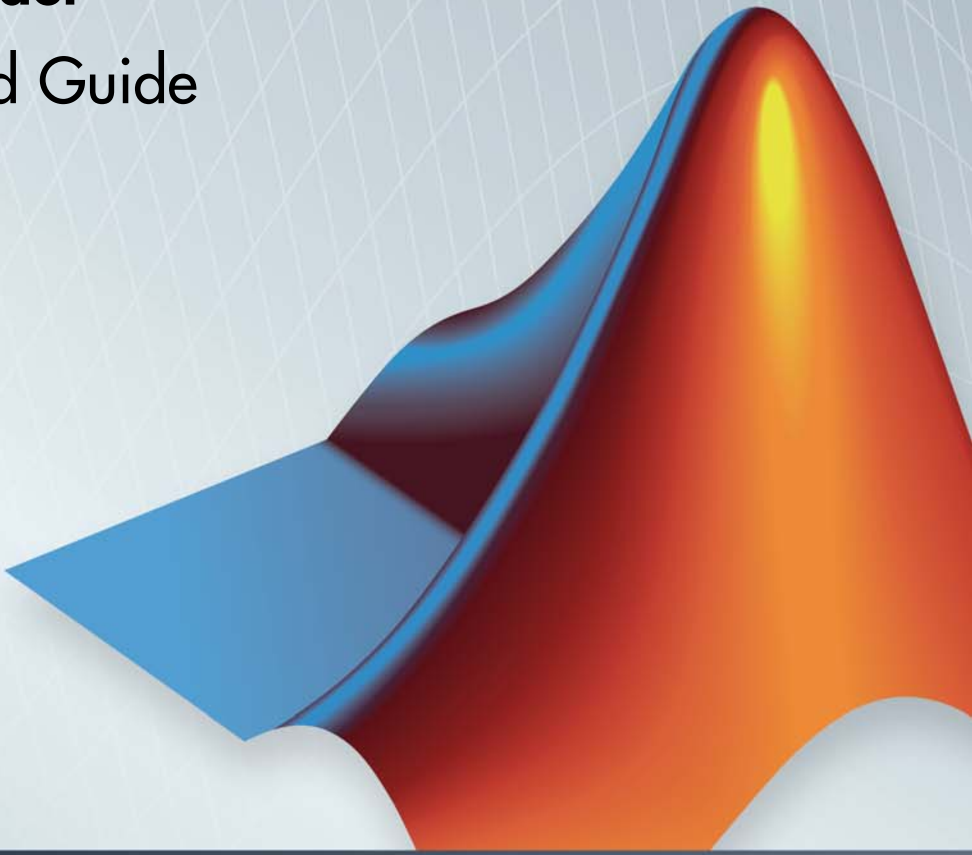
# Embedded Coder®

## Getting Started Guide

**R**2014**a**



MATLAB® & SIMULINK®

MathWorks®

**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) |

## Check Bug Reports for Issues and Fixes

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

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

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

# Contents

## A Installing and Using IDE

# Product Overview

## Embedded Coder Product Description

### Generate C and C++ code optimized for embedded systems

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

Embedded Coder offers built-in support for AUTOSAR and ASAP2 software standards. It also provides traceability reports, code interface documentation, and automated software verification. Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178).

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

### Key Features

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

- Standards support, including ASAP2, AUTOSAR, DO-178, IEC 61508, ISO 26262, and MISRA-C

## Code Generation Technology

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

Code generation technology and related products provide tooling that you can apply to the V-model for system development. The V-model is a representation of system development that highlights verification and validation steps in the development process. For more information, see "Validation and Verification for System Development" on page 1-9.

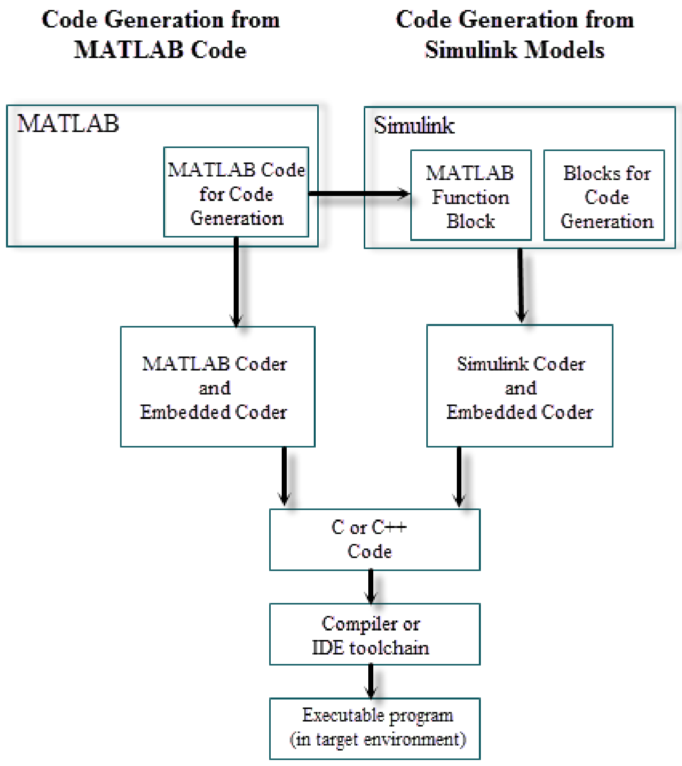
To learn model design patterns that include Simulink blocks, Stateflow® charts, and MATLAB functions, and map to commonly used C constructs, see "Modeling Patterns for C Code" in the Embedded Coder documentation.

## Code Generation Workflows with Embedded Coder

The Embedded Coder product *extends* the MATLAB Coder and Simulink Coder products with key features that you can use for embedded software development. Using the Embedded Coder product, you can generate code that has the clarity and efficiency of professional handwritten code. For example, you can:

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

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


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

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

To develop algorithms with MATLAB code for code generation, see "Code Generation from MATLAB Code" on page 1-6.

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 1-7.

## Code Generation from MATLAB Code

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

- MATLAB
- MATLAB Coder
- Embedded Coder

MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. To generate C or C++ code, you can use MATLAB Coder projects or enter the function `codegen` in the MATLAB Command Window. Embedded Coder provides additional options and advanced optimizations for fine-grain control of the generated code's functions, files, and data.

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

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

## Code Generation from Simulink Models

The code generation from a 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 production-quality C or C++ code from a Simulink model, Embedded Coder provides additional features for implementing, configuring, and verifying your model for code generation.

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

MATLAB code in the MATLAB Function block is also generated into C or C++ code and included in the generated source code.

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

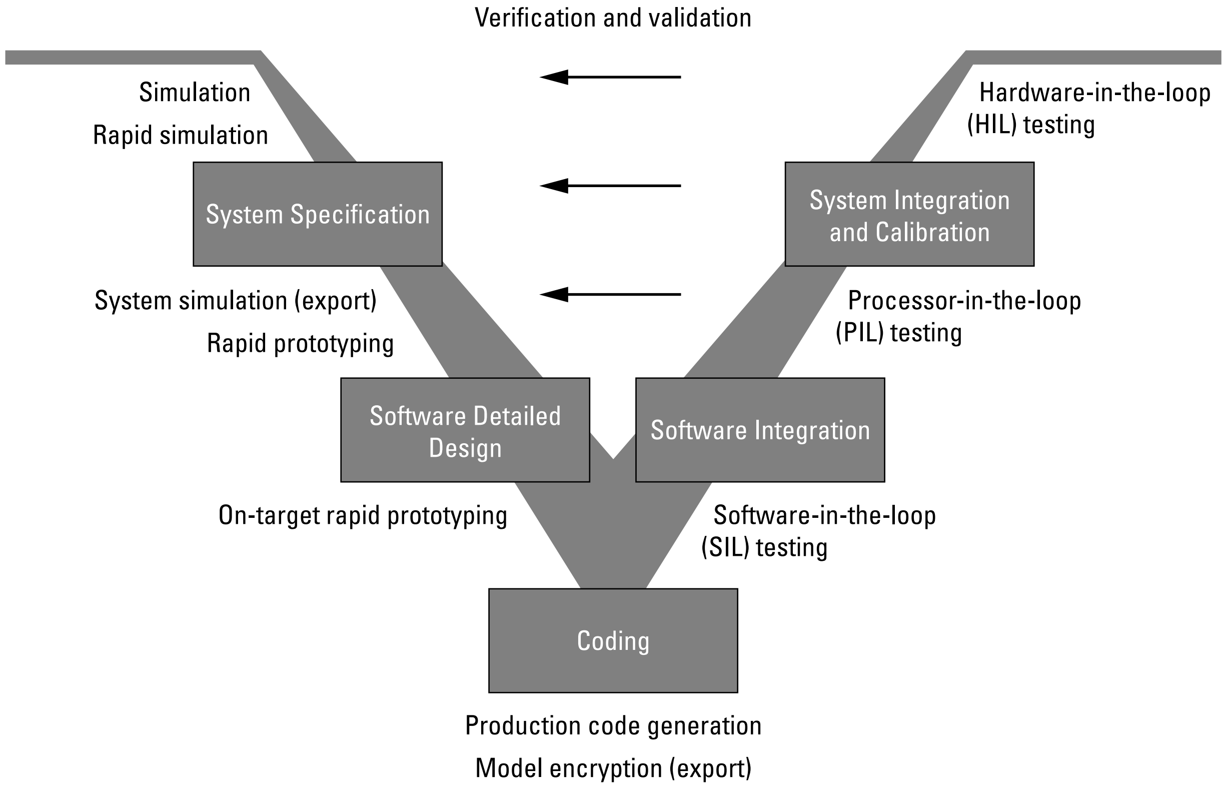
To learn how to model and generate code for commonly used C constructs using Simulink blocks, Stateflow charts, and MATLAB functions, see “Modeling Patterns for C Code”.

## Validation and Verification for System Development

**In this section...**

### V-Model for System Development

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


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

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

- "Types of Simulation and Prototyping in the V-Model" on page 1-11
- "Types of In-the-Loop Testing in the V-Model" on page 1-12
- "Mapping of Code Generation Goals to the V-Model" on page 1-13

## Types of Simulation and Prototyping in the V-Model

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

| | **Host-Based Simulation** | **Standalone Rapid Simulations** | **Rapid Prototyping** | **On-Target Rapid Prototyping** |
|---|---|---|---|---|
| **Purpose** | Test and validate functionality of concept model | Refine, test, and validate functionality of concept model in nonreal time | Test new ideas and research | Refine and calibrate designs during development process |
| **Execution hardware** | Host computer | Host computer<br><br>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)<br><br>Normal mode simulation in Simulink enables you to access, display, and tune data during verification | Easy to simulate models of hybrid dynamic systems that include components and environment models<br><br>Ideal for batch or Monte Carlo simulations<br><br>Can repeat simulations with | Might require custom real-time simulators and hardware<br><br>Might be done with inexpensive off-the-shelf PC hardware and I/O cards | Might use existing hardware, thus less expensive and more convenient |

| | Host-Based Simulation | Standalone Rapid Simulations | Rapid Prototyping | On-Target Rapid Prototyping |
|---|---|---|---|---|
| | Can accelerate Simulink simulations with Accelerated and Rapid Accelerated modes | varying data sets, interactively or programmatically with scripts, without rebuilding the model<br><br>Can connect to Simulink to monitor signals and tune parameters | | |

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

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

| | SIL Testing | PIL Testing on Embedded Hardware | PIL Testing on Instruction Set Simulator | HIL Testing |
|---|---|---|---|---|
| **Purpose** | Verify component source code | Verify component object code | Verify component object code | Verify system functionality |
| **Fidelity and accuracy** | Two options:<br><br>Same source code as target, but might have numerical differences<br><br>Changes source code to emulate word sizes, but is bit accurate for fixed-point math | Same object code<br><br>Bit accurate for fixed-point math<br><br>Cycle accurate because code runs on hardware | Same object code<br><br>Bit accurate for fixed-point math<br><br>Might not be cycle accurate | Same executable code<br><br>Bit accurate for fixed-point math<br><br>Cycle accurate<br><br>Use real and emulated system I/O |

| | SIL Testing | PIL Testing on Embedded Hardware | PIL Testing on Instruction Set Simulator | HIL Testing |
|---|---|---|---|---|
| **Execution platforms** | Host | Target | Host | Target |
| **Ease of use and cost** | Desktop convenience<br>Executes only in Simulink<br>Reduced hardware cost | Executes on desk or test bench<br>Uses hardware — process board and cables | Desktop convenience<br>Executes only on host computer with Simulink and integrated development environment (IDE)<br>Reduced hardware cost | Executes on test bench or in lab<br>Uses hardware — processor, embedded computer unit (ECU), I/O devices, and cables |
| **Real-time capability** | Not real time | Not real time (between samples) | Not real time (between samples) | Hard real time |

## Mapping of Code Generation Goals to the V-Model

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

- Documenting and Validating Requirements on page 1-14
- Developing a Model Executable Specification on page 1-16
- Developing a Detailed Software Design on page 1-19
- Generating the Application Code on page 1-23
- Integrating and Verifying Software on page 1-26
- Integrating, Verifying, and Calibrating System Components on page 1-29

### Documenting and Validating Requirements

| Goals | Related Product Information | Examples |
|---|---|---|
| Capture requirements in a document, spreadsheet, data base, or requirements management tool | "Simulink Report Generator™"<br>Third-party vendor tools such as Microsoft® Word, Microsoft Excel®, raw HTML, or IBM® Rational® DOORS® | |
| Associate requirements documents with objects in concept models<br>Generate a report on requirements associated with a model | "Requirements Traceability" — Simulink Verification and Validation™<br>Bidirectional tracing in Microsoft Word, Microsoft Excel, HTML, and IBM Rational DOORS | `slvnvdemo_fuelsys_docreq` |
| Include requirements links in generated code | "Review of Requirements Links" — Simulink Verification and Validation | `rtwdemo_requirements` |
| Trace model blocks and subsystems to generated code and vice versa | "Code Tracing" — Embedded Coder | `rtwdemo_hyperlinks` |
| Verify, refine, and test concept model in non real time on a host system | "Modeling" — Simulink Coder<br>"Modeling" — Embedded Coder<br>"Simulation" — Simulink<br>"Acceleration" — Simulink | `rtwdemo_fuelsys_publish` |

**Documenting and Validating Requirements (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Run standalone rapid simulations<br><br>Run batch or Monte-Carlo simulations<br><br>Repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model<br><br>Tune parameters and monitor signals interactively<br><br>Simulate models for hybrid dynamic systems that include components and an environment or plant that requires variable-step solvers and zero-crossing detection | "Rapid Simulation" — Simulink Coder<br><br>"Host/Target Communication" — Simulink Coder | `rtwdemo_rsim_param_survey_-script`<br>`rtwdemo_rsim_batch_script`<br>`rtwdemo_rsim_param_tuning` |
| Distribute simulation runs across multiple computers | "SystemTest™"<br><br>"MATLAB Distributed Computing Server™"<br><br>"Parallel Computing Toolbox™" | |

### Developing a Model Executable Specification

| Goals | Related Product Information | Examples |
|---|---|---|
| Produce design artifacts for algorithms that you develop in MATLAB code for reviews and archiving | " MATLAB Report Generator" | |
| Produce design artifacts from Simulink and Stateflow models for reviews and archiving | "System Design Description" — Simulink Report Generator | `rtwdemo_codegenrpt` |
| Add one or more components to another environment for system simulation<br><br>Refine a component model<br><br>Refine an integrated system model<br><br>Verify functionality of a model in nonreal time<br><br>Test a concept model | "Real-Time System Rapid Prototyping" | |
| Schedule generated code | "Scheduling" — Simulink Coder<br><br>"Handle Asynchronous Events" — Simulink Coder | `rtwdemos`, select **Multirate Support** folder |
| Specify function boundaries of systems | "Subsystems" — Simulink Coder | `rtwdemo_atomic`<br>`rtwdemo_ssreuse`<br>`rtwdemo_filepart`<br>`rtwdemo_export_functions` |
| Specify components and boundaries for design and incremental code generation | "Component-Based Modeling" — Simulink Coder<br><br>"Component-Based Modeling" — Embedded Coder | `rtwdemo_mdlreftop` |

**Developing a Model Executable Specification (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Specify function interfaces so that external software can compile, build, and invoke the generated code | "Function and Class Interfaces" — Simulink Coder<br><br>"Function and Class Interfaces" — Embedded Coder | `rtwdemo_fcnprotoctrl`<br>`rtwdemo_cppclass` |
| Manage data packaging in generated code for integrating and packaging data | "File Packaging" — Simulink Coder<br><br>"File Packaging " — Embedded Coder<br><br>"Program Builds" — Simulink Coder | `rtwdemos`, select **Function, File and Data Packaging** folder |
| Generate and control the format of comments and identifiers in generated code | "Add Custom Comments to Generated Code" — Embedded Coder<br><br>"Customize Generated Identifier Naming Rules" — Embedded Coder | `rtwdemo_comments`<br>`rtwdemo_symbols` |
| Create a zip file that contains generated code files, static files, and dependent data to build generated code in an environment other than your host computer | "Relocate Code to Another Development Environment"— Simulink Coder | `rtwdemo_buildinfo` |
| Export models for validation in a system simulator using shared libraries | "Shared Object Libraries" — Embedded Coder | `rtwdemo_shrlib` |

**Developing a Model Executable Specification (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Refine component and environment model designs by rapidly iterating between algorithm design and prototyping<br><br>Verify whether a component can adequately control a physical system in non-real time<br><br>Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design<br><br>Test hardware | "Deployment" — Simulink Coder<br><br>"Deployment" —Embedded Coder | `rtwdemo_profile` |
| Generate code for rapid prototyping | "Function and Class Interfaces" — Simulink Coder<br><br>"Entry Point Functions and Scheduling" — Embedded Coder<br><br>"Atomic Subsystem Code" — Embedded Coder | `rtwdemo_counter`<br>`rtwdemo_async` |
| Generate code for rapid prototyping in hard real time, using PCs | "Simulink Real-Time™" | `doc xpcdemos` |
| Generate code for rapid prototyping in soft real time, using PCs | "Real-Time Windows Target™" | `rtvdp` (and others) |

### Developing a Detailed Software Design

| Goals | Related Product Information | Examples |
|---|---|---|
| Refine a model design for representation and storage of data in generated code | "Data Representation" — Simulink Coder<br>"Data Representation " — Embedded Coder | |
| Select a deployment code format | "Target" — Simulink Coder<br>"Target"— Embedded Coder<br>"Sharing Utility Code" — Embedded Coder<br>"AUTOSAR Code Generation" — Embedded Coder | `rtwdemo_counter`<br>`rtwdemo_async`<br>"AUTOSAR Examples" in the Embedded Coder documentation |
| Specify target hardware settings | "Target" — Simulink Coder<br>"Target"— Embedded Coder | `rtwdemo_targetsettings` |
| Design model variants | "Variant Systems" — Simulink<br>"Variant Systems" — Embedded Coder | |
| Specify fixed-point algorithms in Simulink, Stateflow, and the MATLAB language subset for code generation | "Data Types and Scaling" — Fixed-Point Designer<br>"Fixed-Point Code Generation" — Fixed-Point Designer | `rtwdemo_fixpt1`<br>`rtwdemo_fuelsys_fxp_publish` |
| Convert a floating-point model or subsystem to a fixed-point representation | "Conversion Using Simulation Data" — Fixed-Point Designer<br>"Conversion Using Range Analysis" — Fixed-Point Designer | `fxpdemo_fpa` |
| Iterate to obtain an optimal fixed-point design, using autoscaling | "Data Types and Scaling" — Fixed-Point Designer | `fxpdemo_feedback` |

**Developing a Detailed Software Design (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Create or rename data types specifically for your application | "User-Defined Data Types" — Embedded Coder<br>"Data Type Replacement" — Embedded Coder | `rtwdemo_udt` |
| Control the format of identifiers in generated code | "Customize Generated Identifier Naming Rules" — Embedded Coder | `rtwdemo_symbols` |
| Specify how signals, tunable parameters, block states, and data objects are declared, stored, and represented in generated code | "Custom Storage Classes" — Embedded Coder | `rtwdemo_cscpredef` |
| Create a data dictionary for a model | "Data Definition and Declaration Management" — Embedded Coder | `rtwdemo_advsc` |
| Relocate data segments for generated functions and data using `#pragmas` for calibration or data access | "Memory Sections" — Embedded Coder | `rtwdemo_memsec` |
| Assess and adjust model configuration parameters based on the application and an expected run-time environment | "Configuration" — Simulink Coder<br>"Configuration" — Embedded Coder | `rtwdemo_usingrtw_script`<br>`rtwdemo_usingrtwec_script` |
| Check a model against basic modeling guidelines | "Verify Model Syntax" — Simulink | `rtwdemo_advisor1` |
| Add custom checks to the Simulink Model Advisor | "Customization and Automation" | `slvnvdemo_mdladv` |
| Check a model against custom standards or guidelines | "Consult the Model Advisor" — Simulink | |

**Developing a Detailed Software Design (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Check a model against industry standards and guidelines (MathWorks Automotive Advisory Board (MAAB), IEC 61508, and DO-178B) | "Standards and Guidelines" — Embedded Coder<br>"Model Guidelines Compliance" — Simulink Verification and Validation | `rtwdemo_iec61508` |
| Obtain model coverage for structural coverage analysis such as MC/DC | "Model Coverage Analysis" — Simulink Design Verifier™ | `cvbasic_operation` |
| Prove properties and generate test vectors for models | Simulink Design Verifier | `sldvdemo_cruise_control`<br>`sldvdemo_cruise_control_-verification` |
| Generate reports of models and software designs | " MATLAB Report Generator" — MATLAB Report Generator<br>"Simulink Report Generator" — Simulink Report Generator<br>"System Design Description" — Simulink Report Generator | `rtwdemo_codegenrpt` |
| Conduct reviews of your model and software designs with coworkers, customers, and suppliers who do not have Simulink available | "Web Display of Model Information" — Simulink Report Generator<br>"Model Comparison" — Simulink Report Generator | `slxml_sfcar` |

**Developing a Detailed Software Design (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Refine the concept model of your component or system<br>Test and validate the model functionality in real time<br>Test the hardware<br>Obtain real-time profiles and code metrics for analysis and sizing based on your embedded processor<br>Assess the feasibility of the algorithm based on integration with the environment or plant hardware | "Deployment" — Simulink Coder<br>"Deployment" — Embedded Coder<br>"Code Execution Profiling" — Embedded Coder<br>"Static Code Metrics" — Embedded Coder | `rtwdemos`, select **Desktop IDEs**, **Desktop Targets**, **Embedded IDEs**, or **Embedded Targets** |
| Generate source code for your models, integrate the code into your production build environment, and run it on existing hardware | "Code Generation" — Simulink Coder<br>"Code Generation" — Embedded Coder | `rtwdemo_counter`<br>`rtwdemo_fcnprotoctrl`<br>`rtwdemo_cppclass`<br>`rtwdemo_async`<br>"AUTOSAR Examples" in the Embedded Coder documentation |
| Integrate existing externally written C or C++ code with your model for simulation and code generation | "Block Creation" — Simulink<br>"External Code Integration" — Simulink Coder<br>"External Code Integration" — Embedded Coder | `rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code** |
| Generate code for on-target rapid prototyping on specific embedded microprocessors and IDEs | "Real-Time and Embedded Systems" — Embedded Coder | In `rtwdemos`, select one of the following: **Desktop IDEs**, **Desktop Targets**, **Embedded IDEs**, or **Embedded Targets** |

### Generating the Application Code

| Goals | Related Product Information | Examples |
|---|---|---|
| Optimize generated ANSI® C code for production (for example, disable floating-point code, remove termination and error handling code, and combine code entry points into single functions) | "Performance" — Simulink Coder<br>"Performance" — Embedded Coder | `rtwdemos`, select **Optimizations** |
| Optimize code for a specific run-time environment, using specialized function libraries | "Code Replacement" — Embedded Coder | `rtwdemo_crl_script` |
| Control the format and style of generated code | "Control Code Style" — Embedded Coder | `rtwdemo_parentheses` |
| Control comments inserted into generated code | "Add Custom Comments to Generated Code" — Embedded Coder | `rtwdemo_comments` |
| Enter special instructions or tags for postprocessing by third-party tools or processes | "Customize Post-Code-Generation Build Processing" — Simulink Coder | `rtwdemo_buildinfo` |
| Include requirements links in generated code | "Review of Requirements Links" — Simulink Verification and Validation | `rtwdemo_requirements` |
| Trace model blocks and subsystems to generated code and vice versa | "Code Tracing" — Embedded Coder<br>"Standards and Guidelines" — Embedded Coder | `rtwdemo_comments`<br>`rtwdemo_hyperlinks` |
| Integrate existing externally written code with code generated for a model | "Block Creation" — Simulink<br>"External Code Integration" — Simulink Coder<br>"External Code Integration" — Embedded Coder | `rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code** |

### Generating the Application Code (Continued)

| Goals | Related Product Information | Examples |
|---|---|---|
| Verify generated code for MISRA C®[1] and other run-time violations | "MISRA C Guidelines" — Embedded Coder<br>"Polyspace® Bug Finder™ Documentation"<br>"Polyspace Code Prover™ Documentation" | |
| Protect the intellectual property of component model design and generated code<br>Generate a binary file (shared library) | "Protected Model" — Simulink<br>"Shared Object Libraries" — Embedded Coder | |
| Generate a MEX-file S-function for a model or subsystem so that it can be shared with a third-party vendor | "Generated S-Function Block" — Simulink Coder | |
| Generate a shared library for a model or subsystem so that it can be shared with a third-party vendor | "Shared Object Libraries" — Embedded Coder | |
| Test generated production code with an environment or plant model to verify a conversion of the model to code | "Software-in-the-Loop (SIL) Simulation" — Embedded Coder | `rtwdemo_sil_pil_script` |

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

**Generating the Application Code (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Create an S-function wrapper for calling your generated source code from a model running in Simulink | “Write Wrapper S-Functions” — Simulink Coder | |
| Set up and run SIL tests on your host computer | “Software-in-the-Loop (SIL) Simulation” — Embedded Coder | `rtwdemo_sil_pil_script` |

### Integrating and Verifying Software

| Goals | Related Product Information | Examples |
|---|---|---|
| Integrate existing externally written C or C++ code with a model for simulation and code generation | "Block Creation" — Simulink<br>"External Code Integration" — Simulink Coder<br>"External Code Integration" — Embedded Coder | `rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code** |
| Connect to data interfaces for generated C code data structures | "Data Exchange" — Simulink Coder<br>"Data Exchange" — Embedded Coder | `rtwdemo_capi`<br>`rtwdemo_asap2` |
| Control the generation of code interfaces so that external software can compile, build, and invoke the generated code | "Function and Class Interfaces" — Embedded Coder | `rtwdemo_fcnprotoctrl`<br>`rtwdemo_cppclass` |
| Export virtual and function-call subsystems | "Export Code Generated from Model to External Application" — Embedded Coder | `rtwdemo_export_functions` |
| Include target-specific code | "Code Replacement" — Embedded Coder | `rtwdemo_crl_script` |
| Customize and control the build process | "Build Process" — Simulink Coder | `rtwdemo_buildinfo` |
| Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer | "Relocate Code to Another Development Environment" — Simulink Coder | `rtwdemo_buildinfo` |

**Integrating and Verifying Software (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Integrate software components as a complete system for testing in the target environment | "Component Verification" — Embedded Coder | |
| Generate source code for integration with specific production environments | "Code Generation" — Simulink Coder<br>"Code Generation" — Embedded Coder | `rtwdemo_async`<br>"AUTOSAR Examples" in the Embedded Coder documentation |
| Integrate code for a specific run-time environment, using specialized function libraries | "Code Replacement" — Embedded Coder | `rtwdemo_crl_script` |
| Enter special instructions or tags for postprocessing by third-party tools or processes | "Customize Post-Code-Generation Build Processing" — Simulink Coder | `rtwdemo_buildinfo` |
| Integrate existing externally written code with code generated for a model | "Block Creation" — Simulink<br>"External Code Integration" — Simulink Coder<br>"External Code Integration" — Embedded Coder | `rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code** |
| Connect to data interfaces for the generated C code data structures | "Data Exchange" — Simulink Coder<br>"Data Exchange" — Embedded Coder | `rtwdemo_capi`<br>`rtwdemo_asap2` |
| Customize and control the build process | "Build Process" — Simulink Coder | `rtwdemo_buildinfo` |

**Integrating and Verifying Software (Continued)**

| Goals | Related Product Information | Examples |
|---|---|---|
| Create a zip file that contains generated code files, static files, and dependent data for building the generated code in an environment other than your host computer | "Relocate Code to Another Development Environment" — Simulink Coder | `rtwdemo_buildinfo` |
| Schedule the generated code | "Time-Based Scheduling" — Simulink Coder | `rtwdemos`, select **Multirate Support** |
| Verify object code files in a target environment | "Software-in-the-Loop (SIL) Simulation" — Embedded Coder | `rtwdemo_sil_pil_script` |
| Set up and run PIL tests on your target system | "Processor-in-the-Loop (PIL) Simulation" — Embedded Coder | `rtwdemo_sil_pil_script` `rtwdemo_custom_pil_script` `rtwdemo_rtiostream_script` See the list of `supported hardware` for the Embedded Coder product on the MathWorks Web site, and then find an example for the related product of interest |

### Integrating, Verifying, and Calibrating System Components

| Goals | Related Product Information | Examples |
|---|---|---|
| Integrate the software and its microprocessor with the hardware environment for the final embedded system product<br><br>Add the complexity of the environment (or plant) under control to the test platform<br><br>Test and verify the embedded system or control unit by using a real-time target environment | "Hardware-in-the-Loop (HIL) Simulation" — Embedded Coder | |
| Generate source code for HIL testing | "Code Generation" — Simulink Coder<br><br>"Code Generation" — Embedded Coder<br><br>"Hardware-in-the-Loop (HIL) Simulation" — Embedded Coder | |
| Conduct hard real-time HIL testing using PCs | "Simulink Real-Time" | `doc xpcdemos` |
| Tune ECU properly for its intended use | "Data Exchange" — Simulink Coder<br><br>"Data Exchange" — Embedded Coder | |
| Generate ASAP2 data files | "ASAP2 Data Measurement and Calibration" — Simulink Coder | `rtwdemo_asap2` |
| Generate C API data interface files | "Data Interchange Using C API" — Simulink Coder | `rtwdemo_capi` |

## Target Environments and Applications

| In this section... |
|---|
| "About Target Environments" on page 1-30 |
| "Types of Target Environments" on page 1-30 |
| "Applications of Supported Target Environments" on page 1-33 |

### About Target Environments

In addition to generating source code, the code generator produces make or project files to build an executable program for a specific target environment. The generated make or project files are optional. If you prefer, you can build an executable for the generated source files by using an existing target build environment, such as a third-party integrated development environment (IDE). Applications of generated code range from calling a few exported C or C++ functions on a host computer to generating a complete executable program using a custom build process, for custom hardware, in an environment completely separate from the host computer running MATLAB and Simulink.

The code generator provides built-in *system target files* that generate, build, and execute code for specific target environments. These system target files offer varying degrees of support for interacting with the generated code to log data, tune parameters, and experiment with or without Simulink as the external interface to your generated code.

### Types of Target Environments

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

| **Target Environment** | **Description** |
|---|---|
| Host computer | The same computer that runs MATLAB and Simulink. Typically, a host computer is a PC or UNIX®[2] environment that uses a non-real-time operating system, such as Microsoft Windows® or Linux®[3]. Non-real-time (general purpose) operating systems are nondeterministic. For example, those operating systems might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Therefore, the executable for your generated code might run faster or slower than the sample rates that you specified in your model. |
| Real-time simulator | A different computer than the host computer. A real-time simulator can be a PC or UNIX environment that uses a real-time operating system (RTOS), such as:<br><br>• Simulink Real-Time system<br>• A real-time Linux system<br>• A Versa Module Eurocard (VME) chassis with PowerPC® processors running a commercial RTOS, such as VxWorks® from Wind River® Systems<br><br>The generated code runs in real time. The exact nature of execution varies based on the particular behavior of the system hardware and RTOS.<br><br>Typically, a real-time simulator connects to a host computer for data logging, interactive parameter tuning, and Monte Carlo batch execution studies. |
| Embedded microprocessor | A computer that you eventually disconnect from a host computer and run as a standalone computer as part of an electronics-based product. Embedded microprocessors range in price and performance, from high-end digital signal processors (DSPs) to process communication signals to inexpensive 8-bit fixed-point microcontrollers in mass production (for example, electronic parts produced in the millions of units). Embedded microprocessors can:<br><br>• Use a full-featured RTOS |

2. UNIX® is a registered trademark of The Open Group in the United States and other countries.

3. Linux® is a registered trademark of Linus Torvalds.

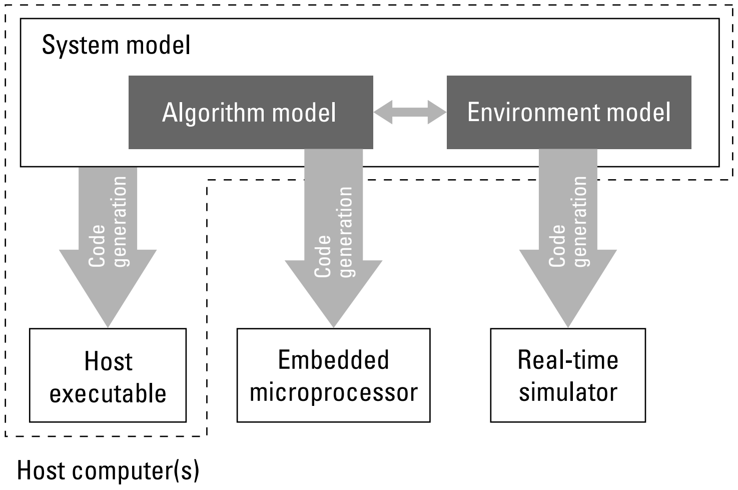
| Target Environment | Description |
|---|---|
| | • Be driven by basic interrupts<br>• Use rate monotonic scheduling provided with code generation |

A target environment can:

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

In addition, you can deploy different parts of a Simulink model on different target environments. For example, it is common to separate the component (algorithm or controller) portion of a model from the environment (or plant). Using Simulink to model an entire system (plant and controller) is often referred to as closed-loop simulation and can provide many benefits, such as early verification of a component.

The following figure shows example target environments for code generated for a model.



## Applications of Supported Target Environments

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

| Application | Description |
|---|---|
| **Host Computer** | |
| “Acceleration” | You apply techniques to speed up the execution of model simulation in the context of the MATLAB and Simulink environments. Accelerated simulations are especially useful when run time is long compared to the time associated with compilation and checking whether the target is up to date. |
| “Rapid Simulation” | You execute code generated for a model in non-real-time on the host computer, but outside the context of the MATLAB and Simulink environments. |
| “Shared Object Libraries” | You integrate components into a larger system. You provide generated source code and related dependencies for building a system in another environment or in a host-based shared library to which other code can dynamically link. |
| “Protect a Referenced Model” | You generate a protected model for use by a third-party vendor in another Simulink simulation environment. |
| **Real-Time Simulator** | |
| “Real-Time System Rapid Prototyping” | You generate, deploy, and tune code on a real-time simulator connected to the system hardware (for example, physical plant or vehicle) being controlled. This design step is crucial for validating whether a component can control the physical system. |

| Application | Description |
|---|---|
| "Integration and Reusable Components" | You integrate generated source code and dependencies for components into a larger system that is built in another environment. You can use shared library files for intellectual property protection. |
| "Real-Time System Rapid Prototyping" | You generate code for a detailed design that you can run in real time on an embedded microprocessor while tuning parameters and monitoring real-time data. This design step allows you to assess, interact with, and optimize code, using embedded compilers and hardware. |
| **Embedded Microprocessor** | |
| "Code Generation" | From a model, you generate code that is optimized for speed, memory usage, simplicity, and possibly, compliance with industry standards and guidelines. |
| "Software-in-the-Loop (SIL) Simulation" | You execute generated code with your plant model within Simulink to verify conversion of the model to code. You might change the code to emulate target word size behavior and verify numerical results expected when the code runs on an embedded microprocessor. Or, you might use actual target word sizes and just test production code behavior. |

| Application | Description |
|---|---|
| "Processor-in-the-Loop (PIL) Simulation" | You test an object code component with a plant or environment model in an open- or closed-loop simulation to verify model-to-code conversion, cross-compilation, and software integration. |
| Hardware-in-the-loop (HIL) testing | You verify an embedded system or embedded computing unit (ECU), using a real-time target environment. |

# MATLAB Tutorials

# Generate C Code from MATLAB Code

**In this section...**

## About MATLAB Coder

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

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

### How Embedded Coder Works With MATLAB Coder

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

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

## Getting Started Tutorials

The following tutorials will help you get started with using Embedded Coder to generate code from MATLAB code for embedded system applications.

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

### Prerequisites

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

- MATLAB
- MATLAB Coder
- Embedded Coder
- C compiler

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

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

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

### Setting Up Tutorial Files

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

## Controlling C Code Style

**In this section...**

### About This Tutorial

#### Learning Objectives

This tutorial shows you how to:

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

#### Prerequisites

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

### Required Files

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

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

## Copying Files Locally

Copy the tutorial files to a local working folder.

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

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

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

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

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

## Setting Up the MATLAB Coder Project

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

   ```
   cd work
   ```

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

2. At the MATLAB command line, enter

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

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

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

4 Define the type of input `x`.

**Why Specify an Input Definition?**

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

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

5 From the menu, select `Define Type`.

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

---

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

---

## Configuring Build Parameters

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

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

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

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

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

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

## Generating the C Code

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

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

## Viewing the Generated C Code

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

To view the generated code:

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

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

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

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

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

     case 2:
     y = 2.0;
     break;

     case 3:
     y = 3.0;
     break;

     default:
     y = 4.0;
     break;
   ```

```
}
```

## Key Points to Remember

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

## Learn More

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

# Generating Reentrant C Code from MATLAB Code

**In this section...**

## About This Tutorial

### Learning Objectives

This tutorial shows you how to:

- Generate reentrant code from MATLAB code that does not use persistent or global data

---

**Note** This example runs on Windows only.

---

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

### Prerequisites

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

### Required Files

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

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

## Copying Files Locally

Copy the tutorial files to a local working folder.

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

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

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

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

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

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

```
cd work
```

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

## About the Example

This example requires libraries that are specific to the Microsoft Windows operating system and, therefore, runs only on Windows platforms. It is a simple, multithreaded example that does not use persistent or global data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

### Contents of matrix_exp.m

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

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

- *`primary_function_name`*`StackData`

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

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

- *`primary_function_name`*`PersistentData`

  If your algorithm uses persistent or global variables, `codegen` provides a separate structure for them and adds a pointer to this structure to the memory allocation structure. Having a separate structure for persistent and global variables allows you to allocate memory for these variables once and share them with all threads if desired. However, if there is not communication between threads, you can choose to allocate memory for these variables per thread or per application.

## Providing a main Function

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

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

### Contents of main.c

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

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

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

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

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

  data1.spillData = sd1;
  data2.spillData = sd2;
for (i=0;i<NUMELEMENTS;i++) {
    data1.in[i] = 1;
    data1.out[i] = 0;
    data2.in[i] = 1.1;
    data2.out[i] = 0;
}

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

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

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

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

free(sd1);
free(sd2);

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

## Configuring Build Parameters

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

1 Create a configuration object.

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

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

2 Enable reentrant code generation.

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

## Generating the C Code

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

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

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

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

## Viewing the Generated C Code

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

To view this header file:

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

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

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

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

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

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

## Running the Code

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

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

The executable runs and reports completion.

## Key Points to Remember

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

## Learn More

| To... | See... |
|---|---|
| Learn more about the generated code API | "Generated Code API" |
| Call reentrant code without persistent or global data on UNIX | "Call Reentrant Code with No Persistent or Global Data (UNIX Only)" |
| Call reentrant code with persistent data on Windows | "Call Reentrant Code — Multithreaded with Persistent Data (Windows Only)" |
| Call reentrant code with persistent data on UNIX | "Call Reentrant Code — Multithreaded with Persistent Data (UNIX Only)" |

## Tracing Between Generated C Code and MATLAB Code

**In this section...**

### About This Tutorial

#### Learning Objectives

This tutorial shows you how to:

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

#### Prerequisites

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

### Required File

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

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

## Copying Files Locally

Copy the tutorial file to a local working folder.

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

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

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

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

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

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

   ```
   cd work
   ```

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

### Contents of polar2cartesian.m

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

## Configuring Build Parameters

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

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

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

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

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

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

## Generating the C Code

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

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

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

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

## Viewing the Generated C Code

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

To view the generated code:

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

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

**3** On this tab, click the `polar2cartesian.c` link.

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

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

## Tracing Back to the Source MATLAB Code

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

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

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

## Key Points to Remember

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

- Use the `-report` option to create a code generation report.
- Use the `-args` option to specify the class, size, and complexity of input parameters.

## Learn More

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

# Simulink Code Generation Tutorials

# Generate C Code from Simulink Models

**In this section...**

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

- "Configure a Model for Code Generation" on page 3-6
- "Generate and Analyze C Code" on page 3-12
- "Customize Function Interface and File Packaging" on page 3-30
- "Define Data in the Generated Code" on page 3-37
- "Customize Code Appearance" on page 3-24
- "Deploy and Verify Executable Program" on page 3-46

## Prerequisites

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

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

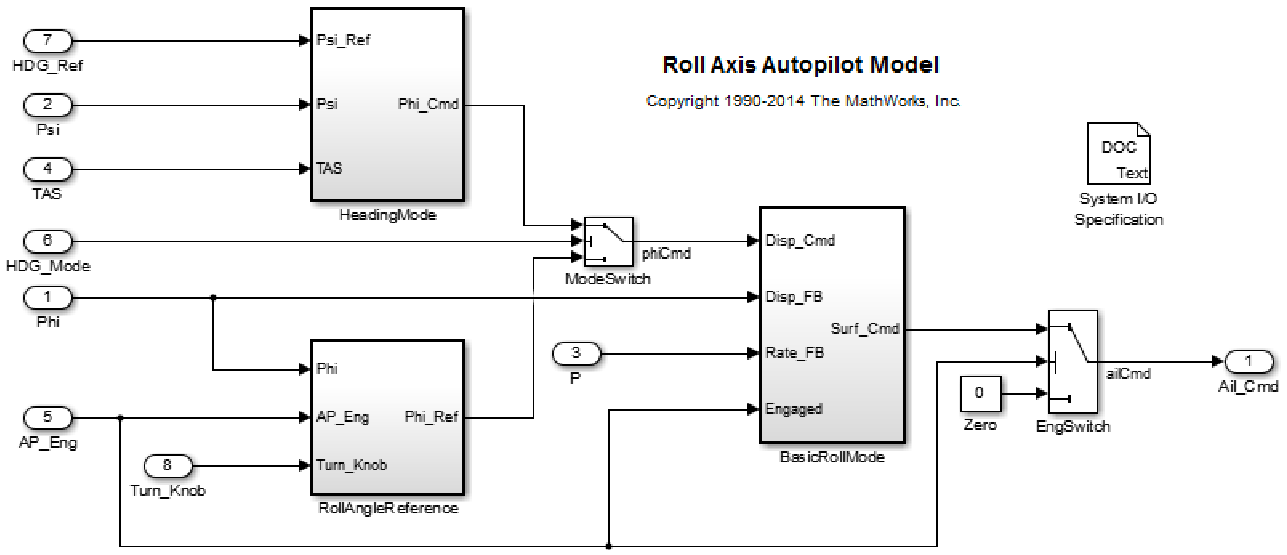
## Example Models in Tutorials

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

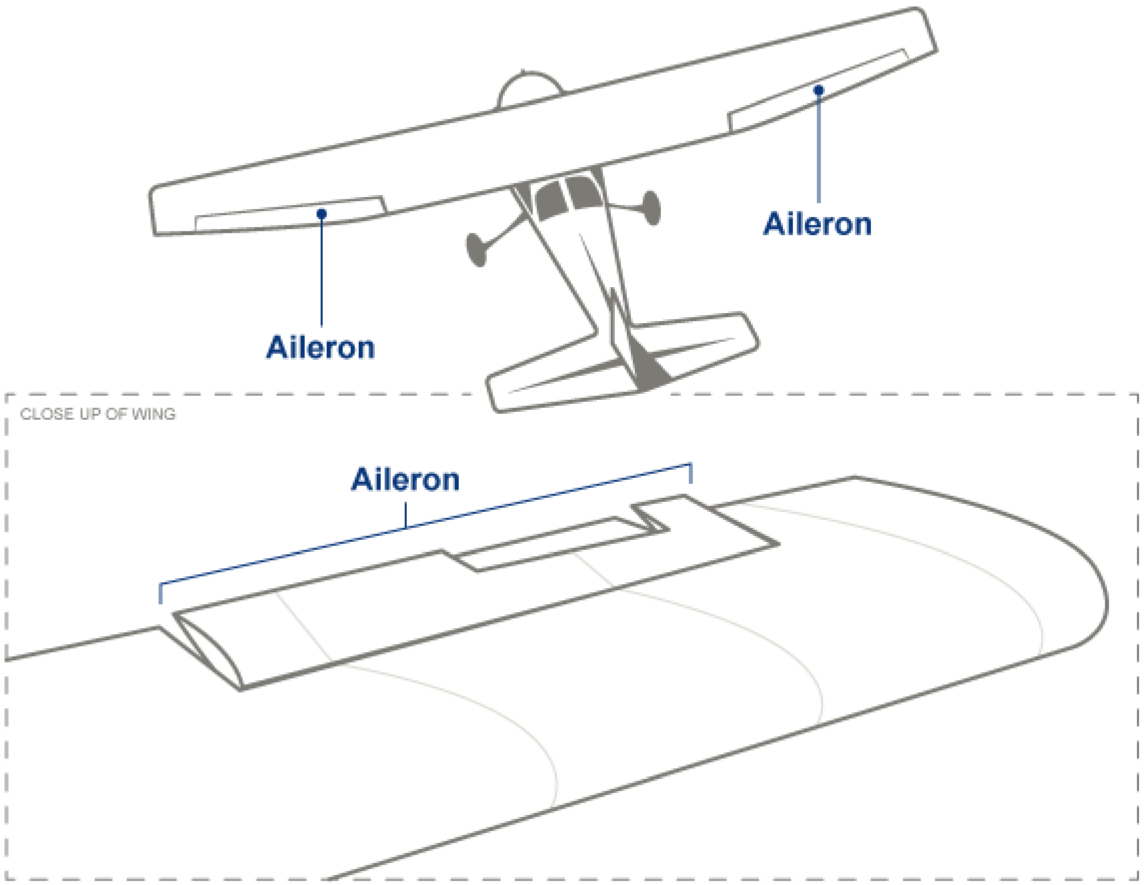
The Embedded Coder tutorials use the `rtwdemo_roll` model, which has been verified for simulation. To open the model, in the Command Window, type:

```
rtwdemo_roll
```

The model opens in the Simulink Editor.



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



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

Two additional models are provided for the Embedded Coder tutorials:

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

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

## Configure a Model for Code Generation

**In this section...**

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

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

### Solver for Code Generation

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

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

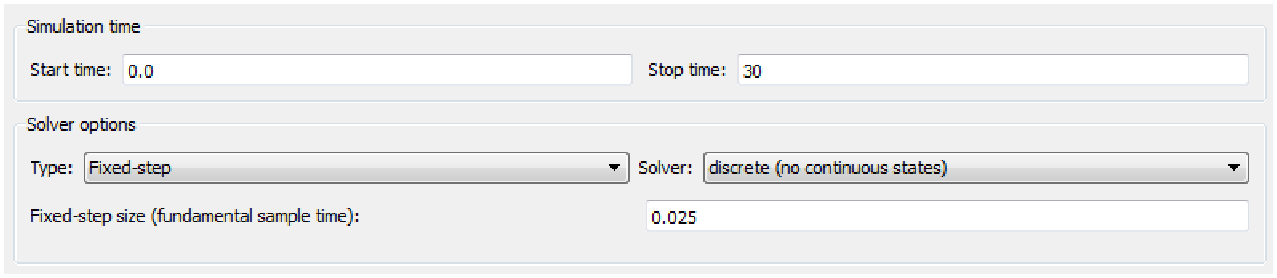
   ```
   rtwdemo_roll
   ```

2. Save the model to a local folder as `roll.slx`.

3. To open the Configuration Parameters dialog box, on the Simulink Editor toolbar, click the **Model Configuration Parameters** icon.

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

To generate code, the model must use a fixed-step solver, which maintains a constant (fixed) step size. In the generated code, the **Solver** parameter applies a fixed-step integration technique for computing the state derivative of the model. The **Fixed-step size** parameter sets the base rate, which must be the lowest common multiple of all rates in the system. For `roll`, the following solver settings are selected.



Simulation time

Start time: 0.0 Stop time: 30

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

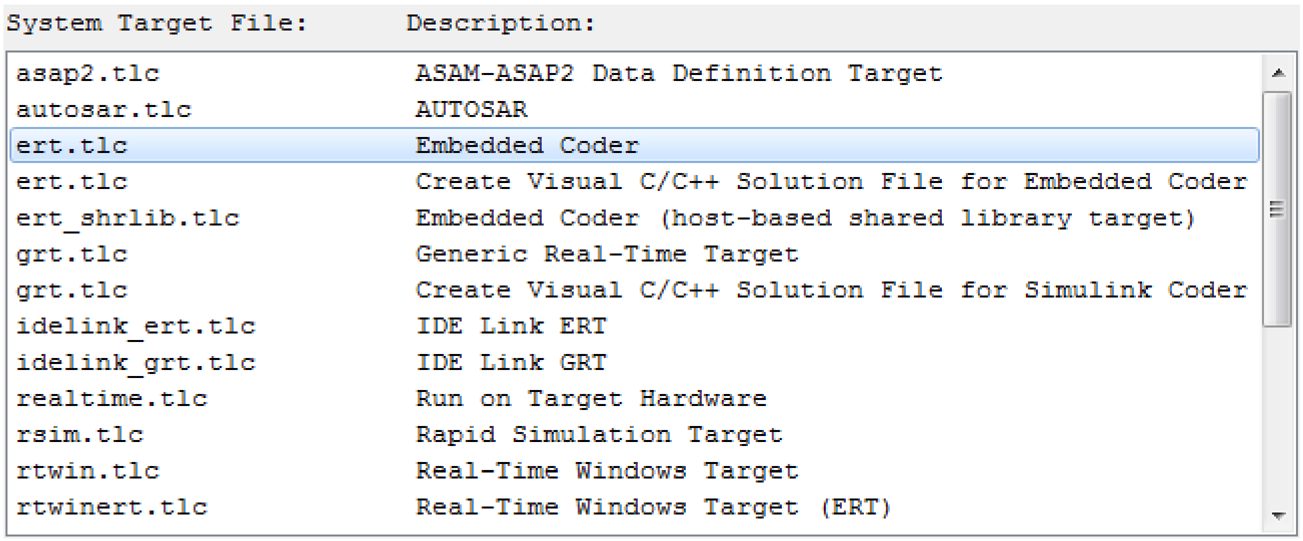
Fixed-step size (fundamental sample time): 0.025

## Code Generation Target

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

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

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



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

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

## Check Model Configuration

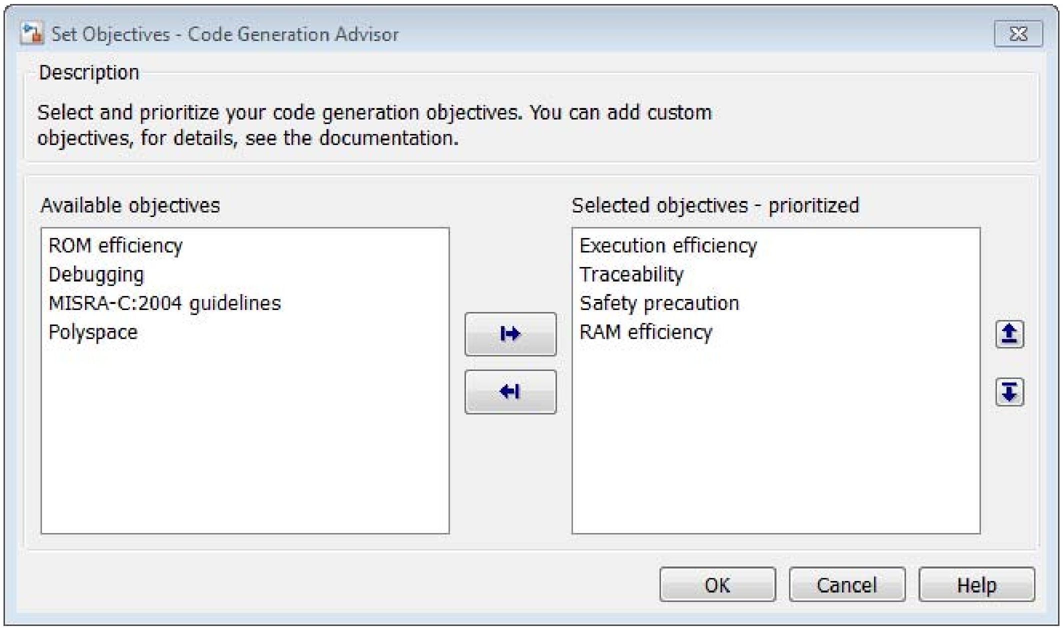
When generating code for real-time deployment, your application might have objectives related to code efficiency, memory usage, safety precaution, and traceability. You can run the Code Generation Advisor to assess whether the model configuration settings meet your set of prioritized objectives. After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

### Set Code Generation Objectives with Code Generation Advisor

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

2 Click **Set Objectives**.

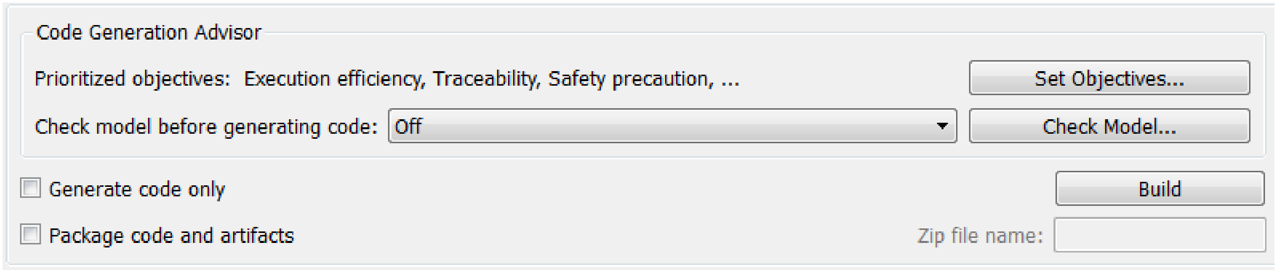
3 In the Select Objectives dialog box, the following objectives are in the **Selected objectives — prioritized** list in the following order: `Execution efficiency`, `Traceability`, `Safety precaution`, and `RAM efficiency`.


Set Objectives - Code Generation Advisor
Description
Available objectives
Selected objectives - prioritized
OK
Cancel
Help

Select and prioritize your code generation objectives. You can add custom objectives, for details, see the documentation.

| Available objectives | Selected objectives - prioritized |
|---|---|
| ROM efficiency | Execution efficiency |
| Debugging | Traceability |
| MISRA-C:2004 guidelines | Safety precaution |
| Polyspace | RAM efficiency |

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



Prioritized objectives: Execution efficiency, Traceability, Safety precaution, ...

Check model before generating code: Off

☐ Generate code only

☐ Package code and artifacts

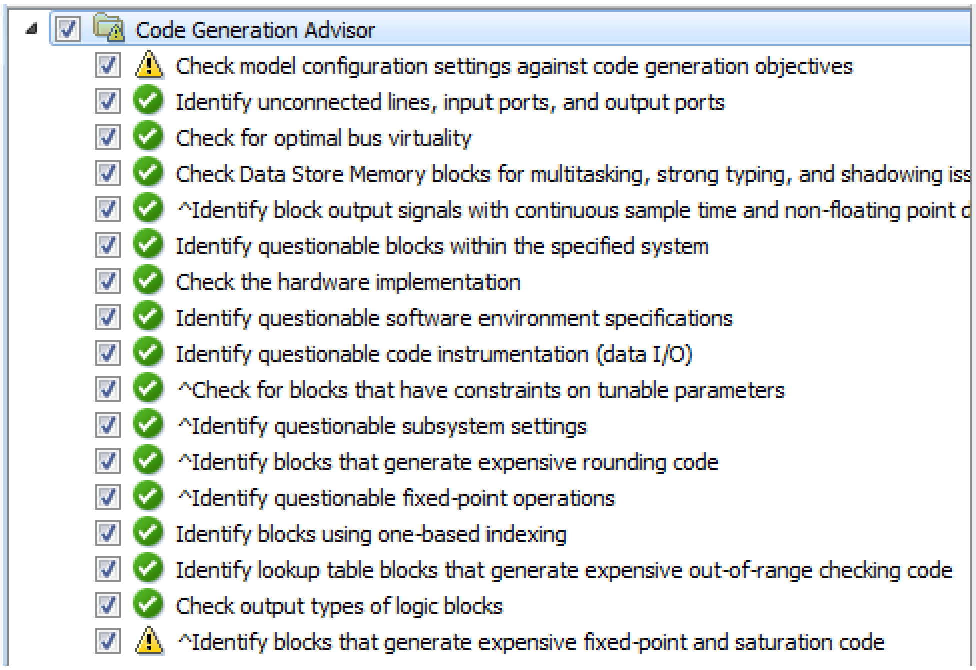
Zip file name:

## Check Model Against Code Generation Objectives

1 In the Configuration Parameters dialog box, on the **Code Generation** pane, click **Check Model**.

2 In the System Selector dialog box, click **OK** to run checks on `roll`.

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



Code Generation Advisor

- ☑ ⚠ Check model configuration settings against code generation objectives
- ☑ ✓ Identify unconnected lines, input ports, and output ports
- ☑ ✓ Check for optimal bus virtuality
- ☑ ✓ Check Data Store Memory blocks for multitasking, strong typing, and shadowing iss
- ☑ ✓ ^Identify block output signals with continuous sample time and non-floating point d
- ☑ ✓ Identify questionable blocks within the specified system
- ☑ ✓ Check the hardware implementation
- ☑ ✓ Identify questionable software environment specifications
- ☑ ✓ Identify questionable code instrumentation (data I/O)
- ☑ ✓ ^Check for blocks that have constraints on tunable parameters
- ☑ ✓ ^Identify questionable subsystem settings
- ☑ ✓ ^Identify blocks that generate expensive rounding code
- ☑ ✓ ^Identify questionable fixed-point operations
- ☑ ✓ Identify blocks using one-based indexing
- ☑ ✓ Identify lookup table blocks that generate expensive out-of-range checking code
- ☑ ✓ Check output types of logic blocks
- ☑ ⚠ ^Identify blocks that generate expensive fixed-point and saturation code

## View Model Configuration Recommendations

In the Code Generation Advisor window:

1. In the left pane, click **Check model configuration settings against code generation objectives**.

2. In the right pane, review the recommendations for the configuration parameters in the table.

3. To change the configuration parameters that caused the warnings to the software-recommended settings, click **Modify Parameters**. The **Result** table displays the parameters and changed values. Clicking a parameter name displays Configuration Parameters dialog box pane where the parameter exists.

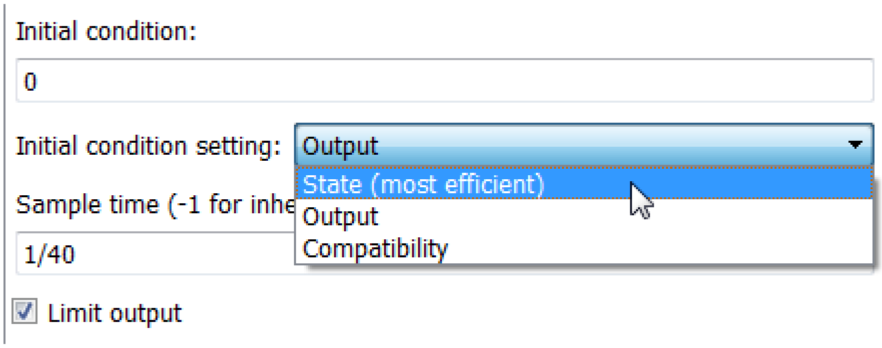
4 In the left pane, click the next warning for **Identify blocks that generate expensive fixed-point and saturation code**.

5 In the right pane, find the first warning, **Identify Discrete Integrator blocks for questionable fixed-point operations**. Under the warning, click the link to the Integrator block.

   In the Simulink Editor, the Integrator block is highlighted in blue.

6 Right-click the Integrator block and in the list, select `Block Parameters(DiscreteIntegrator)`.

7 In the Block Parameter dialog box, set the **Initial condition setting** to `State (most efficient)`.



Initial condition:

0

Initial condition setting: Output

Sample time (-1 for inhe

1/40

[✓] Limit output

8 Click **Apply** and **OK**.

9 Save your model.

The example model `rtwdemo_roll_codegen` contains the modifications made in this example to `rtwdemo_roll`. The next example shows how to generate code, examine the code, and trace between the code and model. See "Generate and Analyze C Code" on page 3-12.

## Generate and Analyze C Code

**In this section...**

After configuring your model for code generation, you generate and view the code. To analyze the generated code, you can generate an HTML code generation report that provides a view of the generated code and information about the code. This example uses the configured model `roll` from the example, "Generate and Analyze C Code" on page 3-12. For this example, open `rtwdemo_roll_codegen` and save it to a local folder as `roll.slx`.

### Generate Code

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

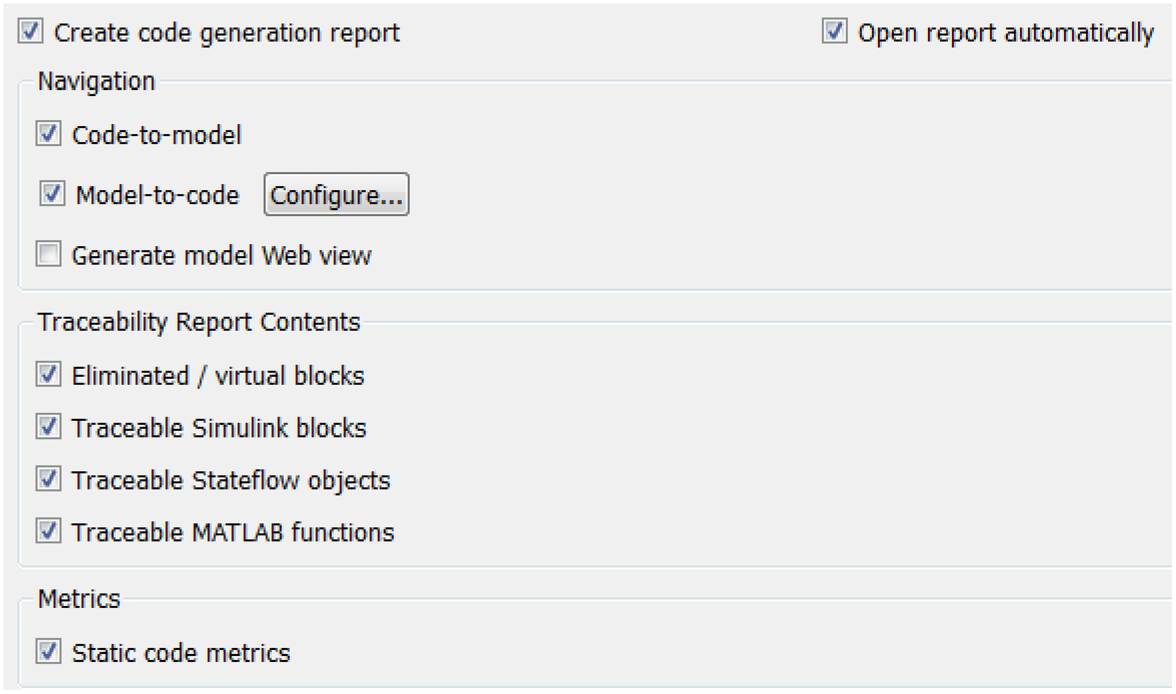
1 Open the Configuration Parameters dialog box.

2 In the left navigation pane, select the **Code Generation > Report** pane.

3 Observe the selected parameters that create a code generation report and include traceability between the code to the model.

   - "Create code generation report"
   - "Open report automatically"
   - "Code-to-model"
   - "Model-to-code", which enables the Traceability Report Contents parameters.

4 To include static code metrics in the code generation report, confirm that "Static code metrics" is selected.



[✓] Create code generation report [✓] Open report automatically

Navigation

[✓] Code-to-model

[✓] Model-to-code

[ ] Generate model Web view

Traceability Report Contents

[✓] Eliminated / virtual blocks

[✓] Traceable Simulink blocks

[✓] Traceable Stateflow objects

[✓] Traceable MATLAB functions

Metrics

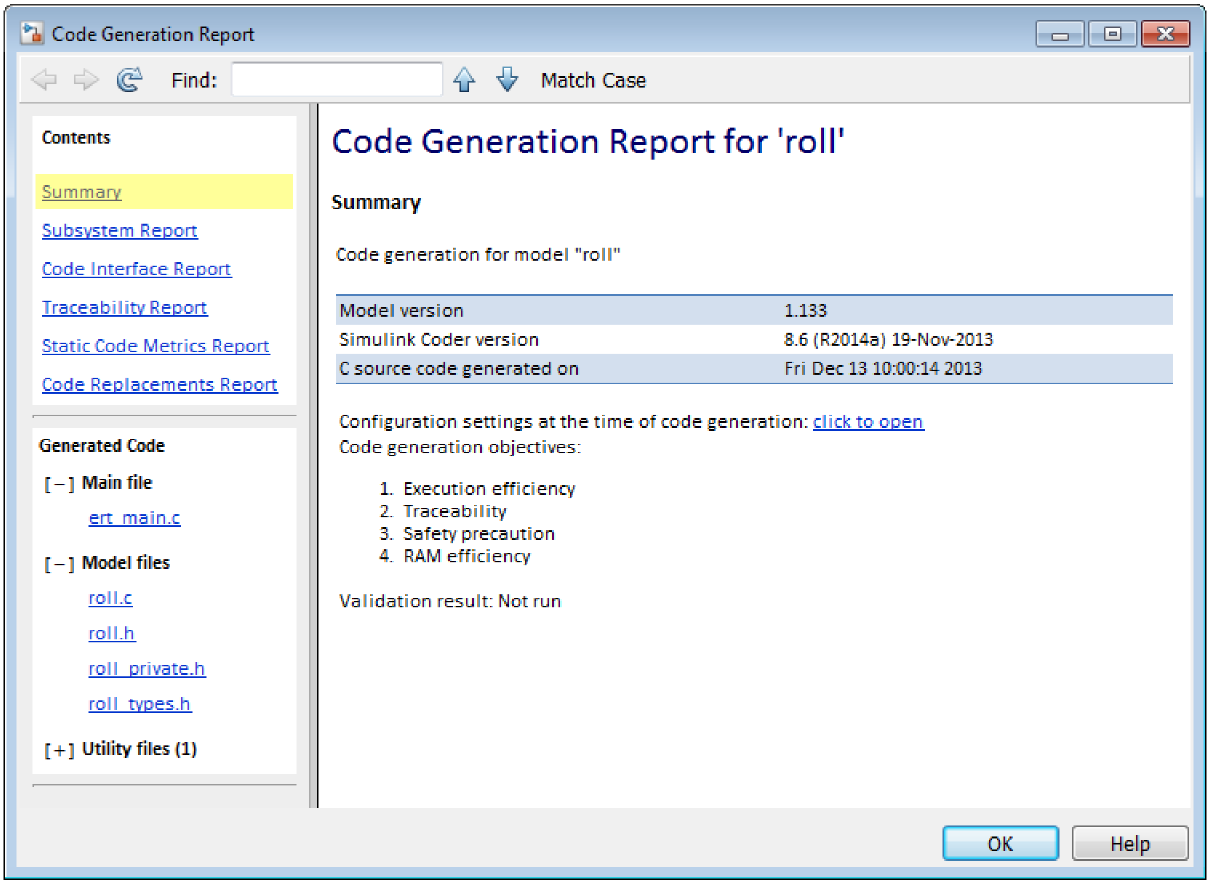
[✓] Static code metrics

5 On the **Code Generation** pane, select the **Generate code only** check box.

6 Click **Apply**.

7 Click **Generate Code**. You can also press **Ctrl+B** to generate code.

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



Code Generation Report for 'roll'

Summary

Code generation for model "roll"

| | |
|---|---|
| Model version | 1.133 |
| Simulink Coder version | 8.6 (R2014a) 19-Nov-2013 |
| C source code generated on | Fri Dec 13 10:00:14 2013 |

Configuration settings at the time of code generation: click to open
Code generation objectives:

1. Execution efficiency
2. Traceability
3. Safety precaution
4. RAM efficiency

Validation result: Not run

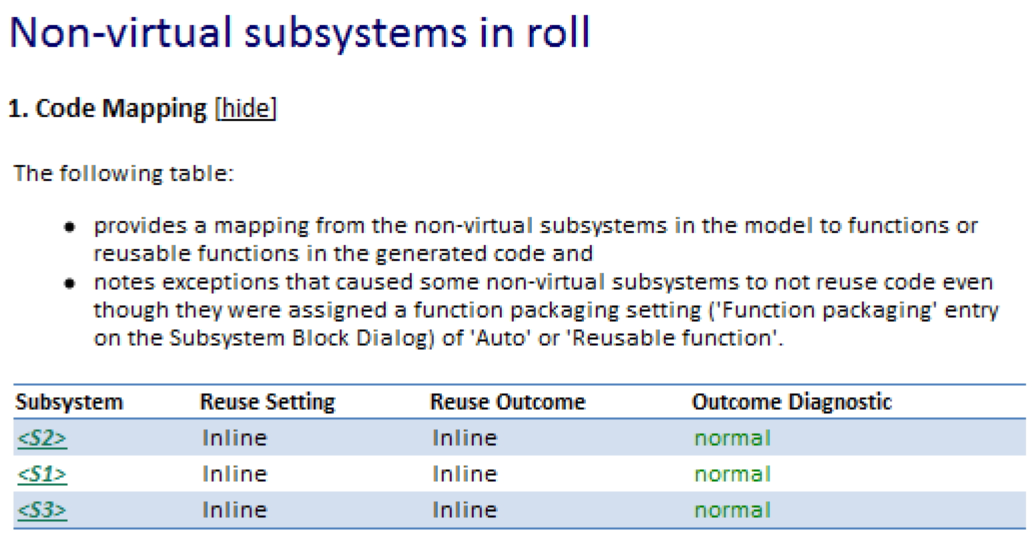
**Note** If you close the code generation report, you can reopen the report from the Simulink Editor by selecting the menu option: **Code > C/C++ Code > Code Generation Report > Open Model Report**.

## Analyze the Generated Code

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

## Subsystem Report

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



## Non-virtual subsystems in roll

### 1. Code Mapping [hide]

The following table:

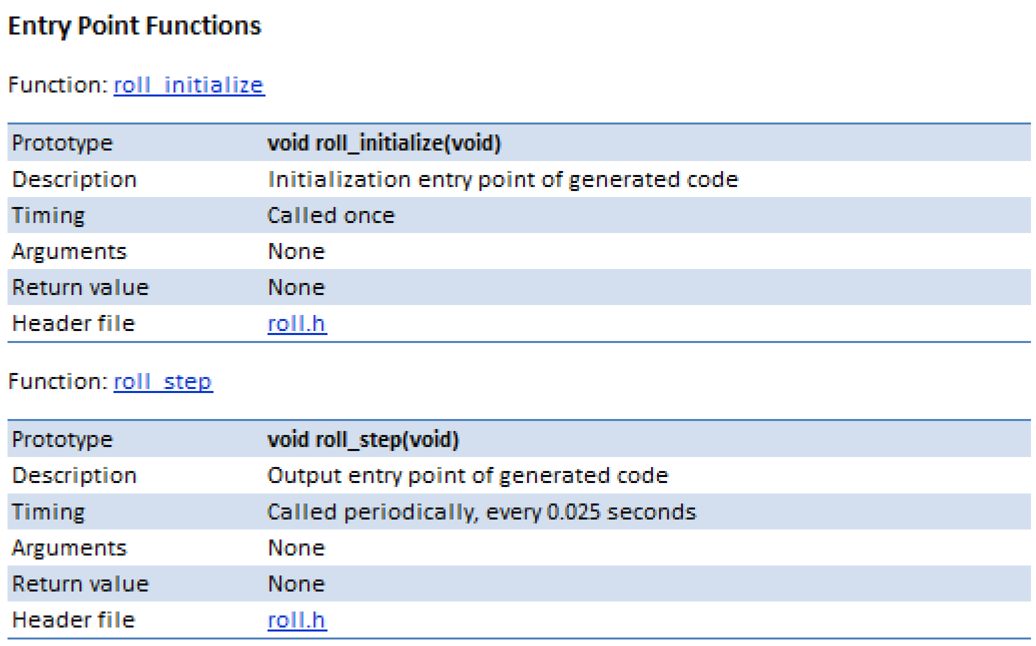
- provides a mapping from the non-virtual subsystems in the model to functions or reusable functions in the generated code and
- notes exceptions that caused some non-virtual subsystems to not reuse code even though they were assigned a function packaging setting ('Function packaging' entry on the Subsystem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

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

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

## Code Interface Report

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



**Entry Point Functions**

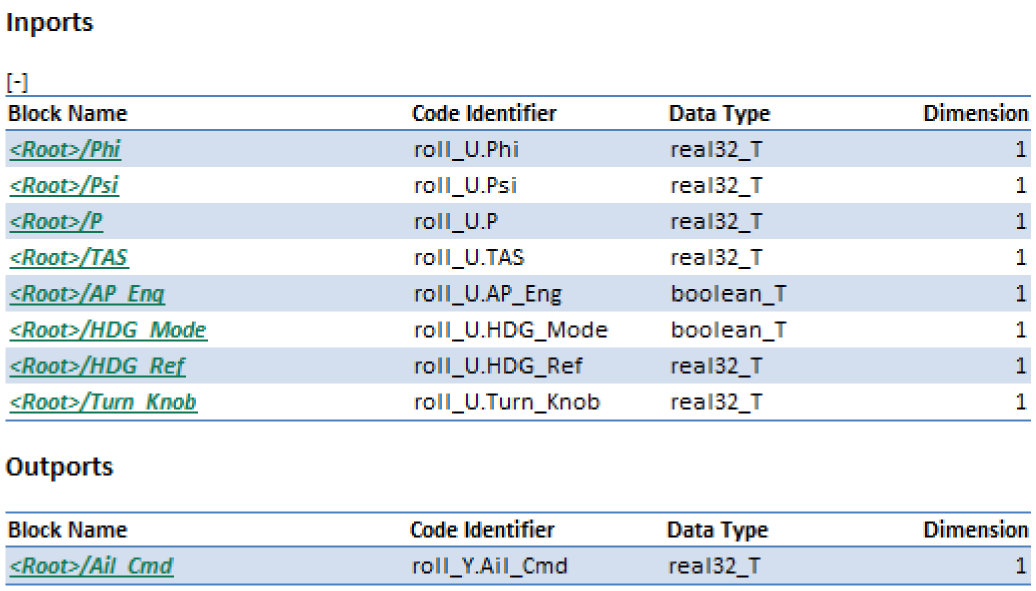
Function: roll_initialize

| | |
|---|---|
| Prototype | **void roll_initialize(void)** |
| Description | Initialization entry point of generated code |
| Timing | Called once |
| Arguments | None |
| Return value | None |
| Header file | roll.h |

Function: roll_step

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

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



**Inports**

[-]

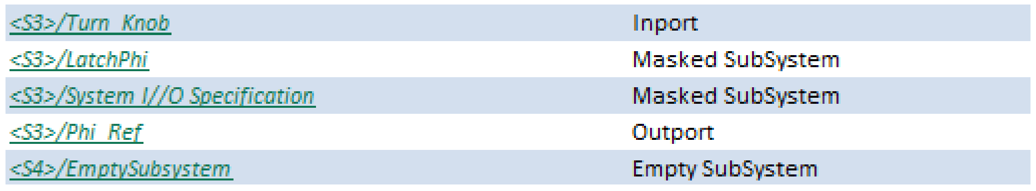
| Block Name | Code Identifier | Data Type | Dimension |
|---|---|---|---|
| <Root>/Phi | roll_U.Phi | real32_T | 1 |
| <Root>/Psi | roll_U.Psi | real32_T | 1 |
| <Root>/P | roll_U.P | real32_T | 1 |
| <Root>/TAS | roll_U.TAS | real32_T | 1 |
| <Root>/AP_Eng | roll_U.AP_Eng | boolean_T | 1 |
| <Root>/HDG_Mode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

**Outports**

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

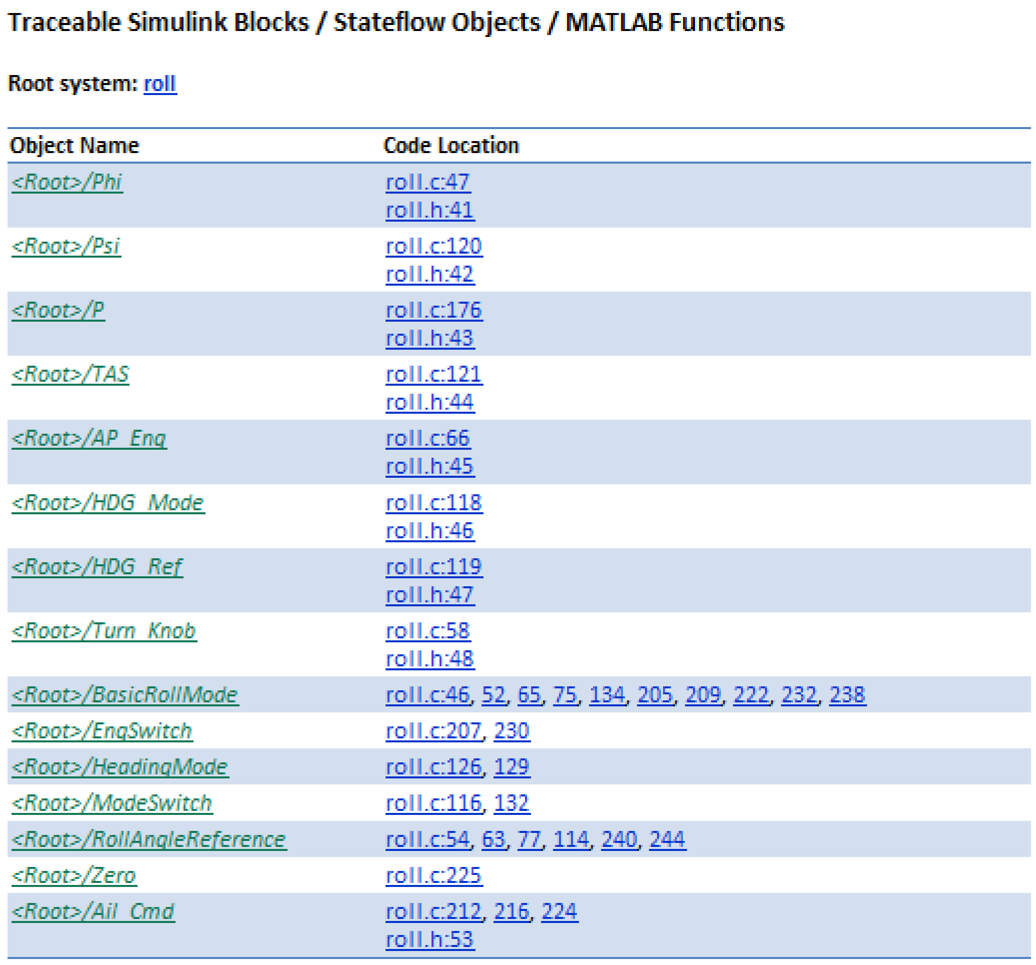
## Traceability Report

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



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

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



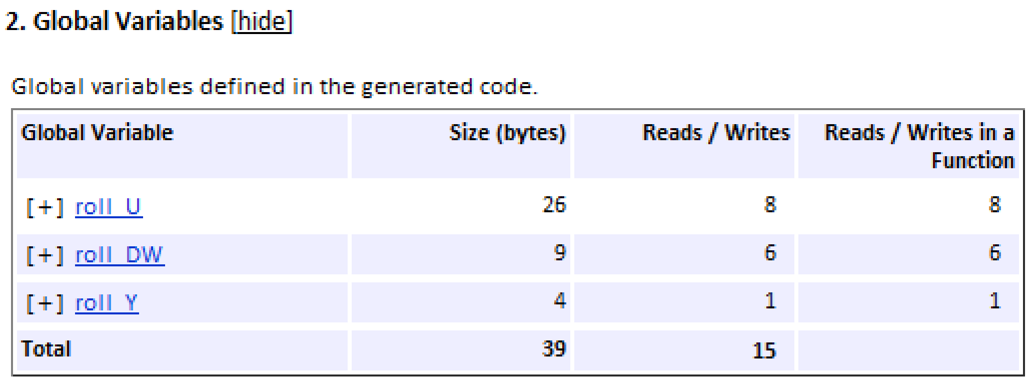
### Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

**Root system:** roll

| Object Name | Code Location |
|---|---|
| <Root>/Phi | roll.c:47<br>roll.h:41 |
| <Root>/Psi | roll.c:120<br>roll.h:42 |
| <Root>/P | roll.c:176<br>roll.h:43 |
| <Root>/TAS | roll.c:121<br>roll.h:44 |
| <Root>/AP_Eng | roll.c:66<br>roll.h:45 |
| <Root>/HDG_Mode | roll.c:118<br>roll.h:46 |
| <Root>/HDG_Ref | roll.c:119<br>roll.h:47 |
| <Root>/Turn_Knob | roll.c:58<br>roll.h:48 |
| <Root>/BasicRollMode | roll.c:46, 52, 65, 75, 134, 205, 209, 222, 232, 238 |
| <Root>/EngSwitch | roll.c:207, 230 |
| <Root>/HeadingMode | roll.c:126, 129 |
| <Root>/ModeSwitch | roll.c:116, 132 |
| <Root>/RollAngleReference | roll.c:54, 63, 77, 114, 240, 244 |
| <Root>/Zero | roll.c:225 |
| <Root>/Ail_Cmd | roll.c:212, 216, 224<br>roll.h:53 |

## Static Code Metrics Report

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



2. Global Variables [hide]

Global variables defined in the generated code.

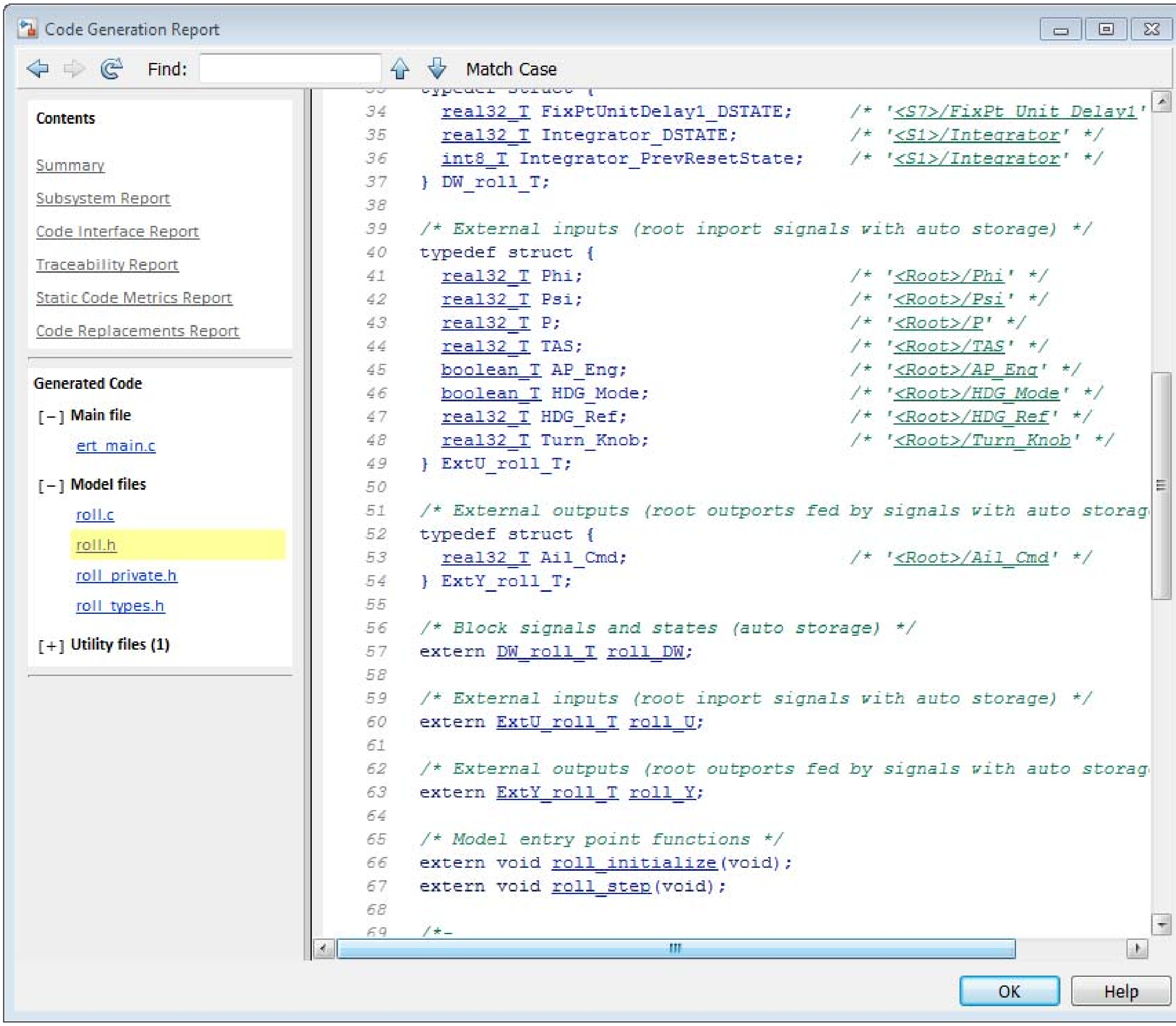
| Global Variable | Size (bytes) | Reads / Writes | Reads / Writes in a Function |
|---|---|---|---|
| [+] roll_U | 26 | 8 | 8 |
| [+] roll_DW | 9 | 6 | 6 |
| [+] roll_Y | 4 | 1 | 1 |
| Total | 39 | 15 | |

## Code Replacements Report

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

## Generated Code

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



```
  real32_T FixPtUnitDelay1_DSTATE;     /* '<S7>/FixPt Unit Delay1'
  real32_T Integrator_DSTATE;          /* '<S1>/Integrator' */
  int8_T Integrator_PrevResetState;    /* '<S1>/Integrator' */
} DW_roll_T;

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

/* External outputs (root outports fed by signals with auto storag
typedef struct {
  real32_T Ail_Cmd;                    /* '<Root>/Ail_Cmd' */
} ExtY_roll_T;

/* Block signals and states (auto storage) */
extern DW_roll_T roll_DW;

/* External inputs (root inport signals with auto storage) */
extern ExtU_roll_T roll_U;

/* External outputs (root outports fed by signals with auto storag
extern ExtY_roll_T roll_Y;

/* Model entry point functions */
extern void roll_initialize(void);
extern void roll_step(void);

/*-
```

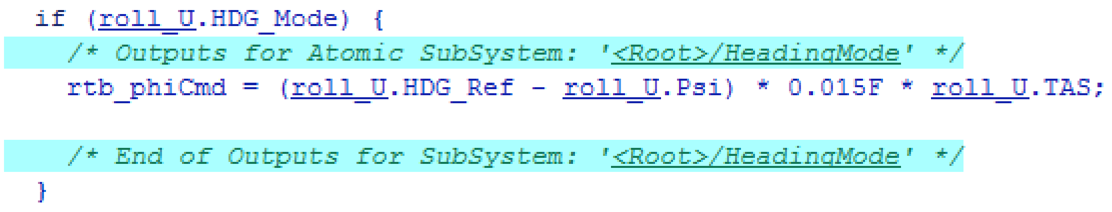
## Trace Between Code and Model

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

- On the **Code Generation > Report** pane:
  - "Code-to-model"
  - "Model-to-code"
- On the **Code Generation > Comments** pane:
  - "Include comments"
  - "Simulink block / Stateflow object comments"

### Trace from Model to Code

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



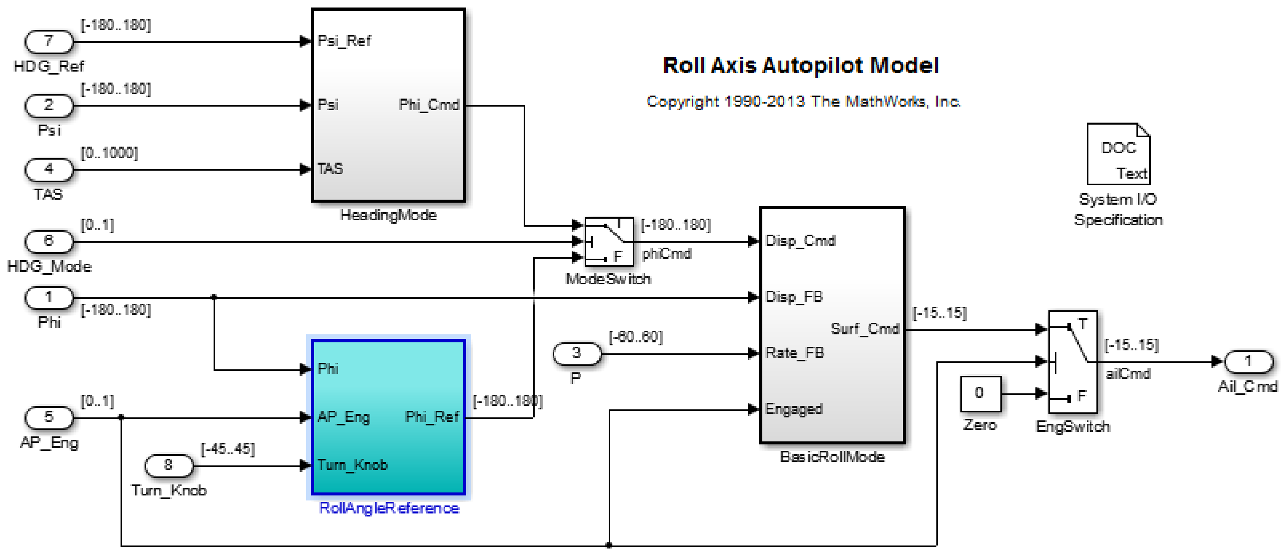
```
if (roll_U.HDG_Mode) {
  /* Outputs for Atomic SubSystem: '<Root>/HeadingMode' */
  rtb_phiCmd = (roll_U.HDG_Ref - roll_U.Psi) * 0.015F * roll_U.TAS;

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

### Trace from Code to Model

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

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

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



After reviewing the reports and analyzing the generated code, you can change the appearance of the generated code according to defined style standards. To change the generation of comments, identifiers, and code style, see the next example, "Customize Code Appearance" on page 3-24.

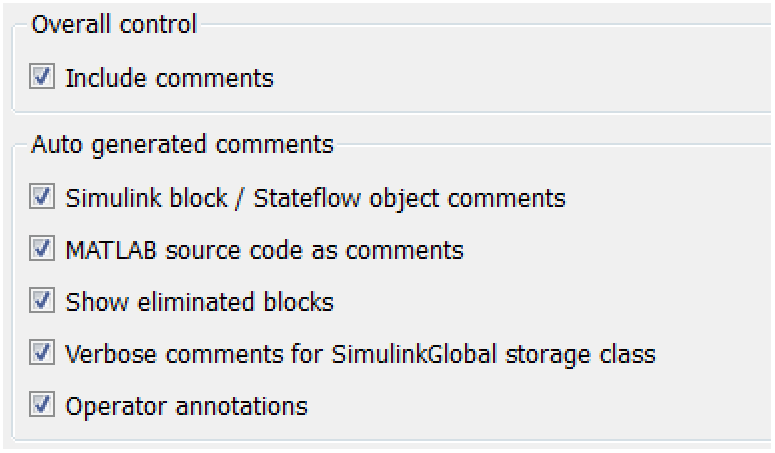
## Customize Code Appearance

**In this section...**

Modifying the code appearance helps you to adhere to your coding standards and enhance the readability of the code for code reviews. You can change the appearance of the generated code by specifying comment style, customizing identifier names, and choosing from several code style parameters. This example uses the configured model `roll` from the example, "Generate and Analyze C Code" on page 3-12. For this example, open `rtwdemo_roll_codegen` and save it to a local folder as `roll.slx`.

### Comments

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



Overall control

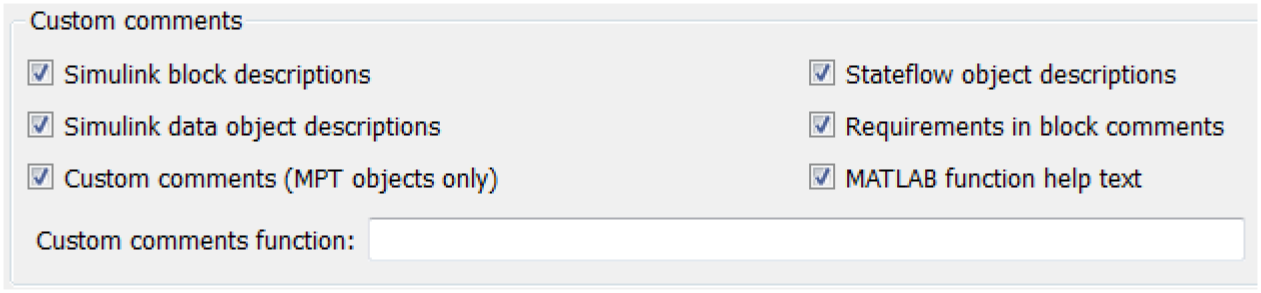
- [✓] Include comments

Auto generated comments

- [✓] Simulink block / Stateflow object comments
- [✓] MATLAB source code as comments
- [✓] Show eliminated blocks
- [✓] Verbose comments for SimulinkGlobal storage class
- [✓] Operator annotations

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

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



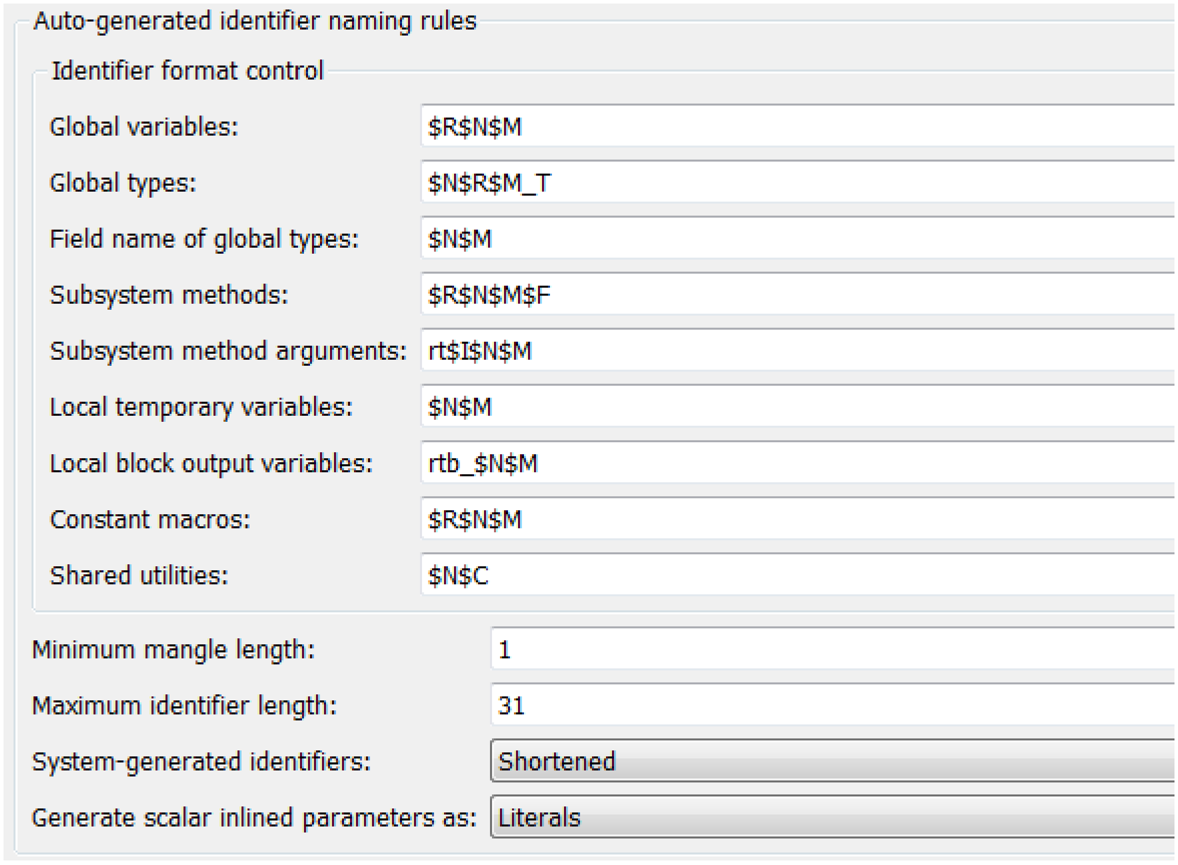
Custom comments

- [✓] Simulink block descriptions
- [✓] Simulink data object descriptions
- [✓] Custom comments (MPT objects only)
- [✓] Stateflow object descriptions
- [✓] Requirements in block comments
- [✓] MATLAB function help text

Custom comments function:

## Identifiers

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



Auto-generated identifier naming rules

Identifier format control

| | |
|---|---|
| Global variables: | $R$N$M |
| Global types: | $N$R$M_T |
| Field name of global types: | $N$M |
| Subsystem methods: | $R$N$M$F |
| Subsystem method arguments: | rt$I$N$M |
| Local temporary variables: | $N$M |
| Local block output variables: | rtb_$N$M |
| Constant macros: | $R$N$M |
| Shared utilities: | $N$C |
| Minimum mangle length: | 1 |
| Maximum identifier length: | 31 |
| System-generated identifiers: | Shortened |
| Generate scalar inlined parameters as: | Literals |

Common tokens to include are:

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

- `$R`: This token inserts the root model name into the identifier. When you use referenced models, this token is required.

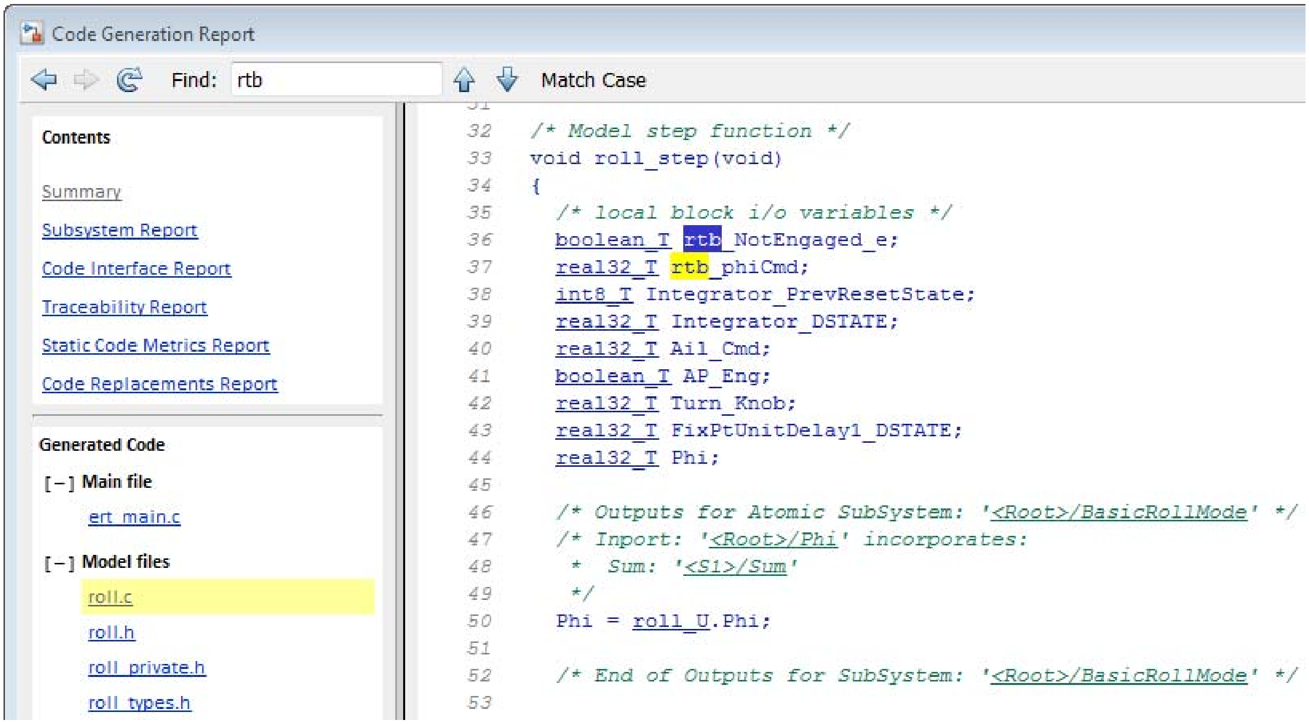
For example, in the `roll` model, the **Local block output variables** is specified with an `rtb` prefix.

**1** If the code generation report is not open, generate code for the `roll` model.

**2** In the HTML code generation report, in the left navigation pane, select `roll.c`.

**3** At the top of the window, in the **Find** box, type `rtb` and press **Enter**.

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

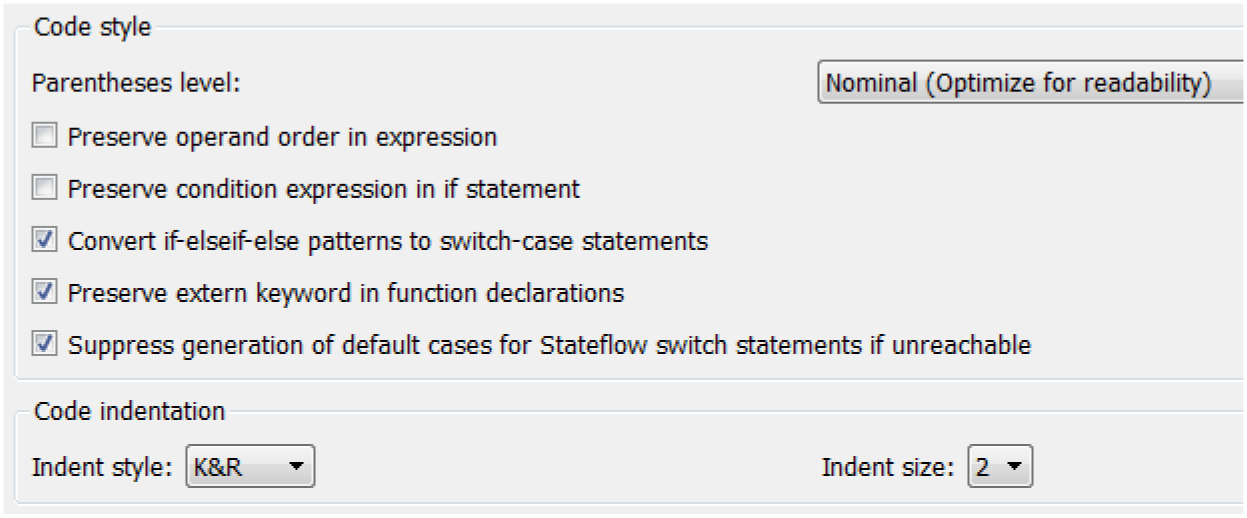


```
/* Model step function */
void roll_step(void)
{
  /* local block i/o variables */
  boolean_T rtb_NotEngaged_e;
  real32_T rtb_phiCmd;
  int8_T Integrator_PrevResetState;
  real32_T Integrator_DSTATE;
  real32_T Ail_Cmd;
  boolean_T AP_Eng;
  real32_T Turn_Knob;
  real32_T FixPtUnitDelay1_DSTATE;
  real32_T Phi;

  /* Outputs for Atomic SubSystem: '<Root>/BasicRollMode' */
  /* Inport: '<Root>/Phi' incorporates:
   *  Sum: '<S1>/Sum'
   */
  Phi = roll_U.Phi;

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

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

## Code Style

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


Code style

Parentheses level: Nominal (Optimize for readability)

☐ Preserve operand order in expression

☐ Preserve condition expression in if statement

☑ Convert if-elseif-else patterns to switch-case statements

☑ Preserve extern keyword in function declarations

☑ Suppress generation of default cases for Stateflow switch statements if unreachable

Code indentation

Indent style: K&R

Indent size: 2

The parameters allow you to control the following code styles:

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

With Embedded Coder, you can specify how the software generates function interfaces and packages the generated files. For more information, see the next example, “Customize Function Interface and File Packaging” on page 3-30.

## Customize Function Interface and File Packaging

**In this section...**

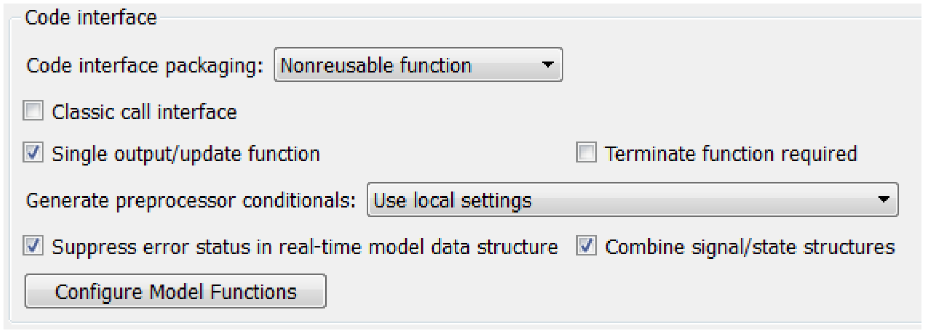
With Embedded Coder, you can specify the function interfaces for models and atomic subsystems in the generated code. You can also specify how the code is placed into files and folders. This example uses the configured model `roll` from the example, "Generate and Analyze C Code" on page 3-12. For this example, open `rtwdemo_roll_codegen` and save it to a local folder as `roll.slx`.

### Model Interface

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

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

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


Code interface

Code interface packaging: Nonreusable function

☐ Classic call interface

☑ Single output/update function ☐ Terminate function required

Generate preprocessor conditionals: Use local settings

☑ Suppress error status in real-time model data structure ☑ Combine signal/state structures

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

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

The software generates this code based on the following assumptions:

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

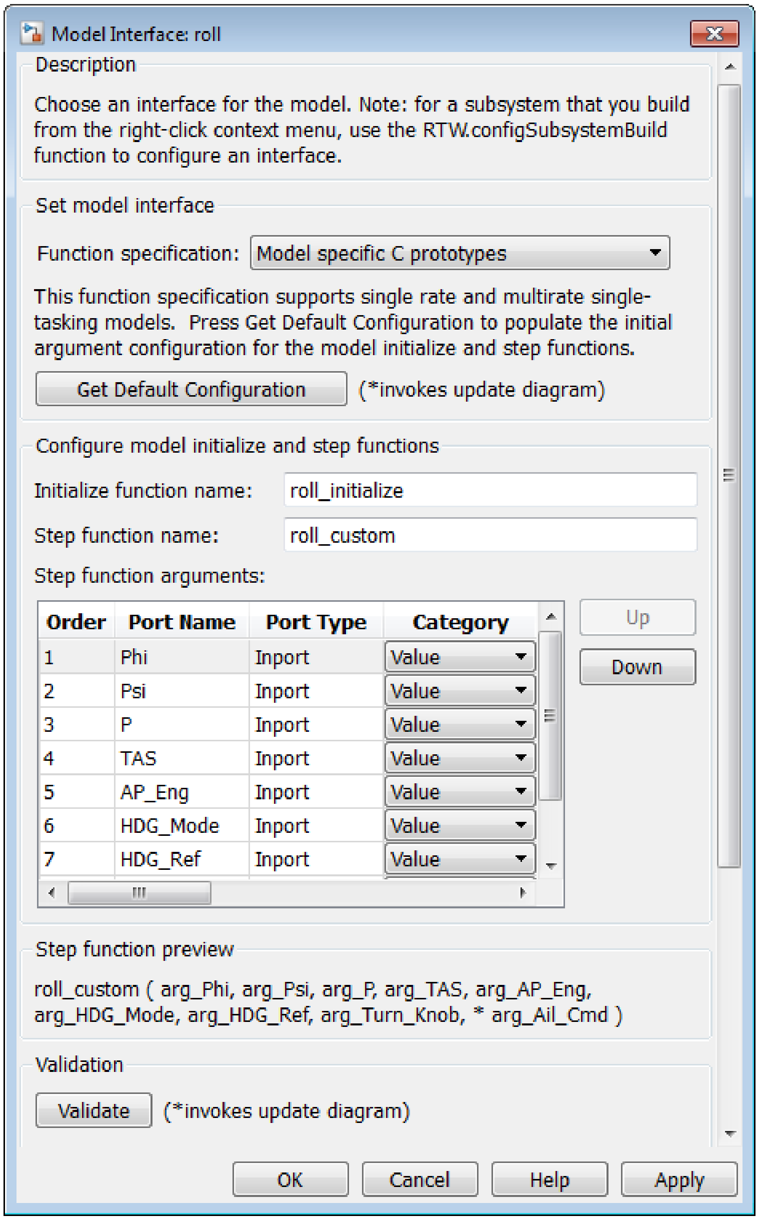
## Configure Model Initialize and Step Functions

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

1 To open the Model Interface dialog box, on the **Interface** pane, click **Configure Model Functions**.

2 In the Model Interface dialog box, specify **Function specification** as `Model specific C prototypes`.

3 Click **Get Default Configuration**. The Model Interface dialog box expands to display the **Configure model initialize and step functions** parameters.



Model Interface: roll

Description

Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.

Set model interface

Function specification: Model specific C prototypes

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.

Get Default Configuration (*invokes update diagram)

Configure model initialize and step functions

Initialize function name: roll_initialize

Step function name: roll_custom

Step function arguments:

| Order | Port Name | Port Type | Category |
|---|---|---|---|
| 1 | Phi | Inport | Value |
| 2 | Psi | Inport | Value |
| 3 | P | Inport | Value |
| 4 | TAS | Inport | Value |
| 5 | AP_Eng | Inport | Value |
| 6 | HDG_Mode | Inport | Value |
| 7 | HDG_Ref | Inport | Value |

Step function preview

roll_custom ( arg_Phi, arg_Psi, arg_P, arg_TAS, arg_AP_Eng, arg_HDG_Mode, arg_HDG_Ref, arg_Turn_Knob, * arg_Ail_Cmd )

Validation

Validate (*invokes update diagram)

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

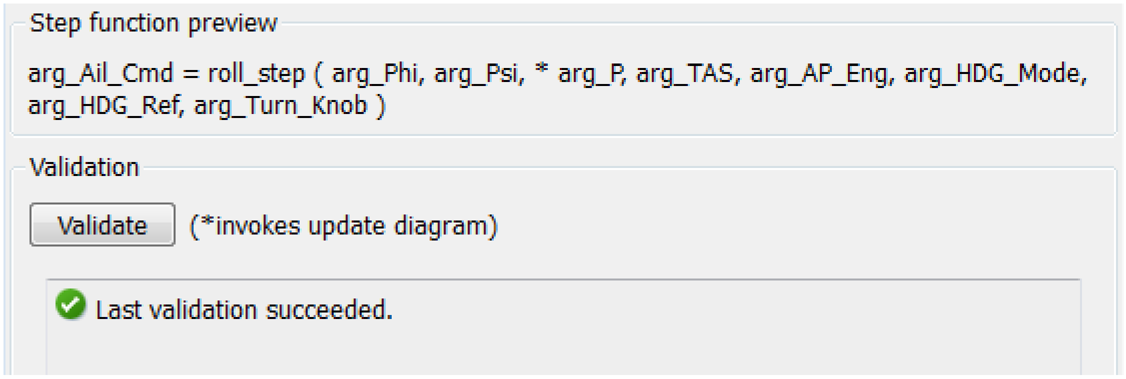
5 Change **Step function name** to `roll_step`.

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

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

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

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


Step function preview

arg_Ail_Cmd = roll_step ( arg_Phi, arg_Psi, * arg_P, arg_TAS, arg_AP_Eng, arg_HDG_Mode, arg_HDG_Ref, arg_Turn_Knob )

Validation

Validate (*invokes update diagram)

✓ Last validation succeeded.

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

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

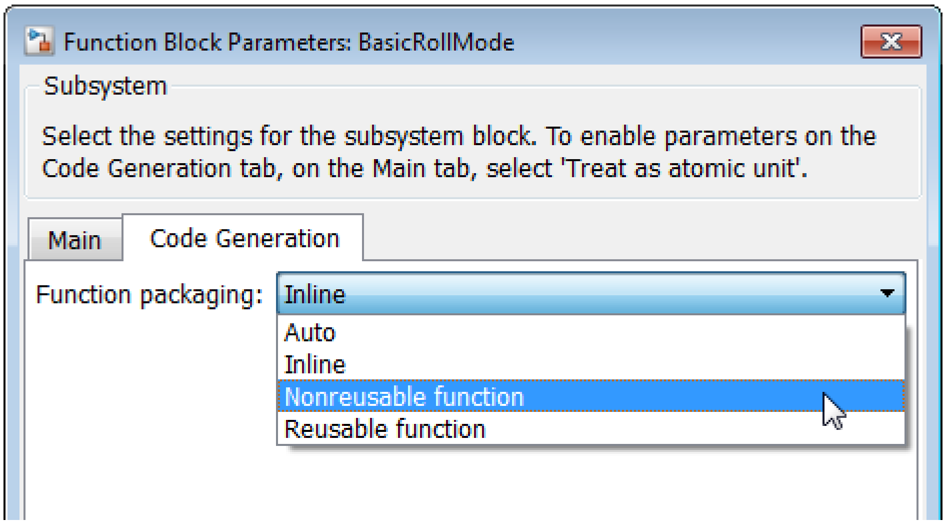
## Subsystem Interface

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

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

2 In the Function Block Parameters dialog box, click the **Code Generation** tab. The **Function packaging** parameter `Auto` option instructs the software to use its heuristic to implement the system efficiently, based on its usage in the model. Otherwise, you can specify the function

implementation based on criteria that you are using for execution speed and memory utilization.


Function Block Parameters: BasicRollMode

Subsystem

Select the settings for the subsystem block. To enable parameters on the Code Generation tab, on the Main tab, select 'Treat as atomic unit'.

Function packaging: Inline

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

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

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

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

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

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

## Customize File Packaging

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

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

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

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

This example showed how to configure the function interfaces for a model and a subsystem. The example showed how to configure your model to modularize the generated code into different file packaging formats. The next example shows how to set up your model to specify how data appears in the generated code. For more information, see "Define Data in the Generated Code" on page 3-37.

# Define Data in the Generated Code

**In this section...**

To control how model data is represented in the generated code, you can specify Simulink data objects. With data objects, you can create variables in the base workspace that allow you to parameterize the specification of model data, such as signals and parameters. You can use the variables to specify parameter and signal attributes. These variables also determine how each parameter and signal is defined and declared in the generated code.

This example uses the configured model `roll` from the example, "Generate and Analyze C Code" on page 3-12. For this example, open `rtwdemo_roll_codegen` and save it to a local folder as `roll.slx`.

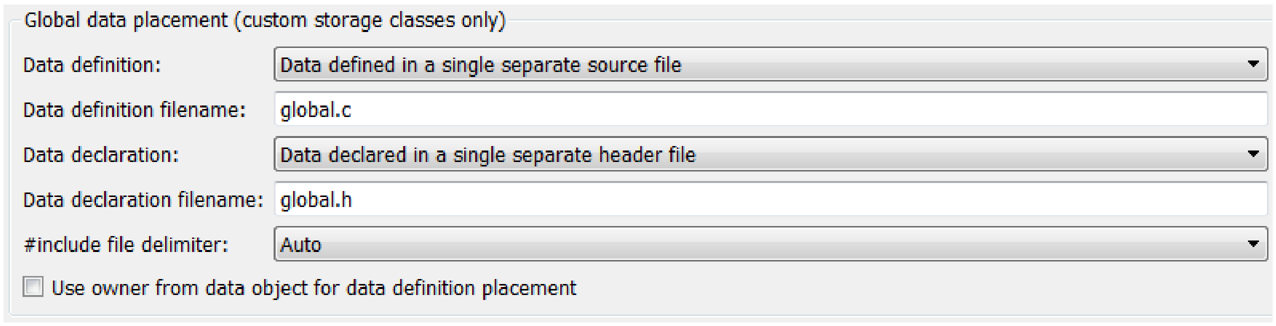
## Control Placement of Data in Generated Files

At the model level, you can control which generated files contain model data definitions and declarations. Separating the data declaration and assignment simplifies the integration of the code into the production environment.

**1** For model `roll`, open the Configuration Parameters dialog box.

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

**3** Set **Data definition** to `Data defined in a single separate source file`. Set **Data declaration** to `Data declared in a single separate header file`.


Global data placement (custom storage classes only)

| | |
|---|---|
| Data definition: | Data defined in a single separate source file |
| Data definition filename: | global.c |
| Data declaration: | Data declared in a single separate header file |
| Data declaration filename: | global.h |
| #include file delimiter: | Auto |

☐ Use owner from data object for data definition placement

4 Click **Apply** and **OK**.

When data objects are defined for this model, code generation places code in `global.c` and `global.h`, which are stored in the `roll_ert_rtw` folder.

## Signal Representation in the Generated Code

### Specify Storage Class for Signal Data

A storage class specification of a signal, parameter, or data object provides you with control over how that entity is declared, stored, and represented in the generated code. As you are preparing to generate code, consider how many data entities in your model that you want to specify as a data object. If there are a large number, you can use the Data Object Wizard to find candidates in your model and create data objects. These steps show this workflow using two signals in the model `roll`.

1 In the Simulink Editor, select **Code > Data Objects > Data Object Wizard**.

2 In the Data Object Wizard, in the Find options group, select all of the check boxes.

3 Click **Find**.

4 Click the check boxes next to `ailCmd` and `phiCmd`.

5 From the **Choose package for selected data objects** drop-down list, verify that `Simulink` is selected.

6. Click **Apply Package** to apply the default Simulink package for the data objects.

7. Click **Create** to create the data objects. When the data objects are created in the base workspace, the signal names no longer appear in the Data Object Wizard.

8. Close the Data Object Wizard.

9. In the Simulink Editor, open the Model Explorer by clicking the icon.

10. In the Model Explorer, on the left pane, in the Model hierarchy tree, click **Base Workspace**. The newly created data objects are now listed in the base workspace (center pane).

The next step is to associate the data objects with the signals.

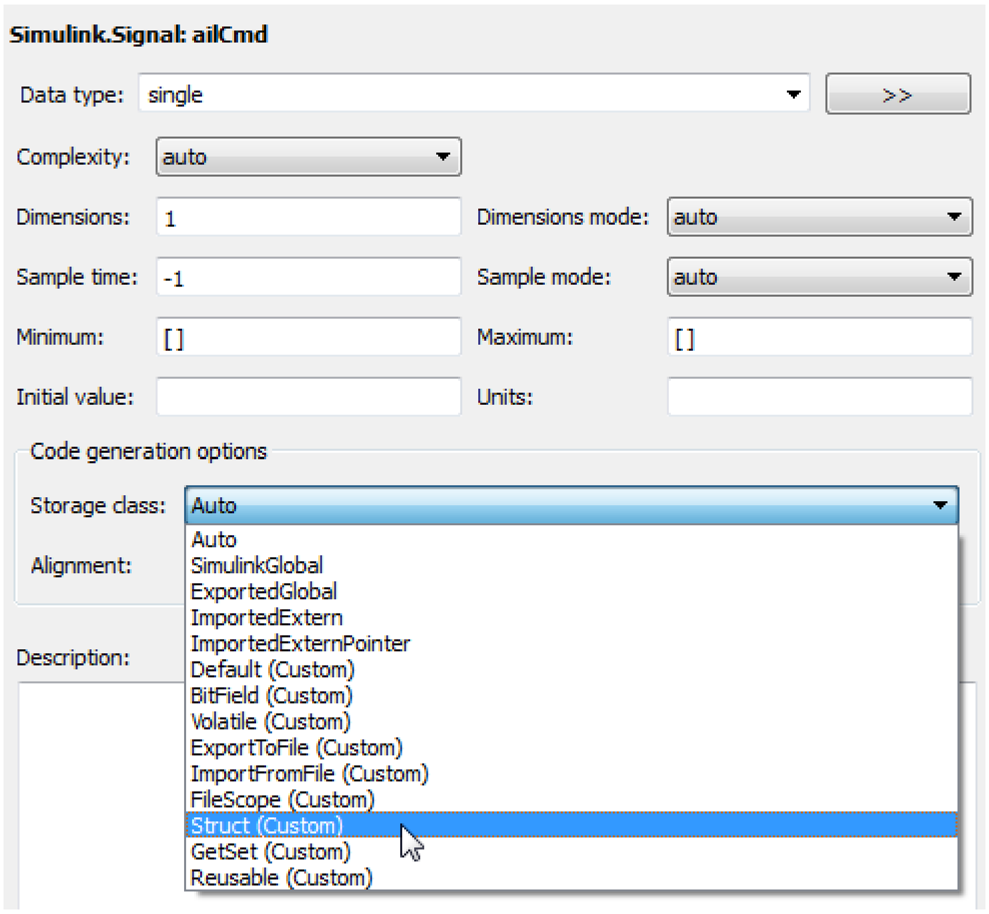
### Configure Signal Data Objects

To configure the new signal data objects `ailCmd` and `phiCmd`, define the data objects such that the code generator places them in a `struct` in the generated code.

1. In the Model Explorer, with the Base Workspace available in the center pane, select `ailCmd`.

   The `Simulink.Signal` properties are displayed in the right pane.

2. Specify the **Storage class** as `Struct (Custom)`.



Simulink.Signal: ailCmd

Data type: single

Complexity: auto

Dimensions: 1

Dimensions mode: auto

Sample time: -1

Sample mode: auto

Minimum: []

Maximum: []

Initial value:

Units:

Code generation options

Storage class: Auto

Alignment:

Description:

3 Specify **StructName** as `signal_data`.

4 Click **Apply**.

5 Repeat steps 1 through 4 for `phiCmd`. `ailCmd` and `phiCmd` use the same **StructName**, `signal_data`.

6 Press **Ctrl+B** to generate code.

Now the variables `ailCmd` and `phiCmd` are defined in a `struct` in `roll_types.h` and used in `roll.c`. The `struct` definition looks similar to the following:

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

## Resolve Signal with Data Object

You must specify that a signal name is associated with a data object in the base workspace. For each signal, `ailCmd` and `phiCmd`, do the following:

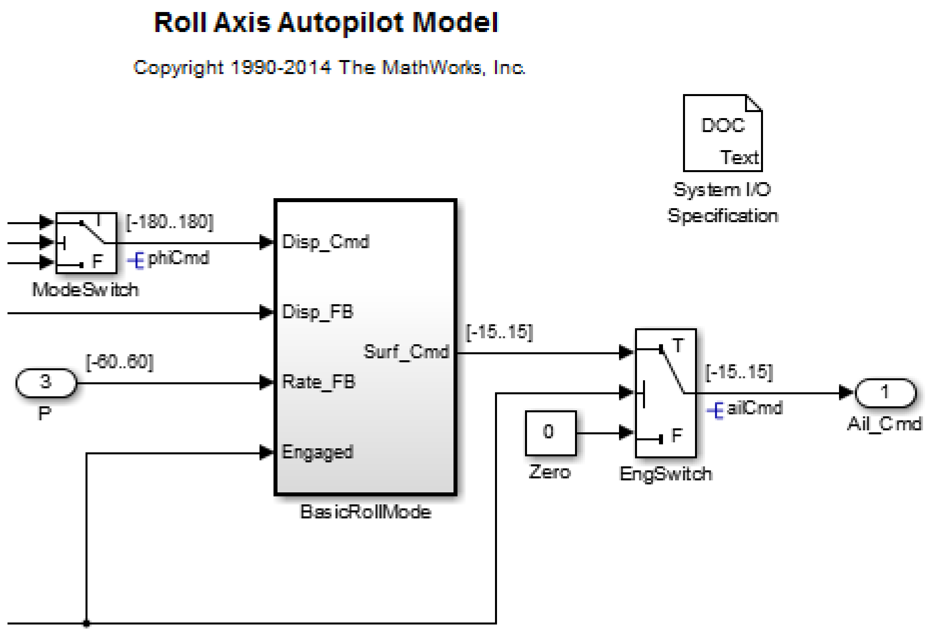
1. In the Simulink Editor, right-click the signal line.

2. From the list, select `Properties`.

3. In the Signal Properties dialog box, observe that the **Signal name must resolve to Simulink signal object** parameter is selected. When the Data Object Wizard created `Simulink.Signal` data objects, the software set this parameter.

4. Click **OK**.

Notice the signal object icon on the signal. This icon is another indication that the signal is associated with a `Simulink.Signal` object in the base workspace.



## Parameter Representation in the Generated Code

An alternative workflow to using the Data Object Wizard is to add objects to the base workspace in the Model Explorer. You can add objects for both signals and parameters. This example shows how to create `Simulink.Parameter` objects for constant values in the model `roll`.

1. In the Simulink Editor, double-click the subsystem `RollAngleReference`. Locate the Constant blocks `UpThr` and `LoThr`.

2. For each Constant block:

   a. Right-click the block and select `Block Parameters (Constant)` from the list.

   b. In the Block Parameter dialog box, replace the numeric **Constant value** with the corresponding variable name: `upThr` for `6`, or `loThr` for `-6`.

   c. Click **Apply** and **OK**.

3 In the Model Explorer, in the left navigation pane, select **Base Workspace**.

4 From the menu bar, select **Add > Simulink.Parameter**. This action creates a `Simulink.Parameter` object with the default name `Param` in the center pane.

5 In the center pane, click the name of the object and rename it to `upThr`. Press **Enter**. The data object properties for `upThr` are displayed in the right pane.

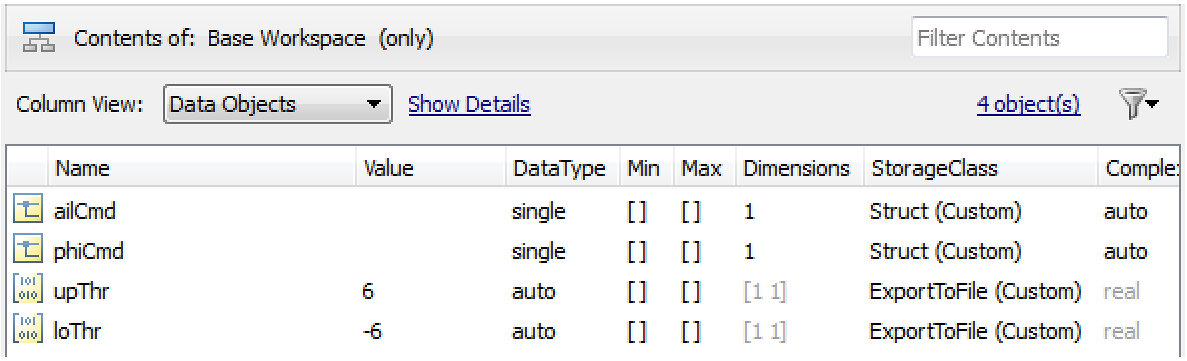
6 In the right pane, specify the properties as follows:

- **Value** is `6`
- **Storage class** is `ExportTofile (Custom)`
- **HeaderFile** is specified as `threshold`

7 Click **Apply**.

8 Repeat steps `4` through `7` for `loThr`, using `-6` as the **Value**.

In the Model Explorer, this graphic is how the base workspace appears.



| Name | Value | DataType | Min | Max | Dimensions | StorageClass | Comple |
|---|---|---|---|---|---|---|---|
| ailCmd | | single | [] | [] | 1 | Struct (Custom) | auto |
| phiCmd | | single | [] | [] | 1 | Struct (Custom) | auto |
| upThr | 6 | auto | [] | [] | [1 1] | ExportToFile (Custom) | real |
| loThr | -6 | auto | [] | [] | [1 1] | ExportToFile (Custom) | real |

From the Simulink Editor, press **Ctrl+B** to generate code. In the code generation report, select `threshold.h`. In the file, find the data declarations for `upThr` and `loThr`.

For further customization, you can define your own custom storage classes tailored to your code generation requirements. Use the Custom Storage Class Designer (`cscdesigner`) tool to design your own custom storage classes.

## Save Data Objects

Data objects stored in the base workspace are not saved with the model. You can save data objects to a MAT-file or a MATLAB-file. Whenever you open your model, you must import the data file or set up your model to use the `PreLoadFcn` callback to populate the base workspace.

### Save Workspace Variables as a MATLAB-file

1 In the base workspace, select all of the data objects so that they are highlighted.

2 Right-click the highlighted variables and select `Export Selected`.

3 In the Export to File dialog box, enter the **File name** `roll_data` and specify **Save as type** (on Linux, **Files of Type**) as `MATLAB-files(*.m)` from the list.

4 Click **Save**.

A MATLAB script is generated. When the script runs, it creates the data objects in the base workspace.

### Add MATLAB-file to a Model Callback

When you have data for a model stored in a file, you can manually import the data file, or set up a model callback so that the data loads when you open a model.

1 For model `roll`, in the Simulink Editor, select **File > Model Properties > Model Properties**.

2 In the Model Properties dialog box, select the **Callbacks** tab.

3 In the left pane, select `PreLoadFcn`.

4 In the **Model pre-load function** text box, type: `roll_data`.

5 Click **Apply** and **OK**.

6 Save the model.

Now, when you load the model `roll`, the MATLAB script, `roll_data.m` runs and creates the data objects in the base workspace.

This example shows how to define data objects to represent signal and parameter data in the generated code. The next example shows how to deploy and verify an executable program for your model. For more information, see "Deploy and Verify Executable Program" on page 3-46.

## Deploy and Verify Executable Program

**In this section...**

To verify code execution results, you can use software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing. A SIL simulation involves compiling and running production source code on your host computer. A PIL simulation involves cross-compiling and running production object code on a target processor or an equivalent instruction set simulator. Embedded Coder automates execution of generated code in Simulink for SIL testing or on the embedded target for PIL testing using Simulink simulation modes or S-function blocks. You can use SIL and PIL simulations to:

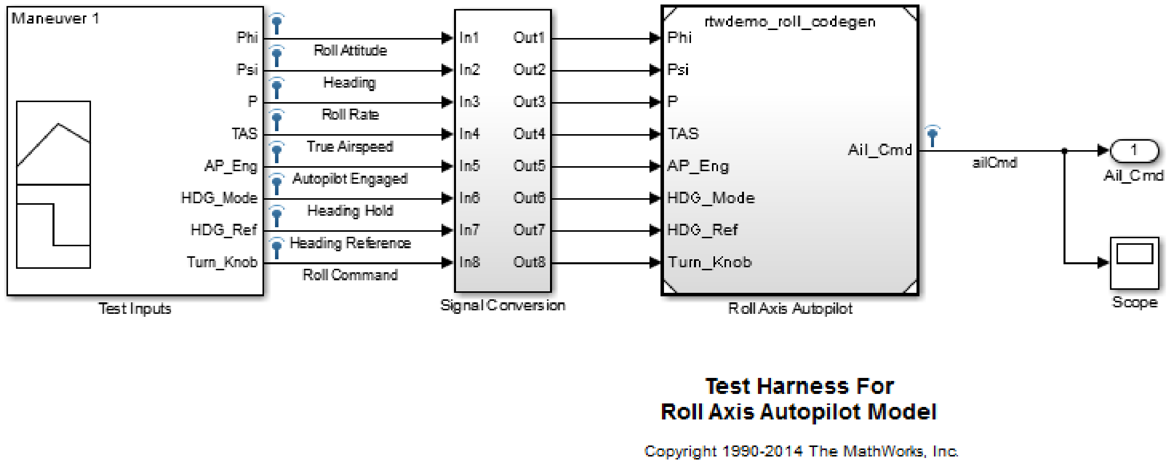
- Verify the numerical behavior of your code.
- Collect code metrics such as code coverage and execution profiling data.
- Optimize your code.
- Progress towards achieving IEC 61508, ISO 26262, or DO-178 certification.

This example simulates a test harness model in normal mode, and then simulates the model in SIL mode. Results are logged to the Simulation Data Inspector. Comparing the results helps you verify that simulating the model produces the same results as executing the generated code.

### Test Harness Model

One method for verifying the code generated for a model is to use a test harness model that includes the model to be tested as a referenced model. You can generate a test harness model with Simulink Verification and

Validation software. With a test harness model, you can easily switch the Model block between normal, SIL, or PIL simulation mode. Test vectors provide a baseline for the behavior of the generated code and verify that the model meets requirements. This example uses a test harness model, `rtwdemo_roll_harness`, which generates test inputs to the referenced model, `rtwdemo_roll_codegen`.



When running in SIL simulation mode, the configuration parameters for the top model and the referenced model must match.

1 Open the example models `rtwdemo_roll_harness` and `rtwdemo_roll_codegen`.

2 Save `rtwdemo_roll_harness` as `roll_harness` to a local folder.

3 Open the Configuration Parameters dialog box for `rtwdemo_roll_codegen`.

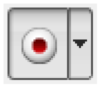
4 On the **Code Generation** pane, verify that the **Generate code only** check box is not selected.

5 If you make changes, click **Apply** and **OK**.

## Simulate in Normal Mode

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

1 To log data to the Simulation Data Inspector for model `roll_harness`, on the Simulink Editor toolbar, click **Record**.



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

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

4 Simulate `roll_harness`.

5 When the simulation is done, view the simulation results in the Simulation Data Inspector. If the tool is not already open, in the Simulink Editor, click the link in the notification bar to open the Simulation Data Inspector.

6 If the **Signal Name** column is not present:

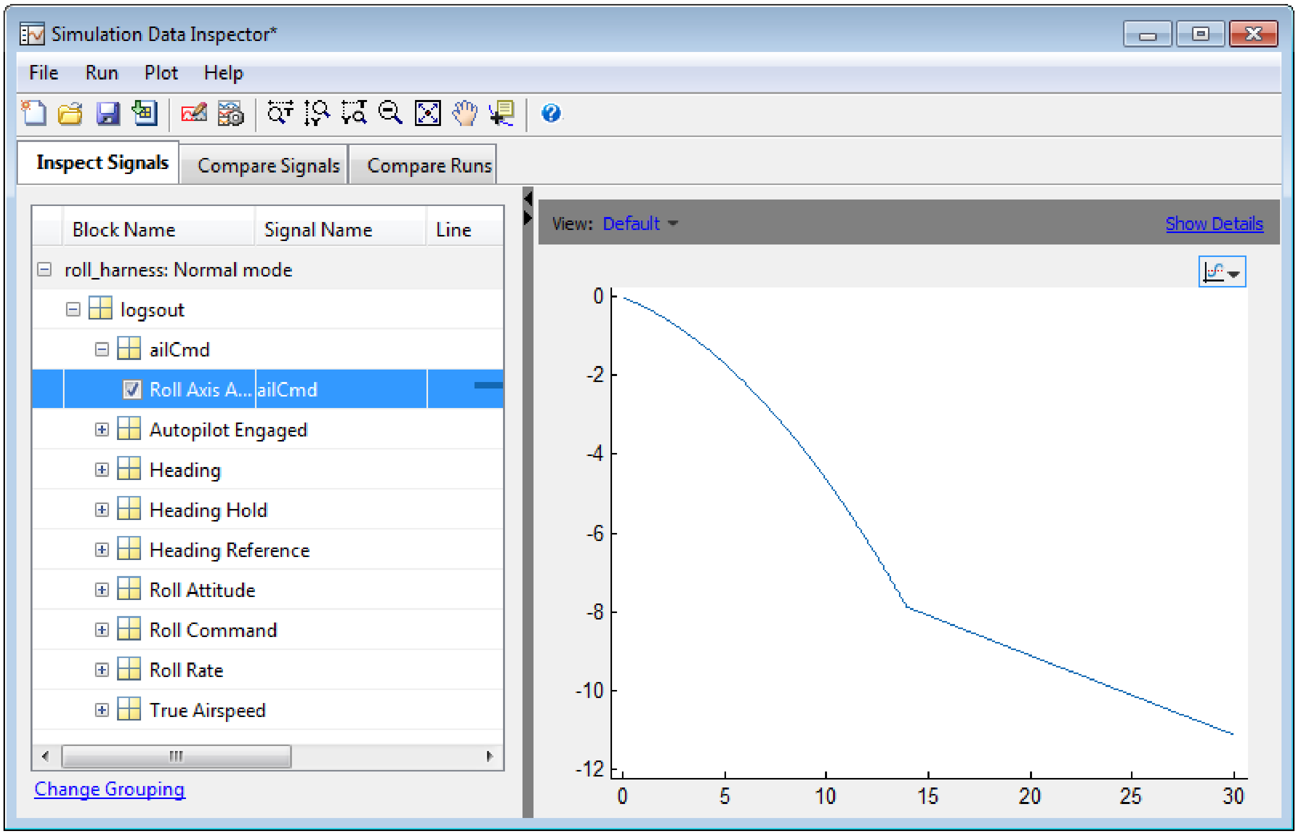
   a Right-click the Signal Browser column titles. From the context menu, add `Signal Name`.

   b Click the **Change Grouping** link below the Signal Browser table. In the Signal Browser, specify the first **Then By** as `Logged Variable`.

   c Click **OK**.

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

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



## Simulate in SIL Mode

In a SIL simulation, code is generated, compiled, and executed on the host computer. Results are logged in the Simulation Data Inspector, in addition to the results from running in normal mode.

1. In the `roll_harness` model window, right-click the `Roll Axis Autopilot` model block and select `Block Parameters (ModelReference)`.
2. In the Block Parameters dialog box, specify **Simulation mode** as `Software-in-the-loop (SIL)`.
3. On the Simulink Editor toolbar, verify that the **Record** button is selected.

4 Simulate the `roll_harness` model.

5 In the Simulation Data Inspector, click the run name field and rename the new run to `roll_harness:  SIL mode`.

6 Expand the run and select `Ail_Cmd` to plot the signal.

## Compare Simulation Results

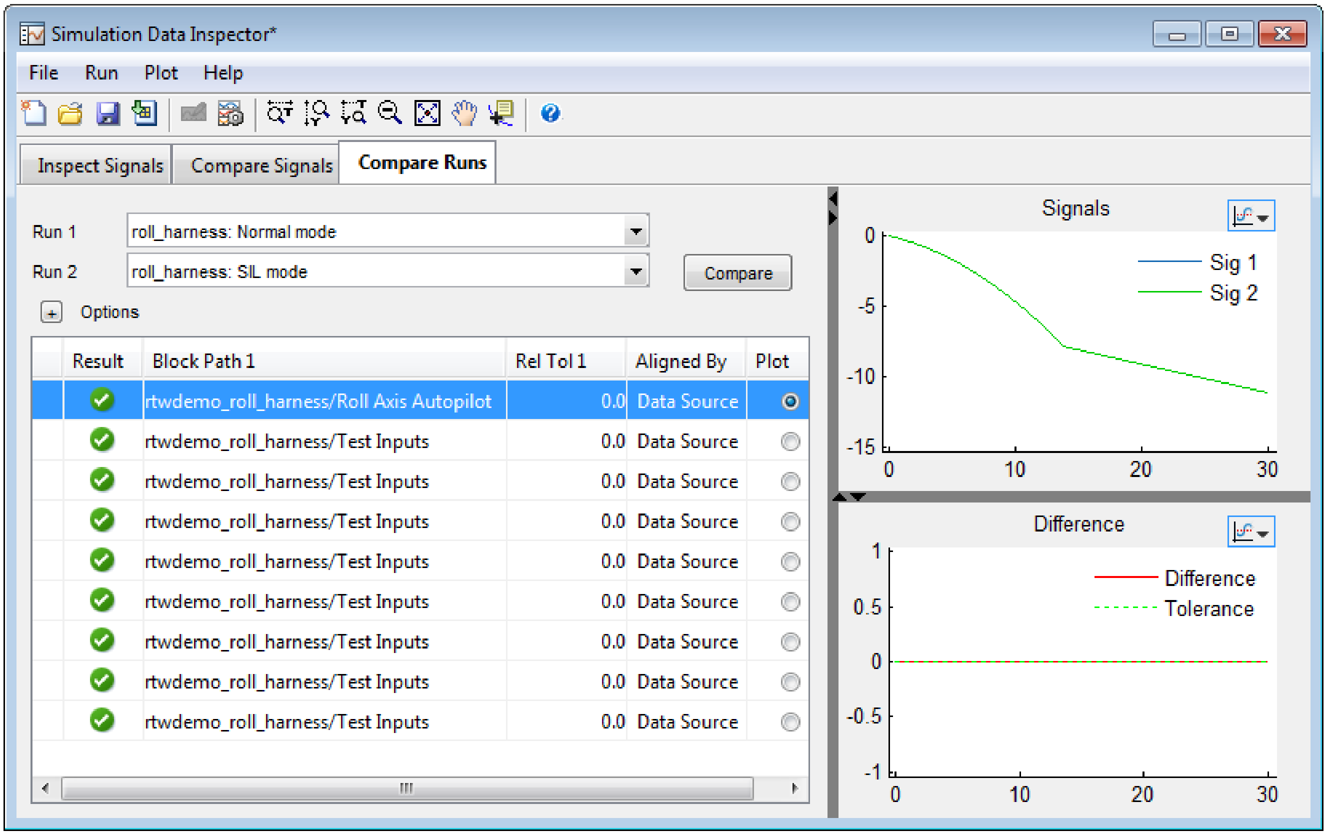
In the Simulation Data Inspector:

1 Click the **Compare Runs** tab.

2 For **Run 1**, select `roll_harness:  Normal mode`. For **Run 2**, select `roll_harness:  SIL mode`.

3 Click **Compare**.

4 Plot the first result, `roll_harness/Roll Axis Autopilot`, by selecting the option button in the Plot column.



Run 1: roll_harness: Normal mode
Run 2: roll_harness: SIL mode

| Result | Block Path 1 | Rel Tol 1 | Aligned By | Plot |
|---|---|---|---|---|
| ✓ | rtwdemo_roll_harness/Roll Axis Autopilot | 0.0 | Data Source | ● |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |
| ✓ | rtwdemo_roll_harness/Test Inputs | 0.0 | Data Source | ○ |

If you find differences in testing results, it is important to investigate and understand the differences. Some differences can be attributed to implementation changes (converting from an interpreted model to an executable implementation), or can be an indication of bugs. You must understand all differences to confirm that the behavior of the system is free of bugs.

## Improve Code Performance

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

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

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

## More Information About Code Generation in Model-Based Design

The following table includes links to additional information for generating, executing, and verifying production code for your model.

| To | See |
|---|---|
| Consider model design and configuration for code generation | "Model Architecture and Design" |
| Generate code variants using macros for preprocessor compilation from models | "Variant Systems" |
| Achieve code reuse | "Subsystems" and "Referenced Models" |
| Define and control model data | "Data Representation " |
| Control generation of function and class interfaces | "Function and Class Interfaces" |
| Control naming and partitioning of data and code across generated source files | "File Packaging " |
| Select and configure the target environment for your application | "Target" |
| Configure model for code generation objectives, such as efficiency or safety | "Application Objectives" |

| To | See |
|---|---|
| Configure parameters for specifying identifier names, code comments, style, and templates | “Code Appearance” |
| Export component XML description and C code for AUTOSAR run-time environment | “AUTOSAR Code Generation” |
| Deployment of standalone programs to target hardware | “Standalone Programs” |
| Choose and apply integration paths and methods | “External Code Integration” |
| Improve code performance, such as memory usage and execution speed | “Performance” |
| Collect code coverage metrics for generated code during SIL or PIL simulation, using a third-party tool to measure test completeness | “Code Coverage” |
| SIL and PIL testing | “Software-in-the-Loop (SIL) Simulation” and “Processor-in-the-Loop (PIL) Simulation” |
| View and analyze execution profiles of code sections | “Code Execution Profiling” |

## Embedded Coder Documentation

The Embedded Coder documentation is divided into two categories:

- **Code Generation from MATLAB Code**

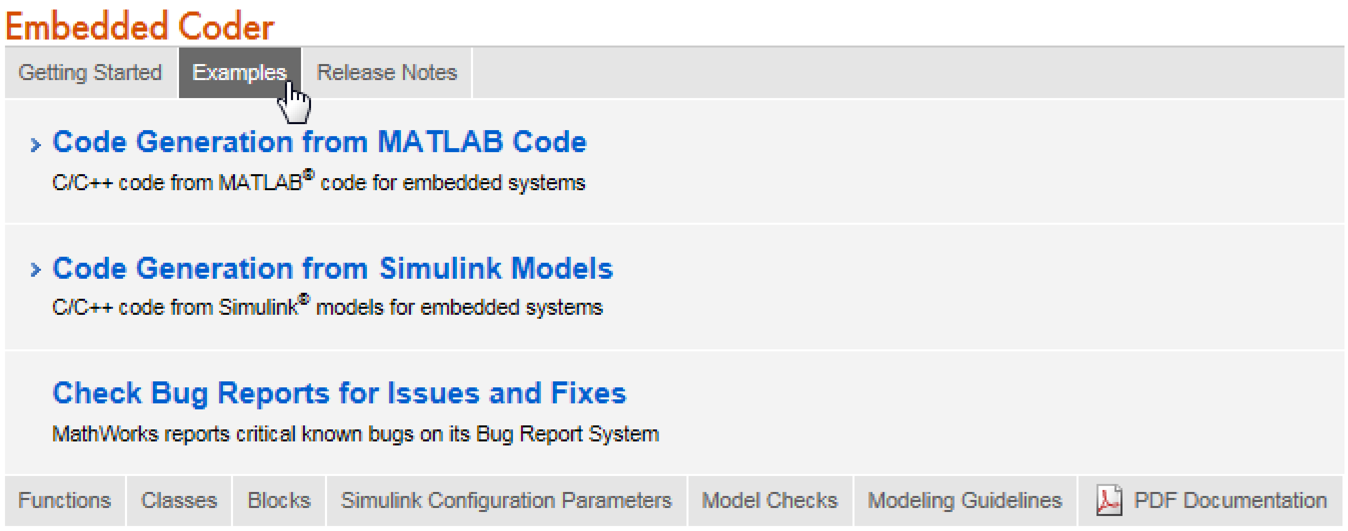
  Provides information on generating C and C++ code from MATLAB code. This documentation is targeted to customers who are not generating code from Simulink models and do not have the Simulink and Simulink Coder products. To get started generating C and C++ code from MATLAB code, see "Generate C Code from MATLAB Code" on page 2-2.

- **Code Generation from Simulink Models**

  Provides information on generating C and C++ code from Simulink models. This documentation is targeted for customers who are generating code from Simulink models and have the Simulink and Simulink Coder products. Typically, these customers generate C and C++ code from MATLAB code by using the MATLAB Function block in a Simulink model. To get started generating C and C++ code from Simulink models, see "Generate C Code from Simulink Models" on page 3-2.

### Embedded Coder Examples

Simulink Coder and Embedded Coder provide a variety of example models that illustrate code generation tasks for rapid simulation and production code generation. You can access these models from the documentation. At the top of the product landing page, click **Examples**.



Embedded Coder

Getting Started | Examples | Release Notes

› Code Generation from MATLAB Code

C/C++ code from MATLAB® code for embedded systems

› Code Generation from Simulink Models

C/C++ code from Simulink® models for embedded systems

Check Bug Reports for Issues and Fixes

MathWorks reports critical known bugs on its Bug Report System

Functions | Classes | Blocks | Simulink Configuration Parameters | Model Checks | Modeling Guidelines | PDF Documentation

# Installing and Using IDE

## Installing Eclipse IDE and Cygwin Debugger

| In this section... |
|---|
| "Installing the Eclipse IDE" on page A-2 |
| "Installing the Cygwin Debugger" on page A-3 |

### Installing the Eclipse IDE

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

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

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

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

2. From the Eclipse CDT Downloads page (http://www.eclipse.org/cdt/downloads.php), download the Eclipse C/C++ Development Tools (CDT) that is compatible with your installed version of the Eclipse IDE.

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

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

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

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

## Installing the Cygwin Debugger

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

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

**3** Follow the installation procedure:

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

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

**5** Add the folder `c:\cygwin\bin` to your system `Path` variable. For example, on a Windows XP system:

**a** Click **Start > Settings > Control Panel > System > Advanced > Environment Variables**.

**b** Under **System variables**, select the `Path` variable and click **Edit**.

**c** Add `c:\cygwin\bin` to the variable value and click **OK**.

---

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

---

## Integrating and Testing Code with Eclipse IDE

**In this section...**

### About Eclipse

Eclipse (`www.eclipse.org`) is an integrated development environment for developing and debugging embedded software. Cygwin (`www.cygwin.com`) is an environment that is similar to the Linux environment, but runs on Windows and includes the GCC compiler and debugger.

This section contains instructions for using the Eclipse IDE with Cygwin tools to build, run, test, and debug projects that include generated code. There are many other software packages and tools that can work with code generation software to perform similar tasks.

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

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

### Define a New C Project

1. In Eclipse, choose **File > New > C Project**. A C Project dialog box opens.

2. In the C Project dialog box:

a In the **Project name** field, type the project name `throttlecntrl_##` (## is `externenv` or `testcode`) .

b In the **Location** field, specify the location of your build folder (for example, `C:\EclipseProjects\throttlecntrl\externenv`).

c In the **Project type** selection box, select and expand **Makefile project**.

d Click the **Empty Project** node.

e Under **Other Toolchains**, select `Cygwin GCC` .

f Click **Next**. A Select Configurations dialog box opens.

3 In the Select Configurations dialog box, click **Advanced settings**. The Properties dialog box appears.

4 In the Properties dialog box:

a Select **C/C++ Build**.

b Select **Generate Makefiles automatically**.

c Select the **Behavior** tab.

d Select **Build on resource save (Auto build)**.

e Click **Apply** and **OK**.

The Properties box closes.

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

## Configure the Debugger

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

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

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

3 Click **Close**.

## Start the Debugger

To start the debugger:

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

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

3 Specify the location of the project files. The Cygwin debugger creates a virtual drive (for example, `main() at /cygdrive/`) during the build process. To run the debugger, Eclipse remaps the drive or locates your project files. Once Eclipse locates the first file, it automatically finds the remaining files. In the Eclipse window, click **Locate File**. The Open dialog box opens.

   For information on using the **Edit Source Lookup Path** button, see "Set the Cygwin Path" on page A-6.

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

## Set the Cygwin Path

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

To provide the path information:

1 Open the Debug Configurations dialog box by selecting **Run > Debug Configurations > C/C++ Application**.

2 Click the **Source** tab.

3 Click **Add**. The Add Source dialog box opens.

4 Select **Path Mapping** and click **OK**. The Path Mappings dialog box opens.

5 Click **Add**. In the **Compilation path** field, type `\cygdrive\c\`.

6 In the **Local file system path** field, click the **Browse** button, navigate to your `C:\` drive, and click **OK**.

7 Click **Apply**.

8 Click **Close**.

## Debugger Actions and Commands

The following actions and commands are available in the debugger.

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