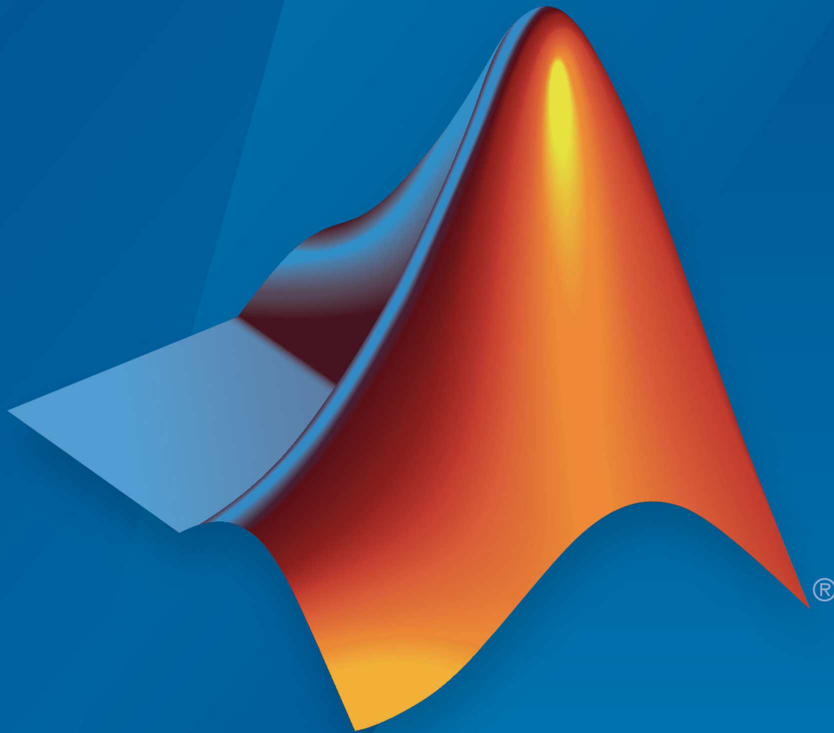
Embedded Coder®

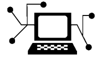
Getting Started Guide



MATLAB® & SIMULINK®

R2019a

How to Contact MathWorks



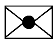
Latest news: www.mathworks.com

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Embedded Coder® Getting Started Guide

© COPYRIGHT 2011–2019 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)
March 2017	Online only	Revised for Version 6.12 (Release 2017a)
September 2017	Online only	Revised for Version 6.13 (Release 2017b)
March 2018	Online only	Revised for Version 7.0 (Release 2018a)
September 2018	Online only	Revised for Version 7.1 (Release 2018b)
March 2019	Online only	Revised for Version 7.2 (Release 2019a)

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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

Embedded Coder Product Description **1-2**
Key Features **1-2**

Code Generation Technology **1-3**

Code Generation Workflows with Embedded Coder **1-4**
Code Generation from MATLAB Code **1-5**
Code Generation from Simulink Models **1-6**

Validation and Verification for System Development **1-8**
V-Model for System Development **1-8**
Types of Simulation and Prototyping in the V-Model **1-10**
Types of In-the-Loop Testing in the V-Model **1-11**
Code Generation Goal Summary **1-12**

Target Environments and Applications **1-30**
About Target Environments **1-30**
Types of Target Environments **1-30**
Applications of Supported Target Environments **1-32**

2 MATLAB Tutorials

Embedded Coder Capabilities for Code Generation from MATLAB Code **2-2**

Controlling C Code Style **2-9**
About This Tutorial **2-9**
Copy File to a Local Working Folder **2-10**

Open the MATLAB Coder App 2-10
Specify Source Files 2-10
Define Input Types 2-11
Check for Run-Time Issues 2-11
Configure Code Generation Parameters 2-12
Generate C Code 2-12
View the Generated Code 2-12
Finish the Workflow 2-13
Key Points to Remember 2-14

Include Comments in Generated C/C++ Code **2-15**
About This Tutorial 2-15
Creating the MATLAB Source File 2-15
Configuring Build Parameters 2-16
Generating the C Code 2-16
Viewing the Generated C Code 2-17
Tracing the Generated Code to the MATLAB Code 2-17

3 Simulink Code Generation Tutorials

Generate C Code from Simulink Models **3-2**
Prerequisites 3-2
Example Models 3-2

Generate Code by Using Embedded Coder Quick Start **3-6**

Configure Data Interface **3-11**
Configure Default Code Generation for Data 3-11
Override Default Settings for Individual Data Elements 3-15

Configure a Model Parameter as a Global Variable for Tuning During Run Time **3-18**

Compare Model Simulation and Generated Code Results **3-21**
Inspect and Configure Test Harness Model 3-21
Simulate the Model in Normal Mode 3-22
Simulate the Model in SIL Mode 3-24
Compare Simulation Results 3-24

Deploy the Generated Code **3-26**
Example Main Program **3-26**
Relocate Generated Code Files **3-26**
Share and Archive Code Generation Report **3-27**
Explore Other Options **3-28**

Product Overview

- "Embedded Coder Product Description" on page 1-2
- "Code Generation Technology" on page 1-3
- "Code Generation Workflows with Embedded Coder" on page 1-4
- "Validation and Verification for System Development" on page 1-8
- "Target Environments and Applications" on page 1-30

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 embedded processors used in mass production. It extends MATLAB® Coder™ and Simulink® Coder with advanced optimizations for precise control of the generated functions, files, and data. These optimizations improve code efficiency and facilitate integration with legacy code, data types, and calibration parameters. You can incorporate a third-party development tool to build an executable for turnkey deployment on your embedded system or rapid prototyping board.

Embedded Coder offers built-in support for AUTOSAR, MISRA C®, and ASAP2 software standards. It also provides traceability reports, code documentation, and automated software verification to support DO178, IEC 61508, and ISO 26262 software development. Embedded Coder code is portable, and can be compiled and executed on any processor. In addition, it offers support packages with advanced optimizations and device drivers for specific hardware.

Key Features

- Optimization and code configuration options extending MATLAB Coder and Simulink Coder
- Storage class, type, and alias definition using data dictionaries
- 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
- Standards support, including ASAP2, AUTOSAR, DO-178, IEC 61508, ISO 26262, and MISRA C (with Simulink)
- Advanced code optimizations and device drivers for specific hardware, including ARM®, Intel®, NXP™, STMicroelectronics®, and Texas Instruments™

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.

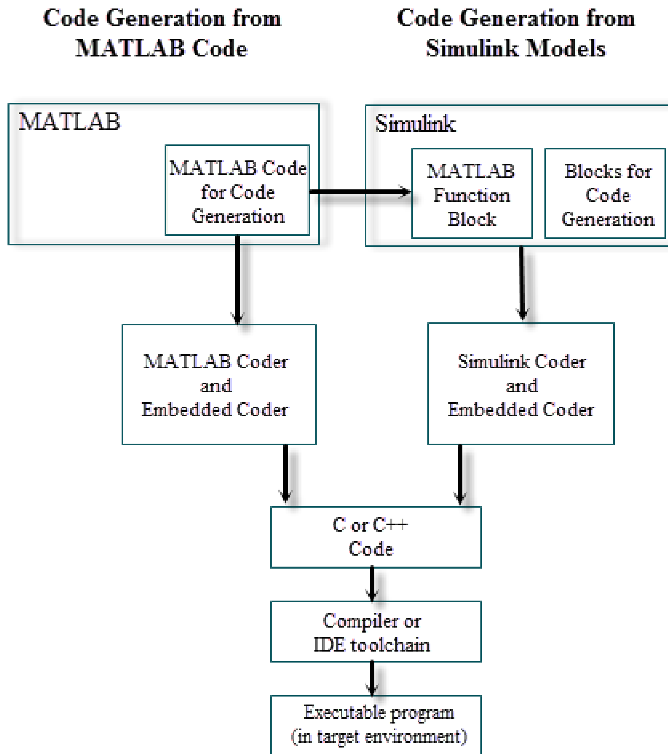
To learn model design patterns that include Simulink blocks, Stateflow® charts, and MATLAB functions, and map to commonly used C constructs, see "Modeling Patterns for C Code".

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, 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.

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.



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

Code generation from the 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 generated code functions, files, and data. For more information about these options and optimizations , see “Embedded Coder Capabilities for Code Generation from MATLAB Code” on page 2-2.

For more information about generating code from MATLAB code, see “Code Generation Workflow” (MATLAB Coder).

To get started generating code from MATLAB code using Embedded Coder, see “Embedded Coder Capabilities for Code Generation from MATLAB Code” on page 2-2.

Code Generation from Simulink Models

Code generation from the 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 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 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.

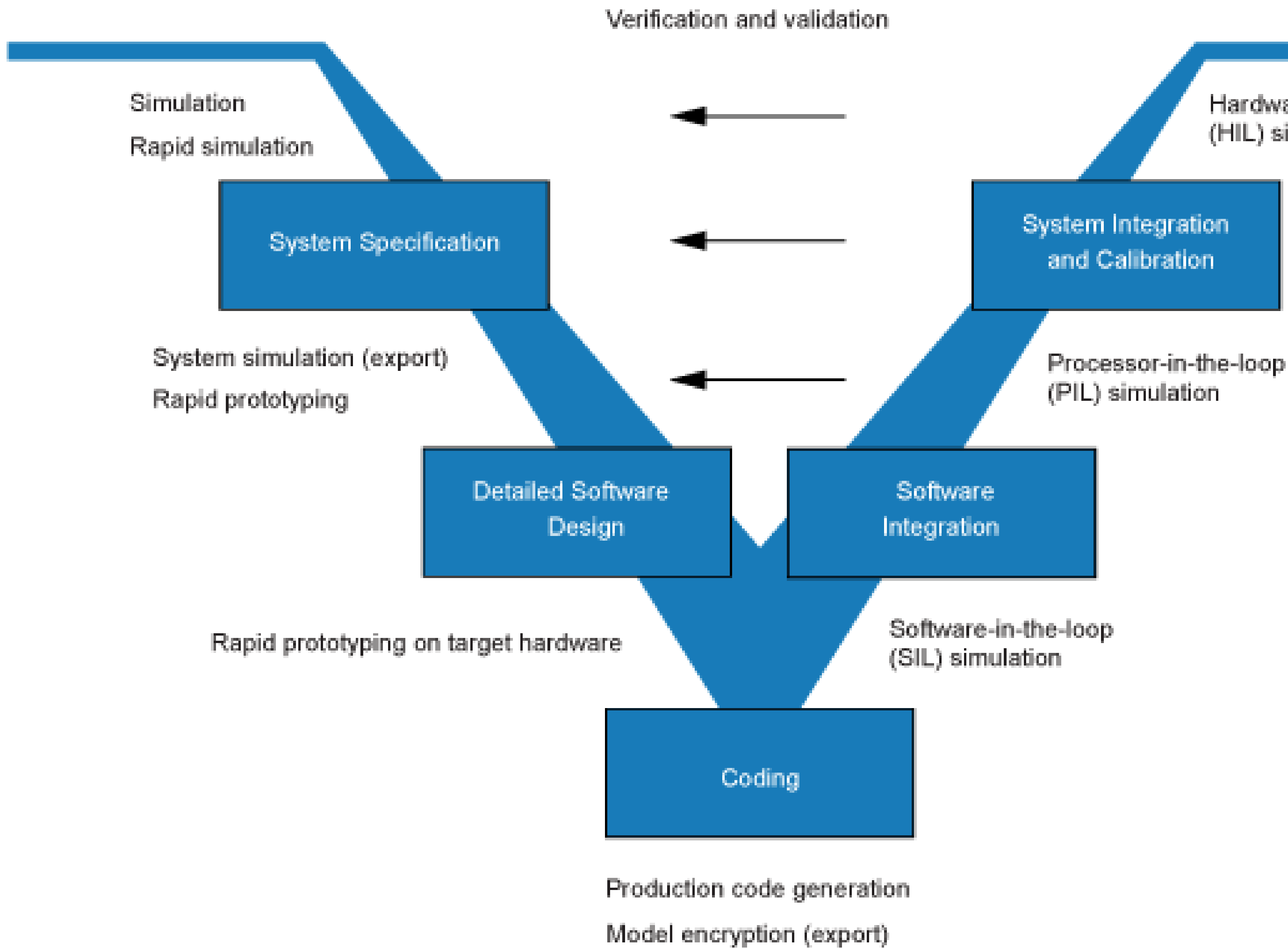
To learn how to model and generate code for commonly used C constructs using Simulink blocks, Stateflow charts, and MATLAB functions, see “Modeling Patterns for C Code”.

Validation and Verification for System Development

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

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. The left side of the 'V' identifies steps that lead to code generation, including system specification and detailed software design. The right side of the V focuses on the verification and validation of steps cited on the left side, including software 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 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
- “Code Generation Goal Summary” on page 1-12

Types of Simulation and Prototyping in the V-Model

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

	Simulation	Rapid Simulation	System Simulation, Rapid Prototyping	Rapid Prototyping on Target Hardware
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 design during development process
Execution hardware	Development computer	Development 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

	Simulation	Rapid Simulation	System Simulation, Rapid Prototyping	Rapid Prototyping on Target Hardware
Ease of use and cost	Can simulate component (algorithm or controller) and environment (or plant) Normal mode simulation in Simulink enables you to access, display, and tune data during verification Can accelerate Simulink simulations with Accelerated and Rapid Accelerated modes	Easy to simulate models of hybrid dynamic systems that include components and environment models Ideal for batch or Monte Carlo simulations Can repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model Can connect to Simulink to monitor signals and tune parameters	Might require custom real-time simulators and hardware Might be done with inexpensive off-the-shelf PC hardware and I/O cards	Might use existing hardware, thus less expensive and more convenient

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

This table compares the types of in-the-loop testing for verification identified on the right side of the V-model diagram.

	SIL Simulation	PIL Simulation on Embedded Hardware	PIL Simulation on Instruction Set Simulator	HIL Simulation
Purpose	Verify component source code	Verify component object code	Verify component object code	Verify system functionality

	SIL Simulation	PIL Simulation on Embedded Hardware	PIL Simulation on Instruction Set Simulator	HIL Simulation
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	Development computer	Target hardware	Development computer	Target hardware
Ease of use and cost	Desktop convenience Executes only in Simulink Reduces hardware cost	Executes on desktop or test bench Uses hardware — process board and cables	Desktop convenience Executes on development computer with Simulink and integrated development environment (IDE) Reduces hardware cost	Executes on test bench or in a lab Uses hardware — processor, embedded computer unit (ECU), I/O devices, and cables
Real-time capability	Not real time	Not real time (between samples)	Not real time (between samples)	Hard real time

Code Generation Goal Summary

These tables list goals that you might have, as you apply code generation technology, and where to find guidance on how to meet those goals.

- Document and Validate Requirements
- Develop System Specification
- Develop Detailed Software Design
- Generate Application Code
- Integrate and Verify Software
- Integrate, Verify, and Calibrate System Components

You can open and run the examples linked below and generate code.

Document and Validate 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®	
Associate requirements documents with objects in concept models Generate a report on requirements associated with a model	"Requirements Management Interface" (Simulink Requirements) Bidirectional tracing in Microsoft Word, Microsoft Excel, HTML, and IBM Rational DOORS	`slvnvdemo_fuelsys_docreq`
Include requirements links in generated code	"Review and Maintain Requirements Links" (Simulink Requirements)	`rtwdemo_requirements`
Trace model elements and subsystems to generated code and vice versa	"Code Tracing"	`rtwdemo_hyperlinks`
Verify, refine, and test concept model in non real time on a development computer	"Model Architecture and Design" (Simulink Coder) "Model Architecture and Design" "Simulation" (Simulink) "Acceleration" (Simulink)	"Air-Fuel Ratio Control System with Stateflow Charts" (Simulink Coder)

Goals	Related Product Information	Examples
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 Simulate models for hybrid dynamic systems that include components and an environment or plant that requires variable-step solvers and zero-crossing detection	“Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” (Simulink Coder) “Host-Target Communication with External Mode Simulation” (Simulink Coder)	“Run Rapid Simulations Over Range of Parameter Values” (Simulink Coder) “Run Batch Simulations Without Recompiling Generated Code” (Simulink Coder) “Use MAT-Files to Feed Data to Inport Blocks for Rapid Simulations” (Simulink Coder)
Distribute simulation runs across multiple computers	“Simulink Test” “MATLAB Parallel Server” “Parallel Computing Toolbox”	

Develop System 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” (Simulink Coder)	
Schedule generated code	“Absolute and Elapsed Time Computation” (Simulink Coder) “Time-Based Scheduling and Code Generation” (Simulink Coder) “Asynchronous Events” (Simulink Coder)	“Time-Based Scheduling Example Models” (Simulink Coder)

Specify function boundaries of system	“Subsystems” (Simulink Coder)	`rtwdemo_atomic`
		`rtwdemo_ssreuse`
		`rtwdemo_filepart`
		`rtwdemo_exporting_functions`

Goals	Related Product Information	Examples
Specify components and boundaries for design and incremental code generation	"Component-Based Modeling" (Simulink Coder) "Component-Based Modeling"	rtwdemo_mdlreftop
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"	rtwdemo_fcnprotoctrl rtwdemo_cppclass
Manage data packaging in generated code for integrating and packaging data	"File Packaging" (Simulink Coder) "File Packaging"	rtwdemo_ssreuse rtwdemo_mdlreftop rtwdemo_advsc
Generate and control the format of comments and identifiers in generated code	"Add Custom Comments to Generated Code" "Construction of Generated Identifiers"	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 Libraries"	rtwdemo_shrlib

Goals	Related Product Information	Examples
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 Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design Test hardware	"Deployment" (Simulink Coder) "Deployment"	

Generate code for rapid prototyping	"Function and Class Interfaces" (Simulink Coder)
	
	"Configure Code Generation for Model Entry-Point Functions" (Simulink Coder)
	
	"Generate Modular Function Code for Nonvirtual Subsystems"
		`rtwdemo_counter`
		`rtwdemo_counter_msvc`
		`rtwdemo_async`

Generate code for rapid prototyping in hard real time, using PCs	"Simulink Real-Time"	"Create and Run Real-Time Application from Simulink Model" (Simulink Real-Time)
Generate code for rapid prototyping in soft real time, using PCs	"Simulink Desktop Real-Time"	`sldrtex_vdp` (and others)

Develop Detailed Software Design

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

Select code generation features for deployment	"Run-Time Environment Configuration" (Simulink Coder)
	
	"Run-Time Environment Configuration"
	
	"Sharing Utility Code"
	
	"AUTOSAR Code Generation"
		`rtwdemo_counter`
		`rtwdemo_counter_msvc`
		`rtwdemo_async`
		"Workflow Samples" (AUTOSAR Blockset)

Specify target hardware settings	"Run-Time Environment Configuration" (Simulink Coder) "Run-Time Environment Configuration"	`rtwdemo_targetsettings`
Design model variants	"Define, Configure, and Activate Variants" (Simulink) "Variant Systems"	
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" (Simulink Coder)
Convert a floating-point model or subsystem to a fixed-point representation	"Convert to Fixed Point" (Fixed-Point Designer)	`fxpdemo_fpa`
Iterate to obtain an optimal fixed-point design, using autoscaling	"Data Types and Scaling" (Fixed-Point Designer)	`fxpdemo_feedback`

Goals	Related Product Information	Examples
Create or rename data types specifically for your application	“Control Data Type Names in Generated Code”	rtwdemo_udt
Control the format of identifiers in generated code	“Construction of Generated Identifiers”	rtwdemo_symbols
Specify how signals, tunable parameters, block states, and data objects are declared, stored, and represented in generated code	“Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements”	rtwdemo_cscpredef
Create a data dictionary for a model	“What Is a Data Dictionary?” (Simulink)	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”	rtwdemo_memsec
Assess and adjust model configuration parameters based on the application and an expected run-time environment	“Model Configuration” (Simulink Coder) “Model Configuration”	“Generate Code Using Simulink® Coder™” (Simulink Coder) “Generate Code Using Embedded Coder®”
Check a model against basic modeling guidelines	“Select and Run Model Advisor Checks” (Simulink)	rtwdemo_advisor1
Add custom checks to the Simulink Model Advisor	“Create Model Advisor Checks” (Simulink Check)	slvnvdemo_mdladv
Check a model against custom standards or guidelines	“Select and Run Model Advisor 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” “Check Model Compliance” (Simulink Check)	rtwdemo_iec61508

Goals	Related Product Information	Examples
Obtain model coverage for structural coverage analysis such as MCDC	"Simulink Coverage"	
Prove properties and generate test vectors for models	Simulink Design Verifier™	`sldvdemo_cruise_control` `sldvdemo_cruise_control_verification`
Generate reports of models and software designs	"MATLAB 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)	`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" "Code Execution Profiling" "Static Code Metrics"	`rtwdemo_sil_topmodel`

Goals	Related Product Information	Examples

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”
		`rtwdemo_counter`
		`rtwdemo_counter_msvc`
		`rtwdemo_fcnprotoctrl`
		`rtwdemo_cppclass`
		`rtwdemo_async`
		“Workflow Samples” (AUTOSAR Blockset)

Integrate existing externally written C or C++ code with your model for simulation and code generation	“Block Authoring and Simulation Integration” (Simulink) “External Code Integration” (Simulink Coder)	`rtwdemos`, select **Model Architecture and Design > External Code Integration**
Generate code for on-target rapid prototyping on specific embedded microprocessors and IDEs	“Deploy Generated Component Software to Application Target Platforms”	In `rtwdemo_vxworks`

Generate 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”	`rtwdemos`, select **Performance**
Optimize code for a specific run-time environment, using specialized function libraries	“Code Replacement” (Simulink Coder) “Code Replacement” “Code Replacement Customization”	“Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®”
Control the format and style of generated code	“Control Code Style”	`rtwdemo_parentheses`
Control comments inserted into generated code	“Add Custom Comments to Generated Code”	`rtwdemo_comments`
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 and Maintain Requirements Links” (Simulink Requirements)	`rtwdemo_requirements`
Trace model blocks and subsystems to generated code and vice versa	“Code Tracing” “Standards, Guidelines, and Block Usage”	`rtwdemo_comments` `rtwdemo_hyperlinks`
Integrate existing externally written code with code generated for a model	“Block Authoring and Simulation Integration” (Simulink) “External Code Integration”	`rtwdemos`, select **Model Architecture and Design > External Code Integration**

Goals	Related Product Information	Examples
Verify generated code for MISRA C[a] and other run-time violations	“MISRA C Guidelines” “Polyspace Bug Finder” “Polyspace Code Prover”	
Protect the intellectual property of component model design and generated code Generate a binary file (shared library)	“Reference Protected Models from Third Parties” (Simulink) “Package Generated Code as Shared Libraries”	
Generate a MEX-file S-function for a model or subsystem so that it can be shared with a third-party vendor	“Generate S-Function from Subsystem” (Simulink Coder)	
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 Libraries”	
Test generated production code with an environment or plant model to verify a conversion of the model to code	“Software-in-the-Loop Simulation”	“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”	“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.

Integrate and Verify Software

Goals	Related Product Information	Examples
Integrate existing externally written C or C++ code with a model for simulation and code generation	"Block Authoring and Simulation Integration" (Simulink) "External Code Integration"	`rtwdemos`, select **Model Architecture and Design > External Code Integration**
Connect to data interfaces for generated C code data structures	"Data Exchange Interfaces" (Simulink Coder) "Data Exchange Interfaces"	`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"	`rtwdemo_fcnprotoctrl` `rtwdemo_cppclass`
Export virtual and function-call subsystems	"Generate Component Source Code for Export to External Code Base"	`rtwdemo_exporting_functions`
Include target-specific code	"Code Replacement" (Simulink Coder) "Code Replacement" "Code Replacement Customization"	"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"	

Goals	Related Product Information	Examples
Generate source code for integration with specific production environments	“Code Generation” (Simulink Coder) “Code Generation”	`rtwdemo_async` “Workflow Samples” (AUTOSAR Blockset)
Integrate code for a specific run-time environment, using specialized function libraries	“Code Replacement” (Simulink Coder) “Code Replacement” “Code Replacement Customization”	“Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®”
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 Authoring and Simulation Integration” (Simulink) “External Code Integration” (Simulink Coder)	`rtwdemos`, select **Model Architecture and Design > External Code Integration**
Connect to data interfaces for the generated C code data structures	“Data Exchange Interfaces” (Simulink Coder) “Data Exchange Interfaces”	`rtwdemo_capi` `rtwdemo_asap2`
Schedule the generated code	“Timers” (Simulink Coder) “Time-Based Scheduling” (Simulink Coder) “Event-Based Scheduling” (Simulink Coder)	“Time-Based Scheduling Example Models” (Simulink Coder)
Verify object code files in a target environment	“Software-in-the-Loop Simulation”	“Test Generated Code with SIL and PIL Simulations”

Goals	Related Product Information	Examples
Set up and run PIL tests on your target system	"Processor-in-the-Loop Simulation"	"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 find an example for the related product of interest

Integrate, Verify, and Calibrate 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" (Simulink Coder) "Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation" (Simulink Coder) "Deploy Generated Standalone Executable Programs To Target Hardware" "Deploy Generated Component Software to Application Target Platforms"	
Generate source code for HIL testing	"Code Generation" (Simulink Coder) "Code Generation" "Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation" (Simulink Coder)	
Conduct hard real-time HIL testing using PCs	"Simulink Real-Time"	"Create and Run Real-Time Application from Simulink Model" (Simulink Real-Time) "Real-Time Simulation and Testing" (Simulink Real-Time)
Tune ECU properly for its intended use	"Data Exchange Interfaces" (Simulink Coder) "Data Exchange Interfaces"	`rtwdemo_capi` `rtwdemo_asap2`

Goals	Related Product Information	Examples
Generate ASAP2 data files	“Export ASAP2 File for Data Measurement and Calibration” (Simulink Coder)	`rtwdemo_asap2`
Generate C API data interface files	“Exchange Data Between Generated and External Code Using C API” (Simulink Coder)	`rtwdemo_capi`

Target Environments and Applications

In this section...
“About Target Environments” on page 1-30
“Types of Target Environments” on page 1-30
“Applications of Supported Target Environments” on page 1-32

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. 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

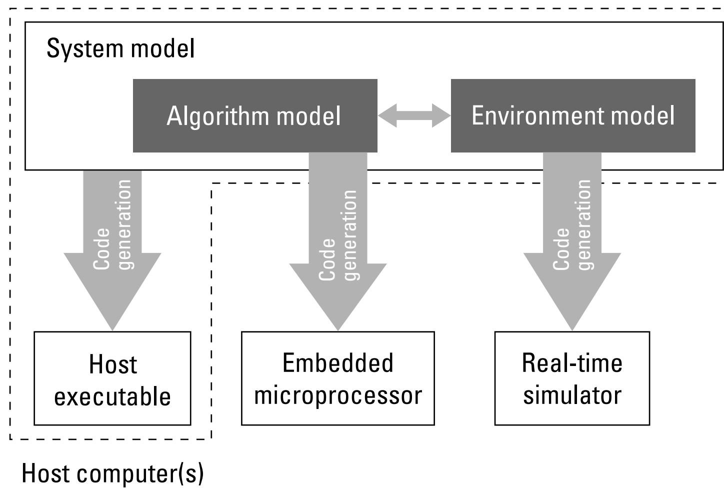
a. UNIX is a registered trademark of The Open Group in the United States and other countries.
b. Linux is a registered trademark of Linus Torvalds.

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	

Application	Description
“Acceleration” (Simulink)	You apply techniques to speed up the execution of model simulation in the context of the MATLAB and Simulink environments. Accelerated simulations are especially useful when run time is long compared to the time associated with compilation and checking whether the target is up to date.
Rapid Simulation (Simulink Coder)	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.
“Protect Models to Conceal Contents” (Simulink Coder)	You generate a protected model for use by a third-party vendor in another Simulink simulation environment.
Real-Time Simulator	
Real-Time Rapid Prototyping (Simulink Coder)	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.

Application	Description
Hardware-in-the-Loop (HIL) Simulation (Simulink Coder)	You generate code for a detailed design that you can run in real time on an embedded microprocessor while tuning parameters and monitoring real-time data. This design step allows you to assess, interact with, and optimize code, using embedded compilers and hardware.
Embedded Microprocessor	
"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 (Simulink Coder)	You verify an embedded system or embedded computing unit (ECU), using a real-time target environment.

2

MATLAB Tutorials

- "Embedded Coder Capabilities for Code Generation from MATLAB Code" on page 2-2
- "Controlling C Code Style" on page 2-9
- "Include Comments in Generated C/C++ Code" on page 2-15

Embedded Coder Capabilities for Code Generation from MATLAB Code

The Embedded Coder product extends the MATLAB Coder product with capabilities that you can use for embedded software development. 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 generated code.
- Optimize generated code for application-specific requirements.
- Enable tracing options that help you to verify the generated 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
Simplify array indexing in loops in the generated code	**Simplify array indexing**	`EnableStrengthReduction`	"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	`CodeReplacementLibrary`	Embedded Coder offers additional libraries and the ability to create and use custom code. See "Code Replacement Customization".

Goal	Project Setting	Code Configuration Object Property	More Information
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”
Include MATLAB source code as comments with traceability tags. In the code generation report, the traceability tags link to the corresponding MATLAB source code	**MATLAB source code as comments**	`MATLABSourceComments`	“Include Comments in Generated C/C++ Code” on page 2-15
Generate MATLAB function help text in the function banner	**MATLAB function help text**	`MATLABFcnDesc`	“Include Comments in Generated C/C++ Code” on page 2-15
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
Specify the level of parenthesization in the generated code	**Parentheses**	`ParenthesesLevel`	N/A

Goal	Project Setting	Code Configuration Object Property	More Information
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**	`EnableSignedLeftShifts`	"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**	`EnableSignedRightShifts`	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"
Specify the maximum number of columns before a line break in the generated code	**Column limit** on the **All Settings** tab	`ColumnLimit`	N/A
Specify custom names for MATLAB data types in generated code	**Enable custom data type replacement**	`EnableCustomReplacementTypes` `ReplacementTypes`	"Customize Data Type Replacement"
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"

Goal	Project Setting	Code Configuration Object Property	More Information
Customize generated identifiers for local temporary variables	**Local temporary variables**	`CustomSymbolStr-TmpVar`	"Customize Generated Identifiers"
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"

Goal	Project Setting	Code Configuration Object Property	More Information
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			
Generate a static code metrics report including generated file information, number of lines, and memory usage	**Static code metrics**	`GenerateCodeMetrics-Report`	"Generating a Static Code Metrics Report for Code Generated from 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			

Goal	Project Setting	Code Configuration Object Property	More Information
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			
Interactively trace between MATLAB source code and generated C/C++ code	`EnableTraceability`	**Enable Code Traceability**	“Interactively Trace Between MATLAB Code and Generated C/C++ Code”
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”

Goal	Project Setting	Code Configuration Object Property	More Information
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"
Run Polyspace® verification on generated C/C++ code by using the integrated workflow	N/A	N/A	"Polyspace Verification of C/C++ Code Generated by MATLAB Coder"

Controlling C Code Style

In this section...

"About This Tutorial" on page 2-9
"Copy File to a Local Working Folder" on page 2-10
"Open the MATLAB Coder App" on page 2-10
"Specify Source Files" on page 2-10
"Define Input Types" on page 2-11
"Check for Run-Time Issues" on page 2-11
"Configure Code Generation Parameters" on page 2-12
"Generate C Code" on page 2-12
"View the Generated Code" on page 2-12
"Finish the Workflow" on page 2-13
"Key Points to Remember" on page 2-14

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 trace between the original MATLAB code and the generated C code.

Required Products

This tutorial requires the following products:

- MATLAB
- MATLAB Coder
- C compiler

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” (MATLAB).

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

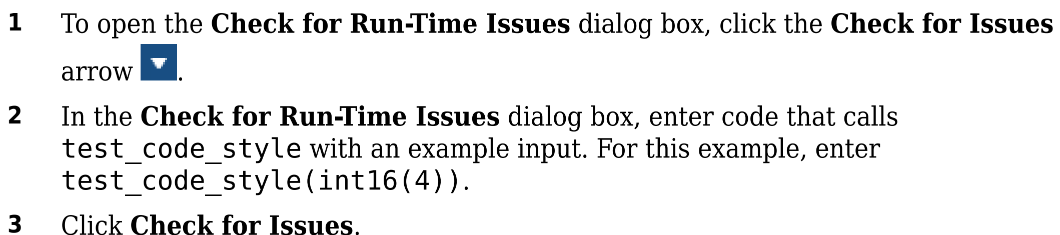
Because C uses static typing, at compile time, the code generator must determine the properties of all variables in the MATLAB files. Therefore, you must specify the properties of all function inputs. To define the properties of the input `x`:

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

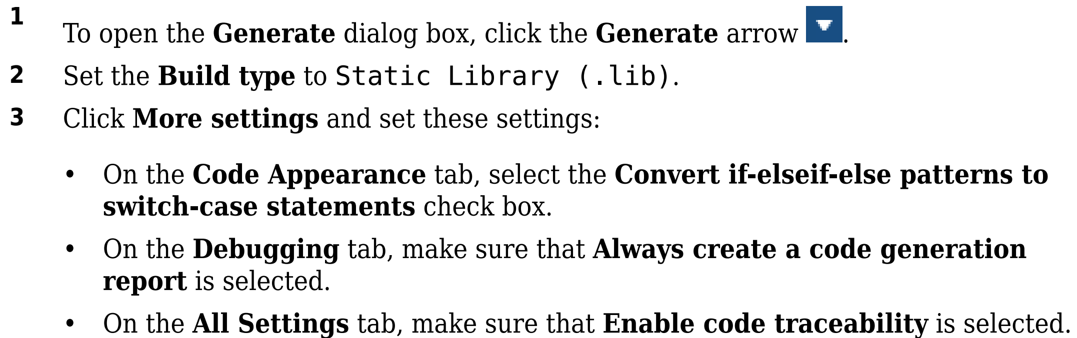
The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. Using this step, you can detect and fix run-time errors that are harder to diagnose in the 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” (MATLAB Coder).

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 Code Generation 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** and set these settings:

   - On the **Code Appearance** tab, select the **Convert if-elseif-else patterns to switch-case statements** check box.
   - On the **Debugging** tab, make sure that **Always create a code generation report** is selected.
   - On the **All Settings** tab, make sure that **Enable code traceability** is selected.

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. The code generator provides a link to the report.

View the Generated Code

1 To open the code generation report, click the **View Report** link.

   The `test_code_style` function is displayed in the code pane.
2 To view the MATLAB code and the C code next to each other, click **Trace Code**.
3 In the MATLAB code, place your cursor over the statement `if (x == 1)`.

   The report traces `if (x == 1)` to a `switch` statement.

Code

test_code_style.m   test_code_style.m - [4-4] | *1 trace found*

```
1  function y = test_code_style(x)
2  %#codegen
3
4  if (x == 1)
5      y = 1;
6  elseif (x == 2)
7      y = 2;
8  elseif (x == 3)
9      y = 3;
10 else
11     y = 4;
12 end
```

```
9
10 /* Include Files */
11 #include "rt_nonfinite.h"
12 #include "test_code_style.h"
13
14 /* Function Definitions */
15
16 /*
17  * Arguments    : short x
18  * Return Type  : double
19  */
20 double test_code_style(short x)
21 {
22   double y;
23   switch (x) {
24    case 1:
25     y = 1.0;
26     break;
27
28    case 2:
29     y = 2.0;
30     break;
31
32    case 3:
33     y = 3.0;
34     break;
35
36    default:
37     y = 4.0;
38     break;
39   }
```

Finish the Workflow

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**.

See Also

More About

- "Generate C Code by Using the MATLAB Coder App" (MATLAB Coder)
- "Generate C Code at the Command Line" (MATLAB Coder)
- "Interactively Trace Between MATLAB Code and Generated C/C++ Code"

Include Comments in Generated C/C++ Code

In this section...

"About This Tutorial" on page 2-15
"Creating the MATLAB Source File" on page 2-15
"Configuring Build Parameters" on page 2-16
"Generating the C Code" on page 2-16
"Viewing the Generated C Code" on page 2-17
"Tracing the Generated Code to the MATLAB Code" on page 2-17

About This Tutorial

Learning Objectives

This tutorial shows you how to generate code that includes:

- The function signature and function help text in the function banner.
- MATLAB source code as comments with traceability tags. In the code generation report, the traceability tags link to the corresponding MATLAB source code.

Prerequisites

To complete this tutorial, you must have these products:

- MATLAB
- MATLAB Coder
- Embedded Coder
- C compiler

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

Creating the MATLAB Source File

In a writable folder, create a copy of the tutorial file.

```
copyfile(fullfile(docroot, 'toolbox', 'ecoder', 'examples', 'polar2cartesian.m'))
```

polar2cartesian

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

Configuring Build Parameters

Create a `coder.EmbeddedCodeConfig` code generation configuration object and set these properties to `true`:

- `GenerateComments` to allow comments in the generated code.
- `MATLABSourceComments` to generate MATLAB source code as comments with traceability tags. In the code generation report, the tags link to the corresponding MATLAB code. When this property is `true`, the code generator also produces the function signature in the function banner.
- `MATLABFcnDesc` to generate the function help text in the function banner.

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

Generating the C Code

To generate C code, call the `codegen` function. Use these 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

View the generated code in the code generation report.

1. To open the code generation report, click `View report`.
2. In the **Generated Code** pane, click `polar2cartesion.c`.

   The generated code includes:

   - The function signature and function help text in the function banner.
   - Comments containing the MATLAB source code that corresponds to the generated C/C++ code. The comment includes a traceability tag that links to the original MATLAB code.

```
/*
 * function [x y] = polar2cartesian(r,theta)
 * Convert polar to Cartesian
 * Arguments    : double r
 *                double theta
 *                double *x
 *                double *y
 * Return Type  : void
 */
void polar2cartesian(double r, double theta, double *x, double *y)
{
  /* 'polar2cartesian:4' x = r * cos(theta); */
  *x = r * cos(theta);

  /* 'polar2cartesian:5' y = r * sin(theta); */
  *y = r * sin(theta);
}
```

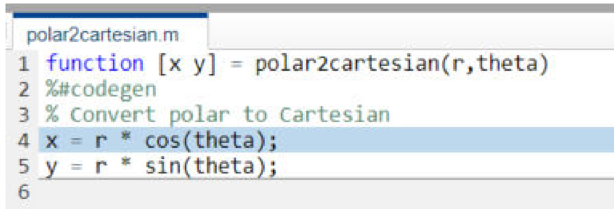
The generated function banner also depends on the code generation template (CGT) file. With the default CGT, the code generator places information about the arguments in the function banner. You can customize the function banner by modifying the CGT. See "Generate Custom File and Function Banners for C/C++ Code".

Tracing the Generated Code to the MATLAB Code

Traceability tags provide information and links that help you to trace the generated code back to the original MATLAB code. For example, click the traceability tag that precedes the code `x = r * cos(theta);`.

```
/* 'polar2cartesian:4' x = r * cos(theta); */
```

The report opens `polar2cartesian.m` and highlights line 4.



To view the MATLAB source code and generated C/C++ code next to each other and to interactively trace between them, in the report, click **Trace Code**. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code”.

See Also

More About

- “Specify Comment Style for C/C++ Code”
- “Tracing Generated C/C++ Code to MATLAB Source Code” (MATLAB Coder)
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code”
- “Code Generation Template Files for MATLAB Code”
- “Generate Custom File and Function Banners for C/C++ Code”

3

Simulink Code Generation Tutorials

- “Generate C Code from Simulink Models” on page 3-2
- “Generate Code by Using Embedded Coder Quick Start” on page 3-6
- “Configure Data Interface” on page 3-11
- “Configure a Model Parameter as a Global Variable for Tuning During Run Time” on page 3-18
- “Compare Model Simulation and Generated Code Results” on page 3-21
- “Deploy the Generated Code” on page 3-26

Generate C Code from Simulink Models

In this section...
“Prerequisites” on page 3-2
“Example Models” on page 3-2

Use the Embedded Coder product to generate C or C++ code that is optimized for deployment on rapid-prototyping boards, embedded processors, or microprocessors. If you are new to Embedded Coder or your application code customization requirements are minimal, you can use graphical tools and default code configuration settings to quickly generate production-quality code. If you need to produce customized code for integration with existing external code or you want to meet code guidelines and standards, tooling is available to configure the code generator to meet requirements for interfacing, code appearance, packaging, and optimizations.

Generating and reviewing code for deployment to an embedded system can be as simple as preparing the model for code generation with the Quick Start tool. Then, with code tools accessible from the Simulink Editor, you can configure code interfaces, initiate code generation, and review the generated code.

Prerequisites

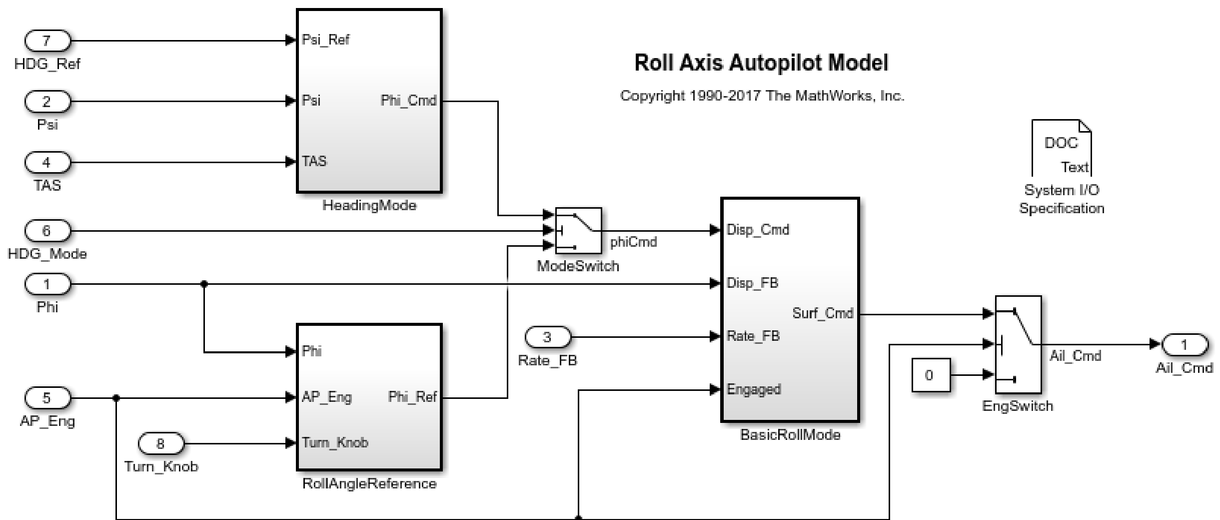
To complete this tutorial, you must have:

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

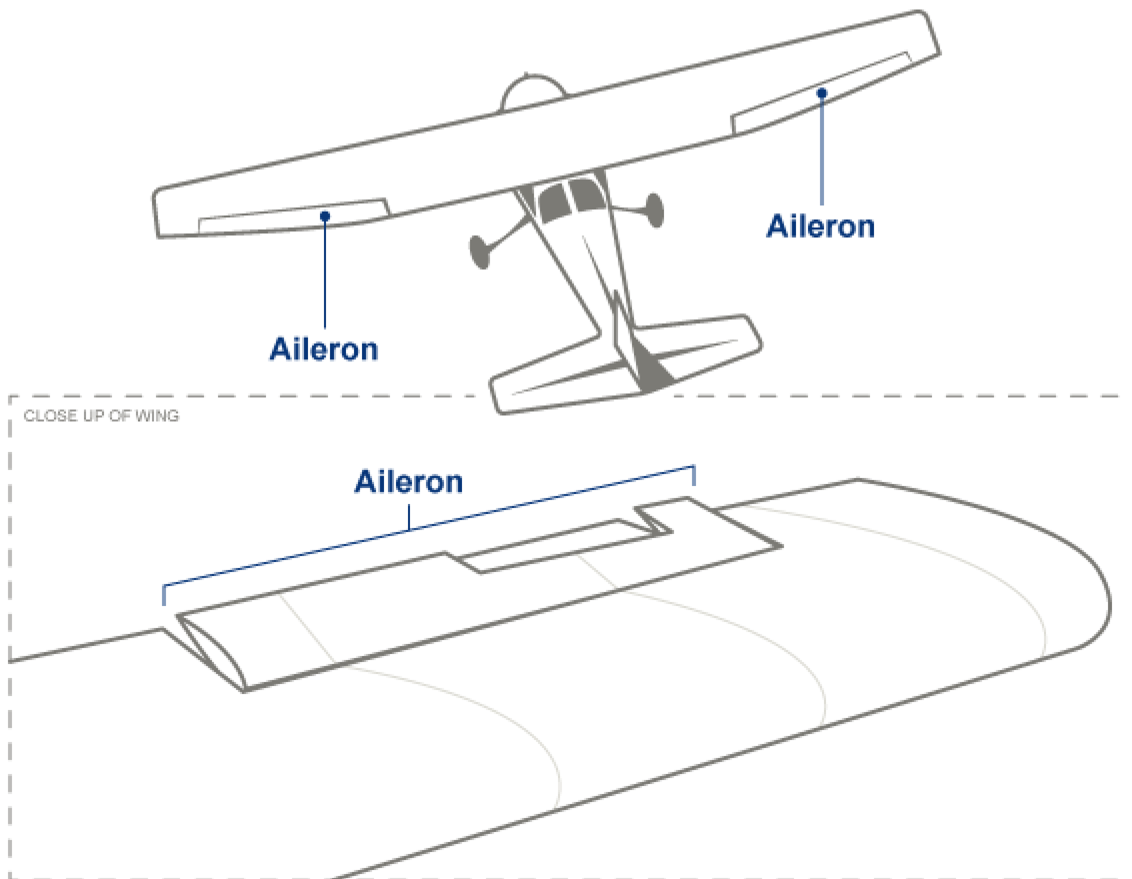
Example Models

The tutorial uses example models `rtwdemo_roll` and `rtwdemo_roll_harness`. The models have been verified for simulation.

Open model `rtwdemo_roll`.



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



The model represents one component in the greater control system of an aircraft. Through the `HDG_Mode` signal, the control system places the model in one of two operating modes: roll attitude hold or heading hold. The `RollAngleReference` and `HeadingMode` subsystems calculate a roll attitude setpoint that supports one of the operating modes. Then, the `BasicRollMode` subsystem, a PID controller, calculates an aileron position command based on the appropriate setpoint and on feedback that indicates the measured roll attitude and rate of change. The model is designed to operate at 40 Hz.

The tutorial uses model `rtwdemo_roll_harness` to test `rtwdemo_roll`.

You will learn how to:

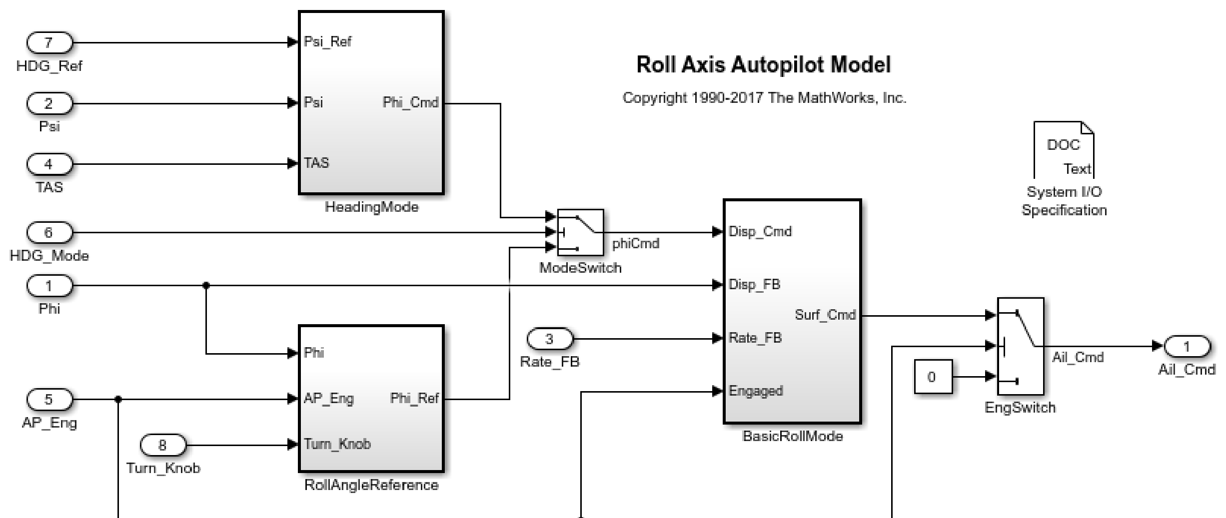
1. Generate code by using the Embedded Coder Quick Start.
2. Configure the data interface.
3. Configure a model parameter as a global variable for tuning during run time.
4. Compare model simulation and generated code results for numeric equivalency.
5. Deploy the generated code.

To start the tutorial, see “Generate Code by Using Embedded Coder Quick Start” on page 3-6.

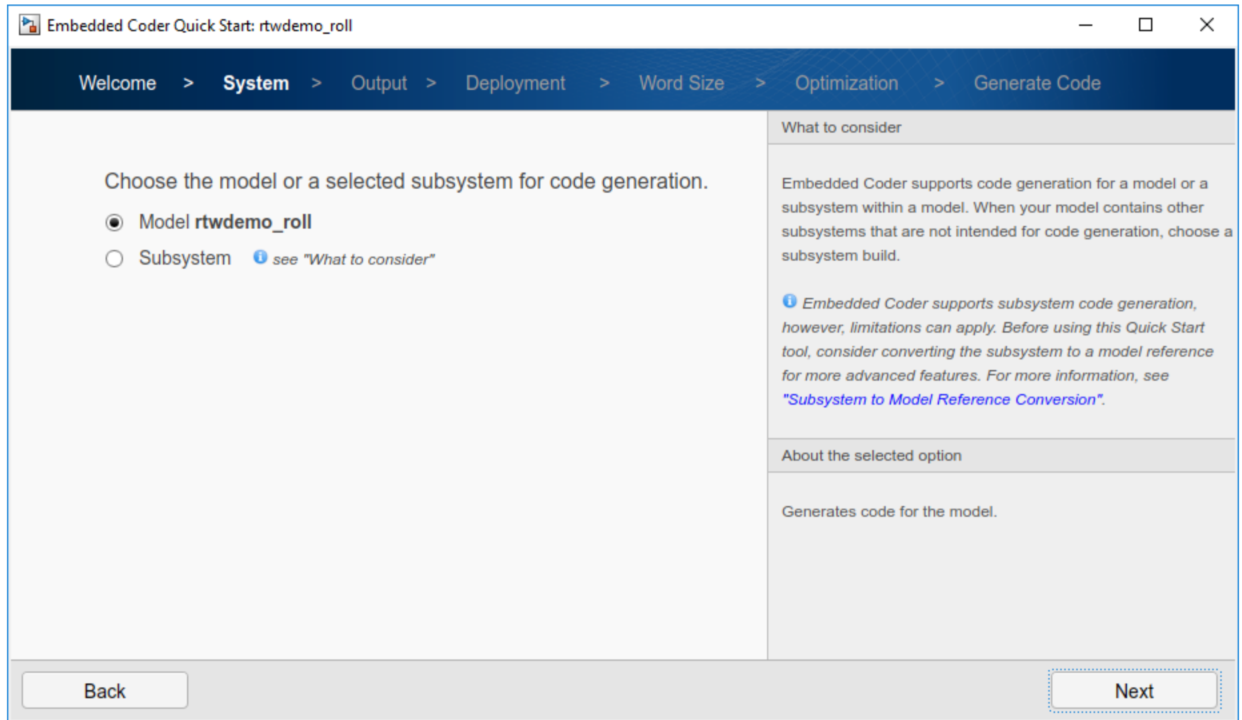
Generate Code by Using Embedded Coder Quick Start

Model `rtwdemo_roll` described in “Generate C Code from Simulink Models” on page 3-2, represents an autopilot control system for an aircraft. You prepare `rtwdemo_roll` for embedded code generation by using Embedded Coder Quick Start, which chooses fundamental code generation settings based on your goals and application.

1 Open model `rtwdemo_roll`.

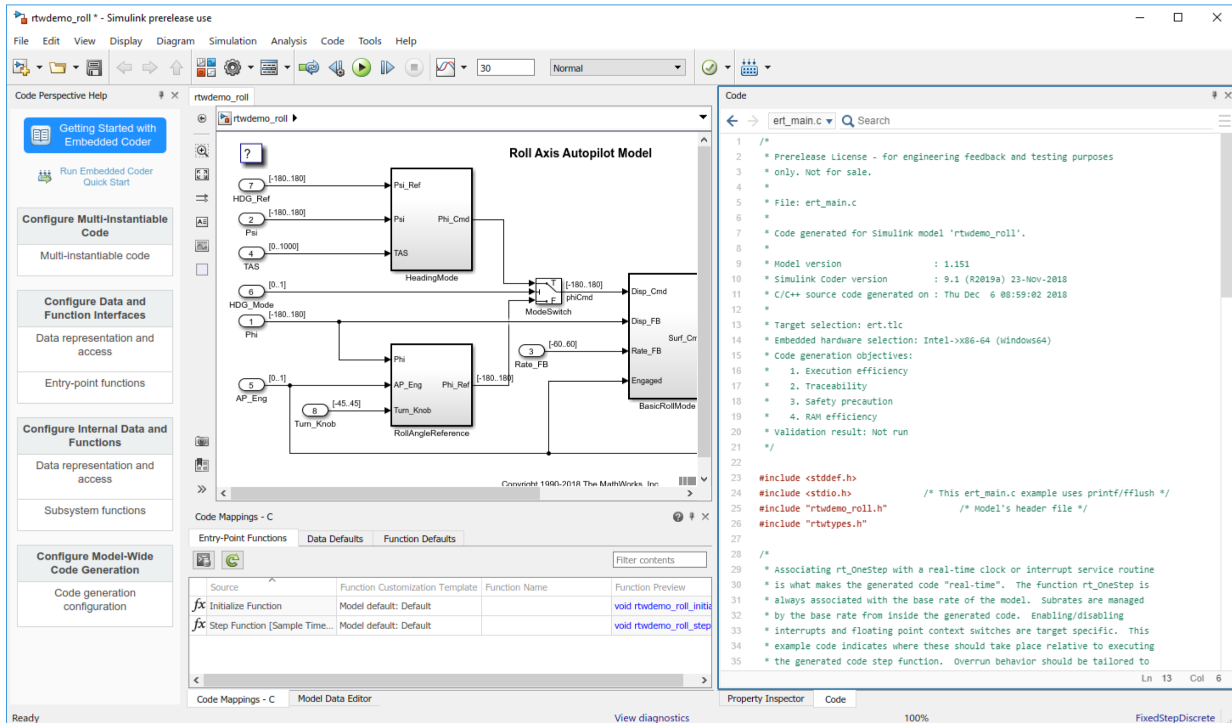


2 Save a copy of the model to a writable location on the search MATLAB path.
3 Open the Quick Start tool. In the Simulink Editor, select **Code** > **C/C++ Code** > **Embedded Coder Quick Start**.
4 Advance through the steps of the Quick Start tool. Each step asks questions about the code that you want to generate. For this tutorial, use the defaults that are already selected. The tool validates your selections against the model and presents the parameter changes required to generate code.



5. In the **Generate Code** step, apply the proposed changes and generate code from `rtwdemo_roll` by clicking **Next**.
6. Close the tool by clicking **Finish**.

   The tool places the model in the Simulink Editor Code perspective. In this perspective, you can configure code generation customizations, and then check that the results are correct in the Code view alongside the model.



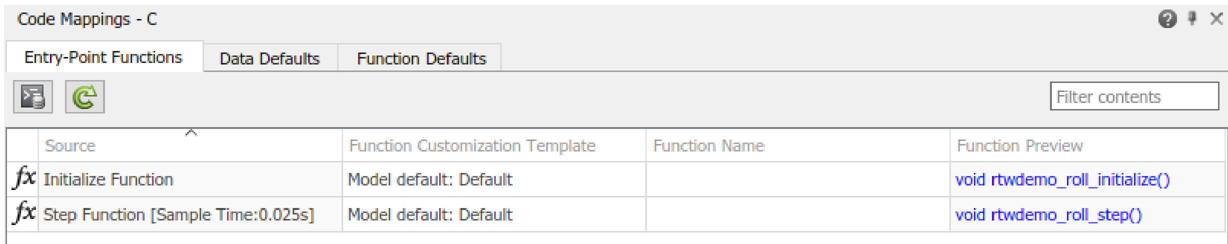
In the Code Mappings editor, on the **Entry-Point Functions** tab, you can see the individual entry-point functions that the code generator produces. You call these generated functions from external code or from a version of a generated main function that you modify. If required, you can change the name of a function. For the base-rate step function of a rate-based model and for step functions for export function models, you can customize the function name and arguments.

The generated code appears in two primary files: `rtwdemo_roll.c` and `rtwdemo_roll.h`. In your MATLAB current folder, the `rtwdemo_roll_ert_rtw` folder contains these primary files.

In your current folder, the code generator creates the `slprj` folder. This folder contains the `rtwtypes.h` file, which defines standard data types that the generated code uses by default. In general, this sibling folder contains generated files that can or must be shared between multiple models.

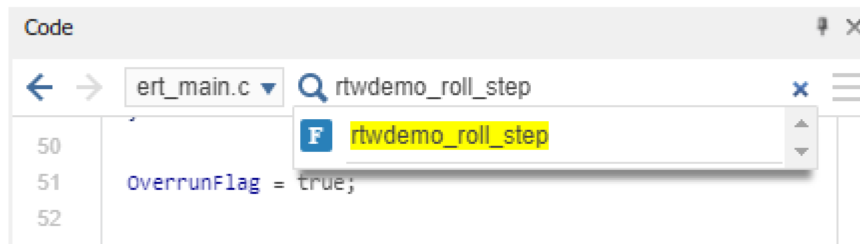
The code that you generate from a model includes entry-point functions, which you call from your application code. For a rate-based model, these functions include an initialization function, an execution function, and, optionally, terminate and reset functions. The functions exchange data with your application code through a data interface that you control.

In the Code Mappings editor, on the **Entry-Point Functions** tab, review the list of entry-point functions that the code generator produces for the model. Use this view to selectively specify for each function a function customization template (code definition) and name. For this tutorial, the code generator uses default (shipped) settings for the customization template and entry-point function names. The code generator names the initialize function `rtwdemo_roll_initialize` and the execution (step) function `rtwdemo_roll_step`. Both entry-point functions have a `void-void` interface (they pass no arguments). The functions gain access to data through shared data structures. Examples of such data include system-level input and output that the functions exchange with application code.

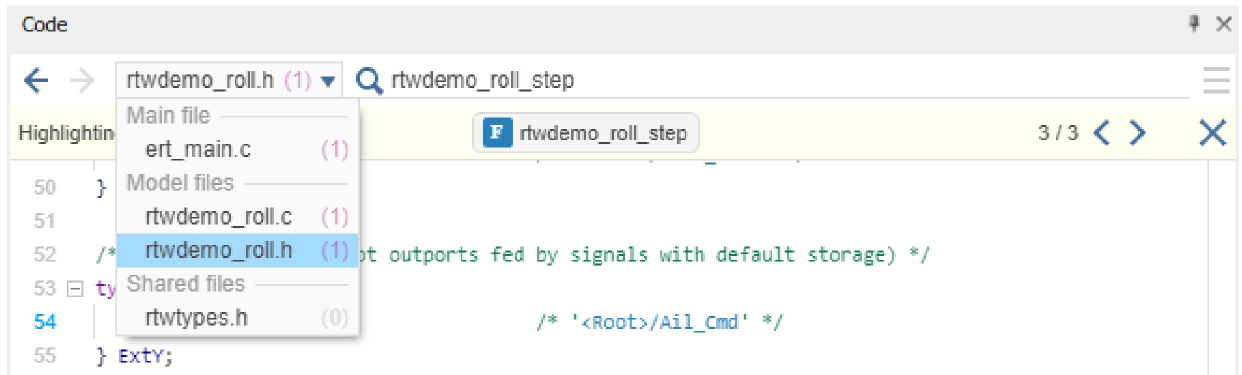


To see these entry-point functions in the generated code:

1 On the right side of the Simulink Editor Code perspective, in the Code view pane, locate the search bar.
2 In the search bar, type `rtwdemo_roll_step`. To find each instance of the step function name across the generated code files, click the search suggestion.



3 Use the arrows on the right to step through each instance, including the step function definition in `rtwdemo_roll.c` and the declaration in `rtwdemo_roll.h`. The step function has a `void-void` interface that does not pass arguments. You can also see the number of search hits in each file from the file menu in the upper left corner.



4 Repeat these search steps to locate the initialize function, `rtwdemo_roll_initialize` in the generated code.

Next, configure the data interface for code generation and review the generated code.

Configure Data Interface

Embedded Coder reduces the effort for configuring data and function interfaces by providing a way to specify default configurations for categories of data elements and functions across a model. Applying default configurations can save time and reduce the risk of introducing errors in code, especially for larger models and models from which you generate multi-instance code. After applying default configurations, you can selectively override the default settings for individual data elements and functions.

Customize the data interface of model `rtwdemo_roll` by configuring function `roll_control_step` to:

- Read input data from global variables that are declared and defined in external files `roll_input_data.h` and `roll_input_data.c`.
- Write output data to global variables that the code generator declares in `output_data.h` and defines in `output_data.c`.

To make these changes, in the MATLAB Command Window, copy these external code files to your current MATLAB working folder.

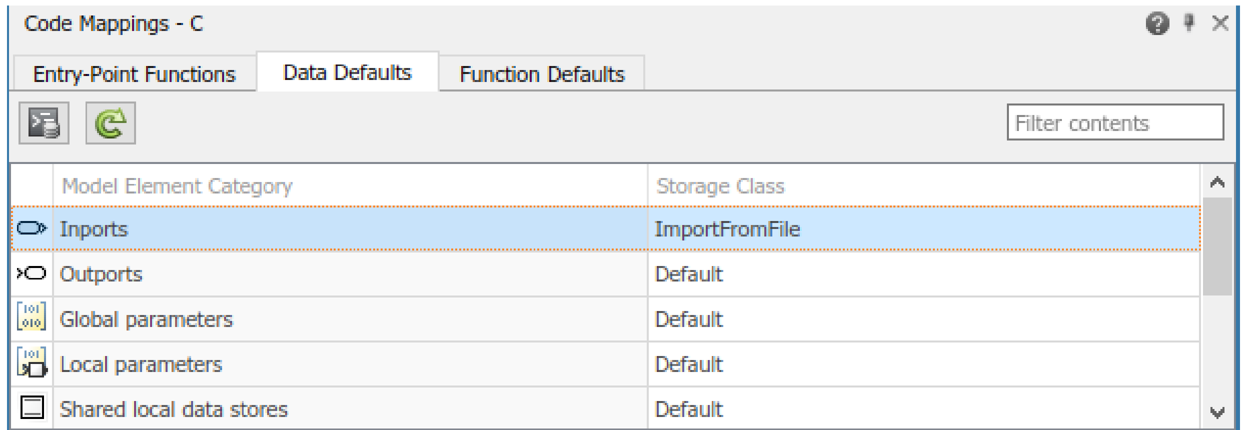
```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.h'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.h'));
```

The data interface configuration changes that you make depend on these files being accessible for code generation and the build process. The build process compiles the generated code with the code that is in these files.

Configure Default Code Generation for Data

Configure default code generation configurations for model inports and outports.

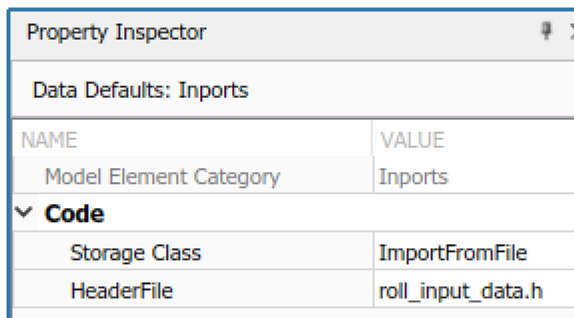
1. Configure Inport blocks at the root level of the model to appear in the generated code as separate global variables defined by external code. In the Code Mappings editor, on the **Data Defaults** tab, for category **Inports**, set **Storage Class** to `ImportFromFile`.



With this setting, the generated code does not define global variables that represent the inport data. Instead, a `#include` statement includes a header file that declares the input variables. You specify the name of the header file with the Property Inspector.

2 Open the Property Inspector. In the lower right corner of the Simulink Editor window, click the **Property Inspector** tab.

3 In the Property Inspector, set property **HeaderFile** to `roll_input_data.h`.



4 To see how the `extern` declarations in external header file `roll_input_data.h` name the input variables, from the MATLAB Command Window, open `roll_input_data.h`.

```
extern boolean_T AP_Eng;
extern real32_T HDG_Ref;
extern real32_T Rate_FB;
```

```
extern real32_T Phi;
extern real32_T Psi;
extern real32_T TAS;
extern real32_T Turn_Knob;
```

5 Configure the code generation naming rule for global variables. By default, the code generator names global variables with the prefix `rt`. For the code generator to produce code that matches the external variable declarations in `roll_input_data.h`, configure the code generation naming rule for global variables accordingly.

   a Open the Model Configuration Parameters dialog box.

   b Navigate to the **Code Generation** > **Symbols** pane.

   c Set parameter **Global variables** to the naming rule `$N$M` (remove the `rt` prefix). Token `$N` represents the name of a data element in the model, for example, the name of an Inport or Outport block. Token `$M` represents name-mangling text that the code generator inserts, if necessary, to avoid name collisions with other global variables in the code.

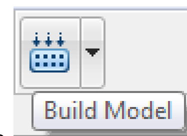
   d Apply the change.

6 Configure Outport blocks at the root level of the model to appear in the generated code as separate global variables. In the Simulink Editor window, in the Code Mappings editor, on the **Data Defaults** tab, for category **Outports**, set **Storage Class** to `ExportToFile`.

   The generated code declares and defines the output variables in header and definition files that you specify with the Property Inspector.

7 In the Property Inspector, specify the names to be used for the generated header and definition files. Set property **HeaderFile** to `roll_output_data.h` and property **DefinitionFile** to `roll_output_data.c`.

8 Configure code generation for the model to include the external source files `roll_input_data.c` and `roll_heading_mode.c`. In the Configuration Parameters dialog box, set **Code Generation** > **Custom Code** > **Additional build information** > **Source files** to `roll_input_data.c roll_heading_mode.c`. Then, click **Apply** and **OK**.

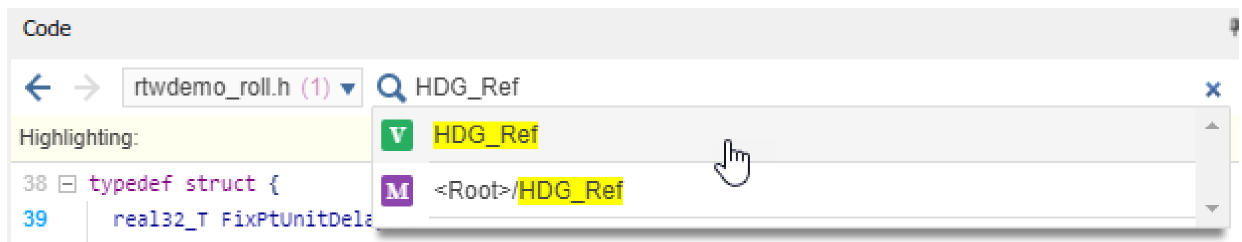
9 

   Save the model. Regenerate the code by clicking the button.

A compiler error indicates that variable `HDG_Mode` is not declared. That variable is not declared in header file `roll_output_data.h`, which you declared as the default header file for inports. You fix this error in the next section of this tutorial.

The model is configured to open the code generation report after code generation is complete. Minimize this report window for exploration later in this tutorial.

10. You configured Inport blocks to use an external header file to declare and define input variables. In the Code view, confirm that the generated code includes this external header file by searching for `roll_input_data.h`.
11. Search for the root level Inport block name, `HDG_Ref`. As you type, choose the search suggestion with the green `V` icon. This search suggestion finds instances of `HDG_Ref` used as a variable in the generated code. Confirm that `HDG_Ref` is defined as a separate global variable.



12. In the model, `rtwdemo_roll`, click the Outport block `Ail_Cmd`. The Code view highlights code in `rtwdemo_roll.c` that corresponds to the block. In the code, place your cursor over the ellipsis menu above the output variable `Ail_Cmd`. The traceability dialog box can show you variable definitions and model elements that correspond to the code. The dialog box confirms that `Ail_Cmd` is defined as a separate global variable. Click the definition code to see the definition in `output_data.c`.

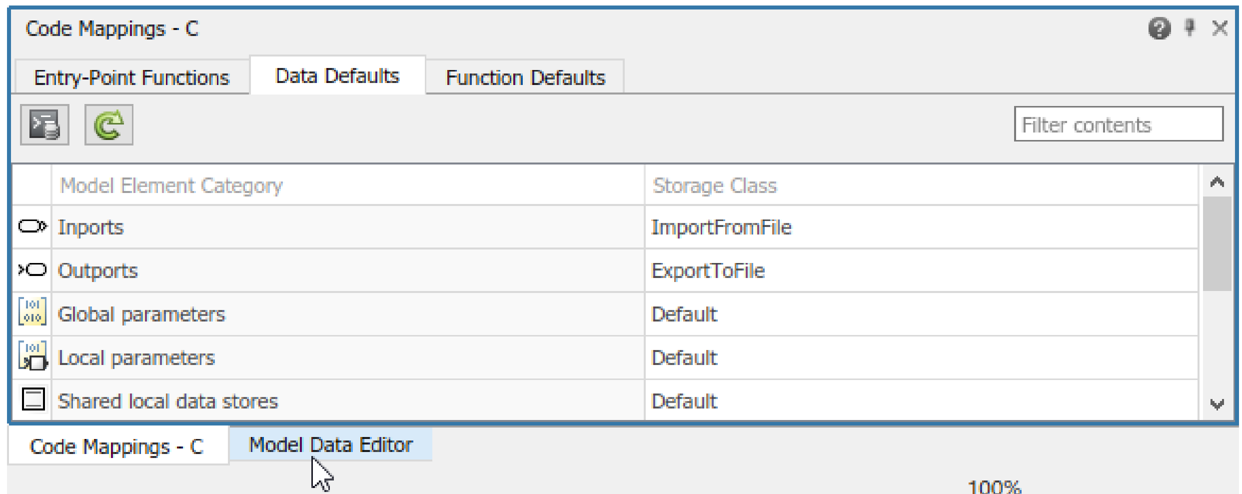
```
if (  Definition
/*    Variable defined in roll_output_data.c                        ' */
/*    28 real32_T Ail_Cmd;
if    Model elements
      M <Root>/EngSwitch   M <Root>/Ail_Cmd   M <S1>/CmdLimit
    Ail_Cmd  15.0F;
  } else if (rtb_TKSwitch < -15.0F) {
    /* Outport: '<Root>/Ail_Cmd' */
    Ail_Cmd = -15.0F;
  } else {
    /* Outport: '<Root>/Ail_Cmd' */
    Ail_Cmd = rtb_TKSwitch;
  }
```

Override Default Settings for Individual Data Elements

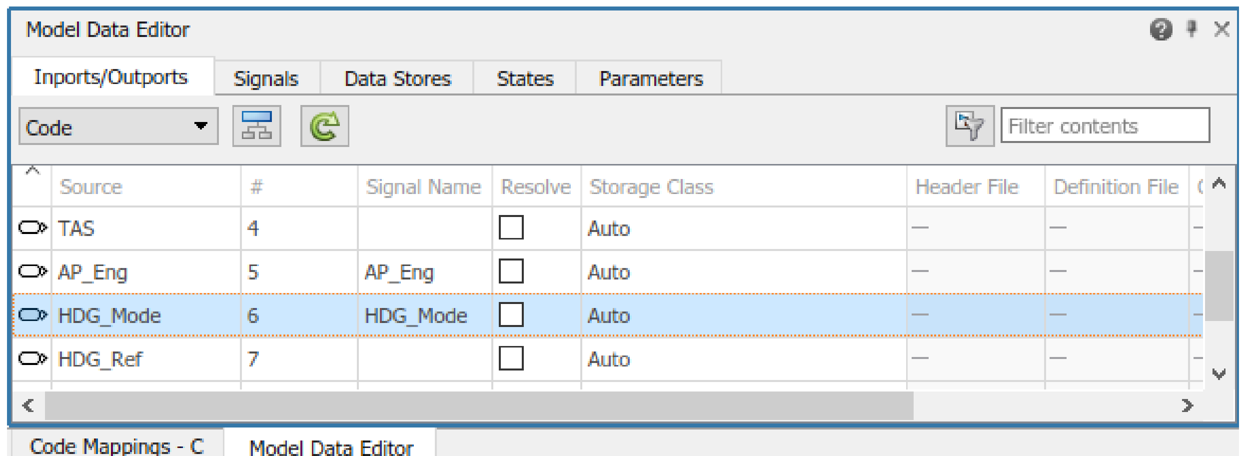
The settings that you choose for a category under **Data Defaults** apply to elements in that category across a model. To override the default settings for an individual data element, use the Model Data Editor.

When you generated code after configuring default settings for inports and outports, a compiler error indicated that variable `HDG_Mode` is not declared. You can fix that error by overriding the default configuration for Inport block `HDG_Mode`.

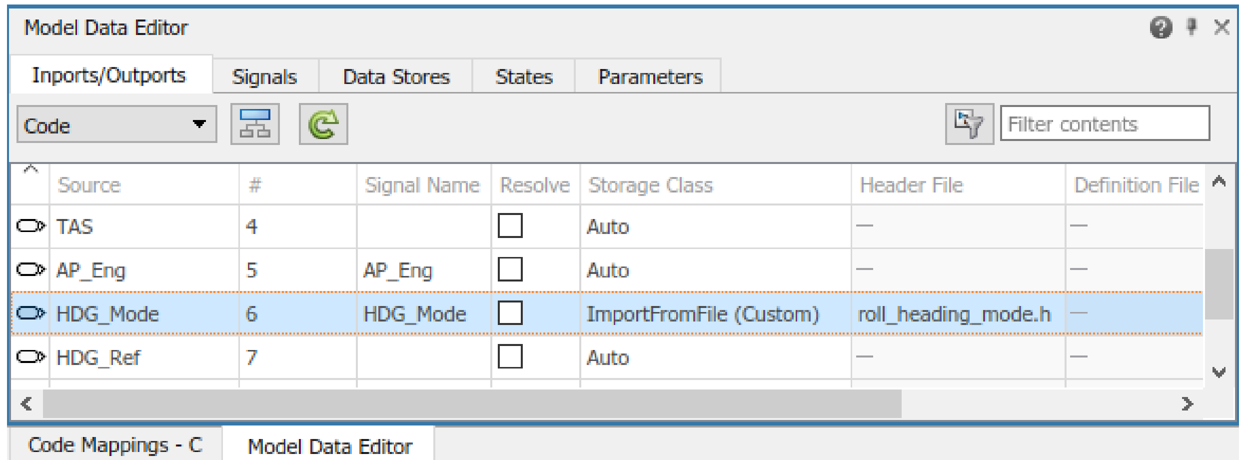
1 Open the Model Data Editor. In the Code perspective, under the Code Mappings editor, click the **Model Data Editor** tab.



2 In the Model Data Editor, on the **Inports/Outports** tab, select source `HDG_Mode`.



3 Set **Storage Class** to `ImportFromFile` and **Header File** to `roll_heading_mode.h`.



Based on these settings, the code generator imports the declaration for external variable `HDG_Mode` from header file `roll_heading_mode.h`.

```
extern boolean_T HDG_Mode;
```

4 Save the model and regenerate the code.

   Minimize the code generation report window for exploration later in this tutorial.

5 In the Code view, search for `roll_heading_mode.h` and confirm that it is included in the generated code with the default configuration file `roll_input_data.h`.

6 Search for `HDG_Mode` and confirm that it is defined as a separate global variable.

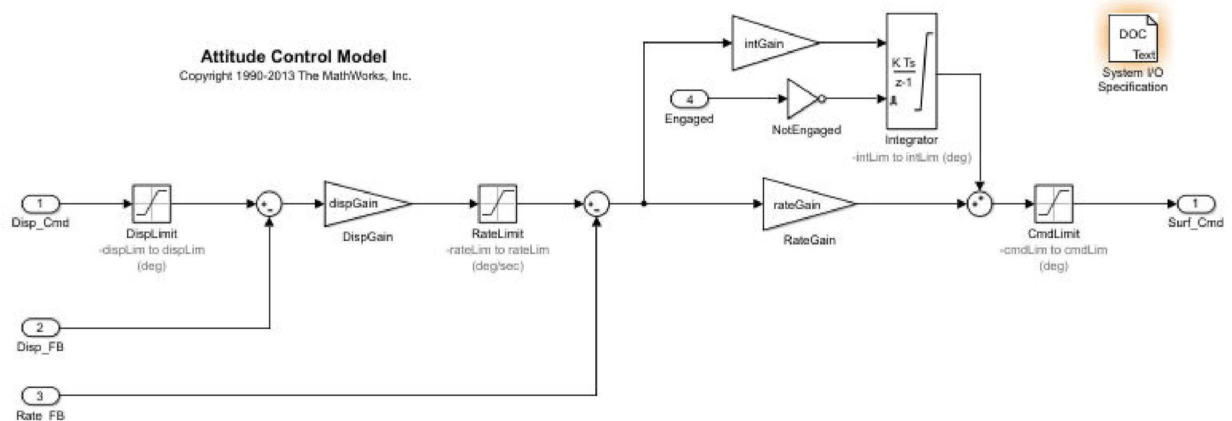
Next, configure a model parameter to be a global variable in the generated code. As a global variable, you can tune the parameter value at run time.

Configure a Model Parameter as a Global Variable for Tuning During Run Time

By default, code generation optimizations eliminate storage for model parameters and most signals that do not participate in the entry-point function interface. To make parameters tunable and related signals accessible, identify them by configuring them explicitly.

In the `BasicRollMode` subsystem of model `rtwdemo_roll`, configure a PID control parameter to appear in the code as a global variable whose value you can tune.

1. Open the `BasicRollMode` subsystem.



2. In the Model Data Editor, select the **Parameters** tab.
3. In the filter field, type `IntGain`. The Model Data Editor shows a row that corresponds to the **Gain** parameter and a row that corresponds to a workspace variable.

Model Data Editor

Inports/Outports | Signals | Data Stores | States | Parameters

Code | IntGain

Source	Name	Argument	Storage Class	Header File	Definition File	Get Function	Set Function	Struct Name
IntGain	Gain	—	—	—	—	—	—	—
Model Work...	intGain	☐	Auto	—	—	—	—	—

4 In the **Source** column, click `IntGain`. That Gain block appears highlighted in the model diagram.

5 In the **Name** column, click the model workspace variable `intGain`.

6 Convert the model workspace variable to a parameter object. In the **Storage Class** column, select `Convert to parameter object`. The **Storage Class** setting changes to `Model default`, which indicates that the parameter object prevents code generation optimizations from eliminating storage for the variable. With this setting, the object uses the storage class specified in the Code Mappings editor as the data default for category **Local parameters**.

7 Save the model and regenerate the code.

   Minimize the code generation report window for exploration later in this tutorial.

8 In the Code view:

   - Search for `intGain`.
   - In `rtwdemo_roll.c`, place your cursor over the ellipsis menu over the `P` in the highlighted code `P.intGain`. In the model editor, notice that the Code view highlights the block corresponding to the generated code.

```
152    *  Gain: '<S1>/RateGain'
153    */
154   rtb_TKSwitch = 2.0F * rtb_
155
156   /* Update for DiscreteInte
157    *  Gain: '<S1>/IntGain'
158    */
159   DW_l.Integrator_DSTATE += P.intGain * rtb_Xnew * 0.025F;
160   if (DW_l.Integrator_DSTATE >= 5.0F) {
161     DW_l.Integrator_DSTATE = 5.0F;
162   } else {
163     if (DW_l.Integrator_DSTATE <= -5.0F) {
164       DW_l.Integrator_DSTATE = -5.0F;
```

- To see the parameter object definition for `intGain` in `rtwdemo_roll_data.c`, click the definition code in the dialog box.

```
25   /* Block parameters (default storage) */
26   P_e P = {
27     /* Variable: intGain
28      * Referenced by: '<S1>/IntGain'
29      */
30     0.5F
31   };
```

The code that you generate from the model stores the parameter object in memory. Because you left the default storage class settings in the Code Mapping Editor for category **Local parameters** set to `Default`, the code generator determines the storage format, for example, as fields of structures.

Next, use a test harness model and software-in-the-loop (SIL) simulation to compare results of model simulation and generated code.

Compare Model Simulation and Generated Code Results

In this section...
“Inspect and Configure Test Harness Model” on page 3-21
“Simulate the Model in Normal Mode” on page 3-22
“Simulate the Model in SIL Mode” on page 3-24
“Compare Simulation Results” on page 3-24

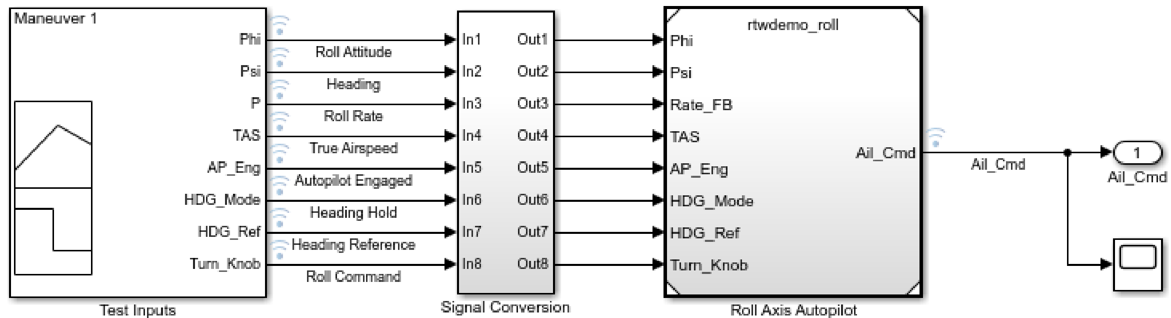
In this step of the tutorial, you verify that when executed, the code is numerically equivalent to the algorithm modeled in Simulink. You use a test harness model to simulate `rtwdemo_roll` in normal mode and in SIL mode, then compare the simulations by using the Simulation Data Inspector.

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. The PIL simulation 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 numeric behavior of your code.
- Collect code coverage and execution-time metrics.
- Optimize your code.
- Progress toward achieving IEC 61508, IEC 62304, ISO 26262, EN 50128, or DO-178 certification.

Inspect and Configure Test Harness Model

Model `rtwdemo_roll_harness` references the model-under-test, `rtwdemo_roll`, through a Model block. The harness model generates test inputs for the referenced model. You can easily switch the Model block between the normal, SIL, or PIL simulation modes.

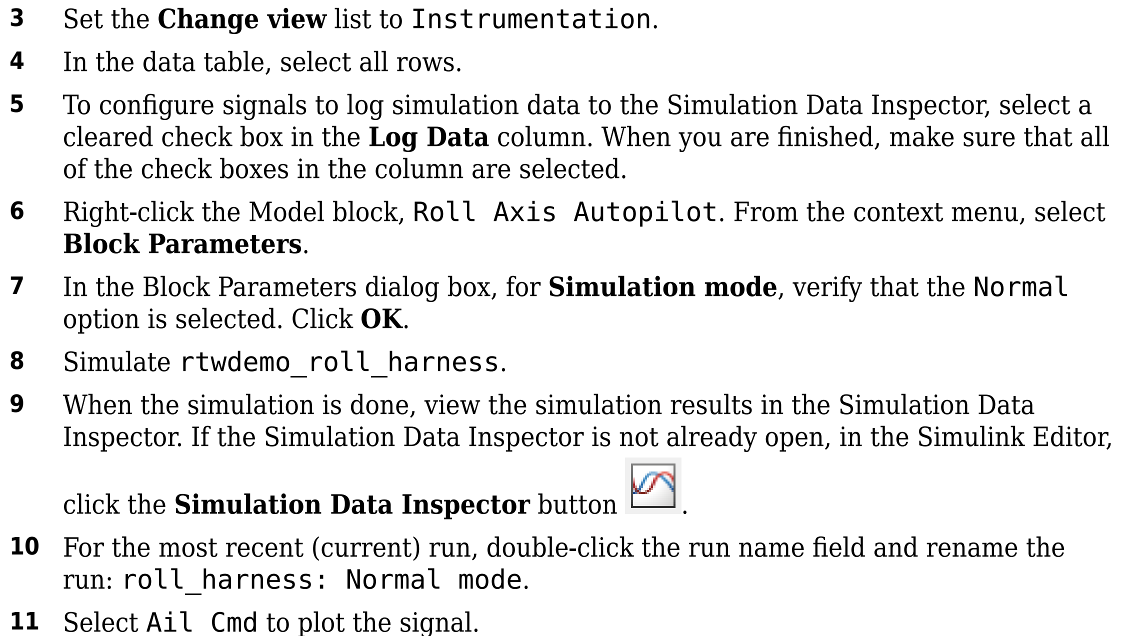


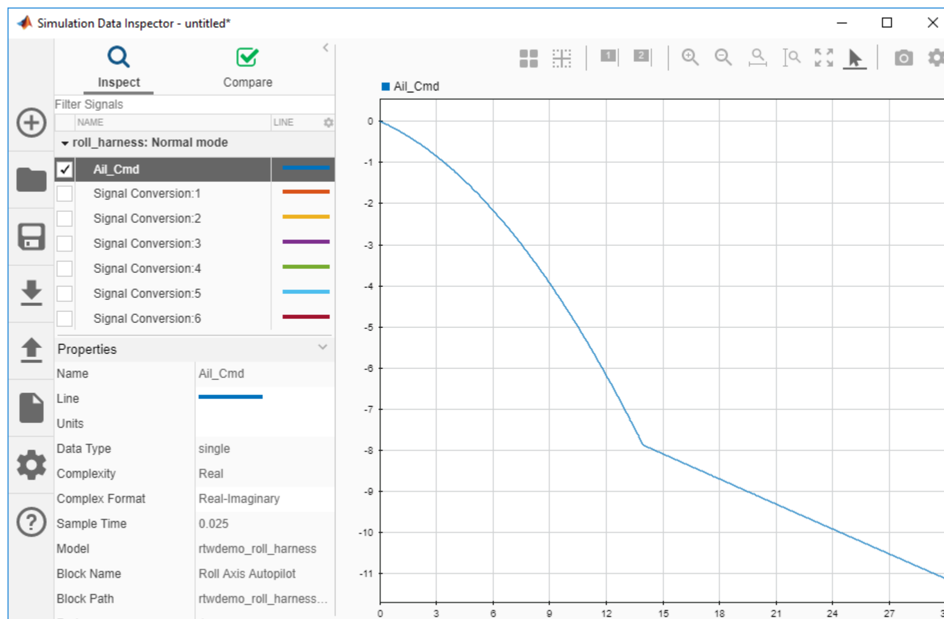
1 Open model `rtwdemo_roll_harness`. If you closed your copy of model `rtwdemo_roll`, reopen it.
2 In the `rtwdemo_roll_harness` model, right-click the Model block and select **Subsystem & Model Reference > Refresh Selected Model Block**.
3 Save a copy of `rtwdemo_roll_harness` in the current working folder.
4 Open the Configuration Parameters dialog boxes for `rtwdemo_roll_harness` and `rtwdemo_roll`.
5 For both models, on the **Code Generation** pane, verify that parameter **Generate code only** is cleared to run SIL and PIL simulations.
6 For both models, on the **Hardware Implementation** pane, expand **Device details**. Verify that **Support long long** is selected.
7 Click **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 In the `rtwdemo_roll_harness` model, select **View > Model Data Editor**.
2 In the Model Data Editor, select the **Signals** tab.

3 Set the **Change view** list to `Instrumentation`.
4 In the data table, select all rows.
5 To configure signals to log simulation data to the Simulation Data Inspector, select a cleared check box in the **Log Data** column. When you are finished, make sure that all of the check boxes in the column are selected.
6 Right-click the Model block, `Roll Axis Autopilot`. From the context menu, select **Block Parameters**.
7 In the Block Parameters dialog box, for **Simulation mode**, verify that the `Normal` option is selected. Click **OK**.
8 Simulate `rtwdemo_roll_harness`.
9 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 .
10 For the most recent (current) run, double-click the run name field and rename the run: `roll_harness: Normal mode`.
11 Select `Ail_Cmd` 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.

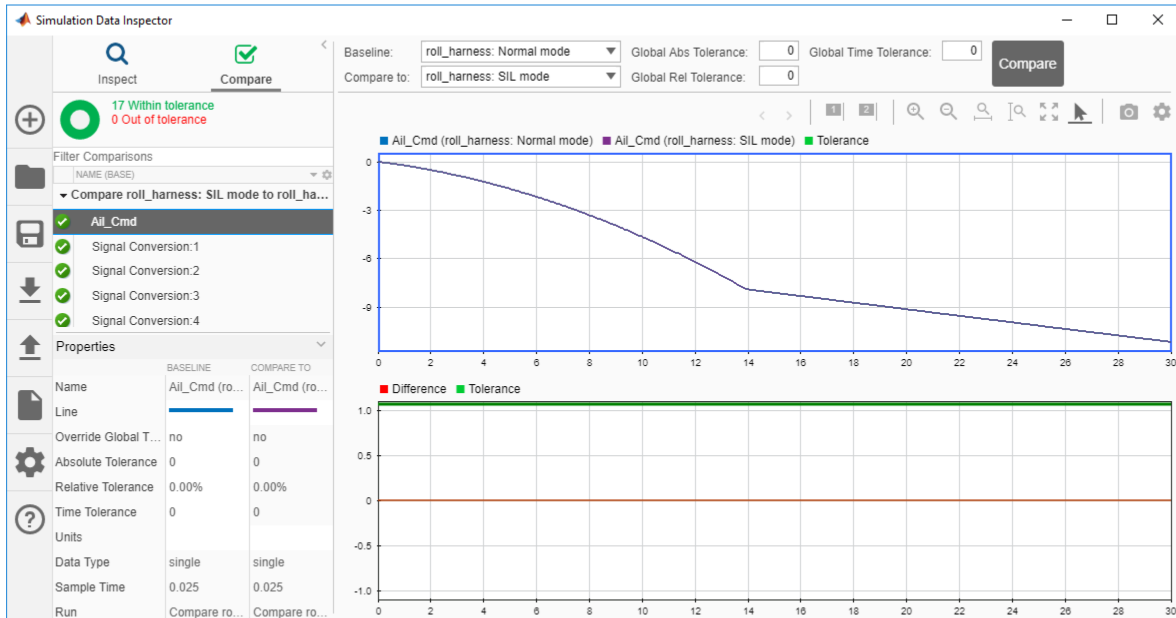
1. In the `rtwdemo_roll_harness` model window, right-click the `Roll Axis Autopilot` model block and select **Block Parameters**.
2. In the Block Parameters dialog box, set **Simulation mode** to `Software-in-the-loop (SIL)` and **Code Interface** to `Top model`. Click **OK**.
3. Exclude external code files from the build process. In the Configuration Parameters dialog box for model `rtwdemo_roll`, set **Code Generation** > **Custom Code** > **Additional build information** > **Source files** to the default value, which is empty. Save the model.
4. Simulate the `rtwdemo_roll_harness` model.

   Minimize the code generation report window for exploration later in this tutorial.
5. In the Simulation Data Inspector, double-click the run name field and rename the new run as `roll_harness: SIL mode`.
6. Select `Ail_Cmd` to plot the signal.
7. Reconfigure the build process for model `rtwdemo_roll` to include the external source files `roll_input_data.c` and `roll_heading_mode.c`. In the Model Configuration Parameters dialog box, set **Code Generation** > **Custom Code** > **Additional build information** > **Source files** to `roll_input_data.c roll_heading_mode.c`. Click **Apply**, close the dialog box, and save the model.

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**.



The Simulation Data Inspector shows that the normal mode and SIL mode results match. Comparing the results of normal mode simulation with SIL and PIL simulations can help you verify that the generated application performs as expected.

Next, explore ways that you can deploy generated code.

Deploy the Generated Code

In this step of the tutorial, you explore mechanisms for deploying the generated code.

Example Main Program

To facilitate deployment of the generated code, the code generator produces an example main program that you can use to get started. The example main program is in the file ert_main.c. To use the algorithmic code (the model entry-point functions) generated for your application, you can copy the incomplete functions defined in ert_main.c, and then complete the functions by inserting your custom scheduling code.

Explore the example main program generated for model rtwdemo_roll.

1. If not already open, open your copy of the model rtwdemo_roll and enable the Simulink Editor Code perspective. In the Simulink Editor, select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**.
2. Regenerate the code.
3. In the Code perspective Code view, select file ert_main.c.
4. Click in the **Search** field and select function rt_OneStep.
5. Explore the incomplete wrapper function rt_OneStep. This function calls the model execution entry-point function, rtwdemo_roll_step. Your application code can call rt_OneStep to run the model algorithm during each execution cycle.
6. Click in the **Search** field and select function main.
7. Explore the incomplete example main function. This function outlines the order and context in which your application code can call rt_OneStep and other model entry-point functions.

For more information, see “Deploy Generated Standalone Executable Programs To Target Hardware”.

Relocate Generated Code Files

Embedded Coder provides a pack-n-go utility for relocating static and generated code files for a model to another development environment. File relocation is necessary when your system or integrated development environment (IDE) does not include MATLAB and Simulink products. The utility packages the files in a compressed file that you can

relocate and unpack by using a standard `zip` utility. You can apply the pack-n-go utility from graphical and programming interfaces. For more information, see “Relocate Code to Another Development Environment”.

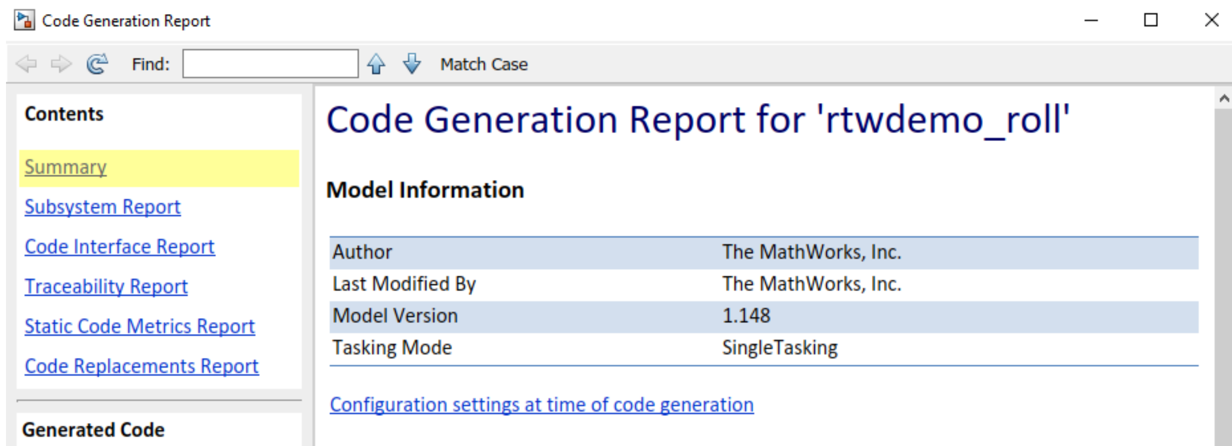
Share and Archive Code Generation Report

The Quick Start tool configures a model to produce an HTML code generation report. In addition to a summary of model and code information, the report includes:

- A subsystem report
- Generated code files
- A code interface report
- A traceability report
- A static code metrics report
- A code replacements report
- Optionally, a model web view

You can use this report outside of the Simulink environment, so it is suitable for sharing or for archival purposes. You can open the report from the tool or, later, select **Code > C/C++ Code > Code Generation Report > Open Model Report**.

The default location for the code generation report files is in the `html` subfolder of the build folder, *model_target*_rtw/html/. *target* is the name of the **System target file** specified on the **Code Generation** pane. The default name for the top-level HTML report file is *model*_codegen_rpt.html or *subsystem*_codegen_rpt.html.



Explore Other Options

Use these links to explore more ways to customize, verify, and deploy generated production code.

Task	Reference
Quickly generate readable, efficient code from your model	"Generate Code by Using the Quick Start Tool"
Consider model design and configuration for code generation	"Design Models for Generated Embedded Code Deployment"
Learn about generated entry-point functions	"Configure Code Generation for Model Entry-Point Functions"
Achieve code reuse	"Choose a Componentization Technique for Code Reuse"
Specify default configurations for categories of data elements and functions across a model	"Configure Default C Code Generation for Categories of Model Data and Functions"
Override default configurations for individual entry-point functions	"Override Default Naming for Individual C Entry-Point Functions" and "Override Default C Step Function Interface"

Task	Reference
Override default configurations for individual data elements	“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” and “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements”
Compare normal mode simulation results against software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation results for numerical equivalency	“SIL and PIL Simulations” and “Choose a SIL or PIL Approach”
Collect code coverage metrics for generated code during SIL or PIL simulation	“Code Coverage”
Use generated example main code as a starting point to deploy generated executable programs	“Deploy Generated Standalone Executable Programs To Target Hardware”