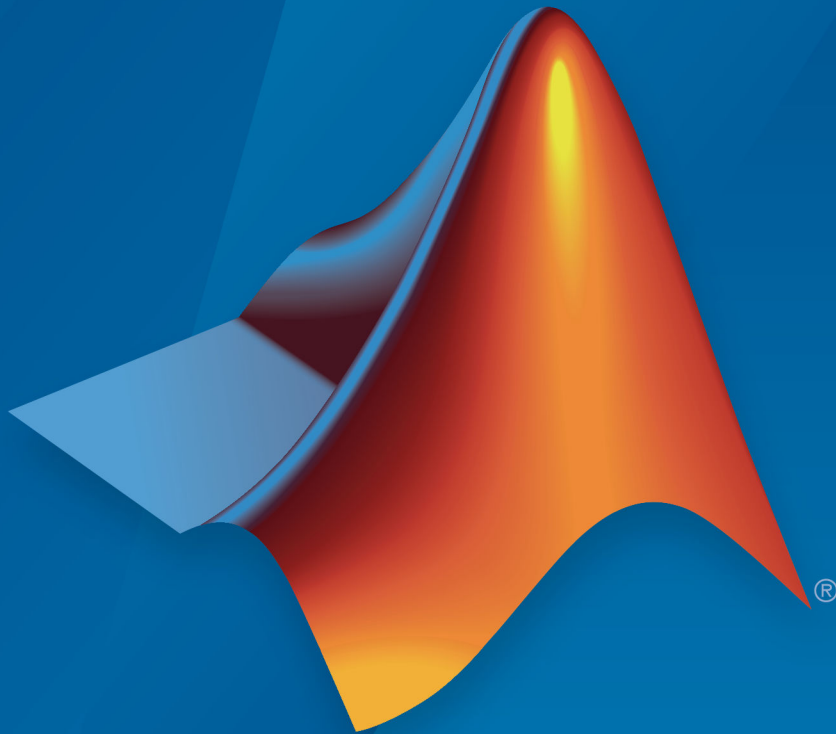


Simulink® Coder™

Getting Started Guide



MATLAB® & SIMULINK®

R2019a



How to Contact MathWorks



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



Phone: 508-647-7000



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

Simulink® Coder™ Getting Started Guide

© COPYRIGHT 2011-2019 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	Online only
September 2011	Online only
March 2012	Online only
September 2012	Online only
March 2013	Online only
September 2013	Online only
March 2014	Online only
October 2014	Online only
March 2015	Online only
September 2015	Online only
October 2015	Online only
March 2016	Online only
September 2016	Online only
March 2017	Online only
September 2017	Online only
March 2018	Online only
September 2018	Online only
March 2019	Online only

New for Version 8.0 (Release 2011a)
Revised for Version 8.1 (Release 2011b)
Revised for Version 8.2 (Release 2012a)
Revised for Version 8.3 (Release 2012b)
Revised for Version 8.4 (Release 2013a)
Revised for Version 8.5 (Release 2013b)
Revised for Version 8.6 (Release 2014a)
Revised for Version 8.7 (Release 2014b)
Revised for Version 8.8 (Release 2015a)
Revised for Version 8.9 (Release 2015b)
Rereleased for Version 8.8.1 (Release 2015aSP1)
Revised for Version 8.10 (Release 2016a)
Revised for Version 8.11 (Release 2016b)
Revised for Version 8.12 (Release 2017a)
Revised for Version 8.13 (Release 2017b)
Revised for Version 8.14 (Release 2018a)
Revised for Version 9.0 (Release 2018b)
Revised for Version 9.1 (Release 2019a)

Check Bug Reports for Issues and Fixes

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

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

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

1 Product Overview

Simulink Coder Product Description	1-2
Key Features	1-2
Code Generation Technology	1-3
Code Generation Workflow with Simulink Coder	1-4
Validation and Verification for System Development	1-7
V-Model for System Development	1-7
Types of Simulation and Prototyping in the V-Model	1-9
Target Environments and Applications	1-11
About Target Environments	1-11
Types of Target Environments	1-11
Applications of Supported Target Environments	1-13

2 Getting Started Examples

Generate C Code for a Model	2-2
Configure Model for Code Generation	2-2
Check Model Configuration for Execution Efficiency	2-4
Simulate the Model	2-6
Generate Code	2-6
View the Generated Code	2-7
Build and Run Executable	2-11
Configure Model to Output Data to MAT-File	2-11
Build Executable	2-13

Run Executable	2-14
View Results	2-15
Tune Parameters and Monitor Signals During Execution . . .	2-18
Configure Data Accessibility	2-18
Build Standalone Executable	2-20
Run Executable	2-21
Connect Simulink to Executable	2-21
Tune Parameter	2-22
More Information	2-23

Product Overview

- “Simulink Coder Product Description” on page 1-2
- “Code Generation Technology” on page 1-3
- “Code Generation Workflow with Simulink Coder” on page 1-4
- “Validation and Verification for System Development” on page 1-7
- “Target Environments and Applications” on page 1-11

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 Technology

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

Code generation technology and related products provide tooling that you can apply to the V-model for system development. The V-model is a representation of system development that highlights verification and validation steps in the development process. For more information, see “Validation and Verification for System Development” on page 1-7.

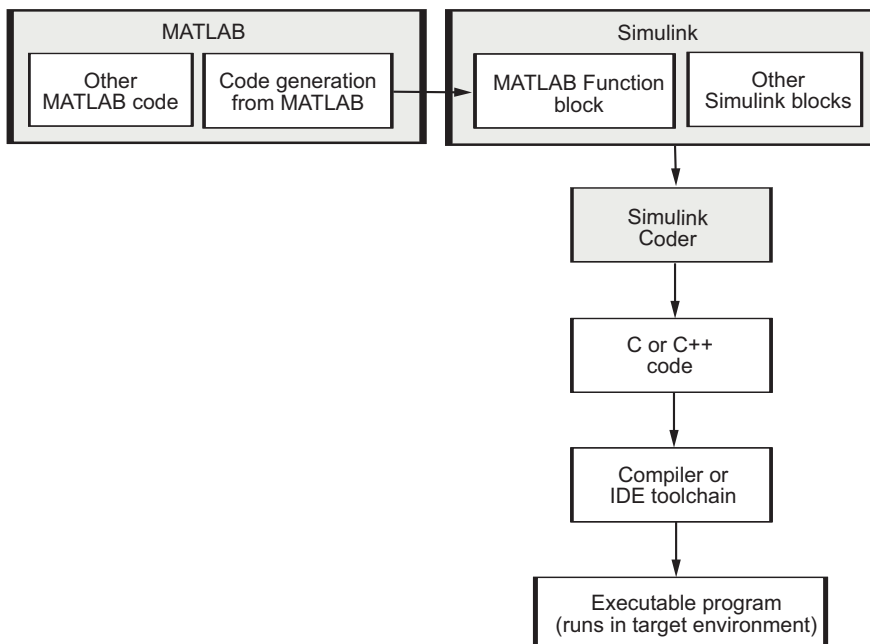
To learn model design patterns that include Simulink blocks, Stateflow charts, and MATLAB functions, and map to commonly used C constructs, see “Modeling Patterns for C Code” (Embedded Coder).

Code Generation Workflow with Simulink Coder

You can use MathWorks code generation technology to generate standalone C or C++ source code for rapid prototyping, simulation acceleration, and hardware-in-the-loop (HIL) simulation:

- By developing Simulink models and Stateflow charts, and then generating C/C++ code from the models and charts with the Simulink Coder product
- By integrating MATLAB code for code generation in MATLAB Function blocks in a Simulink model, and then generating C/C++ code with the Simulink Coder product

You can generate code for most Simulink blocks and many MathWorks products on page 1-3. The following figure shows the product workflow for code generation with Simulink Coder. Other products that support code generation, such as Stateflow software, are available.

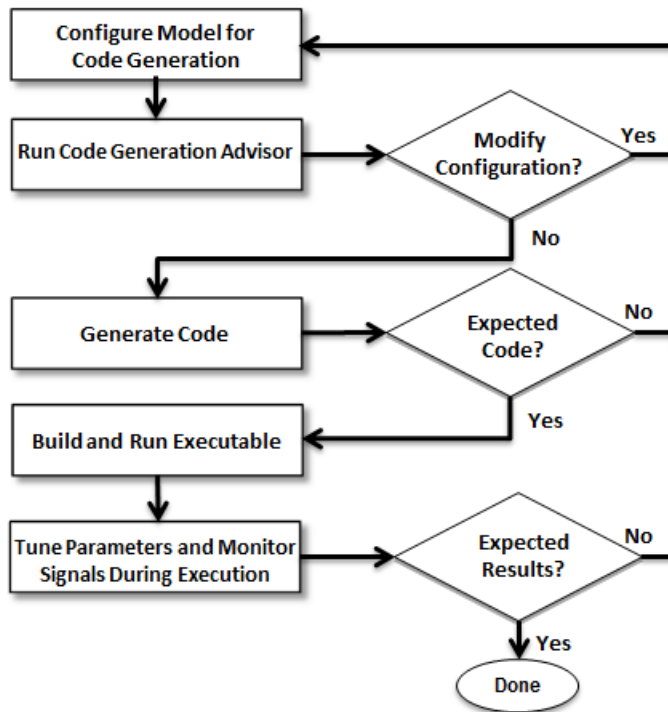


The code generation workflow is a part of the V-model on page 1-7 for system development. The process includes code generation, code verification, and testing of the

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.

A typical workflow for applying the software to the application development process is:



For more information on how to perform these tasks, see the *Getting Started with Simulink Coder* tutorials:

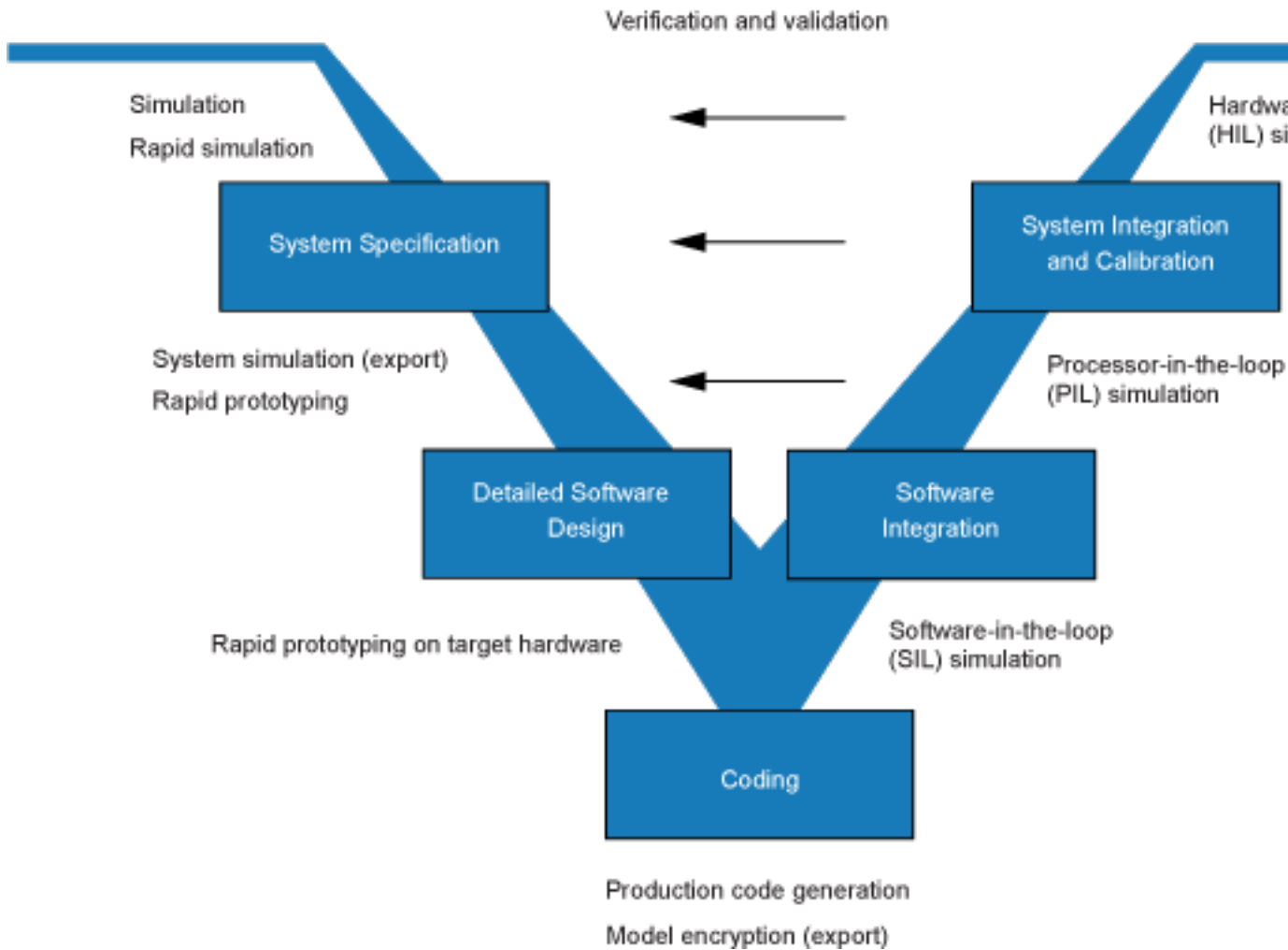
- 1 “Generate C Code for a Model” on page 2-2
- 2 “Build and Run Executable” on page 2-11
- 3 “Tune Parameters and Monitor Signals During Execution” on page 2-18

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 1-9.

Types of Simulation and Prototyping in the V-Model

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

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

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

Target Environments and Applications

In this section...

“About Target Environments” on page 1-11

“Types of Target Environments” on page 1-11

“Applications of Supported Target Environments” on page 1-13

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	<p>The same computer that runs MATLAB and Simulink. Typically, a host computer is a PC or UNIX^a environment that uses a non-real-time operating system, such as Microsoft[®] Windows[®] or Linux^b. Non-real-time (general purpose) operating systems are nondeterministic. For example, those operating systems might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Therefore, the executable for your generated code might run faster or slower than the sample rates that you specified in your model.</p>
Real-time simulator	<p>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:</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, such as VxWorks[®] from Wind River[®] Systems <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>Typically, a real-time simulator connects to a host 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 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:</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.

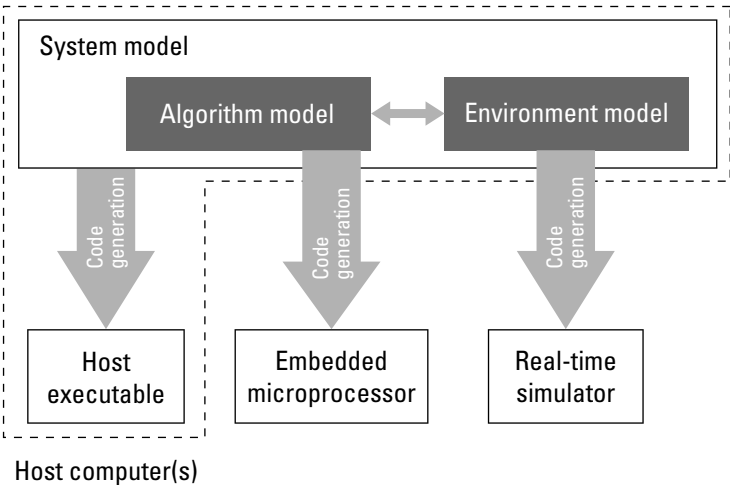
b. Linux is a registered trademark of Linus Torvalds.

A target environment can:

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

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

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



Applications of Supported Target Environments

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

Application	Description
Host Computer	

Application	Description
"Acceleration" (Simulink)	You apply techniques to speed up the execution of model simulation in the context of the MATLAB and Simulink environments. Accelerated simulations are especially useful when run time is long compared to the time associated with compilation and checking whether the target is up to date.
Rapid Simulation	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 (Embedded Coder)	You integrate components into a larger system. You provide generated source code and related dependencies for building a system in another environment or in a host-based shared library to which other code can dynamically link.
"Protect Models to Conceal Contents"	You generate a protected model for use by a third-party vendor in another Simulink simulation environment.
Real-Time Simulator	
Real-Time 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.
Shared Object Libraries (Embedded Coder)	You integrate generated source code and dependencies for components into a larger system that is built in another environment. You can use shared library files for intellectual property protection.

Application	Description
Hardware-in-the-Loop (HIL) Simulation	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" (Embedded Coder)	From a model, you generate code that is optimized for speed, memory usage, simplicity, and possibly, compliance with industry standards and guidelines.
"Software-in-the-Loop Simulation" (Embedded Coder)	You execute generated code with your plant model within Simulink to verify conversion of the model to code. You might change the code to emulate target word size behavior and verify numerical results expected when the code runs on an embedded microprocessor. Or, you might use actual target word sizes and just test production code behavior.
"Processor-in-the-Loop Simulation" (Embedded Coder)	You test an object code component with a plant or environment model in an open- or closed-loop simulation to verify model-to-code conversion, cross-compilation, and software integration.
Hardware-in-the-loop (HIL) Simulation	You verify an embedded system or embedded computing unit (ECU), using a real-time target environment.

Getting Started Examples

- “Generate C Code for a Model” on page 2-2
- “Build and Run Executable” on page 2-11
- “Tune Parameters and Monitor Signals During Execution” on page 2-18

Generate C Code for a Model

In this section...
“Configure Model for Code Generation” on page 2-2
“Check Model Configuration for Execution Efficiency” on page 2-4
“Simulate the Model” on page 2-6
“Generate Code” on page 2-6
“View the Generated Code” on page 2-7

Simulink Coder generates standalone C/C++ code for Simulink models for deployment in a wide variety of applications. The **Getting Started with Simulink Coder** includes three tutorials. It is recommended that you complete **Generate C Code for a Model** first, and then the following tutorials: “Build and Run Executable” on page 2-11 and “Tune Parameters and Monitor Signals During Execution” on page 2-18.

This example shows how to prepare the `rtwdemo_secondOrderSystem` model for code generation and generate C code for real-time simulation. The `rtwdemo_secondOrderSystem` model implements a second-order physical system called an ideal mass-spring-damper system. Components of the system equation are listed as mass, stiffness, and damping.

Set your current MATLAB folder to a writeable folder. Then, to open the model, in the command window, type:

```
rtwdemo_secondOrderSystem
```

Configure Model for Code Generation

To prepare the model for generating C89/C90 compliant C code, you can specify code generation settings in the Configuration Parameters dialog box. To open the Configuration Parameters dialog box, in the Simulink Editor, click the **Model Configuration Parameters** button.



Solver for Code Generation

To generate code for a model, you must configure a solver. Simulink Coder generates only standalone code for a fixed-step solver. On the **Solver** pane, select a solver that meets the performance criteria for real-time execution. For this model, observe the following settings.

Simulation time

Start time: 0.0 Stop time: .2

Solver selection

Type: Fixed-step Solver: ode3 (Bogacki-Shampine)

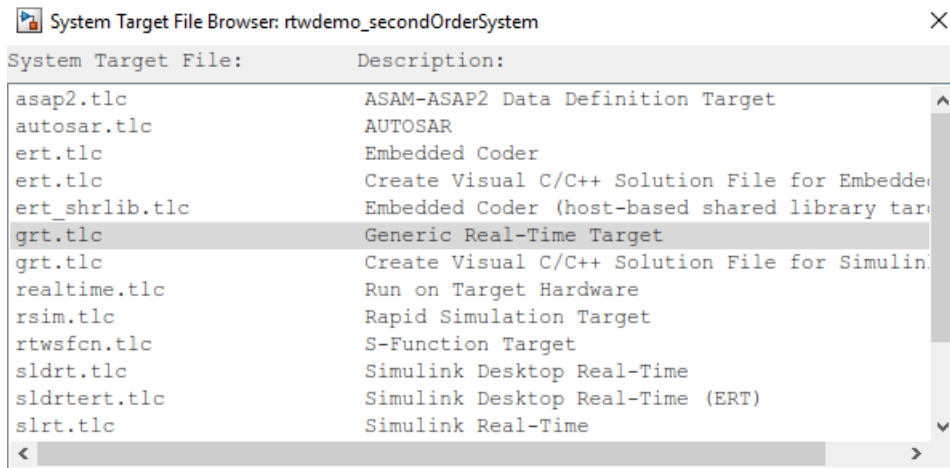
▼ Solver details

Fixed-step size (fundamental sample time): 0.001

Code Generation Target

To specify a target configuration for the model, choose a system target file, a template makefile, and a make command. You can use a ready-to-run Generic Real-Time Target (GRT) configuration.

- 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 `grt.tlc` Generic Real-Time Target.



3 Click **OK**.

Code Generation Report

You can specify that the code generation process generate an HTML report that includes the generated code and information about the model.

- 1 In the Configuration Parameters dialog box, select the **Code Generation > Report** pane.
- 2 For this example, these configuration parameters are selected:
 - **Create code generation report**
 - **Open report automatically**

After the code generation process is complete, an HTML code generation report appears in a separate window.

Check Model Configuration for Execution Efficiency

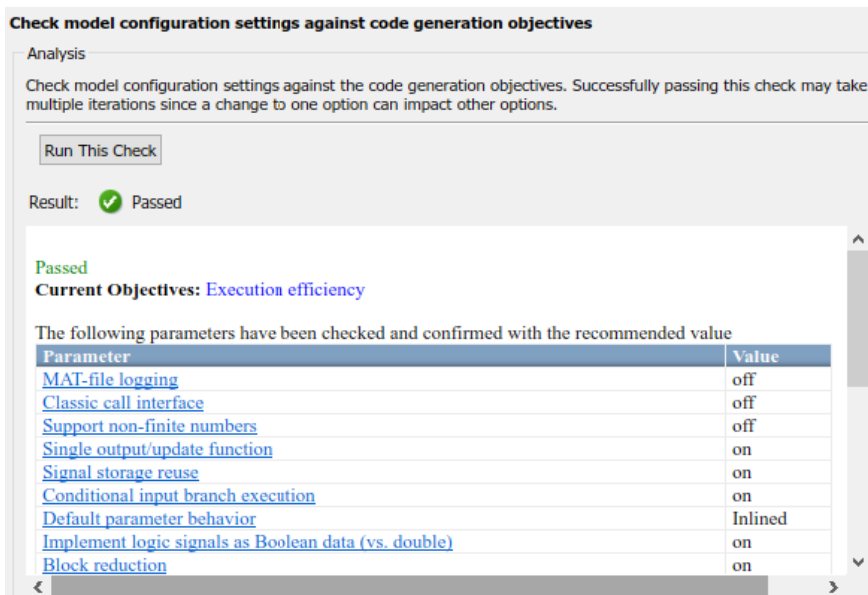
When generating code for real-time deployment, a common objective for the generated code is that it executes efficiently. You can run the Code Generation Advisor on your model for a specified objective such as **Execution efficiency**. The advisor provides information on how to meet code generation objectives for your model.

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

- 2 Under Code generation objectives, select the following, and then click **Apply**:
 - **Select objective**—From the drop-down list, select **Execution efficiency**.
 - **Check model before generating code**—From the drop-down list, select **On (proceed with warnings)**.
- 3 Click **Check Model**.
- 4 In the System Selector dialog box, click **OK** to run checks on the model.

After the advisor runs, there are two warnings indicated by a yellow triangle.

- 5 On the left pane, click **Check model configuration settings against code generation objectives**.
- 6 On the right pane, click **Modify Parameters**. The configuration parameters that caused the warning are changed to the recommended setting.
- 7 On the right pane, click **Run This Check**. The check now passes. The Code Generation Advisor lists the parameters and their recommended settings for **Execution efficiency**. Close the Code Generation Advisor.

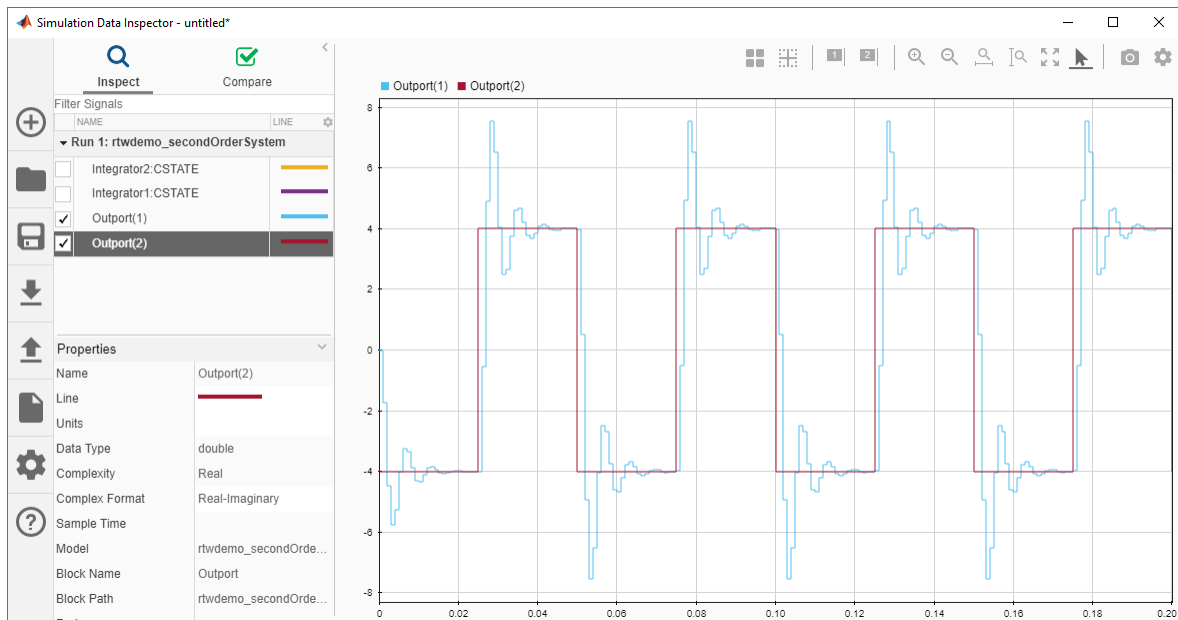


Ignore the warning for the **Identify questionable blocks within the specified system**. This warning is for production code generation, which is not the goal for this example.

Simulate the Model

In the Simulink Editor, simulate the model to verify that the output is as you expect for the specified solver settings.

- 1 Simulate the model.
- 2 When the simulation is done, in the Simulink Editor, click the **Simulation Data Inspector** button to open the Simulation Data Inspector.
- 3 Expand the run, and then select the Output block data check boxes to plot the data.



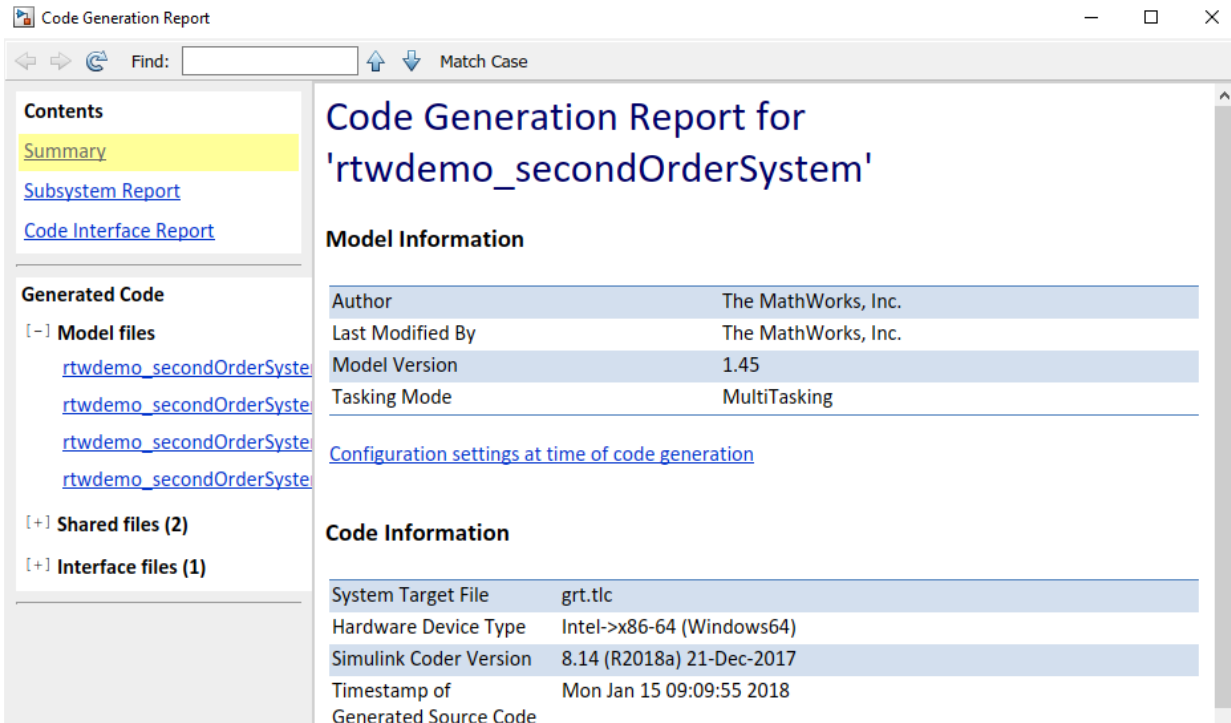
Leave these results in the Simulation Data Inspector. Later, you can compare the simulation data to the output data generated from the executable shown in “Build and Run Executable” on page 2-11.

Generate Code

- 1 In the Configuration Parameters dialog box, on the **Code Generation** pane, select the **Generate code only** check box.

- 2 Click **Apply**.
- 3 In the Simulink Editor, press **Ctrl+B**.

After code generation, the HTML code generation report opens.



View the Generated Code

The code generation process places the source code files in the `rtwdemo_secondOrderSystem_grt_rtw` folder. The HTML code generation report is in the `rtwdemo_secondOrderSystem_grt_rtw/html/rtwdemo_secondOrderSystem_codegen_rpt.html` folder.

Open the HTML code generation report, `rtwdemo_secondOrderSystem_codegen_rpt.html`. The code generation report includes:

- Summary
- Subsystem Report
- Code Interface Report
- Generated Code

Code Interface Report

In the left navigation pane, click **Code Interface Report** to open the report. The code interface report provides information on how an external main program can interface with the generated code. There are three entry-point functions to initialize, step, and terminate the real-time capable code.

Entry Point Functions

Function: [rtwdemo_secondOrderSystem_initialize](#)

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

Function: [rtwdemo_secondOrderSystem_step](#)

Prototype	void rtwdemo_secondOrderSystem_step(void)
Description	Output entry point of generated code
Timing	Must be called periodically, every 0.001 seconds
Arguments	None
Return value	None
Header file	rtwdemo_secondOrderSystem.h

Function: [rtwdemo_secondOrderSystem_terminate](#)

Prototype	void rtwdemo_secondOrderSystem_terminate(void)
Description	Termination entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_secondOrderSystem.h

For `rtwdemo_secondOrderSystem`, the **Outports** section includes a single output variable representing the Output block of the model.

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Output	rtwdemo_secondOrderSystem_Y.Outport	real_T	[2]

Generated Code

The generated `model.c` file `rtwdemo_secondOrderSystem.c` contains the algorithm code, including the ODE solver code. The model data and entry-point functions are accessible to a caller by including `rtwdemo_secondOrderSystem.h`.

On the left navigation pane, click `rtwdemo_secondOrderSystem.h` to view the extern declarations for block outputs, continuous states, model output, entry points, and timing data:

<code>/* Block signals (auto storage) */</code>	
<code>extern B_rtwdemo_secondOrderSystem_T rtwdemo_secondOrderSystem_B;</code>	Block Outputs
<code>/* Continuous states (auto storage) */</code>	
<code>extern X_rtwdemo_secondOrderSystem_T rtwdemo_secondOrderSystem_X;</code>	Continuous States
<code>/* External outputs (root outports fed by signals with auto storage) */</code>	
<code>extern ExtY_rtwdemo_secondOrderSyste_T rtwdemo_secondOrderSystem_Y;</code>	Model Output
<code>/* Model entry point functions */</code>	
<code>extern void rtwdemo_secondOrderSystem_initialize(void);</code>	
<code>extern void rtwdemo_secondOrderSystem_step(void);</code>	Entry Points
<code>extern void rtwdemo_secondOrderSystem_terminate(void);</code>	
<code>/* Real-time Model object */</code>	
<code>extern RT_MODEL_rtwdemo_secondOrderS_T *const rtwdemo_secondOrderSystem_M;</code>	Timing Data

The next example shows how to build an executable. See “Build and Run Executable” on page 2-11.

Build and Run Executable

In this section...
“Configure Model to Output Data to MAT-File” on page 2-11
“Build Executable” on page 2-13
“Run Executable” on page 2-14
“View Results” on page 2-15

For building an executable, Simulink Coder supports these techniques:

- Using toolchain-based controls.
- Using template makefile-based controls.
- Interfacing with an IDE.

The code generation target that you select for your model determines the build process controls that are presented to you. The example model uses the GRT code generation target, which enables the toolchain-based controls. This example shows how to build an executable by using the toolchain controls, and then test the executable results.

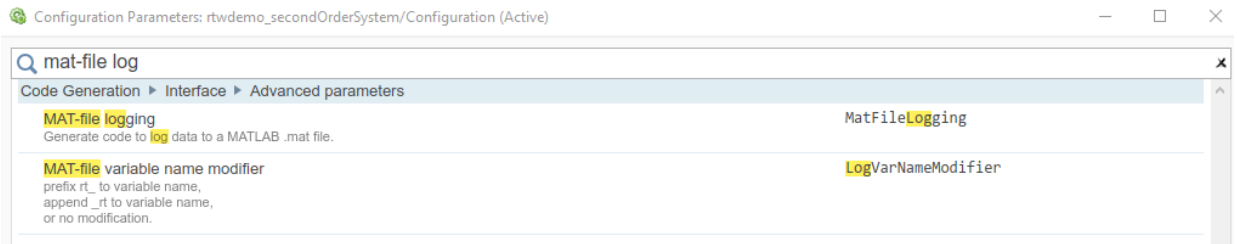
Before following this example, simulate the example model, `rtwdemo_secondOrderSystem`, as described in “Generate C Code for a Model” on page 2-2. Later on, the simulation results are compared to the results from running the executable.

Configure Model to Output Data to MAT-File

Before building the executable, enable the model to log output to a MAT-file instead of the base workspace. You can then view the output data by importing the MAT-file into the Simulation Data Inspector.

- 1 In the Configuration Parameters dialog box, use the search bar to find **MAT-file logging**.

2 Getting Started Examples



- 2 Click the **MAT-file logging** search result.
The **Code Generation > Interface** pane opens.
- 3 Select **MAT-file logging** and set **MAT-file variable name modifier** to `rt_`. Click **Apply**.
- 4 In the **Configuration Parameters > Data Import/Export** pane, under **Save to workspace or file**, specify the parameters, as shown here.

Load from workspace

☐ Input: [t, u] Connect Input

☐ Initial state: xInitial

Save to workspace or file

☒ Time: tout

☒ States: xout Format: Structure with time ▼

☒ Output: yout

☐ Final states: xFinal ☐ Save complete SimState in final state

☒ Signal logging: logsOut Configure Signals to Log...

☒ Data stores: dsmout

☐ Log Dataset data to file: out.mat

☐ Single simulation output: out Logging intervals: [-Inf, Inf]

Simulation Data Inspector

☒ Record logged workspace data in Simulation Data Inspector

▼ Additional parameters

Save options

☐ Limit data points to last: 1000 Decimation: 1

- 5 If necessary, click **Apply**.

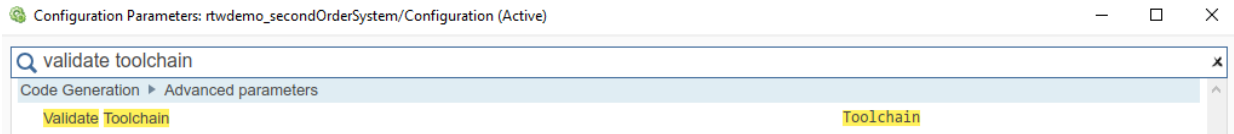
Build Executable

The internal MATLAB function `make_rtw` executes the code generation process for a model. `make_rtw` performs an update diagram on the model, generates code, and builds an executable.

To build an executable in the working MATLAB folder:

- 1 On the **Configuration Parameters > Code Generation** pane, under **Toolchain settings**, set **Toolchain** to Automatically locate an installed toolchain.

- 2 In the Configuration Parameters dialog box, use the search bar to find the **ValidateToolchain** button. Click the button to verify your toolchain.



The Validation Report indicates if the checks passed.

- 3 In the **Configuration Parameters > Code Generation > Interface** pane, select **Software environment > Support non-finite numbers**.
- 4 In the **Configuration Parameters > Code Generation** pane, under **Build process**, clear the **Generate code only** check box.
- 5 Click **Apply**.
- 6 To build the executable, press **Ctrl+B** in the Simulink Editor.

As the code generator builds the executable, **Building** appears on the bottom left of the Simulink Editor window. When the text reads **Ready** and the Code Generation Report appears, the process is complete.

The code generator places the executable in the working folder. On Windows the executable is `rtwdemo_secondOrderSystem.exe`. On Linux the executable is `rtwdemo_secondOrderSystem`.

Run Executable

In the MATLAB Command Window, run the executable by using one of these commands:

- For Windows, type:

```
!rtwdemo_secondOrderSystem
```

- For Linux, type:

```
!./rtwdemo_secondOrderSystem
```


The MATLAB Command Window displays this output:

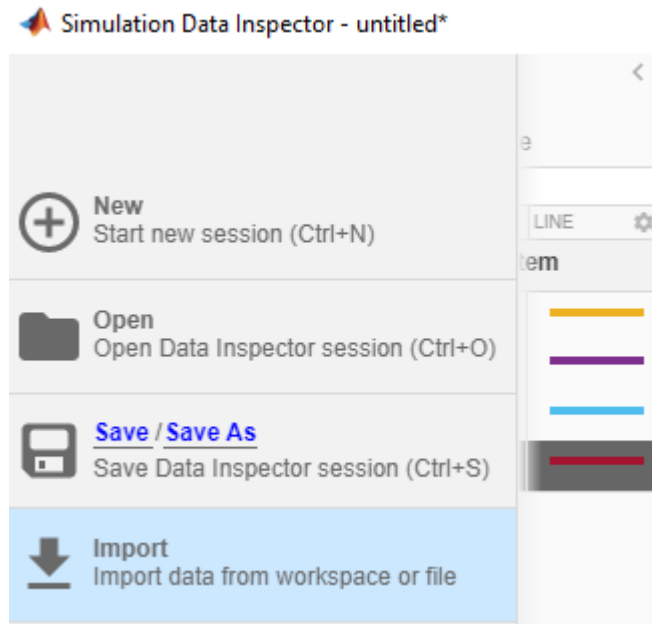
```
** starting the model **  
** created rtwdemo_secondOrderSystem.mat **
```

The code generator outputs a MAT-file, `rtwdemo_secondOrderSystem.mat`. It saves the file to the working folder.

View Results

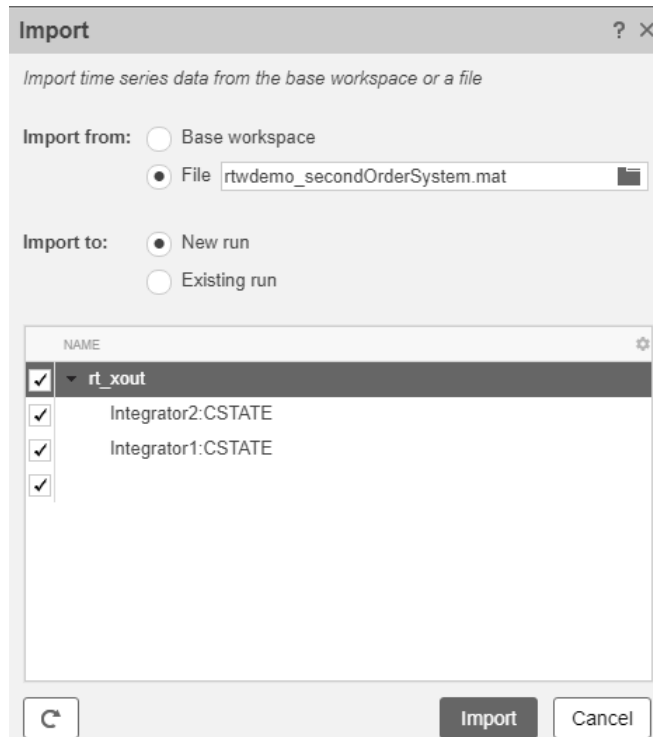
This example shows how to import data into the Simulation Data Inspector, and then compare the executable results to the simulation results. If you have not already sent logged data from the workspace to the Simulation Data Inspector, follow the instructions in “Simulate the Model” on page 2-6.

- 1 If the Simulation Data Inspector is not already open, in the Simulink Editor, click the **Simulation Data Inspector** button .
- 2 To open the Import dialog box, on the left side of the Simulation Data Inspector, click **Import**.



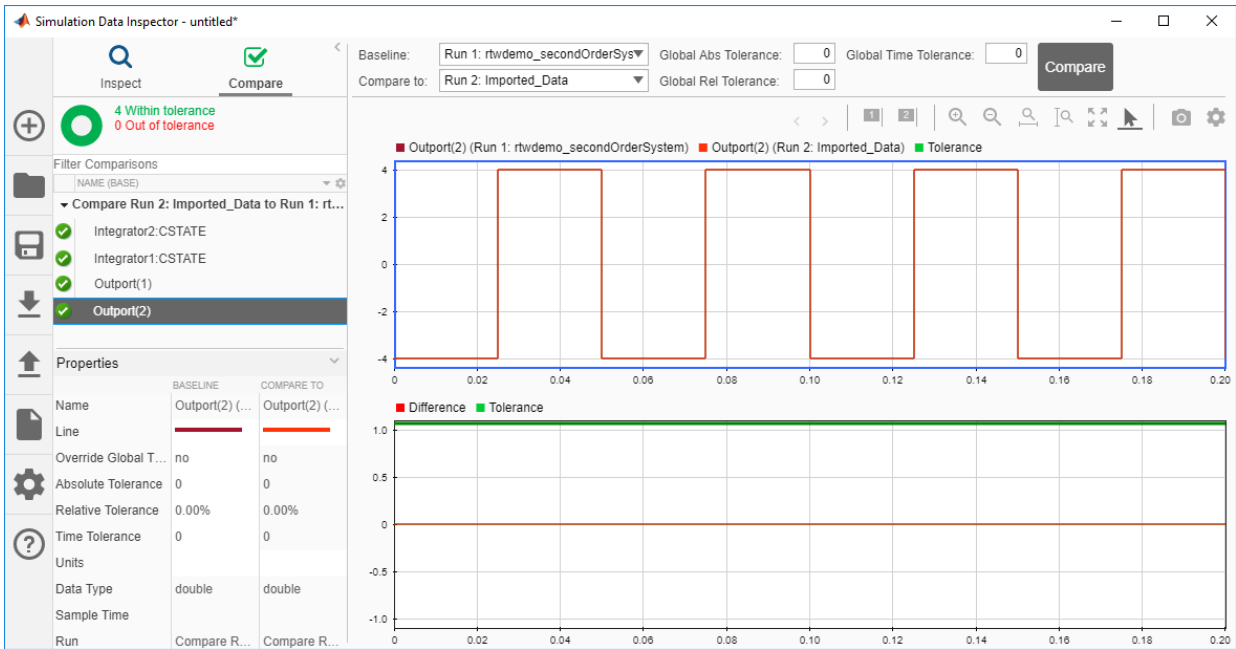
- 3 For **Import from**, select the **File** option button.

Enter the `rtwdemo_secondOrderSystem.mat` file. The data populates the table.



Click **Import**.

- 4 In the Simulation Data Inspector, click **Compare**.
- 5 Select Run 1: `rtwdemo_secondOrderSystem` from the **Baseline** list and Run 2: `Imported_Data` from the **Compare To** list.
- 6 Click the **Compare** button. The Simulation Data Inspector indicates that the output from the executed code is within a reasonable tolerance of the simulation data output previously collected in “Generate C Code for a Model” on page 2-2.



The next example shows how to run the executable on your machine by using Simulink as an interface for testing. See “Tune Parameters and Monitor Signals During Execution” on page 2-18.

Tune Parameters and Monitor Signals During Execution

In this section...
“Configure Data Accessibility” on page 2-18
“Build Standalone Executable” on page 2-20
“Run Executable” on page 2-21
“Connect Simulink to Executable” on page 2-21
“Tune Parameter” on page 2-22
“More Information” on page 2-23

This example shows how to access parameter and signal data while a generated executable runs. Use this approach to experiment with parameters and signal inputs during rapid prototyping.

To interact with a generated program by using Simulink, simulate a model in external mode. In this example, the program runs as a standalone executable in nonreal time on your host computer. Simulink communicates with the executable by using a TCP/IP link.

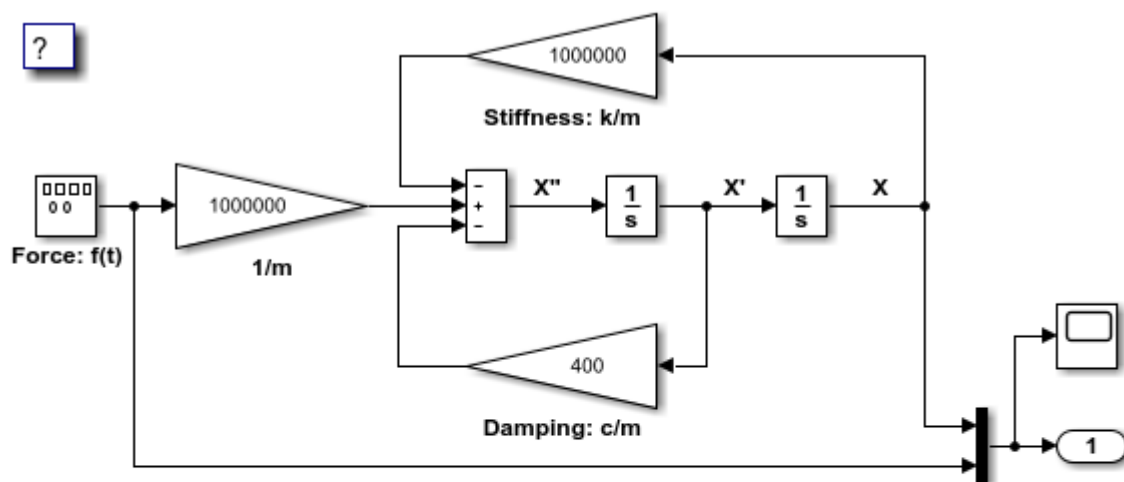
To learn about the example model and how to generate code, see the tutorials “Generate C Code for a Model” on page 2-2 and “Build and Run Executable” on page 2-11.

Configure Data Accessibility

To efficiently implement a model in C code, you typically do not allocate storage in memory for every parameter, signal, and state in the model. If the model algorithm does not require these data items to calculate outputs, code generation optimizations can eliminate storage for the data. To instead allocate storage for the data so that you can access it during prototyping, disable the optimizations.

- 1 Open the example model.

`rtwdemo_secondOrderSystem`



Model of a second-order Mass-Spring-Damper system governed by the system equation:

$m X'' + c X' + k X = f(t)$, where

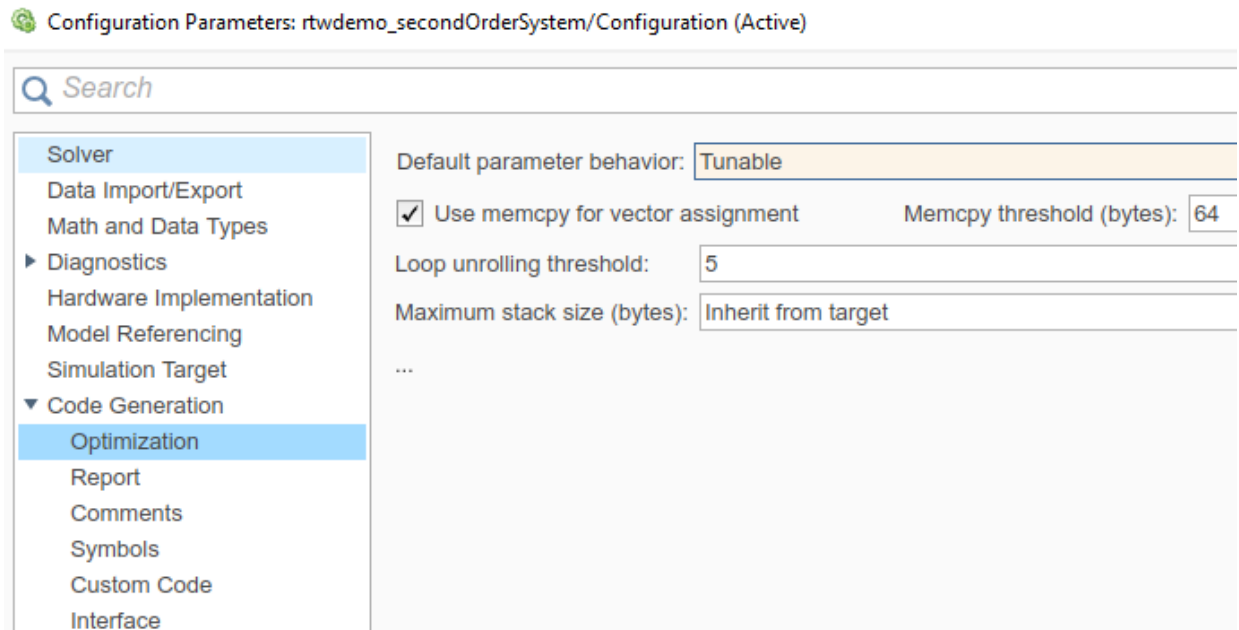
m = Mass of the system: 1.0E-6 kg

c = Damping ratio: 4.0e-4 Ns/m

k = Spring stiffness: 1.0 N/m

$f(t)$ = Forcing function in the x-direction (N)

- 2 Set **Configuration Parameters** > **Code Generation** > **Optimization** > **Default parameter behavior** to Tunable.



With this setting, by default, block parameters (such as the **Gain** parameter of a Gain block) are tunable in the generated code.

- 3 Search for and clear the configuration parameter **Signal storage reuse**.

With this setting, by default, the generated code allocates storage for signal lines. The external mode simulation can access the values of these signals so that you can monitor the signals, for example, by using a Scope block in the model.

- 4 Click **Apply**.

Build Standalone Executable

Generate code and create an executable from the model.

- 1 Select the **Configuration Parameters > Code Generation > Interface > External mode** check box.

This option enables the generated executable to later communicate with Simulink.

- 2 Generate code from the model. For example, in the model, press **Ctrl+B**.

The generated executable, `rtwdemo_secondOrderSystem`, appears in your current folder. A code generation report opens.

Run Executable

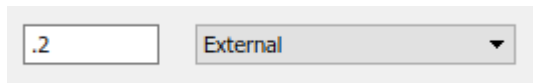
At the command prompt, run the generated executable. Use the option `-tf` to override the stop time so that the executable runs indefinitely.


```
system('rtwdemo_secondOrderSystem -tf inf &')
```

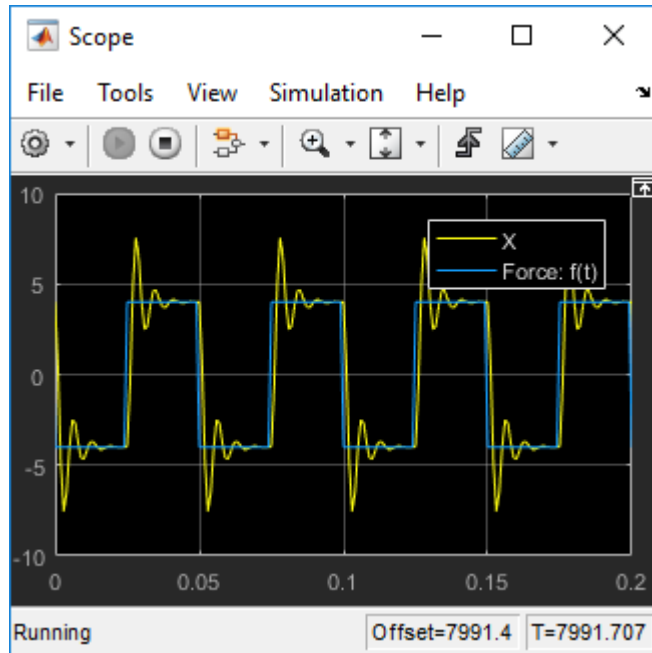
Connect Simulink to Executable

To interact with the running process, use external mode simulation in Simulink.

- 1 On the Simulink Editor toolbar, set the **Simulation mode** drop-down list to **External**.



- 2 Click the **Connect to Target** button .
- 3 In the model, double-click the Scope block. The scope displays the values of the system output signals.

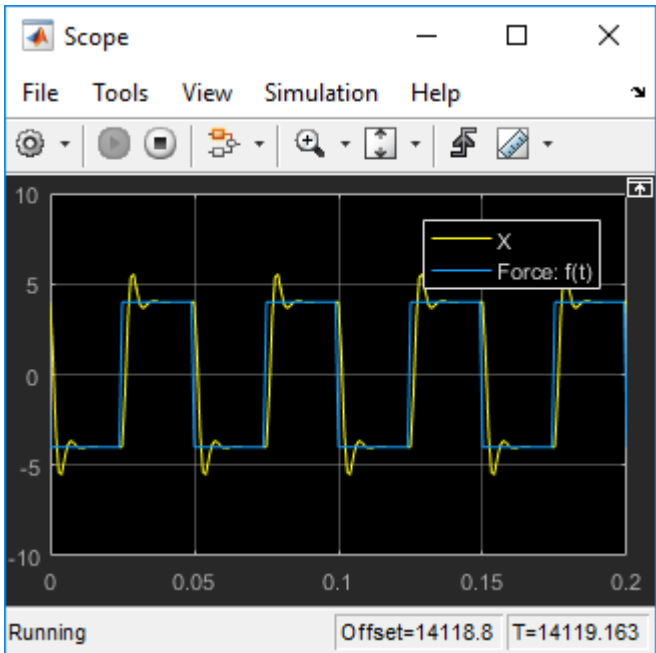


Tune Parameter

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

- 1 In the model, select **View > Model Data Editor**.
- 2 In the Model Data Editor, inspect the **Parameters** tab.
- 3 In the model, click the Gain block named Damping: c/m.
- 4 In the Model Data Editor, change the value of **Gain** from 400 to 800.

The Scope block shows the effect of the change on the signal values.



More Information

For more information, this table includes common capabilities and resources for generating and executing C and C++ code for your model.

Goal	More Information
Configure data accessibility for rapid prototyping	"Access Signal, State, and Parameter Data During Execution"
Model multirate systems	"Scheduling"
Create multiple model configuration sets and share configuration parameter settings across models	"Configuration Reuse" (Simulink)
Control how signals are stored and represented in the generated code	"How Generated Code Stores Internal Signal, State, and Parameter Data"

Goal	More Information
Generate block parameter storage declarations and interface block parameters to your code	"Create Tunable Calibration Parameter in the Generated Code"
Store data separate from the model	"Data Objects" (Simulink)
Interface with legacy code for simulation and code generation	"External Code Integration"
Generate separate files for subsystems and model	"File Packaging"
Configure code comments and reserve keywords	"Code Appearance"
Generate code compatible with C++	"Programming Language"
Export an ASAP2 file containing information about your model during the code generation process	"Export ASAP2 File for Data Measurement and Calibration"
Write host-based or target-based code that interacts with signals, states, root-level inputs/outputs, and parameters in your target-based application code	"Exchange Data Between Generated and External Code Using C API"
Create a protected model that hides all block and line information for sharing with a third party	"Model Protection"
Customize the build process	"Build Process Customization"
Create a custom block	"Block Authoring and Customization"
Create your own target	"Target Development"