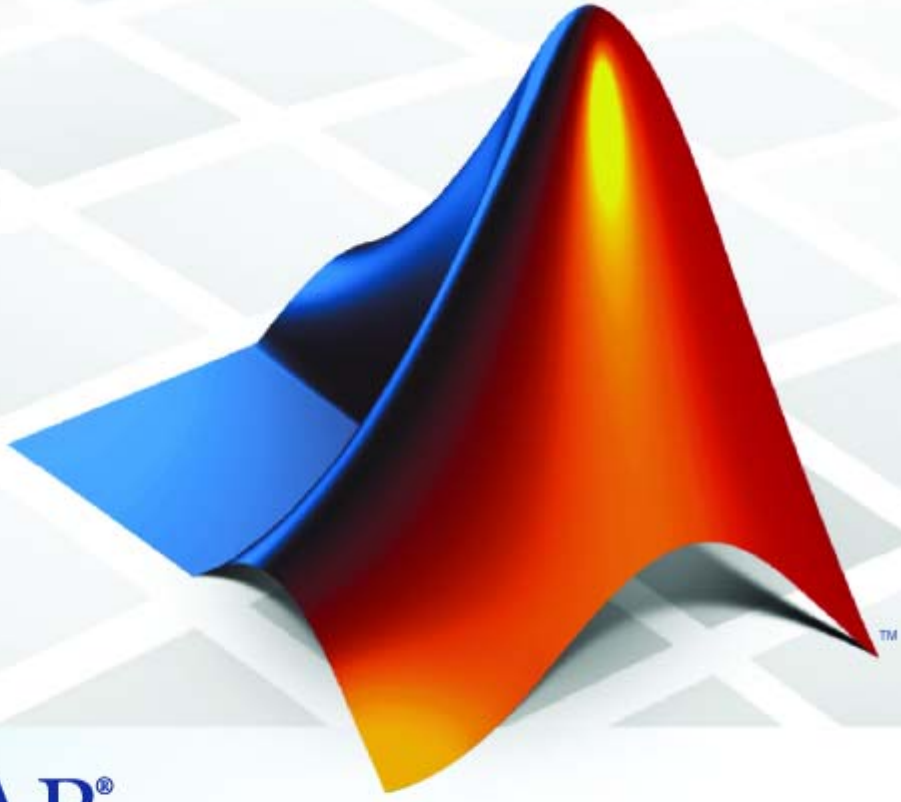
Real-Time Workshop® 7
Getting Started Guide



MATLAB®
& SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The 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.

Real-Time Workshop® 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

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

Revision History

July 2002	First printing	New for Version 5.0 (Release 13)
June 2004	Second printing	Revised for Version 6.0 (Release 14)
October 2004	Third printing	Revised for Version 6.1 (Release 14SP1)
March 2005	Online only	Revised for Version 6.2 (Release 14SP2)
September 2005	Fourth printing	Revised for Version 6.3 (Release 14SP3)
March 2006	Online Only	Revised for Version 6.4 (Release 2006a)
September 2006	Online Only	Revised for Version 6.5 (Release 2006b)
March 2007	Fifth printing	Revised for Version 6.6 (Release 2007a)
September 2007	Online Only	Revised for Version 7.0 (Release 2007b)
March 2008	Online Only	Revised for Version 7.1 (Release 2008a)
October 2008	Sixth printing	Revised for Version 7.2 (Release 2008b)
March 2009	Online Only	Revised for Version 7.3 (Release 2009a)

Contents

1 Getting Started with Real-Time Workshop Technology

2 How to Develop an Application Using Real-Time Workshop Software

3 Working with the Real-Time Workshop Software

Index

1

Getting Started with Real-Time Workshop Technology

What You Need to Know to Use Real-Time Workshop Technology

Before you use Real-Time Workshop® technology, you should be familiar with

- Using the Simulink® and Stateflow® software to create models or state machines as block diagrams, running such simulations in Simulink, and interpreting output in the MATLAB® workspace
- High-level programming language concepts applied to real-time systems

While you do not need to program in C or other programming languages to create, test, and deploy real-time systems using the Real-Time Workshop software, successful emulation and deployment of real-time systems requires familiarity with parameters and design constraints. The Real-Time Workshop documentation assumes you have a basic understanding of real-time system concepts, terminology, and environments.

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 Embedded MATLAB functions, see Technical Solution 1-6AWSQ9 on the MathWorks Web site.

What You Can Accomplish Using Real-Time Workshop Technology

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

You apply Real-Time Workshop technology explicitly with the Real-Time Workshop and Real-Time Workshop® Embedded Coder™ products. Using the Real-Time Workshop product, you can

- Generate source code and executables for discrete-time, continuous-time (fixed-step), and hybrid systems modeled in Simulink
- Use the generated code for real-time and non-real-time applications, including simulation acceleration, rapid prototyping, and hardware-in-the-loop (HIL) testing
- Tune and monitor the generated code by using Simulink blocks and built-in analysis capabilities, or run and interact with the code completely outside the MATLAB and Simulink environment
- Generate code for finite state machines modeled in Stateflow event-based modeling software, using the optional Stateflow® Coder™ product
- Produce source code for many Simulink products and blocksets provided by The MathWorks™ and third-party vendors.

The Real-Time Workshop Embedded Coder product *extends* the Real-Time Workshop product with features that are important for embedded software

development. Using the Real-Time Workshop Embedded Coder add-on product, you gain access to all aspects of Real-Time Workshop technology and can generate code that has the clarity and efficiency of professional handwritten code. For example, you can

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

The following table compares typical applications and key capabilities for these two code generation products.

Product	Typical Applications	Key Capabilities
Real-Time Workshop	Simulation acceleration Simulink model encryption Rapid prototyping HIL testing	Generate code for discrete-time, continuous-time (fixed-step), and hybrid systems modeled in Simulink Tune and monitor the execution of generated code by using Simulink blocks and built-in analysis capabilities or by running and interacting with the code outside the MATLAB and Simulink environment Generate code for finite state machines modeled in Stateflow event-based modeling software, using the optional Stateflow Coder product

Product	Typical Applications	Key Capabilities
		Generate code for many MathWorks™ and third-party Simulink products and blocksets Integrate existing applications, functions, and data
Real-Time Workshop Embedded Coder	All applications listed for the Real-Time Workshop product Embedded systems On-target rapid prototyping boards Microprocessors used in mass production	All capabilities listed for the Real-Time Workshop product Generate code that has the clarity and efficiency of professional handwritten code Customize the appearance and performance of the code for specific target environments Enable tracing, reporting, and testing options that facilitate code verification activities

How the Technology Can Fit Into Your Development Process

In this section...

Tools for Algorithm Development

You can use Real-Time Workshop technology to generate standalone C or C++ source code for algorithms that you develop the following ways:

- With MATLAB code, using the Embedded MATLAB language subset
- As Simulink models
- With MATLAB code that you incorporate into Simulink models

The Embedded MATLAB language subset supports MATLAB operators and functions for floating-point and fixed-point math. 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. Real-Time Workshop technology generates code for most Simulink blocks and many MathWorks products.

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 Embedded MATLAB functions, see Technical Solution 1-6AWSQ9 on the MathWorks Web site.

The following table lists products that the Real-Time Workshop and Real-Time Workshop Embedded Coder software support.

Products Supported by Real-Time Workshop and Real-Time Workshop Embedded Coder	Notes
Aerospace Blockset™	—
Communications Blockset™	—
Control System Toolbox™	—
Embedded IDE Link™ CC	—
Embedded IDE Link MU	—
Embedded IDE Link TS	—
Embedded IDE Link VS	—
Fuzzy Logic Toolbox™	—
Gauges Blockset™	—
MATLAB	Details: Supports Embedded MATLAB
Model-Based Calibration Toolbox™	—
Model Predictive Control Toolbox™	—
PolySpace® Model Link™ SL	Not supported by Real-Time Workshop
Real-Time Windows Target™	—
Signal Processing Blockset™	Details: "Simulink Block Data Type Support for Signal Processing Blockset" Table (enter the MATLAB `showsignalblockdatatypetable` command)
SimDriveline™	—
SimElectronics™	—
SimHydraulics®	—
SimMechanics™	—
SimPowerSystems™	Not supported by Real-Time Workshop Embedded Coder
Simscape™	—

Products Supported by Real-Time Workshop and Real-Time Workshop Embedded Coder	Notes
Simulink	Details: "Simulink Block Support" Table in the Real-Time Workshop documentation
Simulink Fixed Point	—
Simulink® 3D Animation™	—
Simulink® Design Optimization™	—
Simulink® Report Generator™	—
Simulink® Verification and Validation™	—
Stateflow and Stateflow Coder	—
System Identification Toolbox™	Exceptions: • Nonlinear IDNLGREY Model, IDDATA Source, IDDATA Sink, and estimator blocks • Nonlinear ARX models that contain custom regressors • `neuralnet` nonlinearities • `customnet` nonlinearities
Target Support Package™ FM5	—
Target Support Package IC1	—
Target Support Package TC2	—
Target Support Package TC6	—
Vehicle Network Toolbox™	—
Video and Image Processing Blockset™	—
xPC Target™	—
xPC Target Embedded Option™	—

Use of both Embedded MATLAB code and Simulink models is typical for Model-Based Design projects where you start developing an algorithm through research and development or advanced production, using MATLAB,

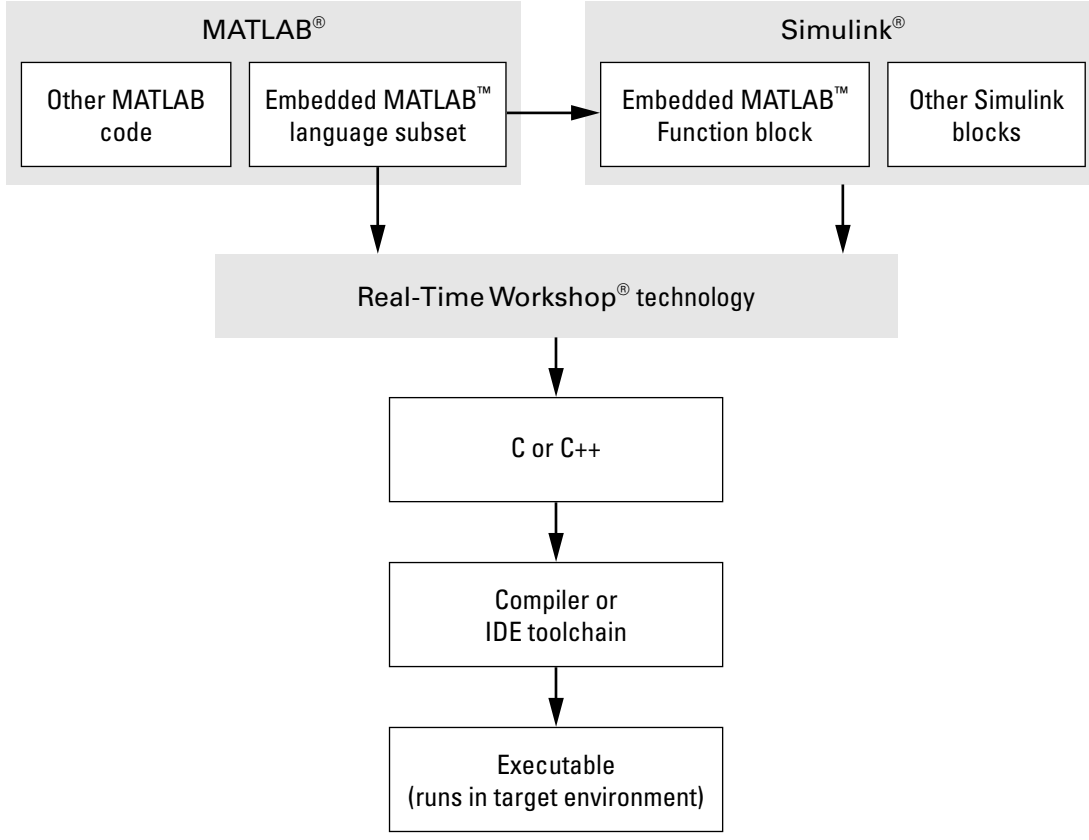
and then use Simulink for system deployment and verification. Benefits of this approach include:

- Richer system simulation environment
- Ability to verify the Embedded MATLAB code
- Real-Time Workshop and Real-Time Workshop Embedded Coder C/C++ code generation for the model and embedded M-code

The following table summarizes how to generate C or C++ code for each of the three approaches and identifies where you can find more information.

If you develop algorithms using...	You generate code by...	For more information, see...
Embedded MATLAB language subset	Entering the Real-Time Workshop function `emlc` in the MATLAB Command Window.	"Working with the Embedded MATLAB Subset" "Working with Embedded MATLAB Coder"
Simulink	Configuring and initiating code generation for your model or subsystem with the Simulink Configuration Parameters dialog.	"Workflow for Developing Applications Using Real-Time Workshop Software" on page 2-2 in Getting Started with Real-Time Workshop
Embedded MATLAB language subset and Simulink	Including Embedded MATLAB code in Simulink models or subsystems by using the Embedded MATLAB Function block. To use this block, you can do one of the following: • Copy your M-code into the block. • Call your M-code from the block by referencing the appropriate M-files on the MATLAB path.	"Working with the Embedded MATLAB Subset" in the Embedded MATLAB documentation

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



Target Environments

In addition to generating source code for a model or subsystem, Real-Time Workshop technology generates make or project files you need to build an executable for a specific target environment. The generated make or project files are optional. That is, 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 code generated with Real-Time Workshop technology 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.

Real-Time Workshop technology 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.

Before you select a system target file, you need to identify the target environment on which you expect to execute your generated code. The three most common target environments include:

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, they might suspend code execution to run an operating system service and then, after providing the service, continue code execution. Thus, the executable for your generated code might run faster or slower than the sample rates you specified in your model.

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.

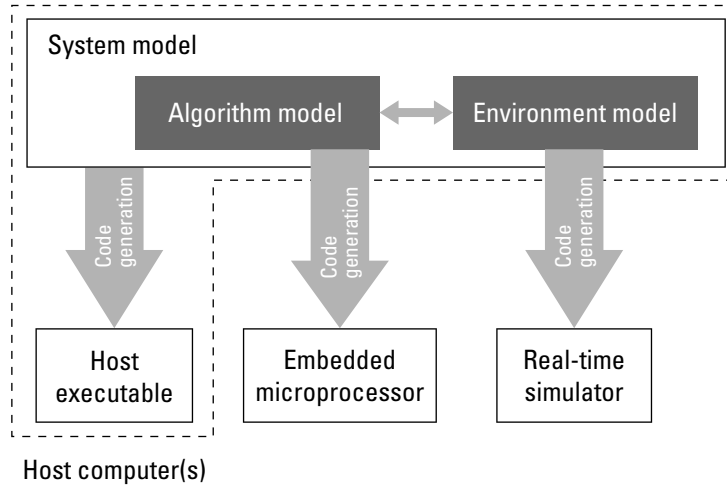
Target Environment	Description
Real-time simulator	A different computer than the host computer. A real-time simulator can be a PC or UNIX environment that uses a real-time operating system (RTOS), such as: • xPC Target system • A real-time Linux system • A Versa Module Eurocard (VME) chassis with PowerPC® processors running a commercial RTOS, such as VxWorks® from Wind River® Systems The generated code runs in real time and behaves deterministically. Although, 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 standalone as part of an electronics-based product. Embedded microprocessors range in price and performance, from high-end digital signal processors (DSPs) used to process communication signals to inexpensive 8-bit fixed-point microcontrollers used in mass production (for example, electronic parts produced in the millions of units). Embedded microprocessors can: • Use a full-featured RTOS • Be driven by basic interrupts • Use rate monotonic scheduling provided with Real-Time Workshop technology

A target environment can:

- Have single- or multiple-core CPUs
- Be standalone 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 component correctness.

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



Applications

The following table lists several ways you can apply Real-Time Workshop 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 environment. Accelerated simulations are especially useful when run time is long compared to the time associated with compilation and checking whether the target is up to date.
Rapid simulation	You execute code generated for a model in non-real time on the host computer, but outside the context of the MATLAB and Simulink environment.
System simulation	You integrate components into a larger system. You provide generated source code and related dependencies for building in another environment or a host-based shared library to which other code can dynamically link.
Model encryption	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 also crucial for validating whether a component can adequately 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 to encrypt components for intellectual property protection.

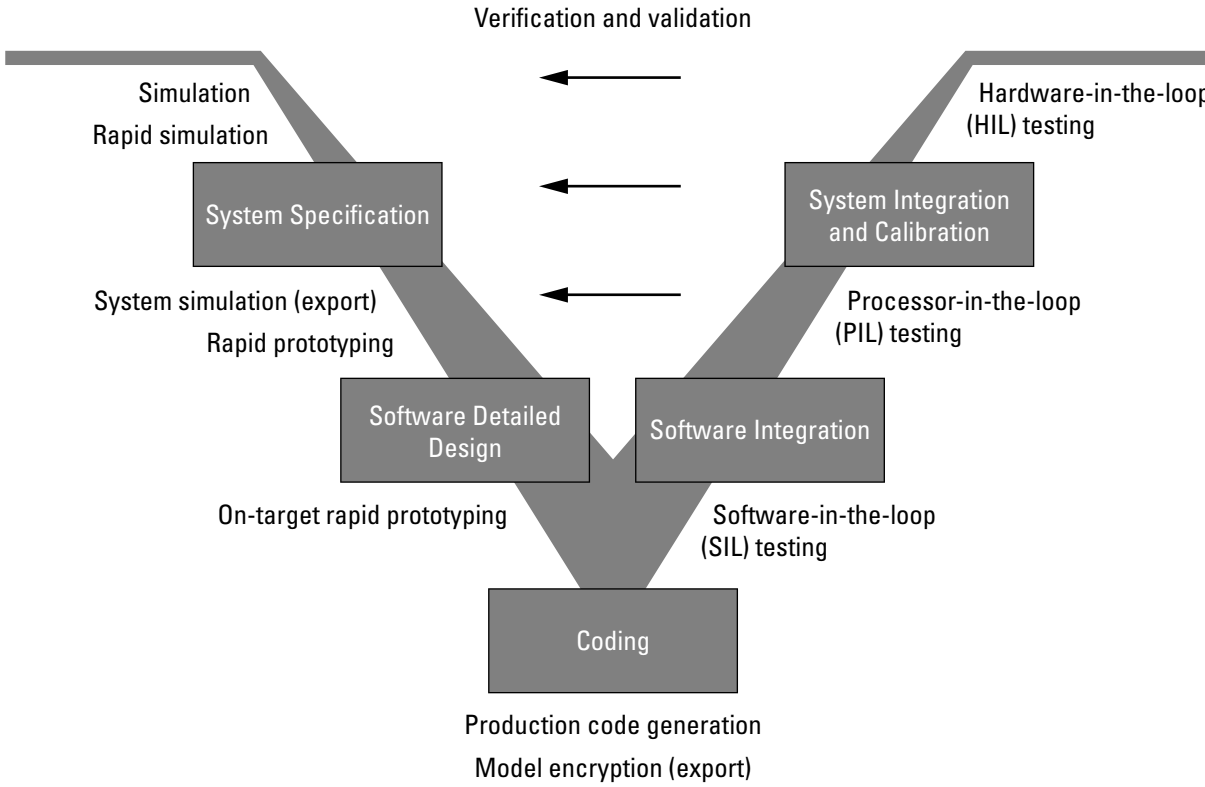
Application	Description
On-target rapid prototyping	You generate code for a detailed design that you can run in real time on an embedded microprocessor while tuning parameters and monitoring real-time data. This design step allows you to assess, interact with, and optimize code, using embedded compilers and hardware.
Embedded Microprocessor	
Production code generation	From a model, you generate code that is optimized for speed, memory usage, simplicity, and if necessary, compliance with industry standards and guidelines.
Software-in-the-loop (SIL) testing	You execute generated code with your plant model within Simulink to verify successful 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 use actual target word sizes and just test production code behavior.
Processor-in-the-loop (PIL) testing	You test an object code component with a plant or environment model in an open- or closed-loop simulation to verify successful 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.

How You Can Apply the Technology to the 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 focuses on the verification and validation of steps cited on the left side, including software integration and system integration.



Depending on your application and role in the process, you might focus on one or more of the steps called out in the V or repeat steps at several stages of the V. Real-Time Workshop technology and related products provide tooling you can apply at each step.

The following sections compare

- Types of simulation and prototyping
- Types of in-the-loop testing for verification and validation

For details on applications of Real-Time Workshop technology for steps identified in the figure, see the following topics in the Real-Time Workshop documentation:

- "Documenting and Validating Requirements"
- "Developing a Model Design Specification"
- "Developing a Detailed Software Design"
- "Developing the Application Code"
- "Integrating Software"
- "Integrating and Calibrating System Components"

Types of Simulation and Prototyping

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 non-real 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 environment	PC or nontarget hardware	Embedded computing unit (ECU) or near-production hardware

	Host-Based Simulation	Standalone Rapid Simulations	Rapid Prototyping	On-Target Rapid Prototyping
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 and parameters while experimenting Can accelerate Simulink simulations with Accelerated and Rapid Accelerated modes	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, without rebuilding the model Can be connected to Simulink to monitor signals and tune parameters	Might require custom real-time simulators and hardware Might be done with inexpensive off-the-shelf PC hardware and I/O cards	Might use existing hardware, thus less expensive and more convenient

Types of In-the-Loop Testing for Verification and Validation

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 since 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
Ease of use and cost	Desktop convenience Executes just in Simulink No cost for hardware	Executes on desk or test bench Uses hardware — process board and cables	Desktop convenience Executes just on host computer with Simulink and integrated development environment (IDE) No cost for hardware	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

2

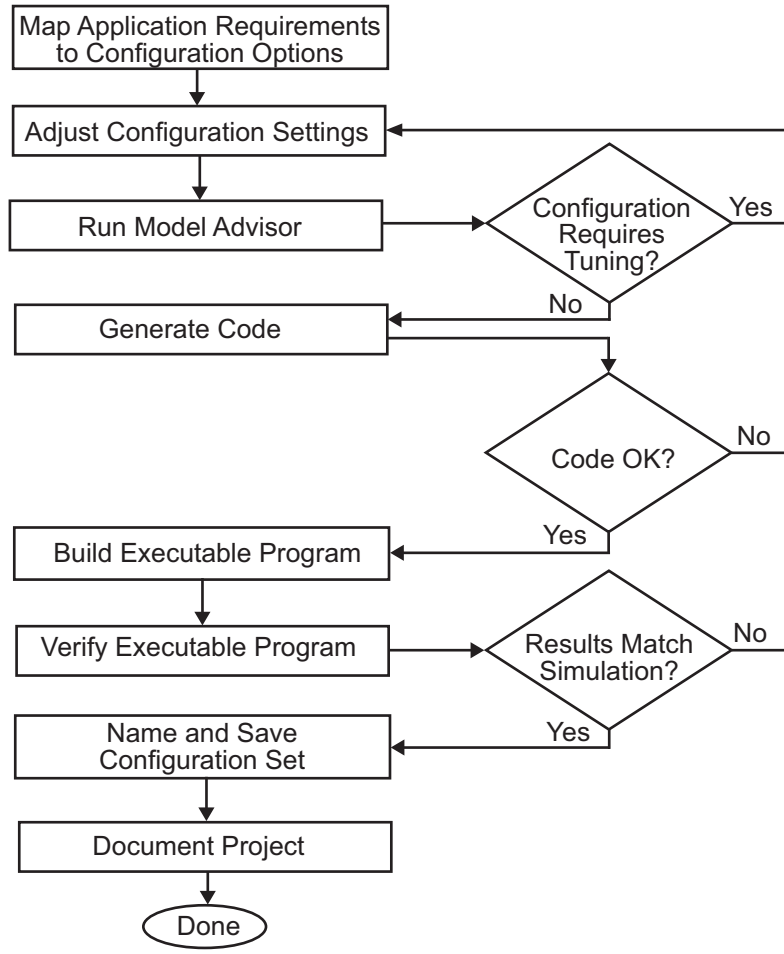
How to Develop an Application Using Real-Time Workshop Software

Workflow for Developing Applications Using Real-Time Workshop Software

The typical workflow for applying the Real-Time Workshop software to the application development process involves the following steps:

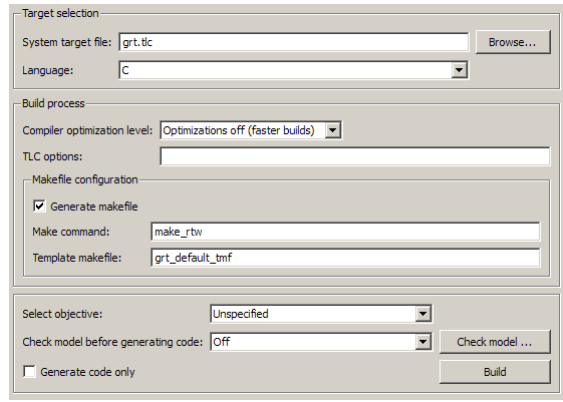
1. Map your application requirements to available configuration options.
2. Adjust configuration options as necessary.
3. Run the Model Advisor tool.
4. If necessary, tune configuration options based on the Model Advisor report.
5. Generate code for your model.
6. Repeat steps 2 to 5, if necessary.
7. 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 ones with the model.
10. Use Simulink Report Generator to automatically document the project.

The following figure shows these steps in a flow diagram. Sections following the figure discuss the steps in more detail.


Map Application Requirements to Configuration Options
Adjust Configuration Settings
Run Model Advisor
Configuration Requires Tuning?
Yes
No
Generate Code
Code OK?
No
Yes
Build Executable Program
Verify Executable Program
Results Match Simulation?
No
Yes
Name and Save Configuration Set
Document Project
Done

Mapping Application Requirements to Configuration Options

The first step in applying the Real-Time Workshop 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 screen display shows the **Real-Time Workshop** 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 Real-Time Workshop 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. However, you should 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 some other criterion?
- What is the second highest priority?

- Can the priority at the start of the project differ from the priority required for the end? What tradeoffs can be made?

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 "Configuring Code Generation Objectives" in the Real-Time Workshop *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.

Note

- Review the "Recommended Settings Summary" to see the settings the Code Generation Advisor recommends.
- 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 appropriate configuration parameter settings, adjust the settings accordingly. Using the Default column in "Mapping Application Requirements to the Solver Pane", identify the configuration parameters you need to modify. Then, open the Configuration Parameters dialog box or Model Explorer and make the necessary adjustments.

Tutorials in Chapter 3, "Working with the Real-Time Workshop Software" guide you through exercises that modify configuration parameter settings. For details on the configuration parameters that pertain to code generation, see "Code Generation and the Build Process" in the Real-Time Workshop documentation.

Note In addition to using the Configuration Parameters dialog box, you can use `get_param` and `set_param` to individually access most configuration parameters both interactively and in scripts. The configuration parameters you can get and set are listed in the "Parameter Reference" in the Real-Time Workshop documentation.

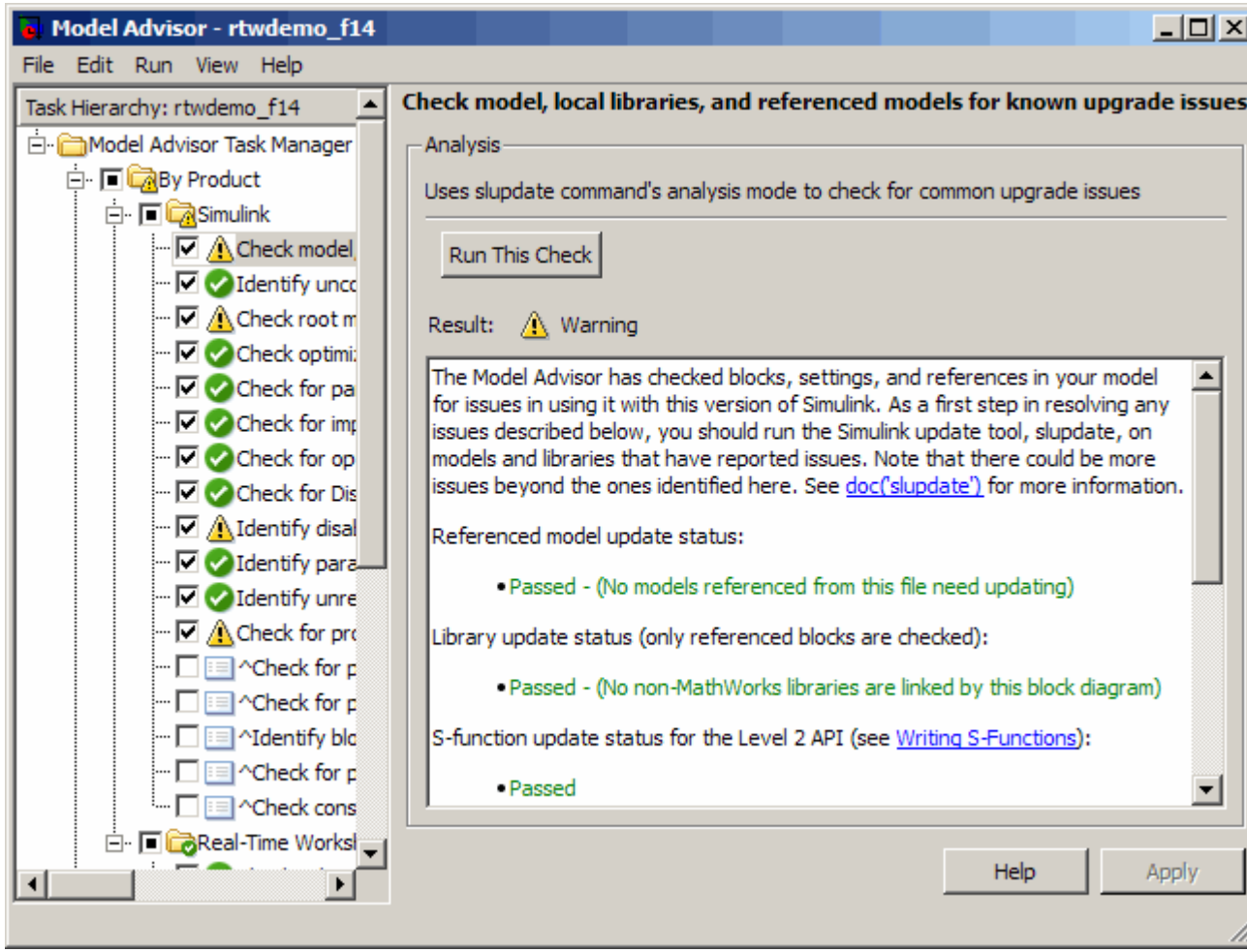
Running the Model Advisor

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

One way of starting the Model Advisor is to select **Tools > Model Advisor** in your model window. A new window appears listing specific diagnostics you can selectively enable or disable. Some examples of the diagnostics follow:

- Identify blocks that generate expensive saturation and rounding code
- Check optimization settings
- Identify questionable software environment specifications

Although you can use the Model Advisor to improve model simulation, it 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 "Getting Advice About Optimizing Models for Code Generation" in the Real-Time Workshop documentation.

Generating Code

After fine-tuning your model and its parameter settings, you are ready to generate code. Typically, the first time through the process of applying The Real-Time Workshop software for an application, you want to generate code without going on to compile and link it into an executable program. Some reasons for doing this include the following:

- You want to inspect the generated code. Is the Real-Time Workshop code generator creating what you expect?
- You need to integrate custom handwritten code.
- You want to experiment with configuration option settings.

You specify code generation only by selecting the **Generate code only** check box available on the **Real-Time Workshop** pane of the Configuration Parameters dialog box (thus changing the label of the **Build** button to **Generate code**). The Real-Time Workshop code generator responds by analyzing the block diagram that represents your model, generating C code, and placing the resulting files in a build directory within your current working directory.

After generating the code, inspect it. Is it what you expected? If not, determine what model and configuration changes you need to make, rerun the Model Advisor, and regenerate the code. When you are satisfied with the generated code, build an executable program image, as explained in "Building an Executable Program" on page 2-10.

For details on the **Generate code only** option, see "Generating Code Only" in the Real-Time Workshop documentation.

Building an Executable Program

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

One way of initiating a build is to click the **Build** button. The Real-Time Workshop code generator

1. Compiles the model — The Real-Time Workshop software analyzes your block diagram (and any 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 directory within your working directory.

   When you click **Generate code**, as explained in "Generating Code" on page 2-9, processing stops here.

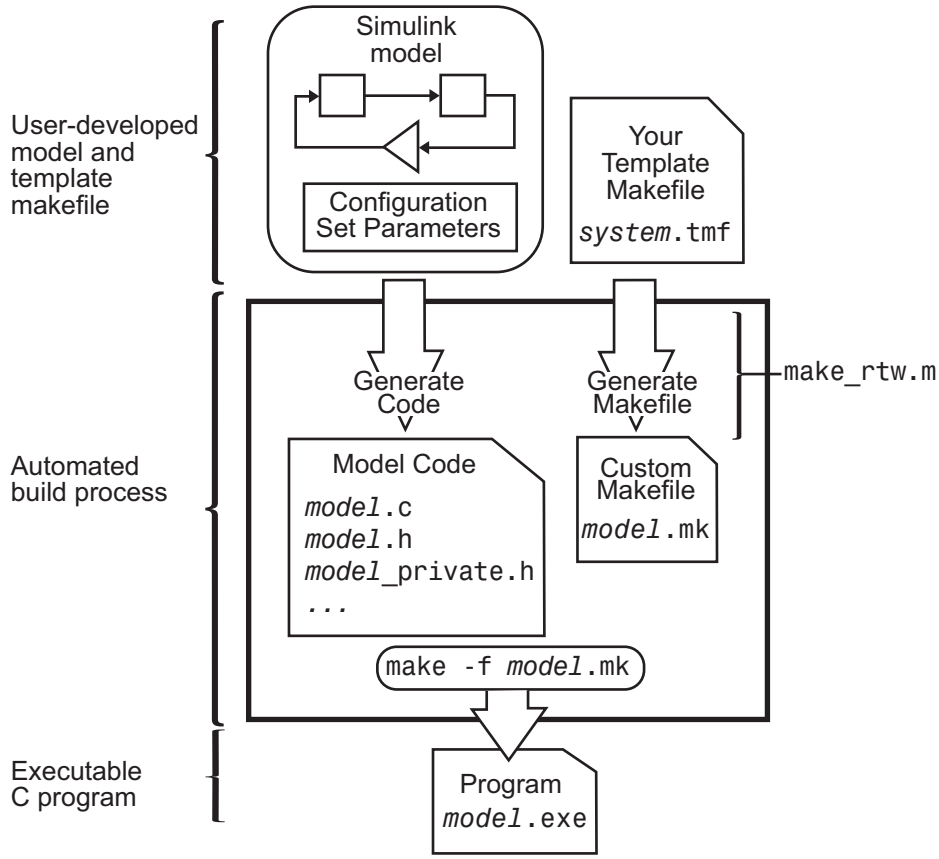
3. Generates a customized makefile — The Real-Time Workshop software constructs a makefile from the appropriate target makefile template and writes it in the build directory.

4. Generates an executable program — Instructing 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 directory.

   If you select **Create code generation report** on the **Real-Time Workshop > Report>** pane, a navigable summary of source files is produced when the model is built. The report files occupy directory `html` in the build directory. The report contents vary depending on the target, but all reports feature links to generated source files.

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 Real-Time Workshop software executes.



The M-file specified by the **Make command** field in the **Build process** section of the **Real-Time Workshop** pane of the Configuration Parameters dialog box controls an internal portion of the Real-Time Workshop build process. By default, the name of the M-file command is `make_rtw`; the Real-Time Workshop build process invokes this M-file for most targets. Any options specified in this field are passed into the makefile-based build process. In some cases, targets customize the `make_rtw` command. However, the arguments used by the function must be preserved.

Although the command may work for a stand-alone model, use of the `make_rtw` command at the command line can be error prone. For example, if you have multiple models open, you need to check whether

- The current subsystem, found by entering `gcs` in the MATLAB command window, contains the model you want to build.
- The **Make command** specified in the Configuration Parameters dialog box 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's simulation. You can do this by

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

Does the output match? Are you able to explain any differences? Do you need to eliminate any differences? At this point, it might be necessary to revisit and possibly fine-tune your block and configuration parameter settings.

For an example, see "Code Verification" on page 3-27.

Naming and Saving the Configuration Set

When you close a model, you should save it to preserve your configuration settings (unless you regard your recent changes as 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 it an identifying name. You can do this easily in Model Explorer.

To name and save a configuration,

1. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
2. In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.
3. Click the `Configuration (active)` node under the model name.

   The Configuration Parameters dialog box appears in the right pane.
4. In the **Configuration Parameters** pane, type a name you want to give the current configuration in the **Name** field.
5. Click **Apply**. The name of the active configuration in the **Model Hierarchy** pane changes to the name 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. To do this, copy the active configuration to a new one, or add a new one, then modify and name the new configuration, as follows:

1. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
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, select **Configuration Set** from the **Add** menu, or click the yellow gear icon on the toolbar:

   A new configuration set named `Configuration` appears in the **Model Hierarchy** pane.

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

   A new configuration set with a numeral (e.g., `1`) appended to its name appears lower in the **Model Hierarchy** pane.

5. If desired, rename the new configuration by right-clicking it, selecting **Properties**, and typing the new name in the **Name** field on the Configuration Parameters dialog box that appears. Then click the **Apply** button.

6. Make the new configuration the active one by right-clicking it in the **Model Hierarchy** pane and selecting **Activate** from the context menu.

   The content of the **Is Active** field in the right pane 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

One way of documenting a Real-Time Workshop code generation project is to use the Simulink Report Generator software. You can generate a comprehensive Rich Text Format (RTF), Extensible Markup Language (XML), or Hypertext Markup Language (HTML) report that includes the following information:

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

To get started with generating a code generation report, see the demo `rtwdemo_codegenrpt` and tutorial "Documenting a Code Generation Project" on page 3-73. For details on using the Report Generator, see the *Simulink Report Generator User's Guide*.

3

Working with the Real-Time Workshop Software

This chapter provides hands-on tutorials that help you get started generating code with the Real-Time Workshop software, as quickly as possible. It includes the following topics:

To get the maximum benefit from this book, The MathWorks recommends that you study and work all the tutorials, in the order presented.

These tutorials assume basic familiarity with the MATLAB and Simulink products. You should also read Chapter 2, "How to Develop an Application Using Real-Time Workshop Software", before proceeding.

The procedures for building, running, and testing your programs are almost identical in UNIX and PC environments. The discussion notes differences where applicable.

Make sure that a MATLAB compatible C compiler is installed on your system before proceeding with these tutorials. For details on supported compiler versions, see

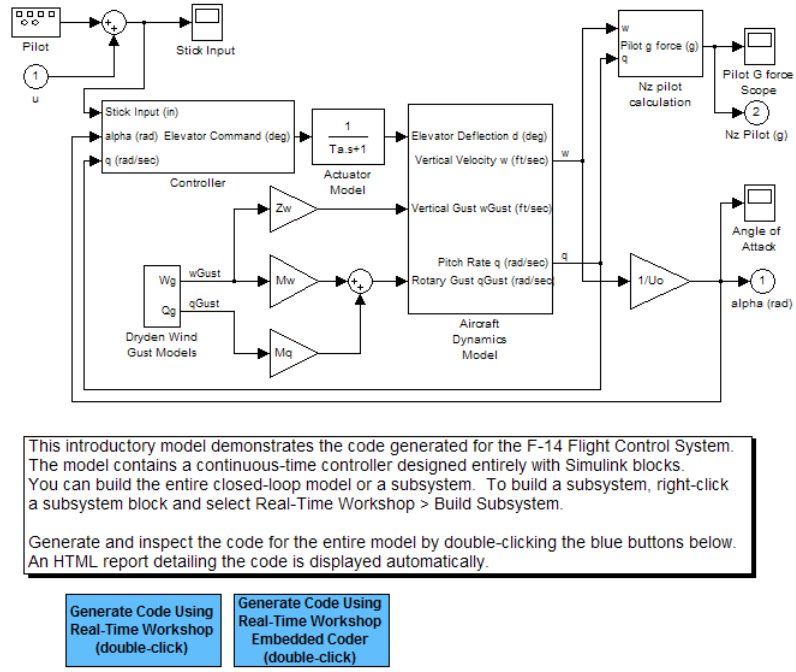
```
http://www.mathworks.com/support/compilers/current_release
```

Demonstration Model: rtwdemo_f14

The first three tutorials use a demonstration Simulink model, `rtwdemo_f14.mdl`, from the directory:

```
matlabroot/toolbox/rtw/rtwdemos/
```

By default, this directory is on your MATLAB path; *`matlabroot`* is the location of MATLAB on your system. The `rtwdemo_f14` model represents a simplified flight controller for the longitudinal motion of a Grumman Aerospace F-14 aircraft. The figure below shows the top level of this model.



The model simulates the pilot's stick input with a square wave having a frequency of 0.5 radians per second and an amplitude of ± 1. The system outputs are the aircraft angle of attack and the G forces experienced by the pilot. The input and output signals are visually monitored by Scope blocks.

Building a Generic Real-Time Program

In this section...

Tutorial Overview

This tutorial walks through the process of generating C code and building an executable program from the demonstration model. The resulting stand-alone program runs on your workstation, independent of external timing and events.

Working and Build Directories

It is convenient to work with a local copy of the `rtwdemo_f14` model, stored in its own directory, `f14example`. Set up your working directory as follows:

1. In the MATLAB Current Directory browser, navigate to a directory where you have write access.
2. Create the working directory from the MATLAB command line by typing:

   ```
   mkdir f14example
   ```

3. Make `f14example` your working directory:

   ```
   cd f14example
   ```

4. Open the `rtwdemo_f14` model:

   ```
   rtwdemo_f14
   ```

   The model appears in the Simulink window.

5 In the model window, choose **File > Save As**. Navigate to your working directory, `f14example`. Save a copy of the `rtwdemo_f14` model as `f14rtw.mdl`.

During code generation, the Real-Time Workshop software creates a *build directory* within your working directory. The build directory name is *`model_target`*`_rtw`, derived from the name of the source model and the chosen target. The build directory stores generated source code and other files created during the build process. You examine the build directory contents at the end of this tutorial.

Note When a model contains Model blocks (which enable one Simulink model to include others), special *project directories* are created in your working directory to organize code for referenced models. Project directories exist alongside of Real-Time Workshop build directories, and are always named `slprj`. "Generating Code for a Referenced Model" on page 3-61 describes navigating project directory structures in **Model Explorer**.

Setting Program Parameters

To generate code correctly from the `f14rtw` model, you must change some of the simulation parameters. In particular, note that generic real-time (GRT) and most other targets require that the model specify a fixed-step solver.

Note The Real-Time Workshop software can generate code for models using variable-step solvers for rapid simulation (`rsim`) and S-function targets only.

To set parameters, use the **Model Explorer** as follows:

1 Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.

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

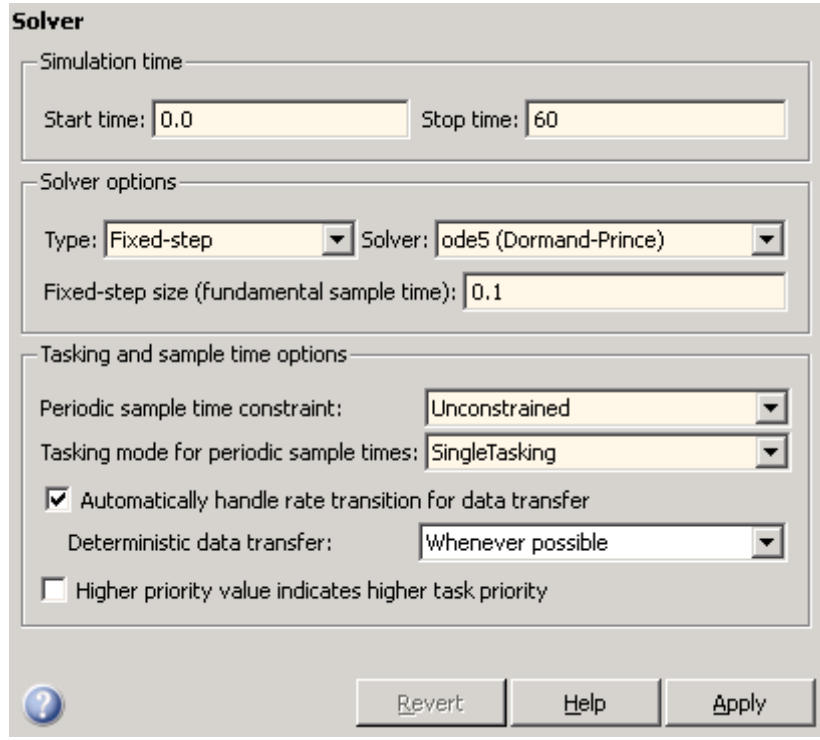
3 Click **Configuration (Active)** in the left pane.

4 Click **Solver** in the center pane. The **Solver** pane appears at the right.

5 Enter the following parameter values on the **Solver** pane (some may already be set):

- **Start time**: `0.0`
- **Stop time**: `60`
- **Type**: `Fixed-step`
- **Solver**: `ode5 (Dormand-Prince)`
- **Fixed step size (fundamental sample time)**: `0.1`
- **Tasking mode for periodic sample times**: `SingleTasking`

The **Solver** pane with the modified parameter settings is shown below. Note the tan background color of the controls you just changed. The color also appears on fields that were set automatically by your choices in other fields. Use this visual feedback to verify that what you set is what you intended. When you apply your changes, the background color reverts to white.



6 Click **Apply** to register your changes.

7 Save the model. Simulation parameters persist with the model, for use in future sessions.

Selecting the Target Configuration

Note Some of the steps in this section do not require you to make changes. They are included to help you familiarize yourself with the Real-Time Workshop user interface. As you step through the dialog boxes, place the mouse pointer on any item of interest to see a tooltip describing its function.

To specify the desired target configuration, you choose a system target file, a template makefile, and a `make` command.

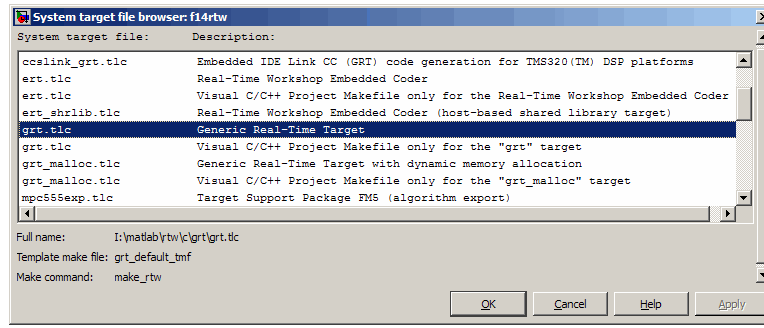
In these tutorials (and in most applications), you do not need to specify these parameters individually. Here, you use the ready-to-run generic real-time target (GRT) configuration. The GRT target is designed to build a stand-alone executable program that runs on your workstation.

To select the GRT target via the **Model Explorer**:

1. With the `f14rtw` model open, open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
2. In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.
3. Click **Configuration (Active)** in the left pane.
4. Click **Real-Time Workshop** in the center pane. The **Real-Time Workshop** pane appears at the right. This pane has several tabs.
5. Click the **General** tab to activate the pane that controls target selection.

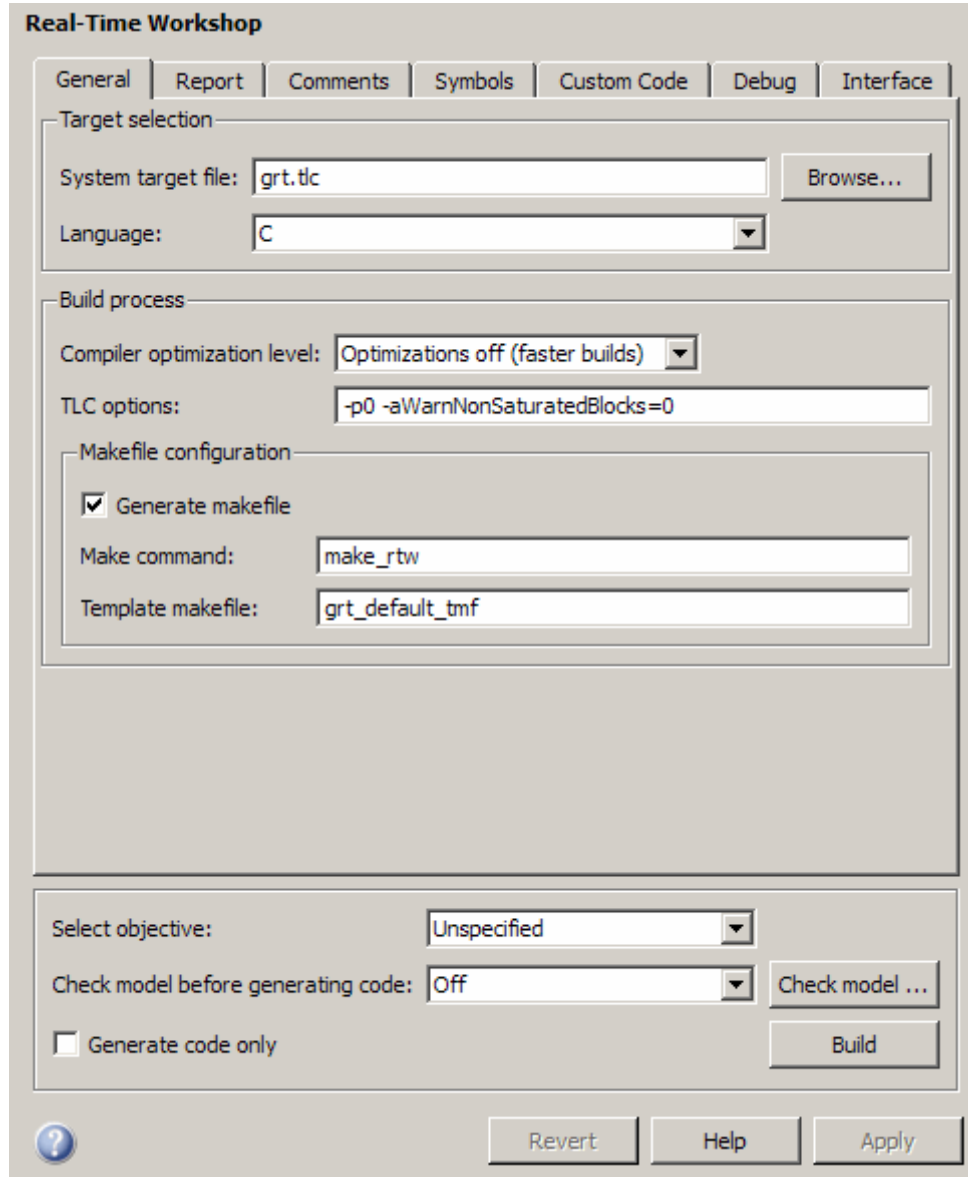
6 Click the **Browse** button next to the **System target file** field. This opens the System Target File Browser, illustrated below. The browser displays a list of all currently available target configurations. Your available configurations may differ. When you select a target configuration, the Real-Time Workshop software automatically chooses the appropriate system target file, template makefile, and `make` command. Their names appear at the bottom left of the window.

Note The system target file browser lists all system target files found on the MATLAB path. Using some of these might require additional licensed products, such as the Real-Time Workshop Embedded Coder product.

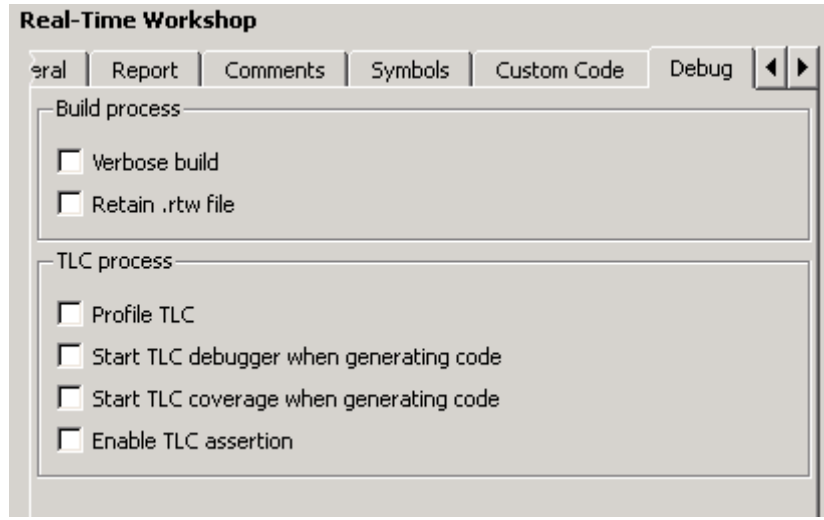


7 From the list of available configurations, select `Generic Real-Time Target` (as shown above) and then click **OK**.

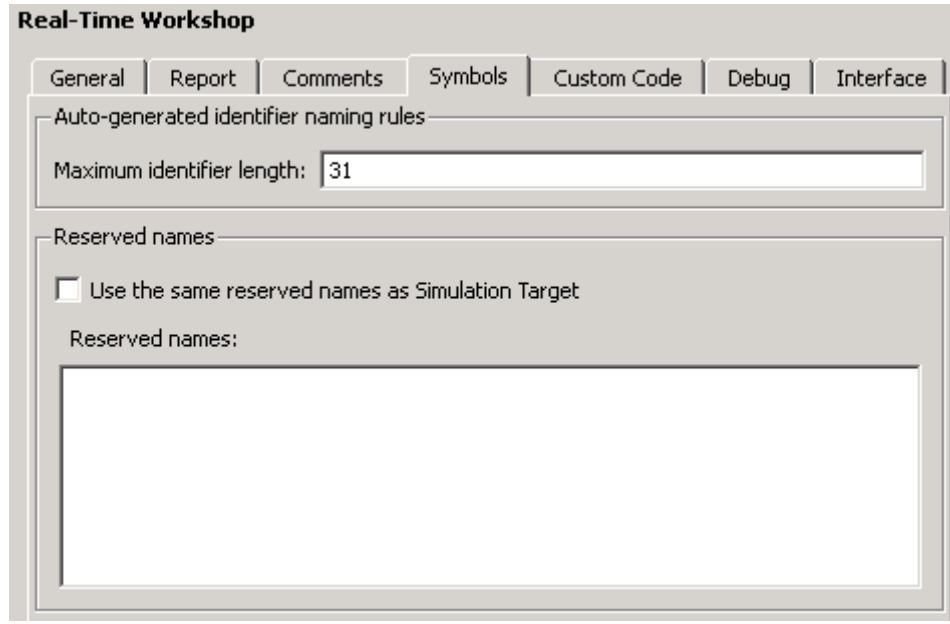
The **Real-Time Workshop** pane displays the correct system target file (`grt.tlc`), `make` command (`make_rtw`), and template makefile (`grt_default_tmf`), as shown below:



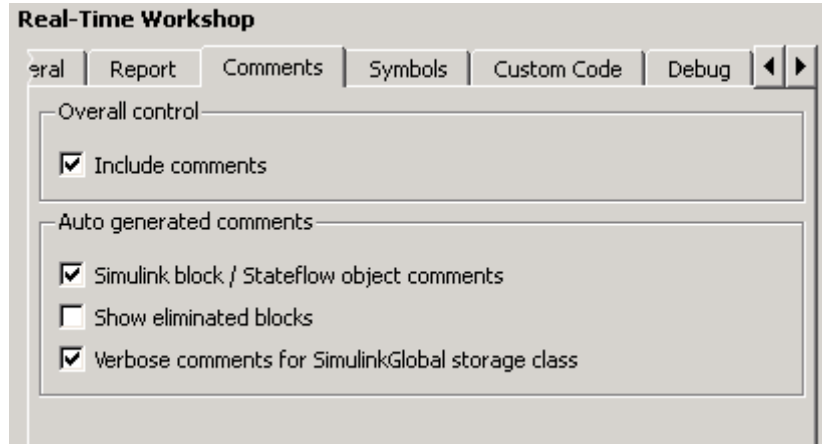
8 Select the **Debug** tab of the **Real-Time Workshop** pane. The options displayed here control build verbosity and debugging support, and are common to all target configurations. Make sure that all options are set to their defaults, as shown below.



9 Select the **Symbols** tab of the **Real-Time Workshop** pane. The options on this pane control the look and feel of generated code.



10 Select the **Comments** tab of the **Real-Time Workshop** pane. The options displayed here control the types of comments included in generated code. Make sure that all options are set to their defaults, as shown below.



11 Make sure that the **Generate code only** check box at the bottom of the pane is cleared.

12 Save the model.

Building and Running the Program

The Real-Time Workshop build process generates C code from the model, and then compiles and links the generated program to create an executable image. To build and run the program,

1 With the `f14rtw` model open, go to the Model Explorer window. In the **Real-Time Workshop** pane, select the **General** tab, then click the **Build** button to start the build process.

A number of messages concerning code generation and compilation appear in the MATLAB Command Window. The initial message is

```
### Starting Real-Time Workshop build procedure for model: f14rtw
```

The contents of many of the succeeding messages depends on your compiler and operating system. The final message is

```
### Successful completion of Real-Time Workshop build procedure for mo
```

The working directory now contains an executable, `f14rtw.exe` (Microsoft Windows platforms) or `f14rtw` (UNIX platforms). In addition, the Real-Time Workshop build process has created a project directory, `slprj`, and a build directory, `f14rtw_grt_rtw`, in your working directory.

Note The Real-Time Workshop build process displays a code generation report after generating the code for the `f14rtw` model. The tutorial "First Look at Generated Code" on page 3-33 provides more information about how to create and use a code generation report.

---

2 To observe the contents of the working directory after the build, type the `dir` command from the MATLAB Command Window.

```
dir

.               f14rtw.exe      f14rtw_grt_rtw
..              f14rtw.mdl      slprj
```

3 To run the executable from the Command Window, type

```
!f14rtw
```

The ! character passes the command that follows it to the operating system, which runs the stand-alone `f14rtw` program.

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

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

No data is output.

4 Finally, to see the files created in the build directory, type

```
dir f14rtw_grt_rtw
```

The exact list of files produced varies among MATLAB platforms and versions. Here is a sample list from a Windows platform.

```
.                  grt_main.obj        rt_nonfinite.h
..                 html                rt_nonfinite.obj
buildInfo.mat      modelsources.txt    rt_rand.c
defines.txt        ode5.obj            rt_rand.h
f14rtw.bat         rtGetInf.c          rt_rand.obj
f14rtw.c           rtGetInf.h          rt_sim.obj
f14rtw.h           rtGetInf.obj        rtmodel.h
f14rtw.mk          rtGetNaN.c          rtw_proj.tmw
f14rtw.obj         rtGetNaN.h          rtwtypes.h
f14rtw_private.h   rtGetNaN.obj        rtwtypeschksum.mat
f14rtw_ref.rsp     rt_logging.obj
f14rtw_types.h     rt_nonfinite.c
```

Contents of the Build Directory

The build process creates a build directory and names it *`model`*`_`*`target`*`_rtw`, where *`model`* is the name of the source model and *`target`* is the target selected for the model. In this example, the build directory is named `f14rtw_grt_rtw`.

The build directory includes the following generated files.

Note The code generation report you created for the `f14rtw` model in the previous section displays a link for each file listed below, which you can click to explore the file contents.

File	Description
`f14rtw.c`	Standalone C code that implements the model

`rt_nonfinite.c`
`rtGetInf.c`
`rtGetNaN.c`
	Functions to initialize nonfinite types (`Inf`, `NaN`, and `-Inf`)

`rt_rand.c`	Random functions, included only if needed by the application
`f14rtw.h`	An include header file containing definitions of parameters and state variables
`f14rtw_private.h`	Header file containing common include definitions
`f14rtw_types.h`	Forward declarations of data types used in the code

`rt_nonfinite.h`
`rtGetInf.h`
`rtGetNaN.h`
	Provides support for nonfinite numbers in the generated code, dynamically generates `Inf`, `NaN`, and `-Inf` as needed

`rt_rand.h`	Imported declarations for random functions, included only if needed by the application
`rtmodel.h`	Master header file for including generated code in the static main program (its name never changes, and it simply includes `f14rtw.h`)
`rtwtypes.h`	Static include file for Simulink `simstruct` data types; some embedded targets tailor this file to reduce overhead, but GRT does not

The build directory contains other files used in the build process, most of which you can disregard for the present:

- `f14rtw.mk` — Makefile generated from a template for the GRT target

- Object (`.obj`) files
- `f14rtw.bat` — Batch control file
- `rtw_proj.tmw` — Marker file
- `buildInfo.mat` — Build information for relocating generated code to another development environment
- `defines.txt` — Parameter definitions for accessing the application
- `f14rtw_ref.rsp` — Data to include as command-line arguments to `mex` (Windows systems only)

The build directory also contains a subdirectory, `html`, which contains the files that make up the code generation report. For more information about the code generation report, see the tutorial "First Look at Generated Code" on page 3-33.

Data Logging

In this section...

Tutorial Overview

Real-Time Workshop MAT-file data logging facility enables a generated program to save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) *model*.mat, where *model* is the name of your model. In this tutorial, data generated by the model `f14rtw.mdl` is logged to the file `f14rtw.mat`. Refer to "Building a Generic Real-Time Program" on page 3-4 for instructions on setting up `f14rtw.mdl` in a working directory if you have not done so already.

To configure data logging, click **Data Import/Export** in the center pane of the Model Explorer. The process is the same as configuring a Simulink model to save output to the MATLAB workspace. For each workspace return variable you define and enable, the Real-Time Workshop software defines a parallel MAT-file variable. For example, if you save simulation time to the variable `tout`, your generated program logs the same data to a variable named `rt_tout`. You can change the prefix `rt_` to a suffix (`_rt`), or eliminate it entirely. You do this by selecting **Real-Time Workshop** in the center pane of the Model Explorer, then clicking the **Interface** tab.

Note Simulink lets you log signal data from anywhere in a model via the **Log signal data** option in the Signal Properties dialog box (accessed via context menu by right-clicking signal lines). The Real-Time Workshop software does not use this method of signal logging in generated code. To log signals in generated code, you must either use the **Data Import/Export** options described below or include To File or To Workspace blocks in your model.

In this tutorial, you modify the `f14rtw` model so that the generated program saves the simulation time and system outputs to the file `f14rtw.mat`. Then

you load the data into the MATLAB workspace and plot simulation time against one of the outputs. The `f14rtw` model should be open and configured as it was at the end of the previous tutorial.

Data Logging During Simulation

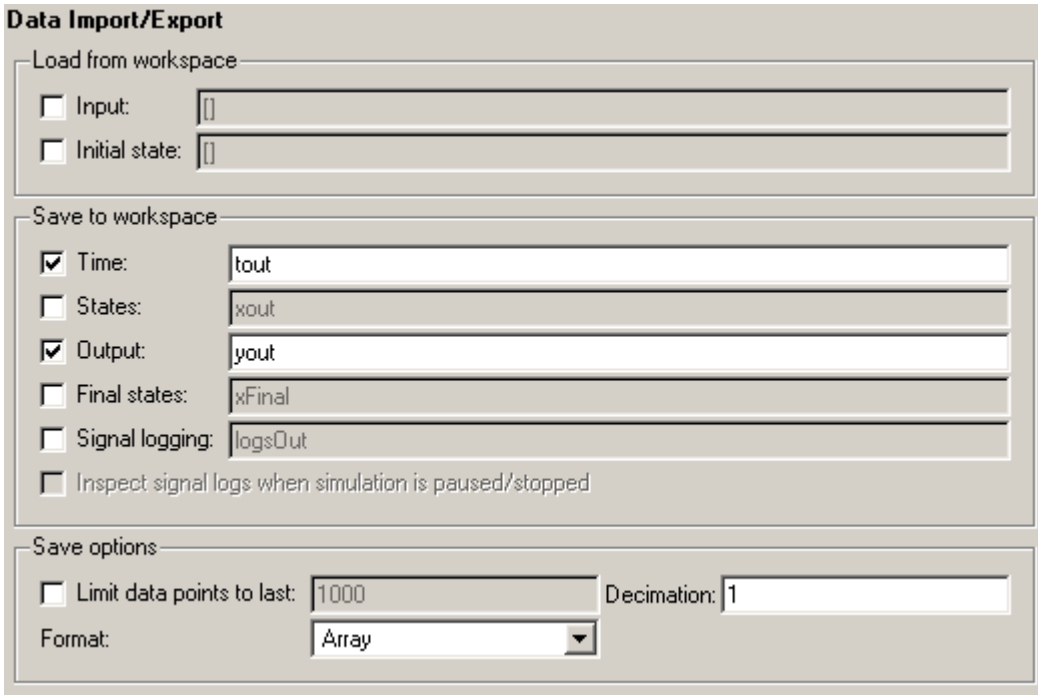
To use the data logging feature:

1. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
2. In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.
3. Click **Configuration (Active)** in the left pane.
4. Click **Data Import/Export** in the center pane. The **Data Import/Export** pane appears at the right. Its **Save to workspace** section lets you specify which outport data is to be saved to the workspace and what variable names to use for it.
5. Select the **Time** option. This tells Simulink to save time step data during simulation as a variable named `tout`. You can enter a different name to distinguish different simulation runs (for example using different step sizes), but take the default for this tutorial. Selecting **Time** enables the Real-Time Workshop code generator to create code that logs the simulation time to a MAT-file.
6. Select the **Output** option. This tells Simulink to save output signal data during simulation as a variable named `yout`. Selecting **Output** enables the Real-Time Workshop code generator to create code that logs the root Output blocks (Angle of Attack and Pilot G Force) to a MAT-file.

Note The sort order of the `yout` array is based on the port number of the Outport blocks, starting with 1. Angle of Attack and Pilot G Force are logged to `yout(:,1)` and `yout(:,2)`, respectively.

---

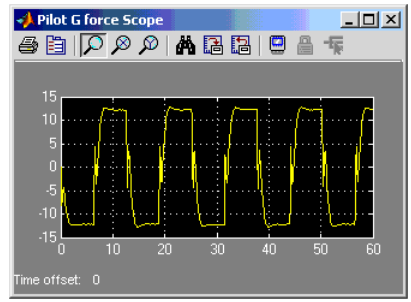
7 If any other options are enabled, clear them. Set **Decimation** to `1` and **Format** to `Array`. The figure below shows the dialog.



8 Click **Apply** to register your changes.

9 Save the model.

10 Open the Pilot G Force Scope block of the model, then run the model by choosing **Simulation > Start** in the model window. The resulting Pilot G Force scope display is shown below.



11 Verify that the simulation time and outputs have been saved to the MATLAB workspace in MAT-files. At the MATLAB prompt, type:

```
whos *out
```

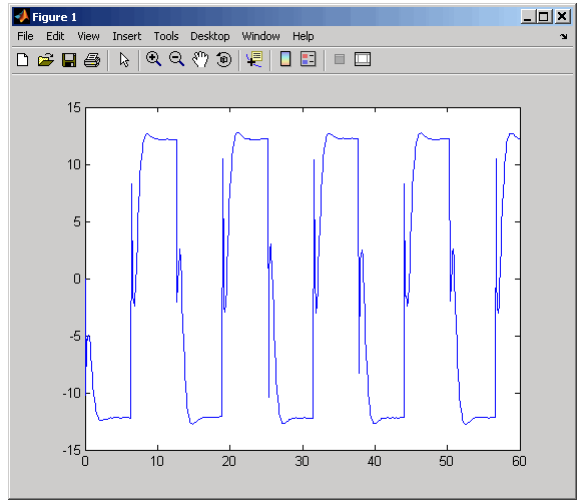
Simulink displays:

```
  Name        Size                    Bytes  Class Attributes

  tout      601x1                      4808  double
  yout      601x2                      9616  double
```

12 Verify that Pilot G Force was correctly logged by plotting simulation time versus that variable. At the MATLAB prompt, type:

```
plot(tout,yout(:,2))
```

The resulting plot is shown below.



Data Logging from Generated Code

In the second part of this tutorial, you build and run a Real-Time Workshop executable of the `f14rtw` model that outputs a MAT-file containing the simulation time and outputs you previously examined. Even though you have already generated code for the `f14rtw` model, you must now regenerate that code because you have changed the model by enabling data logging. The steps below explain this procedure.

To avoid overwriting workspace data with data from simulation runs, the Real-Time Workshop code generator modifies identifiers for variables logged by Simulink. You can control these modifications from the **Model Explorer**:

1 Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.

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

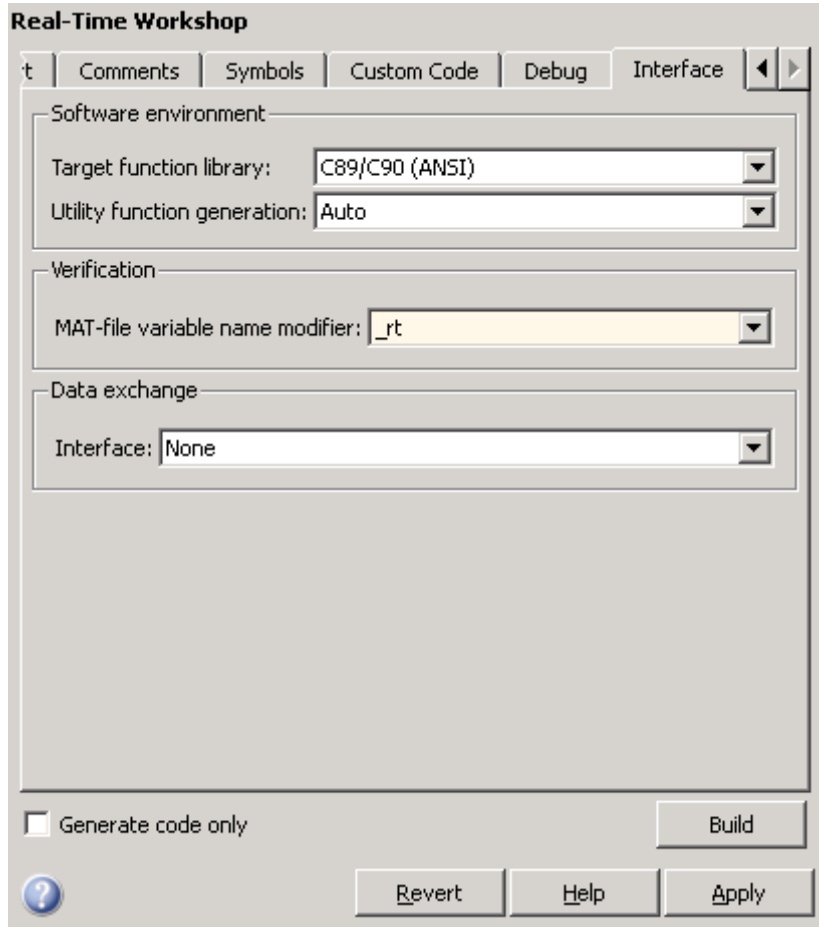
3. Click **Configuration (Active)** in the left pane.

4. In the center pane, click **Real-Time Workshop**. The **Real-Time Workshop** pane appears to the right.

5. Click the **Interface** tab.

6. Set **MAT-file variable name modifier** to `_rt`. This adds the suffix `_rt` to each variable that you selected to be logged in the first part of this tutorial (`tout`, `yout`).

7 Clear the **Generate code only** check box, if it is currently selected. The pane should look like this:



8 Click **Apply** to register your changes.

9 Save the model.

10 To generate code and build an executable, click the **Build** button.

11 When the build concludes, run the executable with the command:

```
!f14rtw
```

12 The program now produces two message lines, indicating that the MAT-file has been written.

```
** starting the model **
** created f14rtw.mat **
```

13 Load the MAT-file data created by the executable and look at the workspace variables from simulation and the generated program by typing:

```
load f14rtw.mat
whos tout* yout*
```

Simulink displays:

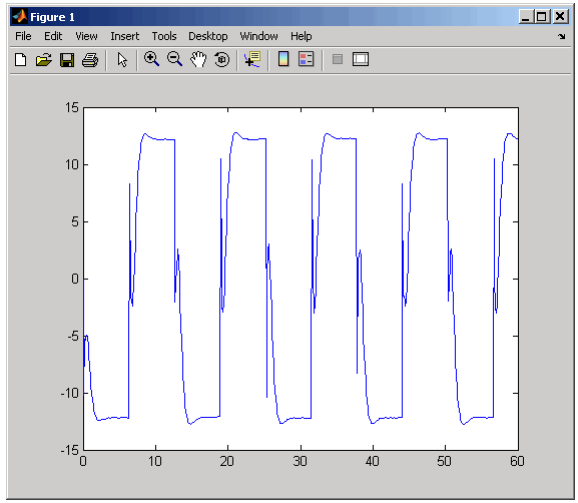
```
  Name          Size                    Bytes  Class Attribute

  tout        601x1                      4808  double
  tout_rt     601x1                      4808  double
  yout        601x2                      9616  double
  yout_rt     601x2                      9616  double
```

Note that all arrays have the same number of elements.

14 Observe that the variables `tout_rt` (time) and `yout_rt` (Pilot G Force and Angle of Attack) have been loaded from the file. Plot Pilot G Force as a function of time.

```
plot(tout_rt,yout_rt(:,2))
```

The resulting plot is identical to the plot you produced in step 10 of the previous part of this tutorial:



Code Verification

In this section...

Tutorial Overview

In this tutorial, you verify the answers computed by code generated from the `f14rtw` model. You do this by capturing two sets of output data and comparing the sets. You obtain one set by running the Simulink model, and the other set by executing the Real-Time Workshop generated code.

Note To obtain a valid comparison between outputs of the model and the generated program, you must use the same **Solver options** and the same **Step size** for both the Simulink run and the Real-Time Workshop build process, and the model must be configured to save simulation time, as shown in the preceding tutorial.

Logging Signals via Scope Blocks

This example uses Scope blocks (rather than Outport blocks) to log output data. The `f14rtw` model should be configured as it was at the end of the previous tutorial, "Data Logging" on page 3-18.

To configure the Scope blocks to log data,

1. Save the model if any unsaved changes exist.
2. Clear the MATLAB workspace to eliminate the results of previous simulation runs. At the MATLAB prompt, type:

```
clear
```

The clear operation clears not only variables created during previous simulations, but all workspace variables, some of which are standard variables that the `f14rtw` model requires.

3. Reload the model so that the standard workspace variables are redeclared and initialized:

   a. Close the model by clicking its window's **Close** box.

   b. At the MATLAB prompt, type:

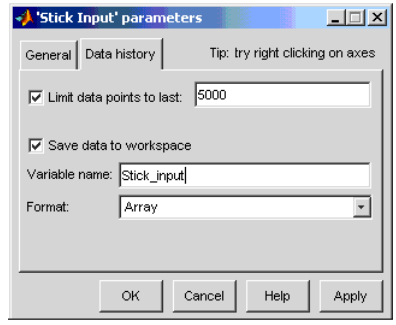
   ```
   f14rtw
   ```

   The model reopens, which declares and initializes the standard workspace variables.

4. Open the Stick Input Scope block and click the **Parameters** button (the second button from the left) on the toolbar of the Scope window. The Stick Input Parameters dialog box opens.

5. Click the **Data History** tab of the Stick Input Parameters dialog box.

6. Select the **Save data to workspace** option and change the **Variable name** to `Stick_input`. The dialog box appears as follows:



7. Click **OK**.

The Stick Input parameters now specify that the Stick Input signal to the Scope block will be logged to the array `Stick_input` during simulation.

The generated code will log the same signal data to the MAT-file variable `rt_Stick_input` during a run of the executable program.

8 Configure the Pilot G Force and Angle of Attack Scope blocks similarly, using the variable names `Pilot_G_force` and `Angle_of_attack`.

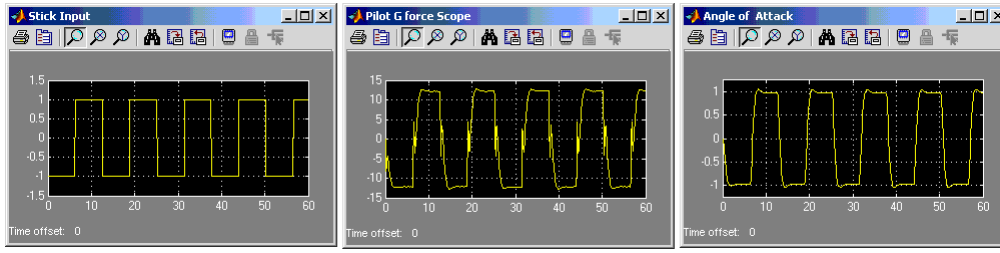
9 Save the model.

Logging Simulation Data

The next step is to run the simulation and log the signal data from the Scope blocks:

1 Open the Stick Input, Pilot G Force, and Angle of Attack Scope blocks.

2 Run the model. The three Scope plots look like this:



3 Use the `whos` command to show that the array variables `Stick_input`, `Pilot_G_force`, and `Angle_of_attack` have been saved to the workspace.

4 Plot one or more of the logged variables against simulation time. For example,

```
plot(tout, Stick_input(:,2))
```

Logging Data from the Generated Program

Because you have modified the model, you must rebuild and run the `f14rtw` executable to obtain a valid data file:

1 Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.

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

3. Click **Configuration (Active)** in the left pane.

4. Select **Real-Time Workshop** on the center pane of the **Model Explorer**, and click the **Interface** tab. The **Interface** pane appears.

5. Set the **MAT-file variable name modifier** menu to `rt_`. This prefixes `rt_` to each variable that you selected to be logged in the first part of this tutorial.

6. Click **Apply**.

7. Save the model.

8. Generate code and build an executable by clicking the **Build** button. Status messages in the MATLAB Command Window track the build process.

9. When the build finishes, run the stand-alone program from MATLAB.

   ```
   !f14rtw
   ```

   The executing program writes the following messages to the MATLAB Command Window.

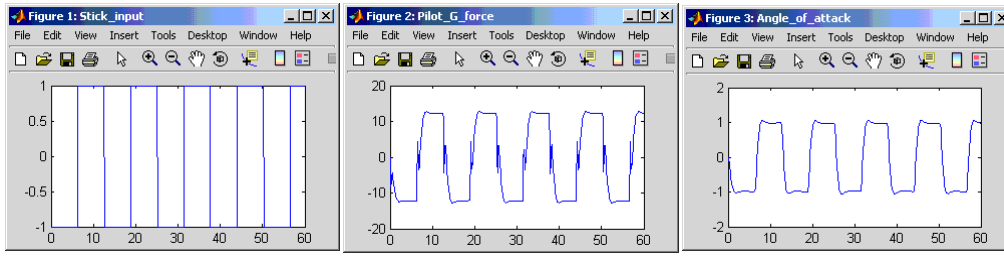
   ```
   ** starting the model **
   ** created f14rtw.mat **
   ```

10. Load the data file `f14rtw.mat` and observe the workspace variables.

```
>> load f14rtw
>> whos rt*
  Name                      Size            Bytes  Class     Attributes

  rt_Angle_of_attack      601x2              9616  double
  rt_Pilot_G_force        601x2              9616  double
  rt_Stick_input          601x2              9616  double
  rt_tout                 601x1              4808  double
  rt_yout                 601x2              9616  double
```

11 Use MATLAB to plot three workspace variables created by the executing program as a function of time.

```
figure('Name','Stick_input')
plot(rt_tout,rt_Stick_input(:,2))
figure('Name','Pilot_G_force')
plot(rt_tout,rt_Pilot_G_force(:,2))
figure('Name','Angle_of_attack')
plot(rt_tout,rt_Angle_of_attack(:,2))
```



Your Simulink simulations and the generated code have apparently produced nearly identical output. The next section shows how to quantify this similarity.

Comparing Numerical Results of the Simulation and the Generated Program

You have now obtained data from a Simulink run of the model and from a run of the program generated from the model. It is a simple matter to compare the `f14rtw` model output to the Real-Time Workshop results. Your comparison results may differ from those shown below.

To compare `Angle_of_attack` (simulation output) to `rt_Angle_of_attack` (generated program output), type:

```
max(abs(rt_Angle_of_attack-Angle_of_attack))
```

MATLAB prints:

```
ans =
1.0e-015 *
         0    0.3331
```

Similarly, the comparison of `Pilot_G_force` (simulation output) to `rt_Pilot_G_force` (generated program output) is:

```
max(abs(rt_Pilot_G_force-Pilot_G_force))
1.0e-013 *
         0    0.4974
```

Overall agreement is within 10^{-13}. The means of residuals are an order of magnitude smaller. This slight error can be caused by many factors, including

- Different compiler optimizations
- Statement ordering
- Runtime libraries

For example, a function call such as `sin(2.0)` might return a slightly different value depending on which C library you are using. Such variations can also cause differences between your results and those shown above.

First Look at Generated Code

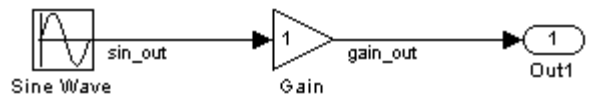
In this section...

Tutorial Overview

In this tutorial, you examine code generated from a simple model to observe the effects of some of the many Real-Time Workshop code optimization features.

Note You can view the code generated from this example using the MATLAB editor. You can also view the code in the MATLAB Help browser if you enable the **Create HTML report** option before generating code. See the following section, "HTML Code Generation Reports" on page 3-44, for an introduction to using the HTML report feature.

The source model, `example.mdl`, is shown below.



Setting Up the Model

First, create the model from Simulink library blocks, and set up basic Simulink and Real-Time Workshop parameters as follows:

1. Create a directory, `example_codegen`, and make it your working directory:

   ```
   !mkdir example_codegen
   cd example_codegen
   ```

2. Create a new model and save it as `example.mdl`.

3. Add Sine Wave, Gain, and Out1 blocks to your model and connect them as shown in the preceding diagram. Label the signals as shown.

4. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.

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

6. Click **Configuration (Active)** in the left pane.

7. Select **Solver** in the center pane. The **Solver** pane appears at the right.

8. In the **Solver Options** pane:

   a. Select `Fixed-step` in the **Type** field.

   b. Select `Discrete (no continuous states)` in the **Solver** field.

   c. Specify `0.1` in the **Fixed-step size** field. (Otherwise, the Real-Time Workshop code generator posts a warning and supplies a default value when you generate code.)

9. Click **Apply**.

10. Click **Data Import/Export** in the center pane and make sure all check boxes in the right pane are cleared. Click **Apply** if you made any changes.

11. Select **Real-Time Workshop** in the center pane. Under **Target Selection** in the right pane, select the default generic real-time (GRT) target `grt.tlc`.

12. Select **Generate code only** at the bottom of the right pane. This option causes the Real-Time Workshop software to generate code and a `make` file, then stop at that point, rather than proceeding to invoke `make` to compile and link the code. Note that the label on the **Build** button changes to **Generate code**.

13 Click **Apply**.

14 Save the model.

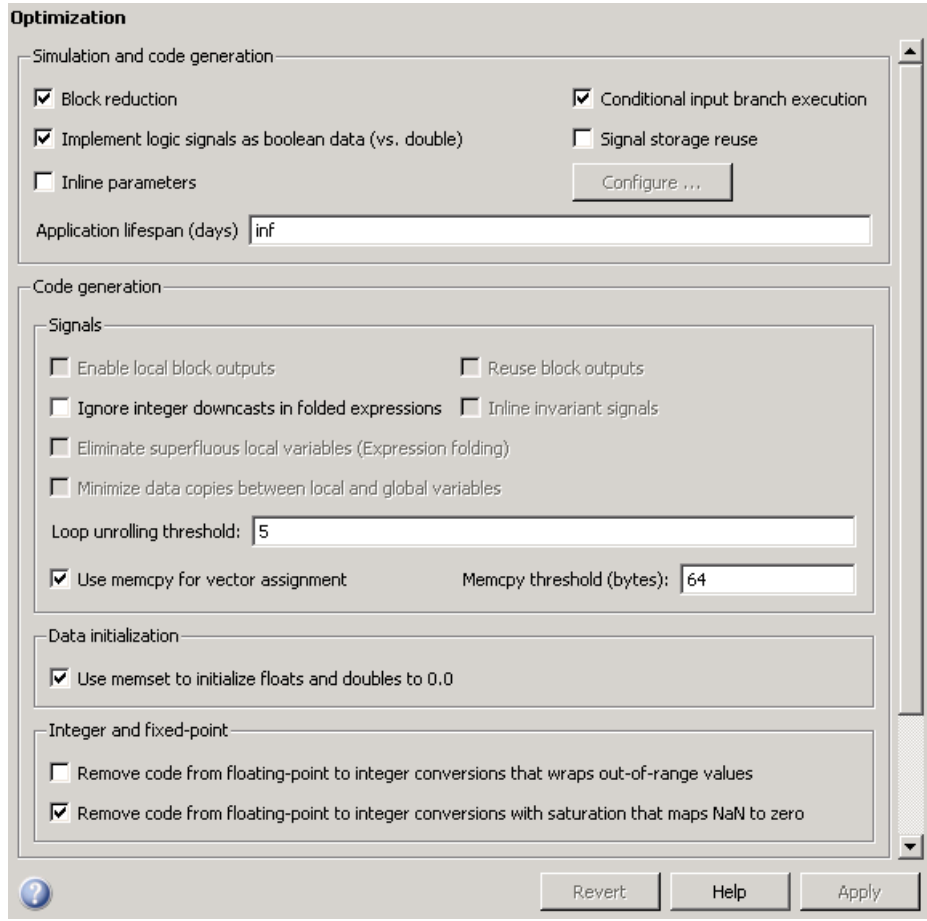
Generating Code Without Buffer Optimization

With buffer optimizations Real-Time Workshop software generates code that reduces memory consumption and execution time. In this tutorial, you disable the buffer optimizations to see what the nonoptimized generated code looks like:

1 Select **Optimization** in the center pane. The **Optimization** pane appears at the right. Clear the **Signal storage reuse** option, as shown below. Change any other attributes as needed to match the figure.

Note Clearing the **Signal storage reuse** option disables the following options:

- **Enable local block outputs**
- **Reuse block outputs**
- **Eliminate superfluous local variables (Expression folding)**
- **Minimize data copies between local and global variables**



2 Click **Apply**.

3 Select **Real-Time Workshop > Report** in the center pane. The **Report** pane appears at the right.

4 Select the **Create code generation report** and **Launch report automatically** check boxes. Selecting these check boxes makes the code generation report display after the build process completes.

5 Select **Real-Time Workshop** in the center pane, and click **Generate code** on the right.

6 Because you selected the **Generate code only** option, the Real-Time Workshop build process does not invoke your `make` utility. The code generation process ends with this message:

```
### Successful completion of Real-Time Workshop
build procedure for model: example
```

7 The generated code is in the build directory, `example_grt_rtw`. The file `example_grt_rtw/example.c` contains the output computation for the model. You can view this file in the code generation report by clicking the `example.c` link in the left pane.

8 In `example.c`, find the function `example_output` near the top of the file.

The generated C code consists of procedures that implement the algorithms defined by your Simulink block diagram. The execution engine calls the procedures in proper succession as time moves forward. The modules that implement the execution engine and other capabilities are referred to collectively as the run-time interface modules. See the Real-Time Workshop User's Guide for a complete discussion of how the Real-Time Workshop software interfaces and executes application, system-dependent, and system-independent modules, in each of the two styles of generated code.

In code generated for `example`, the generated `example_output` function implements the actual algorithm for multiplying a sine wave by a gain. The `example_output` function computes the model's block outputs. The run-time interface must call `example_output` at every time step. With buffer optimizations turned off, `example_output` assigns unique buffers to each block output. These buffers (`rtB.sin_out`, `rtB.gain_out`) are members of a global block I/O data structure, called in this code `example_B` and declared as follows:

```
/* Block signals (auto storage) */
BlockIO_example example_B;
```

The data type `BlockIO_example` is defined in `example.h` as follows:

```
/* Block signals (auto storage) */
extern BlockIO_example example_B;
```

The output code accesses fields of this global structure, as shown below:

```
/* Model output function */
static void example_output(int_T tid)
{
  /* Sin: '<Root>/Sine Wave' */
  example_B.sin_out = sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
    example_P.SineWave_Phase) * example_P.SineWave_Amp + example_P.SineWave_Bias;

  /* Gain: '<Root>/Gain' */
  example_B.gain_out = example_P.Gain_Gain * example_B.sin_out;

  /* Outport: '<Root>/Out1' */
  example_Y.Out1 = example_B.gain_out;

/* tid is required for a uniform function interface.
 * Argument tid is not used in the function. */
UNUSED_PARAMETER(tid);
}
```

9 In GRT targets such as this, the function `example_output` is called by a wrapper function, `MdlOutputs`. In `example.c`, find the function `MdlOutputs` near the end. It looks like this:

```
void MdlOutputs(int_T tid)
{
  example_output(tid);
}
```

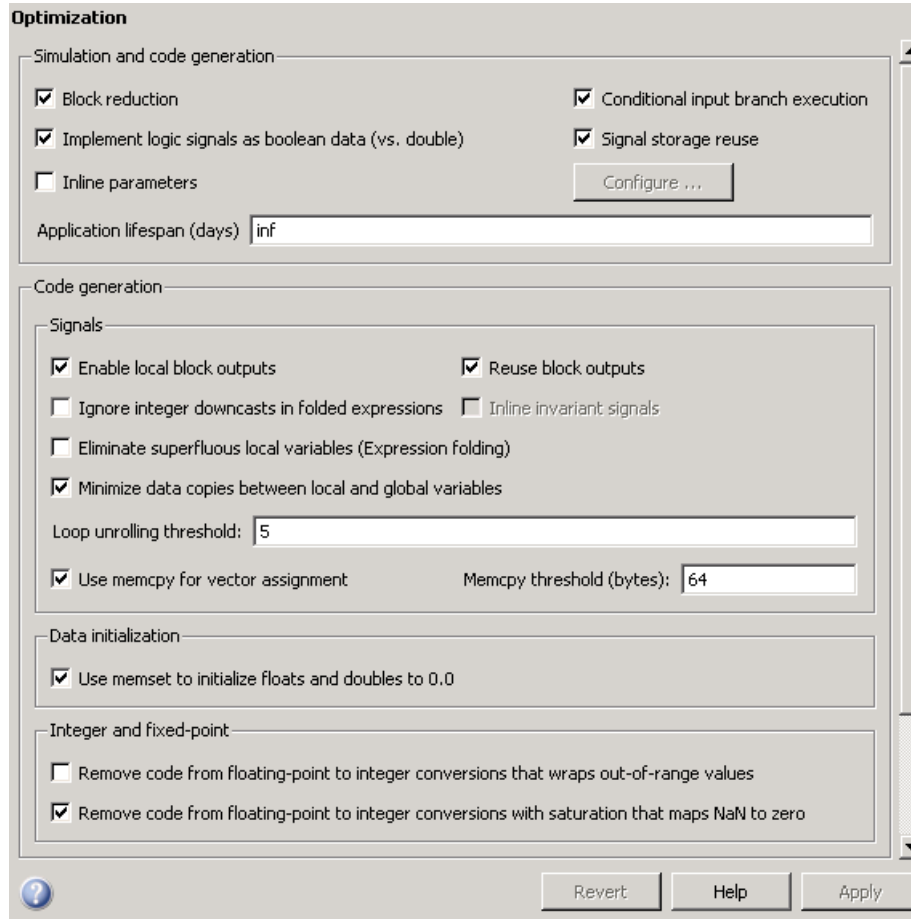
Note In previous releases, `MdlOutputs` was the actual output function for code generated by all GRT-configured models. It is now implemented as a wrapper function to provide greater compatibility among different target configurations.

Generating Code with Buffer Optimization

With buffer optimizations, Real-Time Workshop software generates code that reduces memory consumption and execution time. In this tutorial, you turn buffer optimizations on and observe how they improve the code. Enable signal buffer optimizations and regenerate the code as follows:

1. Change your current working directory to `example_codegen` if you have not already done so.
2. Select **Optimization** in the center pane. The **Optimization** pane appears at the right. Select the **Signal storage reuse** option.
3. Note that the following parameters become enabled in the **Code generation** section:
   - **Enable local block outputs**
   - **Reuse block outputs**
   - **Eliminate superfluous local variables (Expression folding)**
   - **Minimize data copies between local and global variables**

Make sure that **Enable local block outputs**, **Reuse block outputs**, and **Minimize data copies between local and global variables** are selected, and that **Eliminate superfluous local variables (Expression folding)** is cleared, as shown below.



You will observe the effects of expression folding later in this tutorial. Not performing expression folding allows you to see the effects of the buffer optimizations.

4 Click **Apply** to apply the new settings.

5 Select **Real-Time Workshop** in the center pane, and click **Generate code** on the right.

   As before, the Real-Time Workshop software generates code in the `example_grt_rtw` directory. The previously generated code is overwritten.

6 View `example.c` and examine the `example_output` function.

With buffer optimizations enabled, the code in `example_output` reuses the `example_Y.Out1` global buffer.

```
/* Model output function */
static void example_output(int_T tid)
{
  /* Sin: '<Root>/Sine Wave' */
  example_Y.Out1 = sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
                       example_P.SineWave_Phase) * example_P.SineWave_Amp +
    example_P.SineWave_Bias;

  /* Gain: '<Root>/Gain' */
  example_Y.Out1 = example_P.Gain_Gain * example_Y.Out1;

  /* tid is required for a uniform function interface.
  * Argument tid is not used in the function. */
  UNUSED_PARAMETER(tid);
}
```

This code is more efficient in terms of memory usage. The efficiency improvement gained by selecting **Enable local block outputs**, **Reuse block outputs**, and **Minimize data copies between local and global variables** would be more significant in a large model with many signals.

Further Optimization: Expression Folding

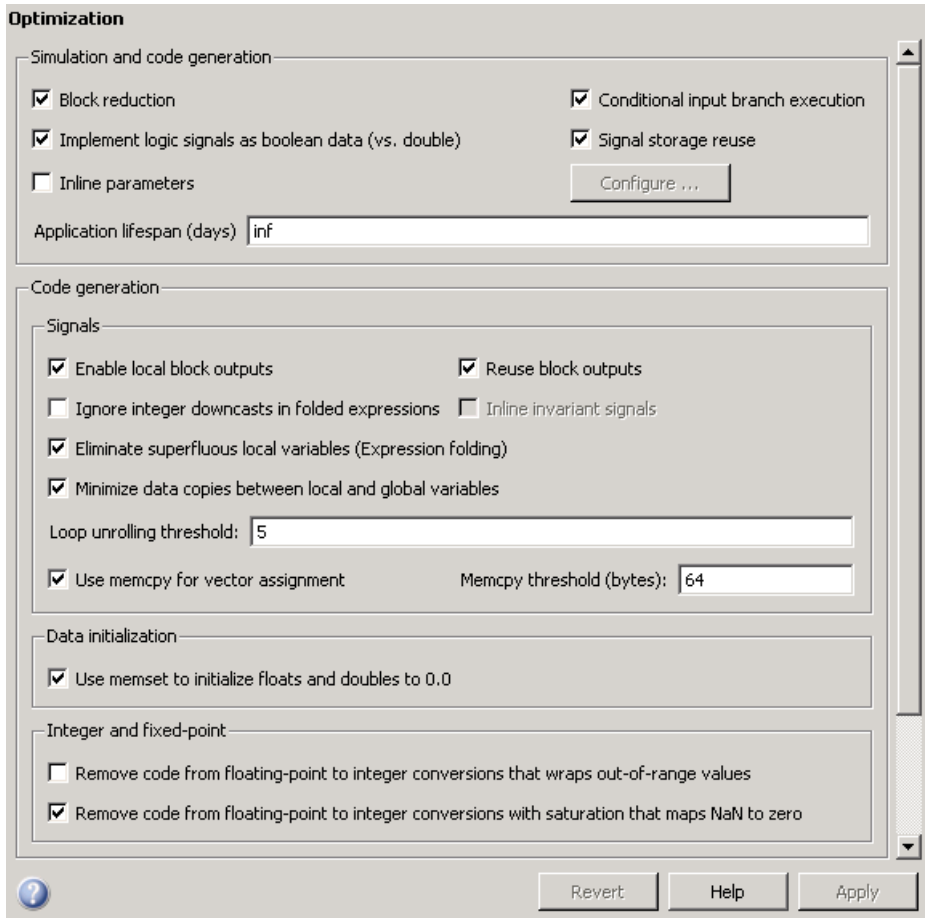
As a final optimization, you turn on expression folding, a code optimization technique that minimizes the computation of intermediate results and the use of temporary buffers or variables.

Enable expression folding and regenerate the code as follows:

1 Change your current working directory to `example_codegen` if you have not already done so.

2 Select **Optimization** in the center pane. The **Optimization** pane appears.

3 Select the **Eliminate superfluous local variables (Expression folding)** option.



4 Click **Apply**.

5 Select **Real-Time Workshop** in the center pane, and click **Generate code** on the right.

   The Real-Time Workshop software generates code as before.

6 View `example.c` and examine the function `example_output`.

In the previous examples, the Gain block computation was computed in a separate code statement and the result was stored in a temporary location before the final output computation.

With **Eliminate superfluous local variables (Expression folding)** selected, there is a subtle but significant difference in the generated code: the gain computation is incorporated (or folded) directly into the Outport computation, eliminating the temporary location and separate code statement. This computation is on the last line of the `example_output` function.

```
/* Model output function */
static void example_output(int_T tid)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Sin: '<Root>/Sine Wave'
   */
  example_Y.Out1 = (sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
                        example_P.SineWave_Phase) * example_P.SineWave_Amp +
                    example_P.SineWave_Bias) * example_P.Gain_Gain;

  /* tid is required for a uniform function interface.
  * Argument tid is not used in the function. */
  UNUSED_PARAMETER(tid);
}
```

In many cases, expression folding can incorporate entire model computations into a single, highly optimized line of code. Expression folding is turned on by default. Using this option will improve the efficiency of generated code.

HTML Code Generation Reports

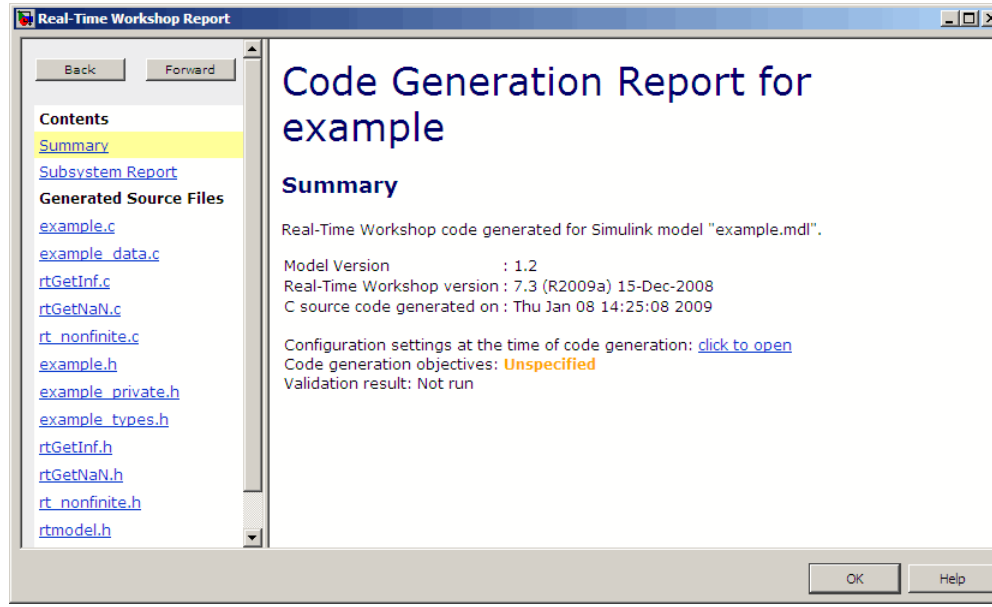
When the **Create code generation report** check box on the **Real-Time Workshop > Report** pane is selected, a navigable summary of source files is produced when the model is built. See the figure below.

Selecting this option causes the Real-Time Workshop software to produce an HTML file for each generated source file, plus a summary and an index file, in a directory named `html` within the build directory. If the **Launch report automatically** option (which is enabled by selecting **Create code generation report**) is also selected, the HTML summary and index are automatically displayed.

In the HTML report, you can click links in the report to inspect source and include files, and view relevant documentation. In these reports,

- Global variable instances are hyperlinked to their definitions.
- Block header comments in source files are hyperlinked back to the model; clicking one of these causes the block that generated that section of code to be highlighted (this feature requires a Real-Time Workshop Embedded Coder license and the ERT target).

An HTML report for the `example.mdl` GRT target is shown below.



One useful feature of HTML reports is the link on the Summary page identifying Configuration Settings at the Time of Code Generation. Clicking this opens a read-only Configuration Parameters dialog box through which you can navigate to identify the settings of every option in place at the time that the HTML report was generated.

You can refer to HTML reports at any time. To review an existing HTML report after you have closed its window, use any HTML browser to open the file `html/`*`model`*`_codgen_rpt.html` within your build directory.

Note The contents of HTML reports for different target types vary, and reports for models with subsystems feature additional information. You can also view HTML-formatted files and text files for generated code and model reference targets within **Model Explorer**. See "Generating Code for a Referenced Model" on page 3-61 for more information.

For further information on configuring and optimizing generated code, consult these sections of the Real-Time Workshop User's Guide documentation:

- "Code Generation and the Build Process" contains overviews of controlling optimizations and other code generation options.
- "Optimizing a Model for Code Generation" has additional details on signal reuse, expression folding, and other code optimization techniques.
- "Program Architecture" has details on the structure and execution of *model*.c files.

Working with External Mode Using GRT

In this section...

Tutorial Overview

This section provides step-by-step instructions for getting started with external mode, a very useful environment for rapid prototyping. The tutorial consists of four parts, each of which depends on completion of the preceding ones, in order. The four parts correspond to the steps that you follow in simulating, building, and tuning an actual real-time application:

1 Set up the model.

2 Build the target executable.

3 Run the external mode target program.

4 Tune parameters.

The example presented uses the generic real-time target, and does not require any hardware other than the computer on which you run the Simulink and Real-Time Workshop software. The generated executable in this example runs on the host computer in a separate process from MATLAB and Simulink.

The procedures for building, running, and testing your programs are almost identical in UNIX and PC environments. The discussion notes differences where applicable.

For a more thorough description of external mode, including a discussion of all the options available, see "Using the External Mode User Interface" in the Real-Time Workshop documentation.

Setting Up the Model

In this part of the tutorial, you create a simple model, `extmode_example`, and a directory called `ext_mode_example` to store the model and the generated executable:

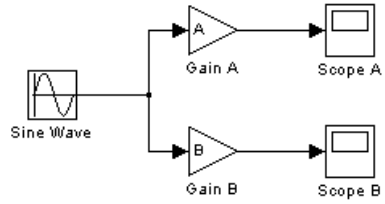
1. Create the directory from the MATLAB command line by typing

   ```
   mkdir ext_mode_example
   ```

2. Make `ext_mode_example` your working directory:

   ```
   cd ext_mode_example
   ```

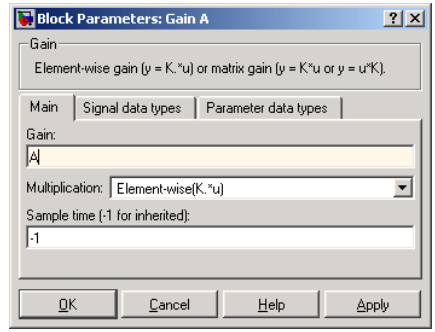
3. Create a model in Simulink with a Sine Wave block for the input signal, two Gain blocks in parallel, and two Scope blocks. The model is shown below. Be sure to label the Gain and Scope blocks as shown, so that subsequent steps will be clear to you.



4. Define and assign two variables `A` and `B` in the MATLAB workspace as follows:

   ```
   A = 2; B = 3;
   ```

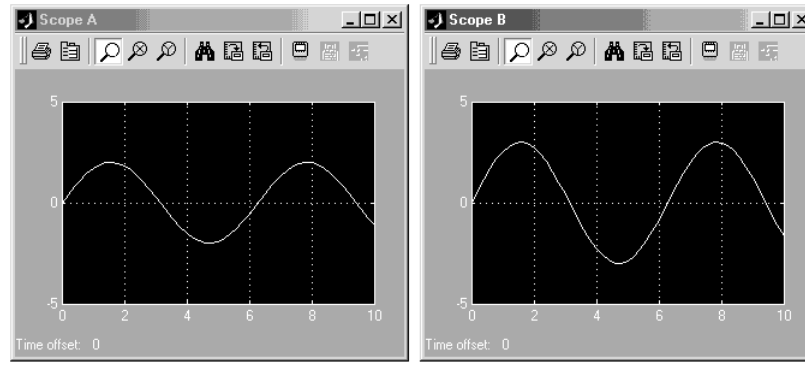
5 Open Gain block A and set its **Gain** parameter to the variable `A`.



6 Similarly, open Gain block B and set its **Gain** parameter to the variable `B`.

When the target program is built and connected to Simulink in external mode, you can download new gain values to the executing target program by assigning new values to workspace variables `A` and `B`, or by editing the values in the block parameters dialog. You explore this in "Tuning Parameters" on page 3-59.

7 Verify correct operation of the model. Open the Scope blocks and run the model. When `A = 2` and `B = 3`, the output looks like this.



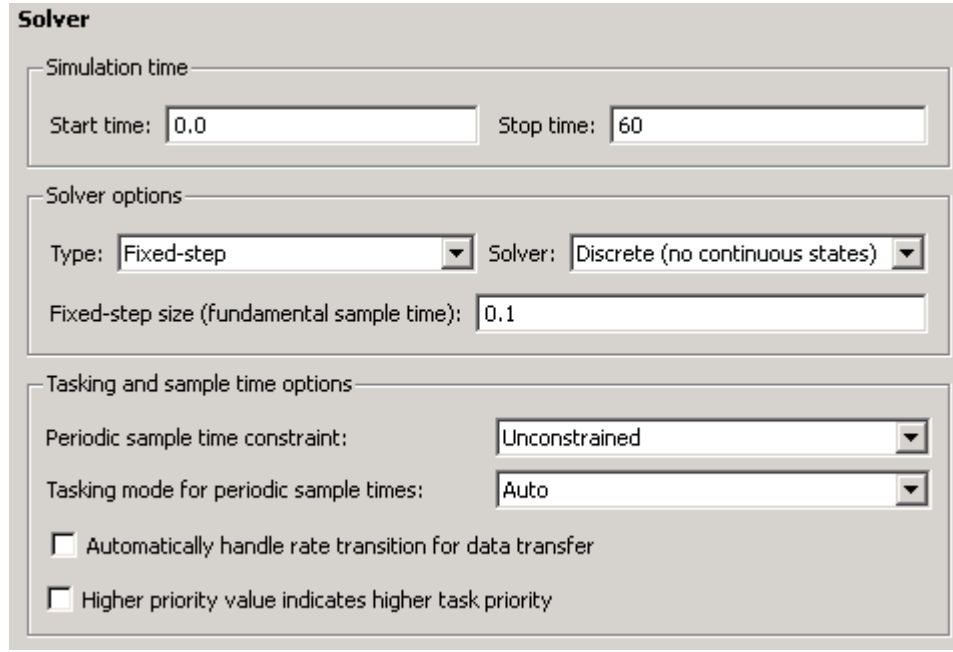
8 From the **File** menu, choose **Save As**. Save the model as `extmode_example.mdl`.

Building the Target Executable

In this section, you set up the model and code generation parameters required for an external mode compatible target program. Then you generate code and build the target executable:

1. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
2. In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.
3. Click **Configuration (Active)** in the left pane.
4. Select **Solver** in the center pane. The **Solver** pane appears at the right.
5. In the **Solver Options** pane:
   a. Select `Fixed-step` in the **Type** field.
   b. Select `Discrete (no continuous states)` in the **Solver** field.
   c. Specify `0.1` in the **Fixed-step size** field. (Otherwise, the Real-Time Workshop build process posts a warning and supplies a value when you generate code.)

6 Click **Apply**. The dialog box appears below. Note that after you click **Apply**, the controls you changed again have a white background color.



7 Click **Data Import/Export** in the center pane, and clear the **Time** and **Output** check boxes. In this tutorial, data is not logged to the workspace or to a MAT-file. Click **Apply**.

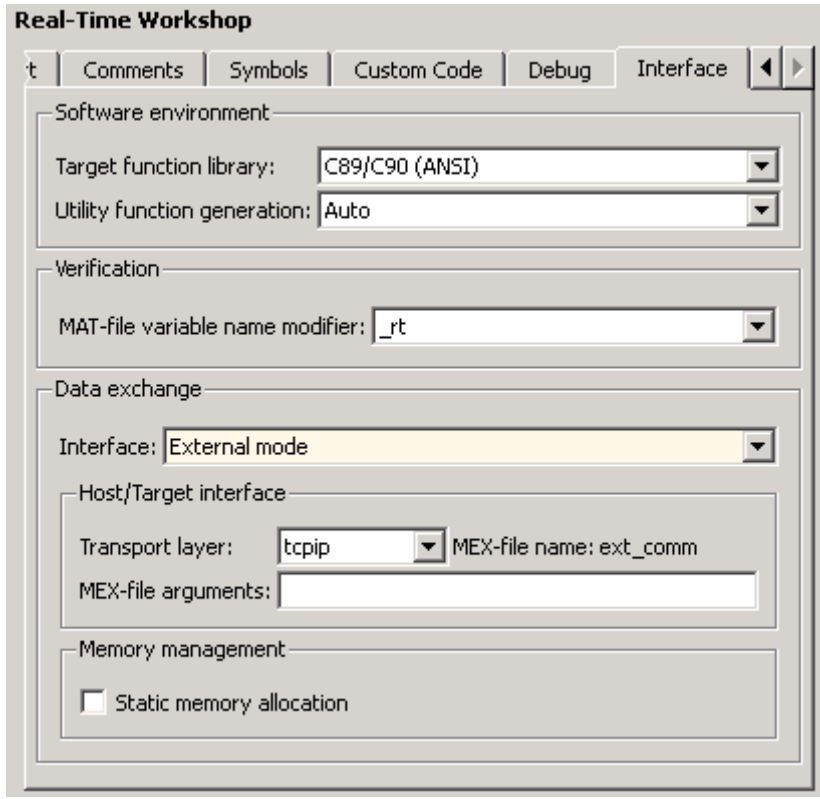
8 Click **Optimization** in the center pane. Make sure that the **Inline parameters** option is *not* selected. Although external mode supports inlined parameters, you will not explore them in this tutorial. Click **Apply** if you have made any changes.

9 Click **Real-Time Workshop** in the center pane. By default, the generic real-time (GRT) target is selected on the **Real-Time Workshop** pane. Select the **Interface** tab. The **Interface** pane appears at the right.

10 In the **Interface** pane, select `External mode` from the **Interface** pull-down menu in the **Data exchange** section. This enables generation

of external mode support code and reveals two more sections of controls: **Host/Target interface** and **Memory management**.

11 Set the **Transport layer** pull-down menu in the **Host/Target interface** section to `tcpip`. The pane now looks like this:



External mode supports communication via TCP/IP, serial, and custom transport protocols. The **MEX-file name** field specifies the name of a MEX-file that implements host and target communications on the host side. The default for TCP/IP is `ext_comm`, a MEX-file provided with the Real-Time Workshop software. You can override this default by supplying appropriate files. See "Creating an External Mode Communication Channel" in the Real-Time Workshop documentation for details if you need to support other transport layers.

The **MEX-file arguments** field lets you specify arguments, such as a TCP/IP server port number, to be passed to the external interface program. Note that these arguments are specific to the external interface you are using. For information on setting these arguments, see "MEX-File Optional Arguments for TCP/IP Transport" and "MEX-File Optional Arguments for Serial Transport" in the Real-Time Workshop documentation.

This tutorial uses the default arguments. Leave the **MEX-file arguments** field blank.

12 Click **Apply** to save the **Interface** settings.

13 Save the model.

14 Click **Real-Time Workshop** in the center pane of the **Model Explorer**. On the right, make sure that **Generate code only** is cleared, then click the **Build** button to generate code and create the target program. The content of subsequent messages depends on your compiler and operating system. The final message is

```
### Successful completion of Real-Time Workshop
build procedure for model: extmode_example
```

In the next section, you will run the `extmode_example` executable and use Simulink as an interactive front end to the running target program.

Running the External Mode Target Program

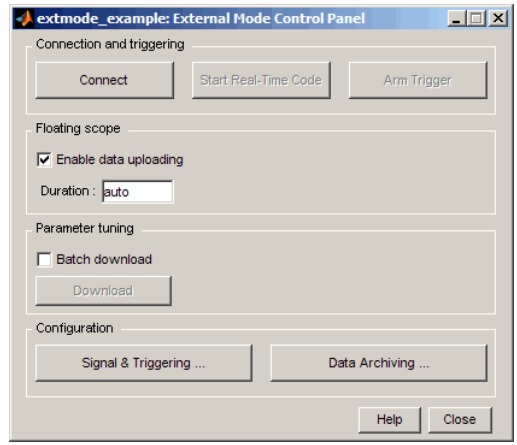
The target executable, `extmode_example`, is now in your working directory. In this section, you run the target program and establish communication between Simulink and the target.

Note An external-mode program like `extmode_example` is a host-based executable. Its execution is not tied to RTOS or a periodic timer interrupt, and it does not run in real time. The program just runs as fast as possible, and the time units it counts off are simulated time units that do not correspond to time in the world outside the program.

The External Signal & Triggering dialog box (accessed via the **External Mode Control Panel**) displays a list of all the blocks in your model that support external mode signal monitoring and logging. The dialog box also lets you configure the signals that are viewed and how they are acquired and displayed. You can use it to reconfigure signals while the target program runs.

In this tutorial, you observe and use the default settings of the External Signal & Triggering dialog box.

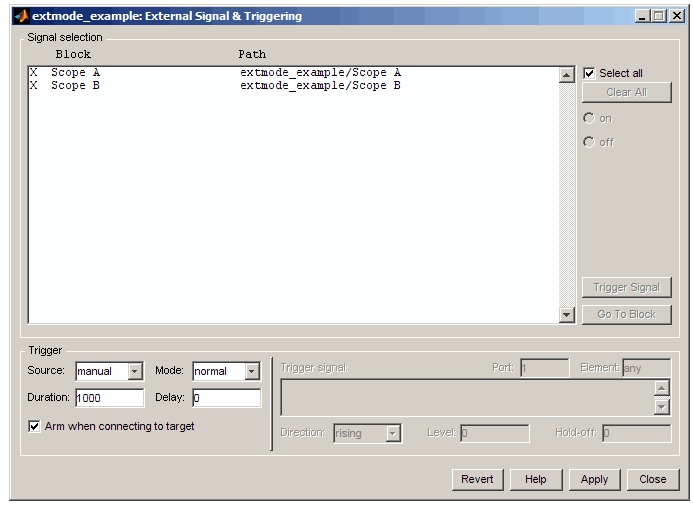
1 From the **Tools** menu of the block diagram, select **External Mode Control Panel**, which lets you configure signal monitoring and data archiving. It also lets you connect to the target program and start and stop execution of the model code.



The top three buttons are for use after the target program has started. The two lower buttons open separate dialog boxes:

- The **Signal & triggering** button opens the **External Signal & Triggering** dialog box. This dialog box lets you select the signals that are collected from the target system and viewed in external mode. It also lets you select a signal that triggers uploading of data when certain signal conditions are met, and define the triggering conditions.
- The **Data archiving** button opens the **External Data Archiving** dialog box. Data archiving lets you save data sets generated by the target program for future analysis. This example does not use data archiving. See “Data Archiving” in the Real-Time Workshop documentation for more information.

2 In the **External Mode Control Panel**, click the **Signal & Triggering** button. The **External Signal & Triggering** dialog box opens. The default configuration of the **External Signal & Triggering** dialog box is designed to ensure that all signals are selected for monitoring. The default configuration also ensures that signal monitoring will begin as soon as the host and target programs have connected. The figure below shows the default configuration for `extmode_example`.



3 Make sure that the **External Signal & Triggering** dialog box is set to the defaults as shown:

- **Select all** check box is selected. All signals in the **Signal selection** list are marked with an `X` in the **Block** column.
- **Trigger Source**: `manual`
- **Trigger Mode**: `normal`
- **Duration**: `1000`
- **Delay**: `0`
- **Arm when connecting to target**: selected

Click **Close**, and then close the **External Mode Control Panel**.

For information on the options mentioned above, see "External Signal Uploading and Triggering" in the Real-Time Workshop documentation.

4 To run the target program, you must open a command prompt window (on UNIX systems, an `Xterm` window). At the command prompt, change to the `ext_mode_example` directory that you created in step 1. The target program is in this directory.

```
cd ext_mode_example
```

Next, type the following command:

```
extmode_example -tf inf -w
```

and press **Return**.

Note On Microsoft Windows platforms, you can also use the "bang" command (!) in the MATLAB Command Window (note that the trailing ampersand is required): `!extmode_example -tf inf -w &`

The target program begins execution. Note that the target program is in a wait state, so no activity occurs in the MATLAB Command Window.

The `-tf` switch overrides the stop time set for the model in Simulink. The `inf` value directs the model to run indefinitely. The model code runs until the target program receives a stop message from Simulink.

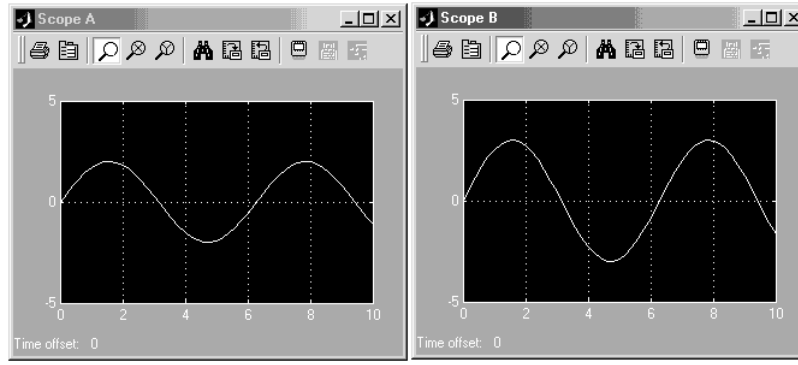
The `-w` switch instructs the target program to enter a wait state until it receives a **Start real-time code** message from the host. This switch is required if you want to view data from time step 0 of the target program execution, or if you want to modify parameters before the target program begins execution of model code.

5 Open Scope blocks A and B. At this point, no signals are visible on the scopes. When you connect Simulink to the target program and begin model execution, the signals generated by the target program will be visible on the scope displays.

6 The model itself must be in external mode before communication between the model and the target program can begin. To enable external mode, select **External** from the simulation mode pull-down menu located on the right side of the toolbar of the Simulink window. Alternatively, you can select **External** from the **Simulation** menu.

7 Reopen the **External Mode Control Panel** (found in the **Tools** menu) and click **Connect**. This initiates a handshake between Simulink and the target program. When Simulink and the target are connected, the **Start Real-Time Code** button becomes enabled, and the label of the **Connect** button changes to **Disconnect**.

8 Click the **Start Real-Time Code** button. The outputs of Gain blocks A and B are displayed on the two scopes in your model. With `A = 2` and `B = 3`, the output looks like this:



Having established communication between Simulink and the running target program, you can tune block parameters in Simulink and observe the effects the parameter changes have on the target program. You do this in the next section.

Tuning Parameters

You can change the gain factor of either Gain block by assigning a new value to the variable `A` or `B` in the MATLAB workspace. When you change block parameter values in the workspace during a simulation, you must explicitly update the block diagram with these changes. When the block diagram is updated, the new values are downloaded to the target program.

To tune the variables `A` and `B`,

1 In the MATLAB Command Window, assign new values to both variables, for example:

```
A = 0.5;B = 3.5;
```

2 Activate the `extmode_example` model window. Select **Update Diagram** from the **Edit** menu, or press **Ctrl+D**. As soon as Simulink has updated the block parameters, the new gain values are downloaded to the target program, and the effect of the gain change becomes visible on the scopes.

3 You can also enter gain values directly into the Gain blocks. To do this, open the Block Parameters dialog box for Gain block A or B in the model. Enter a new numerical value for the gain and click **Apply**. As soon as you click **Apply**, the new value is downloaded to the target program and the effect of the gain change becomes visible on the scope. Similarly, you can change the frequency, amplitude, or phase of the sine wave signal by opening the Block Parameters dialog box for the Sine Wave block and entering a new numerical value in the appropriate field.

   Note that because the Sine Wave is a source block, Simulink pauses while the Block Parameters dialog box is open. You must close the dialog box by clicking **OK**, which allows Simulink to continue and enable you to see the effect of your changes.

   Also note that you cannot change the sample time of the Sine Wave block. Block sample times are part of the structural definition of the model and are part of the generated code. Therefore, if you want to change a block sample time, you must stop the external mode simulation, reset the block's sample time, and rebuild the executable.

4 To simultaneously disconnect host/target communication and end execution of the target program, pull down the **Simulation** menu and select **Stop Real-Time Code**. You can also do this from the **External Mode Control Panel**.

Generating Code for a Referenced Model

In this section...

Tutorial Overview

The Model block allows an existing Simulink model to be used as a block in another model. When a model contains one or more Model blocks, it is called a *parent model*. Models represented by Model blocks are called *referenced models* in that context.

Model blocks are particularly useful for large-scale modeling applications. They work by generating code and creating a binary file for each referenced model, then executing the binary during simulation. The Real-Time Workshop software generates code for referenced models in a slightly different way than for top models and stand-alone models, and generates different code than Simulink does when it simulates them. Follow this tutorial to learn how Simulink and the Real-Time Workshop software handle Model blocks.

In this tutorial, you create a subsystem in an existing model, convert it to a referenced model, call it from the top model via a Model block, and generate code for both models. You accomplish some of these tasks automatically with a function called `Simulink.Subsystem.convertToModelReference`. You also explore the generated code and the project directory using the Model Explorer.

Creating and Configuring a Subsystem Within the vdp Model

In the first part of this tutorial, you define a subsystem for the `vdp` demo model, set configuration parameters for the model, and use the `Simulink.Subsystem.convertToModelReference` function to convert it into

two new models — the top model (`vdptop`) and a referenced model `vdpmultRM` containing a subsystem you created (`vdpmult`):

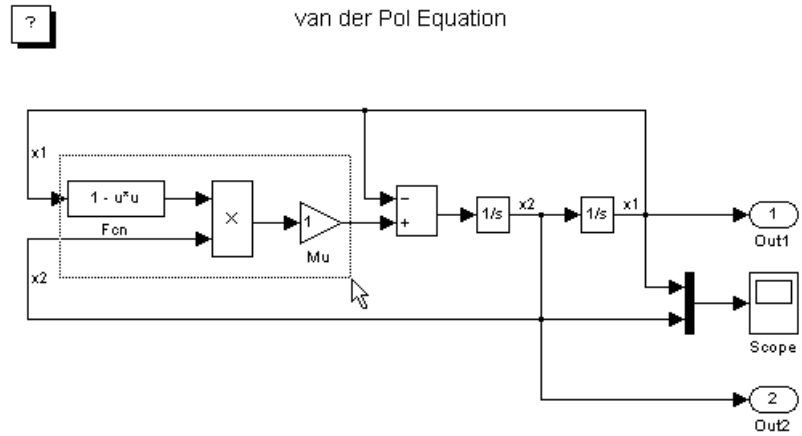
1. In the MATLAB Command Window, create a new working directory wherever you want to work and `cd` into it:

   ```
   mkdir tutorial6
   cd tutorial6
   ```

2. Open the `vdp` demo model by typing:

   ```
   vdp
   ```

3. Drag a box around the three blocks on the left to select them, as shown below:



4. Choose **Create Subsystem** from the model's **Edit** menu.

   A subsystem block replaces the selected blocks.

5. If the new subsystem block is not where you want it, move it to a preferred location.

6. Rename the block `vdpmult`.

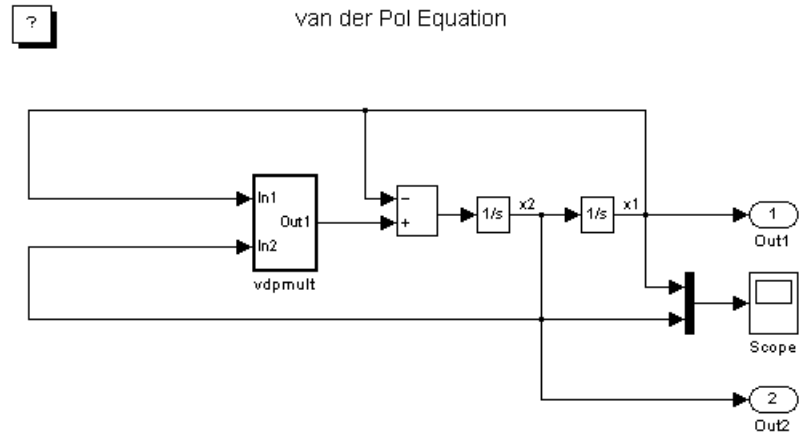
7 Right-click the `vdpmult` block and select **Subsystem Parameters**.

The **Function Block Parameters** dialog box appears.

8 In the **Function Block Parameters** dialog box, select **Treat as atomic unit**, then click **OK**.

The border of the `vdpmult` subsystem thickens to indicate that it is now atomic. An atomic subsystem executes as a unit relative to the parent model: subsystem block execution does not interleave with parent block execution. This property makes it possible to extract subsystems for use as stand-alone models and as functions in generated code.

The block diagram should now appear as follows:



You must set several properties before you can extract a subsystem for use as a referenced model. To set the necessary properties,

1 Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.

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

3 Click **Configuration (Active)** in the left pane.

4. In the center pane, select **Solver**.

5. In the right pane, under **Solver Options** change the **Type** to `Fixed-step`, then click **Apply**. You must use fixed-step solvers when generating code, although referenced models can use different solvers than top models.

6. In the center pane, select **Optimization**. In the right pane, under **Simulation and code generation**, select **Inline parameters**. Click **Apply**.

7. In the center pane, select **Diagnostics**. In the right pane:

   a. Select the **Data Validity** tab. In the **Signals** area, set **Signal resolution** to `Explicit only`.

   b. Select the **Connectivity** tab. In the **Buses** area, set **Mux blocks used to create bus signals** to `error`.

8. Click **Apply**.

   The model now has the properties that model referencing requires.

9. In the center pane, click **Model Referencing**. In the right pane, set **Rebuild options** to `If any changes in known dependencies detected`. Click **Apply**. This setting prevents unnecessary code regeneration.

10. In the `vdp` model window, choose **File > Save as**. Save the model as `vdptop` in your working directory. Leave the model open.

Converting the Model to Use Model Referencing

In this portion of the tutorial, you use the conversion function `Simulink.SubSystem.convertToModelReference` to extract the subsystem `vdpmult` from `vdptop` and convert `vdpmult` into a referenced model named `vdpmultRM`. To see the complete syntax of the conversion function, type at the MATLAB prompt:

```
help Simulink.SubSystem.convertToModelReference
```

For additional information, type:

```
doc Simulink.SubSystem.convertToModelReference
```

If you want to see a demo of `Simulink.SubSystem.convertToModelReference` before using it yourself, type:

```
sldemo_mdlref_conversion
```

Simulink also provides a menu command, **Convert to Model Block**, that you can use to convert a subsystem to a referenced model. The command calls `Simulink.SubSystem.convertToModelReference` with default arguments. See "Converting a Subsystem to a Referenced Model" in the Simulink documentation.

Extracting the Subsystem to a Referenced Model

To use `Simulink.SubSystem.convertToModelReference` to extract `vdpmult` and convert it to a referenced model, type:

```
Simulink.SubSystem.convertToModelReference...
('vdptop/vdpmult', 'vdpmultRM',...
'ReplaceSubsystem', true, 'BuildTarget', 'Sim')
```

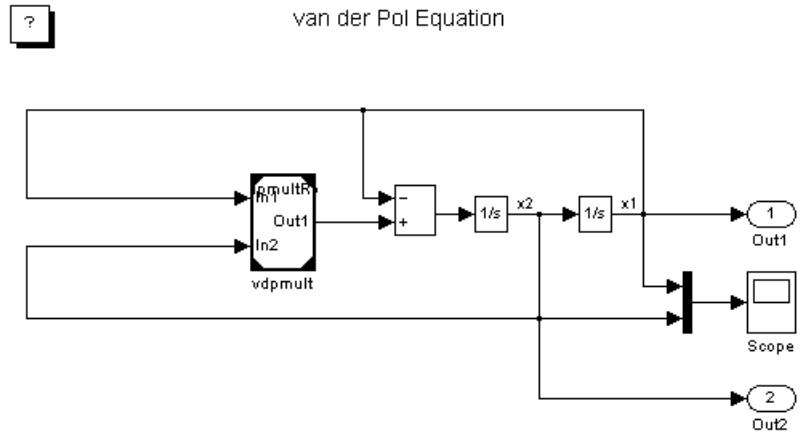
This command:

1. Extracts the subsystem `vdpmult` from `vdptop`.
2. Converts the extracted subsystem to a separate model named `vdpmultRM` and saves the model to the working directory.
3. In `vdptop`, replaces the extracted subsystem with a Model block that references `vdpmultRM`.
4. Creates a simulation target for `vdptop` and `vdpmultRM`.

The converter prints a number of progress messages, and when successful, terminates with

```
ans =
     1
```

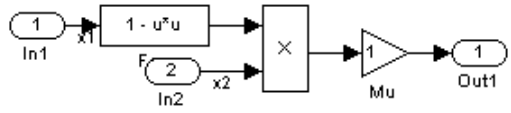
The parent model `vdptop` now looks like this:



Note the changes in the appearance of the block `vdpmult`. These changes indicate that it is now a Model block rather than a subsystem. As a Model block, it has no contents of its own: the previous contents now exist in the referenced model `vdpmultRM`, whose name appears at the top of the Model block. Widen the Model block as needed to expose the complete name of the referenced model.

If the parent model `vdptop` had been closed at the time of conversion, the converter would have opened it. Extracting a subsystem to a referenced model does *not* automatically create or change a saved copy of the parent model. To preserve the changes to the parent model, save `vdptop`.

Right-click the Model block `vdpmultRM` and choose **Open Model 'vdpmultRM'** to open the referenced model. The model looks like this:



Files Created and Changed by the Converter

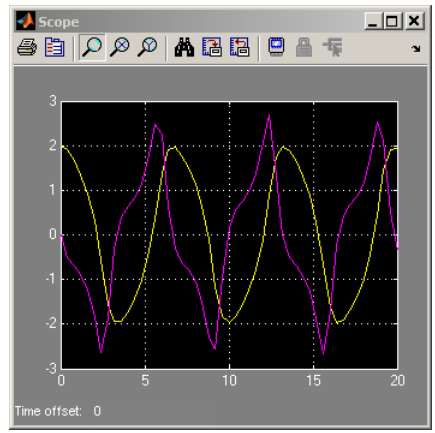
The files in your working directory now consist of the following (not in this order).

File	Description
`vdptop.mdl`	Top-level model that contains a Model block where the `vdpmult` subsystem was
`vdpmultRM.mdl`	Referenced model created for the `vdpmult` subsystem
`vdpmultRM_msf.mexw32`	Static library file (Microsoft Windows platforms only). The last three characters of the suffix are system-dependent and may differ. This file executes when the `vdptop` model calls the model block `vdpmult`. When called, `vdpmult` in turn calls the referenced model `vdpmultRM`.
`/slprj`	Project directory for generated model reference code

Code for model reference simulation targets is placed in the `slprj/sim` subdirectory. Generated code for GRT, ERT, and other Real-Time Workshop targets is placed in `slprj` subdirectories named for those targets. You will inspect some model reference code later in this tutorial. For more information on project directories, see “Working with Project Directories” on page 3-71.

Running the Converted Model

Open the Scope block in `vdptop` if it is not visible. In the `vdptop` window, click the **Start** tool or choose **Start** from the **Simulation** menu. The model calls the `vdpmultRM_msf` simulation target to simulate. The output looks like this:

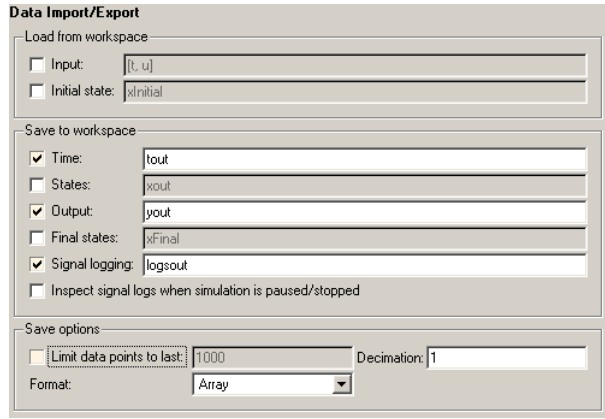


Generating Model Reference Code for a GRT Target

The function `Simulink.SubSystem.convertToModelReference` created the model and the simulation target files for the referenced model `vdpmultRM`. In this part of the tutorial, you generate code for that model and the `vdptop` model, and run the executable you create:

1. Verify that you are still working in the `tutorial6` directory.
2. If the model `vdptop` is not open, open it. Make sure it is the active window.
3. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
4. In the **Model Hierarchy** pane, click the + sign preceding the `vdptop` model to reveal its components.
5. Click **Configuration (Active)** in the left pane.
6. In the center pane, select **Data Import/Export**.

7 Check **Time** and **Output** in the **Save to workspace** section of the right pane, then click **Apply**. The pane shows the following information:



These settings instruct the model `vdptop` (and later its executable) to log time and output data to MAT-files for each time step.

8 Generate GRT code (the default) and an executable for the top model and the referenced model by selecting **Real-Time Workshop** in the center pane and then clicking the **Build** button.

The Real-Time Workshop build process generates and compiles code. The current directory now contains a new file and a new directory:

File	Description
`vdptop.exe`	The executable created by the Real-Time Workshop build process
`vdptop_grt_rtw/`	The Real-Time Workshop build directory, containing generated code for the top model

The Real-Time Workshop build process also generated GRT code for the referenced model, and placed it in the `slprj` directory.

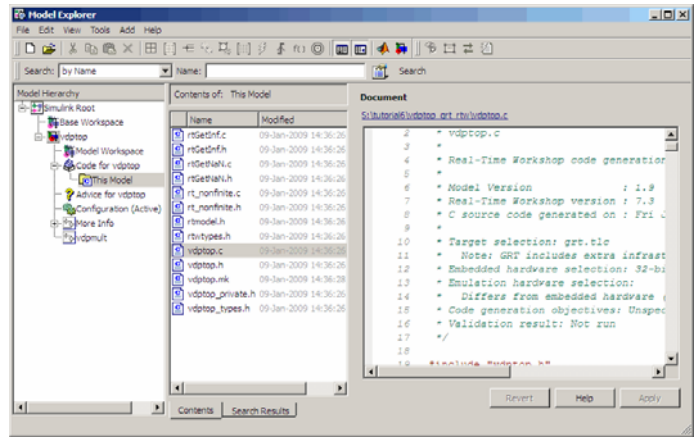
To view a model's generated code in **Model Explorer**, the model must be open. To use the **Model Explorer** to inspect the newly created build directory, `vdptop_grt_rtw`:

1. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
2. In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.
3. Click the + sign preceding `Code for vdptop` to reveal its components.
4. Click `This Model` that appears directly under `Code for vdptop`.

   A list of generated code files for `vdptop_converted` appears in the **Contents** pane:

   ```
   rtmodel.h
   vdptop.c
   vdptop.h
   vdptop.mk
   vdptop_private.h
   vdptop_types.h
   ```

You can browse code in any of these files by selecting a file of interest in the **Contents** pane. The code for the file you select appears in the pane to the right. The figure below illustrates viewing code for `vdptop.c`. Your code may differ.



To open a file in a text editor, click a filename, and then click the hyperlink that appears in the gray area at the top of the **Document** pane.

Working with Project Directories

When you view generated code in **Model Explorer**, the files listed in the **Contents** pane can exist either in a build directory or a project directory. Model reference project directories (always rooted under `slprj`), like build directories, are created in your current working directory, and this implies certain constraints on when and where model reference targets are built, and how they are accessed.

The models referenced by Model blocks can be stored anywhere. A given top model can include models stored on different file systems and directories. The same is not true for the simulation targets derived from these models; under most circumstances, all models referenced by a given top model must be set up to simulate and generate model reference target code in a single project directory. The top and referenced models can exist anywhere on your path, but the project directory is assumed to exist in your current directory.

This means that, if you reference the same model from several top models, each stored in a different directory, you must either

- Always work in the same directory and be sure that the models are on your path
- Allow separate project directories, simulation targets, and Real-Time Workshop targets to be generated in each directory in which you work

The files in such multiple project directories are generally quite redundant. Therefore, to avoid regenerating code for referenced models more times than necessary, you might want to choose a specific working directory and remain in it for all sessions.

As model reference code generated for Real-Time Workshop targets as well as for simulation targets is placed in project directories, the same considerations as above apply even if you are generating target applications only. That is, code for all models referenced from a given model ends up being generated in the same project directory, even if it is generated for different targets and at different times.

Documenting a Code Generation Project

In this section...

Tutorial Overview

As explained in "Documenting the Project" on page 2-16, one way of documenting a Real-Time Workshop code generation project is to use the Simulink Report Generator software. In this tutorial, you adjust the Simulink Report Generator settings to include custom code and then generate a report for the Real-Time Workshop demo `rtwdemo_f14`. A summary of the steps for the tutorial follows:

1 Generate code for the model.

2 Open Report Generator.

3 Set report output options.

4 Specify models and subsystems to be included.

5 Set component options.

6 Generate the report.

7 Review the generated report.

Note You need a Simulink Report Generator license to complete steps 3 through 5. If you omit those steps and use the default option settings, the resulting output will vary from what is documented in step 6.

For details on using Report Generator, see the *Simulink Report Generator User's Guide*.

Generating Code for the Model

Before you can use Report Generator to document your project, you must generate code for the model. To generate code for the `rtwdemo_f14` demo,

1. In the MATLAB Current Directory browser, navigate to a directory where you have write access.
2. Create a working directory from the MATLAB command line by typing:

   ```
   mkdir report_ex
   ```

3. Make `report_ex` your working directory:

   ```
   cd report_ex
   ```

4. Open the `rtwdemo_f14` model by clicking the model name below or by entering the model name on the MATLAB command line.

   ```
   rtwdemo_f14
   ```

   The model appears in a Simulink model window.

5. In the model window, choose **File > Save As**, navigate to the working directory, `report_ex`, and save a copy of the `rtwdemo_f14` model as `myf14.mdl`.
6. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
7. In the **Model Hierarchy** pane, click the + sign preceding the model name to reveal its components.

8 Click **Configuration (Active)** in the left pane.

9 In the **Contents** pane, click **Real-Time Workshop**. The **Real-Time Workshop** pane appears.

10 Select the **Report** tab. Clear the **Create code generation report** and **Launch report automatically** check boxes.

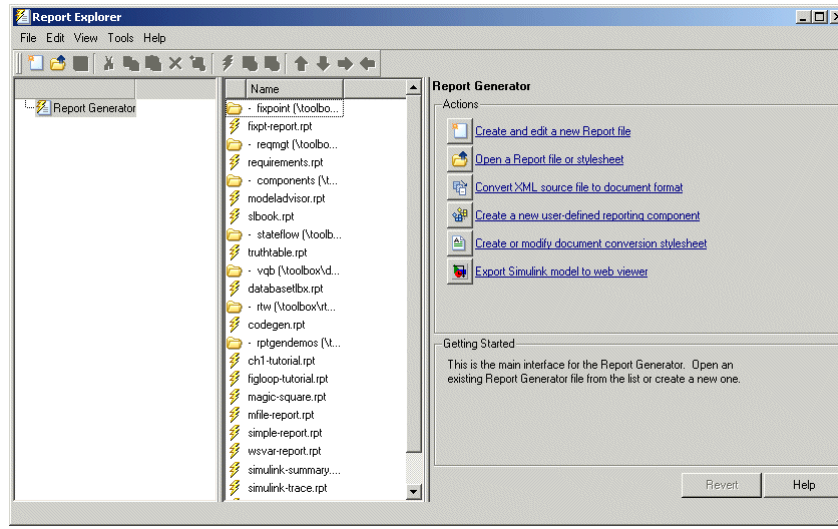
11 Select the **General** tab. Select **Generate code only** and click **Apply**.

12 Click **Generate code**. The Real-Time Workshop build process generates code for the model.

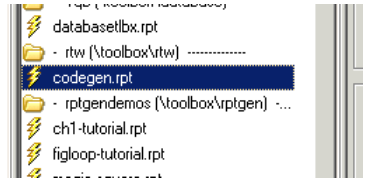
Opening Report Generator

After you generate the code, open the Report Generator.

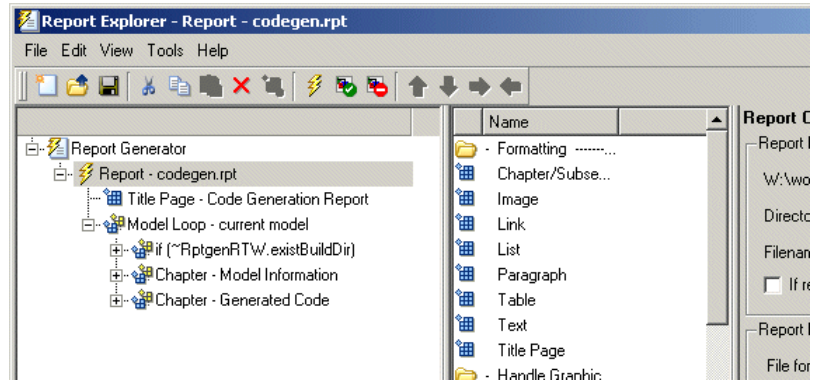
1 In the model window, select **Tools > Report Generator**. The Report Explorer window opens.



2 In the options pane (center), find the folder **rtw (\toolbox\rtw)** and the setup file that it contains — **codegen.rpt**.



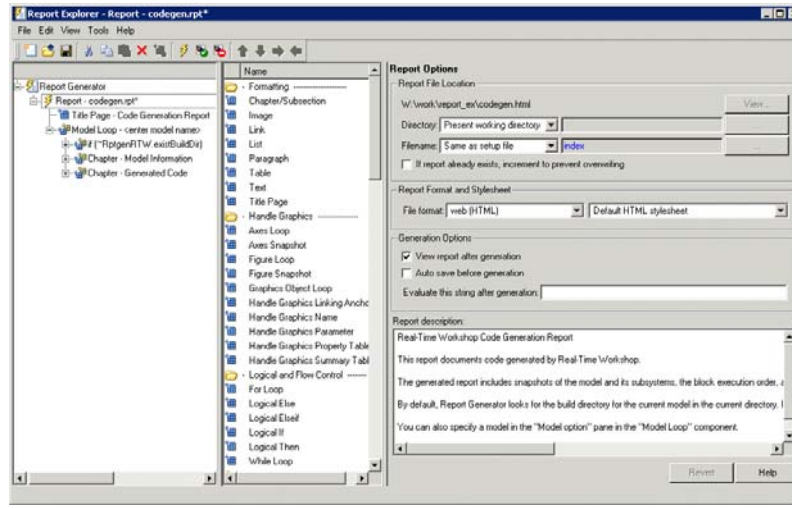
3 Double-click **codegen.rpt** or select it and click the **Open report** button  . Report Generator displays the structure of the setup file in the outline pane (left).



Setting Report Output Options

Before generating a report, you can specify report output options, such as the directory, file name, and format. The following steps explain how to generate a Microsoft Word report named `MyCGModelReport.rtf`.

1 Review the options listed under Report Options in the properties pane.

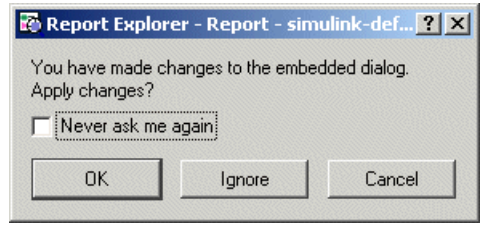


2 Leave the **Directory** field set to `Present working directory`.

3 Select `Custom:` for **Filename** and replace `index` with the name `MyModelCGReport`.

4 For **File format**, specify `Rich Text Format` and replace `Standard Print` with `Numbered Chapters & Sections`.

5 In the outline pane (left), click **Report - codegen.rpt***. The following acknowledgment dialog box appears.



6 Click **OK**.

Specifying Models and Subsystems to Include in a Report

Specify the models and subsystems to be included in the generated report by setting options in the Model Loop component.

1. In the outline pane (left), select **Model Loop**. Report Generator displays Model Loop component options in the properties pane.
2. If not already selected, select `Current block diagram` for the **Model name** option.
3. In the outline pane, click **Report - codegen.rpt***. If you modified the value of the **Model name** option, the change acknowledgment dialog box appears. If the dialog box appears, click **OK**.

Setting Component Options

After setting the report output options, review and, if appropriate, adjust Real-Time Workshop component options.

1. In the outline pane (left), expand the node **Chapter - Generated Code**. By default, the report includes two sections, each containing one of two Real-Time Workshop report components.
2. Expand the node **Section 1 — Code Generation Summary**. The **Code Generation Summary** component appears.
3. Select **Code Generation Summary**. Options for the component appear in the properties pane.
4. Click **Help** to review the report customizations you can make with the Code Generation Summary component. For this tutorial, do not customize the component.
5. Return focus to the Report Explorer window and expand the node **Section 1 — Generated Code Listing**. The **Import Generated Code** component appears.
6. Select **Import Generated Code**. Options for the component appear in the properties pane.

7 Click **Help** to review the report customizations you can make with the Import Generated Code component.

8 Return focus to the Report Explorer window.

Generating the Report

After you adjust report options, from the **Report Explorer** window, generate the report by clicking **File > Report**. A **Message List** dialog box appears, which displays messages you can monitor as the report is generated. Model snapshots also appear during report generation.

For alternative ways of generating reports, see "Generating Reports" in the Simulink Report Generator documentation.

Reviewing the Generated Report

Review your generated report. Make sure the following information is included:

- System snapshots (model and subsystem diagrams)
- Block execution order list
- Real-Time Workshop and model version information for generated code
- List of generated files
- Optimization configuration parameter settings
- Real-Time Workshop target selection and build process configuration parameter settings
- Subsystem map
- File name, path, and generated code listings for the following:
  - `myf14.c`
  - `rt_nonfinite.c`
  - `myf14.h`
  - `myf14_private.h`
  - `myf14_types.h`

- rt_nonfinite.h
- rtmodel.h
- rtwtypes.h

Index

I

M

O

P

R

S

T

V