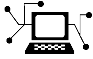
Embedded Coder™ 6
Getting Started Guide

MATLAB®
&SIMULINK®

MathWorks®

How to Contact MathWorks

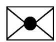


www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

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.

Embedded Coder™ Getting Started Guide

© COPYRIGHT 2011 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

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

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)

Contents

1 Product Overview

When to Use Embedded Coder **1-2**

Related Products **1-3**

Prerequisite Knowledge **1-6**
MATLAB Users **1-6**
Simulink Users **1-6**

About Embedded Coder and MathWorks C/C++ Code Generation **1-8**

Algorithm Development Workflows **1-11**

Target Environments and Applications **1-14**
About Target Environments **1-14**
Types of Target Environments Supported By Embedded Coder **1-14**
Applications of Supported Target Environments **1-16**

V-Model for System Development **1-19**
What Is the V-Model? **1-19**
Types of Simulation and Prototyping **1-21**
Types of In-the-Loop Testing for Verification and Validation **1-22**

2 MATLAB Tutorials

About the Tutorials **2-2**

About MATLAB® Coder 2-2
How Embedded Coder Works With MATLAB® Coder 2-2
Prerequisites 2-3
Setting Up Tutorial Files 2-3

Controlling C Code Style **2-4**
About This Tutorial 2-4
Copying Files Locally 2-5
Setting Up the MATLAB® Coder Project 2-5
Configuring Build Parameters 2-6
Generating the C Code 2-7
Viewing the Generated C Code 2-7
Key Points to Remember 2-8
Learn More 2-8

Generating Reentrant C Code from MATLAB Code **2-9**
About This Tutorial 2-9
Copying Files Locally 2-10
About the Example 2-11
Providing a main Function 2-12
Configuring Build Parameters 2-15
Generating the C Code 2-15
Viewing the Generated C Code 2-15
Running the Code 2-16
Key Points to Remember 2-17
Learn More 2-17

Tracing Between Generated C Code and MATLAB Code **2-18**
About This Tutorial 2-18
Copying Files Locally 2-19
Configuring Build Parameters 2-20
Generating the C Code 2-20
Viewing the Generated C Code 2-20
Tracing Back to the Source MATLAB Code 2-21
Key Points to Remember 2-21
Learn More 2-22

3 Simulink Tutorials

About the Tutorials **3-2**
Introduction **3-2**
Prerequisites **3-3**
Third-Party Software **3-3**
Required Files **3-3**

Tutorial – Configuring a Model and Generating Code for an Embedded System **3-5**
About this Tutorial **3-5**
Configuring a Model for Embedded System Code Generation **3-6**
Checking the Model for Adverse Conditions and Settings for Embedded Systems Code **3-11**
Generating Code for the Model **3-12**
Reviewing the Generated Code **3-13**
Key Points **3-14**
Learn More **3-14**

Tutorial – Controlling the Appearance of Generated Code **3-15**
About this Tutorial **3-15**
Customizing Code Comments **3-16**
Customizing the Appearance of Identifiers **3-18**
Customizing Code Style **3-19**
Key Points **3-20**
Learn More **3-20**

Tutorial – Configuring the Data Interface **3-21**
About this Tutorial **3-21**
Creating Data Objects for Named Data in Base Workspace **3-22**
Configuring Data Objects **3-22**
Controlling Placement of Parameter and Constant Data in Generated Code **3-23**
Including Signal Data Objects in Generated Code **3-25**
Effects of Simulation on Data Typing **3-26**
Viewing Data Objects in Generated Code **3-27**
Saving Base Workspace Data **3-28**
Key Points **3-28**

Learn More 3-28

Tutorial – Partitioning and Exporting Functions in the Generated Code **3-29**
About this Tutorial 3-29
Changing Model Architecture to Control Execution Order 3-30
Controlling Function Location and File Placement in Generated Code 3-32
Using a Mask to Pass Parameters into a Library Subsystem 3-35
Generating Code for the Full Model and Exported Functions 3-37
Changing the Execution Order and Simulation Results .. 3-39
Key Points 3-41
Learn More 3-42

Tutorial – Integrating Generated Code into an External Environment **3-43**
About this Tutorial 3-43
Relocating Code to Another Development Environment .. 3-44
Integrating Generated Code into an Existing System 3-45
Setting Up the Main Function 3-45
Matching the System Interfaces 3-47
Building a Project in the Eclipse Environment 3-50
Key Points 3-51
Learn More 3-51

Tutorial – Verifying Generated Code **3-53**
About this Tutorial 3-53
Methods for Verifying Generated Code 3-54
Reusing Test Data By Importing and Exporting Test Vectors 3-55
Verifying Behavior of a Model with Software-in-the-Loop Testing 3-56
Verifying System Behavior By Importing and Exporting Test Vectors 3-59
Testing via Processor-in-the-Loop (PIL) 3-62
Key Points 3-62
Learn More 3-62

Tutorial – Evaluating the Generated Code **3-63**

About this Tutorial 3-63
Evaluating Code 3-63
About the Compiler 3-64
Viewing the Code Metrics 3-64
About the Build Configurations 3-64
Configuration 1: Reusable Functions, Data Type Double . . 3-65
Configuration 2: Reusable Functions, Data Type Single . . 3-66
Configuration 3: Nonreusable Functions, Data Type Single 3-67

A Installing and Using an IDE for the Integration and Testing Tutorials

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-5**
Introducing Eclipse A-5
Defining a New C Project A-6
Configuring the Debugger A-7
Starting the Debugger A-7
Setting the Cygwin Path A-8
Actions and Commands in the Eclipse Debugger A-8

Product Overview

- "When to Use Embedded Coder" on page 1-2
- "Related Products" on page 1-3
- "Prerequisite Knowledge" on page 1-6
- "About Embedded Coder and MathWorks C/C++ Code Generation" on page 1-8
- "Algorithm Development Workflows" on page 1-11
- "Target Environments and Applications" on page 1-14
- "V-Model for System Development" on page 1-19

When to Use Embedded Coder

Embedded Coder™ generates readable, compact, and fast C and C++ code for use on embedded processors, on-target rapid prototyping boards, and microprocessors used in mass production. Embedded Coder enables additional MATLAB® Coder™ and Simulink® Coder™ configuration options and advanced optimizations for fine-grain control of the generated code's functions, files, and data. These optimizations improve code efficiency and facilitate integration with legacy code, data types, and calibration parameters used in production. You can incorporate a third-party development environment into the build process to produce an executable for turnkey deployment on your embedded system.

Embedded Coder offers built-in support for AUTOSAR and ASAP2 software standards. It also provides traceability reports, code interface documentation, and automated software verification to support DO-178, IEC 61508 and ISO 26262 software development.

Learn more about MathWorks support for certification in automotive (http://www.mathworks.com/automotive/standards/iso-26262.html), aerospace (http://www.mathworks.com/aerospace-defense/), and industrial automation (http://www.mathworks.com/industrial-automation-machinery/) applications.

Related Products

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

Product	Extends Code Generation Capabilities for ...
Aerospace Blockset™	Aircraft, spacecraft, rocket, propulsion systems, and unmanned airborne vehicles
Communications System Toolbox™	Physical layer of communication systems
Computer Vision System Toolbox™	Video processing, image processing, and computer vision systems
Control System Toolbox™	Linear control systems
DSP System Toolbox™	Signal processing systems
Fuzzy Logic Toolbox™	System designs based on fuzzy logic
Gauges Blockset™	Linking generated code executing on a target system with graphical instrumentation in a Simulink® model
MATLAB Coder	Integrating C/C++ code generated from MATLAB® code
Model-Based Calibration Toolbox™	Developing processes for systematically identifying optimal balance of engine performance, emissions, and fuel economy, and reusing statistical models for control design, hardware-in-the-loop testing, or powertrain simulation
Model Predictive Control Toolbox™	Controllers that optimize performance of multi-input and multi-output systems that are subject to input and output constraints
Real-Time Windows Target™	Rapid prototyping or hardware-in-the-loop simulation of control system and signal processing algorithms
SimDriveline™	Driveline (drivetrain) systems

Product	Extends Code Generation Capabilities for ...
SimElectronics®	Electronic and electromechanical systems
SimHydraulics®	Hydraulic power and control systems
SimMechanics™	Three-dimensional mechanical systems
SimPowerSystems™	Systems that generate, transmit, distribute, and consume electrical power
Simscape™	Systems spanning mechanical, electrical, hydraulic, and other physical domains as physical networks
Simulink Coder	Rapid prototyping, host-based simulation, and hardware-in-the-loop (HIL) simulation of control system and signal processing algorithms
Simulink® Fixed Point™	Control and signal processing systems implemented with fixed-point arithmetic
Simulink® 3D Animation™	Systems with 3D visualizations
Simulink® Design Optimization™	Systems requiring maximum overall system performance
Simulink® Report Generator™	Automatically generating project documentation in a standard format
Simulink® Verification and Validation™	Applications requiring automated requirements tracing, model standards compliance checking, and test harness generation

Product	Extends Code Generation Capabilities for ...
System Identification Toolbox™	Systems constructed from measured input-output data Support exceptions: • Nonlinear IDNLGREY Model, IDDATA Source, IDDATA Sink, and estimator blocks • Nonlinear ARX models that contain custom regressors • `neuralnet` nonlinearities • `customnet` nonlinearities
Vehicle Network Toolbox™	Support exception: CAN Configuration, CAN Receive, and CAN Transmit blocks in the CAN Communication library
xPC Target™	Rapid control prototyping, hardware-in-the-loop (HIL) simulation, and other real-time testing applications
xPC Target Embedded Option™	Deploying real-time embedded systems on a PC for production, data acquisition, calibration, and testing applications

Prerequisite Knowledge

In this section...
"MATLAB Users" on page 1-6
"Simulink Users" on page 1-6

MATLAB Users

Be familiar with:

- "MATLAB Coder"
- "Code Generation from MATLAB"
- MATLAB Function block

If you are familiar with C language constructs and want to learn how to map commonly used C constructs to code generated from MATLAB program design patterns, see "Developing Model Patterns that Generate Specific C Constructs" in the Embedded Coder documentation.

Simulink Users

Be familiar with:

- 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.
- Generating code and building executable programs from Simulink models.
- High-level programming language concepts applied to embedded, real-time systems.

While you do not need to program in C or other programming languages to create, test, and deploy embedded systems, using Embedded Coder software, successful emulation and deployment of embedded systems requires familiarity with parameters and design constraints. The Embedded Coder documentation assumes you have a basic understanding of real-time and embedded system concepts, terminology, and environments.

If you have not done so, you should study:

- The tutorials in the *Simulink Coder Getting Started Guide*. The tutorials provide hands-on experience in configuring models for code generation and generating code, setting up a data interface, and integrating external code.
- “Laying Out the Model Architecture” and “Scheduling Considerations” in the Simulink Coder documentation. These sections give a general overview of the architecture and execution of generated code.

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 MATLAB function blocks, see “Developing Model Patterns that Generate Specific C Constructs”.

About Embedded Coder and MathWorks C/C++ Code Generation

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

You apply code generation explicitly with the following coder products:

The Embedded Coder product *extends* the MATLAB Coder and Simulink Coder products with features that are important for embedded software development. Using the Embedded Coder add-on product, you 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 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 of each of the C/C++ coder products:

Product	Typical Applications	Key Capabilities
MATLAB Coder	Accelerate MATLABalgorithms. Verify generated C code within MATLAB Integrate custom C/C++ code into MATLAB	Generate readable, efficient, standalone C/C++ code for MATLABalgorithms. Supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. Generate MEX functions for accelerating computationally intensive portions of MATLAB code and verifying the behavior of generated code.
Simulink Coder	Simulation acceleration Simulink model intellectual property protection 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 environments. Generate code for finite state machines modeled in Stateflow event-based modeling software. Generate code for many MathWorks and third-party Simulink products and blocksets

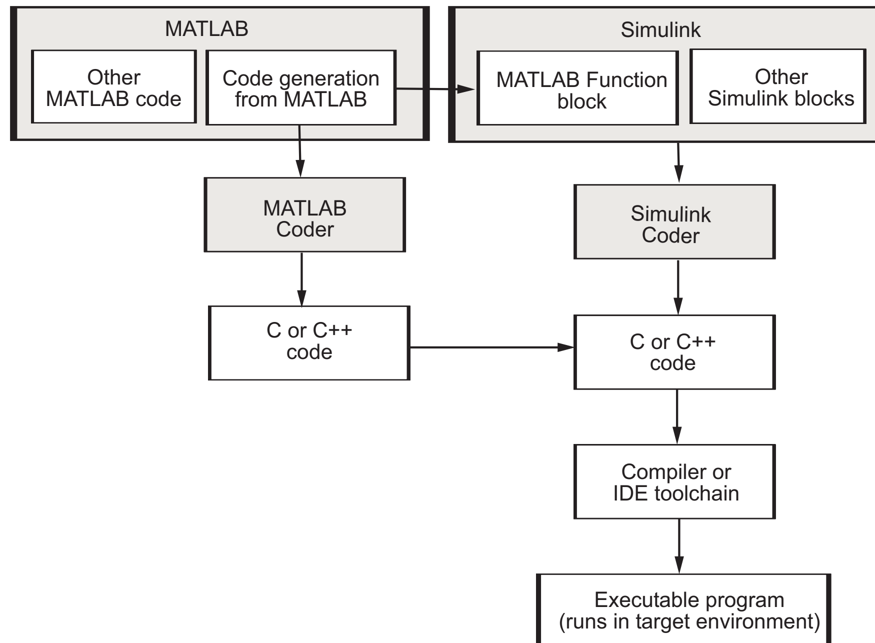
Product	Typical Applications	Key Capabilities
		Integrate existing applications, functions, and data.
Embedded Coder	All applications listed for the MATLAB Coder and Simulink Coder products Embedded systems On-target rapid prototyping boards Microprocessors in mass production	All capabilities listed for the MATLAB Coder and Simulink Coder products. Generate code that has the clarity and efficiency of professional handwritten code. Generate reentrant code. Customize the appearance and performance of the code for specific target environments. Enable tracing, reporting, and testing options that facilitate code verification activities.

Algorithm Development Workflows

You can use MathWorks code generation technology to generate standalone C or C++ source code for embedded systems:

- By developing MATLAB algorithms with Code Generation from MATLAB and then generating C/C++ code with the "MATLAB Coder" and Embedded Coder products
- By developing Simulink models and Stateflow charts and then generating C/C++ code with the "Simulink Coder" and Embedded Coder products
- By integrating MATLAB code into Simulink models, using Code Generation from MATLAB and the Simulink MATLAB Function block, and then generating C/C++ code with the "Simulink Coder" and Embedded Coder products

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



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

If you develop algorithms using...	You generate code by...	For more information, see...
Code generation from MATLAB	Using MATLAB Coder projects Entering the function `codegen` in the MATLAB Command Window	"Workflow Overview" in the MATLAB Coder documentation.
Simulink and Stateflow	Configuring and initiating code generation for your model or subsystem with the Simulink Configuration Parameters dialog.	"Simulink and Stateflow Model Workflow" in the Simulink Coder documentation

If you develop algorithms using...	You generate code by...	For more information, see...
Code generation from MATLAB and Simulink	Including MATLAB code in Simulink models or subsystems by using the MATLAB Function block. To use this block, you can do one of the following: • Copy your code into the block. • Call your code from the block by referencing the appropriate files on the MATLAB path.	"Code Generation from MATLAB" documentation MATLAB Function block in the Simulink documentation

To use MATLAB code and Simulink models for a Model-Based Design project:

- Start by using MATLAB to develop an algorithm for research and early development.
- Later want to integrate the algorithm into a graphical model for system deployment and verification.

Benefits of this approach include:

- Richer system simulation environment
- Ability to verify the MATLAB code
- Simulink Coder and Embedded Coder C/C++ code generation for the model and MATLAB code

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 MATLAB functions, see "Developing Model Patterns that Generate Specific C Constructs" in the Embedded Coder documentation.

Target Environments and Applications

In this section...

"About Target Environments" on page 1-14
"Types of Target Environments Supported By Embedded Coder" on page 1-14
"Applications of Supported Target Environments" on page 1-16

About Target Environments

In addition to generating source code for a model or subsystem, the code generator produces make or project files to build an executable for a specific target environment. The generated make or project files are optional. 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 generated code 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.

The code generator 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.

Types of Target Environments Supported By Embedded Coder

Before you select a system target file, identify the target environment on which you expect to execute your generated code. The most common target environments include those environmens listed in the following table.

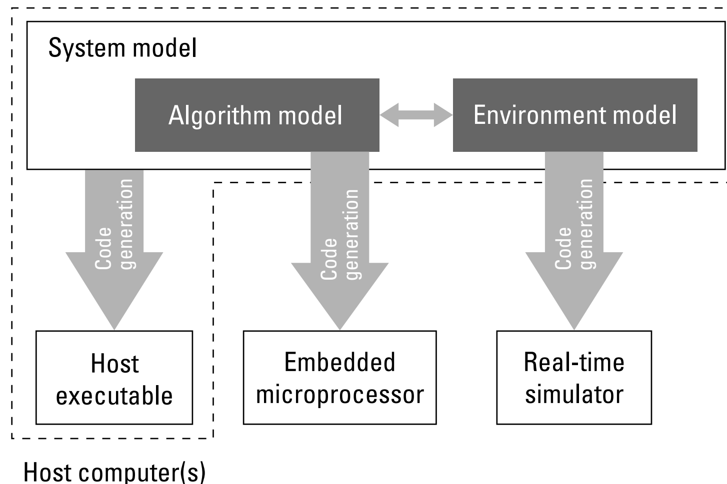
Target Environment	Description
Host computer	The same computer that runs MATLAB and Simulink. Typically, a host computer is a PC or UNIX®[1] environment that uses a non-real-time operating system, such as Microsoft® Windows® or Linux®[2]. Non-real-time (general purpose) operating systems are nondeterministic. For example, those operating systems might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Therefore, the executable for your generated code might run faster or slower than the sample rates that you specified in your model.
Real-time simulator	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: • 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 The generated code runs in real time and behaves deterministically. The exact nature of execution varies based on the particular behavior of the system hardware and RTOS. Typically, a real-time simulator connects to a host computer for data logging, interactive parameter tuning, and Monte Carlo batch execution studies.
Embedded microprocessor	A computer that you eventually disconnect from a host computer and run as a standalone computer as part of an electronics-based product. Embedded microprocessors range in price and performance, from high-end digital signal processors (DSPs) to process communication signals to inexpensive 8-bit fixed-point microcontrollers in mass production (for example, electronic parts produced in the millions of units). Embedded microprocessors can: • Use a full-featured RTOS • Be driven by basic interrupts • Use rate monotonic scheduling provided with code generation

A target environment can:

- Have single- or multiple-core CPUs
- Be a standalone computer 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 of Supported Target Environments

The following table lists several ways that you can apply code generation technology in the context of the different target environments.

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.

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 environments. 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 nonreal time on the host computer, but outside the context of the MATLAB and Simulink environments.
System simulation	You integrate components into a larger system. You provide generated source code and related dependencies for building a system in another environment or in a host-based shared library to which other code can dynamically link.
Model intellectual property protection	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 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 for intellectual property protection.
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	

Application	Description
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.
"Verifying Generated Code With SIL and PIL Simulations"	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, you might use actual target word sizes and just test production code behavior.
"Verifying Generated Code With SIL and PIL Simulations"	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.

V-Model for System Development

In this section...

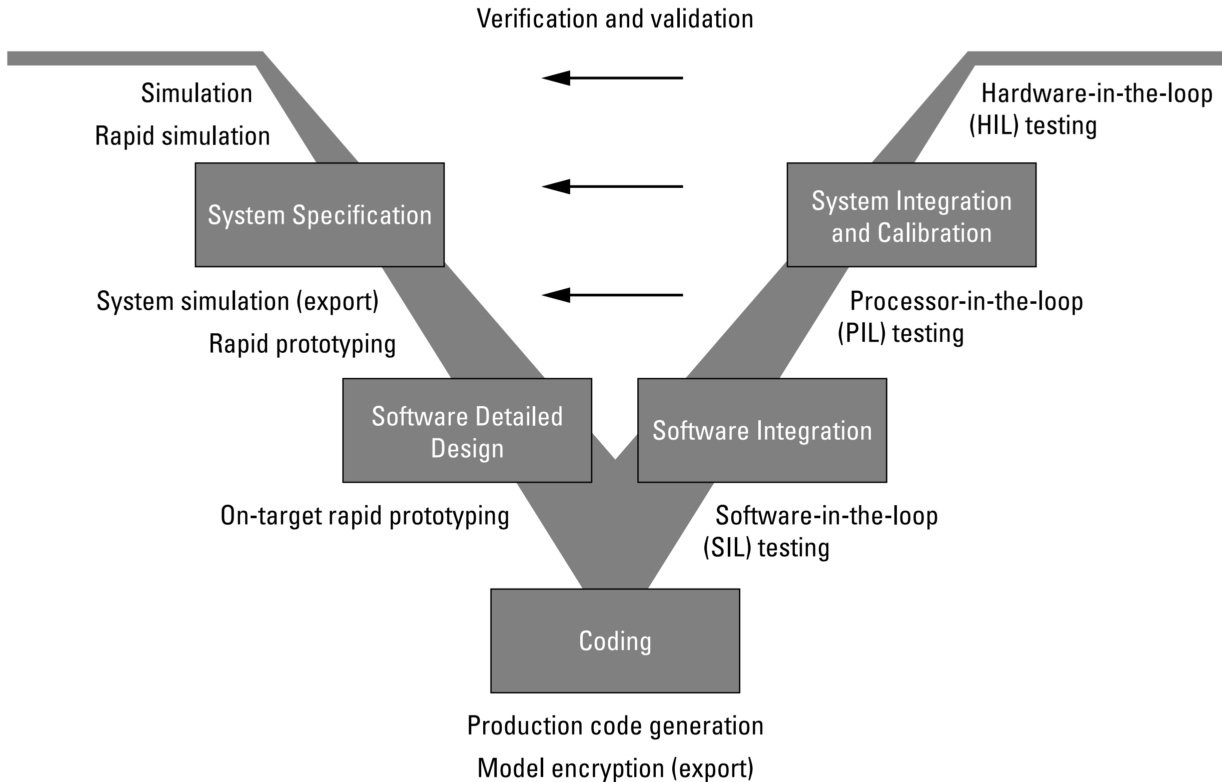
"What Is the V-Model?" on page 1-19

"Types of Simulation and Prototyping" on page 1-21

"Types of In-the-Loop Testing for Verification and Validation" on page 1-22

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 of the V focuses on the verification and validation of steps cited on the left side, including software integration and system integration.



Depending on your application and its role in the process, you might focus on one or more of the steps called out in the V-model or repeat steps at several stages of the V-model. Code generation technology and related products provide tooling that 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 an information map on applications of code generation 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 nonreal 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 environments	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	Can simulate component (algorithm or controller) and environment (or plant) Normal mode simulation in Simulink enables you to access, display, and tune data during verification Can accelerate Simulink simulations with Accelerated and Rapid Accelerated modes	Easy to simulate models of hybrid dynamic systems that include components and environment models Ideal for batch or Monte Carlo simulations Can repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model Can connect to Simulink to monitor signals and tune parameters	Might require custom real-time simulators and hardware Might be done with inexpensive off-the-shelf PC hardware and I/O cards	Might use existing hardware, therefore less expensive and more convenient

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 because 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 only in Simulink No cost for hardware	Executes on desk or test bench Uses hardware — process board and cables	Desktop convenience Executes only 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

MATLAB Tutorials

- "About the Tutorials" on page 2-2
- "Controlling C Code Style" on page 2-4
- "Generating Reentrant C Code from MATLAB Code" on page 2-9
- "Tracing Between Generated C Code and MATLAB Code" on page 2-18

About the Tutorials

In this section...

"About MATLAB® Coder" on page 2-2
"How Embedded Coder Works With MATLAB® Coder" on page 2-2
"Prerequisites" on page 2-3
"Setting Up Tutorial Files" on page 2-3

About MATLAB Coder

MATLAB Coder generates standalone C and C++ from MATLAB code. The generated source code is portable and readable. MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. It can generate MEX functions that let you accelerate computationally intensive portions of MATLAB code and verify the behavior of the generated code.

How Embedded Coder Works With MATLAB Coder

The Embedded Coder product extends the MATLAB Coder product with features that are important for embedded software development. Using the Embedded Coder add-on product, you 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
- Enable tracing options that help you to verify the generated code
- Generate reusable, reentrant code

Prerequisites

To complete these tutorials, you must install the following products:

- MATLAB
- MATLAB Coder
- Embedded Coder
- C compiler

For a list of supported compilers, see
`http://www.mathworks.com/support/compilers/current_release/`.

You must set up the C compiler before generating C code. See "Setting Up the C/C++ Compiler" in the MATLAB Coder documetation.

For instructions on installing MathWorks products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, in the MATLAB Command Window, enter `ver` .

Setting Up Tutorial Files

The tutorial files are available in the following folder:
`matlabroot\toolbox\ecoder\examples`. To run the tutorials, copy these files to a local folder. Each tutorial provides instructions about which files to copy and how to copy them.

Controlling C Code Style

In this section...

"About This Tutorial" on page 2-4
"Copying Files Locally" on page 2-5
"Setting Up the MATLAB® Coder Project" on page 2-5
"Configuring Build Parameters" on page 2-6
"Generating the C Code" on page 2-7
"Viewing the Generated C Code" on page 2-7
"Key Points to Remember" on page 2-8
"Learn More" on page 2-8

About This Tutorial

Learning Objectives

This tutorial shows you how to:

- Generate code for `if-elseif-else` decision logic as `switch-case` statements.
- Automatically generate C code from your MATLAB code using MATLAB Coder.
- Configure code generation configuration parameters in the MATLAB Coder project.
- Generate a code generation report that you can use to debug your MATLAB code and verify that it is suitable for code generation.

Prerequisites

To complete this tutorial, install the required products and set up your C compiler as described in "Prerequisites" on page 2-3.

Required Files

Type	Name	Description
Function code	`test_code_style.m`	MATLAB example that uses `if-elseif-else`.

To run the tutorial, copy this file to a local folder. For instructions, see “Copying Files Locally” on page 2-5.

Copying Files Locally

Copy the tutorial files to a local working folder.

1. Create a local working folder, for example, `c:\ecoder\work`.
2. Change to the `matlabroot\toolbox\ecoder\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'ecoder', 'examples'))
   ```

3. Copy the file `test_code_style.m` to your local working folder.

   Your work folder now contains the file you need to complete this tutorial.

Setting Up the MATLAB Coder Project

1. Set your MATLAB current folder to the work folder that contains the file for this tutorial. At the MATLAB command line, enter:

   ```
   cd work
   ```

   where *`work`* is the full path of the work folder containing your files.

2. At the MATLAB command line, enter

   ```
   coder -new code_style.prj
   ```

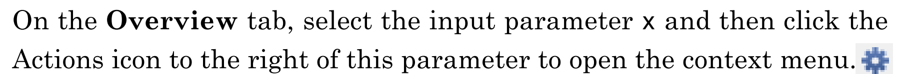
   By default, the project opens in the MATLAB workspace on the right side.

3 On the project **Overview** tab, click the `Add files` link and browse to the file `test_code_style.m` and then click **OK** to add the file to the project.

4 Define the type of input `x`.

Why Specify an Input Definition?

Because C and C++ are statically-typed languages, MATLAB Coder must determine the properties of all variables in the MATLAB files at code generation time. For more information, see "Primary Function Input Specification" in the MATLAB Coder documentation.

On the **Overview** tab, select the input parameter `x` and then click the Actions icon to the right of this parameter to open the context menu. 

5 From the menu, select `Define Type`.

6 In the **Define Type** dialog box, set **Class** to `int16`. Click **OK**.

Note The `Convert if-elseif-else patterns to switch-case statements` optimization works only for integer and enumerated type inputs.

Configuring Build Parameters

1 In the MATLAB Coder project, click the **Build** tab.

2 On the **Build** tab, set the **Output type** to `C/C++ Static Library`.

3 On the **Build** tab, click the `More settings` link to view the project settings.

4 In the **Project Settings** dialog box, click the **Code Style** tab.

5 On the **Code Style** tab, select **Convert if-elseif-else patterns to switch-case statements**.

6 On the **Report** tab, verify that **Always create a code generation report** is selected and then close the dialog box.

Generating the C Code

On the **Build** tab, click the **Build** button.

The Build progress dialog box opens. When the build is complete, MATLAB Coder generates a C library, `test_code_style.lib`, and C code in the `/codegen/lib/test_code_style` subfolder. Because you selected report generation, MATLAB Coder provides a link to the report on the **Results** tab.

Viewing the Generated C Code

MATLAB Coder generates C code in the file `test_code_style.c`.

To view the generated code:

1. On the **Build** tab **Results** pane, click the `View report` link to open the code generation report.

2. In the report, click the **C code** tab.

3. On this tab, click the `test_code_style.c` link.

   MATLAB Coder converts the `if-elseif-else` pattern to the following `switch-case` statements:

   ```
   switch (x) {
     case 1:
     y = 1.0;
     break;

     case 2:
     y = 2.0;
     break;

     case 3:
     y = 3.0;
     break;

     default:
     y = 4.0;
     break;
   ```

```
}
```

Key Points to Remember

- Use the `More settings` option on the MATLAB Coder project Build tab to open the Project Settings dialog box where you can configure code generation options.
- Use the `View Report` option on the MATLAB Coder project Build tab to open the code generation report.

Learn More

To...	See...
Learn how to create and set up a MATLAB Coder project	"Setting Up a MATLAB Coder Project" in the MATLAB Coder documentation.
Learn how to generate C/C++ code from MATLAB code at the command line	`codegen` in the MATLAB Coder documentation.

Generating Reentrant C Code from MATLAB Code

In this section...

"About This Tutorial" on page 2-9
"Copying Files Locally" on page 2-10
"About the Example" on page 2-11
"Providing a main Function" on page 2-12
"Configuring Build Parameters" on page 2-15
"Generating the C Code" on page 2-15
"Viewing the Generated C Code" on page 2-15
"Running the Code" on page 2-16
"Key Points to Remember" on page 2-17
"Learn More" on page 2-17

About This Tutorial

Learning Objectives

This tutorial shows you how to:

- Generate reentrant code from MATLAB code that uses no persistent or global data

Note This example runs on Windows only.

- Automatically generate C code from your MATLAB code.
- Define function input properties at the command line.
- Specify code generation properties.
- Generate a code generation report that you can use to debug your MATLAB code and verify that it is suitable for code generation.

Prerequisites

To complete this tutorial, install the required products and set up your C compiler as described in “Prerequisites” on page 2-3

Required Files

Type	Name	Description
Function code	`matrix_exp.m`	MATLAB Function that computes matrix exponential of the input matrix using Taylor series and returns the computed output.
C main function	`main.c`	Calls the reentrant code.

To run the tutorial, copy these files to a local folder. For instructions, see “Copying Files Locally” on page 2-10.

Copying Files Locally

Copy the tutorial files to a local working folder.

1 Create a local working folder, for example, `c:\ecoder\work`.

2 Change to the `matlabroot\toolbox\ecoder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'ecoder', 'examples'))
```

3 Copy the `reentrant_win` folder to your local working folder.

   Your work folder now contains the files you need to complete this tutorial.

4 Set your MATLAB current folder to the work folder that contains your files for this tutorial. At the MATLAB command line, enter:

```
cd work
```

   where *`work`* is the full path of the work folder containing your files.

About the Example

This example requires libraries that are specific to the Microsoft Windows operating system and, therefore, runs only on Windows platforms. It is a simple, multithreaded example that uses no persistent or global data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

Contents of matrix_exp.m

```
function Y = matrix_exp(X) %#codegen
    %
    % The function matrix_exp computes matrix exponential of
    % the input matrix using Taylor series and returns the
    % computed output.
    E = zeros(size(X));
    F = eye(size(X));
    k = 1;
    while norm(E+F-E,1) > 0
        E = E + F;
        F = X*F/k;
        k = k+1;
    end
    Y = E;
```

When you generate reusable, reentrant code, `codegen` supports dynamic allocation of function variables that are too large for the stack, as well as persistent and global variables. `codegen` generates a header file, *`primary_function_name`*`_types.h`, which you must include when using the generated code. This header file contains the following structures:

- *`primary_function_name`*`StackData`

  Contains the user allocated memory. You must pass a pointer to this structure as the first parameter to all functions that use it either directly, because the function uses a field in the structure, or indirectly, because the function passes the structure to a called function.

If the algorithm uses persistent or global data, the *primary_function_name*StackData structure also contains a pointer to the *primary_function_name*PersistentData structure. Including this pointer means that you have to pass only one parameter to each calling function.

- *primary_function_name*PersistentData

  If your algorithm uses persistent or global variables, codegen provides a separate structure for them and adds a pointer to this structure to the memory allocation structure. Having a separate structure for persistent and global variables allows you to allocate memory for these variables once and share them with all threads if desired. However, if there is no communication between threads, you can choose to allocate memory for these variables per thread or per application.

Providing a main Function

To call the reentrant code, you must provide a main function that:

- Includes the generated header file matrix_exp.h. This file includes the generated header file, matrix_exp_types.h.
- For each thread, allocates memory for stack data.
- Calls the matrix_exp_initialize housekeeping function. For more information, see "Calling Initialize and Terminate Functions" in the MATLAB Coder documentation.
- Calls matrix_exp.
- Calls matrix_exp_terminate.
- Frees the memory used for stack data.

Contents of main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    matrix_expStackData* spillData;
} IODATA;

/* The thread_function calls the matrix_exp function written in MATLAB */
DWORD WINAPI thread_function(PVOID dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize();
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out);
    matrix_exp_terminate();
    return 0;
}

void main() {
  HANDLE thread1, thread2;
  IODATA data1;
  IODATA data2;
  int32_T i;

  /*Initializing data for passing to the 2 threads*/
  matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
  matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

  data1.spillData = sd1;
  data2.spillData = sd2;
```

```
for (i=0;i<NUMELEMENTS;i++) {
    data1.in[i] = 1;
    data1.out[i] = 0;
    data2.in[i] = 1.1;
    data2.out[i] = 0;
}

/*Initializing the 2 threads and passing appropriate data to the thread functions*/
printf("Starting thread 1...\n");
thread1 = CreateThread(NULL , 0, thread_function, (PVOID) &data1, 0, NULL);
if (thread1 == NULL){
    perror( "Thread 1 creation failed.");
   exit(EXIT_FAILURE);
 }

printf("Starting thread 2...\n");
thread2 = CreateThread(NULL, 0, thread_function, (PVOID) &data2, 0, NULL);
if (thread2 == NULL){
    perror( "Thread 2 creation failed.");
    exit(EXIT_FAILURE);
}

/*Wait for both the threads to finish execution*/
if (WaitForSingleObject(thread1, INFINITE) != WAIT_OBJECT_0){
    perror( "Thread 1 join failed.");
  exit(EXIT_FAILURE);
 }

if (WaitForSingleObject(thread2, INFINITE) != WAIT_OBJECT_0){
  perror( "Thread 2 join failed.");
  exit(EXIT_FAILURE);
 }

free(sd1);
free(sd2);

printf("Finished Execution!\n");
exit(EXIT_SUCCESS);
}
```

Configuring Build Parameters

You enable generation of reentrant code using a code generation configuration object.

1 Create a configuration object.

```
e = coder.config('exe', 'ecoder', true);
```

   This command creates a `coder.EmbeddedCodeConfig` object which contains all the configuration parameters that the `codegen` function needs to generate standalone C/C++ static libraries and executables for an embedded target.

2 Enable reentrant code generation.

```
e.MultiInstanceCode = true;
```

Generating the C Code

Call the `codegen` function to generate C code, with the following options:

- `-config` to pass in the code generation configuration object `e`.
- `main.c` to include this file in the compilation.
- `-report` to create a code generation report.
- `-args` to specify an example input with the correct class, size, and complexity.

```
codegen -config e main.c -report matrix_exp.m -args ones(160,160)
```

`codegen` generates a C executable, `matrix_exp.exe`, in the current folder and C code in the `/codegen/exe/matrix_exp` subfolder. Because you selected report generation, `codegen` provides a link to the report.

Viewing the Generated C Code

`codegen` generates a header file `matrix_exp_types.h`, which defines the `matrix_expStackData` global structure. This structure contains local variables that are too large to fit on the stack.

To view this header file:

1 Click the `View report` link to open the code generation report.

2 In the report, click the **C code** tab.

3 On this tab, click the link to `matrix_exp_types.h`.

```
/*
 * matrix_exp_types.h
 *
 * MATLAB Coder code generation for function 'matrix_exp'
 */
#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Type Definitions */
typedef struct {
    struct {
        real_T F[25600];
        real_T Y[25600];
    } f0;
 } matrix_expStackData;

#endif
/* End of code generation (matrix_exp_types.h) */
```

Running the Code

Call the code, first verifying that the example is running on Windows platforms.

```
% This example can only be run on Windows platforms
if ~ispc
 error('This example requires Windows-specific libraries and can only be run on
 Windows.');
end
system('matrix_exp.exe')
```

The executable runs and reports successful completion.

Key Points to Remember

- Create a `main` function that
  - Includes the generated header file, *`primary_function_name`*`_types.h`. This file defines the *`primary_function_name`*`StackData` global structure. This structure contains local variables that are too large to fit on the stack.
  - For each thread, allocates memory for stack data.
  - Calls *`primary_function_name`*`_initialize` .
  - Calls *`primary_function_name`*.
  - Calls *`primary_function_name`*`_terminate`.
  - Frees the memory used for stack data.
- Use the `-config` option to pass the code generation configuration object to the `codegen` function.
- Use the `-args` option to specify input parameters at the command line.
- Use the `-report` option to create a code generation report.

Learn More

To...	See...
Learn more about the generated code API	"Generated Code API"
Call reentrant code with no persistent or global data on UNIX	"Example: Calling Reentrant Code with No Persistent or Global Data (UNIX Only)"
Call reentrant code with persistent data on Windows	"Example: Calling Reentrant Code — Multithreaded with Persistent Data (Windows Only)"
Call reentrant code with persistent data on UNIX	"Example: Calling Reentrant Code — Multithreaded with Persistent Data (UNIX Only)"

Tracing Between Generated C Code and MATLAB Code

In this section...

"About This Tutorial" on page 2-18
"Copying Files Locally" on page 2-19
"Configuring Build Parameters" on page 2-20
"Generating the C Code" on page 2-20
"Viewing the Generated C Code" on page 2-20
"Tracing Back to the Source MATLAB Code" on page 2-21
"Key Points to Remember" on page 2-21
"Learn More" on page 2-22

About This Tutorial

Learning Objectives

This tutorial shows you how to:

- Generate code that includes the MATLAB source code as comments.
- Include the function help text in the function header of the generated code.
- Use the code generation report to trace from the generated code to the source code.

Prerequisites

To complete this tutorial, install the required products and set up your C compiler as described in "Prerequisites" on page 2-3

Required File

Type	Name	Description
Function code	`polar2cartesian.m`	Simple MATLAB function that contains a comment

To run the tutorial, copy this file to a local folder. For instructions, see "Copying Files Locally" on page 2-19.

Copying Files Locally

Copy the tutorial file to a local working folder.

1. Create a local working folder, for example, `c:\ecoder\work`.

2. Change to the `matlabroot\toolbox\ecoder\examples` folder. At the MATLAB command line, enter:

   ```
   cd(fullfile(docroot, 'toolbox', 'ecoder', 'examples'))
   ```

3. Copy the `polar2cartesian.m` file to your local working folder.

   Your work folder now contains the file you need to complete this tutorial.

4. Set your MATLAB current folder to the work folder that contains the file for this tutorial. At the MATLAB command line, enter:

   ```
   cd work
   ```

   where *work* is the full path of the work folder containing your files.

Contents of polar2cartesian.m

```
function [x y] = polar2cartesian(r,theta)
%#codegen
% Convert polar to Cartesian
x = r * cos(theta);
y = r * sin(theta);
```

Configuring Build Parameters

1 Create a `coder.EmbeddedCodeConfig` code generation configuration object.

```
cfg = coder.config('lib', 'ecoder', true);
```

2 Enable the `MATLABSourceCode` option to include MATLAB source code as comments in the generated code and the function signature in the function banner.

```
cfg.MATLABSourceComments = true;
```

3 Enable the `MATLBFcnDesc` option to include the function help text in the function banner.

```
cfg.MATLABFcnDesc = true;
```

Generating the C Code

Call the `codegen` function to generate C code, with the following options:

- `-config` to pass in the code generation configuration object `cfg`.
- `-report` to create a code generation report.

```
codegen -config cfg  -report polar2cartesian
```

`codegen` generates a C library, `polar2cartesian.lib`, in the current folder and C code in the `/codegen/lib/polar2cartesian` subfolder. Because you selected report generation, `codegen` provides a link to the report.

Viewing the Generated C Code

`codegen` generates C code in the file `polar2cartesian.c`.

To view the generated code:

1 Click the `View report` link to open the code generation report.

2 In the report, click the **C code** tab.

3 On this tab, click the `straightline.c` link.

Examine the generated code. The function help text `Convert polar to Cartesian` appears in the function header. The source code appears as comments in the generated code.

```
/*
* function [x y] = polar2cartesian(r,theta)
*  Convert polar to Cartesian
*/
void straightline(real_T r, real_T theta, ...
    real_T *x, real_T *y)
{
   /*  polar2cartesian:4  x = r * cos(theta); */
  *x = r * cos(theta);
  /*  polar2cartesian:5  y = r * sin(theta); */
  *y = r * sin(theta);
}
```

Tracing Back to the Source MATLAB Code

To trace back to the source code, click any traceability tag.

For example, to view the MATLAB code for the C code, `x = r * cos(theta);`, click the `'polar2cartesian:4'` traceability tag.

The source code file `polar2cartesian.m` opens in the MATLAB editor with line 4 highlighted.

Key Points to Remember

- Create a `coder.EmbeddedCodeConfig` configuration object and enable the:
  - `MATLABSourceCode` option to include MATLAB source code as comments in the generated code and the function signature in the function banner
  - `MATLBFcnDesc` option to include the function help text in the function banner
- Use the `-config` option to pass the code generation configuration object to the `codegen` function.
- Use the `-report` option to create a code generation report.

Learn More

To...	See...
Learn more about code traceability	"About Code Traceability" in the MATLAB Coder documentation.
Learn about the location of comments in the generated code	"Location of Comments in Generated Code" in the MATLAB Coder documentation.
See traceability limitations	"Traceability Limitations" in the MATLAB Coder documentation.

3

Simulink Tutorials

- "About the Tutorials" on page 3-2
- "Tutorial – Configuring a Model and Generating Code for an Embedded System" on page 3-5
- "Tutorial – Controlling the Appearance of Generated Code " on page 3-15
- "Tutorial – Configuring the Data Interface" on page 3-21
- "Tutorial – Partitioning and Exporting Functions in the Generated Code" on page 3-29
- "Tutorial – Integrating Generated Code into an External Environment" on page 3-43
- "Tutorial – Verifying Generated Code" on page 3-53
- "Tutorial – Evaluating the Generated Code" on page 3-63

About the Tutorials

In this section...

"Introduction" on page 3-2

"Prerequisites" on page 3-3

"Third-Party Software" on page 3-3

"Required Files" on page 3-3

Introduction

The following tutorials are based on the throttle controller example model described in "Tutorial – Becoming Familiar with the Example Model and Testing Environment" in the Simulink Coder documentation. The tutorials will help you get started with using Embedded Coder to generate code from Simulink models and subsystems for embedded system applications.

- "Tutorial – Configuring a Model and Generating Code for an Embedded System" on page 3-5
- "Tutorial – Configuring the Data Interface" on page 3-21
- "Tutorial – Controlling the Appearance of Generated Code " on page 3-15
- "Tutorial – Partitioning and Exporting Functions in the Generated Code" on page 3-29
- "Tutorial – Integrating Generated Code into an External Environment" on page 3-43
- "Tutorial – Verifying Generated Code" on page 3-53
- "Tutorial – Evaluating the Generated Code" on page 3-63

Each tutorial focuses on a specific aspect of code generation or integration for embedded systems and is self-contained. Use only the tutorials that apply to your needs.

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 to the next tutorial, begin the next tutorial by opening a new model and loading new data.

Prerequisites

You must know how to:

- **MathWorks products**
  - Read, write, and apply MATLAB scripts
  - Create Simulink models
  - Include Stateflow charts in Simulink models
  - Run Simulink simulations and evaluate the results
- **C programming**
  - Use C data types and storage classes
  - Use function prototypes and call functions
  - Compile a C function
- **Metrics for evaluating embedded software**
  - Evaluate code readability
  - Evaluate RAM/ROM usage
  - Evaluate system execution performance

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, Appendix A, “Installing and Using an IDE for the Integration and Testing Tutorials” describes how to install and use the Eclipse™ IDE for C/C++ Developers and the Cygwin™ debugger for integrating and testing your generated code.

Required Files

Each tutorial uses a unique example model file and data set.

- Before you use each example file, place a copy in a in a writable location on your MATLAB path. Proceed through the tutorials from this location.
- As you proceed through a tutorial, save your changes for future examination.
- To avoid potentially introducing errors into the next tutorial, begin each tutorial by opening a new model and loading new data.

Tutorial – Configuring a Model and Generating Code for an Embedded System

In this section...

"About this Tutorial" on page 3-5

"Configuring a Model for Embedded System Code Generation" on page 3-6

"Checking the Model for Adverse Conditions and Settings for Embedded Systems Code" on page 3-11

"Generating Code for the Model" on page 3-12

"Reviewing the Generated Code" on page 3-13

"Key Points" on page 3-14

"Learn More" on page 3-14

About this Tutorial

Learning Objectives

- Configure a model for generating code for an embedded system.
- Apply model-checking tools to discover conditions and configuration settings that can result in generation of inaccurate or inefficient code.
- Generate code optimized for an embedded system from a model.
- Locate and identify generated code files.
- Review generated code.

Prerequisites

- Completed "Tutorial – Becoming Familiar with the Example Model and Testing Environment" and "Tutorial – Configuring the Model and Generating Code" in the Simulink Coder documentation

Required File

`rtwdemo_throttlecntrl_configert.mdl`

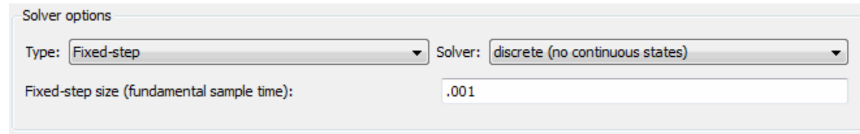
Configuring a Model for Embedded System Code Generation

Model configuration parameters determine the method for generating the code and the resulting format. When generating code for an embedded system, model configuration can be extremely complex. At a minimum, in addition to the solver type and system target file (STF), configure the model for the correct hardware implementation and optimizations that align with the system application.

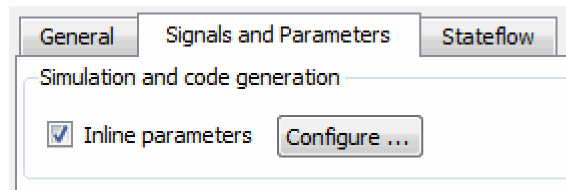
1 Open `rtwdemo_throttlecntrl_configert.mdl`. Save a copy as `throttlecntrl_configert.mdl` in a writable location on your MATLAB path.

2 In the Configuration Parameters dialog box, configure the solver. To generate code, configure the model to use a fixed-step solver. For this example, make sure that the **Type**, **Solver**, and **Fixed-step size** parameters are set as described in the following table.

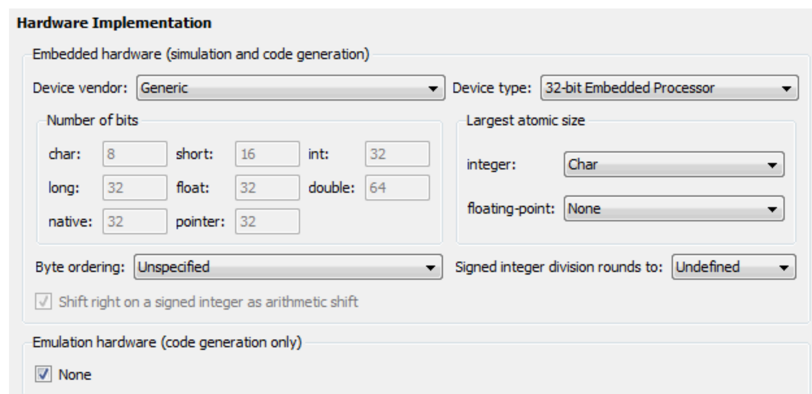
Parameter	Setting	Effect on Generated Code
Type	`Fixed-step`	Maintains a constant (fixed) step size, which is required for code generation
Solver	`discrete (no continuous states)`	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	`.001`	Sets the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



3 Open **Optimization > Signals and Parameters** and select **Inline parameters**. When this parameter is set, the code generator optimizes the code by replacing model parameters with constant values. Unless a model is still under development, consider using this setting when generating code for an embedded system.



4 Open the **Hardware Implementation** pane. Use parameters on this pane to specify the device type, word size, and byte ordering of the target hardware. Assume that the throttle controller model targets a generic 32-bit embedded processor. Set **Device type** to `32-bit Embedded Processor`.



5 Open the **Code Generation > General** pane. Set the **System target file** parameter to the embedded real-time target file, `ert.tlc`. The code

generator uses this target file to generate code that is optimized for embedded system deployment.

The list of Code Generation subpanes expands to include:

- **SIL and PIL Verification**
- **Code Style**
- **Templates**
- **Code Placement**
- **Data Type Replacement**
- **Memory Sections**

6 Set code generation parameters based on your application objectives. Configuring a model to meet specific objectives (requirements) for code generation can be an extremely complex, time consuming task. The Code Generation Advisor simplifies this task by allowing you to select and prioritize one or more of the following objectives:

- Execution efficiency
- ROM efficiency
- RAM efficiency
- MISRA-C:2004 guidelines
- Safety precaution
- Traceability
- 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 your model objectives.

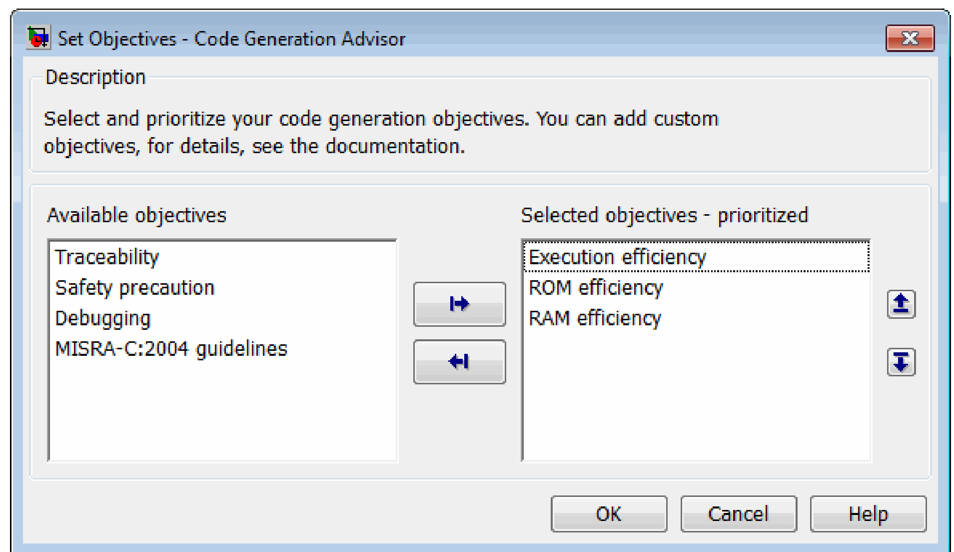
Some objectives recommend different parameter settings and include different checks in the Code Generation Advisor. When objectives conflict,

the priorities of the selected objectives determine which recommendations and checks the advisor presents.

For this tutorial, configure the model for execution and memory efficiency:

a On the **Code Generation** pane, click **Set objectives**. The Set Objectives dialog box opens.

b Select and prioritize categories `Execution efficiency`, `ROM efficiency`, and then `RAM efficiency` as shown below.

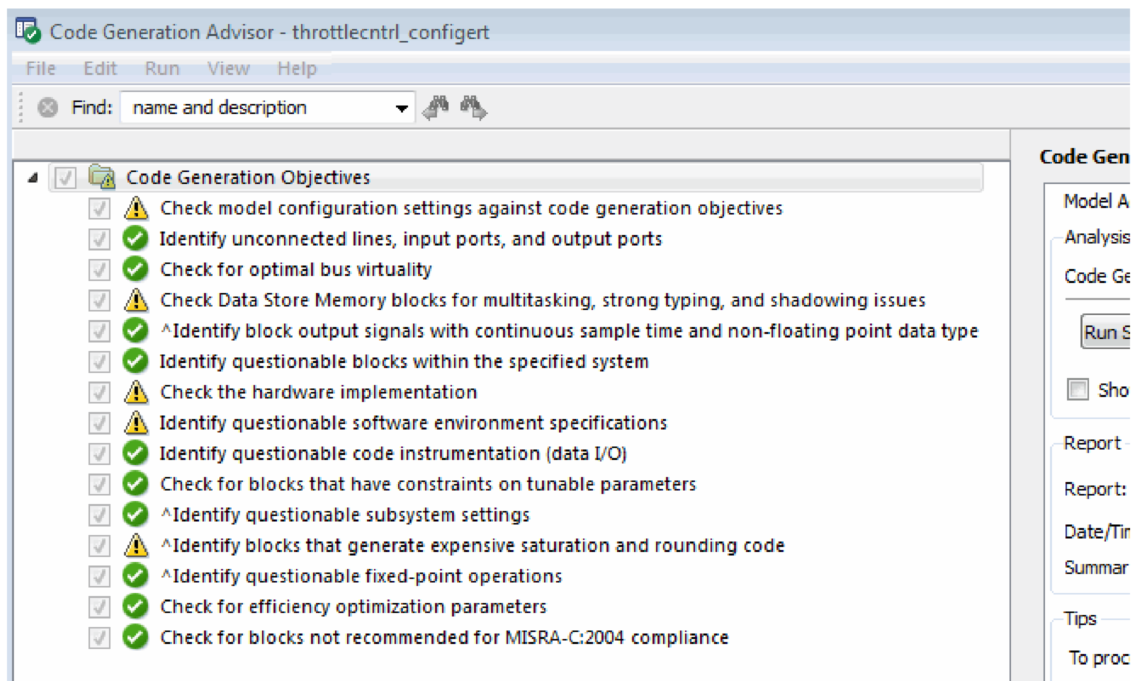


7 Review the model against the selected objectives by using the Code Generation Advisor.

a On the **Code Generation** pane, click **Check model**. A System Selector dialog box opens.

b In the System Selector dialog box, select `throttlecntrl_configert` and click **OK**.

The Code Generation Advisor dynamically creates and runs a list of checks based on the selection and prioritization of the objectives. This task might take a few minutes.



c Correct the warning condition flagged for the first check in the report, which reviews configuration parameter settings and recommends values based on the objectives.

Select the first check and click **Modify Parameters** to set parameters to recommended values. Then, click **Run This Check** to rerun the check. The warning symbol should disappear.

d Select hardware implementation check, which is the second check flagged with a warning. This check identifies inconsistencies and incomplete specification of hardware attributes, which can lead to inefficient or incorrect code for the target hardware.

Under Recommended Action, the report suggests that you specify byte ordering and signed integer division rounding for your target hardware. For this tutorial, assume you are using a target that supports both big-endian and little-endian byte ordering. Therefore, **Byte ordering** can remain `Unspecified`. However, you should click the **Signed integer division rounding** link and set the **Signed integer division**

rounds to parameter to `Zero` (the most common behavior), and rerun the check. Note that only the warning for byte ordering remains.

e. Select the third check that is flagged as a warning. The warning concerns arithmetic exceptions for integer division in generated code. Assume that you verified that your model cannot cause exceptions in division operations, and as the Code Generation Advisor suggests, select **Optimization > Remove code that protects against division arithmetic exceptions**. Update the model diagram and rerun the check. The warning should disappear.

f. Close the Code Generation Advisor window.

For more information, see "Determining Whether the Model is Configured for Specified Objectives".

8. Save the model and the configuration parameter settings as a MATLAB function.

   a. In the Model Explorer window, select `throttlcngrl_configert` in the left pane.

   b. Right-click and select `Export Active Configuration Set`. The Export Active Configuration Set to File dialog box opens.

   c. Save the file as `throttlecntrlModelErtConfig.m`.

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

9. Close the Model Explorer. Save and close the model.

Checking the Model for Adverse Conditions and Settings for Embedded Systems Code

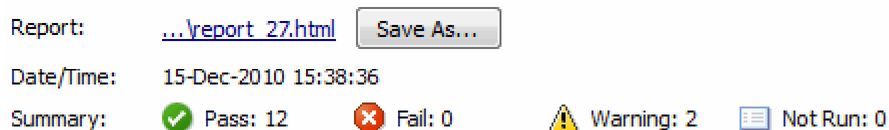
Before generating code for the model, use the Simulink Model Advisor to check the model for conditions and configuration settings that can result in inaccurate or inefficient code.

1. Open your copy of the throttle controller model, `throttlecntrl_configert.mdl`.

2. Start the Model Advisor.

3 In the System Hierarchy dialog box, click `throttlecntrl_configert`, and then click **OK**. It might take a few minutes for the Model Advisor window to open.

4 Expand **By Product** and **Embedded Coder**, enable all checks under **Embedded Coder**, click **Embedded Coder**, and in the right pane, click **Run Selected Checks**. The report summary indicates two warnings remain.



The remaining warnings concern byte ordering and MISRA-C:2004 compliance. Ignore the byte-ordering warning for the reason cited earlier.

5 Correct the MISRA conditions using the links in the analysis results, and then rerun that check.

6 Close the Model Advisor window.

Generating Code for the Model

1 Open `throttlecntrl_configert.mdl`.

2 In the Configuration Parameters dialog box, select **Code Generation > Generate code only** (if not already selected).

3 Click **Generate code** and watch the messages that are displayed in the MATLAB Command Window. The code generator produces standard C and header files and an HTML code generation report. The code generator places the files in a *build folder*, a subfolder named `throttlecntrl_configert_ert_rtw` under your current working folder. The subfolder name is the model name ending with the string `_ert_rtw`. The resulting code, while computationally efficient, is not yet organized for integration into a production environment.

Reviewing the Generated Code

1. From the generated HTML code generation report or the MATLAB Editor, open the generated `throttlecntrl_configert.c` file in the build folder and review the code. Note the following:

   - Identification, version, timestamp, and configuration comments.
   - Data definitions.
   - Controller code is in one function, `throttlecntrl_configert_step`.
   - Operations of multiple blocks are in one equation.
   - Generated data structures (for example, `throttlecntrl_configert_U.pos_rqst`) define all data.
   - `throttlecntrl_configert_initialize` and `throttlecntrl_configert_terminate` functions contain no code.

2. Close the throttle controller model.

Consider examining the following files by clicking the links in the HTML report, or by exploring other generated files in the build folder.

File	Description
`throttlecntrl_configert.c`	C file that contains step and initialization functions
`throttlecntrl_configert_data.c`	C file that assigns values to generated data structures
`ert_main.c`	Example main module that includes a simple scheduler
`throttlecntrl_configert.h`	Header file that defines data structures
`throttlecntrl_configert_private.h`	Header file that defines data used only by the generated code
`throttlecntrl_configert_types.h`	Header file that defines the model data structure

For more information about these and other generated files, see "Code Modules".

Key Points

- To generate code for an embedded system, you must at least change the model configuration to specify:
  - A fixed-step solver
  - Hardware settings that match the target hardware specifications
  - ERT system target format
  - Configuration objectives that align with application requirements
- Consider configuring the model with parameter inlining enabled if C code debugging is not required.
- You can check and modify the configuration of a model to best align with application objectives by using the Code Generation Advisor.
- You can save a model configuration for future use or for applying it to another model. Use the Export Active Configuration Set dialog box, available by right clicking on the model in the Model Explorer.
- Before generating code, consider checking a model with the Model Advisor.
- The code generator places generated files in a subfolder (`throttlecntrl_configert_ert_rtw`) of your working folder.

Learn More

- "Preparing Models for Code Generation"
- "Determining Whether the Model is Configured for Specified Objectives"
- "Mapping Application Objectives to Model Configuration Parameters"
- "Saving Configuration Sets" and "Loading Saved Configuration Sets" in the Simulink documentation.
- "Consulting the Model Advisor" in the Simulinkdocumentation.
- "Generating Code and Building Executables"

Tutorial – Controlling the Appearance of Generated Code

In this section...

"About this Tutorial" on page 3-15

"Customizing Code Comments" on page 3-16

"Customizing the Appearance of Identifiers" on page 3-18

"Customizing Code Style" on page 3-19

"Key Points" on page 3-20

"Learn More" on page 3-20

About this Tutorial

Learning Objectives

Control and customize the following aspects of generated code appearance to optimize the code, simplify maintenance, improve code readability, or comply with company code conventions and style:

- Comments
- Identifiers (symbols)
- Style (for example, conversion of `if-elseif-else` patterns to `switch-case` statements)

Prerequisites

You are able to:

- Open and modify Simulink models and subsystems
- Set block properties
- Set model configuration parameters
- Read C code

Required File

`rtwdemo_throttlecntrl_codeappearance.mdl`

Customizing Code Comments

1 Open `rtwdemo_throttlecntrl_codeappearance.mdl`. Save a copy as `throttlecntrl_codeappearance.mdl` in a writable location on your MATLAB path.

2 Generate code and examine the comments in the generated file `throttlecntrl_codeappearance.c`.

3 Add descriptions to the properties for the three model Inport blocks. For each block, right-click, select Block properties, and enter the descriptions listed in the following table.

For Inport Block...	Add Description...
`pos_rqst`	Throttle position request input for PI controller subsystems
`fbk_1`	Feedback input for PI controller subsystem `PI_ctrl_1`
`fbk_2`	Feedback input for PI controller subsystem `PI_ctrl_2`

4 Open the **Code Generation > Comments** pane of the Configuration Parameters dialog box and explore the available options.

5 Select the **Simulink block descriptions** parameter.

6 Generate code and examine the file `throttlecntrl_codeappearance.c`. Specifically, compare the following before and after code fragments:

Before

```
/* Sum: '<S2>/Sum2' incorporates:
 *  Inport: '<Root>/fbk_1'
 *  Inport: '<Root>/pos_rqst'
 */
```

```
  rtb_Sum3 = (*pos_rqst) - my_throttlecntrl_codeappearan_U.fbk_1;
.
.
.
/* Sum: '<S3>/Sum2' incorporates:
   *  Inport: '<Root>/fbk_2'
   *  Inport: '<Root>/pos_rqst'
   */
  rtb_Sum3 = (*pos_rqst) - fbk_2;
```

After

```
  /* Sum: '<S2>/Sum2' incorporates:
   *  Inport: '<Root>/fbk_1'
   *  Inport: '<Root>/pos_rqst'
   *
   * Block description for '<Root>/fbk_1':
   *  Feedback input for PI controller subsystem PI_ctrl_1
   *
   * Block description for '<Root>/pos_rqst':
   *  Throttle position request input for PI controller subsystems
   */
  rtb_Sum3 = (*pos_rqst) - my_throttlecntrl_codeappearan_U.fbk_1;
.
.
.
 /* Sum: '<S3>/Sum2' incorporates:
   *  Inport: '<Root>/fbk_2'
   *  Inport: '<Root>/pos_rqst'
   *
   * Block description for '/fbk_2':
   *  Feedback input for PI controller subsystem PI_ctrl_2
   *
   * Block description for '<Root>/pos_rqst':
   *  Throttle position request input for PI controller subsystems
   */
  rtb_Sum3 = (*pos_rqst) - fbk_2;
```

7 Close the code and model files.

For more information, see "Customizing Comments in Generated Code"

Customizing the Appearance of Identifiers

The ability to customize identifiers allows for easier integration of code generated by Embedded Coder with legacy code. It also simplifies compliance with company specific C coding standards.

1. Open `throttlecntrl_codeappearance.mdl`.

2. Generate code and examine the identifiers in the generated file `throttlecntrl_codeappearance.c`.

3. Open the **Code Generation > Symbols** pane of the Configuration Parameters dialog box and explore the available options. For token and macro details, see the Help for a specific parameter.

4. Change the setting of the **Local block output variables** parameter to `OutVar_$N$M`.

5. Generate code and examine the file `throttlecntrl_codeappearance.c`. Specifically, compare the following before and after code fragments:

   Before

```
rtb_Sum3 = throttlecntrl_configert_U.pos_rqst -
    throttlecntrl_configert_U.fbk_1;
.
.
.
rtb_Sum3 = throttlecntrl_configert_U.pos_rqst -
    throttlecntrl_configert_U.fbk_2;
```

   After

```
OutVar_Sum3 = throttlecntrl_codeappearance_U.pos_rqst -
    throttlecntrl_codeappearance_U.fbk_1;
.
.
.
OutVar_Sum3 = throttlecntrl_codeappearance_U.pos_rqst -
```

```
throttlecntrl_codeappearance_U.fbk_2;
```

6 Close the code and model files.

For more information, see “Configuring the Appearance of Generated Identifiers”

Customizing Code Style

1 Open `throttlecntrl_codeappearance.mdl`.

2 Generate code and examine the style applied to code in the generated file `throttlecntrl_codeappearance.c`.

3 Open the **Code Generation > Code Style** pane of the Configuration Parameters dialog box and explore the available options.

4 Change the setting of the **Parentheses level** parameter to `Minimum (Rely on C/C++ operators for precedence)`.

5 Generate code and examine the file `throttlecntrl_codeappearance.c`. Specifically, compare the following before and after code fragments:

Before

```
throttlecntrl_configert_DWork.Discrete_Time_Integrator1_DSTAT = (((-0.03 *
    rt_Lookup((const real_T *)throttlecntrl_configert_ConstP.pooled4, 9,
            rtb_Sum3, (const real_T *)throttlecntrl_configert_ConstP.pooled5))
    * rtb_Sum3) * 0.001) +
    throttlecntrl_configert_DWork.Discrete_Time_Integrator1_DSTAT;
```

After

```
throttlecntrl_codeappeara_DWork.Discrete_Time_Integrator1_DSTAT = -0.03 *
    rt_Lookup((const real_T *)throttlecntrl_codeappear_ConstP.pooled4, 9,
              OutVar_Sum3, (const real_T *)
              throttlecntrl_codeappear_ConstP.pooled5) * OutVar_Sum3 * 0.001 +
    throttlecntrl_codeappeara_DWork.Discrete_Time_Integrator1_DSTAT;
```

For more information, see “Controlling Code Style”

Key Points

Control and customize the following aspects of generated code appearance to optimize the code, simplify maintenance, improve code readability, or comply with company code conventions and style:

- Customize aspects of generated code appearance to meet application requirements.
- You can document generated code with the level of code comments appropriate for application objectives.
- You can customize identifiers in generated code, for example, to enhance readability or comply with company code guidelines or standards.
- Available code style customizations pertain to the parentheses level, operand order in expressions, condition expressions and patterns, and use of the `extern` keyword.

Learn More

- "Customizing Comments in Generated Code" and "Code Generation Pane: Comments"
- "Configuring the Appearance of Generated Identifiers" and "Code Generation Pane: Symbols"
- "Controlling Code Style" and "Code Generation Pane: Code Style"

Tutorial – Configuring the Data Interface

In this section...

"About this Tutorial" on page 3-21
"Creating Data Objects for Named Data in Base Workspace" on page 3-22
"Configuring Data Objects" on page 3-22
"Controlling Placement of Parameter and Constant Data in Generated Code" on page 3-23
"Including Signal Data Objects in Generated Code" on page 3-25
"Effects of Simulation on Data Typing" on page 3-26
"Viewing Data Objects in Generated Code" on page 3-27
"Saving Base Workspace Data" on page 3-28
"Key Points" on page 3-28
"Learn More" on page 3-28

About this Tutorial

Learning Objectives

- Configure the data interface for code generated from a model.
- Control the name, data type, and data storage class of signals and parameters in generated code.
- Control the placement of parameter data.

Prerequisites

- Understanding of ways to represent and use data and signals in models.
- Familiarity with representing data constructs as data objects.
- Completed "Tutorial – Configuring the Data Interface" in the Simulink Coder documentation.

- Able to read C code.

Required Files

- rtwdemo_throttlecntrl_dataplacement.mdl
- rtwdemo_throttlecntrl_testharnessert.mdl

Creating Data Objects for Named Data in Base Workspace

Use the Data Object Wizard to find constructs in the base workspace for which you can create data objects.

1. Open rtwdemo_throttlecntrl_dataplacement.mdl. Save a copy as throttlecntrl_dataplacement.mdl in a writable location on your MATLAB path.
2. In the model window, select **Tools** > **Data Object Wizard**. The Data Object Wizard dialog box opens.
3. In the Data Object Wizard dialog box, click **Find** to find candidate constructs. After a few minutes, constructs fbk_1 and pos_cmd_two appear in the dialog box.
4. Click **Check All** to select both constructs.
5. Click **Apply Package** to apply the default Simulink package for the data objects.
6. Click **Create** to create the data objects. Constructs fbk_1 and pos_cmd_two are removed from the dialog box.
7. Close the Data Object Wizard.
8. Open the Model Explorer, click **Base Workspace**, On the Contents pane, find the newly created objects fbk_1 and pos_cmd_two.

Configuring Data Objects

1. In the Model Explorer, examine the contents of the base workspace.

2 Configure the `fbk_1` and `pos_cmd_two` signals with the following settings.

Signal	Data Type	Storage Class
fbk_1	double	ImportedExtern
pos_cmd_two	double	ExportedGlobal

3 Close the Model Explorer.

4 Save and close the model.

Controlling Placement of Parameter and Constant Data in Generated Code

You can control which generated files contain model data definitions and declarations, for example, to adhere to company standards, by specifying files on the **Code Generation > Code Placement** pane of the Configuration Parameters dialog box.

For this tutorial, define and declare parameters in `eval_data.c` and `eval_data.h` Separating the data declaration and assignment simplifies the integration of the code into the production environment.

1 Open your copy of the throttle controller model, `throttlecntrl_dataplacement.mdl`.

2 In the Configuration Parameters dialog box, open **Code Generation > Code Placement**.

3 Set **Data definition** and **Data Declaration** to `Data defined in single separate source file`.

4 Set **Data definition filename** and **Data declaration filename** to `eval_data.c` and `eval_data.h`, respectively.

5 Generate code for the model. When code generation is complete, `eval_data.c` and `eval_data.h` are in the model build folder.

6 Examine the data files.

`eval_data.c`

```
#include "rtwtypes.h"
#include "throttlecntrl_dataplacement_types.h"

/* Const memory section */
/* Definition for custom storage class: Const */
const real_T I_Gain = -0.03;
const real_T I_InErrMap[9] = { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05, 0.25, 0.5,
  1.0 } ;

const real_T I_OutMap[9] = { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6, 0.75, 1.0 } ;

const real_T P_Gain = 0.74;
const real_T P_InErrMap[7] = { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25, 1.0 } ;

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

eval_data.h

```
#ifndef RTW_HEADER_eval_data_h_
#define RTW_HEADER_eval_data_h_
#include "rtwtypes.h"
#include "throttlecntrl_dataplacement_types.h"

/* Const memory section */
/* Declaration for custom storage class: Const */
extern const real_T I_Gain;
extern const real_T I_InErrMap[9];
extern const real_T I_OutMap[9];
extern const real_T P_Gain;
extern const real_T P_InErrMap[7];
extern const real_T P_OutMap[7];

#endif                                /* RTW_HEADER_eval_data_h_ */
```

7 Save and close the code files and model.

For more information, see "Managing Placement of Data Definitions and Declarations".

Including Signal Data Objects in Generated Code

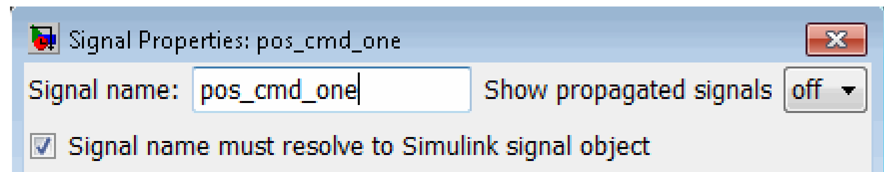
1 Open your copy of the throttle controller model, `throttlecntrl_dataplacement.mdl`.

2 In the Configuration Parameters dialog box, make sure that you select **Optimization > Signals and Parameters > Inline parameters**.

3 Enable signal data object `pos_cmd_one` to appear in the generated code.

   a In the model window, right-click the `pos_cmd_one` signal line and select **Signal Properties**. A Signal Properties dialog box opens.

   b In the Signal Properties dialog box, make sure that you select **Signal name must resolve to a Simulink signal object**.



4 Enable signal object resolution for all signals in the model simultaneously. In the MATLAB Command Window, enter:

```
disableimplicitsignalresolution('throttlecntrl_dataplacement')
```

Messaging in the MATLAB Command Window indicates that the following signal objects are resolved.

Signal...	Used By...
`pos_cmd_two`	`PI_ctrl_2/1`
`error_reset`	`Define_Throt_Param/Constant4/1`
`max_diff`	`Define_Throt_Param/Constant3/1`
`fail_safe_pos`	`Define_Throt_Param/Constant/1`
`fbk_1`	`fbk_1/1`

5 Save and close the model.

Effects of Simulation on Data Typing

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

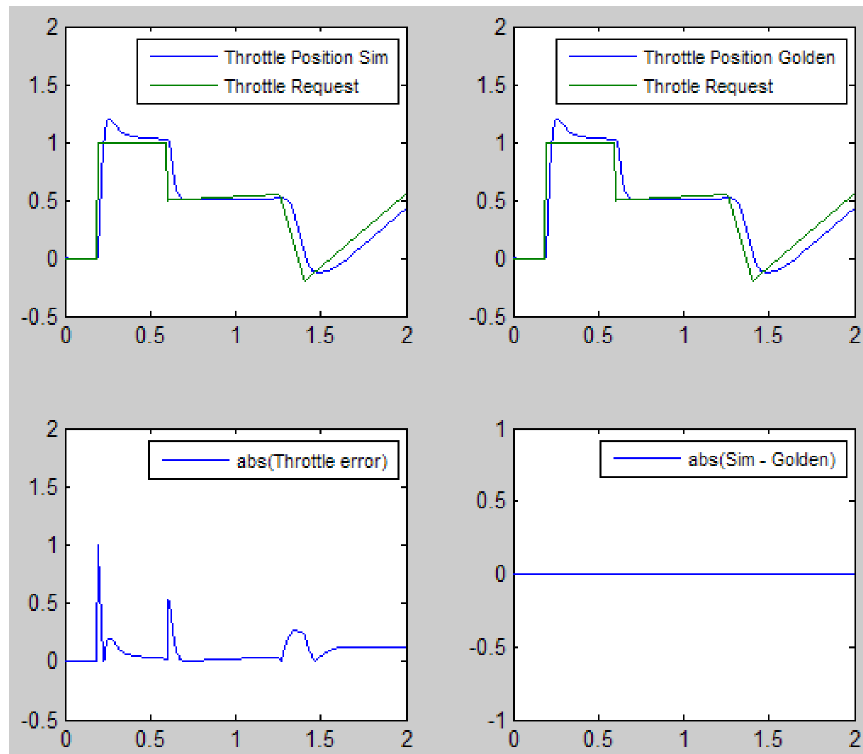
Before you run your test harness, update it to include the `throttlecntrl_dataplacement` model.

Note The following procedure requires a Stateflow license.

---

1. Open `throttlecntrl_dataplacement.mdl`.
2. Open `rtwdemo_throttlecntrl_testharnessert.mdl`. Save a copy as `throttlecntrl_testharnessert.mdl`.
3. In the test harness model, right-click the `Unit_Under_Test` Model block and select **Model Reference Parameters**.
4. Set **Model name** to `throttlecntrl_dataplacement` and click **OK**.
5. Update the test harness model diagram (**Edit > Update Diagram**).
6. Simulate the test harness.

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



7 Save and close the test harness model.

Viewing Data Objects in Generated Code

1 Open your copy of the throttle controller model, `throttlecntrl_dataplacement`.

2 Generate code for the model.

3 Examine the code in the generated file, `throttlecntrl_dataplacement.c`.

The following statement shows a sampling of generated variables for the model before you converted the data to data objects.

```
rtb_Sum3 = my_throttlecntrl_dataplacement_U.pos_rqst...
- my_throttlecntrl_dataplacement_U.fbk_1;
```

After creating data objects for signals `pos_rqst` and `fbk_1`, the same line of generated code appears is:

```
rtb_Sum3 = *pos_rqst - fbk_1;
```

4 Close the model.

Saving Base Workspace Data

In the base workspace, save the data that exists in the base workspace for future reference. In the MATLAB Command Window, enter `save`. Simulink places the data in the `matlab.mat` file in the model build folder.

Key Points

- You can declare data in Simulink models and Stateflow charts by using data objects.
- You can manage (create, view, configure, and so on) base workspace data from the Model Explorer or in the MATLAB Command Window.
- The Data Object Wizard provides a quick way to create data objects for constructs such as signals, buses, and parameters.
- You must explicitly configure data objects to appear by name in generated code.
- Because Simulink software uses the `double` data type for simulation, if all data types are set to `double` for a model, you can expect simulation and generated code behavior to match.
- Separation of the data from the model provides several benefits.

Learn More

- "Working with Data" in the Simulink documentation
- "Defining Data Representation and Storage for Code Generation"
- "Creating and Using Custom Storage Classes" in the Embedded Coder documentation
- "Managing Placement of Data Definitions and Declarations" in the Embedded Coder documentation

Tutorial – Partitioning and Exporting Functions in the Generated Code

In this section...

"About this Tutorial" on page 3-29

"Changing Model Architecture to Control Execution Order" on page 3-30

"Controlling Function Location and File Placement in Generated Code" on page 3-32

"Using a Mask to Pass Parameters into a Library Subsystem" on page 3-35

"Generating Code for the Full Model and Exported Functions" on page 3-37

"Changing the Execution Order and Simulation Results" on page 3-39

"Key Points" on page 3-41

"Learn More" on page 3-42

About this Tutorial

Learning Objectives

- Specify function and file names in generated code.
- Exert direct control over the execution order of model components.
- Identify parts of generated code necessary for integration.
- Generate code for atomic subsystems.
- Know what data you need to execute a generated function.

Prerequisites

- Understand basic model architecture.
- Understand the difference between types of subsystems — see "Systems and Subsystems" in the Simulink documentation.
- Understand the purpose of function-call subsystems.

- Understand what reentrant code is.
- Familiarity with the Subsystem Parameters dialog box.
- Familiarity with the Mask Parameters dialog box.
- Familiarity with different ways to generate code for subsystems.
- Able to read C code.

Required File

- `rtwdemo_throttlecntrl_funcpartition.mdl`
- `rtwdemo_throttlecntrl_testharnessert.mdl`

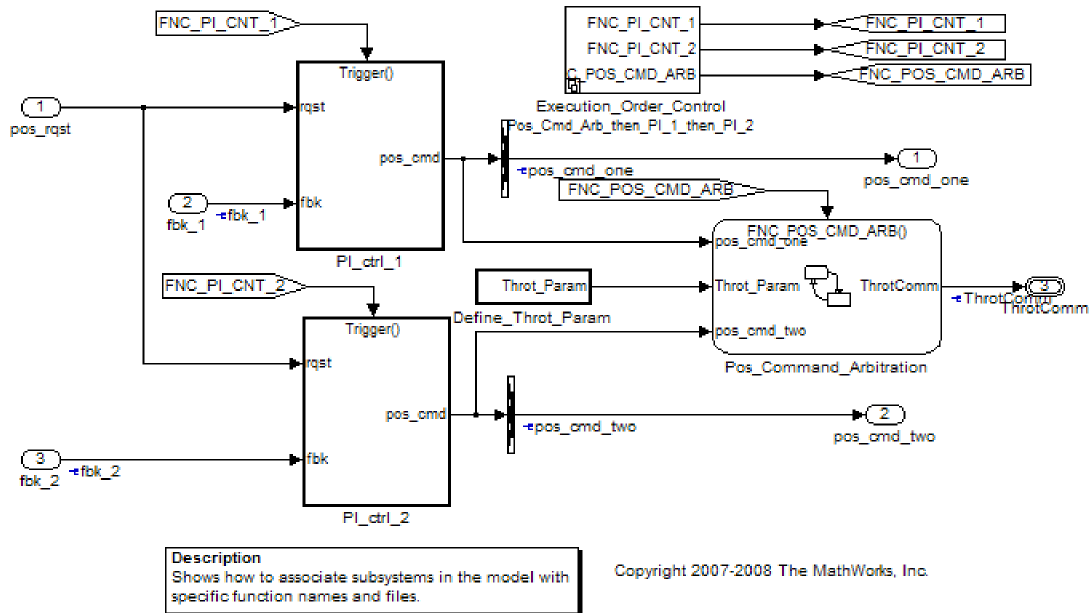
Changing Model Architecture to Control Execution Order

To match the behavior of a physical system, you might have to exert direct control over the execution order of model components. One way of controlling execution order is to use function-call subsystems and a Stateflow chart that models the calling functionality of a scheduler.

The example throttle controller model includes virtual subsystems. This tutorial shows you how to replace the virtual subsystems with function-call subsystems, and use them to control subsystem execution order.

1. Open `rtwdemo_throttlecntrl_funcpartition.mdl`. Save a copy as `throttlecntrl_funcpartition.mdl` in a writable location on your MATLAB path.

   This version of the throttle controller model includes three function-call subsystems and a subsystem consisting of a Stateflow chart, which controls execution order of the other subsystems.



2 Examine the function-call subsystems `PI_ctrl_1`, `PI_ctrl_2`, and `Pos_Command_Arbitration`.

3 Examine the Stateflow chart subsystem `Execution_Order_Control`. This subsystem controls the execution order of the function-call subsystems. Later in the tutorial, you see how changing execution order can change simulation results.

4 Examine the new Signal Conversion blocks for output ports `pos_cmd_one` and `pos_cmd_two` of the PI controllers. The `Contiguous copy` setting for the **Output** block parameter makes it possible for the PI controller functions in the generated code to be reentrant.

5 Close the model.

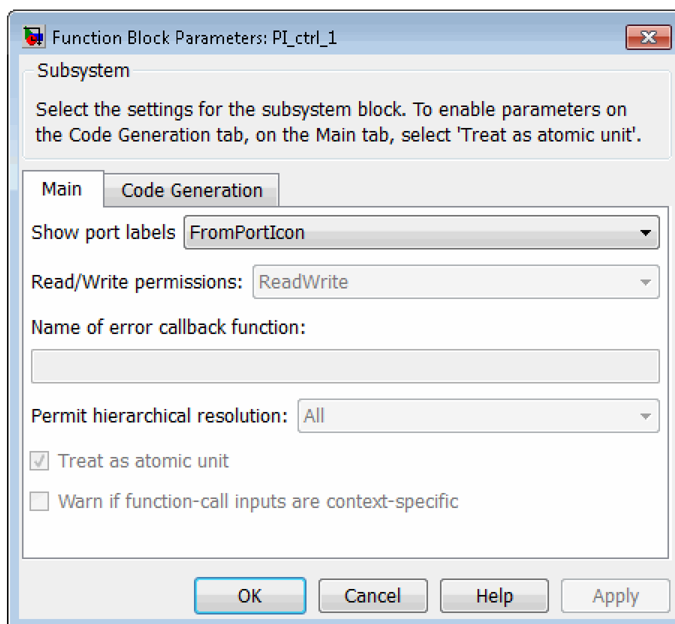
Controlling Function Location and File Placement in Generated Code

So far, the code generator has produced a *model*`_step` function, which contains all control algorithm code for the throttle controller model. However, many applications require a greater level of control over the location of functions in the generated code. By using atomic subsystems, you can instruct the code generator to partition algorithm code across multiple functions. You specify the partitioning by modifying parameters in the Subsystem Parameters dialog box for each subsystem.

1. Open your copy of the throttle controller model, `throttlecntrl_funcpartition.mdl`.

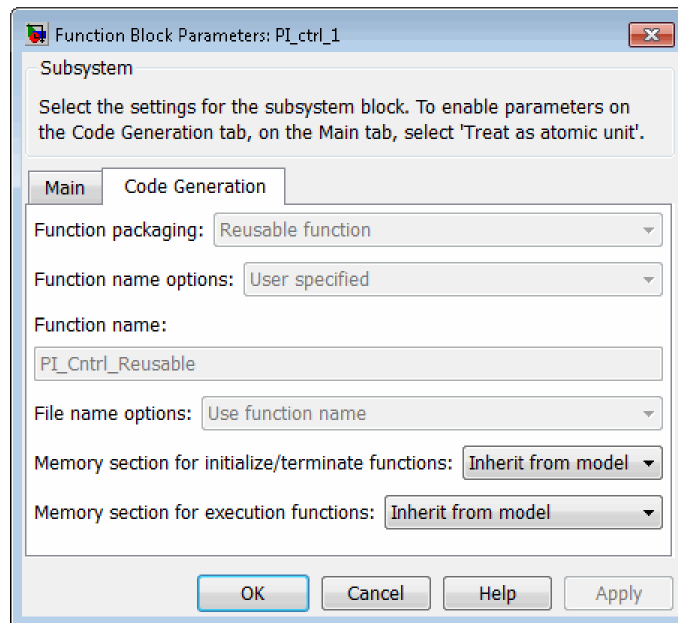
2. Open the Subsystem Parameters dialog box for subsystems `PI_ctrl_1` and `PI_ctrl_2`. Examine the parameter settings.

   a. Right-click the subsystem and select **Subsystem Parameters**. The Function Block Parameters dialog box opens.



b On the **Main** tab, **Treat as atomic unit** is selected. This parameter is automatically selected and unavailable for atomic subsystems. When set, this parameter causes Simulink software to treat block methods associated with a subsystem as a unit when determining execution order. The parameter provides a way to group functional aspects of a model at the execution level. This parameter enables parameters on the **Code Generation** tab.

c Click the **Code Generation** tab.



d Note the following parameter settings:

Parameter	Rationale for Setting
Function packaging	`Reusable function` causes the code generator to produce reentrant code for the subsystem. Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Operating systems and other system software that use 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 state variables and pass them into the function. Any number of users or processes can share one copy of a reentrant routine
Function name options	`User Specified` enables you to specify a unique name for the generated function.
Function name	Enables you to name the function that the code generator produces for a subsystem. In this case, the code generator names the reusable function `PI_Cntrl_Reusable`.
File name options	`Use function name` causes the code generator to place the generated function in a separate file and name it with the same name as the generated function. In this case, the code generator places the function in the model build folder in the file the `PI_Cntrl_Reusable.c`.

3 Open the Subsystem Parameters dialog box for subsystem `Pos_Command_Arbitration`and examine the parameter settings. **Treat as atomic unit** on the **Main** tab is already selected. The following table provides the rationale for the parameter settings on the **Code Generation** tab.

Parameter	Rationale for Setting
Function packaging	`Function` causes the code generator to produce a separate function that is not reentrant and has no arguments.
Function name options	`Auto` assigns the generated function a unique name using the default convention *`model_subsystem`*`()`. *`model`* is the name of the model and *`subsystem`* is the name of the subsystem. In this case, the function name is `throttlecntrl_funcpartition_Pos_Command_-Arbitration`.
File name options	`Auto` causes the code generator to place the function code in the`throttlecntrl_funcpartition.c` file.

4 Open the Subsystem Parameters dialog box for subsystem `Execution_Order_Control`and examine the parameter settings. For this subsystem, the **Treat as atomic unit** parameter is not set, restricting **Function packaging** to `Function` only.

5 Close the model.

For more information, see `Subsystem`.

Using a Mask to Pass Parameters into a Library Subsystem

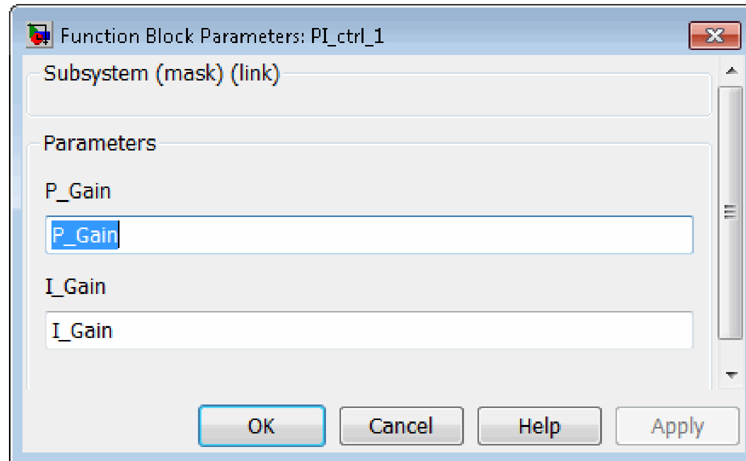
Subsystem *masks* enable Simulink software to define subsystem parameters outside the scope of a library block. By changing the parameter value at the top of the library, you can reuse the same library with multiple sets of parameters within the same model.

When a subsystem is reusable and has a mask, the generated code passes the masked parameters into the reentrant code as arguments. Code generation software fully supports the use of data objects in masks. The data objects are used in the generated code.

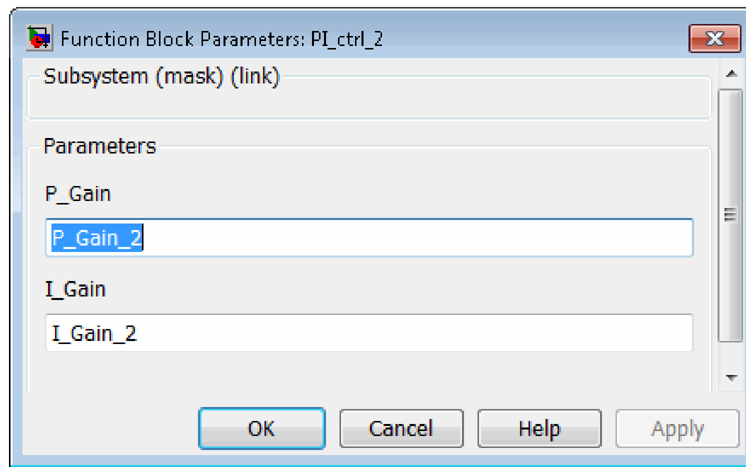
Examine the masks for subsystems `PI_ctrl_1` and `PI_ctrl_2`.

1 Open your copy of the throttle controller model,`throttlecntrl_funcpartition.mdl`.

2 Open the Mask Parameters dialog box for subsystem `PI_ctrl_1` by double-clicking the subsystem. The dialog box opens, showing data objects for gain parameters `P_gain` and `I_gain`.



3 Open the Mask Parameters dialog box for subsystem `PI_ctrl_2`. For this subsystem, Simulink creates two new data objects, `P_Gain_2` and `I_Gain_2`.



4 Close the Mask Parameters dialog boxes and the model.

Generating Code for the Full Model and Exported Functions

The code generator can build code at the system (full model) and subsystem levels. Compare the files generated for the full model build with files generated for exported functions.

1. Open your copy of the throttle controller model, `throttlecntrl_funcpartition.mdl`.

2. Generate code for the model.

3. Export a function for the `PI_ctrl_1` subsystem.

   a. In the model window, right-click `PI_ctrl_1` and select **Code Generation** > **Export Functions**. The **Build code for Subsystem** dialog box opens.

   b. In the **Build code for Subsystem** dialog box, click **Build**. The code generator produces a complete set of code files for the subsystem and places them in the build folder `PI_ctrl_1_ert_rtw`.

4. If you have a Stateflow license, export a function for the `Pos_Command_Arbitration` subsystem. The code generator produces a complete set of code files for the subsystem and places them in the build folder `Pos_Command_Arbitration_ert_rtw`.

5. Examine the generated code listed in the following table by locating and opening the files in the respective build folders.

File	Full Build	`PI_ctrl_1`	`Pos_Command_-Arbitration` (requires Stateflow license)
`throttlecntrl_funcpartition.`	Yes Step function	No	No
`PI_ctrl_1.c`	No	Yes Trigger function	No

File	Full Build	PI_ctrl_1	Pos_Command_-Arbitration (requires Stateflow license)
Pos_Command_Arbitration.c (requires Stateflow license)	No	No	Yes Initialization 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.

6 Close all dialog boxes and the model.

Examining Masked Data in Generated Code

1 Open throttlecntrl_funcpartition.c.

2 Search for PI_Cntrl_Reusable. The function call shows how the code generator passes data objects (P_Gain and I_Gain) from the subsystem masks into the reentrant code.

```
PI_Cntrl_Reusable(pos_rqst, fbk_1, &throttlecntrl_funcpartiti_DWork_DWork->PI_ctrl_1,
I_Gain, P_Gain);
```

3 Search for PI_Cntrl_Reusable again. The second function call passes data objects from the mask for subsystem PI_ctrl_2.

```
PI_Cntrl_Reusable(pos_rqst, fbk_2, &throttlecntrl_funcpartiti_DWork->PI_ctrl_2,
I_Gain_2, P_Gain_2);
```

4 Close the C code file.

Changing the Execution Order and Simulation Results

Without explicit control, subsystems in model `throttlecntrl_funcpartition` execute in the following order:

1 `PI_ctrl_1`

2 `PI_ctrl_2`

3 `Pos_Cmd_Arbitration`

You can use the test harness to see the effect of the execution order on the simulation results. The `Execution_Order_Control` subsystem is set up so that you can switch between two configurations, which change the execution order of the other subsystems in the model.

Note The following procedure requires a Stateflow license.

Change the execution order.

1 Open your copy of the throttle controller model, `throttlecntrl_funcpartition.mdl`.

2 Right-click the `Execution_Order_Control` subsystem. Select **Block Choice > PI_1_then_PI_2_then_Pos_Cmd_Arb** to set the subsystem execution order to `PI_ctrl_1`, `PI_ctrl_2`, then `Pos_cmd_Arbitration`.

3 Save the model.

4 Open the test harness model, `throttlecntrl_testharnessert.mdl`.

5 Set up the test harness to use model `throttlecntrl_funcpartition` as the unit under test.

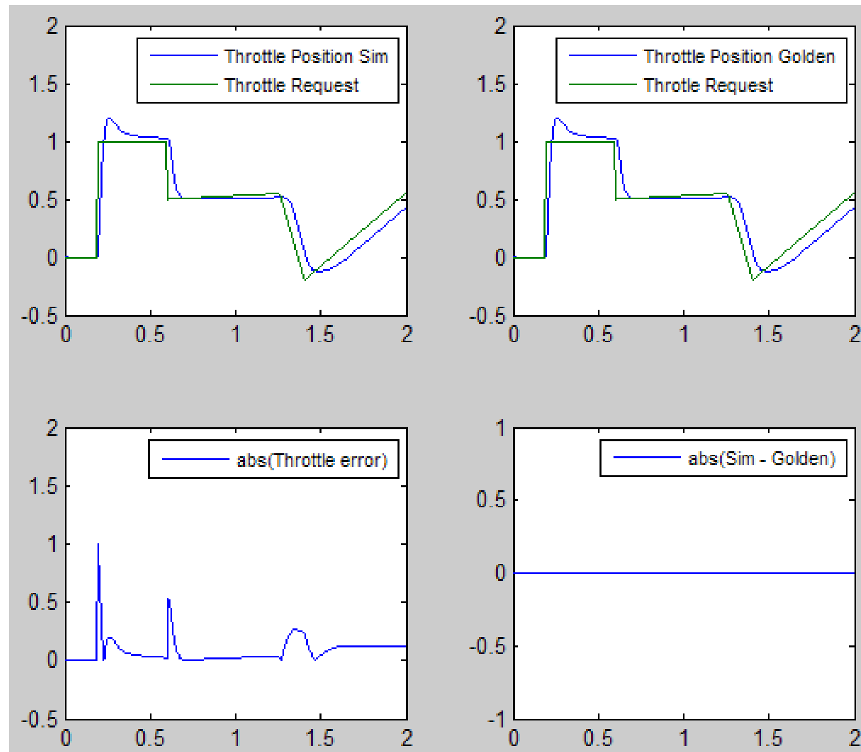
   a Right-click the `Unit_Under_Test` Model block and select **Model Reference Parameters**.

   b Set **Model name** to `throttlecntrl_funcpartition` and click **OK**.

c Update the model diagram.

6 Run the test harness.

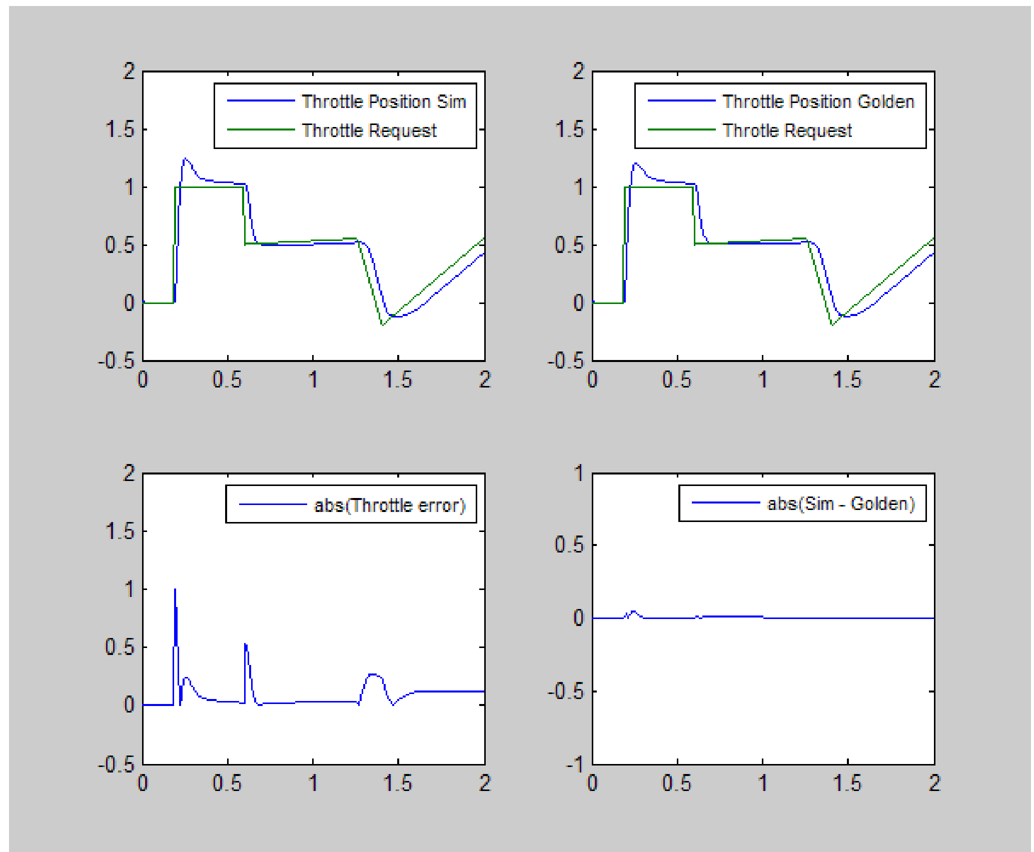
The resulting plot shows that the difference between the golden and simulated versions of the model remains zero.



7 In the model `throttlecntrl_funcpartition`, change the execution order to `Pos_cmd_Arbitration_then_PI_1_then_PI_2`, update the model diagram, and save the model.

8 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.



9 Close the throttle controller and test harness models.

Key Points

- One way of controlling execution order of subsystems during simulation is to use function-call subsystems and a Stateflow chart that models the calling functionality of a scheduler.
- For atomic subsystems, you can instruct the code generator to partition algorithm code across multiple functions.

- When partitioning code across multiple functions, you can specify the name of the function for each subsystem and the name of the separate file for the code with parameters in the Subsystem Parameters dialog box.
- When a subsystem is reusable and has a mask, the generated code passes the masked parameters into the reentrant code as arguments.
- The code generator can build code at the system (full model) and subsystem levels.
- At the subsystem level, the code generator produces a full set of generated files in a separate build folder.
- You can change the execution of a model by using a subsystem containing a Stateflow chart that models the calling functionality of a scheduler and changing the setting of the **Block Choice** option on the subsystem's context menu.

Learn More

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

Tutorial – Integrating Generated Code into an External Environment

In this section...

"About this Tutorial" on page 3-43

"Relocating Code to Another Development Environment" on page 3-44

"Integrating Generated Code into an Existing System" on page 3-45

"Setting Up the Main Function" on page 3-45

"Matching the System Interfaces" on page 3-47

"Building a Project in the Eclipse Environment" on page 3-50

"Key Points" on page 3-51

"Learn More" on page 3-51

About this Tutorial

Learning Objectives

- Collect and relocate files to build an executable outside of the Simulink environment.
- Set up generated code to interface with external variables and functions.
- Build a full system that includes your generated code.

Prerequisites

- Access to installed versions of the Eclipse Integrated Development Environment (IDE) and the Cygwin Debugger. However, required integration tasks demonstrated in the tutorial are common to all integration environments. For information on how to install the Eclipse IDE and Cygwin Debugger, see Appendix A, "Installing and Using an IDE for the Integration and Testing Tutorials"
- Able to read C code.

- Familiarity with debugging tools and capabilities.

Required Files

- `rtwdemo_throttlecntrl_externenv.mdl`
- Files in *`matlabroot`*`/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/externenv_files`, where *`matlabroot`* is your MATLAB installation folder

Relocating Code to Another Development Environment

Generated code depends on static support files that MathWorks provides. If you need to relocate the static and generated code files for a model to another development environment, such as a dedicated build system, use the Simulink Coder pack-and-go utility (`packNGo`). This utility finds and packages all files to build an executable image, including external files you define in the **Code Generation** > **Custom Code** pane of the Configuration Parameters dialog box, and packages the files in a standard zip file.

1. Open `rtwdemo_throttlecntrl_externenv.mdl`. Save a copy to `throttlecntrl_externenv.mdl` in a writable location on your MATLAB path. Proceed through the tutorial from this location.

2. Generate code for the model.

   After code generation, the model is configured to run `packNGo`.

3. In your working folder, find and examine the contents of the generated file `throttlecntrl_externenv.zip`.

   The number of files in the zip file depends on the version of Embedded Coder software that you are running and the configuration of the model. The compiler does not require all 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.

To generate the zip file manually, in the MATLAB Command Window:

1. Load the `buildInfo.mat` file, located in the build folder for the model.

2 Enter the command `packNGo(buildInfo)`.

For more information about using the `packNGo` utility, see "Relocating Code to Another Development Environment" in the Simulink documentation.

Integrating Generated Code into an Existing System

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, code is not generated for any of these components. Instead, you develop interfaces that connect the components. MathWorks provides hardware interface block libraries for many common embedded controllers. For details, see the block libraries under Embedded Targets.

Setting Up the Main Function

For this tutorial, you modify a supplied `main` function to build a full system. The `main` function performs basic actions to exercise the code for a simple system. It is *not* an example of an actual application `main` function.

1 Copy files for the example `main` function to your working folder. The files are in *`matlabroot`*`/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/externenv_files`. *`matlabroot`* represents the name of your MATLAB installation folder.

2 Open and examine the code in the file `example_main.c`.

Note For the `example_main.c` file, name your copy of the model file `rtdemo_throttlecntrl_externenv.mdl` as`throttlecntrl_externenv.mdl`.

```
#include <stdio.h>              /* This ert_main.c example uses printf/fflush */
#include "throttlecntrl_externenv.h"             /* Model's header file */
#include "rtwtypes.h"                   /* MathWorks types */
#include "throttlecntrl_externenv_private.h"     /* Local data for PCG Eval */
#include "defineImportedData.h"         /* The inputs to the system */


/* Observable signals */
static BlockIO_throttlecntrl_externenv throttlecntrl_externenv_B;
/* Observable states */
static D_Work_throttlecntrl_externenv throttlecntrl_externenv_DWork;

real_T pos_cmd_one;                     /* '<Root>/Signal Conversion1' */
real_T pos_cmd_two;                     /* '<Root>/Signal Conversion2' */
ThrottleCommands ThrotComm;             /* '<Root>/Pos_Command_Arbitration' */
ThrottleParams Throt_Param;             /* '<S1>/Bus Creator' */


int simulationLoop = 0;

int_T main(void)
{
  /* Initialize model */
  rt_Pos_Command_Arbitration_Init(); /* Set up the data structures for chart*/
  throttle_cnt_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,&throttlecntrl_externenv_B.PI_ctrl_1,
                      &throttlecntrl_externenv_DWork.PI_ctrl_1);
    PI_Cntrl_Reusable((*pos_rqst),fbk_2,&throttlecntrl_externenv_B.PI_ctrl_2,
                      &throttlecntrl_externenv_DWork.PI_ctrl_2);
    pos_cmd_one = throttlecntrl_externenv_B.PI_ctrl_1.Saturation1;
    pos_cmd_two = throttlecntrl_externenv_B.PI_ctrl_2.Saturation1;
```

```
    throttle_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                     &throttlecntrl_externenv_B.sf_Pos_Command_Arbitration);



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

Identify areas of the code that perform each of the following functions:

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

3 Close `example_main.c`.

The order of execution of functions in `example_main.c` matches the order in which the test harness model and `throttlecntrl_externenv.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 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 with `#include` statements and calls the functions from the generated code.

1 Specify input data.

The system has three input signals: `pos_rqst`, `fbk_1`, and `fbk_2`. The two feedback signals are imported externs (`ImportedExtern`) and the position signal is an imported extern pointer (`ImportedExternPointer`). Because of

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

a Open your copy of the file `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;
}
```

This file contains code for a simple C stub that defines the signal variables. The generated code has access to the data from the `extern` definitions in your generated `throttlecntrl_externenv_Private.h` file. In a real system, the data comes from other software components or from hardware devices.

b Close `defineImportedData.c`.

c In your build folder, open `throttlecntrl_externenv_Private.h`.

d Find and examine the following `extern` definitions.

```
/* 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' */
```

e Close `throttlecntrl_externenv_Private.h`.

2 Specify output data.

You do not have to do anything with the output data. However, you can access the data in your generated `throttlecntrl_externenv.h` file.

a In your build folder, open `throttlecntrl_externenv.h`

b Examine the contents of the file.

c Close `throttlecntrl_externenv.h`.

"Tutorial – Verifying Generated Code" on page 3-53 shows how to save the output data to a log file.

3 Identify additional data.

The code generator creates several data elements that you do not need to access to complete this tutorial. Such data elements include:

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

For this tutorial, the `throttlecntrl_externenv.h` file declares this data.

a In your build folder, open `throttlecntrl_externenv.h`

b Search the file for the data listed in the following table. The table lists the most common data structures. Depending on the configuration of the model, some or all of these structures are in the generated code.

Data Type	Data Name	Data Purpose
Constants	`throttlecntrl_externenv_cP`	Constant parameters
Constants	`throttlecntrl_externenv_cB`	Constant block I/O
Output	`throttlecntrl_externenv_U`	Root and atomic subsystem input
Output	`throttlecntrl_externenv_Y`	Root and atomic subsystem output
Internal data	`throttlecntrl_externenv_B`	Value of block output
Internal data	`throttlecntrl_externenv_D`	State information vectors
Internal data	`throttlecntrl_externenv_M`	Time and other system level data
Internal data	`throttlecntrl_externenv_Zero`	Zero-crossings
Parameters	`throttlecntrl_externenv_P`	Parameters

c Close `throttlecntrl_externenv.h`.

4 Match the function-call interface.

By default, the code generator creates functions that have a `void Func(void)` interface. If you configure the model or an atomic subsystem as reentrant code, the code generator creates a more complex function prototype.

a Open your copy of `example_main.c`.

b The `example_main` function is configured to call the functions with the correct input arguments.

```
throttlecntrl_externenv_B.sf_Pos_Command_Arbitration);
                   ((*pos_rqst),fbk_1,&throttlecntrl_externenv_B.PI_ctrl_1,
                   &throttlecntrl_externenv_DWork.PI_ctrl_1);
PI_Cntrl_Reusable((*pos_rqst),fbk_2,&throttlecntrl_externenv_B.PI_ctrl_2,
                   &throttlecntrl_externenv_DWork.PI_ctrl_2);
pos_cmd_one = throttlecntrl_externenv_B.PI_ctrl_1.Saturation1;
pos_cmd_two = throttlecntrl_externenv_B.PI_ctrl_2.Saturation1;

throttlePos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                   &throttlecntrl_externenv_B.sf_Pos_Command_Arbitration);
```

Calls to the `PI_Cntrl_Reusable` function use a mixture of user-defined variables and default data structures. The build process defines data structures in `throttlecntrl_externenv.h`. The preceding code fragment also shows how the data structures map to 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 folder on your C drive. Name the folder such that the path contains no spaces (for example, `EclipseProjects/throttlecntrl/externenv`). For this tutorial and "Tutorial – Verifying Generated Code" on page 3-53, you use the Cygwin Debugger, which requires that your build folder be on your C drive and the folder path not include spaces.

Note If you have not generated code for the model, or the zip file does not exist, complete the steps in “Relocating Code to Another Development Environment” on page 3-44 before continuing to the next step.

---

2 Unzip the file `throttlecntrl_externenv.zip` into the build folder you just created.

3 Delete the files `ert_main.c` and `throttlecntrl_externenv.c`. Then, add `example_main.c` that you examined in “Setting Up the Main Function” on page 3-45.

4 Use the Eclipse Integrated Development Environment (IDE) and Cygwin Debugger to step through and evaluate the execution behavior of the generated C code. For instructions on installing the IDE, creating a new project, and configuring the debugger, see Appendix A, “Installing and Using an IDE for the Integration and Testing Tutorials”.

5 Close `throttlecntrl_externenv.mdl`.

Key Points

- You can find and package all files you need to build an executable image in an alternative build environment by using the Simulink Coder pack-and-go utility.
- A `main` function performs actions necessary to exercise the code for a system.
- Integration of system components requires matching data and function interfaces of generated code and existing system code.
- Depending on the system, you might need to consider input data, output data, and other data generated in the generated *`model`*`.h` file.

Learn More

- “Relocating Code to Another Development Environment” in the Simulink Coder documentation

- "Setting Up Generated Code To Interface With Components in the Run-Time Environment"
- "Embedded IDEs and Embedded TargetsDesktop IDEs and Desktop Targets"

Tutorial – Verifying Generated Code

In this section...

"About this Tutorial" on page 3-53
"Methods for Verifying Generated Code" on page 3-54
"Reusing Test Data By Importing and Exporting Test Vectors" on page 3-55
"Verifying Behavior of a Model with Software-in-the-Loop Testing" on page 3-56
"Verifying System Behavior By Importing and Exporting Test Vectors" on page 3-59
"Testing via Processor-in-the-Loop (PIL)" on page 3-62
"Key Points" on page 3-62
"Learn More" on page 3-62

About this Tutorial

Learning Objectives

- Identify methods available for testing generated code.
- Test generated code in the Simulink environment.
- Test generated code outside of the Simulink environment.

Prerequisites

- Access to installed versions of the Eclipse Integrated Development Environment (IDE) and the Cygwin Debugger. However, required integration tasks demonstrated in the tutorial are common to all integration environments. For information on how to install the Eclipse IDE and Cygwin Debugger, see Appendix A, "Installing and Using an IDE for the Integration and Testing Tutorials"
- Able to read C code.
- Familiarity with debugging tools and capabilities.

Required Files

- `rtwdemo_throttlecntrl_testcode.mdl`
- `rtwdemo_throttlecntrl_testharnessSIL.mdl`
- Files in *`matlabroot`*`/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/testcode_files`, where *`matlabroot`* is your MATLAB installation folder

Methods for Verifying Generated Code

Simulink software supports multiple methods for verifying the behavior of code generated for a system.

Test Method	What the Method Does	Advantages	Disadvantages
Microsoft Windows run-time executable	Generates a Windows executable and runs 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) testing	Uses an S-function wrapper to include the generated code in a 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

Test Method	What the Method Does	Advantages	Disadvantages
Processor-in-the-loop (PIL) testing	Downloads code to a target processor and communicates with it from Simulink; see "How SIL and PIL Simulations Work"	Allows you to reuse the Simulink test environment Can use C debugger with the simulation Uses actual processor	Requires additional steps to set up test environment
On-target rapid prototyping	Runs 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 By Importing and Exporting Test Vectors

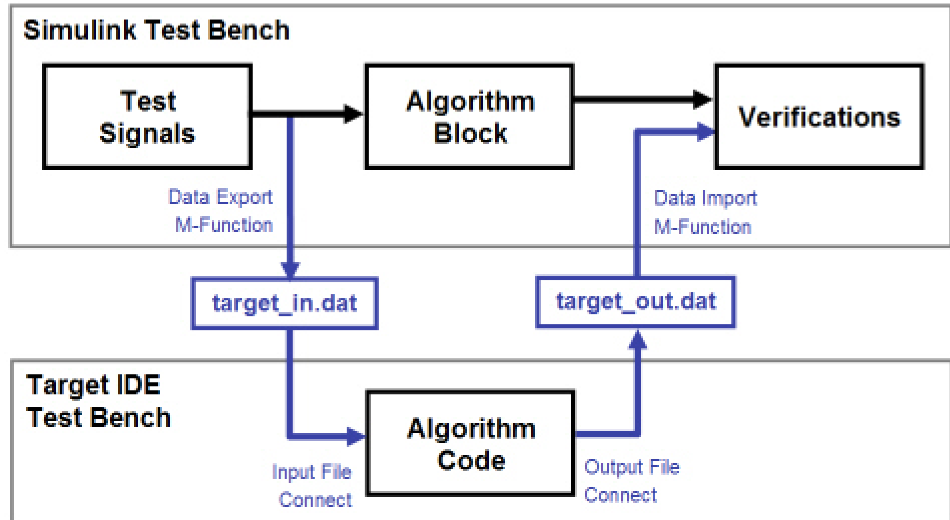
When a unit under test is in the Simulink environment, you can reuse test data inside and outside of the Simulink environment.

1. Save the Simulink data into a file.

2. Format the data in a way that is accessible to the system code.

3. Read the data file as part of the system code procedures.

You can also reuse data from an external environment, such as an interactive development environment (IDE) for a specific target, in the Simulink test environment. To do so, you must save the data in a format that MATLAB software can read.



Verifying Behavior of a Model with Software-in-the-Loop Testing

You can automatically generate code from a Model block, wrap the code in an S-Function, and bring the S-Function into another model for software-in-the-loop testing.

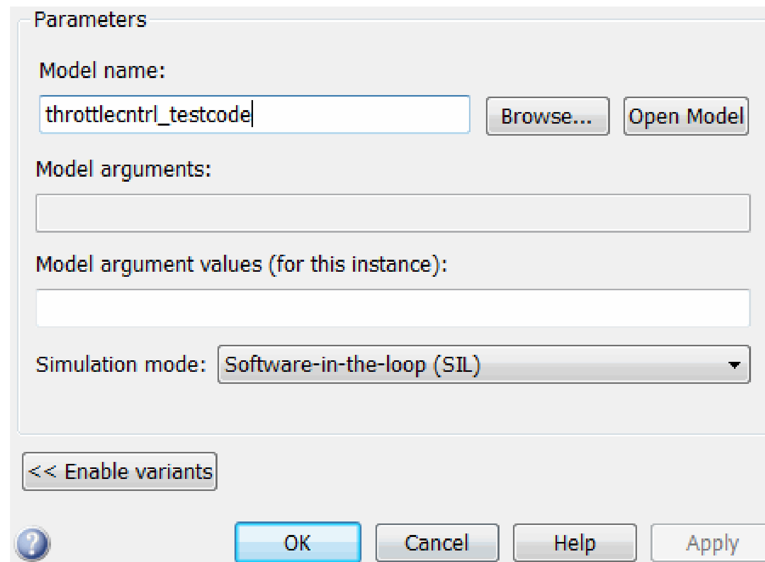
1. Configure the model you want to test.
   a. Open `rtwdemo_throttlecntrl_testcode.mdl`. Save a copy to `throttlecntrl_testcode.mdl` in a writable location on your MATLAB path. Proceed through the tutorial from this location.
   b. Open the Configuration Parameters dialog box. Set **Hardware Implementation > Device vendor** to `Generic` and **Hardware Implementation > Device type** to `32-bit x86 compatible`.
   c. Click **OK** to apply the changes. Close the dialog box.
2. Make sure that you can build an executable for `throttlecntrl_testcode`.
3. Configure the test harness model.

a Open `rtwdemo_throttlecntrl_testharnessSIL.mdl`. Save a copy to `throttlecntrl_testharnessSIL.mdl` in a location on your MATLAB path. The test harness uses a Model block to access the model to verify with software-in-the-loop (SIL) testing.

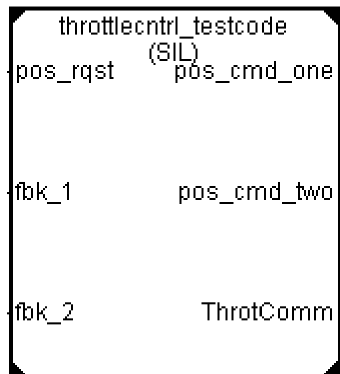
b Right-click the `throttlecntrl_testcode` Model block and select **Model Reference Parameters**.

c Set **Model name** to `throttlecntrl_testcode`.

d Set **Simulation mode** to `Software-in-the-loop (SIL)` and click **OK**.

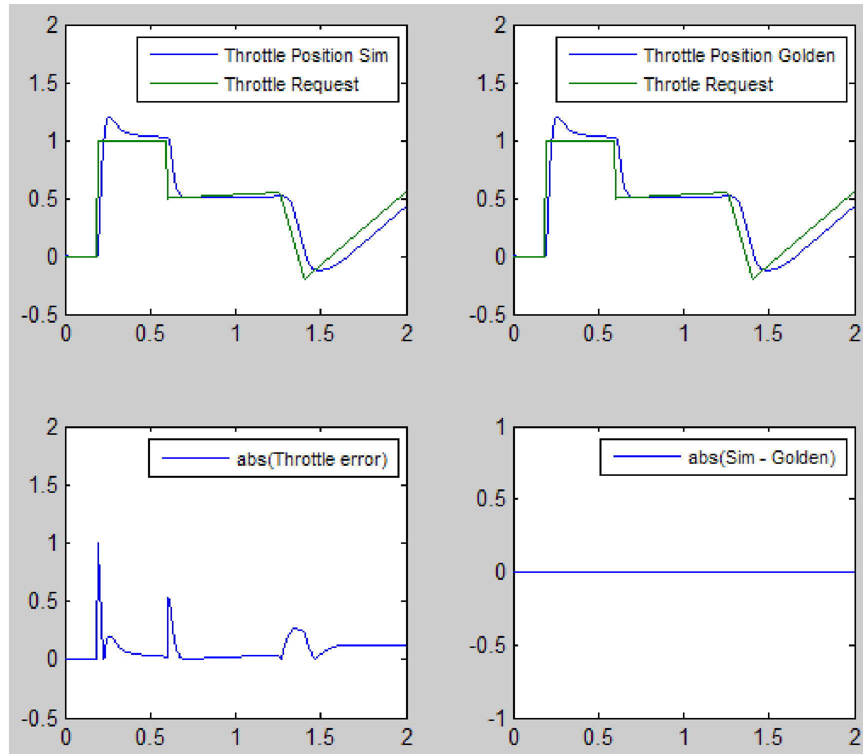


After you configure the Model block for SIL verification, the block includes a (SIL) tag, as the following figure shows:



e Update the test harness model diagram (**Edit > Update Diagram**).

4 Run the test harness. As the following plot shows, the results from running the generated code are the same as the simulation results.



Verifying System Behavior By Importing and Exporting Test Vectors

About the example_main.c

This example extends the example in "Tutorial – Integrating Generated Code into an External Environment" on page 3-43. In this case, `example_main.c` simulates hardware I/O.

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

This version of `example_main.c` has the following order of execution:

- Initialize data (one time)

```
while < endTime
```

- Read simulated hardware input
- `PI_cnrl_1`
- `PI_ctrl_2`
- `Pos_Command_Arbitration`
- Write simulated hardware output

```
end while
```

The `example_main.c` uses two functions, `plant` and `hardwareInputs`.

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

A handwritten function, `WriteDataForEval.c`, logs data. When the test is complete, the function executes and writes test data to the file, `ThrottleCntrl_ExternSimData.m`. You can load this file into the MATLAB environment and compare the data to simulation data.

Importing and Exporting Test Vectors Using the Eclipse Environment

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

1. Open your copy of the throttle controller model, `throttlecntrl_testcode.mdl`.

2. Add the additional files required to build an executable. In the Configuration Parameter dialog box, add the following paths for **Code Generation > Custom Code > Include list of additional > Include directories**:

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

3. Make sure the model configuration parameter **Code Generation > SIL and PIL Verification > Create block** is set to `None`.

4. Build the model.

5. Create a build folder on your C drive. Name the folder such that the path contains no spaces (for example, `EclipseProjects/throttlecntrl/testcode`). For this tutorial, you use the Cygwin Debugger, which requires that your build folder be on your C drive and that the folder path not include spaces.

6. Unzip the `throttlecntrl_testcode.zip` file, in your working folder, into the build folder that you just created.

7. Delete the `ert_main.c` and `throttlecntrl_testcode.c` files. Then, add `example_main.c` that you examined in "Setting Up the Main Function" on page 3-45.

8. Run the control code in Eclipse to generate the `eclipseData.m` file. This is the file that `writeDataForEval.c` generates.

   If necessary, consult "Integrating and Testing Code with the Eclipse IDE" on page A-5 or Eclipse IDE help.

9. Plot the results that you get from the Eclipse environment. Compare the data from the Eclipse run with the data results from the test harness.

10. Close `throttlecntrl_testcode`.

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

For information, instructions, and demos, see "Verifying Generated Code With SIL and PIL Simulations" in the Embedded Coder.

Key Points

- Methods for verifying code generated for an embedded system include running a MicrosoftWindows run-time executable, SIL testing, PIL testing, and on-target rapid prototyping.
- You can reuse test data by importing and exporting test vectors.

Learn More

- "Verifying Generated Code With SIL and PIL Simulations"

Tutorial – Evaluating the Generated Code

In this section...

"About this Tutorial" on page 3-63
"Evaluating Code" on page 3-63
"About the Compiler" on page 3-64
"Viewing the Code Metrics" on page 3-64
"About the Build Configurations" on page 3-64
"Configuration 1: Reusable Functions, Data Type Double" on page 3-65
"Configuration 2: Reusable Functions, Data Type Single" on page 3-66
"Configuration 3: Nonreusable Functions, Data Type Single" on page 3-67

About this Tutorial

Learning Objectives

- Get familiar with the build characteristics of the code you generated for the throttle controller model
- Explore how different model configurations affect the model's RAM/ROM metric.

Prerequisites

Required Files

Evaluating Code

Efficiency of generated code is based on two primary metrics: *execution speed* and *memory usage*. Often, faster execution requires more memory. Memory usage in ROM (read-only memory) and RAM (random access memory) presents comprimises:

- Accessing data from RAM is faster than accessing data from ROM.

- Systems store executables and data using ROM because RAM does not maintain data between power cycles.

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

About the Compiler

The Freescale™ CodeWarrior® is used in this evaluation.

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

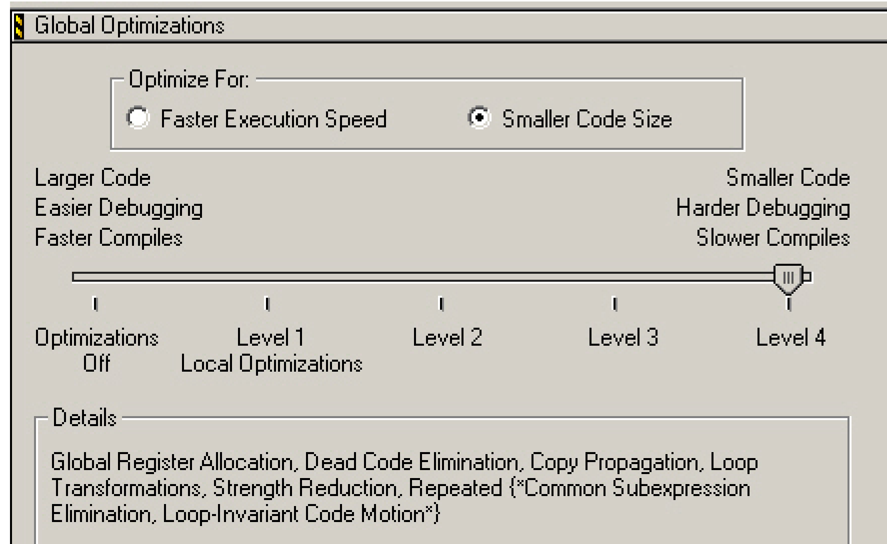
Viewing the Code Metrics

As described in "Tutorial – Integrating Generated Code into an External Environment" on page 3-43 and "Tutorial – Verifying Generated Code" on page 3-53, the generated code might require utility functions. The utility functions have a fixed overhead. Their memory requirements are a one-time cost. Because of this, the data in this module shows the following memory usage.

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

About the Build Configurations

The same model configuration parameter settings are used in 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 “Tutorial – Integrating Generated Code into an External Environment” on page 3-43
- **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 are 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 "Tutorial – Integrating Generated Code into an External Environment" on page 3-43
- **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 configuration 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 "Tutorial – Integrating Generated Code into an External Environment" on page 3-43
- **Function-Call Method:** The function interface is `void void`. Global parameters pass the data.

The memory requirements for the third configuration are higher than the memory requirements for the second configuration. If the data type is doubled, the memory requirements are 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

- "Installing the Eclipse IDE and Cygwin Debugger" on page A-2
- "Integrating and Testing Code with the Eclipse IDE" on page A-5

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

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

Note To use the Eclipse IDE with embedded IDEs and target processors, see "Working with Eclipse IDE" under "Embedded IDEs and Embedded TargetsDesktop IDEs and Desktop Targets".

---

1. From the Eclipse Downloads web page (http://www.eclipse.org/downloads/), download the Eclipse IDE for C/C++ Developers to your C: drive.

   You also need the Eclipse C/C++ Development Tools (CDT) that are compatible with the Eclipse IDE. You can install the CDT as part of the Eclipse C/C++ IDE packaged zip file or you can install it into an existing version of the Eclipse IDE.

If You Install the CDT...	Then...
As part of the Eclipse C/C++ IDE packaged zip file	Go to step 4
Into an existing version of the Eclipse IDE	Go to step 2

2. From the Eclipse CDT Downloads page (http://www.eclipse.org/cdt/downloads.php), download the Eclipse C/C++

Development Tools (CDT) that is compatible with your installed version of the Eclipse IDE.

3. Unzip the downloaded Eclipse CDT zip file. Copy the contents of the directories `features` and `plugins` to the corresponding directories in `c:\eclipse`.

4. Create the folder `c:\eclipse`.

5. Unzip the downloaded Eclipse IDE zip file into `c:\eclipse`.

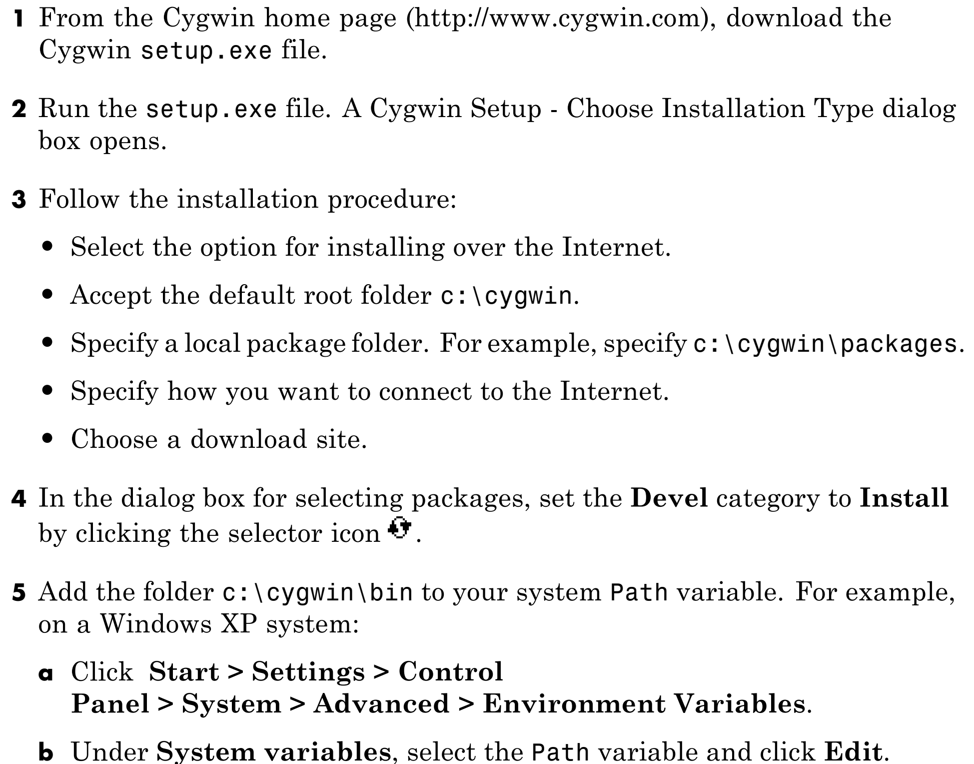
6. On your desktop, create a link to the executable file `c:\eclipse\eclipse.exe`.

Installing the Cygwin Debugger

1. From the Cygwin home page (http://www.cygwin.com), download the Cygwin `setup.exe` file.

2. Run the `setup.exe` file. A Cygwin Setup - Choose Installation Type dialog box opens.

3. Follow the installation procedure:
   - Select the option for installing over the Internet.
   - Accept the default root folder `c:\cygwin`.
   - Specify a local package folder. For example, specify `c:\cygwin\packages`.
   - Specify how you want to connect to the Internet.
   - Choose a download site.

4. In the dialog box for selecting packages, set the **Devel** category to **Install** by clicking the selector icon .

5. Add the folder `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 folder must be on your C drive. The folder path cannot include any spaces.

Integrating and Testing Code with the Eclipse IDE

In this section...

"Introducing Eclipse" on page A-5

"Defining a New C Project" on page A-6

"Configuring the Debugger" on page A-7

"Starting the Debugger" on page A-7

"Setting the Cygwin Path" on page A-8

"Actions and Commands in the Eclipse Debugger" 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 generated code, as described in "Tutorial – Integrating Generated Code into an External Environment" on page 3-43 and "Tutorial – Verifying Generated Code" on page 3-53. There are many other software packages and tools that can work with code generation software to perform similar tasks.

"Installing the Eclipse IDE and Cygwin Debugger" on page A-2 contains instructions for installing Eclipse and Cygwin. Before proceeding, be sure you have installed Eclipse and Cygwin, as described in that section.

Note To use Eclipse IDE with embedded IDEs and target processors, see "Working with Eclipse IDE" under "Embedded IDEs and Embedded TargetsDesktop IDEs and Desktop Targets".

To use Cygwin, your build folder must be on your C drive. The folder path cannot include any spaces.

Project Names and File Names

"Tutorial – Integrating Generated Code into an External Environment" on page 3-43 and "Tutorial – Verifying Generated Code" on page 3-53 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:

- `externenv`, if you are working in "Tutorial – Integrating Generated Code into an External Environment" on page 3-43
- `testcode`, if you are working in "Tutorial – Verifying Generated Code" on page 3-53

Defining a New C Project

1. In Eclipse, choose **File > New > C Project**. A C Project dialog box opens.
2. In the C Project dialog box:
   a. In the **Project name** field, type the project name `throttlecntrl_##` (## is `externenv` or `testcode`) .
   b. In the **Location** field, specify the location of your build folder (for example, `C:\EclipseProjects\throttlecntrl\externenv`).
   c. In the **Project type** selection box, select and expand **Makefile project**.
   d. Click the **Empty Project** node.
   e. Under **Other Toolchains**, select `Cygwin GCC` .
   f. Click **Next**. A Select Configurations dialog box opens.
3. In the Select Configurations dialog box, click **Advanced settings**. The Properties dialog box appears.
4. In the Properties 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 box closes.

5 In the Select Configurations dialog box, click **Finish**.

Configuring the Debugger

1 In Eclipse, choose **Run > Debug Configurations**. The Debug Configurations dialog box opens.

2 Double-click **C/C++ Application**. A **throttlecntrl_externenv Default** entry appears under **C/C++ Application**. The **Main** tab of the configuration pane appears on the right side of the dialog box with the following parameter settings:

Parameter	Setting
Name	`throttlecntrl_externenv Default`
C/C++ Application	`Default\throttlecntrl_externenv.exe`
Project	`throttlecntrl_externenv`
Build configuration	`Default`
Enable auto build	Cleared
Disable auto build	Cleared
Use workspace settings	Selected

3 Click **Close**.

Starting the Debugger

To start the debugger:

1 In the main Eclipse window, select **Run > Debug**. A Confirm Perspective Switch dialog box opens.

2 Click **Yes**. Tabbed debugger panes that display debugging information and controls are displayed in the main Eclipse window.

3 Specify the location of the project files. The Cygwin debugger creates a virtual drive (for example, `main() at /cygdrive/`) during the build

process. To run the debugger, Eclipse remaps the drive or locates your project files. Once Eclipse locates the first file, it automatically finds the remaining files. In the Eclipse window, click **Locate File**. The Open dialog box opens.

For information on using the **Edit Source Lookup Path** button, see "Setting the Cygwin Path" on page A-8.

4. Navigate to the `example_main.c` file and click **Open**. Your program opens in the debugger software.

Setting the Cygwin Path

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

To provide the necessary path information:

1. Open the Debug Configurations dialog box by selecting **Run > Debug Configurations > C/C++ Application**.
2. Click the **Source** tab.
3. Click **Add**. The Add Source dialog box opens.
4. Select **Path Mapping** and click **OK**. The Path Mappings dialog box opens.
5. Click **Add**. In the **Compilation path** field, type `\cygdrive\c\`.
6. In the **Local file system path** field, click the **Browse** button, navigate to your `C:\` drive, and click **OK**.
7. Click **Apply**.
8. Click **Close**.

Actions and Commands in the Eclipse Debugger

The following actions and commands are available in the debugger.

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