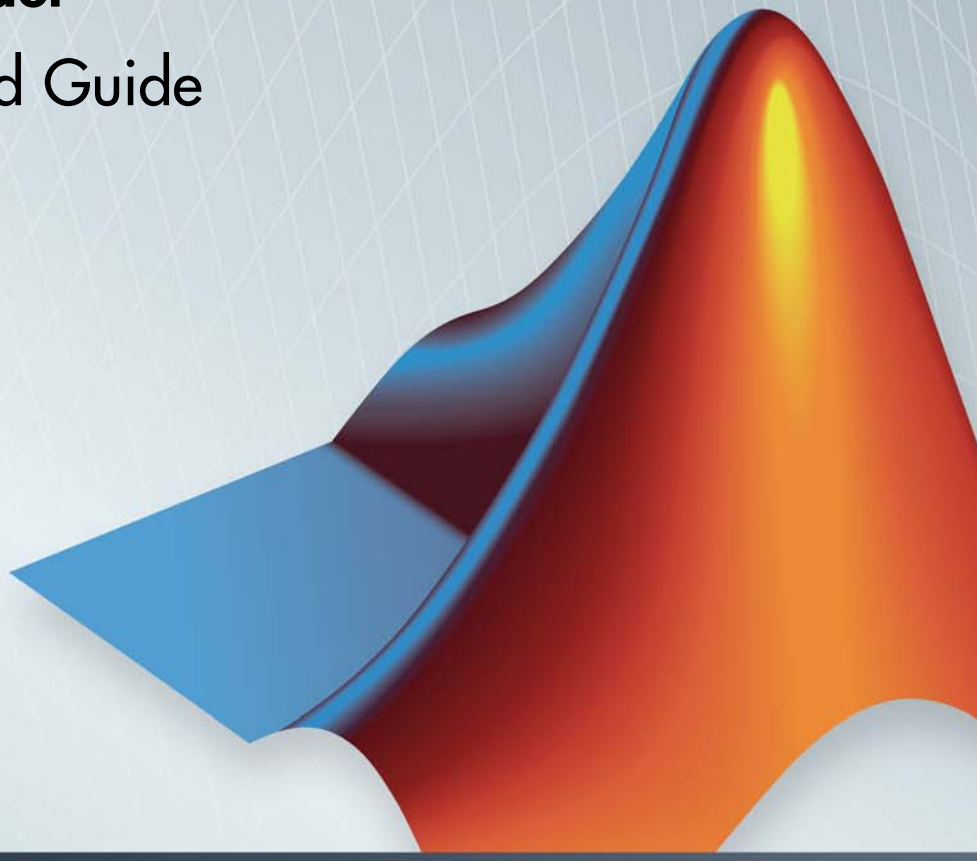
Simulink® Coder™

Getting Started Guide

R2013a



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

Trademarks

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

Patents

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

Revision History

April 2011	Online only	New for Version 8.0 (Release 2011a)
September 2011	Online only	Revised for Version 8.1 (Release 2011b)
March 2012	Online only	Revised for Version 8.2 (Release 2012a)
September 2012	Online only	Revised for Version 8.3 (Release 2012b)
March 2013	Online only	Revised for Version 8.4 (Release 2013a)

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

Index

1

Product Overview

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 nonreal-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 1-24.

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.

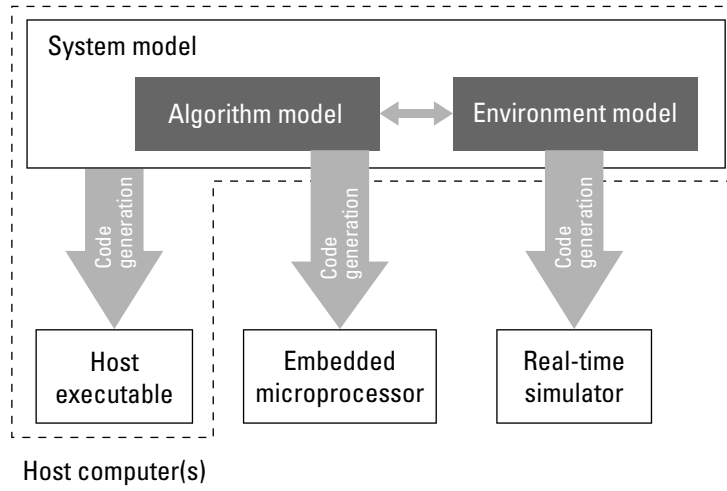
Target Environment	Description
	• Use a full-featured RTOS • Be driven by basic interrupts • Use rate monotonic scheduling provided with code generation

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.

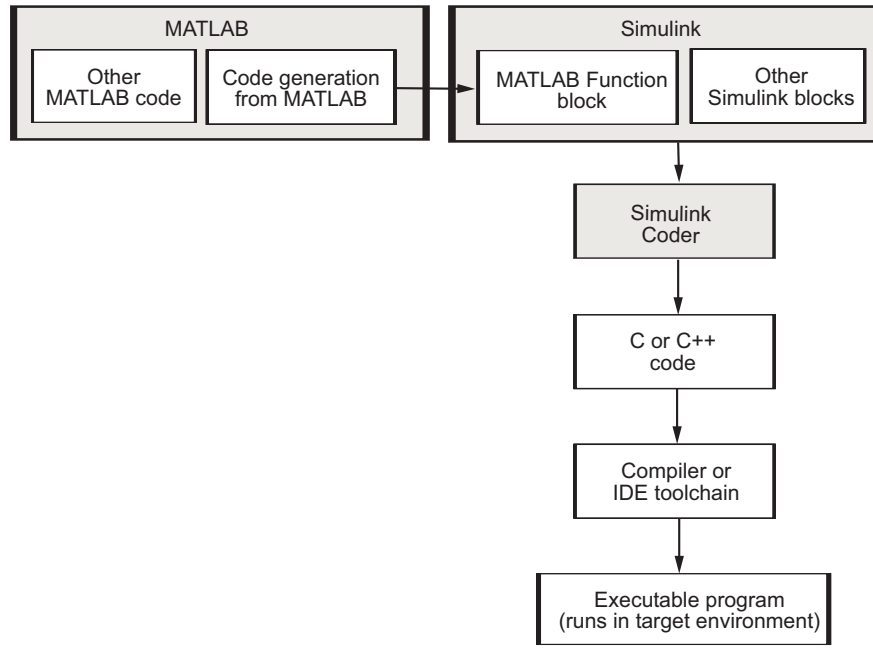
Algorithm Development Options

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

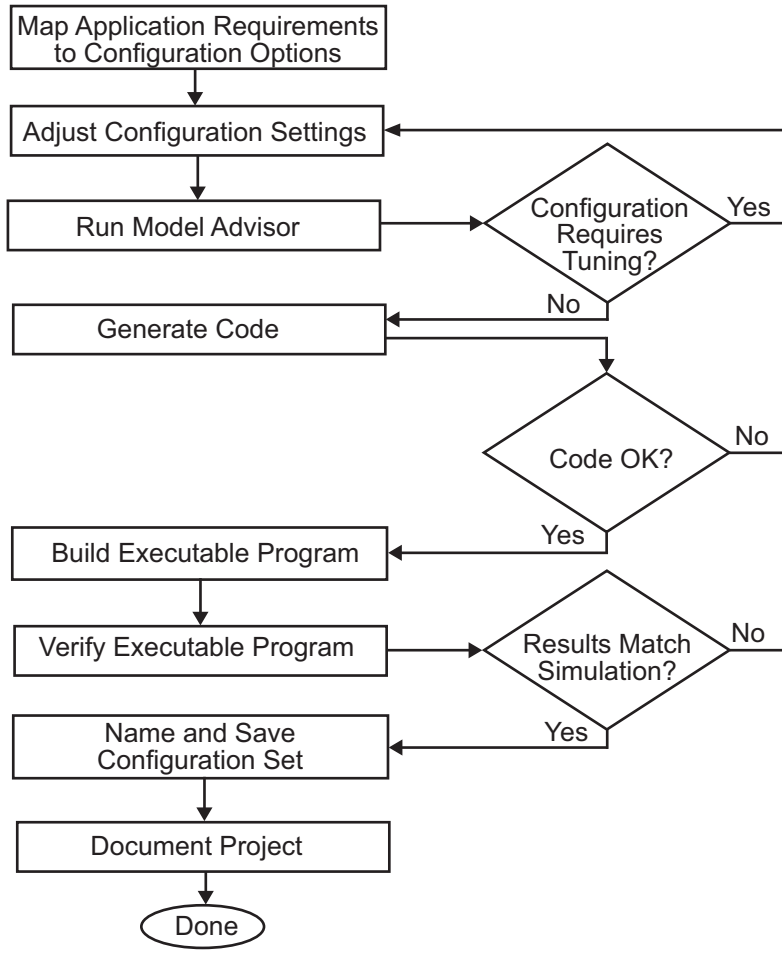
About the Workflow

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:

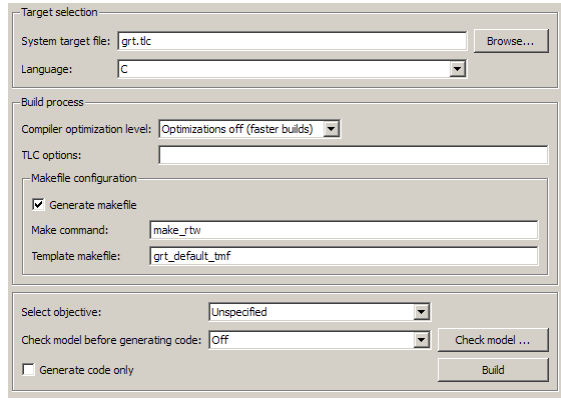
1. Map your application requirements to available configuration options.
2. Adjust configuration settings.
3. Run the Model Advisor tool.
4. Tune configuration options based on the Model Advisor report.
5. Generate code for your model.
6. Repeat steps 2 to 5, until you verify the generated code.
7. Build 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. Using the Default column in “Mapping Application Requirements to the Solver Pane”, 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.

Running the Model Advisor

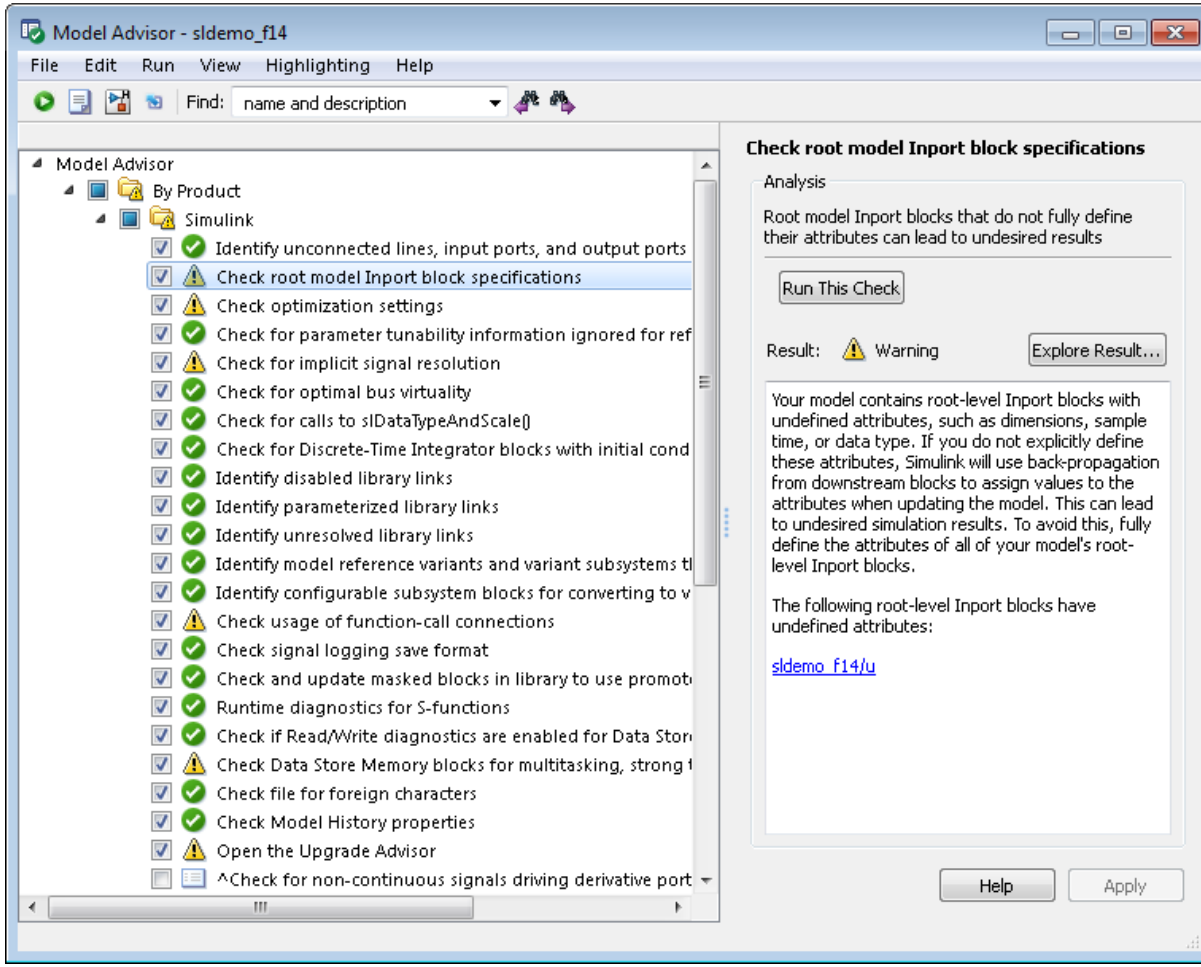
Before you generate code, it is good practice to run the Model Advisor. Based on a list of options that you select, this tool analyzes your model and its parameter settings. The tool then generates results that list findings with information on how to fix and improve the model and its configuration.

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. The following figure shows the Model Advisor.



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 1-17.

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 1-16.

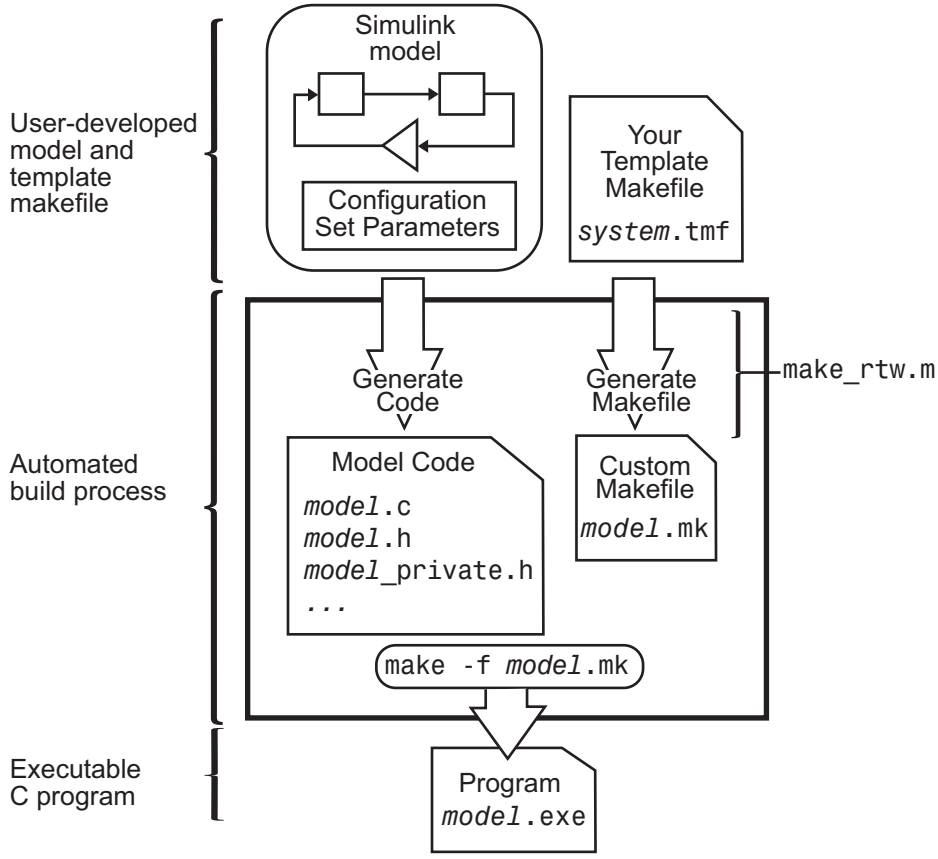
3. Generates a customized makefile — The Simulink Coder software constructs a makefile from a target makefile template and writes it in the build folder.

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.

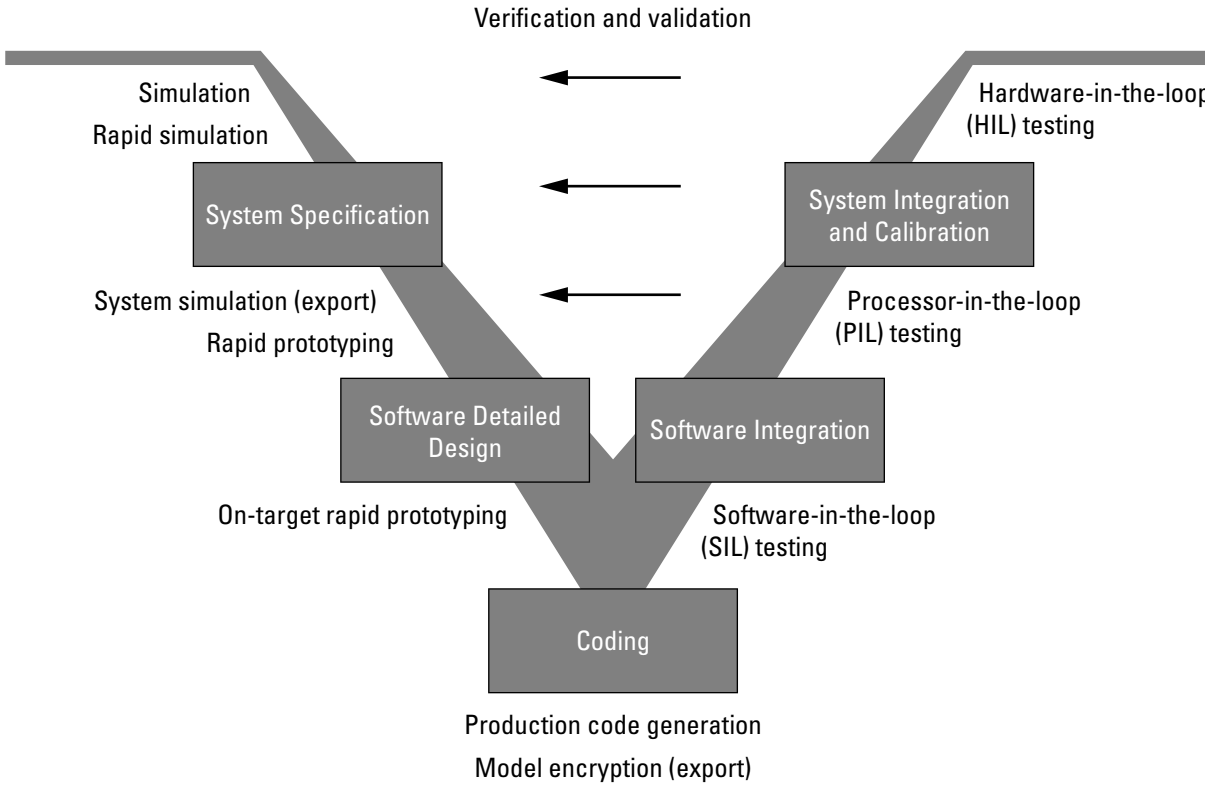
If you develop algorithms using...	You generate code by...	For more information, see...
Code generation from MATLAB and Simulink	Including MATLAB code in Simulink models or subsystems by using the MATLAB Function block. To use this block, you can do one of the following: • Copy your code into the block. • 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 1-29
- Developing a Model Executable Specification on page 1-31
- Developing a Detailed Software Design on page 1-34
- Generating the Application Code on page 1-38
- Integrating and Verifying Software on page 1-40
- Integrating, Verifying, and Calibrating System Components on page 1-43

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 Telelogic® 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
	"Generating Code for AUTOSAR Software Components" — 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`

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Examples
Iterate to obtain an optimal fixed-point design, using autoscaling	"Data Types and Scaling" — Fixed-Point Designer	`fxpdemo_feedback`
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`

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Examples
Check a model against custom standards or guidelines	"Consult the Model Advisor" — Simulink	
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	`rtwdemo_polyspace_script`
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`

2

Getting Started Examples

About the Examples

In this section...

Examples

The following examples will help you get started with using Simulink Coder to generate code from Simulink models and subsystems:

- "Model and Test Environment" on page 2-4
- "Configure Model and Generate Code" on page 2-16
- "Configure Data Interface" on page 2-24
- "Call External C Functions" on page 2-35

Each example focuses on a specific aspect of code generation or integration and is self-contained. Skim or skip examples that do not apply to your needs.

If you are licensed to use the Embedded Coder product, "Getting Started with Embedded Coder" includes additional examples based on the same models and test harness.

Prerequisites

For these examples, you must know how to use MathWorks products to do the following:

- Create Simulink models
- Include Stateflow charts in Simulink models
- Run Simulink simulations and evaluate the results

Required Files

Each example uses a unique example model file and data set.

- Before you use each example model file, place a copy in a writable location and add it to your MATLAB path.
- As you proceed through a example, save your changes for future examination.
- To avoid potentially introducing errors, begin each example by opening a new model and loading new data.

Model and Test Environment

In this section...

About This Example

Learning Objectives

- Learn about the functional behavior of the example model.
- Learn about the role of the example test harness and its components.
- Run simulation tests on a model.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Understand subsystems and how to view subsystem details.
- Understand referenced models and how to view referenced model details.
- Ability to set model configuration parameters.

Required Files

Before you use each example model file, place a copy in a writable location and add it to your MATLAB path.

- `rtwdemo_throttlecntrl` model file
- `rtwdemo_throttlecntrl_testharness` model file

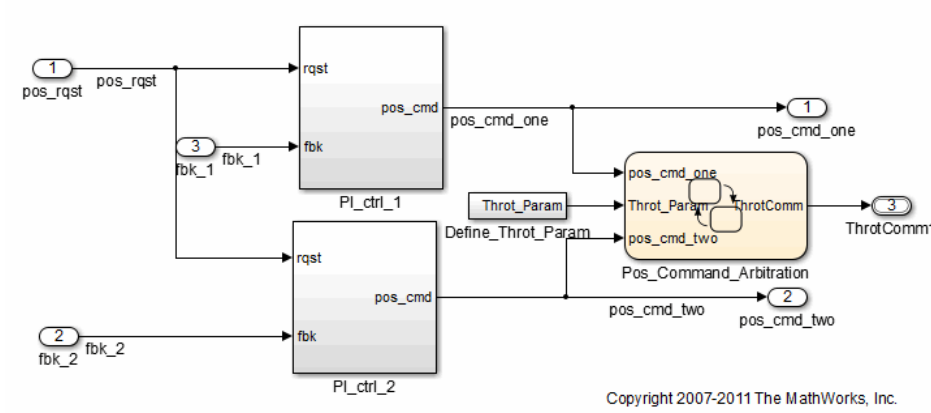
Functional Design of the Model

This example uses a simple, but functionally complete, example model of a throttle controller. The model features redundant control algorithms. The model highlights a standard model structure and a set of basic blocks in algorithm design.

View the Top Model

Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.

Note This model uses Stateflow software.



The top level of the model consists of the following elements:

Subsystems	`PI_ctrl_1`
	`PI_ctrl_2`
	`Define_Throt_Param`
	`Pos_Command_Arbitration`

Top-level input	`pos_rqst`
	`fbk_1`
	`fbk_2`

Top-level output	`pos_cmd_one`
	`pos_cmd_two`
	`ThrotComm1`

Signal routing	
Omit blocks that change the value of a signal, such as Sum and Integrator	

The layout uses a basic architectural style for models:

- Separation of calculations from signal routing (lines and buses)
- Partitioning into subsystems

You can apply this style to a wide range of models.

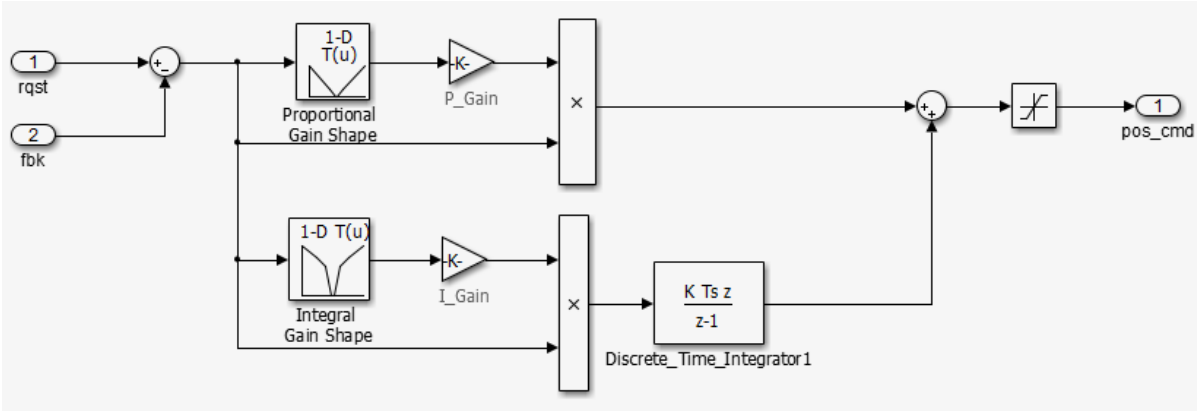
View the Subsystems

Explore two of the subsystems in the top model.

1. If not already open, open `throttlecntrl`.

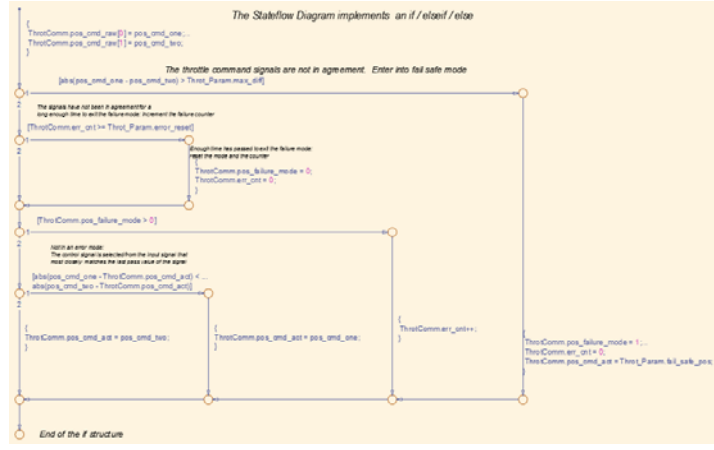
   Two subsystems in the top model represent proportional-integral (PI) controllers, `PI_ctrl_1` and `PI_ctrl_2`. At this stage, these identical subsystems, use identical data. If you have an Embedded Coder license, you can use these subsystems in a example that shows how to create reusable functions.

2. Open the `PI_ctrl_1` subsystem.



The PI controllers in the model are from a *library*, a group of related blocks or models for reuse. Libraries provide one of two methods for including and reusing models. The second method, model referencing, is described in "Simulation Test Environment" on page 2-8. You cannot edit a block that you add to a model from a library. You must edit the block in the library so that instances of the block in different models remain consistent.

3 Open the `Pos_Command_Arbitration` subsystem. This Stateflow chart performs basic error checking on the two command signals. If the command signals are too far apart, the Stateflow diagram sets the output to a `fail_safe` position.



4 Close `throttlecntrl`.

Simulation Test Environment

To test the throttle controller algorithm, you incorporate it into a *test harness*. A test harness is a model that evaluates the control algorithm and offers the following benefits:

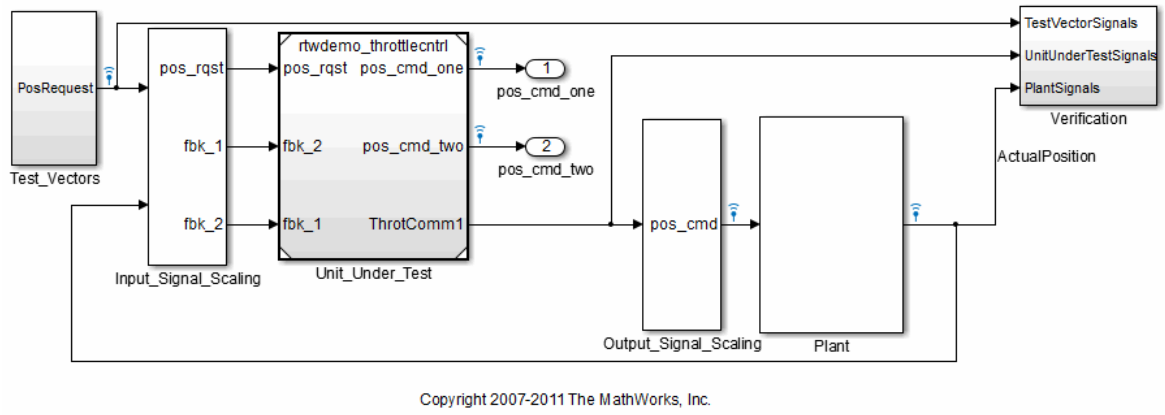
- Separates test data from the control algorithm.
- Separates the plant or feedback model from the control algorithm.
- Provides a reusable environment for multiple versions of the control algorithm.

The test harness model for this example implements a common simulation testing environment consisting of the following parts:

- Unit under test
- Test vector source
- Evaluation and logging
- Plant or feedback system
- Input and output scaling

Explore the simulation testing environment.

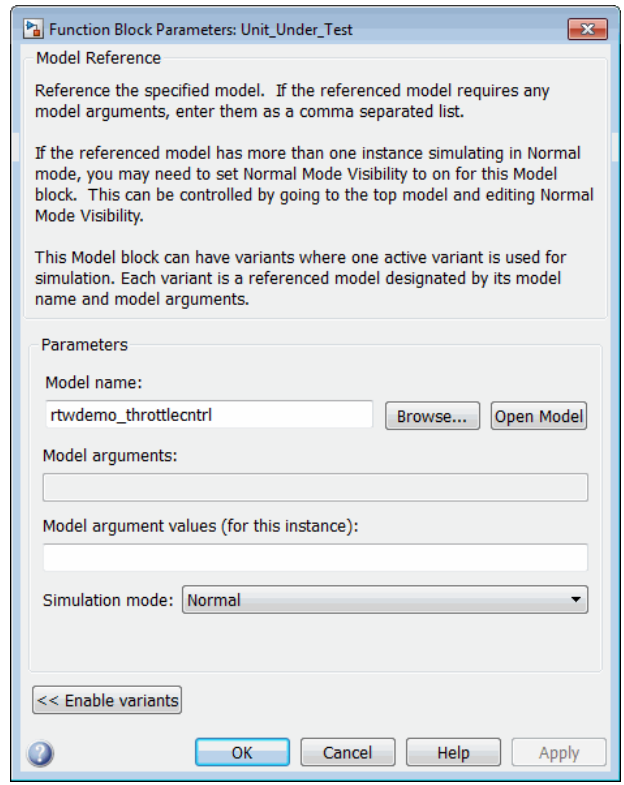
1 Open the test harness model `rtwdemo_throttlecntrl_testharness` and save a copy as `throttlecntrl_testharness` in a writable location on your MATLAB path.



2 Set up your `throttlecntrl` model as the control algorithm of the test harness.

a Open the `Unit_Under_Test` block and view the control algorithm.

b View the model reference parameters by right-clicking the `Unit_Under_Test` block and selecting **Block Parameters (ModelReference)**.



`rtwdemo_throttlecntrl` appears as the name of the referenced model.

c Change the value of **Model name** to `throttlecntrl`.

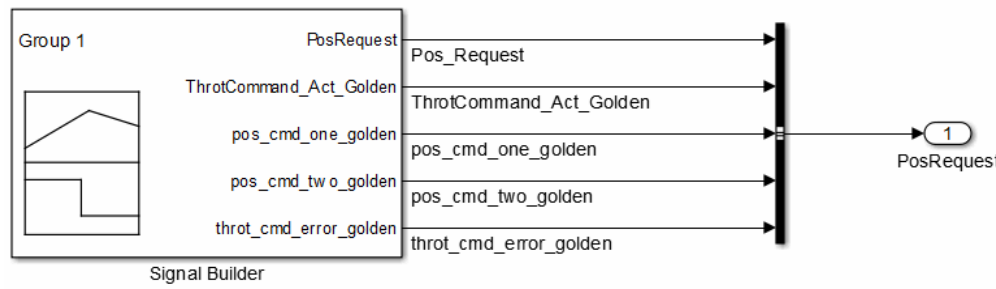
d Update the test harness model diagram by clicking **Simulation > Update Diagram**.

The control algorithm is the *unit under test*, as indicated by the name of the Model block, `Unit_Under_Test`.

The Model block provides a method for reusing components. From the top model, it allows you to reference other models (directly or indirectly) as *compiled functions*. By default, Simulink software recompiles the model when the referenced models change. Compiled functions have the following advantages over libraries:

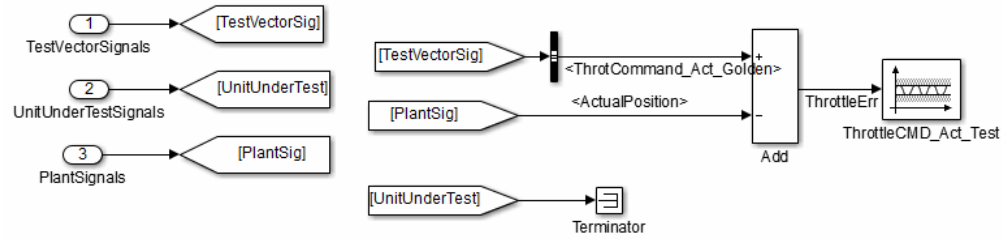
- Simulation time is faster for large models.
- You can directly simulate compiled functions.
- Simulation requires less memory. Only one copy of the compiled model is in memory, even when the model is referenced multiple times.

3 Open the *test vector source*, implemented in this test harness as the `Test_Vectors` subsystem.



The subsystem uses a Signal Builder block for the test vector source. The block has data that drives the simulation (`PosRequest`) and provides the expected results used by the `Verification` subsystem. This example test harness uses only one set of test data. Typically, you create a test suite that fully exercises the system.

4 Open the *evaluation and logging* subsystem, implemented in this test harness as subsystem `Verification`.

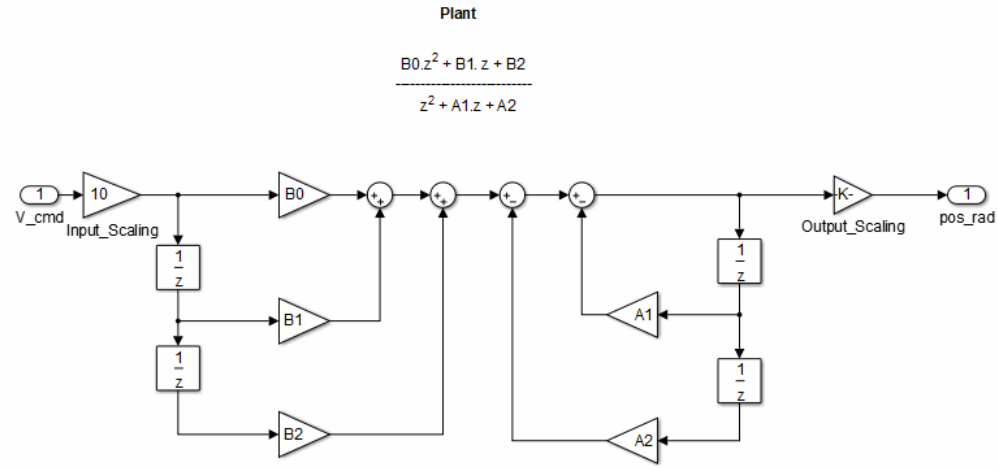


A test harness compares control algorithm simulation results against *golden data* — test results that exhibit the desired behavior for the control algorithm as certified by an expert. In the `Verification` subsystem, an

Assertion block compares the simulated throttle value position from the plant against the golden value from the test harness. If the difference between the two signals is greater than 5%, the test fails and the Assertion block stops the simulation.

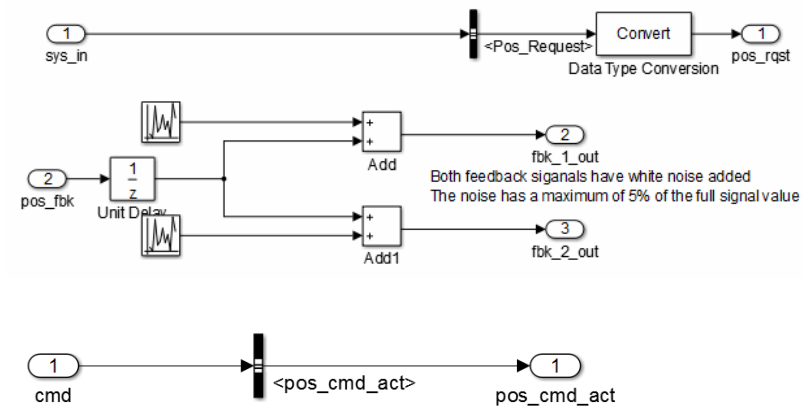
Alternatively, you can evaluate the simulation data after the simulation completes execution. You can use either MATLAB scripts or third-party tools to perform the evaluation. Post-execution evaluation provides greater flexibility in the analysis of data. However, it requires waiting until execution is complete. Combining the two methods can provide a highly flexible and efficient test environment.

5 Open the *plant or feedback system*, implemented in this test harness as the `Plant` subsystem.



The `Plant` subsystem models the throttle dynamics with a transfer function in canonical form. You can create plant models to varying levels of fidelity. It is common to use different plant models at different stages of testing.

6 Open the *input and output scaling* subsystems, implemented in this test harness as `Input_Signal_Scaling` and `Output_Signal_Scaling`.



The subsystems that scale input and output perform the following primary functions:

- Select input signals to route to the unit under test and output signals to route to the plant.
- Rescale signals between engineering units and units that are writable for the unit under test.
- Handle rate transitions between the plant and the unit under test.

7 Save and close `throttlecntrl_testharness`.

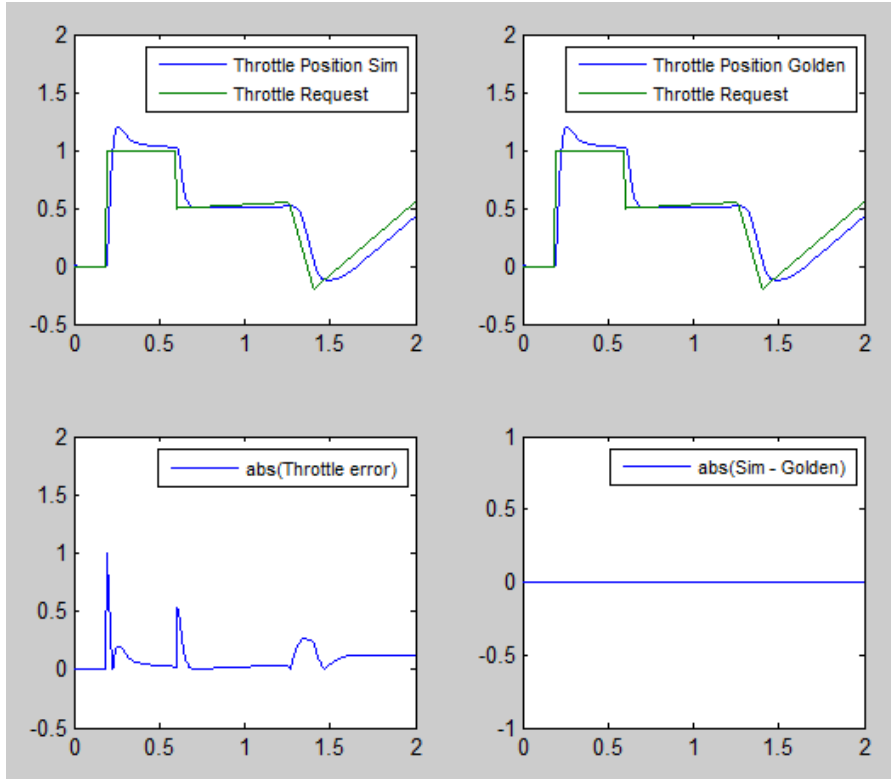
Run Simulation Tests

1 In the MATLAB Command Window, enter `mex -setup` to set up your C compiler. Specify a valid, installed compiler.

2 Check that your working folder is set to a writable folder, such as the folder into which you placed copies of the example model files.

3 Open your copy of the test harness model, `throttlecntrl_testharness`.

4 Start a test harness model simulation. When the simulation is complete, the following results should appear.



The lower-right hand plot shows the difference between the expected (golden) throttle position and the throttle position that the plant calculates. If the difference between the two values is greater than ±0.05, the simulation stops.

5 Save and close throttle controller and test harness models.

Key Points

- A basic model architecture separates calculations from signal routing and partitions the model into subsystems
- Two options for model reuse include block libraries and model referencing.

- If you represent your control algorithm in a test harness as a Model block, be sure that you specify the name of the control algorithm model in the Model Reference Parameters dialog box.
- A test harness is a model that evaluates a control algorithm and typically consists of a unit under test, a test vector source, evaluation and logging, a plant or feedback system, and input and output scaling components.
- The unit under test is the control algorithm being tested.
- The test vector source provides the data that drives the simulation which generates results used for verification.
- During verification, the test harness compares control algorithm simulation results against golden data and logs the results.
- The plant or feedback component of a test harness models the environment that is being controlled.
- When developing a test harness,
  - Scale input and output components.
  - Select input signals to route to the unit under test.
  - Select output signals to route to the plant.
  - Rescale signals between engineering units and units that are writable for the unit under test.
  - Handle rate transitions between the plant and the unit under test.
- Before running simulation or completing verification, consider checking a model with the Model Advisor.

Learn More

- “Support Model Referencing”
- “Code Generation”
- “Signal Groups”

Configure Model and Generate Code

In this section...

About This Example

Learning Objectives

- Configure a model for code generation.
- Apply model checking tools to discover conditions and configuration settings resulting in generation of inaccurate or inefficient code.
- Generate code from a model.
- Locate and identify generated code files.
- Review generated code.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Ability to set model configuration parameters.
- Ability to use the Simulink Model Advisor.

- Ability to read C code.
- Set up a C compiler. In the MATLAB Command Window, enter `mex -setup` and specify a valid, installed compiler.

Required Files

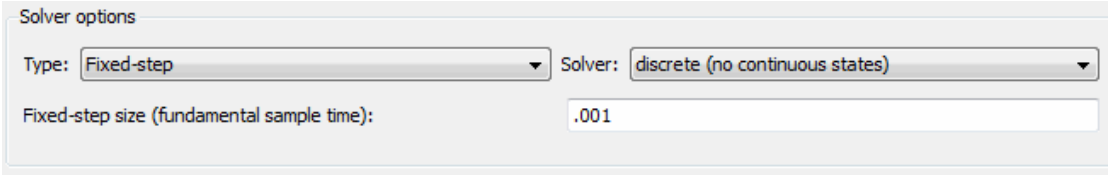
`rtwdemo_throttlecntrl` model file

Configure the Model for Code Generation

Model configuration parameters determine the method for generating the code and the resulting format.

1. Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.
2. Open the Configuration Parameters dialog box, **Solver** pane. To generate code for a model, you must configure the model to use a fixed-step solver. The following table shows the solver configuration for this example.

Parameter	Setting	Effect on Generated Code
Type	`Fixed-step`	Maintains a constant (fixed) step size, which is required for code generation
Solver	`discrete (no continuous states)`	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	`.001`	Sets the base rate; must be the lowest common multiple of all rates in the system



3 Open the **Code Generation > General** pane and note that the **System target file** is set to `grt.tlc`.

Note The GRT (Generic Real-Time Target) configuration requires a fixed-step solver. However, the `rsim.tlc` system target file supports variable step code generation.

The system target file (STF) defines a target, which is an environment for generating and building code for execution on a certain hardware or operating system platform. For example, one property of a target is code format. The grt configuration requires a fixed step solver and the rsim.tlc supports variable step code generation.

4 Open the **Code Generation > Custom Code** pane and under **Include list of additional**, select **Include directories**. Note the following path appears in the text field:

```
"$matlabroot$\toolbox\rtw\rtwdemos\EmbeddedCoderOverview\"
```

This directory includes files that are required to build an executable for the model.

5 Close the dialog box.

Save Your Model Configuration as a MATLAB Function

You can save the settings of model configuration parameters as a MATLAB function by using the `getActiveConfigSet` function. In the MATLAB Command Window, enter:

```
thcntrlAcs = getActiveConfigSet('throttlecntrl');
thcntrlAcs.saveAs('throttlecntrlModelConfig');
```

You can then use the resulting function (for example, `throttlecntrlModelConfig`) to:

- Archive the model configuration.
- Compare different model configurations by using differencing tools.
- Set the configuration of other models.

For example, you can set the configuration of model `myModel` to match the configuration of the throttle controller model by opening `myModel` and entering:

```
myModelAcs = throttlecntrlModelConfig;
attachConfigSet('myModel', myModelAcs, true);
setActiveConfigSet('myModel', myModelAcs.Name);
```

For more information, see “Save a Configuration Set” and “Load a Saved Configuration Set” in the Simulink documentation.

Check the Model for Adverse Conditions and Code Generation Settings

Before generating code for a model, use the Simulink Model Advisor to check the model for conditions and configuration settings that can result in inaccurate or inefficient code.

1. Open `throttlecntrl`.
2. Start the Model Advisor by selecting **Analysis > Model Advisor > Model Advisor**. A dialog box opens showing the model system hierarchy.
3. Click `throttlecntrl` and then click **OK**. The Model Advisor window opens.
4. Expand **By Product** and **Embedded Coder**. By default, checks that do not trigger an Update Diagram, with one exception, are selected.
5. In the left pane, select the remaining checks and select **Embedded Coder**.
6. In the right pane, select **Show report after run** and click **Run Selected Checks**. The report shows a **Run Summary** that flags three warnings.

7 Review the report. The warnings highlight issues for embedded systems. At this point, you can ignore them.

Generate Code for the Model

1 Open `throttlecntrl`.

2 In the Configuration Parameters dialog box, select **Code Generation > Generate code only** and click **Apply** .

3 On the **Code Generation > Report** pane, select **Create code generation report** and click **Apply**.

4 Return to the **Code Generation** pane, click **Generate code**, and watch the messages that appear in the MATLAB Command Window. The code generator produces standard C and header files, and an HTML code generation report. The code generator places the files in a *build folder*, a subfolder named `throttlecntrl_grt_rtw` under your current working folder.

Review the Generated Code

1 Open Model Explorer, and in the **Model Hierarchy** pane, expand the node for the `throttlecntrl` model, and select the **Code for** node.

2 In the **Contents** pane, select **HTML Report**. Model Explorer displays the HTML code generation report for the throttle controller model.

3 In the HTML report, click the link for the generated C model file and review the generated code. Note the following:

- Identification, version, timestamp, and configuration comments.
- Links to help you navigate within and between files
- Data definitions

- Scheduler code
- Controller code
- Model initialization and termination functions
- Call interface for the GRT target — output, update, initialization, start, and terminate

4 Save and close `throttlecntrl`.

Consider examining the following files. In the HTML report **Contents** pane, click the links. Or, in your working folder, explore the generated code subfolder.

File	Description
`throttlecntrl.c`	C file that contains the scheduler, controller, initialization, and interface code
`throttlecntrl_data.c`	C file that assigns values to generated data structures
`throttlecntrl.h`	Header file that defines data structures
`throttlecntrl_private.h`	Header file that defines data used only by the generated code
`throttlecntrl_types.h`	Header file that defines the model data structure

For more information, see “Generated Source Files and File Dependencies”.

At this point you might also want to consider logging data to a MAT-file. For an example, see “Log Data for Analysis”.

Generate an Executable

1 If you have not already done so, set up your C compiler. In the MATLAB Command Window, enter `mex -setup` and specify a valid, installed compiler.

2 Open `throttlecntrl`.

3 In the Configuration Parameters dialog box, clear the **Code Generation > Generate code only** check box and click **Apply**.

4 Click **Build**. Watch the messages in the MATLAB Command Window. The code generator uses a template make file associated with your target selection to create an executable that you can run on your workstation, independent of external timing and events.

5 Check your working folder for the file`throttlecntrl.exe`.

6 Run the executable. In the Command Window, enter `!throttlecntrl`. The ! character passes the command that follows it to the operating system, which runs the standalone program.

The program produces one line of output in the Command Window:

```
** starting the model **
```

At this point you might also want to consider logging data to a MAT-file. For an example, see "Log Data for Analysis".

Key Points

- To generate code change the model configuration to specify a fixed-step solver and then select a system target format. Using the `grt.tlc` file requires a fixed-step solver. If the model contains continuous time blocks then a variable-step solver can be used with the `rsim.tlc` target.
- After debugging a model, consider configuring a model with parameter inlining enabled.
- Use the `getActiveConfigSet` function to save a model configuration for future use or to apply it to another model.
- Before generating code, consider checking a model with the Model Advisor.
- The code generator places generated files in a subfolder (*model*`_grt_rtw`) of your working folder.

Learn More

- “Code Generation”
- “Configuration Sets”
- “Verify Model Syntax”

Configure Data Interface

In this section...

About This Example

Learning Objectives

- Configure the data interface for code generated for a model.
- Control the name, data type, and data storage class of signals and parameters in generated code.

Prerequisites

- Understanding ways to represent and use data and signals in models.
- Familiarity with representing data constructs as data objects.
- Ability to read C code.

Required File

`rtwdemo_throttlecntrl_datainterface` model file

Declare Data

Most programming languages require that you *declare* data before using it. The declaration specifies the following information:

Data Attribute	Description
Scope	The region of the program that has access to the data
Duration	The period during which the data is resident in memory
Data type	The amount of memory allocated for the data
Initialization	An initial value, a pointer to memory, or NULL (if you do not provide an initial value, most compilers assign a zero value or a null pointer)

The following data types are supported for code generation.

Supported Data Types

Name	Description
`double`	Double-precision floating point
`single`	Single-precision floating point
`int8`	Signed 8-bit integer
`uint8`	Unsigned 8-bit integer
`int16`	Signed 16-bit integer
`uint16`	Unsigned 16-bit integer
`int32`	Signed 32-bit integer
`uint32`	Unsigned 32-bit integer
Fixed point data types	8-, 16-, 32-bit word lengths

A *storage class* is the scope and duration of a data item. For more information about storage classes, see

- "Tunable Parameter Storage Classes"

- "Signals Storage Classes"
- "State Storage Classes"

Use Data Objects

In Simulink models and Stateflow charts, the following methods are available for declaring data: *data objects* and *direct specification*. This example uses the data object method. Both methods allow full control over the data type and storage class. You can mix the two methods in a single model.

In the MATLAB and Simulink environment, you can use data objects in a variety of ways. This example focuses on the following types of data objects:

- Signal
- Parameter
- Bus

To configure the data interface for your model using the data object method, in the MATLAB base workspace, you define data objects and then associate them with your Simulink model or embedded Stateflow chart. When you build your model, the build process uses the associated base workspace data objects in the generated code.

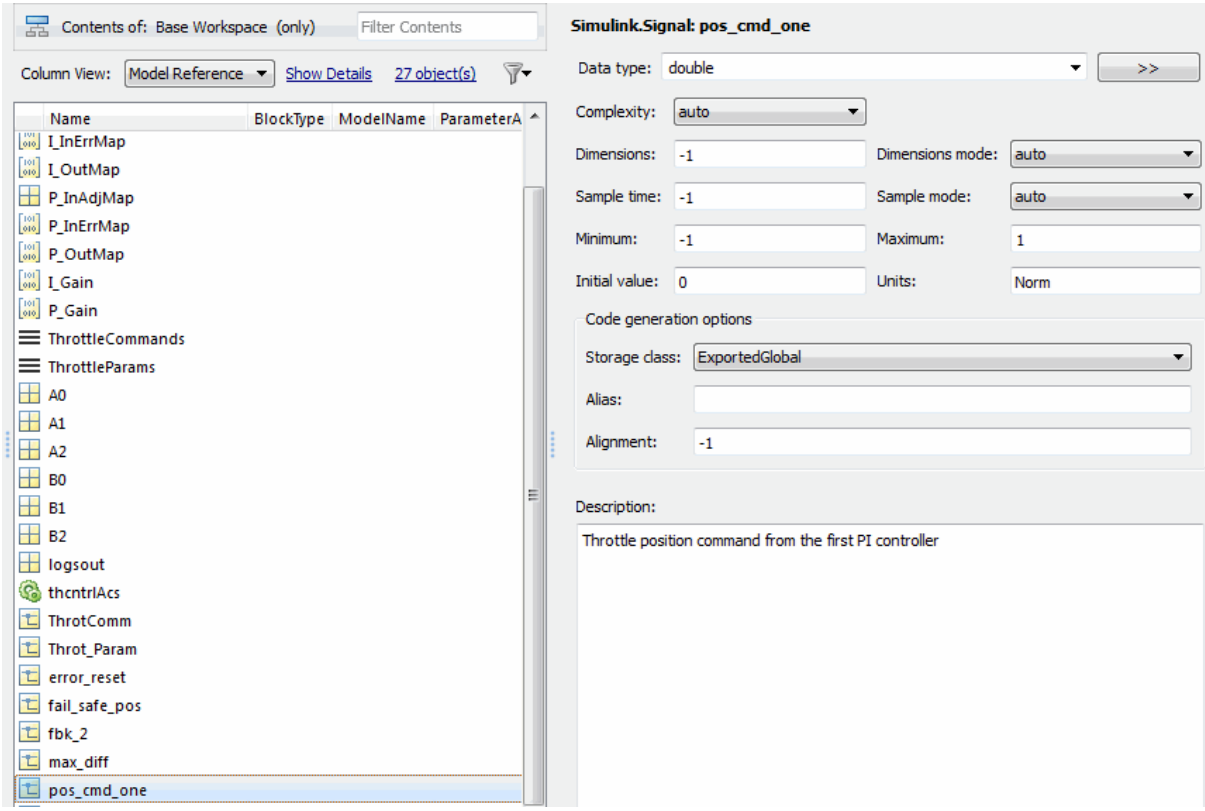
A data object has a mixture of *active* and *descriptive* fields. Active fields potentially affect simulation or code generation. Descriptive fields do not affect simulation or code generation. They are used with data dictionaries and model-checking tools.

- Active fields:
  - Data type
  - Storage class
  - Value (parameters)
  - Initial value (signals)
  - Alias (define a different name in the generated code)
  - Dimension (inherited for parameters)

  - Complexity (inherited for parameters)
- Descriptive fields:
  - Minimum
  - Maximum
  - Units
  - Description

You can create and inspect base workspace data objects by entering commands in the MATLAB Command Window or by using Model Explorer. Perform the following steps to explore base workspace signal data objects.

1. Open `rtwdemo_throttlecntrl_datainterface` and save a copy as `throttlecntrl_datainterface` in a writable location on your MATLAB path.
2. Open Model Explorer.
3. Select **Base Workspace**.
4. Select the `pos_cmd_one` signal object for viewing.



You can also view the definition of a signal object. In the MATLAB Command Window, enter `pos_cmd_one`:

```
pos_cmd_one =

Simulink.Signal handle
Package: Simulink

Properties:
       CoderInfo: [1x1 Simulink.SignalCoderInfo]
     Description: 'Throttle position command from the first PI controller'
        DataType: 'double'
             Min: -1
             Max: 1
```

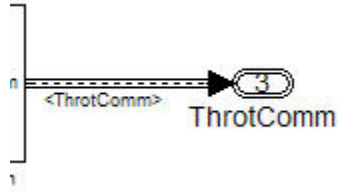
```
          DocUnits: 'Norm'
        Dimensions: -1
    DimensionsMode: 'auto'
        Complexity: 'auto'
        SampleTime: -1
      SamplingMode: 'auto'
      InitialValue: '0'

  Methods, Events, Superclasses
```

5 To view other signal objects, in Model Explorer, click the object name or in the MATLAB Command Window, enter the object name. The following table summarizes object characteristics for some of the data objects in this model.

Object Characteristics	`pos_cmd_one`	`pos_rqst`	`P_InErrMap`	`ThrotComm*`	`ThrottleCommands*`
Description	Top-level output	Top-level input	Calibration parameter	Top-level output structure	Bus definition
Data type	Double	Double	Auto	Auto	Structure
Storage class	Exported global	Imported extern pointer	Constant	Exported global	None

* `ThrottleCommands` defines a Bus object; `ThrotComm` is an instantiation of the bus. If the bus is a nonvirtual bus, the signal generates a structure in the C code.

You can use a bus definition (`ThrottleCommands`) to instantiate multiple instances of the structure. In a model diagram, a bus object appears as a wide line with central dashes, as shown below.



Add New Data Objects

You can create data objects for named signals, states, and parameters. To associate a data object with a construct, the construct must have a name.

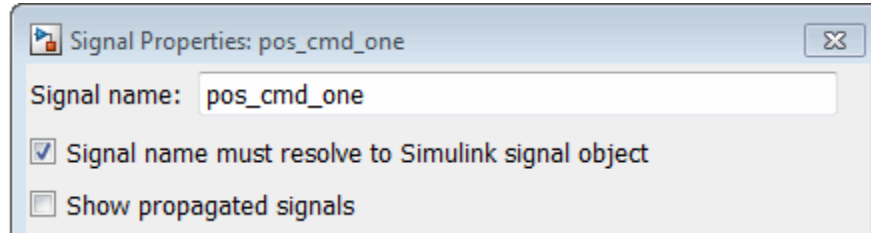
To find constructs for which you can create data objects, use the Data Object Wizard. This tool finds the constructs and then creates the objects for you. The model includes two signals that are not associated with data objects: `fbk_1` and `pos_cmd_two`.

To find the signals and create data objects for them:

1. In the model window, select **Code > Data Objects > Data Object Wizard**. The Data Object Wizard dialog box opens.
2. To find candidate constructs, click **Find**. Constructs `fbk_1` and `pos_cmd_two` appear in the dialog box.
3. To select both constructs, click **Check All**.
4. To apply the default Simulink package for the data objects, click **Apply Package**.
5. To create the data objects, click **Create**. Constructs `fbk_1` and `pos_cmd_two` are removed from the dialog box.
6. Close the Data Object Wizard.
7. In the **Contents** pane of the Model Explorer, find the newly created objects `fbk_1` and `pos_cmd_two`.

Enable Data Objects for Generated Code

1. In the Model Explorer **Model Hierarchy**, expand the `throttlecntrl_datainterface` model node.
2. Click the **Configuration (Active)** node. Make sure that you select, **Optimization > Signals and Parameters > Inline parameters**.
3. Enable a signal to appear in generated code.
   a. In the model window, right-click the `pos_cmd_one` signal line and select **Properties**. A Signal Properties dialog box opens.
   b. Make sure that you select the **Signal name must resolve to Simulink signal object** parameter.



4. Enable signal object resolution for all signals in the model. In the MATLAB Command Window, enter:

```
disableimplicitsignalresolution('throttlecntrl_datainterface')
```

5. Save and close `throttlecntrl_datainterface`.

Effects of Simulation on Data Typing

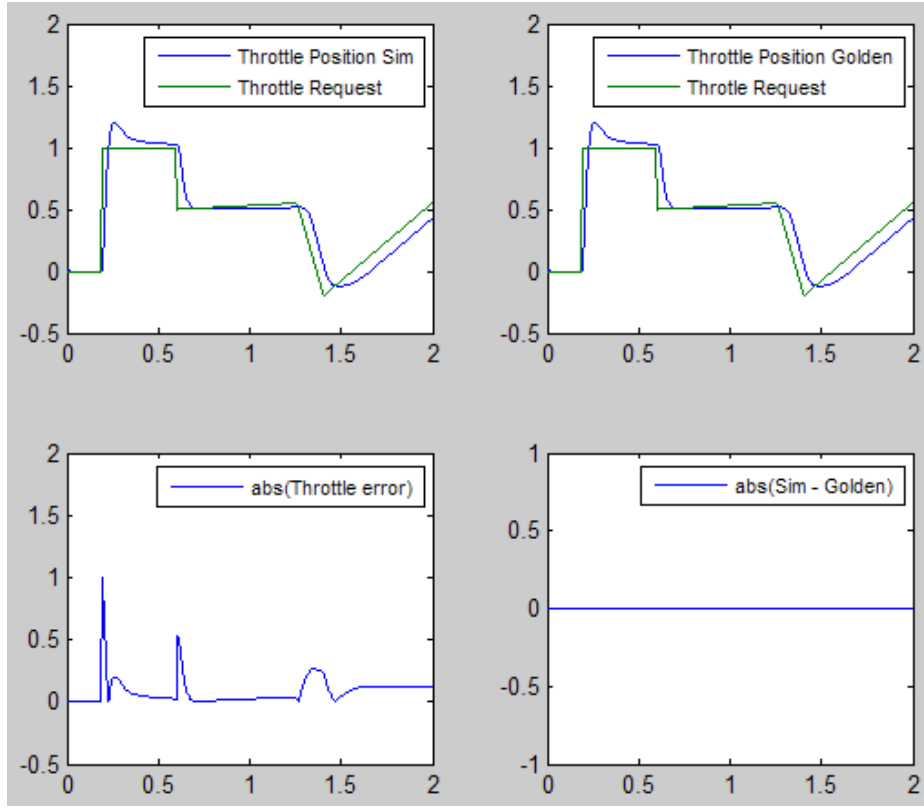
In the throttle controller model, all data types are set to `double`. Because Simulink software uses the `double` data type for simulation, do not expect changes in the model behavior when you run the generated code. You verify this by running the test harness.

Before you run your test harness, update it to include the `throttlecntrl_datainterface` model.

Note The following procedure requires a Stateflow license.

1. Open `throttlecntrl_datainterface`.
2. Open your copy of test harness, `throttlecntrl_testharness`.
3. Right-click the `Unit_Under_Test` Model block and select **Block Parameters (ModelReference)**.
4. Set **Model name** to `throttlecntrl_datainterface`. Click **OK**.
5. Update the test harness model diagram.
6. Simulate the test harness.

   The resulting plot shows that the difference between the golden and simulated versions of the model remains zero.



7 Save and close `throttlecntrl_testharness`.

Manage Data

Data objects exist in a separate file from the model in the base workspace. To save the data manually, in the MATLAB Command Window, enter `save`.

The separation of data from the model provides the following benefits:

- One model, multiple data sets:
  - Use of different parameter values to change the behavior of the control algorithm (for example, for reusable components with different calibration values)

  - Use of different data types to change targeted hardware (for example, for floating-point and fixed-point targets)
- Multiple models, one data set:
  - Sharing data between models in a system
  - Sharing data between projects (for example, transmission, engine, and wheel controllers might use the same CAN message data set)

Key Points

- You can declare data in Simulink models and Stateflow charts by using data objects or direct specification.
- From the Model Explorer or from the command line in the MATLAB Command Window, you manage (create, view, configure, and so on) base workspace data.
- The Data Object Wizard provides a quick way to create data objects for constructs such as signals, buses, and parameters.
- You must explicitly configure data objects to appear by name in generated code.
- Separation of data from model provides several benefits.

Learn More

- “Import Data”
- “Data Representation”
- “Custom Storage Classes”
- “Manage Placement of Data Definitions and Declarations”

Call External C Functions

In this section...

About This Example

Learning Objectives

- Evaluate a C function as part of a model simulation.
- Call an external C function from generated code.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Ability to set model configuration parameters.
- Ability to read C code.
- Set up a C compiler. In the MATLAB Command Window, enter `mex -setup` and specifying a valid, installed compiler.

Required Files

- `rtwdemo_throttlecntrl_extfunccall` model file
- `rtwdemo_ValidateLegacyCodeVrsSim` model file

- `/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.c`
- `/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.h`

Include External C Functions in a Model

Simulink models are one part of Model-Based Design. For many applications, a design also includes a set of preexisting C functions created, tested (verified), and validated outside of a MATLAB and Simulink environment. You can integrate these functions easily into a model and the generated code. External C code can be used in the generated code to access hardware devices and external data files during rapid simulation runs.

This example shows you how to create a custom block that calls an external C function. Once the block is part of the model, you can take advantage of the simulation environment to test the system further.

Create a Block That Calls a C Function

To specify a call to an external C function, use an S-Function block. You can automate the process of creating the S-Function block by using the Simulink Legacy Code Tool. Using this tool, you specify an interface for your external C function. The tool then uses that interface to automate creation of an S-Function block.

1. Make copies of the files `SimpleTable.c` and `SimpleTable.h`, located in *`matlabroot`*`/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files`. Put the copies in your working folder.

   Note *`matlabroot`* represents the name of your top-level MATLAB installation folder.

2. Create an S-Function block that calls the specified function at each time step during simulation:

   a. In the MATLAB Command Window, create a function interface definition structure:

   ```
   def=legacy_code('initialize')
   ```

The data structure `def` defines the function interface to the external C code.

```
def =

                  SFunctionName: ''
    InitializeConditionsFcnSpec: ''
                OutputFcnSpec: ''
                 StartFcnSpec: ''
              TerminateFcnSpec: ''
                   HeaderFiles: {}
                   SourceFiles: {}
                  HostLibFiles: {}
                TargetLibFiles: {}
                      IncPaths: {}
                      SrcPaths: {}
                      LibPaths: {}
                    SampleTime: 'inherited'
                       Options: [1x1 struct]
```

b Populate the function interface definition structure by entering the following commands:

```
def.OutputFcnSpec=['double y1 = SimpleTable(double u1,',...
   'double p1[], double p2[], int16 p3)'];
def.HeaderFiles = {'SimpleTable.h'};
def.SourceFiles = {'SimpleTable.c'};
def.SFunctionName = 'SimpTableWrap';
```

c Create the S-function:

```
legacy_code('sfcn_cmex_generate', def)
```

d Compile the S-function:

```
legacy_code('compile', def)
```

e Create the S-Function block:

```
legacy_code('slblock_generate', def)
```

A new model window opens that contains the `SimpTableWrap` block.

Tip Creating the S-Function block is a one-time task. Once the block exists, you can reuse it in multiple models.

3. Save the model to your working folder as: `s_func_simptablewrap`.

4. Create a Target Language Compiler (TLC) file for the S-Function block:

   ```
   legacy_code('sfcn_tlc_generate', def)
   ```

   The TLC file is the component of an S-function that specifies how the code generator produces the code for a block.

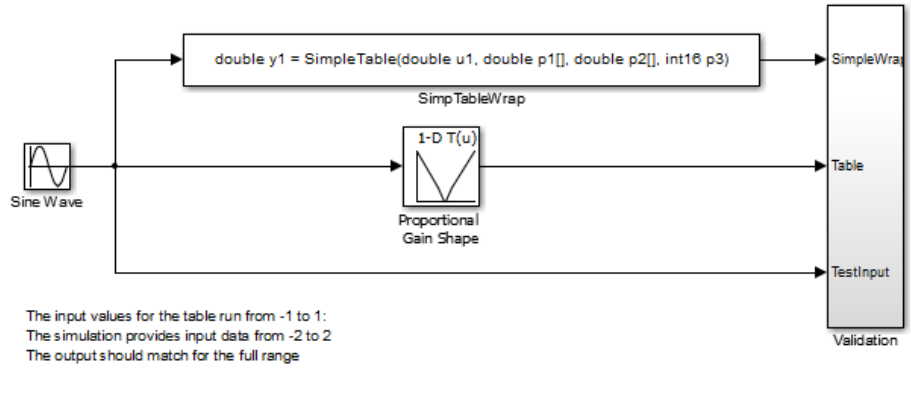
For more information on using the Legacy Code Tool, see:

- "Integrate C Functions Using Legacy Code Tool" in the Simulink documentation
- "Integrate External Code Using Legacy Code Tool"

Validate External Code in the Simulink Environment

When you integrate external C code with a Simulink model, before using the code, validate the functionality of the external C function code as a standalone component.

1. Open the model `rtwdemo_ValidateLegacyCodeVrsSim`. This model validates the S-function block that you just created.

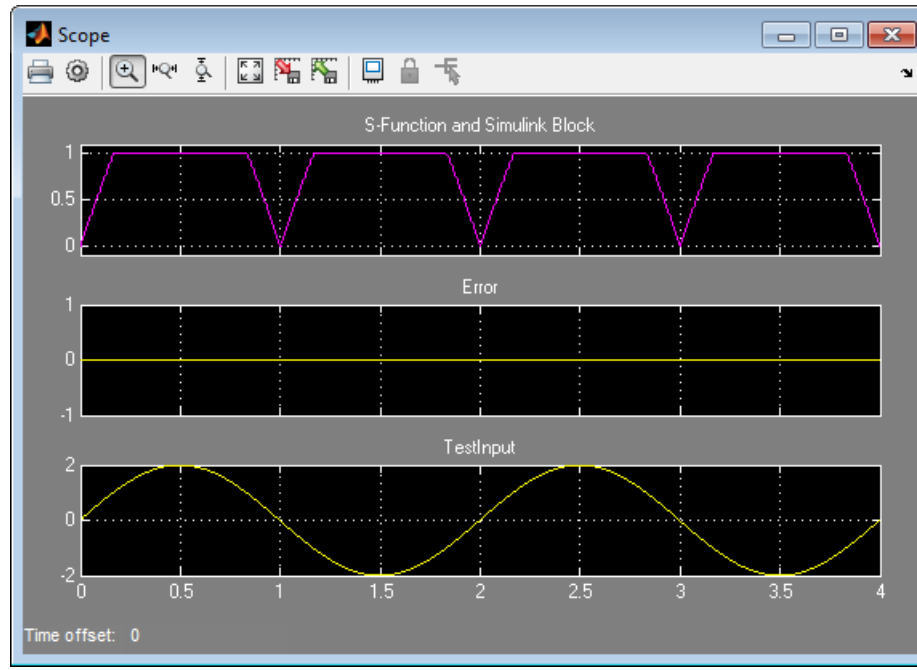


- The Sine Wave block produces output values from [-2 : 2].
- The input range of the lookup table is from [-1 : 1].
- The output from the lookup table is the absolute value of the input.
- The lookup table output clips the output at the input limits.

2 Simulate the model.

3 View the validation results by opening the `Validation` subsystem and, in that subsystem, clicking the Scope block.

The following figure shows the validation results. The external C code and the Simulink Lookup table block provide the same output values.



4 Close the validation model.

Validate C Code as Part of a Model

After you validate the functionality of the external C function code as a standalone component, validate the S-function in the model. Use the test harness model to complete the validation.

Note The following procedure requires a Stateflow license.

---

1 Open `rtwdemo_throttlecntrl_extfunccall` and save a copy as `throttlecntrl_extfunccall` in a writable folder on your MATLAB path.

2 Examine the `PI_ctrl_1` and `PI_ctrl_2` subsystems.

   a Lookup blocks have been replaced with the block you created using the Legacy Code Tool.

b Note the block parameter settings for `SimpTableWrap` and `SimpTableWrap1`.

c Close the Block Parameter dialog boxes and the PI subsystem windows.

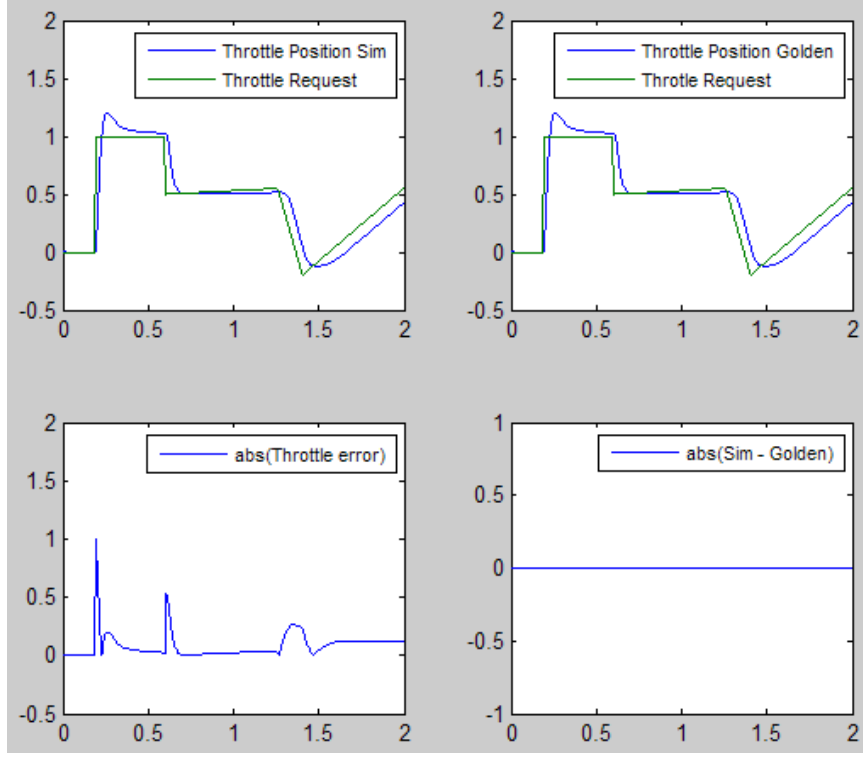
3 Open the test harness model, right-click the `Unit_Under_Test` Model block, and select **Block Parameters (ModelReference)**.

4 Set **Model name** to `throttlecntrl_extfunccall`. Click **OK**.

5 Update the test harness model diagram.

6 Simulate the test harness.

The simulation results match the expected golden values.



7 Save and close `throttlecntrl_extfunccall` and `throttlecntrl_testharness`.

Call a C Function from Generated Code

The code generator uses a TLC file to process the S-Function block. Calls to C code embedded in an S-Function block:

- Can use data objects.
- Are subject to *expression folding*, an operation that combines multiple computations into a single output calculation.

1 Open `throttlecntrl_extfunccall`.

2 Generate code for the model.

3 Examine the generated code in the file`throttlecntrl_extfunccall.c`.

4 Close `throttlecntrl_extfunccall` and `throttlecntrl_testharness`.

Key Points

- You can easily integrate external functions into a model and generated code by using the Legacy Code Tool.
- Validate the functionality of external C function code which you integrate into a model as a standalone component.
- After you validate the functionality of external C function code as a standalone component, validate the S-function in the model.

Learn More

- “Integrate C Functions Using Legacy Code Tool”
- “Insert S-Function Code”

Index

S

T

V