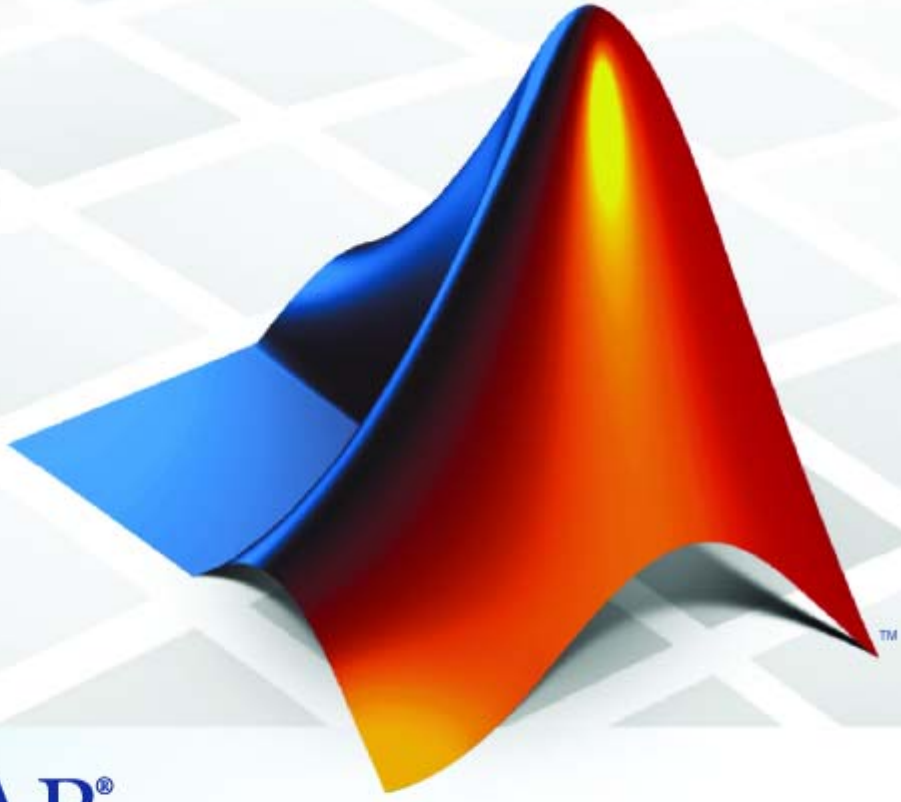
Real-Time Workshop® Embedded Coder™ 5

Getting Started Guide



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

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 2007–2008 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	New for Version 5.0 (Release 2007b)
March 2008	Online only	Revised for Version 5.1 (Release 2008a)

Contents

1 Introducing Real-Time Workshop® Embedded Coder™ Software

Product Overview 1-2

What Can You Do with Real-Time Workshop® Embedded Coder™ Software? 1-3

What You Need to Know to Use This Product 1-5
Prerequisites 1-5
Real-Time Workshop® Embedded Coder™ Documentation Collection 1-5
Related Documentation 1-7

Installing Real-Time Workshop® Embedded Coder™ Software 1-8

Accessing Real-Time Workshop® Embedded Coder™ Demos 1-9

2 Learning and Using Real-Time Workshop® Embedded Coder™ Software

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 Model's Functional Design 2-5
Viewing the Top-Level Model 2-6
Viewing Subsystems 2-7
Understanding the Simulation Testing Environment 2-8
Running the Simulation Tests 2-12
Viewing the Configuration Options for Code Generation .. 2-13
Generating Code for the Model 2-19
Examining the Generated Code 2-20
See Also 2-22

Configuring the Data Interface 2-23
Introduction 2-23
Declaring Data and Functions 2-23
Controlling Data in Simulink® Software and Stateflow® Software 2-26
Adding New Data Objects 2-29
Configuring Data Objects 2-30
Controlling File Placement of Parameter Data 2-30
Enabling Data Objects in Generated Code 2-31
Effects of Simulation on Data Typing 2-32
Viewing Data Objects in Generated Code 2-33
Managing Data 2-36
See Also 2-37

Partitioning Functions in the Code 2-38
Introduction 2-38
About Atomic and Virtual Subsystems 2-38
Viewing Changes in the Model Architecture 2-39
Controlling Function Location and File Placement in Generated Code 2-40
Understanding Reentrant Code 2-43
Using a Mask to Pass Parameters into a Library Subsystem 2-44
Generating Code from an Atomic Subsystem 2-45
Generating Code: Full Model versus Exported Functions 2-47
Effect of Execution Order on Simulation Results 2-48
See Also 2-50

Calling External C Functions from the Model and Generated Code 2-51
Introduction 2-51
Including Preexisting C Functions in a Simulink® Model .. 2-51

Creating a Block That Calls a C Function 2-52
Validating the External Code in the Simulink® Environment 2-53
Validating the C Code as Part of the Simulink® Model ... 2-55
Calling the C Function from the Generated Code 2-56
See Also 2-57

Integrating the Generated Code into the External Environment 2-58
Introduction 2-58
Building and Collecting the Required Data and Files 2-58
Integrating the Generated Code into an Existing System 2-59
About the Integration Environment 2-59
Matching the System Interfaces 2-61
Matching Function Call Interfaces 2-63
Building a Project in the Eclipse Environment 2-64
See Also 2-65

Testing the Generated Code 2-66
Introduction 2-66
Methods for Validating Generated Code 2-66
Reusing Test Data: Test Vector Import/Export 2-68
Testing via Software in the Loop (S-Functions) 2-69
Configuring the System for Testing via Test Vector Import/Export 2-72
Testing with Test Vector Import/Export Using the Eclipse Environment 2-73
See Also 2-74

Evaluating the Generated Code 2-75
Introduction 2-75
Evaluating Code 2-75
About the Compiler Used 2-76
Viewing the Code Metrics 2-76
About the Build Option Configurations 2-76
Configuration 1: Reusable Functions Data Type Double .. 2-77
Configuration 2: Reusable Functions Data Type Single ... 2-78
Configuration 3: Nonreusable Functions Data Type Single 2-79

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**
Creating a New CDT-Managed Make C Project **A-5**
Configuring the Debugger **A-6**
Starting the Debugger **A-8**
Setting the Cygwin Path **A-9**
What the Eclipse Debugger Can Do **A-10**

Introducing Real-Time Workshop® Embedded Coder™ Software

Product Overview (p. 1-2)	Provides a high-level overview of the Real-Time Workshop® Embedded Coder™ product, describing its purpose and major features.
What Can You Do with Real-Time Workshop® Embedded Coder™ Software? (p. 1-3)	Lists features of the Real-Time Workshop Embedded Coder product.
What You Need to Know to Use This Product (p. 1-5)	Describes the prerequisite knowledge and experience assumed for using the Real-Time Workshop Embedded Coder software and lists related documentation.
Installing Real-Time Workshop® Embedded Coder™ Software (p. 1-8)	Provides installation notes.
Accessing Real-Time Workshop® Embedded Coder™ Demos (p. 1-9)	Provides information about interactive demos and example code provided to help you learn about the Real-Time Workshop Embedded Coder product.

Product Overview

Real-Time Workshop® Embedded Coder™ software is a separate, add-on product for use with Real-Time Workshop® software. It is intended for use in embedded systems development. The Real-Time Workshop Embedded Coder software generates code that is easy to read, trace, and customize for your production environment.

Real-Time Workshop Embedded Coder software provides a framework for the development of production code that is optimized for speed, memory usage, and simplicity. The software generates optimized ANSI-C or ISO-C code for fixed-point and floating-point microprocessors. It extends the capabilities provided by Real-Time Workshop software to support specification, integration, deployment, and testing of production applications on embedded targets. It also addresses targeting considerations such as RAM, ROM, and CPU constraints, code configuration, and code verification.

The embedded real-time (ERT) target provided by Real-Time Workshop Embedded Coder software is designed for customization. Most users want to generate code for a particular microprocessor or development board, and to deploy the code on target hardware with a cross-development system. To do this, some modifications to the ERT target files are required. This document and its companion, the Developing Embedded Targets document, describe how to customize the ERT target for your production requirements.

For large-scale, multimodel projects involving teams of engineers, the Real-Time Workshop Embedded Coder software offers Module Packaging Features (MPF) you can use to control the number and organization of files generated, the location of global identifiers, registration of user-defined data types, customized comments, and the location of target variables.

For a comparison of Real-Time Workshop and Real-Time Workshop Embedded Coder, see “What You Can Accomplish Using Real-Time Workshop Technology” in the Real-Time Workshop documentation.

What Can You Do with Real-Time Workshop® Embedded Coder™ Software?

In addition to supporting the features of the Real-Time Workshop® software, Real-Time Workshop® Embedded Coder™ software:

- Generates ANSI/ISO C or C++ code and executables from Simulink® models and Stateflow® charts with memory usage, execution speed, and readability comparable to handwritten code.
- Extends Real-Time Workshop software and Stateflow® Coder™ software with the optimizations and code configuration features essential for production deployment.
- Supports all Simulink data objects and data dictionary capabilities, including user-defined storage classes, types, and aliases.
- Provides an intuitive graphical user interface for creating custom data.
- Concisely partitions multirate code for efficient scheduling with or without a real-time operating system (RTOS).
- Provides a rich set of commenting capabilities to trace code to models and requirements.
- Verifies code by automatically importing it into Simulink models for software-in-the-loop testing.
- Generates code documentation that is integrated with the Simulink Model Explorer and hyperlinked to the model.
- Provides a Model Advisor that checks your model configuration and offers advice on how to optimize or tune a configuration set based on your stated goals or style.
- Generates an extensible main program based on information you provide for deploying the code in your real-time environment.
- Generates single-rate or multirate code using periodic sample times specified in a model.
- Applies a strategy called *rate grouping* for multirate, multitasking models, which generates separate functions for the base rate task and for each subrate task in the model.

- Provides an option to easily transition between the Real-Time Workshop generic real-time (GRT) target and the Real-Time Workshop Embedded Coder embedded real-time (ERT) target.
- Provides extensible module packaging features that let you package generated code to comply with specific software styles and standards.
- Provides capabilities for verifying generated code, including the ability to import generated code back into Simulink models as an S-function for software-in-the-loop testing with a plant model, generation of user-controlled comments and descriptions to improve code readability and traceability, inclusion of requirements in generated code, and persistent identifier names for minimizing code differences between model revisions.
- Documents generated code in an HTML report that comprehensively describes code modules and model configuration settings applied during code generation.
- Supports international (non-US-ASCII) characters encountered during code generation when found in Simulink block names and block descriptions, user comments on Stateflow diagrams, Stateflow object descriptions, custom TLC files, and code generation template files. For details about international character support, see “Support for International (Non-US-ASCII) Characters” in the Real-Time Workshop documentation.

For information on applications of the Real-Time Workshop Embedded Coder product and how and when you might use it during system development, see “Introduction to Real-Time Workshop Technology” in the Real-Time Workshop documentation.

What You Need to Know to Use This Product

In this section...
"Prerequisites" on page 1-5 "Real-Time Workshop® Embedded Coder™ Documentation Collection" on page 1-5 "Related Documentation" on page 1-7

Prerequisites

To use the Real-Time Workshop® Embedded Coder™ software, you should have basic familiarity with MATLAB®, Simulink®, and Real-Time Workshop® software. 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.
- "Program Architecture" and "Models with Multiple Sample Rates" 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.

Real-Time Workshop® Embedded Coder™ Documentation Collection

The Real-Time Workshop Embedded Coder documentation collection consists of the following:

Document	Description
(this document)	Provides an overview of the Real-Time Workshop Embedded Coder product and a collection of tutorials to help you get started with using the product.

Document	Description
User's Guide	Describes ERT model execution, timing, and task management; explains how to interface to and call model code; describes default ERT code generation options; and discusses advanced configuration options.
Module Packaging Features	Explains how to use the Module Packaging Features.
Reference	Provides reference descriptions of Real-Time Workshop Embedded Coder functions and blocks.
Developing Embedded Targets	Describes requirements and implementation details for creating custom embedded targets based on the supplied ERT.

Related Documentation

If you are planning to implement custom embedded targets, the following documentation might interest you:

Document	Description
Real-Time Workshop User's Guide: Writing S-Functions for Real-Time Workshop	Discusses inlining and code generation issues relevant to device drivers and other S-functions
Real-Time Workshop User's Guide: Data Exchange APIs	Explains how to interface signals and parameters within generated code to your own code; combine code generated from multiple models into a single system; and implement external mode communication with your own low-level protocol layer.
Target Language Compiler	Provides details about the Target Language Compiler (TLC) needed to make nontrivial modifications to the ERT system target file, use TLC hooks into the build process, use information from the *model*`.rtw` file, implement inlined device drivers, or pass information into or out of the TLC phase of the build process.
Writing S-Functions	Explains how to write fully inlined S-functions. This information is necessary for developing device driver blocks for a target.

Installing Real-Time Workshop® Embedded Coder™ Software

Your platform-specific MATLAB® installation documentation provides all of the information you need to install Real-Time Workshop® Embedded Coder™ software.

Prior to installing the software, you must obtain a License File or Personal License Password (PLP) from The MathWorks. The License File or PLP identifies the products you are permitted to install and use.

If you customize your installation, the installer displays a dialog box that lets you select the MATLAB products to install. You can select and install only products for which you are licensed.

Real-Time Workshop Embedded Coder software requires certain other products, described in the following table, that must be installed for proper installation and execution.

Licensed Product	Required Products	Additional Information
Simulink®	MATLAB	—
Real-Time Workshop®	Simulink	Requires host platform C compiler (Microsoft Visual C/C++ or Watcom C/C++). Stateflow® Coder™ software is also required when generating code for Simulink models containing Stateflow® charts.
Real-Time Workshop Embedded Coder	Real-Time Workshop	Requires a cross-compiler for the target processor.

If you experience installation difficulties and have Web access, use the resources available on the MathWorks Web site Installation and Licensing page at `http://www.mathworks.com/support/install.html`.

Accessing Real-Time Workshop® Embedded Coder™ Demos

The Real-Time Workshop® demo suite contains many demos to familiarize you with features of Real-Time Workshop® Embedded Coder™ software and to inspect generated code. These demos illustrate features specific to Real-Time Workshop Embedded Coder software and also general Real-Time Workshop features as used with Real-Time Workshop Embedded Coder software.

To open the demo suite by entering the following demo library name at the MATLAB® command prompt:

```
rtwdemos
```

Most of the demos provide a button titled **Generate Code Using Real-Time Workshop Embedded Coder**. When you click this button, the demo auto-configures itself for code generation using the ERT target, and then initiates the code generation process. If your installation is licensed for Real-Time Workshop Embedded Coder software, use this button.

If your installation is not licensed for Real-Time Workshop Embedded Coder software, you can run most of the demos by clicking on the **Generate Code Using Real-Time Workshop** button. When you click this button, the demo auto-configures itself for code generation using the GRT target, and then initiates the code generation process. Note that the GRT target provides a subset of the capabilities of the ERT target.

2

Learning and Using Real-Time Workshop® Embedded Coder™ Software

Using the Tutorials (p. 2-2)	Describes the tutorials.
Understanding the Demo Model (p. 2-5)	Acquaints you with the model used in the tutorials.
Configuring the Data Interface (p. 2-23)	Guides you through the steps for configuring a data interface.
Partitioning Functions in the Code (p. 2-38)	Guides you through the steps for partitioning functions in generated code.
Calling External C Functions from the Model and Generated Code (p. 2-51)	Guides you through the steps for calling external C functions.
Integrating the Generated Code into the External Environment (p. 2-58)	Guides you through steps for integrating generated code.
Testing the Generated Code (p. 2-66)	Guides you through steps for testing generated code.
Evaluating the Generated Code (p. 2-75)	Guides you through the evaluation of generated code.

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 contains seven tutorials that apply MathWorks products to those common steps. 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 2-23
- "Partitioning Functions in the Code" on page 2-38
- "Calling External C Functions from the Model and Generated Code" on page 2-51
- "Integrating the Generated Code into the External Environment" on page 2-58
- "Testing the Generated Code" on page 2-66
- "Evaluating the Generated Code" on page 2-75

In these tutorials, you work with a Simulink® model, read data into it, and use Real-Time Workshop® Embedded Coder™ software to generate code for the model, integrate the generated code with an existing system, and validate simulation and executable results. Each tutorial uses a unique version of the model and its associated data, which you load as instructed.

As you proceed through the tutorials, you save each model after you have worked on it, capturing your modifications to the model and model data. Saving the model and data preserves them for future examination. To prevent any errors from carrying over, you begin the next tutorial by opening a preexisting model and loading preexisting data.

These tutorials are intended for online use. Each contains cross-references to related documentation and provides instructions for performing specific tasks. If a task fails for any reason, error messages appear in the MATLAB® Command Window.

Prerequisites

The tutorials in this book assume the following prerequisite knowledge:

MathWorks products

- How to read, write, and apply M-file 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 methods of calling 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

Installing and Using an IDE for Integration and Testing explains how to install third-party software, the Eclipse IDE and the Cygwin/gcc compiler, which you can use to integrate code that you generate into an external

development environment for testing. That section also explains how to integrate the generated code into the external environment.

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. This tutorial refers to the created working directory as `MyPlayArea`.

2. Copy the following files from *`matlabroot`*`/toolbox/rtw/rtwdemos` to the directory you just created:

   - `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 Model's Functional Design" on page 2-5
"Viewing the Top-Level 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
"Viewing the Configuration Options for Code Generation" on page 2-13
"Generating Code for the Model" on page 2-19
"Examining the Generated Code" on page 2-20
"See Also" on page 2-22

Introduction

This tutorial introduces the model from a behavioral and structural perspective, and explains how code is generated and shows the basics of configuring the model.

In this tutorial you:

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

Understanding the Model's Functional Design

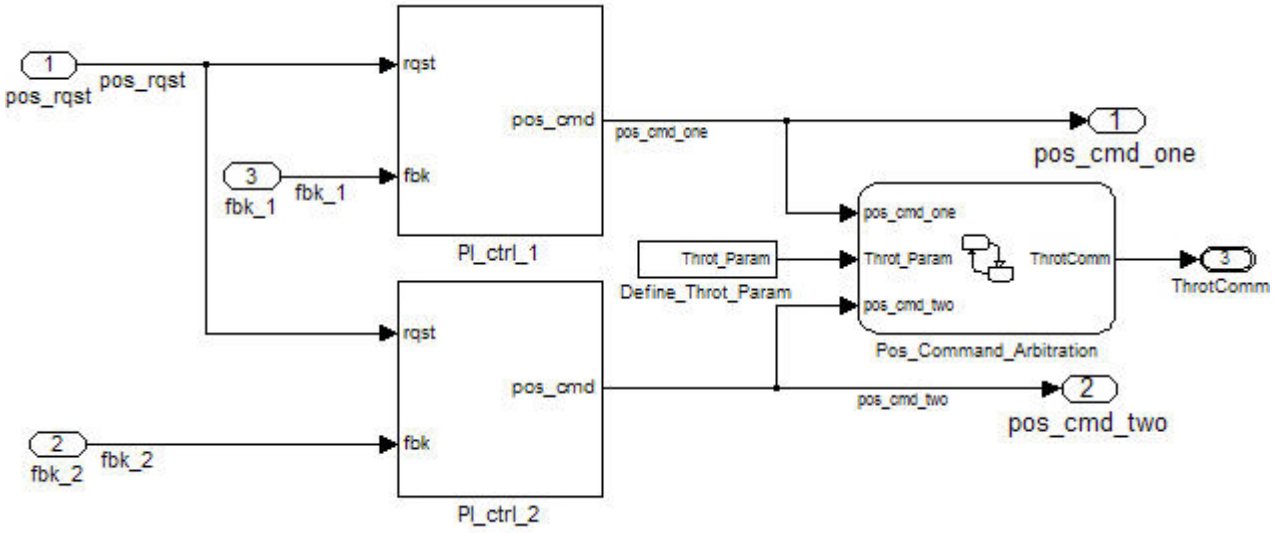
This demo uses a simple but functionally complete model of a throttle controller. The model features redundancy, which is common for safety-critical

drive-by-wire applications. The model highlights a standard model structure and a set of basic blocks used in algorithm design.

In the current configuration, the model generates code. However, the code is not configured for a production target system. This demo 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-Level Model

1 Open the top-level model by entering `rtwdemo_PCG_Eval_P1` at the MATLAB® command line.



The top-level 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 `ThrotComm`
- Signal routing

- *No* transformative blocks (blocks that change the value of a signal, such as Sum and Integrator blocks)

The layout shows a basic model architectural style.

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

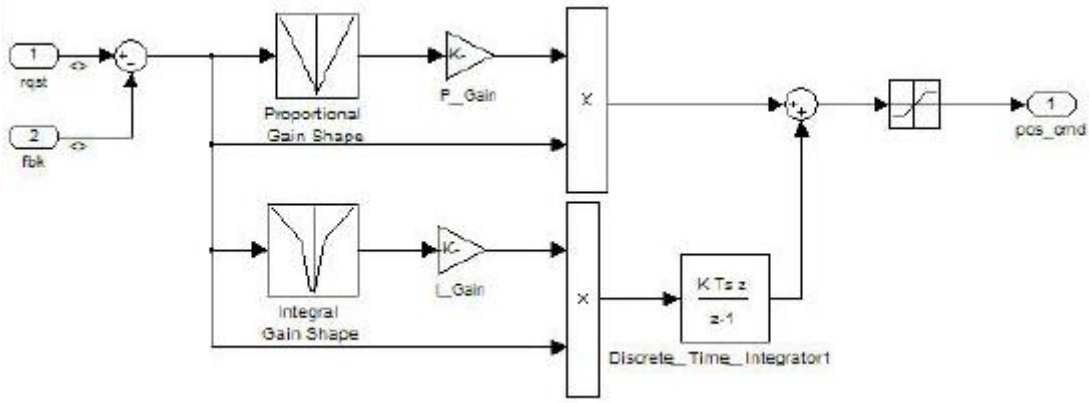
You can apply this style to all types of models.

Viewing Subsystems

Two subsystems represent PI controllers, `PI_cntrl_1` and `PI_cntrl_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.

PI Controller Subsystems

Open the `PI_cntrl_1` subsystem by double clicking the subsystem block.



The PI controllers are included in the model from a *library*, a group of related blocks or models intended for reuse. Libraries provide one of two methods for including and reusing models. The second method, model referencing, is covered later. 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 ensures that instances of the block in different models remain consistent.

Command Signal Error Checking Subsystem

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

Understanding the Simulation Testing Environment

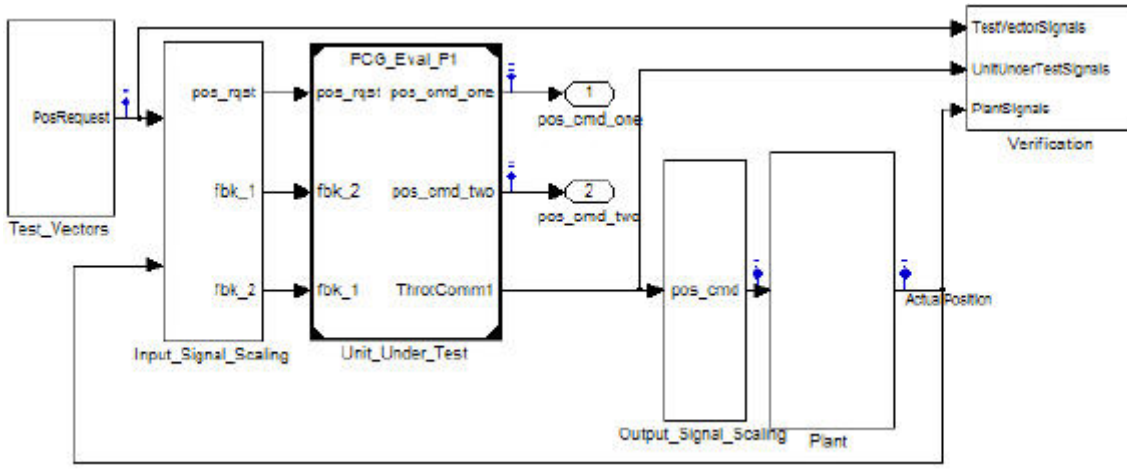
You test the throttle controller model in a separate model called a *test harness*. A test harness is a model that evaluates the control algorithm and offers the following advantages:

- 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

A common simulation testing environment consists of the following parts:

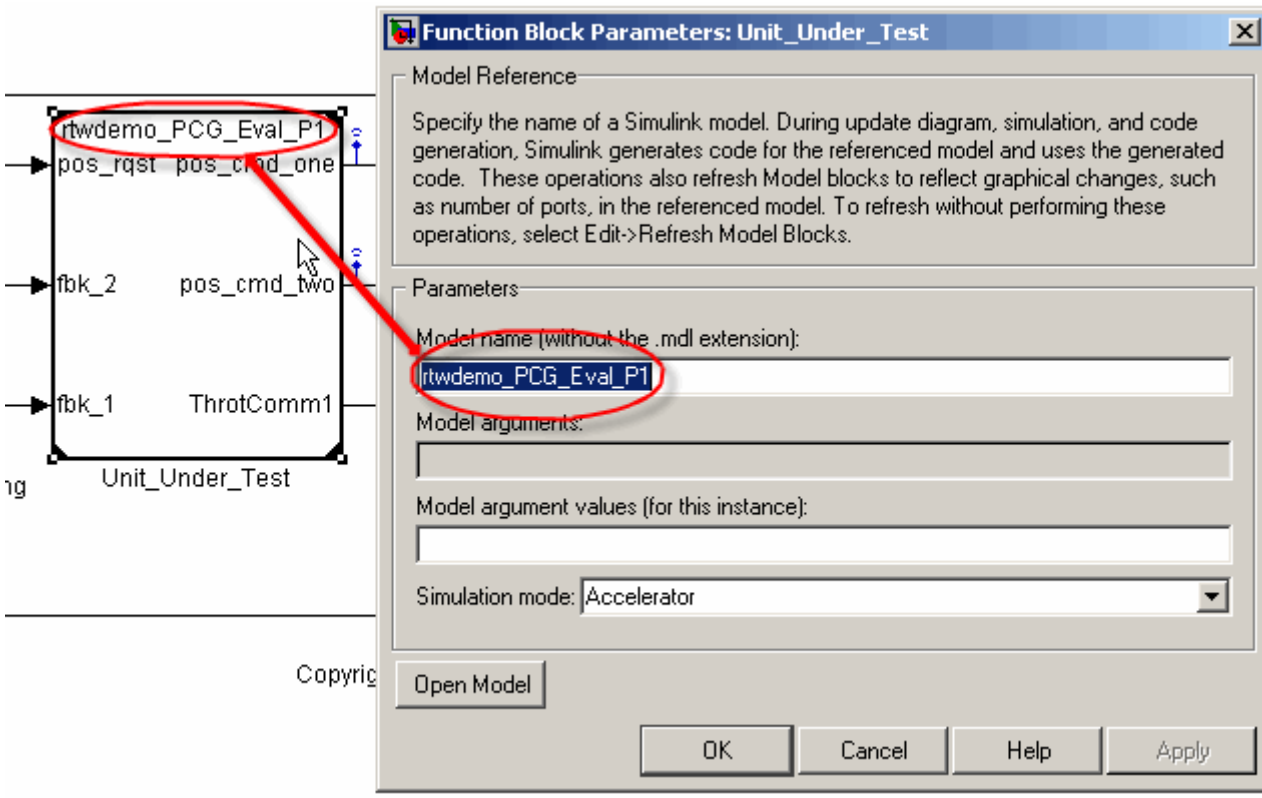
- Unit under test
- Test vector source
- Evaluation and logging
- Plant or feedback system
- Input and output scaling

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



2 View the unit under test model reference parameters by right-clicking the `Unit_Under_Test` subsystem and selecting **Model Reference Parameters**.

The control algorithm is the *unit under test*. The control algorithm is referenced in the test harness using a Model Reference block. The Model Reference block provides a second method for reusing components. The referenced model is selected in the Model Reference dialog.



The Model Reference block allows other models to be *referenced* (directly or indirectly) from the top model as *compiled functions*. By default, Simulink® software compiles the model when the referenced model is changed. 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. One copy of the compiled model is in memory, even when the model is referenced multiple times.

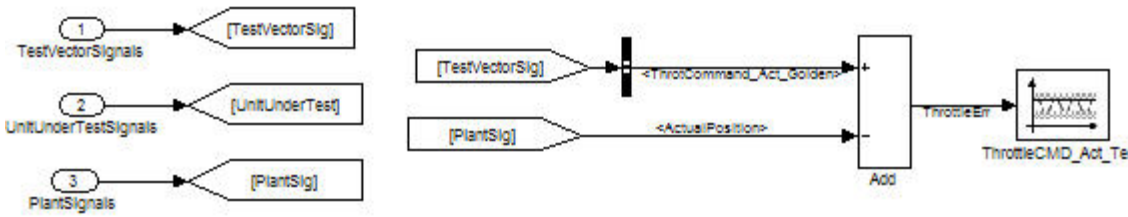
3 Open the test vector source subsystem double clicking the subsystem labeled `Test_Vectors`.

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

4 Open the `Verification` subsystem.

The test harness compares the simulation results against *golden data*: a set of test results that have been certified by an expert to exhibit the desired behavior for the model. In this model, the V&V Assertion block compares the plant's simulated throttle value position against the golden value provided by 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 M-file 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` subsystem.

The throttle dynamics are modeled with a transfer function that is broken down into its canonical form. You can create plant models to any level of fidelity. It is not uncommon to have different plant models used at different stages of testing.

6 Open the `Input_Signal_Scaling` and `Output_Signal_Scaling` subsystems.

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

- Select signals to route to the unit under test and plant.
- Rescale signals between engineering units and units required by the unit under test.
- Handle rate transitions between the plant and the unit under test.

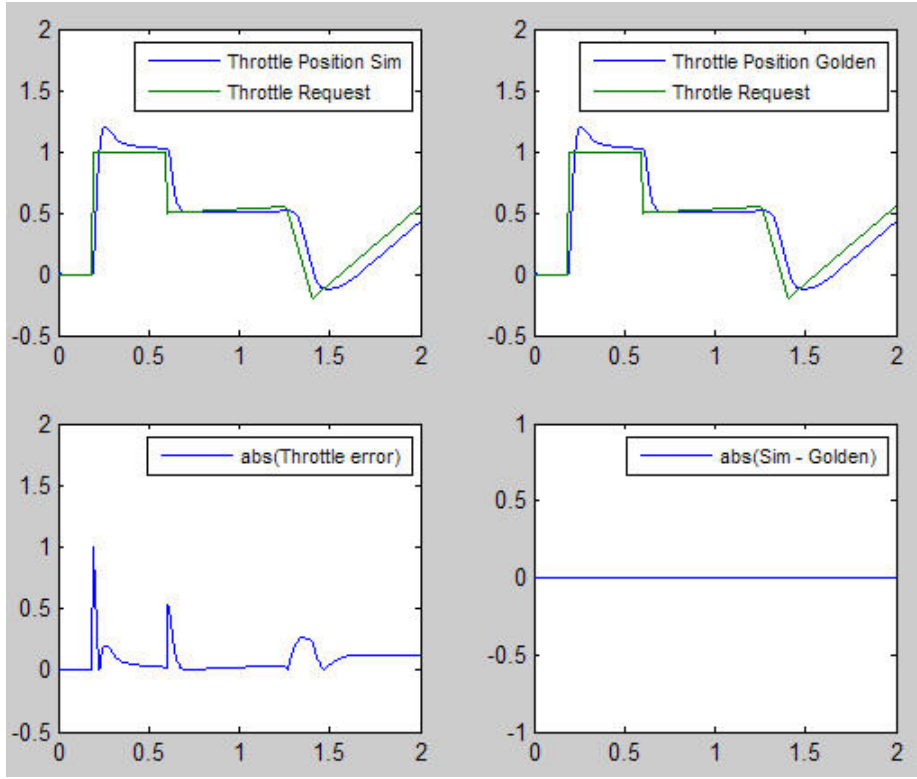
Running the Simulation Tests

1. Set up your C compiler by entering `mex -setup` at the MATLAB command line and specifying a valid, installed compiler.
2. Set your working directory to a directory to which you have write access.
3. In the toolbar of the `rtwdemo_PCGEvalHarness` model, click the **Start** button to run the test harness model simulation.

   The first time the test harness runs, the referenced model is compiled. 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.

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



Viewing the Configuration Options for Code Generation

The first step to 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. One way of setting the parameters is to use the **Configuration Parameters** dialog, directly accessible through the model's **Simulation** menu. Equivalently, open the Model Explorer, accessible through the model's **View** menu.

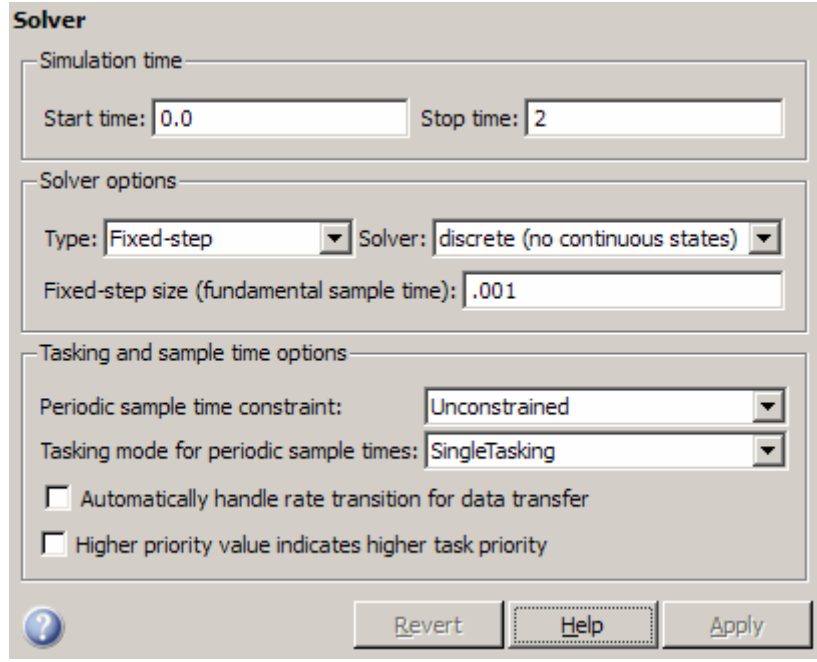
This tutorial focuses on four areas of model configuration:

- Solver options

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

1 Open the Model Explorer and click **Configuration Preferences** in the **Model Hierarchy** pane.

2 Open the **Solver** pane.

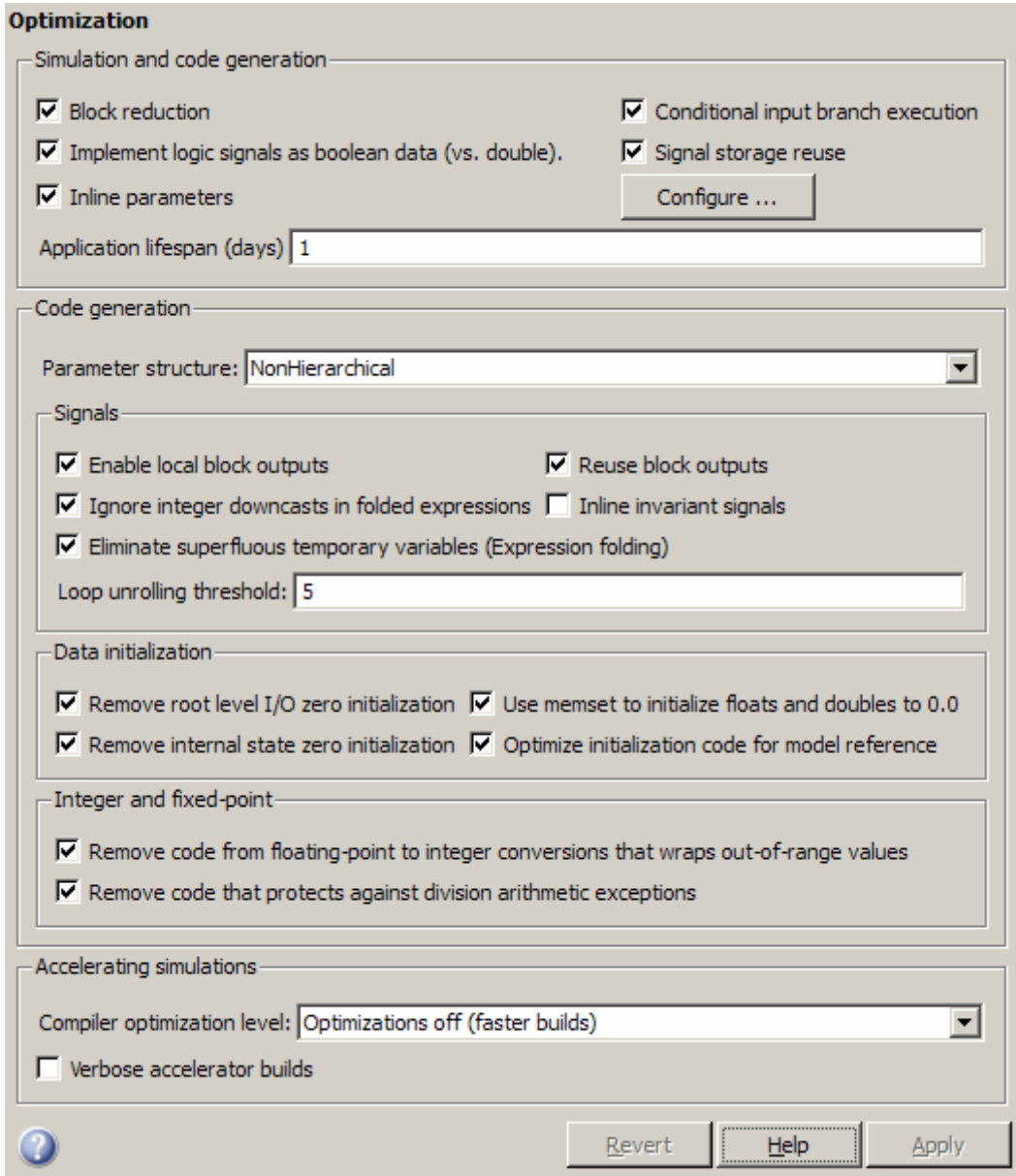


For Real-Time Workshop software to generate code for a model, you must configure the model to use a fixed-step solver. The start and stop time do not affect generated code.

Option	Required Setting	Effect on Generated Code
Start time and **Stop time**	Any	No effect

Option	Required Setting	Effect on Generated Code
Type	`Fixed-step`	Code is not generated unless fixed step
Solver	Any	Controls selected integration algorithms
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

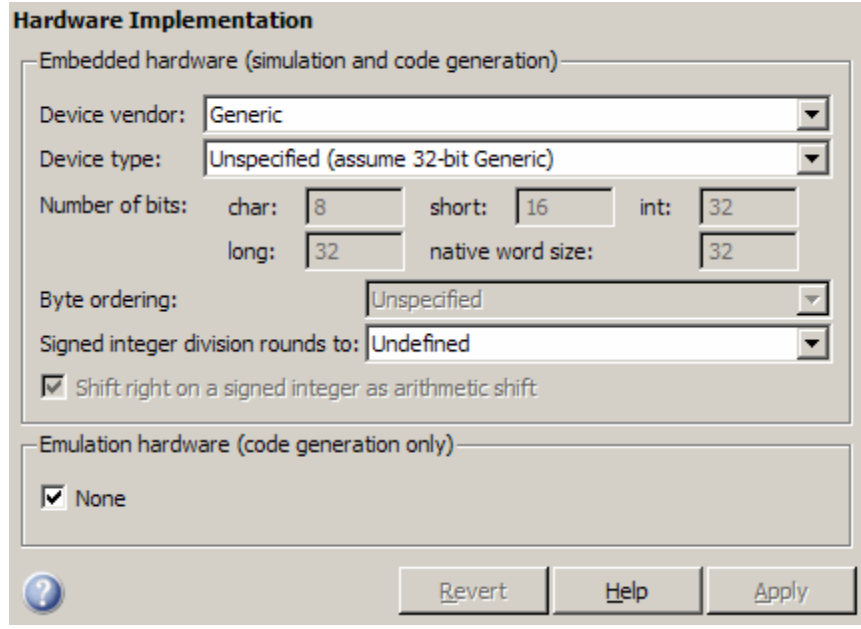
3 Open the **Optimization** pane.



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

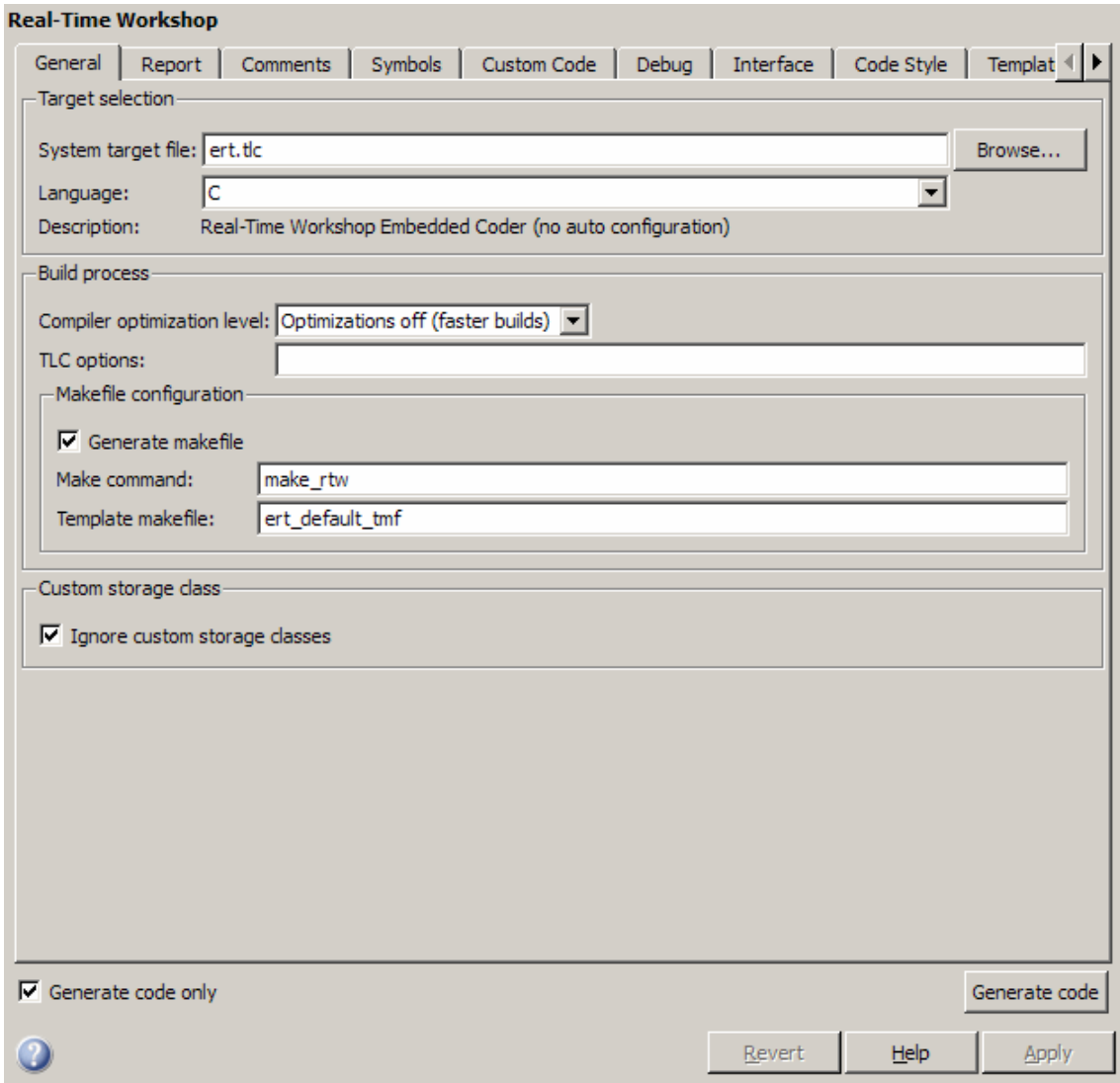
Subpane	Effect
Simulation and code generation	Removes unused branches from the code and controls creation of temporary variables
Signals	Reduces the number of temporary variables created by collapsing multiple computations into a single assignment and by reusing temporary variables
Data initialization	Controls which signals have explicit initialization code
Integer and fixed-point	Enables and disables use of overflow and division-by-zero protection code

4 Open the **Hardware Implementation** pane.



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

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



The **Real-Time Workshop** pane is where you specify the system target file (STF). This demo uses the Real-Time Workshop® Embedded Coder™ STF (`ert.tlc`). You can extend this 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
- Selected make file
- 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

Generating Code for the Model

Use one of the following methods to generate code for the `rtwdemo_PCG_eval_P1` model:

- With the model in focus, enter **Ctrl+B** from your keyboard.
- Choose **Configuration Parameters** > **Real-Time Workshop** > **Generate code**.
- Choose **Tools** > **Real-Time Workshop** > **Build Model**.

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

Examining the Generated Code

The code generation process results in multiple files that you can view from the Model Explorer. In addition to the standard C and H files, a set of HTML files is generated. The HTML files provided active links between the code and the model.

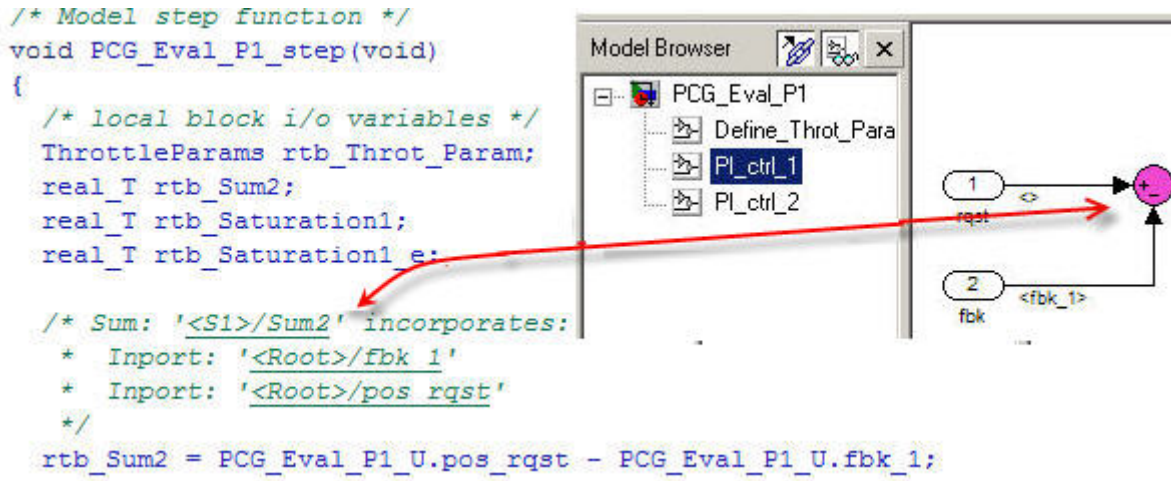
Note You must generate code before you can view the files.

---

1. In the Model Explorer Model Hierarchy pane, select **Code for rtwdemo_PCG_Eval_P1**.
2. In the Contents pane, select **HTML Report**.
3. In the Real-Time Workshop HTML report, click `rtwdemo_PCG_Eval_P1.c`.

   In the generated code, note that:

   - All of the controller code is contained in one function called `rtwdemo_PCG_Eval_P1_step` and found in the file `rtwdemo_PCG_Eval_P1.c`.
   - The operations of multiple blocks are collapsed into one equation.
   - Variables are initialized in the function `rtwdemo_PCG_Eval_P1_initialize`.
   - All data is defined using Real-Time Workshop data structures (for example, `rtwdemo_PCG_Eval_P1_U.pos_rqst`).



You can view any of the files by clicking on the links in the table below, or by exploring the generated code subdirectory from your working directory and locating the individual files.

File	Description
`rtwdemo_PCG_Eval_P1.c`	C file with step and initialization function
`rtwdemo_PCG_Eval_P1_data.c`	C file that assigns values to Real-Time Workshop data structures
`ert_main.c`	Example Main file that includes a simple scheduler
`rtwdemo_PCG_Eval_P1.h`	H file that defines data structures
`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

See Also

- “Supporting Model Referencing” in the Real-Time Workshop Embedded Coder documentation
- “Code Generation and the Build Process” 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 2-23

"Declaring Data and Functions" on page 2-23

"Controlling Data in Simulink® Software and Stateflow® Software" on page 2-26

"Adding New Data Objects" on page 2-29

"Configuring Data Objects " on page 2-30

"Controlling File Placement of Parameter Data" on page 2-30

"Enabling Data Objects in Generated Code" on page 2-31

"Effects of Simulation on Data Typing" on page 2-32

"Viewing Data Objects in Generated Code" on page 2-33

"Managing Data" on page 2-36

"See Also" on page 2-37

Introduction

This tutorial covers the specification of signals and parameters in the generated code.

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 and Functions

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

Data and Function 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	A value, a pointer to memory, or NULL

The combination of scope and duration is the *storage class*. If you do not provide an initial value, most compilers assign a zero value or a null pointer.

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	8-, 16-, 32-bit word lengths

Supported Storage Classes

Name	Description	Parameters Supported	Signals Supported	Data Types
`Const`	Use `const` type qualifier in declaration	Y	N	All

Supported Storage Classes (Continued)

Name	Description	Parameters Supported	Signals Supported	Data Types
`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
`BitField`	Embed Boolean data in a named bit field	Y	Y	Boolean

Supported Storage Classes (Continued)

Name	Description	Parameters Supported	Signals Supported	Data Types
`Define`	Represent parameters with a `#define` macro	Y	N	All
`Struct`	Embed data in a named struct to encapsulate sets of data	Y	Y	All

Controlling Data in Simulink® Software and Stateflow® Software

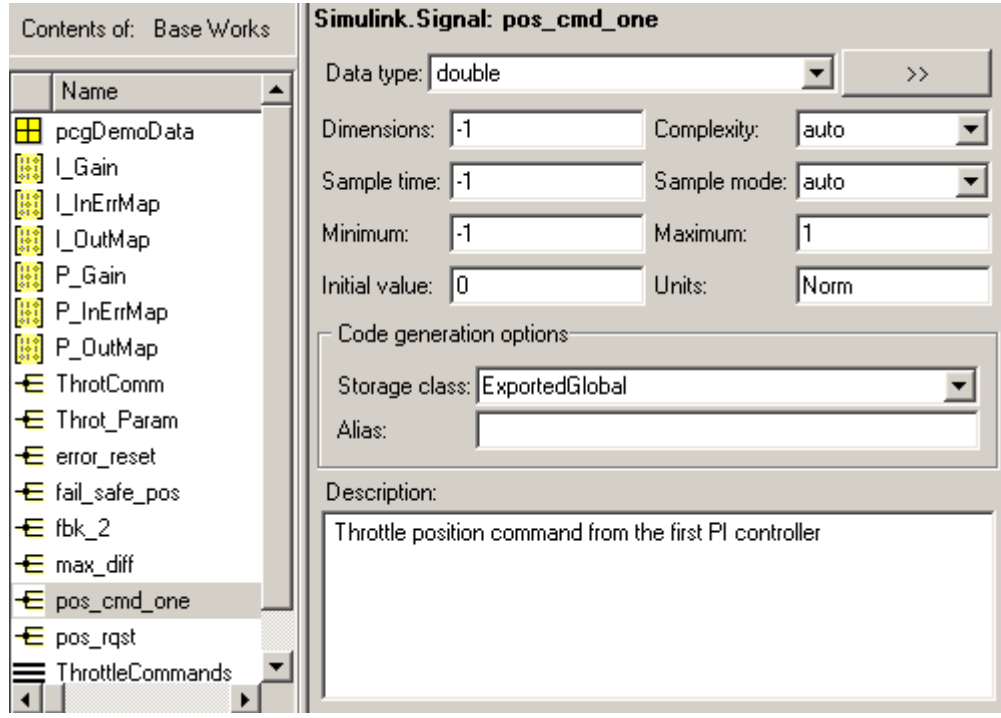
Two methods are available for declaring data in Simulink® software and Stateflow® software: *data objects* and *direct specification*. This tutorial uses the data object method. Both methods allow full control over the data type and storage class. You can mix the two methods in a single model.

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

The code generator uses data objects from the MATLAB base workspace. You can create and inspect them by entering commands at the MATLAB command line or by using the Model Explorer.

1. Open the model `rtwdemo_PCG_Eval_P2`.
2. Open the Model Explorer and view the `pos_cmd_one` data object in the base workspace.



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'
```

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.

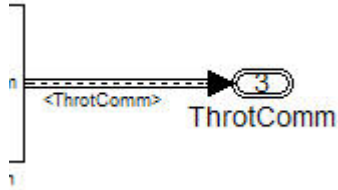
- Active fields:
  - Data type
  - Storage class
  - Value (parameters)
  - Initial value (signals)
  - Alias (define a different name in the generated code)
  - Dimension (inherited for parameters)
  - Complexity (inherited for parameters)
- Descriptive fields:
  - Minimum
  - Maximum
  - Units
  -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

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	Struct
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: **Tools** > **Data Object Wizard** in the model menu bar.
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 the 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 field settings in the **Data** pane to match those in the table.

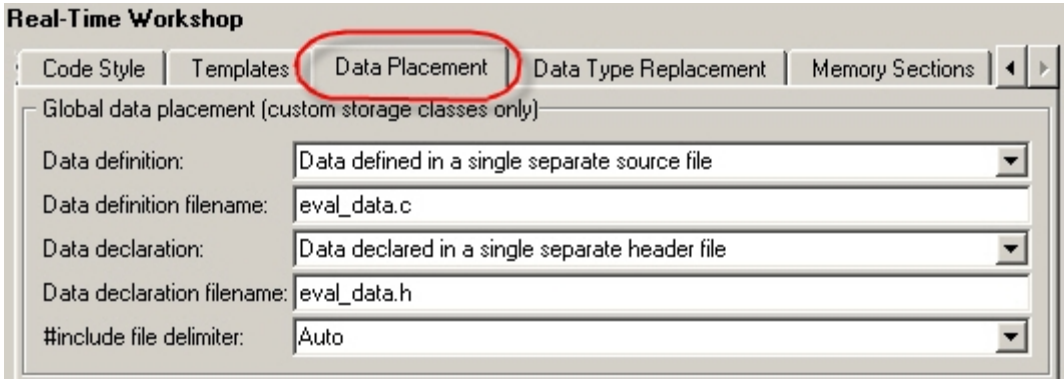
Data Object	Data Type	Storage Class
`fbk_1`	Double	Imported Extern
`pos_cmd_two`	Double	Exported Global

Controlling File Placement of Parameter Data

Real-Time Workshop® Embedded Coder™ software allows you to control the files in which the parameters and constants are defined. In this tutorial, all parameters were written to the file `eval_data.c`.

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

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



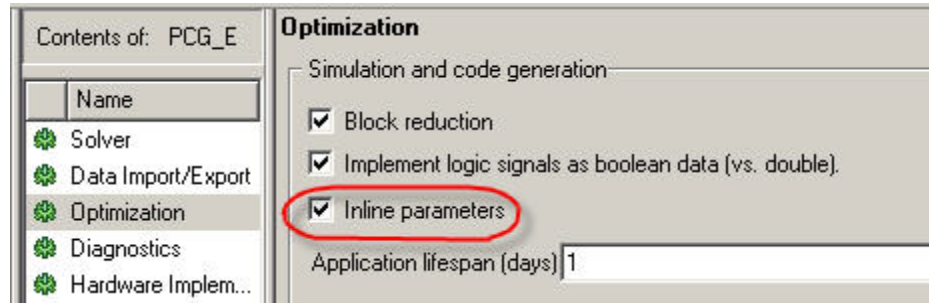
2 The generated code that results for `eval_data.c` is shown below:

```
/* 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 the Model Explorer, make sure the **Configuration** > **Optimizations** > **Inline parameters** check box is selected.

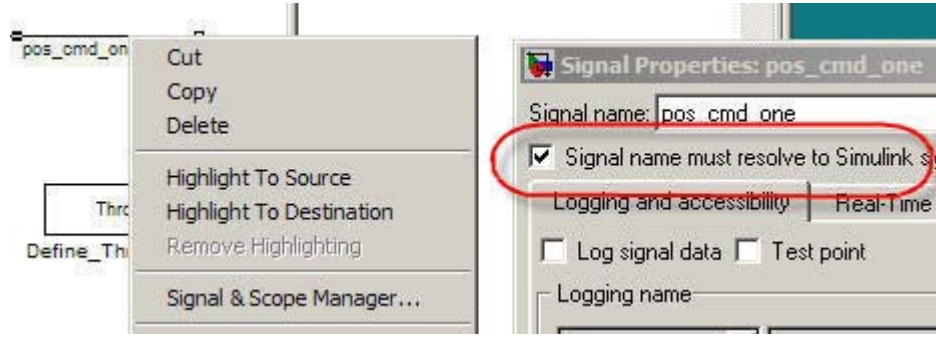


2 Enable a signal in generated code:

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 at the MATLAB command line

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

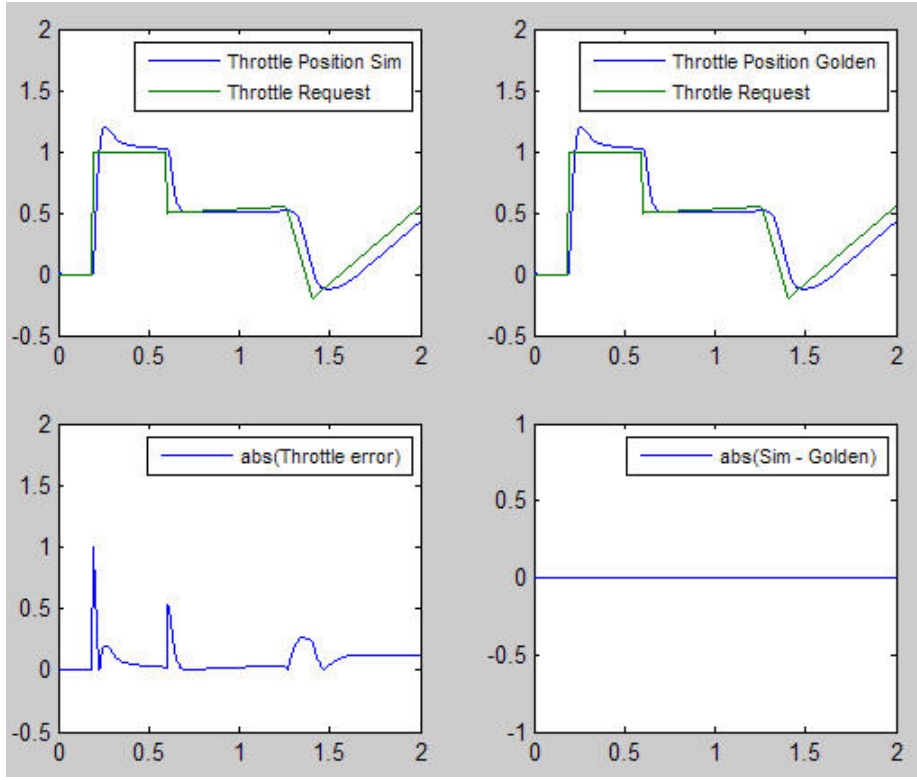
Effects of Simulation on Data Typing

In `rtwdemo_PCG_Eval_P2`, all data types are set to `double`. Since Simulink software uses the `double` data type for simulation, no changes are expected in the model behavior when you run the generated code. To verify this, run the test harness model. The test harness model is automatically updated to include the `rtwdemo_PCG_Eval_P2` model. That is the only change made to the test harness.

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

2 Start the test harness.

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

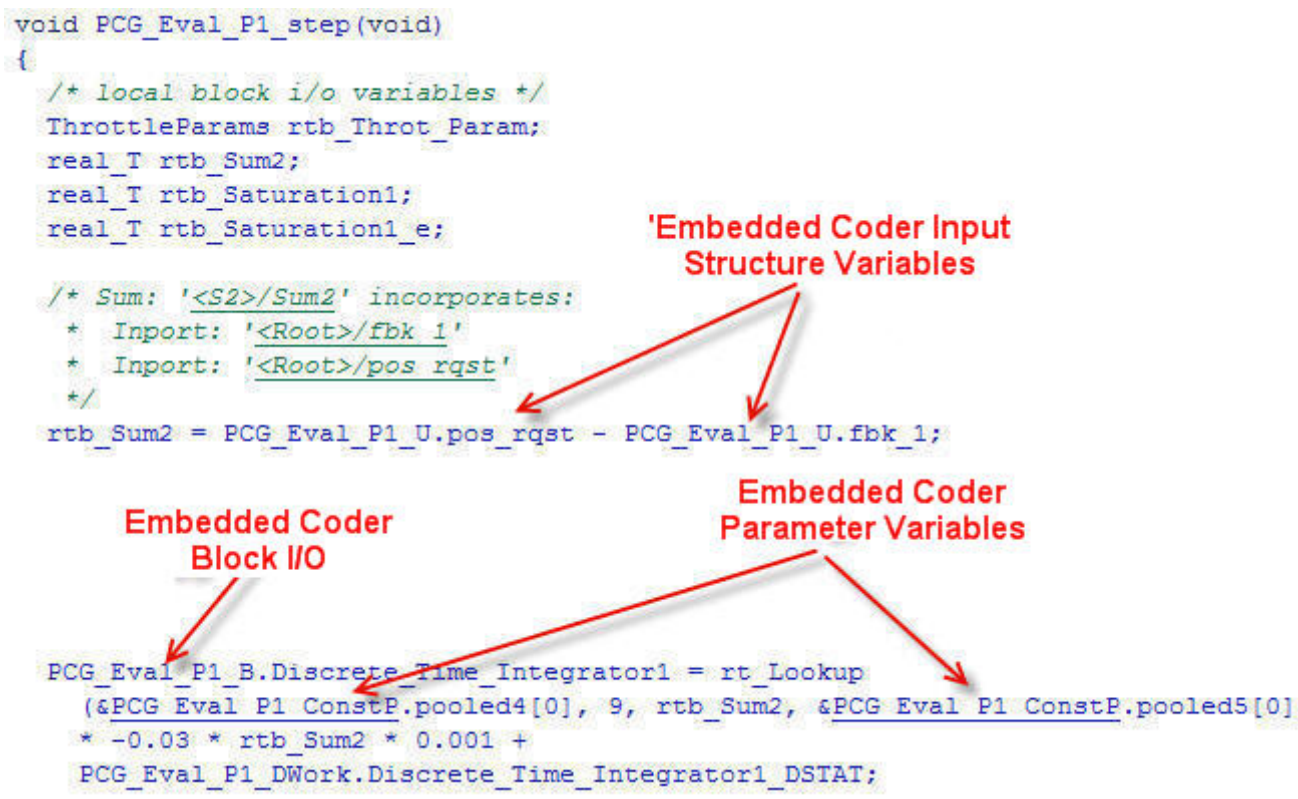


Viewing Data Objects in Generated Code

View the file `rtwdemo_PCG_Eval_P2.c` to see how the use of data objects changed the generated code.

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

The following figure shows the code for function `rtwdemo_PCG_Eval_P2_step` as it appears in `rtwdemo_PCG_Eval_P2.c` before the use of data objects.

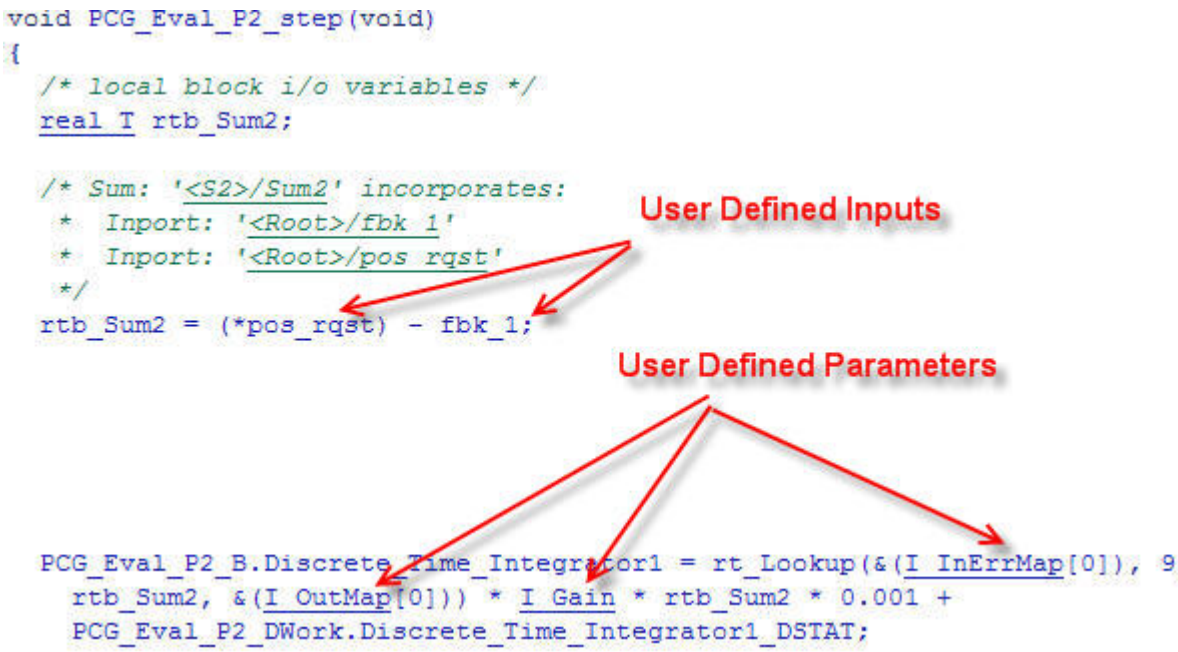



```
void PCG_Eval_P1_step(void)
{
  /* local block i/o variables */
  ThrottleParams rtb_Throt_Param;
  real_T rtb_Sum2;
  real_T rtb_Saturation1;
  real_T rtb_Saturation1_e;

  /* Sum: '<S2>/Sum2' incorporates:
   *  Inport: '<Root>/fbk_1'
   *  Inport: '<Root>/pos_rqst'
   */
  rtb_Sum2 = PCG_Eval_P1_U.pos_rqst - PCG_Eval_P1_U.fbk_1;

  PCG_Eval_P1_B.Discrete_Time_Integrator1 = rt_Lookup
    (&PCG_Eval_P1_ConstP.pooled4[0], 9, rtb_Sum2, &PCG_Eval_P1_ConstP.pooled5[0])
    * -0.03 * rtb_Sum2 * 0.001 +
    PCG_Eval_P1_DWork.Discrete_Time_Integrator1_DSTAT;
```

The figure below shows the code as it appears in `rtwdemo_PCG_Eval_P2.c` with data objects used.



This figure shows that most of the Real-Time Workshop® data structures have been replaced with user-defined data objects. The local variable `rtb_Sum2` and the state variable `rtwdemo_PCG_Eval_P2_DWork.Discrete_Time_Integrator1_DSAT` still use the Real-Time Workshop data structures.

Click the file names in the following table to view generated code.

Files generated for `rtwdemo_PCG_Eval_P2`

File	Definition	Notes
`rtwdemo_PCG_Eval_P2.c`	Provides step and initialization function	Uses the defined data objects
`eval_data.c`	Assigns values to the defined parameters	Has the file name specifically defined

Files generated for `rtwdemo_PCG_Eval_P2` (Continued)

File	Definition	Notes
`eval_data.h`	Provides extern definitions to the defined parameters	Has the file name specifically defined
`ert_main.c`	Provides scheduling functions	No change
`rtwdemo_PCG_Eval_P2.h`	Defines data structures	Using data objects shifted some parameters out of this file into `user_data.h`
`PCG_Eval_p2_private.h`	Defines private (local) data for the generated functions	Objects now defined in `eval_data` were removed
`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. They are saved 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)

See Also

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

Partitioning Functions in the Code

In this section...

"Introduction" on page 2-38

"About Atomic and Virtual Subsystems" on page 2-38

"Viewing Changes in the Model Architecture" on page 2-39

"Controlling Function Location and File Placement in Generated Code" on page 2-40

"Understanding Reentrant Code" on page 2-43

"Using a Mask to Pass Parameters into a Library Subsystem" on page 2-44

"Generating Code from an Atomic Subsystem" on page 2-45

"Generating Code: Full Model versus Exported Functions" on page 2-47

"Effect of Execution Order on Simulation Results" on page 2-48

"See Also" on page 2-50

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 required for integration
- How to generate code for atomic subsystems
- Data required 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 2-23 use *virtual subsystems*. Virtual subsystems visually organize blocks, but have no effect on the functional 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.

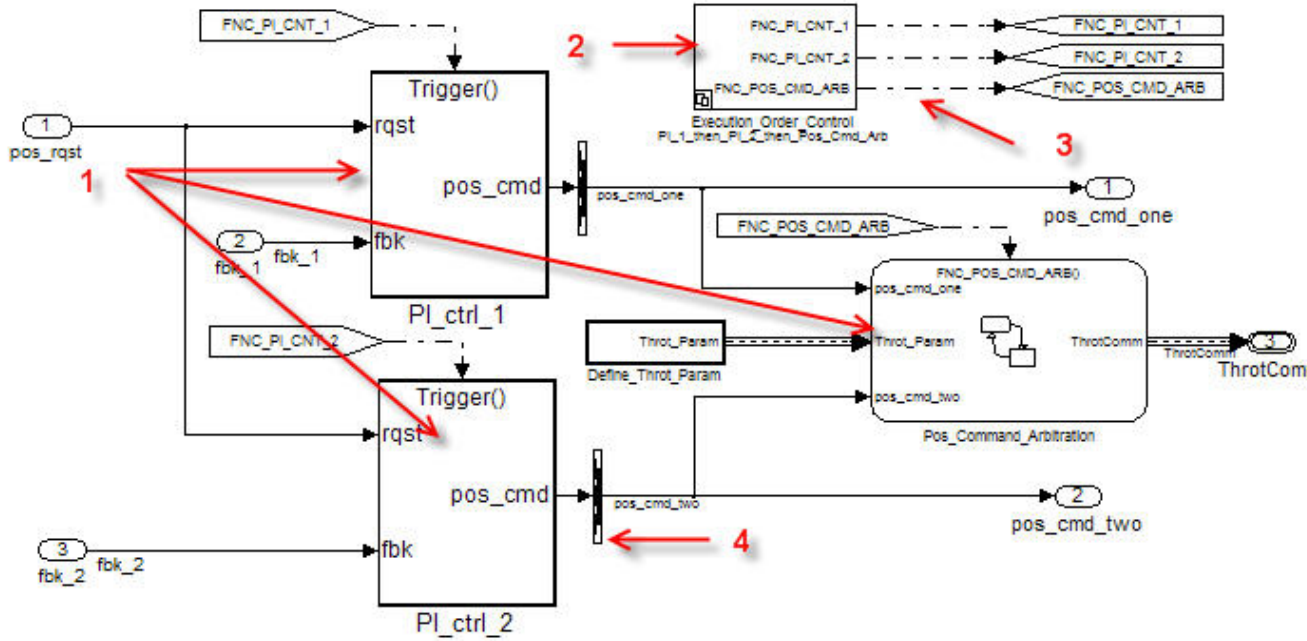
Viewing Changes in the Model Architecture

This module 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 (1) as `PI_ctrl_1`, `PI_ctrl_2`, and `Pos_Command_Arbitration`:



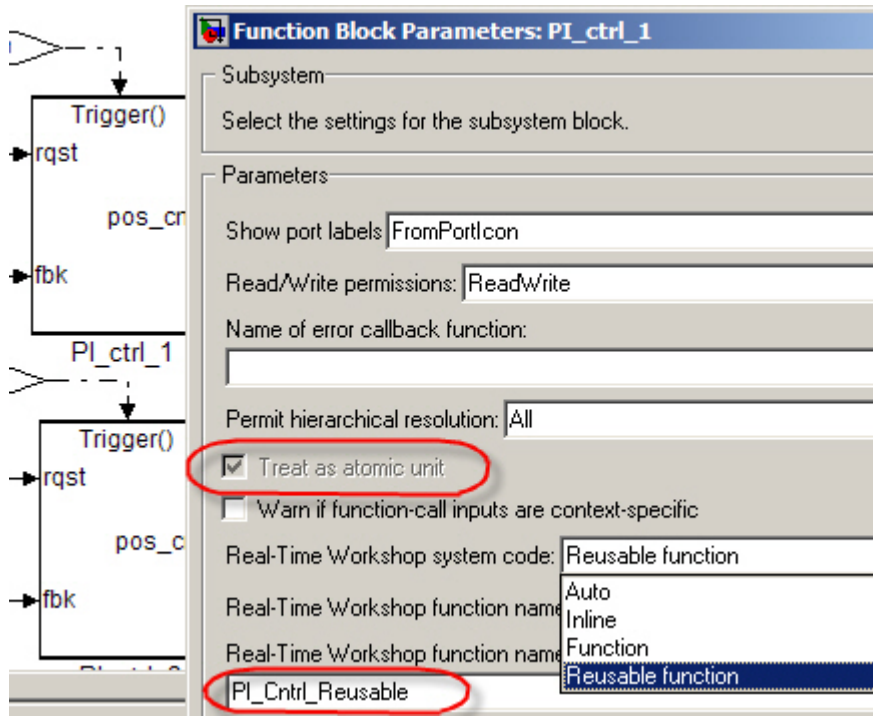
The subsystem `Execution_Order_Control` (2) has been added to the model. It is a Stateflow® chart that models the calling functionality of a scheduler. It controls the execution order of the function call subsystems. Later, this tutorial examines how changing execution order (3) can change the simulation results.

Four signal conversion blocks (4) were added to the outports for the PI controllers to make the functions reentrant. This is discussed in more detail later in this module.

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 2-23, 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.

The following figure shows the subsystem parameters for `PI_ctrl_1` and key parameters are described below the 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.

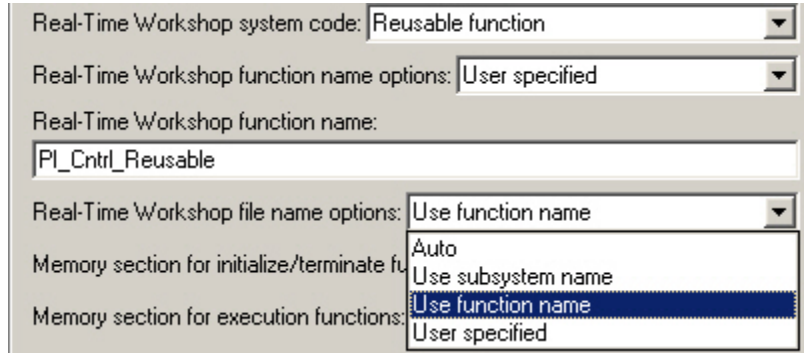
Parameter	What the Parameter Does
Real-Time Workshop system code	Depends on setting, as follows.
	`Auto`: Real-Time Workshop software determines how the subsystem appears in the generated code. This is the default.
	`Inline`: Real-Time Workshop software places the subsystem code inline with the rest 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. All input and output is passed into the function by argument or by reference. Global variables are not passed into the function.
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`: The function is based on the subsystem name.
	`User Specified`: You specify a unique file name.

Parameter	What the Parameter Does
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 module generated from the subsystem's parent system, or, if the subsystem's parent is the model itself, within the *`model`*`.c` 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 programming routine that can be used by multiple programs simultaneously. Reentrant code is used in operating systems and other system software that uses multithreading to handle concurrent events. Reentrant code does not maintain state data: there are no persistent variables in the function. Calling programs maintain their state variables and are required to pass them into the function. Any number of users or processes can share one copy of a reentrant routine.

To generate reentrant (reusable) code, you must first specify the subsystem as a candidate for reuse. You do this through the subsystem parameter 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 outport of the subsystem	Add a Signal Conversion block between the subsystem and the signal definition.
Passing data into the system as pointers	In the Model Explorer, select the **Configuration** > **Model Referencing** > **Pass 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.

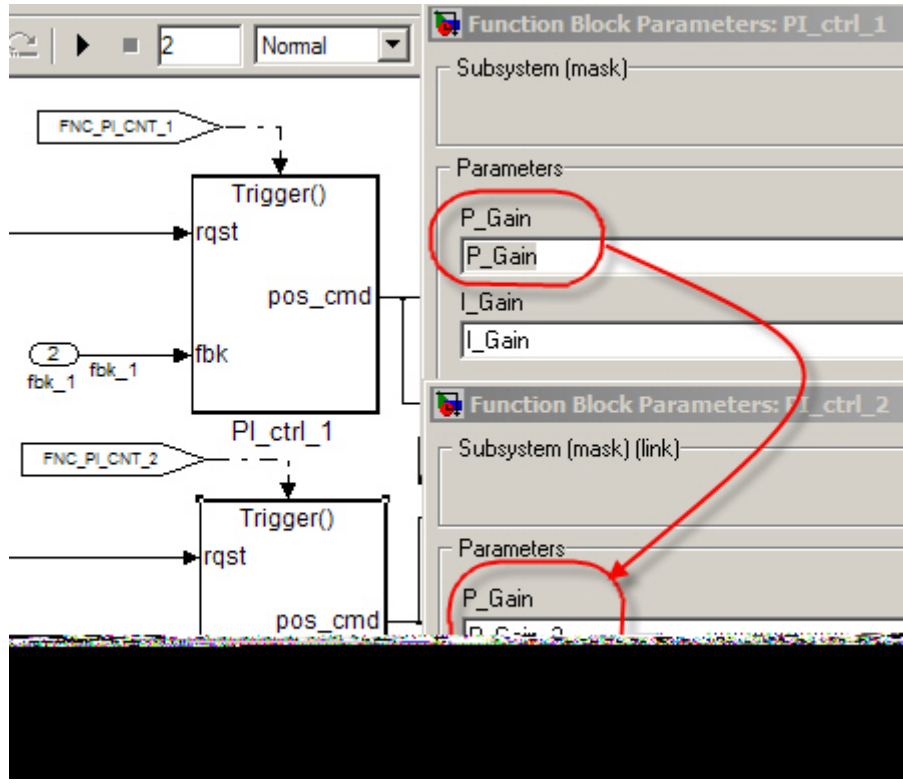
Using a Mask to Pass Parameters into a Library Subsystem

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 this model, the subsystems `PI_ctrl_1` and `PI_ctrl_2` have been masked. The value of the `P` and `I` gains are set in the subsystem mask. Two new data objects are created: `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 2-23, you generated code at the model root level. In addition to building at the system level, it is possible to build at the subsystem level, as the following figure shows.

- **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 we see 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:** `PCG_Eval_File_3.zip`
- **Data Type:** All singles
- **Included Data:** All data required for the build is included 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 2-58
- **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 they 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

A

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

Installing the Eclipse IDE and Cygwin Debugger (p. A-2)

Installing the Eclipse IDE and the Cygwin debugger for use with the integration and testing tutorials.

Integrating and Testing Code with the Eclipse IDE (p. A-4)

Testing and debugging generated code with the Eclipse IDE and the Cygwin debugger.

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.
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. 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.
2. Run the file `setup.exe`. 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's `Path` variable. For example, on an Microsoft 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**.

Integrating and Testing Code with the Eclipse IDE

In this section...

"Introducing Eclipse" on page A-4
"Creating a New CDT-Managed Make C Project" on page A-5
"Configuring the Debugger" on page A-6
"Starting the Debugger" on page A-8
"Setting the Cygwin Path" on page A-9
"What the Eclipse Debugger Can Do" on page A-10

Introducing Eclipse

Eclipse (`www.eclipse.org`) is an Integrated Development Environment for developing and debugging embedded software. Cygwin (`www.cygwin.com`) is a Linux-like environment for Windows that 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 2-58 and "Testing the Generated Code" on page 2-66. 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.

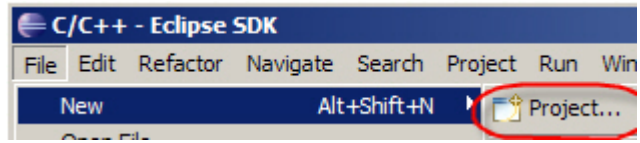
About Project Names and Filenames Used in This Section

"Integrating the Generated Code into the External Environment" on page 2-58 and "Testing the Generated Code" on page 2-66 both use the instructions in this section, but the project names and filenames differ. Where you see ## in a project name or filename in this section, substitute:

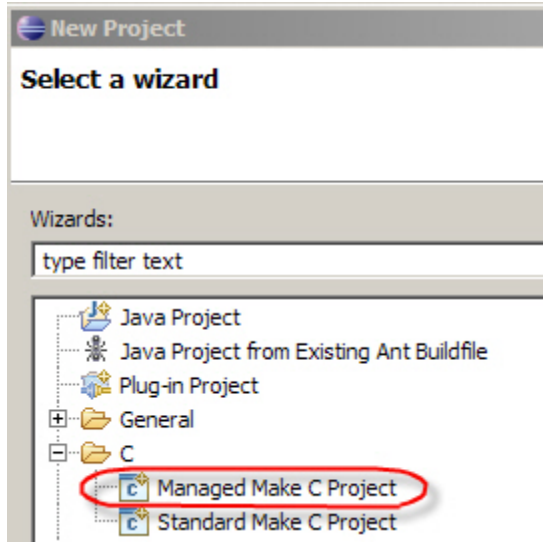
- `P6` if you are working in “Integrating the Generated Code into the External Environment” on page 2-58
- `P7` if you are working in“Testing the Generated Code” on page 2-66

Creating a New CDT-Managed Make C Project

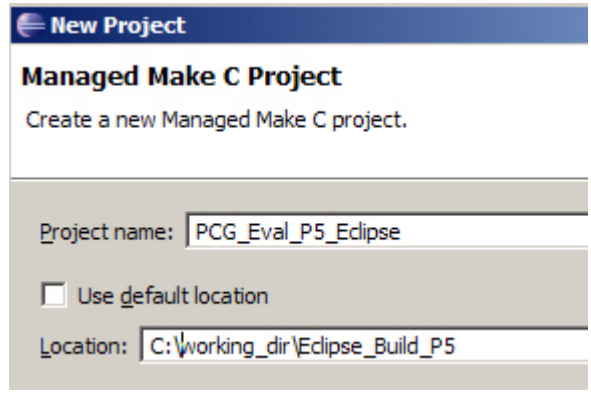
1 In Eclipse, choose **File > New > Project**.



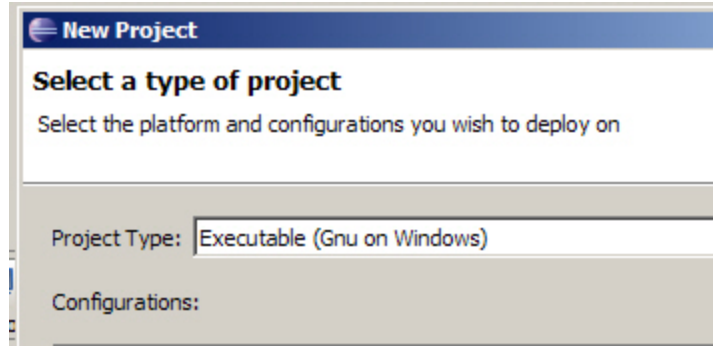
2 In the New Project dialog, click **C > Managed Make C Project**.



3 Set the **Project name** to `PCG_Eval_##_Eclipse` (where `##` is `P5` or `P6`) and set the **Location** to the build directory.

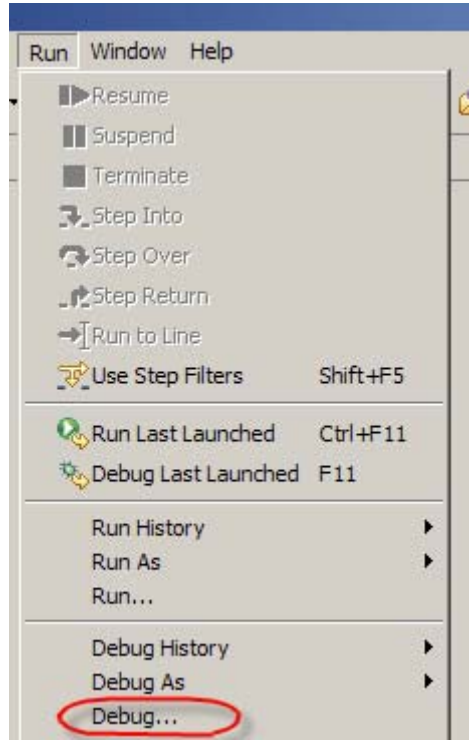


4 Set the **Project Type** to **Executable (Gnu on Windows)**

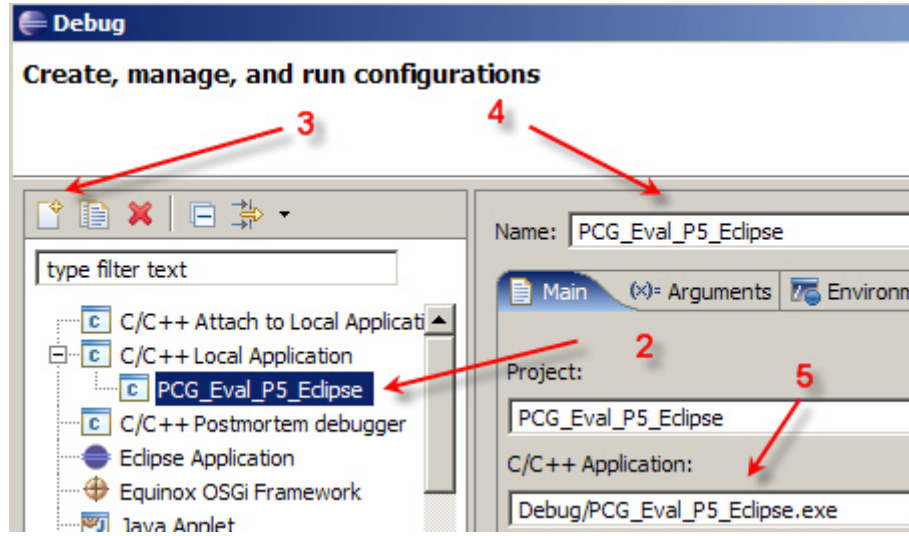


Configuring the Debugger

1 In Eclipse, choose **Run > Debug**.



2 Click node **C/C++ Local Application**.



3 Click the **Launch New Configuration** toolbar button.

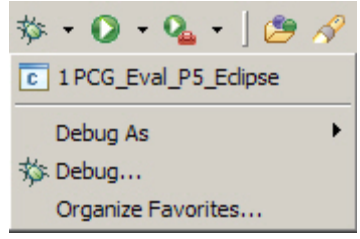
4 Click the project name **PCG_Eval_##_Debug**.

The project name appears in the **Name** field.

5 Select the executable file.

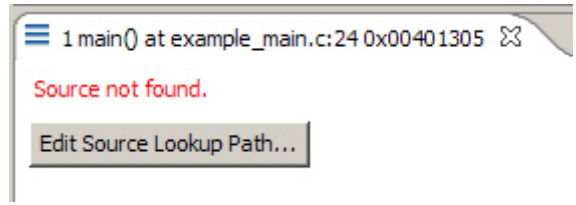
Starting the Debugger

1 Click the **Debug** toolbar button and from the menu select project `PCG_Eval_##_Debug`.



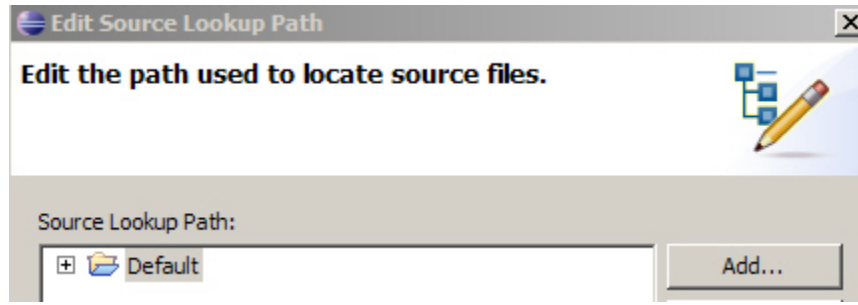
Setting the Cygwin Path

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



To provide the necessary path information:

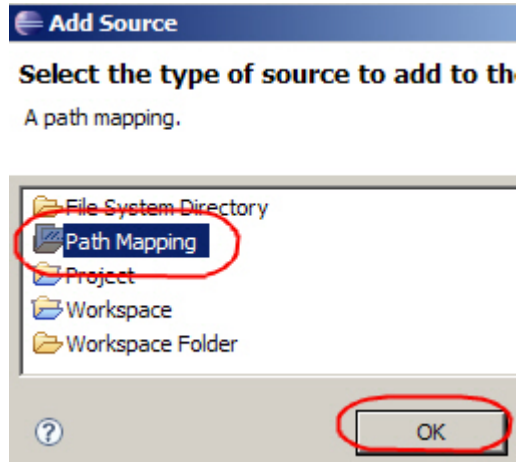
1 Click **Edit Source Lookup Path** in the error message.



2 Click the **Add** button 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 the **OK** button. The Edit Source Lookup Path dialog box appears.

5 In the Edit Source Lookup Path dialog box, select **Path Mapping**.

6 Click the **Edit** button.

The Path Mappings dialog box appear.

7 In the Path Mappings dialog box, click the **Add** button.

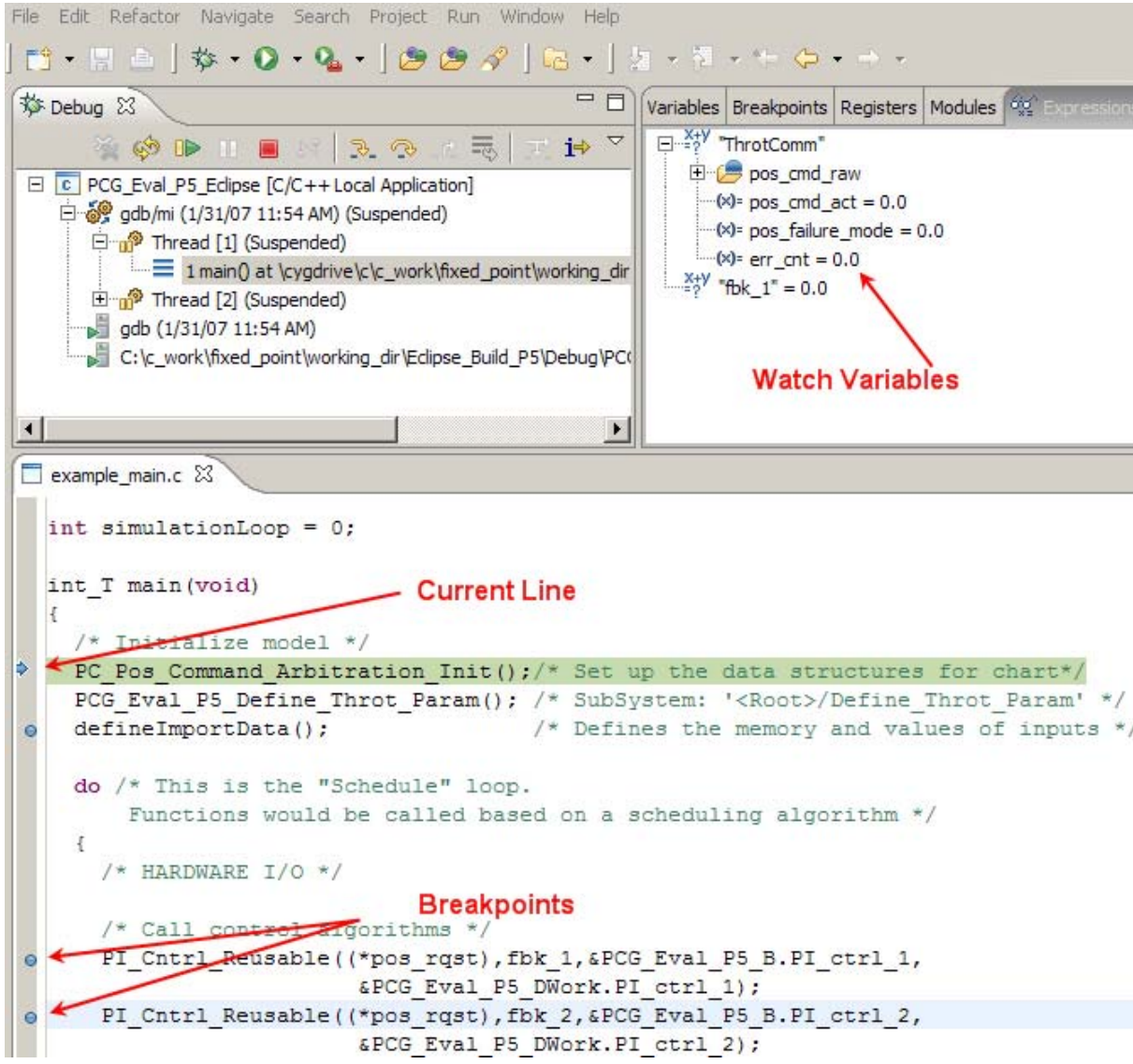
8 Type `\cygdrive\c\` in the **Compilation path** field.

9 Type `c:\` in the **Local file system path** field.

10 Click the **OK** button.

What the Eclipse Debugger Can Do

The next figure shows the Eclipse debugger open on a project.



Actions and Commands

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