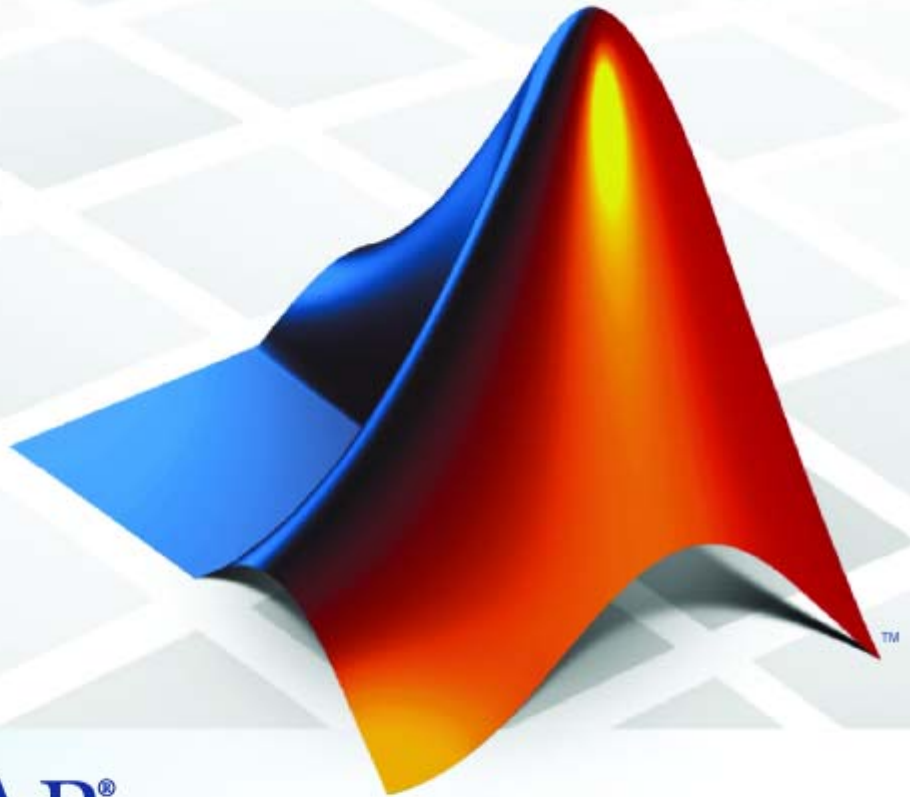


Real-Time Workshop[®] Embedded Coder[™] 5

Getting Started Guide



MATLAB[®]
& **SIMULINK[®]**

How to Contact The MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html 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

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

Real-Time Workshop® Embedded Coder™ Getting Started Guide

© COPYRIGHT 2007–2010 by The MathWorks, Inc.

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

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

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

September 2007 First printing
March 2008 Online only
October 2008 Online only
March 2009 Online only
September 2009 Online only
March 2010 Online only

New for Version 5.0 (Release 2007b)
Revised for Version 5.1 (Release 2008a)
Revised for Version 5.2 (Release 2008b)
Revised for Version 5.3 (Release 2009a)
Revised for Version 5.4 (Release 2009b)
Revised for Version 5.5 (Release 2010a)

Getting Started with Real-Time Workshop® Embedded Coder Software

1

What You Need to Know to Use Real-Time Workshop® Embedded Coder	1-2
What You Can Accomplish Using Real-Time Workshop Technology	1-3
How the Technology Can Fit Into Your Development Process	1-6
Tools for Algorithm Development	1-6
Target Environments	1-10
Applications	1-14
How You Can Apply the Technology to the V-Model for System Development	1-16
What Is the V-Model?	1-16
Types of Simulation and Prototyping	1-18
Types of In-the-Loop Testing for Verification and Validation	1-19

Learning and Using Real-Time Workshop® Embedded Coder Software

2

Using the Tutorials	2-2
Introduction	2-2
Prerequisites	2-3
Third-Party Software	2-3
Setting Up the Tutorial Files	2-4

Understanding the Demo Model	2-5
Introduction	2-5
Understanding the Functional Design of the Model	2-6
Viewing the Top Model	2-6
Viewing Subsystems	2-7
Understanding the Simulation Testing Environment	2-8
Running the Simulation Tests	2-12
Setting the Configuration Options for Code Generation ..	2-13
Saving the Configuration Parameters as a MATLAB Function	2-24
Generating Code for the Model	2-24
Examining the Generated Code	2-25
Topics for Further Study	2-27
Configuring the Data Interface	2-28
Introduction	2-28
Declaring Data	2-28
Using Data Objects in Simulink Models and Stateflow Charts	2-31
Adding New Data Objects	2-34
Configuring Data Objects	2-35
Controlling File Placement of Parameter Data	2-35
Enabling Data Objects in Generated Code	2-36
Effects of Simulation on Data Typing	2-37
Viewing Data Objects in Generated Code	2-39
Managing Data	2-42
Topics for Further Study	2-42
Partitioning Functions in the Generated Code	2-43
Introduction	2-43
About Atomic and Virtual Subsystems	2-43
Viewing Changes in the Model Architecture	2-44
Controlling Function Location and File Placement in Generated Code	2-45
Understanding Reentrant Code	2-46
Using a Mask to Pass Parameters into a Library Subsystem	2-47
Generating Code from an Atomic Subsystem	2-48
Generating Code: Full Model vs. Exported Functions	2-49
Effect of Execution Order on Simulation Results	2-51
Topics for Further Study	2-53

Calling External C Functions from the Model and Generated Code	2-54
Introduction	2-54
Including Preexisting C Functions in a Simulink Model ..	2-54
Creating a Block That Calls a C Function	2-55
Validating the External Code in the Simulink Environment	2-56
Validating the C Code as Part of the Simulink Model	2-58
Calling the C Function from the Generated Code	2-59
Topics for Further Study	2-60
Integrating the Generated Code into the External Environment	2-61
Introduction	2-61
Building and Collecting the Required Data and Files	2-61
Integrating the Generated Code into an Existing System	2-62
About the Integration Environment	2-62
Matching the System Interfaces	2-64
Matching Function-Call Interfaces	2-66
Building a Project in the Eclipse Environment	2-67
Topics for Further Study	2-68
Testing the Generated Code	2-69
Introduction	2-69
Methods for Validating Generated Code	2-69
Reusing Test Data: Test Vector Import/Export	2-71
Testing via Software-in-the-Loop (S-Functions)	2-72
Configuring the System for Testing via Test Vector Import/Export	2-74
Testing with Test Vector Import/Export Using the Eclipse Environment	2-76
Testing via Processor-in-the-Loop (PIL)	2-77
Evaluating the Generated Code	2-78
Introduction	2-78
Evaluating Code	2-78
About the Compiler Used	2-79
Viewing the Code Metrics	2-79
About the Build Option Configurations	2-79
Configuration 1: Reusable Functions, Data Type Double ..	2-80
Configuration 2: Reusable Functions, Data Type Single ..	2-81

Configuration 3: Nonreusable Functions, Data Type	
Single	2-82

Installing and Using an IDE for the Integration and Testing Tutorials (Optional)

A

Installing the Eclipse IDE and Cygwin Debugger	A-2
Installing the Eclipse IDE	A-2
Installing the Cygwin Debugger	A-3
Integrating and Testing Code with the Eclipse IDE ...	A-4
Introducing Eclipse	A-4
Defining a New C Project	A-5
Configuring the Debugger	A-6
Starting the Debugger	A-7
Setting the Cygwin Path	A-7
What the Eclipse Debugger Can Do	A-8

Getting Started with Real-Time Workshop Embedded Coder Software

- “What You Need to Know to Use Real-Time Workshop® Embedded Coder” on page 1-2
- “What You Can Accomplish Using Real-Time Workshop Technology” on page 1-3
- “How the Technology Can Fit Into Your Development Process” on page 1-6
- “How You Can Apply the Technology to the V-Model for System Development” on page 1-16

What You Need to Know to Use Real-Time Workshop Embedded Coder

Before you use the Real-Time Workshop® Embedded Coder™ software, you should be familiar with

- Using the Simulink® and Stateflow® software to create models or state machines as block diagrams, running such simulations in Simulink, and interpreting output in the MATLAB® workspace
- Using Real-Time Workshop® software to generate code and build executable programs from Simulink models
- High-level programming language concepts applied to embedded, real-time systems

If you have not done so, you should read:

- The tutorials in the *Real-Time Workshop Getting Started Guide*. The tutorials provide hands-on experience in configuring models for code generation and generating code.
- “Laying Out the Model Architecture” and “Scheduling Considerations” in the Real-Time Workshop documentation. These sections give a general overview of the architecture and execution of programs generated by Real-Time Workshop software.

If you are familiar with C language constructs and want to learn about how to map commonly used C constructs to code generated from model design patterns that include Simulink blocks, Stateflow charts, and Embedded MATLAB® functions, see Technical Solution 1-6AWSQ9 on the MathWorks™ Web site.

What You Can Accomplish Using Real-Time Workshop Technology

Real-Time Workshop technology generates C or C++ source code and executables for algorithms that you model graphically in the Simulink environment or programmatically with the Embedded MATLAB language subset. You can generate code for any Simulink blocks and MATLAB functions that are useful for real-time or embedded applications. The generated source code and executables for floating-point algorithms match the functional behavior of Simulink simulations and Embedded MATLAB code execution to high degrees of fidelity. Using the Simulink® Fixed Point™ product, you can generate fixed-point code that provides a bit-wise accurate match to model simulation results. Such broad support and high degrees of accuracy are possible because Real-Time Workshop technology is tightly integrated with the MATLAB and Simulink execution and simulation engines. In fact, the built-in accelerated simulation modes in Simulink use Real-Time Workshop technology.

You apply Real-Time Workshop technology explicitly with the Real-Time Workshop and Real-Time Workshop Embedded Coder products. Using the Real-Time Workshop product, you can

- Generate source code and executables for discrete-time, continuous-time (fixed-step), and hybrid systems modeled in Simulink
- Use the generated code for real-time and non-real-time applications, including simulation acceleration, rapid prototyping, and hardware-in-the-loop (HIL) testing
- Tune and monitor the generated code by using Simulink blocks and built-in analysis capabilities, or run and interact with the code completely outside the MATLAB and Simulink environment
- Generate code for finite state machines modeled in Stateflow event-based modeling software, using the optional Stateflow® Coder™ product
- Produce source code for many Simulink products and blocksets provided by The MathWorks™ and third-party vendors.

The Real-Time Workshop Embedded Coder product *extends* the Real-Time Workshop product with features that are important for embedded software

development. Using the Real-Time Workshop Embedded Coder add-on product, you gain access to all aspects of Real-Time Workshop technology and can generate code that has the clarity and efficiency of professional handwritten code. For example, you can

- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems
- Customize the appearance of the generated code
- Optimize the generated code for a specific target environment
- Integrate existing (legacy) applications, functions, and data
- Enable tracing, reporting, and testing options that facilitate code verification activities

The following table compares typical applications and key capabilities for these two code generation products.

Product	Typical Applications	Key Capabilities
Real-Time Workshop	Simulation acceleration Simulink model encryption Rapid prototyping HIL testing	Generate code for discrete-time, continuous-time (fixed-step), and hybrid systems modeled in Simulink Tune and monitor the execution of generated code by using Simulink blocks and built-in analysis capabilities or by running and interacting with the code outside the MATLAB and Simulink environment Generate code for finite state machines modeled in Stateflow event-based modeling software, using the optional Stateflow Coder product

Product	Typical Applications	Key Capabilities
		<p>Generate code for many MathWorks and third-party Simulink products and blocksets</p> <p>Integrate existing applications, functions, and data</p>
<p>Real-Time Workshop Embedded Coder</p>	<p>All applications listed for the Real-Time Workshop product</p> <p>Embedded systems</p> <p>On-target rapid prototyping boards</p> <p>Microprocessors used in mass production</p>	<p>All capabilities listed for the Real-Time Workshop product</p> <p>Generate code that has the clarity and efficiency of professional handwritten code</p> <p>Customize the appearance and performance of the code for specific target environments</p> <p>Enable tracing, reporting, and testing options that facilitate code verification activities</p>

How the Technology Can Fit Into Your Development Process

In this section...

“Tools for Algorithm Development” on page 1-6

“Target Environments” on page 1-10

“Applications” on page 1-14

Tools for Algorithm Development

You can use Real-Time Workshop technology to generate standalone C or C++ source code for algorithms that you develop the following ways:

- With MATLAB code, using the Embedded MATLAB language subset
- As Simulink models
- With MATLAB code that you incorporate into Simulink models

The Embedded MATLAB language subset supports MATLAB operators and functions for floating-point and fixed-point math. Simulink support for dynamic system simulation, conditional execution of system semantics, and large model hierarchies provides an environment for modeling periodic and event-driven algorithms commonly found in embedded systems. Real-Time Workshop technology generates code for most Simulink blocks and many MathWorks products.

If you are familiar with C language constructs and want to learn about how to map commonly used C constructs to code generated from model design patterns that include Simulink blocks, Stateflow charts, and Embedded MATLAB functions, see Technical Solution 1-6AWSQ9 on the MathWorks Web site.

The following table lists products that the Real-Time Workshop and Real-Time Workshop Embedded Coder software support.

Products Supported by Real-Time Workshop and Real-Time Workshop Embedded Coder	Notes
Aerospace Blockset™	—
Communications Blockset™	—
Control System Toolbox™	—
Embedded IDE Link™	—
Fuzzy Logic Toolbox™	—
Gauges Blockset™	—
MATLAB	Details: Supports Embedded MATLAB
Model-Based Calibration Toolbox™	—
Model Predictive Control Toolbox™	—
PolySpace® Model Link™ SL	Not supported by Real-Time Workshop
Real-Time Windows Target™	—
Signal Processing Blockset™	Details: “Simulink Block Data Type Support for Signal Processing Blockset” Table (enter the MATLAB show <code>signalblockdatatype</code> table command)
SimDriveline™	—
SimElectronics®	—
SimHydraulics®	—
SimMechanics™	—
SimPowerSystems™	Not supported by Real-Time Workshop Embedded Coder
Simscape™	—
Simulink	Details: “Simulink Built-In Blocks That Support Code Generation” Table in the Real-Time Workshop documentation
Simulink Fixed Point	—
Simulink® 3D Animation™	—

Products Supported by Real-Time Workshop and Real-Time Workshop Embedded Coder	Notes
Simulink® Design Optimization™	—
Simulink® Report Generator™	—
Simulink® Verification and Validation™	—
Stateflow and Stateflow Coder	—
System Identification Toolbox™	Exceptions: <ul style="list-style-type: none"> • Nonlinear IDNLGREY Model, IDDATA Source, IDDATA Sink, and estimator blocks • Nonlinear ARX models that contain custom regressors • neuralnet nonlinearities • customnet nonlinearities
Target Support Package™	—
Vehicle Network Toolbox™	Exception: CAN Configuration, CAN Receive, and CAN Transmit blocks in the CAN Communication library
Video and Image Processing Blockset™	—
xPC Target™	—
xPC Target Embedded Option™	—

Use of both Embedded MATLAB code and Simulink models is typical for Model-Based Design projects where you start developing an algorithm through research and development or advanced production, using MATLAB, and then use Simulink for system deployment and verification. Benefits of this approach include:

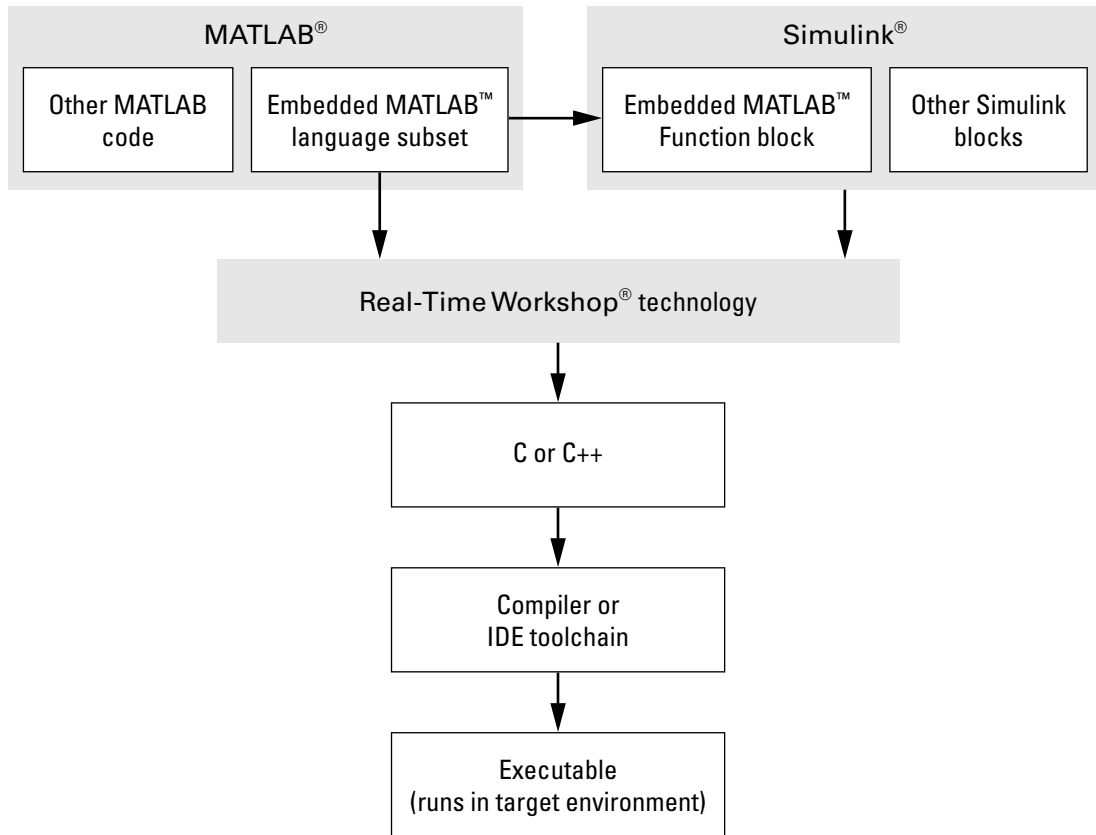
- Richer system simulation environment
- Ability to verify the Embedded MATLAB code

- Real-Time Workshop and Real-Time Workshop Embedded Coder C/C++ code generation for the model and embedded MATLAB code

The following table summarizes how to generate C or C++ code for each of the three approaches and identifies where you can find more information.

If you develop algorithms using...	You generate code by...	For more information, see...
Embedded MATLAB language subset	Entering the Real-Time Workshop function <code>emlc</code> in the MATLAB Command Window.	“Working with the Embedded MATLAB Subset” “Converting MATLAB Code to C/C++ Code”
Simulink	Configuring and initiating code generation for your model or subsystem with the Simulink Configuration Parameters dialog.	“Workflow for Developing Applications Using Real-Time Workshop Software” in Getting Started with Real-Time Workshop
Embedded MATLAB language subset and Simulink	Including Embedded MATLAB code in Simulink models or subsystems by using the Embedded MATLAB Function block. To use this block, you can do one of the following: <ul style="list-style-type: none"> • Copy your code into the block. • Call your code from the block by referencing the appropriate files on the MATLAB path. 	“Working with the Embedded MATLAB Subset” in the Embedded MATLAB documentation

The following figure shows the three design and deployment environment options. Although not shown in the figure, other products that support code generation, such as Stateflow software, are available.



Target Environments

In addition to generating source code for a model or subsystem, Real-Time Workshop technology generates make or project files you need to build an executable for a specific target environment. The generated make or project files are optional. That is, if you prefer, you can build an executable for the generated source files by using an existing target build environment, such as a third-party integrated development environment (IDE). Applications of code generated with Real-Time Workshop technology range from calling a few exported C or C++ functions on a host computer to generating a complete executable using a custom build process, for custom hardware, in an environment completely separate from the host computer running MATLAB and Simulink.

Real-Time Workshop technology provides built-in *system target files* that generate, build, and execute code for specific target environments. These system target files offer varying degrees of support for interacting with the generated code to log data, tune parameters, and experiment with or without Simulink as the external interface to your generated code.

Before you select a system target file, you need to identify the target environment on which you expect to execute your generated code. The three most common target environments include:

Target Environment	Description
Host computer	<p>The same computer that runs MATLAB and Simulink. Typically, a host computer is a PC or UNIX^{®1} environment that uses a non-real-time operating system, such as Microsoft[®]Windows[®] or Linux^{®2}. Non-real-time (general purpose) operating systems are nondeterministic. For example, they might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Thus, the executable for your generated code might run faster or slower than the sample rates you specified in your model.</p>
Real-time simulator	<p>A different computer than the host computer. A real-time simulator can be a PC or UNIX environment that uses a real-time operating system (RTOS), such as:</p> <ul style="list-style-type: none"> • xPC Target system • A real-time Linux system • A Versa Module Eurocard (VME) chassis with PowerPC[®] processors running a commercial RTOS, such as VxWorks[®] from Wind River[®] Systems <p>The generated code runs in real time and behaves deterministically. Although, the exact nature of execution varies based on the particular behavior of the system hardware and RTOS.</p>

1. UNIX[®] is a registered trademark of The Open Group in the United States and other countries.
2. Linux[®] is a registered trademark of Linus Torvalds.

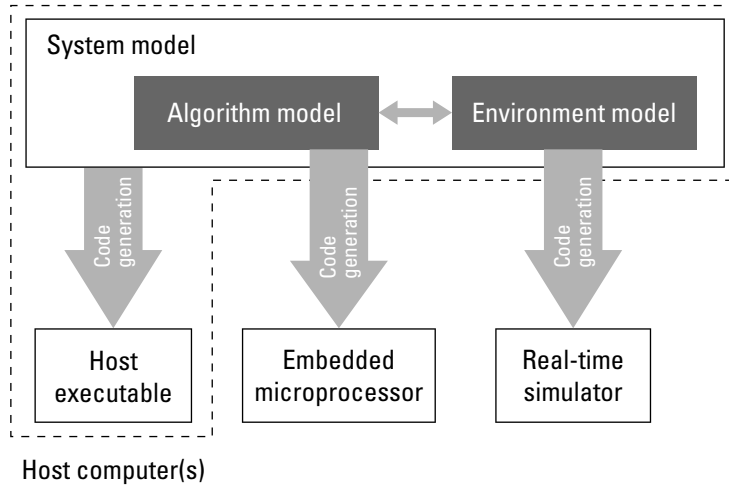
Target Environment	Description
	Typically, a real-time simulator connects to a host computer for data logging, interactive parameter tuning, and Monte Carlo batch execution studies.
Embedded microprocessor	<p>A computer that you eventually disconnect from a host computer and run standalone as part of an electronics-based product. Embedded microprocessors range in price and performance, from high-end digital signal processors (DSPs) used to process communication signals to inexpensive 8-bit fixed-point microcontrollers used in mass production (for example, electronic parts produced in the millions of units). Embedded microprocessors can:</p> <ul style="list-style-type: none"> • Use a full-featured RTOS • Be driven by basic interrupts • Use rate monotonic scheduling provided with Real-Time Workshop technology

A target environment can:

- Have single- or multiple-core CPUs
- Be standalone or communicate as part of a computer network

In addition, you can deploy different parts of a Simulink model on different target environments. For example, it is common to separate the component (algorithm or controller) portion of a model from the environment (or plant). Using Simulink to model an entire system (plant and controller) is often referred to as closed-loop simulation and can provide many benefits such as early verification of component correctness.

The following figure shows example target environments for code generated for a model.



Applications

The following table lists several ways you can apply Real-Time Workshop technology in the context of the different target environments.

Application	Description
Host Computer	
Accelerated simulation	You apply techniques to speed up the execution of model simulation in the context of the MATLAB and Simulink environment. Accelerated simulations are especially useful when run time is long compared to the time associated with compilation and checking whether the target is up to date.
Rapid simulation	You execute code generated for a model in non-real time on the host computer, but outside the context of the MATLAB and Simulink environment.
System simulation	You integrate components into a larger system. You provide generated source code and related dependencies for building in another environment or a host-based shared library to which other code can dynamically link.
Model encryption	You generate a Simulink shareable object library for a model or subsystem for use by a third-party vendor in another Simulink simulation environment.
Real-Time Simulator	
Rapid prototyping	You generate, deploy, and tune code on a real-time simulator connected to the system hardware (for example, physical plant or vehicle) being controlled. This design step is also crucial for validating whether a component can adequately control the physical system.
System simulation	You integrate generated source code and dependencies for components into a larger system that is built in another environment. You can use shared library files to encrypt components for intellectual property protection.

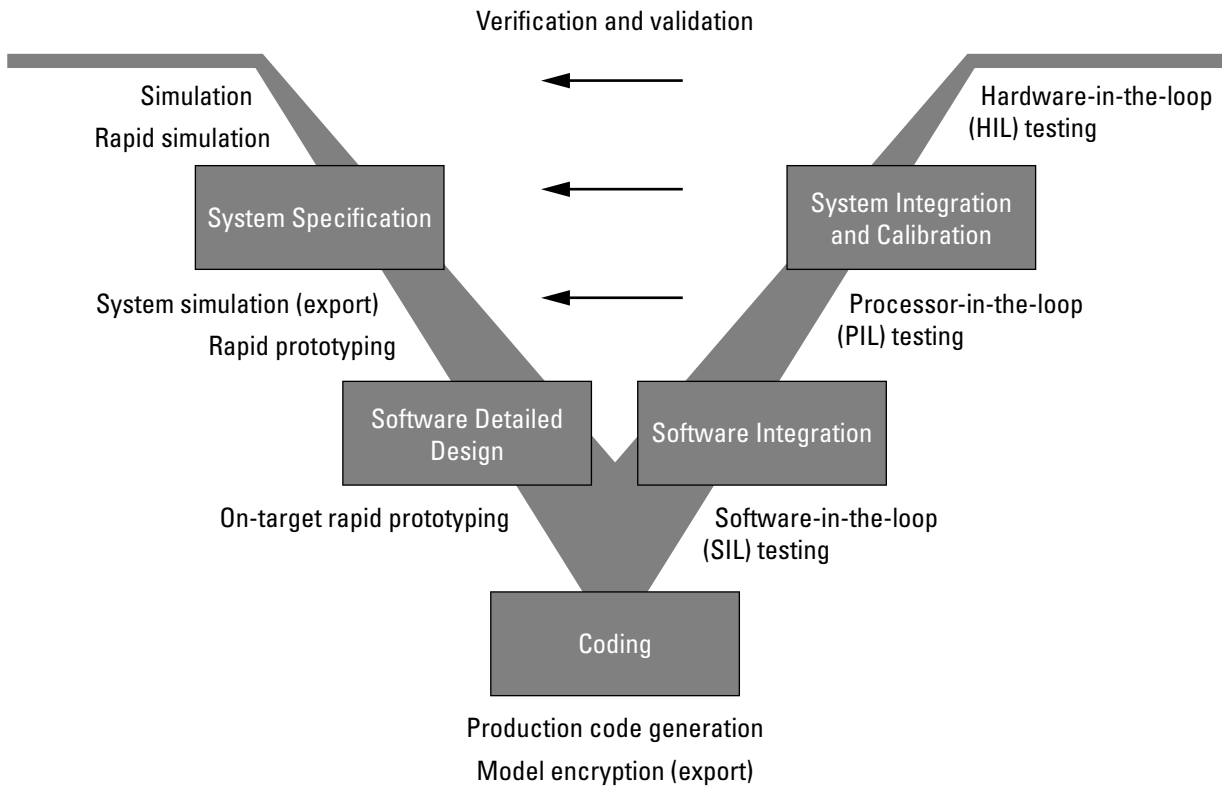
Application	Description
On-target rapid prototyping	You generate code for a detailed design that you can run in real time on an embedded microprocessor while tuning parameters and monitoring real-time data. This design step allows you to assess, interact with, and optimize code, using embedded compilers and hardware.
Embedded Microprocessor	
Production code generation	From a model, you generate code that is optimized for speed, memory usage, simplicity, and if necessary, compliance with industry standards and guidelines.
Software-in-the-loop (SIL) testing	You execute generated code with your plant model within Simulink to verify successful conversion of the model to code. You might change the code to emulate target word size behavior and verify numerical results expected when the code runs on an embedded microprocessor, or use actual target word sizes and just test production code behavior.
Processor-in-the-loop (PIL) testing	You test an object code component with a plant or environment model in an open- or closed-loop simulation to verify successful model-to-code conversion, cross-compilation, and software integration.
Hardware-in-the-loop (HIL) testing	You verify an embedded system or embedded computing unit (ECU), using a real-time target environment.

How You Can Apply the Technology to the V-Model for System Development

In this section...
“What Is the V-Model?” on page 1-16
“Types of Simulation and Prototyping” on page 1-18
“Types of In-the-Loop Testing for Verification and Validation” on page 1-19

What Is the V-Model?

The V-model is a representation of system development that highlights verification and validation steps in the system development process. As the following figure shows, the left side of the V identifies steps that lead to code generation, including requirements analysis, system specification, detailed software design, and coding. The right side focuses on the verification and validation of steps cited on the left side, including software integration and system integration.



Depending on your application and role in the process, you might focus on one or more of the steps called out in the V or repeat steps at several stages of the V. Real-Time Workshop technology and related products provide tooling you can apply at each step.

The following sections compare

- Types of simulation and prototyping
- Types of in-the-loop testing for verification and validation

For a map of information on applications of Real-Time Workshop technology identified in the figure, see the following tables:

- “Documenting and Validating Requirements”

- “Developing a Model Executable Specification”
- “Developing a Detailed Software Design”
- “Generating the Application Code”
- “Integrating and Verifying Software”
- “Integrating, Verifying, and Calibrating System Components”

Types of Simulation and Prototyping

The following table compares the types of simulation and prototyping identified on the left side of the V-model diagram.

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
Purpose	Test and validate functionality of concept model	Refine, test, and validate functionality of concept model in non-real time	Test new ideas and research	Refine and calibrate designs during development process
Execution hardware	Host computer	Host computer Standalone executable runs outside of MATLAB and Simulink environment	PC or nontarget hardware	Embedded computing unit (ECU) or near-production hardware

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
Code efficiency and I/O latency	Not applicable	Not applicable	Less emphasis on code efficiency and I/O latency	More emphasis on code efficiency and I/O latency
Ease of use and cost	<p>Can simulate component (algorithm or controller) and environment (or plant)</p> <p>Normal mode simulation in Simulink enables you to access, display, and tune data and parameters while experimenting</p> <p>Can accelerate Simulink simulations with Accelerated and Rapid Accelerated modes</p>	<p>Easy to simulate models of hybrid dynamic systems that include components and environment models</p> <p>Ideal for batch or Monte Carlo simulations</p> <p>Can repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model</p> <p>Can be connected to Simulink to monitor signals and tune parameters</p>	<p>Might require custom real-time simulators and hardware</p> <p>Might be done with inexpensive off-the-shelf PC hardware and I/O cards</p>	<p>Might use existing hardware, thus less expensive and more convenient</p>

Types of In-the-Loop Testing for Verification and Validation

The following table compares the types of in-the-loop testing for verification and validation identified on the right side of the V-model diagram.

	SIL Testing	PIL Testing on Embedded Hardware	PIL Testing on Instruction Set Simulator	HIL Testing
Purpose	Verify component source code	Verify component object code	Verify component object code	Verify system functionality
Fidelity and accuracy	Two options: Same source code as target, but might have numerical differences Changes source code to emulate word sizes, but is bit accurate for fixed-point math	Same object code Bit accurate for fixed-point math Cycle accurate since code runs on hardware	Same object code Bit accurate for fixed-point math Might not be cycle accurate	Same executable code Bit accurate for fixed-point math Cycle accurate Use real and emulated system I/O
Execution platforms	Host	Target	Host	Target
Ease of use and cost	Desktop convenience Executes just in Simulink No cost for hardware	Executes on desk or test bench Uses hardware — process board and cables	Desktop convenience Executes just on host computer with Simulink and integrated development environment (IDE) No cost for hardware	Executes on test bench or in lab Uses hardware — processor, embedded computer unit (ECU), I/O devices, and cables
Real time capability	Not real time	Not real time (between samples)	Not real time (between samples)	Hard real time

Learning and Using Real-Time Workshop Embedded Coder Software

- “Using the Tutorials” on page 2-2
- “Understanding the Demo Model” on page 2-5
- “Configuring the Data Interface” on page 2-28
- “Partitioning Functions in the Generated Code” on page 2-43
- “Calling External C Functions from the Model and Generated Code” on page 2-54
- “Integrating the Generated Code into the External Environment” on page 2-61
- “Testing the Generated Code” on page 2-69
- “Evaluating the Generated Code” on page 2-78

Using the Tutorials

In this section...
“Introduction” on page 2-2
“Prerequisites” on page 2-3
“Third-Party Software” on page 2-3
“Setting Up the Tutorial Files” on page 2-4

Introduction

The process for designing and implementing a control algorithm for an embedded real-time application varies among different organizations. However, some basic steps in the process are common. This getting started documentation provides seven tutorials that apply MathWorks products to those common steps. In these tutorials, you configure a Simulink model and use Real-Time Workshop Embedded Coder software to

- Generate code for the model
- Integrate the generated code with an application framework outside the Simulink environment
- Test and analyze the generated code

Each tutorial focuses on a specific aspect of code generation or integration and is self-contained. You can step through them in any order, and skim or skip any that do not apply to your needs. The seven tutorials are:

- “Understanding the Demo Model” on page 2-5
- “Configuring the Data Interface” on page 2-28
- “Partitioning Functions in the Generated Code” on page 2-43
- “Calling External C Functions from the Model and Generated Code” on page 2-54
- “Integrating the Generated Code into the External Environment” on page 2-61
- “Testing the Generated Code” on page 2-69

- “Evaluating the Generated Code” on page 2-78

Each tutorial uses a unique Simulink demo model and data set. As you proceed through the tutorials, you save each model after you have worked on it, preserving your modifications to the model and model data for future examination. To prevent any errors from carrying over, you begin the next tutorial by opening a new model and loading new data.

These tutorials provide instructions for performing specific tasks and references related documentation. If a task fails for any reason, error messages appear in the MATLAB Command Window.

Prerequisites

The tutorials assume familiarity with the following techniques:

MathWorks products

- How to read, write, and apply MATLAB scripts
- How to create a basic Simulink model with Stateflow charts
- How to run Simulink simulations and evaluate the results

C programming

- How to use C data types and storage classes
- How to use function prototypes and call functions
- How to compile a C function

Metrics for evaluating embedded software

- How to evaluate basic code readability
- How to evaluate RAM/ROM usage

Third-Party Software

To compile and build generated code for the integration and testing tutorials, you can use an Integrated Development Environment (IDE) or equivalent tools such as command-line compilers and makefiles. Appendix A, “Installing

and Using an IDE for the Integration and Testing Tutorials (Optional)” explains how to install and use the Eclipse™ IDE for C/C++ Developers and the Cygwin™ debugger, for integrating and testing your generated code.

Setting Up the Tutorial Files

Set up a directory for your tutorial work:

- 1 Create a writable working directory outside the scope of your MATLAB installation directory.
- 2 Copy the following files from *matlabroot/toolbox/rtw/rtwdemos* to your working directory:

```
rtwdemo_PCG_Eval_P1.mdl  
rtwdemo_PCG_Eval_P2.mdl  
rtwdemo_PCG_Eval_P3.mdl  
rtwdemo_PCG_Eval_P4.mdl  
rtwdemo_PCG_Eval_P5.mdl  
rtwdemo_PCG_Eval_P6.mdl  
rtwdemo_PCGEvalHarness.mdl  
rtwdemo_PCGEvalHarnessSFun.mdl
```


Understanding the Demo Model

In this section...

“Introduction” on page 2-5

“Understanding the Functional Design of the Model” on page 2-6

“Viewing the Top Model” on page 2-6

“Viewing Subsystems” on page 2-7

“Understanding the Simulation Testing Environment” on page 2-8

“Running the Simulation Tests” on page 2-12

“Setting the Configuration Options for Code Generation” on page 2-13

“Saving the Configuration Parameters as a MATLAB Function” on page 2-24

“Generating Code for the Model” on page 2-24

“Examining the Generated Code” on page 2-25

“Topics for Further Study” on page 2-27

Introduction

This tutorial introduces a Simulink demo model, `rtwdemo_PCG_Eval_P1`, from a behavioral and structural perspective. It explains how to generate code and shows the basics of configuring a model.

In this tutorial, you:

- Understand the functional behavior of the model
- Understand how to validate the model
- Become familiar with model checking tools
- Become familiar with configuration options that affect code generation
- Learn how to generate code from a model

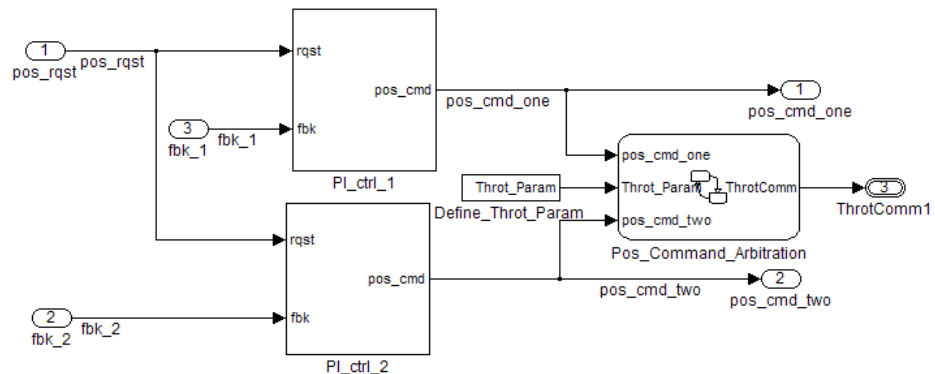
Understanding the Functional Design of the Model

This tutorial uses a simple but functionally complete demo model of a throttle controller. The model features redundancy, which is common for safety-critical, drive-by-wire applications. The model highlights a standard model structure and a set of basic blocks used in algorithm design.

In the provided configuration, the model generates code. However, the code is not configured for a production target system. This tutorial guides you through the steps necessary to change the target configuration and shows how the format of the generated code changes with the completion of each task.

Viewing the Top Model

Open the top model by entering `rtwdemo_PCG_Eval_P1` at the MATLAB command line.



The top model consists of:

- Four subsystems: `PI_ctrl_1`, `PI_ctrl_2`, `Define_Throt_Param`, and `Pos_Command_Arbitration`
- Top-level inputs: `pos_rqst`, `fbk_1`, and `fbk_2`
- Top-level outputs: `pos_cmd_one`, `pos_cmd_two`, and `ThrotComm1`
- Signal routing
- No blocks that change the value of a signal, such as Sum and Integrator

The layout uses a basic architectural style for models:

- Separation of calculations from signal routing (lines and buses)
- Partitioning into subsystems

You can apply this style to all types of models.

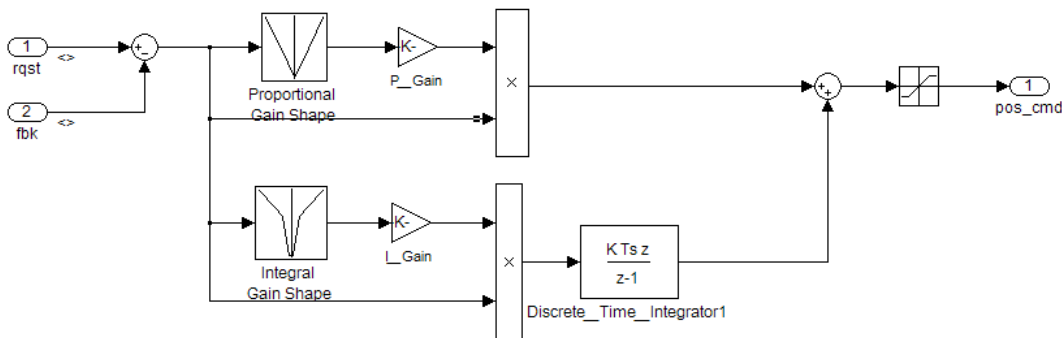
Viewing Subsystems

Perform the following steps to explore two of the key subsystems in the top model.

- 1 Open the `rtwdemo_PCG_Eval_P1` demo model.

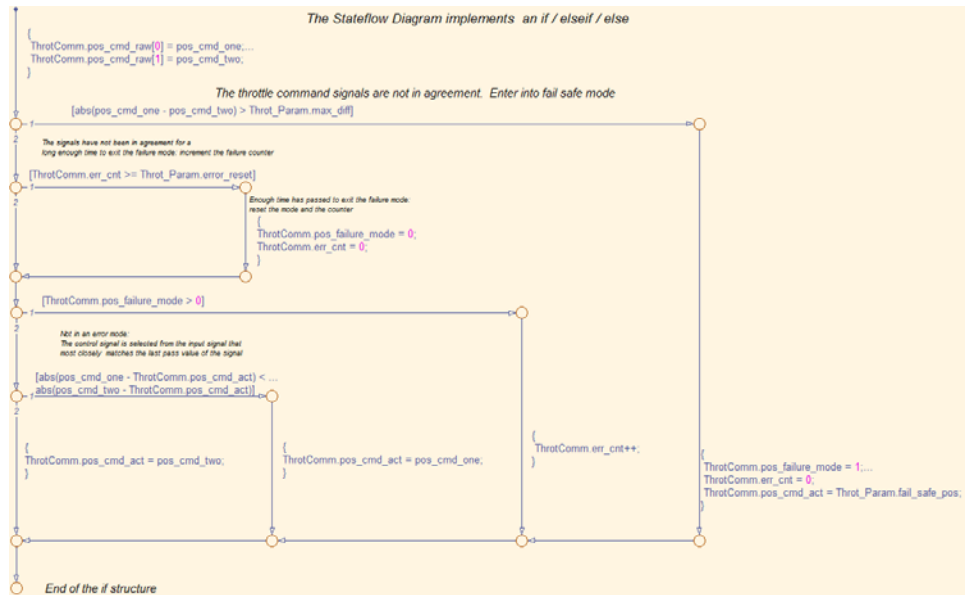
Two subsystems in the top model represent proportional-integral (PI) controllers, `PI_ctrl1_1` and `PI_ctrl1_2`. These identical subsystems, at this stage, use identical data. Later, you use the subsystems to learn how Real-Time Workshop software can create reusable functions.

- 2 Open the `PI_ctrl1_1` subsystem by double-clicking the subsystem block.



The PI controllers in the model are from a *library*, a group of related blocks or models for reuse. Libraries provide one of two methods for including and reusing models. The second method, model referencing, is covered below in “Understanding the Simulation Testing Environment” on page 2-8. You cannot edit a block that you add to a model from a library in the context of the model. To edit the block, you must do so in the library. This restriction ensures that instances of the block in different models remain consistent.

- 3 Open the Pos_Command_Arbitration subsystem by double-clicking the subsystem block. The Stateflow chart performs basic error checking on the two command signals. If the command signals are too far apart, the Stateflow diagram sets the output to a fail_safe position.



Understanding the Simulation Testing Environment

To test your throttle controller algorithm, you incorporate it into a *test harness*. A test harness is a model that evaluates the control algorithm and offers the following benefits:

- Separates test data from the control algorithm
- Separates the plant or feedback model from the control algorithm
- Provides a reusable environment for multiple versions of the control algorithm

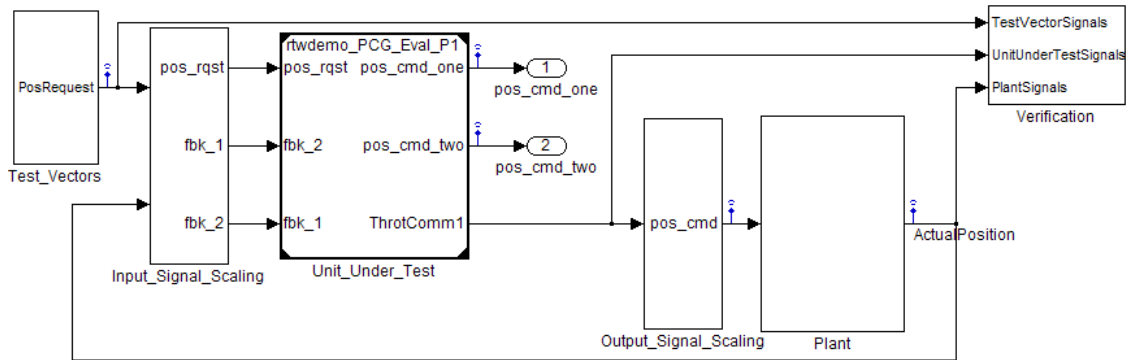
The test harness model provided with this tutorial implements a common simulation testing environment, consisting of the following parts:

- Unit under test

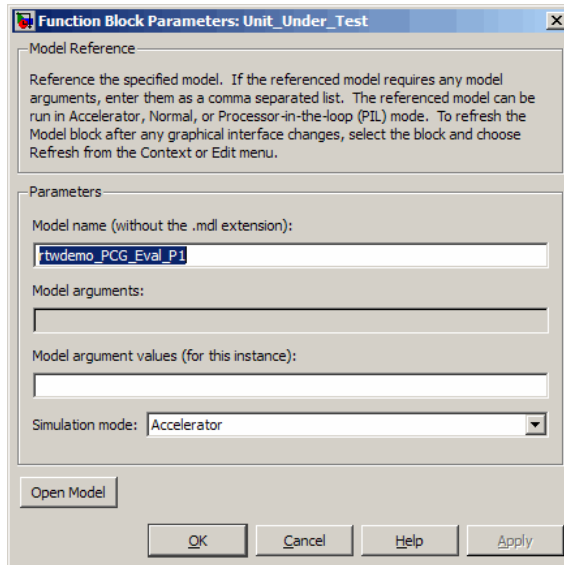
- Test vector source
- Evaluation and logging
- Plant or feedback system
- Input and output scaling

Perform the following steps to explore the simulation testing environment.

- 1 Open the test harness model by entering `rtwdemo_PCGEvalHarness` at the MATLAB command line.



- 2 In this test harness, the control algorithm is the *unit under test*. The control algorithm from `rtwdemo_PCG_Eval_P1` is referenced using a Model block named `Unit_Under_Test`. To confirm this, you can open the `Unit_Under_Test` block and view the control algorithm. Also, you can view the model reference parameters by right-clicking the `Unit_Under_Test` block and selecting **Model Reference Parameters**. The name of the referenced model `rtwdemo_PCG_Eval_P1` is shown in the **Model name** field of the Model Reference dialog box. The Model block provides a second method for reusing components.

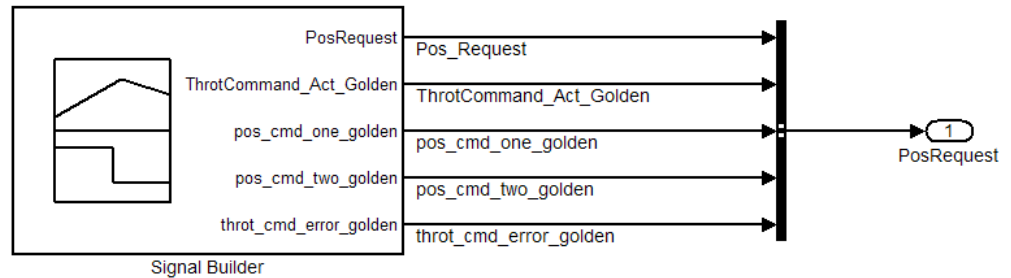


The Model block allows you to reference other models (directly or indirectly) from the top model as *compiled functions*. By default, Simulink software recompiles the model when the referenced models change. Compiled functions have several advantages over libraries:

- Simulation time is faster for large models.
- You can directly simulate compiled functions.
- The simulation requires less memory. Only one copy of the compiled model is in memory, even when the model is referenced multiple times.

- 3** Open the *test vector source*, implemented in this test harness as the Test_Vectors subsystem.

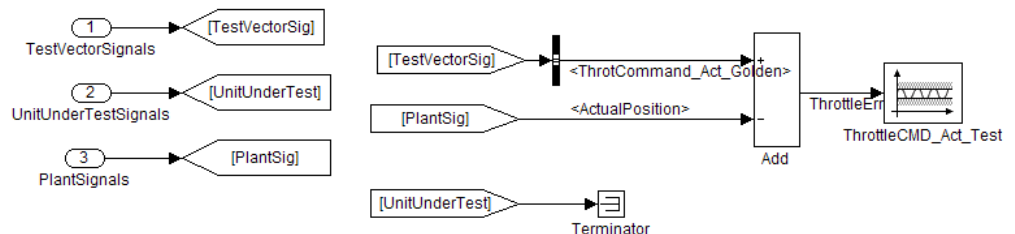
The test harness model uses a Signal Builder block for the test vector source. The block has data that drives the simulation (pos_rqst) and provides the expected results used by the Verification subsystem. This demo model uses only one set of test data. Typically, you would create a test suite that fully exercises the system.



- 4 Open the *evaluation and logging* subsystem, implemented in this test harness as `Verification`.

The test harness compares the control algorithm simulation results against *golden data* — a set of test results that have been certified by an expert to exhibit the desired behavior for the control algorithm. In this subsystem, an Assertion block compares the simulated throttle value position from the plant against the golden value from the test harness. If the difference between the two signals is greater than 5%, the test fails and the Assertion block stops the simulation.

Alternatively, you can evaluate the simulation data after the simulation completes execution. You can use either MATLAB scripts or third-party tools to perform the evaluation. Post-execution evaluation provides greater flexibility in the analysis of the data. However, it requires waiting until execution is complete. Combining the two methods can provide a highly flexible and efficient test environment.



- 5 Open the *plant or feedback system*, implemented in this test harness as the `Plant` subsystem.

The Plant subsystem models the throttle dynamics with a transfer function in canonical form. You can create plant models to any level of fidelity. It is common to use different plant models at different stages of testing.

- 6 Open the *input and output scaling* subsystems, implemented in this test harness as `Input_Signal_Scaling` and `Output_Signal_Scaling`.

The subsystems that scale input and output perform three primary functions:

- Select input signals to route to the unit under test and output signals to route to the plant.
- Rescale signals between engineering units and units for writable the unit under test.
- Handle rate transitions between the plant and the unit under test.

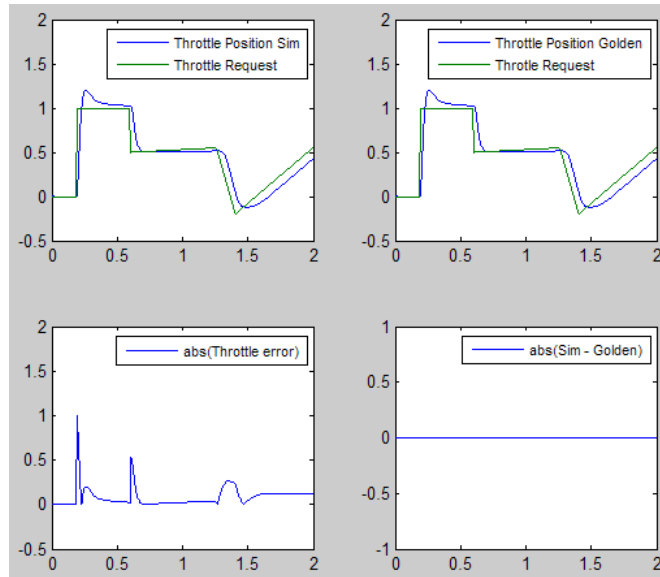
Running the Simulation Tests

- 1 Set up your C compiler by entering `mex -setup` at the MATLAB command line and specifying a valid, installed compiler.
- 2 Check that your working directory is set to a writable directory, such as the directory to which you copied the tutorial demos.
- 3 In the toolbar of the `rtwdemo_PCGEvalHarness` model, click the **Start simulation** icon to run the test harness model simulation.

The first time the test harness runs, the Real-Time Workshop software compiles the referenced model. You can monitor the compilation progress in the MATLAB Command Window.

When the model simulation is complete, Simulink software displays the results in a plot window, shown below.

The lower right plot shows the difference between the expected (golden) throttle position and the throttle position that the plant calculates. If the difference between the two values had been greater than ± 0.05 , the simulation would have stopped.



4 Close the `rtwdemo_PCGEvalHarness` model.

Setting the Configuration Options for Code Generation

- “Overview of Configuration Options” on page 2-13
- “Automatically Setting Configuration Options with Code Generation Objectives” on page 2-14
- “Manually Setting Configuration Options” on page 2-17

Overview of Configuration Options

The first step in preparing a model for code generation is to set model configuration parameters. The configuration parameters determine the method Real-Time Workshop software uses to generate the code and the resulting format.

You can:

- Automatically configure the model configuration parameters based on your code generation objectives. See “Automatically Setting Configuration Options with Code Generation Objectives” on page 2-14.
- Manually configure the model configuration parameters. See “Manually Setting Configuration Options” on page 2-17.

Automatically Setting Configuration Options with Code Generation Objectives

There are six high-level code generation objectives that you can use to automatically set the model configuration parameters:

- Execution efficiency
- ROM efficiency
- RAM efficiency
- Traceability
- Safety precaution
- Debugging

Each objective includes a set of Code Generation Advisor checks that you can use to:

- Review the model configuration parameters against the recommended values of the objectives.
- Verify that the model configuration parameters are set to create code that meets the objectives.

Some of the code generation objectives recommend different values for configuration parameters and include different checks in the Code Generation Advisor. When the objectives are in conflict, the priority of the selection determines which recommendations and checks are presented.

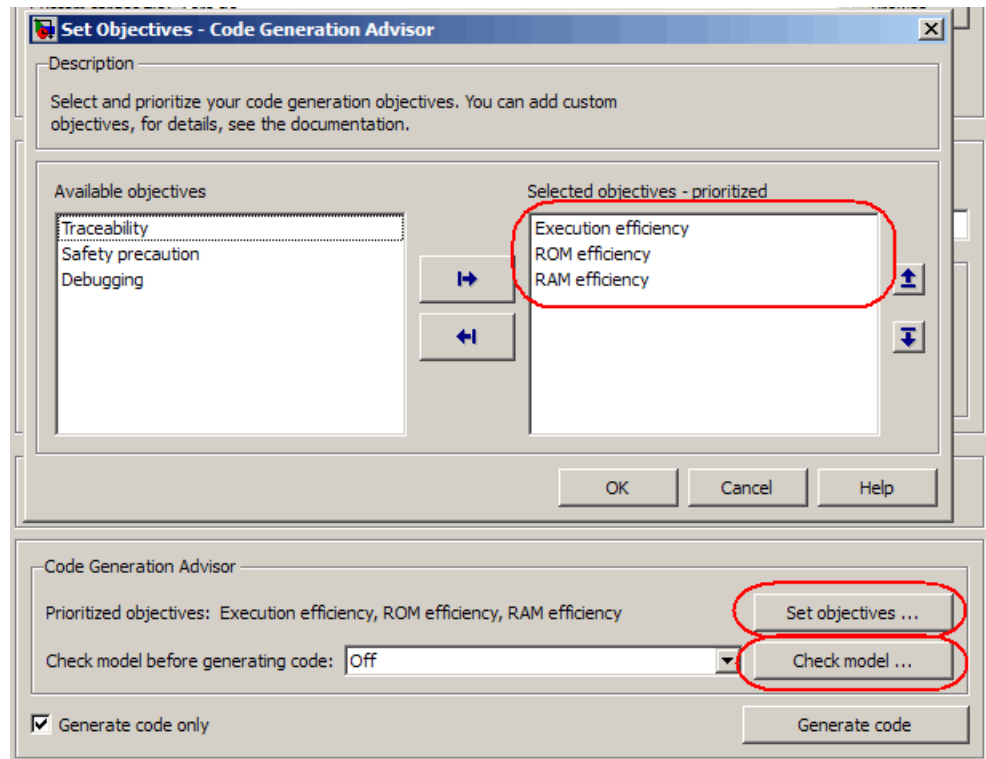
To specify code generation objectives:

- 1** Open the Configuration Parameters dialog box.
- 2** Select the **Real-Time Workshop** pane.

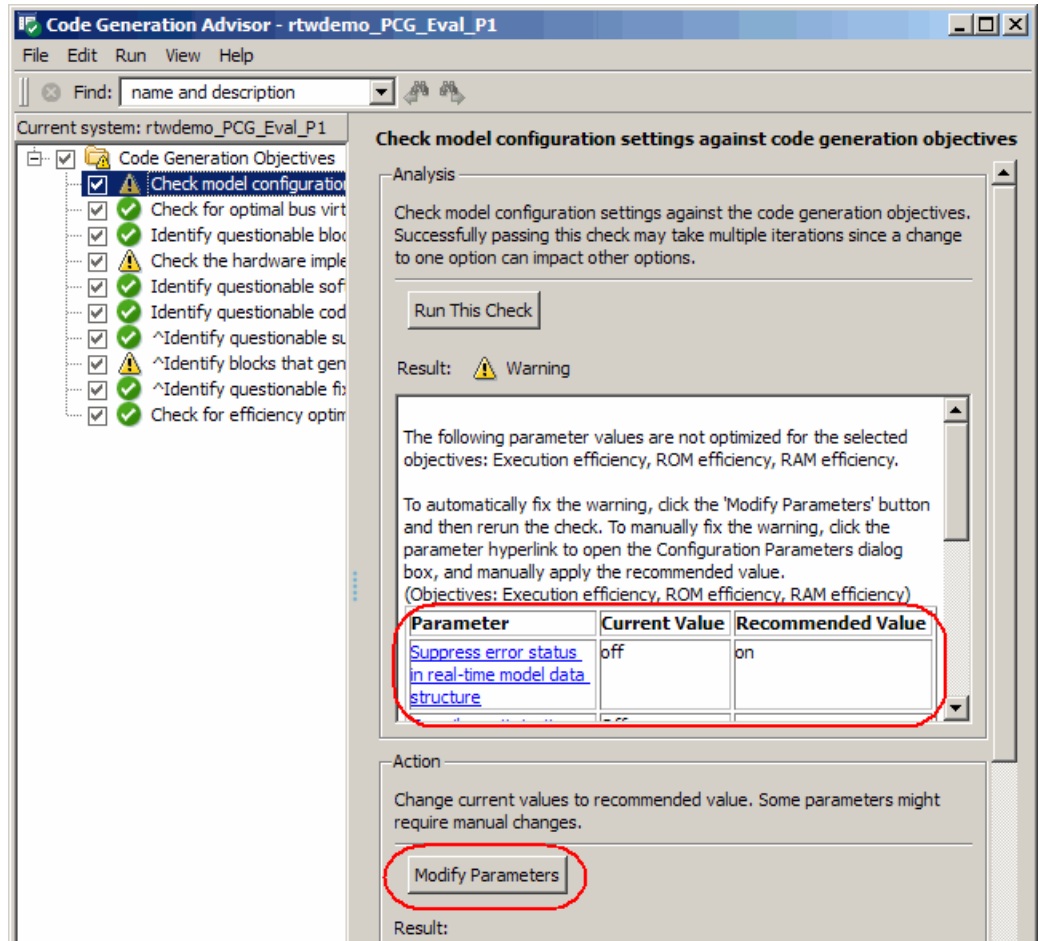
- 3 Ensure that **System target file** is set to an ERT-based target.
- 4 Click **Select objectives**. The Set Objectives dialog box opens.
- 5 In the Set Objectives dialog box, specify your objectives.

To review the model against the specified objectives, use the Code Generation Advisor. To open the Code Generation Advisor, in the Configuration Parameters dialog box, on the **Real-Time Workshop** pane, click **Check model**.

In the following figure, the priority is Execution efficiency, ROM efficiency, and then RAM efficiency.



The Code Generation Advisor dynamically creates the list of checks based on the objectives that you select. The first check reviews the current values of the configuration parameters and recommends values based on the objectives. The check provides an automated method for setting the parameters to the recommended values.



For more information about using the Code Generation Advisor, see “Determining Whether the Model is Configured for Specified Objectives”.

Manually Setting Configuration Options

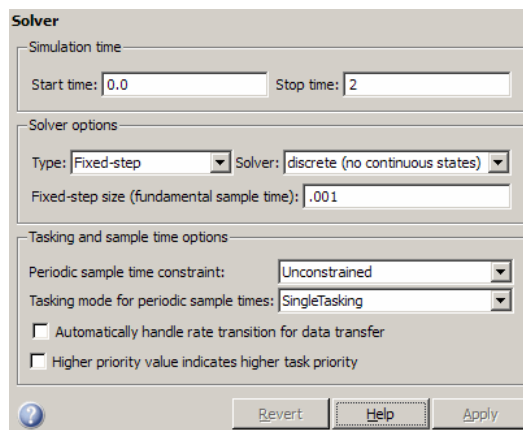
You can manually specify the configuration parameters. To view the Configuration Parameters dialog box, from the Model Editor **Simulation** menu, select **Configuration Parameters**. Alternatively, to view the Configuration Parameters dialog box in the Model Explorer window, from the Model Editor **View** menu, select **Model Explorer**.

This tutorial focuses on four areas of model configuration:

- Solver options
- Optimization options
- Hardware implementation options
- Real-Time Workshop options

Perform the following steps to explore model configuration options.

- 1** Open the demo model `rtwdemo_PCG_Eval_P1` by entering `rtwdemo_PCG_Eval_P1` at the MATLAB command line.
- 2** Open Model Explorer and click **Configuration (Active)** in the **Model Hierarchy** pane.
- 3** Open the **Solver** pane.



For Real-Time Workshop software to generate code for a model, you must configure the model to use a fixed-step solver. The start and stop time do not affect generated code.

Option	Required Setting	Effect on Generated Code
Start time and Stop time	Any	No effect
Type	Fixed-step	Enables code generation
Solver	Any	Controls selection of integration technique used to compute the state derivative of the model
Fixed-step size	Must be lowest common multiple of all rates in the system	Sets base rate of the system
Tasking mode for periodic sample times	SingleTasking or MultiTasking	MultiTasking generates one entry point function for each rate in the system

4 Open the **Optimization** pane.

Simulation and code generation

Block reduction Conditional input branch execution

Implement logic signals as Boolean data (vs. double) Signal storage reuse

Inline parameters

Application lifespan (days)

Use integer division to handle net slopes that are reciprocals of integers

Code generation

Parameter structure:

Signals

Enable local block outputs Reuse block outputs

Inline invariant signals

Eliminate superfluous local variables (Expression folding) Simplify array indexing

Minimize data copies between local and global variables

Pack Boolean data into bitfields

Loop unrolling threshold: Maximum stack size (bytes):

Use memcpy for vector assignment Memcpy threshold (bytes):

Pass reusable subsystem outputs as:

Data initialization

Remove root level I/O zero initialization Use memset to initialize floats and doubles to 0.0

Remove internal data zero initialization Optimize initialization code for model reference

Integer and fixed-point

Remove code from floating-point to integer conversions that wraps out-of-range values

Remove code from floating-point to integer conversions with saturation that maps NaN to zero

Remove code that protects against division arithmetic exceptions

Stateflow

Use bitsets for storing state configuration Use bitsets for storing Boolean data

Accelerating simulations

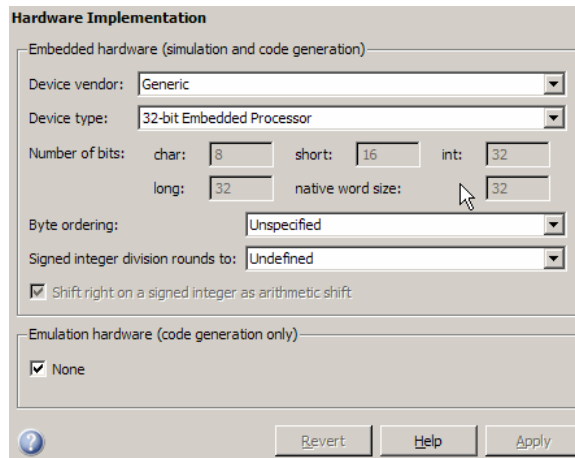
Compiler optimization level:

Verbose accelerator builds

The **Optimization** pane includes the following subpanes.

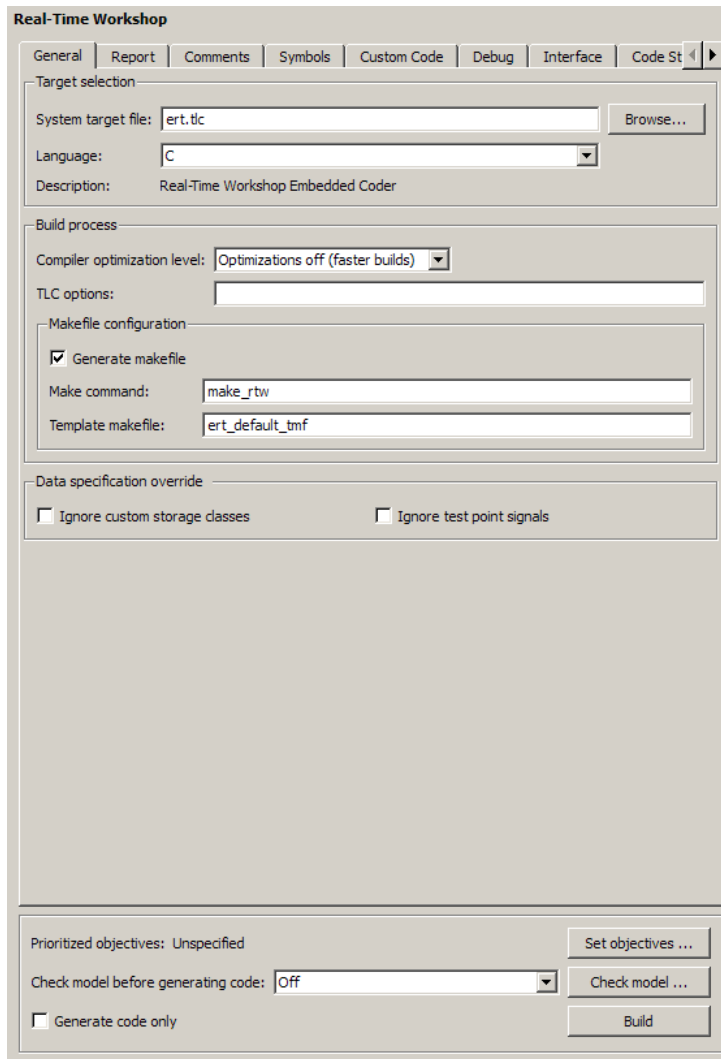
Subpane	Effect
Simulation and code generation	Removes unused branches from the code and controls creation of temporary variables
Signals	Controls code optimizations related to signals — for example, reduces the number of temporary variables created by collapsing multiple computations into a single assignment and by reusing temporary variables
Data initialization	Controls which signals have explicit initialization code
Integer and fixed-point	Enables and disables use of overflow and division-by-zero protection code
Stateflow	Controls how the Stateflow software stores bit-wise information
Accelerating simulations	Provides control over builds for simulations involving accelerator modes or referenced models — no effect on Real-Time Workshop code generation

5 Open the **Hardware Implementation** pane.



Use hardware implementation parameters to specify the word size and byte ordering of the target hardware. This demo model targets a generic 32-bit embedded processor.

6 Open the **Real-Time Workshop** pane.



The **Real-Time Workshop** pane is where you specify the system target file (STF) and the language for your generated code. This demo model uses the Real-Time Workshop Embedded Coder STF (`ert.tlc`) and the C language. You can extend the STF to create a customized configuration. Some of the basic configuration options reachable from the **Real-Time Workshop** pane include:

- Selection of the code generator target:
 - `ert.tlc` — base Embedded Real-Time Target
 - `grt.tlc` — base Generic Real-Time Target
 - Hardware-specific targets
- Selection of code generation language:
 - C — C code
 - C++ — C++ compatible code
 - C++ (Encapsulated) — C++ code with a model class that encapsulates model data and entry-point functions
- Build process options including selection of make file and compiler optimizations
- Additional categories of options on subsidiary panes, such as **Report**, **Comments**, **Symbols**, **Custom Code**, **Debug**, **Interface**, **Code Style**, and **Templates**, among others.
- Code formatting options:
 - Line length
 - Use of parentheses
 - Header file information
 - Variable naming conventions
- Inclusion of custom code:
 - C files
 - H files
 - Object files
 - Directory paths
- Generation of ASAP2 files
- Code generation objectives options to identify changes to model constructs and settings that improve the generated code (see “Mapping Application Objectives to Model Configuration Parameters”).

The **Real-Time Workshop** pane also contains a **Build** button that you can use to build your model. If you select the option **Generate code only**, the button is relabeled to **Generate code**.

Saving the Configuration Parameters as a MATLAB Function

You can save the settings of configuration parameters as a MATLAB function. At the command line, enter:

```
hCs = getActiveConfigSet('rtwdemo_PCG_Eval_P1');  
hCs.saveAs('ConfiguredData');
```

You can use the MATLAB function to:

- Archive the configuration parameters for a model.
- Compare different versions of the configuration parameters using traditional differencing tools.
- Set the configuration parameters of other models.

Running the MATLAB function sets the configuration parameters of other models. For example, to set the configuration parameters for a model called `myModel`, at the command line, enter:

```
hCs2 = ConfiguredData;  
attachConfigSet('myModel', hCs2, true);  
setActiveConfigSet('myModel', hCs2.Name);
```

For more information, see “Saving Configuration Sets” and “Loading Saved Configuration Sets” in the Simulink documentation.

Generating Code for the Model

Perform the following steps to generate code for the demo model that implements the control algorithm.

- 1 Set up your C compiler by entering `mex -setup` at the MATLAB command line and specifying a valid, installed compiler.

- 2 Check that your working directory is set to a writable directory, such as the directory to which you copied the tutorial demos.
- 3 Open the `rtwdemo_PCG_eval_P1` demo model and use one of the following methods to generate code:
 - Click the **Generate code** button in the **Configuration Parameters > Real-Time Workshop** pane.
 - Select **Tools > Real-Time Workshop > Build Model**.

The Real-Time Workshop build process generates several files. The resulting code, while computationally efficient, is not yet organized for integration into the production environment.

Examining the Generated Code

Building the `rtwdemo_PCG_eval_P1` demo model generates multiple files into a subdirectory of your current working directory. In addition to the standard C and H files, the build process generates an HTML code generation report, which provides active links between the code and the model.

Perform the following steps to examine the generated code for the demo model that implements the control algorithm.

- 1 Generate code for the `rtwdemo_PCG_eval_P1` demo model using one of the methods described in the previous section, “Generating Code for the Model” on page 2-24.
- 2 Open Model Explorer, and in the **Model Hierarchy** pane, select **Code for `rtwdemo_PCG_Eval_P1`**.
- 3 In Model Explorer **Contents** pane for the generated model code, select **HTML Report**. Model Explorer displays the HTML code generation report for the `rtwdemo_PCG_eval_P1` demo model.
- 4 In the HTML report, click the link for the generated file `rtwdemo_PCG_Eval_P1.c` and examine the generated code. Notice that:
 - All of the controller code is in one function, `rtwdemo_PCG_Eval_P1_step`.
 - The operations of multiple blocks are in one equation.
 - The `rtwdemo_PCG_Eval_P1_initialize` function initializes variables.

- Real-Time Workshop data structures (for example, `rtwdemo_PCG_Eval_P1_U.pos_rqst`) define all data.
- You can click links in the HTML report to display and highlight the corresponding model block. For example, as shown below, you can click the link `<S2>/Sum2` in the HTML report to highlight and display the corresponding Sum block in your model (as well as the `PI_ctrl1_1` subsystem block that contains it).

```

/* Model step function */
void rtwdemo_PCG_Eval_P1_step(void)
{
    {
        real_T rtb_Sum3;
        real_T rtb_Saturation1;

        /* Sum: '<S2>/Sum2' incorporates:
         * Inport: '<Root>/fbk_1'
         * Inport: '<Root>/pos_rqst'
         */
        rtb_Sum3 = rtwdemo_PCG_Eval_P1_U.pos_rqst - rtwdemo_PCG_Eval_P1_U.fbk_1;
    }
}

```

5 Close the `rtwdemo_PCG_eval_P1` demo model.

You can view any of the files listed below by clicking their links in the HTML report **Contents** pane, or by exploring the generated code subdirectory created in your working directory by the build process.

File	Description
<code>rtwdemo_PCG_Eval_P1.c</code>	C file with step and initialization functions
<code>rtwdemo_PCG_Eval_P1_data.c</code>	C file that assigns values to Real-Time Workshop data structures
<code>ert_main.c</code>	Example main module that includes a simple scheduler
<code>rtwdemo_PCG_Eval_P1.h</code>	H file that defines data structures

File	Description
PCG_Eval_p1_private.h	File that defines data used only by the generated code
rtwdemo_PCG_Eval_P1_types.h	H file that defines the model data structure

Topics for Further Study

- “Supporting Model Referencing” in the Real-Time Workshop documentation
- “Building Executables” in the Real-Time Workshop documentation
- “Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation
- “Working with Signal Groups” in the Simulink documentation
- Simulink Verification and Validation documentation

Configuring the Data Interface

In this section...

- “Introduction” on page 2-28
- “Declaring Data” on page 2-28
- “Using Data Objects in Simulink Models and Stateflow Charts” on page 2-31
- “Adding New Data Objects” on page 2-34
- “Configuring Data Objects” on page 2-35
- “Controlling File Placement of Parameter Data” on page 2-35
- “Enabling Data Objects in Generated Code” on page 2-36
- “Effects of Simulation on Data Typing” on page 2-37
- “Viewing Data Objects in Generated Code” on page 2-39
- “Managing Data” on page 2-42
- “Topics for Further Study” on page 2-42

Introduction

This tutorial explains how to configure the data interface for the generated code of a model. In this tutorial, you learn how to control the following attributes of signals and parameters in the generated code:

- Name
- Data type
- Data storage class

Declaring Data

Most programming languages require that you *declare* data before using it. The declaration specifies the following:

Data Attribute	Description
Scope	The region of the program that has access to the data
Duration	The period during which the data is resident in memory
Data type	The amount of memory allocated for the data
Initialization	An initial value, a pointer to memory, or NULL (if you do not provide an initial value, most compilers assign a zero value or a null pointer)

The following data types are supported for code generation.

Supported Data Types

Name	Description
double	Double-precision floating point
single	Single-precision floating point
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer
Fixed point data types	8-, 16-, 32-bit word lengths

The combination of scope and duration comprises the *storage class* of a data item. The following predefined storage classes are supported for code generation.

Supported Predefined Storage Classes

Name	Description	Parameters Supported	Signals Supported	Data Types
Const	Use const type qualifier in declaration	Y	N	All
ConstVolatile	Use const volatile type qualifier in declaration	Y	N	All
Volatile	Use volatile type qualifier in declaration	Y	Y	All
ExportToFile	Generate and include files, with user-specified name, containing global variable declarations and definitions	Y	Y	All
ImportFromFile	Include predefined header files containing global variable declarations	Y	Y	All
Exported Global	Declare and define variables of global scope	Y	Y	All
Imported Extern	Import a variable defined outside of the scope of the model	Y	Y	All

Supported Predefined Storage Classes (Continued)

Name	Description	Parameters Supported	Signals Supported	Data Types
BitField	Embed Boolean data in a named bit field	Y	Y	Boolean
Define	Represent parameters with a <code>#define</code> macro	Y	N	All
Struct	Embed data in a named structure to encapsulate sets of data	Y	Y	All

Using Data Objects in Simulink Models and Stateflow Charts

Two methods are available for declaring data in Simulink models and Stateflow charts: *data objects* and *direct specification*. This tutorial uses the data object method. Both methods allow full control over the data type and storage class. You can mix the two methods in a single model.

You can use data objects in a variety of ways in the MATLAB and Simulink environment. The tutorial focuses on three types of data objects:

- Signal
- Parameter
- Bus

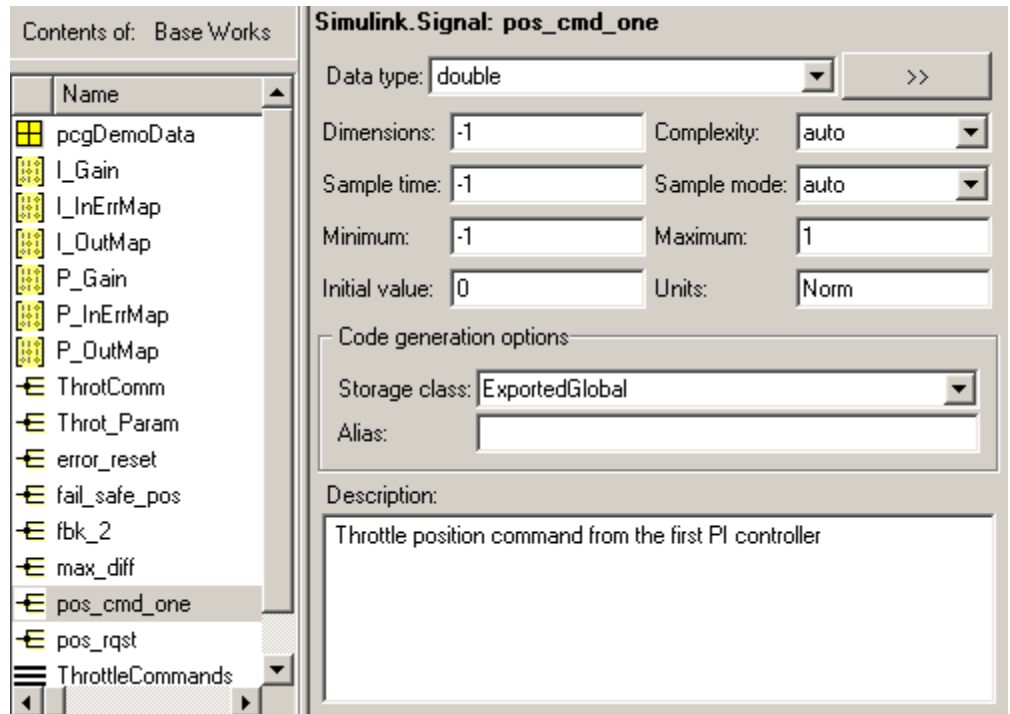
To configure the data interface for your model using the data object method, you define data objects in the MATLAB base workspace and then associate them with your Simulink model or embedded Stateflow chart. When you build your model, the Real-Time Workshop build process uses the associated base workspace data objects in the generated code.

A data object has a mixture of *active* and *descriptive* fields. Active fields affect simulation or code generation. Descriptive fields do not affect simulation or code generation, but are used with data dictionaries and model-checking tools.

- Active fields:
 - Data type
 - Storage class
 - Value (parameters)
 - Initial value (signals)
 - Alias (define a different name in the generated code)
 - Dimension (inherited for parameters)
 - Complexity (inherited for parameters)
- Descriptive fields:
 - Minimum
 - Maximum
 - Units
 - Description

You can create and inspect base workspace data objects by entering commands at the MATLAB command line or by using Model Explorer. Perform the following steps to explore base workspace signal data objects declared for the `rtwdemo_PCG_Eval_P2` demo model.

- 1** Open the `rtwdemo_PCG_Eval_P2` demo model by entering `rtwdemo_PCG_Eval_P2` at the MATLAB command line.
- 2** Open Model Explorer.
- 3** Select **Base Workspace**.
- 4** Select the `pos_cmd_one` signal data object for viewing



You can also view the definition of Simulink signal object `pos_cmd_one` by entering `pos_cmd_one` at the MATLAB command line:

```
pos_cmd_one =

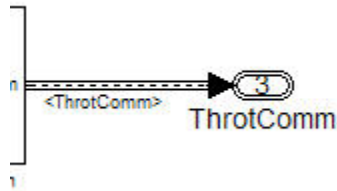
Simulink.Signal (handle)
  RTWInfo: [1x1 Simulink.SignalRTWInfo]
  Description: 'Throttle position command from the first PI controller'
  DataType: 'double'
  Min: -1
  Max: 1
  DocUnits: 'Norm'
  Dimensions: -1
  Complexity: 'auto'
  SampleTime: -1
  SamplingMode: 'auto'
  InitialValue: '0'
```

- 5 To view other signal objects, click the object name in Model Explorer or enter the object name at the MATLAB command line. The following table summarizes object characteristics for this model.

Object Characteristics	pos_cmd_one	pos_rqst	P_InErrMap	ThrotComm*	ThrottleCommands*
Description	Top-level output	Top-level input	Calibration parameter	Top-level output structure	Bus definition
Data type	Double	Double	Auto	Auto	Structure
Storage class	Exported global	Imported extern pointer	Constant	Exported global	None

* ThrottleCommands defines a Simulink Bus object; ThrotComm is the instantiation of the bus. If the bus is a nonvirtual bus, the signal generates a structure in the C code.

As in C, you can use a bus definition (ThrottleCommands) to instantiate multiple instances of the structure. In a model diagram, a bus object appears as a wide line with central dashes, as shown below.



Adding New Data Objects

You can create data objects for named signals, states, and parameters. To associate a data object with a construct, the construct must have a name.

The Data Object Wizard is a tool that finds constructs for which you can create data objects, and then creates the objects for you. The model includes two signals that are not associated with data objects: fbk_1 and pos_cmd_two.

To find the signals and create data objects for them:

- 1** Open the Data Object Wizard by selecting **Tools > Data Object Wizard** in the Model Editor.
- 2** Click the **Find** button to find candidate constructs.
- 3** Click the **Check All** button to select all candidates.
- 4** Click the **Apply Package** button to apply the default Simulink package for the data objects.
- 5** Click the **Create** button to create the data objects.

Configuring Data Objects

The next step is to set the data type and storage class:

- 1** Open Model Explorer and view the base workspace.
- 2** For each object listed in the following table:
 - a** Click the signal name in the **Contents** pane.
 - b** Change the settings in the **Data** pane to match those in the table.
 - c** Click the **Apply** button.

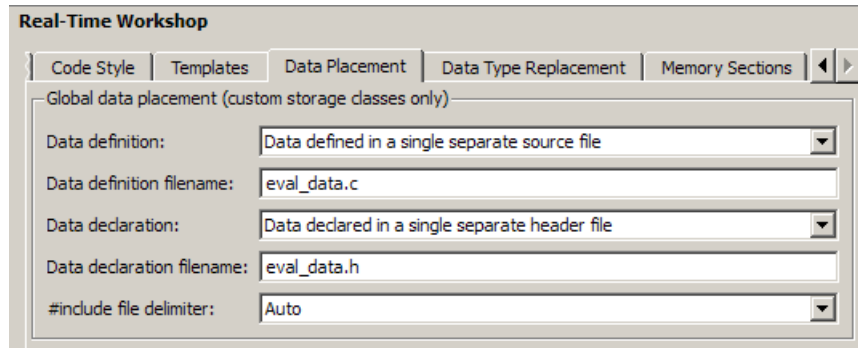
Signal	Data Type	Storage Class
fbk_1	double	ImportedExtern
pos_cmd_two	double	ExportedGlobal

Controlling File Placement of Parameter Data

Real-Time Workshop Embedded Coder software allows you to control the files that define the parameters and constants. In this tutorial, all parameters are in `eval_data.c`.

To change the placement of parameter and constant definitions, set the appropriate data placement options.

- 1 Within Model Explorer, enter the data options in the **Configuration > Real-Time Workshop > Data Placement** pane, as shown in the following figure.



- 2 eval_data.c is shown below:

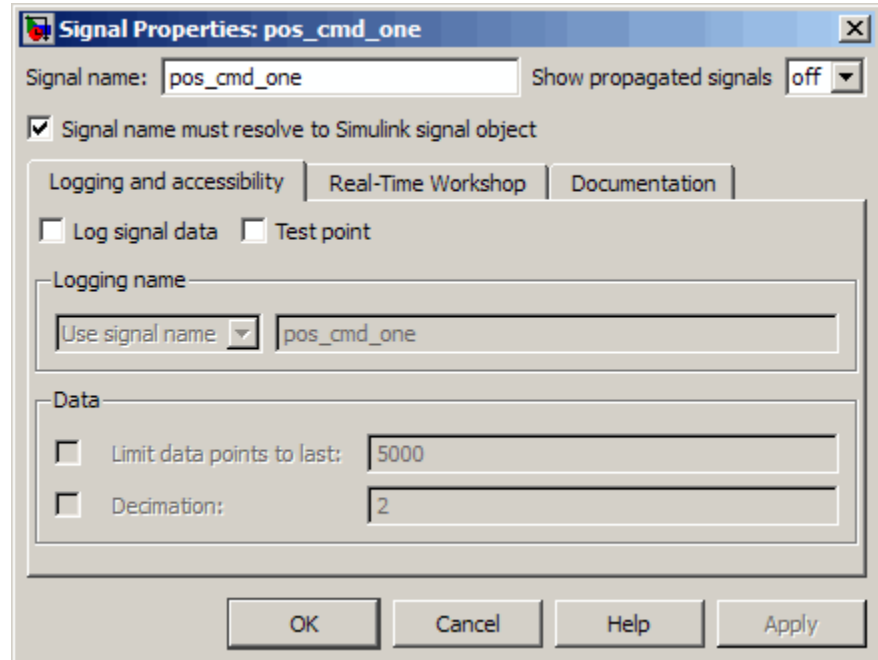
```
/* Const memory section */
17  /* Definition for custom storage class: Const */
18  const real_T I_Gain = -0.03;
19  const real_T I_InErrMap[9] = { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05, 0.25, 0.5,
20    1.0 } ;
21
22  const real_T I_OutMap[9] = { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6, 0.75, 1.0 } ;
23
24  const real_T P_Gain = 0.74;
25  const real_T P_InErrMap[7] = { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25, 1.0 } ;
26
27  const real_T P_OutMap[7] = { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0 } ;
```

Enabling Data Objects in Generated Code

The next step is to ensure that the data objects you have created appear in the generated code:

- 1 In Model Explorer, make sure the **Configuration > Optimizations > Inline parameters** check box is selected.
- 2 Enable a signal in generated code:

- a In the model, right-click the `pos_cmd_one` signal line.
- b Select **Signal Properties**. A Signal Properties dialog box appears.
- c Make sure the **Signal name must resolve to a Simulink signal object** check box is selected.



- 3 Enable all of the signals in the model simultaneously by entering the following at the MATLAB command line:

```
disableimplicitsignalresolution('rtwdemo_PCG_Eval_P2')
```

- 4 Save the model (requires a Stateflow license) for use in the next section.

Effects of Simulation on Data Typing

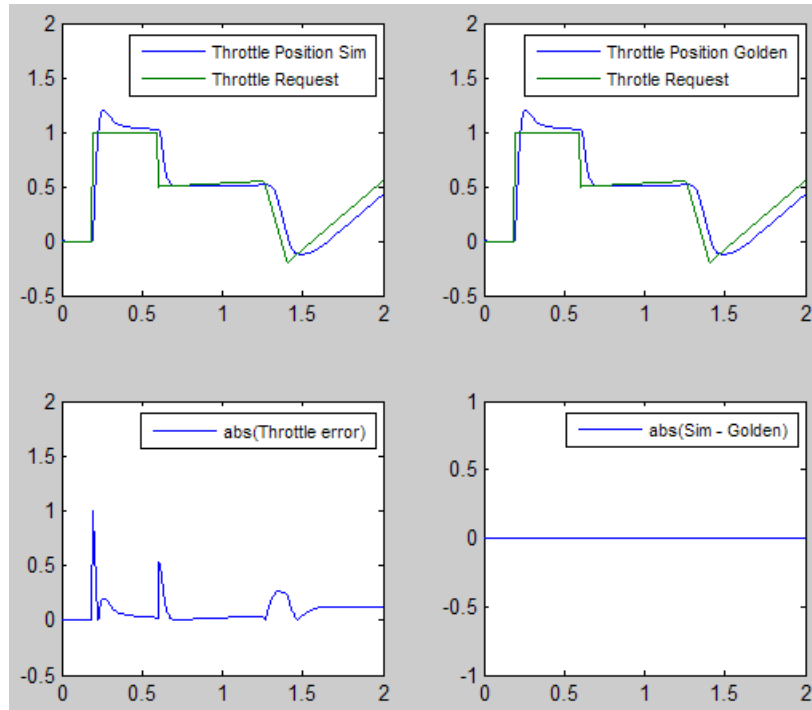
In the `rtwdemo_PCG_Eval_P2` model, all data types are set to `double`. Since Simulink software uses the `double` data type for simulation, you should not expect changes in the model behavior when you run the generated code. You verify this by running the test harness. You update the test harness to

include the `rtwdemo_PCG_Eval_P2` model. That change is the only one made to the test harness.

Note The following procedure requires a Stateflow license.

- 1** Open the test harness by entering `rtwdemo_PCGEvalHarness` at the MATLAB command line.
- 2** Right-click the `Unit_Under_Test Model` block and select **ModelReference Parameters**.
- 3** Set **Model name (without the .mdl extension)** to:
`rtwdemo_PCG_Eval_P2`.
- 4** Click **OK**.
- 5** Simulate the test harness.

The resulting plot shows that the difference between the golden and simulated versions of the model remains zero.



6 Close the `rtwdemo_PCGEval1Harness` model.

Viewing Data Objects in Generated Code

View the `rtwdemo_PCG_Eval_P2.c` file to see how using data objects changes the generated code.

- 1 Generate code from the `rtwdemo_PCG_Eval_P2` model.
- 2 View the generated code for the model step function in `rtwdemo_PCG_Eval_P2.c`, which is in the generated code subdirectory for this model.

The following code for the `rtwdemo_PCG_EVAL_P1_step` function is equivalent to `rtwdemo_PCG_Eval_P2_step` function before the use of data objects. The highlighted portions indicate parameter and structure variables that user defined data objects replace.

```
/* Model step function */
void rtwdemo_PCG_Eval_P1_step(void)
{
    {
        real_T rtb_Sum3;
        real_T rtb_Saturation1;

        /* Sum: '<S2>/Sum2' incorporates:
         * Inport: '<Root>/fbk_1'
         * Inport: '<Root>/pos_rqst'
         */
        rtb_Sum3 = rtwdemo_PCG_Eval_P1_U.pos_rqst - rtwdemo_PCG_Eval_P1_U.fbk_1;

        /* DiscreteIntegrator: '<S2>/Discrete_Time_Integrator1' incorporates:
         * Gain: '<S2>/Int Gain1'
         * Lookup: '<S2>/Integral Gain Shape'
         * Product: '<S2>/Product3'
         */
        rtwdemo_PCG_Eval_P1_B.Discrete_Time_Integrator1 = -0.03 * rt_Lookup((real_T *)
            (&rtwdemo_PCG_Eval_P1_ConstP.pooled4[0]), 9, rtb_Sum3, (real_T *)
            (&rtwdemo_PCG_Eval_P1_ConstP.pooled5[0])) * rtb_Sum3 * 0.001 +
            rtwdemo_PCG_Eval_P1_DWork.Discrete_Time_Integrator1_DSAT;
        ...
    }
}
```

The following code for the `rtwdemo_PCG_Eval_P2_step` function shows that most of the Real-Time Workshop data structures have been replaced with user-defined data objects (highlighted). The local variable `rtb_Sum3` and the state variable `rtwdemo_PCG_Eval_P2_DWork.Discrete_Time_Integrator1_DSAT` use Real-Time Workshop data structures.

```
/* Model step function */
void rtwdemo_PCG_Eval_P2_step(void)
{
    {
        real_T rtb_Sum3;
        /* Sum: '<S2>/Sum2' incorporates:
         * Inport: '<Root>/fbk_1'
         * Inport: '<Root>/pos_rqst'
         */
    }
}
```

```

rtb_Sum3 = (*pos_rqst) - fbk_1;

/* DiscreteIntegrator: '<S2>/Discrete_Time_Integrator1' incorporates:
 * Gain: '<S2>/Int_Gain1'
 * Lookup: '<S2>/Integral_Gain_Shape'
 * Product: '<S2>/Product3'
 */
rtwdemo_PCG_Eval_P2_B.Discrete_Time_Integrator1 = I_Gain * rt_Lookup((real_T*)
(&(I_InErrMap[0])), 9, rtb_Sum3, (real_T *)
(&(I_OutMap[0]))) * rtb_Sum3 *
0.001 + rtwdemo_PCG_Eval_P2_DWork.Discrete_Time_Integrator1_DSTAT;
...

```

3 Close the `rtwdemo_PCG_eval_P2` demo model.

The following table lists the files that the code generator creates:

Files Generated for `rtwdemo_PCG_Eval_P2`

File	Definition	Notes
<code>rtwdemo_PCG_Eval_P2.c</code>	Provides step and initialization function	Uses the defined data objects
<code>eval_data.c</code>	Assigns values to the defined parameters	Has the file name specifically defined
<code>eval_data.h</code>	Provides extern definitions to the defined parameters	Has the file name specifically defined
<code>ert_main.c</code>	Provides scheduling functions	No change
<code>rtwdemo_PCG_Eval_P2.h</code>	Defines data structures	Using data objects shifted some parameters out of this file into <code>user_data.h</code>
<code>PCG_Eval_p2_private.h</code>	Defines private (local) data for the generated functions	Objects now defined in <code>eval_data</code> were removed

Files Generated for rtwdemo_PCG_Eval_P2 (Continued)

File	Definition	Notes
rtwdemo_PCG_Eval_P2_types.h	Defines the model data structure	No change
rtwtypes.h	Provides mapping to data types defined by Real-Time Workshop software	Used for integration with external systems

Managing Data

Data objects exist in the MATLAB base workspace, in a separate file from the model. To save the data manually, enter `save` at the MATLAB command line.

The separation of data from the model provides many benefits:

- One model, multiple data sets:
 - Use of different data types to change the targeted hardware (for example, for floating-point and fixed-point targets)
 - Use of different parameter values to change the behavior of the control algorithm (for example, for reusable components with different calibration values)
- Multiple models, one data set:
 - Sharing of data between Simulink models in a system
 - Sharing of data between projects (for example, transmission, engine, and wheel controllers might all use the same CAN message data set)

Topics for Further Study

- “Working with Data” in the Simulink documentation
- “Creating and Using Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation
- “Managing Placement of Data Definitions and Declarations” in the Real-Time Workshop Embedded Coder documentation

Partitioning Functions in the Generated Code

In this section...

- “Introduction” on page 2-43
- “About Atomic and Virtual Subsystems” on page 2-43
- “Viewing Changes in the Model Architecture” on page 2-44
- “Controlling Function Location and File Placement in Generated Code” on page 2-45
- “Understanding Reentrant Code” on page 2-46
- “Using a Mask to Pass Parameters into a Library Subsystem” on page 2-47
- “Generating Code from an Atomic Subsystem” on page 2-48
- “Generating Code: Full Model vs. Exported Functions” on page 2-49
- “Effect of Execution Order on Simulation Results” on page 2-51
- “Topics for Further Study” on page 2-53

Introduction

This tutorial shows how to associate subsystems in the model with specific function names and files. You examine:

- How to specify function and file names in generated code
- Parts of generated code that you must have for integration
- How to generate code for atomic subsystems
- Data that you must have to execute a generated function

About Atomic and Virtual Subsystems

The models in “Understanding the Demo Model” on page 2-5 and “Configuring the Data Interface” on page 2-28 use *virtual subsystems*. Virtual subsystems visually organize blocks but have no effect on the behavior of the model. *Atomic subsystems* evaluate all included blocks as a unit. In addition, atomic subsystems allow you to specify additional function partitioning information. Atomic subsystems display graphically with a bold border.

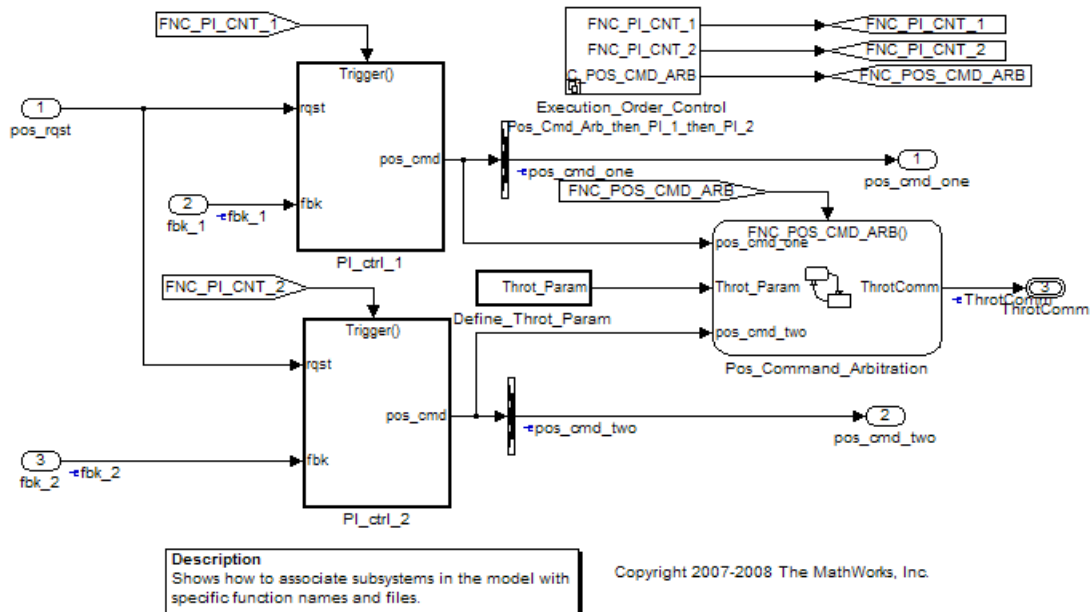
Viewing Changes in the Model Architecture

This section shows you how to replace the virtual subsystems in the model with *function-call subsystems*. Function-call subsystems:

- Are always atomic subsystems
- Allow the direct control of subsystem execution order
- Are associated with a function-call signal
- Are executed when the function-call signal is triggered

You might have to exert direct control over execution order if you intend the model to match an existing system with a specific execution order.

The following figure of the `rtwdemo_PCG_Eval_P3` model identifies function-call subsystems as `PI_ctrl_1`, `PI_ctrl_2`, and `Pos_Command_Arbitration`.



The `Execution_Order_Control` subsystem has been added to the model. It is a Stateflow chart that models the calling functionality of a scheduler. It

controls the execution order of the function-call subsystems. Later, this tutorial examines how changing execution order can change the simulation results.

Four signal conversion blocks were added to the outputs for the PI controllers to make the functions reentrant.

Controlling Function Location and File Placement in Generated Code

In “Understanding the Demo Model” on page 2-5 and “Configuring the Data Interface” on page 2-28, Real-Time Workshop software generates a single *model_step* function that contains all the control algorithm code. However, many applications require a greater level of control over the location of functions in the generated code. By using atomic subsystems, you can specify multiple functions within a single model. You specify this information by modifying subsystem parameters, shown in the following figure.

Parameter	What the Parameter Does
Treat as atomic unit	Enables other submenus. This parameter is automatically selected and grayed out for atomic subsystems.
Sample time	Specifies a sample time. Not present for function-call subsystems.
Real-Time Workshop system code	Depends on setting, as follows.
	Auto: Real-Time Workshop software determines how the subsystem appears in the generated code. This is the default.
	Inline: Real-Time Workshop software places the subsystem code inline with the rest of the model code.
	Function: Real-Time Workshop software generates the code for the subsystem as a function.
	Reusable function: Real-Time Workshop software generates a reusable function from the subsystem. Passes all input and output into the function by argument or by reference. Does not pass global variables into the function.

Parameter	What the Parameter Does
Real-Time Workshop function name options	If you select Function or Reusable function , function name options are enabled as follows.
	Auto: Real-Time Workshop software determines the function.
	Use subsystem name: Base the function name on the subsystem name.
	User Specified: You specify a unique file name.
Real-Time Workshop file name options	If you select Function or Reusable function , file name options are enabled as follows.
	Auto: Real-Time Workshop software generates the function code within the generated code from the parent system or, if the parent of the subsystem is the model itself, within the <i>model.c</i> file.
	Use subsystem name: Real-Time Workshop software generates a separate file and names it with the name of the subsystem or library block.
	Use function name: Real-Time Workshop software generates a separate file and names it with the function name specified for Real-Time Workshop function name options .
	User Specified: You specify a unique file name.
Function with separate data	Enabled when you set Real-Time Workshop system code to Function . If selected, Real-Time Workshop Embedded Coder software generates subsystem function code in which the internal data for an atomic subsystem is separated from its parent model and is owned by the subsystem.

Understanding Reentrant Code

Real-Time Workshop Embedded Coder software supports *reentrant code*. Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Operating systems and other system software that uses multithreading to handle concurrent events use reentrant code.

Reentrant code does not maintain state data: no persistent variables are in the function. Calling programs maintain their state variables and pass them into the function. Any number of users or processes can share one copy of a reentrant routine.

To generate reentrant code, you must first specify the subsystem as a candidate for reuse by specifying the subsystem parameters in the Function Block Parameters dialog box.

In some cases, the configuration of the model prevents Real-Time Workshop software from generating reusable code. Common issues that prevent the generation of reentrant code and corresponding solutions follow.

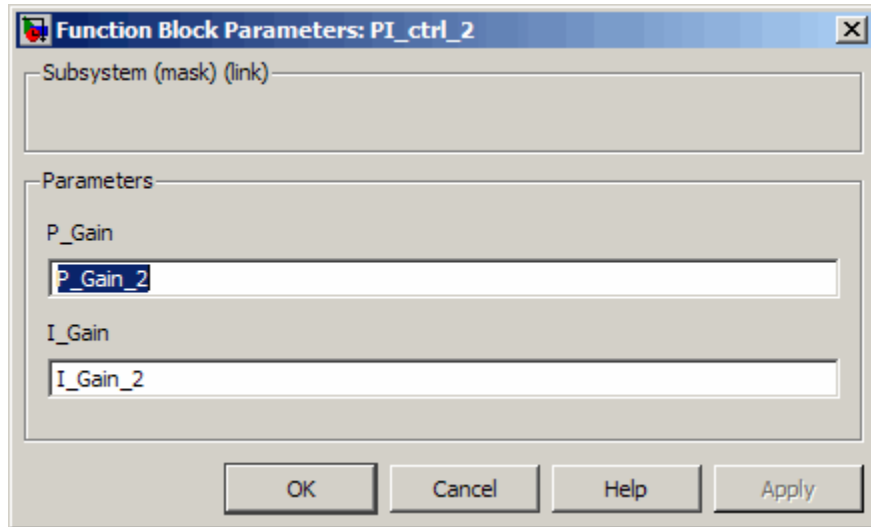
Cause	Solution
Use of global data on the output of the subsystem	Add a Signal Conversion block between the subsystem and the signal definition.
Passing data into the system as pointers	In Model Explorer, select the Configuration > Model Referencing > Pass fixed-size scalar root inputs by value check box.
Use of global data inside the subsystem	Use a port to pass the global data in and out of the subsystem.

Using a Mask to Pass Parameters into a Library Subsystem

Subsystem *masks* enable Simulink software to define subsystem parameters outside the scope of a library block. By changing the parameter value at the top of the library, the same library is usable with multiple sets of parameters within the same model.

When a subsystem is reusable and masked, Real-Time Workshop software passes the masked parameters into the reentrant code as arguments. Real-Time Workshop software fully supports the use of data objects in masks. The data objects are used in the generated code.

In the `rtwdemo_PCG_Eval_P3` model, the `PI_ctrl_1` and `PI_ctrl_2` subsystems are masked. The value of the P and I gains are set in the subsystem Mask Parameters dialog box, shown in the following figure. Simulink creates two new data objects: `P_Gain_2` and `I_Gain_2`.



Generating Code from an Atomic Subsystem

In “Understanding the Demo Model” on page 2-5 and “Configuring the Data Interface” on page 2-28, the Real-Time Workshop software generates code at the model root level. In addition to building at the system level, it is possible to build at the subsystem level

You start a subsystem build from the right-click context menu. Three different options are available for a subsystem build.

Build Option	What the Option Does
Build Subsystem	Treats the subsystem as a separate model. The full set of source C files and header files are created for the subsystem. Does not support function-call subsystems.
Generate S-Function	Generates C code for the subsystem and creates an S-function wrapper. You can then simulate the code in the original Simulink model. Does not support function-call subsystems.
Export Functions	Generates C code without the scheduling code associated with the Build Subsystem option. Export functions is required when building subsystems that use triggers.

Generating Code: Full Model vs. Exported Functions

In this section, you compare the files generated for the full model build with files generated for exported functions. This module also examines how the masked data appears in the code.

1 Open `rtwdemo_PCG_Eval_P3`.

2 Generate code for the model.

The code generator creates code for the `rtwdemo_PCG_Eval_P3` model.

3 Export a function for the `PI_ctrl_1` subsystem:

- a** In the Model Editor, right-click `PI_ctrl_1` and select **Real-Time Workshop > Export Functions**.

The **Build code for Subsystem: PI_ctrl_1** dialog box opens.

- b** Click the **Build** button.

Code is generated for `PI_ctrl_1`.

4 If you have a Stateflow license, export a function for the `Pos_Command_Arbitration` chart:

- a** In the Model Editor, right-click `Pos_Command_Arbitration` and select **Real-Time Workshop > Export Functions**.

The **Build code for Subsystem: Pos_Command_Arbitration** dialog box opens.

- b** Click the **Build** button.

Code is generated for Pos_Command_Arbitration.

- 5** Examine the generated code listed in the table by locating and opening the files in the respective generated code subdirectories.

File	Full Build	PI_ctrl_1	Pos_Command_Arbitration (requires Stateflow license)
rtwdemo_PCG_Eval_P3.c	Yes Step function	No	No
PI_ctrl_1.c	No	Yes Trigger function	No
Pos_Command_Arbitration.c (requires Stateflow license)	No	No	Yes Init and Function
PI_Ctrl_Reusable.c	Yes Called by main	Yes Called by PI_ctrl_1	No
ert_main.c	Yes	Yes	Yes
eval_data.c	Yes*	Yes*	No Eval data not used in diagram

* The content of eval_data.c differs between the full model and export function builds. The full model build includes all parameters that the model uses while the export function contains only variables that the subsystem uses.

Masked Data in the Generated Code

The code in rtwdemo_PCG_Eval_P3.c illustrates how data objects from the mask (P_Gain and I_Gain) and P_Gain_2 and I_Gain_2 pass into the reentrant code.

```

PI_Cntrl_Reusable((*pos_rqst), fbk_1, &rtwdemo_PCG_Eval_P3_B->PI_ctrl1_1,
                  &rtwdemo_PCG_Eval_P3_DWork->PI_ctrl1_1, I_Gain, P_Gain);
PI_Cntrl_Reusable((*pos_rqst), fbk_2, &rtwdemo_PCG_Eval_P3_B->PI_ctrl1_2,
                  &rtwdemo_PCG_Eval_P3_DWork->PI_ctrl1_2, I_Gain_2, P_Gain_2);

```

Effect of Execution Order on Simulation Results

Without explicit control, Simulink software sets the execution order of the subsystems as:

- 1 PI_ctrl1_1
- 2 PI_ctrl1_2
- 3 Pos_Cmd_Arbitration

You use the test harness to see the effect of the execution order on the simulation results. The `Execution_Order_Control` subsystem is a configurable subsystem with two configurations that change the execution order of the subsystems.

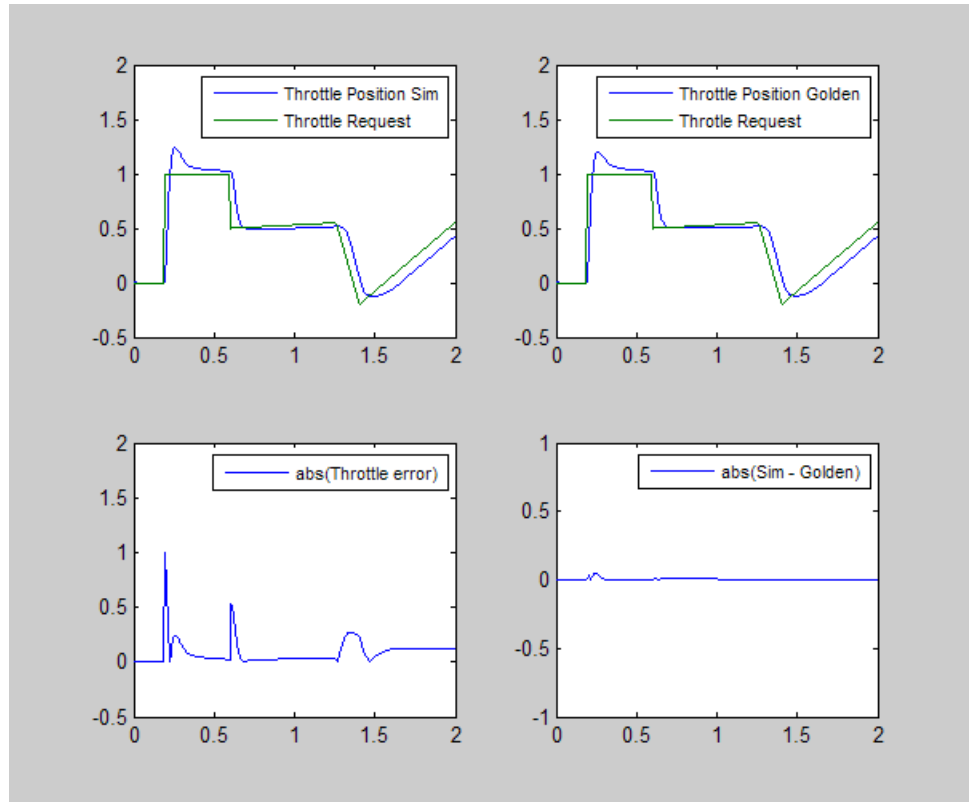
Note The following procedure requires a Stateflow license.

Change the execution order as follows:

- 1 In the `rtwdemo_PCG_Eval_P3` model, set the execution order to `PI_ctrl1_1`, `PI_ctrl1_2`, `Pos_cmd_Arbitration`:
 - a Right-click the `Execution_Order_Control` subsystem.
 - b On the **Block Choice** menu, select **PI_1_then_PI_2_then_Pos_Cmd_Arb.**
- 2 Save `rtwdemo_PCG_Eval_P3`.
- 3 Open the test harness, `rtwdemo_PCGEvalHarness`.
- 4 Right-click the `Unit_Under_Test Model` block and select **ModelReference Parameters**.

- 5** Set **Model name (without the .mdl extension)** to:
rtwdemo_PCG_Eval_P3.
- 6** Click **OK**.
- 7** Run the test harness.
- 8** In `rtwdemo_PCG_Eval_P3`, change the execution order to
`Pos_cmd_Arbitration_then_PI_1_then_PI_2`.
- 9** Run the test harness again.

A slight variation exists in the output results depending on the order of execution. The difference is most noticeable when the desired input changes.



10 Close the `rtwdemo_PCG_Eval_P3` model.

Topics for Further Study

- “Architecture Considerations” in the Real-Time Workshop documentation
- “Integrating External Code With Generated C and C++ Code” in the Real-Time Workshop documentation
- “Exporting Function-Call Subsystems” in the Real-Time Workshop Embedded Coder documentation
- “Controlling Generation of Function Prototypes” in the Real-Time Workshop Embedded Coder documentation
- “Working with Block Masks” in the Simulink documentation

Calling External C Functions from the Model and Generated Code

In this section...

- “Introduction” on page 2-54
- “Including Preexisting C Functions in a Simulink Model” on page 2-54
- “Creating a Block That Calls a C Function” on page 2-55
- “Validating the External Code in the Simulink Environment” on page 2-56
- “Validating the C Code as Part of the Simulink Model” on page 2-58
- “Calling the C Function from the Generated Code” on page 2-59
- “Topics for Further Study” on page 2-60

Introduction

This tutorial introduces the Legacy Code Tool as a method for calling external functions. The Legacy Code Tool enables you to call the external function from within the simulation and in the generated code.

In this tutorial, you examine:

- How to evaluate a C function as part of a Simulink model simulation
- How to call a C function from code that Real-Time Workshop software generates

Including Preexisting C Functions in a Simulink Model

Simulink models are one part of Model-Based Design. For many applications, a design also includes a set of C functions that you have created, tested, and validated. The ability to integrate these functions easily into a Simulink model and generated code is critical to using Simulink software in the controls development process.

This section demonstrates how to create a custom Simulink block that calls an existing C function. Once the block is part of the model, you can take advantage of the simulation environment to test the system further.

In “Creating a Block That Calls a C Function” on page 2-55, you replace the Lookup Table blocks in the PI controllers with calls to an existing C function. The function is defined in the files

```
matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.c
matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.h
```

Note *matlabroot* represents the name of your top-level MATLAB installation directory.

Creating a Block That Calls a C Function

To specify a call to an existing C function, you use an S-Function block. You can automate the process of creating the S-Function block by using the Simulink Legacy Code Tool. Using this tool, you specify an interface for your existing C function. The tool then uses that interface to automate creation of an S-Function block.

Complete the steps below to create an S-Function block for an existing C function `SimpleTable.c`.

- 1 Copy the `SimpleTable.c` and `SimpleTable.h` files to your working directory.
- 2 Create an S-Function block that calls the specified function at each time step during simulation:
 - a Create the function interface definition structure at the command line:

```
def=legacy_code('initialize')
```

The data structure `def` defines the function interface to the existing C code.

- b Populate the function interface definition structure by entering the following commands:

```
def.OutputFcnSpec=['double y1 = SimpleTable(double u1,',...
  'double p1[], double p2[], int16 p3)'];
def.HeaderFiles = {'SimpleTable.h'};
def.SourceFiles = {'SimpleTable.c'};
```

```
def.SFunctionName = 'SimpTableWrap';
```

- c Create the S-function:

```
legacy_code('sfcn_cmex_generate',def)
```

- d Compile the S-function:

```
legacy_code('compile',def)
```

- e Create the S-Function block:

```
legacy_code('slblock_generate',def)
```

Simulink creates a new model that contains the `SimpTableWrap` block.

Tip Creating the S-Function block is a one-time task. Once the block exists, you can reuse it in any model.

- 3 Save the model to your working directory as: `s_fun_simpablewrap.mdl`.

- 4 Create the TLC file for the S-Function block:

```
legacy_code('sfcn_tlc_generate',def);
```

The TLC file is the component of an S-function that specifies how Real-Time Workshop software generates code for the block.

For more information on using the Legacy Code Tool, see “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation.

Validating the External Code in the Simulink Environment

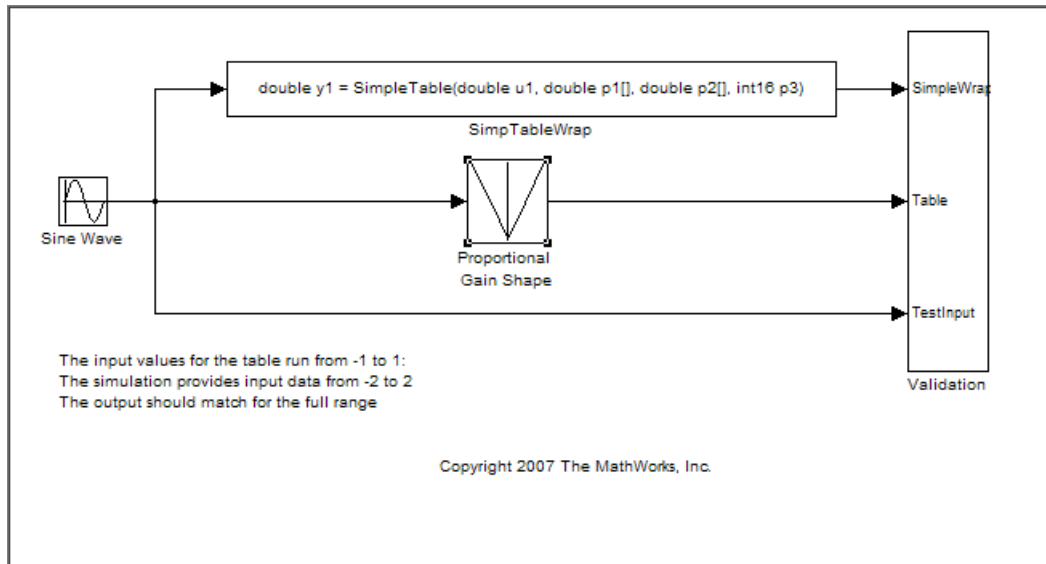
When you integrate existing C code with a Simulink model, always validate the results before using the code.

In this tutorial, you replace Lookup Table blocks with an existing C function. To validate the replacement, you compare simulation results from the Lookup

Table block with results from the new S-Function block you created in the preceding section.

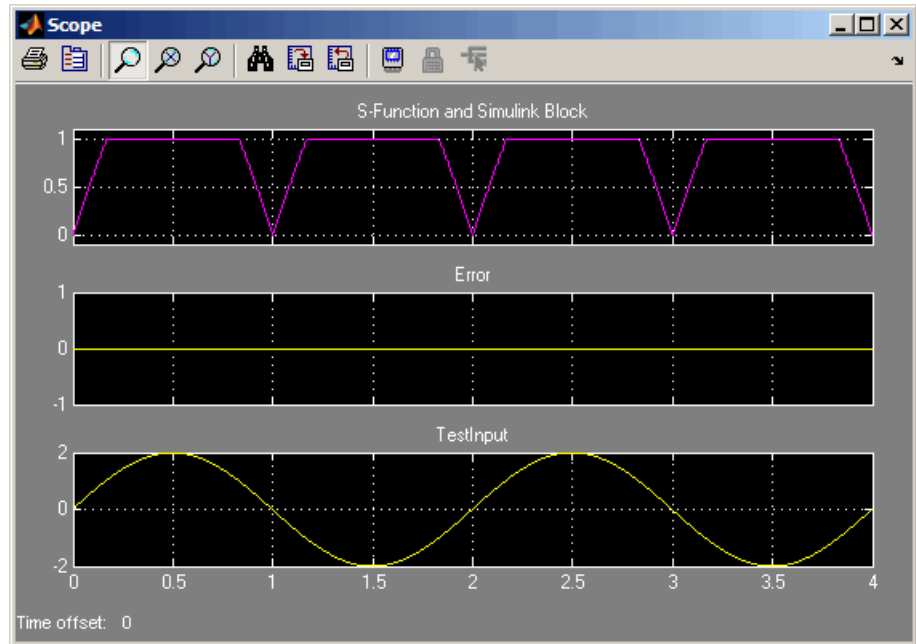
1 Open the validation model, `rtwdemo_ValidateLegacyCodeVrsSim`.

- The Sine Wave block produces output values from [-2 : 2].
- The input range of the lookup table is from [-1 : 1].
- The output from the lookup table is the absolute value of the input.
- The lookup table output clips the output at the input limits.



2 Run the validation model.

The following figure shows the validation results. The existing C code and the Simulink table block provide the same output values.



Validating the C Code as Part of the Simulink Model

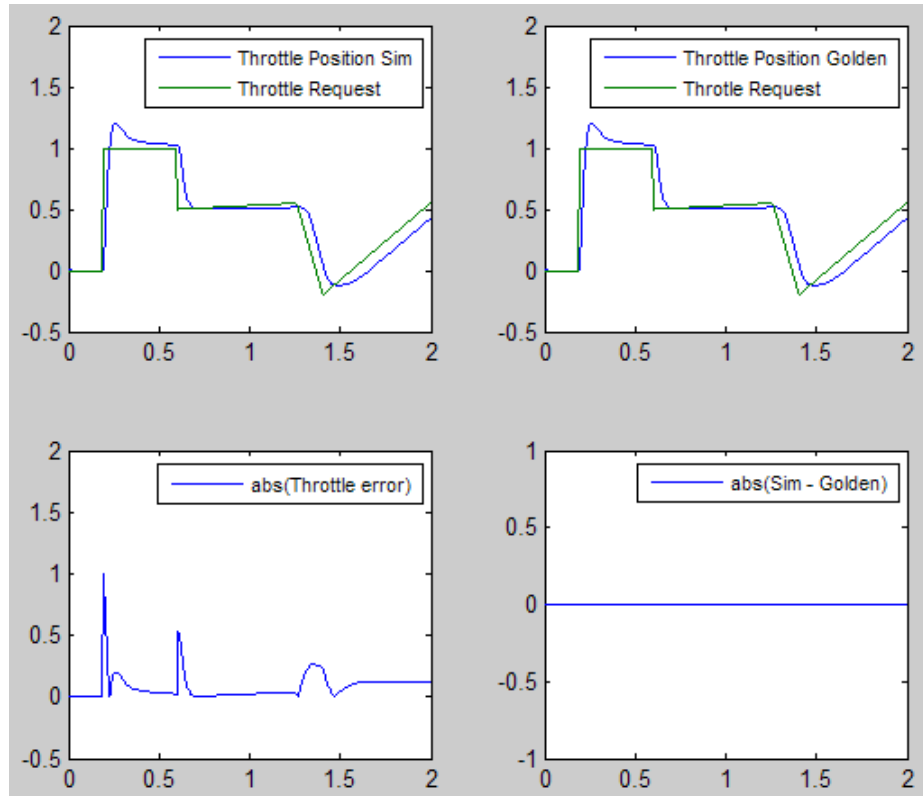
After you validate the functionality of the existing C function code as a standalone component, validate the S-function in the model. Use the test harness model to complete the validation.

Note The following procedure requires a Stateflow license.

- 1 Open the test harness, `rtwdemo_PCGEvalHarness`.
- 1 Right-click the `Unit_Under_Test` Model block and select **ModelReference Parameters**.
- 2 Set **Model name (without the .mdl extension)** to:
`rtwdemo_PCG_Eval_P4`.
- 3 Click **OK**.

4 Run the test harness.

The simulation results match the expected golden values.

**5** Close the `rtwdemo_PCGEvalHarness` model.

Calling the C Function from the Generated Code

Real-Time Workshop software uses the TLC file to process the S-Function block like any other block in the system. Calls to the C code of the S-Function block:

- Can use data objects

- Are subject to *expression folding*, an operation that combines multiple computations into a single output calculation

1 Open the `rtwdemo_PCG_Eval_P4` model.

2 Build the code for the model.

3 Examine the generated code (`PI_Control_Reusable.c`).

The generated code now calls the `SimpleTable` C function.

Before the integration, the generated code called `rt_Lookup`.

```
localB->Discrete_Time_Integrator1 = rtp_Masked_I_Gain * rt_Lookup((real_T *)  
    (&(I_InErrMap[0])), 9, rtb_Sum3, (real_T *)(&(I_OutMap[0]))) * rtb_Sum3 *  
    0.001 + localDW->Discrete_Time_Integrator1_DSTAT;
```

After the integration, the generated code calls the C function `SimpleTable`.

```
localB->Discrete_Time_Integrator1 = I_Gain * SimpleTable( rtb_Sum2,  
    (&(I_InErrMap[0])), (&(I_OutMap[0])), 9) * rtb_Sum2 * 0.001 +  
    localDW->Discrete_Time_Integrator1_DSTAT;
```

4 Close the `rtwdemo_PCG_eval_P4` demo model.

Topics for Further Study

- “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation
- “Integrating External Code With Generated C and C++ Code” in the Real-Time Workshop documentation

Integrating the Generated Code into the External Environment

In this section...

- “Introduction” on page 2-61
- “Building and Collecting the Required Data and Files” on page 2-61
- “Integrating the Generated Code into an Existing System” on page 2-62
- “About the Integration Environment” on page 2-62
- “Matching the System Interfaces” on page 2-64
- “Matching Function-Call Interfaces” on page 2-66
- “Building a Project in the Eclipse Environment” on page 2-67
- “Topics for Further Study” on page 2-68

Introduction

This tutorial provides an overview of the external build process.

In this tutorial, you explore:

- How to collect files that you must have for building outside of the Simulink environment
- How to interface with external variables and functions

Building and Collecting the Required Data and Files

The code that Real-Time Workshop software generates depends on support files that The MathWorks provides. If you need to relocate generated code to another development environment, such as a dedicated build system, you must also relocate the required support files. You can automatically collect all generated and necessary support files and package them in a zip file by using the Real-Time Workshop `packNGo` utility. This utility uses tools for customizing the build process after code generation, including a `buildinfo_data` structure, and a `packNGo` function to find and package all files that you need to build an executable image, including external files you define in the **Real-Time Workshop > Custom Code** pane of the

Configuration Parameters dialog box. The utility packages the files in a standard zip file. The `buildinfo` MAT-file is saved automatically in the directory `model_ert_rtw`.

- 1 Open the `rtwdemo_PCG_Eval_P5` model.
- 2 Generate code for the model `rtwdemo_PCG_Eval_P5`.

The model is configured to run `packNGo` automatically after code generation.

- 3 To generate the zip file manually, do these steps at the MATLAB command line:
 - a Load the `buildInfo.mat` file (located in the `rtwdemo_PCG_Eval_P5_ert_rtw` subdirectory).
 - b Enter the `packNGo(buildInfo)` command.

The number of files in the zip file depends on the version of Real-Time Workshop Embedded Coder software and the configuration of the model you use. The compiler does not require all of the files in the zip file. The compiled executable size (RAM/ROM) is dependent on the link process. You must configure the linker to include only required object files.

Integrating the Generated Code into an Existing System

This section covers tasks required to integrate the generated code into an existing code base. For this evaluation, you use the Eclipse Integrated Development Environment (IDE) and Cygwin's GCC compiler. The required integration tasks are common to all integration environments.

About the Integration Environment

A full embedded controls system has multiple components, both hardware and software. Control algorithms are just one type of component. The other standard types of components include:

- An operating system (OS)
- A scheduling layer

- Physical hardware I/O
- Low-level hardware device drivers

In general, Real-Time Workshop Embedded Coder software does not generate code for any of these components. Instead, it generates interfaces that connect with the components. The MathWorks provides hardware interface block libraries for many common embedded controllers. For examples, see the Target Support Package block libraries.

For this evaluation, the following main function demonstrates how you can build a full system.

It is a simple main function that performs the basic actions to exercise the code. It is *not* an example of an actual application main function.

Note The file is available at:

`matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_5_files/example_main.c`

where `matlabroot` represents the name of your top-level MATLAB installation directory.

```
int_T main(void)
{
    /* Initialize model */
    PC_Pos_Command_Arbitration_Init(); /* Set up the data structures for chart*/
    PCG_Eval_P5_Define_Throt_Param(); /* SubSystem: '<Root>/Define_Throt_Param' */
    defineImportData();             /* Defines the memory and values of inputs */

    do /* This is the "Schedule" loop.
        Functions would be called based on a scheduling algorithm */
    {
        /* HARDWARE I/O */

        /* Call control algorithms */
        PI_Cntrl_Reusable((*pos_rqst),fbk_1,&PCG_Eval_P5_B.PI_ctrl_1,
                        &PCG_Eval_P5_DWork.PI_ctrl_1);
        PI_Cntrl_Reusable((*pos_rqst),fbk_2,&PCG_Eval_P5_B.PI_ctrl_2,
```

```
        &PCG_Eval_P5_DWork.PI_ctrl_2);
pos_cmd_one = PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
pos_cmd_two = PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

PCG_Eva_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two);

simulationLoop++;
} while (simulationLoop < 2);
return 0;
}
```

Functions of `example_main.c` include the following:

- Defines function interfaces (function prototypes)
- Includes required files for data definition
- Defines extern data
- Initializes data
- Calls simulated hardware
- Calls algorithmic functions

The order of execution of functions in `example_main.c` matches the order in which the test harness and `rtwdemo_PCG_Eval_P5.h` call the subsystems. If you change the order of execution in `example_main.c`, results from the executable image differ from simulation results.

Matching the System Interfaces

Integration requires matching both the *Data* and *Function* interfaces of the generated code and the existing system code. In this example, the `example_main.c` file defines the data through `#includes` and calls the functions from the generated code.

Specifying Input Data

The system has three input signals: `pos_rqst`, `fbk_1`, and `fbk_2`. The two feedback signals are imported externs and the position signal is an imported extern pointer. Because of how the signals are defined, Real-Time Workshop

software does not create variables for them. Instead, the signal variables are defined in a file that is external to the MATLAB environment.

For the tutorial, the `defineImportedData.c` file, a simple C stub, defines the signal variables. The generated code has access to the data from the `extern` definitions in the `rtwdemo_PCG_Eval_P5_Private.h` file. In a real system, the data would come from other software components or from hardware devices.

- 1 Open `matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/-stage_5_files/defineImportedData.c`:

```
/* Define imported data */
#include "rtwtypes.h"
real_T fbk_1;
real_T fbk_2;
real_T dummy_pos_value = 10.0;
real_T *pos_rqst;
void defineImportData(void)
{
    pos_rqst = &dummy_pos_value;
}
```

- 2 In your working directory, open `rtwdemo_PCG_Eval_P5_private.h`:

```
/* Imported (extern) block signals */
extern real_T fbk_1;           /* '<Root>/fbk_1' */
extern real_T fbk_2;           /* '<Root>/fbk_2' */

/* Imported (extern) pointer block signals */
extern real_T *pos_rqst;       /* '<Root>/pos_rqst' */
```

Specifying Output Data

The system does not require you to do anything with the output data. However, you can access the data by referring to the `rtwdemo_PCG_Eval_P5.h` file.

Open `rtwdemo_PCG_Eval_P5.h`.

The “Testing the Generated Code” on page 2-69 section shows how to save the output data to a standard log file.

Accessing Additional Data

Real-Time Workshop Embedded Coder software creates several data structures during the code generation process although this tutorial does not require access to these structures. Data elements include:

- Block state values (integrator, transfer functions)
- Local parameters
- Time

The following table lists the most common Real-Time Workshop data structures. Depending on the configuration of the model, some or all of these structures appear in the generated code. In this tutorial, `rtwdemo_PCG_Eval_P5.h` file declares the data.

Data Type	Data Name	Data Purpose
Constants	<code>model_cP</code>	Constant parameters
Constants	<code>model_cB</code>	Constant block I/O
Output	<code>model_U</code>	Root and atomic subsystem input
Output	<code>model_Y</code>	Root and atomic subsystem output
Internal data	<code>model_B</code>	Value of block output
Internal data	<code>model_D</code>	State information vectors
Internal data	<code>model_M</code>	Time and other system level data
Internal data	<code>model_Zero</code>	Zero-crossings
Parameters	<code>model_P</code>	Parameters

Matching Function-Call Interfaces

By default, the Real-Time Workshop software generates functions that have a `void Func(void)` interface. If you configure the model or atomic subsystem as reentrant code, the Real-Time Workshop software creates a more complex

function prototype. As shown below, the `example_main` function is configured to call the functions with the correct input arguments.

```

rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
    ((*pos_rqst), fbk_1, &rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
    &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
PI_Cntrl_Reusable((*pos_rqst), fbk_2, &rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
    &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,

    &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);

```

Calls to the `PI_Cntrl_Reusable` function use a mixture of user-defined variables and Real-Time Workshop structures. The Real-Time Workshop build process defines structures in `rtwdemo_PCG_Eval_P5.h`. The preceding code fragment also shows how the structures map onto user-defined variables.

Building a Project in the Eclipse Environment

This tutorial uses the Eclipse IDE to build the embedded system.

- 1 Create a build directory (`Eclipse_Build_P5`).

Note If you have not generated code for the model, or the zip file does not exist, complete the steps in “Building and Collecting the Required Data and Files” on page 2-61 before continuing to the next step.

- 2 Unzip the file `rtwdemo_PCG_Eval_P5.zip` into your build directory.
- 3 Delete the following files, which `example_main.c` replaces:
 - `rtwdemo_PCG_Eval_P5.c`
 - `ert_main.c`
 - `rt_logging.c`

- 4 Follow the link for instructions on Appendix A, “Installing and Using an IDE for the Integration and Testing Tutorials (Optional)”.

You can use the Eclipse debugger to step through and evaluate the execution behavior of the generated C code. “Testing the Generated Code” on page 2-69 includes an example on how to exercise the model with input data.

- 5 Close the `rtwdemo_PCG_eval_P5` demo model.

Topics for Further Study

“Relocating Code to Another Development Environment”

Testing the Generated Code

In this section...

“Introduction” on page 2-69

“Methods for Validating Generated Code” on page 2-69

“Reusing Test Data: Test Vector Import/Export” on page 2-71

“Testing via Software-in-the-Loop (S-Functions)” on page 2-72

“Configuring the System for Testing via Test Vector Import/Export” on page 2-74

“Testing with Test Vector Import/Export Using the Eclipse Environment” on page 2-76

“Testing via Processor-in-the-Loop (PIL)” on page 2-77

Introduction

This tutorial shows two approaches for validating the generated code: the use of system-level S-functions and running code in an external environment.

In this tutorial, you examine:

- Different methods available for testing generated code
- How to test generated code in the Simulink environment
- How to test generated code outside of the Simulink environment

Methods for Validating Generated Code

Simulink software supports multiple system-testing methods for validating the behavior of generated code.

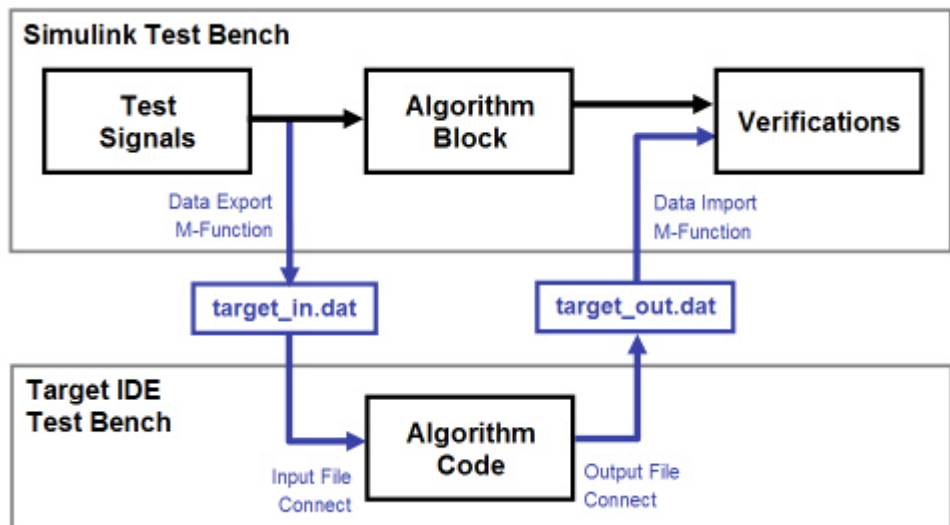
Test Method	What the Method Does	Advantages	Disadvantages
Microsoft Windows run-time executable	Generate a Windows executable and run the executable from the command prompt.	Easy to create. Can use C debugger to evaluate code.	Emulates only part of the target hardware
Software in the loop (SIL)	Use an S-function wrapper to include the generated code back into the Simulink model.	Easy to create Allows you to reuse the Simulink test environment. Can use C debugger to evaluate code.	Emulates only part of the target hardware
Processor in the loop (PIL)	Download code to a target processor and communicate with it from Simulink. See “How PIL Simulation Works” in the Real-Time Workshop Embedded Coder documentation.	Allows you to reuse the Simulink test environment. Can use C debugger with the simulation. Actual processor is used.	Requires additional steps to set up test environment.
On-target rapid prototyping	Run generated code on the target processor as part of the full system.	Can determine actual hardware constraints. Allows testing of component within the full system. Processor runs in real time.	Requires hardware. Requires additional steps to set up test environment.

Reusing Test Data: Test Vector Import/Export

When the unit under test is in the Simulink environment, you can easily reuse test data. However, test data can be reused outside of the Simulink environment. To accomplish this task:

- Save the Simulink data into a file.
- Format the data in a way that the system code can access.
- Read the data file as part of the system code procedures.

Likewise, you can reuse external environment data in the Simulink test environment if you save the data from the external environment in a format that MATLAB software can read. In this example, the `hardwareInputs.c` file contains the output data from the Signal Builder block in the test harness model.

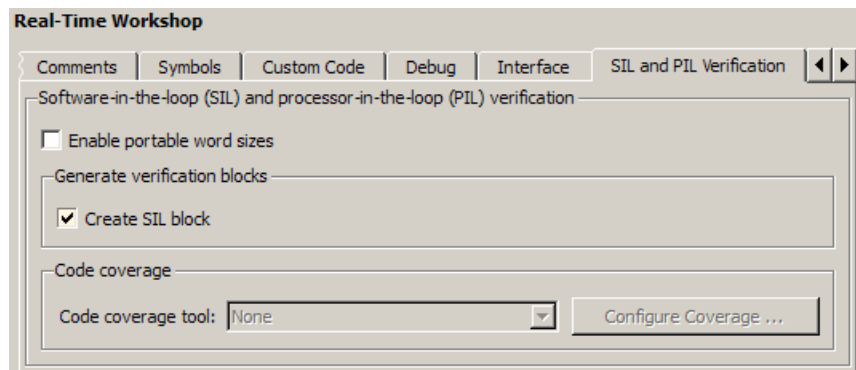


Testing via Software-in-the-Loop (S-Functions)

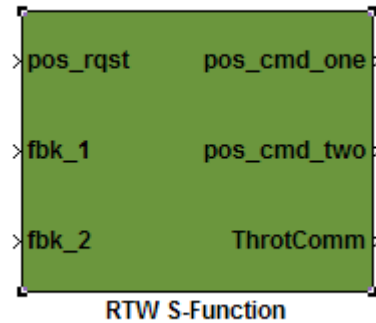
Creating the S-Function

Simulink software can automatically create an S-function wrapper for generated C code. You can enable this option from Model Explorer:

- 1 Open the `rtwdemo_PCG_Eval_P6` model.
- 2 In Model Explorer, select the **Configuration > Real-Time Workshop > SIL and PIL Verification** pane.
- 3 Select the **Create SIL block** check box.



- 4 In the **Real-Time Workshop > General** pane, make sure the **Generate code only** check box is cleared.
- 5 Build the code for the model.
Building the model creates an S-function.



After you create the S-function, you can save it as a model, and then use it with the test harness.

Running the S-Function

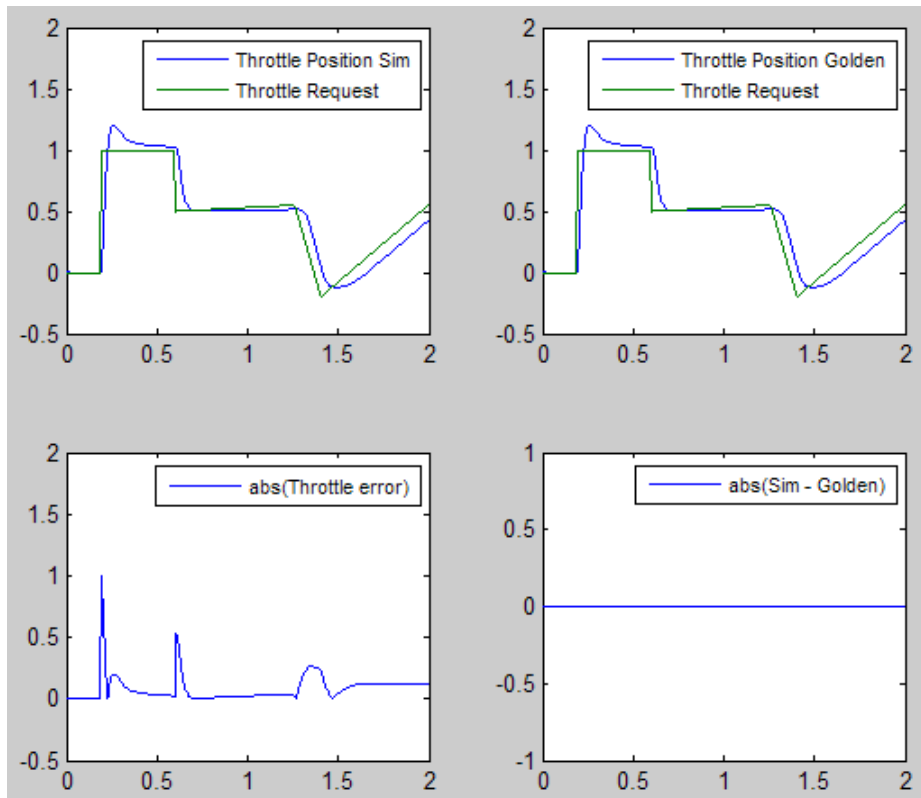
In the following tutorial, the demo model is a version of the test harness with a modification: the Model block is replaced with the automatically generated S-function. Model blocks and the automatically generated S-function model are based on the same technological infrastructure. As a result, Model blocks cannot include automatically generated S-functions. However, Model blocks can include standard S-functions.

- 1 Open the test harness, `rtwdemo_PCGEvalHarnessSFun`.

Notice that the model uses the S-function.

- 2 Run the test harness.

Again, the results from running the generated code are the same as the simulation results.



Topics for Further Study

“Generating S-Function Wrappers”

Configuring the System for Testing via Test Vector Import/Export

This section extends the integration example in “Integrating the Generated Code into the External Environment” on page 2-61. In this case, `example_main.c` has simulated hardware I/O.

Open `matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/-stage_6_files/example_main.c`.

The augmented `example_main.c` file now has the following order of execution:

- 1 Initialize data (one time)


```
while < endTime
```
- 2 Read simulated hardware inputs
- 3 `PI_cnrl_1`
- 4 `PI_ctrl_2`
- 5 `Pos_Command_Arbitration`
- 6 Write simulated hardware outputs:


```
end while
```

Two functions, `plant` and `hardwareInputs`.

File Name	Function Signature	Comments
<code>Plant.c</code>	<code>void Plant(void)</code>	Code generated from the plant section of the test harness. Simulates the throttle body response to throttle commands.
<code>HardwareInputs</code>	<code>void hardwareInputs(void)</code>	Provides the <code>pos_req</code> signal and adds noise from the <code>Input_Signal_Scaling</code> subsystems into the plant feedback signal.

The handwritten function `WriteDataForEval.c` provides data logging. The function executes and writes test data to the file, `PCG_Eval_ExternSimData.m` once the test is complete. You can load the file into the MATLAB environment and compare it to the simulated data.

Testing with Test Vector Import/Export Using the Eclipse Environment

This tutorial uses the Eclipse Integrated Development Environment (IDE) debugger to build the embedded system.

- 1** Before building an executable in the Eclipse environment, regenerate the code without the S-function interface.
 - a** Open the `rtwdemo_PCG_Eval_P6` model.
 - b** In Model Explorer, open the **Configuration > Real-Time Workshop > SIL and PIL Verification** pane.
 - c** Make sure the **Create SIL block** check box is cleared.
 - d** Build the model.
- 2** Create a build directory (`Eclipse_Build_P6`).
- 3** Unzip the `rtwdemo_PCG_Eval_P6.zip` file into your build directory.
- 4** Delete these files, which `example_main.c` replaces:
 - `rtwdemo_PCG_Eval_P6.c`
 - `ert_main.c`
 - `rt_logging.c`
- 5** Follow the link for instructions on Appendix A, “Installing and Using an IDE for the Integration and Testing Tutorials (Optional)”.

Running the control code in Eclipse generates the `eclipseData.m` file. The `writeDataForEval.c` file generated this file.

- 6** Plot the Eclipse results.

Compare the data from the Eclipse run and the standard test harness.

- 7** Close the `rtwdemo_PCG_eval_P6` demo model.

Testing via Processor-in-the-Loop (PIL)

See “Verifying Compiled Object Code with Processor-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation for information, instructions, and demos.

Evaluating the Generated Code

In this section...

“Introduction” on page 2-78
“Evaluating Code” on page 2-78
“About the Compiler Used” on page 2-79
“Viewing the Code Metrics” on page 2-79
“About the Build Option Configurations” on page 2-79
“Configuration 1: Reusable Functions, Data Type Double” on page 2-80
“Configuration 2: Reusable Functions, Data Type Single” on page 2-81
“Configuration 3: Nonreusable Functions, Data Type Single” on page 2-82

Introduction

This tutorial reviews the build characteristics of the generated code. It also provides RAM/ROM data for several model configurations.

In this tutorial, you explore how different configurations affect the RAM/ROM metric.

Evaluating Code

Generated code efficiency is based on two primary metrics: *execution speed* and *memory usage*. Often, though not always, faster execution requires more memory. Memory usage in ROM (read-only memory) and RAM (random access memory) presents tradeoffs:

- Accessing data from RAM is faster than accessing ROM.
- Systems store executables and data using ROM because RAM does not maintain data between power cycles.

This section shows memory requirements divided into function and data components. Execution speed is not evaluated.

About the Compiler Used

The Freescale™ CodeWarrior® is used in this evaluation.

Compiler	Version	Target Processor
Freescale CodeWarrior	v5.5.1.1430	Power PC 565

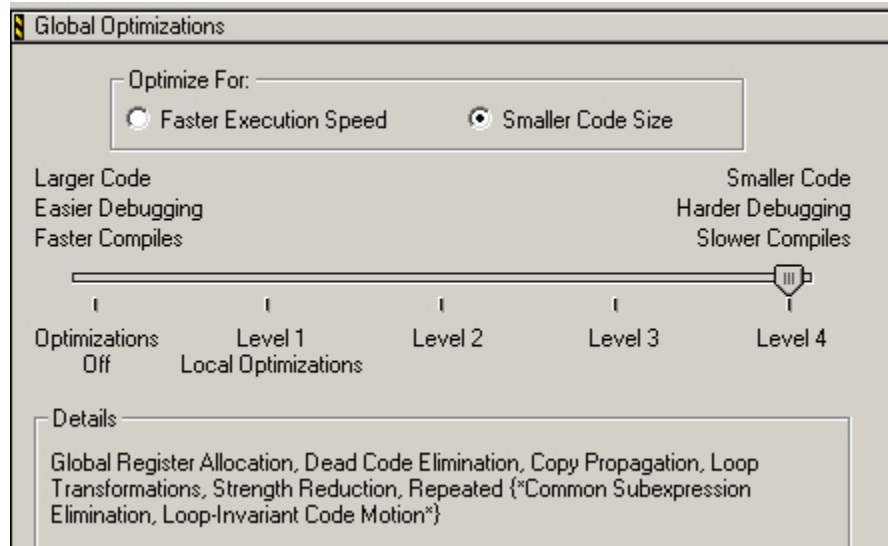
Viewing the Code Metrics

As described in “Integrating the Generated Code into the External Environment” on page 2-61 and “Testing the Generated Code” on page 2-69, the generated code might require the use of utility functions. The utility functions have a fixed overhead; their memory requirements is a one-time cost. Because of this, the data in this module shows memory usage for:

- Algorithms: The C code generated from the Simulink diagrams and the data definition functions
- Utilities: Functions that are part of the Real-Time Workshop library source
- Full: The sum of both the Algorithm and Utilities

About the Build Option Configurations

The same configuration options are used in all three evaluations. CodeWarrior is configured to minimize memory usage and apply all allowed optimizations.



Configuration 1: Reusable Functions, Data Type Double

- **Source files:**
`matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/CodeMetricFiles/PCG_Eval_CodeMetrics_1`
- **Data Type:** All doubles
- **Included Data:** All data required for the build is in the project (including data defined as extern: `pos_rqst`, `fbk_1`, and `fbk_2`)
- **Main Function:** A modified version of `example_main` from “Integrating the Generated Code into the External Environment” on page 2-61
- **Function-Call Method:** Reusable functions for the PI controllers

defineImportedData.c	12	28	•
defineImportedData.h	0	0	
eval_data.c	0	288	•
eval_data.h	0	0	
example_main.c	260	88	•
PCG_Eval_Fil_Define_Throt_Param.c	52	8	•
PCG_Eval_Fil_Define_Throt_Param.h	0	0	
PCG_Eval_File_1.c	464	92	•
PCG_Eval_File_1.h	0	0	
PCG_Eval_File_1_private.h	0	0	
PCG_Eval_File_1_types.h	0	0	
PI_Cntrl_Reusable.c	384	44	•
PI_Cntrl_Reusable.h	0	0	
rt_look1d.c	220	20	•
rt_look.c	372	20	•

Utilities

Func | Data

Memory Usage

Usage Type	Function (bytes)	Data (bytes)
Full	1764	589
Algorithms	1172	549
Utilities	592	40

Configuration 2: Reusable Functions, Data Type Single

In this configuration, the data types for the model were changed from the default of double to single.

- **Source files:**
`matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/CodeMetricFiles/PCG_Eval_CodeMetrics_2`
- **Data Type:** All singles

- **Included Data:** All data required for the build is in the project (including data defined as extern: pos_rqst, fbk_1, and fbk_2)
- **Main Function:** A modified version of example_main from “Integrating the Generated Code into the External Environment” on page 2-61
- **Function-Call Method:** Reusable functions for the PI controllers

Memory Usage

Usage Type	Function (bytes)	Data (bytes)
Full	1392	348
Algorithms	800	308
Utilities	592	40

Comparing the memory used by the algorithms in the first configuration to the current configuration, there is a large drop in the data memory, from 549 bytes to 308 bytes or 56 percent. The function size also decreased from 1172 to 800 bytes, or 68 percent. Running the simulation with data type set to single does not reduce the accuracy of the control algorithm. Therefore, this is an acceptable design decision.

Configuration 3: Nonreusable Functions, Data Type Single

- **Source files:**
`matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/CodeMetricFiles/PCG_Eval_CodeMetrics_3`
- **Data Type:** All singles
- **Included Data:** All data required for the build is in the project (including data defined as extern: pos_rqst, fbk_1, and fbk_2)
- **Main Function:** A modified version of example_main from “Integrating the Generated Code into the External Environment” on page 2-61
- **Function-Call Method:** The function interface is void void. Data is passed by global parameters.

The memory requirements for the third configuration are higher than the second configuration. Had the data type been doubled, the memory requirements would have been higher than the first configuration, as well.

Memory Usage

Usage Type	Function (bytes)	Data (bytes)
Full	1540	388
Algorithms	948	348
Utilities	592	40

Installing and Using an IDE for the Integration and Testing Tutorials (Optional)

- “Installing the Eclipse IDE and Cygwin Debugger” on page A-2
- “Integrating and Testing Code with the Eclipse IDE” on page A-4

Installing the Eclipse IDE and Cygwin Debugger


In this section...
“Installing the Eclipse IDE” on page A-2
“Installing the Cygwin Debugger” on page A-3

Installing the Eclipse IDE

Note This section explains how to install the Eclipse IDE for C/C++ Developers and the Cygwin debugger for use with the integration and testing tutorials. Alternatively, you can use another Integrated Development Environment (IDE) or use equivalent tools such as command-line compilers and makefiles.

- 1** Download the Eclipse IDE for C/C++ Developers from the Eclipse Downloads web page (<http://www.eclipse.org/downloads/>).
- 2** Download the Eclipse C/C++ Development Tools (CDT) that is compatible with the Eclipse IDE you downloaded in step 1 from the Eclipse CDT Downloads page (<http://www.eclipse.org/cdt/downloads.php>). For example, the Eclipse IDE 3.3 requires Eclipse CDT 4.0.
- 3** Unzip the downloaded Eclipse IDE zip file.
- 4** Create the directory `c:\eclipse`.
- 5** Copy the unzipped Eclipse IDE files to `c:\eclipse`.
- 6** Unzip the downloaded Eclipse CDT zip file.
- 7** Copy the contents of the directories `features` and `plugins` to the corresponding directories in `c:\eclipse`.
- 8** Create a link to the executable file `c:\eclipse\eclipse.exe` on your desktop.

Installing the Cygwin Debugger

- 1** Download the Cygwin `setup.exe` file from the Cygwin home page (<http://www.cygwin.com>).
- 2** Run the `setup.exe` file. A Cygwin Setup - Choose Installation Type dialog appears.
- 3** As you follow the installation procedure:
 - Select the option for installing over the Internet.
 - Accept the default root directory `c:\cygwin`.
 - Specify a local package directory. For example, specify `c:\cygwin\packages`.
 - Specify how you want to connect to the Internet.
 - Choose a download site.
- 4** On the dialog for selecting packages, set the **Devel** category to **Install** by clicking the selector icon .
- 5** Add the directory `c:\cygwin\bin` to your system Path variable. For example, on a Windows XP system:
 - a** Click **Start > Settings > Control Panel > System > Advanced > Environment Variables**.
 - b** Under **System variables**, select the Path variable and click **Edit**.
 - c** Add `c:\cygwin\bin` to the variable value and click **OK**.

Note To use Cygwin, your build directory must be on your C drive and the directory path cannot include any spaces.

Integrating and Testing Code with the Eclipse IDE

In this section...
“Introducing Eclipse” on page A-4
“Defining a New C Project” on page A-5
“Configuring the Debugger” on page A-6
“Starting the Debugger” on page A-7
“Setting the Cygwin Path” on page A-7
“What the Eclipse Debugger Can Do” on page A-8

Introducing Eclipse

Eclipse (www.eclipse.org) is an integrated development environment for developing and debugging embedded software. Cygwin (www.cygwin.com) is an environment that is similar to the Linux environment, but runs on Windows and includes the GCC compiler and debugger.

This section contains instructions for using the Eclipse IDE with Cygwin tools to build, run, test, and debug projects that include code generated by Real-Time Workshop Embedded Coder software, as described in “Integrating the Generated Code into the External Environment” on page 2-61 and “Testing the Generated Code” on page 2-69. Many other software packages and tools also exist that can work with Real-Time Workshop Embedded Coder software to perform similar tasks.

“Installing the Eclipse IDE and Cygwin Debugger” on page A-2 contains instructions for installing Eclipse and Cygwin. Be sure you have installed Eclipse and Cygwin, as explained in that section, before you proceed.

Note To use Cygwin, your build directory must be on your C drive and the directory path cannot include any spaces.

About Project Names and File Names Used in This Section

“Integrating the Generated Code into the External Environment” on page 2-61 and “Testing the Generated Code” on page 2-69 both use the instructions in this section, but the project names and file names differ. Where you see ## in a project name or file name, substitute:

- P5 if you are working in “Integrating the Generated Code into the External Environment” on page 2-61
- P6 if you are working in “Testing the Generated Code” on page 2-69

Defining a New C Project

- 1** In Eclipse, choose **File > New > Project**. A New Project dialog box appears.
- 2** In the New Project dialog box,
 - a** Expand **C**.
 - b** Click **C Project**.
 - c** Click **Next**.A C Project dialog box appears.
- 3** In the C Project dialog box,
 - a** Type the project name `rtwdemo_PCG_Eval_##` (where ## is P5 or P6) in the **Project name** field.
 - b** Specify the location of your build directory in the **Location** field, for example `C:\work\Eclipse_Projects\Eclipse_Build_P5`.
 - c** In the **Project types** selection box, select **Makefile project**. A list of corresponding toolchains appears.
 - d** Select toolchain `Cygwin GCC`.
 - e** Click **Next**. A Select Configurations dialog box appears.
- 4** In the Select Configurations dialog box, click the **Advanced settings** button. The Properties for *configuration* dialog box appears.
- 5** In the Properties for *configuration* dialog box,

- a** Select **C/C++ Build**.
- b** Select **Generate Makefiles automatically**.
- c** Select the **Behavior** tab.
- d** Select **Build on resource save (Auto build)**.
- e** Click **Apply** and **OK**.

The Properties for *configuration* dialog box closes.

- 6** In the Select Configurations dialog box, click **Finish**.

Configuring the Debugger

- 1** In Eclipse, choose **Run > Open Debug Dialog**. The Debug dialog box appears.
- 2** Double-click **C/C++ Local Application**. A **New_configuration** entry appears under **C/C++ Local Application**.
- 3** Type the name of your configuration (for example, `rtwdemos_PCG_Eval_P5_CygwinGCC`) in the **Name** field.
- 4** Enter the name of your project (for example, `rtwdemo_PCG_Eval_P5`) in the **Project** field. If you click **Browse**, a Project Section dialog box appears. Select a project and click **OK**.
- 5** Enter the path for the location of your executable file (for example, `C:\Work\Eclipse_Projects\Eclipse_Build_P5\CygwinGCC\rtwdemo_PCG_Eval_P5.exe`) in the **C/C++ Application** field. If you click **Browse**, an Open dialog box appears. Navigate to and select your executable file and click **Open**.
- 6** Click **Apply**. The configuration name you specified replaces **New_configuration** under **C/C++ Local Application**.

Note Do *not* click **Run**.

- 7** Click **Close**.

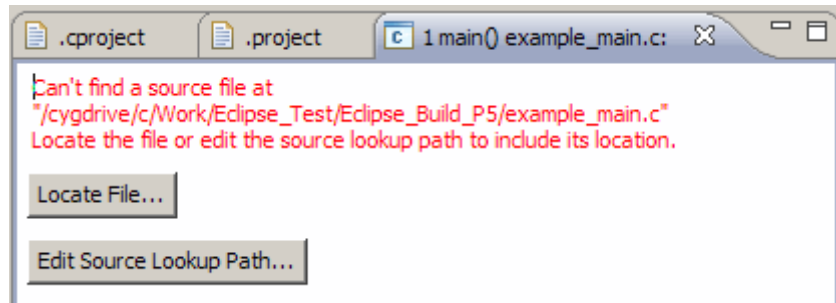
Starting the Debugger

To start the debugger,

- 1 In the main Eclipse window, select **Run > Debug**.

Tabbed debugger panes that display debugging information and controls appear in the main Eclipse window. In addition, a console window pointing to your executable file (for example, `C:\Work\Eclipse_Projects\Eclipse_Build_P5\Cygwin GCC\rtwdemo_PCG_Eval_P5.exe`) opens.

- 2 Specify the location of the project files. The Cygwin debugger creates a virtual drive, `/cygdrive/c/` during the build process. To run the debugger, Eclipse needs to remap the drive or locate your project files. Once Eclipse locates the first file, it automatically finds the remaining files. In the Eclipse window, click **Locate File**.



For information on using the **Edit Source Lookup Path** button, see “Setting the Cygwin Path” on page A-7

- 3 An Open dialog box appears. Navigate to the `example_main.c` file and click **Open**. Your program opens in the context of the debugger.

Setting the Cygwin Path

The first time you run Eclipse, you will get an error related to the Cygwin path.

To provide the necessary path information,

- 1** Click **Edit Source Lookup Path** in the error message dialog box.
- 2** Click **Add** in the Edit Source Lookup Path dialog box.
The Add Source dialog box appears.
- 3** Select **Path Mapping** in the Add Source dialog box.
- 4** Click **OK**. The Edit Source Lookup Path dialog box appears.
- 5** In the Edit Source Lookup Path dialog box, select **Path Mapping**.
- 6** Click **Edit**. The Path Mappings dialog box appears.
- 7** In the Path Mappings dialog box, click **Add**.
- 8** In the **Compilation path** field, type `\cygdrive\c\`.
- 9** In the **Local file system path** field, type `c:\`.
- 10** Click **OK**.

What the Eclipse Debugger Can Do

Actions and commands available in the debugger include:

Action	Command
Step into	F5
Step over	F6
Step out	F7
Resume	F8
Toggle break point	Ctrl + Shift + B