Real-Time Workshop®

For Use with Simulink®

Modeling

Simulation

Implementation

Getting Started

Version 5

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

Printing History: July 2002 First printing New for Version 5 (Release 13)
October 2004 Online only Revised for Version 5.2 (Release 13SP2)

Contents

About This Guide

1 Introducing Real-Time Workshop

2 Building an Application

3 Working with Real-Time Workshop

A Glossary

About This Guide

Real-Time Workshop® builds applications from Simulink diagrams for prototyping, testing, and deploying real-time systems on a variety of target computing platforms. Users of Real-Time Workshop can direct it to generate source code that accommodates the compilers, input and output devices, memory models, communication modes, and other characteristics that their applications may require.

This guide summarizes the concepts, capabilities, user interface, and applications of Real-Time Workshop, to help you become productive with it as quickly as possible. It includes the following chapters:

- **Introducing Real-Time Workshop** — Presents an overview of installation procedures, Real-Time Workshop demos, available documentation, and descriptions of related products.
- **Building an Application** — Describes how Real-Time Workshop compiles models to construct stand-alone applications, what files are generated, and how they are organized.
- **Working with Real-Time Workshop** — Summarizes Real-Time Workshop concepts and terms, and provides a set of tutorials to get you started generating code right away.

This Getting Started guide is an introduction to the Real-Time Workshop documentation, which covers in detail technical topics such as code formats, rapid prototyping, communications, optimizations, and targeting. You may further customize code for targets and blocks by preparing Target Language Compiler scripts. The Target Language Compiler Reference Guide, which is also part of this documentation set, describes these advanced techniques.

1

Introducing Real-Time Workshop

We begin this guide 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?

Real-Time Workshop® is an extension of capabilities found in Simulink® and MATLAB® to enable rapid prototyping of real-time software applications on a variety of systems. Real-Time Workshop, along with other tools and components from The MathWorks, 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

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 create 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 a number of environments, including Tornado and DOS. The generic real-time and embedded real-time targets provide a framework for developing customized rapid prototyping or production target environments. In addition to the bundled targets, the 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.
- *Rapid Simulations* — Using Simulink Accelerator (part of the Simulink Performance Tools product), the S-function Target, or the Rapid Simulation Target, you can accelerate your simulations by 5 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, these targets apply many optimizations, such as eliminating ones and zeros in computations for filter blocks.

Capabilities and Benefits

Specific capabilities and benefits of Real-Time Workshop include

- **Code generator 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 data types.
  - Fixed-point capabilities of Real-Time Workshop 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).
  - Generated code is processor independent. The generated code represents your model exactly. A separate run-time interface is used to execute this code. We provide several example run-time interfaces as well as production run-time interfaces.
  - Supports any single or multitasking operating system, as well as "bare-board" (no operating system) environments.
  - The flexible scripting capabilities of the Target Language Compiler enable you to fully customize generated code.
  - Efficient code for S-functions (user-created blocks) can be crafted using Target Language Compiler instructions (called TLC scripts) and automatically integrated 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 parametric 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 validation. You can generate code for your model and create a stand-alone 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.
- **Rapid simulations**
  - 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.
  - Add-on targets (Real-Time Windows Target and xPC Target) for PC-based hardware are available from The MathWorks. These targets enable you to turn a PC with fast, high-quality, low-cost hardware into a rapid prototyping system.
  - 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.

In addition to the above benefits, the Real-Time Workshop Embedded Coder provides:

- Customizable, portable, and readable C code that is designed to be placed in a production embedded environment.
- More efficient code, because inlined S-functions are required and continuous time states are not allowed.
- Software-in-the-loop. With the Real-Time Workshop Embedded Coder, you can generate code for your embedded application and bring it back into Simulink for verification via simulation.
- Web-viewable code generation report, which describes in detail code modules, analyzes the generated code, and helps to identify code generation optimizations relevant to your program.
- Annotation of the generated code using the Description block property.
- Signal logging options and external parameter tuning, enabling easy interfacing of the generated code in your real-time system.

Accelerating Your Development Process

The MathWorks does not tell you how to work by suggesting or imposing a particular software design methodology. The MathWorks gives you the ability to simplify and accelerate most phases of software development, and at the same time to eliminate paperwork and other mundane tasks. Our tools lend themselves particularly well to the spiral design process shown below.

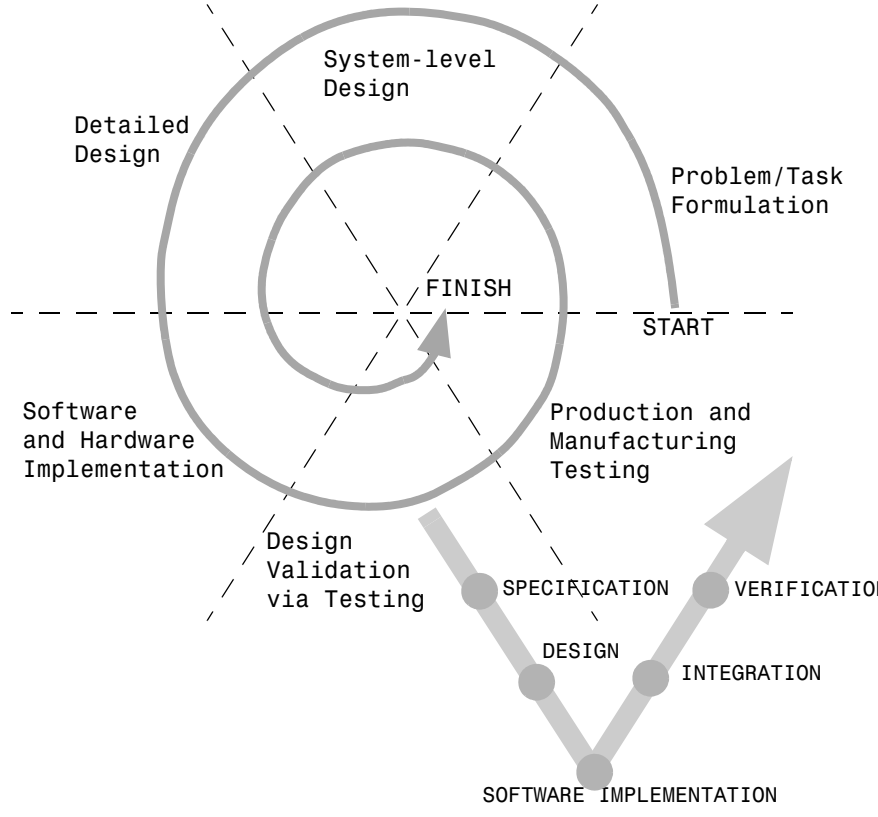


Figure 1-1: Spiral Design Process

When you work with tools from The MathWorks, *your model represents your understanding of your system*. This understanding is passed 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 one or more phases, because the same model and tools are used throughout.

A spiral design process iterates quickly between phases, enabling engineers to work on innovative features. To do this cost effectively, they 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 validate an algorithm 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. This helps compress the overall development cycle while increasing quality.

Simulink facilitates the first three phases described in Figure 1-1. You can build applications from built-in blocks from the Simulink and Stateflow® libraries, incorporate specialized blocks from the Communications, DSP, Nonlinear Control Design, and other MathWorks blocksets, and develop your own blocks by writing S-functions.

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

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

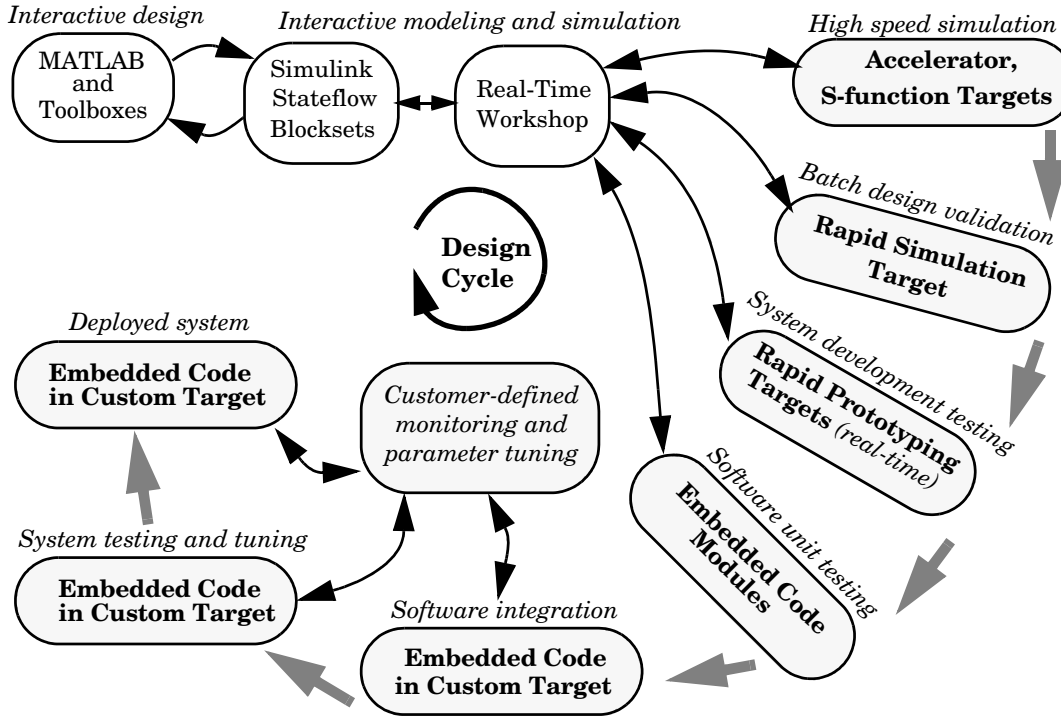


Figure 1-2: Roles of MathWorks Products in Software Development

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 helps with this process by enabling high-speed simulations via Simulink Accelerator (part of Simulink Performance Tools), and the S-function Target to componentize and speed up models.

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

Once you've tuned your model, you can move into system development testing by exercising your model on a rapid prototyping system such as the Real-Time Windows Target or the 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.

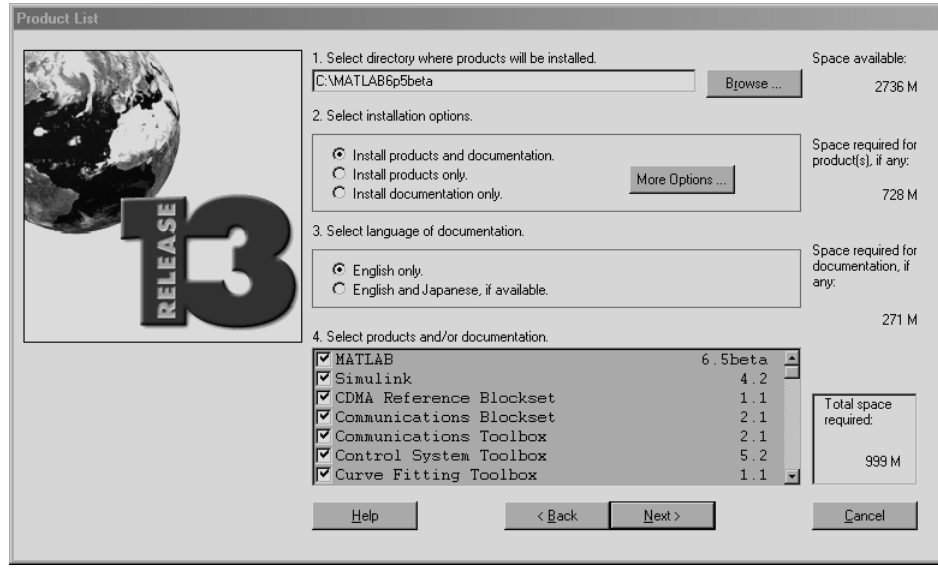
After your prototype system is created, you can use the Real-Time Workshop Embedded Coder to create code for deployment on your custom target. The signal monitoring and parameter tuning capabilities enable you to easily integrate the embedded code into a production environment equipped with debugging and upgrade capabilities.

Installing Real-Time Workshop

Your platform-specific MATLAB installation documentation provides all of the information you need to install the 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 identifies the products you are permitted to install and use.

As the installation process proceeds, it displays a dialog similar to the one below, letting you indicate which products to install.



In the product window you can only select for installation MATLAB products for which you are licensed.

Real-Time Workshop has certain product prerequisites that must be met for proper installation and execution.

Licensed Product	Prerequisite Products	Additional Information
Simulink	MATLAB 6.5 (Release 13)	Allows installation of Simulink.
Real-Time Workshop	Simulink 5 (Release 13)	Requires Borland C, LCC, Visual C/C++, or Watcom C compiler to create MATLAB MEX-files on your platform.
The Real-Time Workshop Embedded Coder	Real-Time Workshop 5	

If you experience installation difficulties and have Web access, connect to the MathWorks home page (`http://www.mathworks.com`). Use the resources found 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

Several of the 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. The following sections describe how to configure your system so that Real-Time Workshop can access your 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 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 on the **Advanced** tab, select **Environment**, and define BORLAND to be the path to your compiler.

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++

Define the environment variable MSDevDir to be

```
MSDevDir=<path to compiler>\SharedIDE          for Visual C/C++ 5.0
MSDevDir=<path to compiler>\Common\MSDev98     for Visual C/C++ 6.0
```

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, as of this printing, 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 Windows 95 or 98, add

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

to your `autoexec.bat` file.

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

Out-of-Environment Error Message

If you are receiving out-of-environment space error messages, you can 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. eWe have tested the Real-Time Workshop with these compilers on Windows.

Compiler	Versions
Borland	5.2, 5.3, 5.4, 5.5, 5.6
LCC	Use the version of LCC shipped with MATLAB.
Microsoft Visual C/C++	5.0, 6.0, 7.0
Watcom	10.6, 11.0 (see “Watcom” 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. `cc` is the default on all platforms except SunOS, where `gcc` is the default.

For further information, please 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, due to compiler defects, compiler optimizations applied to Real-Time Workshop generated code may 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 turning off optimizations. Please 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 running a set of demos we provide, and then inspecting code generated from these models. These demos illustrate a number of Real-Time Workshop features, though certainly not all of them. Note that one of the suites demonstrates features of the Real-Time Workshop Embedded Coder, which you will need in order to run those demos. Some of the following demos are set up to build ERT targets, but will default to GRT if Real-Time Workshop Embedded Coder is not installed.

If you are using the MATLAB Help browser to read this, you can launch the demos by clicking on the links in the **Command** column of Table 1-1.

Alternatively, you can access the demo suite by typing commands from the **Demo Command** column of the following table, at the MATLAB command prompt, as in this example:

```
rtwdemos
```

Table 1-1: Real-Time Workshop and Related Simulink Demos

Demo Command	Demo Topic
`rtwdemos`	Top-level demo containing buttons to launch specific Real-Time Workshop demos (double-click to activate)
`asyncdemo`	Simulate and generate code for single-, multi-, and asynchronous rate models
`sl_subsys_semantics`	Illustrates differences among types of subsystem, when and how to use them, and common mistakes.
`atomicdemo`	How to preserve the boundary of a virtual subsystem using reusable atomic subsystems
`condinputexec`	Illustrates how conditional input branch execution improves simulation and generated code efficiency
`cbdemo`	Demonstrates Simulink's ability to recognize patterns and eliminate redundant operations

Table 1-1: Real-Time Workshop and Related Simulink Demos (Continued)

Demo Command	Demo Topic
`exprfolding`	Demonstrates how Real-Time Workshop folds (combines) expressions to dramatically improve code efficiency and readability.
`ecifdemo`	Demonstrates how Real-Time Workshop generates production code for Simulink software constructs `if`, `case`, `for`, and `while`.
`sfexfold`	Demonstrates the seamless integration between Simulink and Stateflow, using a variation of the model for the `exprfolding` demo.
`hierdemo`	Demonstrates hierarchical name resolution in a masked subsystem
`objectdemo`	Demonstrates how attributes of data objects are available throughout all stages of simulation, report generation, and code generation.
`tunabledemo`	Demonstrates the preservation of tunable expressions in the generated code despite mask transformations
`asap2demo`	Demonstrates ASAP2 (a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems) data export
`rsimtfdemo`	Demonstrates using the Rapid Simulation (rsim) Target to create an accelerated, stand-alone simulation environment.
`signalcapidemo`	Automatically generate a C-API interface for the signals in your model
`ptdemo`	Automatically generate a C-API interface for the parameters in your model
`ecoderdemos`	Tour of the Real-Time Workshop Embedded Coder (menu for demo suite)

Table 1-1: Real-Time Workshop and Related Simulink Demos (Continued)

Demo Command	Demo Topic
`rtwprofiledemo`	Demonstrates the hooks provided by Real-Time Workshop to profile generated code
`multimallocdemo`	Illustrates how to combine code for multiple models

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 as block diagrams, and 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 in order to create, test, and deploy real-time systems using Real-Time Workshop, successful emulation and deployment of real-time systems involves working familiarity with their 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 online at `http://www.mathworks.com`, through the MATLAB Help browser, and also in the form of PDF documents that you can view online or print.

This section includes the following topics:

- "Online Documentation" on page 1-18—Where to find Help online (HTML documents)
- "Printing the Documentation" on page 1-19—Printable versions (PDF documents)
- "For Further Information" on page 1-19—A guide to major help topics

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 on that page's **Documentation** link. To access the documentation with the MATLAB Help browser, use the following procedure:

1 In the MATLAB window, and from the **View** menu, click **Help**. Or from the **Help** menu, click **Full Product Family Help**.

   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 on any link there, or click on 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 "-", which will hide the topics under it when you click on it.

Printing the Documentation

The following manuals are available on the documentation CD as PDF files. You can also download them from the Real-Time Workshop Roadmap Web page:
`http://www.mathworks.com/access/helpdesk/help/toolbox/rtw/rtw.shtml`

The CD locations listed below assume your CD-ROM drive is listed as `Z:`.

- Getting Started with Real-Time Workshop—Located at `Z:\help\pdf_doc\rtw\rtw_gs.pdf`
- Real-Time Workshop User's Guide—Located at `Z:\help\pdf_doc\rtw\rtw_ug.pdf`
- Target Language Compiler Reference Guide—Located at `Z:\help\pdf_doc\rtw\targetlanguagecompiler.pdf`

The last two books are not distributed in printed form. You are welcome to print the PDF versions. When preparing to print, please be aware that each one has more than 500 pages.

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..." on page 1-14.

You can extensively customize output from Real-Time Workshop at the block, target, and makefile levels. For advanced uses, you may have to prepare or modify Target Language Compiler files. See the Target Language Compiler documentation for further descriptions.

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with Real-Time Workshop. They are listed in the table below.

Real-Time Workshop needs to run in the same environment as MATLAB and *requires* these products:

- MATLAB 6.5 (Release 13) or later
- Simulink 5.0 (Release 13) or later
- A supported compiler (See "Supported Compilers" on page 1-13 and "Third-Party Compiler Installation on Windows" on page 1-11)

MATLAB documentation—For information on using MATLAB, see the MATLAB documentation collection. It explains how to work with data and how to use the functions supplied with MATLAB. For a reference describing the functions supplied with MATLAB, see the online MATLAB Function Reference.

Simulink documentation—For information on using Simulink, see the Simulink documentation. It explains how to connect blocks to build models and change block parameters. It also provides a reference that describes each block in the standard Simulink library.

For more information about any of these products, see either

- The online documentation for that product, if it is installed or if you are reading the documentation from the CD
- Appropriate sections within MATLAB Release Notes for Release 13
- The "products" section on the MathWorks Web site at `http://www.mathworks.com`

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB. The blocksets listed below all include blocks that extend capabilities of Simulink.

Product	Description
CDMA Reference Blockset	Design and simulate IS-95A mobile phone equipment
Communications Blockset	Design and simulate communication systems
Communications Toolbox	Design and analyze communication systems
Control System Toolbox	Design and analyze feedback control systems
Data Acquisition Toolbox	Acquire and send out data from plug-in data acquisition boards
Gauges Blockset	Monitor signals with graphical instruments
DSP Blockset	Design and simulate DSP systems
Embedded Target for Motorola® MPC555	Generate Real-Time Workshop Embedded Coder production code for the Motorola MPC555
Embedded Target for the TI TMS320C6000™ DSP Platform	Deploy and validate DSP designs on Texas Instruments C6000 digital signal processors
Fixed-Point Blockset	Design and simulate fixed-point systems
Fuzzy Logic Toolbox	Design and simulate fuzzy logic systems
Instrument Control Toolbox	Control and communicate with test and measurement instruments
MATLAB Compiler	Convert MATLAB M-files to C and C++ code
Nonlinear Control Design Blockset	Optimize design parameters in nonlinear control systems

Product	Description
Real-Time Windows Target	Run Simulink and Stateflow models on a PC in real time
Real-Time Workshop Embedded Coder	Generate production code for embedded systems
Simulink	Design and simulate continuous- and discrete-time systems
Simulink Performance Tools	Manage and optimize the performance of large Simulink models
Simulink Report Generator	Automatically generate documentation for Simulink and Stateflow models
Stateflow	Design and simulate event-driven systems
Stateflow Coder	Generate C code from Stateflow charts
xPC Target	Perform real-time rapid prototyping using PC hardware
xPC Target Embedded Option	Deploy real-time applications on PC hardware

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'whole')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace` font	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'method')`

2

Building an Application

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

Automatic Program Building

The Real-Time Workshop automatic program building process creates programs for real-time applications in a variety of host environments. Automatic program building uses the `make` utility to control the compilation and linking of generated source code.

The figure below illustrates the complete process. The shaded box highlights the portions of it executed by Real-Time Workshop.

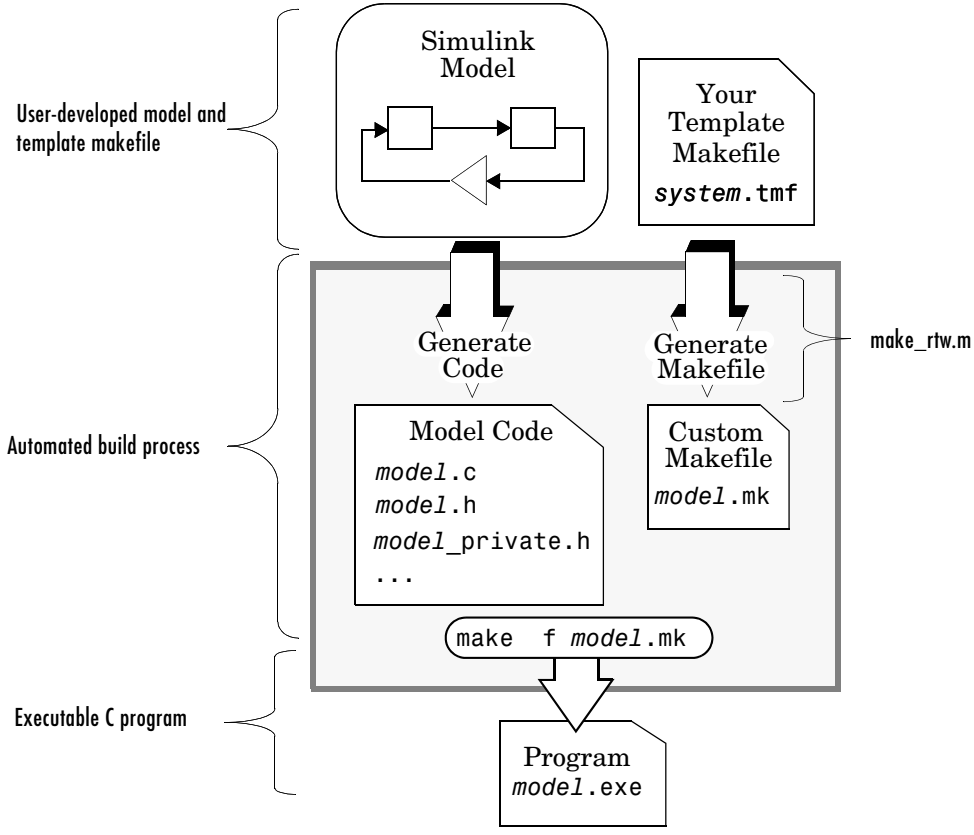


Figure 2-1: Automatic Program Building

A high-level M-file command controls the Real-Time Workshop build process. The default command, used with most targets, is `make_rtw`. Real-Time Workshop normally issues this command, although you may do so from the MATLAB control window at any time. If you are curious about how Real-Time Workshop choreographs its activities, you can inspect this M-file (but never edit it), located in *`matlabroot`*`/toolbox/rtw/rtw`.

Steps in the Build Process

Code generation begins with a two-step process, which is followed by two more steps whenever an executable is being compiled. The four steps (also summarized in "The Build Process" on page 3-5) are automatically completed when you click the **Build** button on the Real-Time Workshop dialog (assuming that Real-Time Workshop detects no constraints to generating code for your model; if it does, it will issue warnings):

1. Analyze the Model. Real-Time Workshop analyzes your block diagram and compiles it into an intermediate hierarchical representation called *model*`.rtw`.

2. Call the Target Language Compiler to Generate Code. The Target Language Compiler reads *model*`.rtw` and translates it to C code, which it places in a build directory within your working directory.

   If you have selected the **Generate code only** check box (in which case the **Build** button will be labeled **Generate code**), the process halts there.

3. Generate a Customized Makefile. The Target Language Compiler constructs a makefile from the appropriate target makefile template, and places it in the build directory.

4. Create the Executable. Your system's make utility reads the makefile to compile source code, link object files and libraries, and generate an executable (called *model* or *model*`.exe`), which is left in your working directory.

   If **Generate HTML report** was selected (the default) under **General code generation options**, the MATLAB Help browser will display the report once 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, as well as hyperlinks in source header comments that cause the respective block in the model diagram to be highlighted.

   The *model*`.rtw` file will be deleted unless you selected **Retain .rtw file** under **TLC Debugging** options. There is no benefit to preserving this file (which is produced every time Real-Time Workshop generates code for a

model, even if no changes were made), other than to refer to when debugging TLC scripts.

Additional details about each of the four steps follow.

Analyze the Model

The build process begins with the analysis of your Simulink block diagram. The analysis process consists of these tasks:

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

During this phase, Real-Time Workshop reads your model file (*model*`.mdl`) and 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. "Overview of a model.rtw File" on page 2-10 explains the basic structure of a `.rtw` file. For a detailed description of the *model*`.rtw` architecture, see "model.rtw File Contents" in the Target Language Compiler documentation.

Call the Target Language Compiler to Generate Code

In the second stage of the build procedure, the Target Language Compiler transforms the intermediate model description stored in *model*`.rtw` into target-specific code.

The Target Language Compiler is an interpreted programming language designed for the sole purpose of converting a model description into code. The Target Language Compiler executes a *TLC program* comprising several *target files* (`.tlc` script files). The TLC scripts specify how to generate code from the model, using the *model*`.rtw` file as input.

The TLC program consists of

- The *system target file*

  The system target file is the entry point or main file.
- *Block target files*

  For each block in a Simulink model, there is a block target file that specifies how to translate that block into target-specific code.
- The *Target Language Compiler function library*

  The Target Language Compiler function library contains functions that support the code generation process.

The Target Language Compiler begins by reading in the *model*`.rtw` file. It then compiles and executes the commands in the target files — first the system target file, then the individual block target files. The output of the Target Language Compiler is a source code version of the Simulink block diagram.

Generate a Customized Makefile

The third step in the build procedure is to generate a customized makefile, *model*`.mk`. The generated makefile instructs the `make` 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 makefile*, *system*`.tmf` (where *system* stands for the selected target name). The system template makefile is designed for your target environment. The template makefile allows you to specify compilers, compiler options, and additional information used during the creation of the executable.

The *model*`.mk` file is created 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 Chapter 2 of the Real-Time Workshop documentation lists all template makefiles that are bundled with Real-Time Workshop.

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

Create the Executable

Creation of an executable program is the final stage of the build process. This stage is optional, as illustrated by the control logic in Figure 2-2.

If you are targeting a system such as an embedded micro controller or a DSP board, you can choose to generate only source code. You can then cross compile your code and download it to your target hardware. “Making an Executable” in Chapter 2 of the Real-Time Workshop documentation discusses the options that control whether or not the build creates an executable.

The creation of the executable, if enabled, takes place after the *model*`.mk` file has been created. At this point, the build process invokes the `make` utility, which in turn runs the compiler. 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, `make` can also download the executable to your target hardware.

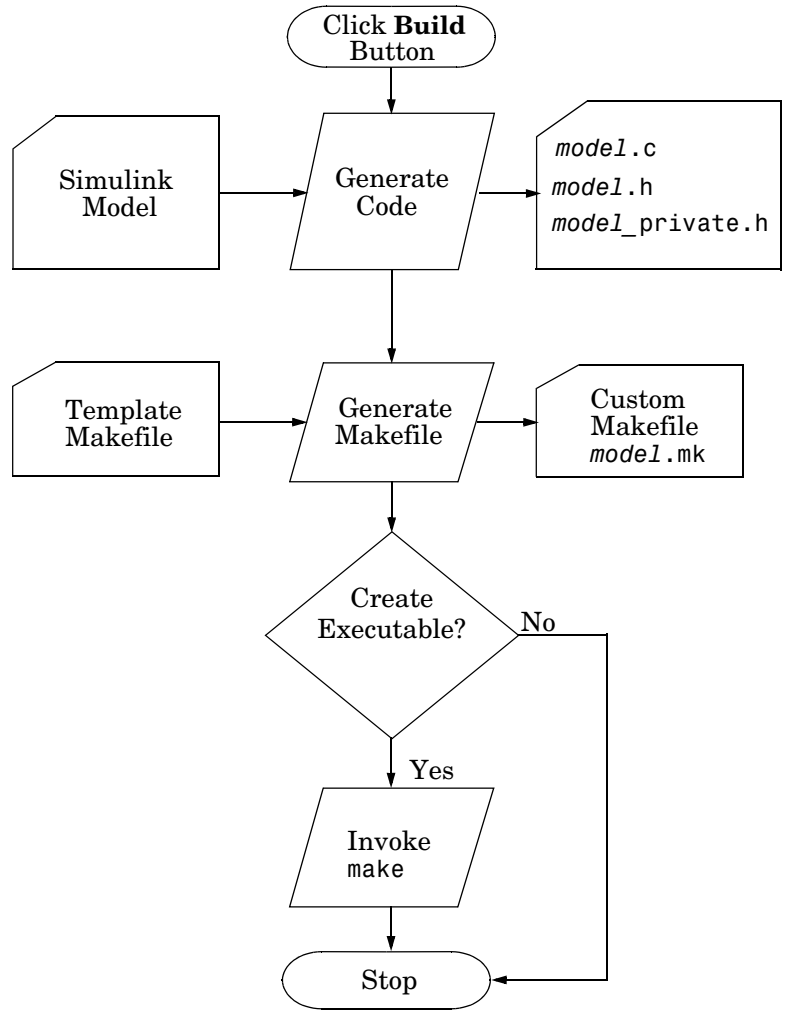


Figure 2-2: How Automatic Program Building Is Controlled

Summary of Files Created by the Build Procedure

The following is a list of the *model*.* files created during the code generation and build process for the GRT and GRT malloc targets. Many of the files derive from *model*.mdl, created by Simulink, which you can think of as a very high-level programming language source file. Files generated for embedded applications by Real-Time Workshop Embedded Coder are packaged slightly differently. Depending on model architectures and code generation options, other files may also be created by the build process:

- *model*.rtw, generated by Real-Time Workshop build process, is analogous to the object file created from a high-level language source program. This "compiled model" is deleted by default once the build is over, but may be retained for inspection.
- *model*.c, generated by the Target Language Compiler, is the C source code corresponding to the *model*.mdl file. It contains
  - All data except data placed in *model*_data.c
  - Include files *model*.h and *model*_private.h
  - Algorithm code
- *model*_data.c, generated by the Target Language Compiler, is a conditionally-generated C source file that when present contains
  - Constant block i/o parameters
  - Include files *model*.h and *model*_private.h
  - Constant parameters
- *model*_private.h, generated by the Target Language Compiler, is a header file that contains
  - Imported Simulink data symbols
  - Imported Stateflow machine parented data
  - Stateflow entry points
  - Real-Time Workshop-specific details (various macros, enums, etc. private to the code)
- *model*.h, generated by the Target Language Compiler, is a header file that includes *model*_private.h and also defines
  - Exported Simulink data symbols
  - Exported Stateflow machine parented data

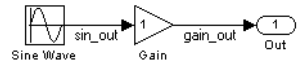
  - Model data structures, including `rtM`
  - Model entry point functions
- *`subsystem`*`.c`, containing executable code for each noninlined nonvirtual subsystem or copy thereof (when its code cannot be *reused*).
- *`subsystem`*`.h`, containing exported symbols for noninlined nonvirtual subsystems, analogous to *`model`*`.h`.
- *`model`*`.mk`, generated by the Real-Time Workshop build process, is the customized makefile used to build an executable.
- *`model`*`.exe` (on PC) or *`model`* (on UNIX), is an executable program, created under control of the `make` utility by your development system (unless you have specified **Generate code only** on the `Target configuration` portion of the Real-Time Workshop pane).

In addition, for each build when the **HTML report** option is selected, a set of `.html` files (one for each source file plus a *`model`*`_contents.html` index file) is generated in the `/html` subdirectory within your build directory.

For more information, see "Generated Source Files" in Chapter 2 of the Real-Time Workshop documentation.

Overview of a model.rtw File

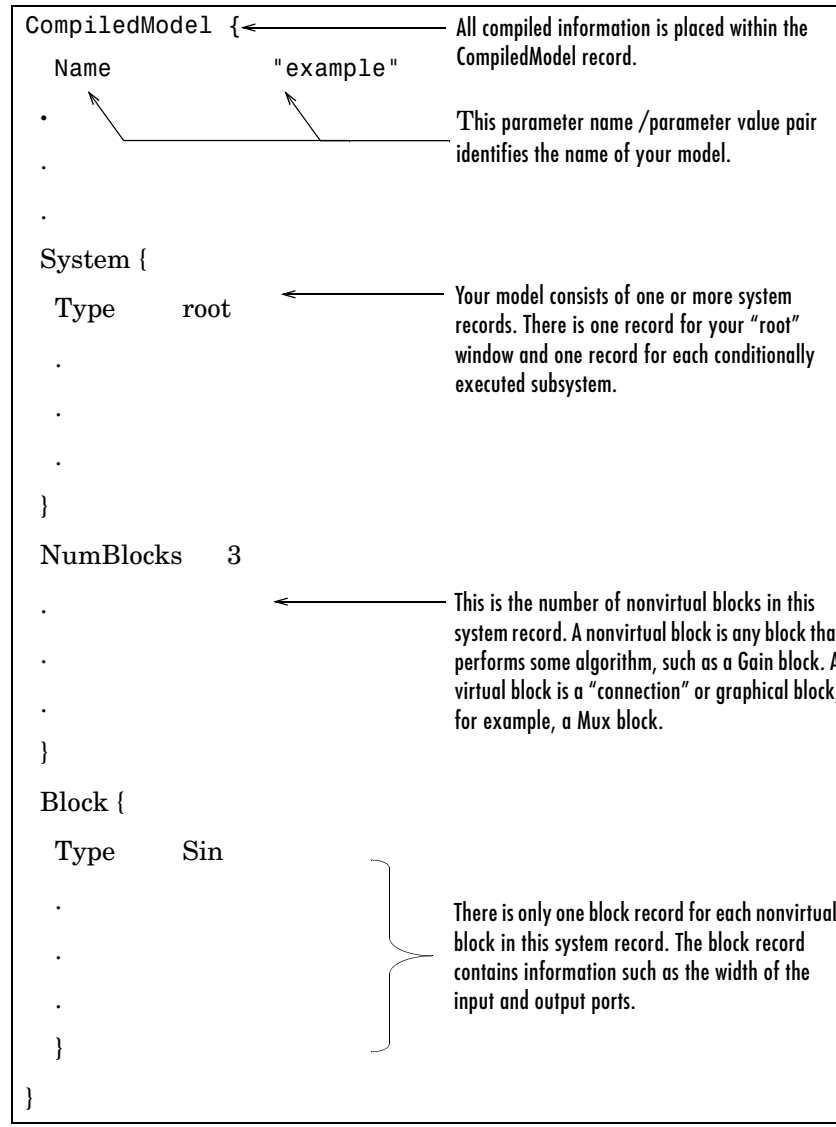
This section examines the basic features of a *`model`*`.rtw` file, which serves as input to the code generation process. The `.rtw` file shown is generated from the source model shown below.



This model is saved in a file called `example.mdl`. Real-Time Workshop generates `example.rtw`., an ASCII file by *compiling* the model into structured records from which code can be generated. The `example.rtw` file consists of parameter name/parameter value pairs, stored in a hierarchical structure of nested text records.

Below is an excerpt from `example.rtw`. The majority of lines have been elided to highlight on the structure of the file rather than its specific contents. You can find a more extensive example of a *`model`*`.rtw` file in the "Code

Generation Architecture" section of the Target Language Compiler documentation.



For more information on these files, see the Appendix A of the Target Language Compiler documentation, which details all records comprising *model*.rtw files. The documentation also provides tutorials on processing such record files.

Directories Used in the Build Process

Real-Time Workshop creates output files in two directories during the build process:

- The working directory

  If an executable is created, it is written to your working directory. The executable is named *model*.exe (on PC) or *model* (on UNIX).

- The build directory

  The build process creates a subdirectory, called the build directory, 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 type of the chosen target (e.g., grt for generic real-time). The build directory stores generated source code and all other files created during the build process (except the executable).

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

Depending upon the target and code generation and build options selected, additional files in the build directory may include

- *model*.rtw
- Object (.obj or .o) files
- Code modules generated from subsystems
- HTML summary reports of files generated (in its /html subdirectory)
- TLC profiler report files
- Block I/O (*model*_bio.c) and parameter tuning (*model*_pt.c) information files
- Real-Time Workshop project (*model*.tmw) files

3

Working with Real-Time Workshop

This chapter provides an overview of the ideas and technology behind Real-Time Workshop, and hands-on exercises to help you to get started generating code as quickly as possible. It includes the following topics:

Basic Real-Time Workshop Concepts (p. 3-2)	Terms and definitions, such as *host*, *target*, *code format*, *makefile*, and more, used in the documentation
Quick Start Tutorials (p. 3-7)	Hands-on exercises that demonstrate essential features of Real-Time Workshop

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

Basic Real-Time Workshop Concepts

Even if you have experience in building real-time systems, you may find it helpful to review the following descriptions to be sure of understanding the nomenclature that Real Time Workshop documentation uses. A more extensive list of terms may be found in Appendix A, "Glossary."

The process of generating source code from Simulink models is shown in the following diagram. The terms and concepts involved are described below.

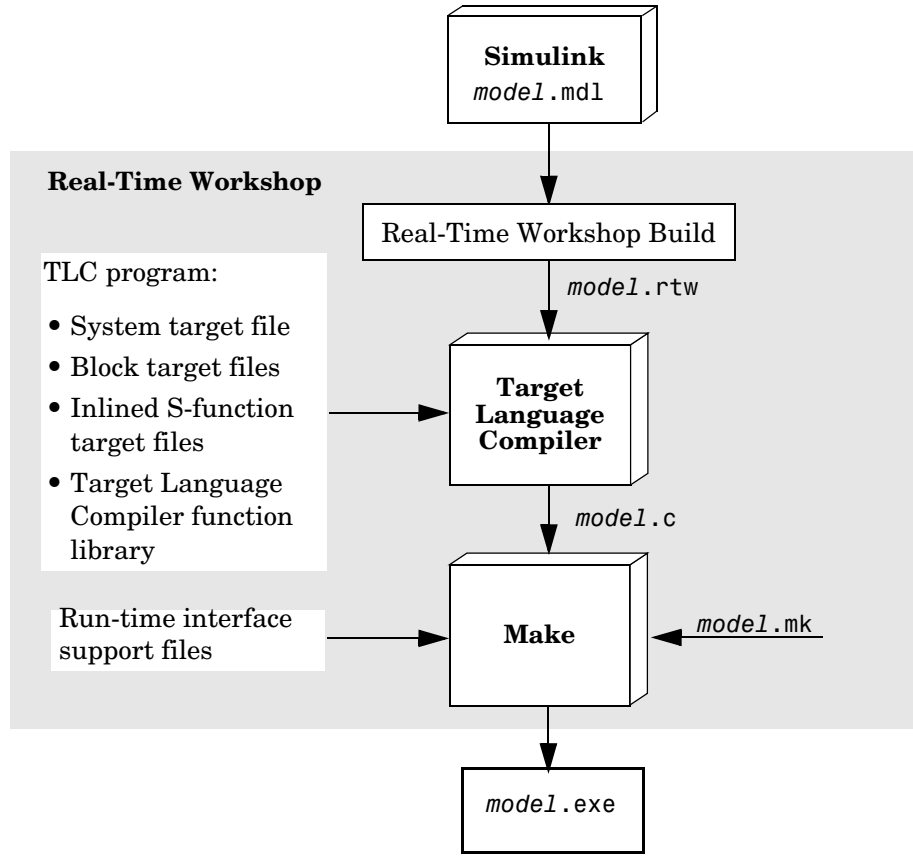


Figure 3-1: Real-Time Workshop Code Generation Process

Target and Host

A *target* is an environment—hardware or operating system—on which your generated code will run. The process of specifying this environment is called *targeting*.

The process of generating target-specific code is controlled by a *system target file*, a *template makefile*, and a *make command*. To select a desired target, you can specify these items individually, or you can choose from a wide variety of ready-to-run configurations.

The *host* is the system you use to run MATLAB, Simulink, and Real-Time Workshop. Using the build tools on the host, you create code and an executable that runs on your target system.

Available Target Configurations

Real-Time Workshop supports many target environments. These include ready-to-run configurations and third-party targets. You can also develop your own custom target.

For a complete list of bundled targets, with their associated system target files and template makefiles, see "The System Target File Browser" in Chapter 2 of the Real-Time Workshop documentation.

Code Formats

A *code format* specifies a framework for code generation suited for specific applications.

When you choose a target configuration, you implicitly choose a code format. If you use Real-Time Workshop Embedded Coder, for example, the code generated will be in *embedded C* format. The embedded C code format is a compact format designed for production code generation. Its small code size, use of static memory, and simple call structure make it optimal for embedded applications.

Many other targets, such as the generic real-time (GRT) target, use the *real-time* code format. This format, less compact but more flexible, is optimal for rapid prototyping applications.

For a complete discussion of available code formats, see "Generated Code Formats" in Chapter 3 of the Real-Time Workshop documentation.

The Generic Real-Time Target

Real-Time Workshop provides a generic real-time development target. The GRT target provides an environment for simulating fixed-step models in single or multitasking mode. A program generated with the GRT target runs your model, in simulated time, as a stand-alone program on your workstation.

The GRT target allows you to perform code validation by logging system outputs, states, and simulation time to a data file. The data file can then be loaded into the MATLAB workspace for analysis or comparison with the output of the original model.

The GRT target also provides a starting point for targeting custom hardware. You can modify the GRT harness program, `grt_main.c`, to execute code generated from your model at interrupt level under control of a real-time clock.

Target Language Compiler Files

Real-Time Workshop generates source code for models and blocks through the Target Language Compiler, which reads script files (or *TLC files)* that specify the format and content of output source files. Two types of TLC files are used:

1. A *system target file*, which describes how to generate code for a chosen target, is the entry point for the TLC program that creates the executable.
2. *Block target files* define how the code looks for each of the Simulink blocks in your model.

System and block target files have the extension `.tlc`. By convention, a system target file has a name corresponding to your target. For example, `grt.tlc` is the system target file for the GRT target.

If you need to incorporate legacy code or maximize the efficiency of code for models containing S-functions, you will need to prepare your own TLC files to augment code generation for such models. These and other advanced uses of TLC scripts are explained in the Target Language Compiler Reference Guide, which provides a set of tutorials to help you get started.

Template Makefiles

Real-Time Workshop uses template makefiles to build an executable from the generated code.

The Real-Time Workshop build process creates a makefile from the template makefile. Each line from the template makefile is copied into the makefile; tokens encountered during this process are expanded into the makefile.

The name of the makefile created by the build process is *model*.mk (where *model* is the name of the Simulink model). The *model*.mk file is passed to a make utility, which compiles and links an executable from a set of files.

By convention, a template makefile has an extension of .tmf and a name corresponding to your target and compiler. For example, grt_vc.tmf is the template makefile for building a generic real-time program under Visual C/C++.

You specify system target files and template makefiles using the Real-Time Workshop pane of the **Simulation Parameters** dialog box, either by typing their filenames or choosing them with the Target File Browser. "Tutorial 1: Building a Generic Real-Time Program" on page 3-8 below introduces these interfaces. See "Target Configuration Options" in Chapter 2 for additional documentation.

The Build Process

A high-level M-file command controls the Real-Time Workshop build process. The default command, used with most targets, is make_rtw. When you initiate a build, Real-Time Workshop invokes make_rtw. The make_rtw command, in turn, invokes the Target Language Compiler and utilities such as make. The build process consists of the following stages:

1. First, make_rtw compiles the block diagram and generates a model description file, *model*.rtw.

2. Next, make_rtw invokes the Target Language Compiler to generate target-specific code, processing *model*.rtw as specified by the selected system target file.

3. Next, make_rtw creates a makefile, *model*.mk, from the selected template makefile.

4 Finally, `make` is invoked. `make` compiles and links a program from the generated code, as instructed in the generated makefile.

"Automatic Program Building" on page 2-2 gives an overview of the build process. "Simulation Parameters and Code Generation" in Chapter 2 of the Real-Time Workshop documentation expands on how model data affects code generation.

Model Parameters and Code Generation

The simulation parameters of your model directly affect code generation and program building. For example, if your model is configured to stop execution after 60 seconds, the program generated from your model will also run for 60 seconds.

Before you generate code and build an executable, you must verify that you have set the model parameters correctly in the **Simulation Parameters** dialog box. See "Simulation Parameters and Code Generation" in Chapter 2 of the Real-Time Workshop documentation for more information. Real-Time workshop will refuse to generate code and issue diagnostics when certain parameters (such as solver type) are set inappropriately. However, when other parameters (such as **Production Hardware Characteristics**) have inappropriate values, code generated by Real-Time Workshop can yield incorrect results.

Real-Time Workshop imposes certain requirements and restrictions on the model from which code is generated. Many of these, for example, the use of variable-step solvers, are target specific. The following sections in the Real-Time Workshop documentation may help you understand restrictions on code generation that may apply:

- "Choosing a Code Format for Your Application" in Chapter 3 compares features that Real-Time Workshop supports for different targets.
- "Parameters: Storage, Interfacing, and Tuning" in Chapter 5 describes how Real-Time Workshop structures parameter data.
- "Interfacing Parameters and Signals" in Chapter 14 provides more advanced information on how to access parameter data in generated code.

Quick Start Tutorials

This section provides hands-on experience with the code generation, program building, data logging, and code validation capabilities of Real-Time Workshop.

- "Tutorial 1: Building a Generic Real-Time Program" on page 3-8 shows how to generate C code from a Simulink demonstration model and build an executable program.
- "Tutorial 2: Data Logging" on page 3-15 explains how to modify the demonstration program to save data in a MATLAB MAT-file, for plotting.
- "Tutorial 3: Code Validation" on page 3-19 demonstrates how to validate the generated program by comparing its output to that of the original model.
- "Tutorial 4: A First Look at Generated Code" on page 3-23 examines code generated from a very simple model, illustrating the effect of some of the Real-Time Workshop code generation options.
- "Tutorial 5: Getting Started with External Mode Using GRT" on page 3–33 acquaints you with the basics of using external mode on a single computer, and demonstrates the value of external mode to rapid prototyping.

These tutorials assume basic familiarity with MATLAB and Simulink. You should also read "Building an Application" on page 2-1 before proceeding.

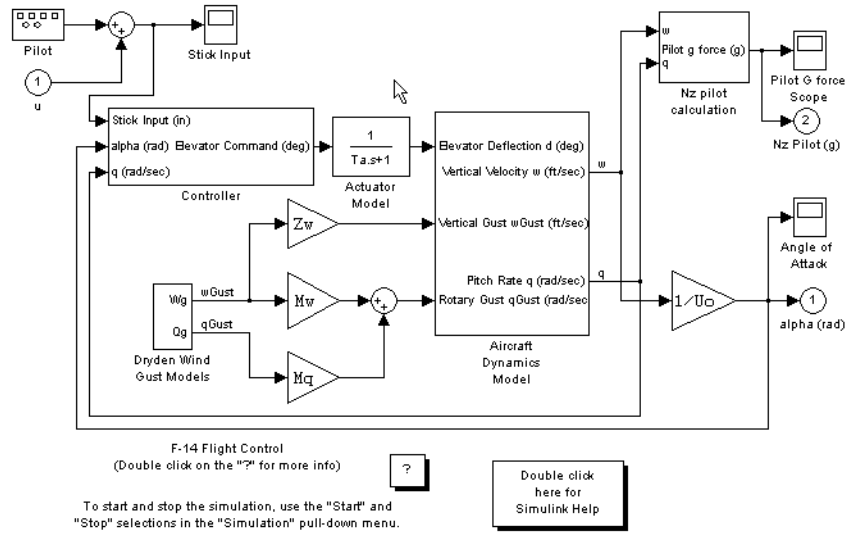
The procedures for building, running, and testing your programs are almost identical in UNIX and PC environments. The discussion notes differences where applicable.

Make sure that a MATLAB compatible C compiler is installed on your system before proceeding with these tutorials. See "Supported Compilers" in Chapter 1 of the Real-Time Workshop documentation or more information on supported compilers and compiler installation.

The f14 Demonstration Model

Tutorials 1-3 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 outputs are visually monitored by Scope blocks.

Tutorial 1: 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 stand-alone program runs on your workstation, independent of external timing and events.

Working and Build Directories

It is convenient to work with a local copy of the `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**. Save a copy of the `f14` model as `d:/f14example/f14rtw.mdl`.

Be aware that 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. We examine the build directory and its contents at the end of this tutorial.

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 Chapter 11 of the Real-Time Workshop documentation for details.

To set parameters, use the **Simulation Parameters** dialog box as follows:

1 From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog box opens.

2 Click the **Solver** tab and enter the following parameter values on the Solver pane.

Start Time: `0.0`

Stop Time: `60`

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

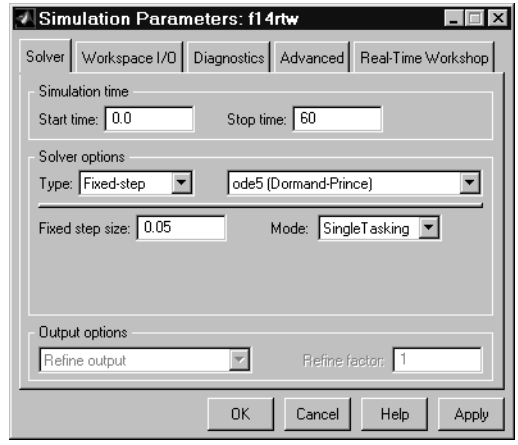
Fixed step size: `0.05`

Mode: `SingleTasking`

3 Click **Apply**. Then click **OK** to close the dialog box.

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

The Solver pane with the correct parameter settings is shown below.



Selecting the Target Configuration

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

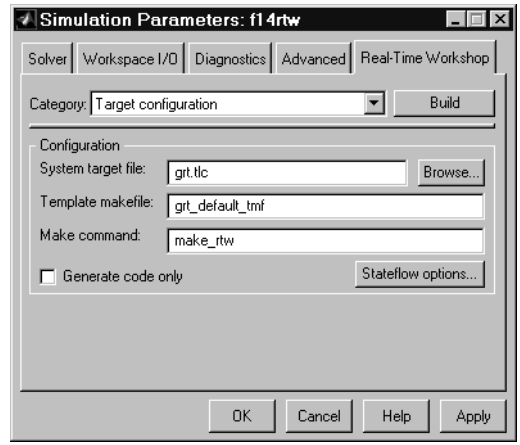
In these tutorials, you do not need to specify these parameters individually. Instead, you use the ready-to-run generic real-time target configuration. The GRT target is designed to build a stand-alone executable program that runs on your workstation.

To select the GRT target:

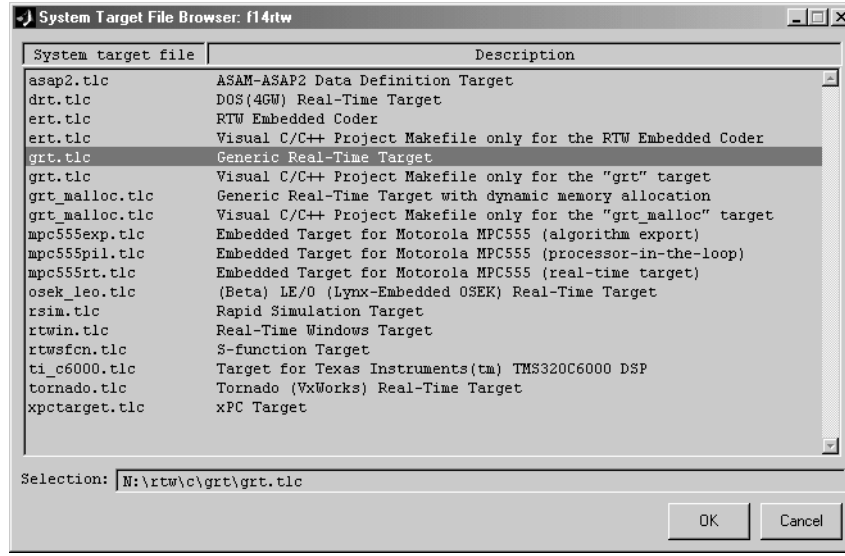
1. From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog box opens.

2. Click on the **Real-Time Workshop** tab of the **Simulation Parameters** dialog box. The Real-Time Workshop pane activates.

3. The Real-Time Workshop pane has several parts, which are selected via the **Category** menu. Select `Target configuration` from the **Category** menu, as shown below.



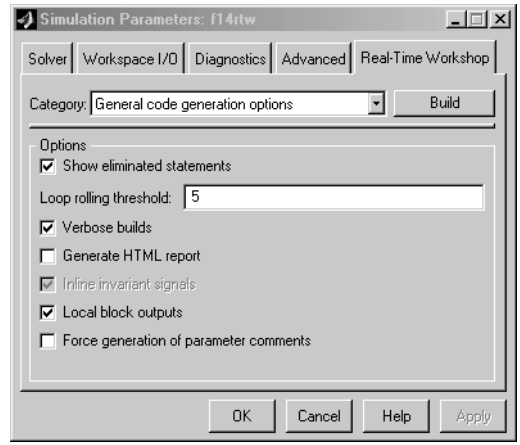
4. Click the **Browse** button next to the **System target file** field. This opens the System Target File Browser, illustrated below. The browser displays a list of all currently available target configurations. When you select a target configuration, Real-Time Workshop automatically chooses the appropriate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



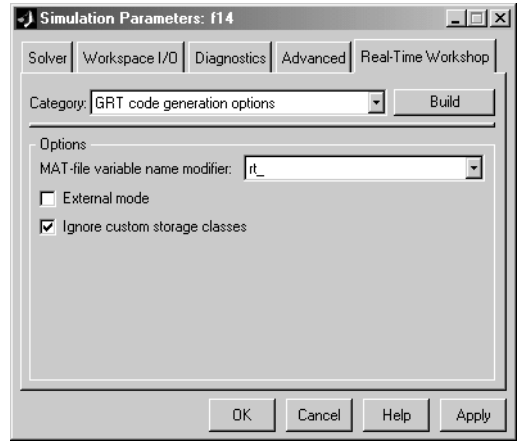
5 From the list of available configurations, select `Generic Real-Time Target` (as shown above) and then click **OK**.

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

7 Select `General code generation options` from the **Category** menu. The options displayed here are common to all target configurations. Make sure that all options are set to their defaults, as shown below.



8 Select `GRT code generation options` from the **Category** menu. The options displayed here are specific to the GRT target. Make sure that all options are set to their defaults, as below.



9 Select `TLC debugging` from the **Category** menu. Make sure that all options in this category are cleared.

10 Select `Target configuration` from the **Category** menu. Make sure that the **Generate code only** option is not selected.

11 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 build and run the program:

1 Click the **Build** button in the **Simulation Parameters** dialog box to start the build process.

2 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: .\f14rtw_grt_rtw
```

The content 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
```

3 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.

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

```
dir
.              f14rtw.exe     f14rtw_grt_rtw
..             f14rtw.mdl
```

4 To run the executable from the MATLAB command window, type

```
!f14rtw
```

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

The program produces one line of output:

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

5 Finally, to see the contents of 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 code files:

- `f14rtw.c` — the stand-alone C code that implements the model.
- `f14rtw_data.c` — initial parameter values used by the model.
- `f14rtw.h` — an include header file containing definitions of parameter and state variables
- `f14rtw_types.h` — forward declarations of data types used in the code.
- `f14rtw_private.h` — a header file containing common include definitions
- `rtmodel.h` — a master header file for including generated code in the static main program (its name never changes, and it simply includes `f14rtw.h`).

The build directory also contains other files used in the build process, such as the object (`.obj`) files and the generated makefile (`f14rtw.mk`).

Tutorial 2: Data Logging

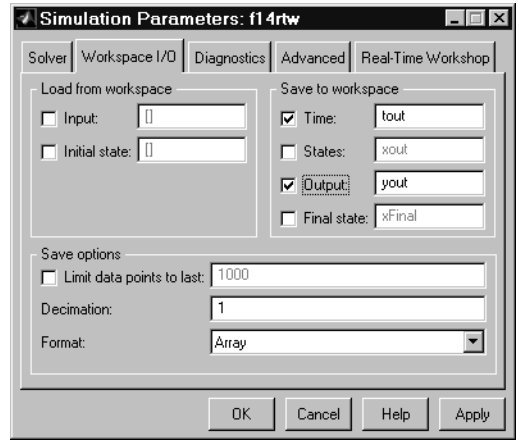
The Real-Time Workshop MAT-file data logging facility enables a generated program to save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) `model.mat`, where `model` is the name of your model. In this tutorial, data generated by the model `f14rtw.mdl` is logged to the file `f14rtw.mat`. Refer to "Tutorial 1: Building a Generic Real-Time Program" on page 3-8 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 Workspace I/O pane of the **Simulation Parameters** dialog box. The process is nearly the same as configuring a Simulink model to save output to the MATLAB workspace. 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 (by default) `rt_tout`.

In this tutorial, you will modify the `f14rtw` model such that the generated program saves the simulation time and system outputs to the file `f14rtw.mat`. Then, you will load the data into the MATLAB workspace and plot simulation time against one of the outputs.

To use the data logging feature:

1. Select the **Workspace I/O** tab of the **Simulation Parameters** dialog box. The Workspace I/O pane specifies how outport data is loaded from and saved to the workspace.



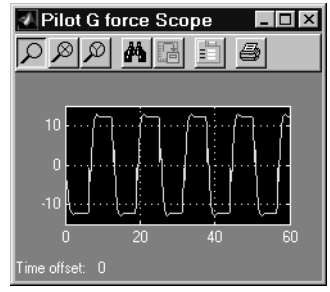
2. Select the **Time** option. Enabling the **Time** option causes Real-Time Workshop to generate code that logs the simulation time to the MAT-file matrix `rt_tout`.

3 Select the **Output** option. Enabling the **Output** option causes Real-Time Workshop to generate code that logs the root Output blocks (Angle of Attack and Pilot G Force) to the MAT-file matrix `rt_yout`.

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

4 If any other options are enabled, clear them. Set **Decimation** to `1` and **Format** to `Array`. Then click **Apply**.

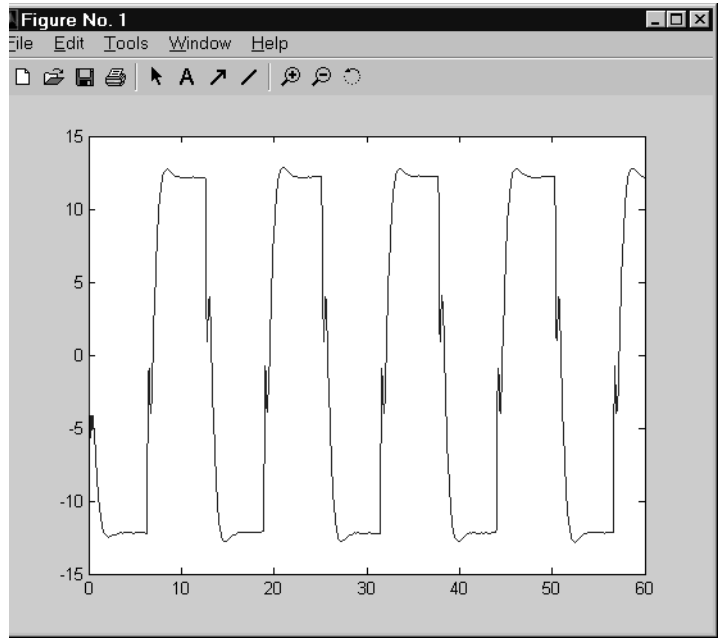
5 Open the Pilot G Force Scope block. To run the model, click on the **Start** button in the toolbar of the Simulink window. The scope displays below.



6 Verify that the simulation time and Pilot G Force outputs have been correctly saved to the workspace by plotting simulation time versus Pilot G Force.

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

   The resultant plot is shown below.



7 The `f14rtw` program must be rebuilt, because you have changed the model by enabling data logging. Select **Build Model** from the **Real-Time Workshop** menu of the **Tools** menu in the Simulink window. This is an alternative way to start the Real-Time Workshop build process. It is identical to using **Build** button in the **Simulation Parameters** dialog box.

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

```
!f14rtw
```

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

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

10 Clear the workspace, load the MAT-file data, and look at the workspace variables:

```
clear
load f14rtw.mat
whos
```

11 Observe that the variables `rt_tout` (time) and `rt_yout` (G Force and Angle of Attack) have been loaded from the file. Plot G Force as a function of time.

```
plot(rt_tout,rt_yout(:,2))
```

12 The plot should appear identical to the plot you produced in step 5 above.

Tutorial 3: Code Validation

In this tutorial, the code generated from the `f14rtw` model is validated against the model. The code is validated by capturing and comparing data from runs of the Simulink model and the generated program.

Note To obtain a valid comparison between outputs of the model and the generated program, make sure that you have selected the same integration scheme (`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 Tutorial 2.

Logging Signals via Scope Blocks

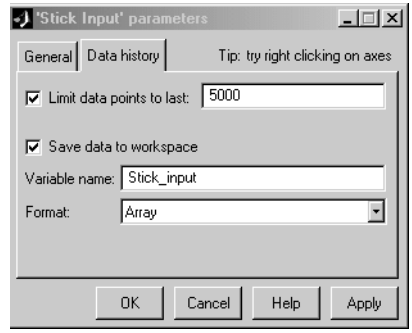
This example uses Scope blocks (rather than Outport blocks) to log both input and output data. To configure the Scope blocks to log data:

1 Before proceeding with this tutorial, 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 on the **Parameters** button on the toolbar of the Scope window. The **Scope Properties** dialog box opens.

3 Select the **Data History** tab of the **Scope 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.

In the example above, the Stick Input signal to the scope block will be logged to the array `Stick_input` during simulation. The generated code will log the same signal data to the MAT-file variable `rt_Stick_input` during a run of the executable program.

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`.

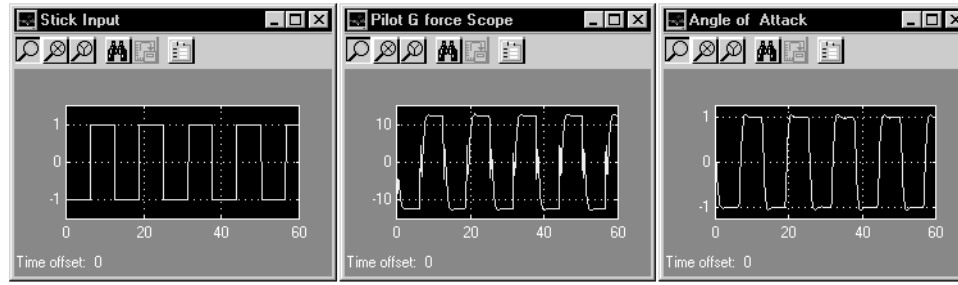
7 Save the model.

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 blocks display.



3 Use the `whos` command to observe 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

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

1 Select **Build Model** from the **Real-Time Workshop** menu of the **Tools** menu in the Simulink window.

2 When the build completes, run the stand-alone program from MATLAB:

```
!f14rtw
```

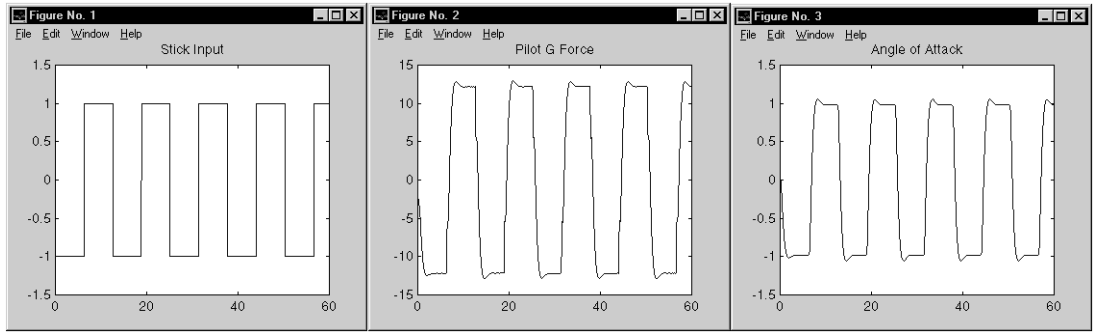
3 Load the data file `f14rtw.mat` and observe the workspace variables:

```
load f14rtw
whos
```

The data loaded from the MAT-file will include `rt_Pilot_G_force`, `rt_Angle_of_attack`, `rt_Stick_input`, and `rt_tout`.

4 You can now use MATLAB to plot the three workspace variables as a function of time.

```
plot(rt_tout,rt_Stick_input(:,2))
figure
plot(rt_tout,rt_Pilot_G_force(:,2))
figure
plot(rt_tout,rt_Angle_of_attack(:,2))
```



Comparing Results of the Simulation and the Generated Program

Your Simulink simulations and the generated code should produce nearly identical output.

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    0.4441
```

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

```
max(abs(rt_Pilot_G_force-Pilot_G_force))
```

```
ans =
  1.0e-013 *
         0    0.7283
```

Overall agreement is within 10^{-13}. This slight error can be caused by many factors, including

- Different compiler optimizations
- Statement ordering
- Run-time libraries

For example, a function such as `sin(2.0)` may return a slightly different value, depending on which C library you are using.

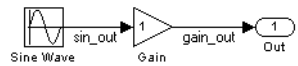
For the same reasons, your comparison results may not be identical to those above.

Tutorial 4: 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 may 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 3-31 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 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 above diagram. Label the signals as shown.

4. From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog box opens.

5. Click the **Solver** tab and 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.

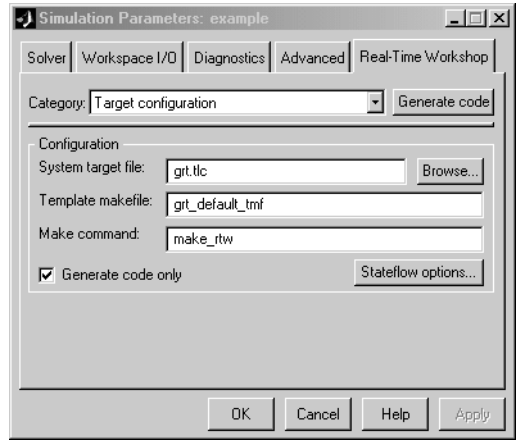
6. Click **Apply**.

7. Click the **Workspace I/O** tab and make sure all check boxes are cleared.

8. Click **Apply**.

9. Click the **Real-Time Workshop** tab. Select `Target configuration` from the **Category** menu. Next, select the **Generate code only** option. This option causes Real-Time Workshop to generate code without invoking `make` to compile and link the code. This option is convenient for this exercise, as we are only interested in looking at the generated code. Note that the **Build** button caption changes to **Generate code**.

   Also, make sure that the generic real-time (GRT) target is selected. The pane should appear as below.



10 Click **Apply**.

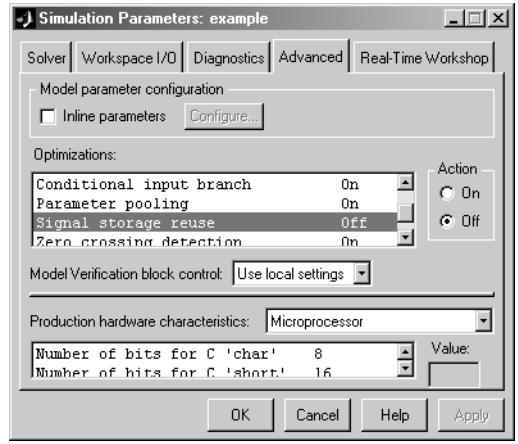
11 Save the model.

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. We now disable this option to see what the nonoptimized generated code looks like:

1 From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog box opens.

2 Click the **Advanced** tab. Select the `Signal storage reuse` option and select the **Off** radio button, as shown below.



3 Click **Apply**.

4 Click the **Real-Time Workshop** tab and select `Target configuration` from the **Category** menu. Then click the **Generate code** button.

5 Because the **Generate code only** option was selected, Real-Time Workshop does not invoke your make utility. The code generation process ends with the message

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

6 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 into the MATLAB editor:

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

7 In `example.c`, find the function `MdlOutputs`.

The generated C code consists of procedures that implement the algorithms defined by your Simulink block diagram. Your target's execution engine executes the procedures 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 "Program Architecture" in Chapter 7 of the Real-Time Workshop 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 our example, the generated `MdlOutputs` function implements the actual algorithm for multiplying a sine wave by a gain. The `MdlOutputs` function computes the model's block outputs. The run-time interface must call `MdlOutputs` at every time step.

With buffer optimizations turned off, `MdlOutputs` assigns unique buffers to each block output. These buffers (`rtB.sin_out`, `rtB.gain_out`) are members of a global data structure, `rtB`. The code is shown below:

```
void MdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sine Wave */
  rtB.sin_out = rtP.Sine_Wave_Amp *
    sin(rtP.Sine_Wave_Freq * ssGetT(rtS) +
    rtP.Sine_Wave_Phase) + rtP.Sine_Wave_Bias;

  /* Gain Block: <Root>/Gain */
  rtB.gain_out = rtB.sin_out * rtP.Gain_Gain;

  /* Outport Block: <Root>/Out1 */
  rtY.Out1 = rtB.gain_out;
}
```

We now turn buffer optimizations on and observe how these optimizations improve the code.

Generating Code with Buffer Optimization

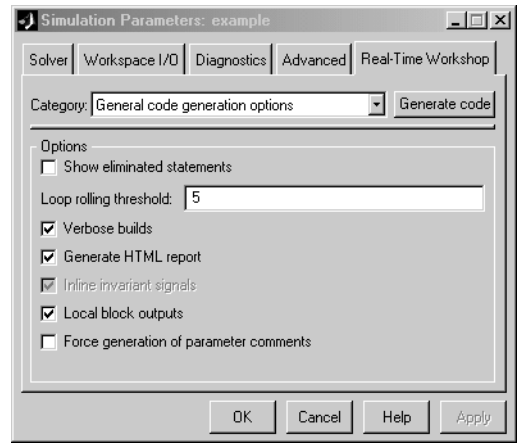
Enable buffer optimizations and regenerate the code as follows:

1. From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog box opens.

2. Click the **Advanced** tab. Select the `Signal storage reuse` option and select the **On** radio button.

3. Click **Apply**.

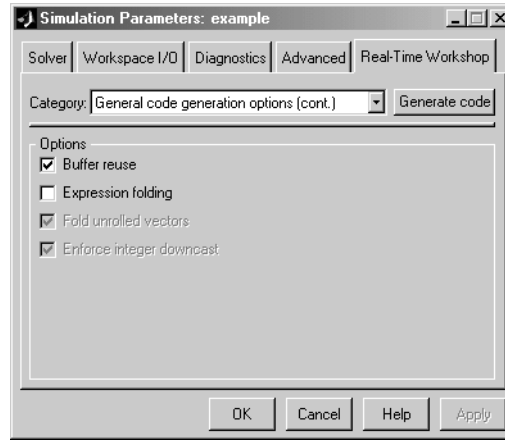
4 Click the **Real-Time Workshop** tab. Select `General code generation options` from the **Category** menu.



5 Make sure that the **Local block outputs** option is selected, as shown above.

6 Click **Apply**.

7 Select `General code generation options (cont.)` from the **Category** menu.



8. Make sure that the **Buffer reuse** option is selected, as shown above. Make sure that the **Expression folding** option is off, which will disable the two options below it, as shown. We will observe the effects of **Expression folding** later in this tutorial.

9. Click **Apply**.

10. Click the **Generate code** button.

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

12. Edit `example_grt_rtw/example.c`, and examine the function `MdlOutputs`.

With buffer optimizations enabled, the code in `MdlOutputs` reuses `rtb_temp0`, a temporary buffer with local scope, rather than assigning global buffers to each input and output:

```
void MdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_temp0;

  /* Sin Block: <Root>/Sine Wave  */
rtb_temp0 = rtP.Sine_Wave_Amp *
sin(rtP.Sine_Wave_Freq * ssGetT(rtS) + rtP.Sine_Wave_Phase) +
rtP.Sine_Wave_Bias;

/* Gain Block: <Root>/Gain
 *   Gain value: rtP.Gain_Gain
 */
  rtb_temp0 *= rtP.Gain_Gain;

  /* Outport Block: <Root>/Out1  */
  rtY.Out1 = rtb_temp0;
}
```

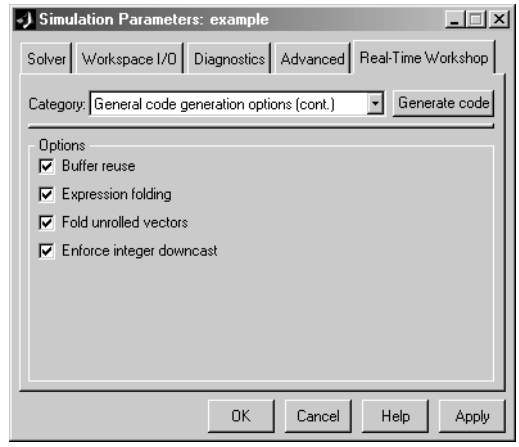
This code is more efficient in terms of memory usage. The efficiency improvement gained by enabling **Buffer reuse** and **Local block outputs** would be more significant in a large model with many signals.

A Further Optimization: Expression Folding

As a final optimization, we will 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 `General code generation options (cont.)` from the **Category** menu, if you have not already done so.



2. Select the **Expression folding** option. Make sure that the options **Fold unrolled vectors** and **Enforce integer downcast** (below **Expression folding**) are selected, as shown.

3. Make sure that **Buffer reuse** is still selected, as shown.

4. Click **Apply**.

5. Click the **Generate code** button.

6. As before, Real-Time Workshop generates code in the `example_grt_rtw` directory.

7. Edit `example_grt_rtw/example.c`, and examine the function `MdlOutputs`.

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.

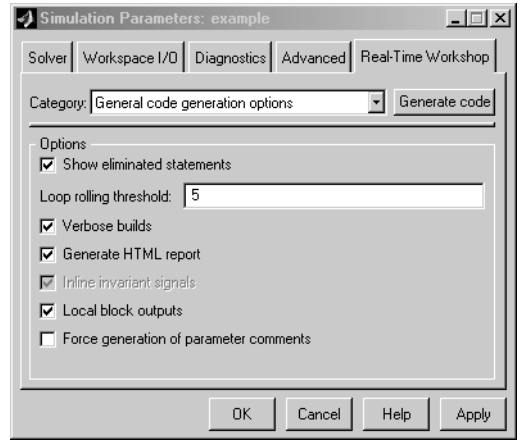
With **Expression folding** selected, there is a subtle but significant difference in the generated code: the gain computation is incorporated (or "folded") directly into the Outport computation, eliminating the temporary location and separate code statement. This computation is on the last line of the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

```
void MdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_sin_out;

  /* Sin Block: <Root>/Sine Wave  */
  rtb_sin_out = rtP.Sine_Wave_Amp *
   sin(rtP.Sine_Wave_Freq * ssGetT(rtS) + rtP.Sine_Wave_Phase)
+
    rtP.Sine_Wave_Bias;

  /* Outport: <Root>/Out1  incorporates:
   *   Gain: <Root>/Gain
   *
   * Regarding <Root>/Gain :
   *   Gain value: rtP.Gain_Gain
   */
  rtY.Out1 = (rtP.Gain_Gain * rtb_sin_out);
}
```

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. We strongly recommend that you use this option.

HTML Code Generation Reports

When the **Generate HTML report** option under `General code generation options` is selected (see figure below), a navigable summary of source files is produced when the model is built.



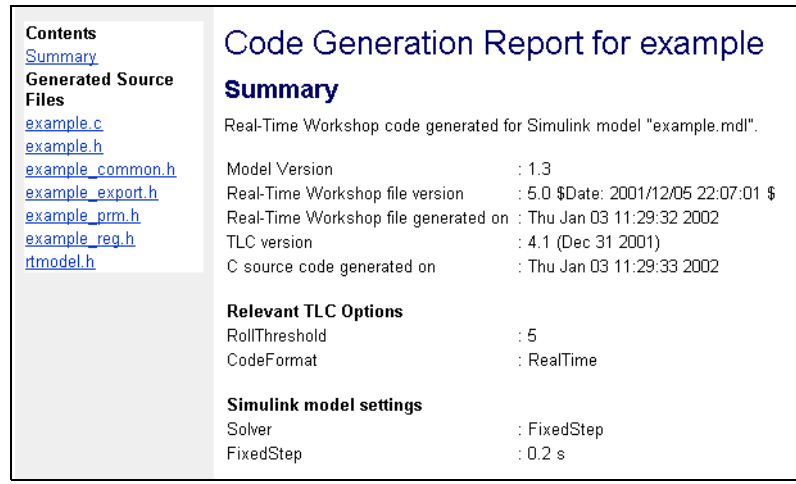
Code generation causes Real-Time Workshop to produce an HTML file for each source file, plus a summary and an index file, in a directory named `/html` within the build directory. Unless you are running MATLAB in `-nodesktop` mode (a UNIX option only), the HTML summary and index are automatically loaded into the MATLAB Help browser and displayed, as the following figure illustrates. You can click on links in the report to inspect source and include files, and view relevant documentation. Block header comments in source files displayed in the browser have hyperlinks back to the model that cause the block that generated that section of code to be highlighted. To review any HTML report at a later time, use any Web browser to open the file `html/`*`model`*`_codgen_rpt.html` within your build directory.

Note The contents of HTML reports for different target types will vary, and reports for models with subsystems will feature additional information.

For further information, consult these sections of the Real-Time Workshop documentation:

- "Code Generation and the Build Process" in Chapter 2 contains details on buffer optimizations and other code generation options.
- See "Optimizing the Model for Code Generation" in Chapter 9 for full details on Expression folding and on other code optimization techniques.

- For details on the structure and execution of *model*.c files, refer to "Program Architecture" in Chapter 7 of the Real-Time Workshop documentation.



HTML Report for Code Generated for a GRT Target

Tutorial 5: Getting Started 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 would follow in simulating, building, and tuning an actual real-time application:

- Part 1: Setting Up the Model
- Part 2: Building the Target Executable
- Part 3: Running the External Mode Target Program
- Part 4: Tuning Parameters

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 under 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 Chapter 6 of the Real-Time Workshop documentation.

Part 1: 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:

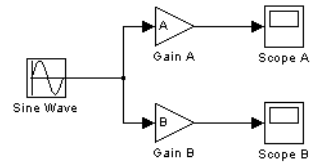
1. Create the directory from the MATLAB command line by typing

   ```
   mkdir ext_mode_example
   ```

2. Make `ext_mode_example` your working directory:

   ```
   cd ext_mode_example
   ```

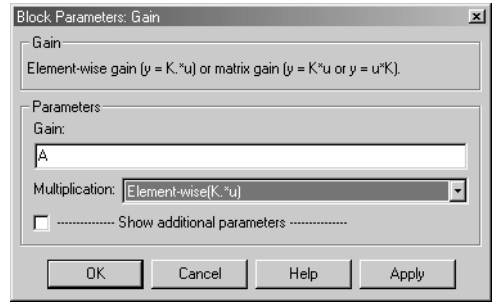
3. Create a model in Simulink with a Sine Wave block for the input signal, two Gain blocks in parallel, and two Scope blocks. The model is shown below. Label the Gain and Scope blocks as shown.



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

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

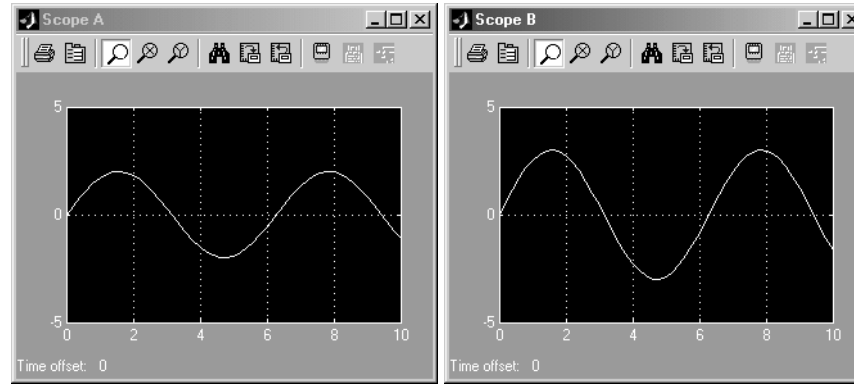
5 Open Gain block A and set its Gain parameter to the variable `A` as shown below.



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, new gain values can be downloaded to the executing target program by assigning new values to workspace variables `A` and `B`, or by editing the values in the block parameter dialog boxes.

7 Verify correct operation of the model. Open the Scope blocks and run the model. Given that `A=2` and `B=3`, the output should look like this.



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

Part 2: Building the Target Executable

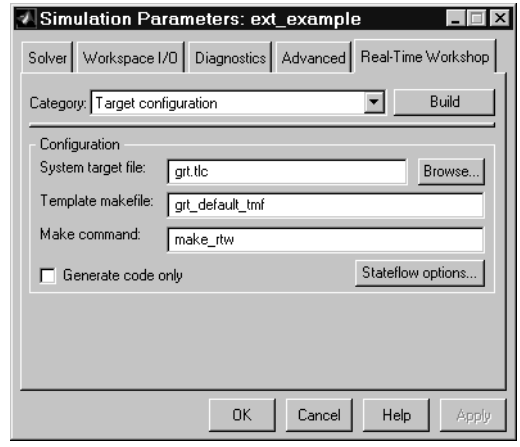
In this section, you set up the model and code generation parameters required for an external mode compatible target program. Then you generate code and build the target executable.

1. Open the **Simulation Parameters** dialog box. On the Solver pane, set the **Solver options Type** to `Fixed-step`, select the `discrete (no continuous states)` solver algorithm. Set **Fixed step size** to `0.01`. Leave the other parameters at their default values.

2. On the Workspace I/O pane, clear the **Time** and **Output** check boxes. In this exercise, data will not be logged to the workspace or to a MAT-file.

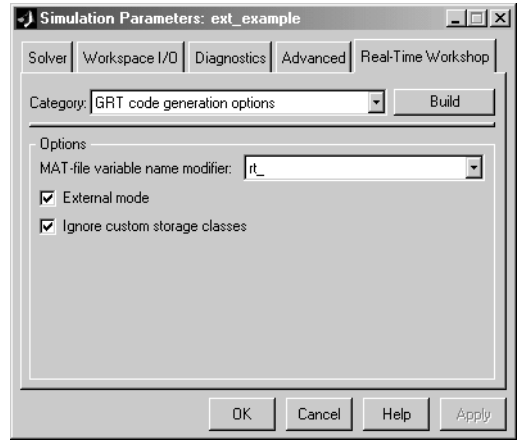
3. On the Real-Time Workshop pane, select `Target configuration` from the **Category** menu.

   By default, the GRT target should be selected, as shown in this figure.



   If the GRT target is *not* selected, click the **Browse** button and select the GRT target from the System Target File Browser. Then click **OK** to close the browser. Return to the Real-Time Workshop pane and click **Apply**.

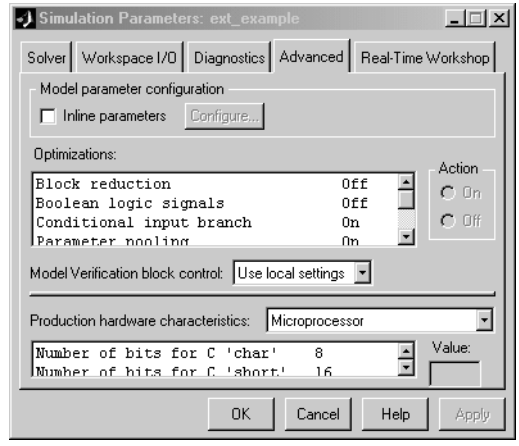
4 Select `GRT code generation options` from the **Category** menu and select the **External mode** option.This enables generation of external mode support code.



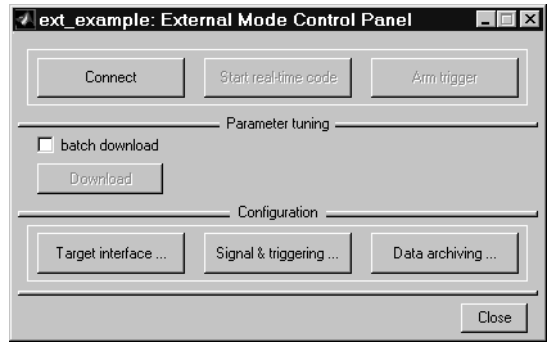
5 Click **Apply**.

6 On the Advanced pane, make sure that the **Inline parameters** option is not selected. External mode supports inlined parameters, but we will not be using inlined parameters in this tutorial.

The Advanced pane looks like the figure below.



7 From the **Tools** menu, select **External Mode Control Panel**. The **External Mode Control Panel** lets you configure host and target communications, 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 four buttons are for use after the target program has started. The three lower buttons open three separate dialog boxes:

- The **Target interface** button opens the **External Target Interface** dialog box, which configures the external mode communications channel.
- The **Signal & triggering** button opens the **External Signal & Triggering** dialog box, which configures which signals are viewed and how signals are acquired.
- 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 Chapter 6 of the Real-Time Workshop documentation for more information.

8 Click the **Target interface** button to open the **External Target Interface** dialog box. This dialog box configures the external mode interface options.

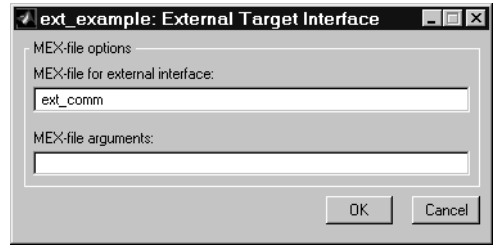
The **MEX-file for external interface** field specifies the name of a MEX-file that supports host and target communications on the host side. The default is `ext_comm`, a MEX-file provided by Real-Time Workshop. `ext_comm` supports communication via the TCP/IP communications protocol.

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 file you are using.

For information on these arguments, see "The External Interface MEX-File" in Chapter 6 of the Real-Time Workshop documentation.

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

The **External Target Interface** dialog box should appear as shown below.



9 Click **OK** to close the **External Target Interface** dialog box.

10 Close the **External Mode Control Panel**.

11 Save the model.

12 Return to the Real-Time Workshop pane. Click **Build** 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.

Part 3: 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.

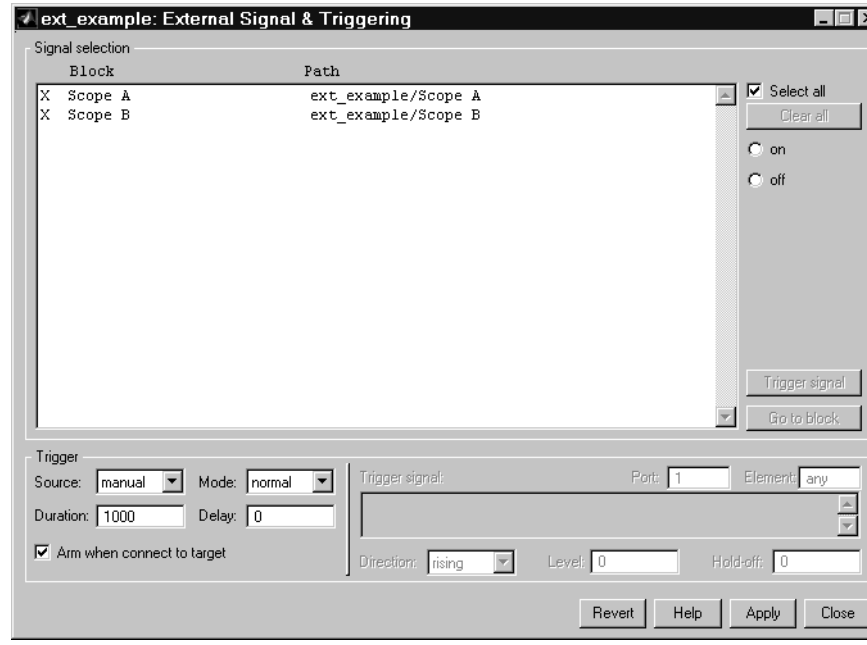
The **External Signal & Triggering** dialog box displays a list of all the blocks in your model that support external mode signal monitoring and logging. The **External Signal & Triggering** dialog box also lets you configure which signals are viewed and how they are acquired and displayed. You can reconfigure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

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

1. From the **Tools** menu, select **External Mode Control Panel**.

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

3. 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`.



4 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**.

5 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

```
ext_example -tf inf -w
```

and press **Return**. 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.

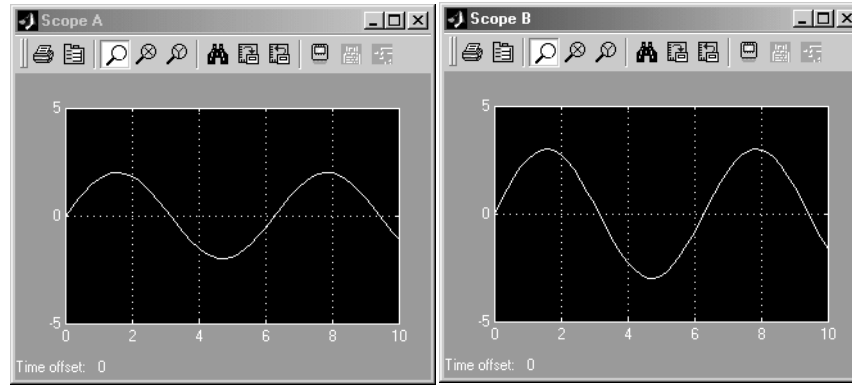
The `-w` switch instructs the target program to enter a wait state until it receives a **Start real-time code** message from the host. This switch is required if you want to view data from time step 0 of the target program execution, or if you want to modify parameters before the target program begins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

6. 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.

7. The model 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.

8. 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 caption of the **Connect** button changes to **Disconnect**.

9 Click the **Start real-time code** button.You should see the outputs of Gain blocks A and B 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 will do this in the next section.

Part 4: Tuning Parameters

You can change the gain factor of either Gain block by assigning new values to the variables `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 `ext_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 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 parameter dialog 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 will pause while the parameter dialog box is open. You must close the dialog by clicking **OK**, which will allow 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**.

A

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 — A 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.

Block target file — A 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 — An 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.

Embedded Real-Time (ERT) target – A target configuration that generates model code for execution on an independent embedded real-time system. Requires Real-Time Workshop Embedded Coder.

Expression folding — A 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 to 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`	Target Language Compiler	The generated C code
`.h`	Target Language Compiler	A C include header file used by the `.c` program

Extension	Created by	Description
`.mdl`	Simulink	Contains structures associated with Simulink block diagrams
`.mk`	Real-Time Workshop	A makefile specific to your model that is derived from the template makefile
`.rtw`	Real-Time Workshop	An intermediate compilation ("*model*`.rtw`") of a `.mdl` file used in generating 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	A project marker file inside a build directory that identifies the date and product version of generated code

Generic Real-Time (GRT) target — A target configuration that generates model code for a real-time system, with the resulting code executed on your workstation. (Note that execution is not tied to a real-time clock.) You can use GRT as a starting point for targeting custom hardware.

Host system — The computer system on which you create and may compile your real-time application.

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

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 S-function residing in an external MEX-file.

Interrupt Service Routine (ISR) — A 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 — A 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, etc., to interact. Makefiles are executed by your development system's make utility.

Multitasking — A process by which your microprocessor schedules the handling of multiple tasks. The number of tasks is equal to the number of sample times in your model.

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 an 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 — A 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.

Pseudomultitasking — n 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 — A computer that processes real-world events as they happen, under the constraint of a real-time clock, and which may implement algorithms in dedicated hardware. Examples include mobile telephones, test and measurement devices, and avionic and automotive control systems.

Run-time interface — A wrapper around the generated code that can be built into a stand-alone executable. The run-time interface consists of routines to move the time forward, save logged variables at the appropriate time steps, etc. The run-time interface is responsible for managing the execution of the real-time program created from your Simulink block diagram.

S-function — A 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 through preparing TLC scripts for them.

Simstruct — A Simulink data structure and associated application programming 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.

Singletasking — A mode in which a model runs in one task.

System target file — The entry point to the Target Language Compiler program, used to transform the Real-Time Workshop file into target specific code.

Target file — A 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.

Targeting — The process of creating software modules appropriate for execution on your target system.

Target Language Compiler (TLC) — A 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 in order to inline S-functions or to "wrap" existing user code.

Target system — The specific or generic computer system on which your real-time application executes.

Template makefile — A line-for-line makefile used by a make utility. The template makefile is converted 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.

Task identifier (tid) — In generated code, each sample rate in a multirate model is assigned a task identifier (`tid`). The `tid` is passed to the model output and update routines to control which portion of your model should execute at a given time step. Single-rate systems ignore the `tid`.

Virtual block — A 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.

Index

H

M

N

P

R

S

T

V

W