Simulink® Coder™

Getting Started Guide

R2013**b**

MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Simulink® Coder™ Getting Started Guide

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)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

1 Product Overview

2 Getting Started Examples

1

Product Overview

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++ from Simulink diagrams, 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
- Incremental code generation for large models
- Integer, floating-point, and fixed-point data type support
- 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

Code Generation Technology

MathWorks® Code generation technology generates C or C++ code and executables for algorithms that you model 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 high degrees 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 degrees 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 about the V-model and how MathWorks code generation technology and related products provide tooling that you can apply to the process, see “V-Model for System Development” on page 1-23.

Target Environments and Applications

In this section...

About Target Environments

In addition to generating source code, the code generator produces make or project files to build an executable 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 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 Supported By Simulink Coder

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 those environments listed in the following table.

Target Environment	Description
Host computer	The same computer that runs MATLAB and Simulink. Typically, a host computer is a PC or UNIX®[1] environment that uses a non-real-time operating system, such as Microsoft® Windows® or Linux®[2]. Non-real-time (general purpose) operating systems are nondeterministic. For example, those operating systems might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Therefore, the executable for your generated code might run faster or slower than the sample rates that you specified in your model.
Real-time simulator	A different computer than the host computer. A real-time simulator can be a PC or UNIX environment that uses a real-time operating system (RTOS), such as: • xPC Target™ system • A real-time Linux system • A Versa Module Eurocard (VME) chassis with PowerPC® processors running a commercial RTOS, such as VxWorks® from Wind River® Systems The generated code runs in real time and behaves deterministically. The exact nature of execution varies based on the particular behavior of the system hardware and RTOS. Typically, a real-time simulator connects to a host computer for data logging, interactive parameter tuning, and Monte Carlo batch execution studies.
Embedded microprocessor	A computer that you eventually disconnect from a host computer and run as a standalone computer as part of an electronics-based product. Embedded microprocessors range in price and performance, from high-end digital signal processors (DSPs) that 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:

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

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

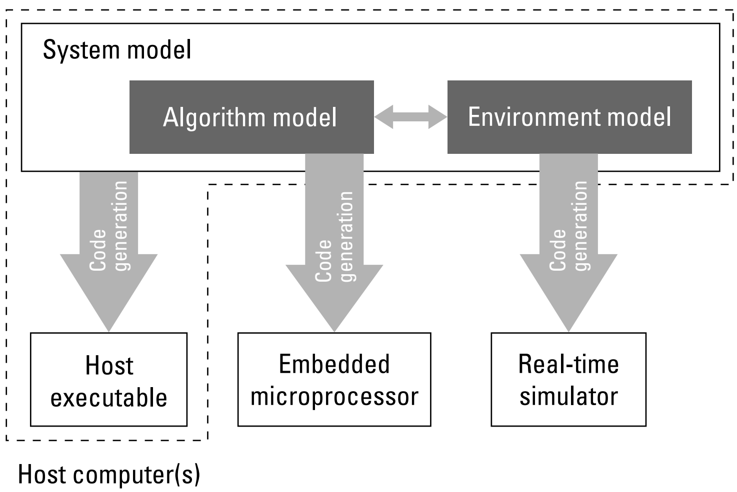
Target Environment	Description
	• Use a full-featured RTOS • Be driven by basic interrupts • Use rate monotonic scheduling provided with code generation

A target environment can:

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

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

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	
Accelerated simulation	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 nonreal time on the host computer, but outside the context of the MATLAB and Simulink environments.
System simulation	You integrate components into a larger system. You provide generated source code and related dependencies for building a system in another environment or in a host-based shared library to which other code can dynamically link.
Model intellectual property protection	You generate a Simulink shareable object library for a model or subsystem for use by a third-party vendor in another Simulink simulation environment.
Real-Time Simulator	
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.
System simulation	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
On-target rapid prototyping	You generate code for a detailed design that you can run in real time on an embedded microprocessor while tuning parameters and monitoring real-time data. This design step allows you to assess, interact with, and optimize code, using embedded compilers and hardware.
Embedded Microprocessor	
Production code generation	From a model, you generate code that is optimized for speed, memory usage, simplicity, and potentially, compliance with industry standards and guidelines.
"Software-in-the-Loop (SIL) Simulation"	You execute generated code with your plant model within Simulink to verify conversion of the model to code. You might change the code to emulate target word size behavior and verify numerical results expected when the code runs on an embedded microprocessor. Or, you might use actual target word sizes and just test production code behavior.
"Processor-in-the-Loop (PIL) Simulation"	You test an object code component with a plant or environment model in an open- or closed-loop simulation to verify model-to-code conversion, cross-compilation, and software integration.
Hardware-in-the-loop (HIL) testing	You verify an embedded system or embedded computing unit (ECU), using a real-time target environment.

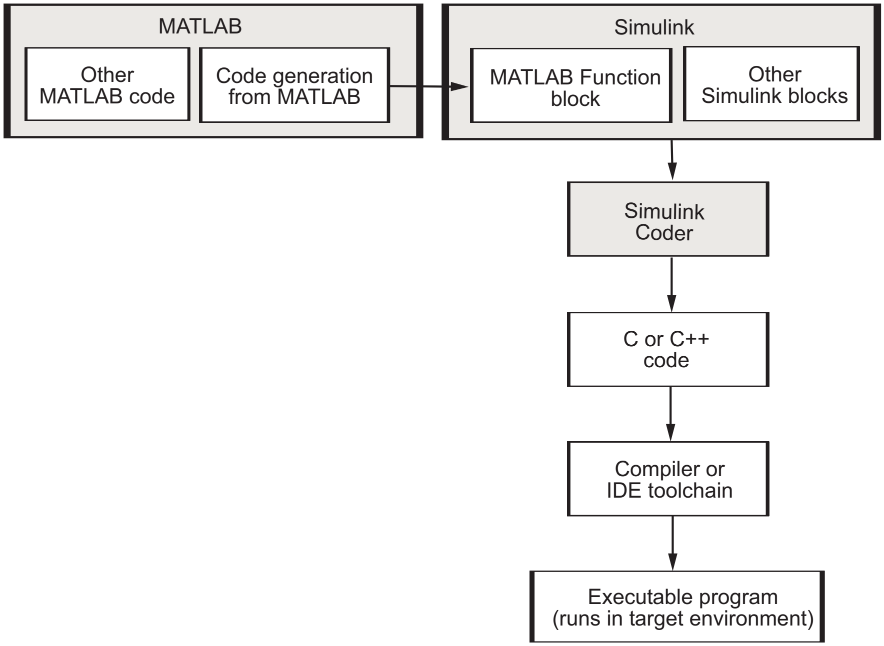
Algorithm Development Options

In this section...

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 into Simulink models, using code generation from MATLAB and the Simulink MATLAB Function block, and then generating C/C++ code with the Simulink Coder product

The following figure shows these design and deployment environment options. Although not shown in the figure, other products that support code generation, such as Stateflow software, are available.



If you are familiar with C language constructs and want to learn about how to map commonly used C constructs to code generated from model design patterns that include Simulink blocks, Stateflow charts, and MATLAB functions, see "Patterns for C Code".

Simulink and Stateflow Model

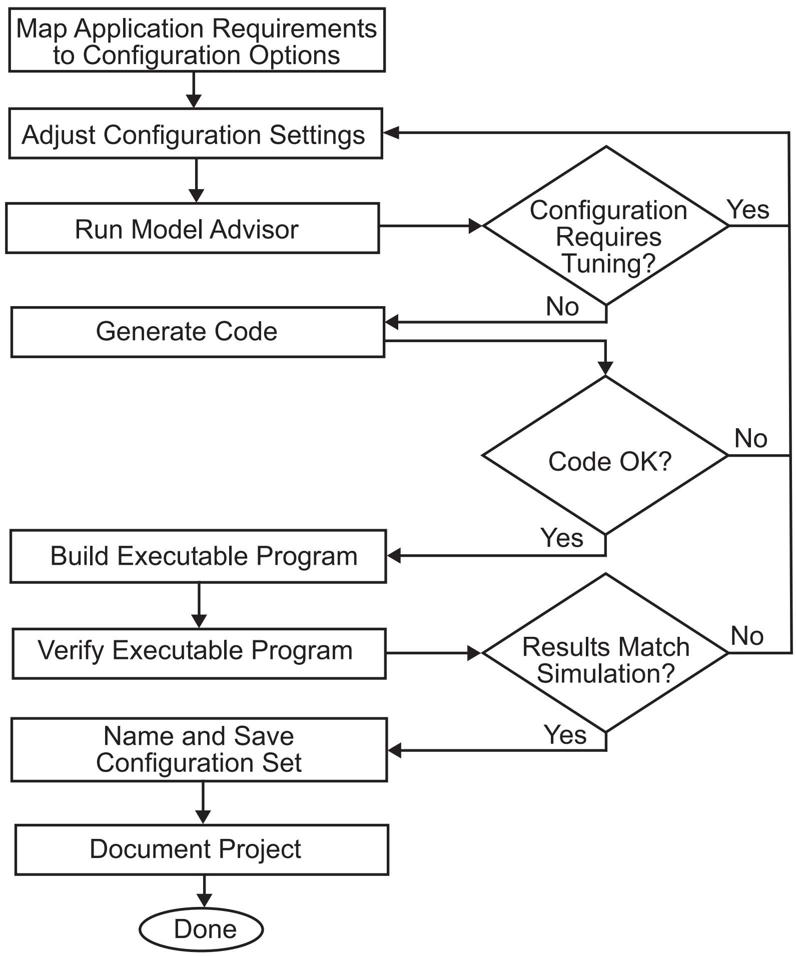
About the Workflow

Simulink support for dynamic system simulation, conditional execution of system semantics, and large model hierarchies provides an environment for modeling periodic and event-driven algorithms commonly found in embedded systems. You can generate code for most Simulink blocks and many MathWorks products.

The typical workflow for applying the Simulink Coder software to the application development process is:

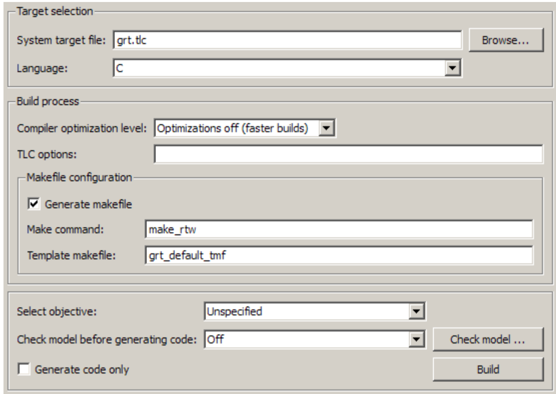
1. Map your application requirements to available configuration options.
2. Adjust configuration settings.
3. Run the Model Advisor tool.
4. Tune configuration options based on the Model Advisor report.
5. Generate code for your model.
6. Repeat steps 2 to 5, until you verify the generated code.
7. Build an executable program image.
8. Verify that the generated program produces results that are equivalent to those of your model simulation.
9. Save the configuration, and alternative configurations, with the model.
10. Use Simulink Report Generator™ to automatically document the project.

Sections following the figure describe the steps in more detail.



Mapping Application Requirements to Configuration Options

The first step in applying the Simulink Coder software to the application development process is to consider how your application requirements, particularly with respect to debugging, traceability, efficiency, and safety, map to code generation options available through the Simulink Configuration Parameters dialog box. The following graphic shows the **Code Generation** pane of the Configuration Parameters dialog box.



Parameters that you set in the various panes of the Configuration Parameters dialog box affect the behavior of a model in simulation and the code generated for the model. The Simulink Coder software automatically adjusts the available configuration parameters and their default settings based on your target selection. For example, the preceding dialog box display shows default settings for the generic real-time (GRT) target. Become familiar with the various parameters and be prepared to adjust settings to optimize a configuration for your application.

As you review the parameters, consider: questions such as the following:

- What settings will help you debug your application?
- What is the highest priority for your application — efficiency, traceability, extra safety precaution, or other criteria?
- What is the second highest priority?
- Can the priority at the start of the project differ from the priority required for the end of the project? What tradeoffs can you make?

Once you have answered these questions, you can either:

- Use the Code Generation Advisor to identify changes to model constructs and settings that improve the generated code. For more information, see “Application Objectives” in the Simulink Coder *User’s Guide*.

- Review “Recommended Settings Summary”, which summarizes the impact of each configuration option on efficiency, traceability, safety precautions, and debugging, and indicates the default (factory) configuration settings for the GRT target. For additional details, click the links in the Configuration Parameter column.

To see the settings that the Code Generation Advisor recommends, review the “Recommended Settings Summary”.

If you use a specific embedded target, a Stateflow target, or fixed-point blocks, consider the mapping of many other configuration parameters. For details, see the documentation specific to your target environment.

Adjusting Configuration Settings

Once you have mapped your application requirements to configuration parameter settings, adjust the settings accordingly. In “Recommended Settings Summary”, using the Default column in the mapping tables, identify the configuration parameters to modify. Then, open the Configuration Parameters dialog box or Model Explorer and make adjustments.

Note You also can use `get_param` and `set_param` to individually access most configuration parameters both interactively and in scripts. The relevant configuration parameters are listed in the “Parameter Reference” in the Simulink Coder documentation.

Run the Model Advisor

Before you generate code, it is good practice to run the Model Advisor. Based on a list of options that you select, this tool analyzes your model and its parameter settings. The tool then generates results that list findings with information on how to fix and improve the model and its configuration.

To start the Model Advisor, in your model window, select **Analysis > Model Advisor > Model Advisor**. A new window opens listing specific diagnostics that you can individually select or clear. Some examples of the diagnostics are:

- Identify blocks that generate expensive saturation and rounding code

- Check optimization settings
- Identify questionable software environment specifications

The Model Advisor is particularly useful for identifying aspects of your model that limit code efficiency or impede deployment of production code.

For more information on using the Model Advisor, see “Advice About Optimizing Models for Code Generation” in the Simulink Coder documentation.

Generating Code

After fine-tuning your model and its parameter settings, you can generate code. Typically, the first time through the process of applying Simulink Coder software for an application, you want to generate code without compiling and linking it into an executable program. Some reasons for not compiling and linking the code are:

- Inspecting the generated code. Is the Simulink Coder code generator creating what you expect?
- Integrating custom handwritten code.
- Experimenting with configuration option settings.

You specify code generation by selecting the **Generate code only** check box available on the **Code Generation** pane of the Configuration Parameters dialog box (changing the label of the **Build** button to **Generate code**). The code generator then analyzes the block diagram that represents your model, generating C code, and placing the resulting files in a build folder within your current working folder.

After generating the code, inspect it. Is it what you expected? If not, determine what model and configuration changes to make, rerun the Model Advisor, and regenerate the code. When you are satisfied with the generated code, build an executable program image, as described in “Building an Executable Program” on page 1-16.

For details on the **Generate code only** option, see “Generate code only”.

Verifying the Generated Code

Verify whether the generated code behaves as expected, generates expected results, and meets performance requirements by using these verification techniques:

- "Log Data for Analysis"
- "Simulation and Code Comparison"

Building an Executable Program

When you are satisfied with the code generated for your model, build an executable program image. If the **Generate code only** option on the **Code Generation** pane of the Configuration Parameters dialog box is selected, clear it. This action changes the label of the **Generate code** button back to **Build**.

To initiate a build, click the **Build** button. The code generator:

1. Compiles the model — The Simulink Coder software analyzes your block diagram (and models referenced by Model blocks) and compiles an intermediate hierarchical representation in a file called `*model*.rtw`.

2. Generates C code — The Target Language Compiler reads `*model*.rtw`, translates it to C code, and places the C file in a build folder within your working folder.

   When you click **Generate code** processing stops. See "Generating Code" on page 1-15.

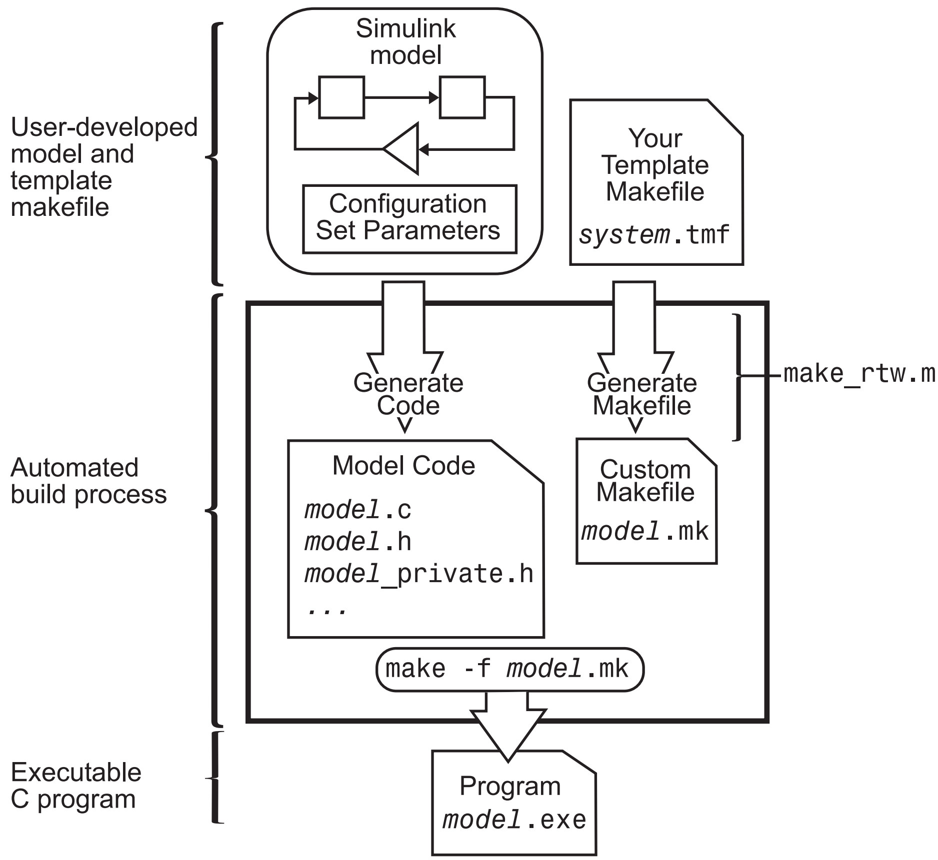
3. Generates a customized makefile — The Simulink Coder software constructs a makefile from a target makefile template and writes it in the build folder.

4. Generates an executable program — Instructs your system's `make` utility to use the generated makefile to compile the generated source code, link object files and libraries, and generate an executable program file called *`model`* (UNIX) or *`model`*`.exe` (Microsoft Windows). The makefile places the executable image in your working folder.

If you select **Create code generation report** on the **Code Generation > Report** pane, a navigable summary of source files is produced when the model is built. The report files occupy folder `html` in the build folder. The reports provide links to generated source files. Report contents vary depending on the target.

If the software detects code generation constraints for your model, it issues warning or error messages.

The following figure illustrates the complete process. The box labeled "Automated build process" highlights portions of the process that the Simulink Coder software executes.



In the Configuration Parameters dialog box, in the **Build process** section of the **Code Generation** pane, the MATLAB command file specified by the **Make command** field controls an internal portion of the build process. By default, the name of the command file is `make_rtw`. The build process invokes this file for most targets. Options specified in this field are passed into the makefile-based build process. In some cases, targets customize the `make_rtw` command. However, preserve the arguments used by the function.

Although the command may work for a standalone model, if you use the `make_rtw` command at the command line you might get an error. For example, if you have multiple models open, verify that:

- The current subsystem contains the model that you want to build. You can find the current subsystem by entering `gcs` in the MATLAB Command Window.
- In the Configuration Parameters dialog box, the **Make command** specified for the target environment is `make_rtw`.
- The model includes Model blocks. Models containing Model blocks do not build by using `make_rtw` directly.

To build (or generate code for) a model from the MATLAB Command Window, use one of the following `rtwbuild` commands, where *`model`* is the name of the model:

```
rtwbuild model
rtwbuild('model')
```

Verifying the Executable Program

Once you have an executable image, run the image and compare the results to the results of your model simulation.

1. Log output data produced by simulation runs.
2. Log output data produced by executable program runs.
3. Compare the results of the simulation and executable program runs.

Does the output match? Can you explain any differences? Do you need to eliminate any differences? You might need to revisit and possibly fine-tune your block and configuration parameter settings.

For an example, see "Verifying the Generated Code" on page 1-16.

Naming and Saving the Configuration Set

When you close a model, save it to preserve your configuration settings (unless your recent changes are dispensable). If you want to maintain several alternative configurations for a model (e.g., GRT and Rapid Simulation targets, inline parameters on/off, different solvers, etc.), you can set up a configuration set for each set of configuration parameters and give each set an identifying name. You can do this easily in Model Explorer.

To name and save a configuration:

1. Open Model Explorer from the model window by selecting **Model Explorer > View**.
2. In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.
3. Under the mode name, click the `Configuration (active)` node.

   The Configuration Parameters dialog box opens in the right pane.
4. In the **Configuration Parameters** pane, in the **Name** field, type a name you want to give the current configuration.
5. Click **Apply**. In the **Model Hierarchy** pane, the name of the active configuration changes to the name that you typed.
6. Save the model.

Adding and Copying Configuration Sets. You can save the model with more than one configuration so that you can instantly reconfigure it at a later time. Copy the active configuration to a new one, or add a new one, then modify and name the new configuration:

1 Open Model Explorer from your model window by selecting **Model Explorer > View**.

2 In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.

3 To add a new configuration set, while the model is selected in the **Model Hierarchy** pane, from the **Add** menu, select **Configuration Set** or on the toolbar, click the yellow gear icon:

In the **Model Hierarchy** pane, you see a new configuration set named `Configuration`.

4 To copy an existing configuration set, in the **Model Hierarchy** pane, right-click its name and drag it to the + sign in front of the model name.

In the **Model Hierarchy** pane, you see a new configuration set with a numeral (for example, `1`) appended to its name.

5 If you want, rename the new configuration by right-clicking it, selecting **Properties**, and in the Configuration Parameters dialog box that opens, type the new name in the **Name** field. Then click **Apply**.

6 Make the new configuration the active one. In the **Model Hierarchy** pane, right-click the new configuration. From the context menu, select **Activate**.

In the right pane, the content of the **Is Active** field changes from **no** to **yes**.

7 Save the model.

Documenting the Project

Consider documenting the design and implementation details of your project to facilitate:

- Project verification and validation.
- Collaboration with other individuals or teams, particularly if dependencies exist.

- Archiving the project for future reference.

Use the Simulink Report Generator software to document a code generation project. You can generate a comprehensive Rich Text Format (RTF), Extensible Markup Language (XML), or Hypertext Markup Language (HTML) report that includes:

- Model name and version
- Simulink Coder product version
- Date and time the code generator created the code
- List of generated source and header (include) files
- Optimization and Simulink Coder target selection and build process configuration settings
- Mapping of subsystem numbers to subsystem labels
- Listings of generated and custom code for the model

To generate a code generation report, see the example `rtwdemo_codegenrpt` and "Document Generated Code with Simulink Report Generator". For details about the Report Generator, see "Simulink Report Generator".

MATLAB Code with Simulink Model

You might use both MATLAB code and Simulink models for a Model-Based Design project if you:

- Start by using MATLAB to develop an algorithm for research and early development.
- Later want to integrate the algorithm into a graphical model for system deployment and verification.

Benefits of this approach include:

- Richer system simulation environment
- Ability to verify the MATLAB code

- Simulink Coder and Embedded Coder® C/C++ code generation for the model and MATLAB code

The following table summarizes how to generate C or C++ code, using this approach, and identifies where you can find more information.

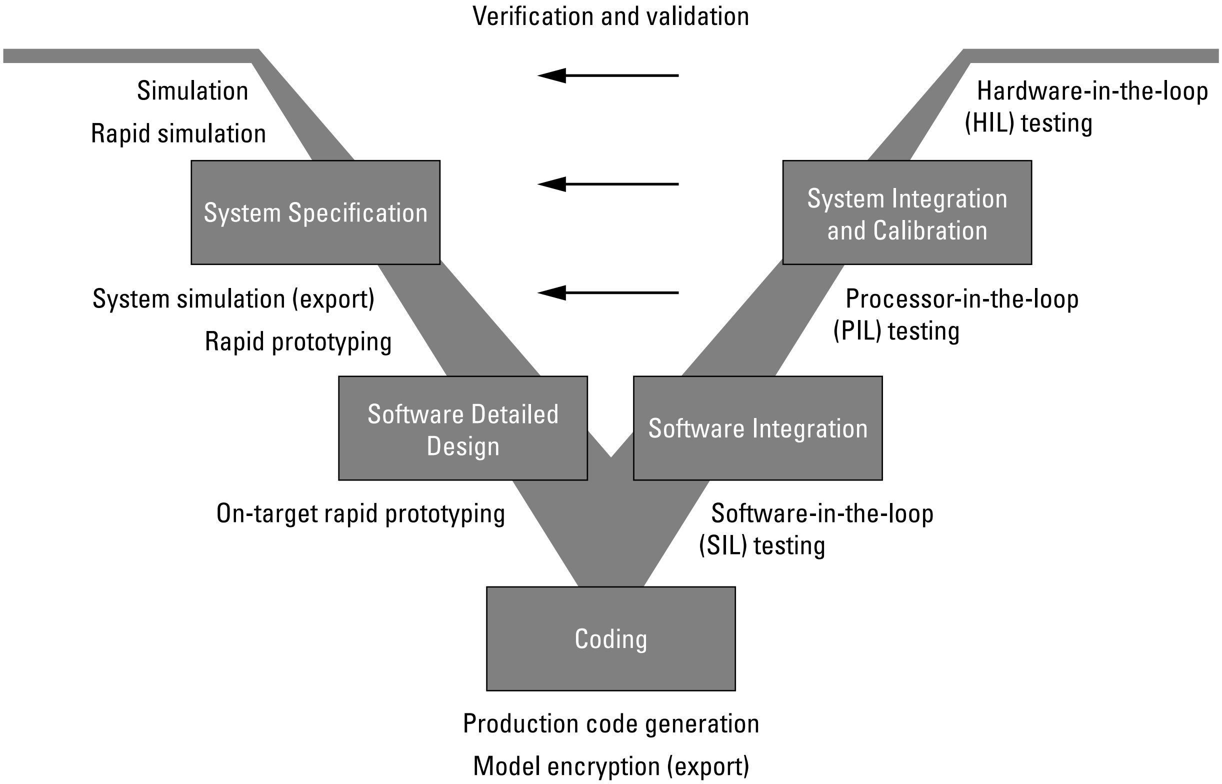
If you develop algorithms using...	You generate code by...	For more information, see...
Code generation from MATLAB and Simulink	Including MATLAB code in Simulink models or subsystems by using the MATLAB Function block. To use this block, you can do one of the following: • Copy your code into the block. • Call your code from the block by referencing files on the MATLAB path.	Code generation from MATLAB documentation MATLAB Function block in the Simulink documentation

V-Model for System Development

In this section...

What Is the V-Model?

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



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 at each step.

Types of Simulation and Prototyping in the V-Model

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

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
Purpose	Test and validate functionality of concept model	Refine, test, and validate functionality of concept model in nonreal time	Test new ideas and research	Refine and calibrate designs during development process
Execution hardware	Host computer	Host computer Standalone executable runs outside of MATLAB and Simulink environments	PC or nontarget hardware	Embedded computing unit (ECU) or near-production hardware
Code efficiency and I/O latency	Not applicable	Not applicable	Less emphasis on code efficiency and I/O latency	More emphasis on code efficiency and I/O latency
Ease of use and cost	Can simulate component (algorithm or controller) and environment (or plant) Normal mode simulation in Simulink enables you to access, display, and tune data during verification Can accelerate Simulink simulations with Accelerated and	Easy to simulate models of hybrid dynamic systems that include components and environment models Ideal for batch or Monte Carlo simulations Can repeat simulations with varying data sets, interactively or programmatically with scripts,	Might require custom real-time simulators and hardware Might be done with inexpensive off-the-shelf PC hardware and I/O cards	Might use existing hardware, thus less expensive and more convenient

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
	Rapid Accelerated modes	without rebuilding the model Can connect to Simulink to monitor signals and tune parameters		

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

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

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

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

Mapping of Code Generation Goals to the V-Model

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

- Documenting and Validating Requirements on page 1-28
- Developing a Model Executable Specification on page 1-30
- Developing a Detailed Software Design on page 1-33
- Generating the Application Code on page 1-37
- Integrating and Verifying Software on page 1-39
- Integrating, Verifying, and Calibrating System Components on page 1-42

Documenting and Validating Requirements

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

Documenting and Validating Requirements (Continued)

Goals	Related Product Information	Examples

Run standalone rapid simulations
 	
Run batch or Monte-Carlo simulations
 	
Repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model
 	
Tune parameters and monitor signals interactively
 	
Simulate models for hybrid dynamic systems that include components and an environment or plant that requires variable-step solvers and zero-crossing detection
	"Rapid Simulation"
	
	"Host/Target Communication"
		`rtwdemo_rsim_param_survey_-script`
		`rtwdemo_rsim_batch_script`
		`rtwdemo_rsim_param_tuning`

Distribute simulation runs across multiple computers	"SystemTest™" "MATLAB Distributed Computing Server™" "Parallel Computing Toolbox™"	

Developing a Model Executable Specification

Goals	Related Product Information	Examples
Produce design artifacts for algorithms that you develop in MATLAB code for reviews and archiving	" MATLAB Report Generator"	
Produce design artifacts from Simulink and Stateflow models for reviews and archiving	"Simulink Report Generator" "System Design Description" — Simulink Report Generator	`rtwdemo_codegenrpt`
Add one or more components to another environment for system simulation Refine a component model Refine an integrated system model Verify functionality of a model in nonreal time Test a concept model	"Real-Time System Rapid Prototyping"	
Schedule generated code	"Scheduling" "Handle Asynchronous Events"	`rtwdemos`, select **Multirate Support** folder

Specify function boundaries of systems	"Subsystems"	`rtwdemo_atomic`
		`rtwdemo_ssreuse`
		`rtwdemo_filepart`
		`rtwdemo_export_functions`

Specify components and boundaries for design and incremental code generation	"Component-Based Modeling" — Simulink Coder "Component-Based Modeling" — Embedded Coder	`rtwdemo_mdlreftop`

Developing a Model Executable Specification (Continued)

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

Developing a Model Executable Specification (Continued)

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

Developing a Detailed Software Design

Goals	Related Product Information	Examples
Refine a model design for representation and storage of data in generated code	"Data Representation" — Simulink Coder "Data Representation " — Embedded Coder	

Select a deployment code format	"Target" — Simulink Coder
	"Target"— Embedded Coder
	"Sharing Utility Code" — Embedded Coder
	"AUTOSAR Code Generation" — Embedded Coder
		`rtwdemo_counter`
		`rtwdemo_async`
		"AUTOSAR Examples" in the Embedded Coder documentation

Specify target hardware settings	"Target" — Simulink Coder "Target"— Embedded Coder	`rtwdemo_targetsettings`
Design model variants	"Variant Systems" — Simulink "Variant Systems" — Embedded Coder	
Specify fixed-point algorithms in Simulink, Stateflow, and the MATLAB language subset for code generation	"Data Types and Scaling" — Fixed-Point Designer "Fixed-Point Code Generation" — Fixed-Point Designer	`rtwdemo_fixpt1` `rtwdemo_fuelsys_fxp_publish`
Convert a floating-point model or subsystem to a fixed-point representation	"Conversion Using Simulation Data" — Fixed-Point Designer "Conversion Using Range Analysis" — Fixed-Point Designer	`fxpdemo_fpa`
Iterate to obtain an optimal fixed-point design, using autoscaling	"Data Types and Scaling" — Fixed-Point Designer	`fxpdemo_feedback`

Developing a Detailed Software Design (Continued)

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

Developing a Detailed Software Design (Continued)

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

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Examples
Refine the concept model of your component or system Test and validate the model functionality in real time Test the hardware Obtain real-time profiles and code metrics for analysis and sizing based on your embedded processor Assess the feasibility of the algorithm based on integration with the environment or plant hardware	"Deployment" — Simulink Coder "Deployment" — Embedded Coder "Code Execution Profiling" — Embedded Coder "Static Code Metrics" — Embedded Coder	`rtwdemos`, select **Desktop IDEsDesktop TargetsEmbedded IDEsEmbedded Targets**

Generate source code for your models, integrate the code into your production build environment, and run it on existing hardware	"Code Generation" — Simulink Coder
	"Code Generation" — Embedded Coder
		`rtwdemo_counter`
		`rtwdemo_fcnprotoctrl`
		`rtwdemo_cppencap`
		`rtwdemo_async`
		"AUTOSAR Examples" in the Embedded Coder documentation

Integrate existing externally written C or C++ code with your model for simulation and code generation	"Block Creation" — Simulink "External Code Integration" — Simulink Coder "External Code Integration" — Embedded Coder	`rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code**
Generate code for on-target rapid prototyping on specific embedded microprocessors and IDEs	"Real-Time and Embedded Systems"	In `rtwdemos`, select one of the following: **Desktop IDEs**, **Desktop Targets**, **Embedded IDEs**, or **Embedded Targets**

Generating the Application Code

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

Generating the Application Code (Continued)

Goals	Related Product Information	Examples
Verify generated code for MISRA C®[3] and other run-time violations	"MISRA C Guidelines" — Embedded Coder Documentation for Polyspace® Products	
Protect the intellectual property of component model design and generated code Generate a binary file (shared library)	"Protected Model" — Simulink "Shared Object Libraries" — Embedded Coder	
Generate a MEX-file S-function for a model or subsystem so that it can be shared with a third-party vendor	"Generated S-Function Block"	
Generate a shared library for a model or subsystem so that it can be shared with a third-party vendor	"Shared Object Libraries" — Embedded Coder	
Test generated production code with an environment or plant model to verify a conversion of the model to code	"Software-in-the-Loop (SIL) Simulation" — Embedded Coder	`rtwdemo_sil_pil_script`

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

Generating the Application Code (Continued)

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

Integrating and Verifying Software

Goals	Related Product Information	Examples
Integrate existing externally written C or C++ code with a model for simulation and code generation	“Block Creation” — Simulink “External Code Integration” — Simulink Coder “External Code Integration” — Embedded Coder	`rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code**
Connect to data interfaces for generated C code data structures	“Data Exchange” — Simulink Coder “Data Exchange” — Embedded Coder	`rtwdemo_capi` `rtwdemo_asap2`
Control the generation of code interfaces so that external software can compile, build, and invoke the generated code	“Function and Class Interfaces” — Embedded Coder	`rtwdemo_fcnprotoctrl` `rtwdemo_cppencap`
Export virtual and function-call subsystems	“Export Code Generated from Model to External Application” — Embedded Coder	`rtwdemo_export_functions`

Integrating and Verifying Software (Continued)

Goals	Related Product Information	Examples
Include target-specific code	"Code Replacement" — Embedded Coder	`rtwdemo_crl_script`
Customize and control the build process	"Build Process"	`rtwdemo_buildinfo`
Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer	"Relocate Code to Another Development Environment"	`rtwdemo_buildinfo`
Integrate software components as a complete system for testing in the target environment	"Component Verification"	
Generate source code for integration with specific production environments	"Code Generation" — Simulink Coder "Code Generation" — Embedded Coder	`rtwdemo_async` "AUTOSAR Examples" in the Embedded Coder documentation
Integrate code for a specific run-time environment, using specialized function libraries	"Code Replacement" — Embedded Coder	`rtwdemo_crl_script`
Enter special instructions or tags for postprocessing by third-party tools or processes	"Customize Post-Code-Generation Build Processing"	`rtwdemo_buildinfo`
Integrate existing externally written code with code generated for a model	"Block Creation" — Simulink "External Code Integration" "External Code Integration" — Embedded Coder	`rtwdemos`, select **Integrating with C Code** or **Integrating with C++ Code**

Integrating and Verifying Software (Continued)

Goals	Related Product Information	Examples
Connect to data interfaces for the generated C code data structures	"Data Exchange" — Simulink Coder "Data Exchange" — Embedded Coder	`rtwdemo_capi` `rtwdemo_asap2`
Customize and control the build process	"Build Process"	`rtwdemo_buildinfo`
Create a zip file that contains generated code files, static files, and dependent data for building the generated code in an environment other than your host computer	"Relocate Code to Another Development Environment"	`rtwdemo_buildinfo`
Schedule the generated code	"Time-Based Scheduling"	`rtwdemos`, select **Multirate Support**
Verify object code files in a target environment	"Software-in-the-Loop (SIL) Simulation"	`rtwdemo_sil_pil_script`

Set up and run PIL tests on your target system	"Processor-in-the-Loop (PIL) Simulation"	`rtwdemo_sil_pil_script`
		`rtwdemo_custom_pil_script`
		`rtwdemo_rtiostream_script`
		See the list of `supported hardware` for the Embedded Coder product on the MathWorks Web site, and then find an example for the related product of interest

Integrating, Verifying, and Calibrating System Components

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

Getting Started Examples

Generate C Code for a Model

In this section...

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 2-13 and "Tune Parameters and Monitor Signals During Execution" on page 2-19.

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. To open the model, in the command window, type:

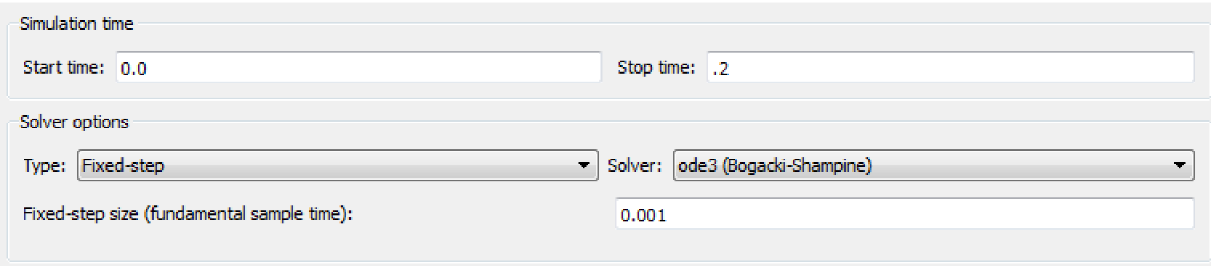
```
rtwdemo_secondOrderSystem
```

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.

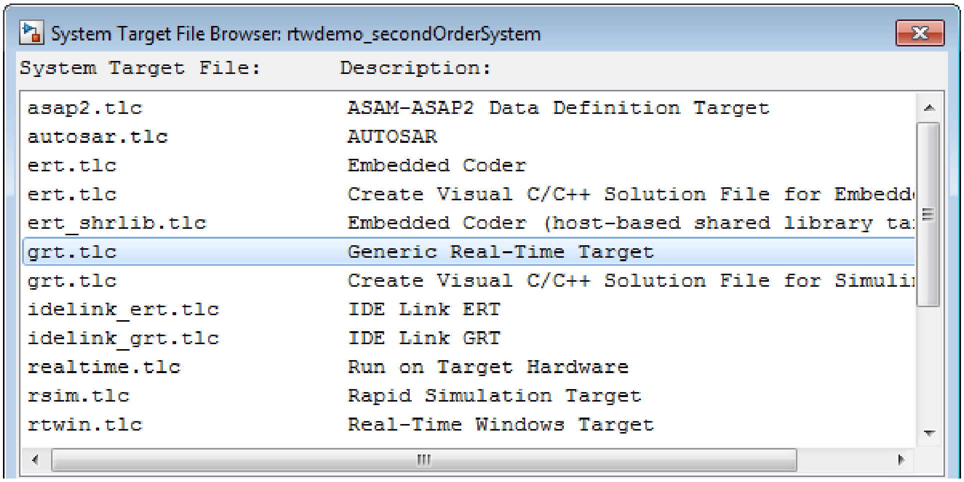


Code Generation Target

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 automatically generates 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, the following 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. From the **Select objective** drop-down list, select `Execution efficiency`. Click **Apply**.

3. Click **Check model**.

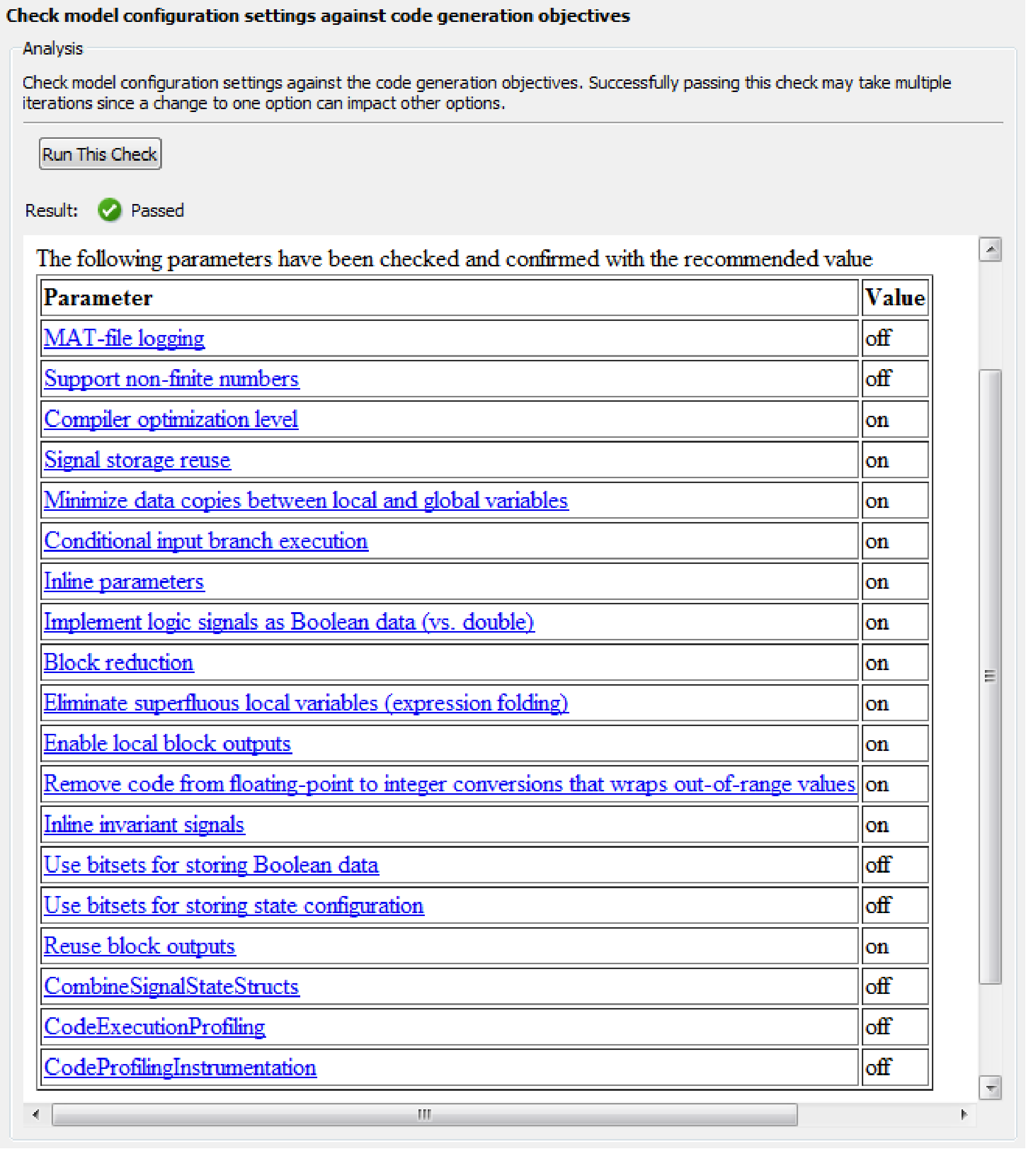
4. In the System Selector dialog box, click **OK** to run checks on the model.

   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 software-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`.



Check model configuration settings against code generation objectives

Analysis

Check model configuration settings against the code generation objectives. Successfully passing this check may take multiple iterations since a change to one option can impact other options.

Run This Check

Result: Passed

The following parameters have been checked and confirmed with the recommended value

Parameter	Value
MAT-file logging	off
Support non-finite numbers	off
Compiler optimization level	on
Signal storage reuse	on
Minimize data copies between local and global variables	on
Conditional input branch execution	on
Inline parameters	on
Implement logic signals as Boolean data (vs. double)	on
Block reduction	on
Eliminate superfluous local variables (expression folding)	on
Enable local block outputs	on
Remove code from floating-point to integer conversions that wraps out-of-range values	on
Inline invariant signals	on
Use bitsets for storing Boolean data	off
Use bitsets for storing state configuration	off
Reuse block outputs	on
CombineSignalStateStructs	off
CodeExecutionProfiling	off
CodeProfilingInstrumentation	off

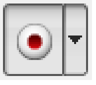
Close the Code Generation Advisor.

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.

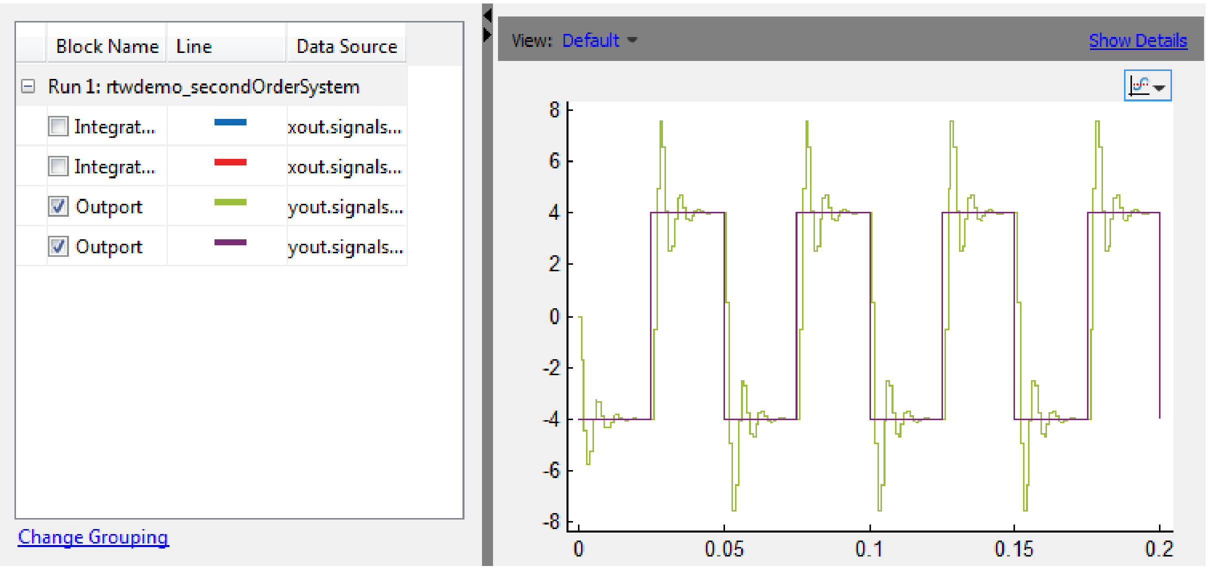
1. To log data to the Simulation Data Inspector, on the Simulink Editor toolbar, verify that the **Record** button is selected.



2. Simulate the model.

3. When the simulation is done, in the Simulink Editor, click the link in the notification bar to open the Simulation Data Inspector.

4. Expand the run and then select the Outport block 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 2-13.

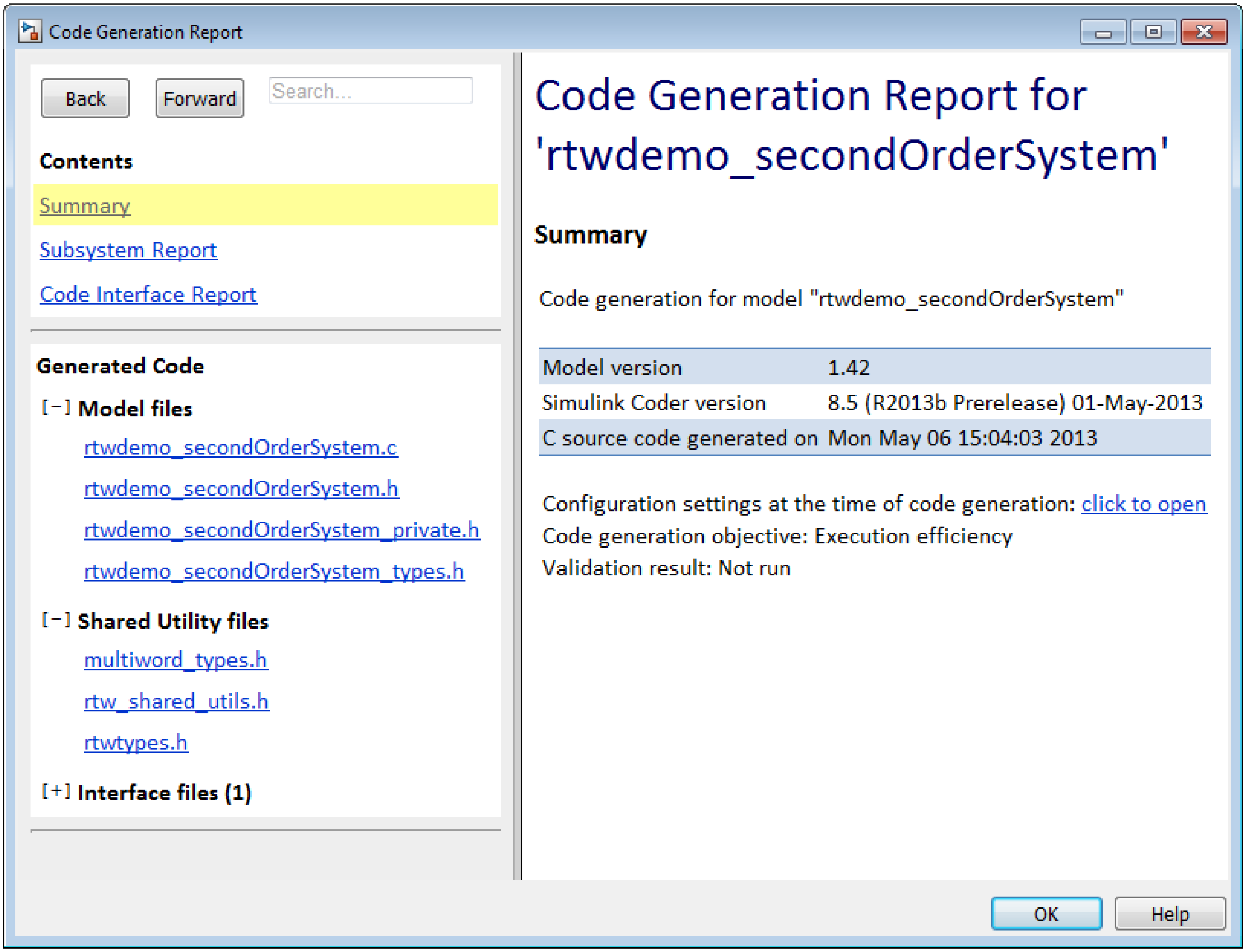
Generate Code

1 Select the **Generate code only** check box.

2 Click **Apply**.

3 Click **Generate code**.

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` folder. The code generation report includes:

- 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	Called 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	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	Called once
Arguments	None
Return value	None
Header file	rtwdemo_secondOrderSystem.h

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

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Outport	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:

```
/* Block signals (auto storage) */
extern B_rtwdemo_secondOrderSystem_T rtwdemo_secondOrderSystem_B;

/* Continuous states (auto storage) */
extern X_rtwdemo_secondOrderSystem_T rtwdemo_secondOrderSystem_X;

/* External outputs (root outports fed by signals with auto storage) */
extern ExtY_rtwdemo_secondOrderSyste_T rtwdemo_secondOrderSystem_Y;

/* Model entry point functions */
extern void rtwdemo_secondOrderSystem_initialize(void);
extern void rtwdemo_secondOrderSystem_step(void);
extern void rtwdemo_secondOrderSystem_terminate(void);

/* Real-time Model object */
extern RT_MODEL_rtwdemo_secondOrderS_T *const rtwdemo_secondOrderSystem_M;
```

Block Outputs
Continuous States
Model Output
Entry Points
Timing Data

The next example shows how to build an executable. See “Build and Run Executable” on page 2-13.

Build and Run Executable

In this section...

Simulink Coder supports several methods for building an executable:

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

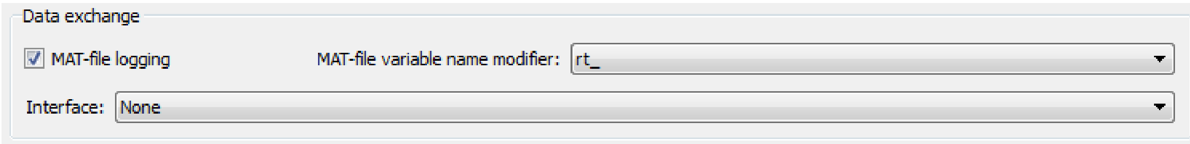
The code generation target that you select for your model determines the build process controls that are presented to you. The example model uses the GRT code generation target, which enables the toolchain based controls. This example shows how to build an executable using the toolchain controls, and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

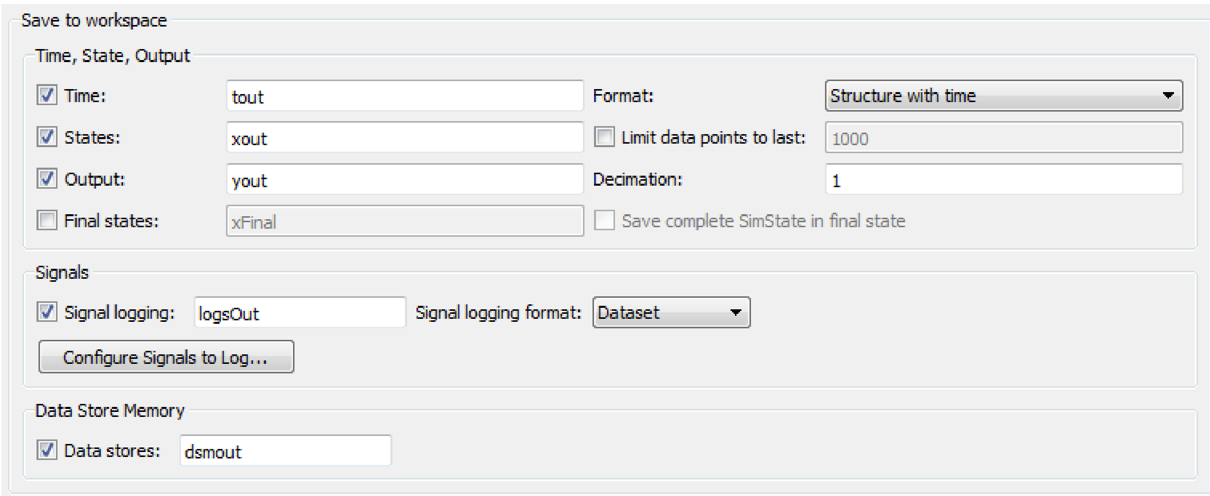
Configure Model to Output Data to MAT-File

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, select the **Code Generation > Interface** pane.
2. Under **Data exchange**, the **MAT-file logging** check box is selected.
3. The **MAT-file variable name modifier** parameters is specified as `rt_`.



4 Click the **Data Import/Export** pane and specify the **Save to workspace** parameters, as shown here.



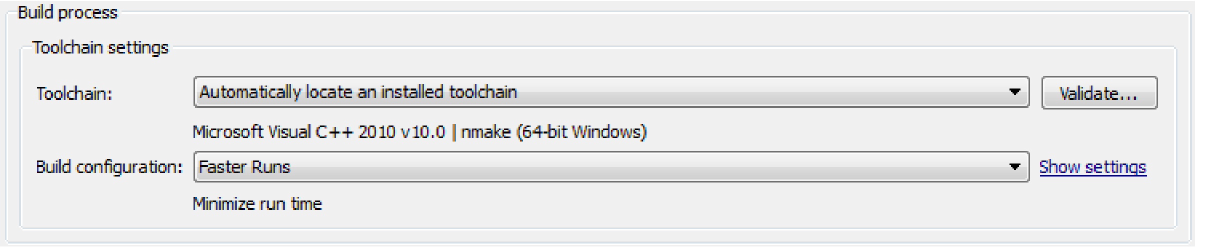
5 Click **Apply**.

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 **Code Generation** pane, in the **Build process** section, specify the **Toolchain** and **Build configuration** parameters.



Here, the default toolchain is `Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)`.

2 To verify your toolchain, click **Validate**.

   The Validation Report indicates if the checks passed.

3 Clear the **Generate code only** check box.

4 Click **Apply**.

5 To build the executable, click **Build** (previously the **Generate code** button).

The MATLAB command window displays the following output:

```
### Starting build procedure for model: rtwdemo_secondOrderSystem
### Successful completion of build procedure for model: rtwdemo_secondOrderSystem
```

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. For Windows, type

```
!rtwdemo_secondOrderSystem
```

For Linux, type

```
!./rtwdemo_secondOrderSystem
```

MATLAB displays the following 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

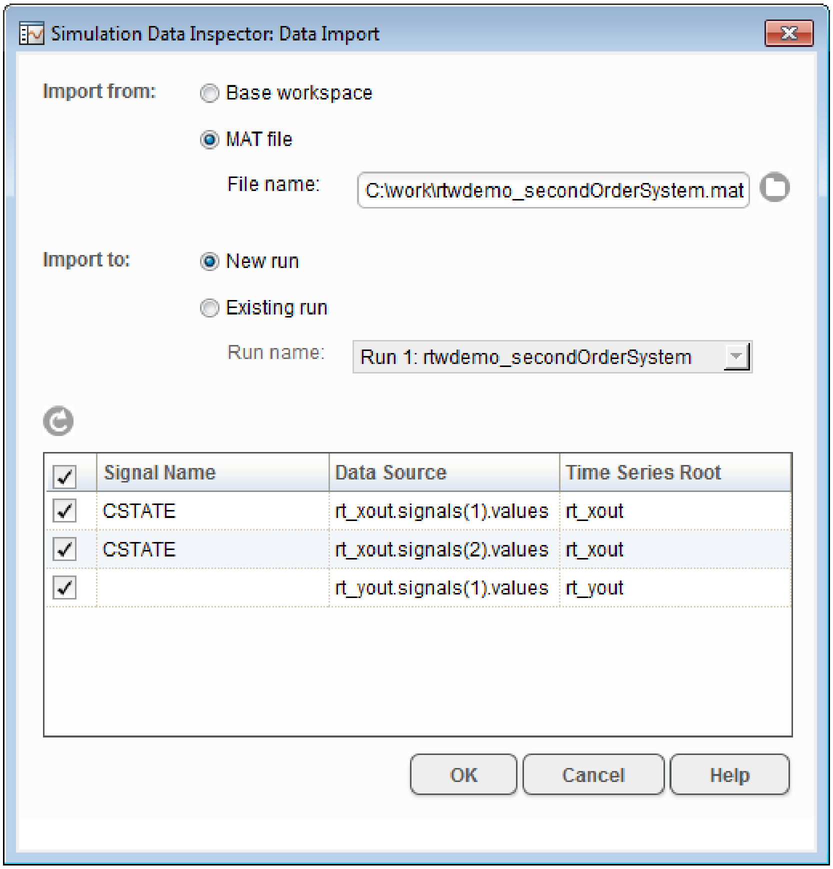
This example shows you how to import data into the Simulation Data Inspector, and then compare the executable results with the simulation results. If you have not already recorded the simulation data to the Simulation Data Inspector, follow the instructions in “Simulate the Model” on page 2-7.

1 If the Simulation Data Inspector is not already open, in the Simulink Editor, click the down arrow of the **Record** button and select `Simulation Data Inspector`.

2 To open the Import Data dialog box, click the **Import Data** button.

3 In the Import Data dialog box, for **Import from**, select the **MAT-file** option button.

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

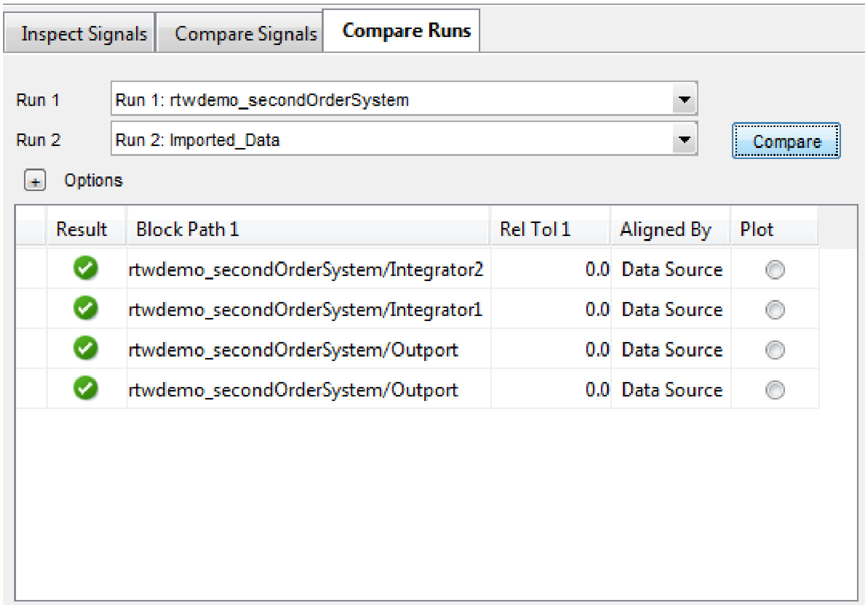


Click **OK**.

4 On the **Inspect Signals** tab, select signals from each run to view them in the right pane.

5 Select the **Compare Runs** tab.

6 Specify `Run 1` and `Run 2`. Click **Compare**.



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 using Simulink as an interface for testing. See "Tune Parameters and Monitor Signals During Execution" on page 2-19.

Tune Parameters and Monitor Signals During Execution

In this section...

This example shows how to tune parameters and monitor signals of the standalone executable using the example model, `rtwdemo_secondOrderSystem`. Using Simulink External Mode simulation, Simulink communicates to a standalone executable that can be running in real time or nonreal time depending on the target code generation configuration. The example model uses the default GRT target implementation. Simulink communicates to a separate and standalone non-real-time executable running on the host computer over a TCP/IP communication link.

Before working through this example, consider doing these getting started tutorials: "Generate C Code for a Model" on page 2-2 and "Build and Run Executable" on page 2-13.

Set Up Signal Monitoring

To view signal data during execution, you can use Scope blocks in your model. For this example, the Scope block is sufficient for viewing the output from an external program.

To avoid placing many scopes throughout your model, you can use a Floating Scope block. By default, the code generator attempts to implement all signals in local memory. A floating scope cannot access local memory. Therefore, you must place signals in memory that are available to the floating scope. Once signals are in global memory, you can add signals to a floating scope. To place a signal into global memory in the generated code you can add a test point to a signal or you can configure your model to place all signals into global memory.

Add a Test Point to a Signal

If your model is large, placing all signals into global memory generates less efficient code. Consider using test points which place only specified signals into global memory. A signal specified as a test point is defined in the block I/O data structure. Specify a test point for a signal by selecting the **Test point** check box in the Signal Properties dialog box.

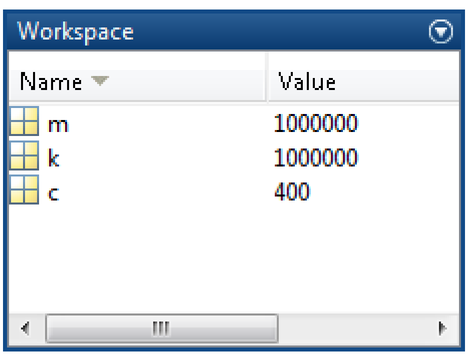
Place All Signals into Global Memory

You can configure the model such that the code generator places each signal in the global block I/O data structure in the generated code. On the **Optimization > Signals and Parameters** pane, clear the **Signal storage reuse** check box. All signals are placed into global memory in the generated code, which makes the signal data available to a floating scope. You can add signals to a Floating Scope block using the Signal Selector dialog box.

Set Up Tunable Parameters

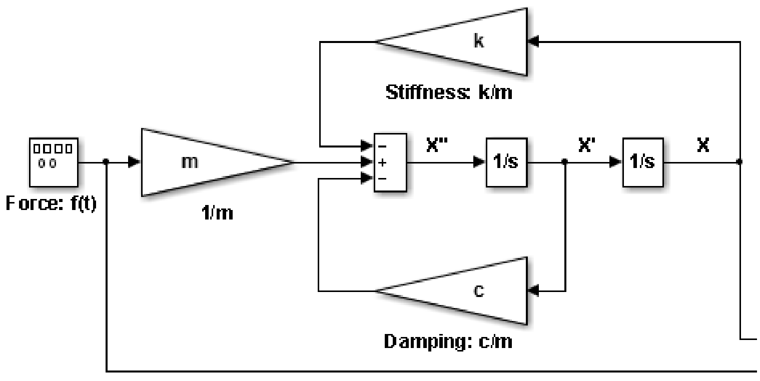
You can tune parameters directly in the Block Parameter dialog box while an external program is running. Alternatively, you can tune parameters that are in the base workspace.

1 Declare the following variables in the base workspace.



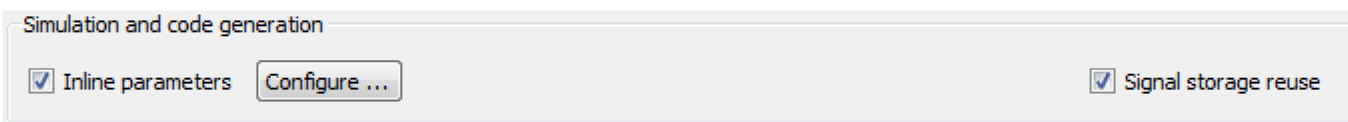
2 For each Gain block in the model, double-click the block to open the Block Parameters dialog box.

3 Replace the **Gain** parameter value with the name of the corresponding workspace variable.



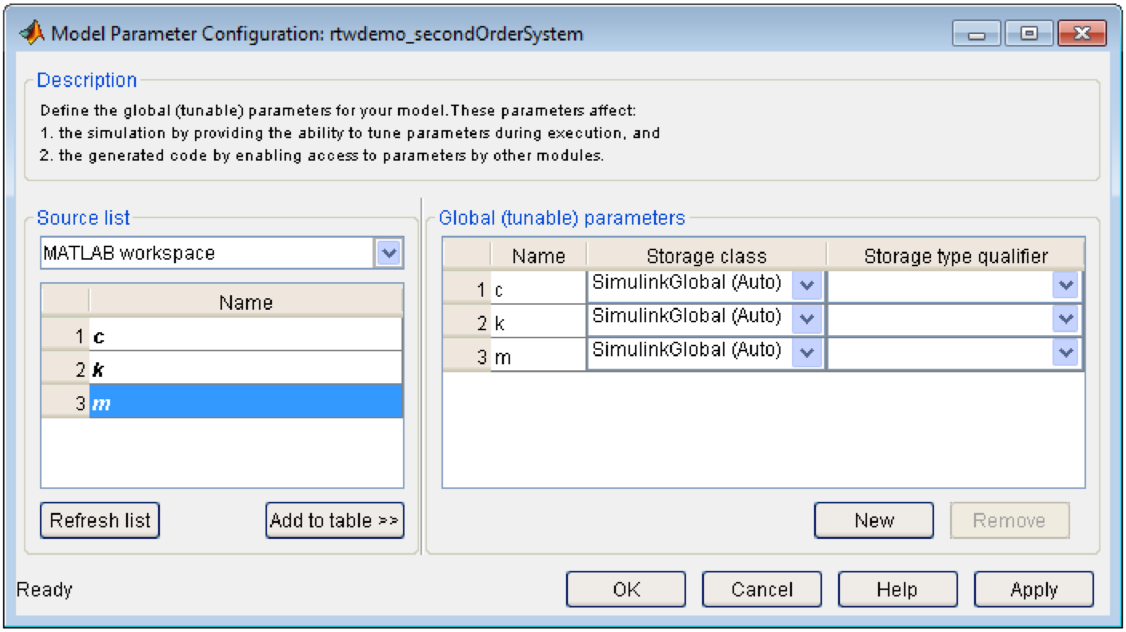
To use tunable parameters, the variables must be preserved by name in the generated code. Before generating code, you must inline all parameters in the model before generating code.

1 In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, select **Inline parameters**. The code generator numerically inlines parameter values into the generated code to maximize code efficiency. Therefore, you must define global tunable parameters.



2 Click **Configure** to open the Model Parameter Configuration dialog.

3 To specify the variables that you want to preserve in the code, add each variable to the **Global (tunable) parameters** table. Click a variable name in the **Source list**, and then click **Add to table**.



Each variable uses the default **Storage class** `SimulinkGlobal(Auto)`. A variable specified as a `SimulinkGlobal` is placed in the model parameter data structure in the generated code.

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

Now your model is set up to change the **Gain** parameters in the base workspace once the external program is executing.

Build the Target Executable

This example uses the default TCP/IP communication protocol for a GRT target.

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

2 For the **Interface** parameter, select `External mode`.

3 Click **Apply**.

4 To build the executable, on the **Code Generation** pane, click **Build**. Alternatively, from the model diagram, press **Ctrl-B**.

The code generation process creates the executable, `rtwdemo_secondOrderSystem.exe`, and places it in the current folder.

The tunable parameters and signal parameters are defined in `rtwdemo_secondOrderSystem.h`.

```
/* Parameters (auto storage) */
struct P_rtwdemo_secondOrderSystem_T_ {
  real_T c;                            /* Variable: c
                                        * Referenced by: '<Root>/Damping: c//m'
                                        */

  real_T k;                            /* Variable: k
                                        * Referenced by: '<Root>/Stiffness: k//m'
                                        */

  real_T m;                            /* Variable: m
                                        * Referenced by: '<Root>/1//m'
                                        */

};
```

```
/* Block signals (auto storage) */
typedef struct {
  real_T X;                            /* '<Root>/Integrator2' */
  real_T Forceft;                      /* '<Root>/Force: f(t)' */
  real_T m;                            /* '<Root>/1//m' */
  real_T X_h;                          /* '<Root>/Integrator1' */
  real_T Dampingcm;                    /* '<Root>/Damping: c//m' */
  real_T Stiffnesskm;                  /* '<Root>/Stiffness: k//m' */
  real_T X_p;                          /* '<Root>/Sum' */
} B_rtwdemo_secondOrderSystem_T;
```

Run External Mode Target Program

Open a command window and go to the folder where the executable is saved. Run the executable:

```
>> rtwdemo_secondOrderSystem -tf inf
```

The `tf` option overrides the stop time so that the executable runs indefinitely.

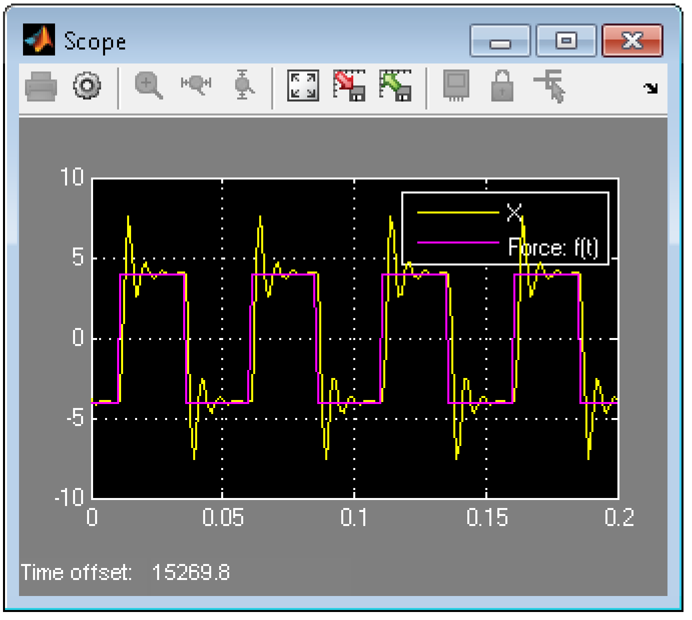
Connect Simulink to the External Process

To connect `rtwdemo_secondOrderSystem` to the running executable:

1. From the Simulink Editor, select **Code > External Mode Control Panel**.

2. Click **Connect** to establish a connection.

View the data from the external process in the scope.

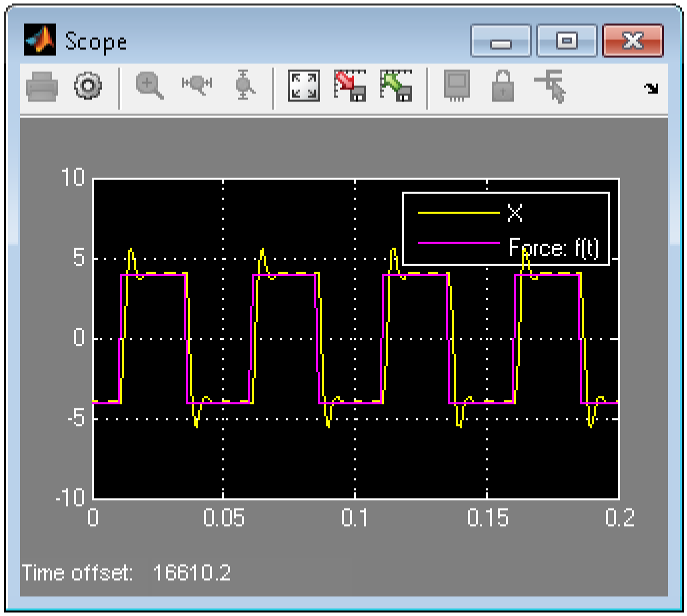


Parameter Tuning

You can now change block parameter settings in Simulink and observe the effects the changes have on the target program.

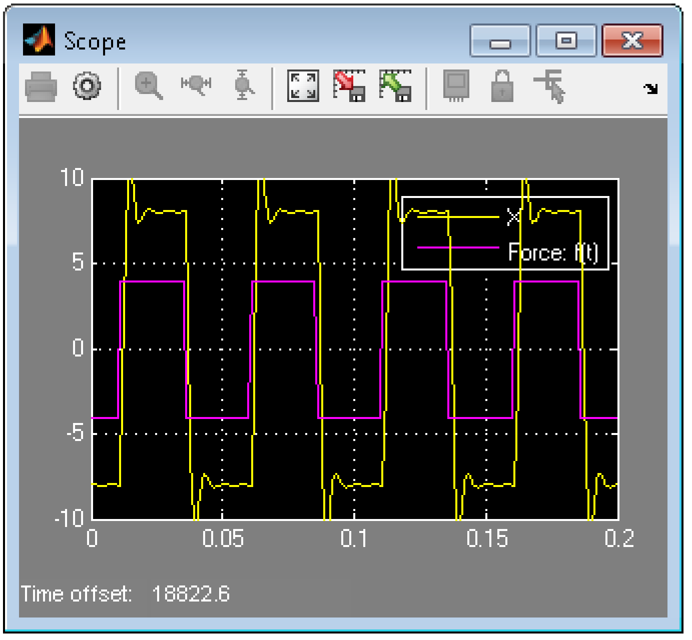
1. Change the value of base workspace variable `c` from `400` to `800`.

2. Perform an update diagram, **Ctrl-D**. After changing the value of a base workspace variable, you must perform an update diagram in order to see the change in the ongoing simulation output.



3 At the MATLAB command line, change the mass parameter, `m`, from `1.0E-6` kg to `2.0E-6` kg.

4 Perform an update diagram, **Ctrl-D**.



5 To stop the simulation, on the External Mode Control Panel dialog box, click **Disconnect**.

Next Steps

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

To...	See....
Model multirate systems	"Scheduling"
Create multiple model configuration sets and share configuration parameter settings across models	"Configuration Sets"
Control how signals are stored and represented in the generated code	"Signal Storage Basics" and "Signal Objects"

To...	See....
Generate block parameter storage declarations and interface block parameters to your code	“Tunable Parameter Storage Classes” and “Parameter Objects”
Store data separate from the model	“Data Objects”
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 C++ compatible code	“Language”
Export an ASAP2 file containing information about your model during the code generation process	“ASAP2 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	“Data Interchange Using C API”
Create a protected model that hides all block and line information to share with third-party	“Model Protection”
Customize the build process	“Build Process”
Create a custom block	“Block Authoring”
Create your own target	“Target Development”

Index