Embedded Coder®

Getting Started Guide

MATLAB®&SIMULINK®

R2016b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Embedded Coder® Getting Started Guide

© COPYRIGHT 2011–2016 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)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.3 (Release 2012b)
March 2013	Online only	Revised for Version 6.4 (Release 2013a)
September 2013	Online only	Revised for Version 6.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.6 (Release 2014a)
October 2014	Online only	Revised for Version 6.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.8 (Release 2015a)
September 2015	Online only	Revised for Version 6.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.10 (R2016a)
September 2016	Online only	Revised for Version 6.11 (Release 2016b)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

1 Product Overview

2 MATLAB Tutorials

A Installing and Using IDE

Product Overview

Embedded Coder Product Description

Generate C and C++ code optimized for embedded systems

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. Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178).

Learn more about MathWorks support for certification in automotive, aerospace, and industrial automation.

Key Features

- Optimization and code configuration options that extend MATLAB Coder and Simulink Coder
- Storage class, type, and alias definition using Simulink data dictionary capabilities
- Processor-specific code optimization
- Multirate, multitask, and multicore code execution with or without an RTOS
- Code verification, including SIL and PIL testing, custom comments, and code reports with tracing of models to and from code and requirements
- Integration with Texas Instruments™ Code Composer Studio™, Analog Devices® VisualDSP++®, and other third-party embedded development environments
- Standards support, including ASAP2, AUTOSAR, DO-178, IEC 61508, ISO 26262, and MISRA-C

Code Generation Technology

MathWorks® code generation technology produces C or C++ code and executables for algorithms. You can write algorithms programmatically with MATLAB or graphically in the Simulink environment. You can generate code for MATLAB functions and Simulink blocks 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 a high degree of fidelity. Using the Fixed-Point Designer product, you can generate fixed-point code that provides a bit-wise match to model simulation results. Such broad support and high degree 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.

Code generation technology and related products provide tooling that you can apply to the V-model for system development. The V-model is a representation of system development that highlights verification and validation steps in the development process. For more information, see “Validation and Verification for System Development” on page 1-8.

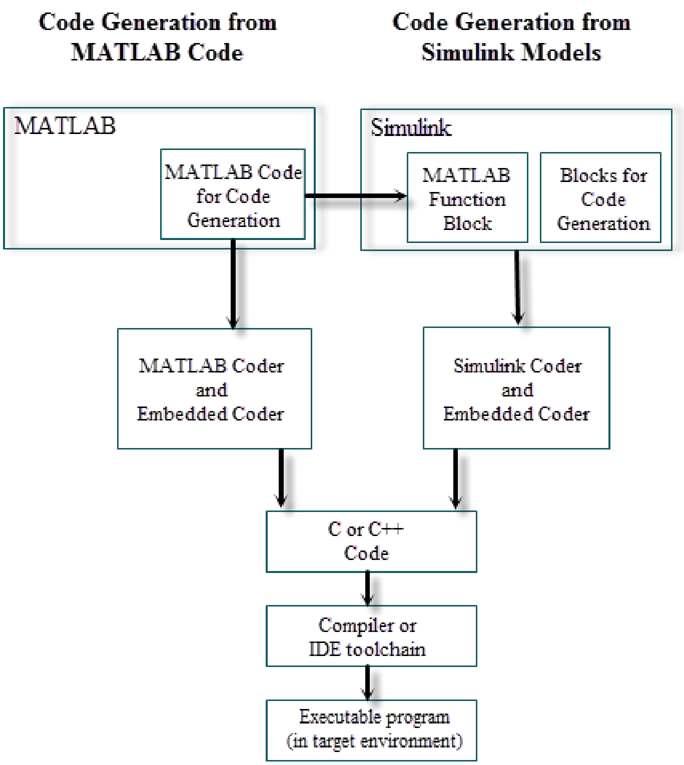
To learn model design patterns that include Simulink blocks, Stateflow® charts, and MATLAB functions, and map to commonly used C constructs, see “Modeling Patterns for C Code” (Embedded Coder).

Code Generation Workflows with Embedded Coder

The Embedded Coder product *extends* the MATLAB Coder and Simulink Coder products with key features that you can use for embedded software development. Using the Embedded Coder 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 applications, functions, and data.
- Enable tracing, reporting, and testing options that facilitate code verification activities.

The code generator supports two workflows for designing, implementing, and verifying generated C or C++ code. The following figure shows the design and deployment environment options.



Although not shown in this figure, other products that support code generation, such as Stateflow software, are available.

To develop algorithms with MATLAB code for code generation, see "Code Generation from MATLAB Code" on page 1-5.

To implement algorithms as Simulink blocks and Stateflow charts in a Simulink model, and generate C or C++ code, see "Code Generation from Simulink Models" on page 1-6.

Code Generation from MATLAB Code

The code generation from MATLAB code workflow with Embedded Coder requires the following products:

- MATLAB
- MATLAB Coder
- Embedded Coder

MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. To generate C or C++ code, you can use MATLAB Coder projects or enter the function `codegen` in the MATLAB Command Window. Embedded Coder provides additional options and advanced optimizations for fine-grain control of the generated code's functions, files, and data. For more information about these options and optimizations , see "Embedded Coder Capabilities for Code Generation from MATLAB Code" on page 2-3.

For more information about generating code from MATLAB code, see "MATLAB Code for Code Generation Workflow Overview" in the MATLAB Coder documentation.

To get started generating code from MATLAB code using Embedded Coder, see "Generate C Code from MATLAB Code" on page 2-2.

Code Generation from Simulink Models

The code generation from Simulink models workflow with Embedded Coder requires the following products:

- MATLAB
- MATLAB Coder
- Simulink
- Simulink Coder
- Embedded Coder

You can implement algorithms as Simulink blocks and Stateflow charts in a Simulink model. To generate C or C++ code from a Simulink model, Embedded Coder provides additional features for implementing, configuring, and verifying your model for code generation.

If you have algorithms written in MATLAB code, you can include the MATLAB code in a Simulink model or subsystem by using the `MATLAB Function` block. When you generate C or C++ code for a Simulink model, the MATLAB code in the MATLAB Function block is also generated into C or C++ code and included in the generated source code.

To get started generating code from Simulink models using Embedded Coder, see "Generate C Code from Simulink Models" on page 3-2.

To learn how to model and generate code for commonly used C constructs using Simulink blocks, Stateflow charts, and MATLAB functions, see "Modeling Patterns for C Code".

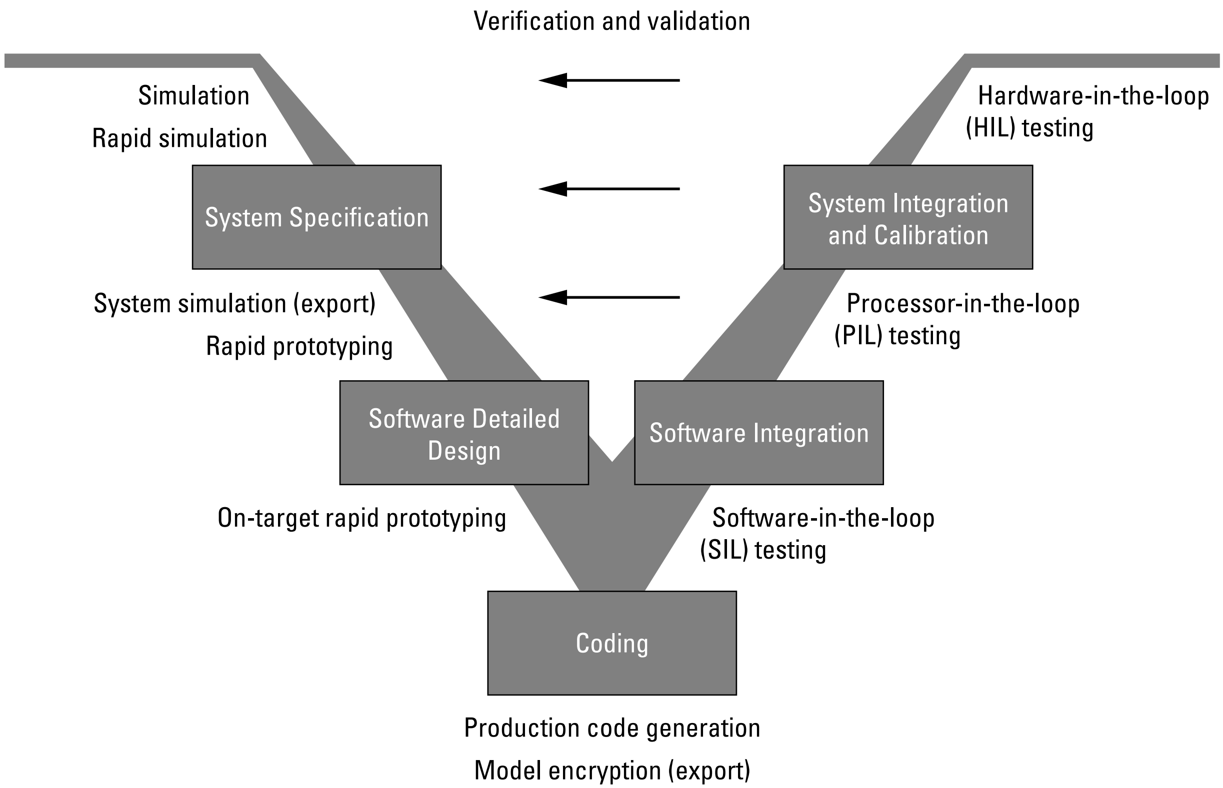
Validation and Verification for System Development

An approach to validating and verifying system development is the V-model.

In this section...

V-Model for System Development

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 to the V-model for system development. For more information about how you can apply MathWorks code generation technology and related products provide tooling to the V-model process, see:

- "Types of Simulation and Prototyping in the V-Model" on page 1-10
- "Types of In-the-Loop Testing in the V-Model" on page 1-11
- "Mapping of Code Generation Goals to the V-Model" on page 1-12

Types of Simulation and Prototyping in the V-Model

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
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	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	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, thus less expensive and more convenient

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
	Can accelerate Simulink simulations with Accelerated and Rapid Accelerated modes	with scripts, without rebuilding the model Can connect to Simulink to monitor signals and tune parameters		

Types of In-the-Loop Testing in the V-Model

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 on desk or test bench	Desktop convenience	Executes on test bench or in lab

	SIL Testing	PIL Testing on Embedded Hardware	PIL Testing on Instruction Set Simulator	HIL Testing
	Executes only in Simulink Reduced hardware cost	Uses hardware — process board and cables	Executes only on host computer with Simulink and integrated development environment (IDE) Reduced hardware cost	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

Mapping of Code Generation Goals to the V-Model

The following tables list goals that you might have, as you apply code generation technology, and where to find guidance on how to meet those goals. Each table focuses on goals that pertain to a step of the V-model for system development.

- 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

Documenting and Validating Requirements

Goals	Related Product Information	Examples
Capture requirements in a document, spreadsheet, data base, or requirements management tool	"Simulink Report Generator" Third-party vendor tools such as Microsoft® Word, Microsoft Excel®, raw HTML, or IBM® Rational® DOORS®	

Goals	Related Product Information	Examples
Associate requirements documents with objects in concept models Generate a report on requirements associated with a model	"Requirements Traceability" (Simulink Verification and Validation™) Bidirectional tracing in Microsoft Word, Microsoft Excel, HTML, and IBM Rational DOORS	`slvnvdemo_fuelsys_docreq`
Include requirements links in generated code	"Review of Requirements Links" (Simulink Verification and Validation)	`rtwdemo_requirements`
Trace model blocks and subsystems to generated code and vice versa	"Code Tracing" (Embedded Coder)	`rtwdemo_hyperlinks`
Verify, refine, and test concept model in non real time on a host system	"Model Architecture and Design" (Simulink Coder) "Model Architecture and Design" (Embedded Coder) "Simulation" (Simulink) "Acceleration" (Simulink)	"Air-Fuel Ratio Control System with Stateflow Charts"
Run standalone rapid simulations Run batch or Monte-Carlo simulations Repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model Tune parameters and monitor signals interactively	"Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File" "Set Up and Use Host/Target Communication Channel"	"Run Rapid Simulations Over Range of Parameter Values" "Run Batch Simulations Without Recompiling Generated Code" "Use MAT-Files to Feed Data to Inport Blocks for Rapid Simulations"

Goals	Related Product Information	Examples
Simulate models for hybrid dynamic systems that include components and an environment or plant that requires variable-step solvers and zero-crossing detection		
Distribute simulation runs across multiple computers	"Simulink Test" "MATLAB Distributed Computing Server" "Parallel Computing Toolbox"	

Developing a Model Executable Specification

Goals	Related Product Information	Examples
Produce design artifacts for algorithms that you develop in MATLAB code for reviews and archiving	"MATLAB Report Generator"	
Produce design artifacts from Simulink and Stateflow models for reviews and archiving	"System Design Description" (Simulink Report Generator™)	`rtwdemo_codegenrpt`
Add one or more components to another environment for system simulation Refine a component model Refine an integrated system model Verify functionality of a model in nonreal time Test a concept model	"Deploy Algorithm Model for Real-Time Rapid Prototyping"	
Schedule generated code	"Absolute and Elapsed Time Computation" "Time-Based Scheduling and Code Generation" "Asynchronous Events"	`rtwdemos`, select **Multirate Support**

Specify function boundaries of systems	"Subsystems"	`rtwdemo_atomic`
		`rtwdemo_ssreuse`
		`rtwdemo_filepart`
		`rtwdemo_exporting_functions`

Specify components and boundaries for design and incremental code generation	"Component-Based Modeling" (Simulink Coder) "Component-Based Modeling" (Embedded Coder)	`rtwdemo_mdlreftop`

Goals	Related Product Information	Examples
Specify function interfaces so that external software can compile, build, and invoke the generated code	"Function and Class Interfaces" (Simulink Coder) "Function and Class Interfaces" (Embedded Coder)	`rtwdemo_fcnprotoctrl` `rtwdemo_cppclass`

Manage data packaging in generated code for integrating and packaging data	"File Packaging" (Simulink Coder)
	
	"File Packaging" (Embedded Coder)
		`rtwdemo_ssreuse`
		`rtwdemo_mdlreftop`
		`rtwdemo_advsc`

Generate and control the format of comments and identifiers in generated code	"Add Custom Comments to Generated Code" (Embedded Coder) "Construction of Generated Identifiers" (Embedded Coder)	`rtwdemo_comments` `rtwdemo_symbols`
Create a zip file that contains generated code files, static files, and dependent data to build generated code in an environment other than your host computer	"Relocate Code to Another Development Environment" (Simulink Coder)	`rtwdemo_buildinfo`
Export models for validation in a system simulator using shared libraries	"Package Generated Code as Shared Object Libraries" (Embedded Coder)	`rtwdemo_shrlib`
Refine component and environment model designs by rapidly iterating between algorithm design and prototyping Verify whether a component can adequately control a physical system in non-real time	"Deployment" (Simulink Coder) "Deployment" (Embedded Coder)	`rtwdemo_profile`

Goals	Related Product Information	Examples
Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design Test hardware		
Generate code for rapid prototyping	"Function and Class Interfaces" "Entry-Point Functions and Scheduling" "Generate Modular Function Code" (Embedded Coder)	`rtwdemo_counter` `rtwdemo_async`
Generate code for rapid prototyping in hard real time, using PCs	"Simulink Real-Time"	"Simulink Real-Time Examples"
Generate code for rapid prototyping in soft real time, using PCs	"Simulink Desktop Real-Time"	`sldrtex_vdp` (and others)

Developing a Detailed Software Design

Goals	Related Product Information	Examples
Refine a model design for representation and storage of data in generated code	"Data Representation" (Simulink Coder) "Data Representation" (Embedded Coder)	

Select code generation features for deployment	"Target Environment Configuration" (Simulink Coder)
	
	"Target Environment Configuration" (Embedded Coder)
	
	"Sharing Utility Code" (Embedded Coder)
		`rtwdemo_counter`
		`rtwdemo_async`
		"Sample Workflows" in the Embedded Coder documentation

Goals	Related Product Information	Examples
	"AUTOSAR Code Generation" (Embedded Coder)	
Specify target hardware settings	"Target Environment Configuration" (Simulink Coder) "Target Environment Configuration" (Embedded Coder)	`rtwdemo_targetsettings`
Design model variants	"Define, Configure, and Activate Variants" (Simulink) "Variant Systems" (Embedded Coder)	
Specify fixed-point algorithms in Simulink, Stateflow, and the MATLAB language subset for code generation	"Data Types and Scaling" (Fixed-Point Designer) "Fixed-Point Code Generation Support" (Fixed-Point Designer)	`rtwdemo_fixpt1` "Air-Fuel Ratio Control System with Fixed-Point Data"
Convert a floating-point model or subsystem to a fixed-point representation	"Conversion Using Simulation Data" (Fixed-Point Designer) "Conversion Using Range Analysis" (Fixed-Point Designer)	`fxpdemo_fpa`
Iterate to obtain an optimal fixed-point design, using autoscaling	"Data Types and Scaling" (Fixed-Point Designer)	`fxpdemo_feedback`
Create or rename data types specifically for your application	"What Are User-Defined Data Types?" (Embedded Coder) "Data Type Replacement" (Embedded Coder)	`rtwdemo_udt`
Control the format of identifiers in generated code	"Construction of Generated Identifiers" (Embedded Coder)	`rtwdemo_symbols`

Goals	Related Product Information	Examples
Specify how signals, tunable parameters, block states, and data objects are declared, stored, and represented in generated code	"Custom Storage Classes" (Embedded Coder)	`rtwdemo_cscpredef`
Create a data dictionary for a model	"Data Definition and Declaration Management" (Embedded Coder)	`rtwdemo_advsc`
Relocate data segments for generated functions and data using `#pragmas` for calibration or data access	"Control Data and Function Placement in Memory by Inserting Pragmas" (Embedded Coder)	`rtwdemo_memsec`
Assess and adjust model configuration parameters based on the application and an expected run-time environment	"Configuration" (Simulink Coder) "Configuration" (Embedded Coder)	"Generate Code Using Simulink® Coder™" "Generate Code Using Embedded Coder®"
Check a model against basic modeling guidelines	"Run Model Checks" (Simulink)	`rtwdemo_advisor1`
Add custom checks to the Simulink Model Advisor	"Customization and Automation"	`slvnvdemo_mdladv`
Check a model against custom standards or guidelines	"Run Model Checks" (Simulink)	
Check a model against industry standards and guidelines (MathWorks Automotive Advisory Board (MAAB), IEC 61508, IEC 62304, ISO 26262, EN 50128 and DO-178)	"Standards, Guidelines, and Block Usage" (Embedded Coder) "Model Guidelines Compliance" (Simulink Verification and Validation)	`rtwdemo_iec61508`
Obtain model coverage for structural coverage analysis such as MC/DC	"Model Coverage Analysis" (Simulink Design Verifier™)	
Prove properties and generate test vectors for models	Simulink Design Verifier	`sldvdemo_cruise_control`

Goals	Related Product Information	Examples
		`sldvdemo_cruise_control_-verification`
Generate reports of models and software designs	"MATLAB Report Generator" (MATLAB Report Generator) "Simulink Report Generator" (Simulink Report Generator) "System Design Description" (Simulink Report Generator)	`rtwdemo_codegenrpt`
Conduct reviews of your model and software designs with coworkers, customers, and suppliers who do not have Simulink available	"Model Web Views" (Simulink Report Generator) "Model Comparison" (Simulink Report Generator)	`slxml_sfcar`
Refine the concept model of your component or system Test and validate the model functionality in real time Test the hardware Obtain real-time profiles and code metrics for analysis and sizing based on your embedded processor Assess the feasibility of the algorithm based on integration with the environment or plant hardware	"Deployment" (Simulink Coder) "Deployment" (Embedded Coder) "Code Execution Profiling" (Embedded Coder) "Static Code Metrics" (Embedded Coder)	`rtwdemos`, select **Embedded IDEs** or **Embedded Targets**

Generate source code for your models, integrate the code into your production build environment, and run it on existing hardware	"Code Generation" (Simulink Coder)
	
	"Code Generation" (Embedded Coder)
		`rtwdemo_counter`
		`rtwdemo_fcnprotoctrl`
		`rtwdemo_cppclass`
		`rtwdemo_async`
		"Sample Workflows" in the Embedded Coder documentation

Goals	Related Product Information	Examples
Integrate existing externally written C or C++ code with your model for simulation and code generation	"Block Creation" (Simulink) "External Code Integration" (Simulink Coder) "External Code Integration" (Embedded Coder)	`rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code**
Generate code for on-target rapid prototyping on specific embedded microprocessors and IDEs	"Deploy Generated Embedded System Software to Application Target Platforms" (Embedded Coder)	In `rtwdemos`, select one of the following: **Embedded IDEs** or **Embedded Targets**

Generating the Application Code

Goals	Related Product Information	Examples
Optimize generated ANSI® C code for production (for example, disable floating-point code, remove termination and error handling code, and combine code entry points into single functions)	"Performance" (Simulink Coder) "Performance" (Embedded Coder)	`rtwdemos`, select **Optimizations**
Optimize code for a specific run-time environment, using specialized function libraries	"Code Replacement" (Simulink Coder) "Code Replacement" (Embedded Coder) "Code Replacement Customization" (Embedded Coder)	"Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®"
Control the format and style of generated code	"Control Code Style" (Embedded Coder)	`rtwdemo_parentheses`
Control comments inserted into generated code	"Add Custom Comments to Generated Code" (Embedded Coder)	`rtwdemo_comments`

Goals	Related Product Information	Examples
Enter special instructions or tags for postprocessing by third-party tools or processes	"Customize Post-Code-Generation Build Processing" (Simulink Coder)	`rtwdemo_buildinfo`
Include requirements links in generated code	"Review of Requirements Links" (Simulink Verification and Validation)	`rtwdemo_requirements`
Trace model blocks and subsystems to generated code and vice versa	"Code Tracing" (Embedded Coder) "Standards, Guidelines, and Block Usage" (Embedded Coder)	`rtwdemo_comments` `rtwdemo_hyperlinks`
Integrate existing externally written code with code generated for a model	"Block Creation" (Simulink) "External Code Integration" (Simulink Coder) "External Code Integration" (Embedded Coder)	`rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code**
Verify generated code for MISRA C®[a] and other run-time violations	"MISRA C Guidelines" (Embedded Coder) "Polyspace Bug Finder" "Polyspace Code Prover"	
Protect the intellectual property of component model design and generated code Generate a binary file (shared library)	"Protected Model" (Simulink) "Package Generated Code as Shared Object Libraries" (Embedded Coder)	
Generate a MEX-file S-function for a model or subsystem so that it can be shared with a third-party vendor	"Automate S-Function Generation with S-Function Builder" (Simulink Coder)	

Goals	Related Product Information	Examples
Generate a shared library for a model or subsystem so that it can be shared with a third-party vendor	"Package Generated Code as Shared Object Libraries" (Embedded Coder)	
Test generated production code with an environment or plant model to verify a conversion of the model to code	"Software-in-the-Loop Simulation" (Embedded Coder)	"Test Generated Code with SIL and PIL Simulations"
Create an S-function wrapper for calling your generated source code from a model running in Simulink	"Write Wrapper S-Function and TLC Files" (Simulink Coder)	
Set up and run SIL tests on your host computer	"Software-in-the-Loop Simulation" (Embedded Coder)	"Test Generated Code with SIL and PIL Simulations"

a. MISRA® and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

Integrating and Verifying Software

Goals	Related Product Information	Examples
Integrate existing externally written C or C++ code with a model for simulation and code generation	"Block Creation" (Simulink) "External Code Integration" (Simulink Coder) "External Code Integration" (Embedded Coder)	`rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code**
Connect to data interfaces for generated C code data structures	"Data Exchange Interfaces" (Simulink Coder) "Data Exchange Interfaces" (Embedded Coder)	`rtwdemo_capi` `rtwdemo_asap2`
Control the generation of code interfaces so that external software can compile, build, and invoke the generated code	"Function and Class Interfaces" (Embedded Coder)	`rtwdemo_fcnprotoctrl` `rtwdemo_cppclass`

Goals	Related Product Information	Examples
Export virtual and function-call subsystems	"Export Function-Call Subsystems" (Embedded Coder)	`rtwdemo_exporting_functions`
Include target-specific code	"Code Replacement" (Simulink Coder) "Code Replacement" (Embedded Coder) "Code Replacement Customization" (Embedded Coder)	"Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®"
Customize and control the build process	"Build Process Customization" (Simulink Coder)	`rtwdemo_buildinfo`
Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer	"Relocate Code to Another Development Environment" (Simulink Coder)	`rtwdemo_buildinfo`
Integrate software components as a complete system for testing in the target environment	"Target Environment Verification" (Embedded Coder)	
Generate source code for integration with specific production environments	"Code Generation" (Simulink Coder) "Code Generation" (Embedded Coder)	`rtwdemo_async` "Sample Workflows" in the Embedded Coder documentation
Integrate code for a specific run-time environment, using specialized function libraries	"Code Replacement" (Simulink Coder) "Code Replacement" (Embedded Coder) "Code Replacement Customization" (Embedded Coder)	"Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®"

Goals	Related Product Information	Examples
Enter special instructions or tags for postprocessing by third-party tools or processes	"Customize Post-Code-Generation Build Processing" (Simulink Coder)	`rtwdemo_buildinfo`
Integrate existing externally written code with code generated for a model	"Block Creation" (Simulink) "External Code Integration" (Simulink Coder) "External Code Integration" (Embedded Coder)	`rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code**
Connect to data interfaces for the generated C code data structures	"Data Exchange Interfaces" (Simulink Coder) "Data Exchange Interfaces" (Embedded Coder)	`rtwdemo_capi` `rtwdemo_asap2`
Schedule the generated code	"Timers" "Time-Based Scheduling" "Event-Based Scheduling"	`rtwdemos`, select **Multirate Support**
Verify object code files in a target environment	"Software-in-the-Loop Simulation" (Embedded Coder)	"Test Generated Code with SIL and PIL Simulations"
Set up and run PIL tests on your target system	"Processor-in-the-Loop Simulation" (Embedded Coder)	"Test Generated Code with SIL and PIL Simulations" "Configure Processor-In-The-Loop (PIL) for a Custom Target" "Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation" See the list of `supported hardware` for the Embedded Coder product on the MathWorks Web site, and then

Goals	Related Product Information	Examples
		find an example for the related product of interest

Integrating, Verifying, and Calibrating System Components

Goals	Related Product Information	Examples
Integrate the software and its microprocessor with the hardware environment for the final embedded system product Add the complexity of the environment (or plant) under control to the test platform Test and verify the embedded system or control unit by using a real-time target environment	"Deploy Algorithm Model for Real-Time Rapid Prototyping" "Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation" "Deploy Generated Standalone Executables To Target Hardware" (Embedded Coder) "Deploy Generated Embedded System Software to Application Target Platforms" (Embedded Coder)	
Generate source code for HIL testing	"Code Generation" (Simulink Coder) "Code Generation" (Embedded Coder) "Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation"	
Conduct hard real-time HIL testing using PCs	"Simulink Real-Time"	"Simulink Real-Time Examples"
Tune ECU properly for its intended use	"Data Exchange Interfaces" (Simulink Coder) "Data Exchange Interfaces" (Embedded Coder)	`rtwdemo_capi` `rtwdemo_asap2`
Generate ASAP2 data files	"Export ASAP2 File for Data Measurement and Calibration"	`rtwdemo_asap2`

Goals	Related Product Information	Examples
Generate C API data interface files	"Exchange Data Between Generated and External Code Using C API"	`rtwdemo_capi`

Target Environments and Applications

In this section...

About Target Environments

In addition to generating source code, the code generator produces make or project files to build an executable program 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 program 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

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 environments 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®[a] environment that uses a non-real-time operating system, such as Microsoft Windows® or Linux®[b]. 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.

Target Environment	Description
	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: • Simulink Real-Time 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. 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

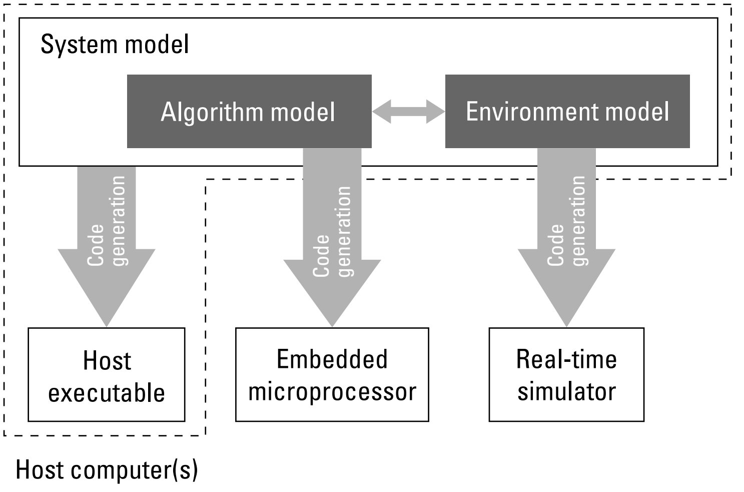
a. UNIX is a registered trademark of The Open Group in the United States and other countries.
b. Linux is a registered trademark of Linus Torvalds.

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 a component.

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.

Application	Description
Host Computer	
"Acceleration"	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.

Application	Description
Rapid Simulation	You execute code generated for a model in non-real-time on the host computer, but outside the context of the MATLAB and Simulink environments.
Shared Object Libraries	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 Protection	You generate a protected model for use by a third-party vendor in another Simulink simulation environment.
Real-Time Simulator	
Real-Time 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 control the physical system.
Shared Object Libraries	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.
Hardware-in-the-Loop (HIL) Simulation	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.

Application	Description
Embedded Microprocessor	
"Code Generation"	From a model, you generate code that is optimized for speed, memory usage, simplicity, and possibly, compliance with industry standards and guidelines.
"Software-in-the-Loop Simulation"	You execute generated code with your plant model within Simulink to verify 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.
"Processor-in-the-Loop Simulation"	You test an object code component with a plant or environment model in an open- or closed-loop simulation to verify model-to-code conversion, cross-compilation, and software integration.
Hardware-in-the-loop (HIL) Simulation	You verify an embedded system or embedded computing unit (ECU), using a real-time target environment.

MATLAB Tutorials

Generate C Code from MATLAB Code

In this section...

About MATLAB Coder

MATLAB Coder generates standalone C and C++ from MATLAB code. The generated source code is portable and readable. The code generator 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.

When generating C and C++ code from MATLAB code, follow this workflow.

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 application-specific requirements
- Enable tracing options that help you to verify the generated code

See "Embedded Coder Capabilities for Code Generation from MATLAB Code" on page 2-3.

Getting Started Tutorials

The following tutorials will help you get started with using the code generator to produce code from MATLAB code for embedded system applications.

- "Controlling C Code Style" on page 2-9
- "Tracing Between Generated C Code and MATLAB Code" on page 2-14

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 `https://www.mathworks.com/support/compilers/current_release/`.

Before generating C code, you must set up the C compiler. See "Setting Up the C or C++ Compiler" in the MATLAB Coder documentation.

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

Setting Up Tutorial Files

The tutorial files are available in the following folder: `*matlabroot*/help/toolbox/ecoder/examples` (open). 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.

Embedded Coder Capabilities for Code Generation from MATLAB Code

The Embedded Coder product extends the MATLAB Coder product with the following options and optimizations for C/C++ code generation.

Goal	Project Setting	Code Configuration Object Property	More Information
Execution Time			
Control generation of floating-point data and operations	**Support only purely-integer numbers**	`PurelyIntegerCode`	N/A

Goal	Project Setting	Code Configuration Object Property	More Information
Simplify array indexing in loops in the generated code	**Simplify array indexing**	`EnableStrengthReduct:`	“Simplify Multiply Operations for Array Indexing in Loops”
Replace functions and operators in the generated code to meet application-specific code requirements	**Code replacement library** on the **Custom Code** tab	`CodeReplacement-Library`	Embedded Coder offers additional libraries and the ability to create and use custom code. See “Code Replacement Customization”.
Create and register application-specific implementations of functions and operators	N/A	N/A	“Code Replacement Customization”
Code Appearance			
Specify use of single-line or multiline comments in the generated code	**Comment Style**	`CommentStyle`	“Specify Comment Style for C/C++ Code”
Generate traceable code that includes the MATLAB function help text in the function banner	**MATLAB function help text**	`MATLABFcnDesc`	“Tracing Between Generated C Code and MATLAB Code” on page 2-14
Convert if-elseif-else patterns to switch-case statements	**Convert if-elseif-else patterns to switch-case statements**	`ConvertIfToSwitch`	“Controlling C Code Style” on page 2-9
Specify that the extern keyword is included in declarations of generated external functions	**Preserve extern keyword in function declarations**	`PreserveExtern-InFcnDecls`	N/A

Goal	Project Setting	Code Configuration Object Property	More Information
Specify the level of parenthesization in the generated code	**Parentheses**	`ParenthesesLevel`	N/A
Specify whether to replace multiplications by powers of two with signed left bitwise shifts in the generated code	**Use signed shift left for fixed-point operations and multiplication by powers of 2**	`EnableSignedLeftShif`	"Control Signed Left Shifts in Generated Code"
Specify whether to allow signed right bitwise shifts in the generated code	**Allow right shifts on signed integers**	`EnableSignedRightShi`	N/A
Control data type casts in the generated code	**Casting mode** on the **All Settings** tab	`CastingMode`	"Control Data Type Casts in Generated Code"
Specify the indent style for the generated code	**Indent style** on the **All Settings** tab **Indent size** on the **All Settings** tab	`IndentStyle` `IndentSize`	"Specify Indent Style for C/C++ Code"
Customize generated global variable identifiers	**Global variables**	`CustomSymbolStr-GlobalVar`	"Customize Generated Identifiers"
Customize generated global type identifiers	**Global types**	`CustomSymbolStrType`	"Customize Generated Identifiers"
Customize generated field names in global type identifiers	**Field name of global types**	`CustomSymbolStrField`	"Customize Generated Identifiers"
Customize generated local functions identifiers	**Local functions**	`CustomSymbolStrFcn`	"Customize Generated Identifiers"
Customize generated identifiers for local temporary variables	**Local temporary variables**	`CustomSymbolStr-TmpVar`	"Customize Generated Identifiers"

Goal	Project Setting	Code Configuration Object Property	More Information
Customize generated identifiers for constant macros	**Constant macros**	`CustomSymbolStrMacro`	"Customize Generated Identifiers"
Customize generated identifiers for EMX Array types (Embeddable mxArray types)	**EMX Array Types**	`CustomSymbolStr-EMXArray`	"Customize Generated Identifiers"
Customize generated identifiers for EMX Array (Embeddable mxArrays) utility functions	**EMX Array Utility Functions**	`CustomSymbolStrEMX-ArrayFcn`	"Customize Generated Identifiers"
Customize function interface in the generated code	**Terminate function required**	`IncludeTerminateFcn`	N/A
Customize file and function banners	N/A	`CodeTemplate`	• "Generate Custom File and Function Banners for C/C++ Code" • "Code Generation Template Files for MATLAB Code"
Control declarations and definitions of global variables in the generated code	N/A	N/A	• "Storage Classes for Code Generation from MATLAB Code" • "Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code"
Debugging			

Goal	Project Setting	Code Configuration Object Property	More Information
Generate a static code metrics report including generated file information, number of lines, and memory usage	**Static code metrics**	`GenerateCodeMetrics-Report`	"Generate a Static Code Metrics Report for MATLAB Code"
Generate a code replacement report that summarizes the replacements used from the selected code replacement library	**Code replacements**	`GenerateCode-ReplacementReport`	• "Replace Code Generated from MATLAB Code" • "Verify Code Replacements"
Highlight single-precision, double-precision, and expensive fixed-point operations in the code generation report	**Highlight potential data type issues**	`HighlightPotential-DataTypeIssues`	"Highlight Potential Data Type Issues in a Report"
Custom Code			
Replace functions and operators in the generated code to meet application-specific code requirements	**Code replacement library**	`CodeReplacement-Library`	Embedded Coder offers additional libraries and the ability to create and use custom code. See "Code Replacement Customization".
Create and register application-specific implementations of functions and operators	N/A	N/A	"Code Replacement Customization"
Verification			

Goal	Project Setting	Code Configuration Object Property	More Information
Verify generated code using software-in-the-loop and processor-in-the-loop execution	N/A	`VerificationMode`	"Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution"
Debug code during software-in-the-loop execution	**Enable source-level debugging for SIL** on the **Debugging** pane	`SILDebugging`	"Debug Generated Code During SIL Execution"
Profile execution times during software -in-the-loop and processor-in-the-loop execution	**Enable entry point execution profiling for SIL/PIL** on the **Debugging** pane	`CodeExecution-Profiling`	"Execution Time Profiling for SIL and PIL"
Verify and profile ARM optimized code	**Hardware Board** on the **Hardware** pane	`Hardware`	• "PIL Execution with ARM Cortex-A at the Command Line" • "PIL Execution with ARM Cortex-A by Using the MATLAB Coder App"

Controlling C Code Style

In this section...

About This Tutorial

Learning Objectives

This tutorial shows you how to:

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

Required Products

This tutorial requires the following products:

- MATLAB
- MATLAB Coder
- C compiler

For most platforms, a default compiler is supplied with MATLAB.

MATLAB Coder locates and uses a supported installed compiler. See Supported and Compatible Compilers on the MathWorks website.

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

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, at the prompt, enter `ver`.

Required Files

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

Copy File to a Local Working Folder

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

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

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

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

Open the MATLAB Coder App

On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon.

The app opens the **Select Source Files** page.

Specify Source Files

1 On the **Select Source Files** page, type or select the name of the entry-point function `test_code_style.m`.

2 In the **Project location** field, change the project name to `code_style.prj`.

3 Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

Because C uses static typing, at compile time, the code generator must determine the properties of all variables in the MATLAB files. Therefore, you must specify the properties of all function inputs. To define the properties of the input `x`:

1 Click **Let me enter input or global types directly**.

2 Click the field to the right of `x`.

3 From the list of options, select `int16`. Then, select `scalar`.

4 Click **Next** to go to the **Check for Run-Time Issues** step.

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

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. Using this step, you can detect and fix run-time errors that are harder to diagnose in the generated C code. By default, the MEX function includes memory integrity checks. These checks perform array bounds and dimension checking. The checks detect violations of memory integrity in code generated for MATLAB functions. For more information, see “Control Run-Time Checks”.

1 To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow .

2 In the **Check for Run-Time Issues** dialog box, enter code that calls `test_code_style` with an example input. For this example, enter `test_code_style(int16(4))`.

3 Click **Check for Issues**.

   The app generates a MEX function. It runs the MEX function with the example input. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.

4 Click **Next** to go to the **Generate Code** step.

Configure Build Parameters

1 To open the **Generate** dialog box, click the **Generate** arrow .
2 Set the **Build type** to `Static Library (.lib)`.
3 Click **More settings**.
4 On the **Code Appearance** tab, select the **Convert if-elseif-else patterns to switch-case statements** check box.
5 On the **Debugging** tab, verify that **Always create a code generation report** is selected, and then click **Close**.

Generate C Code

Click **Generate**.

When code generation is complete, the code generator produces a C static library, `test_code_style.lib`, and C code in the `/codegen/lib/test_code_style` subfolder. Because you selected report generation, the code generator provides a link to the report.

View the Generated C Code

The code generator produces C code in the file `test_code_style.c`.

To view the generated code:

1 On the **Generate Code** page, under **Output Files**, click `test_code_style.c`. In the code window, you can see that in the C code, the `switch-case` statement replaces the `if-elseif-else` pattern.

```
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;
}
```

2 Click **Next** to open the **Finish Workflow** page.

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to the generated output.

Key Points to Remember

- To check for run-time issues before code generation, perform the **Check for Run-Time Issues** step.
- To access build configuration settings, on the **Generate Code** page, open the **Generate** dialog box, and then click **More Settings**.

Learn More

To	See
Learn how to generate C/C++ code using the MATLAB Coder app.	"C Code Generation Using the MATLAB Coder App"
Learn how to generate C/C++ code using MATLAB code using the command-line interface.	"C Code Generation at the Command Line"

Tracing Between Generated C Code and MATLAB Code

In this section...

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 these tutorials, you must install the following products:

- MATLAB
- MATLAB Coder
- Embedded Coder
- C compiler

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

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 2-15.

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\help\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.
- `-args` to specify the class, size, and complexity of the input parameters.

```
codegen -config cfg  -report polar2cartesian -args {0, 0}
```

`codegen` generates a C static library, `polar2cartesian.lib`, 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 `polar2cartesian.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 a 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.
- Use the `-args` option to specify the class, size, and complexity of input parameters.

More About

- “Generate Traceable Code”

Simulink Code Generation Tutorials

Generate C Code from Simulink Models

In this section...

The code generator produces readable, compact, and fast C and C++ code for use on embedded processors, on-target rapid prototyping boards, and microprocessors used in mass production. You can generate code for a wide variety of applications. These tutorials focus on real-time deployment of a discrete-time control system. The tutorials include how to:

- "Configure a Model for Code Generation" on page 3-6
- "Generate and Analyze C Code" on page 3-11
- "Customize Function Interface and File Packaging" on page 3-26
- "Define Data in the Generated Code" on page 3-32
- "Customize Code Appearance" on page 3-21
- "Deploy and Test Executable Program" on page 3-39

Prerequisites

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

- MATLAB
- MATLAB Coder
- Simulink
- Simulink Coder
- Embedded Coder

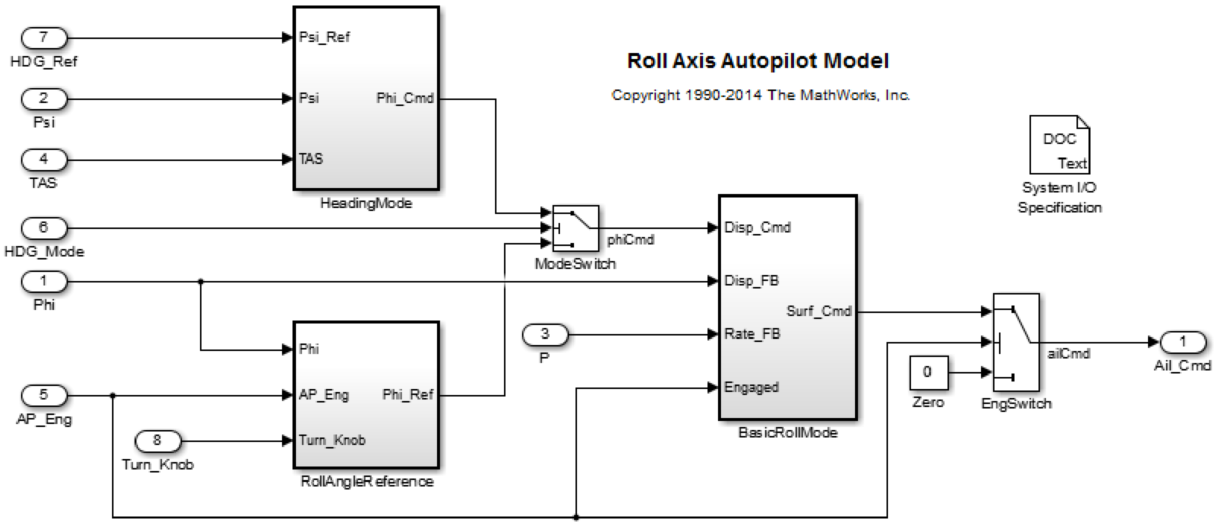
Example Models in Tutorials

The code verification and validation process depends on your model meeting your requirements and exactly representing your design. Functionality in the model must be traceable back to model requirements. You can use reviews, analysis, simulations, and requirements-based tests to prove that your original requirements are met by your design and that the design does not contain unintended functionality. Performing verification and validation activities at each step of the process can reduce expensive errors during production.

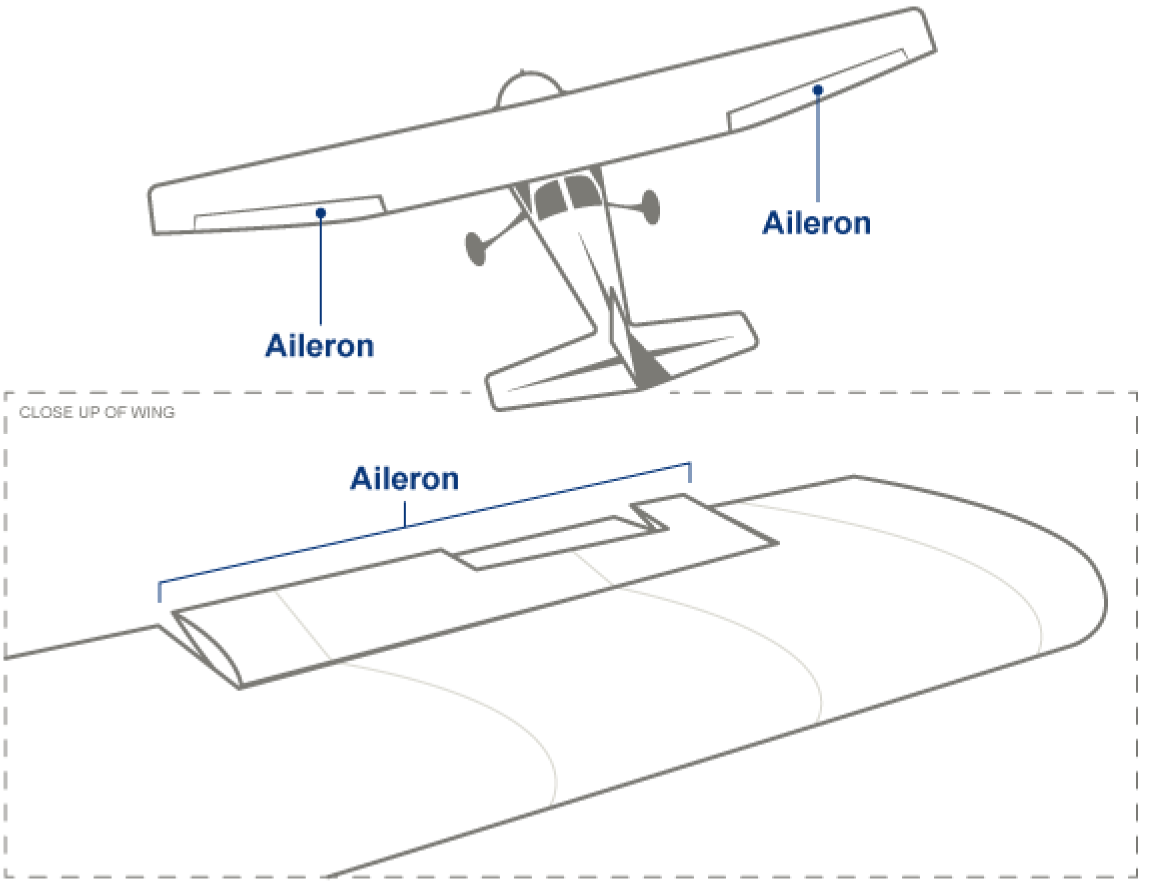
The Embedded Coder tutorials use the `rtwdemo_roll` model, which has been verified for simulation. To open the model, in the Command Window, type:

```
rtwdemo_roll
```

The model opens in the Simulink Editor.



The `rtwdemo_roll` model implements a basic roll axis autopilot algorithm, which controls the aileron position of an aircraft.



There are two operating modes: roll attitude hold and heading hold. The mode logic for these modes is external to this model. The model architecture uses atomic subsystems to represent the roll angle reference (`RollAngleReference`), heading hold mode (`HeadingMode`), and basic roll attitude (`BasicRollMode`) functions as atomic subsystems. The roll attitude control function is a PID controller that uses roll attitude and roll rate feedback to produce an aileron command. The input to the controller is either a basic roll angle reference or a roll command to track the desired heading. The controller operates at `40 Hz`.

Two additional models are provided for the Embedded Coder tutorials:

- `rtwdemo_roll_codegen`: This model is `rtwdemo_roll` configured for code generation with optimizations applied according to the code generation objectives.
- `rtwdemo_roll_harness`: This model is a harness model to test `rtwdemo_roll_codegen`.

To begin the tutorials for code generation, see the first example, “Configure a Model for Code Generation” on page 3-6.

Configure a Model for Code Generation

In this section...

Model configuration parameter settings determine how a model simulates and how the code generator produces code and builds an executable for the model. You specify the model configuration parameters on the Configuration Parameters dialog box or at the command line. The settings in the Configuration Parameters dialog box specify the model's active configuration set, which is saved with the model.

When generating code for an embedded system, choosing the model configuration settings can be complex. At a minimum, you must configure the solver, system target file, hardware implementation, and optimizations according to your application requirements.

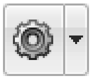
Solver for Code Generation

To prepare the model for generating C89/C90 compliant C code:

1. If `rtwdemo_roll` is not already open, in the Command Window, type:

   ```
   rtwdemo_roll
   ```

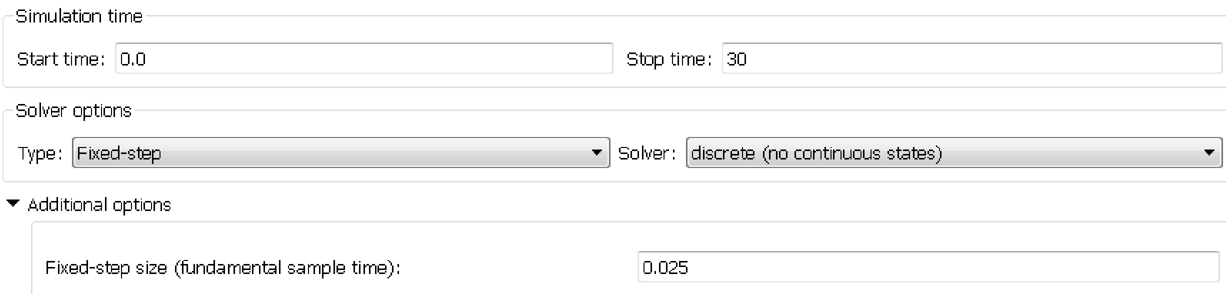
2. Save the model to a local folder as `roll.slx`.
3. To open the Configuration Parameters dialog box, on the Simulink Editor toolbar, click the **Model Configuration Parameters** icon.

4. In the Configuration Parameters dialog box, in the left navigation pane, select the **Solver** pane.

To generate code, the model must use a fixed-step solver, which maintains a constant (fixed) step size. In the generated code, the **Solver** parameter applies a fixed-step integration technique for computing the state derivative of the model. The **Fixed-step**

size parameter sets the base rate, which must be the lowest common multiple of all rates in the system. For `roll`, the following solver settings are selected.

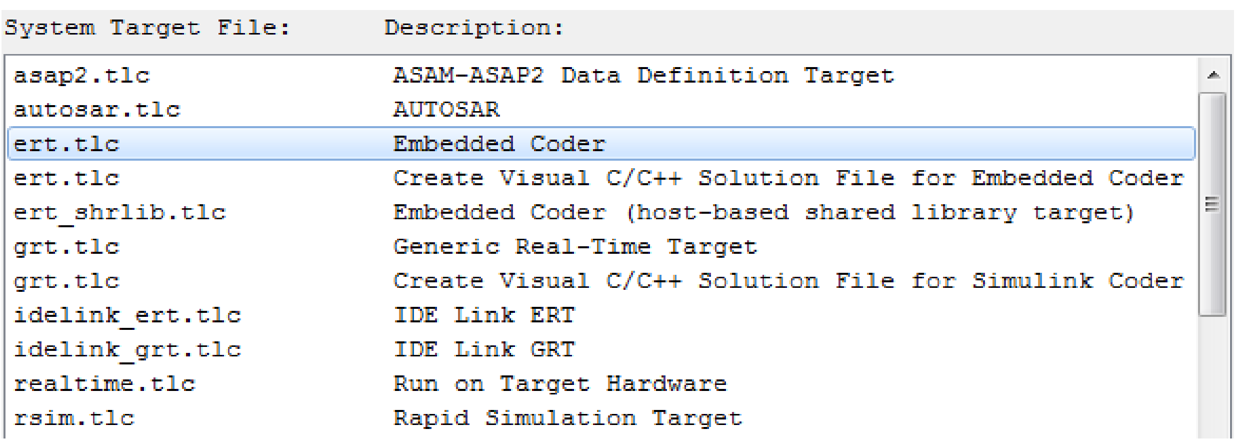


Code Generation Target

To specify a target configuration for the model, you can choose a ready-to-run Embedded Real-Time Target (ERT) configuration. The code generator uses this target file to generate code that is optimized for embedded system deployment.

1 In the Configuration Parameters dialog box, select the **Code Generation** pane.

2 To open the System Target File Browser dialog box, click the **System target file** parameter **Browse** button. The System Target File Browser dialog box includes a list of available targets. This example uses the system target file `ert.tlc Embedded Coder`, which is already set.



System Target File:	Description:
asap2.tlc	ASAM-ASAP2 Data Definition Target
autosar.tlc	AUTOSAR
ert.tlc	Embedded Coder
ert.tlc	Create Visual C/C++ Solution File for Embedded Coder
ert_shrlib.tlc	Embedded Coder (host-based shared library target)
grt.tlc	Generic Real-Time Target
grt.tlc	Create Visual C/C++ Solution File for Simulink Coder
idelink_ert.tlc	IDE Link ERT
idelink_grt.tlc	IDE Link GRT
realtime.tlc	Run on Target Hardware
rsim.tlc	Rapid Simulation Target

3 In the System Target File Browser dialog box, click **OK**.

Check Model Configuration

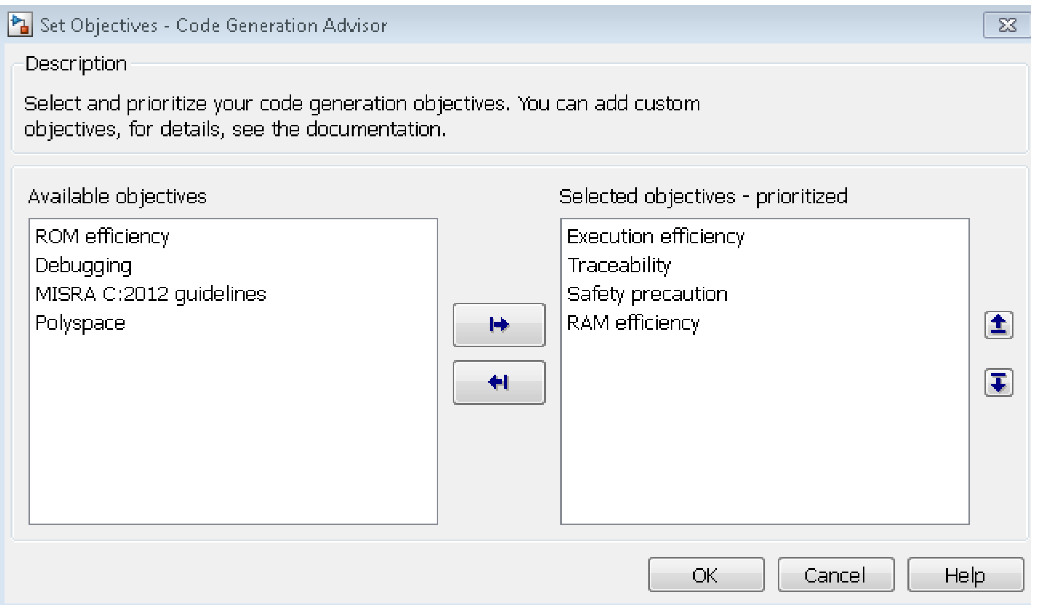
When generating code for real-time deployment, your application might have objectives related to code efficiency, memory usage, safety precaution, and traceability. You can run the Code Generation Advisor to assess whether the model configuration settings meet your set of prioritized objectives. After running the advisor, you get information on how to modify your model configuration parameters to meet the specified objectives.

Set Code Generation Objectives with Code Generation Advisor

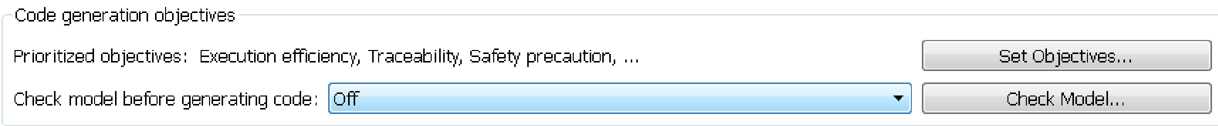
1 In the Configuration Parameters dialog box, select the **Code Generation** pane.

2 Click **Set Objectives**.

3 In the Select Objectives dialog box, the following objectives are in the **Selected objectives — prioritized** list in the following order: `Execution efficiency`, `Traceability`, `Safety precaution`, and `RAM efficiency`.



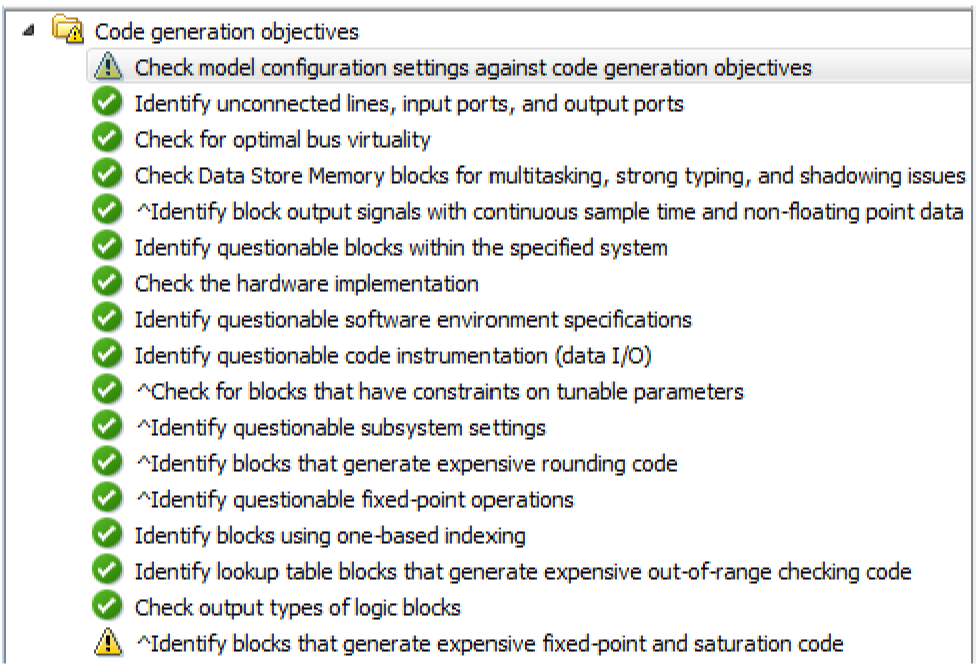
4 Click **OK**. In the Configuration Parameters dialog box, the selected objectives are shown in the **Prioritized objectives** list.



Check Model Against Code Generation Objectives

1. In the Configuration Parameters dialog box, on the **Code Generation** pane, click **Check Model**.
2. In the System Selector dialog box, click **OK** to run checks on `roll`.

   The Code Generation Advisor window opens. After the advisor runs, in the left pane, there are two warnings indicated by yellow triangles.

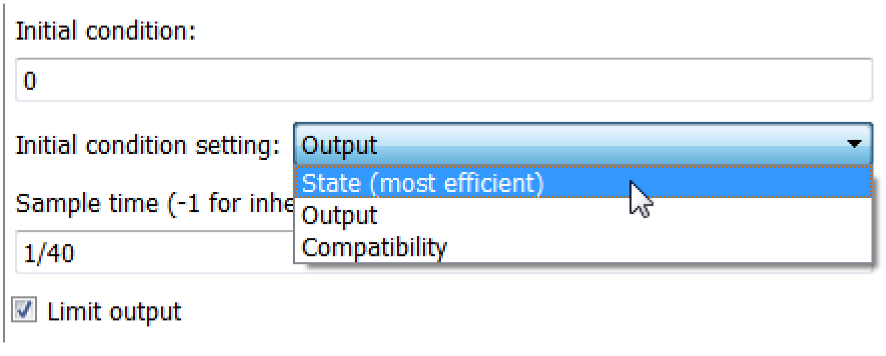


View Model Configuration Recommendations

In the Code Generation Advisor window:

1 In the left pane, click **Check model configuration settings against code generation objectives**.
2 In the right pane, review the recommendations for the configuration parameters in the table.
3 To change the configuration parameters that caused the warnings to the software-recommended settings, click **Modify Parameters**. The **Result** table displays the parameters and changed values. Clicking a parameter name displays Configuration Parameters dialog box pane where the parameter exists.
4 In the left pane, click the next warning for **Identify blocks that generate expensive fixed-point and saturation code**.
5 In the right pane, find the warning, **Identify Discrete Integrator blocks for questionable fixed-point operations**. Under the warning, click the link to the Integrator block.

   In the Simulink Editor, the Integrator block is highlighted in blue.
6 Right-click the Integrator block and in the list, select `Block Parameters(DiscreteIntegrator)`.
7 In the Block Parameter dialog box, set the **Initial condition setting** to `State (most efficient)`.



8 Click **Apply** and **OK**.
9 Save your model.

The example model `rtwdemo_roll_codegen` contains the modifications made in this example to `rtwdemo_roll`. The next example shows how to generate code, examine the code, and trace between the code and model. See "Generate and Analyze C Code" on page 3-11.

Generate and Analyze C Code

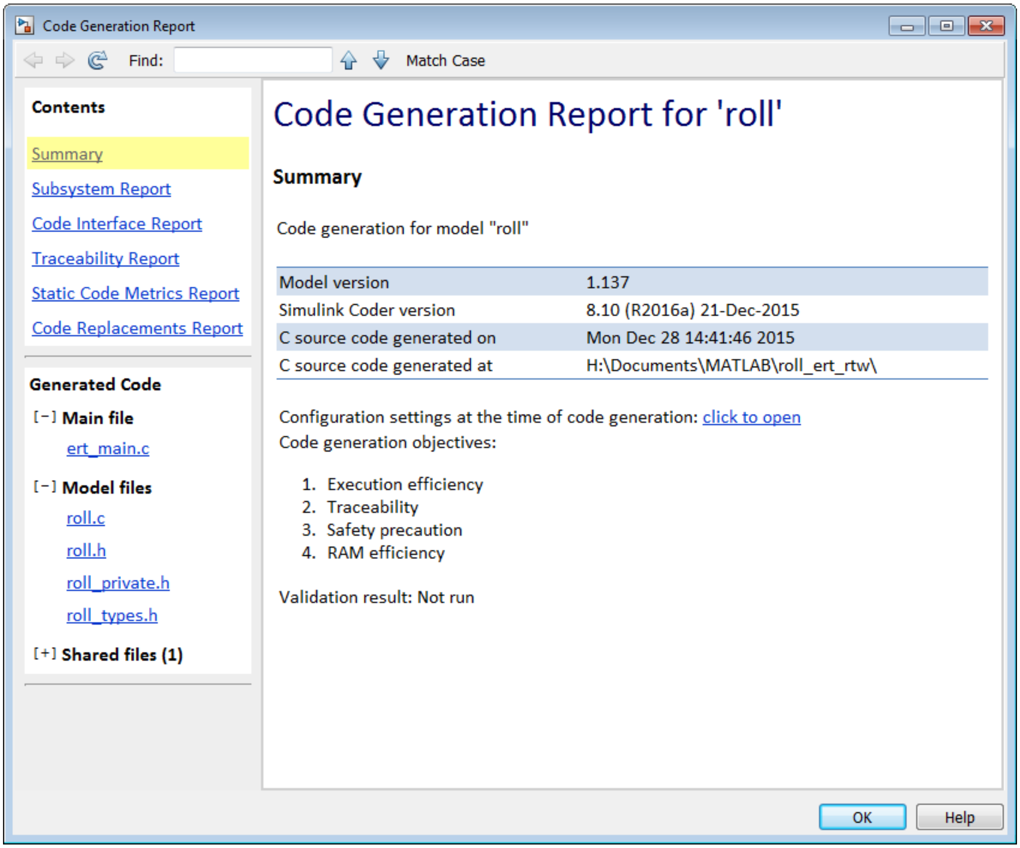
After configuring your model for code generation, you generate and view the code. To analyze the generated code, you can generate an HTML code generation report that provides a view of the generated code and information about the code. This example uses the configured model `roll` from the example, "Generate and Analyze C Code" on page 3-11. For this example, open `rtwdemo_roll_codegen` and save it to a local folder as `roll.slx`.

Generate Code

Before generating code, you can specify that the code generator produce an HTML report that includes the generated code and information about the model. This information helps you to evaluate the generated code.

1. Open the Configuration Parameters dialog box.
2. In the left navigation pane, select the **Code Generation > Report** pane.
3. The selected parameters creates a code generation report and include traceability between the code and the model.
   - "Create code generation report"
   - "Open report automatically"
   - "Code-to-model" (located on the **All Parameters** tab)
   - "Model-to-code", which enables the **Traceability report contents** (located on the **All Parameters** tab).
4. To include static code metrics in the code generation report, confirm that "Static code metrics" is selected.
5. On the **Code Generation** pane, select the **Generate code only** check box.
6. Click **Apply**.
7. Press **Ctrl+B** to generate code.

After the code generation process is complete, the HTML code generation report opens.



Note: If you close the code generation report, you can reopen the report from the Simulink Editor by selecting the menu option: **Code > C/C++ Code > Code Generation Report > Open Model Report**.

Analyze the Generated Code

The code generation process places the source code files in the build folder `roll_ert_rtw`. The HTML code generation report files are placed in the `roll_ert_rtw/html` folder. The code generation report includes the generated code and several reports that provide information for evaluating the generated code. The following sections describe each of these reports:

- “Subsystem Report” on page 3-13
- “Code Interface Report” on page 3-14
- “Traceability Report” on page 3-15
- “Static Code Metrics Report” on page 3-17
- “Code Replacements Report” on page 3-17
- “Generated Code” on page 3-17

Subsystem Report

To open the subsystem report, in the left pane of the code generation report, click **Subsystem Report**. You can implement nonvirtual subsystems as inlined, void/void functions, or reusable functions. In the subsystem report, you can view information on how nonvirtual subsystems are configured in the model and implemented in the code. In the **Code Mapping** section, the subsystem report provides traceability from the table to the Subsystem block in the model. The table includes information about whether a subsystem is configured for reuse and if the subsystem function code is reused.

Non-virtual subsystems in roll

1. Code Mapping [hide]

The following table:

- provides a mapping from the non-virtual subsystems in the model to functions or reusable functions in the generated code and
- notes exceptions that caused some non-virtual subsystems to not reuse code even though they were assigned a function packaging setting ('Function packaging' entry on the Subsystem Block Dialog) of 'Auto' or 'Reusable function'.

Subsystem	Reuse Setting	Reuse Outcome	Outcome Diagnostic
<S2>	Inline	Inline	normal
<S1>	Inline	Inline	normal
<S3>	Inline	Inline	normal

The **Code Reuse Exceptions** section provides information on subsystems configured for reuse, but code reuse does not occur. For this model, there are no reuse exceptions.

Code Interface Report

The code interface report provides documentation of the generated code interface for consumers of the generated code. The generated code interface includes model entry point functions and interface data. The information in the report can help facilitate code reviews and code integration. There are potentially three entry point functions to initialize, step, and terminate the real-time capable code. The code generated for this model has an initialize and step function.

Entry Point Functions

Function: roll_initialize

Prototype	**void roll_initialize(void)**
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	roll.h

Function: roll_step

Prototype	**void roll_step(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 0.025 seconds
Arguments	None
Return value	None
Header file	roll.h

For `roll`, the **Inports** and **Outports** sections include block names that you can click to navigate to the corresponding block in the model. The other columns in the table include the name for the block, the data type, and dimension as it is represented in the generated code.

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/Phi	roll_U.Phi	real32_T	1
<Root>/Psi	roll_U.Psi	real32_T	1
<Root>/P	roll_U.P	real32_T	1
<Root>/TAS	roll_U.TAS	real32_T	1
<Root>/AP_Eng	roll_U.AP_Eng	boolean_T	1
<Root>/HDG_Mode	roll_U.HDG_Mode	boolean_T	1
<Root>/HDG_Ref	roll_U.HDG_Ref	real32_T	1
<Root>/Turn_Knob	roll_U.Turn_Knob	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Ail_Cmd	roll_Y.Ail_Cmd	real32_T	1

Traceability Report

To map model objects to and from the generated code, open the traceability report. The **Eliminated / Virtual Blocks** table lists objects that are virtual or eliminated from the generated code due to an optimization.

<S3>/Turn_Knob	Inport
<S3>/LatchPhi	Masked SubSystem
<S3>/System I//O Specification	Masked SubSystem
<S3>/Phi_Ref	Outport
<S4>/EmptySubsystem	Empty SubSystem

In the **Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions** table, click an **Object Name** to highlight the object in the model diagram. You can also click the corresponding **Code Location**, which displays the generated code for that object.

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Root system: roll

Object Name	Code Location
<Root>/Phi	roll.c:51, 127 roll.h:41
<Root>/Psi	roll.c:74 roll.h:42
<Root>/P	roll.c:143 roll.h:43
<Root>/TAS	roll.c:75 roll.h:44
<Root>/AP_Eng	roll.c:43, 100, 172 roll.h:45
<Root>/HDG_Mode	roll.c:72 roll.h:46
<Root>/HDG_Ref	roll.c:73 roll.h:47
<Root>/Turn_Knob	roll.c:76 roll.h:48
<Root>/BasicRollMode	roll.c:98, 169, 175, 188
<Root>/EngSwitch	roll.c:171, 196
<Root>/HeadingMode	roll.c:83, 86
<Root>/ModeSwitch	roll.c:68, 96
<Root>/RollAngleReference	roll.c:38, 66, 88, 93
<Root>/Zero	roll.c:191
<Root>/Ail_Cmd	roll.c:178, 182, 190 roll.h:53

Static Code Metrics Report

You can monitor code metrics as you develop your model and refine its configuration. The code generator performs static analysis of the generated C code and provides these metrics in the static code metrics report. Static analysis of the generated code is performed only on the source code without executing the program. Information in the report includes metrics on files, global variables, and functions. For example, the **Global Variables** table includes information for each global variable: size, number of reads and writes, and number of reads and writes in a function.

2. Global Variables [hide]

Global variables defined in the generated code.

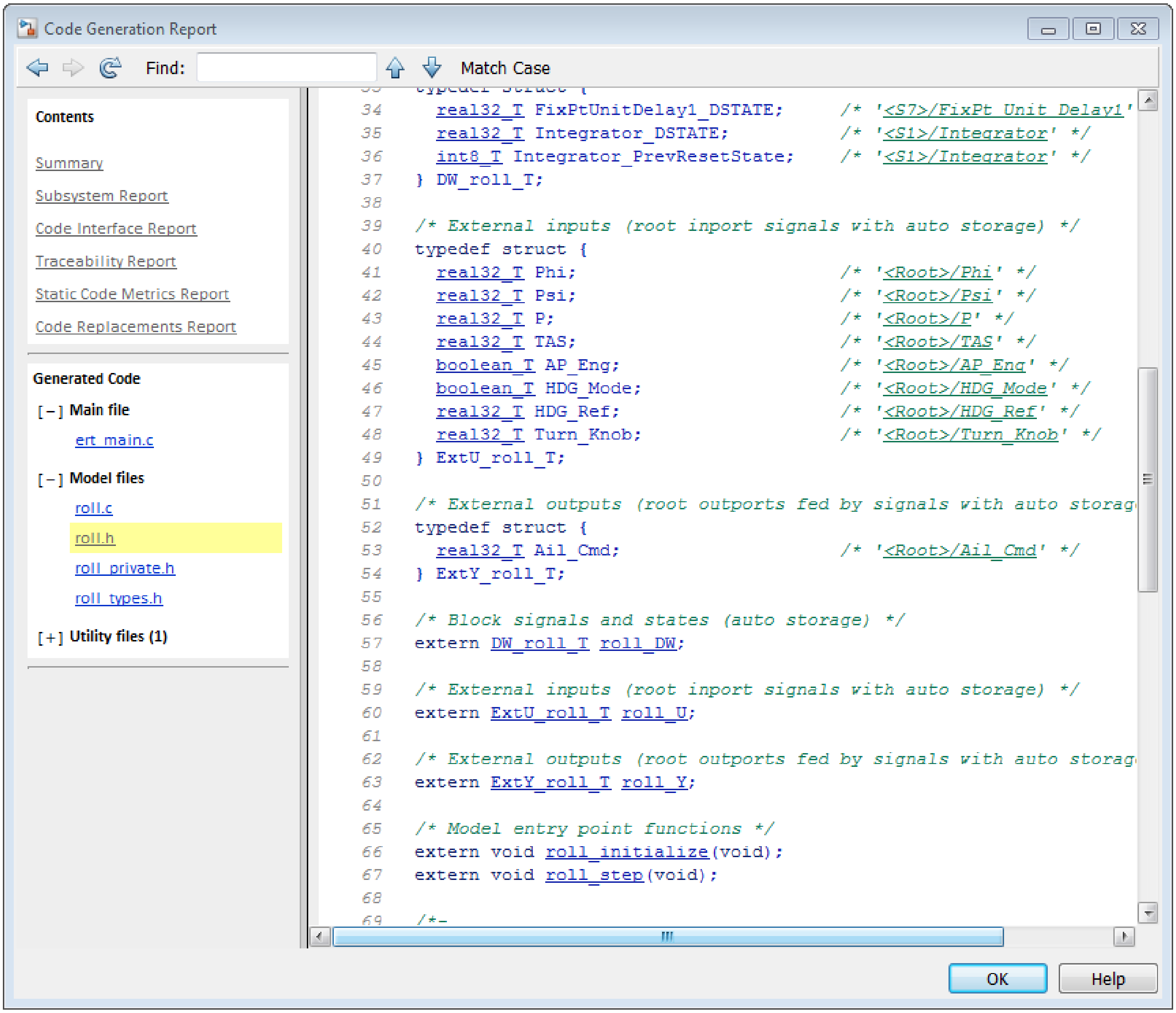
Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] roll_U	26	14	14
[+] roll_DW	9	17	17
[+] roll_Y	4	4	4
Total	**39**	**35**	

Code Replacements Report

You can use the code replacements report to determine which code replacement library (CRL) functions you use in the generated code. The report includes traceability from each replacement instance back to the block that triggered the replacement. For this model, no code replacement library is specified.

Generated Code

You can view the generated code source files in the code generation report. Click the file names in the left navigation pane. The generated `model.c` file `roll.c` contains the algorithm code, including the ODE solver code. The model data and entry point functions are accessible to a caller by including `roll.h`. In the left navigation pane, click `roll.h` to view the `extern` declarations for block outputs, continuous states, model output, entry points, and timing data.



Trace Between Code and Model

To verify the generated code, you can specify that the code generator produce hyperlinks in the source code in the code generation report for your model. The hyperlinks trace to the corresponding element in the model. For this example, `roll` is set up to include traceability. To enable traceability and generate hyperlinks the following parameters must be selected:

- On the **All Parameters** tab:
  - “Code-to-model”
  - “Model-to-code”
- On the **Code Generation** > **Comments** pane:
  - “Include comments”
  - “Simulink block / Stateflow object comments”

Trace from Model to Code

To trace from `roll` to the code generation report, in the Simulink Editor, right-click the `HeadingMode` subsystem. From the menu list, select **C/C++ code > Navigate to C/C++ code**. In the code generation report, the source code for `HeadingMode` is highlighted.

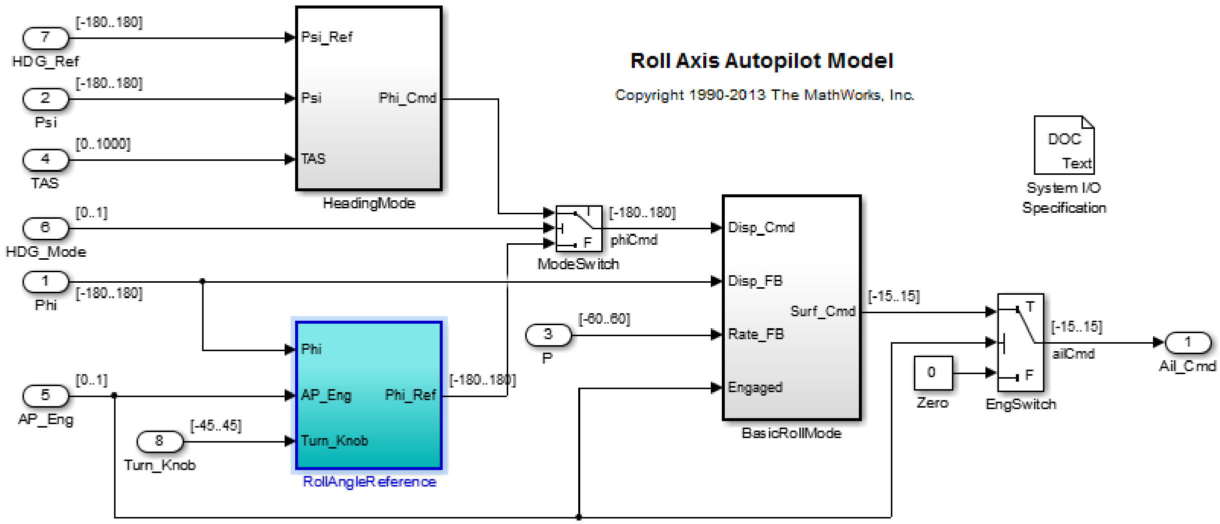
```
if (roll_U.HDG_Mode) {
  /* Outputs for Atomic SubSystem: '<Root>/HeadingMode' */
  rtb_phiCmd = (roll_U.HDG_Ref - roll_U.Psi) * 0.015F * roll_U.TAS;

  /* End of Outputs for SubSystem: '<Root>/HeadingMode' */
}
```

Trace from Code to Model

To trace from the code generation report to the model, in the left navigation pane of the code generation report, select `roll.c`. Comments in the code contain underlined text that are hyperlinks to the model. For example, when you click the hyperlink for `RollAngleReference`:

```
/* Outputs for Atomic SubSystem: '<Root>/RollAngleReference' */
/* UnitDelay: '<S7>/FixPt Unit Delay1' */
FixPtUnitDelay1_DSTATE = roll_DW.FixPtUnitDelay1_DSTATE;
```

the `RollAngleReference` subsystem is highlighted in the Simulink Editor.



After reviewing the reports and analyzing the generated code, you can change the appearance of the generated code according to defined style standards. To change the generation of comments, identifiers, and code style, see the next example, “Customize Code Appearance” on page 3-21.

Customize Code Appearance

In this section...

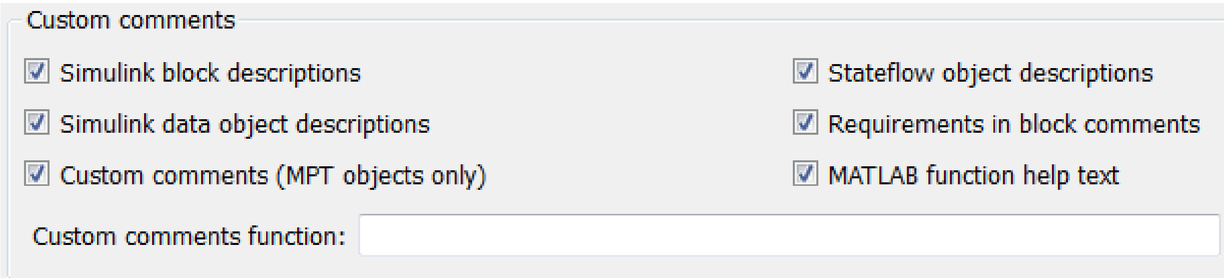
Modifying the code appearance helps you to adhere to your coding standards and enhance the readability of the code for code reviews. You can change the appearance of the generated code by specifying comment style, customizing identifier names, and choosing from several code style parameters. This example uses the configured model `roll` from the example, "Generate and Analyze C Code" on page 3-11. For this example, open `rtwdemo_roll_codegen` and save it to a local folder as `roll.slx`.

Comments

To customize the appearance of comments in the generated code for model `roll`, open the Configuration Parameters dialog box and select the **Code Generation > Comments** pane.

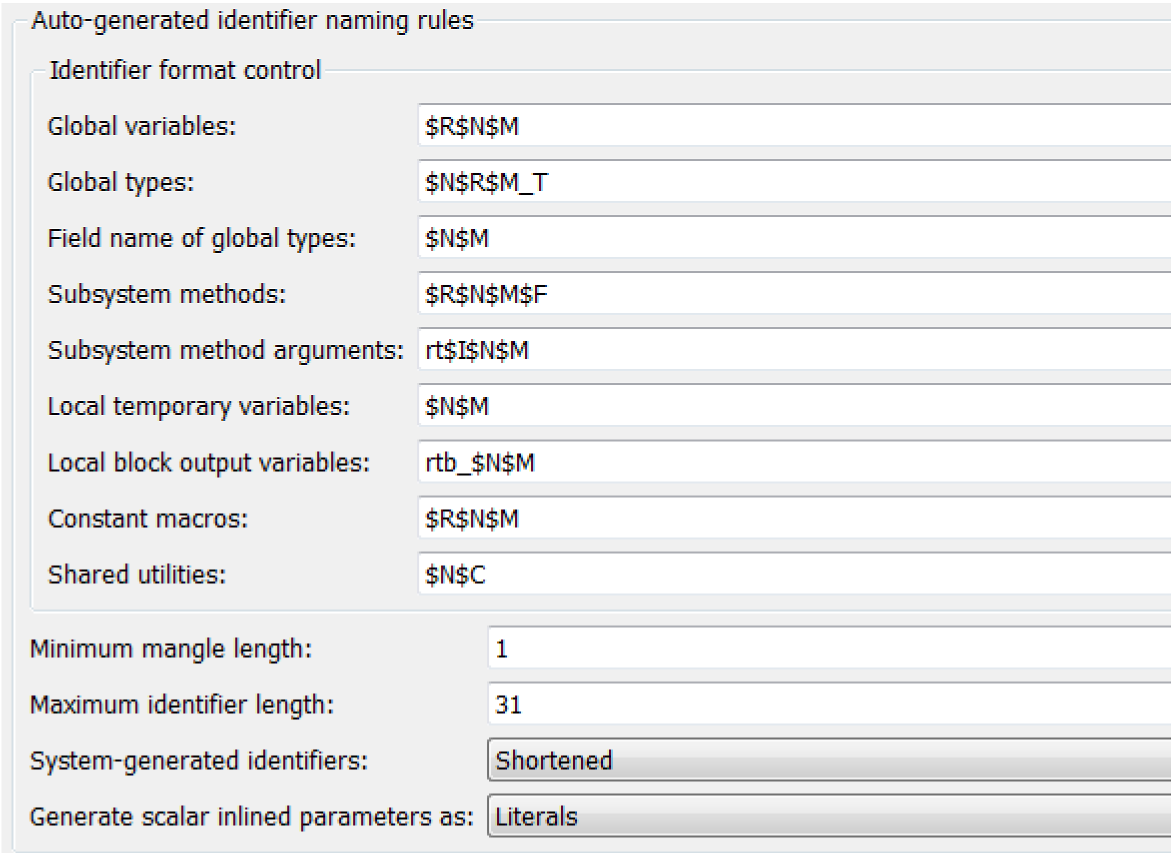
In the model `roll`, "Include comments" is selected to include comments in the generated code and enable the other comment parameters. If you configured your model for traceability, enabling comments provides traceability hyperlinks in the generate code comments.

The **Custom comments** group of parameters provides additional options for controlling specific comments for model elements.



Identifiers

To customize the appearance of identifiers in the generated code, in the Configuration Parameters dialog box, select the **Code Generation > Symbols** pane. The **Auto-generated identifier naming rules** group of parameters allows you to include predefined tokens to customize the generated identifier names. In the model `roll`, the tokens specified are the default values.



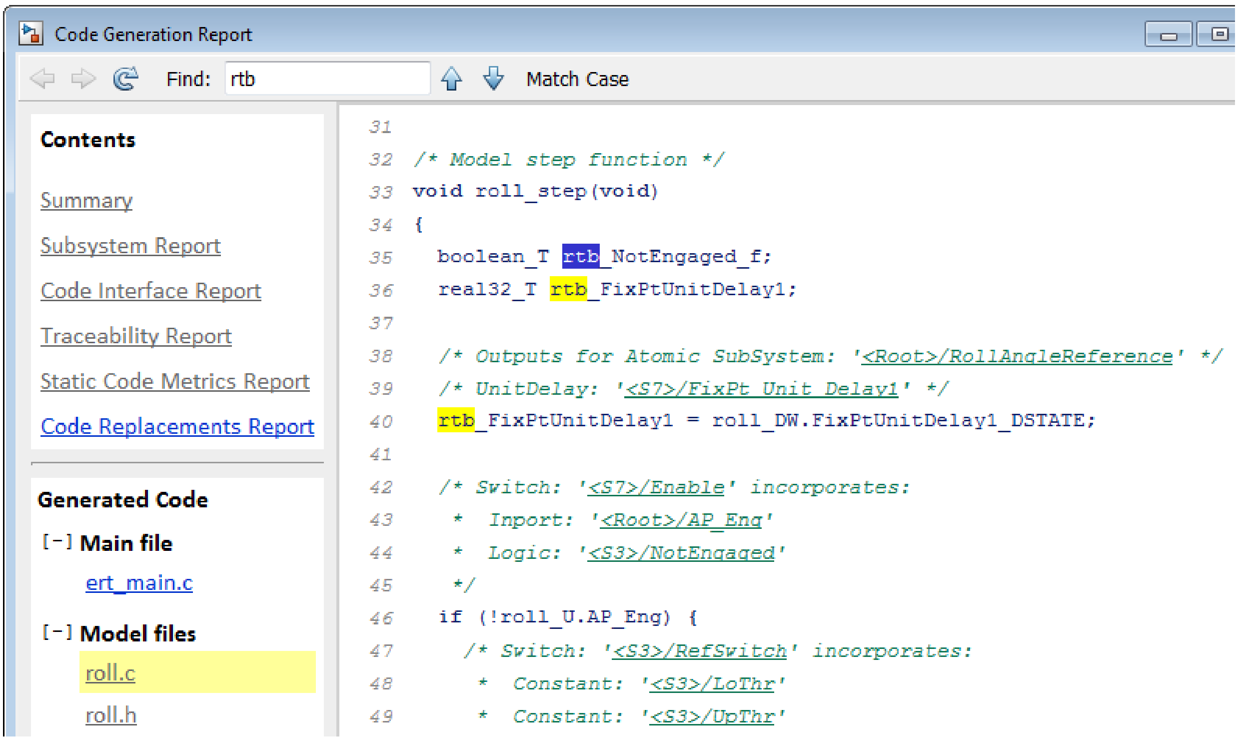
Common tokens to include are:

- `$M`: The `$M` token is a name mangling scheme to avoid naming collisions. The position of the `$M` token determines the position of the name mangling text in the generated identifier. For most of the variables that you customize, this token is required in the specification. The **Minimum mangle length** parameter determines the size of the mangling text.
- `$N`: This token includes the name of a model object (block, signal or signal object, state, parameter, shared utility function, or parameter object) for which the identifier is being generated.
- `$R`: This token inserts the root model name into the identifier. When you use referenced models, this token is required.

For example, in the `roll` model, the **Local block output variables** is specified with an `rtb` prefix.

1 If the code generation report is not open, generate code for the `roll` model.
2 In the HTML code generation report, in the left navigation pane, select `roll.c`.
3 At the top of the window, in the **Find** box, type `rtb` and press **Enter**.

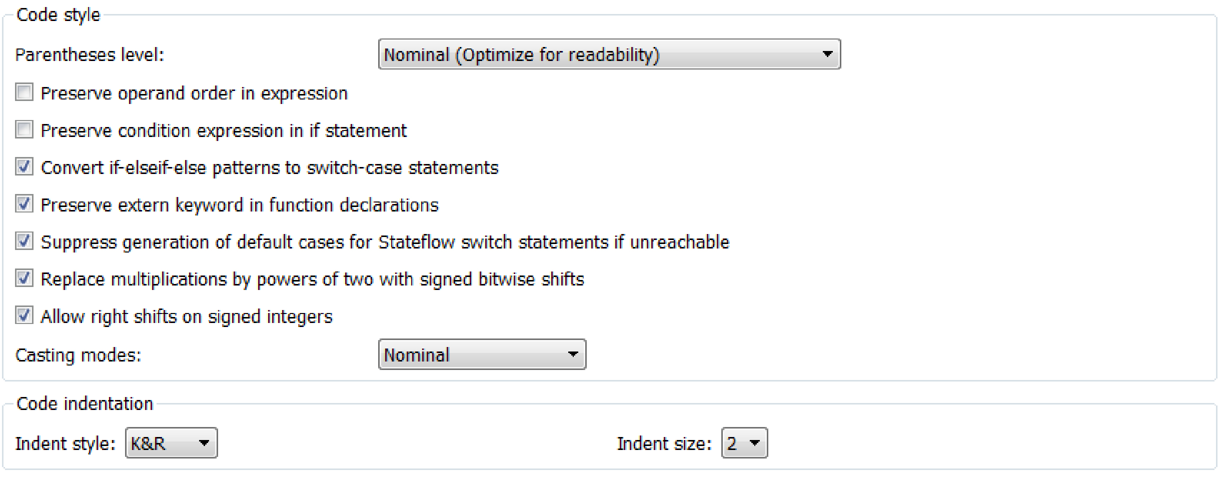
   Variables beginning with `rtb` are highlighted in the report.



4 To navigate between instances, use the up and down arrows in the code generation report.

Code Style

To customize the appearance of the generated code, in the Configuration Parameters dialog box, click the **Code Generation > Code Style** pane.



The parameters allow you to control the following code styles:

- Level of parenthesization
- Order of operands in expressions
- Empty primary condition expressions in `if` statements
- Whether to generate code for `if-elseif-else` decision logic as `switch-case` statements
- Whether to include the `extern` keyword in function declarations
- Whether to generate default cases for `switch-case` statements in the code for Stateflow charts
- How to specify casting for variable data types in the generated code
- Code indentation

With Embedded Coder, you can specify how the code generator produces function interfaces and packages the generated files. For more information, see the next example, “Customize Function Interface and File Packaging” on page 3-26.

Customize Function Interface and File Packaging

In this section...

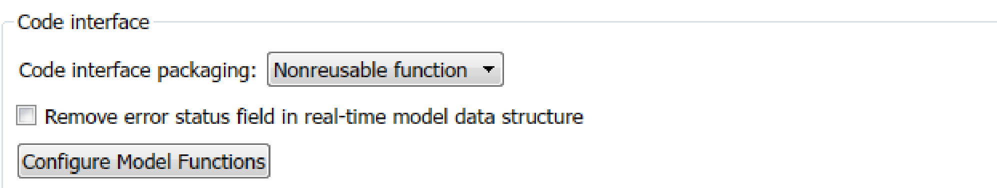
With Embedded Coder, you can specify the function interfaces for models and atomic subsystems in the generated code. You can also specify how the code is placed into files and folders. This example uses the configured model `roll` from the example, "Generate and Analyze C Code" on page 3-11. For this example, open `rtwdemo_roll_codegen` and save it to a local folder as `roll.slx`.

Model Interface

You can configure the interface of the code for the model in the Configuration Parameters dialog box, on the **Code Generation > Interface** pane. By default, the model's entry points are implemented as void/void functions. Model `roll` is set to generate nonreusable code with a minimal function interface.

When configuring the model interface, you can choose whether to produce reusable or nonreusable code. Reusable code consists of reentrant functions that can be called with different data sets. In general, nonreusable code executes more efficiently in an embedded system because nonreentrant functions can avoid pointer dereference.

For this example, `roll` is configured to generate a nonreusable function interface. Verify that **Code interface packaging** is set to `Nonreusable function`.



For `roll`, the interface settings direct the code generator to create two entry point functions to initialize and step through algorithm code.

Configuration Parameter	Description
Remove error status field in real-time model data structure is selected	Remove the error status field in the real-time model data structure
All Parameters > **Single output/update function** is selected	Produce a single entry point function to step the model
All Parameters > **Terminate function required** is cleared	Eliminate the entry point function to terminate the model
All Parameters > **Combine signal/state structures** is selected	Produce a single data structure for the model's global data

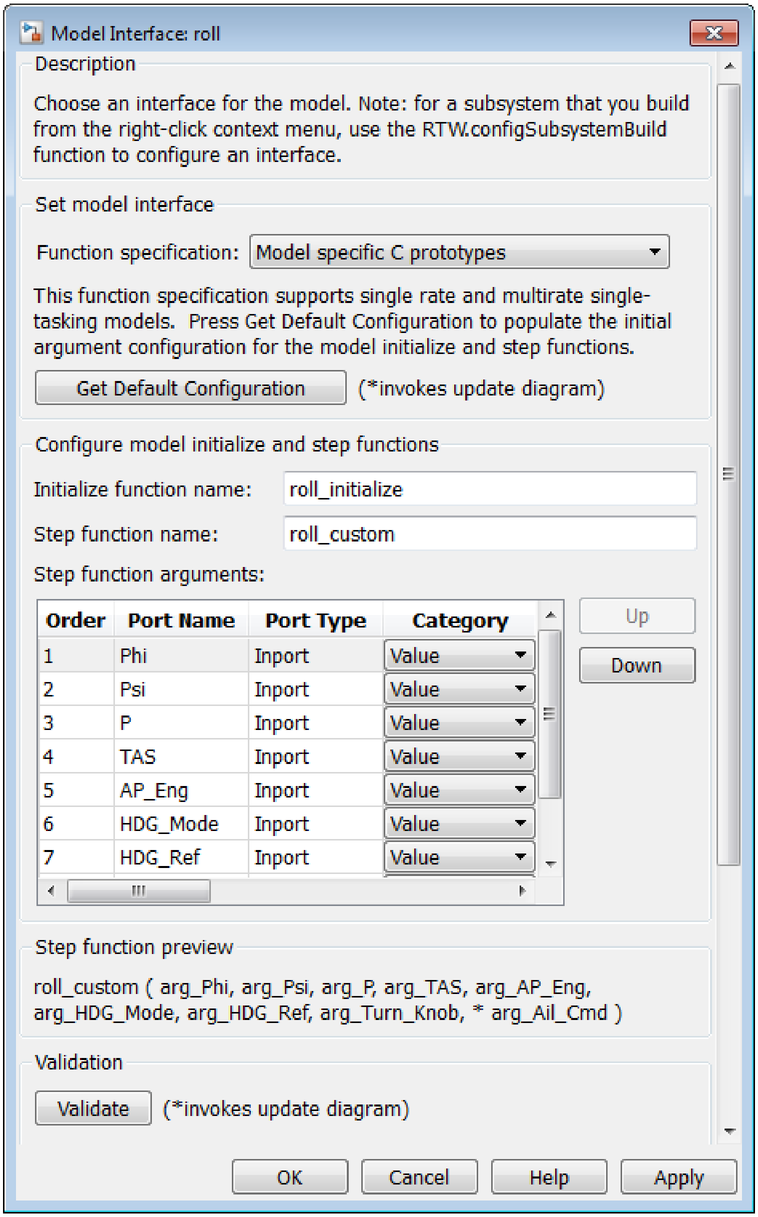
The code generator produces this code based on the following assumptions:

- The solver setting indicates that the step function executes the code at `40 Hz`.
- The initialize function is called once prior to executing the step function.

Configure Model Initialize and Step Functions

You can specify the names of the model initialize and step function, and the function prototype of the model step function.

1. To open the Model Interface dialog box, on the **Interface** pane, click **Configure Model Functions**.
2. In the Model Interface dialog box, specify **Function specification** as `Model specific C prototypes`.
3. Click **Get Default Configuration**. The Model Interface dialog box expands to display the **Configure model initialize and step functions** parameters.


Model Interface: roll
Description
Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.
Set model interface
Function specification: Model specific C prototypes
This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.
Get Default Configuration
(*invokes update diagram)
Configure model initialize and step functions
Initialize function name: roll_initialize
Step function name: roll_custom
Step function arguments:
Order Port Name Port Type Category
1 Phi Inport Value
2 Psi Inport Value
3 P Inport Value
4 TAS Inport Value
5 AP_Eng Inport Value
6 HDG_Mode Inport Value
7 HDG_Ref Inport Value
Up
Down
Step function preview
roll_custom (arg_Phi, arg_Psi, arg_P, arg_TAS, arg_AP_Eng, arg_HDG_Mode, arg_HDG_Ref, arg_Turn_Knob, * arg_Ail_Cmd)
Validation
Validate
(*invokes update diagram)
OK
Cancel
Help
Apply

4 Change **Initialize function name** to `roll_init`.

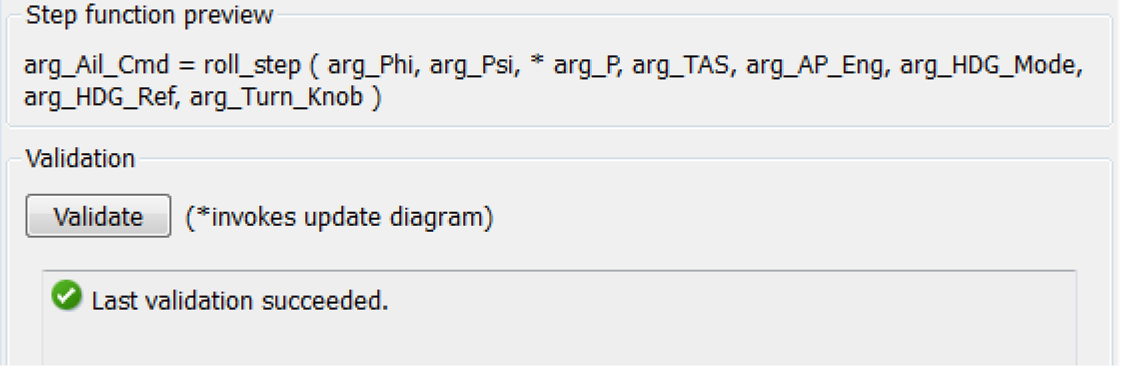
5 Change **Step function name** to `roll_step`.

6 In the **Step function arguments** table, verify that the order of the first two arguments is `Phi` and then `Psi`. You can use the **Up** and **Down** buttons to reorder the arguments.

7 Modify the third return argument, `P`, to a `Pointer` in the **Category** column.

8 Change the Outport, `Ail_Cmd`, to return by `Value` in the **Category** column.

9 Click **Validate**, so that consistency checking is performed on the interface specification.



10 Click **Apply** and **OK**.

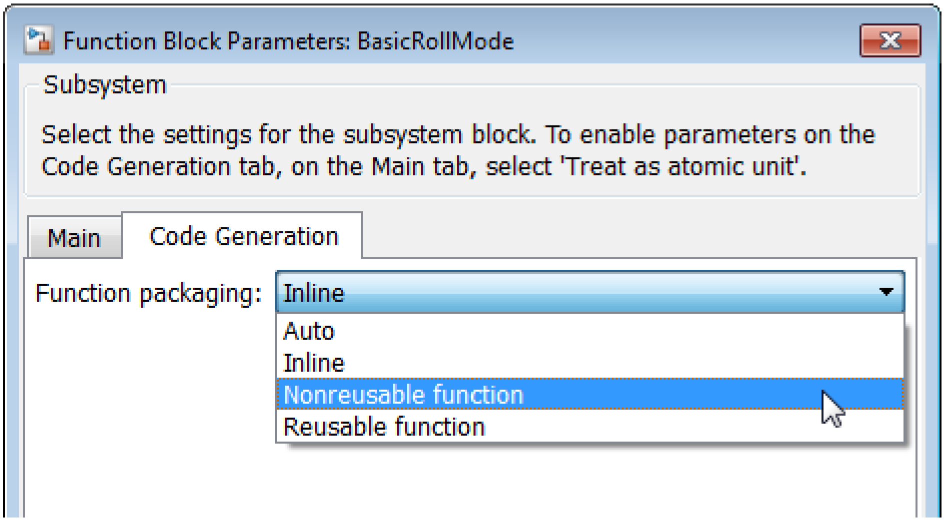
11 In the Simulink Editor, press **Ctrl+B** to generate the code. In the code generation report, see that the code matches the specification.

Subsystem Interface

You can configure how the software implements atomic subsystems in the generated code. In the Simulink editor, locate the subsystem, `BasicRollMode`.

1 Right-click `BasicRollMode` and select `Block Parameters (Subsystem)`.

2 In the Block Parameters dialog box, click the **Code Generation** tab. The **Function packaging** parameter `Auto` option instructs the software to use its heuristic to implement the system efficiently, based on its usage in the model. Otherwise, you can specify the function implementation based on criteria that you are using for execution speed and memory utilization.



3 Specify the **Function packaging** parameter as `Nonreusable function`.

4 To use the block name as the function name, in the dialog box, specify **Function name options** as `Use subsystem name`.

5 To place the code for the function in a separate file and use the function name, specify **File name options** as `Use function name`.

6 To pass the subsystem inputs and outputs as arguments to the function, specify **Function interface** as `Allow arguments`.

7 Click **Apply** and **OK**.

8 Press **Ctrl+B** to generate the code. Verify that the subsystem code is in `roll_BasicRollMode.c` and its declaration is in `roll_BasicRollMode.h`.

Customize File Packaging

In the previous section, using the Subsystem block parameters, you specified file packaging of the generated code at the subsystem level. You can also configure file packaging at the model level.

In the Configuration Parameters dialog box, on the **Code Generation > Code Placement** pane, you can specify the **File packaging format** parameter with the following options: `Modular`, `Compact (with separate data file)`, and `Compact`. This parameter instructs the code generator to modularize the code into many files or compact the generated code into a few files. If your model contains referenced models, you can specify a different file packaging format for each referenced model.

For your model `roll`, the **File packaging format** is set to `Modular`. Therefore, the code generator creates the following files:

- `roll.c`
- `roll.h`
- `roll_private.h`
- `roll_types.h`
- Subsystem files: `roll_BasicRollMode.c` and `roll_BasicRollMode.c`

This example showed how to configure the function interfaces for a model and a subsystem. The example showed how to configure your model to modularize the generated code into different file packaging formats. The next example shows how to set up your model to specify how data appears in the generated code. For more information, see “Define Data in the Generated Code” on page 3-32.

Define Data in the Generated Code

To access and organize the data (signals, parameters, and states) that a model contains, you can apply *storage classes* and custom storage classes to individual data items. You can specify:

- Storage in memory, for example, as a global structure or bitfield.
- Placement of definitions and declarations in generated source and header files.

To apply storage classes to signals and data stores, use the Model Data Editor. To apply storage classes to parameters, use the Model Data Editor to create `Simulink.Parameter` objects.

In the example "Generate and Analyze C Code" on page 3-11, you generated code from the model `rtwdemo_roll_codegen`. In this example, you configure data representation for the same model. Save the model as `roll.slx` in a local folder.

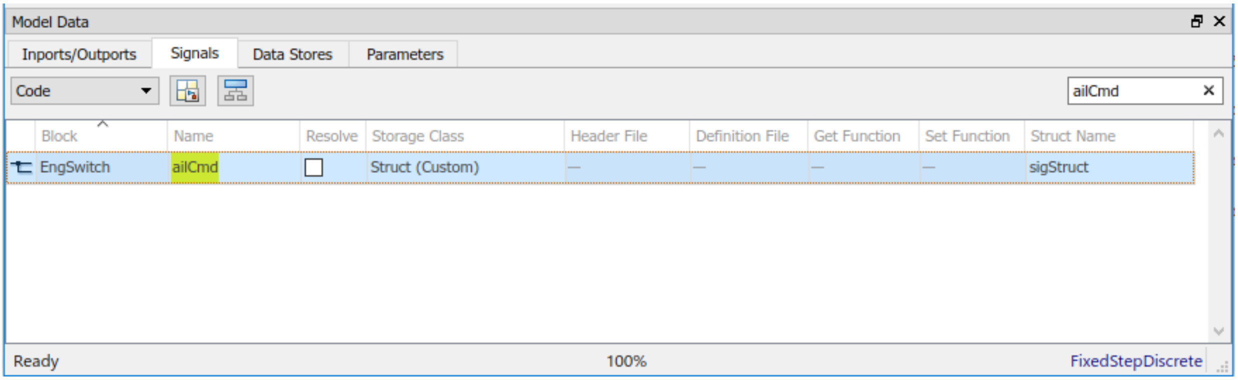
Configure Signal Data

The model has two named signals, `ailCmd` and `phiCmd`. Configure the generated code to store these signals in a structure.

1. In the model, select **View** > **Model Data**.
2. In the Model Data Editor, select the **Signals** tab.
3. Set the **Change View** drop-down list to `Code`.
4. Use the **Filter Contents** box to search for the `ailCmd` signal.
5. Use the **Storage Class** column to apply the custom storage class `Struct`.

   The custom storage class `Struct` causes each data item to appear in the generated code as a field of a flat structure.
6. Set **Struct Name** to `sigStruct`.

   This property controls the name of the structure variable in the generated code.



7 Use the **Filter Contents** box to search for the `phiCmd` signal. Configure the signal to use the same structure variable, `sigStruct`.

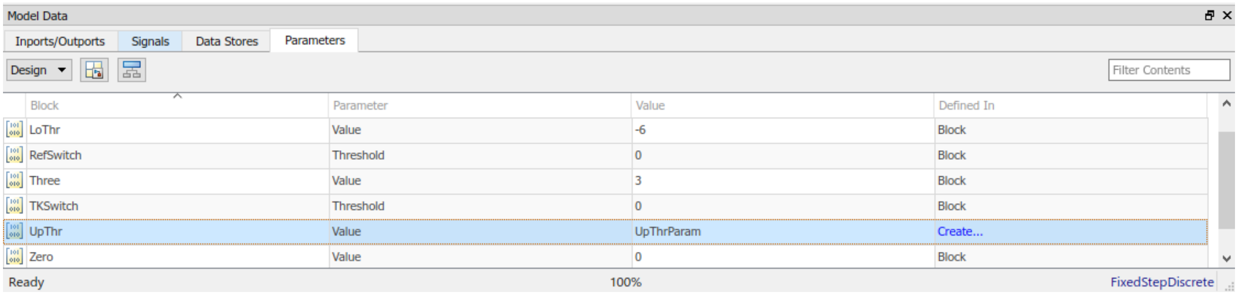
Configure Parameter Data

In the `RollAngleReference` subsystem, two Constant blocks represent upper and lower threshold values for the vehicle roll angle. Configure these threshold parameters to appear in the generated code as tunable global variables.

To apply a storage class to a block parameter, such as the **Constant value** parameter of a Constant block, you must first use a `Simulink.Parameter` object to set the parameter value. Then, you apply the storage class to the parameter object.

1 Navigate into the `RollAngleReference` subsystem.
2 In the Model Data Editor, select the **Parameters** tab.
3 In the model, click the Constant block named `UpThr`.

   In the Model Data Editor, the highlighted row represents the **Constant value** parameter of the block.
4 Set **Value** to `UpThrParam`.



5 In the **Defined In** column, click **Create**.

6 In the Create New Data dialog box, set **Value** to `Simulink.Parameter(6)`. Click **Create**.

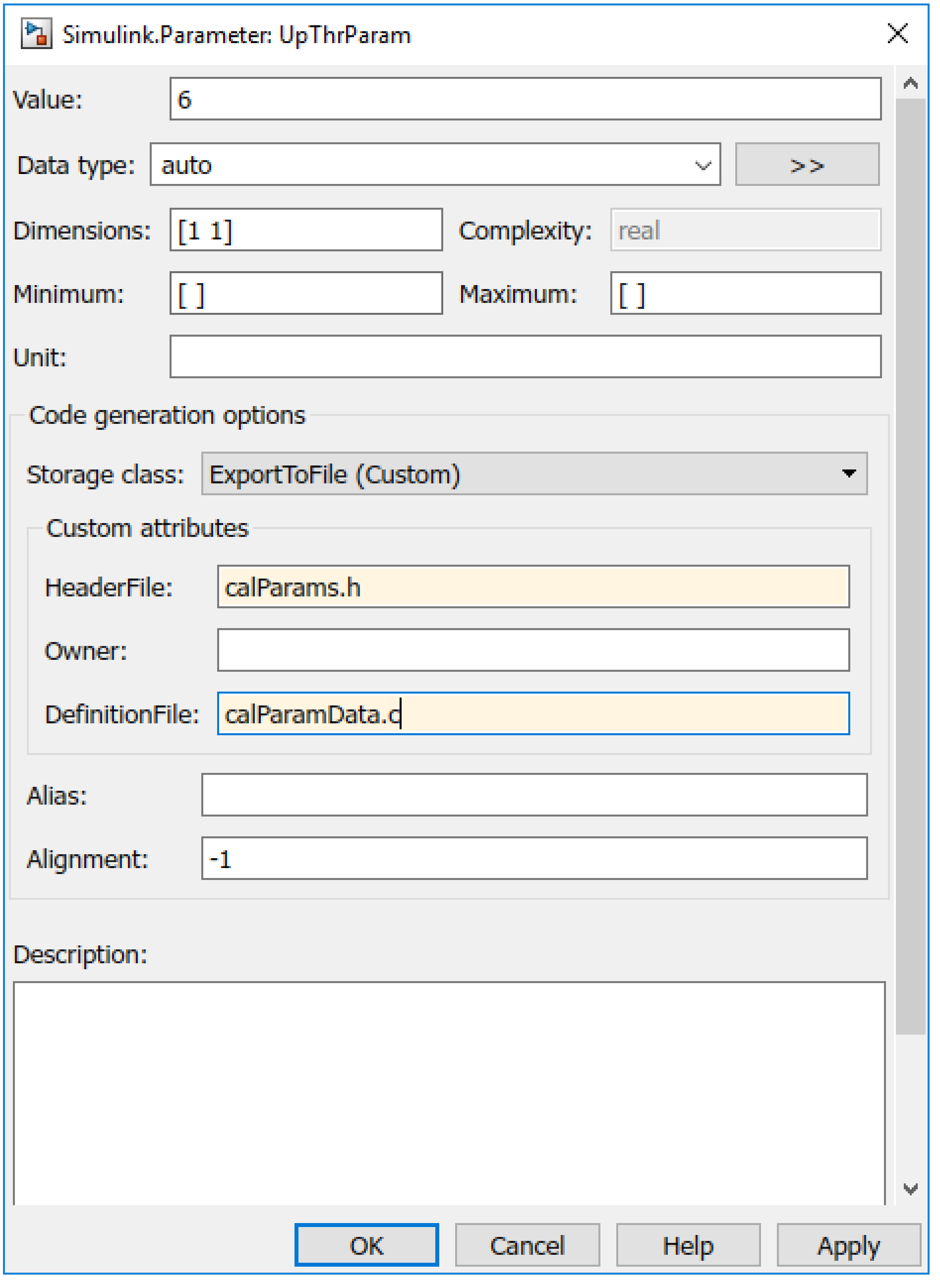
A `Simulink.Parameter` object named `UpThrParam` appears in the base workspace. The **Constant value** block parameter uses the numeric value, 6, from the parameter object.

7 In the `UpThrParam` property dialog box, set **Storage class** to `ExportToFile`.

With the custom storage class `ExportToFile`, you can control the file placement of the data definition and declaration.

8 Set **HeaderFile** to `calParams.h`.

9 Set **DefinitionFile** to `calParamData.c`.


Simulink.Parameter: UpThrParam
Value:
6
Data type:
auto
>>
Dimensions:
[1 1]
Complexity:
real
Minimum:
[ ]
Maximum:
[ ]
Unit:
Code generation options
Storage class:
ExportToFile (Custom)
Custom attributes
HeaderFile:
calParams.h
Owner:
DefinitionFile:
calParamData.c
Alias:
Alignment:
-1
Description:
OK
Cancel
Help
Apply

10 Click **OK**.
11 For the Constant block named `LoThr`, create another `Simulink.Parameter` object named `LoThrParam`, but use the value `-6` instead of `6`. Configure the object to use the same header file, `calParams.h`, and the same definition file, `calParamData.c`.

Generate and Inspect Code

1 Generate code from the model.
2 View the generated file `roll.h`. This header file defines the structure type `sigStruct_type`.

```
/* Type definition for custom storage class: Struct */
typedef struct sigStruct_tag {
  real32_T phiCmd;
  real32_T ailCmd;
} sigStruct_type;
```

The data type alias `real32_T` corresponds to the single-precision floating-point data type `float`.

The file also declares a global structure variable, `sigStruct`, of type `sigStruct_type`.

```
/* Declaration for custom storage class: Struct */
extern sigStruct_type sigStruct;
```

3 View the file `calParams.h`. The file declares the global variables `LoThrParam` and `UpThrParam`.

```
/* Declaration for custom storage class: ExportToFile */
extern real32_T LoThrParam;
extern real32_T UpThrParam;
```

4 View the file `roll.c`. This source file defines `sigStruct`.

```
/* Definition for custom storage class: Struct */
sigStruct_type sigStruct;
```

5 View the file `calParamData.c`. The file defines and initializes `LoThrParam` and `UpThrParam`.

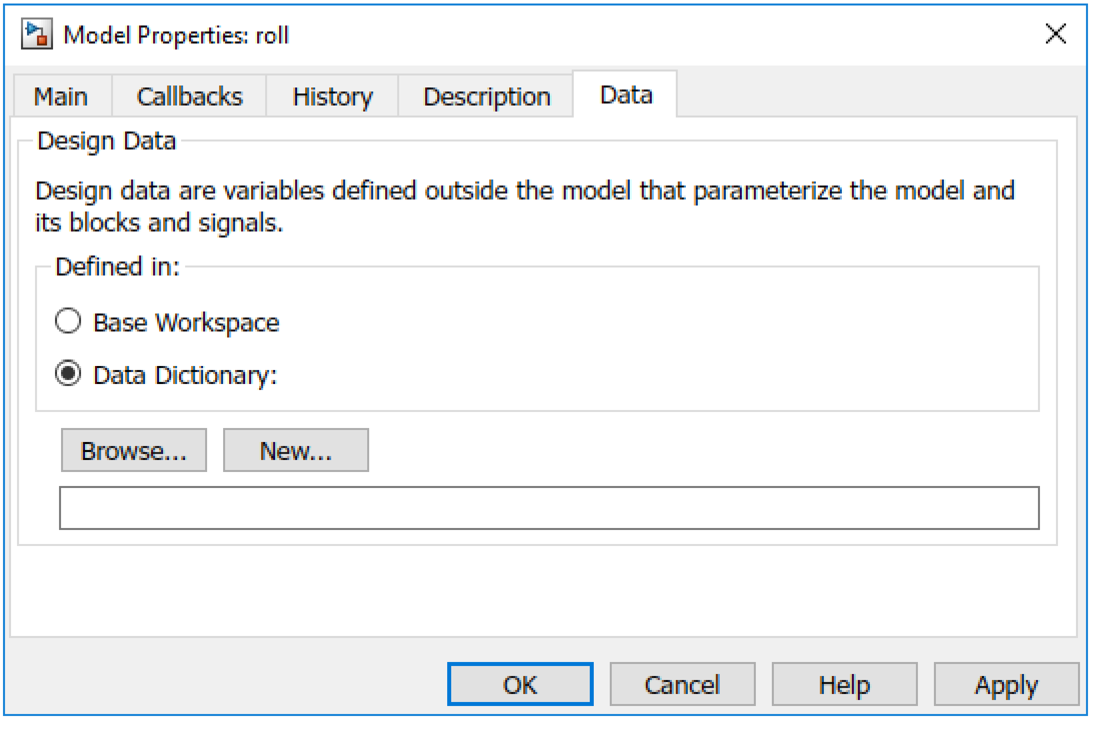
```
/* Definition for custom storage class: ExportToFile */
real32_T LoThrParam = -6.0F;
real32_T UpThrParam = 6.0F;
```

To use the data in your code, include the header files `calParams.h` and `roll.h`. To modularize the generated code and establish ownership of data, you can control the file placement of the generated definitions.

Save Data Objects in Data Dictionary

Data objects that you create in the base workspace (for example, the `Simulink.Parameter` objects) are not saved with the model. When you end your MATLAB session, these objects do not persist. To permanently store these objects, save them in a data dictionary and link the model to the dictionary.

1 In the `roll` model, select **File** > **Model Properties** > **Link to Data Dictionary**.
2 In the Model Properties dialog box, select **Data Dictionary**.



3 Click **New**.

4 In the Create a new Data Dictionary dialog box, set **File name** to `rollDict`. Click **Save**.

The data dictionary file `rollDict.sldd` appears in your current folder.

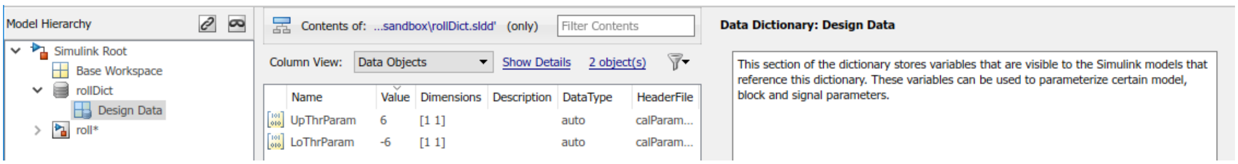
5 In the Model Properties dialog box, click **OK**.

6 Click **Yes** in response to the message about migrating base workspace data.

7 Click **Yes** in response to the message about removing imported items from the base workspace.

8 In the lower-left corner of the model, click the data dictionary badge .

The contents of the data dictionary appear in the Model Explorer.



The data dictionary `rollDict.sldd` permanently stores the parameter objects that the model uses.

This example shows how to control data representation in the generated code. The next example shows how to deploy and verify an executable program for your model. For more information, see “Deploy and Test Executable Program” on page 3-39.

Related Examples

- “Control Data Representation by Applying Custom Storage Classes”

More About

- “Data Objects”
- “Default Data Structures in the Generated Code”

Deploy and Test Executable Program

In this section...

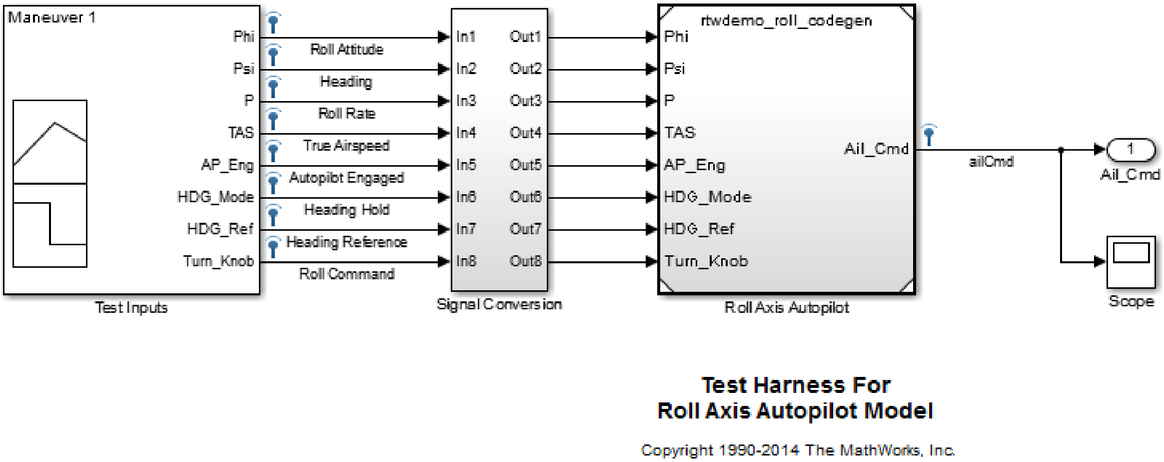
To test generated code, you can run software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. A SIL simulation compiles and runs the generated code on your development computer. A PIL simulation cross-compiles source code on your development computer. It then downloads and runs the object code on a target processor or an equivalent instruction set simulator. You can use SIL and PIL simulations to:

- Verify the numerical behavior of your code.
- Collect code coverage and execution-time metrics.
- Optimize your code.
- Progress to achieving IEC 61508, IEC 62304, ISO 26262, EN 50128, or DO-178 certification.

This example uses a test harness model to run a model in normal mode, and then in SIL mode. The Simulation Data Inspector logs and compares the results. The comparison determines whether the model and the generated code are numerically equivalent.

Test Harness Model

One method for testing generated model code is to use a test harness model that references the model-under-test through a Model block. You can generate a test harness model with Simulink Verification and Validation software. You can easily switch the Model block between the normal, SIL, or PIL simulation modes. This example uses a test harness model, `rtwdemo_roll_harness`, which generates test inputs for the referenced model, `rtwdemo_roll_codegen`.

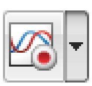


1 Open the example models `rtwdemo_roll_harness` and `rtwdemo_roll_codegen`.
2 In a local folder, save `rtwdemo_roll_harness` and `rtwdemo_roll_codegen`.
3 Open the Configuration Parameters dialog boxes for `rtwdemo_roll_harness` and `rtwdemo_roll_codegen`.
4 For `rtwdemo_roll_harness`, on the **Code Generation** pane, verify that the **Generate code only** check box is cleared.
5 For both models, on the **Hardware Implementation** pane, expand **Device details**. If **Support long long** is not already selected, select the check box.
6 If you make changes, click **Apply** and **OK**. Then save the models.

Simulate the Model in Normal Mode

Run the harness model in normal mode and capture the results in the Simulation Data Inspector.

1 To send logged data from the workspace to the Simulation Data Inspector for model `roll_harness`, on the Simulink Editor toolbar, click the **Simulation Data Inspector** button arrow and select **Send Logged Workspace Data to Data Inspector**.



2 Right-click the Model block, `Roll Axis Autopilot`. From the context menu, select **Block Parameters (ModelReference)**.

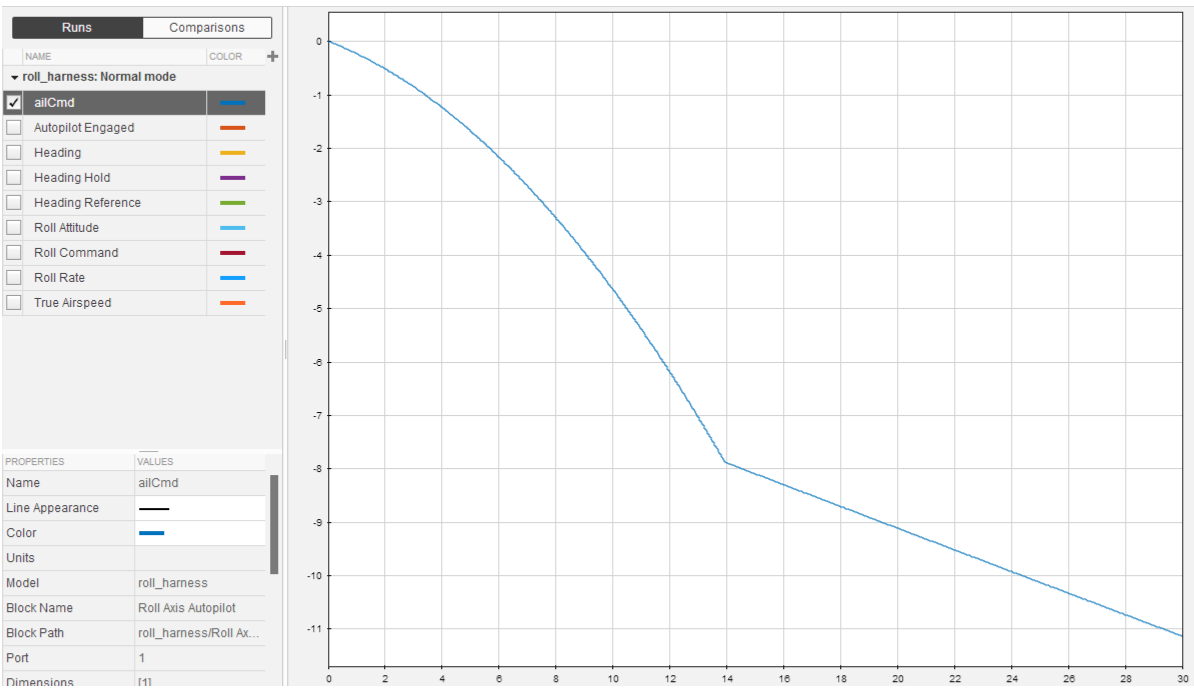
3 In the Block Parameters dialog box, for **Simulation mode**, verify that the `Normal` option is selected. Click **OK**.

4 Simulate `rtwdemo_roll_harness`.

5 When the simulation is done, view the simulation results in the Simulation Data Inspector. If the Simulation Data Inspector is not already open, in the Simulink Editor, click the **Simulation Data Inspector** button.

6 For the new run, double-click the run name field and rename the run: `roll_harness: Normal mode`.

7 Expand the run and select `ailCmd` to plot the signal.



Simulate the Model in SIL Mode

The SIL simulation generates, compiles, and executes code on your development computer. The Simulation Data Inspector logs results.

1 In the rtwdemo_roll_harness model window, right-click the Roll Axis Autopilot model block and select **Block Parameters (ModelReference)**.

2 In the Block Parameters dialog box, specify **Simulation mode** as Software-in-the-loop (SIL). Click **Apply** and **OK**.

3 On the Simulink Editor toolbar, verify that logged data recording is on.

4 Simulate the rtwdemo_roll_harness model.

5 In the Simulation Data Inspector, double-click the run name field and rename the new run as roll_harness: SIL mode.

6 Expand the run and select ailCmd to plot the signal.

Compare Simulation Results

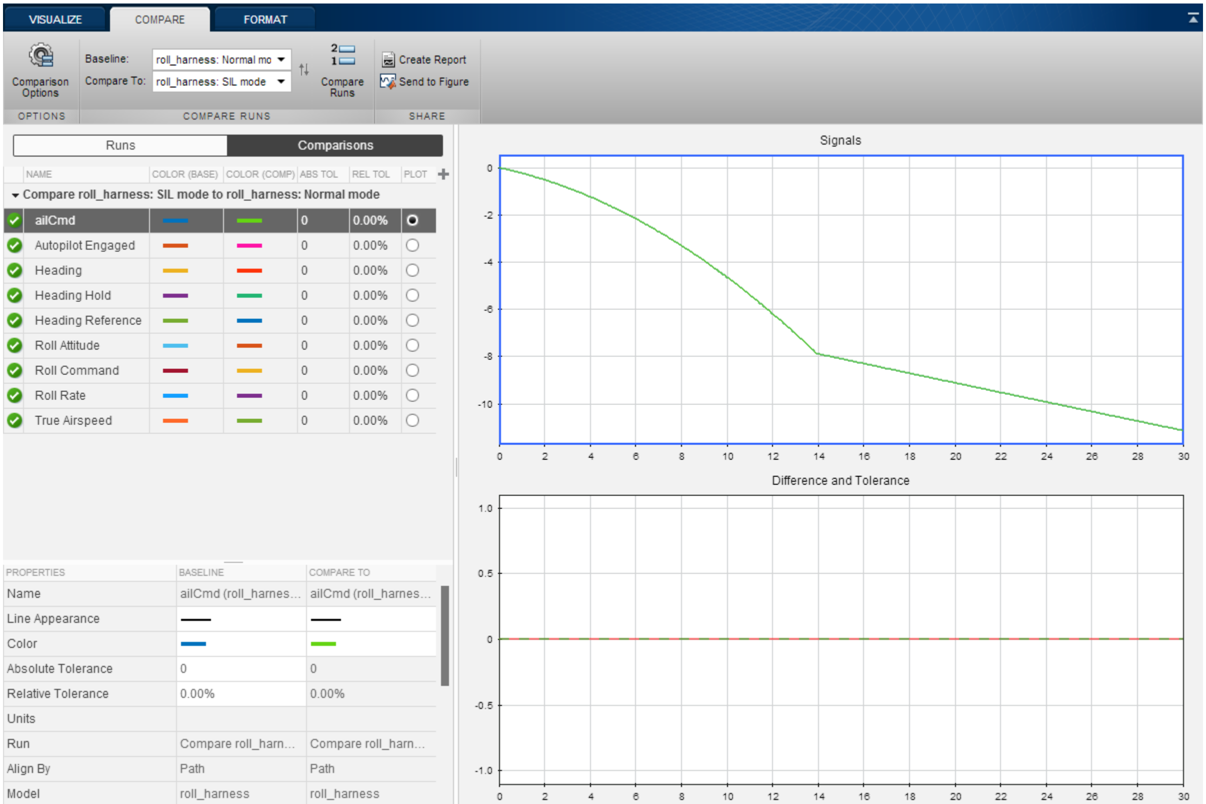
In the Simulation Data Inspector:

1 Click the **Compare** tab.

2 In the **Baseline** field, select roll_harness: Normal mode.

3 In the **Compare To** field, select roll_harness: SIL mode.

4 Click **Compare**.



If normal mode and SIL mode results differ, it is important to investigate and understand the differences. For example, differences can arise from:

- Implementation changes (converting an interpreted model to an executable implementation)
- Bugs

You must understand all differences to verify that the generated code does not have bugs.

Improve Code Performance

To generate code that meets code performance requirements, you can design and configure your model to support code generator optimizations. Use the Code Generation Advisor for guidance on configuring your model to meet code generation objectives.

After testing your model and code, you can fine-tune your model configuration to improve the code performance. For a specific performance goal, Embedded Coder provides model configuration parameters on the **Optimization** panes to help you improve memory usage and reduce execution time.

When choosing optimization parameters for your model, consider the trade-offs of one optimization over another optimization. Improving one area of performance can sacrifice another area of performance. For example, reducing memory usage can sacrifice execution efficiency.

More Information About Code Generation in Model-Based Design

This table provides links to additional information for generating, executing, and verifying production code for your model.

To	See
Quickly generate readable, efficient code from your model	"Generate Code with the Quick Start Tool"
Consider model design and configuration for code generation	"Model Architecture and Design"
Generate code variants by using macros for preprocessor compilation from models	"Variant Systems"
Achieve code reuse	"Subsystems" and "Referenced Models"
Define and control model data	"Data Representation"
Control generation of function and class interfaces	"Function and Class Interfaces"
Control naming and partitioning of data and code across generated source files	"File Packaging"
Select and configure the target environment for your application	"Target Environment Configuration"
Configure model for code generation objectives, such as efficiency or safety	"Configure Model for Code Generation Objectives Using Code Generation Advisor"
Configure parameters for specifying identifier names, code comments, style, and templates	"Code Appearance"

To	See
Export component XML description and C code for AUTOSAR run-time environment	"AUTOSAR Code Generation"
Deployment of standalone programs to target hardware	"Deploy Generated Standalone Executables To Target Hardware"
Choose and apply integration paths and methods	"External Code Integration"
Improve code performance, such as memory usage and execution speed	"Performance"
Collect code coverage metrics for generated code during SIL or PIL simulation	"Code Coverage"
SIL and PIL testing	"Software-in-the-Loop Simulation" and "Processor-in-the-Loop Simulation"
View and analyze execution profiles of code sections	"Code Execution Profiling"

A

Installing and Using IDE

Installing Eclipse IDE and Cygwin Debugger

In this section...

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.

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 spaces.

Integrating and Testing Code with Eclipse IDE

In this section...

About 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. There are many other software packages and tools that can work with code generation software to perform similar tasks.

“Installing 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.

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

Define 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**.

Configure 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**.

Start 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 "Set the Cygwin Path" on page A-6.
4 Navigate to the `example_main.c` file and click **Open**. Your program opens in the debugger software.

Set the Cygwin Path

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

To provide the 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**.

Debugger Actions and Commands

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**