

Simulink® Coder™
Getting Started Guide



MATLAB® & SIMULINK®

R2023a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Simulink® Coder™ Getting Started Guide

© COPYRIGHT 2011–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

| | | |
|----------------|-------------|---|
| April 2011 | Online only | New for Version 8.0 (Release 2011a) |
| September 2011 | Online only | Revised for Version 8.1 (Release 2011b) |
| March 2012 | Online only | Revised for Version 8.2 (Release 2012a) |
| September 2012 | Online only | Revised for Version 8.3 (Release 2012b) |
| March 2013 | Online only | Revised for Version 8.4 (Release 2013a) |
| September 2013 | Online only | Revised for Version 8.5 (Release 2013b) |
| March 2014 | Online only | Revised for Version 8.6 (Release 2014a) |
| October 2014 | Online only | Revised for Version 8.7 (Release 2014b) |
| March 2015 | Online only | Revised for Version 8.8 (Release 2015a) |
| September 2015 | Online only | Revised for Version 8.9 (Release 2015b) |
| October 2015 | Online only | Rereleased for Version 8.8.1 (Release 2015aSP1) |
| March 2016 | Online only | Revised for Version 8.10 (Release 2016a) |
| September 2016 | Online only | Revised for Version 8.11 (Release 2016b) |
| March 2017 | Online only | Revised for Version 8.12 (Release 2017a) |
| September 2017 | Online only | Revised for Version 8.13 (Release 2017b) |
| March 2018 | Online only | Revised for Version 8.14 (Release 2018a) |
| September 2018 | Online only | Revised for Version 9.0 (Release 2018b) |
| March 2019 | Online only | Revised for Version 9.1 (Release 2019a) |
| September 2019 | Online only | Revised for Version 9.2 (Release 2019b) |
| March 2020 | Online only | Revised for Version 9.3 (Release 2020a) |
| September 2020 | Online only | Revised for Version 9.4 (Release 2020b) |
| March 2021 | Online only | Revised for Version 9.5 (Release 2021a) |
| September 2021 | Online only | Revised for Version 9.6 (Release 2021b) |
| March 2022 | Online only | Revised for Version 9.7 (Release 2022a) |
| September 2022 | Online only | Revised for Version 9.8 (Release 2022b) |
| March 2023 | Online only | Revised for Version 9.9 (Release 2023a) |

Check Bug Reports for Issues and Fixes

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

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

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

| | | |
|---|---|------------|
| 1 | Product Overview | |
| | Simulink Coder Product Description | 1-2 |
| | Key Features | 1-2 |
| | Code Generation Using Simulink Coder | 1-3 |
| | Code Generation Technology | 1-3 |
| | Code Generation with Simulink Coder | 1-3 |
| | Validation and Verification for System Development | 1-6 |
| | V-Model for System Development | 1-6 |
| | Types of Simulation and Prototyping in the V-Model | 1-6 |
| | Target Environments and Applications | 1-8 |
| | About Target Environments | 1-8 |
| Types of Target Environments | 1-8 | |
| Applications of Supported Target Environments | 1-10 | |
| Code Generation Goal Summary | 1-13 | |
| Customize Code Generation Environment | 1-15 | |

| | | |
|--|---|-------------|
| 2 | Getting Started Tutorials | |
| | Generate C Code for a Model | 2-2 |
| | Generate C Code by Using Simulink Coder Quick Start Tool | 2-4 |
| | Generate Code with Quick Start Tool | 2-4 |
| | Inspect the Generated Code | 2-4 |
| | Verify Generated Executable Program Results | 2-6 |
| | Configure Model for Verification | 2-6 |
| | Simulate Model and View Results | 2-7 |
| | Build and Run Executable Program and View Results | 2-8 |
| | Compare Simulation and Executable Program Results | 2-8 |
| | Tune Parameter During Program Execution | 2-10 |
| Configure Data Accessibility and Communication Channel | 2-10 | |
| Build and Run Executable Program | 2-10 | |
| Tune Parameter and Observe Results | 2-10 | |

| | |
|--|-------------|
| Deploy Prototype Code and Artifacts | 2-12 |
| Package Generated Code and Artifacts in a Zip File | 2-12 |
| Explore Other Options | 2-12 |

Product Overview

- “Simulink Coder Product Description” on page 1-2
- “Code Generation Using Simulink Coder” on page 1-3
- “Validation and Verification for System Development” on page 1-6
- “Target Environments and Applications” on page 1-8
- “Code Generation Goal Summary” on page 1-13
- “Customize Code Generation Environment” on page 1-15

Simulink Coder Product Description

Generate C and C++ code from Simulink and Stateflow models

Simulink Coder (formerly Real-Time Workshop®) generates and executes C and C++ code from Simulink models, Stateflow® charts, and MATLAB® functions. The generated source code can be used for real-time and non-real-time applications, including simulation acceleration, rapid prototyping, and hardware-in-the-loop testing. You can tune and monitor the generated code using Simulink or run and interact with the code outside MATLAB and Simulink.

Key Features

- ANSI/ISO C and C++ code and executables for discrete, continuous, or hybrid Simulink and Stateflow models
- Integer, floating-point, and fixed-point data types using row- and column-major layout
- Code generation for single-rate, multirate, and asynchronous models
- Single-task, multitask, and multicore code execution with or without an RTOS
- External mode simulation for parameter tuning and signal monitoring using XCP, TCP/IP, and serial communication protocols
- Incremental and parallel code generation builds for large models

Code Generation Using Simulink Coder

In this section...

“Code Generation Technology” on page 1-3

“Code Generation with Simulink Coder” on page 1-3

Code Generation Technology

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

Code generation technology and related products provide tooling that you can apply to the V-model for system development. The V-model is a representation of system development that highlights verification and validation steps in the development process. For more information, see “Validation and Verification for System Development” on page 1-6.

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

Code Generation with Simulink Coder

Use Simulink Coder to generate C or C++ source code for:

- Problem-based learning
- Rapid prototyping and real-time simulation for proof-of-concept development, testing, and optimization
- Development of rapid prototyping and hardware-in-the-loop (HIL) turnkey solutions
- Application binaries that run on desktop platforms
- Development of model designs and generated code that provide intellectual property protection

“Rapid Prototyping and Real-Time Simulation” and “Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation” introduce the use of Simulink Coder for deploying algorithm models for rapid prototyping and deploying environment models for real-time HIL simulation.

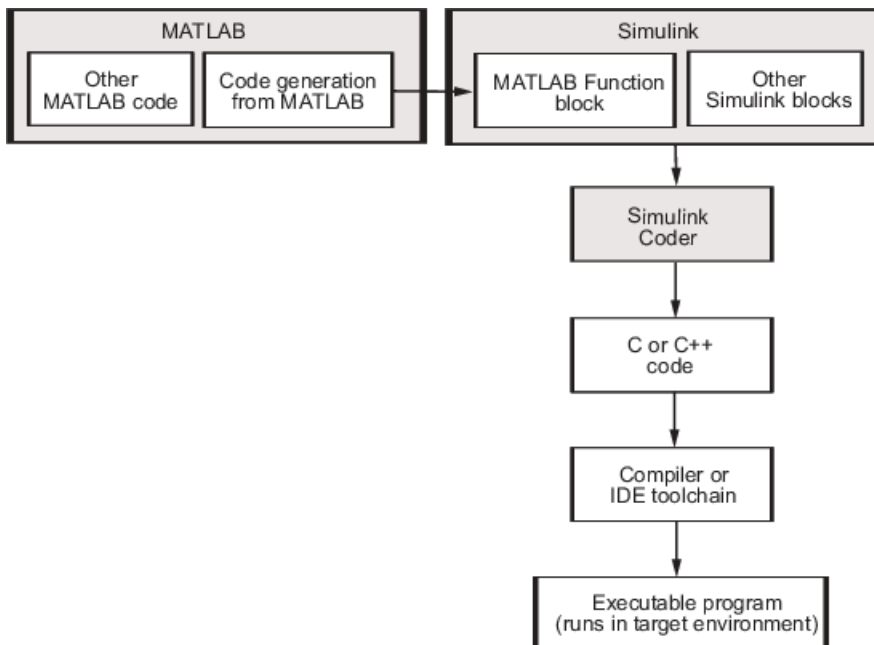
For information on developing rapid prototyping and HIL turnkey solutions, see “Code and Tool Customization”, Simulink Real-Time™, and Simulink Desktop Real-Time™.

“Model Protection” provides information on developing models and generated code that provide intellectual property protection.

To generate and deploy code for embedded processors used in mass production, use Embedded Coder®. Embedded Coder extends capabilities of Simulink Coder, including, but not limited to:

- Generation of function code modules that you can integrate with external code
- Advanced code optimizations
- Control over code appearance, including code interfaces
- Variety of tools for verification, testing, and certification

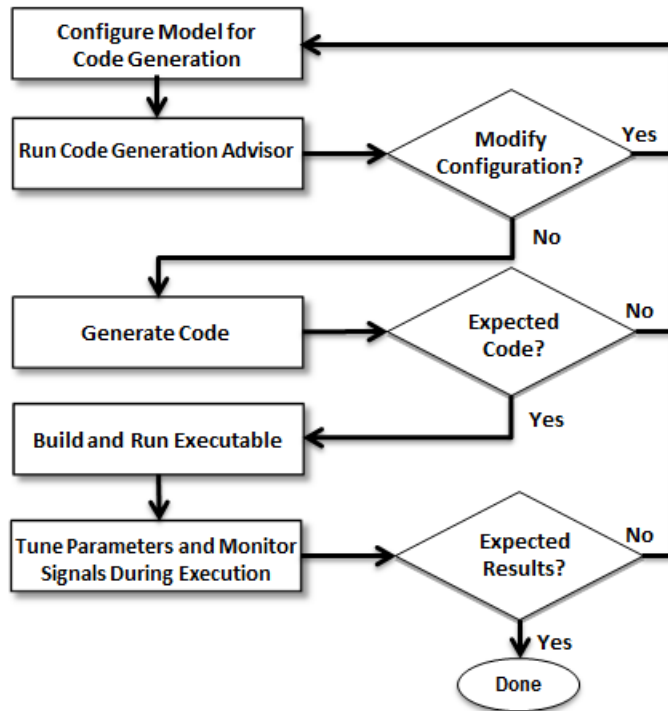
You can generate code for most Simulink blocks and many MathWorks products as “Use Products and Blocks Supported for Code Generation” shows. This figure shows how to use Simulink Coder to generate and deploy code.



The code generation workflow is a part of the V-model on page 1-6 for system development. The process includes model design, code generation, code verification, and testing of the executable program in real-time. For rapid prototyping of a real-time application, typical tasks are:

- Configure the model for code generation in the model configuration set.
- Check the model configuration for execution efficiency using the Code Generation Advisor.
- Generate and view the C code.
- Create and run the executable of the generated code.
- Verify the execution results.
- Build the target executable.
- Run the external model target program.
- Connect Simulink to the external process for testing.
- Use signal monitoring and parameter tuning to further test your program.

Here is a typical workflow for applying the software to the application development process.



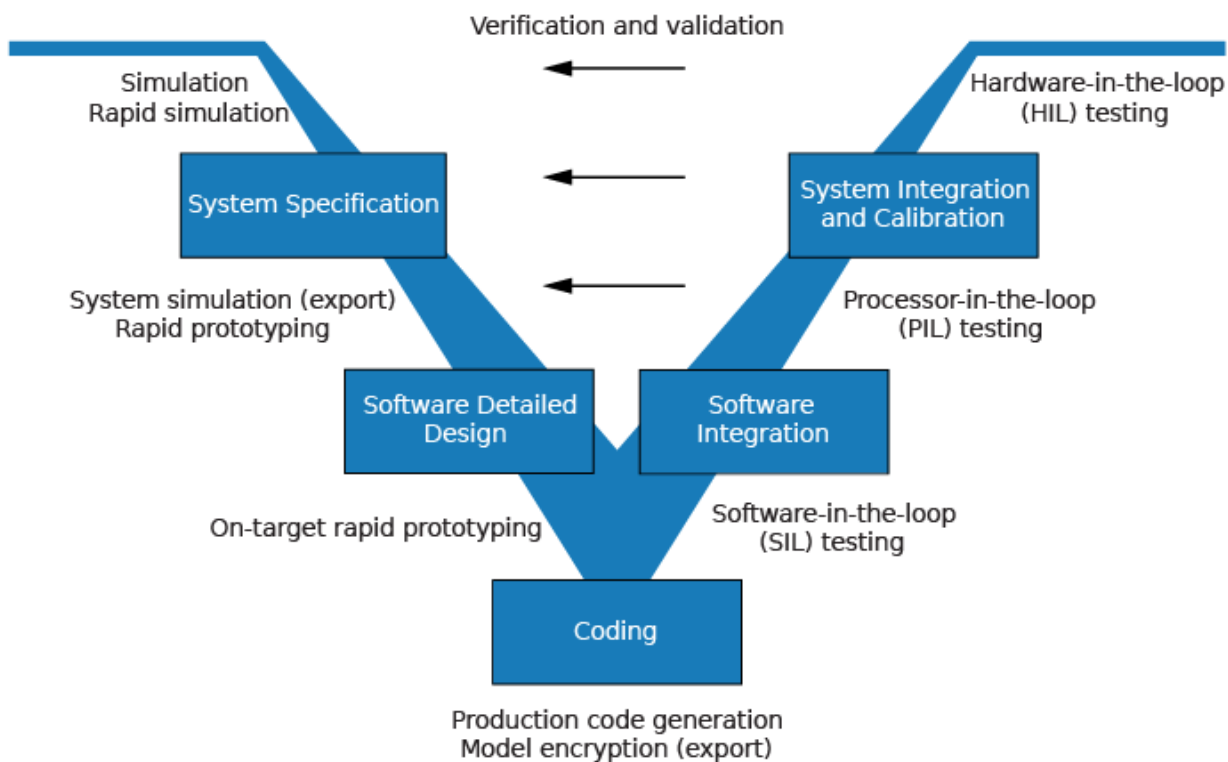
For more information on how to perform these tasks, see “Generate C Code for a Model” on page 2-2.

Validation and Verification for System Development

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

V-Model for System Development

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



Depending on your application and its role in the process, you might focus on one or more of the steps called out in the V-model or repeat steps at several stages of the V-model. Code generation technology and related products provide tooling that you can apply to the V-model for system development. For more information about how you can apply MathWorks code generation technology and related products to the V-model process, see “Types of Simulation and Prototyping in the V-Model” on page 1-6.

Types of Simulation and Prototyping in the V-Model

Use the V-model for system development for different types of simulation and prototyping, such as rapid simulation, system simulation, rapid prototyping, and rapid prototyping on target hardware. This table compares the types of simulation and prototyping identified on the left side of the V-model diagram shown in “V-Model for System Development” (Embedded Coder).

| | Simulation | Rapid Simulation | System Simulation, Rapid Prototyping | Rapid Prototyping on Target Hardware |
|--|--|---|---|---|
| Purpose | Test and validate functionality of concept model | Refine, test, and validate functionality of concept model in nonreal time | Test new ideas and research | Refine and calibrate design during development process |
| Execution hardware | Development computer | Development computer Standalone executable runs outside of MATLAB and Simulink environments | PC or nontarget hardware | Embedded computing unit (ECU) or near-production hardware |
| Code efficiency and I/O latency | Not applicable | Not applicable | Less emphasis on code efficiency and I/O latency | More emphasis on code efficiency and I/O latency |
| Ease of use and cost | Can simulate component (algorithm or controller) and environment (or plant) Normal mode simulation in Simulink enables you to access, display, and tune data during verification Can accelerate Simulink simulations | Easy to simulate models of hybrid dynamic systems that include components and environment models Ideal for batch or Monte Carlo simulations Can repeat simulations with varying data sets, interactively or programmatically by using scripts, without rebuilding the model Can connect to Simulink to monitor signals and tune parameters | Might require custom real-time simulators and hardware Might be done with inexpensive, off-the-shelf PC hardware and I/O cards | Might use existing hardware for less expense and more convenience |

Target Environments and Applications

In this section...

“About Target Environments” on page 1-8

“Types of Target Environments” on page 1-8

“Applications of Supported Target Environments” on page 1-10

About Target Environments

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

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

Types of Target Environments

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

| Target Environment | Description |
|----------------------|---|
| Development computer | The computer that runs MATLAB and Simulink. A development computer is a PC or UNIX ^{®a} environment that uses a non-real-time operating system, such as Microsoft [®] Windows [®] or Linux ^{®b} . Non-real-time (general purpose) operating systems are nondeterministic. For example, those operating systems might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Therefore, the executable for your generated code might run faster or slower than the sample rates that you specified in your model. |

| Target Environment | Description |
|-------------------------|---|
| Real-time simulator | <p>A different computer from the development computer. A real-time simulator can be a PC or UNIX environment that uses a real-time operating system (RTOS), such as:</p> <ul style="list-style-type: none"> • Simulink Real-Time system • A real-time Linux system • A Versa Module Eurocard (VME) chassis with PowerPC™ processors running a commercial RTOS <p>The generated code runs in real time. The exact nature of execution varies based on the particular behavior of the system hardware and RTOS.</p> <p>A real-time simulator connects to a development computer for data logging, interactive parameter tuning, and Monte Carlo batch execution studies.</p> |
| Embedded microprocessor | <p>A computer that you eventually disconnect from a development computer and run as a standalone computer as part of an electronics-based product. Embedded microprocessors range in price and performance, from high-end digital signal processors (DSPs) to process communication signals to inexpensive 8-bit fixed-point microcontrollers in mass production (for example, electronic parts produced in the millions of units). Embedded microprocessors can:</p> <ul style="list-style-type: none"> • Use a full-featured RTOS • Be driven by basic interrupts • Use rate monotonic scheduling provided with code generation |

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

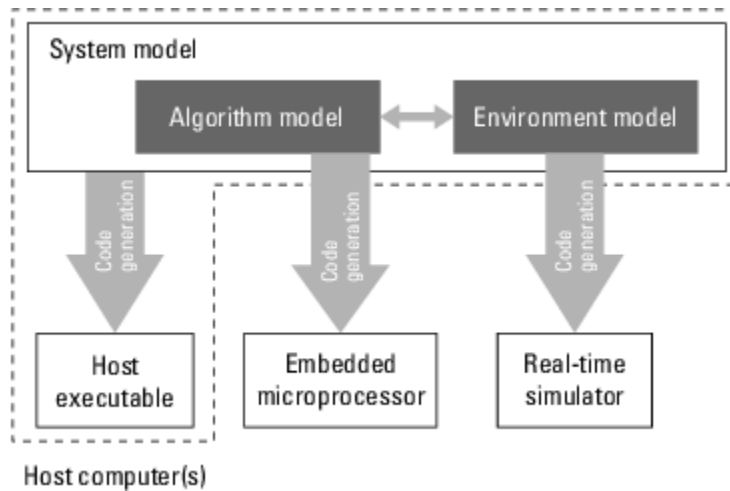
b Linux is a registered trademark of Linus Torvalds.

A target environment can:

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

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

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



Applications of Supported Target Environments

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

| Application | Description |
|--|--|
| Development Computer | |
| "Acceleration" | Techniques to speed up the execution of model simulation in the context of the MATLAB and Simulink environments. Accelerated simulations are especially useful when run time is long compared to the time associated with compilation and checking whether the target is up to date. |
| Rapid Simulation | Execute code generated for a model in non-real-time on the development computer, but outside the context of the MATLAB and Simulink environments. |
| Shared Object Libraries (Embedded Coder) | Integrate components into a larger system. You provide generated source code and related dependencies for building a system in another environment or in a shared library to which other code can dynamically link. |
| "Protect Models to Conceal Contents" | Generate a protected model for use by a third-party vendor in another Simulink simulation environment. |
| Real-Time Simulator | |

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

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

Code Generation Goal Summary

Use this table to find Simulink Coder Embedded Coder information based on your project goals.

| Goals | Related Information |
|--|---|
| Get familiar with the product by using tutorials and learning about concepts. | "Get Started with Simulink Coder" |
| Avoid or minimize project rework by considering code generation from the beginning of the model design process <ul style="list-style-type: none"> • Divide software problem into parts and map aspects of your system architecture to Simulink modeling environment elements. • Model software interfaces . • Modularize design by using Simulink component-based modeling options and reuse techniques. • Import external MATLAB®, C, or C++ code. • Design timer and scheduling support. • Design models based on recommended guidelines. | "Architecture and Component Design" |
| Generate code from model design. <ul style="list-style-type: none"> • Configure code interfaces. • Check whether model is ready for code generation. • Generate source code. • Compile generated source code and link object code. • Generate reports of code generation results. | "Code Generation" |
| Deploy generated code. <ul style="list-style-type: none"> • Adapt a generated program to run without an external real-time executive or operating system on target hardware. • Integrate a generated program with an operating system or board support package. • Relocate static and generated code files for a model to another development environment. • Use a generated shared library in application code. • Package and share artifacts generated for a model with a third party without revealing intellectual property. • Run accelerated simulations and deploy standalone simulations outside of the MATLAB and Simulink environment. | "Deployment, Integration, and Supported Hardware" |

| Goals | Related Information |
|---|-----------------------------------|
| <p>Optimize generated code by minimizing memory requirements and speeding up execution.</p> <ul style="list-style-type: none"> • Optimize use of RAM, ROM, and stack space by generated code. • Improve execution speed of generated code. • Reduce data copies. • Apply code replacement libraries. • Make tradeoffs when generating optimized code. | <p>“Code Efficiency”</p> |
| <p>Customize generated code and code generation tools for project or organization.</p> <ul style="list-style-type: none"> • Customize data representation to support interactivity and instrumentation. • Customize how the code generator produces code. • Define code generation behavior for custom blocks. • Use the Target Language Compiler to customize generated code, inline code generated for custom blocks, and generate additional or different types of files. • Capture target hardware information for MathWorks software. • Provide the code generator with information about a target platform and required code features. • Customize the compilation stage of the build process. • Create software for deploying MATLAB and Simulink designs to hardware platforms. | |
| <p>Verify and test generated code.</p> <ul style="list-style-type: none"> • Monitor signals and tune model parameters in your target application. • Profile code execution to determine whether generated code meets performance requirements. • Test numeric equivalence. • Verify generated code by using code analysis tools. | <p>“Verification and Testing”</p> |

Customize Code Generation Environment

If you are responsible for customizing the code generation environment for a deployment project or for code deployment across an organization, consider these options.

| Customization Goal | Description | More Information |
|------------------------------|--|---|
| Data representation | Customize how the code generator stores data, including matrices and arrays. Control the naming and placement of data types in generated code. | "Data Representation in Generated Code" |
| Model configuration settings | Establish standard model configuration parameter settings that define how the code generator produces and compiles code from models. | "Model Configuration Set Customization" |
| Custom blocks | Define code generation behavior for custom blocks by developing S-functions. | "Code Generation for Custom Blocks" |
| Target Language Compiler | Use the Target Language Compiler to customize generated code, inline code generated for S-function blocks, and generate additional or different types of files. | "Target Language Compiler" |
| Target platform devices | For a target platform device, select a system target file that specifies required target platform environment and code features, use the <code>target</code> package to extend the range of hardware that the code generator supports, and configure the code generator to use code replacement libraries. | "Target Platform Device Customization" |
| Code compilation | Customize the compilation stage of the build process by using a toolchain or template makefile approach. | "Code Compilation Customization" |

| Customization Goal | Description | More Information |
|-------------------------------------|--|---------------------------------------|
| Custom software for target hardware | Create software that supports the deployment of MATLAB and Simulink designs to hardware platforms. Register custom toolchains for the build process, customize XCP server software for target hardware, and create a target hardware application that checks the validity of code generation assumptions. Use the target package to extend the range of supported hardware and provide connectivity between Simulink and your target hardware for external mode and processor-in-the-loop (PIL) simulations. | "Custom Software for Target Hardware" |

Getting Started Tutorials

- “Generate C Code for a Model” on page 2-2
- “Generate C Code by Using Simulink Coder Quick Start Tool” on page 2-4
- “Verify Generated Executable Program Results” on page 2-6
- “Tune Parameter During Program Execution” on page 2-10
- “Deploy Prototype Code and Artifacts” on page 2-12

Generate C Code for a Model

To generate C or C++ code from Simulink® models, Stateflow® charts, and MATLAB® functions, use the Simulink Coder™ product. Use the generated code in applications such as simulation acceleration, rapid prototyping, and hardware-in-the-loop (HIL) simulations.

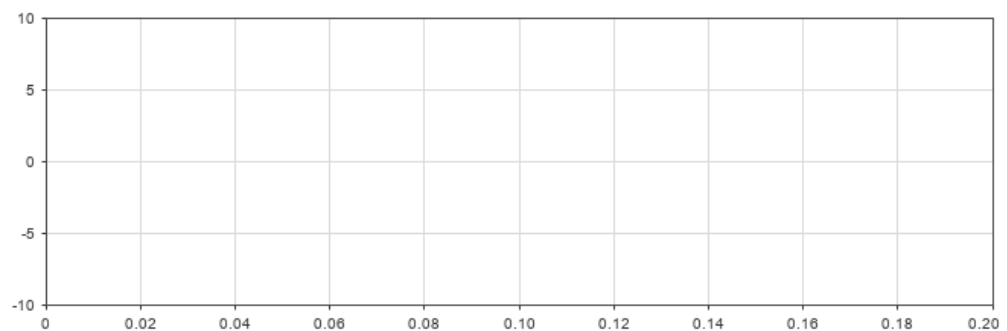
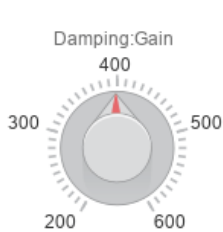
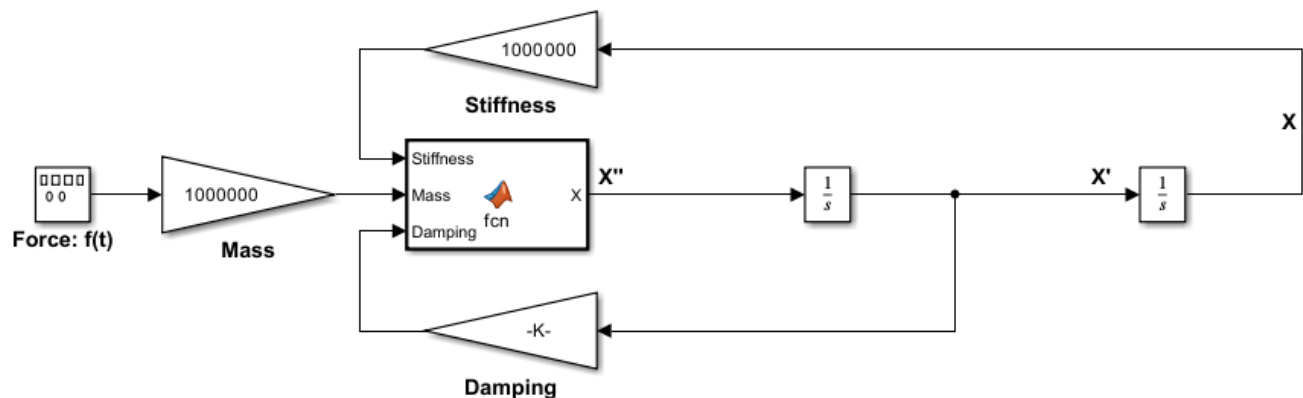
If you are new to Simulink Coder or your application code customization requirements are minimal, you can use graphical tools and default model configuration settings to quickly generate code.

Generating and reviewing code can be as simple as preparing the model for code generation with the Quick Start tool. Then, using code tools accessible from the Simulink Editor, you can configure data interfaces, initiate code generation, and verify the generated code.

This tutorial uses example model `SecondOrderSystem`.

Open the model.

```
open_system("SecondOrderSystem");
```



The model implements a second-order physical system called an ideal mass-spring-damper system. Gain blocks represent components of the system equation: Mass, Stiffness, and Damping. The equation for the system is $mX'' + cX' + kX = f(t)$.

- m = mass of the system (1.0E-6 kg)

- c = damping ratio (4.0e-4 Ns/m)
- k = spring stiffness (1.0 N/m)
- $f(t)$ = forcing function in the x-direction (N)

A Signal Generator block injects a square wave form with an amplitude of 4 and frequency of 20 Hz. The block uses simulation time as the source of values for the waveform time variable. Because the model is configured with a fixed-step solver, which is required for code generation, Simulink uses the same step size for an entire simulation. The consistent step size provides a uniformly sampled representation of the ideal waveform.

The example model shows how you can use MATLAB Function blocks to integrate existing MATLAB function code into Simulink models from which you can generate embeddable C code. The MATLAB function block in the example model integrates a MATLAB function that computes the sum of the component variables.

The Integrator blocks compute integrals of the MATLAB Function block output with respect to time. The solver computes the output of the Integrator block at the current time step, by using the current input value and the value of the state at the previous time step. To support this computational model, the Integrator block saves its output at the current time step for use by the solver to compute its output at the next time step. The block also provides the solver with an initial condition for use in computing the block's initial state at the beginning of a simulation. The default initial condition and the setting for this example model is 0.

The dashboard blocks, Knob and Dashboard Scope, provide visual tooling for tuning the damping and monitoring the waveform. The Knob block is connected to the Damping Gain block. The Dashboard Scope block connects to signals Force: $f(t)$:1 and X.

Use this model to learn how to:

- 1 Generate code by using the Simulink Coder Quick Start tool.
- 2 Verify whether generated executable program results match simulation results.
- 3 Tune a parameter during program execution.
- 4 Deploy prototype code and artifacts.

To start the tutorial, see “Generate C Code by Using Simulink Coder Quick Start Tool” on page 2-4.

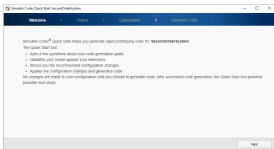
Generate C Code by Using Simulink Coder Quick Start Tool

Prepare model `SecondOrderSystem` for code generation and generate C89/C90 compliant C code by using the Simulink Coder Quick Start tool. Then, inspect the generated code.

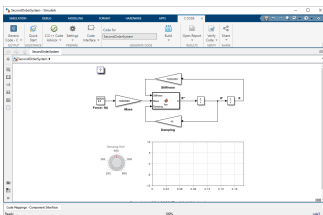
Generate Code with Quick Start Tool

The Quick Start tool chooses fundamental code generation settings based on your goals and application. For example, the Quick Start tool configures the model with a fixed-step solver, which is required for code generation.

- 1 Open model `SecondOrderSystem`, if not already open, by entering the command `openExample("SecondOrderSystem")` in the command Window.
- 2 If the **C Code** tab is not already open, in the Apps gallery, under **Code Generation**, click **Simulink Coder**.
- 3 Open the Simulink Coder Quick Start tool. On the **C Code** tab, click **Quick Start**.



- 4 Advance through the steps of the Quick Start tool. Each step asks questions about the code that you want to generate. For this tutorial, use the default settings. The tool validates your selections against the model and presents the parameter changes required to generate code.
- 5 In the **Generate Code** step, apply the proposed changes and generate code from the model by clicking **Next**.
- 6 Click **Finish**. In the Simulink Editor, return to the **C Code** tab.



Inspect the Generated Code

The code generator creates folder `SecondOrderSystem_grt_rtw` in your current working folder and places source code files in that folder. The generated code is in two primary files: `SecondOrderSystem.c` and `SecondOrderSystem.h`. The file `SecondOrderSystem.c` contains the algorithm code, including the ODE solver code. Model data and entry-point functions are accessible to a caller by including `SecondOrderSystem.h`. The `SecondOrderSystem.h` file includes the extern declarations for block outputs, continuous states, model output, entry points, and timing data.

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

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

- 1 Open the code generation report. In the **C Code** tab, click **Open Report**.
- 2 Open the **Code Interface Report** section. Review the entry-point functions that the code generator produces for the model. For the initialize, execution (step), and terminate functions, the code generator uses these names:
 - `SecondOrderSystem_initialize`
 - `SecondOrderSystem_step`
 - `SecondOrderSystem_terminate`

The functions have a `void-void` interface, which means that they do not pass arguments. The functions gain access to data through shared data structures. Examples of such data include system-level input and output that the functions exchange with application code.

- 3 Review the entry-point functions in the generated code. In the left pane of the code generation report, under **Code**, click file name `SecondOrderSystem.c`. Use the search field to find instances of the string `SecondOrderSystem_step`. Use the arrows to step through each instance. Do the same for the header file `SecondOrderSystem.h`. Then, review code for the initialize and terminate functions.

You can configure code generation customizations, regenerate code, and check results in the code generation report.

Next, verify whether model simulation results match generated executable program results.

Verify Generated Executable Program Results

Verify whether generated executable program results for the model match simulation results.

Configure Model for Verification

- 1 Configure the Dashboard Scope block to monitor the values of signals `Force: f(t):1` and `X`. Double-click the Dashboard Scope block. In the Block Parameters dialog box, confirm that:
 - The block is connected to signals `Force: f(t):1` and `X`. To connect a Dashboard block to a signal, in the model canvas, select the signal. In the Block Parameters dialog box, select the signal name.
 - **Min** is set to `-10`.
 - **Max** is set to `10`.

Apply changes and close the dialog box.

- 2 Configure the Knob block so that you can use the knob to change the value of the damping gain. Double-click the Knob block. In the Block Parameters dialog box, confirm that:
 - The block is connected to parameter `Damping:Gain`. To connect a Dashboard block to a parameter, in the model canvas, select the block that uses the parameter. In the Block Parameters dialog box, select the parameter name.
 - **Minimum** is set to `200`.
 - **Maximum** is set to `600`.
 - **Tick Interval** is set to `100`.

Apply changes and close the dialog box.

- 3 Open the Model Configuration Parameters dialog box. On the **C Code** tab, click **Settings > C/C++ Code generation settings**.
- 4 Configure the model such that Simulink and the generated executable program log workspace data in the Simulation Data Inspector. Click **Data Import/Export**. Confirm that the model is configured with these settings

| Parameter Selected | Name Set To |
|--|-------------|
| Time | tout |
| States | xout |
| Output | yout |
| Signal logging | logsout |
| Data stores | dsmout |
| Record logged workspace data in Simulation Data Inspector | |

- 5 Configure the model for building an executable program. Click **Code Generation**. Confirm that parameter `Generate code only` is cleared.
- 6 Configure and validate the toolchain for building the executable program. Confirm that parameter `Toolchain` is set to `Automatically locate an installed toolchain`.
- 7 Point to the three dots at the bottom of the parameters to reveal the **Advanced parameters** label. Then, under **Advanced parameters** click the **Validate Toolchain** button to generate and

open the Validation Report. The Validation Report indicates which compiler is being used and whether the checks for this compiler passed.

- 8 Configure parameters and signals so that the data is stored in memory and is accessible while the executable program runs. To efficiently implement a model in C code, you do not allocate memory for every parameter, signal, and state in the model. If the model algorithm does not require data to calculate outputs, code generation optimizations eliminate storage for that data. To allocate storage for the data so that you can access it during prototyping, you disable some optimizations.

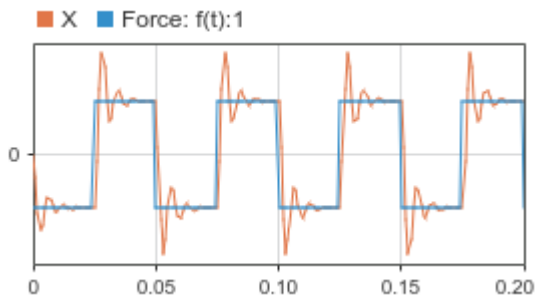
Click **Code Generation > Optimization**.

- 9 Confirm that Default parameter behavior is set to **Tunable**. With this setting, block parameters, such as the **Gain** parameter of a Gain block, are tunable in the generated code.
- 10 Point to the three dots at the bottom of the parameters to reveal the **Advanced parameters** label. Then, under **Advanced parameters**, confirm that Signal storage reuse is cleared. With this setting, the code generator allocates storage for individual signal lines. While running the executable program, you can monitor the values of the signals.
- 11 Configure the code generator to produce nonfinite data (for example, NaN and Inf) and related operations. Click **Code Generation > Interface**. Confirm that parameter Support: non-finite numbers is selected.
- 12 Configure a communication channel. For Simulink® to communicate with an executable program generated from a model, the model must include support for a communication channel. This example uses XCP on TCP/IP as the transport layer for a communication channel. Confirm these parameter settings:
 - External mode is selected.
 - Transport layer is set to XCP on TCP/IP. This selection specifies ext_xcp for parameter **Mex-file name**.
 - Automatically allocate static memory is selected.
- 13 Disable MAT-file logging. Load the data into the Simulation Data Inspector from the MATLAB base workspace. Point to the three dots at the bottom of the parameters to reveal the **Advanced parameters** label. Then, under **Advanced parameters**, confirm that parameter MAT-file logging is cleared.
- 14 Apply your configuration changes, close the Model Configuration Parameters dialog box, and save the model.

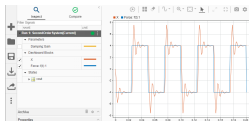
Simulate Model and View Results

- 1 From the Simulink Editor, in the **Simulation** tab, click **Run**. The clock on the **Run** button indicates that simulation pacing is enabled. Simulation pacing slows down a simulation so that you can observe system behavior. Visualizing simulations at a slower rate makes it easier to understand the underlying system design and identify design issues while demonstrating near real-time behavior.

During the simulation, the Dashboard Scope block displays the behavior of signals Force : f(t) : 1 and X.



- 2 In the Simulink Editor, in the **Simulation** tab, click **Data Inspector**. The Simulation Data Inspector opens with data from your simulation run imported.
- 3 Expand the run (if not already expanded). Then, to plot the data, select data signals X and Force: f(t):1.



Leave these results in the Simulation Data Inspector. Later, you compare the simulation data to the output data produced by the executable program generated from the model.

Build and Run Executable Program and View Results

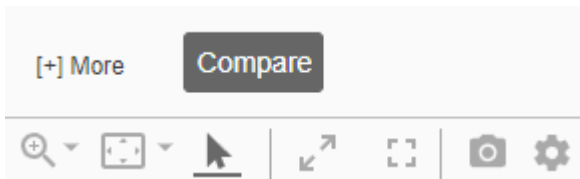
Build and run the model executable program.

- 1 In the Simulink Editor, in the **Hardware** tab, click **Monitor & Tune** to start the process. Simulink:
 - a Builds the target application files, which include the executable file. The debugging symbols for signals are created as part of the executable file or in a separate file, depending on the configuration of Code Generation. To learn more about different configurations for Code Generation, see “Approaches for Building Code Generated from Simulink Models”.
 - b Deploys the executable program as a separate process on your development computer.
 - c Connects the Simulink model to the executable program.
 - d Starts the model executable program code.

Compare Simulation and Executable Program Results

Use the Simulation Data Inspector to compare the executable program results with the simulation results.

- 1 In the Simulation Data Inspector, inspect the results of your executable program run, Run 2: SecondOrderSystem.
- 2 Click **Compare**.
- 3 Select the data runs that you want to compare. For this example, from the **Baseline** list, select Run 1: SecondOrderSystem. From the **Compare to** list, select Run 2: SecondOrderSystem.
- 4 In the upper-right corner of the Simulation Data Inspector, click **Compare**.



- 5 The Simulation Data Inspector indicates that the output for X and Force: $f(t):1$ from the executable program code is out of tolerance from the simulation data output. To see a plot of the results for X, under **File Comparisons**, select the row for X.



- 6 Inspect the comparison plot for Force: $f(t):1$. Under **File Comparisons**, select the row for Force: $f(t):1$.



- 7 Determine whether the numerical discrepancies are significant by specifying an absolute relative tolerance value. For this tutorial, set the global absolute tolerance to $1e-12$. In the upper-right corner click the **More** button. In the dialog box, under **Global Tolerances**, set **Absolute** to $1e-12$. Then, click **Compare**. The comparisons for X and Force: $f(t):1$ are within tolerance.

| Category | Value | Unit | Status |
|----------|-------|------|--------|
| Absolute | 1e-12 | | OK |
| Relative | 1e-06 | | OK |
| Force | 1e-06 | | OK |
| Position | 1e-06 | | OK |
| Velocity | 1e-06 | | OK |

For more information, about numerical consistency verification and tolerances, see “Numerical Consistency of Model and Generated Code Simulation Results”.

Next, tune a parameter during program execution.

Tune Parameter During Program Execution

| |
|---|
| In this section... |
| “Configure Data Accessibility and Communication Channel” on page 2-10 |
| “Build and Run Executable Program” on page 2-10 |
| “Tune Parameter and Observe Results” on page 2-10 |

Interact with a generated executable program while the program runs in nonreal time on your development computer by tuning a parameter and observing the results.

Configure Data Accessibility and Communication Channel

This part of the tutorial assumes that you have configured the example model `SecondOrderSystem` as described in “Configure Model for Verification” on page 2-6.

Build and Run Executable Program

- 1 To allow time for you to monitor changes that you make to the parameter, set the simulation stop time to `Inf`. In the Simulink Editor, click the **Hardware** tab. In the **Stop Time** field, set the simulation stop time to `Inf`.
- 2 Click **Monitor & Tune**. The software:
 - a Builds the executable program.
 - b Deploys the program as a separate process on your development computer.
 - c Connects the Simulink model to the program.
 - d Runs the model executable program code.

To stop the simulation, in the **Hardware** tab, click **Stop**.

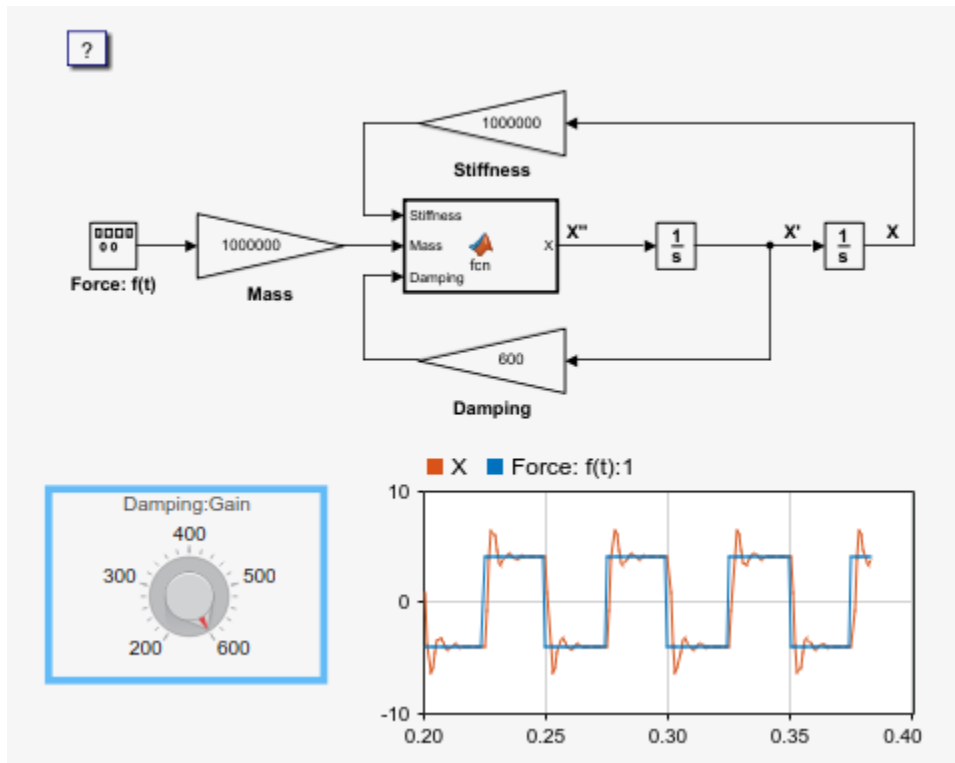
Tune Parameter and Observe Results

Experiment with the value of a block parameter during execution. Observe the impact of the change.

While the executable program is simulating on your development computer, in the model canvas, use the Knob block to change the value of the damping gain. For example, change the value to 600.

Observe:

- The changes in the plot are displayed in the Dashboard Scope block.
- On the **Hardware** tab, open the Model Data Editor by clicking **Tune Parameters**.



Next, package the generated program code and artifacts for deployment.

Deploy Prototype Code and Artifacts

Package prototype code and artifacts in a Zip file so that you can share or relocate project results.

Package Generated Code and Artifacts in a Zip File

- 1 In the Simulink Editor, in the **C Code** tab, click **Share**.
- 2 Under **Package Code & Artifacts**, specify a file name for the zip file. By default, the code generator uses the model name and file extension `.zip`. For this example, use the default name.
- 3 Click **Generate Code and Package**. The code generator produces zip file `SecondOrderSystem.zip`.
- 4 Explore the contents of the generated zip file.

Explore Other Options

To explore more ways to customize, verify, and deploy generated rapid-prototyping code and artifacts, see the information that is listed in this table.

| Goal | More Information |
|---|--|
| Configure data accessibility for rapid prototyping | "Access Signal, State, and Parameter Data During Execution" |
| Model multirate systems | "Timers and Scheduling" |
| Create multiple model configuration sets and share configuration parameter settings across models | "Model Configuration Sets" |
| Control how signals are stored and represented in the generated code | "How Generated Code Stores Internal Signal, State, and Parameter Data" |
| Generate block parameter storage declarations and interface block parameters to your code | "Create Tunable Calibration Parameter in the Generated Code" |
| Interface with legacy code for simulation and code generation | "Choose an External Code Integration Workflow" |
| Generate code compatible with C++ | "Select C or C++ Programming Language" |
| Create a protected model that hides block and line information for sharing with a third party | "Model Protection" |
| Customize the build process | "Code Compilation Customization" |