Real-Time Workshop®

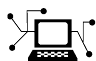
For Use with Simulink®

- Modeling
- Simulation
- Implementation

Getting Started

Version 6

How to Contact The MathWorks:



`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	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. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

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

Getting Started with Real-Time Workshop

© COPYRIGHT 2002–2005 by The MathWorks, Inc.

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

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

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Revision History:

July 2002	First printing	New for Version 5 (Release 13)
June 2004	Second printing	Updated for Version 6.0 (Release 14)
October 2004	Third printing	Updated for Version 6.1 (Release 14SP1)
March 2005	Online only	Updated for Version 6.2 (Release 14SP2)

Contents

1 Introduction

What Is Real-Time Workshop? **1-2**
Components and Features **1-3**
Accelerating Your Development Process **1-7**

Installing Real-Time Workshop **1-12**
Third-Party Compiler Installation on Windows **1-13**
Supported Compilers **1-15**
Compiler Optimization Settings **1-16**

Real-Time Workshop Demos **1-17**

MathWorks Products Supported by Real-Time Workshop **1-18**

Help and Documentation **1-21**
Online Documentation **1-21**
Related Products **1-22**
For Further Information **1-22**

2 Building an Application

Real-Time Workshop Workflow **2-2**
Mapping Application Requirements to Configuration Options **2-4**
Setting Configuration Options **2-9**
Running the Model Advisor **2-10**
Generating Code **2-12**
Building an Executable Program Image **2-12**
Verifying the Generated Results **2-13**
Saving the Model Configuration **2-13**

Automatic Program Building **2-16**

The Build Process **2-18**
Model Compilation **2-20**
Code Generation **2-20**
Customized Makefile Generation **2-21**
Executable Program Generation **2-22**
Files and Directories Created by the Build Process **2-24**

3 Working with Real-Time Workshop

The f14 Demonstration Model **3-3**

Building a Generic Real-Time Program **3-4**
Working and Build Directories **3-4**
Setting Program Parameters **3-5**
Selecting the Target Configuration **3-7**
Building and Running the Program **3-12**
Contents of the Build Directory **3-14**

Data Logging **3-15**
Data Logging During Simulation **3-15**
Data Logging from Generated Code **3-18**

Code Verification **3-21**
Logging Signals via Scope Blocks **3-21**
Logging Simulation Data **3-22**
Logging Data from the Generated Program **3-23**
Comparing Numerical Results of the Simulation and the Generated Program **3-25**

A First Look at Generated Code **3-27**
Setting Up the Model **3-27**
Generating Code Without Buffer Optimization **3-28**
Generating Code with Buffer Optimization **3-31**
Further Optimization: Expression Folding **3-33**
HTML Code Generation Reports **3-35**

Working with External Mode Using GRT **3-38**
Setting Up the Model **3-38**
Building the Target Executable **3-40**
Running the External Mode Target Program **3-43**
Tuning Parameters **3-48**

Generating Code for a Referenced Model **3-50**
Create and Configure A Subsystem Within the vdp Model **3-50**
Convert the Model to Use Model Reference **3-54**
Generate Model Reference Code for a GRT Target **3-56**
Working with Project Directories **3-62**

Glossary

A Examples

Verification **A-2**

Optimizations **A-3**

External Mode **A-4**

Model Reference **A-5**

Index

1

Introduction

This guide begins with a high-level overview of Real-Time Workshop®, describing its purpose, its component parts, its major features, and the ways in which it leverages the modeling power of Simulink® for developing real-time applications on a variety of platforms. You will also find here helpful information about installing Real-Time Workshop, including discussions of related products from The MathWorks and compilers from third parties, as well as pointers to demos and online and printable documentation. The chapter is laid out as follows:

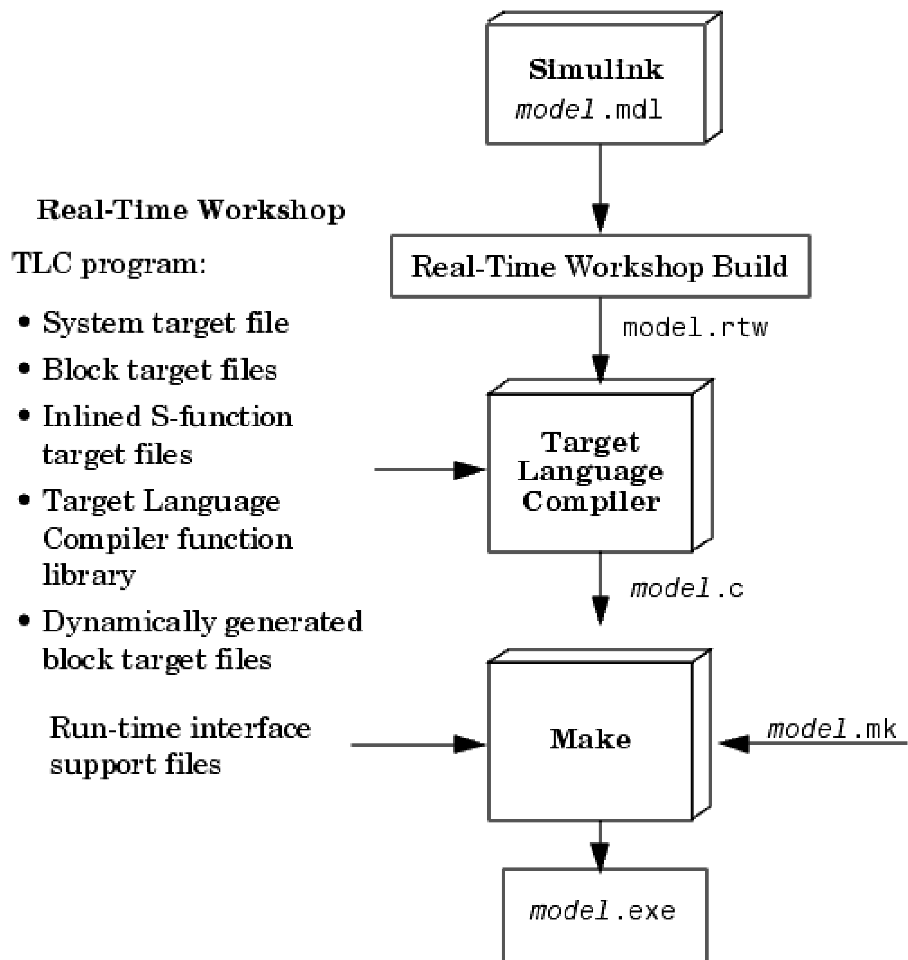
"What Is Real-Time Workshop?" (p. 1-2)	What it is, and what it can do for you
"Installing Real-Time Workshop" (p. 1-12)	Information on supported compilers
"Real-Time Workshop Demos" (p. 1-17)	Demonstrations you can summon that illustrate code generation capabilities
"MathWorks Products Supported by Real-Time Workshop" (p. 1-18)	Products for which code can be generated or which otherwise work with Real-Time Workshop
"Help and Documentation" (p. 1-21)	Locating and using online and printed help documents

What Is Real-Time Workshop?

Real-Time Workshop® is an extension of capabilities of Simulink® and MATLAB® that automatically generates, packages, and compiles source code from Simulink models to create real-time software applications on a variety of systems. By providing a code generation environment for rapid prototyping and deployment, Real-Time Workshop is the foundation for production code generation capabilities. Along with other tools and components from The MathWorks, Real-Time Workshop provides

- Automatic code generation tailored for a variety of target platforms
- A rapid and direct path from system design to implementation
- Seamless integration with MATLAB and Simulink
- A simple graphical user interface
- An open architecture and extensible make process

The process of generating source code from Simulink models using Real-Time Workshop is shown in the following diagram.



Real-Time Workshop Code Generation Process

Components and Features

The principal components and features of Real-Time Workshop are

- **Simulink Code Generator** — Automatically generates C code from your Simulink model.

- **Make process** — The Real-Time Workshop user-extensible make process lets you customize compilation and linking of generated code for your own production or rapid prototyping target.
- **Simulink external mode** — External mode enables communication between Simulink and a model executing on a real-time test environment, or in another process on the same machine. External mode lets you perform parameter tuning, data logging, and viewing using Simulink.
- **Targeting support** — Using the targets bundled with Real-Time Workshop, you can build systems for real-time and prototyping environments. The generic real-time and other bundled targets provide a framework for developing customized rapid prototyping or production target environments. In addition to the bundled targets, the optional Real-Time Windows Target and the xPC Target let you turn almost any PC into a rapid prototyping target or a small to medium volume production target. To supplement core capabilities, the optional Real-Time Workshop Embedded Coder and embedded target products extend and tailor Real-Time Workshop code to run in a growing suite of microprocessor environments.
- **Rapid simulations** — Using Simulink Accelerator, the S-Function Target, or the Rapid Simulation Target, you can accelerate your simulations by five to 20 times on average. Executables built with these targets bypass normal Simulink interpretive simulation mode. Code generated by Simulink Accelerator, S-Function Target, and Rapid Simulation Target is highly optimized to execute only the algorithms used in your specific model. In addition, the code generator applies many optimizations, such as eliminating ones and zeros in computations for filter blocks.
- **Large-scale modeling** — Support for multilevel modeling (termed *model referencing*) in Simulink is mirrored in Real-Time Workshop, which lets you generate code incrementally for a hierarchy of independent component models, as they evolve.

Features of Real-Time Workshop include

- **Code generation for Simulink models**
  - Generates optimized, customizable code. There are several styles of generated code, which can be classified as either embedded (production phase) or rapid prototyping.

- Supports all Simulink features, including 8-, 16-, and 32-bit integers and floating-point double and single data types.
- Allows fixed-point capabilities allow for scaling of integer words ranging from 2 to 128 bits. Code generation is limited by the implementation of `char`, `short`, `int`, and `long` in embedded C compiler environments (usually 8, 16, and 32 bits, respectively). You can choose hardware characteristics for more than 20 preconfigured target processors by name or create your own custom processor definition.
- Generates processor-independent code. The generated code represents your model exactly. A separate run-time interface is used to execute this code. Real-Time Workshop provides several example run-time interfaces, as well as production run-time interfaces.
- Supports single- or multitasking operating system environments, as well as bare-board (no operating system) environments.
- Provides the flexible scripting capabilities of the Target Language Compiler to enable you to fully customize generated code.
- Allows you to craft efficient code for S-functions (user-created blocks) using Target Language Compiler instructions (called TLC scripts) that automatically integrates with generated code.

- **Extensive model-based debugging support**
  - External mode enables you to examine what the generated code is doing by uploading data from your target to the graphical display elements in your model. There is no need to use a conventional source-level debugger to look at your generated code.
  - External mode also enables you to tune the generated code via your Simulink model. When you change a parameter value of a block in your model, the new value is passed down to the generated code running on your target, and the corresponding target memory location is updated. Again, there is no need to use an embedded compiler debugger to perform this type of operation. Your model is your debugger user interface.
- **Integration with Simulink**
  - Code verification. You can generate code for your model and create a standalone executable that exercises the generated code and produces a MAT-file containing the execution results.

- Generated code contains system and block identification tags to help you identify the block in your source model that generated a given line of code. The MATLAB command `hilite_system` recognizes these tags and highlights the corresponding blocks in your model.
- Support for Simulink data objects lets you define how your signals and block parameters interface to the external world.

- **Integration with Stateflow®**
  - Code generation. Seamless code generation support for models that contain Stateflow charts.
  - Full support for Stateflow Coder features.
- **Rapid simulation**
  - Real-Time Workshop supports several ways to speed up your simulations by creating optimized, model-specific executables.
- **Target support**
  - Turnkey solutions for rapid prototyping substantially reduce design cycles, allowing for fast turnaround of design iterations.
  - Bundled rapid prototyping example targets provide working code you can modify and use quickly. For a complete list of bundled targets, with their associated system target files and template makefiles, see "Selecting a Target Configuration"in the Real-Time Workshop documentation.
  - The optional Real-Time Windows and xPC targets for PC-based hardware from The MathWorks enable you to turn a PC with fast, high-quality, low-cost hardware into a rapid prototyping system.
  - Real-Time Workshop Embedded Coder, an add-on product, provides extensive support for tailoring generated code to special requirements of embedded hardware, software environments, data formats, and tool chain protocols.
  - The MathWorks offers a growing set of code generation products for embedded processors, such as the Embedded Target for Infineon C166™® Microcontrollers, the Embedded Target for Motorola® HC12, and the Embedded Target for Motorola® MPC555, which via turnkey hardware support extend the benefits of Real-Time Workshop and Real-Time Workshop Embedded Coder into production environments.

  - Supports a variety of third-party hardware and tools, with extensible device driver support.

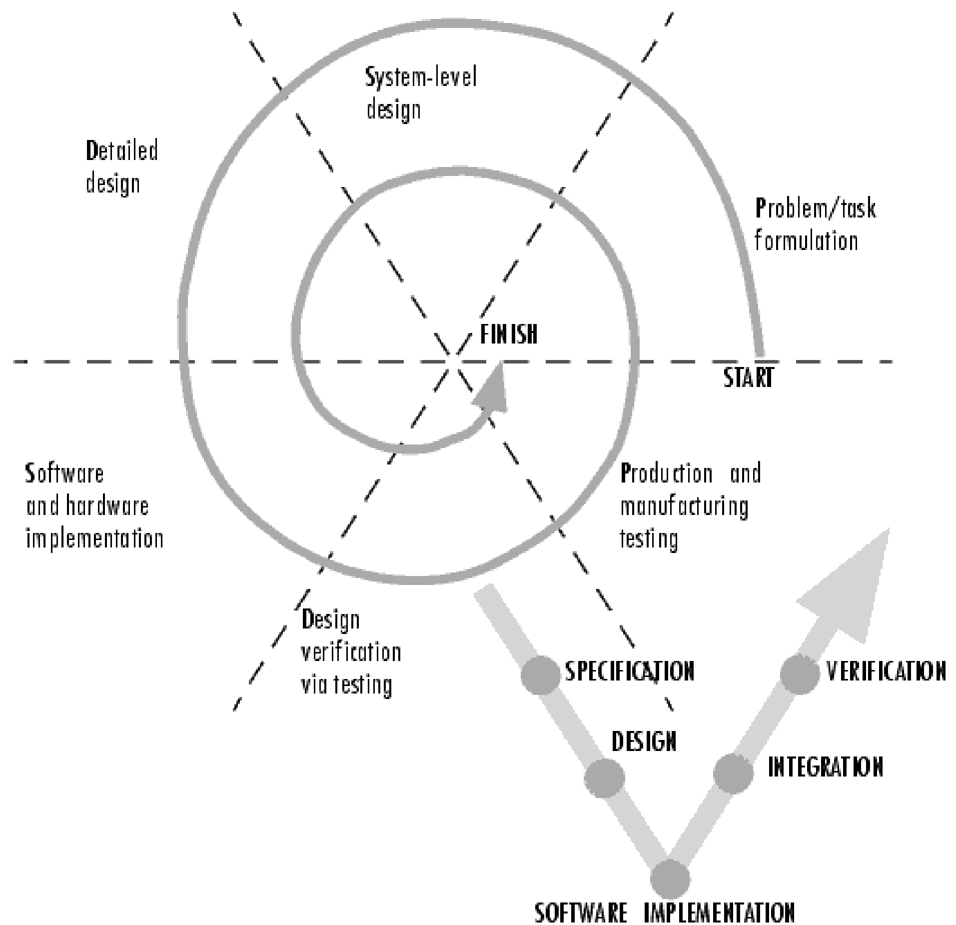
- **Extensible make process**
  - Allows for easy integration with any embedded compiler and linker.
  - Provides for easy linkage with your hand-written supervisory or supporting code.

The Real-Time Workshop Embedded Coder, mentioned above,

- Generates ANSI/ISO C code and executables for discrete, continuous, or hybrid models
- Uses model blocks to incrementally generate and build code for large applications
- Supports Simulink data dictionary features for integer, floating-point, and fixed-point data
- Generates code for single-rate, multirate, and asynchronous models
- Supports single-tasking and multitasking operating systems and bare-board (no operating system) environments
- Performs code optimizations that improve code execution speed
- Provides capabilities for code customization and legacy code integration
- Lets you interactively tune and monitor the generated code inside or outside of Simulink

Accelerating Your Development Process

The MathWorks gives you the ability to simplify and accelerate most phases of software development, and at the same time to eliminate repetitive and error-prone tasks, including some design document preparation. These tools lend themselves particularly well to the spiral design process shown below.



Spiral Design Process

When you work with tools from The MathWorks, *your model represents your understanding of your system*. This understanding is preserved from one phase of modeling to the next, reducing the need to backtrack. In the event that rework is necessary in a previous phase, it is easier to step back because the same model and tools are used throughout.

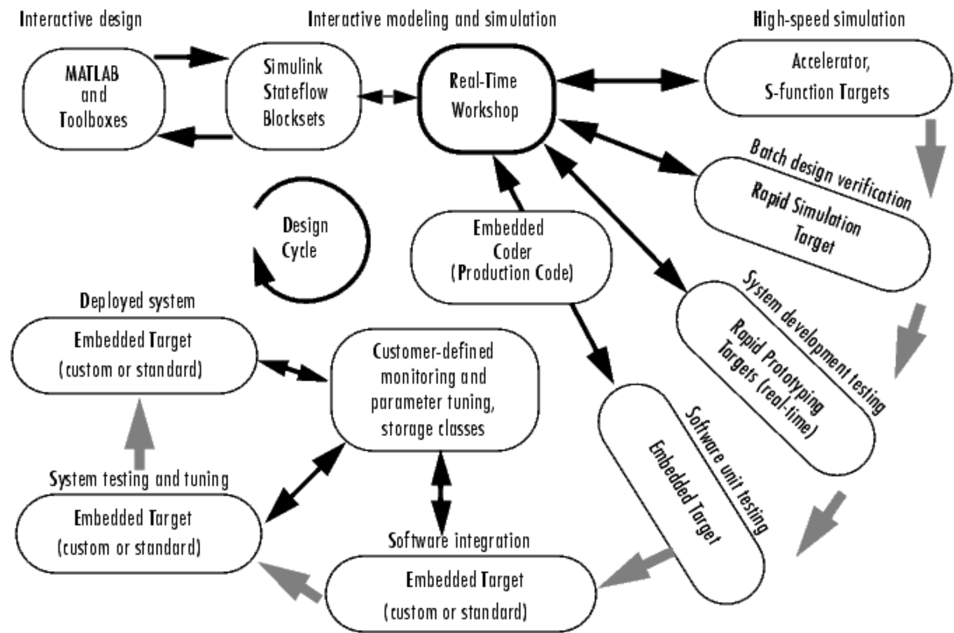
A spiral design process allows quick iterations between phases, enabling you to focus on design work. To do this cost-effectively, you need to use tools that

make it easy to move from one phase to another. For example, in a matter of minutes a control system engineer or a signal processing engineer can verify that an algorithm works on a real-world rapid prototyping system. The spiral process lends itself naturally to parallelism in the overall development process. You can provide early working models to validation and production groups, involving them in your system development process from the start. Once unit models are prototyped, tested, and ready, they can be packaged into larger assemblies using Model blocks (the Simulink *model referencing* facility). This helps to compress overall development time while increasing quality.

Simulink facilitates the first three phases described in the above figure. You can build applications using built-in blocks from the Simulink and Stateflow libraries, incorporate specialized blocks from the Aerospace, Communications, Signal Processing, and other MathWorks blocksets, and develop your own blocks by writing S-functions.

Real-Time Workshop (optionally extended by Real-Time Workshop Embedded Coder, Real-Time Windows Target, and xPC Target) completes the spiral process. It closes the rapid prototyping loop by generating and optimizing code for given tasks and production environments.

The figure below illustrates where products from The MathWorks, including Real-Time Workshop, help you in your development process.



MathWorks Products in Software Development

Early in the design process, you use MATLAB and Simulink to help you formulate your objectives, problems, and constraints to create your initial design. Real-Time Workshop speeds up models by enabling high-speed simulations via Simulink Accelerator and by model referencing, which includes models in other models as blocks.

After you have a functional model, you might need to tune your model's parameters. You can do this quickly using the Real-Time Workshop Rapid Simulation Target for Monte Carlo and other batch-oriented simulations (varying coefficients over many simulations).

Once you have tuned your model, you can move into system development testing by exercising your model on a rapid prototyping system, such as Real-Time Windows Target or xPC Target. With a rapid prototyping target, you connect your model to your physical system. This lets you locate design flaws and modeling errors quickly.

After you create your prototype system and verify its outputs, you can use Real-Time Workshop Embedded Coder to deploy generated code on your custom target. The signal monitoring and parameter tuning capabilities enable you to easily integrate the embedded code into a production environment equipped with debugging and upgrade capabilities. See the Real-Time Workshop Embedded Coder documentation for an overview of this process. First, however, see "Real-Time Workshop Workflow" on page 2-2 for a process view of using Real-Time Workshop.

Installing Real-Time Workshop

Your platform-specific MATLAB installation documentation provides all the information you need to install Real-Time Workshop.

Prior to installing Real-Time Workshop, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password (PLP) identifies the products you are permitted to install and use.

If you choose to customize your installation, as the process proceeds, it displays a dialog box, letting you indicate which products to install. Note that, in the installer product window, you can only select for installation MATLAB products for which you are licensed.

Simulink, Real-Time Workshop, and Real-Time Workshop Embedded Coder each have product prerequisites for proper installation and execution described in the following table.

Licensed Product	Prerequisite Products	Additional Information
Simulink	MATLAB 7 (Release 14)	Allows installation of Simulink
Real-Time Workshop	Simulink 6 (Release 14)	Requires Borland C, LCC, Visual C/C++, or Watcom C compiler to create MATLAB MEX-files or other executables on your platform
Real-Time Workshop Embedded Coder	Real-Time Workshop 6	

If you experience installation difficulties and have Web access, connect to the MathWorks home page (`http://www.mathworks.com`). Use the resources on the Installation, License Changes, and Passwords page at `http://www.mathworks.com/support/install/` to help you through the process.

Third-Party Compiler Installation on Windows

Most Real-Time Workshop targets create an executable that runs on your workstation. When creating the executable, Real-Time Workshop must be able to access an appropriate compiler. Real-Time Workshop can automatically find a compiler to use based on your default MEX compiler. For more information, see "Choosing and Configuring Your Compiler on Windows" in the Real-Time Workshop documentation. Alternatively, you can set up a compiler-specific environment variable as described below. The environment variable enables Real-Time Workshop to find and invoke the compiler independently of the default MEX compiler.

Borland

Make sure that your Borland environment variable is defined and correctly points to the directory in which your Borland compiler resides. To check this, type

```
set BORLAND
```

at the DOS prompt. The return from this includes the selected directory.

If the `BORLAND` environment variable is not defined, you must define it to point to where you installed your Borland compiler.

On Microsoft Windows, in the control panel select **System**, click the **Advanced** tab, select **Environment**, and define `BORLAND` to be the path to your compiler.

Intel

Real-Time Workshop includes support for the Intel compiler (Version 7.1 for Microsoft Windows). The Intel compiler requires Microsoft Visual C/C++ Version 6.0 or newer. However, only Version 6.0 has been verified to work with Real-Time Workshop in conjunction with the Intel compiler.

To use the Intel compiler when compiling Real-Time Workshop generated code, use `mex -setup` and select the Intel compiler.

To set up the Intel compiler manually,

- Copy `$(MATLAB)\bin\win32\mexopts\intelc71opts.bat` to `mexopts.bat` in your preferences directory (`prefdir`).
- Edit the new `mexopts.bat` file and replace `%MSVCDir%` with the root directory of your Microsoft Visual C/C++ Version 6.0 installation.
- Replace `%INTELC71%` with the root directory of the Intel compiler installation.

LCC

The freeware LCC C compiler is shipped with MATLAB, and is installed with the product. If you want to use LCC to build programs generated by Real-Time Workshop, use the version that is currently shipped with the product. Information about LCC is available at `http://www.cs.virginia.edu/~lcc-win32/`.

Microsoft Visual C/C++

Use the MATLAB command

```
mex -setup
```

to define the environment for Visual C/C++ Versions 5, 6, and 7.1.

On Windows platforms, S-function targets and model reference simulation targets are generated as DLL files via the MATLAB `mex` command. The compiler support for these targets is thus limited to that provided by`mex`.

Watcom

Note The Watcom C compiler is no longer available from the manufacturer. Development of this compiler has been taken over by the Open Watcom organization (`http://www.openwatcom.org`), which has released a binary patch update (11.0c) for existing Watcom C/C++ and Fortran customers. Real-Time Workshop continues to ship with Watcom-related target configurations. However, this policy may be subject to change in the future.

Make sure that your Watcom environment variable is defined and correctly points to the directory in which your Watcom compiler resides. To check this, type

```
set WATCOM
```

at the DOS prompt. The return from this includes the selected directory.

If the WATCOM environment variable is not defined, you must define it to point to where you installed your Watcom compiler. On Microsoft Windows 95 or 98, add

```
set BORLAND=<path to your compiler>
```

to your autoexec.bat file.

On Microsoft Windows NT or 2000, in the control panel select **System**, click the **Advanced** tab, select **Environment**, and define WATCOM to be the path to your compiler.

Out-of-Environment Error Message

If you receive out-of-environment space error messages, right-click your mouse on the program that is causing the problem (for example, dosprmpt or autoexec.bat) and choose **Properties**. From there choose **Memory**. Set the Initial Environment to the maximum allowed and click **Apply**. This should increase the amount of environment space available.

Supported Compilers

On Windows

Real-Time Workshop has been tested with these compilers on Windows.

Compiler	Versions
Borland	5.2, 5.3, 5.4, 5.5, 5.6
Intel	7.1
LCC	Use the version of LCC shipped with MATLAB.

Compiler	Versions
Microsoft Visual C/C++	5.0, 6.0, 7.0
Watcom	11.0 (see “Watcom” on page 1-14 above)

Typically you must make modifications to your setup when a new version of your compiler is released. See the MathWorks home page, `http://www.mathworks.com`, for up-to-date information on newer compilers.

On UNIX

On UNIX, the Real-Time Workshop build process uses the default compiler. The `cc` compiler is the default on all platforms except SunOS, where `gcc` is the default.

For further information, see Technical Note 1601, “What Compilers Are Supported?” at `http://www.mathworks.com/support/tech-notes/1600/1601.shtml`.

Compiler Optimization Settings

In some very rare instances, because of compiler defects, compiler optimizations applied to Real-Time Workshop generated code can cause the executable program to produce incorrect results, even though the code itself is correct.

Real-Time Workshop uses the default optimization level for each supported compiler. You can usually work around problems caused by compiler optimizations by lowering the optimization level of the compiler or by turning off optimizations. Refer to your compiler’s documentation for information on how to do this.

Real-Time Workshop Demos

A good way to familiarize yourself with Real-Time Workshop is by browsing through its suite of demos, running the ones of interest, and then inspecting code generated from these demo models. The demos illustrate many (but not all) Real-Time Workshop features.

To access the current set of demos for Real-Time Workshop, type

```
rtwdemos
```

at the MATLAB prompt. You can also find Real-Time Workshop demos by navigating to **Real-Time Workshop**, (found under **Simulink**) in the **Demos** pane of the Help browser, and clicking the **Feature Tour** example.

Note that many of the Real-Time Workshop demos illustrate features of Real-Time Workshop Embedded Coder, and are thus set up to generate code for the ERT target. Should MATLAB find no license for Real-Time Workshop Embedded Coder on your system, you can still run the demos, but all code generated will default to the GRT target.

MathWorks Products Supported by Real-Time Workshop

Many blocksets and other MathWorks products can work with Real-Time Workshop. The following table lists products that Real-Time Workshop and Real-Time Workshop Embedded Coder support.

MathWorks Products Supported by Real-Time Workshop

Product	Real-Time Workshop Support	Real-Time Workshop Embedded Coder Support
Aerospace Blockset	Yes	Yes
CDMA Reference Blockset	Yes	Yes
Communications Blockset	Yes	Yes
Gauges Blockset	Yes	Yes
Embedded Target for Infineon C166® Microcontrollers	No	Yes
Embedded Target for Motorola® HC12	No	Yes
Embedded Target for Motorola® MPC555	No	Yes
Embedded Target for OSEK/VDX®	No	Yes
Embedded Target for TI C2000™ DSP	Yes	Yes
Embedded Target for TI C6000™ DSP	Yes	Yes
Fuzzy Logic Toolbox	Yes	Yes
Link for Code Composer Studio	Yes	Yes

MathWorks Products Supported by Real-Time Workshop (Continued)

Product	Real-Time Workshop Support	Real-Time Workshop Embedded Coder Support
MATLAB Report Generator	Yes	Yes
Model Predictive Control Toolbox	Yes	Yes
Model-Based Calibration Toolbox	Yes	Yes
Real-Time Windows Target	Yes	No
Signal Processing Blockset	Yes	Yes
SimDriveLine	Yes	Yes
SimMechanics	Yes	No
SimPowerSystems	Yes	No
Simulink Fixed Point	Yes	Yes
Simulink Parameter Estimation	Yes	Yes
Simulink Report Generator	Yes	Yes
Simulink Verification and Validation	Yes	Yes
Stateflow and Stateflow Coder	Yes	Yes
Virtual Reality Toolbox	Yes	Yes
xPC Target	Yes	No

MathWorks Products Supported by Real-Time Workshop (Continued)

Product	Real-Time Workshop Support	Real-Time Workshop Embedded Coder Support
xPC Target Embedded Option	Yes	No
xPC TargetBox	Yes	No

Help and Documentation

Real-Time Workshop software is shipped with this Getting Started guide. Users of this book should be familiar with

- Using Simulink and Stateflow to create models/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 Real-Time Workshop, 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. The documentation is available in the following locations:

- Online at `http://www.mathworks.com`
- Through the MATLAB Help browser
- As PDF documents that you can view online or print

This section includes the following topics:

- "Online Documentation" on page 1-21
- "Related Products" on page 1-22
- "For Further Information" on page 1-22

Online Documentation

Access to the online information for Real-Time Workshop is through MATLAB or from the MathWorks Web site at `http://www.mathworks.com/support/`. Click the **Documentation** link.

To access the documentation with the MATLAB Help browser, use the following procedure:

1 In the MATLAB window, click **Help > Full product family help**, or click the **?** icon on the toolbar.

   The Help browser window opens.

2 In the left pane, click the **Real-Time Workshop** book icon.

   The Help browser displays the Real-Time Workshop roadmap page in the right pane. Click any link there, or click the + sign to the left of the book icon in the left pane to reveal the table of contents. When you do so, the + changes to a -.

Related Products

The MathWorks provides other products that are especially relevant to the kinds of tasks you can perform with Real-Time Workshop, including many custom targets. For information, see the MathWorks Web site.

For Further Information

The Real-Time Workshop User's Guide documents, in detail, the capabilities of Real-Time Workshop. For a topical overview of its contents, see "How Do I... in" the Real-Time Workshop User's Guide documentation.

You can customize output from Real-Time Workshop at the block, target, and makefile levels. For advanced uses, you might have to prepare or modify Target Language Compiler files, as explained in the Target Language Compiler documentation.

Building an Application

This chapter expands the high-level discussion of code generation and the build process given in Chapter 1, “Introduction”. It provides a foundation of understanding for tutorial exercises in Chapter 3, “Working with Real-Time Workshop”.

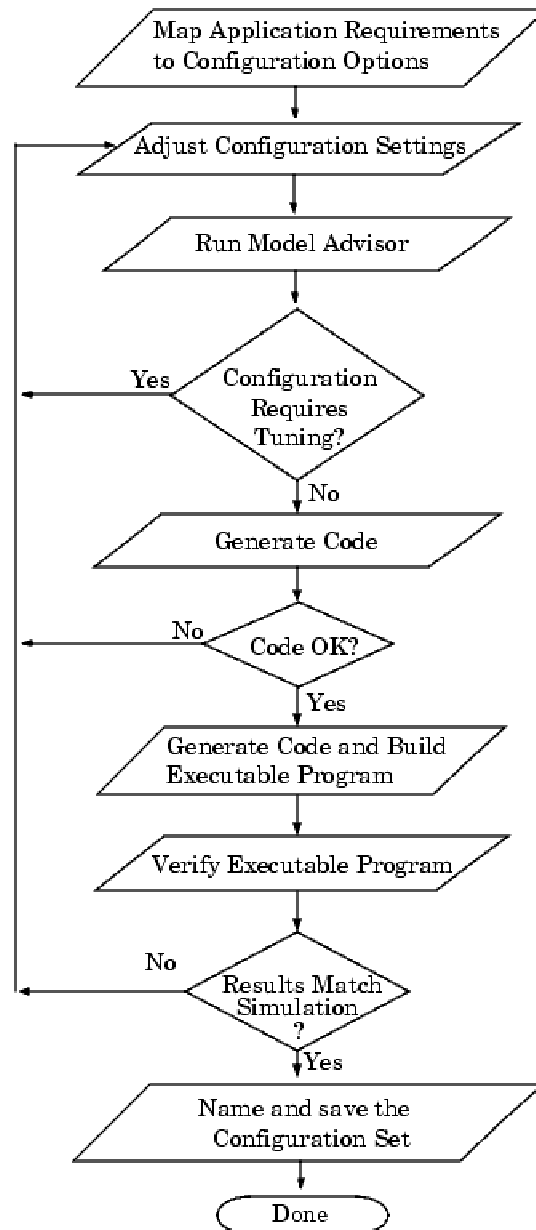
“Real-Time Workshop Workflow” (p. 2-2)	Explains the typical workflow for applying Real-Time Workshop to the application development process
“Automatic Program Building” (p. 2-16)	Describes the flow of control for code generation
“The Build Process” (p. 2-18)	Describes the sequence of events that takes place when you click the Build button, including the files that the Target Language Compiler uses and creates

Real-Time Workshop Workflow

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

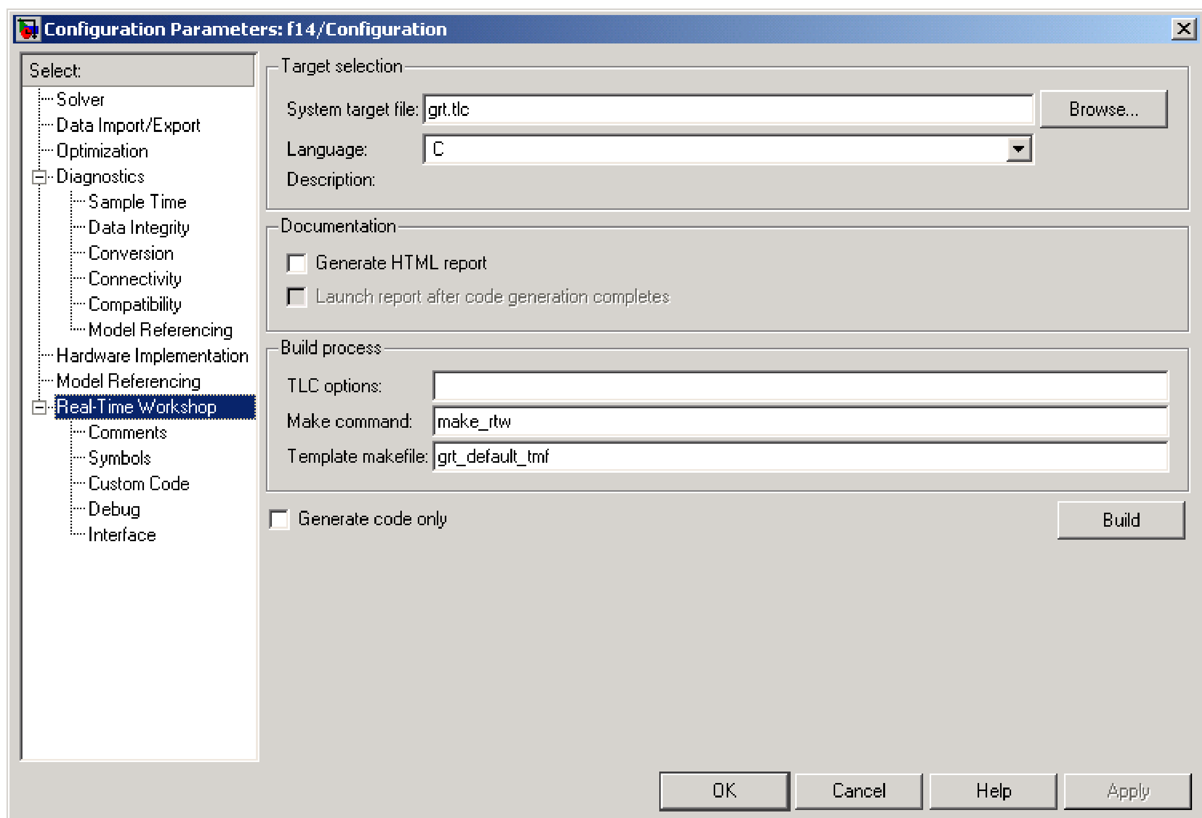
1. Map your application requirements to available configuration options.
2. Adjust configuration options as necessary.
3. Run the Model Advisor Tool.
4. If necessary, tune configuration options based on the Model Advisor report.
5. Generate code for your model.
6. Repeat steps 2 to 5 if necessary.
7. 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.

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
Generate Code and Build Executable Program
Verify Executable Program
Results Match Simulation ?
No
Yes
Name and save the Configuration Set
Done

Mapping Application Requirements to Configuration Options

The first step in applying Real-Time Workshop 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. Real-Time Workshop 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 — traceability, efficiency, 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 result? What tradeoffs can be made?

Once you have answered these questions, review the following table, which summarizes the impact of each configuration option on debugging, traceability, efficiency, and safety considerations, and indicates the default (factory) configuration settings for the generic real-time (GRT) target. For additional details, click the links in the Configuration Parameters column.

Note If you use a Real-Time Workshop Embedded Coder target, a specific embedded target, a Stateflow target, or fixed-point blocks, you need to consider the mapping of many other configuration parameters. For details, see the Real-Time Workshop Embedded Coder documentation and any other documentation specific to your target environment.

Mapping of Application Requirements to Configuration Parameters

Configuration Parameter	Debugging	Traceability	Efficiency	Safety	Default
Solver					
Stop time	No impact	No impact	No impact	No impact	10.0 sec
Type	No impact	No impact	No impact	No impact	Variable step
Data Import/Export					
Load from workspace	N/A	N/A	N/A	N/A	Clear
Save to workspace	Maybe	Maybe	Clear	No impact	Set
Save options	Maybe	Maybe	No impact	No impact	Set
Optimization					
Block reduction optimization	Clear	Clear	Set	No impact	Set
Implement logic signals as Boolean data (versus double)	No impact	No impact	Set	No impact	Set
Inline parameters	Clear	Set	Set	No impact	Clear
Conditional input branch execution	No impact	Set	Set	No impact	Set
Signal storage reuse	Clear	Clear	Set	No impact	Set

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety	Default
Application lifespan (days)	No impact	No impact	Finite value	`Inf`	`Inf`
Enable local block outputs	Clear	No impact	Set	No impact	Set
Ignore integer downcasts in fold expressions	Clear	No impact	Set	Clear	Clear
Eliminate superfluous temporary variables (Expression folding)	Clear	No impact	Set	No impact	Set
Loop unrolling threshold	No impact	No impact	>0	No impact	5
Reuse block outputs	Clear	Clear	Set	No impact	Set
Inline invariant signals	Clear	Clear	Set	No impact	Set
Hardware Implementation					
Number of bits	No impact	No impact	Target specific	No impact	8, 16, 32, 32
Real-Time Workshop					
Generate HTML report	Set	Set	No impact	No impact	Clear

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety	Default
Real-Time Workshop: Comments					
Include comments	Set	Set	No impact	No impact	Set
Simulink block comments	Set	Set	No impact	No impact	Set
Show eliminated statements	Set	Set	No impact	No impact	Clear
Verbose comments for Simulink Global storage class	Set	Set	No impact	No impact	Clear
Real-Time Workshop: Symbols					
Maximum identifier length	Set	>30	No impact	No impact	31
Real-Time Workshop: Debug					
Verbose builds	Set	No impact	No impact	Set	Set
Real-Time Workshop: Interface					
Target floating point math environment	No impact	No impact	Set	No impact	`ANSI-C`

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety	Default
Utility function generation	Shared	Shared	Shared	No impact	`Auto`*
MAT-file variable name modifier	No impact	No impact	No impact	No impact	`rt_`

* Backward compatibility with previous release

Setting Configuration Options

Once you have mapped your application requirements to appropriate configuration parameters settings, adjust the settings accordingly. Using the Default column in the table in "Mapping Application Requirements to Configuration Options" on page 2-4, 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 Real-Time Workshop", guide you through exercises that modify configuration parameters 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 "Configuration Parameter Reference" in the Real-Time Workshop documentation.

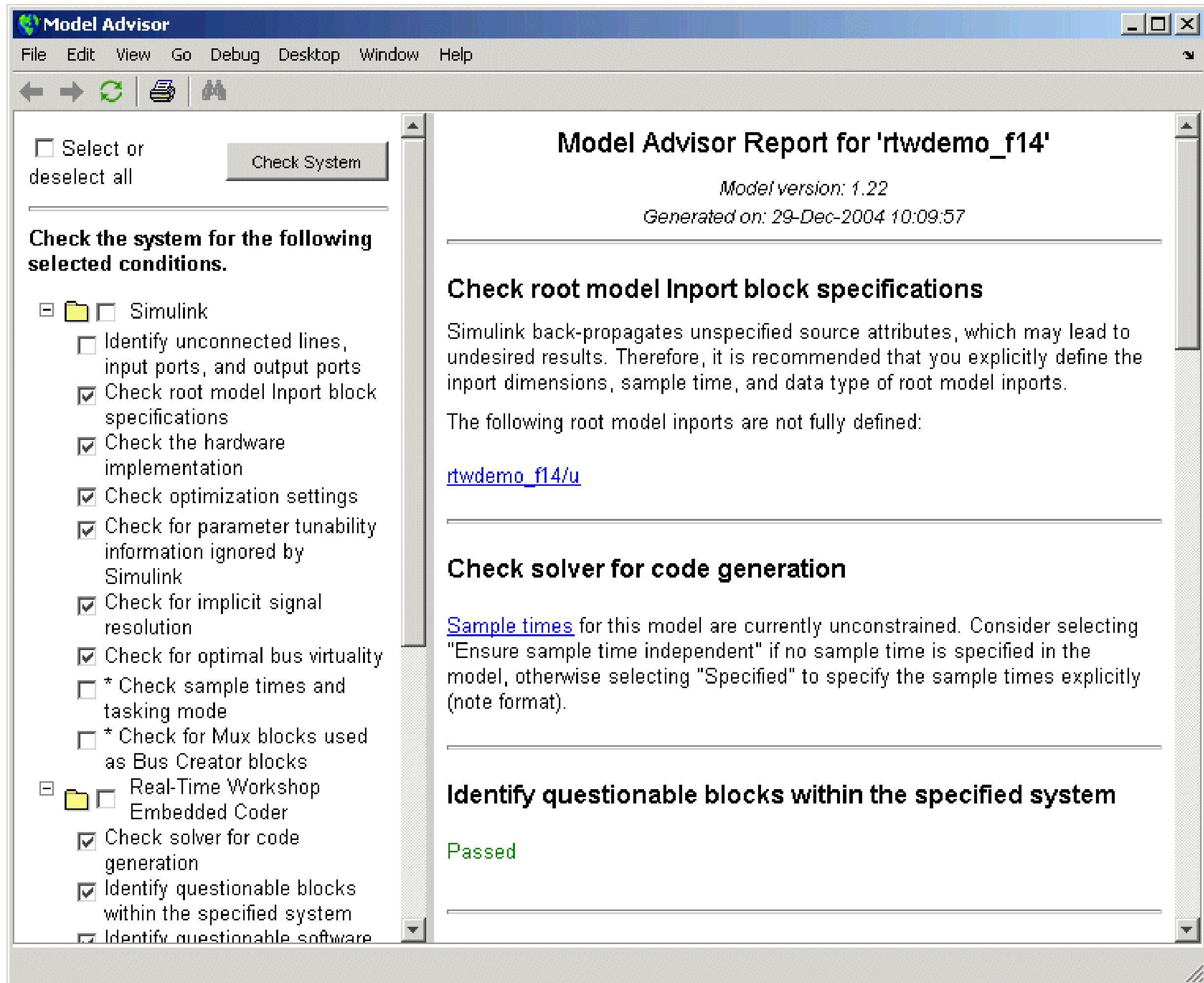
Running the Model Advisor

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 a report that lists 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 part of a sample Model Advisor report.



For more information on using the Model Advisor, see "Using the Model Advisor" in the Simulink and 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 Real-Time Workshop 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 Real-Time Workshop generating 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**). Real-Time Workshop 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 Image" on page 2-12.

For a more detailed discussion of the code generation process, see "Automatic Program Building" on page 2-16 and "The Build Process" on page 2-18. For details on the **Generate code only** option, see "Generate Code Only" in the Real-Time Workshop documentation.

Building an Executable Program Image

When you are satisfied with the code Real-Time Workshop generates for your model, build an executable 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. Real-Time Workshop responds by

1. Generating the C code, as explained in “Generating Code” on page 2-12.

2. Creating a customized makefile based on a template makefile for your target of choice and placing the makefile in the build directory.

3. 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`* or *`model`*`.exe`. The executable image resides in your working directory.

For a more detailed discussion of the build process, see “Automatic Program Building” on page 2-16 and “The Build Process” on page 2-18.

Verifying the Generated Results

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 3-21.

Saving the Model Configuration

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** while your model is still open by selecting **Model Explorer** from the **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.

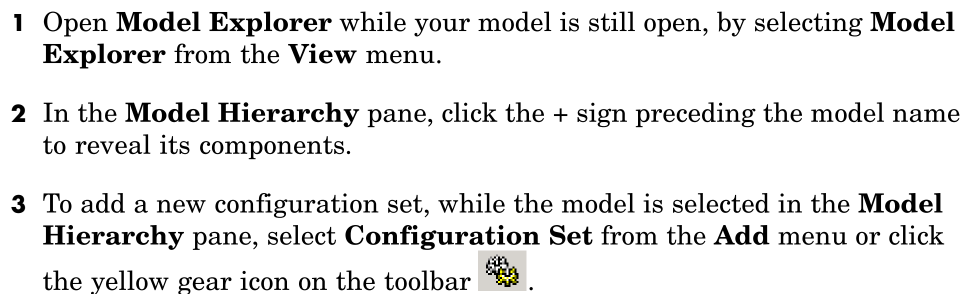
6. Save the model.

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** while your model is still open, by selecting **Model Explorer** from the **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.

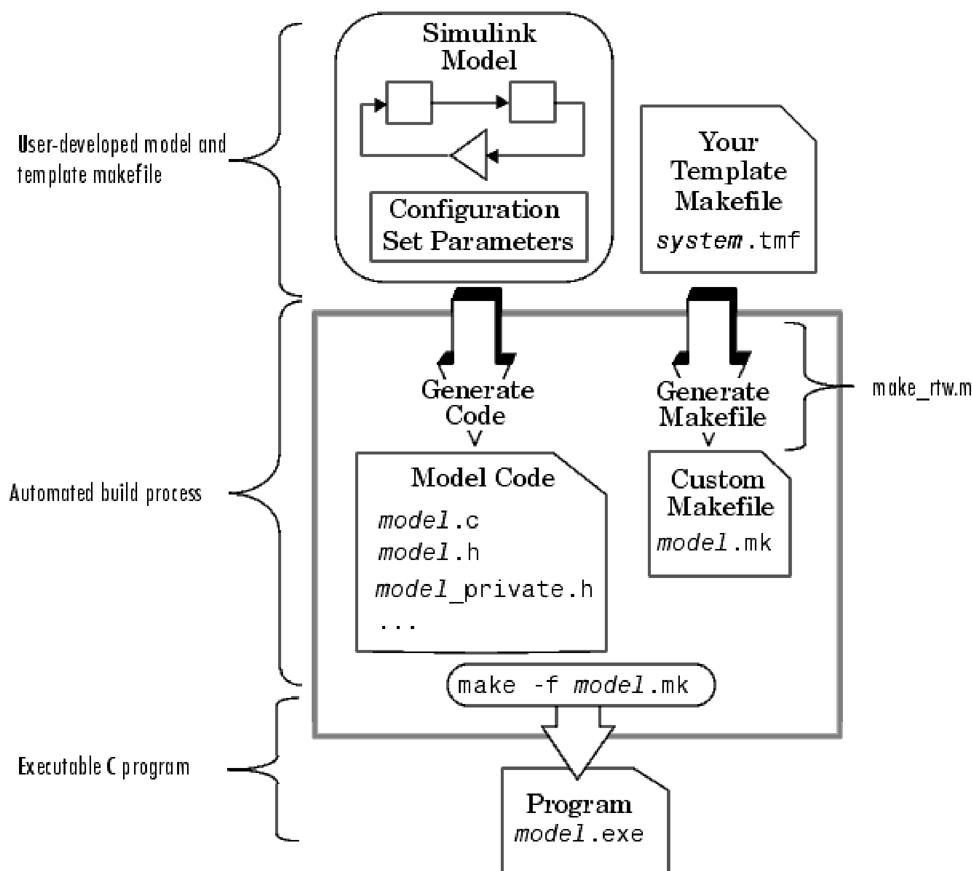
   The content of the **Is Active** field in the right pane changes from **no** to **yes**.

7. Save the model.

Automatic Program Building

Real-Time Workshop builds programs automatically for real-time applications in a variety of host environments. Using the make utility, Real-Time Workshop controls how it compiles and links the generated source code.

The following figure illustrates the complete process. The shaded box highlights portions of the process that Real-Time Workshop executes.



A high-level M-file command controls the Real-Time Workshop build process. By default, Real-Time Workshop issues the command `make_rtw` with appropriate options for most targets. You can customize this command by editing the contents of the **Make command** field in the **Build process** section of the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

Although the command may work for a standalone 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 make sure `gcs` is the model you want to build. Also, some target environments supply a **Make command** in the Configuration Parameters dialog box that is not `make_rtw`. Finally, models containing Model blocks do not build by using `make_rtw` directly.

To build (or generate code for) a model as currently configured from the MATLAB Command Window, use one of the following `rtwbuild` commands.

```
rtwbuild modelname
rtwbuild('modelname')
```

The Build Process

Real-Time Workshop generates C code only or generates the C code and produces an executable image, depending on the level of processing you choose. By default, a **Build** button appears on the **Real-Time Workshop** pane of the Configuration Parameters dialog box. This button completes the entire build process and an executable image results. If you select the **Generate code only** check box to the left of the button, the button label changes to **Generate code**.

When you click the **Build** or **Generate code** button, Real-Time Workshop completes the following process.

Note If Real-Time Workshop detects code generation constraints for your model, Real-Time Workshop issues warning or error messages.

---

1. "Model Compilation" on page 2-20 — Real-Time Workshop analyzes your block diagram (and any models referenced by Model blocks) and compiles an intermediate hierarchical representation in a file called `*model*.rtw`.

2. "Code Generation" on page 2-20 — 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.

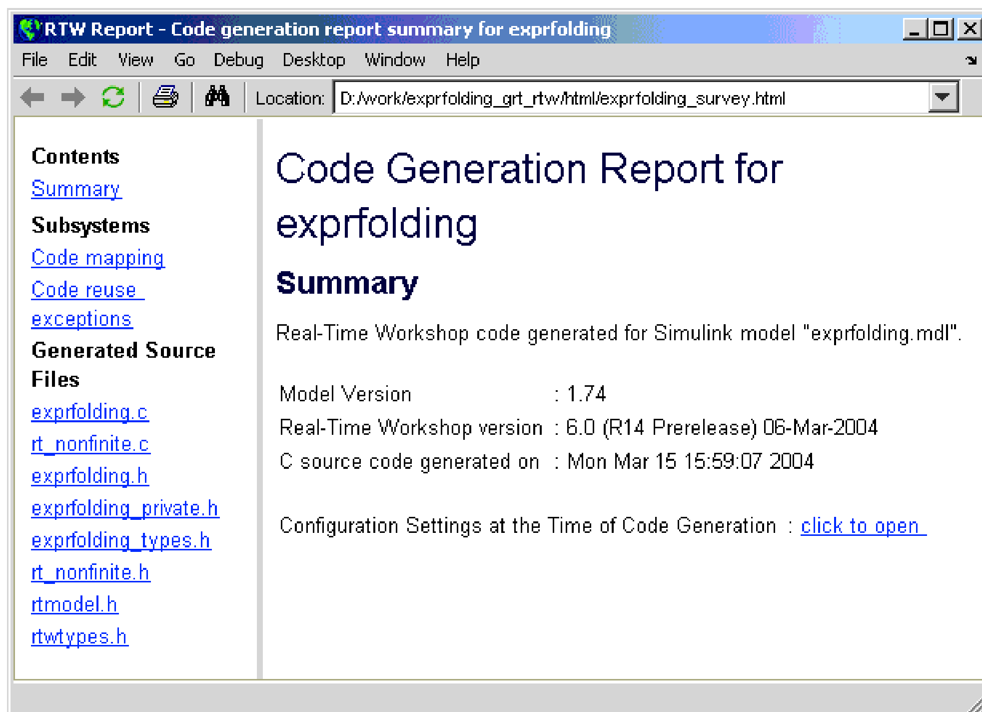
   When you click **Generate code**, processing stops here.

3. "Customized Makefile Generation" on page 2-21 — Real-Time Workshop constructs a makefile from the appropriate target makefile template and writes it in the build directory.

4. "Executable Program Generation" on page 2-22—Your system's `make` utility reads the makefile to compile source code, link object files and libraries, and generate an executable image (called *`model`* or `*model*.exe`). The makefile places the executable image in your working directory.

   If you select **Generate HTML report** on the **Real-Time Workshop** pane, a MATLAB browser window displays a report when the build is done. The report files occupy a directory called `/html` within the build directory.

The report contents vary depending on the target, but all reports feature clickable links to generated source files. The following display shows an example of an HTML code generation report for a generic real-time (GRT) target.



For more information, see "Generate HTML Report" in the Real-Time Workshop documentation. You can also view an HTML report in **Model Explorer**. See the last part of “Generating Code for a Referenced Model” on page 3-50 for details.

Details about each of the four steps in the process follow.

Model Compilation

The build process begins with Simulink compiling the block diagram. During this stage, Simulink

- Evaluates simulation and block parameters
- Propagates signal widths and sample times
- Determines the execution order of blocks within the model
- Computes work vector sizes, such as those used by S-functions. (For more information about work vectors, refer to the Simulink Writing S-Functions documentation.)

When Simulink completes this processing, it compiles an intermediate representation of the model. This intermediate description is stored in a language-independent format in the ASCII file `model.rtw`. The `model.rtw` file is the input to the next stage of the build process.

`model.rtw` files are similar in format to Simulink model (`.mdl`) files, but are only used for automated code generation. For a general description of the `model.rtw` file format, see the Target Language Compiler documentation.

Code Generation

During code generation, Real-Time Workshop uses the TLC and a supporting TLC function library to transform the intermediate model description stored in `model.rtw` into target-specific code.

The target language compiled by the TLC is an interpreted programming language designed to convert a model description to code. The TLC executes a TLC program comprising several target files (`.tlc` scripts). The TLC scripts specify how to generate code from the model, using the `model.rtw` file as input.

The TLC

1. Reads the `model.rtw` file
2. Compiles and executes commands in a system target file

The system target file is the entry point or main file. You select it from those available on the MATLAB path with the system target file browser or you can type the name of any such file on your system prior to building.

3 Compiles and executes commands in block-level target files.

For blocks in the Simulink model, there is a corresponding target file that is either dynamically generated or statically provided.

Note Real-Time Workshop executes all user-written S-function target files, but optionally executes block target files for Simulink blocks.

---

4 Writes a source code version of the Simulink block diagram

Customized Makefile Generation

After generating the code, Real-Time Workshop generates a customized makefile, `*model*.mk`. The generated makefile instructs the `make` system utility to compile and link source code generated from the model, as well as any required harness program, libraries, or user-provided modules.

Real-Time Workshop creates `*model*.mk` from a system template file, `*system*.tmf` (where `*system*` stands for the selected target name). The system template makefile is designed for your target environment. You have the option of modifying the template makefile to specify compilers, compiler options, and additional information used during the creation of the executable.

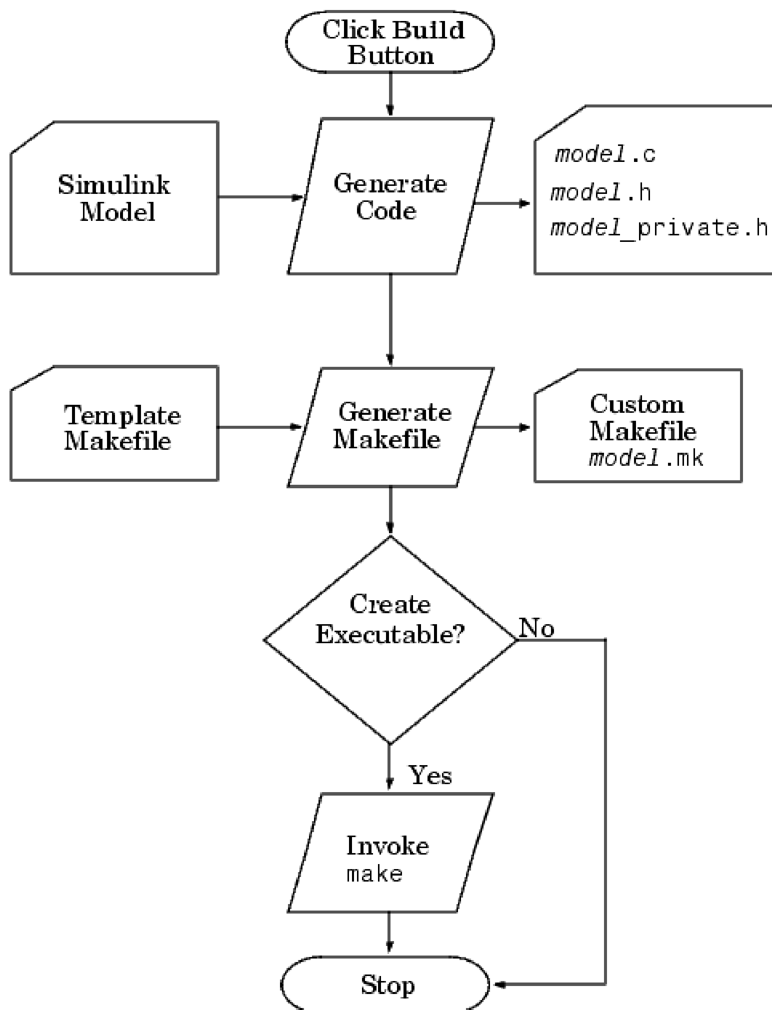
Real-Time Workshop creates the `*model*.mk` file by copying the contents of `*system*.tmf` and expanding lexical tokens (symbolic names) that describe your model's configuration.

Real-Time Workshop provides many system template makefiles, configured for specific target environments and development systems. "The System Target File Browser" in the Real-Time Workshop documentation lists all template makefiles that are bundled with Real-Time Workshop. To see an example template makefile, navigate to `rtw\c\grt` from the MATLAB root, and open with an editor the file `grt_msvc.tmf`. (You can determine the MATLAB root by typing `matlabroot` on the MATLAB command line.)

You can fully customize your build process by modifying an existing template makefile or providing your own template makefile.

Executable Program Generation

The following figure shows how Real-Time Workshop controls automatic program building.



During the final stage of processing, Real-Time Workshop invokes the generated makefile, *model*.mk, which in turn compiles and links the generated code. On PC platforms, a batch file is created to invoke the generated makefile. The batch file sets up the proper environment for invoking the make utility and related compiler tools. To avoid unnecessary recompilation of C files, the make utility performs date checking on the dependencies between the object

and C files; only out-of-date source files are compiled. Optionally, the makefile can download the resulting executable image to your target hardware.

This stage is optional, as illustrated by the control logic in the preceding figure. You might choose to omit this stage, for example, if you are targeting an embedded microcontroller or a digital signal processing (DSP) board.

To omit this stage of processing, select the **Generate code only** option on the **Real-Time Workshop** pane of the Configuration Parameters dialog box. You can then cross-compile your code and download it to your target hardware. "Making an Executable" in the Real-Time Workshop documentation discusses the options that control whether or not the build creates an executable image.

Files and Directories Created by the Build Process

The following sections discuss

- “Files Created During Build Process” on page 2-24
- “Directories Used During the Build Process” on page 2-30

Files Created During Build Process

This section lists `model.*` files created during the code generation and build process for the GRT and GRT `malloc` targets when used with standalone models. Additional directories and files are created to support shared utilities and model references (see "Building Subsystems and Working with Referenced Models" in the Real-Time Workshop documentation).

Real-Time Workshop derives many of the files from the `model.mdl` file you create with Simulink. You can think of the `model.mdl` file as a very high-level programming language source file.

Note Files generated by the Real-Time Workshop Embedded Coder are packaged slightly differently. Depending on model architectures and code generation options, Real-Time Workshop might generate other files.

Below is a directory listing for the F14 demo model, which is configured for the GRT target.

```
F14_GRT_RTW\
| f14.mk
| rtw_proj.tmw
| f14.bat
| f14_targ_data_map.m
| rtwtypes.h
| f14_types.h
| f14.h
| f14.c
| f14_private.h
| rtmodel.h
| f14_data.c
| rt_nonfinite.h
| rt_nonfinite.c
| modelsources.txt
| f14_dt.h
|
+---tlc
| This directory contains TLC files generated during the build
| process. These files are used to create the generated C source
| and header files.
[[NOT-EQUAL]]
+---html
| This directory contains generated HTML files used for the Code
| Generation report.
```

Descriptions of the principal generated files follow. Note that these descriptions use the generic term `model` for the model name:

- *`model`*`.rtw`

  An ASCII file, representing the compiled model, generated by the Real-Time Workshop build process. This file is analogous to the object file created from a high-level language source program. By default, Real-Time Workshop deletes this file when the build process is complete. However, you can choose to retain the file for inspection.

- *`model`*`.c`

C source code that corresponds to *model*.mdl and is generated by the Target Language Compiler. This file contains

- Include files *model*.h and *model*_private.h
- All data except data placed in *model*_data.c
- Model-specific scheduler code
- Model-specific solver code
- Model registration code
- Algorithm code
- Optional GRT wrapper functions MdlStart, MdlOutputs, MdlUpdate, MdlInitializeSizes, MdlInitializeSampleTimes

• *model*.h

Defines model data structures and a public interface to the model entry points and data structures. Also provides an interface to the real-time model data structure (*model*_rtM) via access macros. *model*.h is included by subsystem .c files in the model. It includes

- Exported Simulink data symbols
- Exported Stateflow machine parented data
- Model data structures, including rtM
- Model entry point functions

• *model*_private.h

Contains local define constants and local data required by the model and subsystems. This file is included by the generated source files in the model. You might need to include *model*_private.h when interfacing legacy hand-written code to a model. See "Header Dependencies When Interfacing Legacy/Custom Code with Generated Code" in the Real-Time Workshop documentation for more information. This header file contains

- Imported Simulink data symbols
- Imported Stateflow machine parented data
- Stateflow entry points

   - Real-Time Workshop details (various macros, `enums`, and so forth that are private to the code)

- *`model`*`_types.h`

  Provides forward declarations for the real-time model data structure and the parameters data structure. These might be needed by function declarations of reusable functions. *`model`*`_types.h` is included by all the generated header files in the model.

- *`model`*`_data.c`

  A conditionally generated C source code file containing declarations for the parameters data structure and the constant block I/O data structure, and any zero representations for structure data types that are used in the model. If these data structures are not used in the model, *`model`*`_data.c` is not generated. Note that these structures are declared `extern` in *`model`*`.h`. When present, this file contains

   - Constant block I/O parameters
   - Include files *`model`*`.h` and *`model`*`_private.h`
   - Definitions for the zero representations for any user-defined structure data types used by the model
   - Constant parameters

- *`model`*`.exe` (on PC) or *`model`* (on UNIX), generated in the current directory, not in the build directory

  Executable program file created under control of the `make` utility by your development system (unless you have explicitly specified that Real-Time Workshop generate code only and skip the rest of the build process)

- *`model`*`.mk`

  Customized makefile generated by the Real-Time Workshop build process. This file builds an executable program file.

- `rtmodel.h`

  Contains `#include` directives required by static main program modules such as `grt_main.c` and `grt_malloc_main.c`. Since these modules are not created at code generation time, they include `rt_model.h` to access model-specific data structures and entry points. If you create your own main program module, take care to include `rtmodel.h`

- `rtwtypes.h`

  For GRT targets, a header file that includes `simstruc_types.h`, which, in turn, includes `tmwtypes.h`. For Real-Time Workshop Embedded Coder ERT targets, `rtwtypes.h` itself provides the necessary `defines`, `enums`, and so on, instead of including `tmwtypes.h` and `simstruc_types.h`. The `rtwtypes.h` file generated for ERT is an optimized (reduced) file based on the settings provided with the model that is being built. See "Header Dependencies When Interfacing Legacy/Custom Code with Generated Code" in the Real-Time Workshop documentation for more information.

- `rt_nonfinite.c`

  C source file that declares and initializes global nonfinite values for `inf`, minus `inf`, and `nan`. Nonfinite comparison functions are also provided. This file is always generated for GRT-based targets and optionally generated for other targets.

- `rt_nonfinite.h`

  C header file that defines `extern` references to nonfinite variables and functions. This file is always generated for GRT-based targets and optionally generated for other targets.

- `rtw_proj.tmw`

  File Real-Time Workshop generated for the `make` utility to use to determine when to rebuild objects when the name of the current Real-Time Workshop project changes

- *`model`*`.bat`

  Windows batch file used to set up the appropriate compiler environment and invoke the make command

  `modelsources.txt`

  List of additional sources that should be included in the compilation.

Optional files:

- *`model`*`_targ_data_map.m`

  M-file used by external mode to initialize the external mode connection

- *`model_dt`*`.h`

C header file used for supporting external mode. Declares structures that contain data type and data type transition information for generated model data structures.

- *subsystem*.c

  C source code for each noninlined nonvirtual subsystem or copy thereof when the subsystem is configured to place code in a separate file

- *subsystem*.h
- C header file containing exported symbols for noninlined nonvirtual subsystems. Analogous to *model*.h.
- *model*_capi.h

  An interface header file between the model source code and the generated C-API. See "C-API for Interfacing with Signals and Parameters" in the Real-Time Workshop User's Guide documentation for more information.

- *model*_capi.c

  C source file that contains data structures that describe the model's signals and parameters without using external mode. See "C-API for Interfacing with Signals and Parameters" in the Real-Time Workshop User's Guide documentation for more information.

- rt_sfcn_helper.h, rt_sfcn_helper.c

  Header and source files providing functions needed by noninlined S-functions in a model. The functions rt_CallSys, rt_enableSys, and rt_DisableSys are used when noninlined S-functions call downstream function call subsystems.

In addition, when you select the **Generate HTML report** option, Real-Time Workshop generates a set of .html files (one for each source file plus a *model*_contents.html index file) in the /html subdirectory within your build directory.

The above source files have dependency relationships, and there are additional file dependencies that you might need to take into account. These are described in "Generated Source Files and File Dependencies" in the the Real-Time Workshop documentation.

Directories Used During the Build Process

Real-Time Workshop places output files in three directories during the build process:

- The working directory

  If you choose to generate an executable program file, Real-Time Workshop writes the file `model.exe` (on PC) or `model` (on UNIX) to your working directory.

- The build directory — `model_target_rtw`

  A subdirectory within your working directory. The build directory name is `model_target_rtw`, where `model` is the name of the source model and `target` is the selected target type (for example, `grt` for the generic real-time target). The build directory stores generated source code and all other files created during the build process (except the executable program file).

- Project directory — `slprj`

  A subdirectory within your working directory. When models referenced via Model blocks are built for simulation or Real-Time Workshop code generation, files are placed in `slprj`. The Real-Time Workshop Embedded Coder has an option that places generated shared code in `slprj` without the use of model reference. Subdirectories in `slprj` provide separate places for simulation code, some Real-Time Workshop code, utility code shared between models, and other files. Of particular importance to Real-Time Workshop users are

  - Model reference RTW target files

    `slprj/target/modelName/`

  - MAT–files used during code generation of model reference RTW target and standalone code generation

    `slprj/target/modelName/tmwinternal`

  - Shared (fixed-point) utilities

    `slprj/target/_sharedutils`

  See “Working with Project Directories” on page 3-62 for more information on organizing your files with respect to project directories.

The build directory always contains the generated code modules *model*.c, *model*.h, and the generated makefile *model*.mk.

Depending on the target, code generation, and build options you select, the build directory might also include

- *model*.rtw
- Object files (.obj or .o)
- Code modules generated from subsystems
- HTML summary reports of files generated (in the /html subdirectory)
- TLC profiler report files
- Block I/O and parameter tuning information file (*model*_capi.c)
- C-API code for parameters and signals
- Real-Time Workshop project (*model*.tmw) files

For additional information about using project directories, see "Project Directory Structure for Model Reference Targets" and "Supporting the Shared Utilities Directory in the Build Process" in the Real-Time Workshop documentation.

Working with Real-Time Workshop

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

"The f14 Demonstration Model" (p. 3-3)	Describes the Simulink model that the first three tutorial use
"Building a Generic Real-Time Program" (p. 3-4)	Shows how to generate C code from a Simulink demo model and build an executable program
"Data Logging" (p. 3-15)	Explains how to modify the demonstration program to save data in a MATLAB MAT-file for plotting
"Code Verification" (p. 3-21)	Demonstrates how to verify a generated program by comparing its output to that of the original model under simulation
"A First Look at Generated Code" (p. 3-27)	Examines code generated for a very simple model, illustrating the effects of some of the Real-Time Workshop code generation options

"Working with External Mode Using GRT" (p. 3-38)	Acquaints you with the basics of using external mode on a single computer, and demonstrates the value of external mode to rapid prototyping
"Generating Code for a Referenced Model" (p. 3-50)	Introduces generating code for models referenced with Model blocks, and using **Model Explorer** to browse code files

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

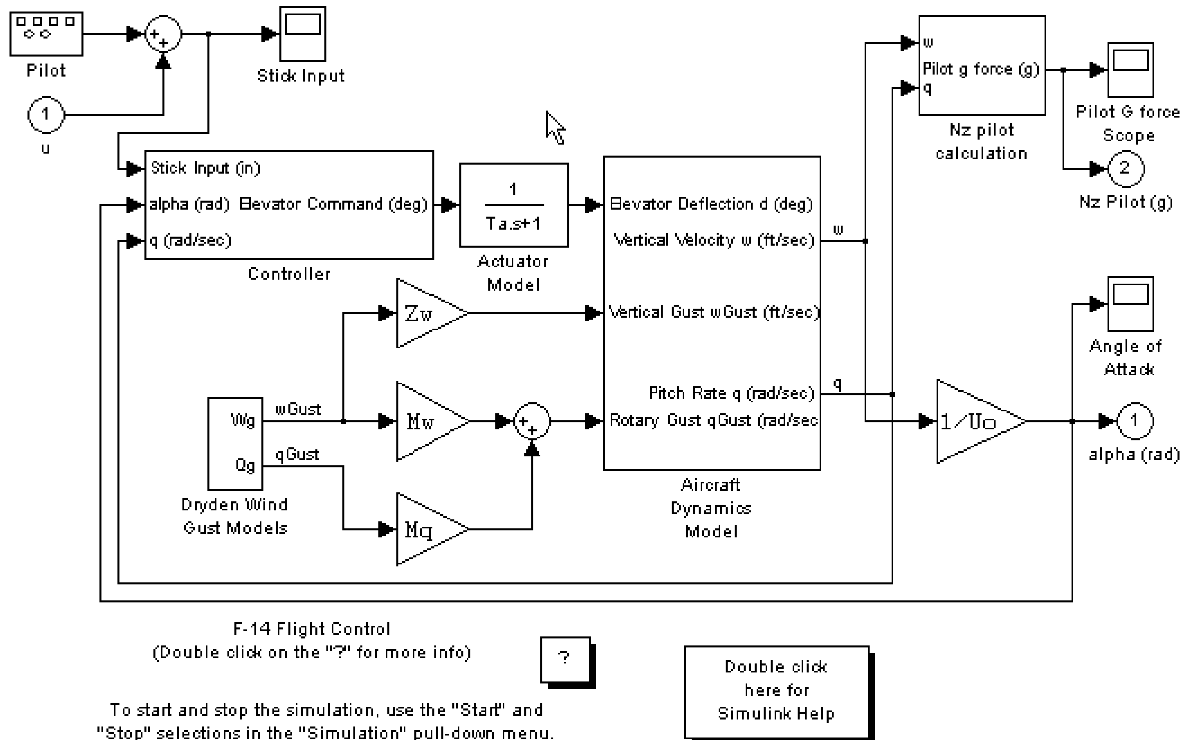
These tutorials assume basic familiarity with MATLAB and Simulink. You should also read Chapter 2, "Building an Application", before proceeding.

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. See "Supported Compilers" on page 1-15 in the Real-Time Workshop documentation for more information on supported compilers and compiler installation.

The f14 Demonstration Model

The first three tutorials use a demonstration Simulink model, `f14.mdl`, from the *`matlabroot`*`/toolbox/simulink/simdemos/aerospace` directory. (By default, this directory is on your MATLAB path; *`matlabroot`* is the location of MATLAB on your system.) `f14` is a model of a flight controller for the longitudinal motion of a Grumman Aerospace F-14 aircraft. Activate the F-14 model by typing `f14` at the MATLAB prompt. The diagram below displays 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

This tutorial walks through the process of generating C code and building an executable program from the demonstration model. The resultant standalone 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 `f14` model, stored in its own directory, `f14example`. This discussion assumes that the `f14example` directory resides on drive `d:`. Set up your working directory as follows:

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

   ```
   !mkdir d:\f14example (on PC)
   ```

   or

   ```
   !mkdir ~/f14example (on UNIX)
   ```

2. Make `f14example` your working directory (drive `d:` used as example).

   ```
   cd d:/f14example
   ```

3. Open the `f14` model.

   ```
   f14
   ```

   The model appears in the Simulink window.

4. From the **File** menu, choose **Save As**. You need to navigate to or to specify your working directory. Save a copy of the `f14` model as `d:/f14example/f14rtw.mdl`.

During code generation, Real-Time Workshop 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 3-50 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 Real-Time Workshop can generate code for models using variable-step solvers for rapid simulation (`rsim`) and S-function targets only. A Simulink license is checked out when `rsim` targets execute. See "Licensing Protocols for Simulink Solvers in Executables" in the Real-Time Workshop documentation for details.

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

1. From the **View** menu, choose **Model Explorer**. The **Model Explorer** opens.

2. Expand **f14rtw** in the **Model Hierarchy** pane.

3. Click **Configuration (Active)** and then, in the center pane, click **Solver**. The **Solver** pane appears at the right.

4. Enter the following parameter values on the **Solver** pane in their respective edit fields and menus:

   Start Time: `0.0`

   Stop Time: `60`

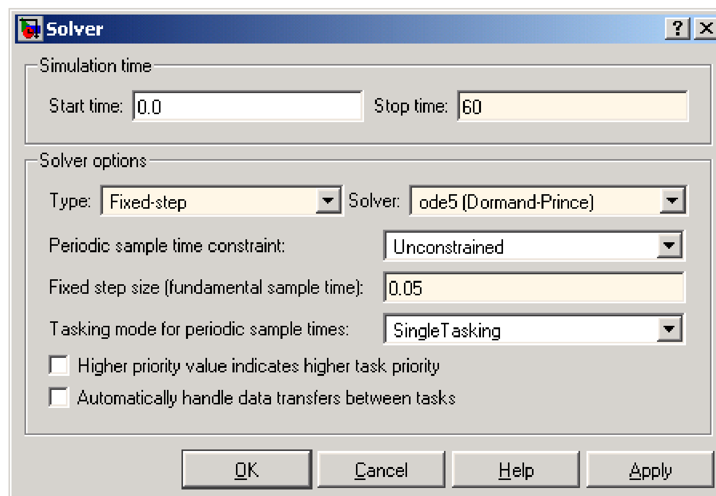
Solver options: set **Type** to `Fixed-step`. Select the `ode5 (Dormand-Prince)` solver algorithm.

Fixed step size: `0.05`

Mode: `SingleTasking`

The **Solver** pane with the modified parameter settings is shown below. Note the tan background color of the controls you just changed; use this visual feedback to double check that what you set is what you intended. When you apply your changes, the background color reverts to white.

Note There are several ways to access simulation and code generation parameters. The views of Configuration Parameters dialog box shown in these tutorials are those you obtain from expanding the `Configuration` node in the **Model Hierarchy** pane of **Model Explorer**, and then right-clicking a subnode (such as `Solver`) in the **Contents** pane and selecting **Properties** from the menu. The same dialog panes appear in the Configuration Parameters dialog box and the **Model Explorer**.



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

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

Selecting the Target Configuration

Note 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.

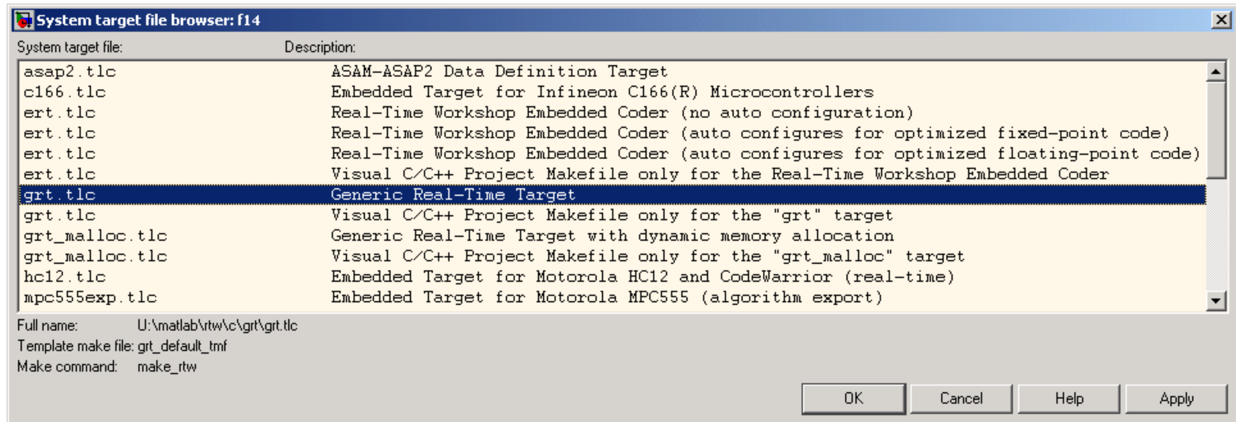
In these exercises (and in most applications), you do not need to specify these parameters individually. Here, you use the ready-to-run generic real-time target (GRT) configuration. The GRT target is designed to build a standalone executable program that runs on your workstation.

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

1. Click **Real-Time Workshop** in the center pane. The **Real-Time Workshop** pane appears at the right. This pane has several tabs.

2. Click **General** to activate the pane that controls target selection.

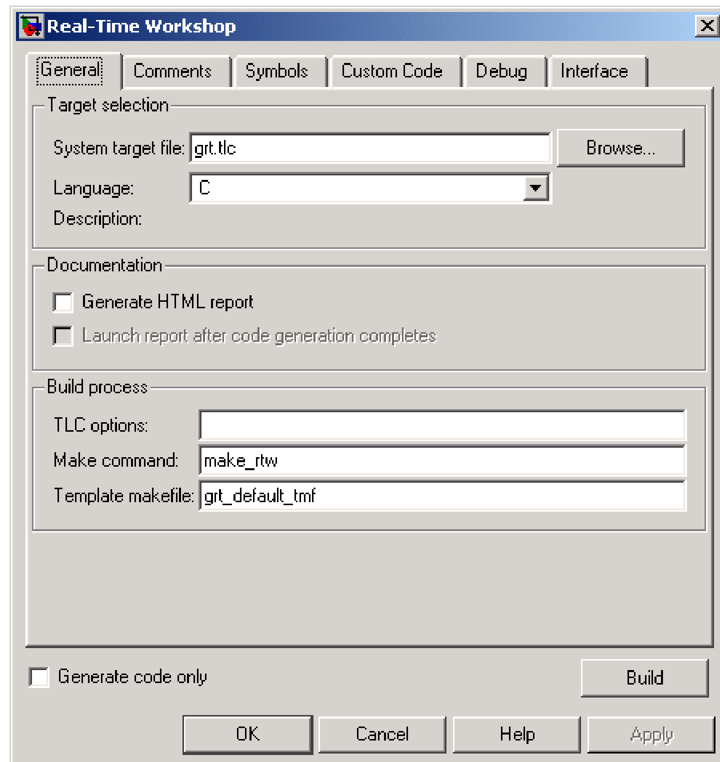
3. Click the **Browse** button next to the **RTW system target file** field. This opens the System Target File Browser, illustrated below. The browser displays a list of all currently available target configurations. When you select a target configuration, Real-Time Workshop automatically chooses the appropriate system target file, template makefile, and `make` command.



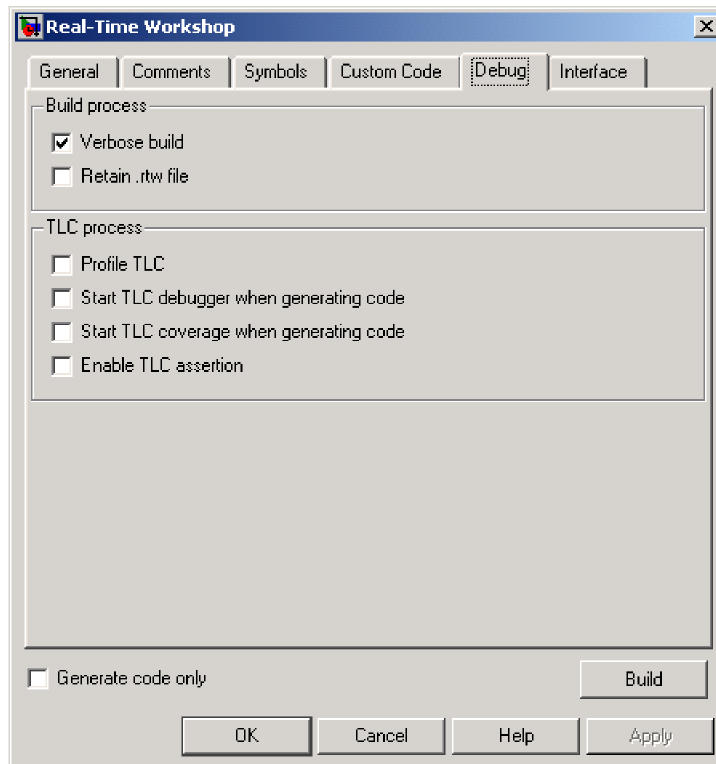
4 From the list of available configurations, select `Generic Real-Time Target` (as shown above) and then click **OK**.

Note The system target file browser lists all system target files found on the MATLAB path. Some of these might require additional licensed products (for example, Real-Time Workshop Embedded Coder) in order to use them.

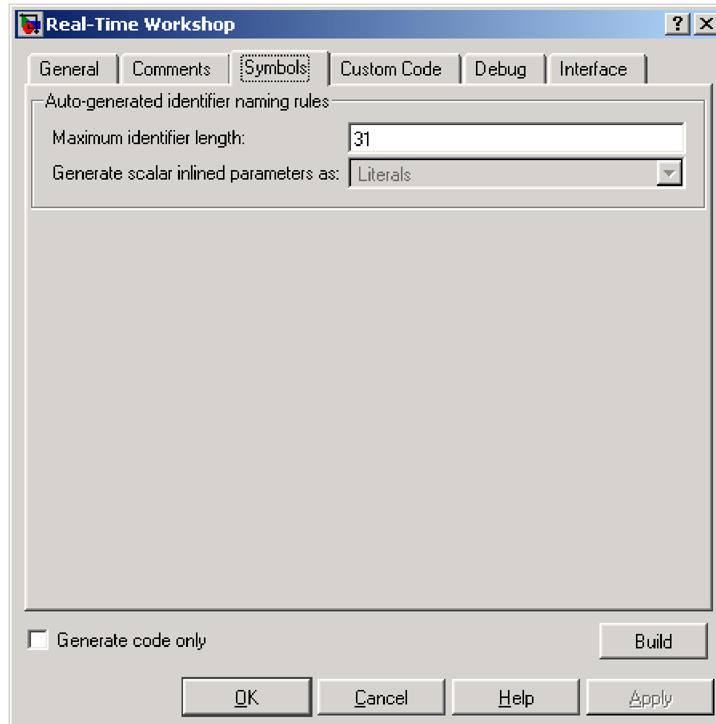
The **General Real-Time Workshop** pane displays the correct system target file (`grt.tlc`), `make` command (`make_rtw`), and template makefile (`grt_default_tmf`), as shown below:


Real-Time Workshop
General
Comments
Symbols
Custom Code
Debug
Interface
Target selection
System target file:
grt.tlc
Browse...
Language:
C
Description:
Documentation
Generate HTML report
Launch report after code generation completes
Build process
TLC options:
Make command:
make_rtw
Template makefile:
grt_default_tmf
Generate code only
Build
OK
Cancel
Help
Apply

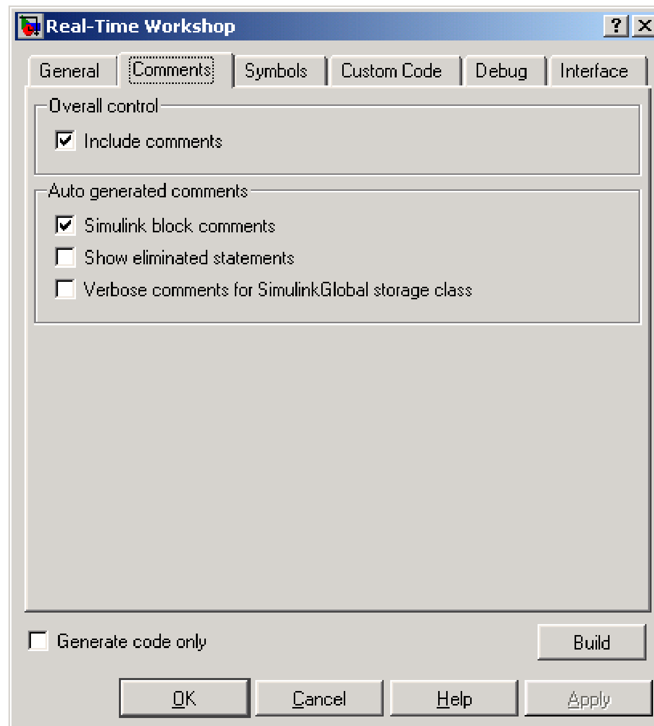
5 Select the **Debug** tab of the **Real-Time Workshop** pane. The options displayed here control build verbosity and debugging support, and are common to all target configurations. Make sure that all options are set to their defaults, as shown below.



6 Select the **Symbols** tab of the **Real-Time Workshop** pane. The options on this pane control the look and feel of generated code; there is only one for the GRT target, **Maximum identifier length** (the number of characters allowed in variable and other symbol names). The default for this option is `31`, as shown below.



7 Select the **Comments** tab of the **Real-Time Workshop** pane. The options displayed here control the types of comments included in generated code. The default is to include only block-level comments, as shown below.



8 Make sure that the **Generate code only** option is not selected, and 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 Select the **General** tab of the **Real-Time Workshop** pane, and 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 messages are

```
### Starting Real-Time Workshop build procedure for model:
f14rtw
### Generating code into build directory: D:\Work\f14rtw_grt_rtw
```

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 model: f14rtw
```

The working directory now contains an executable, `f14rtw.exe` (on PC) or `f14rtw` (on UNIX). In addition, a build directory, f14rtw_grt_rtw, has been created in your working directory.

2 To observe the contents of the working directory after the build, type the `dir` command from the 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 standalone `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
```

Contents of the Build Directory

The build process creates a build directory and names it model_target_rtw, concatenating the name of the source model and the chosen target. In this example, the build directory is named f14rtw_grt_rtw.

f14rtw_grt_rtw contains these generated source files:

File	Description
f14rtw.c	Standalone C code that implements the model
f14rtw_data.c	Initial parameter values used by the model
rt_nonfinite.c	Function to initialize nonfinite types (Inf, NaN, and -Inf)
f14rtw.h	An include header file containing definitions of parameters and state variables
f14rtw_types.h	Forward declarations of data types used in the code
f14rtw_private.h	Header file containing common include definitions
rt_nonfinite.h	Imported declarations for nonfinite types
rtwtypes.h	Static include file for Simulink simstruct data types; some embedded targets tailor this file to reduce overhead, but GRT does not
rtmodel.h	Master header file for including generated code in the static main program (its name never changes, and it simply includes f14rtw.h)
f14rtw.mk	Makefile generated from a template for the GRT target

The build directory also contains other files used in the build process, such as the object (.obj) files, a batch control file (f14rtw.bat), and a marker file (rtw_proj.tmw).

Data Logging

The 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, you use the **Real-Time Workshop > Data Import/Export** pane. The process is the same as configuring a Simulink model to save output to the MATLAB workspace. In addition, however, for each workspace return variable you define and enable, Real-Time Workshop 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 from the **Real-Time Workshop > Interface** pane.

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). Real-Time Workshop 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 such 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.

Data Logging During Simulation

To use the data logging feature,

1. From the **View** menu, select **Model Explorer**. The **Model Explorer**opens.

2. Expand **f14rtw** in the **Model Hierarchy** pane.

3. Click **Configuration (Active)** and then, in the center pane, click **Data Import/Export**. 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.

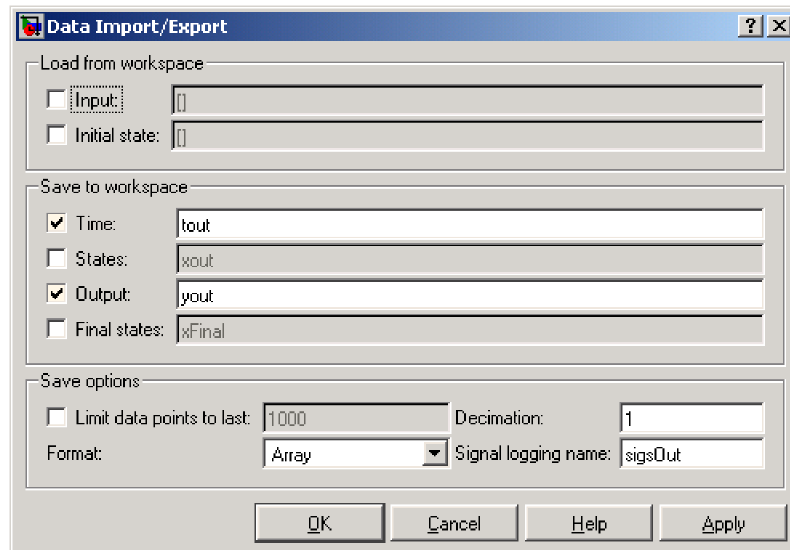
4. Select the **Time** option. This tells Simulink to save time step data during simulation as a variable named `tout`. You can enter a different name to distinguish different simulation runs (for example using different step sizes), but allow the name to default for this exercise. Selecting **Time** enables Real-Time Workshop to generate code that logs the simulation time to the MAT-file with a related name.

5. Select the **Output** option. This tells Simulink to save time step data during simulation as a variable named `yout`. Selecting **Output** enables Real-Time Workshop to generate code that logs the root Output blocks (Angle of Attack and Pilot G Force) to the MAT-file with related names.

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.

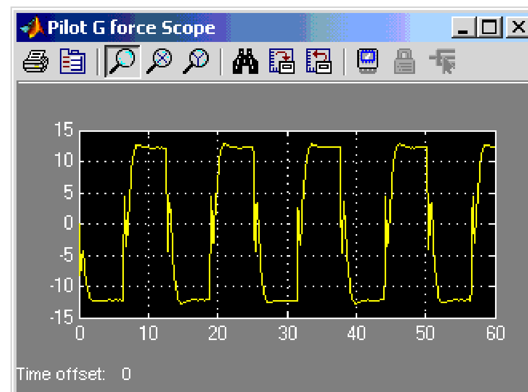
---

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



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

8 Open the Pilot G Force Scope block, and run the model by clicking the **Start** button in the model's toolbar. The resulting Pilot G Force scope display is shown below.



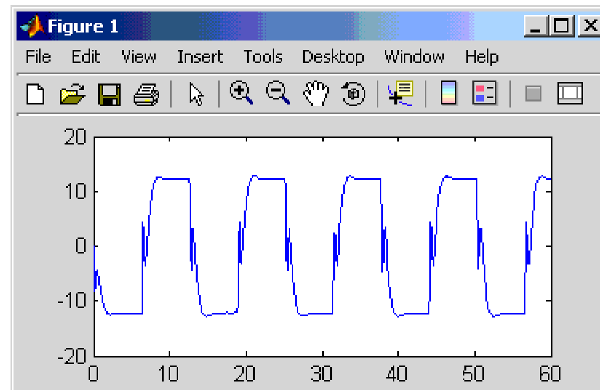
9 Verify that the simulation time and outputs have been saved to the workspace.

```
whos *out
  Name       Size                    Bytes  Class

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

Grand total is 1803 elements using 14424 bytes
```

10 Verify that Pilot G Force was correctly logged by plotting simulation time versus that variable.

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

The resulting plot is shown below.



Data Logging from Generated Code

In the second part of the exercise, you build and run a Real-Time Workshop executable of the `f14rtw` model that outputs a MAT-file containing the variables that you logged in. Even if you have already generated code for the `f14rtw` model, you must register code for it because you have changed the model by enabling data logging. The steps below explain this.

Real-Time Workshop modifies identifiers for variables logged by Simulink in order to avoid overwriting workspace data from simulation runs. You control these modifications from the **Model Explorer**:

1 From the **View** menu, choose **Model Explorer**. The **Model Explorer** opens.

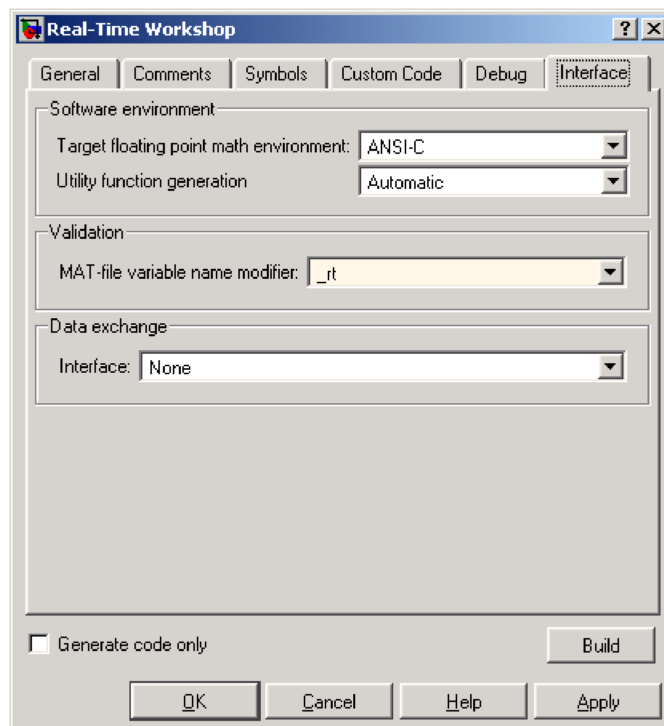
2 Expand **f14rtw** in the **Model Hierarchy** pane.

3 Click **Configuration Active** and then, in the center pane, click **Real-Time Workshop**. The **Real-Time Workshop** pane appears to the right.

4 Click the **Interface** tab.

5 Set the **MAT-file variable name modifier** menu to `_rt`. This will add the suffix `_rt` to each variable that you selected to be logged in the first part of this exercise (`tout`, `yout`).

6 Deselect the **Generate code only** check box, if it is currently selected. The dialog box appears as below.



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

8 Generate code and build an executable: Click the **Build** button, type **Ctrl+B**, or select **Build Model** from the **Real-Time Workshop** submenu of the **Tools** menu in the model window.

9 When the build concludes, run the executable with the command

```
!f14rtw
```

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

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

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

```
load f14rtw.mat
whos tout* yout*
  Name          Size                   Bytes  Class

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

Grand total is 3606 elements using 28848 bytes
```

Note that all arrays have the same number of elements.

12 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))
```

This plot is identical to the plot you produced in step 10 of the first part of this exercise.

Code Verification

In this tutorial, you verify the answers computed by code generated from the `f14rtw` model. You do this by capturing output data from runs of the Simulink model and the generated program and comparing their respective values.

Note To obtain a valid comparison between outputs of the model and the generated program, make sure that you select the same **Solver options** (`fixed-step`, `ode5 (Dormand-Prince)`) and the same step size (`0.05`) for both the Simulink run and the Real-Time Workshop build process. Also, make sure that the model is configured to save simulation time, as 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 exercise, "Data Logging" on page 3-15.

To configure the Scope blocks to log data,

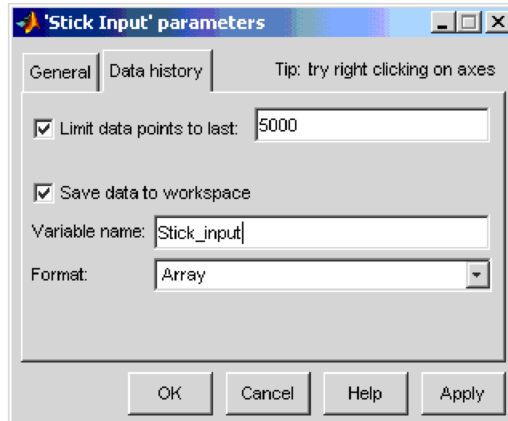
1. Clear the workspace and reload the model so that the proper workspace variables are declared and initialized:

   ```
   clear
   f14rtw
   ```

2. Open the Stick Input Scope block and click the **Parameters** button on the toolbar of the Scope window. The **Stick Input parameters** dialog box opens.

3. Select the **Data History** tab of the **Scope Input Properties** dialog box.

4. Select the **Save data to workspace** option and enter the name of the variable (`Stick_input`) that is to receive the scope data. The dialog box appears as shown below.



The Stick Input signal to the Scope block is logged to the array `Stick_input` during simulation. The generated code logs the same signal data to the MAT-file variable `rt_Stick_input` during a run of the executable program.

5. Click the **Apply** button.

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

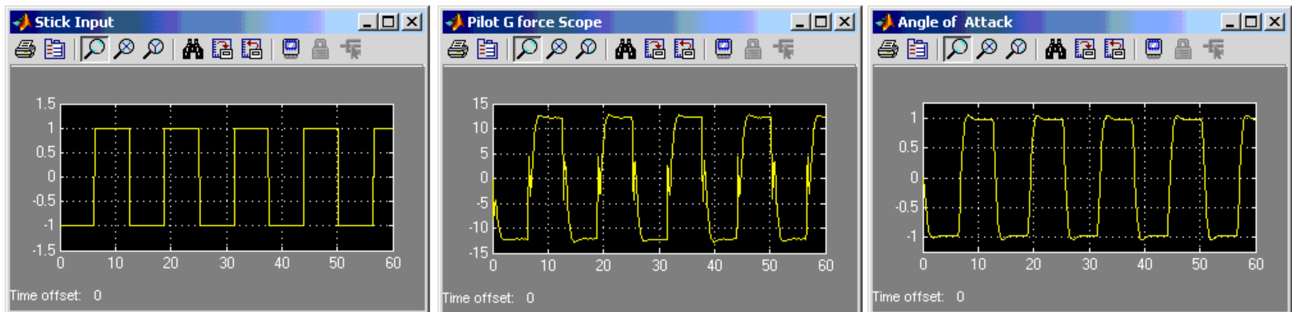
7. Save the model.

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 scope plots are displayed.



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 in order to obtain a valid data file:

1 Set the variable name modifier to be used by Real-Time Workshop. Select **Real-Time Workshop** on the center pane of the **Model Explorer**, and click the **Interface** tab. The **Interface** pane appears.

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

3 Click **Apply**.

4 Generate code and build an executable: Click the **Build** button, type **Ctrl+B**, or select **Build Model** from the **Real-Time Workshop** submenu of the **Tools** menu in the model window. Status messages in the console window track the build process.

5 When the build finishes, run the standalone program from MATLAB.

```
!f14rtw
```

The executing program writes the following messages to the console.

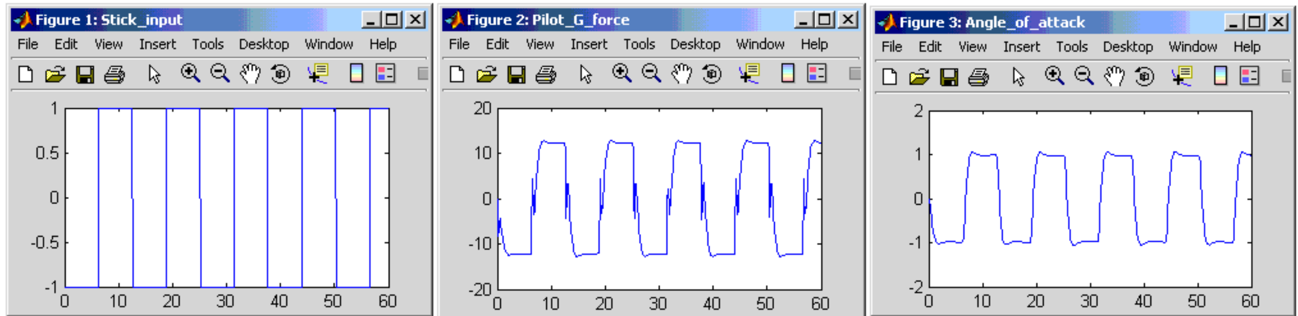
```
** starting the model **
** created f14rtw.mat **
```

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

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

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

Grand total is 5409 elements using 43272 bytes
```

7 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))
```



Comparing Numerical Results of the Simulation and the Generated Program

Your Simulink simulations and the generated code should produce nearly identical output. (Note that for non-`double` signals, results should be written using the `Structure` or `Structure with Time` format. For details, see the description of the To Workspace block in the Simulink documentation.)

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 results achieved by Real-Time Workshop.

Comparing `Angle_of_attack` (simulation output) to `rt_Angle_of_attack` (generated program output) produces

```
max(abs(rt_Angle_of_attack-Angle_of_attack))
ans =
1.0e-015 *
          0     0.3331
```

Comparing `Pilot_G_force` (simulation output) to `rt_Pilot_G_force` (generated program output) produces

```
max(abs(rt_Pilot_G_force-Pilot_G_force))
1.0e-013 *
          0     0.6395
```

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
- Run-time 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.

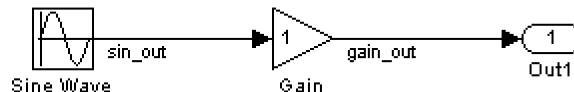
For the same reasons, your comparison results might not be identical to those above.

A First Look at Generated Code

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

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 3-35, 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 From the **View** menu, select **Model Explorer**. The **Model Explorer** appears.

5 Select **Real-Time Workshop** on the center pane, and click the **Solver** tab. The **Solver** pane appears at the right.

6 Enter the following parameter values on the **Solver** pane:

   Solver options: set **Type** to `Fixed-step`. Select the `ode5 (Dormand-Prince)` solver algorithm.

   Leave the other **Solver** pane parameters set to their default values.

7 Click **Apply**.

8 Click the **Data Import/Export** tab and make sure all check boxes are cleared.

9 Click **Apply** if you made any changes.

10 Click the **Real-Time Workshop** in the center pane, and select the **Generate code only** option at the bottom. This option causes Real-Time Workshop to generate code without invoking `make` to compile and link the code. This is convenient for this exercise, as it focuses on the generated source code. Note that the caption on the **Build** button changes to **Generate code**.

   Also, make sure that the default generic real-time (GRT) target is selected (system target file is `grt.tlc`).

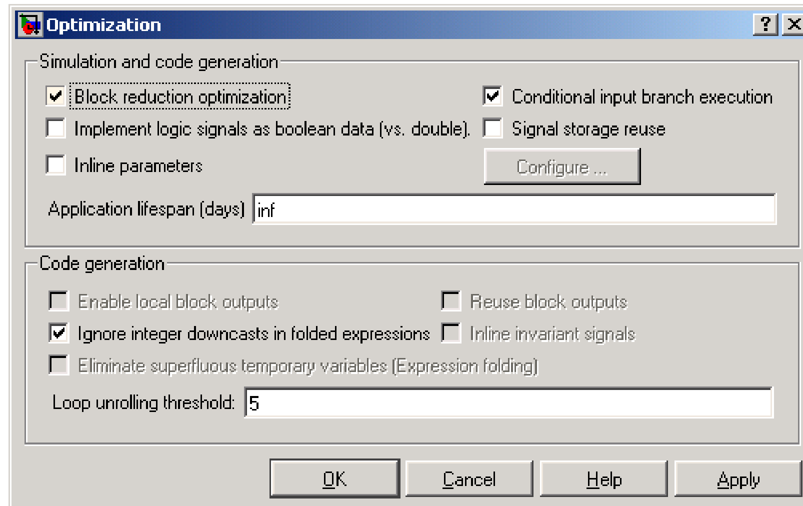
11 Click **Apply**.

12 Save the model.

Generating Code Without Buffer Optimization

When the block I/O optimization feature is enabled, Real-Time Workshop uses local storage for block outputs wherever possible. In this exercise, you disable this option 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.



2 Click **Apply**.

3 Click **Real-Time Workshop** in the center pane, and click the **Generate code** button.

4 Because the **Generate code only** option was selected, Real-Time Workshop does not invoke your make utility. The code generation process ends with this set of messages:

```
### Writing header file example_types.h
.
### Writing header file example.h
### Writing source file example.c
### Writing header file example_private.h
### Writing header file rtmodel.h
.
### Writing source file example_data.c
### Writing header file rt_nonfinite.h
### Writing source file rt_nonfinite.c
### TLC code generation complete.
### Creating project marker file: rtw_proj.tmw
### Creating example.mk from N:\matlab\rtw\c\grt\grt_lcc.tmf
### Successful completion of Real-Time Workshop build procedure
for model: example
```

5 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. Open this file in the MATLAB editor:

```
edit example_grt_rtw\example.c
```

6 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 documentation for a complete discussion of how Real-Time Workshop 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:

```
BlockIO_example example_B;
```

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

```
typedef struct _BlockIO_example {
  real_T sin_out;
  real_T gain_out;
} BlockIO_example;
```

The output code (without comments) accesses fields of this global structure, as shown below:

```
static void example_output(int_T tid)
```

```
{

  example_B.sin_out = example_P.SineWave_Amp *
    sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
    example_P.SineWave_Phase) + example_P.SineWave_Bias;

  example_B.gain_out = example_B.sin_out * example_P.Gain_Gain;

  example_Y.Out1 = example_B.gain_out;
}
```

7. 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.

In the steps below, you turn buffer optimizations on and observe how these optimizations improve the code.

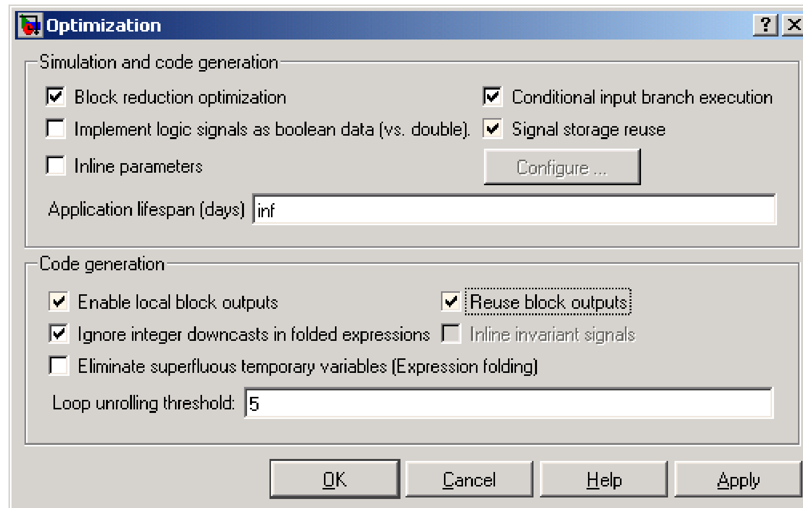
Generating Code with Buffer Optimization

Enable signal buffer optimizations and regenerate the code as follows:

1. Click **Optimization** in the center pane. The **Optimization** pane appears at the right. Select the **Signal storage reuse** option.

2. Note that three controls become enabled in the **Code generation** section below the check box, **Enable local block outputs**, **Reuse block outputs** and **Eliminate superfluous temporary variables (Expression**

folding). Make sure that the first two options are selected, and **Eliminate superfluous temporary variables (Expression folding)** is not selected, 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 block output optimizations.

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

4. Click **Generate Code**, or type **Ctrl+B** in the model window to generate code again.

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

5. Edit `example_grt_rtw/example.c` and examine the function `example_output`.

With buffer optimizations enabled, the code in `example_output` reuses temporary buffers with local scope, `rtb_sin_out` and `rtb_gain_out`, rather than assigning global buffers to each input and output.

```
static void example_output(int_T tid)
```

```
{

  real_T rtb_sin_out;
  real_T rtb_gain_out;

  rtb_sin_out = example_P.SineWave_Amp *
    sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
    example_P.SineWave_Phase) + example_P.SineWave_Bias;

  rtb_gain_out = rtb_sin_out * example_P.Gain_Gain;

  example_Y.Out1 = rtb_gain_out;
}
```

This code is more efficient in terms of memory usage. The efficiency improvement gained by enabling **Enable local block outputs** and **Reuse block outputs** would be more significant in a large model with many signals.

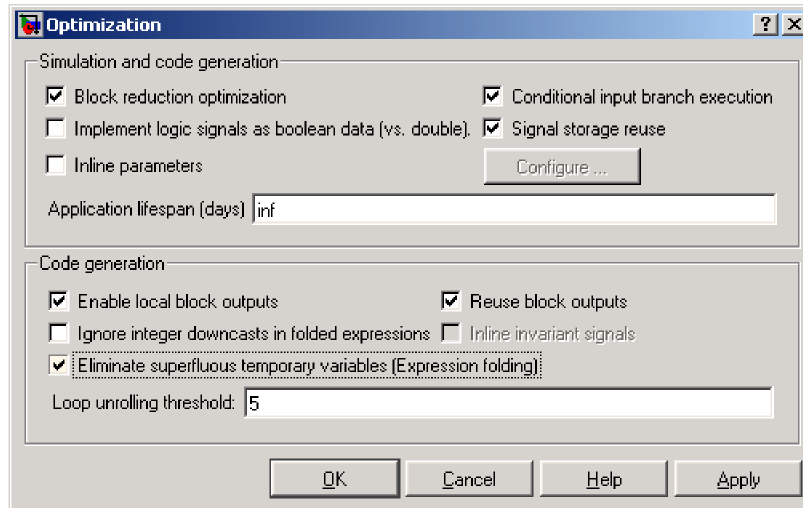
Further Optimization: Expression Folding

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

2. Select the **Eliminate superfluous temporary variables (Expression folding)** option.



3. Make sure that **Signal storage reuse** is still selected, as shown above.

4. Click **Apply**.

5. Click the **Generate code** button. As before, Real-Time Workshop generates code in the `example_grt_rtw` directory.

6. Edit `example_grt_rtw/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 temporary 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.

```
static void example_output(int_T tid)
{

  real_T rtb_sin_out;
```

```
    rtb_sin_out = example_P.SineWave_Amp *
      sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
      example_P.SineWave_Phase) + example_P.SineWave_Bias;

    example_Y.Out1 = rtb_sin_out * example_P.Gain_Gain;
  }
```

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

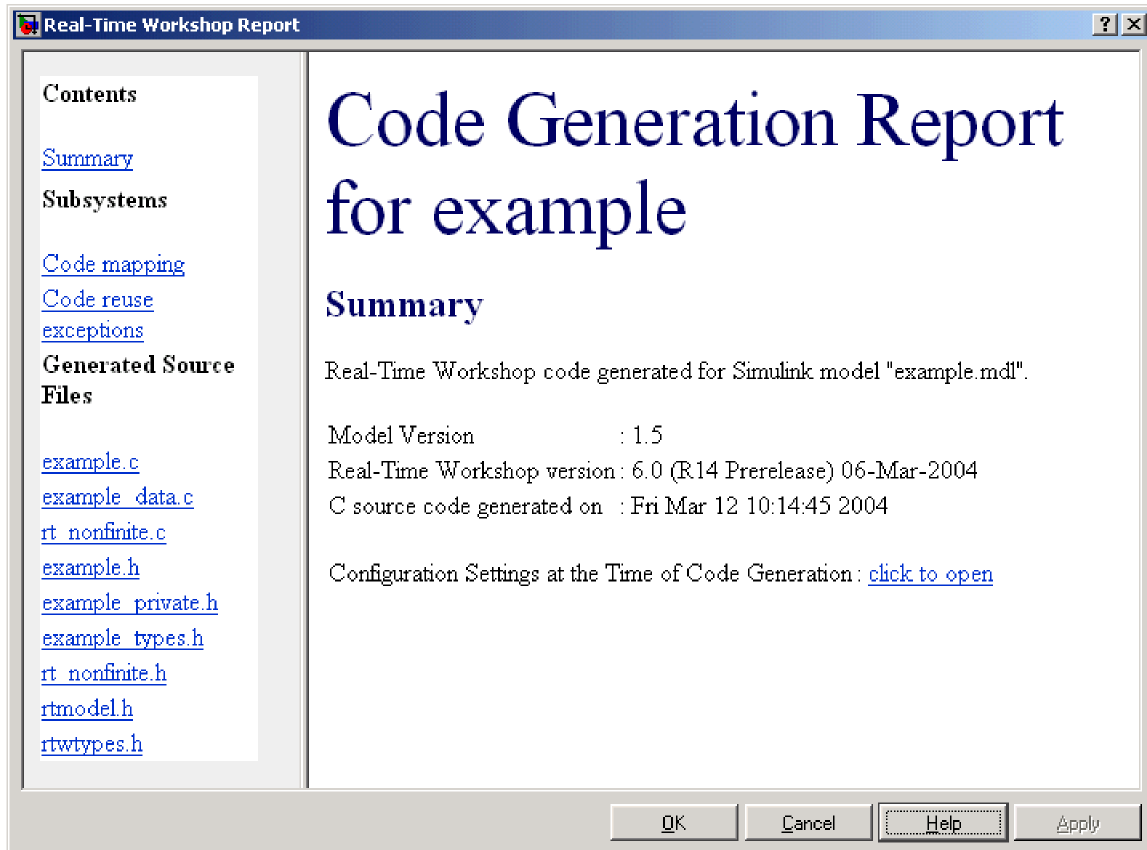
When the **Generate HTML report** option under the **Real-Time Workshop General** tab is selected, a navigable summary of source files is produced when the model is built. See the figure below.

Enabling this option causes Real-Time Workshop 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 after code generation completes** option (which is enabled by selecting **Generate HTML report**) is also selected, the HTML summary and index are automatically loaded into a new browser window and 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 Real-Time Workshop Embedded Coder).

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 3-50 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

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:

- “Setting Up the Model” on page 3-38
- “Building the Target Executable” on page 3-40
- “Running the External Mode Target Program” on page 3-43
- “Tuning Parameters” on page 3-48

The example presented uses the generic real-time target, and does not require any hardware other than the computer on which you run Simulink and Real-Time Workshop. 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, `ext_example`, and a directory called `ext_mode_example` to store the model and the generated executable:

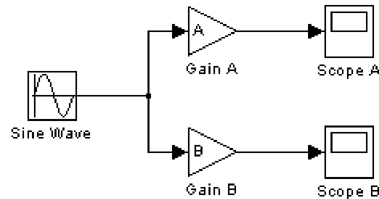
1 Create the directory from the MATLAB command line by typing

```
mkdir ext_mode_example
```

2 Make `ext_mode_example` your working directory:

```
cd ext_mode_example
```

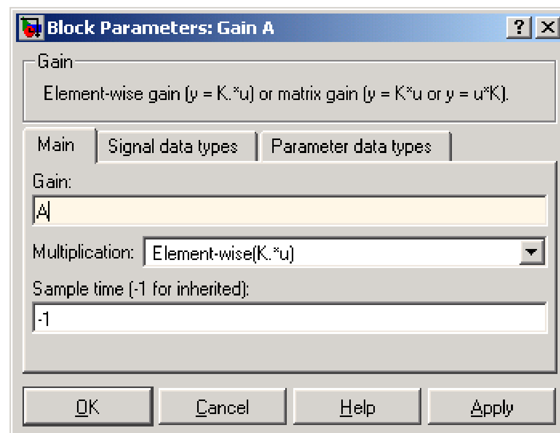
3 Create a model in Simulink with a Sine Wave block for the input signal, two Gain blocks in parallel, and two Scope blocks. The model is shown below. Be sure to label the Gain and Scope blocks as shown, so that subsequent steps will be clear to you.



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` as shown below.

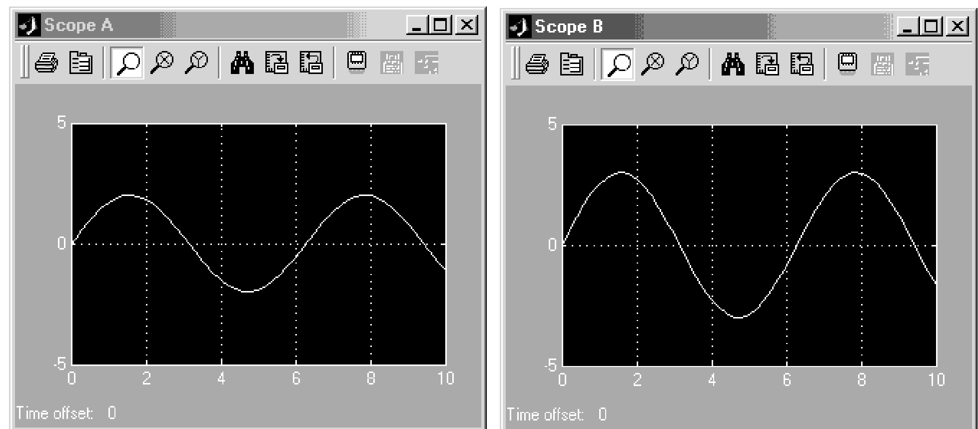


6 Similarly, open Gain block B and set its **Gain** parameter to the variable `B`.

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 3-48.

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.

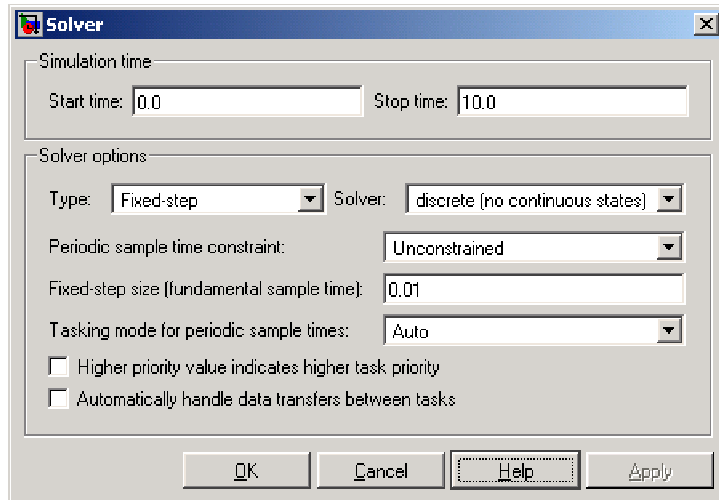


8 From the **File** menu, choose **Save As**. Save the model as `ext_example.mdl`.

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 the **Model Explorer** from the **View** menu. Click **Solver** in the center pane, set the **Solver type** to `Fixed-step`, and select the `discrete (no continuous states)` for the solver algorithm. Set **Fixed step size** to `0.01`. Leave the other parameters at their default values. The dialog box appears below; note that after you click **Apply**, the controls you changed again have a white background color.



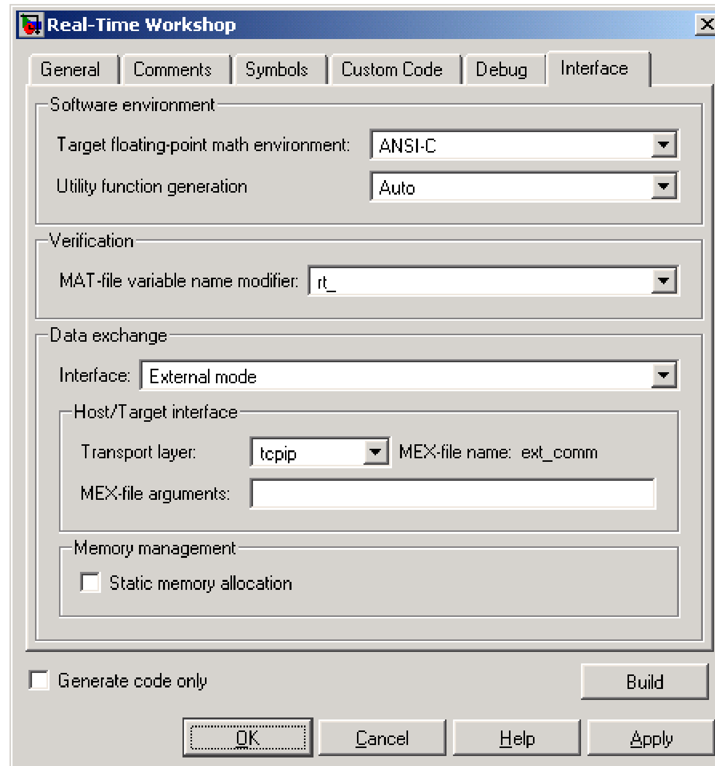
2 Click **Data Import/Export** in the center pane, and clear the **Time** and **Output** check boxes. In this exercise, data is not logged to the workspace or to a MAT-file.

3 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 this in this exercise.

4 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.

5 In the **Interface** pane, select `External mode` from the **Interface** 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**.

6 Set the **Transport layer** menu in the **Host/Target interface** section to `tcpip`. The dialog box now appears as follows:



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 by Real-Time Workshop. 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 exercise uses the default arguments. Leave the **MEX-file arguments** field blank.

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

8. Save the model.

9. Click **Real-Time Workshop** in the center pane of the **Model Explorer**, and 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: ext_example
```

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

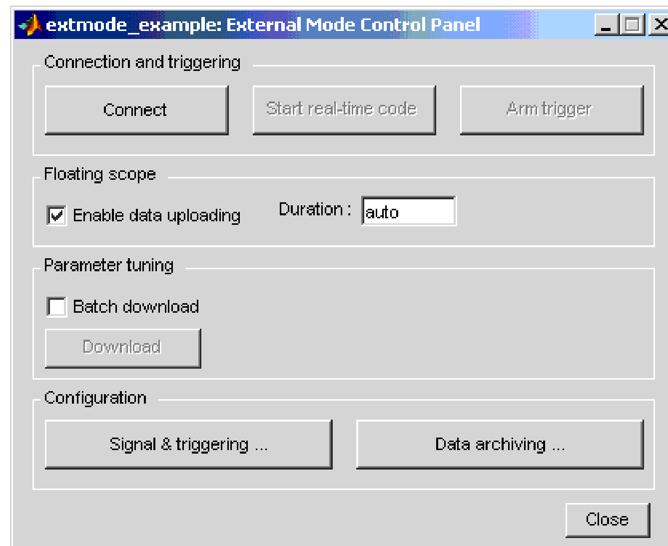
Running the External Mode Target Program

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

The **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 exercise you observe and use the default settings of the **External Signal & Triggering** dialog box.

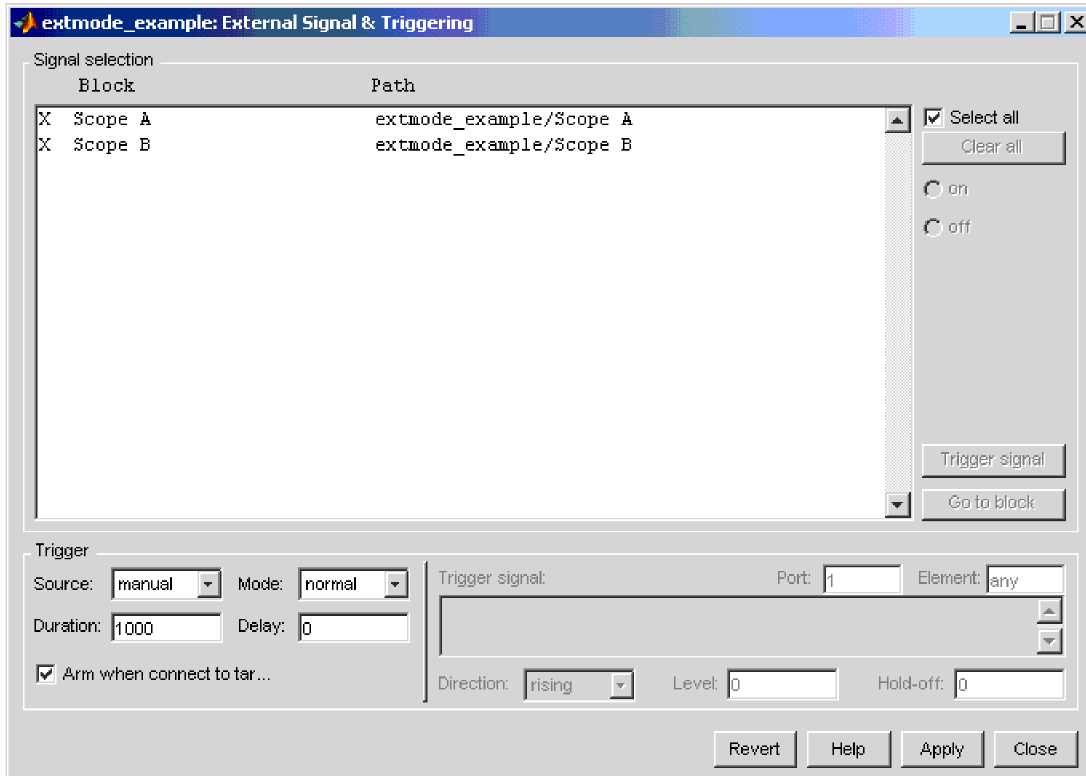
1. From the **Tools** menu of the block diagram, select **External Mode Control Panel**, which lets you configure signal monitoring and data archiving. It also lets you connect to the target program and start and stop execution of the model code.



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.

  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 `ext_example`.



2 In the **External Mode Control Panel**, click the **Signal & triggering** button.

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 connect 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 Windows platforms, you can also use the "bang" command (!) in the MATLAB console (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 there is no activity in the command prompt 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 will run until the target program receives a stop message from Simulink.

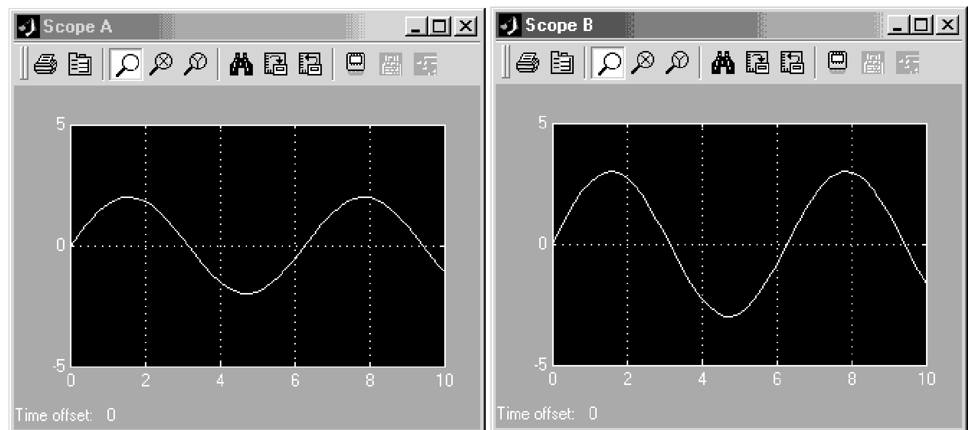
The `-w` switch instructs the target program to enter a wait state until it receives a **Start real-time code** message from the host. This switch is required if you want to view data from time step 0 of the target program execution, or if you want to modify parameters before the target program begins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

   You can also enter gain values directly into the Gain blocks. To do this,

3. Open the Block Parameters dialog box for Gain block A or B in the model.

4. 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.

5 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

The Model block, introduced in Version 6 of Simulink, enables 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. Real-Time Workshop generates code for referenced models in a slightly different way than for top models and standalone models, and generates different code than Simulink does when it simulates them. Follow this tutorial to learn how Simulink and Real-Time Workshop handle model blocks.

In this exercise you create a subsystem in an existing model, convert it to a model, call it from the top model via a Model block, and generate code for both models. You accomplish much of this automatically with a demo function called `sl_convert_to_model_reference`. You also explore the generated code and the project directory using the **Model Explorer**.

Create and Configure A Subsystem Within the vdp Model

In the first part of this exercise, you define a subsystem for the `vdp` demo model, set configuration parameters for the model, and use the `sl_convert_to_model_reference` function to convert it into two new model — the top model (`vdptop`) and a referenced model containing the subsystem you created (`vdpsub`):

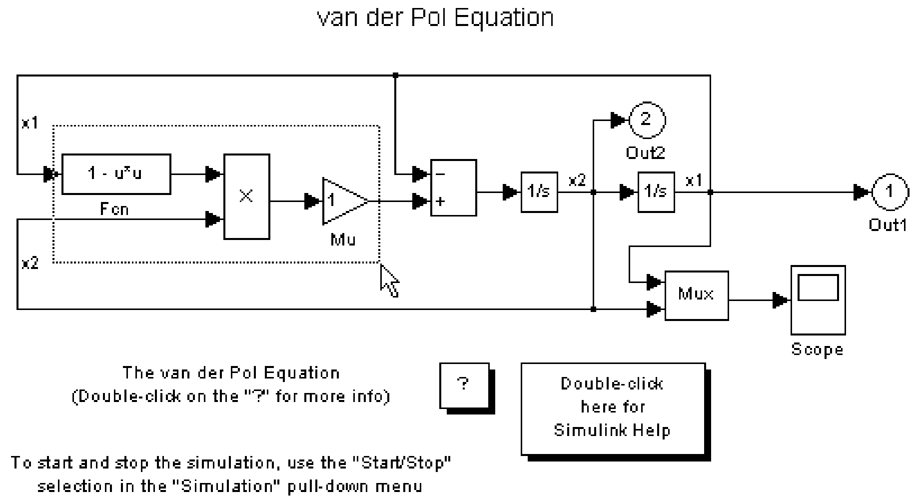
1 In MATLAB, create a new working directory wherever you want to work and `cd` into it:

```
mkdir tutorial6
cd tutorial6
```

2 Open the `vdp` demo model:

```
vdp
```

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



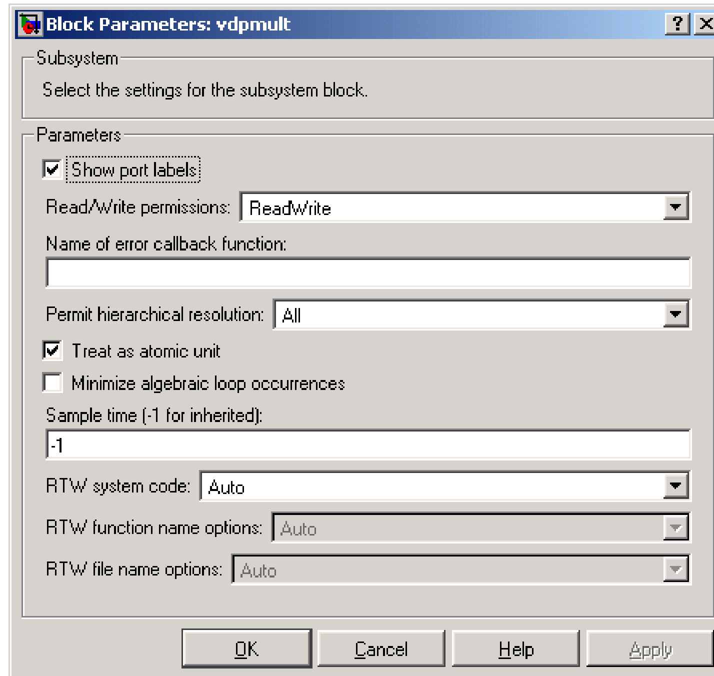
4 Right-click any of the three highlighted blocks and select **Create Subsystem** from the context menu. You can also select **Create Subsystem** from the **Edit** menu or type **Ctrl+G** instead of right-clicking.

5 If the new Subsystem block is not where you want it, move it to a preferred location. Rename the block `vdpmult`.

6 Right-click the vdpmult block and select **SubSystem Parameters**.

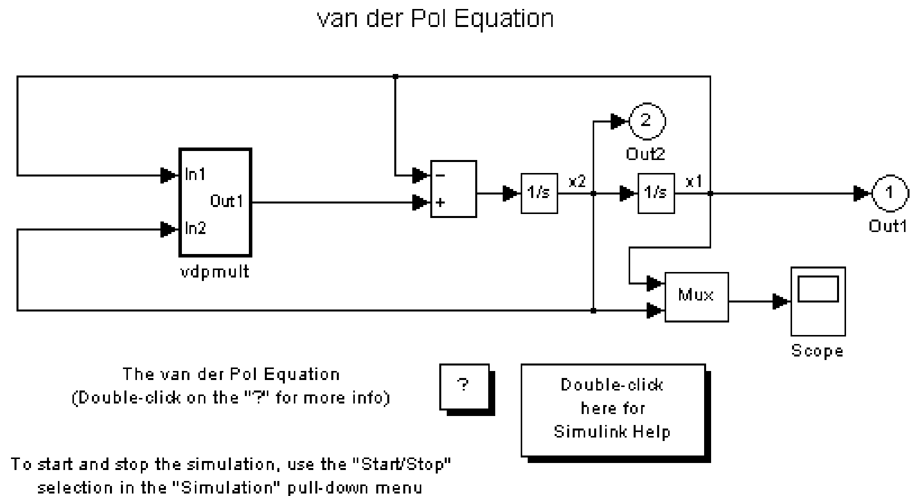
A **Block Parameters** dialog box appears.

7 In the **Block Parameters** dialog box, select the **Treat as atomic unit** check box, click **Apply**, as shown below, and then click **OK** to dismiss the dialog box.



The border of the `vdpmult` subsystem thickens to indicate that it is now atomic. An atomic subsystem executes as a unit rather than having its blocks executed at different stages during simulation. This makes it possible to abstract subsystems as standalone models and as functions in generated code.

The block diagram now looks like this.



8 Open the **Model Advisor** from the **View** menu.

The next four steps bypass certain model reference error checks, to minimize the number of times code must be generated later on.

9 Select the **Solver** node in the center pane and then change the Solver **Type** to `Fixed-step`. Click **Apply**. You must use fixed-step solvers when generating code, although referenced models can use different solvers than top models.

10 Click**Optimization** in the center pane, and then select the **Inline parameters** option.

11 Click **Diagnostics** in the center pane and then click the **Data Integrity** tab. Change the setting for **Signal resolution control** to `Use local settings`, and click **Apply**.

12 Select **Model Referencing** in the center pane. Change the setting for **Rebuild options** to `If any changes in known dependencies detected`, and click **Apply**.

13 On the model, choose **File > Save as**, and navigate to your working directory. Save the model as vdptop there, and then close it.

Convert the Model to Use Model Reference

In this portion of the exercise, you transform the vdptop model into two new models: a top-level model and a referenced model for the vdpmult subsystem. You do this using an M-function provided as a demo. You can run the demo by typing

```
mdlref_conversionscript
```

The function that converts models with subsystems to new models is named sl_convert_to_model_reference. To learn how this function is used, you can type

```
help sl_convert_to_model_reference
```

In this exercise you only supply the first two calling arguments (a model name and a target directory) and no return arguments:

1 Run the conversion utility by typing at the MATLAB prompt

```
sl_convert_to_model_reference('vdptop','.')
```

This converts the vdptop model and places the resulting files in the current directory. The converter prints a number of progress messages to the console, and when successful, terminates with

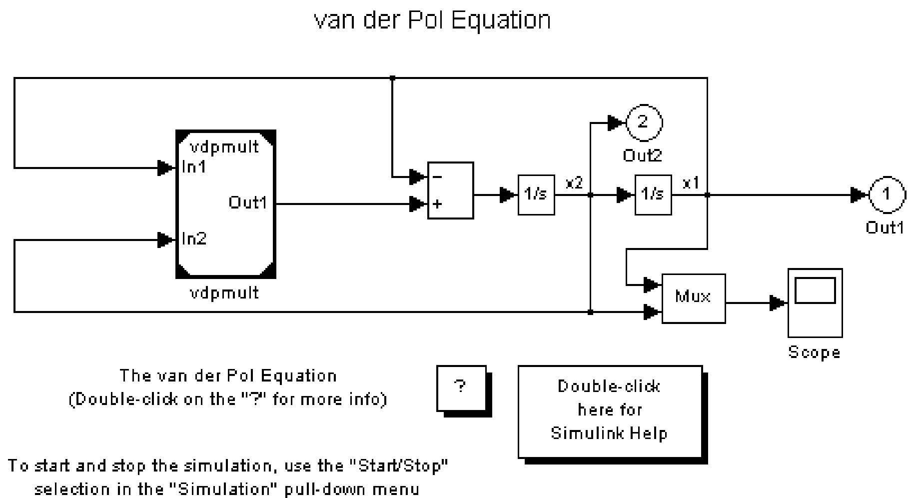
```
ans =
      1
```

Note that the model being converted need not be open. The files in your working directory now consist of the following.

File	Description
vdptop.mdl	The modified vdp model you saved

File	Description
vdptop_converted.mdl	Top-level model created by the M-function that contains a Model block where the vdpmult subsystem was
vdpmult.mdl	Model created for the vdpmult subsystem
vdpmult_msf.dll	Static library file (on Windows) that executes when the vdptop model calls what was formerly the vdpmult subsystem
/slprj	Project directory for generated model reference code

2 Type `vdptop_converted` to open the converted top-level model. Open the Scope if it is not visible, and click **Start** or choose **Start** from the **Simulation** menu. The model calls the `vdpmult_msf` simulation target to simulate. The model and its output look like this.



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 exercise. For more information on project directories, see "Working with Project Directories" on page 3-62.

Generate Model Reference Code for a GRT Target

The `sl_convert_to_model_reference` M-function took care of creating the model and the simulation target files for the vdpmult subsystem. In this part of the exercise, you generate code for that model and the `vdptop` model, and run the executable you create:

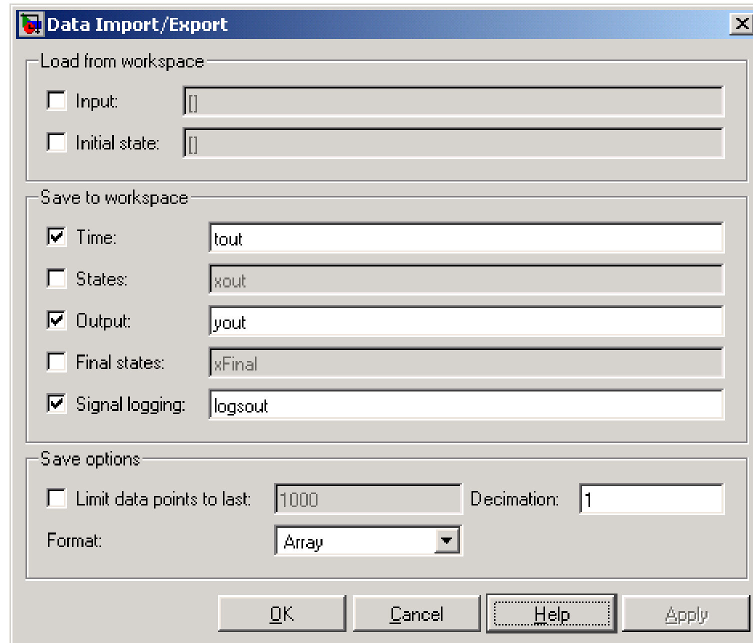
1. Verify that you are still working in the `tutorial6` directory.

2. If the model `vdptop_converted` is not open, open it. Make sure it is the active window.

3. Choose **Model Explorer** from the **View** menu.

   The **Model Explorer** view for `vdptop_converted` opens.

4. Select **Data Import/Export** in the center pane. Select **Time:** and **Output:** in the **Save to workspace** section of the **Data Import/Export** pane, then click **Apply**. The pane looks like this.



This instructs the `vdptop_converted` model (and later its executable) to log time and output data to MAT-files for each time step.

5 Generate GRT code (the default) and an executable for the top model and the referenced model. To start the build process, do one of the following:

- Select the **Real-Time Workshop** in the center pane and then click the **Build** button.
- Choose **Real-Time Workshop > Build Model** from the model's **Tools** menu.
- Type **Ctrl+B**.

Real-Time Workshop generates and compiles code. The current directory now contains two new files:

File	Description
vdptop_converted.exe	The executable created by Real-Time Workshop
/vdptop_converted_grt_rtw	The Real-Time Workshop build directory, containing generated code for the top model

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

6 Use the **Model Explorer** to inspect the newly created build directory, vdptop_converted_grt_rtw, in your current directory. Choose **Model Explorer** from the **View** menu.

7 In Model Explorer's **Model Hierarchy** pane, expand the node for the vdptop_converted model.

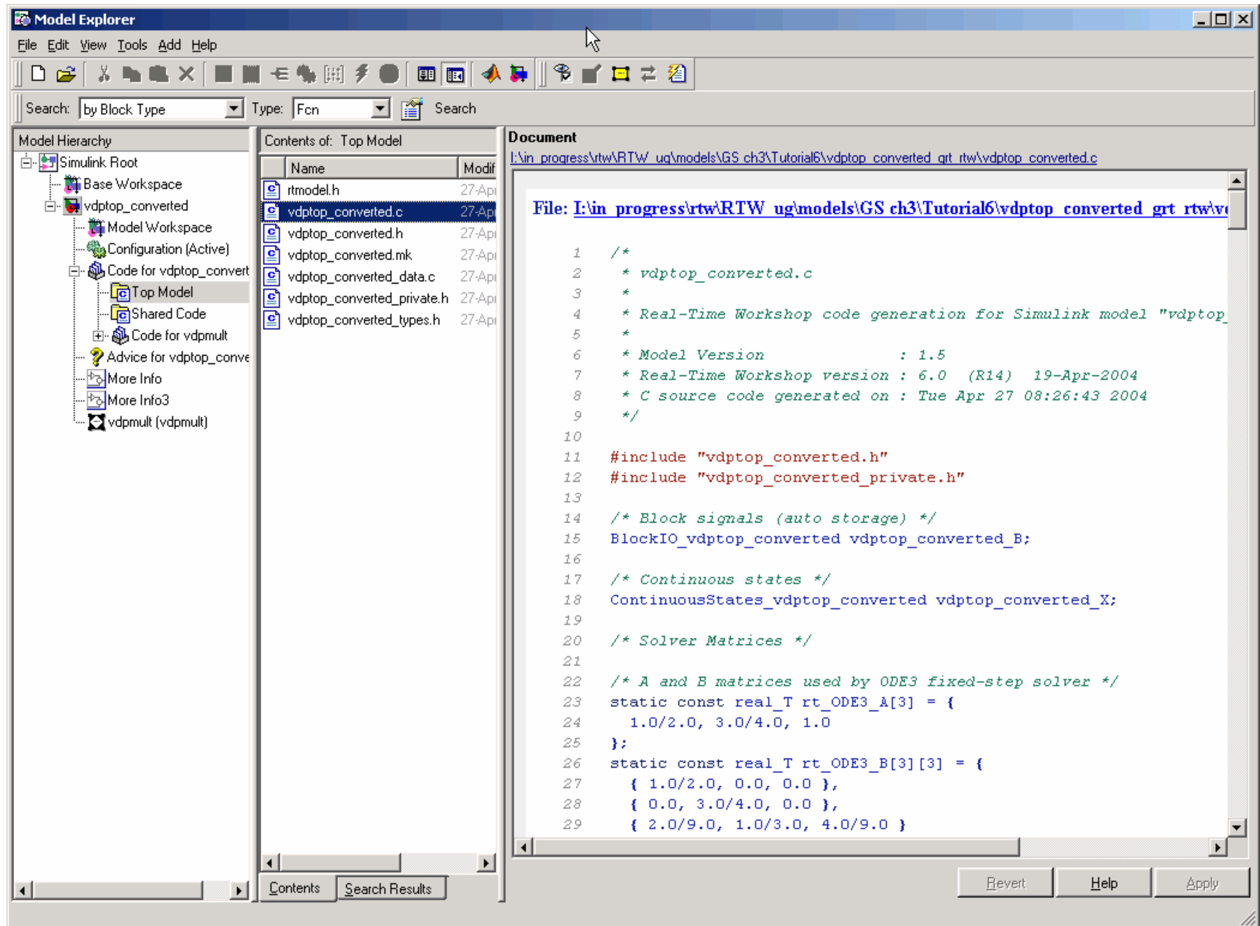
8 Click the plus sign in front of the node named Code for vdptop_converted to expand the node.

9 Click the Top Model node that appears directly under the Code for vdptop_converted node.

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

```
vdptop_converted.c
vdptop_converted_data.c
rtmodel.h
vdptop_converted_types.h
vdptop_converted.h
vdptop_converted_private.h
vdptop_converted.mk
```

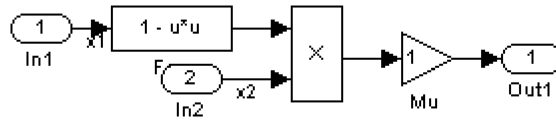
You can browse code in any of these files by selecting a file of interest in the **Contents** pane. The code for the file you select appears in the pane to the right. The picture below illustrates viewing code for vdptop_converted.c.



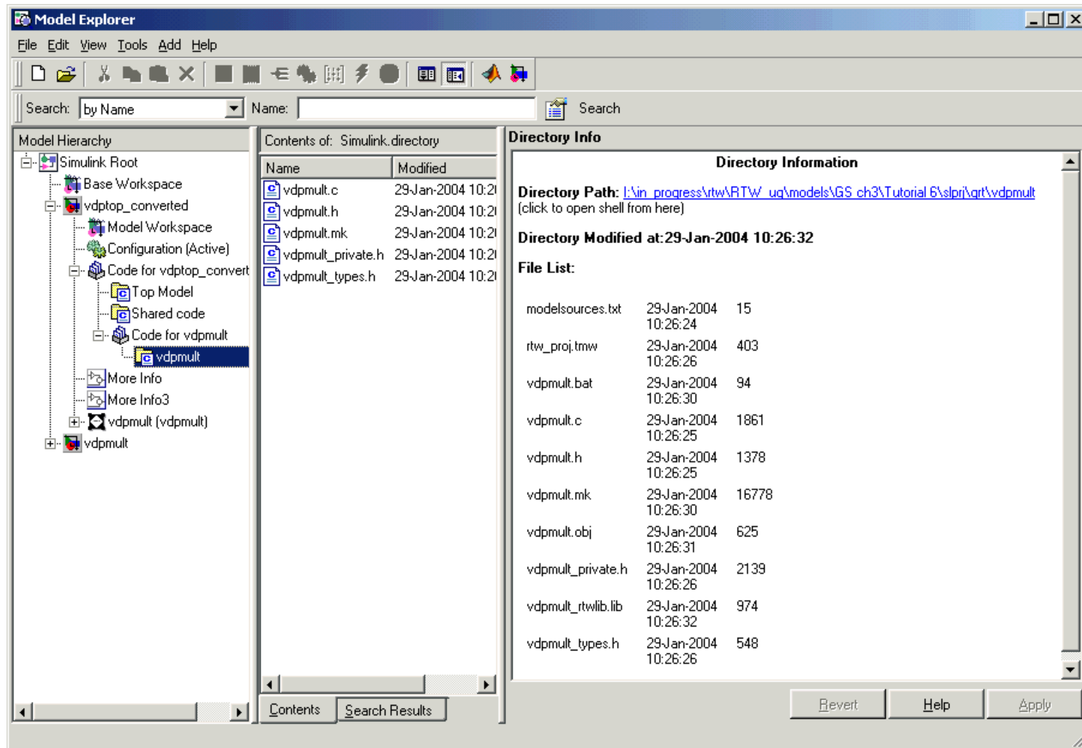
To view a model's generated code in **Model Explorer**, the model must be open. To open a file in a text editor, click a filename, and then click the hyperlink that appears in the gray area at the top of the **Dialog** pane.

10 Examine code for `vdpmult.mdl`. In the **Model Hierarchy** pane of **Model Explorer**, right-click the Model block icon labeled `vdpmult (vdpmult)` and select `Open Model 'vdpmult'` from the menu that appears.

The block diagram for vdpmult opens in a separate window, and a new node named vdpmult appears in the **Model Hierarchy** pane. The model looks like this.



11 A **Model Explorer** node named `Code for vdpmult` also appears under the node `Code for vdptop_converted`. Expand this node by clicking its plus sign, and then select `vdpmult` below it. **Model Explorer** now shows the files for the referenced model (which are in the `/slprj/grt` directory).



12 Click `vdpmult.c` in the **Contents** pane to view the model's output and initialization functions in the pane to the right. Here is the code for the `vdpmult` output function:

```
/* Output and update for referenced model: 'vdpmult'*/
void mr_vdpmult(const real_T *rtu_0, const real_T *rtu_1,
                real_T *rty_0, RT_MODEL_vdpmult *vdpmult_M)
{

  /* local block i/o variables */

  /* Gain: '<Root>/Mu' incorporates:
   *  Fcn: '<Root>/Fcn'
   *  Product: '<Root>/Product'
   */
  (*rty_0) = (1.0 - (*rtu_0) * (*rtu_0)) * (*rtu_1);
```

```
}
```

Note the function's prefix, mr_; this indicates that it comes from a referenced model, which also uses inlined parameters. Here is generated GRT code for the vdpmult subsystem in the vdptop model; the **RTW system code** property has been set to Reusable function.

```
/* Output and update for atomic system: '<Root>/vdpmult' */
void vdptop_vdpmult(real_T rtu_In1, real_T rtu_In2,
                    rtB_vdptop_vdpmult *localB)
{

  /* local block i/o variables */

  /* Gain: '<S1>/Mu' incorporates:
   *  Fcn: '<S1>/Fcn'
   *  Product: '<S1>/Product'
   */
  localB->Mu = (1.0 - rtu_In1 * rtu_In1) * rtu_In2;
}
```

The reusable function is prefixed with the model name. You can modify the construction of such identifiers when using ERT and ERT-derived targets.

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.

Glossary

application modules

With respect to Real-Time Workshop program architecture, these are collections of programs that implement functions carried out by the system-dependent, system-independent, and application components.

atomic subsystem

Subsystem whose blocks are executed as a unit before moving on. Conditionally executed subsystems are atomic, and atomic subsystems are nonvirtual. Unconditionally executed subsystems are virtual by default, but can be designated as atomic. Real-Time Workshop can generate reusable code only for nonvirtual subsystems.

base sample rate

Fundamental sample time of a model; in practice, limited by the fastest rate at which a processor's timer can generate interrupts. All sample times must be integer multiples of the base rate.

block I/O structure (model_B)

Global data structure for storing block output signals. The number of block output signals is the sum of the widths of the data output ports of all nonvirtual blocks in your model. By default, Simulink and Real-Time Workshop try to reduce the size of the `model_B` structure by reusing the entries in the `model_B` structure and making other entries local variables.

block target file

File that describes how a specific Simulink block is to be transformed to a language such as C, based on the block's description in the Real-Time Workshop file (`model.rtw`). Typically, there is one block target file for each Simulink block.

code reuse

Optimization whereby code generated for identical nonvirtual subsystems is collapsed into one function that is called for each subsystem instance with appropriate parameters. Code reuse, along with *expression folding*, can dramatically reduce the amount of generated code.

configuration

Set of attributes for a model which defines parameters governing how a model simulates and generates code. A model can have one or more such configuration sets, and users can switch between them to change code generation targets or to modify the behavior of models in other ways.

configuration component

Named element of a configuration set. Configuration components encapsulate settings associated with the **Solver**, **Data Import/Export**, **Optimization**, **Diagnostics**, **Hardware Implementation**, **Model Referencing**, and **Real-Time Workshop** panes in the Configuration Parameters dialog box. A component may contain subcomponents.

embedded real-time (ERT) target

Target configuration that generates model code for execution on an independent embedded real-time system. Requires Real-Time Workshop Embedded Coder.

expression folding

Code optimization technique that minimizes the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. It can dramatically improve the efficiency of generated code, achieving results that compare favorably with hand-optimized code.

file extensions

The table below lists the file extensions associated with Simulink, the Target Language Compiler, and Real-Time Workshop.

Extension	Created by	Description
`.c` or `.cpp`	Target Language Compiler	The generated C or C++ code
`.h`	Target Language Compiler	C/C++ include header file used by the `.c` or `.cpp` program

Extension	Created by	Description
.mdl	Simulink	Contains structures associated with Simulink block diagrams
.mk	Real-Time Workshop	Makefile specific to your model that is derived from the template makefile
.rtw	Real-Time Workshop	Intermediate compilation (*model*.rtw) of a .mdl file used in generating C or C++ code
.tlc	The MathWorks and Real-Time Workshop users	Target Language Compiler script files that Real-Time Workshop uses to generate code for targets and blocks
.tmf	Supplied with Real-Time Workshop	Template makefiles
.tmw	Real-Time Workshop	Project marker file inside a build directory that identifies the date and product version of generated code

generic real-time (GRT) target

Target configuration that generates model code for a real-time system, with the resulting code executed on your workstation. (Execution is not tied to a real-time clock.) You can use GRT as a starting point for targeting custom hardware.

host system

Computer system on which you create and may compile your real-time application. Also referred to as emulation hardware.

inline

Generally, this means to place something directly in the generated source code. You can inline parameters and S-functions using Real-Time Workshop and the Target Language Compiler.

inlined parameters

(Target Language Compiler Boolean global variable: `InlineParameters`) The numerical values of the block parameters are hard-coded into the generated code. Advantages include faster execution and less memory use, but you lose the ability to change the block parameter values at run time.

inlined S-function

An S-function can be inlined into the generated code by implementing it as a `.tlc` file. The code for this S-function is placed in the generated model code itself. In contrast, noninlined S-functions require a function call to an S-function residing in an external MEX-file.

interrupt service routine (ISR)

Piece of code that your processor executes when an external event, such as a timer, occurs.

loop rolling

(Target Language Compiler global variable: `RollThreshold`) Depending on the block's operation and the width of the input/output ports, the generated code uses a `for` statement (rolled code) instead of repeating identical lines of code (flat code) over the signal width.

make

Utility to maintain, update, and regenerate related programs and files. The commands to be executed are placed in a *makefile*.

makefiles

Files that contain a collection of commands that allow groups of programs, object files, libraries, and so on, to interact. Makefiles are executed by your development system's make utility.

`model`.rtw
: Intermediate record file into which Real-Time Workshop compiles the blocks, signals, states, and parameters a model, which the Target Language Compiler reads to generate code to represent the model.

multitasking
: Process by which a microprocessor schedules the handling of multiple tasks. In generated code, the number of tasks is equal to the number of sample times in your model. *See also* pseudo multitasking.

noninlined S-function
: In the context of Real-Time Workshop, this is any C-MEX S-function that is not implemented using a customized `.tlc` file. If you create a C-MEX S-function as part of a Simulink model, it is by default noninlined unless you write your own `.tlc` file that inlines it.

nonreal-time
: Simulation environment of a block diagram provided for high-speed simulation of your model. Execution is not tied to a real-time clock.

nonvirtual block
: Any block that performs some algorithm, such as a Gain block. Real-Time Workshop generates code for all nonvirtual blocks, either inline or as separate functions and files, as directed by users.

pseudo multitasking
: On processors that do not offer *multitasking* support, you can perform pseudomultitasking by scheduling events on a fixed time-sharing basis.

real-time model data structure
: Real-Time Workshop encapsulates information about the root model in the real-time model data structure, often abbreviated as rtM. rtM contains global information related to timing, solvers, and logging, and model data such as inputs, outputs, states, and parameters.

real-time system
: Computer that processes real-world events as they happen, under the constraint of a real-time clock, and that can implement algorithms in dedicated hardware. Examples include mobile telephones, test and measurement devices, and avionic and automotive control systems.

Real-Time Workshop target

Set of code files generated by Real-Time Workshop for a standard or custom target, specified by a Real-Time Workshop configuration component. These source files can be built into an executable program that will run independently of Simulink. *See also* simulation target, configuration.

run-time interface

Wrapper around the generated code that can be built into a standalone executable. The run-time interface consists of routines to move the time forward, save logged variables at the appropriate time steps, and so on The run-time interface is responsible for managing the execution of the real-time program created from your Simulink block diagram.

S-function

Customized Simulink block written in C, Fortran, or M-code. Real-Time Workshop can target C-code S-functions as is or users can *inline* C-code S-functions by preparing TLC scripts for them.

simstruct

Simulink data structure and associated application program interface (API) that enables S-functions to communicate with other entities in models. Simstructs are included in code generated by Real-Time Workshop for noninlined S-functions.

simulation target

Set of code files generated for a model which is referenced by a Model block. Simulation target code is generated into `/slprj/sim` project directory in the working directory. Also an executable library compiled from these codes that implements a Model block. *See also* Real-Time Workshop Target.

single-tasking

Mode in which a model runs in one task, regardless of the number of sample rates it contains.

stiffness

Property of a problem that forces a numerical method, in one or more intervals of integration, to use a step length that is excessively small in relation to the smoothness of the exact solution in that interval.

system target file
: Entry point to the Target Language Compiler program, used to transform the Real-Time Workshop file into target-specific code.

target file
: File that is compiled and executed by the Target Language Compiler. The block and system target TLC files used specify how to transform the Real-Time Workshop file (`model.rtw`) into target-specific code.

Target Language Compiler (TLC)
: Program that compiles and executes system and target files by translating a `model.rtw` file into a target language by means of TLC scripts and template makefiles.

Target Language Compiler program
: One or more TLC script files that describe how to convert a `model.rtw` file into generated code. There is one TLC file for the target, plus one for each built-in block. Users can provide their own TLC files to inline S-functions or to wrap existing user code.

target system
: Specific or generic computer system on which your real-time application is intended to execute. Also referred to as embedded hardware.

targeting
: Process of creating software modules appropriate for execution on your target system.

task identifier (tid)
: In generated code, each sample rate in a multirate model is assigned a task identifier (`tid`). The `tid` is used by the model output and update routines to control the portion of your model that should execute at a given time step. Single-rate systems ignore the `tid`. *See also* base sample rate.

template makefile
: Line-for-line makefile used by a make utility. Real-Time Workshop converts the template makefile to a makefile by copying the contents of the template makefile (usually `system.tmf`) to a makefile (usually `system.mk`) replacing tokens describing your model's configuration.

virtual block

Connection or graphical block, for example a Mux block, that has no algorithmic functionality. Virtual blocks incur no real-time overhead as no code is generated for them.

work vector

Data structures for saving internal states or similar information, accessible to blocks that may require such work areas. These include state work (`rtDWork`), real work (`rtRWork`), integer work (`rtIWork`), and pointer work (`rtPWork`) structures. For example, the Memory block uses a real work element for each signal.

Examples

Use this list to find examples in the documentation.

Verification

“Data Logging” on page 3-15
“Code Verification” on page 3-21

Optimizations

"A First Look at Generated Code" on page 3-27

External Mode

"Working with External Mode Using GRT" on page 3-38

Model Reference

"Generating Code for a Referenced Model" on page 3-50

Index

A

acceleration of development process 1-7
application requirements 2-4

B

block target file 2-21
build directory
  contents of 2-31
  F14 example 3-14
  naming convention 2-30
  seeing files 3-13
build directory optional contents
  C-API files 2-31
  HTML report files 2-31
  `model.rtw` 2-31
  object files 2-31
  subsystem code modules 2-31
  TLC profiler files 2-31
build process
  files and directories created 2-24
  messages in MATLAB Command Window 3-13
  steps in 2-18

C

code generation for Simulink models 1-4
code generation tutorial 3-27
  product support for 1-18
code verification tutorial 3-21
code with buffer optimization 3-33
  efficiency 3-32
code with expression folding 3-33
code without buffer optimization 3-30
comments options 3-12
compilers
  Borland 1-13
  Intel 1-13
  LCC 1-14
  list of supported 1-15
  MEX 1-13
  Microsoft Visual C/C++ 1-14
  optimization settings 1-16
  supported on UNIX 1-16
  supported on Windows 1-15
  Watcom 1-14
configuration parameters 2-9
  different ways of displaying 3-6
  factory defaults 2-6
  impacts of settings 2-5
  questions to consider 2-5

D

data logging 3-15
  from generated code 3-23
  tutorial 3-15
  via Scope blocks
    example 3-21
debug options 3-10
debugging
  and configuration parameter settings 2-6
dialog boxes
  Block Parameters 3-39
  Configuration Parameters 2-4
  External Mode Control Panel 3-43
  External Signal and Triggering 3-45
  Model Explorer 2-9
directories
  build 3-4
  working 3-4
documentation
  online 1-21

E

efficiency
  and configuration parameter settings 2-6
executable

running 3-13
extensible make process 1-7
extensive model-based debugging support 1-5
external mode
building executable 3-40
control panel 3-43
definition of 1-4
model setup 3-38
parameter tuning 3-48
running executable 3-43
tutorial 3-38

F

F14 GRT code generation exercise 3-4
F14 model 3-3
files
generated, *see* generated files
fixed-step solver 3-5

G

generated files 2-24
`model` (UNIX executable) 2-27
`model.c` 2-25
`model_capi.c` 2-29
`model_capi.h` 2-29
`model_data.c` 2-27
`model_dt.h` 2-28
`model.exe` (PC executable) 2-27
`model.h` 2-26
`model.mdl` 2-24
`model.mk` 2-27
`model_private.h` 2-26
`model.rtw` 2-25
`model_targ_data_map.m` 2-28
`model_target_rtw` 2-30
`model_types.h` 2-27
`modelsources.txt` 2-28
`rt_nonfinite.c` 2-28
`rt_nonfinite.h` 2-28
`rt_sfcn_helper.c`, `rt_sfcn_helper.h` 2-29
`rtmodel.h` 2-27
`rtw_proj.tmw` 2-28
`rtwtypes.h` 2-28
`subsystem.c` 2-29
`subsystem.h` 2-29
generic real-time (GRT) target
tutorial 3-4

I

integration
Real-Time Workshop with Simulink 1-5
Real-Time Workshop with Stateflow 1-6
integration with Simulink 1-5
integration with Stateflow 1-6

L

large-scale modeling 1-4

M

make process 1-4
make utility 2-16
makefile 2-21
MAT-files
creating 3-18
loading 3-24
MATLAB 1-2
`model` (on UNIX) 2-27
Model Advisor 2-10
model compiling process 2-20
model explorer
viewing code in 3-58
model referencing
converting to 3-54
definition of 3-50
generating code 3-56

tutorial 3-50
model.bat 2-28
model.c 2-25
model_capi.c 2-29
model_capi.h 2-29
model_data.c 2-27
model_dt.h 2-28
model.exe 2-27
model.h 2-26
model.mdl 2-24
model.mk 2-27
model_private.h 2-26
model.rtw 2-25
model_targ_data_map.m 2-28
model_target_rtw 2-30
model_types.h 2-27
modelsources.txt 2-28

O

optimizations
expression folding 3-33
signal storage reuse 3-31
out-of-environment error message 1-15

P

parameters
setting correctly 3-5
pilot G Force plot 3-17
project directory 2-30
working with 3-62

R

rapid simulations 1-4
Real-Time Workshop
capabilities and benefits 1-4
components and features 1-3
demos 1-17
extensible make process 1-7
features 1-7
installing 1-12
integration with Simulink 1-5
integration with Stateflow 1-6
model-based debugging support 1-5
products supported by 1-18
report 3-35
software development with 1-7
third-party compiler support 1-13
typical workflow 2-2
Real-Time Workshop Report 3-35
related products 1-22
rt_nonfinite.c 2-28
rt_nonfinite.h 2-28
rt_sfcn_helper.c 2-29
rt_sfcn_helper.h 2-29
rtmodel.h 2-27
rtw_proj.tmw 2-28
rtwtypes.h 2-28
run-time interface modules 3-30

S

safety precautions
and configuration parameter settings 2-6
save to workspace options 3-16
Simulink 1-2
slprj 2-30
Stateflow 1-6
subsystem.c 2-29
subsystem.h 2-29
subsystems
converting to referenced models 3-54
treating as atomic units 3-51
symbols options 3-11
system target file 2-20

T

target

how to specify 3-7
Real-Time Workshop support 1-6
target file
block 2-21
system 2-20
Target Language Compiler
function library 2-20
generation of code by 2-20
TLC scripts 2-20
traceability
and configuration parameter settings 2-6
tuning parameters 3-48
tutorials
building generic real-time program 3-4
code generation 3-27
code verification 3-21
data logging 3-15
external mode 3-38
model referencing 3-50

U

UNIX compilers 1-16

V

variable-step solver 3-5

W

Windows compilers 1-15
working directory 2-30