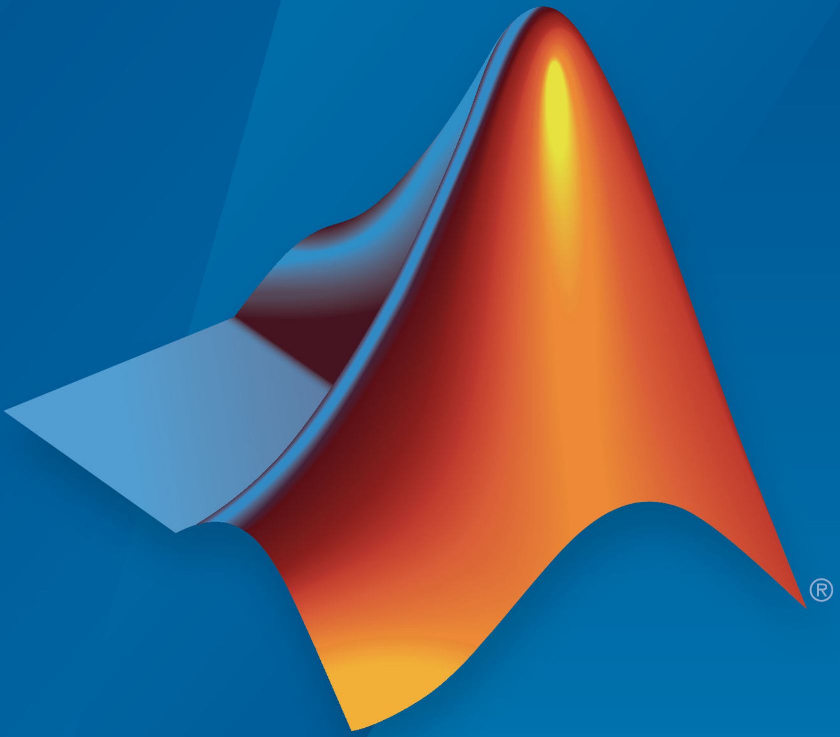
Embedded Coder®

Getting Started Guide



MATLAB®&SIMULINK®

R2018a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Embedded Coder® Getting Started Guide

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)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

1 Product Overview

2 MATLAB Tutorials

Product Overview

Embedded Coder Product Description

Generate C and C++ code optimized for embedded systems

Embedded Coder generates readable, compact, and fast C and C++ code for 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.

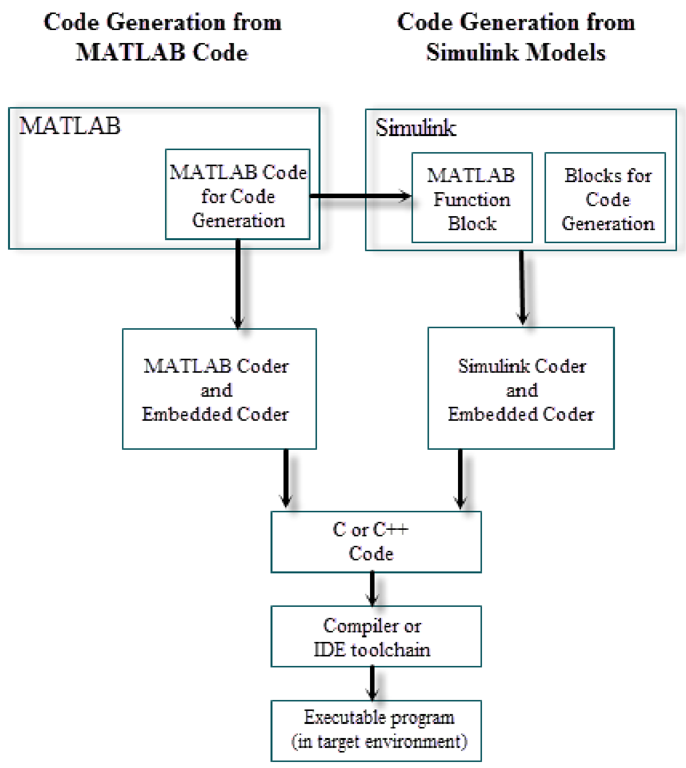
To learn model design patterns that include Simulink blocks, Stateflow® charts, and MATLAB functions, and map to commonly used C constructs, see “Modeling Patterns for C Code”.

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, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize generated code for a specific target environment.
- Integrate existing applications, functions, and data.
- Enable tracing, reporting, and testing options that facilitate code verification activities.

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


Code Generation from MATLAB Code
Code Generation from Simulink Models
MATLAB
MATLAB Code for Code Generation
Simulink
MATLAB Function Block
Blocks for Code Generation
MATLAB Coder and Embedded Coder
Simulink Coder and Embedded Coder
C or C++ Code
Compiler or IDE toolchain
Executable program (in target environment)

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

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

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

Code Generation from MATLAB Code

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

- MATLAB
- MATLAB Coder
- Embedded Coder

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

For more information about generating code from MATLAB code, see "MATLAB Code for Code Generation Workflow Overview" (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

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

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

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

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

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

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

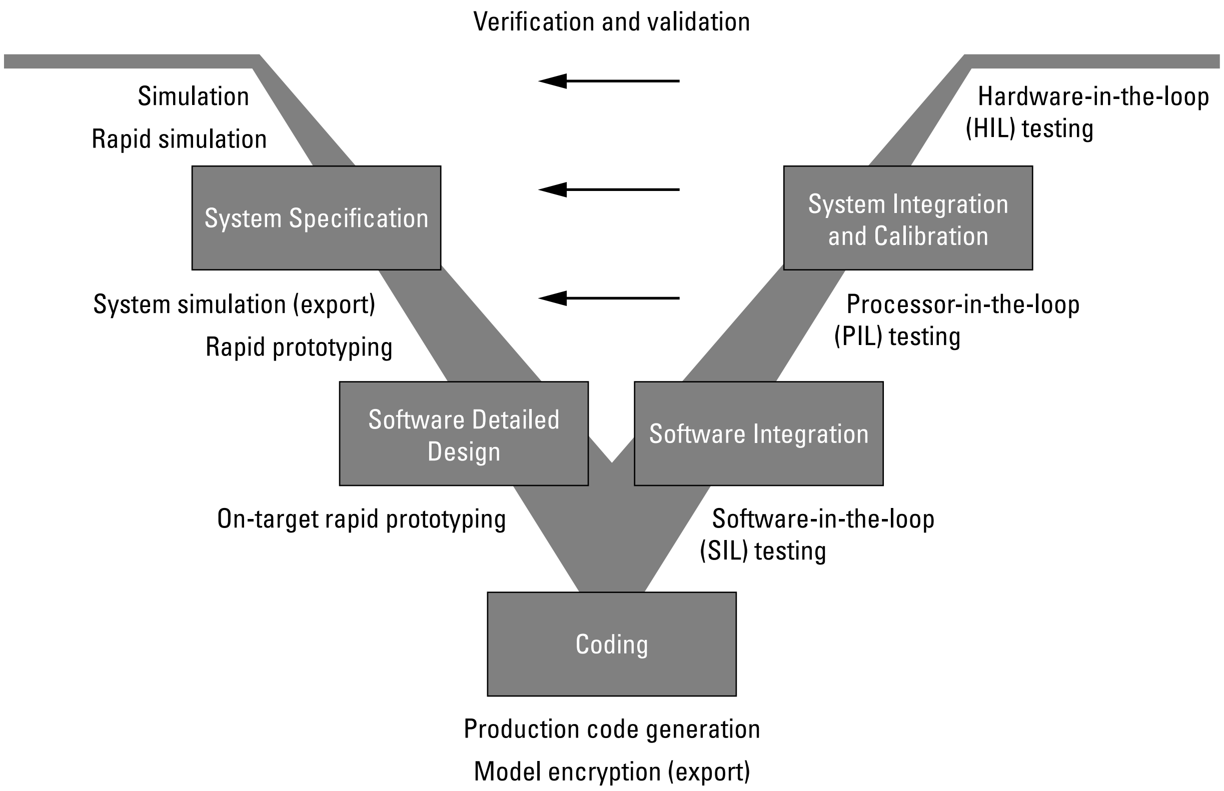
Validation and Verification for System Development

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

In this section...

V-Model for System Development

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


Verification and validation
Simulation
Rapid simulation
System Specification
System simulation (export)
Rapid prototyping
Software Detailed Design
On-target rapid prototyping
Coding
Production code generation
Model encryption (export)
Software-in-the-loop (SIL) testing
Software Integration
Processor-in-the-loop (PIL) testing
System Integration and Calibration
Hardware-in-the-loop (HIL) testing

Depending on your application and its role in the process, you might focus on one or more of the steps called out in the V-model or repeat steps at several stages of the V-model. Code generation technology and related products provide tooling that you can apply to the V-model for system development. For more information about how you can apply MathWorks code generation technology and related products provide tooling to the V-model process, see:

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

Types of Simulation and Prototyping in the V-Model

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

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
Purpose	Test and validate functionality of concept model	Refine, test, and validate functionality of concept model in nonreal time	Test new ideas and research	Refine and calibrate designs during development process
Execution hardware	Host computer	Host computer Standalone executable runs outside of MATLAB and Simulink environments	PC or nontarget hardware	Embedded computing unit (ECU) or near-production hardware
Code efficiency and I/O latency	Not applicable	Not applicable	Less emphasis on code efficiency and I/O latency	More emphasis on code efficiency and I/O latency

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
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

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

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

	SIL Testing	PIL Testing on Embedded Hardware	PIL Testing on Instruction Set Simulator	HIL Testing
Fidelity and accuracy	Two options: Same source code as target, but might have numerical differences Changes source code to emulate word sizes, but is bit accurate for fixed-point math	Same object code Bit accurate for fixed-point math Cycle accurate because code runs on hardware	Same object code Bit accurate for fixed-point math Might not be cycle accurate	Same executable code Bit accurate for fixed-point math Cycle accurate Use real and emulated system I/O
Execution platforms	Host	Target	Host	Target
Ease of use and cost	Desktop convenience Executes only in Simulink Reduced hardware cost	Executes on desk or test bench Uses hardware — process board and cables	Desktop convenience Executes only on host computer with Simulink and integrated development environment (IDE) Reduced hardware cost	Executes on test bench or in lab Uses hardware — processor, embedded computer unit (ECU), I/O devices, and cables
Real-time capability	Not real time	Not real time (between samples)	Not real time (between samples)	Hard real time

Mapping of Code Generation Goals to the V-Model

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

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

Documenting and Validating Requirements

Goals	Related Product Information	Examples
Capture requirements in a document, spreadsheet, data base, or requirements management tool	“Simulink Report Generator” Third-party vendor tools such as Microsoft® Word, Microsoft Excel®, raw HTML, or IBM® Rational® DOORS®	
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 host system	“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 Distributed Computing Server” “Parallel Computing Toolbox”	

Developing a Model Executable Specification

Goals	Related Product Information	Examples
Produce design artifacts for algorithms that you develop in MATLAB code for reviews and archiving	“MATLAB Report Generator”	
Produce design artifacts from Simulink and Stateflow models for reviews and archiving	“System Design Description” (Simulink Report Generator)	`rtwdemo_codegenrpt`
Add one or more components to another environment for system simulation Refine a component model Refine an integrated system model Verify functionality of a model in nonreal time Test a concept model	“Deploy Algorithm Model for Real-Time Rapid Prototyping” (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 systems	“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”
		`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)

Developing a 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`
		"Sample Workflows"

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	"Run Model 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	"Run Model Checks" (Simulink)	
Check a model against industry standards and guidelines (MathWorks Automotive Advisory Board (MAAB), IEC 61508, IEC 62304, ISO 26262, EN 50128 and DO-178)	"Standards, Guidelines, and Block Usage" "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) “Compare and Merge Simulink Models” (Simulink)	`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`
		“Sample Workflows”

Integrate existing externally written C or C++ code with your model for simulation and code generation	“Block Creation” (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`

Generating the Application Code

Goals	Related Product Information	Examples
Optimize generated ANSI® C code for production (for example, disable floating-point code, remove termination and error handling code, and combine code entry points into single functions)	"Performance" (Simulink Coder) "Performance"	`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 Creation" (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)	“Simulate 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.

Integrating and Verifying Software

Goals	Related Product Information	Examples
Integrate existing externally written C or C++ code with a model for simulation and code generation	"Block Creation" (Simulink) "External Code Integration"	`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` “Sample Workflows”
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 Creation” (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

Integrating, Verifying, and Calibrating System Components

Goals	Related Product Information	Examples
Integrate the software and its microprocessor with the hardware environment for the final embedded system product Add the complexity of the environment (or plant) under control to the test platform Test and verify the embedded system or control unit by using a real-time target environment	“Deploy Algorithm Model for Real-Time Rapid Prototyping” (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”	“Simulink Real-Time Examples” (Simulink Real-Time)
Tune ECU properly for its intended use	“Data Exchange Interfaces” (Simulink Coder) “Data Exchange Interfaces”	`rtwdemo_capi` `rtwdemo_asap2`
Generate ASAP2 data files	“Export ASAP2 File for Data Measurement and Calibration” (Simulink Coder)	`rtwdemo_asap2`

Goals	**Related Product Information**	**Examples**
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

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

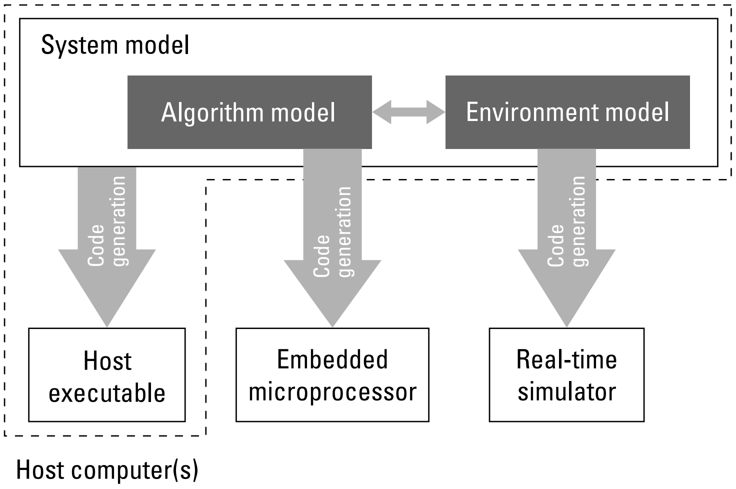
a. UNIX is a registered trademark of The Open Group in the United States and other countries.
b. Linux is a registered trademark of Linus Torvalds.

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.
Model Protection (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.

MATLAB Tutorials

Embedded Coder Capabilities for Code Generation from MATLAB Code

The Embedded Coder product extends the MATLAB Coder product with features that are important for embedded software development. Using the Embedded Coder add-on product, you can generate code that has the clarity and efficiency of professional handwritten code. For example, you can:

- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize the generated code for application-specific requirements.
- Enable tracing options that help you to verify the generated code.

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	`CodeReplacement-Library`	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 2-14
Generate MATLAB function help text in the function banner	**MATLAB function help text**	`MATLABFcnDesc`	"Include Comments in Generated C/C++ Code" on page 2-14
Convert if-elseif-else patterns to switch-case statements	**Convert if-elseif-else patterns to switch-case statements**	`ConvertIfToSwitch`	"Controlling C Code Style" on page 2-8
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”
Customize generated global variable identifiers	**Global variables**	`CustomSymbolStr-GlobalVar`	“Customize Generated Identifiers”
Customize generated global type identifiers	**Global types**	`CustomSymbolStrType`	“Customize Generated Identifiers”
Customize generated field names in global type identifiers	**Field name of global types**	`CustomSymbolStrField`	“Customize Generated Identifiers”
Customize generated local functions identifiers	**Local functions**	`CustomSymbolStrFcn`	“Customize Generated Identifiers”
Customize generated identifiers for local temporary variables	**Local temporary variables**	`CustomSymbolStr-TmpVar`	“Customize Generated Identifiers”
Customize generated identifiers for constant macros	**Constant macros**	`CustomSymbolStrMacro`	“Customize Generated Identifiers”

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

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

Goal	Project Setting	Code Configuration Object Property	More Information
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"
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

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**.

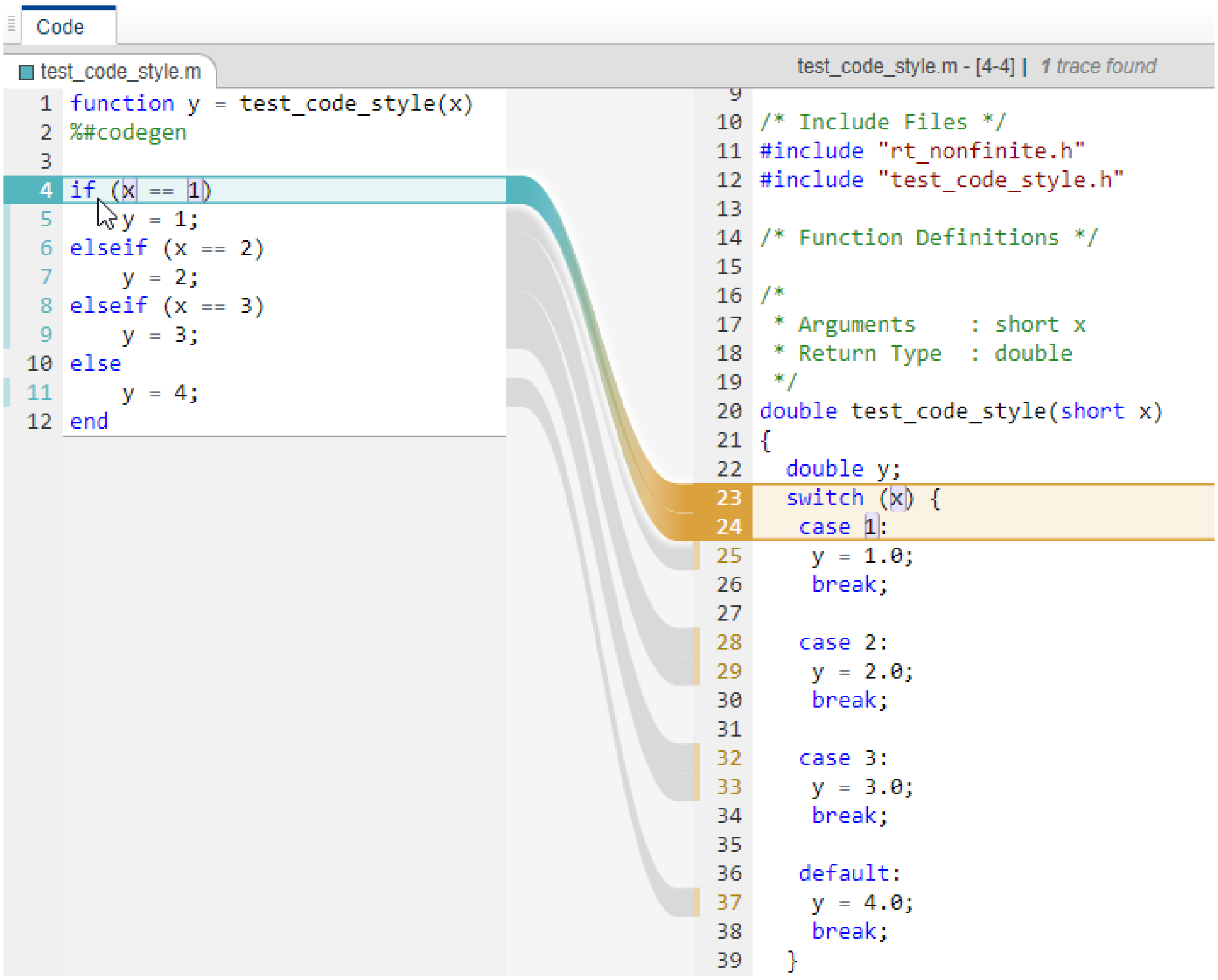
When code generation is complete, the code generator produces a C static library, `test_code_style.lib`, and C code in the `/codegen/lib/test_code_style` subfolder. The code generator provides a link to the report.

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.



```
function y = test_code_style(x)
%#codegen

if (x == 1)
    y = 1;
elseif (x == 2)
    y = 2;
elseif (x == 3)
    y = 3;
else
    y = 4;
end
```

```
/* Include Files */
#include "rt_nonfinite.h"
#include "test_code_style.h"

/* Function Definitions */

/*
 * Arguments    : short x
 * Return Type  : double
 */
double test_code_style(short x)
{
  double y;
  switch (x) {
   case 1:
    y = 1.0;
    break;

   case 2:
    y = 2.0;
    break;

   case 3:
    y = 3.0;
    break;

   default:
    y = 4.0;
    break;
  }
```

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

- “C Code Generation Using the MATLAB Coder App” (MATLAB Coder)
- “C Code Generation 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

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.

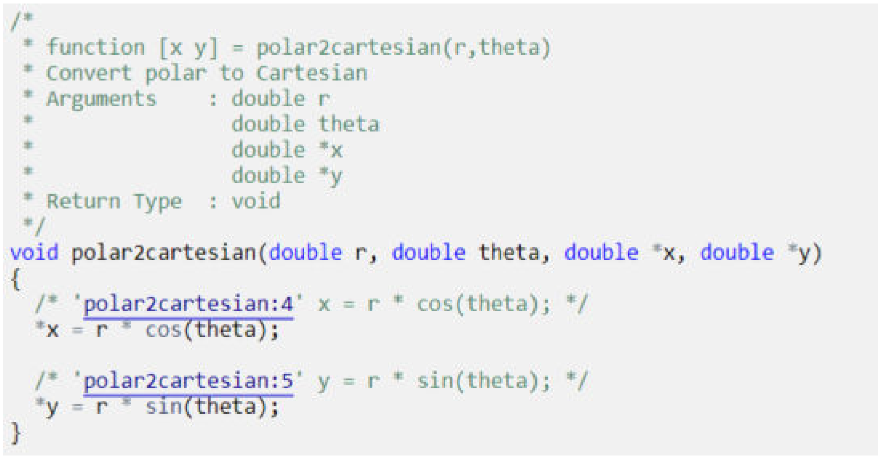
Viewing the Generated C Code

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.



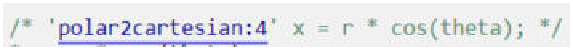
```
/*
 * function [x y] = polar2cartesian(r,theta)
 * Convert polar to Cartesian
 * Arguments    : double r
 *                double theta
 *                double *x
 *                double *y
 * Return Type  : void
 */
void polar2cartesian(double r, double theta, double *x, double *y)
{
  /* 'polar2cartesian:4'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

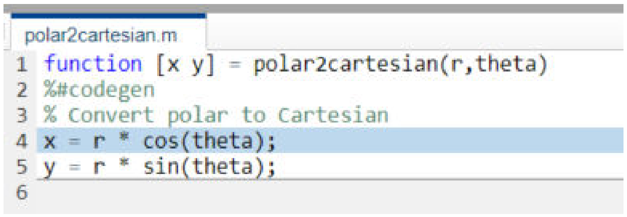
Tracing the Generated Code to theMATLAB Code

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.



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

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”

Simulink Code Generation Tutorials

Generate C Code from Simulink Models

The code generator produces readable, compact, and fast C and C++ code for use on embedded processors, rapid prototyping boards, and microprocessors used in mass production. You can generate code for a wide variety of applications. In this series of examples, you use Embedded Coder for real-time deployment of a discrete-time control system.

Prerequisites

To complete these examples, you must have:

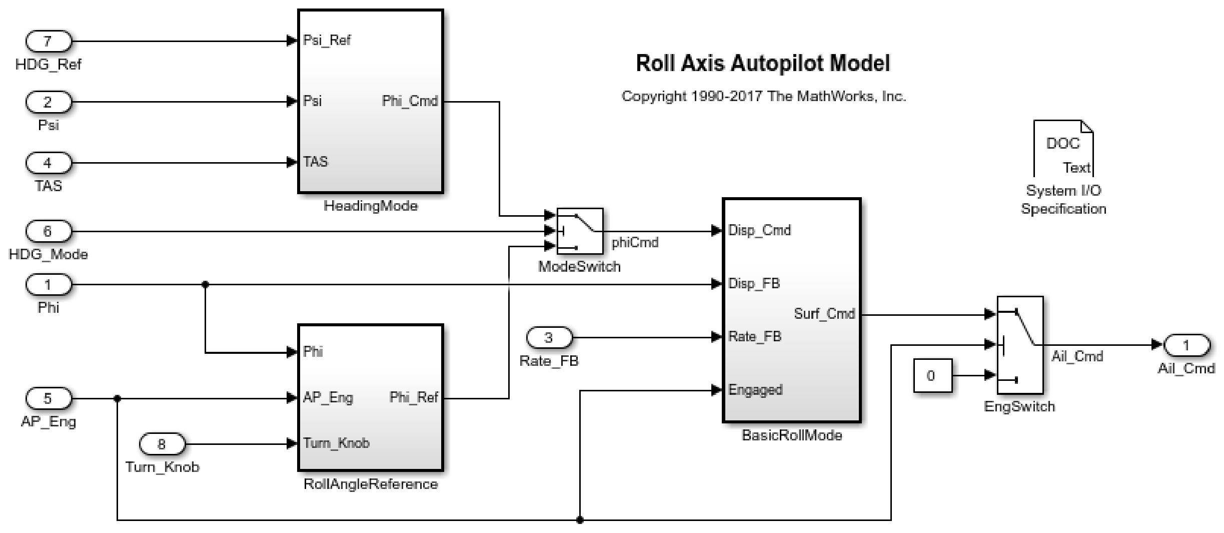
- MATLAB
- MATLAB Coder
- Simulink
- Simulink Coder
- Embedded Coder

Example Models

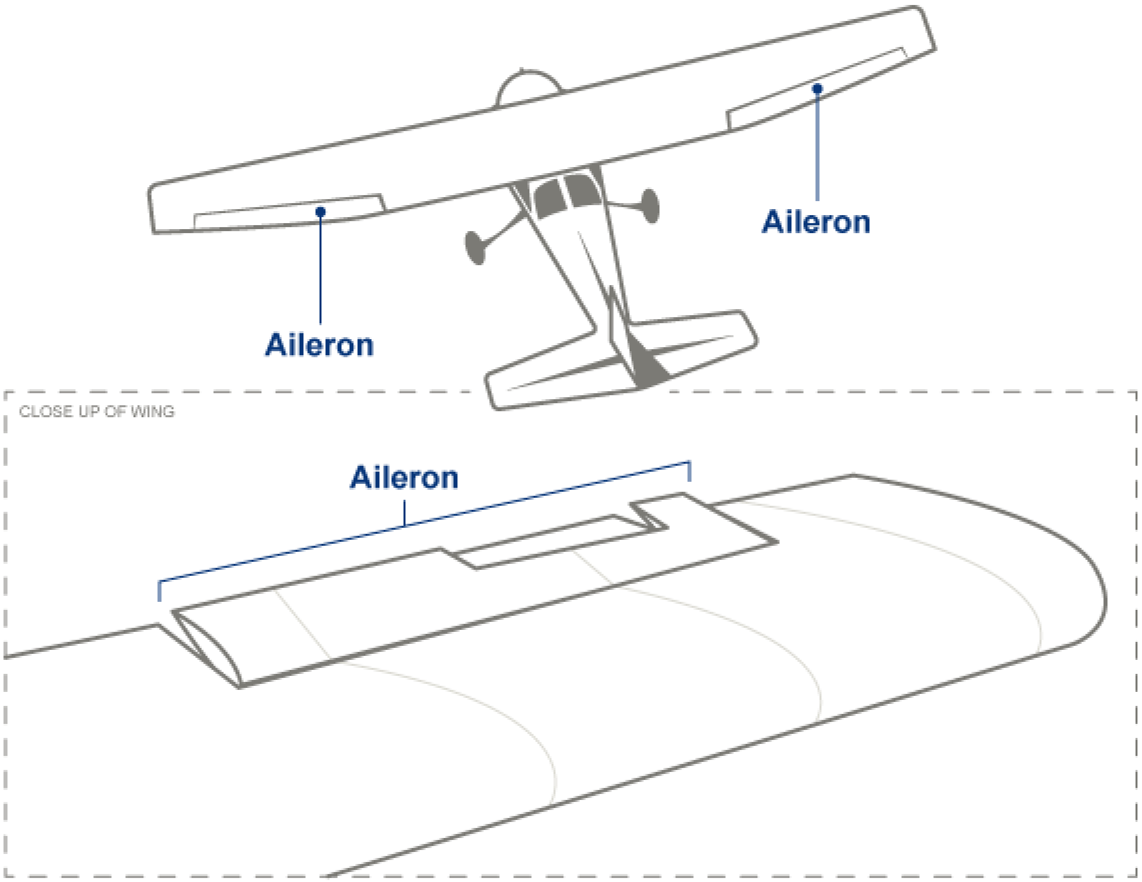
Functionality in your model must be traceable back to your model requirements. You can use reviews, analysis, simulations, and requirements-based tests to prove that your design meets your original requirements and does not contain unintended functionality. Performing verification and validation activities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

The examples in this series use the `rtwdemo_roll` model, which has been verified for simulation. To open the model in the Simulink Editor, at the command prompt, enter:

```
rtwdemo_roll
```



The `rtwdemo_roll` 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 the entire aircraft. Through the `HDG_Mode` signal, the control system places the model in one of two operating modes: roll attitude hold or heading hold. Each of the `RollAngleReference` and `HeadingMode` subsystems calculates 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 examples in this series also use the model `rtwdemo_roll_harness`, which is a harness model to test `rtwdemo_roll`.

The examples show how to:

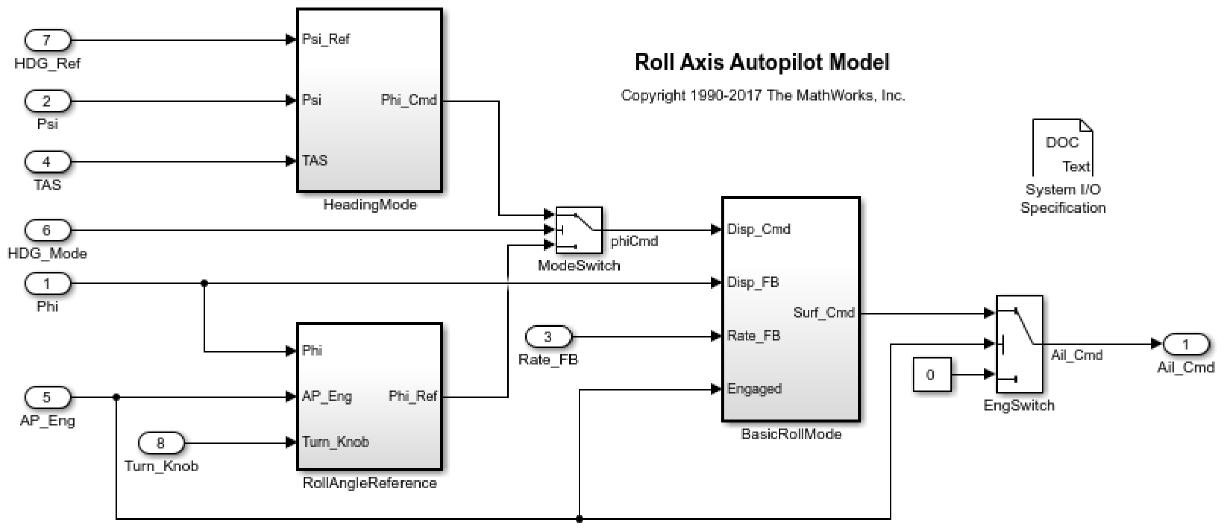
1. Prepare a model for initial code generation. See “Prepare a Model for Embedded Code Generation” on page 3-6.
2. Inspect and understand the generated code. See “Inspect and Analyze Generated Code” on page 3-10.
3. Configure the interfaces of the generated entry-point functions, which helps you to integrate the generated code with external code. See “Customize Interfaces of Generated Entry-Point Functions” on page 3-18.
4. Modularize and partition the code into separate files and functions. See “Partition and Modularize Generated Code” on page 3-27.
5. Verify that executing the code yields expected numeric results. See “Deploy and Test Executable Program” on page 3-30.

Prepare a Model for Embedded Code Generation

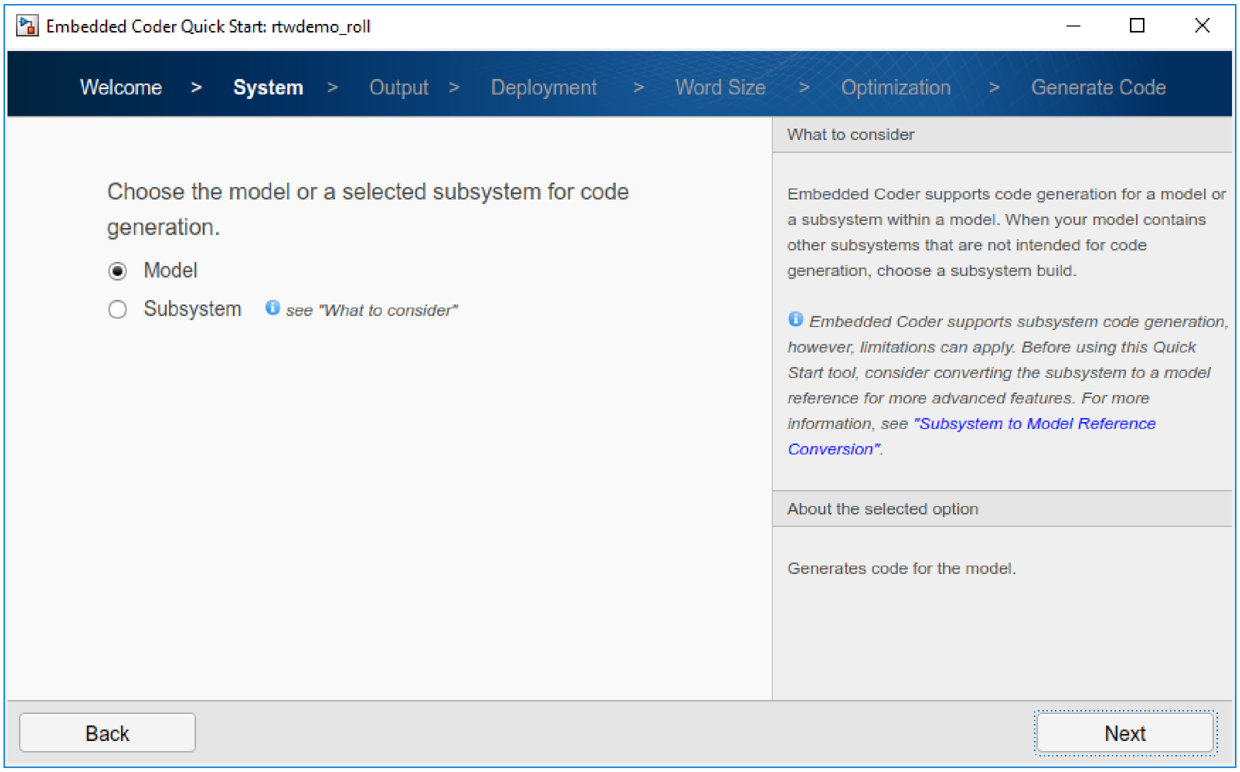
The example model `rtwdemo_roll` described in “Generate C Code from Simulink Models” on page 3-2 represents an autopilot control system for an aircraft. In this example, you prepare `rtwdemo_roll` for embedded code generation by using Embedded Coder Quick Start, which chooses fundamental code generation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

1. Open the example model.

   ```
   rtwdemo_roll
   ```



2. Set your MATLAB current folder to a writable location.
3. Open the Quick Start tool. To do so, in the model window, select **Code** > **C/C++ Code** > **Embedded Coder Quick Start**.
4. In the tool, advance to the **System** page by clicking **Next**. On this page, the default setting, **Model**, means that the tool prepares the entire model for code generation.


Embedded Coder Quick Start: rtwdemo_roll
Welcome > System > Output > Deployment > Word Size > Optimization > Generate Code
Back
Next

Choose the model or a selected subsystem for code generation.

(● Model) ○ Subsystem — see "What to consider"

What to consider

Embedded Coder supports code generation for a model or a subsystem within a model. When your model contains other subsystems that are not intended for code generation, choose a subsystem build.

Embedded Coder supports subsystem code generation, however, limitations can apply. Before using this Quick Start tool, consider converting the subsystem to a model reference for more advanced features. For more information, see "Subsystem to Model Reference Conversion".

About the selected option

Generates code for the model.

5 Advance to the **Output** page. On this page:

- The default setting **C code** means that the code generator yields C code (as opposed to, for example, C++ code).
- The default setting **Single instance** means that the code generator yields nonreentrant code, so your application code can maintain only one instance of the model.

6 Advance to the **Deployment** page and run a model analysis by clicking **Next**.

With the results of the analysis, the tool chooses settings related to scheduling and solvers. In this case, the tool finds that `rtwdemo_roll` is a rate-based model with only one sample rate, 0.025 seconds.

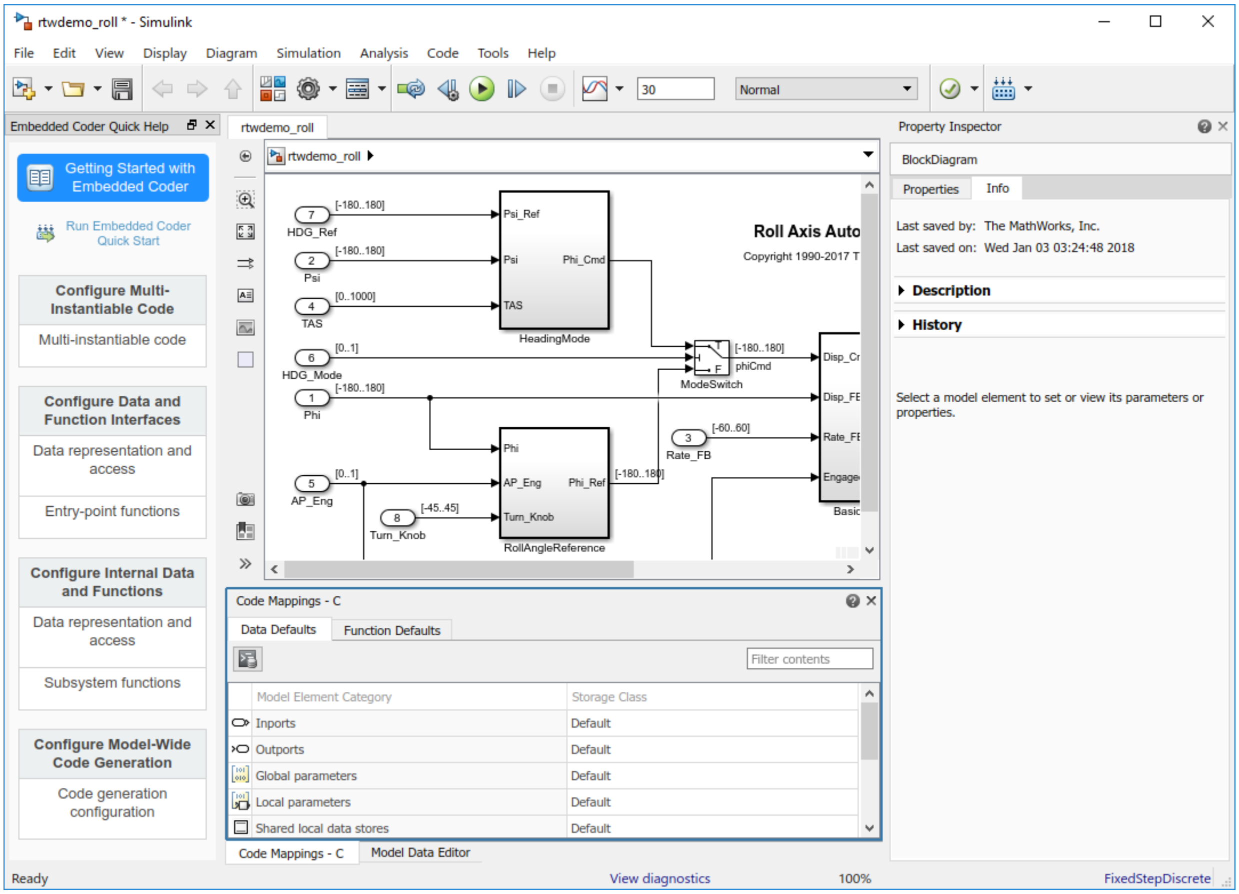
7 Advance to the **Word Size** page.

For this example, you use the default settings for hardware implementation parameters. You can specify these parameters according to the characteristics of your target hardware. The code generator then customizes the code for the hardware.

8. Advance to the **Optimization** page. On this page, the default setting, **Execution efficiency**, means that the tool chooses settings so that the generated code executes more quickly. This optimization strategy can lead to increased memory consumption.

   Later, you can manually inspect code metrics that estimate the amount of memory the code consumes, and adjust the optimization strategy to favor other computing resources.
9. Advance to the **Generate Code** page. On this page, inspect the proposed changes to the model configuration parameters. The changes are based on the options that you chose on previous pages, such as **Execution efficiency** on the **Optimization** page, and on typical requirements for embedded system code.
10. Apply the changes and generate code from `rtwdemo_roll` by clicking **Next**.
11. Close the tool by clicking **Finish**.

    The tool places the model in the code perspective, which enables you to view and further configure code generation settings.



BlockDiagram

Last saved by: The MathWorks, Inc.
Last saved on: Wed Jan 03 03:24:48 2018

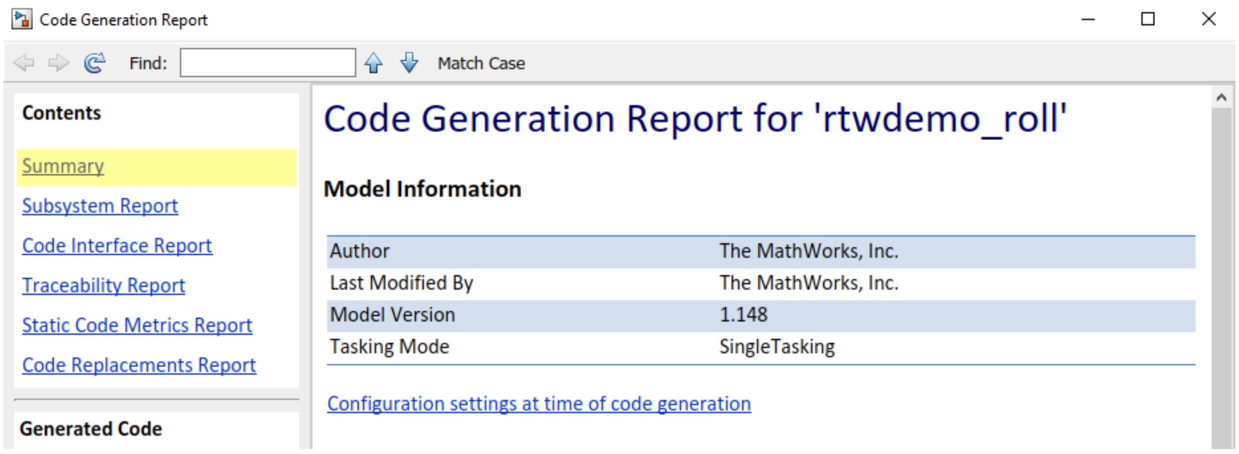
Select a model element to set or view its parameters or properties.

Model Element Category	Storage Class
Inports	Default
Outports	Default
Global parameters	Default
Local parameters	Default
Shared local data stores	Default

Inspect and Analyze Generated Code

In the example “Prepare a Model for Embedded Code Generation” on page 3-6, you used the Embedded Coder Quick Start tool to configure the example model `rtwdemo_roll` for embedded code generation. In this example, you use a code generation report to inspect the generated code.

The Quick Start tool configures a model to produce a report. You can open the report from the tool or, at any time, select **Code** > **C/C++ Code** > **Code Generation Report** > **Open Model Report**. The report opens to the **Summary** section.



Code Generation Report for 'rtwdemo_roll'

Model Information

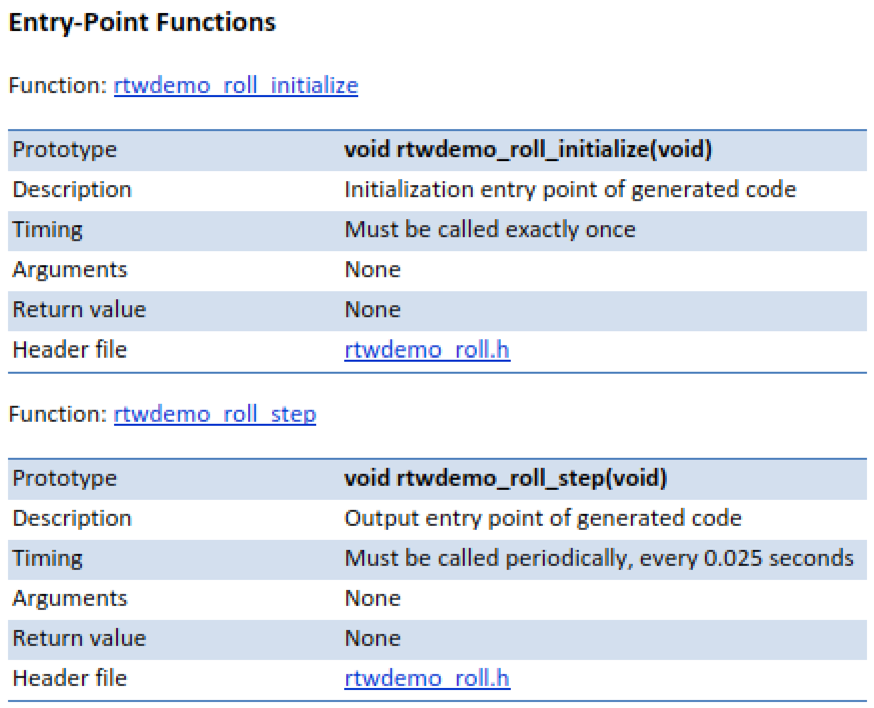
Author	The MathWorks, Inc.
Last Modified By	The MathWorks, Inc.
Model Version	1.148
Tasking Mode	SingleTasking

Configuration settings at time of code generation

Interfaces of Entry-Point Functions

The code that you generate from a model includes entry-point functions, which you call from your application code. 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 can control. To inspect the interfaces and prepare to write your application code, use the code interface report.

1. In the code generation report, under **Contents**, click **Code Interface Report**.
2. In the interface report, inspect the **Entry-Point Functions** section.



Entry-Point Functions

Function: rtwdemo_roll_initialize

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

Function: rtwdemo_roll_step

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

The report shows that the generated code defines two entry-point functions: `rtwdemo_roll_step` (execution function) and `rtwdemo_roll_initialize` (initialization function, which, for this model, does nothing).

The functions are `void-void`. Instead of accepting data through arguments, the functions interact directly with global variables. With this data interface, the functions are not reentrant.

3. Inspect the **Inports** and **Outports** sections. The report shows that the generated code defines two global structure variables, `rtU` and `rtY`, whose fields represent the Inport and Outport blocks at the root level of the model.

   Root-level Inport and Outport blocks represent high-level inputs and outputs of a model. Your application code can write input data to the fields of the `rtU` structure, execute the algorithmic code by calling `rtwdemo_roll_step`, and read the resulting output data from `rtY`.
4. Inspect the **Interface Parameters** section. The report shows that the model code does not define tunable parameters.

By default, the code generator generates efficient code by eliminating storage for block parameters, such as the **Gain** parameter of a Gain block. The parameter values appear in the code as literal numbers, so you cannot tune them.

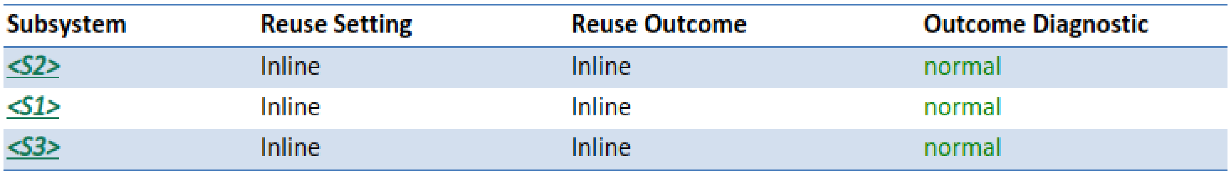
To make individual parameters tunable, such as the parameters of the PID controller modeled by the `BasicRollMode` subsystem, you later configure them to appear in the code as global variables whose values you can change. You can also configure downstream signal lines to appear as variables whose values you can monitor.

The generated header file `rtwdemo_roll.h` declares the elements of the model interface, including the entry-point functions and global variables. To call the functions and interact with the global variables, you can include (`#include`) this header file in your application code.

Subsystem Partitioning

The model contains atomic subsystems such as `BasicRollMode`, which organize algorithmic blocks and improve the appearance of the model. By default, organizing blocks into subsystems does not affect the generated code.

In the code generation report, under **Contents**, select **Subsystem Report**. The subsystem report indicates that the three subsystems appear inline in the generated code.



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

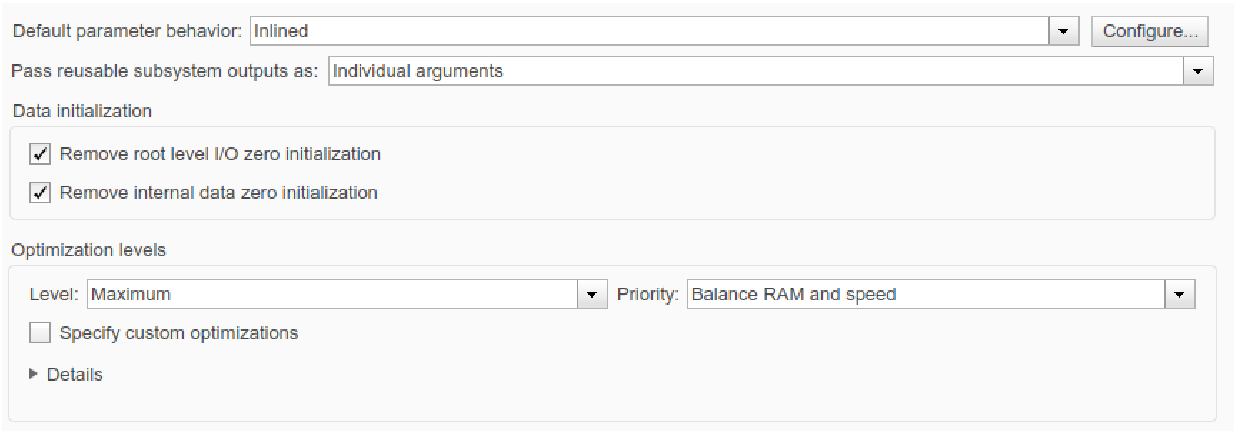
Later, you can modularize the code by configuring each subsystem to appear as a separate function.

Execution Efficiency and Consumption of Computing Resources

By default, optimizations make the generated code more efficient. You can adjust the high-level optimization strategy in the model configuration parameters.

1 In `rtwdemo_roll`, inspect the **Configuration Parameters > Code Generation > Optimization** pane.

2. Clear **Specify custom optimizations**.
3. In the Warning dialog box, click **OK**.
4. In the Configuration Parameters dialog box, click **Apply**.


Default parameter behavior: Inlined
Pass reusable subsystem outputs as: Individual arguments

Data initialization

- [✓] Remove root level I/O zero initialization
- [✓] Remove internal data zero initialization

Optimization levels

Level: Maximum Priority: Balance RAM and speed

- [ ] Specify custom optimizations

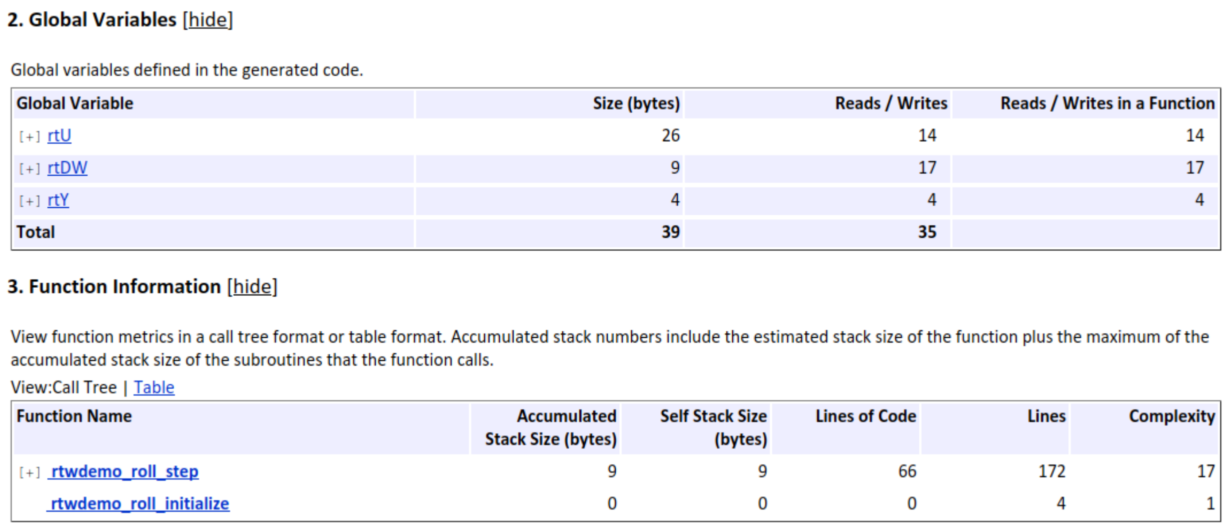
- To favor one computing resource over another, under **Optimization levels**, you can adjust the **Priority** parameter. For example, you can favor increased execution speed over reduced memory consumption.

  For `rtwdemo_roll`, the value `Balance RAM and speed` means that the code generator does not favor one resource over another.
- To allocate more memory for data so that you can tune and monitor it, you can reduce the strength of the optimizations by adjusting the **Level** parameter.

  For `rtwdemo_roll`, the value `Maximum` means that the code generator optimizes the code by eliminating storage for unnecessary data.

To estimate the amount of some computing resources, such as RAM and ROM, that the generated code consumes on your hardware, use the code generation report. In the report, under **Contents**, select **Static Code Metrics Report**. For `rtwdemo_roll`:

- The **Global Variables** section estimates that global variables that are defined by the generated code consume 39 bytes of memory.
- The **Function Information** section estimates that the model execution function, `rtwdemo_roll_step`, consumes 9 bytes of stack memory.



2. Global Variables [hide]

Global variables defined in the generated code.

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

3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View:Call Tree | Table

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
[+] rtwdemo_roll_step	9	9	66	172	17
rtwdemo_roll_initialize	0	0	0	4	1

Example Main Program

The generated code includes an example `main` program in the file `ert_main.c`. To use the generated algorithmic code (the model entry-point functions) in your application, you can copy the incomplete functions defined in `ert_main.c`, then complete the functions by inserting your custom scheduling code.

In the code generation report, under **Generated Code**, inspect `ert_main.c`. In the file, inspect the incomplete wrapper function `rt_OneStep`, which your application code can call to run the model algorithm each execution cycle. This function calls the model execution function, `rtwdemo_roll_step`.

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = false;

  /* Disable interrupts here */

  /* Check for overrun */
  if (OverrunFlag) {
    return;
  }
  OverrunFlag = true;

  /* Save FPU context here (if necessary) */
  /* Re-enable timer or interrupt here */
  /* Set model inputs here */

  /* Step the model */
  rtwdemo_roll_step();

  /* Get model outputs here */

  /* Indicate task complete */
  OverrunFlag = false;

  /* Disable interrupts here */
  /* Restore FPU context here (if necessary) */
  /* Enable interrupts here */
}
```

The file also defines an incomplete example `main` function, which outlines the order and context in which your application code can call `rt_OneStep` and the other model entry-point functions.

```
int_T main(int_T argc, const char *argv[])
{
  /* Unused arguments */
  (void)(argc);
  (void)(argv);

  /* Initialize model */
  rtwdemo_roll_initialize();

  /* Attach rt_OneStep to a timer or interrupt service routine with
   * period 0.025 seconds (the model's base sample time) here.  The
   * call syntax for rt_OneStep is
   *
   *  rt_OneStep();
   */
  printf("Warning: The simulation will run forever. "
         "Generated ERT main won't simulate model step behavior. "
         "To change this behavior select the 'MAT-file logging' option.\n");
  fflush((NULL));
  while (1) {
    /*  Perform other application tasks here */
  }

  /* The option 'Remove error status field in real-time model data structure'
   * is selected, therefore the following code does not need to execute.
   */
#if 0

  /* Disable rt_OneStep() here */
#endif
```

```
  return 0;
}
```

Code Placement in Files and Folders

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. The folder also contains supporting files such as `rtwtypes.h`, which defines standard data types that the generated code uses by default.

In your current folder, in addition to `rtwdemo_roll_ert_rtw`, the code generator creates the `slprj` folder. In general, this sibling folder contains generated files that can or must be shared between multiple models. In this case, because you generated code from one model, `slprj` does not contain meaningful code files.

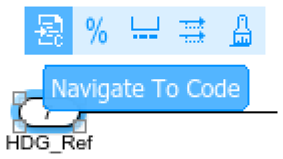
Traceability Between Model and Generated Code

You can locate the representation of a model element in the generated code. You can also trace from the code to an element in a model. This traceability helps you write your application code, debug the generated code, and verify that the system satisfies requirements.

At the root level of `rtwdemo_roll`, the Inport block `HDG_Ref` represents a system input. To find this input data in the generated code:

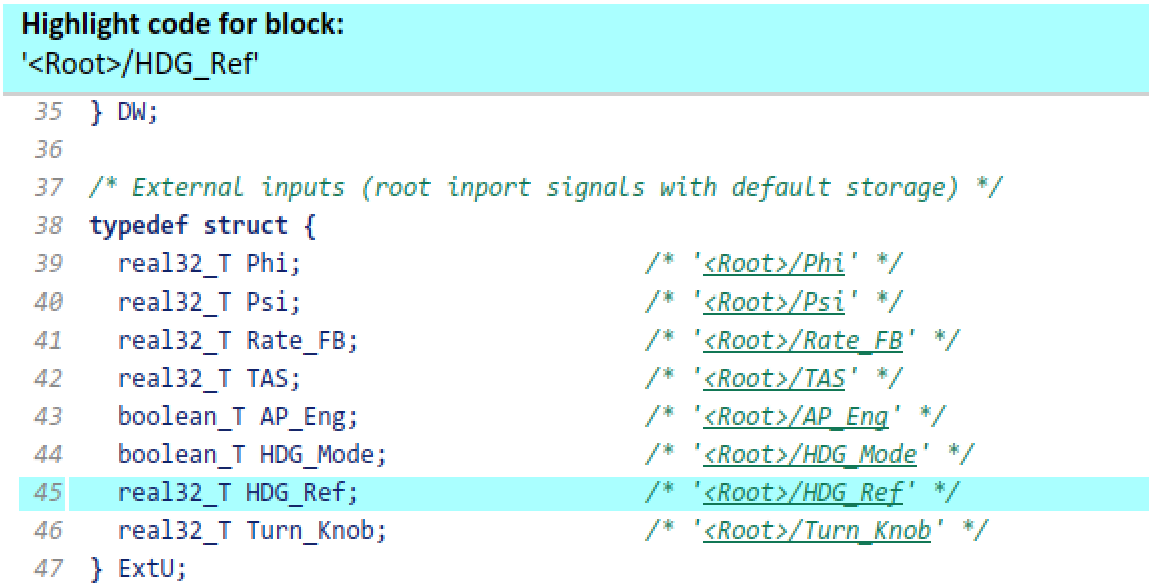
1 In the model, click the `HDG_Ref` block.
2 Move the cursor over the ellipsis that appears above the block ⋯ .

  Several icons appear. To display the name of each icon, move your cursor over the icon.
3 Select the **Navigate to Code** icon.



The code generation report highlights the code corresponding to the Inport block.

4 Inspect the header file `rtwdemo_roll.h`. The highlighting identifies the code that defines the structure field `HDG_Ref`, which represents the block.



Highlight code for block:
'<Root>/HDG_Ref'

```
} DW;

/* External inputs (root inport signals with default storage) */
typedef struct {
  real32_T Phi;                        /* '<Root>/Phi' */
  real32_T Psi;                        /* '<Root>/Psi' */
  real32_T Rate_FB;                    /* '<Root>/Rate_FB' */
  real32_T TAS;                        /* '<Root>/TAS' */
  boolean_T AP_Eng;                    /* '<Root>/AP_Eng' */
  boolean_T HDG_Mode;                  /* '<Root>/HDG_Mode' */
  real32_T HDG_Ref;                    /* '<Root>/HDG_Ref' */
  real32_T Turn_Knob;                  /* '<Root>/Turn_Knob' */
} ExtU;
```

5 Inspect `rtwdemo_roll.c`. The highlighting identifies the algorithmic code in `rtwdemo_roll_step` that reads data from the structure field.

To navigate from the code to the blocks and other elements in the model, click hyperlinks in the code generation report. In `rtwdemo_roll.h`, in the definition of the structure type `ExtY`, the field `Ail_Cmd` represents an Outport block at the root level of the model.

```
/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real32_T Ail_Cmd;                    /* '<Root>/Ail_Cmd' */
} ExtY;
```

The comment `<Root>/Ail_Cmd` is a hyperlink.

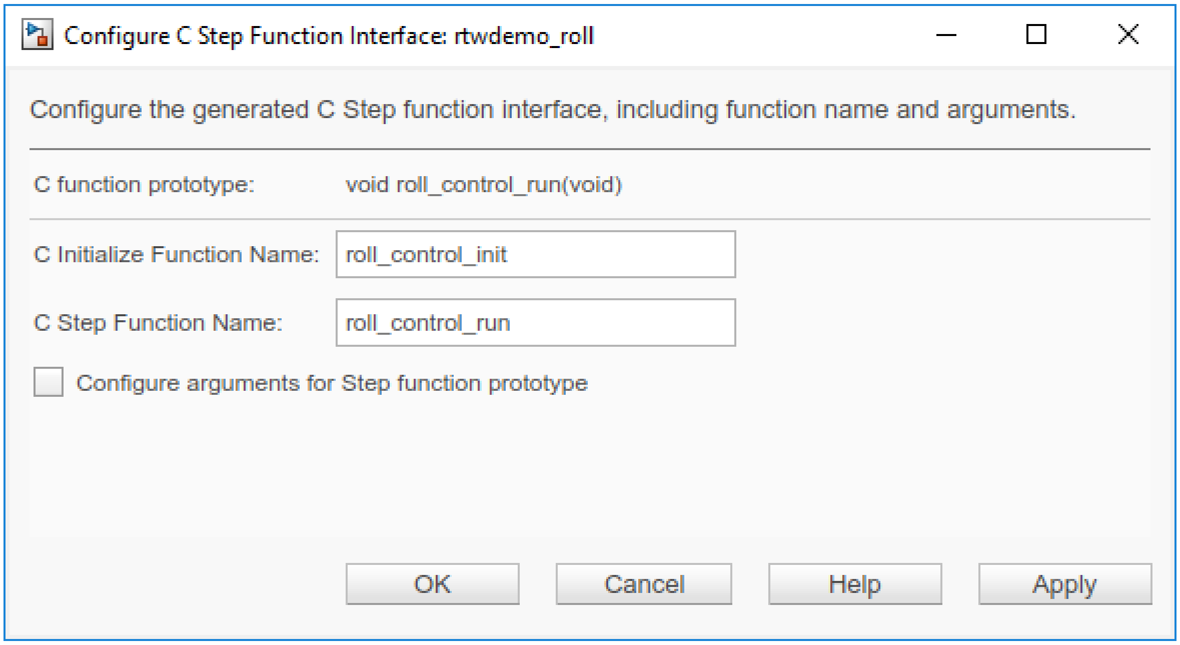
Customize Interfaces of Generated Entry-Point Functions

In the example “Inspect and Analyze Generated Code” on page 3-10, you inspected the code that the code generator produced from the example model `rtwdemo_roll`. In this example, you customize the interfaces of the model execution function, `rtwdemo_roll_step`, and the model initialization function, `rtwdemo_roll_initialize`. The interface of a function includes the function name and the data, such as system-level inputs and outputs, that the function exchanges with your application code.

Function Names

Set the names of the model entry-point functions to `roll_control_run` (execution) and `roll_control_init` (initialization).

1. In the model, navigate to the **Configuration Parameters** > **Code Generation** > **Interface** pane.
2. Click the **Configure Model Functions** button to open the **Configure C/C++ Function Interface** dialog box.
3. To configure the function names, in the dialog box, set **C/C++ Initialize Function Name** to `roll_control_init` and **C/C++ Step Function Name** to `roll_control_run`. Then, click **OK**.



Configure C Step Function Interface: rtwdemo_roll

Configure the generated C Step function interface, including function name and arguments.

C function prototype: void roll_control_run(void)

C Initialize Function Name: roll_control_init

C Step Function Name: roll_control_run

☐ Configure arguments for Step function prototype

Input and Output Data

In this part of the example, you customize the data interface of the model by configuring the execution function (`roll_control_run`) to:

- Read input data from existing global variables that are defined by external code. Most of the variables are declared in an external header file, `roll_input_data.h`, and defined in `roll_input_data.c`. As an exception, the input `HDG_Mode` is declared in a different header file, `roll_heading_mode.h`, and defined in `roll_heading_mode.c`.
- Write output data to global variables that the generated code defines in `output_data.c` and declares in `output_data.h`.

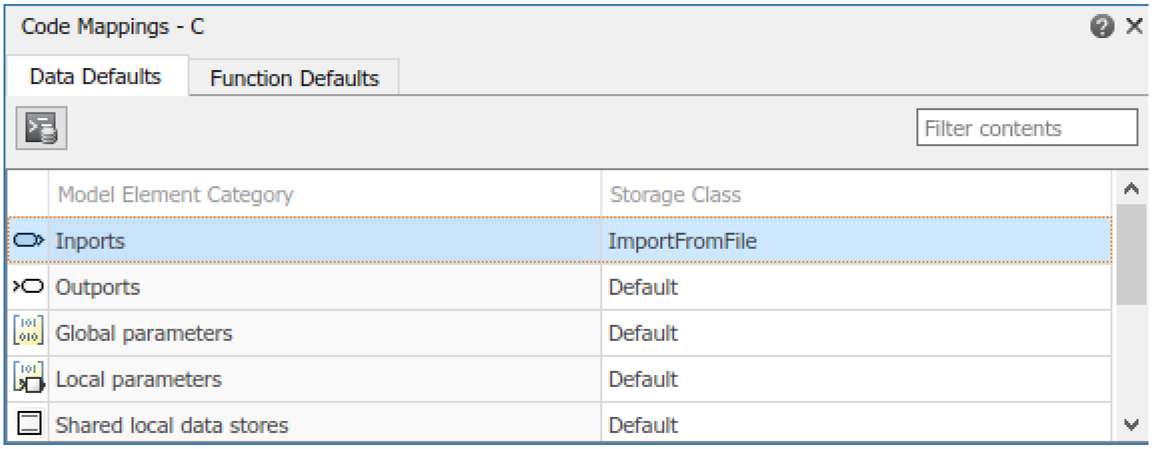
First, at the command prompt, copy the external code files into your MATLAB current folder.

```
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'));
```

These files represent existing, external code with which you compile the generated code.

Now, configure the data interface of the model.

1 If the model is not already in the code perspective, select **Code > C/C++ Code > Configure Model in Code Perspective**.
2 Configure Inport blocks at the root level of the model to appear in the generated code as separate global variables defined by external code. Under **Code Mappings > Data Defaults**, for the **Inports** row, in the **Storage Class** column, select `ImportFromFile` from the drop-down list.

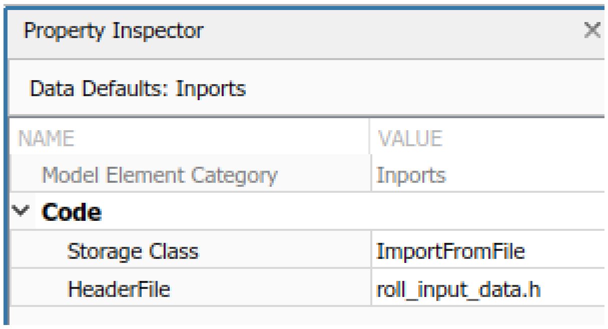

Code Mappings - C

Data Defaults | Function Defaults

Filter contents

Model Element Category	Storage Class
Inports	ImportFromFile
Outports	Default
Global parameters	Default
Local parameters	Default
Shared local data stores	Default

With this setting, the generated code does not define the variables, instead including (`#include`) a declaration header file whose name you specify with the **HeaderFile** property.
3 In the Property Inspector, for **HeaderFile**, enter `roll_input_data.h`.


Property Inspector

Data Defaults: Inports

NAME	VALUE
Model Element Category	Inports
Code	
Storage Class	ImportFromFile
HeaderFile	roll_input_data.h

4 Configure root-level Outport blocks to appear in the generated code as separate global variables. For the **Outports** row, in the **Storage Class** column, select `ExportToFile`.

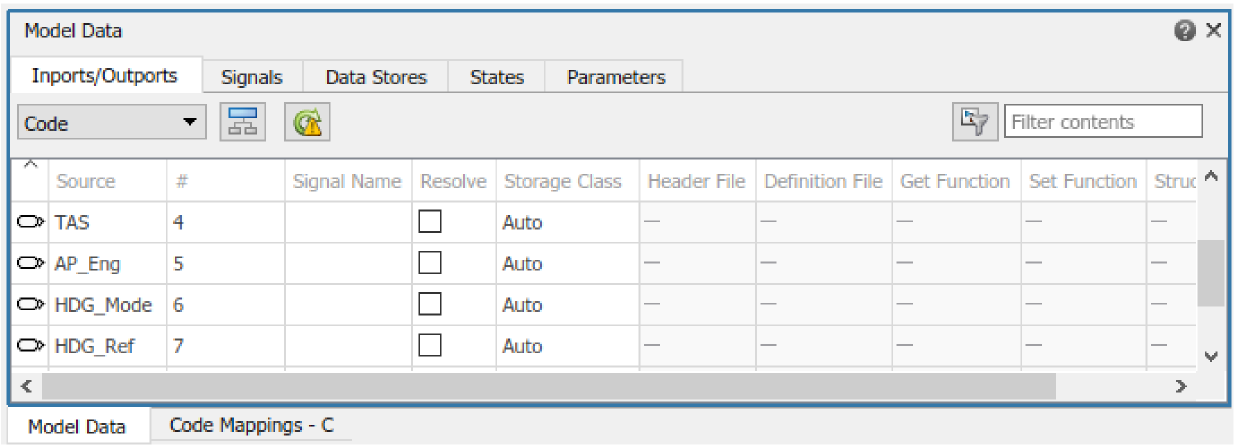
   The generated code defines and declares the variables in files whose names you specify with the **DefinitionFile** and **HeaderFile** properties.
5 In the Property Inspector, set **HeaderFile** to `output_data.h` and **DefinitionFile** to `output_data.c`.
6 Configure each variable that appears in the code to use the name of the corresponding Inport or Outport block in the model. Change the setting of **Configuration Parameters** > **Code Generation** > **Symbols** > **Global variables** from `rt$N$M` to `$N$M` and click **Apply**.

   The default value, `rt$N$M`, means that in the generated code, the global variables have names that begin with `rt`. In `roll_input_data.h` and `roll_heading_mode.h`, the variable names do not begin with `rt`, which is why you removed the prefix.

   The token `$N` stands for the name of each data element in the model, for example, the name of each Inport or Outport block. The token `$M` represents name-mangling text that the code generator inserts, if necessary, to avoid name collisions with other global symbols in the code.

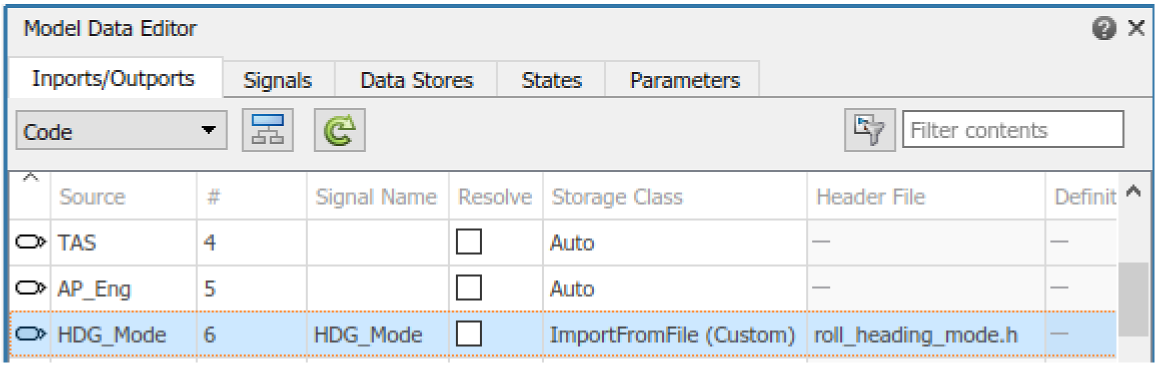
The settings that you choose in **Data Defaults** and **Function Defaults** apply to each category of model elements by default. You can override these defaults for an individual element by using other tools such as the Model Data Editor. To configure the generated code to import `HDG_Mode` from `roll_heading_mode.h` instead of `roll_input_data.h`:

1 Open the Model Data Editor. Underneath the block diagram, select **Model Data Editor** > **Inports/Outports**.



Source	#	Signal Name	Resolve	Storage Class	Header File	Definition File	Get Function	Set Function	Struc
TAS	4		☐	Auto	—	—	—	—	—
AP_Eng	5		☐	Auto	—	—	—	—	—
HDG_Mode	6		☐	Auto	—	—	—	—	—
HDG_Ref	7		☐	Auto	—	—	—	—	—

2 For the `HDG_Mode` row, set **Storage Class** to `ImportFromFile`.
3 Set **Header File** to `roll_heading_mode.h`.



Source	#	Signal Name	Resolve	Storage Class	Header File	Definit
TAS	4		☐	Auto	—	—
AP_Eng	5		☐	Auto	—	—
HDG_Mode	6	HDG_Mode	☐	ImportFromFile (Custom)	roll_heading_mode.h	—

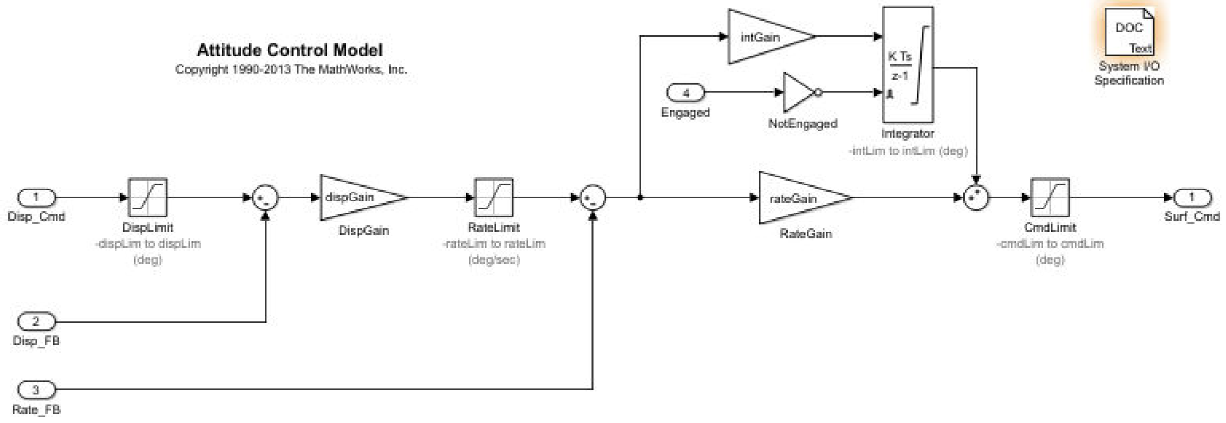
`HDG_Mode` uses the code generation settings that you specified in the Model Data Editor. The other Inport blocks use the default setting for the **Storage Class** column, `Auto`, which means they acquire the code generation settings that you specified for **Code Mappings** > **Data Defaults** > **Inports**.

Tunable Parameters

You can configure the PID control parameters in the `BasicRollMode` subsystem, and some associated signals, to appear in the code as global variables whose values you can tune and monitor.

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

1 In the model, double-click the `BasicRollMode` Subsystem block.



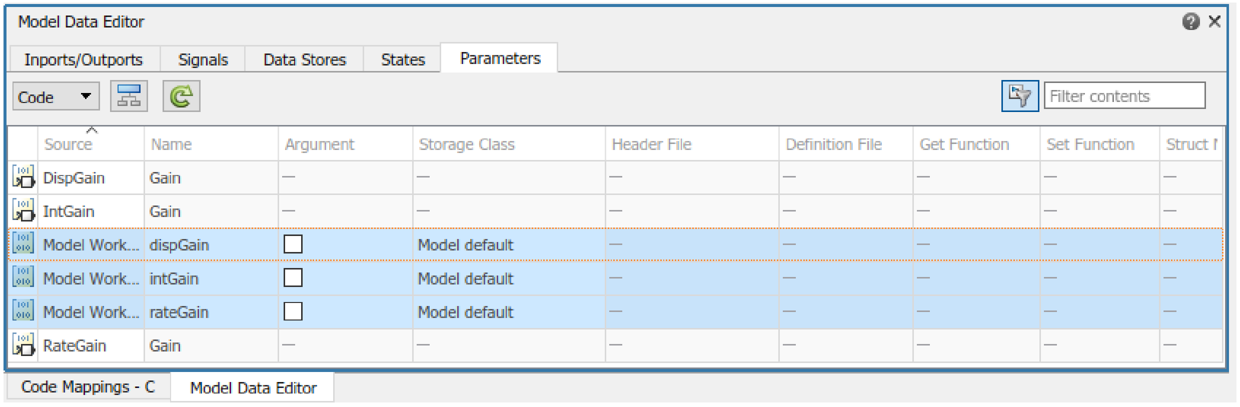
2 Underneath the block diagram, select **Model Data Editor** > **Parameters**.
3 Configure the Model Data Editor to filter the data table as you select blocks and signals in the model. To do so, next to the **Filter contents** box, activate the **Filter using selection** button.
4 Configure the Model Data Editor to show information about numeric MATLAB variables that the subsystem uses. To do so, click the **Show/refresh additional information** button.

The Gain blocks use some variables, such as `intGain`, that are stored in the model workspace. Each variable appears as a row in the data table under **Parameters**.
5 In the model, select the three Gain blocks. For example, hold **Shift**, and then select each block.

The Model Data Editor shows three rows that correspond to the **Gain** parameters and three rows that correspond to the workspace variables.
6 In the Model Data Editor, select the three rows that correspond to the variables.
7 Convert the variables to `Simulink.Parameter` objects. To do so, in the **Storage Class** column, select `Convert to parameter object`.

You cannot apply a storage class to a numeric variable, but you can apply a storage class to a parameter object.

8 Confirm that the parameter objects prevent optimizations from eliminating storage for them. In the **Storage Class** column, verify that the objects use the storage class setting `Model default`.



Model Data Editor

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

Code

	Source	Name	Argument	Storage Class	Header File	Definition File	Get Function	Set Function	Struct N
	DispGain	Gain	—	—	—	—	—	—	—
	IntGain	Gain	—	—	—	—	—	—	—
	Model Work...	dispGain	☐	Model default	—	—	—	—	—
	Model Work...	intGain	☐	Model default	—	—	—	—	—
	Model Work...	rateGain	☐	Model default	—	—	—	—	—
	RateGain	Gain	—	—	—	—	—	—	—

Code Mappings - C | Model Data Editor

With this setting, the objects use the storage class that you specify for **Code Mappings > Data Defaults > Local parameters**.

9 In the Model Data Editor, inspect the **Signals** tab.

10 In the data table, select the three available rows, which correspond to the output signals of the Gain blocks.

11 Prevent the code generator from eliminating storage for the signals. To do so, in the **Storage Class** column, select `Model default`.

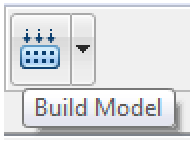
The signals acquire the storage class that you specify for **Code Mappings > Data Defaults > Internal data**.

The code that you generate from the model stores the parameter objects and signals in memory. Because you left the storage class settings for **Code Mappings > Data Defaults > Local parameters** and **Internal data** at the default value, `Default`, the code generator determines the storage format, for example, as fields of structures.

Generate and Inspect Code

1 Include the external source files `roll_input_data.c` and `roll_heading_mode.c` in the build process. In the model, set **Configuration Parameters > Code**

Generation > Custom Code > Additional build information > Source files to `roll_input_data.c roll_heading_mode.c`. Then, click **Apply**.

2. To refresh the generated code and the code generation report, generate code from the model again. To generate code, in the model, click the  button.
3. In the code generation report, under **Contents**, select **Code Interface Report**.
4. Inspect the **Entry-Point Functions** section. The functions use the names that you specified.
5. Inspect the **Inports** section. The root-level Inport blocks appear as imported data, which means the generated code does not define the corresponding global variables.
6. Inspect the **Outports** section. The root-level Outport block appears as a separate global variable. The variable is defined in `output_data.c` and declared in `output_data.h`, which you can inspect under **Generated Code > Data files** and **Generated Code > Shared files**.
7. Inspect the **Interface Parameters** section. The three parameter objects appear as fields of a structure named `P`. Your application code can tune the values of these fields.

The interface report does not identify the C-code representations of the three signals that you configured. Instead, you can trace from the signals in the model to the corresponding data in the code.

1. In the model, select the DispGain block.
2. Move the cursor over the ellipsis that appears above the block.
3. Select the option on the left, **Navigate To Code**.

   The code generation report highlights some code in `rtwdemo_roll.c`.
4. In the report, to the right of **Highlight code for block**, click the right arrow button until you find the definition of the structure type named `DW` in `rtwdemo_roll.h`.

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real32_T DispGain;                  /* '<S1>/DispGain' */
  real32_T RateGain;                  /* '<S1>/RateGain' */
  real32_T IntGain;                   /* '<S1>/IntGain' */
  real32_T FixPtUnitDelay1_DSTATE;    /* '<S7>/FixPt Unit Delay1' */
  real32_T Integrator_DSTATE;         /* '<S1>/Integrator' */
  int8_T Integrator_PrevResetState;   /* '<S1>/Integrator' */
} DW;
```

Among other data, this structure type contains three fields that correspond to the output signals of the Gain blocks. The file also declares a global structure variable named `DW_l` from which your application code can read the signal data.

By controlling the interfaces of the generated entry-point functions, you can generate code that interacts with your existing, external algorithmic and scheduling code.

Partition and Modularize Generated Code

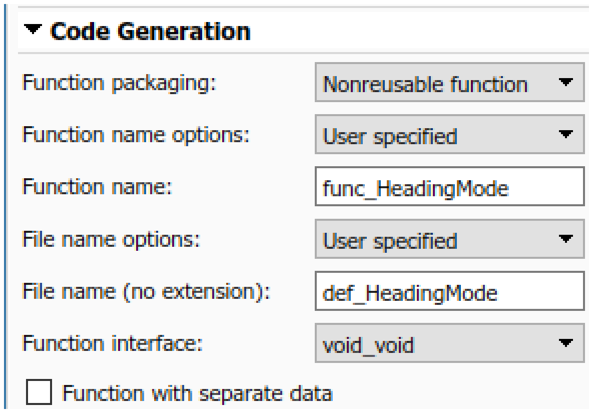
In the example "Inspect and Analyze Generated Code" on page 3-10, you inspected the code generated from the example model `rtwdemo_roll`. By default, the algorithmic code appeared in the entry-point function `rtwdemo_roll_step` in the file `rtwdemo_roll.c`. In this example, you partition the code by generating more files and a separate function for each subsystem in the model. Organizing the code into separate functions and files can make it easier to read and maintain.

Configure Model and Subsystems

1. In the model, inspect the value of **Configuration Parameters > Code Generation > Code Placement > File packaging format**.

   The configuration parameter is set to `Compact (with separate data file)`, which means the code generator produces fewer files by aggregating data, function, and type definitions.
2. Configure the code generator to distribute the code among more files by setting **File packaging format** to `Modular`.
3. At the root level of the model, select the subsystem `HeadingMode`.
4. In the Property Inspector, under **Code Generation**:

   - Configure the subsystem to appear in the code as a separate function. To do so, set **Function packaging** to `Nonreusable function`.
   - Set **Function name options** to `User specified`. Then, set **Function name** to `func_HeadingMode`.
   - Place the function definition and declaration in separate generated files. To do so, set **File name options** to `User specified`. Then, set **File name** to `def_HeadingMode`. This configuration sets the names of the generated files to `def_HeadingMode.c` and `def_HeadingMode.h`.



▼ Code Generation

Function packaging:	Nonreusable function
Function name options:	User specified
Function name:	func_HeadingMode
File name options:	User specified
File name (no extension):	def_HeadingMode
Function interface:	void_void

☐ Function with separate data

5 Configure the subsystems `RollAngleReference` and `BasicRollMode` in a similar way. When specifying function and file names, make sure you use the name of each subsystem.

Inspect Generated Code

1 Generate code from the model.

2 Inspect the more modular placement of the model code. In the code generation report, under **Generated Code** > **Model files**, inspect the new files `rtwdemo_roll_private.h` and `rtwdemo_roll_types.h`.

   These files can separate code constructs from `rtwdemo_roll.h`, such as the declarations of some global variables and the definitions of some types.

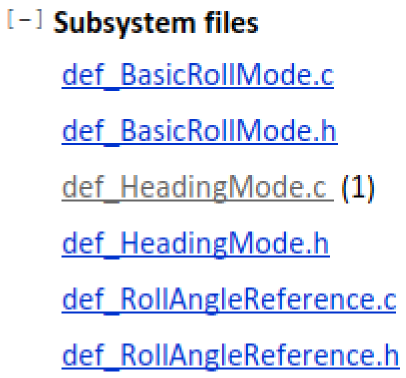
3 Find the representation of the `HeadingMode` subsystem in the generated code. To do so, at the root level of the model, click the Subsystem block and, using the ellipsis, select **Navigate To Code**.

   The code generation report shows the generated file `def_HeadingMode.c`, which defines the `func_HeadingMode` function.

4 To the right of **Highlight code for block**, use the arrow buttons to navigate to the code highlighted in `rtwdemo_roll.c`.

   The model execution function `roll_control_run` calls `func_HeadingMode`.

5 Under **Generated Code** > **Subsystem files**, inspect the files generated for the other subsystems. Each subsystem yields a function defined in a `.c` file and declared in a `.h` file.



[-] **Subsystem files**

def_BasicRollMode.c

def_BasicRollMode.h

def_HeadingMode.c (1)

def_HeadingMode.h

def_RollAngleReference.c

def_RollAngleReference.h

As design iterations result in changes to the generated code, you can use external versioning tools to manage the generated code files and track the changes.

Deploy and Test Executable Program

In this section...

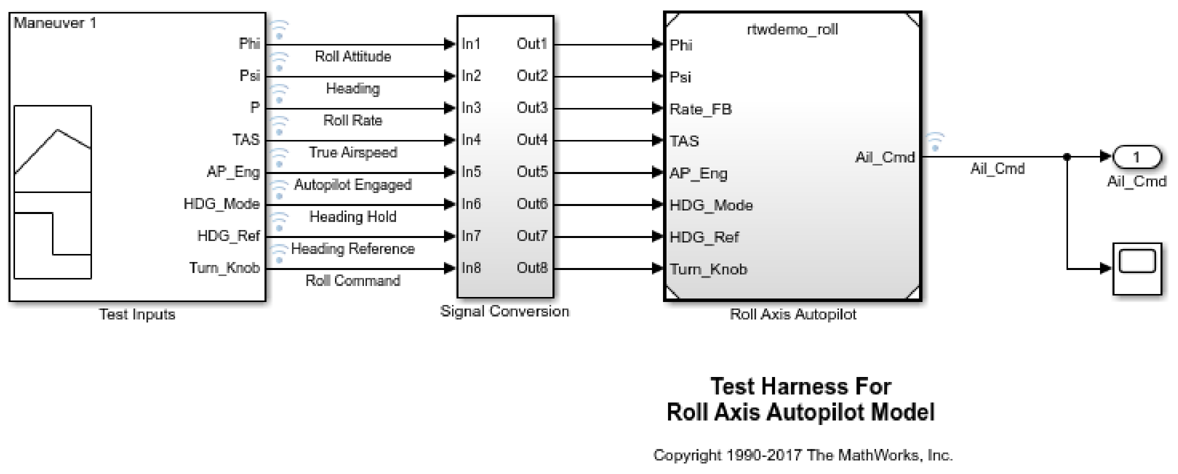
In the example "Inspect and Analyze Generated Code" on page 3-10, you inspected the code generated from the example model `rtwdemo_roll`. In this example, you verify that when executed, the code is numerically equivalent to the algorithm modeled in Simulink. You use a test harness model to simulate `rtwdemo_roll` in normal mode and again 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 towards achieving IEC 61508, IEC 62304, ISO 26262, EN 50128, or DO-178 certification.

Inspect and Configure Test Harness Model

The example 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 the example models `rtwdemo_roll_harness` and `rtwdemo_roll`.
2 In the MATLAB current folder, save a copy of `rtwdemo_roll`.
3 In the `rtwdemo_roll_harness` model, right-click the Model block and select **Subsystem & Model Reference > Refresh Selected Model Block**.
4 Save a copy of `rtwdemo_roll_harness` in the current folder.
5 Open the Configuration Parameters dialog boxes for `rtwdemo_roll_harness` and `rtwdemo_roll`.
6 For both models, on the **Code Generation** pane, verify that the **Generate code only** check box is cleared.

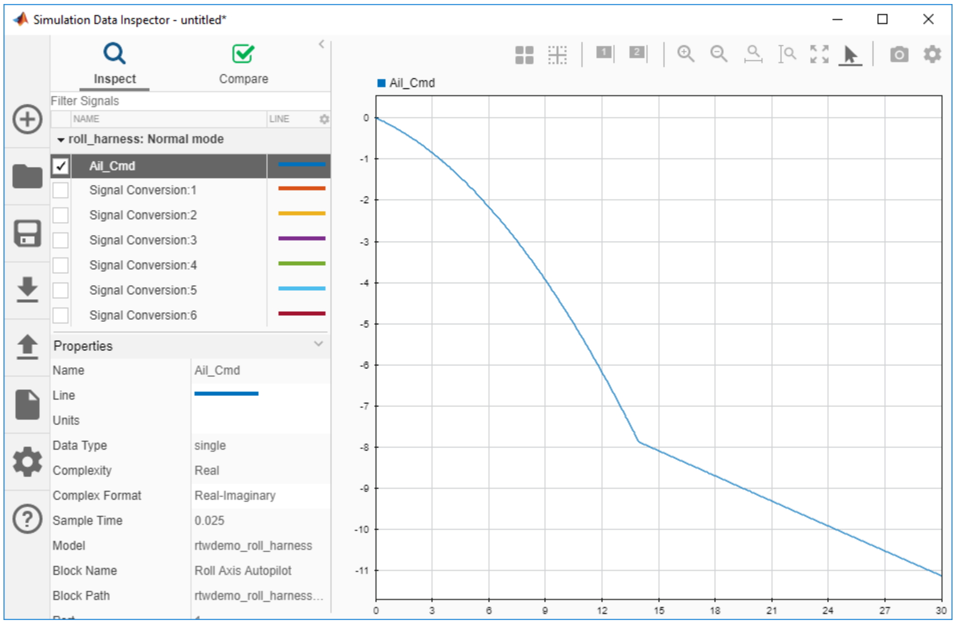
   To run SIL and PIL simulations, you must clear **Generate code only**.
7 For both models, on the **Hardware Implementation** pane, expand **Device details**. Verify that **Support long long** is selected.

   For this example, given the configurations of the models, to run SIL and PIL simulations, you must select **Support long long**.
8 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** drop-down list to `Instrumentation`.
4. In the data table, click a row in the **Source** column. Then, select all of the rows by pressing **Ctrl+A**.
5. To configure all of the 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 new run, double-click the run name field and rename the run: `roll_harness: Normal mode`.
11. Select `Ail_Cmd` to plot the signal.



Name	Ail_Cmd
Line	
Units	
Data Type	single
Complexity	Real
Complex Format	Real-Imaginary
Sample Time	0.025
Model	rtwdemo_roll_harness
Block Name	Roll Axis Autopilot
Block Path	rtwdemo_roll_harness…

Simulate the Model in SIL Mode

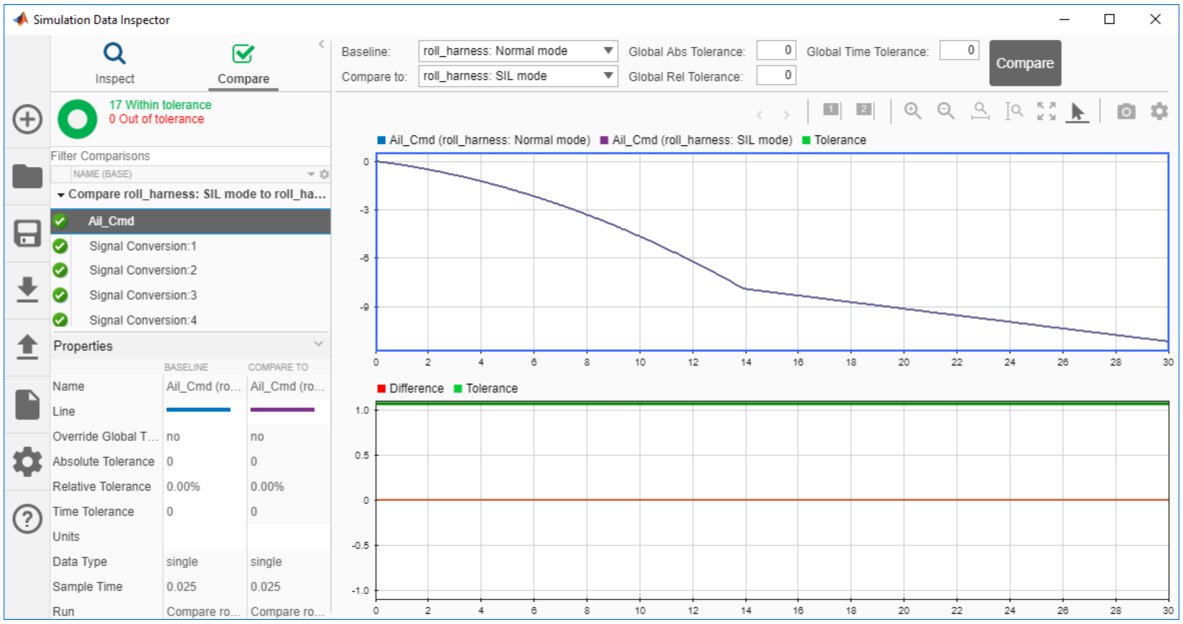
The SIL simulation generates, compiles, and executes code on your development computer. The Simulation Data Inspector logs results.

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, specify **Simulation mode** as `Software-in-the-loop (SIL)`. Click **OK**.
3. Exclude the external code files from the build process. In `rtwdemo_roll`, set **Configuration Parameters > Code Generation > Custom Code > Additional build information > Source files** to the default value, which is empty. Then, save the model.
4. Simulate the `rtwdemo_roll_harness` model.
5. In the Simulation Data Inspector, double-click the run name field and rename the new run as `roll_harness: SIL mode`.
6. Select `Ail_Cmd` to plot the signal.

Compare Simulation Results

In the Simulation Data Inspector:

1. Click the **Compare** tab.
2. In the **Baseline** field, select `roll_harness: Normal mode`.
3. In the **Compare To** field, select `roll_harness: SIL mode`.
4. Click **Compare**.


Simulation Data Inspector
Inspect
Compare
Compare
Filter Comparisons
NAME (BASE)
Properties
Ail_Cmd (roll_harness: Normal mode)
Ail_Cmd (roll_harness: SIL mode)
Tolerance
Difference
Tolerance

Baseline: roll_harness: Normal mode
Compare to: roll_harness: SIL mode
Global Abs Tolerance: 0
Global Rel Tolerance: 0
Global Time Tolerance: 0

17 Within tolerance
0 Out of tolerance

Compare roll_harness: SIL mode to roll_ha...
- ✓ Ail_Cmd
- ✓ Signal Conversion:1
- ✓ Signal Conversion:2
- ✓ Signal Conversion:3
- ✓ Signal Conversion:4

	BASELINE	COMPARE TO
Name	Ail_Cmd (ro...	Ail_Cmd (ro...
Line		
Override Global T...	no	no
Absolute Tolerance	0	0
Relative Tolerance	0.00%	0.00%
Time Tolerance	0	0
Units		
Data Type	single	single
Sample Time	0.025	0.025
Run	Compare ro...	Compare ro...

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.

More Information About Code Generation in Model-Based Design

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

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	“Model Architecture and Design”
Generate code variants by using macros for preprocessor compilation from models	“Variant Systems”
Achieve code reuse	“Subsystems” and “Referenced Models”
Define and control model data	“Data Representation and Access”
Control generation of function and class interfaces	“Function and Class Interfaces”
Control naming and partitioning of data and code across generated source files	“File Packaging”
Select and configure the target environment for your application	“Run-Time Environment Configuration”
Configure model for code generation objectives, such as efficiency or safety	“Configure Model for Code Generation Objectives by Using Code Generation Advisor”
Configure parameters for specifying identifier names, code comments, style, and templates	“Code Appearance”
Export component XML description and C code for AUTOSAR run-time environment	“AUTOSAR Code Generation”
Deploy standalone programs to target hardware	“Deploy Generated Standalone Executable Programs To Target Hardware”
Choose and apply integration paths and methods	“External Code Integration”
Improve code performance, such as memory usage and execution speed	“Performance”
For element-wise arithemtic operations involving floating-point data types, generate more efficient code containing SIMD intrinsics	“Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®”

Task	Reference
Collect code coverage metrics for generated code during SIL or PIL simulation	“Code Coverage”
Test model through SIL and PIL simulations	“Software-in-the-Loop Simulation” and “Processor-in-the-Loop Simulation”
View and analyze execution profiles of code sections	“Code Execution Profiling”