Embedded Coder®

Getting Started Guide

MATLAB® & SIMULINK®

R2023a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Embedded Coder® Getting Started Guide

Trademarks

Patents

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.3 (Release 2012b)
March 2013	Online only	Revised for Version 6.4 (Release 2013a)
September 2013	Online only	Revised for Version 6.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.6 (Release 2014a)
October 2014	Online only	Revised for Version 6.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.8 (Release 2015a)
September 2015	Online only	Revised for Version 6.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.10 (R2016a)
September 2016	Online only	Revised for Version 6.11 (Release 2016b)
March 2017	Online only	Revised for Version 6.12 (Release 2017a)
September 2017	Online only	Revised for Version 6.13 (Release 2017b)
March 2018	Online only	Revised for Version 7.0 (Release 2018a)
September 2018	Online only	Revised for Version 7.1 (Release 2018b)
March 2019	Online only	Revised for Version 7.2 (Release 2019a)
September 2019	Online only	Revised for Version 7.3 (Release 2019b)
March 2020	Online only	Revised for Version 7.4 (Release 2020a)
September 2020	Online only	Revised for Version 7.5 (Release 2020b)
March 2021	Online only	Revised for Version 7.6 (Release 2021a)
September 2021	Online only	Revised for Version 7.7 (Release 2021b)
March 2022	Online only	Revised for Version 7.8 (Release 2022a)
September 2022	Online only	Revised for Version 7.9 (Release 2022b)
March 2023	Online only	Revised for Version 7.10 (Release 2023a)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

1 Product Overview

2 MATLAB Tutorials

3 Simulink Code Generation 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 by Using Embedded Coder

Code Generation Technology

MathWorks® code generation technology produces C or C++ code and executable programs for algorithms. You can write algorithms programmatically by using MATLAB or graphically in the Simulink environment. You can generate code for MATLAB functions and Simulink blocks that are useful for real-time and embedded applications. Generated source code and executable programs 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 bitwise 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 1-6.

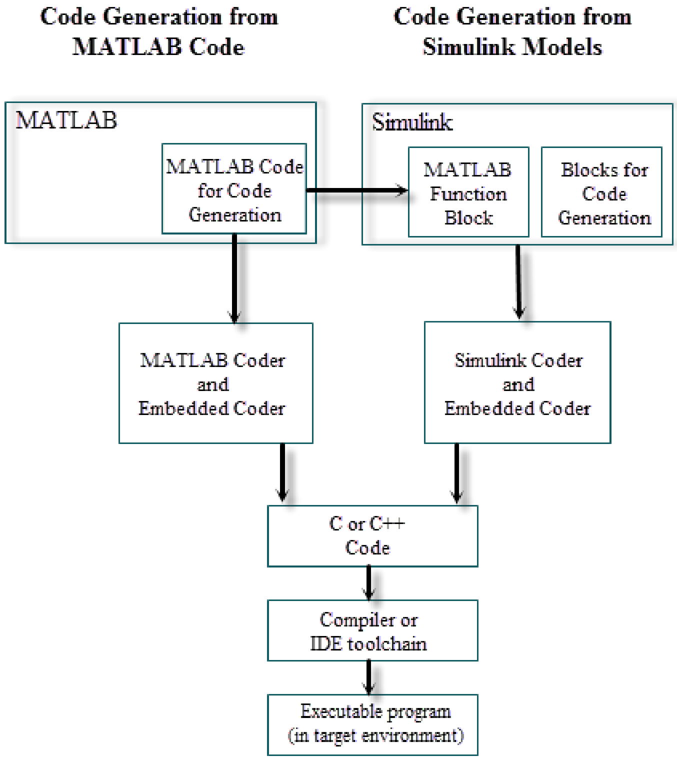
To learn about model design patterns that include Simulink blocks, Stateflow® charts, and MATLAB functions, and map to commonly used C constructs, see “Modeling Patterns for C Code Constructs”.

Code Generation Workflows with Embedded Coder

The Embedded Coder product *extends* the MATLAB Coder and Simulink Coder products with 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 handwritten code. For example, you can:

- Generate code that is compact and fast, which is essential for real-time simulators, rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize generated code for a specific target environment.
- Integrate existing applications, functions, and data.
- Enable tracing, reporting, and testing options that facilitate code verification.

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


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)

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 1-4.

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 1-5.

Code Generation from MATLAB Code

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

- MATLAB
- MATLAB Coder
- Embedded Coder

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

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

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

Code Generation from Simulink Models

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

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

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

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

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

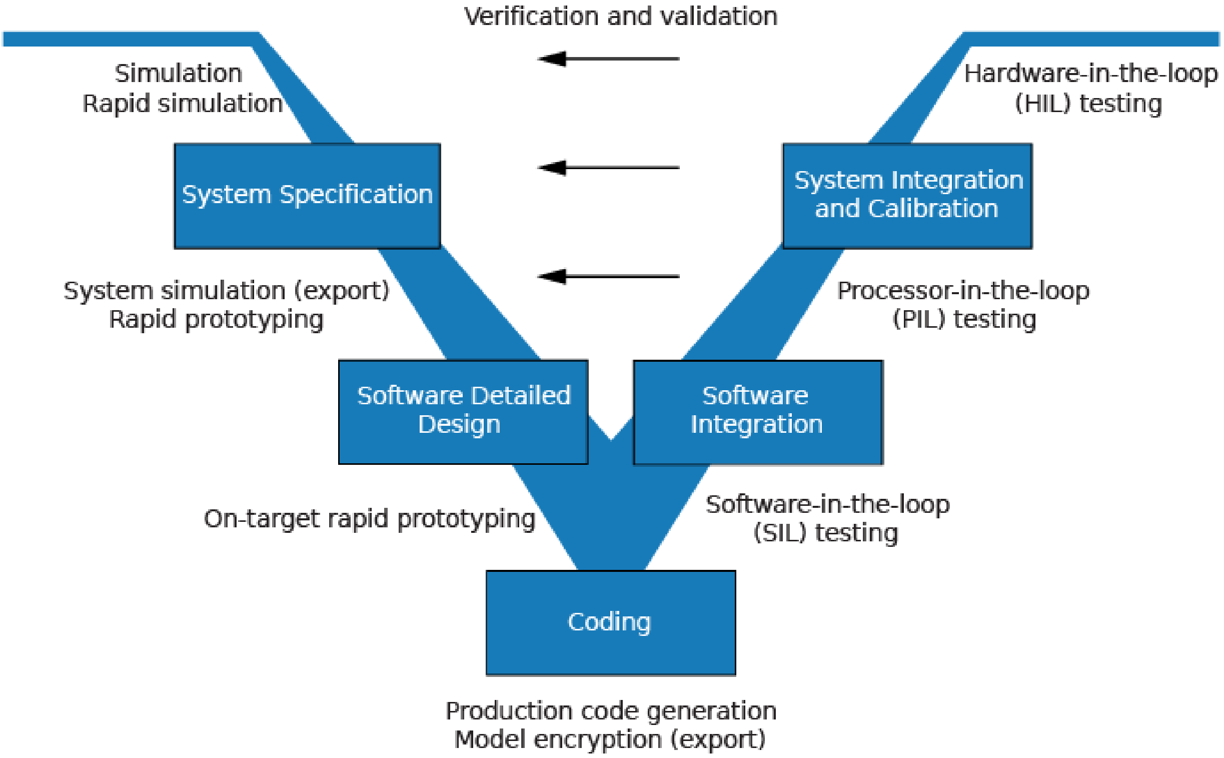
To learn how to model and generate code for commonly used C constructs using Simulink blocks, Stateflow charts, and MATLAB functions, see “Modeling Patterns for C Code Constructs”.

Validation and Verification for System Development

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

V-Model for System Development

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



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

- "Types of Simulation and Prototyping in the V-Model" on page 1-6
- "Types of In-the-Loop Testing in the V-Model" on page 1-7

Types of Simulation and Prototyping in the V-Model

Use the V-model for system development for different types of simulation and prototyping, such as rapid simulation, system simulation, rapid prototyping, and rapid prototyping on target hardware.

This table compares the types of simulation and prototyping identified on the left side of the V-model diagram shown in “V-Model for System Development” on page 1-6.

	Simulation	Rapid Simulation	System Simulation, Rapid Prototyping	Rapid Prototyping on Target Hardware
Purpose	Test and validate functionality of concept model	Refine, test, and validate functionality of concept model in nonreal time	Test new ideas and research	Refine and calibrate design during development process
Execution hardware	Development computer	Development computer Standalone executable runs outside of MATLAB and Simulink environments	PC or nontarget hardware	Embedded computing unit (ECU) or near-production hardware
Code efficiency and I/O latency	Not applicable	Not applicable	Less emphasis on code efficiency and I/O latency	More emphasis on code efficiency and I/O latency
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	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 by using 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 for less expense and more convenience

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

This table compares types of in-the-loop testing for verification identified on the right side of the V-model diagram shown in “V-Model for System Development” on page 1-6.

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

Target Environments and Applications

In this section...
"About Target Environments" on page 1-9
"Types of Target Environments" on page 1-9
"Applications of Supported Target Environments" on page 1-11

About Target Environments

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 program 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 development computer to generating a complete executable program that uses a custom build process for custom hardware, in an environment completely separate from the development 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 this table.

Target Environment	Description
Development computer	The computer that runs MATLAB and Simulink. A development 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.

Target Environment	Description
Real-time simulator	A different computer from the development 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 The generated code runs in real time. The exact nature of execution varies based on the particular behavior of the system hardware and RTOS. A real-time simulator connects to a development computer for data logging, interactive parameter tuning, and Monte Carlo batch execution studies.
Embedded microprocessor	A computer that you eventually disconnect from a development 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.

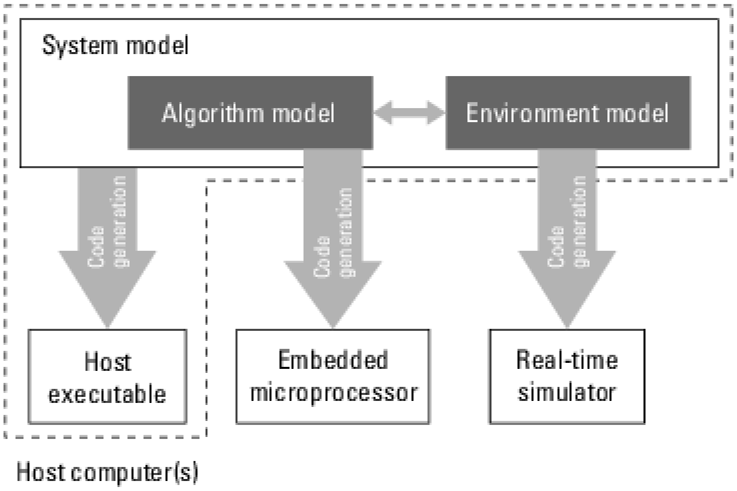
[b] Linux is a registered trademark of Linus Torvalds.

A target environment can:

- Have single- or multiple-core CPUs
- Be a standalone computer or communicate as part of a computer network

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

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

Application	Description
Development Computer	
"Acceleration"	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	Execute code generated for a model in non-real-time on the development computer, but outside the context of the MATLAB and Simulink environments.
Shared Object Libraries	Integrate components into a larger system. You provide generated source code and related dependencies for building a system in another environment or in a shared library to which other code can dynamically link.
"Protect Models to Conceal Contents"	Generate a protected model for use by a third-party vendor in another Simulink simulation environment.
Real-Time Simulator	

Application	Description
Real-Time Rapid Prototyping	Generate, deploy, and tune code on a real-time simulator connected to the system hardware, for example, physical plant or vehicle. being controlled. Crucial for validating whether a component can control the physical system.
Shared Object Libraries	Integrate generated source code and dependencies for components into a larger system that is built in another environment. You can use shared library files for intellectual property protection.
Hardware-in-the-Loop (HIL) Simulation	Run a simulation that pairs physical hardware, such as a controller, with a virtual real-time implementation of physical components on a real-time target computer, including a plant, sensors, actuators, and the environment. Use HIL simulations to test and validate physical hardware and a controller algorithm by including the effects of component response in real time to realistic stimuli. Testing commonly compares the HIL simulation results to system requirements. Validation compares HIL simulation results to user requirements. Often HIL simulations are referred to as closed-loop simulations due to the component response to the physical environment stimuli.
Embedded Microprocessor	
“Code Generation”	From a model, generate code that is optimized for speed, memory usage, simplicity, and compliance with industry standards and guidelines.
“Software-in-the-Loop Simulation”	Compile generated or external source code intended for production and execute the code as a separate process from the rest of the Simulink model on your development computer. Goals include initial source code testing and verification by comparing SIL and model simulation results or comparing SIL results to requirements by using back-to-back testing. Commonly used with external code integration, bit-accurate fixed-point math, and coverage analysis.

Application	Description
"Processor-in-the-Loop Simulation"	Cross-compile generated or external source code intended for production on a development computer, and then download and run the object code on a target processor or an equivalent instruction set simulator. Goals include verification by comparing PIL simulation results against model or SIL simulation results and collecting execution time profiling data. Commonly used with external code integration, bit-accurate fixed-point math, and coverage analysis.
Hardware-in-the-loop (HIL) Simulation	Run a simulation that pairs physical hardware, such as a controller, with a virtual real-time implementation of physical components on a real-time target computer, including a plant, sensors, actuators, and the environment. Use HIL simulations to test and validate physical hardware and a controller algorithm by including the effects of component response in real time to realistic stimuli. Testing commonly compares the HIL simulation results to system requirements. Validation compares HIL simulation results to user requirements. Often HIL simulations are referred to as closed-loop simulations due to the component response to the physical environment stimuli.

MATLAB Tutorials

Embedded Coder Capabilities for Code Generation from MATLAB Code

The Embedded Coder product extends the MATLAB Coder product with capabilities that you can use for embedded software development. You can generate code that has the clarity and efficiency of professional handwritten code. For example, you can:

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

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

Goal	Project Setting	Code Configuration Object Property	More Information
Execution Time			
Control generation of floating-point data and operations	**Support only purely-integer numbers**	`PurelyIntegerCode`	N/A
Simplify array indexing in loops in the generated code	**Simplify array indexing**	`EnableStrengthReduction`	“Simplify Multiply Operations for Array Indexing in Loops”
Replace functions and operators in the generated code to meet application-specific code requirements	**Code replacement library** on the **Custom Code** tab	`CodeReplacementLibrary`	Embedded Coder offers additional libraries and the ability to create and use custom code. See “Code Replacement Customization”.
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 2-12

Goal	Project Setting	Code Configuration Object Property	More Information
Generate MATLAB function help text in the function banner	**MATLAB function help text**	`MATLABFcnDesc`	"Include Comments in Generated C/C++ Code" on page 2-12
Include comments in the generated code that contain summaries for requirements linked to MATLAB code. In the code generation report, the comments are hyperlinks that to the requirements in the **Requirements Editor**	**Requirement summaries as comments**	`ReqsInCode`	"Requirements Traceability for Code Generated from MATLAB Code" (Requirements Toolbox)
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 2-7
Specify that the extern keyword is included in declarations of generated external functions	**Preserve extern keyword in function declarations**	`PreserveExternInFcnDecls`	N/A
Specify the level of parenthesization in the generated code	**Parentheses**	`ParenthesesLevel`	N/A
Specify whether to replace multiplications by powers of two with signed left bitwise shifts in the generated code	**Use signed shift left for fixed-point operations and multiplication by powers of 2**	`EnableSignedLeftShifts`	"Control Signed Left Shifts in Generated Code"
Specify whether to allow signed right bitwise shifts in the generated code	**Allow right shifts on signed integers**	`EnableSignedRightShifts`	N/A
Control data type casts in the generated code	**Casting mode** on the **All Settings** tab	`CastingMode`	"Control Data Type Casts in Generated Code"
Specify the indent style for the generated code	**Indent style** on the **All Settings** tab **Indent size** on the **All Settings** tab	`IndentStyle` `IndentSize`	"Specify Indent Style for C/C++ Code"
Specify the maximum number of columns before a line break in the generated code	**Column limit** on the **All Settings** tab	`ColumnLimit`	N/A
Specify custom names for MATLAB data types in generated code	**Enable custom data type replacement**	`EnableCustomReplacementTypes` `ReplacementTypes`	"Customize Data Type Replacement"

Goal	Project Setting	Code Configuration Object Property	More Information
Import custom data type definitions from external header files	**Import custom types from external header files**	`IsExtern HeaderFiles`	"Import Custom Data Type Definitions from External Header Files"
Customize generated C/C++ file names	**Generated source and header file name format**	`CustomFileNameStr`	"Customize C/C++ File Names Generated from MATLAB Code"
Customize generated global variable identifiers	**Global variables**	`CustomSymbolStrGlobalVar`	"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**	`CustomSymbolStrTmpVar`	"Customize Generated Identifiers"
Customize generated identifiers for constant macros	**Constant macros**	`CustomSymbolStrMacro`	"Customize Generated Identifiers"
Customize generated identifiers for EMX Array types (Embeddable mxArray types)	**EMX Array Types**	`CustomSymbolStrEMXArray`	"Customize Generated Identifiers"
Customize generated identifiers for EMX Array (Embeddable mxArrays) utility functions	**EMX Array Utility Functions**	`CustomSymbolStrEMXArrayFcn`	"Customize Generated Identifiers"
Customize function interface in the generated code	**Initialize function required** on the **All Settings** tab **Terminate function required** on the **All Settings** tab	`IncludeInitializeFcn` `IncludeTerminateFcn`	N/A
Customize file and function banners	N/A	`CodeTemplate`	• "Generate Custom File and Function Banners for C/C++ Code" • "Code Generation Template Files for MATLAB Code"

Goal	Project Setting	Code Configuration Object Property	More Information
Control declarations and definitions of global variables in the generated code	N/A	N/A	• “Storage Classes for Code Generation from MATLAB Code” • “Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code”
Debugging			
Generate a static code metrics report including generated file information, number of lines, and memory usage	**Static code metrics**	`GenerateCodeMetricsReport`	“Generating a Static Code Metrics Report for Code Generated from MATLAB Code”
Generate a code replacement report that summarizes the replacements used from the selected code replacement library	**Code replacements**	`GenerateCodeReplacementReport`	• “Verify Code Replacement Library”
Highlight single-precision, double-precision, and expensive fixed-point operations in the code generation report	**Highlight potential data type issues**	`HighlightPotentialDataTypeIssues`	“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**	`CodeReplacementLibrary`	Embedded Coder offers additional libraries and the ability to create and use custom code. See “Code Replacement Customization”.
Create and register application-specific implementations of functions and operators	N/A	N/A	“Code Replacement Customization”
Verification			
Interactively trace between MATLAB source code and generated C/C++ code	**Enable Code Traceability**	`EnableTraceability`	“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”

Goal	Project Setting	Code Configuration Object Property	More Information
Debug code during software-in-the-loop or processor-in-the-loop execution	**Enable source-level debugging for SIL or PIL** on the **Debugging** pane	`SILPILDebugging`	“Debug Generated Code During SIL or PIL 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	`CodeExecutionProfiling`	“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”.

Required Files

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

Create File in a Local Working Folder

1. Create a local working folder, for example, `c:\ecoder\work`.
2. Create a file `test_code_style.m` that contains this code:

```
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
```

Open the MATLAB Coder App

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

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

Specify Source Files

1. On the **Select Source Files** page, type or select the name of the entry-point function `test_code_style.m`.
2. In the **Project location** field, change the project name to `code_style.prj`.
3. Click **Next** to go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

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

1. Click **Let me enter input or global types directly**.
2. Click the field to the right of `x`.
3. From the list of options, select `int16`. Then, select `scalar`.
4. Click **Next** to go to the **Check for Run-Time Issues** step.

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

Check for Run-Time Issues

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

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow ▼.
2. In the **Check for Run-Time Issues** dialog box, enter code that calls `test_code_style` with an example input. For this example, enter `test_code_style(int16(4))`.
3. Click **Check for Issues**.

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

Configure 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 report** is selected.
   - On the **All Settings** tab, make sure that **Enable code traceability** is selected.

Generate C Code

Click **Generate**.

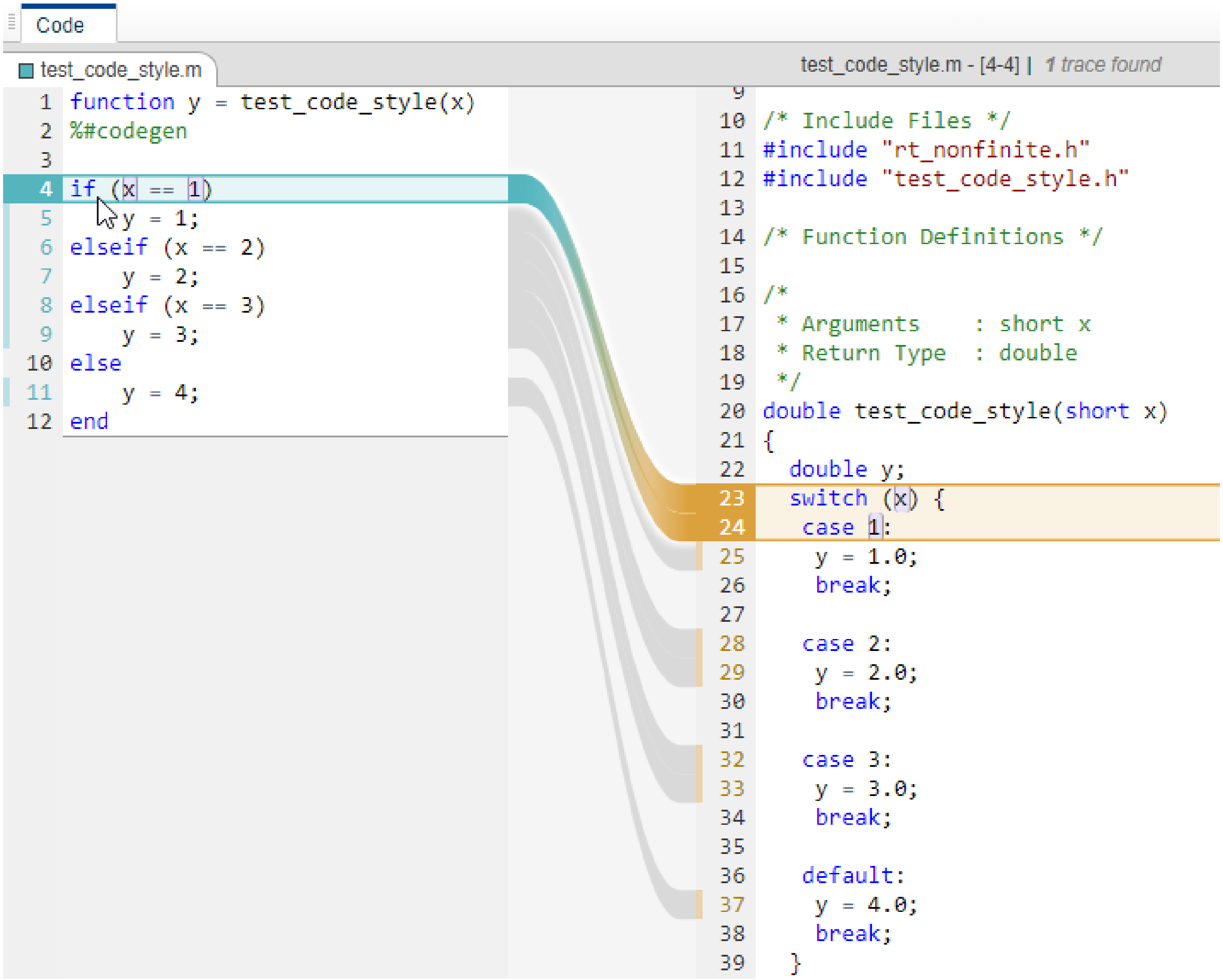
When code generation is complete, the code generator produces a C static library, `test_code_style.lib`, and C code in the `/codegen/lib/test_code_style` subfolder. The code generator provides a link to the report.

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

- “Generate C Code by Using the MATLAB Coder App”
- “Generate C Code at the Command Line”
- “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 the MATLAB source file `polar2cartesian.m` that contains this code:

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

Configuring Build Parameters

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

- `GenerateComments` to allow comments in the generated code.

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

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

Generating the C Code

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

- `-config` to pass in the code generation configuration object `cfg`.
- `-report` to create a code generation report.
- `-args` to specify the class, size, and complexity of the input parameters.

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

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

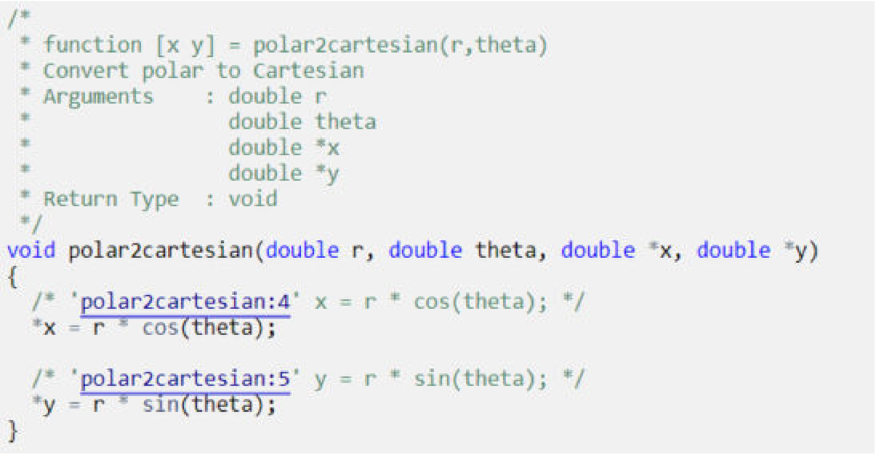
Viewing the Generated C Code

View the generated code in the code generation report.

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

   The generated code includes:

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

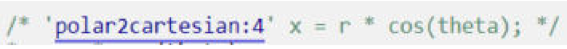


```
/*
 * function [x y] = polar2cartesian(r,theta)
 * Convert polar to Cartesian
 * Arguments    : double r
 *                double theta
 *                double *x
 *                double *y
 * Return Type  : void
 */
void polar2cartesian(double r, double theta, double *x, double *y)
{
  /* 'polar2cartesian:4'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

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

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

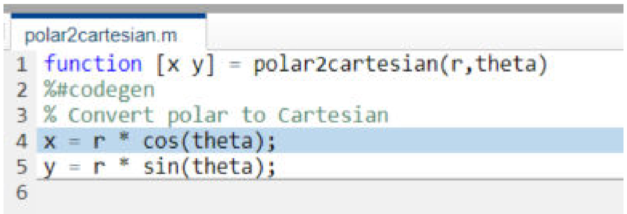
Tracing the Generated Code to the MATLAB Code

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



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

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



```
1 function [x y] = polar2cartesian(r,theta)
2 %#codegen
3 % Convert polar to Cartesian
4 x = r * cos(theta);
5 y = r * sin(theta);
6
```

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”
- “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

This example shows you how to generate C code from a Simulink® model of a roll axis autopilot algorithm by using the Embedded Coder® product.

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

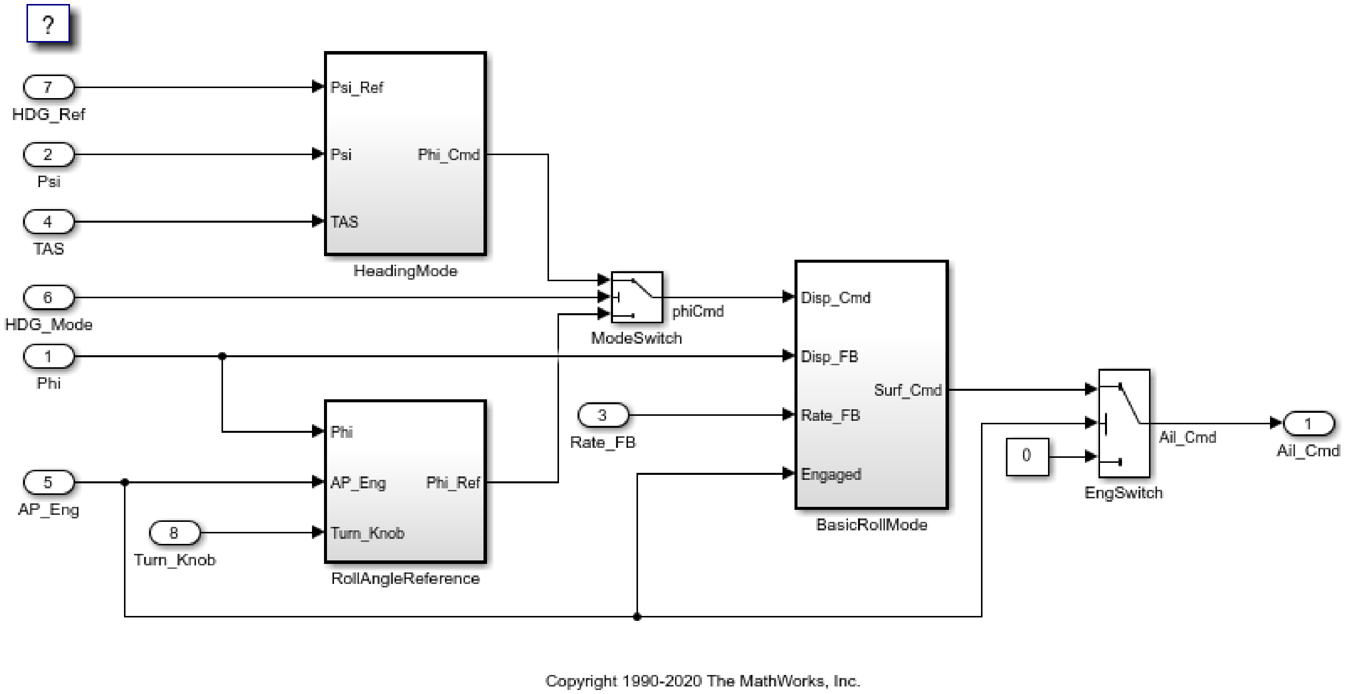
Generating and reviewing code for deployment to an embedded system can be as simple as preparing the model for code generation with the Quick Start tool. Then, with code tools accessible from the Simulink Editor, you can configure code interfaces, initiate code generation, and review the generated code.

Example Models

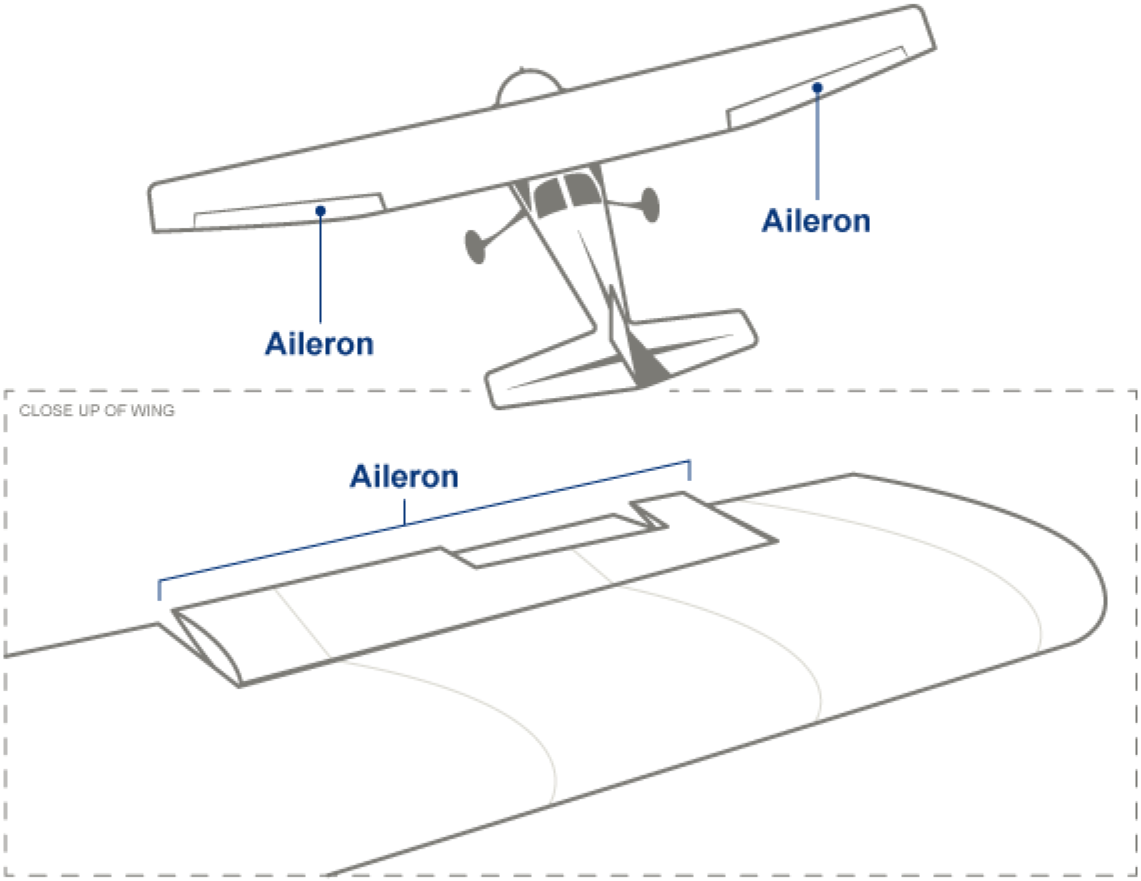
The tutorial uses example models `RollAxisAutopilot` and `RollAxisAutopilotHarness`. The models have been verified for simulation.

Open model `RollAxisAutopilot`.

```
open_system('RollAxisAutopilot.slx')
```

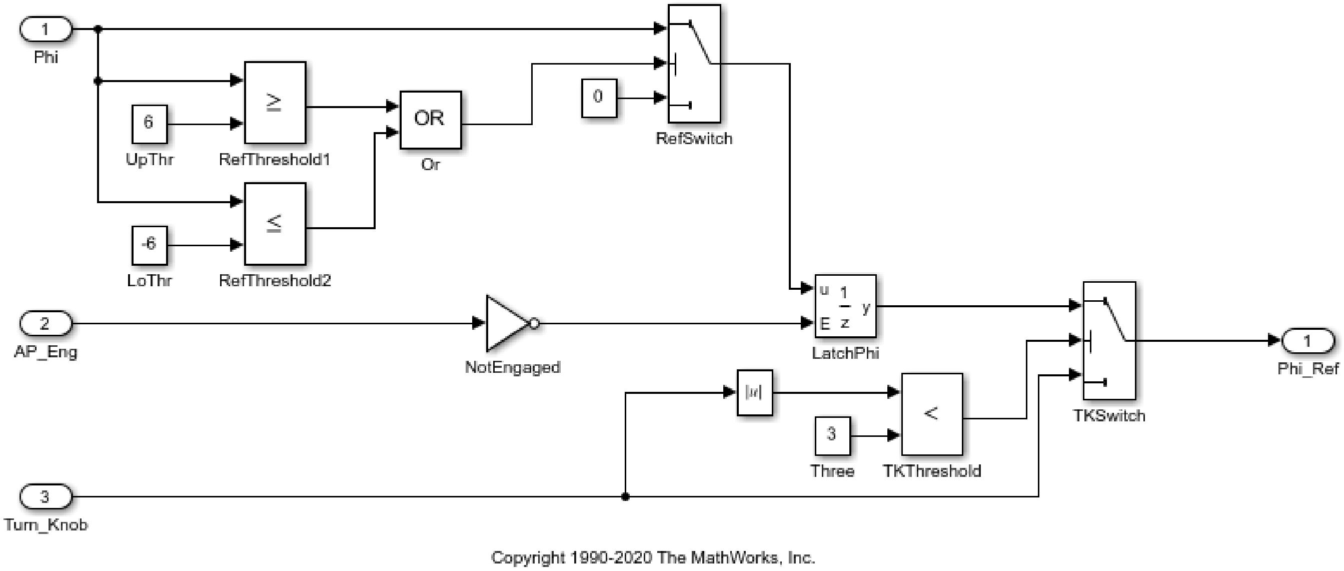


This model implements a basic roll axis autopilot algorithm, which controls the aileron position of an aircraft.

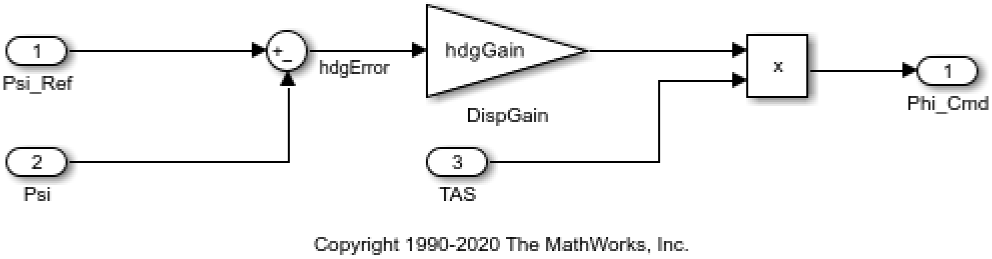


The model represents one component in the greater control system of an aircraft. Through the `HDG_Mode` signal, the control system places the model in one of two operating modes: roll attitude hold or heading hold. The `RollAngleReference` and `HeadingMode` subsystems calculate a roll attitude setpoint that supports one of the operating modes.

```
open_system('RollAxisAutopilot/RollAngleReference')
```

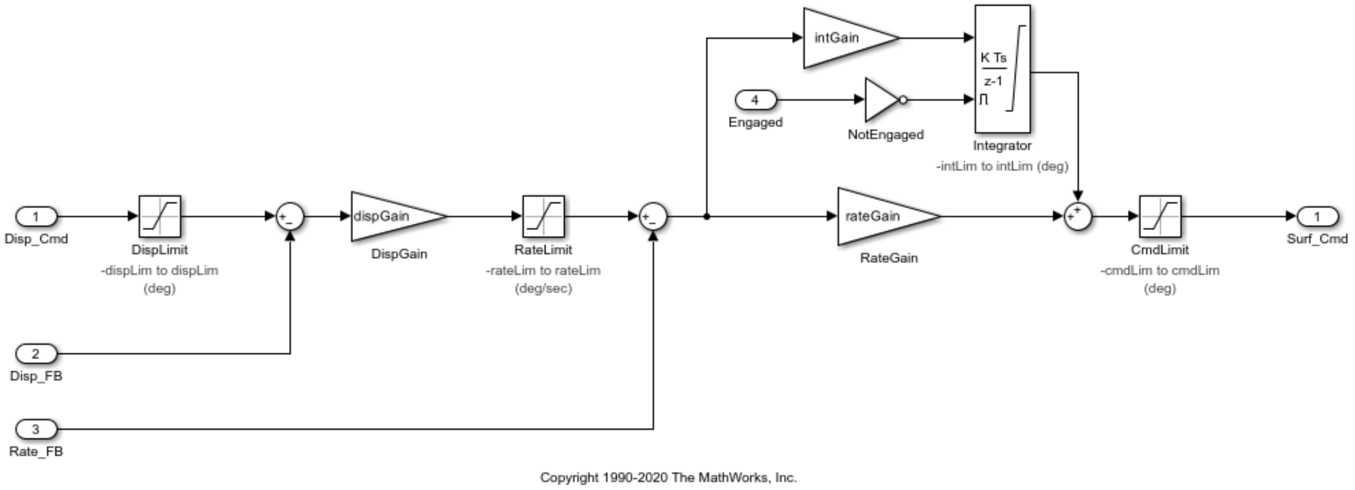


```
open_system('RollAxisAutopilot/HeadingMode')
```



Then, the `BasicRollMode` subsystem, a PID controller, calculates an aileron position command based on the setpoint and on feedback that indicates the measured roll attitude and rate of change. The model is designed to operate at 40 Hz.

```
open_system('RollAxisAutopilot/BasicRollMode')
```



The tutorial uses model `RollAxisAutopilotHarness` to test `RollAxisAutopilot`.

You will learn how to:

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

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

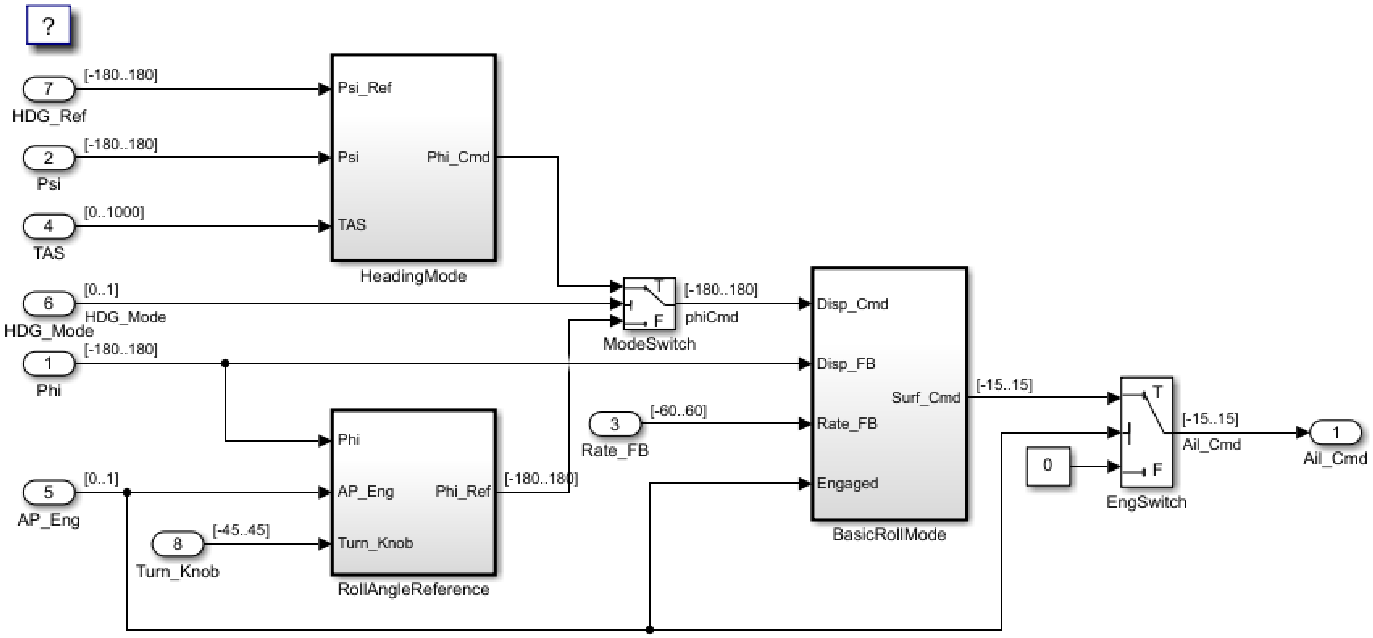
Generate Code by Using Embedded Coder Quick Start

Model `RollAxisAutopilot` represents an autopilot control system for an aircraft. You prepare `RollAxisAutopilot` for embedded code generation by using Embedded Coder Quick Start, which chooses fundamental code generation settings based on your goals and application.

Generate Code with Quick Start Tool

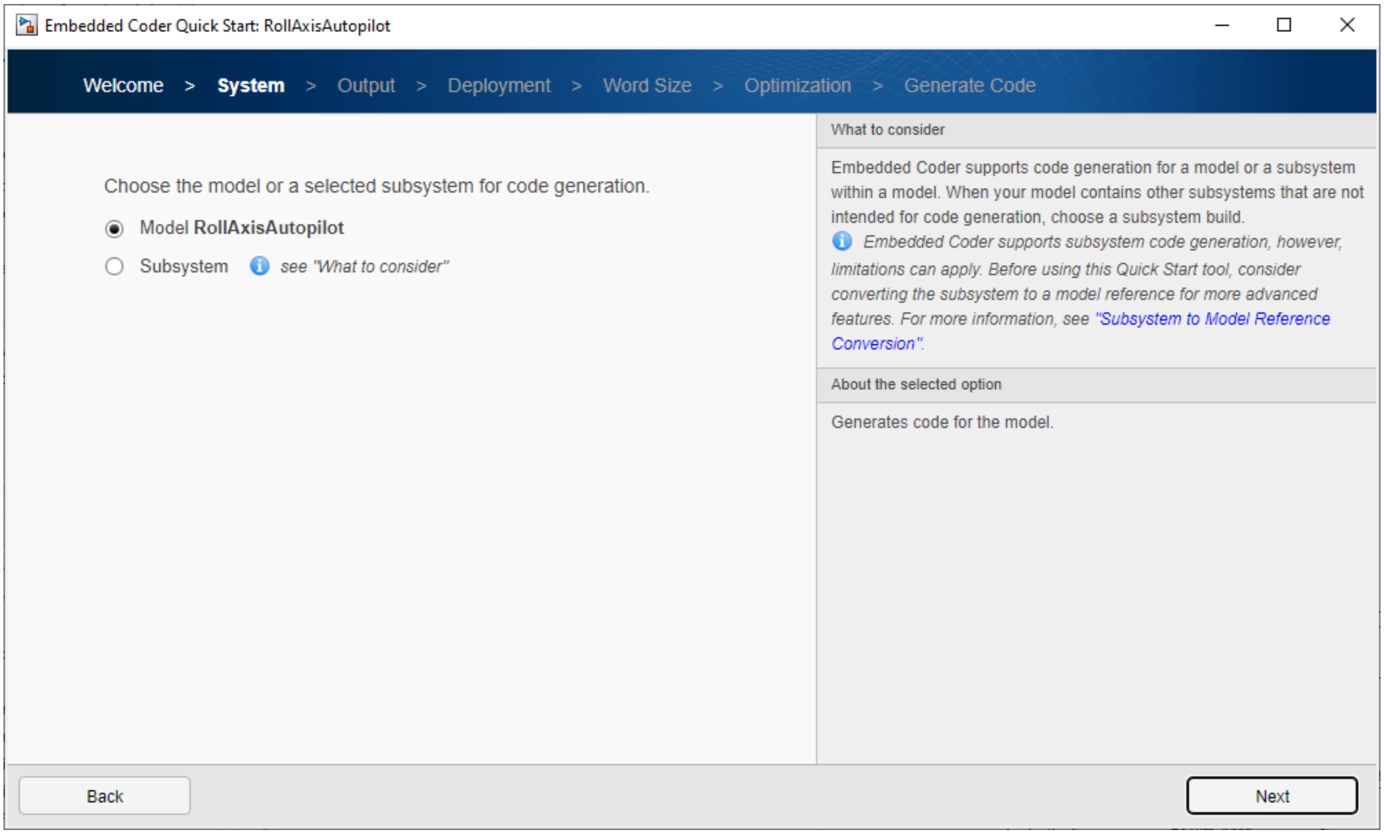
1 Open the model `RollAxisAutopilot` by typing this command:

```
openExample('RollAxisAutopilot');
```



2 If the **C Code** tab is not already open, in the Apps gallery, under **Code Generation**, click **Embedded Coder**.

3 On the **C Code** tab, click **Quick Start**.


Embedded Coder Quick Start: RollAxisAutopilot
Welcome > System > Output > Deployment > Word Size > Optimization > Generate Code

Choose the model or a selected subsystem for code generation.

● Model **RollAxisAutopilot**

○ Subsystem *see "What to consider"*

What to consider

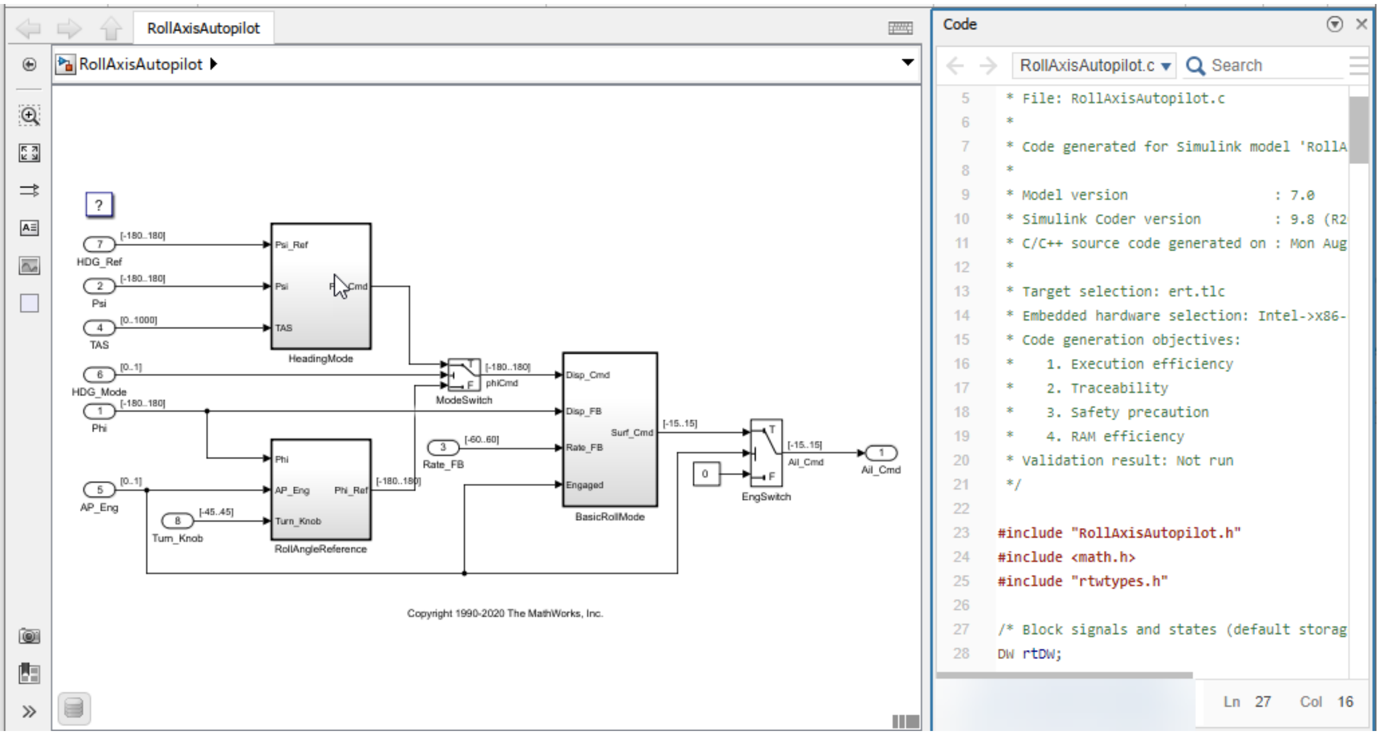
Embedded Coder supports code generation for a model or a subsystem within a model. When your model contains other subsystems that are not intended for code generation, choose a subsystem build.

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.

4. Advance through the steps of the Quick Start tool, stopping at the **Generate Code** step. Each step asks questions about the code that you want to generate. For this tutorial, use the defaults that are already selected. The tool validates your selections against the model and presents the parameter changes required to generate code.
5. In the **Generate Code** step, apply the proposed changes and generate code from `RollAxisAutopilot` by clicking **Next**.
6. Click **Finish**, then return to the **C Code** tab. From this tab you can configure code generation customizations, and then check the results in the Code view next to the model.



```
 * File: RollAxisAutopilot.c
 *
 * Code generated for Simulink model 'RollA
 *
 * Model version                  : 7.0
 * Simulink Coder version         : 9.8 (R2
 * C/C++ source code generated on : Mon Aug
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. Traceability
 *    3. Safety precaution
 *    4. RAM efficiency
 * Validation result: Not run
 */

#include "RollAxisAutopilot.h"
#include <math.h>
#include "rtwtypes.h"

/* Block signals and states (default storag
DW rtDW;
```

Inspect the Generated Code

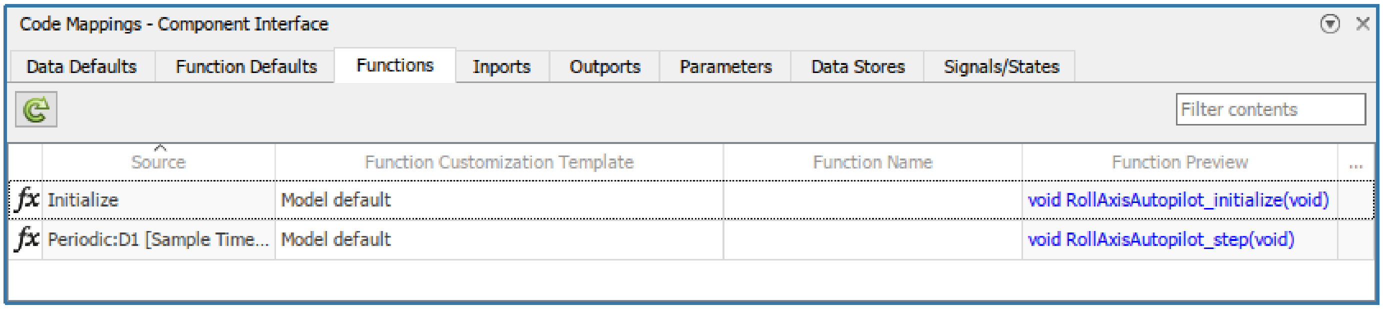
The generated code appears in two primary files: `RollAxisAutopilot.c` and `RollAxisAutopilot.h`. In your MATLAB current folder, the `RollAxisAutopilot_ert_rtw` folder contains these primary files.

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

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

Open the Code Mappings editor by clicking **Code Mappings - Component Interface** below the model diagram. On the **Functions** tab, you can see the individual entry-point functions that the code generator produces. You call these generated functions from external code or from a version of a generated main function that you modify. For the base-rate step function of a rate-based model and for step functions for export function models, you can customize the function name and arguments.

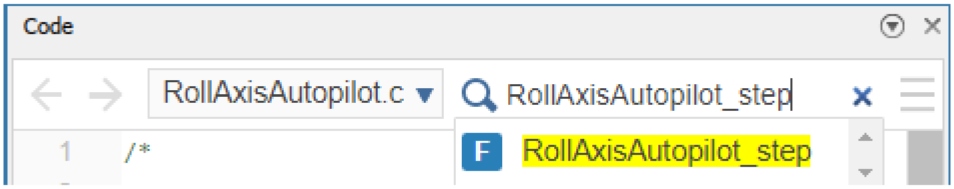
Review the list of entry-point functions that the code generator produces for the model. Use this view to selectively specify for each function a function customization template (code definition) and name. For this tutorial, the code generator uses default (shipped) settings for the customization template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



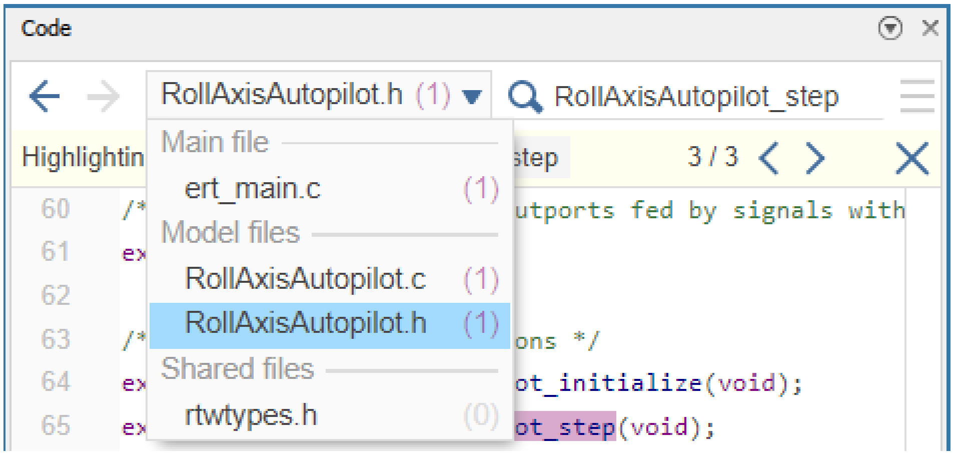
Source	Function Customization Template	Function Name	Function Preview
Initialize	Model default		void RollAxisAutopilot_initialize(void)
Periodic:D1 [Sample Time...	Model default		void RollAxisAutopilot_step(void)

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

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



3. Use the arrows on the right to step through each instance, including the step function definition in `RollAxisAutopilot.c` and the declaration in `RollAxisAutopilot.h`. You can also see the number of search hits in each file from the file menu in the upper left corner.



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

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

Configure Data Interface

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

Customize the data interface of model `RollAxisAutopilot` by configuring function `RollAxisAutopilot_step` to:

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

To make the changes, use these external code files in your current MATLAB working folder.

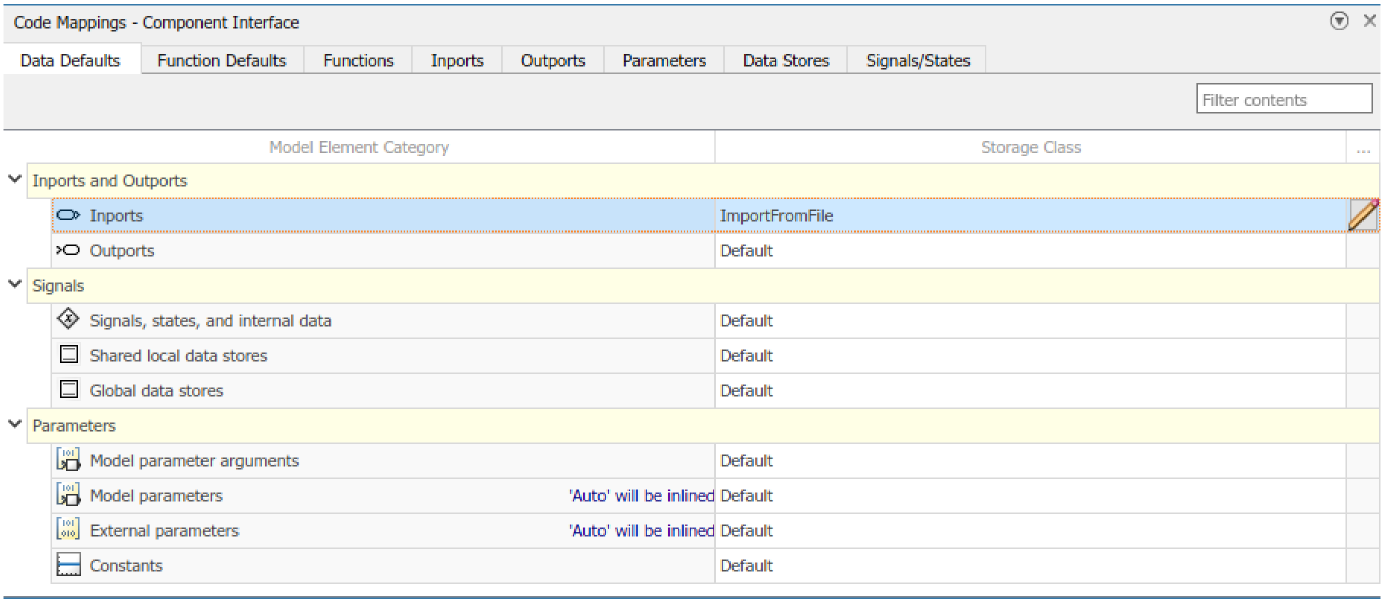
- `roll_input_data.c`
- `roll_input_data.h`
- `roll_heading_mode.c`
- `roll_heading_mode.h`

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

Configure Default Code Generation for Data

Configure default code generation configurations for model inports and outports.

1. In the **C Code** tab, select **Code Interface** > **Default Code Mappings**.
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. In the Code Mappings editor, under **Inports and Outports**, select category **Inports**. Set the default storage class to `ImportFromFile`.



Model Element Category		Storage Class
Inports and Outports		
Inports		ImportFromFile
Outports		Default
Signals		
Signals, states, and internal data		Default
Shared local data stores		Default
Global data stores		Default
Parameters		
Model parameter arguments		Default
Model parameters	'Auto' will be inlined	Default
External parameters	'Auto' will be inlined	Default
Constants		Default

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

3 Click the ✎ icon and set **HeaderFile** to `roll_input_data.h`.

4 To see how the `extern` declarations in external header file `roll_input_data.h` name the input variables, in the MATLAB Command Window, open `roll_input_data.h` located in your current working folder.

```
extern boolean_T AP_Eng;
extern real32_T HDG_Ref;
extern real32_T Rate_FB;
extern real32_T Phi;
extern real32_T Psi;
extern real32_T TAS;
extern real32_T Turn_Knob;
```

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

   a Open the Model Configuration Parameters dialog box. In the toolstrip, on the **C Code** tab, click **Settings**.

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

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

   d Apply the change.

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

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

7 Click the ✎ icon and set **HeaderFile** to `roll_output_data.h` and **DefinitionFile** to `roll_output_data.c`.

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

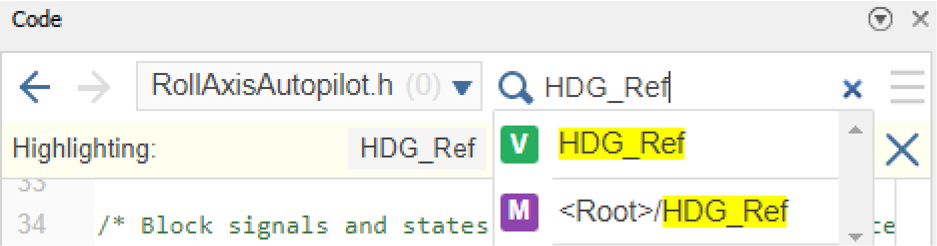
9 Save the model. Regenerate the code by clicking **Build**.

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

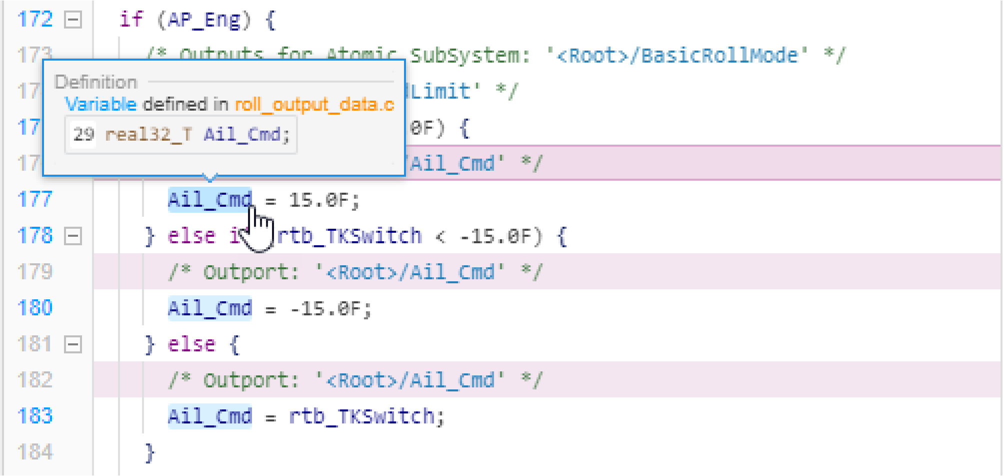
The model is configured to open the code generation report after code generation is complete. Minimize this report window for exploration later in this tutorial.

10 You configured Inport blocks to use an external header file to declare and define input variables. In the Code view, confirm that the generated code includes this external header file by searching for `roll_input_data.h`.

11 Search for the root level Inport block name, `HDG_Ref`. As you type, choose the search suggestion with the green V icon. This search suggestion finds instances of `HDG_Ref` used as a variable in the generated code. Confirm that `HDG_Ref` is defined as a separate global variable.



12 In the model, `RollAxisAutopilot`, click the Outport block `Ail_Cmd`. Place your cursor over the ellipsis menu above the block and click **Navigate To Code**. The Code view highlights code in `RollAxisAutopilot.c` that corresponds to the block. In the code, place your cursor over the output variable `Ail_Cmd`. The traceability dialog box displays variable definitions. The dialog box confirms that `Ail_Cmd` is defined as a separate global variable. Click the definition code to see the definition in `roll_output_data.c`.



Definition
Variable defined in roll_output_data.c
29 real32_T Ail_Cmd;

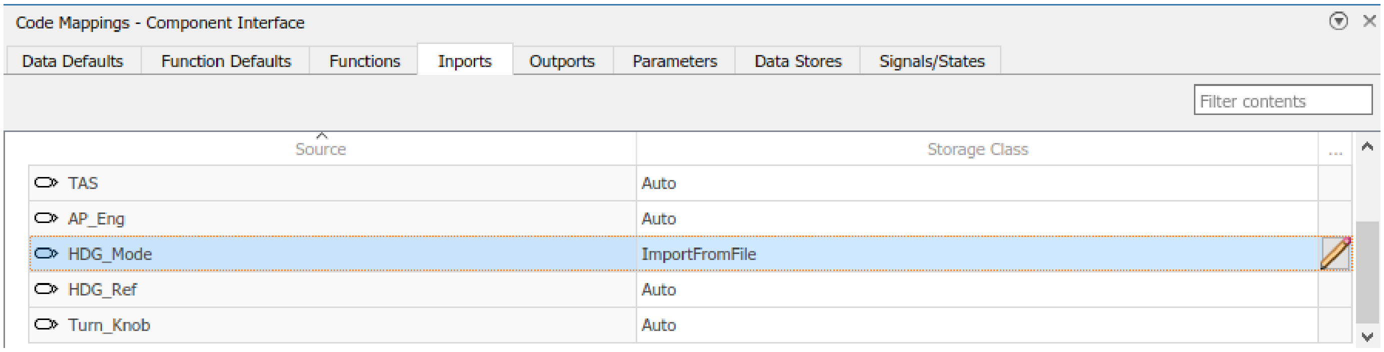
```
if (AP_Eng) {
  /* Outputs for Atomic SubSystem: '<Root>/BasicRollMode' */
                                 Limit' */
                                 0F) {
                                 Ail_Cmd' */
    Ail_Cmd = 15.0F;
  } else if (rtb_TKSwitch < -15.0F) {
    /* Outport: '<Root>/Ail_Cmd' */
    Ail_Cmd = -15.0F;
  } else {
    /* Outport: '<Root>/Ail_Cmd' */
    Ail_Cmd = rtb_TKSwitch;
  }
```

Override Default Settings for Individual Data Elements

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

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

1. In the Code Mappings editor, on the **Inports** tab, select source `HDG_Mode`.
2. Set **Storage Class** to `ImportFromFile`.



Source	Storage Class
TAS	Auto
AP_Eng	Auto
HDG_Mode	ImportFromFile
HDG_Ref	Auto
Turn_Knob	Auto

3. Click the ✎ icon and set **Identifier** to `HDG_Mode` and **Header File** to `roll_heading_mode.h`.

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

   ```
   extern boolean_T HDG_Mode;
   ```
4. Save the model and regenerate the code.

   Minimize the code generation report window for exploration later in this tutorial.
5. In the Code view, search for `roll_heading_mode.h` and confirm that it is included in the generated code with the default configuration file `roll_input_data.h`.

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

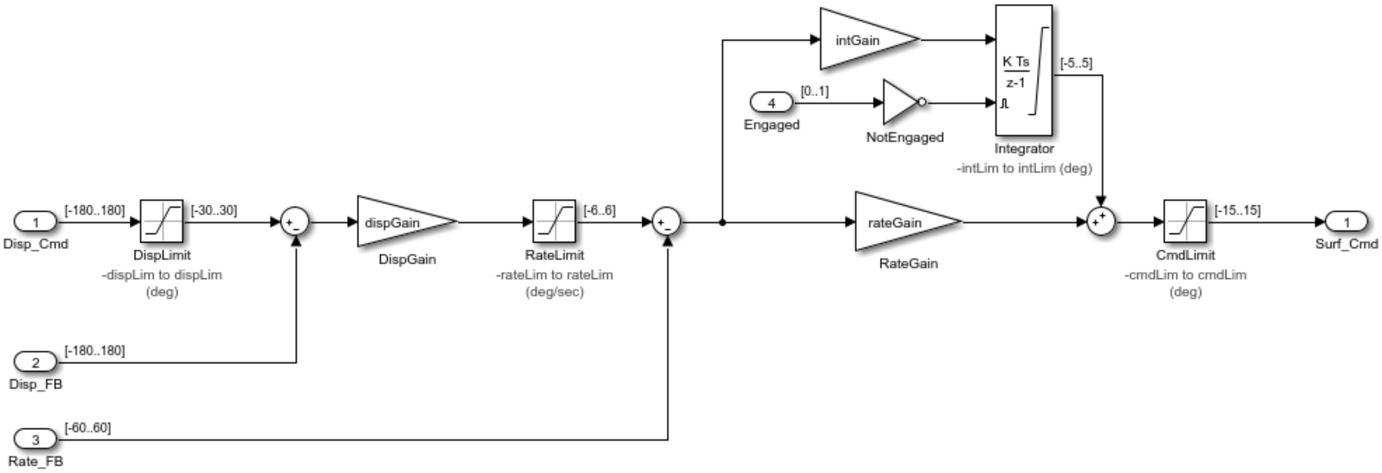
Next, configure a model parameter to be a global variable in the generated code. As a global variable, you can tune the parameter value at run time.

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

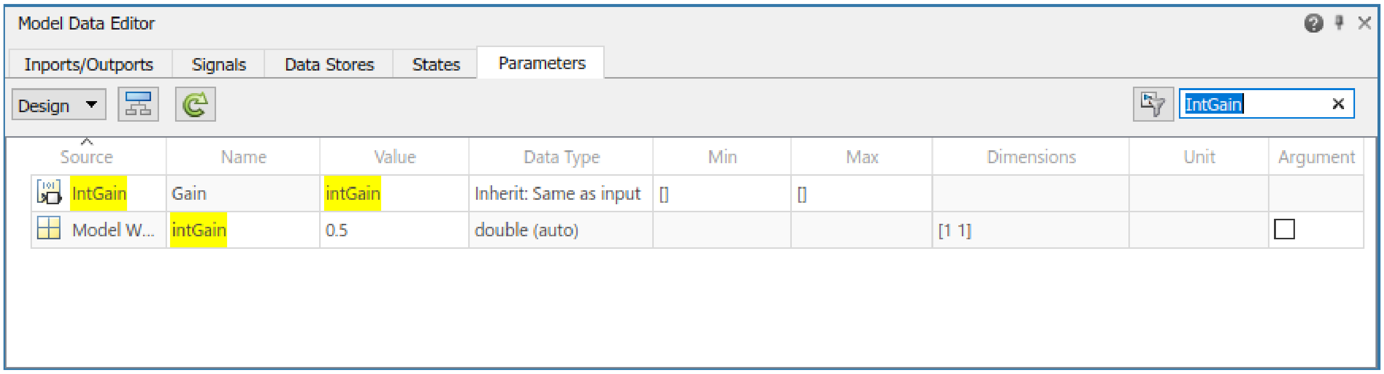
By default, code generation optimizations eliminate storage for model parameters and most signals that do not participate in the entry-point function interface. To make parameters tunable and related signals accessible, identify them by configuring them explicitly.

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

1. Open the `BasicRollMode` subsystem.



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



Source	Name	Value	Data Type	Min	Max	Dimensions	Unit	Argument
IntGain	Gain	intGain	Inherit: Same as input	[]	[]			
Model W...	intGain	0.5	double (auto)			[1 1]		□

5. In the **Source** column, click `IntGain`. That Gain block appears highlighted in the model diagram.
6. In the **Value** column, next to `intGain`, click the action button (button with three vertical dots) and select **Explore**.
7. Convert the model workspace variable to a parameter object. In the Model Explorer, right-click `intGain` and select `Convert to parameter object`.

8 In the **Dialog** pane, on the **Code Generation** tab, click **Configure in Coder App**.

9 In the Code Mappings editor, on the **Parameters** tab, change the **Storage Class** setting for `intGain` to `Model default`, which indicates that the parameter object prevents code generation optimizations from eliminating storage for the variable. With this setting, the object uses the storage class specified in the Code Mappings editor as the data default for category **Model parameters**.

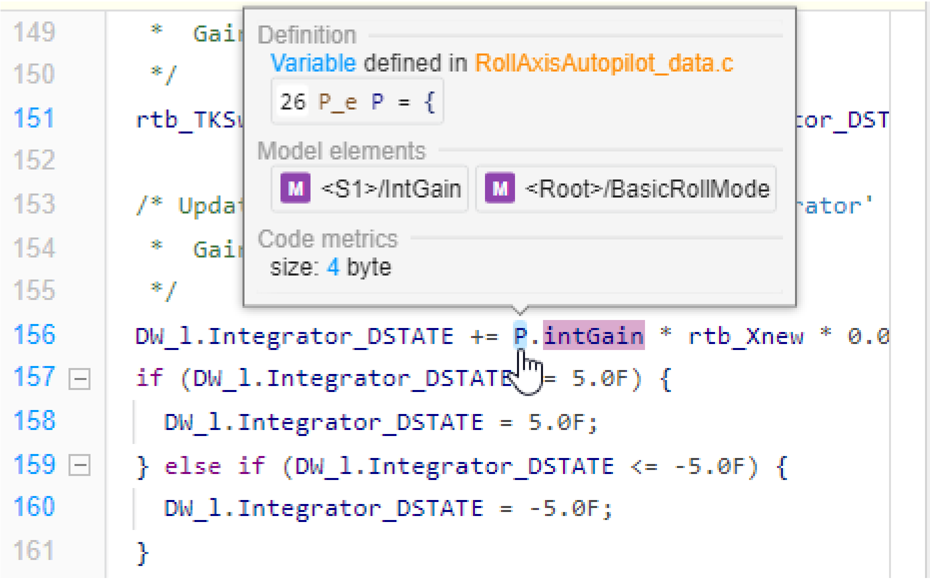
10 Save the model and regenerate the code.

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

11 In the Code view:

   a Search for `intGain`.

   b In `RollAxisAutopilot.c`, place your cursor over the `P` in the highlighted code `P.intGain`.



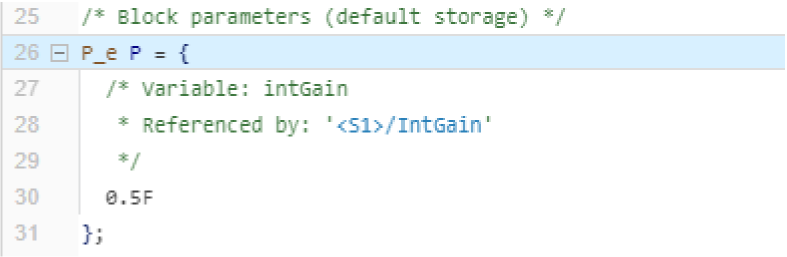
```
 *  Gai
 */
rtb_TKS                                                      tor_DST

/* Upda                                                      ator'
 *  Gai
 */
DW_l.Integrator_DSTATE += P.intGain * rtb_Xnew * 0.0
if (DW_l.Integrator_DSTATE >= 5.0F) {
  DW_l.Integrator_DSTATE = 5.0F;
} else if (DW_l.Integrator_DSTATE <= -5.0F) {
  DW_l.Integrator_DSTATE = -5.0F;
}
```

Definition
Variable defined in RollAxisAutopilot_data.c
26 P_e P = {

Model elements
M <S1>/IntGain
M <Root>/BasicRollMode

Code metrics
size: 4 byte

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



```
/* Block parameters (default storage) */
P_e P = {
  /* Variable: intGain
   * Referenced by: '<S1>/IntGain'
   */
  0.5F
};
```

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

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

Compare Model Simulation and Generated Code Results

In this section...

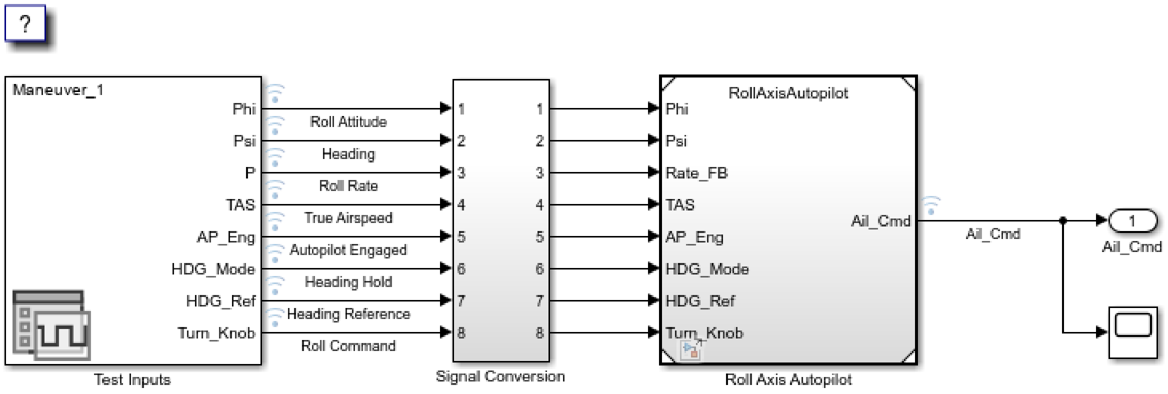
In this step of the tutorial, you verify that when executed, the code is numerically equivalent to the algorithm modeled in Simulink. You use a test harness model to simulate `RollAxisAutopilot` in normal mode and in SIL mode, then compare the simulations by using the Simulation Data Inspector.

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

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

Inspect and Configure Test Harness Model

Model `RollAxisAutopilotHarness` references the model-under-test, `RollAxisAutopilot`, 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.



Copyright 1990-2022 The MathWorks, Inc.

1. Open the model `RollAxisAutopilotHarness` by typing this command:

```
openExample('RollAxisAutopilotHarness');
```

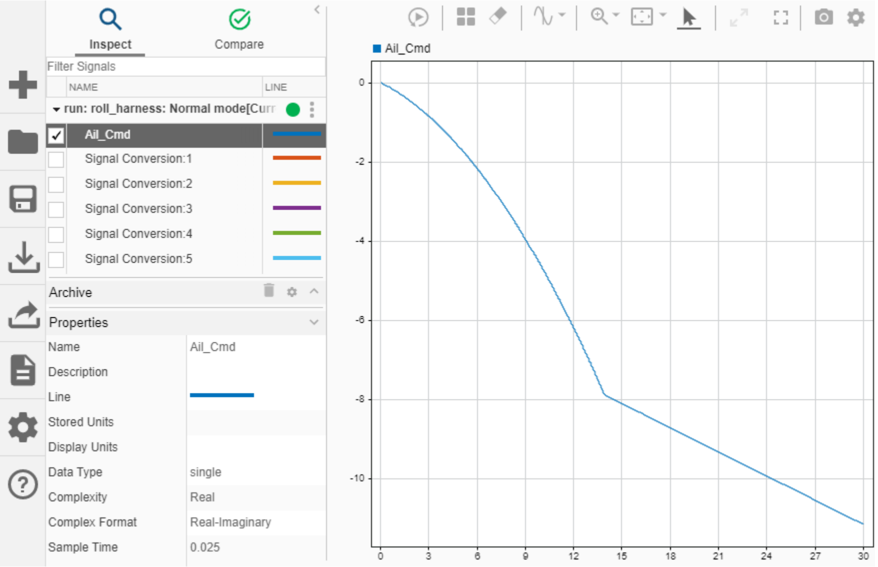
2. If you closed your copy of model `RollAxisAutopilot`, reopen it.

3. In the RollAxisAutopilotHarness model, right-click the Model block and select **Subsystem & Model Reference** > **Refresh Selected Model Block**.
4. Save a copy of RollAxisAutopilotHarness in the current working folder.
5. Open the Configuration Parameters dialog boxes for RollAxisAutopilotHarness and RollAxisAutopilot.
6. To run SIL and PIL simulations, on the **Code Generation** pane, verify that parameter **Generate code only** is cleared. Do this for both models.
7. For both models, on the **Hardware Implementation** pane, expand **Device details**. Verify that **Support long long** is selected.
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 RollAxisAutopilotHarness model, open the Model Data Editor. On the **Modeling** tab, click **Model Data Editor**.
2. In the Model Data Editor, select the **Signals** tab.
3. Set the **Change view** list to Instrumentation.
4. In the data table, select all rows.
5. To configure signals to log simulation data to the Simulation Data Inspector, select a cleared check box in the **Log Data** column. When you are finished, make sure that all of the check boxes in the column are selected.
6. Right-click the Model block, Roll Axis Autopilot. From the context menu, select **Block Parameters**.
7. In the Block Parameters dialog box, for **Simulation mode**, verify that the Normal option is selected. Click **OK**.
8. Simulate RollAxisAutopilotHarness.
9. When the simulation is done, view the simulation results in the Simulation Data Inspector. If the Simulation Data Inspector is not already open, on the **Simulation** tab, click **Data Inspector**.
10. For the most recent (current) run, double-click the run name field and rename the run: roll_harness: Normal mode.
11. Select Ail_Cmd to plot the signal.



Name	Ail_Cmd
Description	
Line	
Stored Units	
Display Units	
Data Type	single
Complexity	Real
Complex Format	Real-Imaginary
Sample Time	0.025

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

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

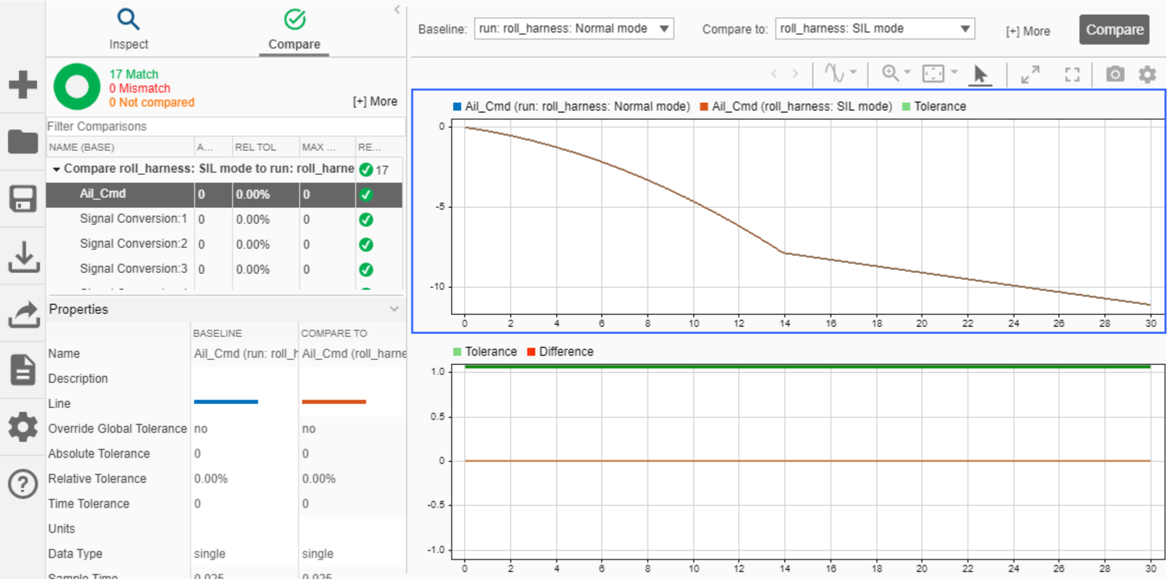
Compare Simulation Results

In the Simulation Data Inspector:

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

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

4 Click **Compare**.



Baseline: run: roll_harness: Normal mode

Compare to: roll_harness: SIL mode

17 Match
0 Mismatch
0 Not compared

NAME (BASE)	A...	REL TOL	MAX ...	RE...
Compare roll_harness: SIL mode to run: roll_harne				✓ 17
Ail_Cmd	0	0.00%	0	✓
Signal Conversion:1	0	0.00%	0	✓
Signal Conversion:2	0	0.00%	0	✓
Signal Conversion:3	0	0.00%	0	✓

Properties

	BASELINE	COMPARE TO
Name	Ail_Cmd (run: roll_h	Ail_Cmd (roll_harne
Description		
Line		
Override Global Tolerance	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

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

Next, explore ways that you can deploy generated code.

Deploy the Generated Code

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

Example Main Program

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

Explore the example `main` program generated for model `RollAxisAutopilot`.

1. If not already open, open your copy of the model `RollAxisAutopilot`.
2. In the Apps gallery, click **Embedded Coder**.
3. Regenerate the code.
4. In the Code view, select file `ert_main.c`.
5. Click in the **Search** field and select function `rt_OneStep`.
6. Explore the incomplete wrapper function `rt_OneStep`. This function calls the model execution entry-point function, `RollAxisAutopilot_step`. Your application code can call `rt_OneStep` to run the model algorithm during each execution cycle.
7. Click in the **Search** field and select function `main`.
8. Explore the incomplete example `main` function. This function outlines the order and context in which your application code can call `rt_OneStep` and other model entry-point functions.

For more information, see “Deploy Applications to Target Hardware”.

Relocate Generated Code Files

Embedded Coder provides a pack-n-go utility for relocating static and generated code files for a model to another development environment. File relocation is necessary when your system or integrated development environment (IDE) does not include MATLAB and Simulink products. The utility packages the files in a compressed file that you can relocate and unpack by using a standard `zip` utility. You can apply the pack-n-go utility from graphical and programming interfaces. For more information, see “Relocate or Share Generated Code”.

Share and Archive Code Generation Report

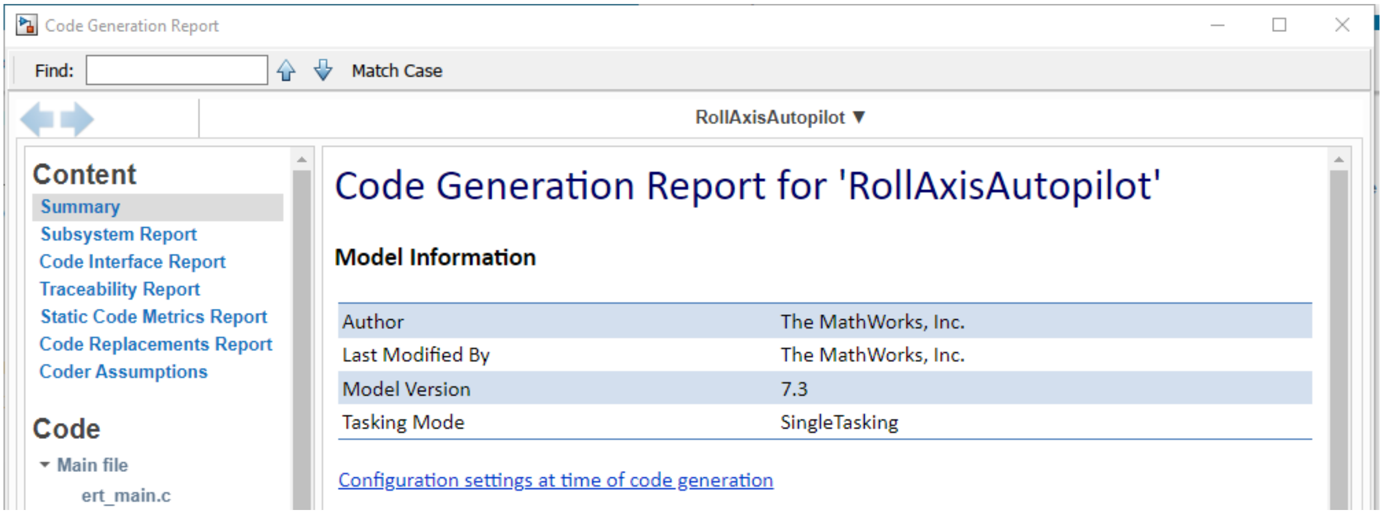
The Quick Start tool configures a model to produce an HTML code generation report. In addition to a summary of model and code information, the report includes:

- A subsystem report
- Generated code files
- A code interface report
- A traceability report
- A static code metrics report

- A code replacements report
- A coder assumptions report
- Optionally, a model web view

You can use this report outside of the Simulink environment, so it is suitable for sharing or for archival purposes. You can open the report from the tool or, on the **C Code** tab, click **Open Report**.

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



Code Generation Report for 'RollAxisAutopilot'

Model Information

Author	The MathWorks, Inc.
Last Modified By	The MathWorks, Inc.
Model Version	7.3
Tasking Mode	SingleTasking

Configuration settings at time of code generation

Explore Other Options

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

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

Task	Reference
Override default configurations for individual data elements	“C Code Generation Configuration for Model Interface Elements” and “Organize Parameter Data into a Structure by Using Struct Storage Class”
Compare normal mode simulation results against software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation results for numerical equivalency	“SIL and PIL Simulations” and “Choose a SIL or PIL Approach”
Collect code coverage metrics for generated code during SIL or PIL simulation	“Code Coverage”
Use generated example main code as a starting point to deploy generated executable programs	“Deploy Applications to Target Hardware”

Generate C++ Code from Simulink Models

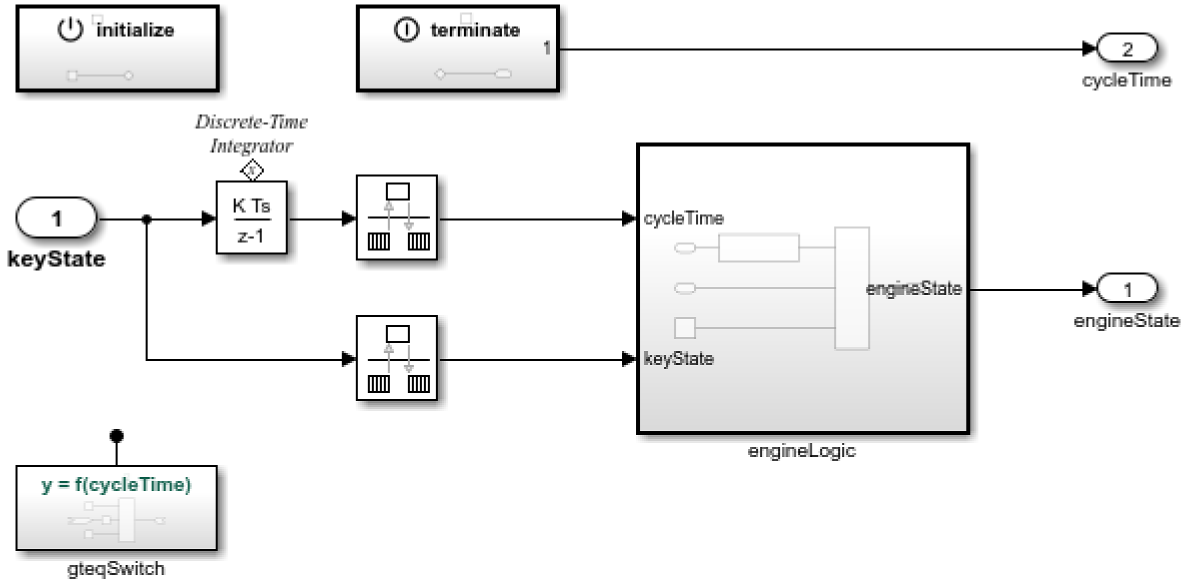
This example shows you how to generate C++ code from a Simulink® model of a key ignition system by using the Embedded Coder® product.

To generate and review code for deployment to an embedded system, you can prepare your model for code generation by using the Embedded Coder Quick Start tool. Then, using code tools accessible from the Simulink Editor, you configure code interfaces, initiate code generation, and review the generated code.

Example model

This tutorial uses the `CppClassWorkflowKeyIgnition` example model.

The `CppClassWorkflowKeyIgnition` model represents a key ignition system in the greater control system of a vehicle engine. When the ignition is turned on, the `keyState` input signal changes state and the control system waits a delay before switching the output `engineState`.



By using this tutorial, you learn how to:

- Generate C++ code by using the Embedded Coder Quick Start tool.
- Configure the C++ class interface.
- Deploy the generated C++ code.

For the first task, see “Generate C++ Code by Using Embedded Coder Quick Start” on page 3-26.

Generate C++ Code by Using Embedded Coder Quick Start

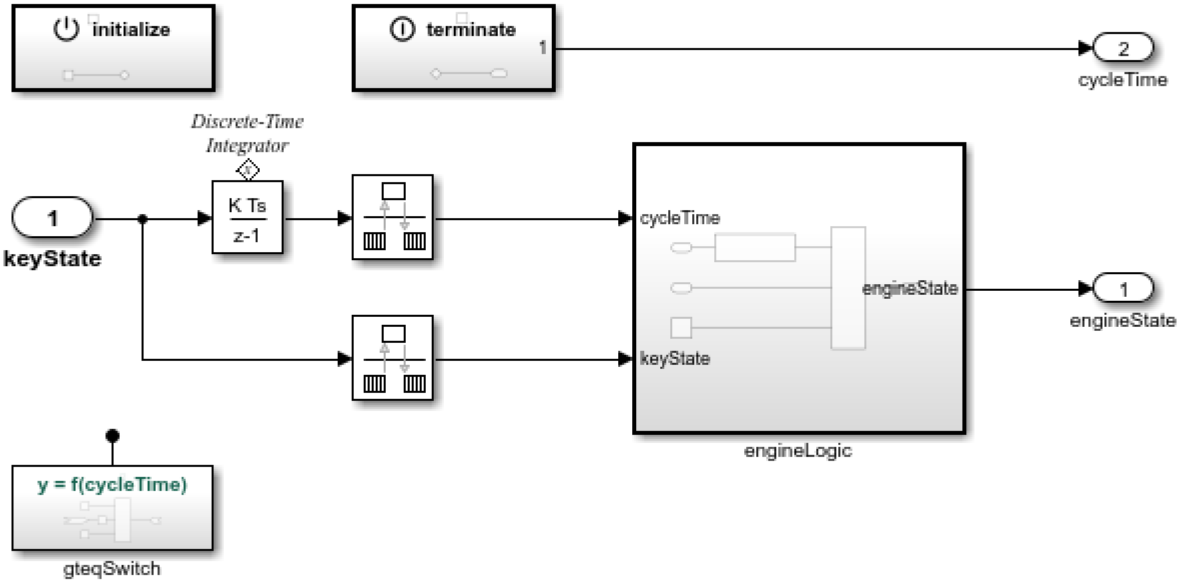
Prepare the `CppClassWorkflowKeyIgnition` model for embedded code generation by using the Embedded Coder Quick Start tool. The Quick Start tool chooses fundamental code generation settings based on your goals and application.

For this step of the tutorial, you generate code for the `CppClassWorkflowKeyIgnition` model, and then inspect the generated files. The `CppClassWorkflowKeyIgnition` model represents an ignition system for a vehicle.

Generate C++ Code by Using Quick Start Tool

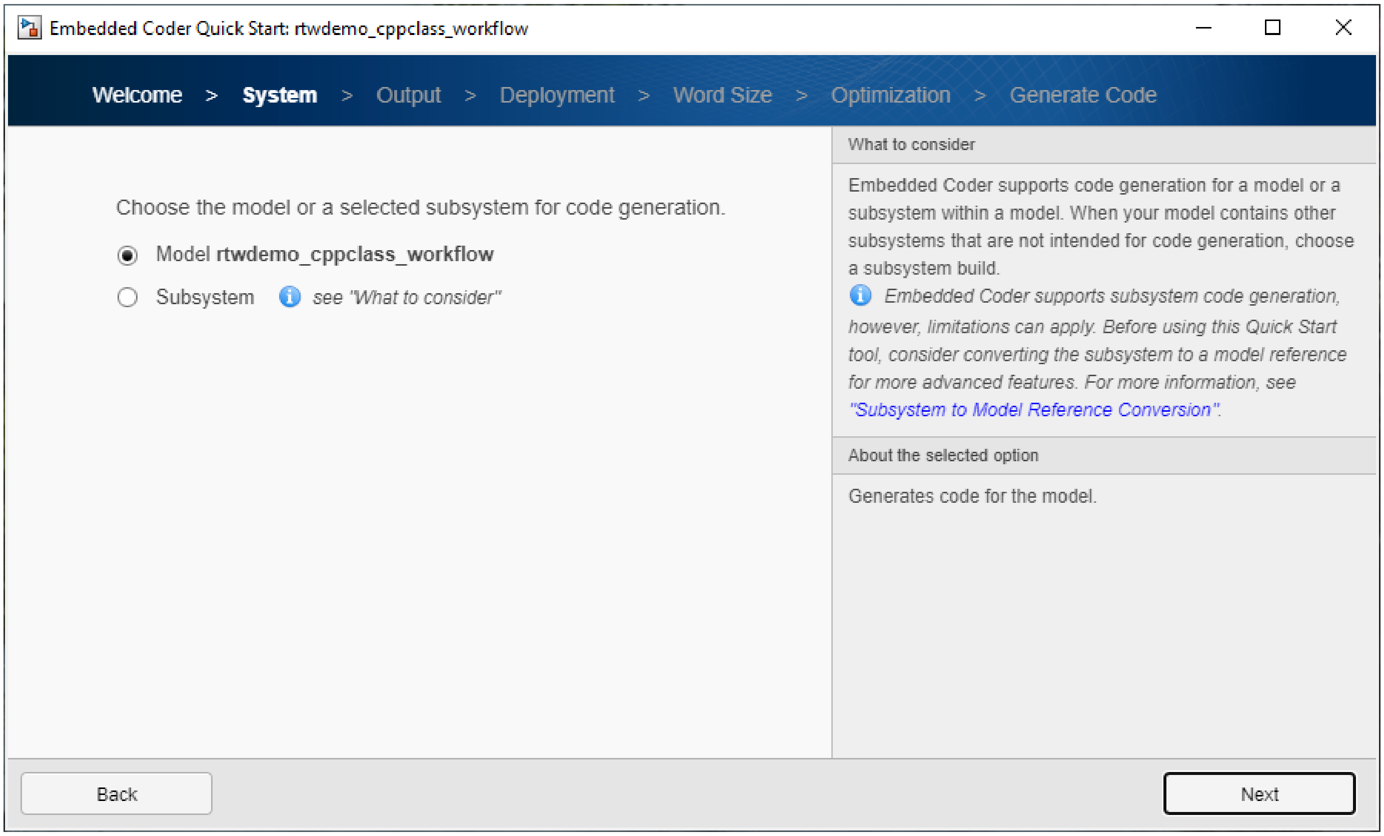
1 Open the model `CppClassWorkflowKeyIgnition`.

```
openExample('ecoder/CppCodeGenerationWorkflowForKeyIgnitionSystemExample', ...
            supportingFile='CppClassWorkflowKeyIgnition.slx')
```



2 Save a copy of the model to a writable location on the MATLAB search path.
3 If the **C++ Code** tab is not already open, on the **Apps** tab, in the Apps gallery, under **Code Generation**, click **Embedded Coder**.
4 On the **C++ Code** tab, click **Settings** and select **C/C++ Code generation settings**.

The Configuration Parameters dialog box opens.
5 On the Configuration Parameters dialog box, open the **Code Generation** pane, and under **Target selection**, verify the parameter **Language** is set to `C++`. Click **OK** to close the dialog box.
6 On the **C++ Code** tab, click **Quick Start**.


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

Choose the model or a selected subsystem for code generation.

● Model **rtwdemo_cppclass_workflow**

○ 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

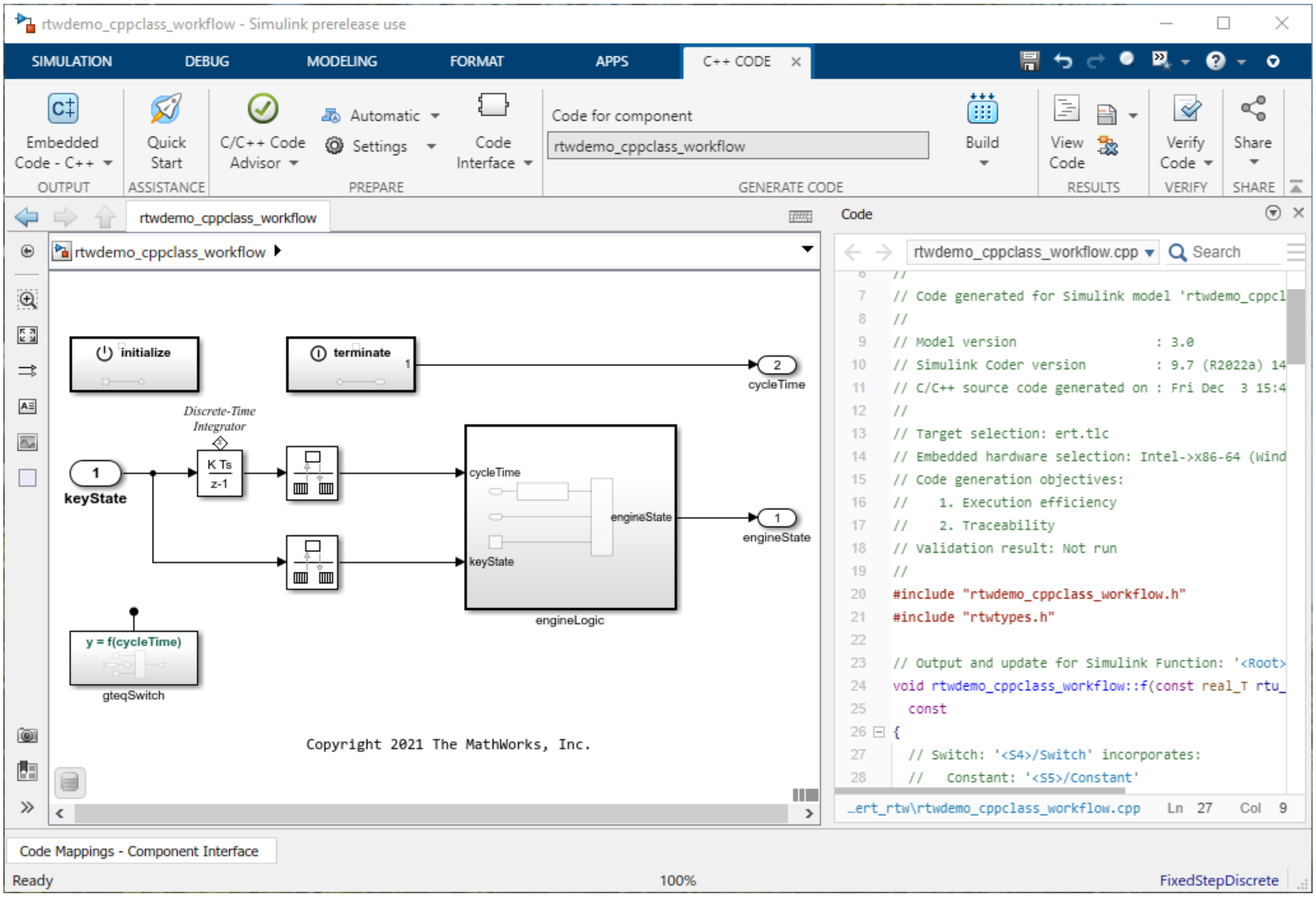
Generates code for the model.

7 Advance through the Quick Start tool by clicking **Next** in each step.

Each step asks questions about the code that you want to generate. For this tutorial, use the defaults. The tool validates your selections against the model and presents the parameter changes required to generate code.

8 In the **Generate Code** step, apply the proposed changes and generate code from `CppClassWorkflowKeyIgnition` by clicking **Next**.

9 Click **Finish**, then return to the **C++ Code** tab.



```
//
// Code generated for Simulink model 'rtwdemo_cppcl
//
// Model version                  : 3.0
// Simulink Coder version         : 9.7 (R2022a) 14
// C/C++ source code generated on : Fri Dec  3 15:4
//
// Target selection: ert.tlc
// Embedded hardware selection: Intel->x86-64 (Wind
// Code generation objectives:
//    1. Execution efficiency
//    2. Traceability
// Validation result: Not run
//
#include "rtwdemo_cppclass_workflow.h"
#include "rtwtypes.h"

// Output and update for Simulink Function: '<Root>
void rtwdemo_cppclass_workflow::f(const real_T rtu_
  const
{
  // Switch: '<S4>/Switch' incorporates:
  //   Constant: '<S5>/Constant'
```

Inspect Generated C++ Code

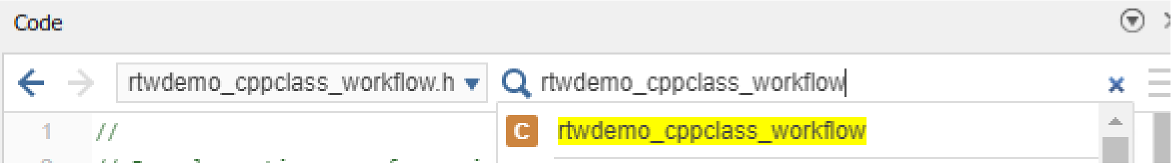
The code generator converts the `CppClassWorkflowKeyIgnition` model into a C++ class, which you access from your application code. The model data elements appear as class members. The Simulink functions appear as class methods.

The `CppClassWorkflowKeyIgnition` model incorporates elements of rate-based and export-function modeling. For rate-based modeling, the entry-point class methods, which you call from your application code, include an initialization method, an execution method, a terminate method, and, optionally, a reset method. To integrate with external code or interface requirements, you can customize the generated class interface.

For this tutorial, the code generated by using the Quick Start tool uses default settings for the class elements. The default name for the C++ class is the model name `CppClassWorkflowKeyIgnition`.

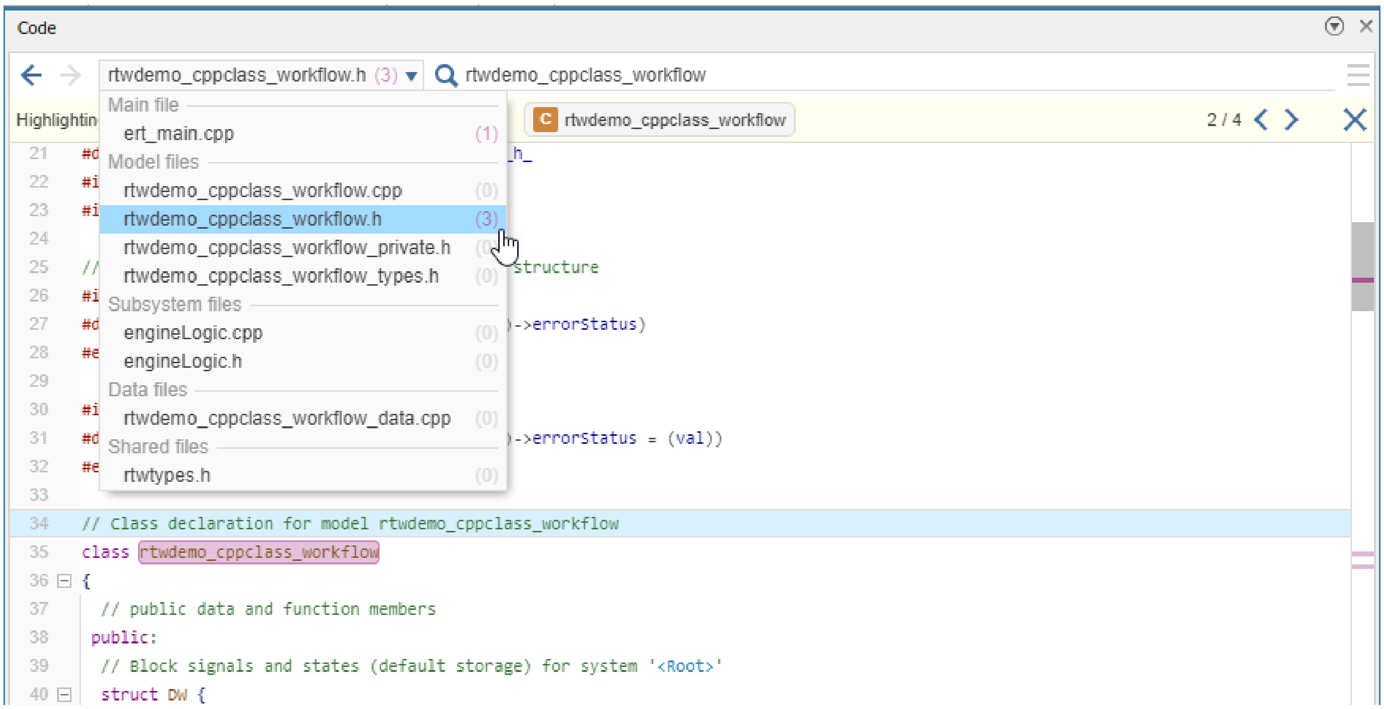
To inspect the C++ class information generated for the model:

1. On the right side of the Simulink Editor window, in the Code view pane, locate the search bar.
2. In the search bar, type the model class name `CppClassWorkflowKeyIgnition` to find each instance of the class name across the generated code, and then click the highlighted search suggestion.


Code
rtwdemo_cppclass_workflow.h
rtwdemo_cppclass_workflow
C
rtwdemo_cppclass_workflow
//

3 Use the arrows on the right under the search bar to step through each instance, including the class definition in `CppClassWorkflowKeyIgnition.h` and the class instantiation in `ert_main.cpp`.

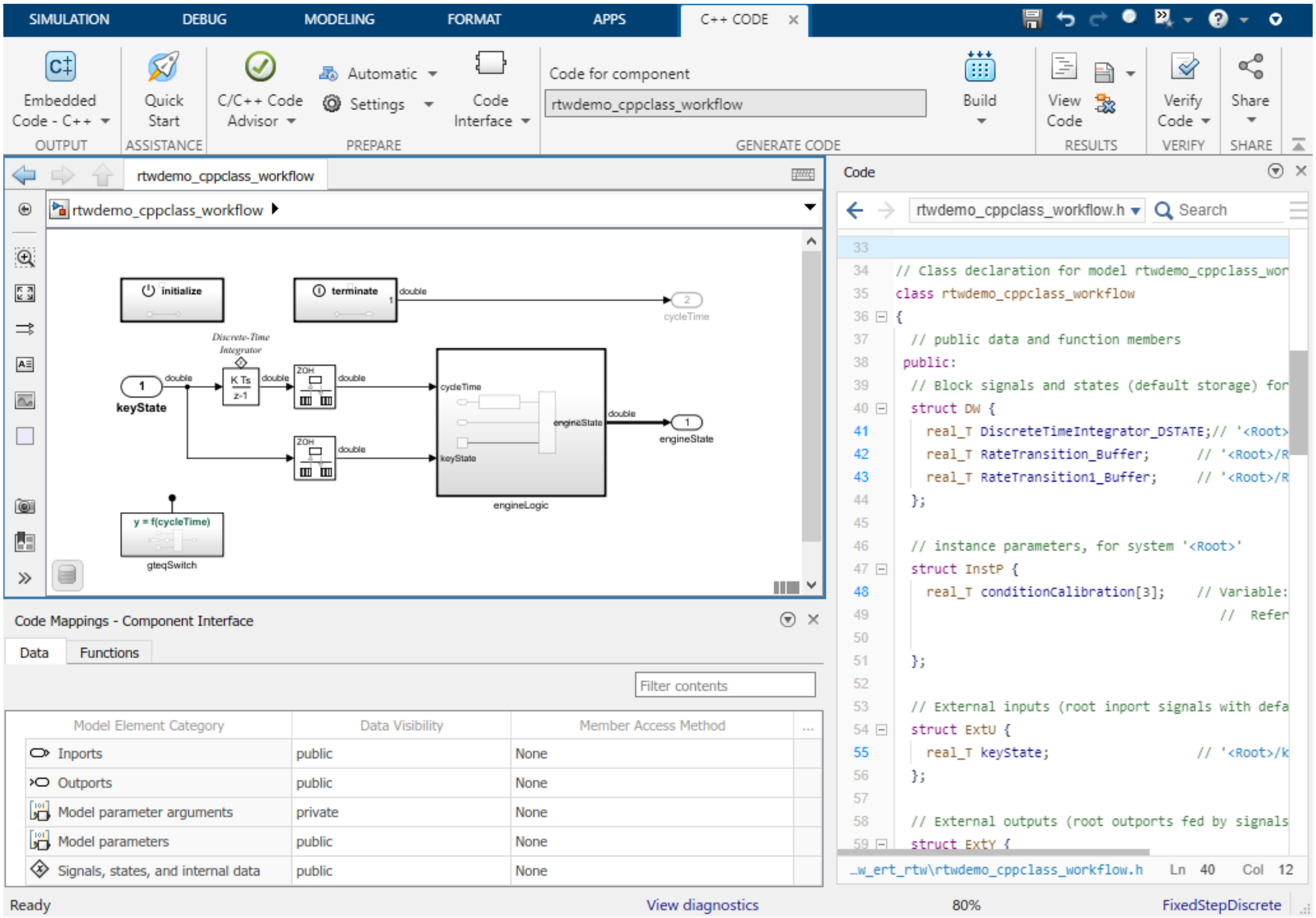
You can also see the number of search results in each file from the file menu in the upper-left corner.


Code
rtwdemo_cppclass_workflow.h (3)
rtwdemo_cppclass_workflow
Highlightin
C
rtwdemo_cppclass_workflow
2 / 4
Main file
ert_main.cpp (1)
Model files
rtwdemo_cppclass_workflow.cpp (0)
rtwdemo_cppclass_workflow.h (3)
rtwdemo_cppclass_workflow_private.h (0)
rtwdemo_cppclass_workflow_types.h (0)
Subsystem files
engineLogic.cpp (0)
engineLogic.h (0)
Data files
rtwdemo_cppclass_workflow_data.cpp (0)
Shared files
rtwtypes.h (0)

```
// Class declaration for model rtwdemo_cppclass_workflow
class rtwdemo_cppclass_workflow
{
  // public data and function members
 public:
  // Block signals and states (default storage) for system '<Root>'
  struct DW {
```

4 Review the data and function code mappings.

In the Simulink Editor window, click **Code Interface**, and select **Code Mappings** to open the Code Mappings editor.



Code for component: rtwdemo_cppclass_workflow

rtwdemo_cppclass_workflow.h

```

// Class declaration for model rtwdemo_cppclass_wor
class rtwdemo_cppclass_workflow
{
  // public data and function members
 public:
  // Block signals and states (default storage) for
  struct DW {
    real_T DiscreteTimeIntegrator_DSTATE;// '<Root>
    real_T RateTransition_Buffer;       // '<Root>/R
    real_T RateTransition1_Buffer;      // '<Root>/R
  };

  // instance parameters, for system '<Root>'
  struct InstP {
    real_T conditionCalibration[3];     // Variable:
                                        //  Refer


  };

  // External inputs (root inport signals with defa
  struct ExtU {
    real_T keyState;                    // '<Root>/k
  };

  // External outputs (root outports fed by signals
  struct ExtY {
```

..w_ert_rtw\rtwdemo_cppclass_workflow.h Ln 40 Col 12

Code Mappings - Component Interface

Data | Functions

Filter contents

Model Element Category	Data Visibility	Member Access Method
Inports	public	None
Outports	public	None
Model parameter arguments	private	None
Model parameters	public	None
Signals, states, and internal data	public	None

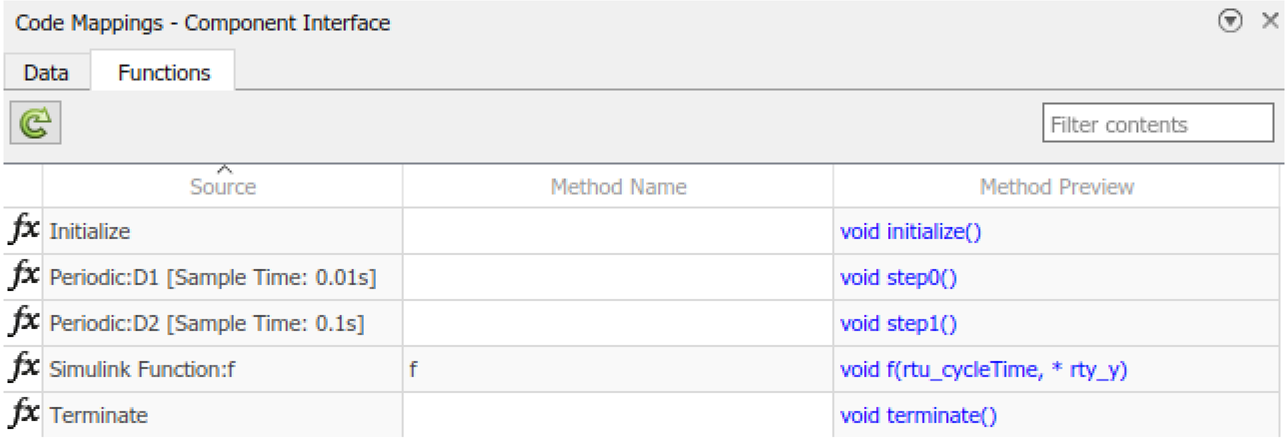
5 Click the **Data** tab to view class member visibility and access methods.

Simulink data elements are grouped into categories of modeling elements:

- **Inports**: Root-level data input ports of a model.
- **Outports**: Root-level data output ports of a model.
- **Model parameter arguments**: Workspace variables that appear as instance (nonstatic) class data members.
- **Model parameters**: Workspace variables that are shared across instances of the model class that are generated as static class data members.
- **Signals, states, and internal data**: Data elements that are internal to a model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

6 Click the **Functions** tab to view class methods.

The generated class methods are entry-point methods and are locations in code where a transfer of program control occurs.



	Source	Method Name	Method Preview
fx	Initialize		void initialize()
fx	Periodic:D1 [Sample Time: 0.01s]		void step0()
fx	Periodic:D2 [Sample Time: 0.1s]		void step1()
fx	Simulink Function:f	f	void f(rtu_cycleTime, * rty_y)
fx	Terminate		void terminate()

The default name for the initialize class method is `initialize`, the execution (step) method is `step0`, and the terminate method is `terminate`.

7. Repeat the search steps to locate class methods `initialize`, `step0`, and `terminate` in the generated code.

Next, configure a customized class interface for code generation and review the generated code.

Configure Class Interface

In this step of the tutorial, you configure the C++ class interface. A customized C++ class interface enables the generated classes to meet specific code standards or interface requirements so that the generated code can compile and integrate into larger architectures requiring minimal post-generation customization.

To customize the class interface of model `CppClassWorkflowKeyIgnition`:

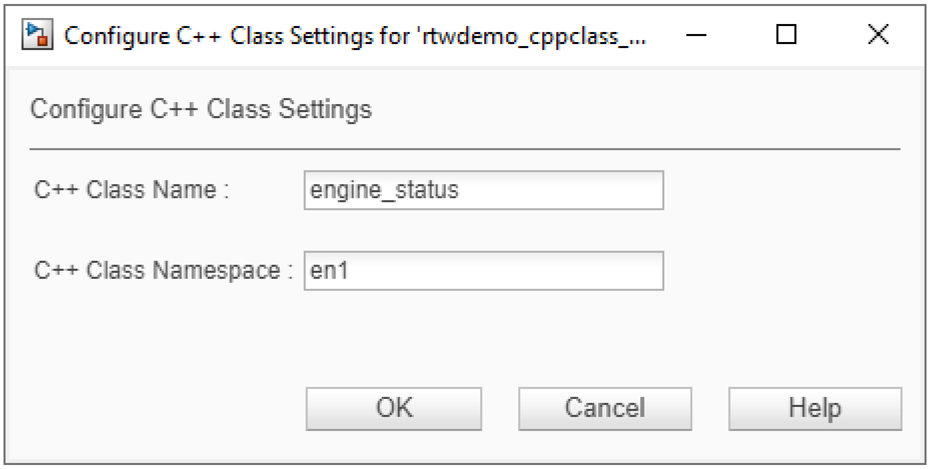
- Change the class name to `engine_status`.
- Set the class namespace to `en1`.
- Configure the inports of the model as public with a set method for access.
- Configure the outports of the model as public with an aggregate structure-based get method to access all output ports.
- Update the names of the initialize and terminate entry-point methods.

Configure Model Class Name and Namespace

1. If the model `CppClassWorkflowKeyIgnition` is not open, open the model.

```
openExample('ecoder/CppCodeGenerationWorkflowForKeyIgnitionSystemExample', ...
            supportingFile='CppClassWorkflowKeyIgnition.slx')
```

2. On the **C++ Code** tab, click **Code Interface** and select **Class Name & Namespace**.
3. Edit the **C++ Class Name** field to `engine_status`.
4. Enter `en1` in the **C++ Class Namespace** field.



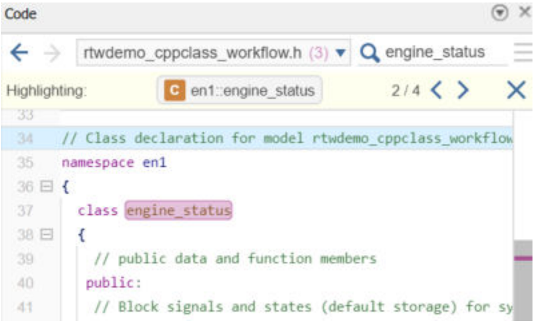
Configure C++ Class Settings for 'rtwdemo_cppclass_...

Configure C++ Class Settings

C++ Class Name : engine_status

C++ Class Namespace : en1

5. Click **OK**. Validation is performed interactively with field warnings that alert you if you enter an invalid name or namespace.
6. Save the model, and then click **Build** to build and regenerate the code.
7. In the Code view pane, search for `engine_status` to view the change in the generated code.



```
// Class declaration for model rtwdemo_cppclass_workflow
namespace en1
{
  class engine_status
  {
    // public data and function members
   public:
    // Block signals and states (default storage) for sy
```

Configure Visibility and Access of Class Members

1 Open the Code Mappings editor.

To open the Code Mappings editor, on the **C++ Code** tab, click **Code Interface** and select **Code Mappings**.

2 Click the **Data** tab to open the **Data** pane.

The Code Mappings spreadsheet displays visibility and access information for each category of model element.

3 Configure **Inports**.

- In the **Data Visibility** column, select `public`.
- In the **Member Access Method** column, select `Inlined method`.

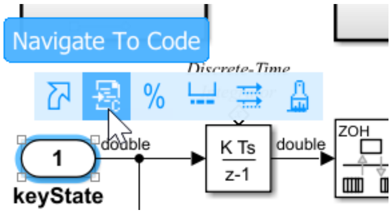
4 Configure **Outports**.

- In the **Data Visibility** column, select `public`.
- In the **Member Access Method** column, select `Structure-based method`.

5 Save the model, and click **Build** to build and regenerate the code.

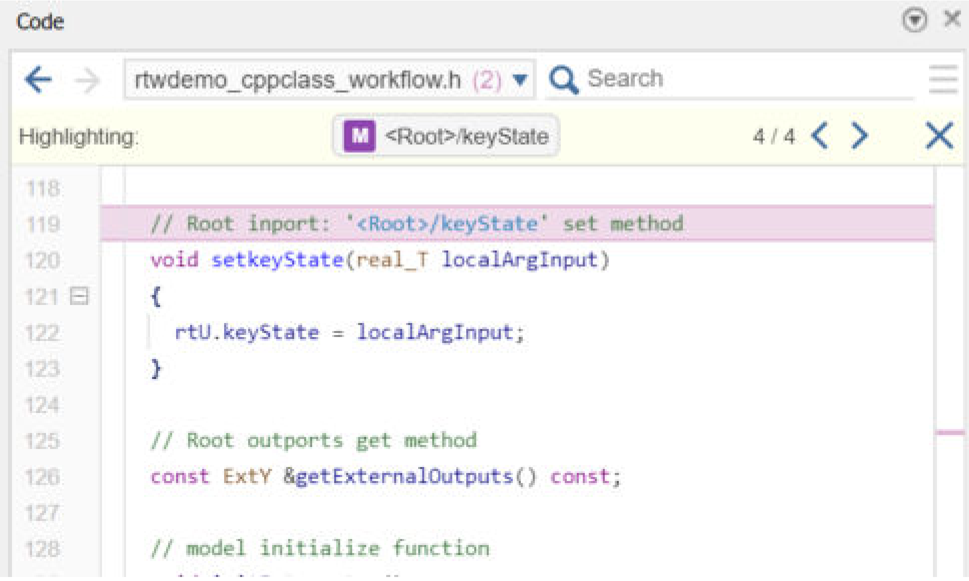
6 In the model `CppClassWorkflowKeyIgnition`, click the Inport block `keyState`.

7 Place your cursor over the ellipsis menu above the block and click **Navigate To Code**.



The Code view highlights code that corresponds to the block.

8 In the Code view, click the search arrows to locate the public declaration and definition of the set method `setkeyState` for the root inport `keyState`.

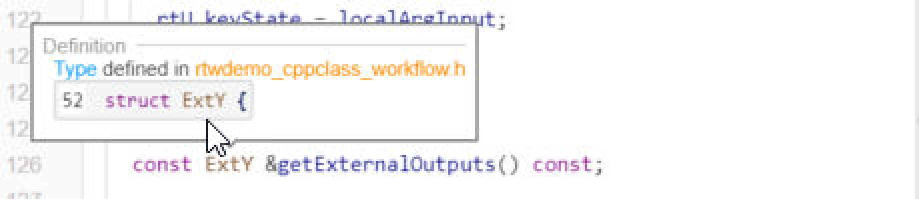


```
// Root inport: '<Root>/keyState' set method
void setkeyState(real_T localArgInput)
{
  rtU.keyState = localArgInput;
}

// Root outports get method
const ExtY &getExternalOutputs() const;

// model initialize function
```

The get method for the aggregated root outports, `getExternalOutputs`, is declared directly beneath the inport set method.

9 Place your cursor over the identifier `ExtY` in the `getExternalOutputs` method declaration.

A traceability dialog box displays definitions that correspond to the code.



```
rtU.keyState = localArgInput;
52  struct ExtY {
const ExtY &getExternalOutputs() const;
```

10 In the traceability dialog box, click `struct ExtY` to locate the structure.

```
// External outputs (root outports fed by signals with default storage)
struct ExtY {
  real_T engineState[3];           // '<Root>/engineState'
  real_T cycleTime;                // '<Root>/cycleTime'
};
```

11 In the search bar, type `getExternalOutputs` and press **Enter** to locate the method definition in `CppClassWorkflowKeyIgnition.cpp`.

Configure Model Functions

Configure the class method names. When you generate C++ code from a model, model entry-point functions appear as class methods in the generated code. To integrate with external code or interface requirements, you can customize the name of the generated methods.

1 In the Code Mappings editor, click the **Functions** tab to view the class methods.

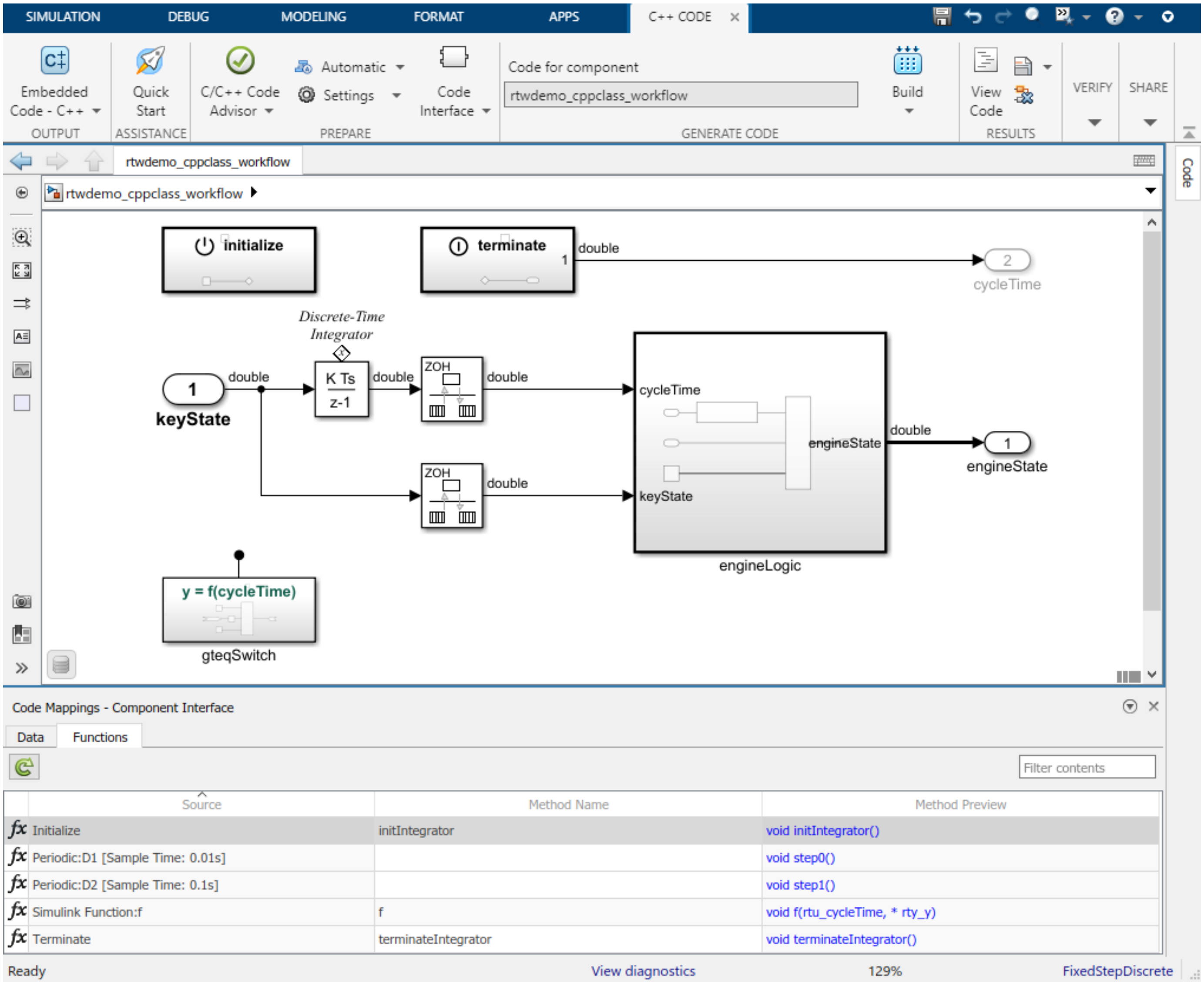
2 Configure the **Initialize** function name.

In the **Method Name** column, click and edit the spreadsheet to change the name to `initIntegrator`.

3 Configure the **Terminate** function name.

In the **Method Name** column, click and edit the spreadsheet to change the name to `terminateIntegrator`.

4 Verify the updated names in the **Method Preview** column.



Code Mappings - Component Interface

Data Functions

Filter contents

Source	Method Name	Method Preview
Initialize	initIntegrator	void initIntegrator()
Periodic:D1 [Sample Time: 0.01s]		void step0()
Periodic:D2 [Sample Time: 0.1s]		void step1()
Simulink Function:f	f	void f(rtu_cycleTime, * rty_y)
Terminate	terminateIntegrator	void terminateIntegrator()

5 Save the model, and then click **Build** to build and regenerate code.

6 In the Code view, search for the updated method names for the entry-point functions to view the generated code.

Next, deploy the C++ generated code.

Deploy the C++ Generated Code

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

Example Main Program

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

Explore the example main program generated for model CppClassWorkflowKeyIgnition.

1. If not already open, open your copy of the model CppClassWorkflowKeyIgnition.
2. In the Apps gallery, click **Embedded Coder**.
3. Regenerate the code.
4. In the Code view, select file ert_main.cpp.
5. Click in the **Search** field and select function rt_OneStep.
6. Explore the incomplete wrapper function rt_OneStep. This function calls the model execution entry-point function, step0. Your application code can call rt_OneStep to run the model algorithm during each execution cycle.
7. Click in the **Search** field and select function main.
8. Explore the incomplete example main function. This function outlines the order and context in which your application code can call rt_OneStep and other model entry-point functions.

Overview of Generated Code Files

The code generator creates two folders in your MATLAB current folder for the generated code files:

- slprj
- CppClassWorkflowKeyIgnition_ert_rtw

The slprj/ert/_sharedutils folder contains generated files that are shared between multiple models. This folder contains the file rtwtypes.h, which defines standard data types that the generated code uses by default.

The CppClassWorkflowKeyIgnition_ert_rtw folder contains the model-specific files of your generated code, including the two primary files CppClassWorkflowKeyIgnition.cpp and CppClassWorkflowKeyIgnition.h.

Relocate Generated Code Files

Embedded Coder provides a packNGo utility for relocating static and generated code files for a model to another development environment. The utility packages the files in a compressed file that you can relocate and unpack by using a standard zip utility. You can apply the packNGo utility from graphical and programming interfaces.

For more information, see “Relocate or Share Generated Code”.

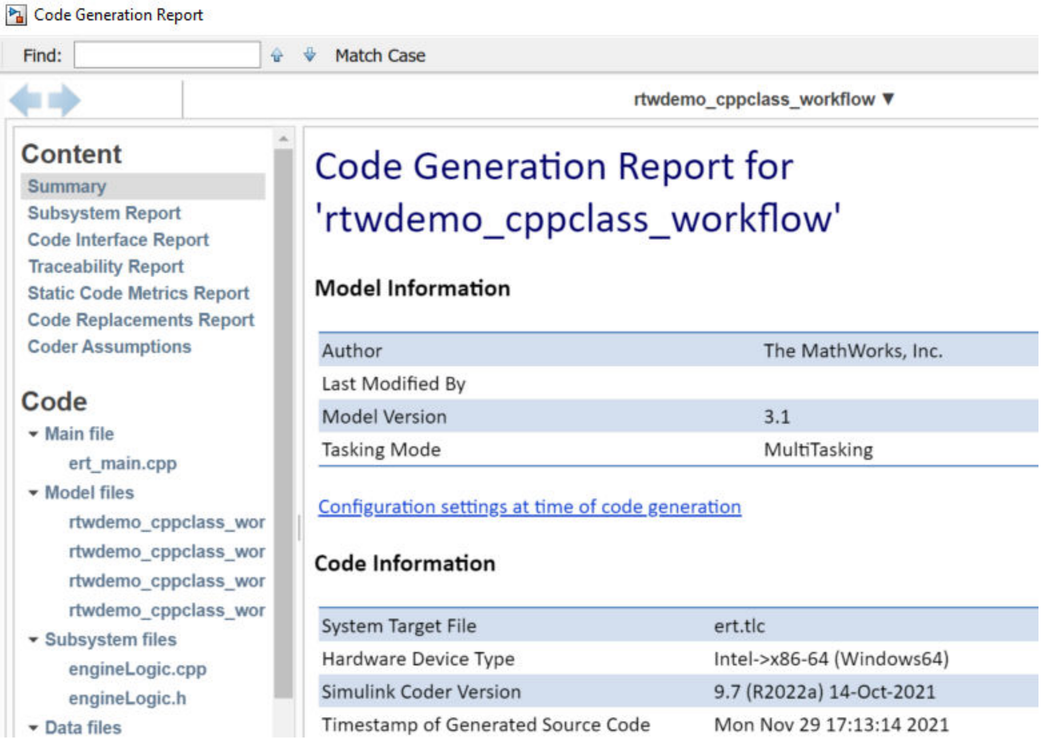
Share and Archive a Code Generation Report

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

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

You can use this report outside of the Simulink environment, so it is suitable for sharing or for archival purposes. You can open the report from the tool or, on the **C++ Code** tab, click **Open Report**.

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



Code Generation Report for 'rtwdemo_cppclass_workflow'

Model Information

Author	The MathWorks, Inc.
Last Modified By	
Model Version	3.1
Tasking Mode	MultiTasking

Configuration settings at time of code generation

Code Information

System Target File	ert.tlc
Hardware Device Type	Intel->x86-64 (Windows64)
Simulink Coder Version	9.7 (R2022a) 14-Oct-2021
Timestamp of Generated Source Code	Mon Nov 29 17:13:14 2021

Explore Other Options

To explore more ways to customize, verify, and deploy generated production code, use the links listed in the table.

Task	Reference
Consider model design and configuration for code generation	“Architecture and Component Design”
Learn about generated entry-point functions	“Configure Generated C Function Interface for Model Entry-Point Functions”
Learn about configuring C++ code interfaces for code generation	“Interactively Configure C++ Interface”
Achieve code reuse	“Choose a Componentization Technique for Code Reuse”
Compare normal mode simulation results against software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation results for numeric equivalency	“SIL and PIL Simulations” and “Choose a SIL or PIL Approach”
Collect code coverage metrics for generated code during SIL or PIL simulation	“Code Coverage”
Use generated example main code as a starting point to deploy generated executable programs	“Deploy Applications to Target Hardware”

See Also

Related Examples

- “Interactively Configure C++ Interface”
- “Programmatically Configure C++ Interface”
- “Configure C++ Class Interface for Rate-Based Models”
- “Configure C++ Class Interface for Export-Function Models”

Get Started with Embedded Coder

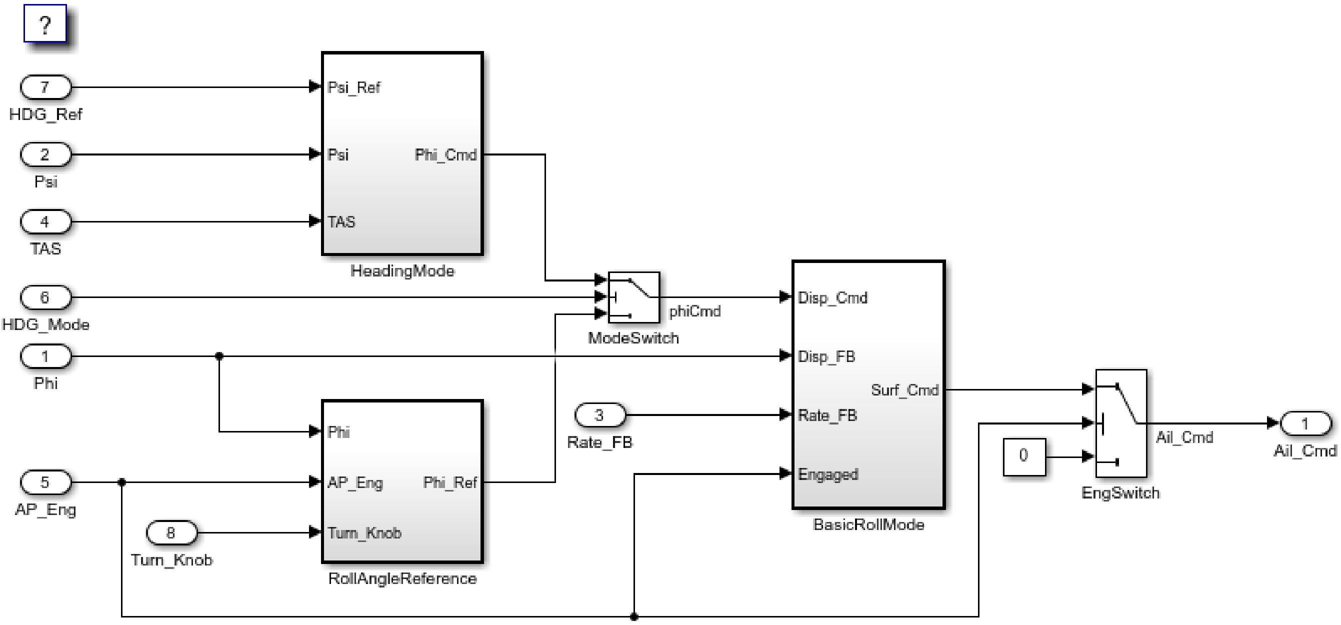
This model shows an implementation of a roll axis autopilot control system, that is designed for code generation.

About the Model

This model represents a basic roll axis autopilot with two operating modes: roll attitude hold and heading hold. The mode logic for these modes is external to this model. The model architecture represents the heading hold mode and basic roll attitude function as atomic subsystems.

The roll attitude control function is a PID controller that uses roll attitude and roll rate feedback to produce an aileron command. The input to the controller is either a basic roll angle reference or a roll command to track the desired heading. The model is as follows:

```
open_system('RollAxisAutopilot');
```

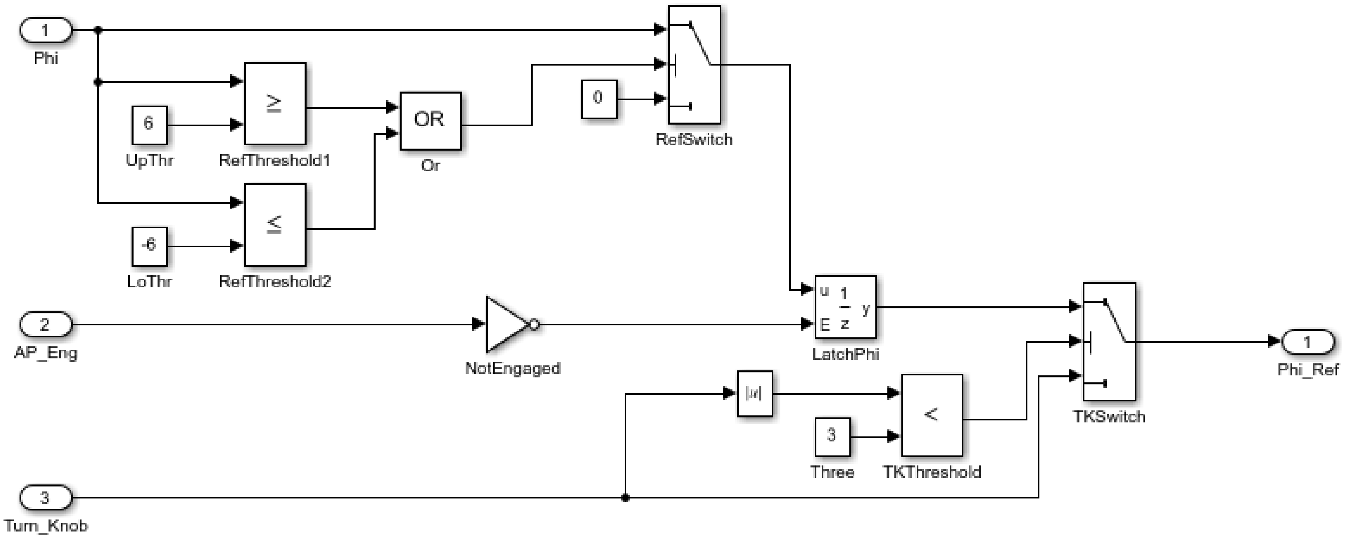


Copyright 1990-2020 The MathWorks, Inc.

Subsystem `RollAngleReference`

The basic roll angle reference calculation is implemented as the subsystem `RollAngleReference`. Embedded Coder® inlines this calculation directly into the main function for `RollAxisAutopilot`.

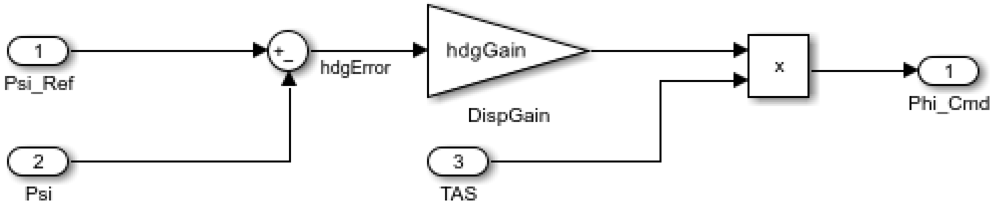
```
open_system('RollAxisAutopilot/RollAngleReference');
```



Subsystem HeadingMode

The subsystem HeadingMode computes the roll command to track the desired heading.

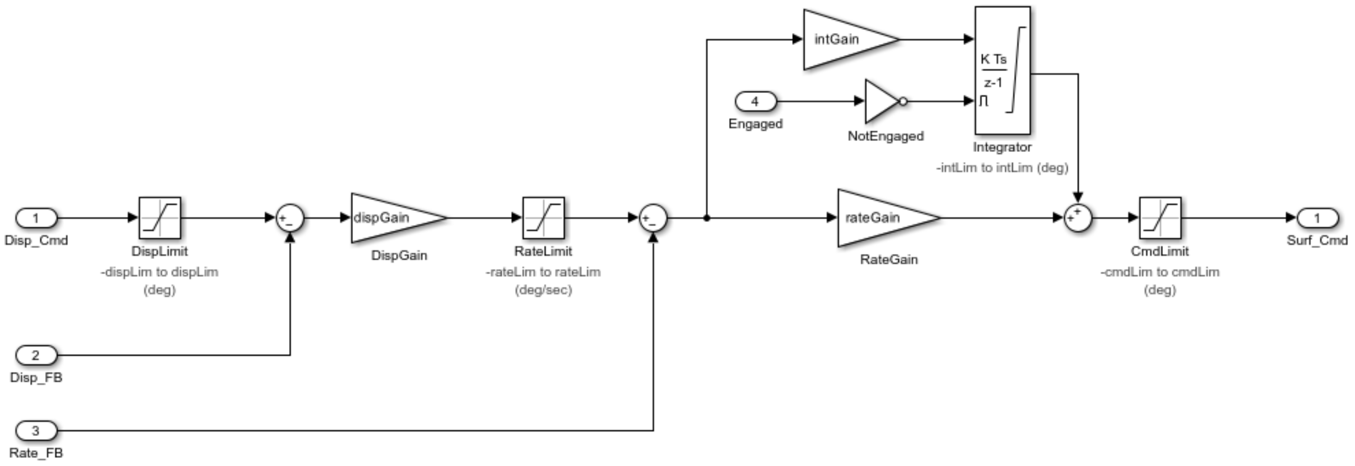
```
close_system('RollAxisAutopilot/RollAngleReference');
open_system('RollAxisAutopilot/HeadingMode');
```



Subsystem BasicRollMode

The subsystem BasicRollMode computes the roll attitude control function (PID).

```
close_system('RollAxisAutopilot/HeadingMode');
open_system('RollAxisAutopilot/BasicRollMode');
```



Generate Code for the Model

The model is preconfigured to generate code using Embedded Coder. To generate code using Simulink® Coder™ only, reconfigure the model or at the command prompt type cs = getActiveConfigSet('RollAxisAutopilot'); switchTarget(cs,'grt.tlc',[]);

Generate code.

```
slbuild('RollAxisAutopilot');
```

```
### Starting build procedure for: RollAxisAutopilot
### Successful completion of build procedure for: RollAxisAutopilot

Build Summary

Top model targets built:

Model              Action                        Rebuild Reason
================================================================================================
RollAxisAutopilot  Code generated and compiled.  Code generation information file does not exist

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 12.891s
```

You can view the entire generated code in a detailed HTML report, with bi-directional traceability between model and code.

```
currentDir = pwd;
web(fullfile(currentDir,'RollAxisAutopilot_ert_rtw','html','index.html'))
```

Embedded Coder Get Started Tutorials

For more information on generating code with Embedded Coder, see “Generate C Code from Simulink Models” on page 3-2 in the *Get Started with Embedded Coder* documentation.