Real-Time Workshop®

For Use with SIMULINK®

Modeling

Simulation

Implementation

The MATH WORKS Inc.

User's Guide

Version 3

How to Contact The MathWorks:

508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail
`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`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
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Printing History:			
	May 1994	First printing	Version 1.0
	May 1997	Second printing	Version 2.1
	January 1999	Third printing	Version 3.0 (Release 11)

Contents

Foreword

3 Code Generation and the Build Process

8 Targeting Tornado for Real-Time Applications

10 Targeting Custom Hardware

11 Real-Time Workshop Libraries

12 Configuring Real-Time Workshop for Your Application

13 Real-Time Workshop Rapid Simulation Target

14 Real-Time Workshop Ada Coder

Foreword

Related Products and Documentation

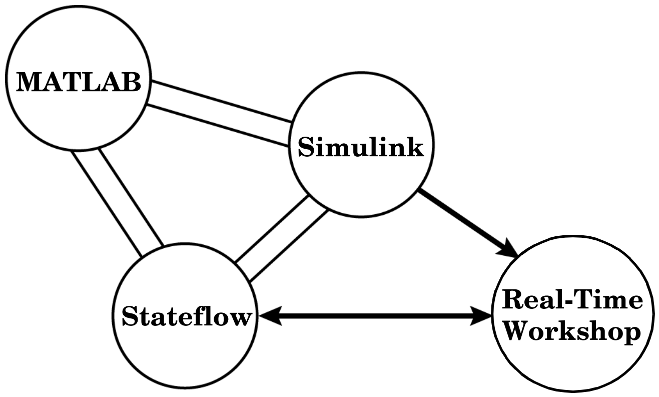
Requirements

The Real-Time Workshop® is a multiplatform product, running on Microsoft Windows 95, Windows 98, Windows NT, and UNIX systems.

The Real-Time Workshop requires:

- MATLAB® 5.3 (Release 11)
- Simulink® 3.0 (Release 11)

In addition, you can use Stateflow® to add finite-state machine models to your model. The Real-Time Workshop generates production quality code for models that consist of elements from MATLAB, Simulink, and Stateflow.



Several of the Real-Time Workshop targets, including generating a nonreal-time simulation program that runs on your workstation, require one of the following compilers for Windows:

- Microsoft Visual C/C++
- Watcom
- Borland

What Is MATLAB?

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state-of-the-art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

See the MATLAB documentation set for more information.

What Is Simulink?

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a far cry from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high-level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in MATLAB's command window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (for example, if you are doing Monte Carlo simulations or want to sweep a parameter across a range of values). Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, you can change parameters and immediately see what happens, for "what if" exploration. The simulation results can be put in the MATLAB workspace for postprocessing and visualization.

Model analysis tools include linearization and trimming tools, which can be accessed from the MATLAB command line, plus the many tools in MATLAB and its application toolboxes. And because MATLAB and Simulink are integrated, you can simulate, analyze, and revise your models in either environment at any point.

Stateflow is part of this environment. The Stateflow block is a masked Simulink model. Stateflow builds an S-function that corresponds to each

Stateflow machine. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow diagrams into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

The *Using Simulink* document describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

What Is Stateflow?

Stateflow is a powerful graphical design and development tool for complex control and supervisory logic problems. Stateflow supports flow diagram notation as well as state transition notation. Using Stateflow you can:

- Visually model and simulate complex reactive systems based on *finite state machine* theory.
- Design and develop deterministic, supervisory control systems.
- Use flow diagram notation and state transition notation seamlessly in the same Stateflow diagram.
- Easily modify your design, evaluate the results, and verify the system's behavior at any stage of your design.
- Automatically generate integer or floating-point code directly from your design.
- Take advantage of the integration with the MATLAB and Simulink environments to model, simulate, and analyze your system.

Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like `for` loops and `if-then-else` constructs.

Stateflow also provides clear, concise descriptions of complex system behavior using finite state machine theory, flow diagram notations, and state-transition diagrams. Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behavior.

The Stateflow® Coder is an add-on to Stateflow that supports code generation. It works in conjunction with Real-Time Workshop in these ways:

- Stateflow Coder generates high quality integer-based code (runs on a fixed-point processor; a floating-point processor is not required).
- Real-Time Workshop generates fixed-point (integer-based code) or floating-point-based code.
- Stateflow blocks are fully integrated for real-time code generation and compatible with Real-Time Workshop.

How to Use This Guide

Typographical Conventions

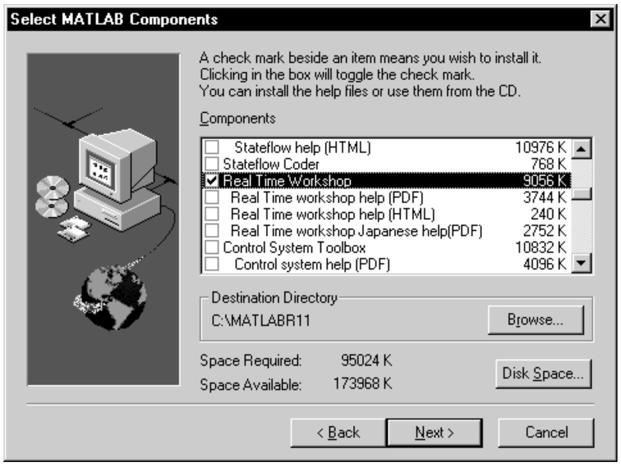
To Indicate...	This Guide Uses...	Example
New terms	*Italics*	An *array* is an ordered collection of information.
Keys, menu names, items, and GUI controls	**Boldface** with an initial capital letter	Choose the **File** menu.
Function names, variables, example code, user-entered text	`Monospace` type	The `sfopen` command opens a model.
User-supplied strings	*`Italic monospace`* type	The option `-I`*`path`* adds your path name to the list of paths in which to search for target files.

Installation

Your platform-specific MATLAB *Installation Guide* provides essentially all of the information you need to install the Real-Time Workshop.

Prior to installing the 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, a screen similar to this, where you can indicate which products to install, is displayed:



Select MATLAB Components

A check mark beside an item means you wish to install it.
Clicking in the box will toggle the check mark.
You can install the help files or use them from the CD.

Components

☐	Stateflow help (HTML)	10976 K
☐	Stateflow Coder	768 K
✓	Real Time Workshop	9056 K
☐	Real Time workshop help (PDF)	3744 K
☐	Real Time workshop help (HTML)	240 K
☐	Real Time workshop Japanese help(PDF)	2752 K
☐	Control System Toolbox	10832 K
☐	Control system help (PDF)	4096 K

Destination Directory: C:\MATLABR11

Space Required: 95024 K
Space Available: 173968 K

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

Licensed Product	**Prerequisite Products**	**Additional Information**
Simulink 3.0	MATLAB 5.3 (Release 11)	Allows installation of Simulink.
The Real-Time Workshop	Simulink 3.0 (Release 11)	Requires a Watcom C, Borland C, or Visual C/C++ compiler for creating MATLAB MEX-files on your platform.

If you experience installation difficulties and have Web access, connect to The MathWorks home page (`http://www.mathworks.com`). Look for the license manager and installation information under the `Tech Notes/FAQ` link under `Tech Support Info`.

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, the Real-Time Workshop must be able to access a compiler. The following sections describe how to configure your system so that the Real-Time Workshop has access to your compiler.

Watcom

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 Windows NT, in the control panel select **System**, go to the **Environment** page, and define `WATCOM` to be the path to your compiler.

Borland

The procedure for Borland compilers is exactly the same as for Watcom compilers. The only difference is that the environment variable is named `BORLAND`.

Microsoft Visual C/C++

Define the environment variable `MSDevDir` to be

```
MSDevDir=<path to compiler>                    for Visual C/C++ 4.2
MSDevDir=<path to compiler>\SharedIDE          for Visual C/C++ 5.0
MSDevDir=<path to compiler>\Common\MSDev98     for Visual C/C++ 6.0
```

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

As of this printing, we have tested the Real-Time Workshop with these compilers.

Compiler	**Versions**
Microsoft Visual C/C++	4.2, 5.0, 6.0
Watcom	10.6, 11.0
Borland	5.0, 5.2, 5.3

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.

Where to Go from Here

Chapter 1, “Introduction to the Real-Time Workshop,” is a quick introduction to the rapid prototyping process and the open architecture of the Real-Time Workshop.

Chapter 2, “Getting Started with the Real-Time Workshop,” discusses basic concepts and terminology of the Real-Time Workshop. It provides a road map that connects basic real-time development tasks to corresponding sections of this book, making it a good place to start if you are unsure of where you need to go in the documentation. Code validation and third-party environments (dSPACE, for example) are also discussed.

Chapter 3, “Code Generation and the Build Process,” describes the automatic program building process in detail, including a discussion of template makefiles. It also discusses the Real-Time Workshop’s graphical user interface (GUI).

Chapter 4, “External Mode,” contains information about external mode, which is a simulation environment that supports on-the-fly parameter tuning, signal monitoring, and data logging.

Chapter 5, “Data Logging and Signal Monitoring,” discusses how to save data from a simulation and monitor signal changes while a simulation is running.

Chapter 6, “Program Architecture,” discusses program architecture, including real-time program architecture, and the run-time interface.

Chapter 7, “Models with Multiple Sample Rates,” describes how to handle multirate systems.

Chapter 8, “Targeting Tornado for Real-Time Applications,” contains information that is specific to developing programs that target Tornado and signal monitoring using StethoScope.

Chapter 9, “Targeting DOS for Real-Time Applications,” contains information that is specific to developing programs that target DOS.

Chapter 10, “Targeting Custom Hardware,” discusses the targeting of custom hardware, including the use of S-functions as device drivers.

Chapter 11, “Real-Time Workshop Libraries,” contains information about the Real-Time Workshop Library, which is a collection of blocks and templates you can use to customize code generation for your application.

Chapter 12, “Configuring Real-Time Workshop for Your Application,” compares and contrasts various targets, including the generic real-time, embedded-C, and RTW S-function targets.

Chapter 13, “Real-Time Workshop Rapid Simulation Target,” discusses the Rapid Simulation Target (`rsim`), which executes your model in nonreal-time on your host computer. You can use this feature to generate fast, stand-alone simulations that allow batch parameter tuning and the loading of new simulation data (signals) from a standard MATLAB MAT-file without needing to recompile your model.

Chapter 14, “Real-Time Workshop Ada Coder,” discusses the Real-Time Workshop Ada Target, which generates Ada code from your models. This is a separate product from the Real-Time Workshop.

Appendix A lists the directory structure and the files shipped with the Real-Time Workshop.

Appendix B is a glossary that contains definitions of terminology associated with the Real-Time Workshop and real-time development.

Note See the *Target Language Compiler Reference Guide* for information about TLC files, including a discussion of custom target files.

Introduction to the Real-Time Workshop

Introduction

The Real-Time Workshop, for use with MATLAB and Simulink, produces code directly from Simulink models and automatically builds programs that can be run in a variety of environments, including real-time systems and stand-alone simulations.

With the Real-Time Workshop, you can run your Simulink model in real-time on a remote processor. The Real-Time Workshop also enables you to run high-speed stand-alone simulations on your host machine or on an external computer.

Using the rapid prototyping process, you can shorten development cycles and reduce costs. You can also use the Real-Time Workshop to implement hardware-in-the-loop (HIL) simulations.

This chapter presents an introduction to the Real-Time Workshop and describes its relationship to Simulink and MATLAB. The chapter concludes with a summary of the material contained in this manual and a description of the examples that are bundled with the product.

The Real-Time Workshop

The Real-Time Workshop provides a real-time development environment that features:

- A rapid and direct path from system design to implementation.
- Seamless integration with MATLAB and Simulink.
- A simple, easy to use interface.
- An open and extensible architecture.
- A fully configurable code generator — virtually every aspect of the generated code can be configured by using the Target Language Compiler™.
- Fast design iterations by editing block diagrams and automatically building a new executable.

The package includes application modules that allow you to build complete programs targeting a wide variety of environments. Program building is fully automated.

Real-Time Workshop Applications

The Real-Time Workshop is designed for a variety of applications. Some examples include:

- Real-time control — You can design your control system using MATLAB and Simulink and generate code from your block diagram model. You can then compile and download it directly to your target hardware.
- Real-time signal processing — You can design your signal processing algorithm using MATLAB and Simulink. The generated code from your block diagram can then be compiled and downloaded to your target hardware.
- Hardware-in-the-loop simulation — You can create Simulink models that mimic real-life measurement, system dynamics, and actuation signals. Generated code from the model can be targeted on special purpose hardware to provide a real-time representation of the physical system. Applications include control system validation, training simulation, and fatigue testing using simulated load variations.
- Interactive real-time parameter tuning — You can use Simulink as a front end to your real-time program. This allows you to change parameters while the program is executing.

- High-speed stand-alone simulations.
- Generation of portable C code for export to other simulation programs.

The Generated Code

The generated code (i.e., the model code) is by default highly optimized and fully commented C code that can be generated from any Simulink model, including linear, nonlinear, continuous, discrete, or hybrid models.

All Simulink blocks are automatically converted to code, with the exception of MATLAB function blocks and S-function blocks that invoke M-files. You must rewrite these blocks as C MEX S-functions if you want to use them with the Real-Time Workshop.

The Real-Time Workshop includes a set of target files that are compiled by the Target Language Compiler (TLC) to produce ANSI C code. The target files are ASCII text files that describe how to convert the Simulink model to code. For advanced users, target files enable you to customize generated code for individual blocks or throughout the entire model. See the *Target Language Compiler Reference Guide* for more information about customizing target files.

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to *inline* the S-function, thus improving performance by eliminating function calls to the S-function itself.

Types of Output

The Real-Time Workshop's interface allows you to select various types of output:

- C code — Generate code that contains system equations and initialization functions for the Simulink model. You can use this code in nonreal-time simulation environments or for real-time applications.
- Ada code — Generate Ada code from your Simulink model. This option requires that you have installed the Real-Time Workshop Ada Coder, which is a separate product. See Chapter 14, "Real-Time Workshop Ada Coder," for more information.
- A real-time program — Transform the generated code into a real-time program suitable for use with dedicated real-time hardware. The resulting code is designed to interface with an external clock source and hence runs at

a fixed, user-specified sample rate. Continuous time models are incorporated into this code with the simple provision that their states are propagated using fixed-step size integration algorithms.

- A high-performance stand-alone simulation — use the generated code with the generic real-time system target file to produce an executable for stand-alone simulations. At the end of the simulation, the executable produces a `model.mat` file that contains the MATLAB variables that were logged in Simulink. This MAT-file is used for analysis in MATLAB.

  You can also use the stand-alone generic real-time simulation environment for code validation since you can directly compare results against Simulink.

The Rapid Prototyping Process

The Real Time Workshop allows you to do *rapid prototyping*, a process that allows you to conceptualize solutions using a block diagram modeling environment and take an early look at system performance prior to laying out hardware, writing any production software, or committing to a fixed design.

By using rapid prototyping, you can refine your real-time system by continuously iterating your model. Further, you can tune parameters on the fly by using Simulink as the front-end of your real-time model. This is known as using Simulink in *external mode*.

Key Aspects of Rapid Prototyping

The key to rapid prototyping is automatic code generation. It reduces algorithm coding to an automated process; this includes coding, compiling, linking, and downloading to target hardware. This automation allows design changes to be made directly to the block diagram. This figure show the rapid prototyping development process:

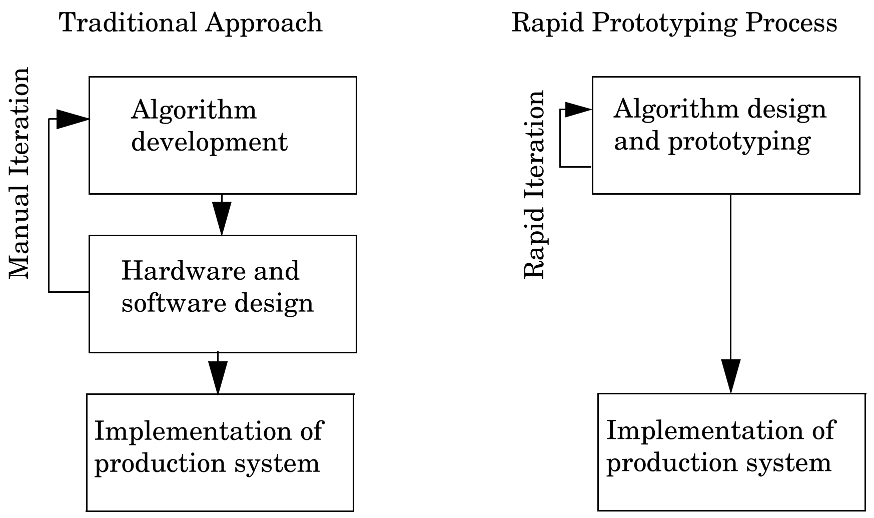

Traditional Approach
Rapid Prototyping Process
Manual Iteration
Algorithm development
Hardware and software design
Implementation of production system
Rapid Iteration
Algorithm design and prototyping
Implementation of production system

Figure 1-1: Comparison of Traditional and Rapid Prototyping Development Processes

The traditional approach to real-time design and implementation typically involves multiple teams of engineers, including an algorithm design team, software design team, hardware design team, and an implementation team. When the algorithm design team has completed its specifications, the software design team implements the algorithm in a simulation environment and then specifies the hardware requirements. The hardware design team then creates the production hardware. Finally, the implementation team integrates the hardware into the larger overall system. This approach leads to long development processes, because the algorithm design engineers do not work with the actual hardware. The rapid prototyping process combines the algorithm, software, and hardware design phases, thus eliminating potential bottlenecks. The process allows engineers to see the results and rapidly iterate on the design before expensive hardware is developed.

The rapid prototyping process begins in Simulink. First, you develop a model in Simulink. In control engineering, this involves modeling plant dynamics and including additional dynamic components that constitute a controller and/or an observer. In digital signal processing, the model is typically an exploration of the signal-to-noise ratio and other characteristics of the input signal. You then simulate your model in Simulink; you can use MATLAB, Simulink, and toolboxes to aid in the development of algorithms and analysis of the results. If the results are not satisfactory, you can iterate the modeling/analysis process until results are acceptable.

Once you have achieved the desired results, you can use the Real-Time Workshop to generate downloadable C code (for the appropriate portions of the model). Using Simulink in external mode, you can tune parameters and further refine your model, again rapidly iterating to achieve required results. At this stage, the rapid prototyping process is complete. You can begin the final implementation for production with confidence that the underlying algorithms work properly in your real-time production system.

The figure below shows the rapid prototyping process in more detail.

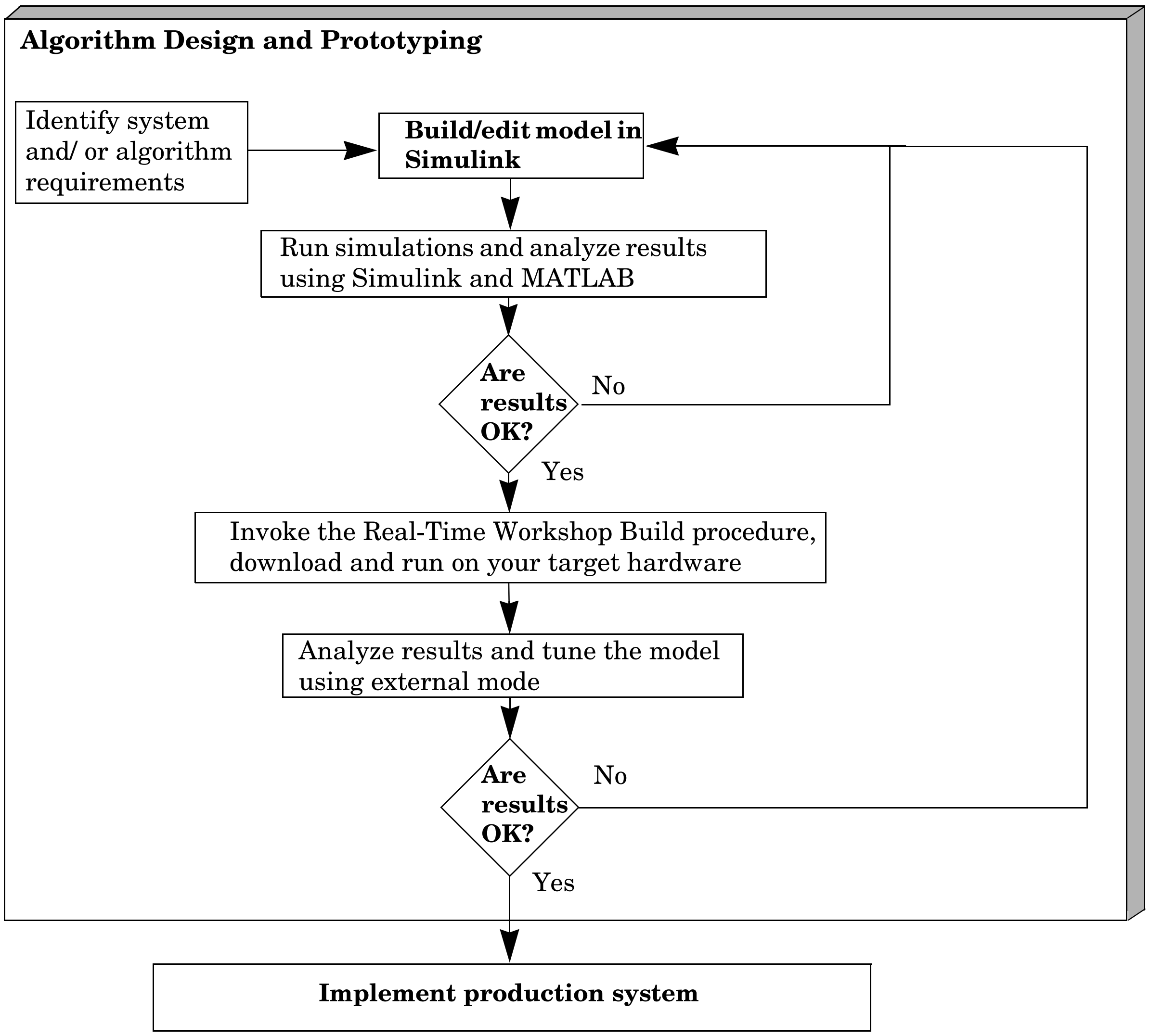

Algorithm Design and Prototyping
Identify system and/ or algorithm requirements
Build/edit model in Simulink
Run simulations and analyze results using Simulink and MATLAB
Are results OK?
No
Yes
Invoke the Real-Time Workshop Build procedure, download and run on your target hardware
Analyze results and tune the model using external mode
Are results OK?
No
Yes
Implement production system

Figure 1-2: The Rapid Prototyping Development Process

This highly productive development cycle is possible because the Real-Time Workshop is closely tied to MATLAB and Simulink. Each package contributes to the design of your application:

- MATLAB — Design, analysis, and data visualization tools
- Simulink — System modeling, simulation, and validation
- Real-Time Workshop — C code generation from Simulink model and framework for running generated code in real-time, tuning (modifying) parameters, and viewing real-time data

Rapid Prototyping for Digital Signal Processing

The first step in the rapid prototyping process for digital signal processing is to consider the kind and quality of the data to be worked on and to relate it to the system requirements. Typically this includes examining the signal-to-noise ratio, distortion, and other characteristics of the incoming signal, and relating them to algorithm and design choices.

System Simulation and Algorithm Design

In the rapid prototyping process, the role of the block diagram in algorithm development is twofold. It supplies a way to identify processing bottlenecks and to optimize the algorithm or system architecture. It also provides a high-level system description, that is, a hierarchical framework for evaluating the behavior and accuracy of alternative algorithms under a range of operating conditions.

Analyzing Results, Parameter Tuning, and Signal Monitoring Using External Mode

Once an algorithm (or a set of candidate algorithms) has been created, the next stage is to consider architectural and implementation issues such as complexity, speed, and accuracy. In the conventional case, this meant recoding the algorithm in C or in a hardware design and simulation package.

After building the executable and downloading it to your hardware, you can tune (modify) block parameters in Simulink and automatically download the new values to the hardware. To change these parameters from the Simulink block diagram, you can run Simulink in external mode. Simulink's external mode allows you to change parameters interactively without stopping the real-time execution of your signal processing algorithms on the hardware.

You can monitor signals using Scope blocks while running external mode. Simply connect Scope blocks to signals that you want to monitor. You can then view signals changing as you tune parameters on-the-fly.

Rapid Prototyping for Control Systems

Rapid prototyping for control systems is similar to digital signal processing, with one exception: in control systems design, it is necessary to develop a model of your plant prior to algorithm development in order to simulate closed loop performance. Once a simulation has been developed that models the plant with sufficient accuracy, the rapid prototyping process for control system design continues in much the same manner as digital signal processing design.

Rapid prototyping begins with developing block diagram plant models of sufficient fidelity for preliminary system design and simulation. Once simulations show encouraging system performance, the controller block diagram is separated from the plant mode and I/O device drivers are attached. Using automatic code generation, the entire system is immediately converted to real-time executable code. The executable can be automatically loaded onto target hardware, allowing the implementation of real-time control systems in very short time.

Modeling Systems in Simulink

You can use MATLAB and Simulink to design, test, and analyze a model of your system. The first step in the design process is to develop a plant model. You can build models involving plant, sensor, and actuator dynamics using the Simulink collection of linear and nonlinear components. Because Simulink is customizable, you can further simplify modeling by creating custom blocks and block libraries from continuous and discrete-time components.

You can also use the Systems Identification Toolbox to analyze test data to develop an empirical plant model, or the Symbolic Math Toolbox to translate the equations of the plant dynamics into state-variable form.

Analyze Results of the Simulation

You can use MATLAB and Simulink to analyze the results produced from the model you developed in the first step of the rapid prototyping process. It is at this stage that you can design and add a controller to your plant.

Algorithm Design and Analysis

From the block diagrams developed during the modeling stage, you can extract state-space models through linearization techniques. These matrices can be used in control system design. You can use these toolboxes to facilitate control system design, and work with the matrices that you derived:

- Control System Toolbox
- Robust Control Toolbox
- Model Predictive Control Toolbox
- μ-Analysis and Synthesis Toolbox
- LMI Control Toolbox
- QFT Control Toolbox

Once you have your controller designed, you can create a closed-loop system by connecting it to the Simulink plant model. Closed-loop simulations allow you to determine how well the initial design meets performance requirements.

Once you have a model that you're satisfied with, it is a simple matter to generate C code directly from the Simulink block diagram, compile it for the target processor, and link it with supplied or user-written application modules.

Analyzing Results, Parameter Tuning, and Signal Monitoring Using External Mode

You can load the program data into MATLAB for analysis or display the data with third party monitoring tools. You can easily make design changes to the Simulink model and then regenerate the C code.

After building the executable and downloading it to your hardware, you can tune (modify) block parameters in Simulink and automatically download the new values to the hardware. To change these parameters from the Simulink block diagram, you can run Simulink in external mode. Simulink's external mode allows you to change parameters interactively without stopping the real-time execution of the model on the hardware.

You can monitor signals using Scope blocks while running external mode. Simply connect Scope blocks to signals that you want to monitor. You can then view signals changing as you tune parameters on-the-fly.

Open Architecture of the Real-Time Workshop

The Real-Time Workshop is an open and extensible system designed for use with a wide variety of operating environments and hardware types. Its features include:

- The ability to generate code from any fixed-step Simulink block diagram
- A framework for building real-time programs
- An extensible device driver library supporting a variety of hardware
- Automatic program building and a fully customizable build process
- Bundled sample implementations in DOS, Tornado, and generic real-time environments
- Support for third-party hardware and tools
- Fully customizable code generation, including inlined custom blocks
- Automatic loop-rolling, which allows vectorized operations to be expanded out or placed in a `for` loop, depending upon a rolling threshold that you can set
- Automatic function inlining, which allows the direct embedding of functions in the generated code. This feature eliminates the overhead of a function call.
- The ability to configure the build process to virtually any version of `make`. The Real Time Workshop includes examples that use Watcom C/C++, Microsoft Visual C/C++, and UNIX versions of `make`.
- Multiple code formats that you can use for rapid prototyping and embedded systems applications
- Ability to interface parameters and signals outside of the Real-Time Workshop (that is, interface with hand-written code)
- Fully configurable Real-Time Workshop graphical user interface

You can customize the code generation and build process of the Real-Time Workshop. The following picture highlights the open architecture of the Real-Time Workshop. The next few pages discuss concepts in this figure:

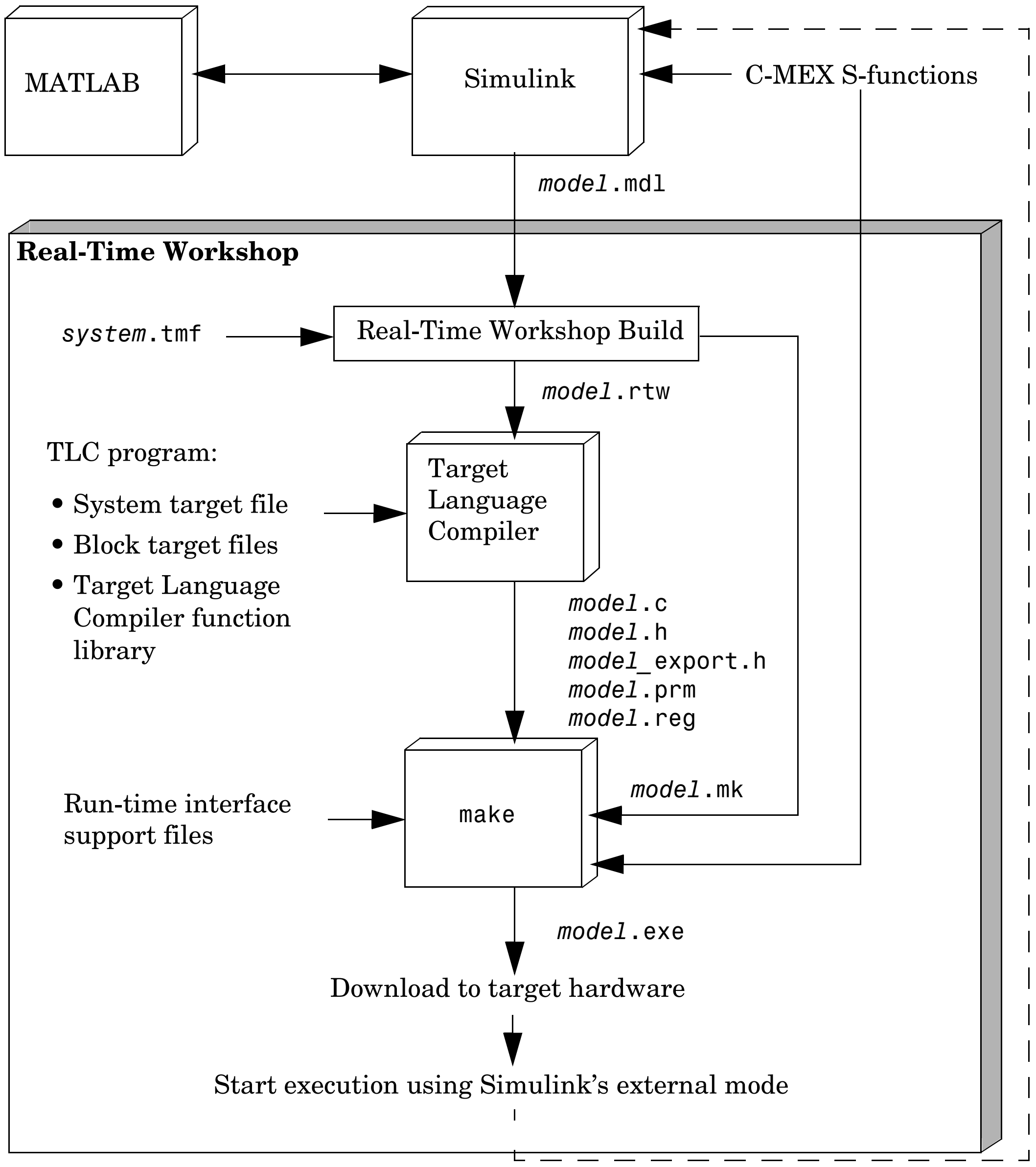


Figure 1-3: The Real-Time Workshop's Open Architecture

Target Language Compiler

To generate code, the Real-Time Workshop invokes the Target Language Compiler (TLC). The Target Language Compiler transforms an intermediate model description generated by the Real-Time Workshop of your Simulink block diagram into target specific code. The intermediate description of your model is saved in an ASCII file called `model.rtw`.

The Target Language Compiler allows you to modify most aspects of the generated code. The compiler reads the `model.rtw` file and executes a TLC program that consists of a set of *target files* (`.tlc` files). These are ASCII files written for the Target Language Compiler. The *TLC program* specifies how to transform the `model.rtw` file into generated code.

The TLC program consists of:

- The entry point or main file. This is called the *system target file*.
- A set of *block target files*. These specify how to translate each block in your model into target-specific code.
- A *Target Language Compiler function library*. This is a set of library functions that the TLC program uses when converting the `model.rtw` file into generated code.

The complete TLC program is provided with the Real-Time Workshop.

If you are familiar with HTML, Perl, and MATLAB, you may see similarities between these and the Target Language Compiler. The Target Language Compiler has the mark-up language features, like HTML, the power and flexibility of Perl and other scripting languages, and the data handling power of MATLAB. The Target Language Compiler is designed for one purpose — to convert the model description file, `model.rtw`, into target-specific code.

Make Utility

The Real-Time Workshop invokes `make` to build the real-time executable. `Make` is a utility that compiles and links the generated code to create an executable. The `model.mk` makefile, which the system template makefile, `system.tmf`, creates, in turn invokes `make`. You can fully configure the `make` utility by modifying the system template makefile.

The Real-Time Workshop passes `model.mk` to the make utility, which directs the compilation and linking of model code along with any libraries or user provided modules.

S-Functions

S-functions allow you to add custom code to your Simulink model. You can embed the code directly into the generated code or, by default, allow the S-function Application Program Interface (API) to call the S-function. Embedding the S-function code directly into the generated code is called *inlining* the S-function. The default is called *noninlined*.

For more information on S-functions, see *Using Simulink*. For information on inlining S-functions see the *Target Language Compiler Reference Guide*.

The Target Language Compiler is the key for customizing the generated code. Customizations usually involve inlining S-functions.

The Build Procedure

The open and modular structure of the Real-Time Workshop allows you to configure the Real-Time Workshop for your environment, or use an existing environment, such as dSPACE or Wind River Systems Tornado Development Environment, provided by a third-party vendor.

The Real-Time Workshop build procedure first creates the `model.rtw` file, which is an intermediate representation of a Simulink block diagram that contains information such as the parameter values, vector widths, sample times, and execution order for the blocks in your model. This information is stored in a language-independent format.

After creating the `model.rtw` file, the build procedure invokes the Target Language Compiler to transform it into target-specific code. The Target Language Compiler begins by reading in the `model.rtw` file. It then compiles and executes the commands in the target files. The target files (referred to as `.tlc` or TLC files) specify how to transform the `model.rtw` file into target-specific code. The Target Language Compiler starts execution with the system target file. It then loads the individual block target files to transform the block information in the `model.rtw` file into target-specific code for the blocks. The output of the Target Language Compiler is a source code version of the Simulink block diagram.

The Target Language Compiler includes a Target Language Compiler function library, which is a set of routines for use by the various target files.

The next step of the build procedure is to create a system makefile (`system.mk`) from a template makefile (`system.tmf`). `system.tmf` is the template makefile for your target environment; it allows you to specify compilers, compiler options, and additional information for the destination (target) of the generated executable.

The *model*.mk file is created by copying the contents of *system*.tmf and expanding out tokens describing your model's configuration. You can fully customize your build process by modifying an existing template makefile or providing your own template makefile.

After the *model*.mk file is created, the make command is invoked to create your executable. make can also optionally download the executable to your target hardware.

After downloading the file to the target hardware, if you are using external mode, you can connect back with Simulink to tune your model's parameters while the code is running on your target hardware.

Files Created by the Build Procedure

Each of the *model*.* files performs a specific function in the Real-Time Workshop:

- *model*.mdl, created by Simulink, is analogous to a high-level programming language source file.
- *model*.rtw, created by the Real-Time Workshop build process, is analogous to the object file created from a high-level language.
- *model*.c, created by the Target Language Compiler, is the C source code corresponding to the model.mdl file.
- *model*.h, created by the Target Language Compiler, is a header file that maps the links between blocks in the model.
- *model*_export.h, created by the Target Language Compiler, is a header file that contains exported signal, parameter, and function symbols.
- *model*.prm, created by the Target Language Compiler, contains the parameter settings of the blocks in the model.
- *model*.reg, created by the Target Language Compiler, contains the model registration function responsible for model initialization.

A First Look at Real-Time Workshop Files

An example of a Simulink model is this block diagram:

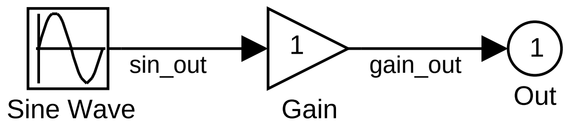


Figure 1-4: A Simple Simulink Model

This model is saved in a file called `example.mdl`. Generating C code from `example.mdl` is done by using the Real-Time Workshop user interface for generating code and executables. The next chapter, “Getting Started with the Real-Time Workshop,” explains the details of the user interface.

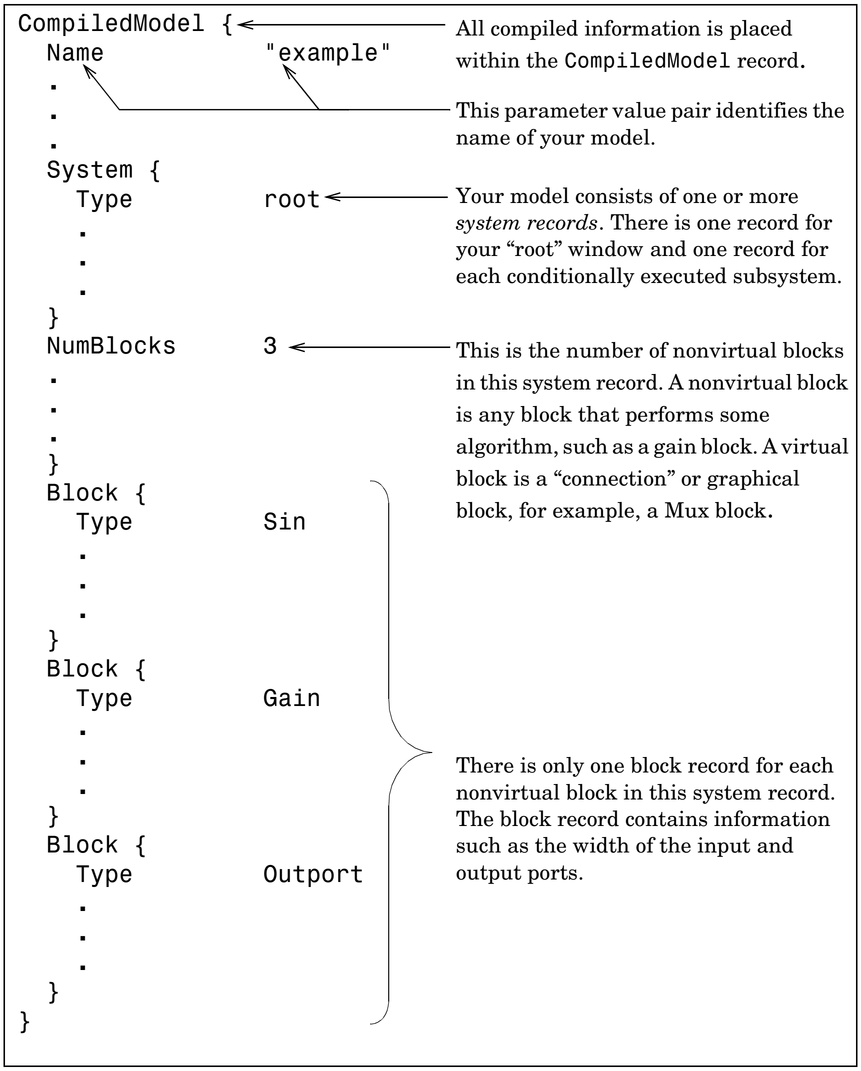
Later in this section are excerpts from the associated `.rtw` files that the Real-Time Workshop uses to generate the real-time version of this model in C.

Basic Features of example.rtw

When you invoke the Real-Time Workshop build procedure to generate code, it first compiles your model. This compilation process consists of these tasks:

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

The Real-Time Workshop writes this information out to `example.rtw`. The `example.rtw` file is an ASCII file consisting of parameter value pairs stored in a hierarchical structure consisting of records. Below is an excerpt from example.rtw, which is the .rtw file associated with “A Simple Simulink Model” on page 1-17:



```
CompiledModel {
  Name              "example"
  .
  .
  .
  System {
    Type            root
    .
    .
    .
  }
  NumBlocks         3
  .
  .
  .
  }
  Block {
    Type            Sin
    .
    .
    .
  }
  Block {
    Type            Gain
    .
    .
    .
  }
  Block {
    Type            Outport
    .
    .
    .
  }
}
```

All compiled information is placed within the CompiledModel record.

This parameter value pair identifies the name of your model.

Your model consists of one or more *system records*. There is one record for your “root” window and one record for each conditionally executed subsystem.

This is the number of nonvirtual blocks in this system record. A nonvirtual block is any block that performs some algorithm, such as a gain block. A virtual block is a “connection” or graphical block, for example, a Mux block.

There is only one block record for each nonvirtual block in this system record. The block record contains information such as the width of the input and output ports.

Note that *model*.rtw files are similar in appearance to *model*.mdl files generated by Simulink. For more information on .rtw files, see the *Target*

Language Compiler Reference Guide, which contains detailed descriptions of the contents of *`model`*`.rtw` files.

Basic Features of example.c

Using the `example.rtw` file and target files, the Target Language Compiler creates C code that you can use in stand-alone or real-time applications. The generated C code consists of procedures that must be called by your target's execution engine.

These procedures consist of the algorithms defined by your Simulink block diagram. The execution engine executes the procedures as time moves forward. In addition to executing the generated procedures, your target may provide capabilities such as data logging. The modules that implement the execution engine and other capabilities are referred to collectively as the *run-time interface modules*.

For our example, the generated `MdlOutputs` function, which is where the actual algorithm (multiplying a sine wave by a gain) occurs, is shown below.

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

`MdlOutputs` must be called at every time step by the run-time interface. This example demonstrates the efficiency of generated Real-Time Workshop code. The code in `MdlOutputs` reuses `rtb_temp0`, a temporary buffer, rather than assigning buffers to each input and output. Compare this to the `MdlOutputs` function generated with buffer optimizations turned off.

```
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);
  /* Gain Block: <Root>/Gain */
  rtB.gain_out = rtB.sin_out * rtP.Gain_Gain;
  /* Outport Block: <Root>/Out1 */
  rtY.Out1 = rtB.gain_out;
}
```

Note that the labels `sin_out` and `gain_out` that were placed in the model depicted in Figure 1-4 appear in the code of `MdlOutputs` when no optimizations take place. For further information on the contents of *`model`*`.c` files, refer to Chapter 3, “Code Generation and the Build Process.”

Bundled Target Systems

The Real-Time Workshop provides sample implementations that illustrate the development of real-time programs under DOS and Tornado, as well as generic real-time programs under Windows95, Windows 98, Windows NT, and UNIX.

These sample implementations are located in

- *`matlabroot`*`/rtw/c/grt` — Generic real-time examples
- `matlabroot/rtw/c/dos` — DOS examples
- `matlabroot/rtw/c/tornado` — Tornado examples

In addition to bundled systems provided by The MathWorks, third party vendors such as dSPACE provide real-time targets for a wide variety of rapid prototyping environments. For an example of a dSPACE implementation, see “Building a Real-Time Executable with dSPACE’s RTI” on page 2–30.

Using Stateflow and Blocksets with the Real-Time Workshop

Stateflow is a graphical design and development tool for complex control and supervisory logic problems. It supports flow diagram notation as well as state transition notation. The Stateflow Coder is an add-on to Stateflow that supports code generation and seamless integration with the Real-Time Workshop. You can include Stateflow blocks in your Simulink model; the Real-Time Workshop in conjunction with the Stateflow Coder generates code from them in much the same manner as the Real-Time Workshop does for Simulink blocks and S-functions. Refer to the *Stateflow User's Guide* for information about Stateflow and the Stateflow Coder.

In addition to Stateflow, these toolboxes and blocksets are compatible with the Real-Time Workshop:

- DSP Blockset
- The Communications Toolbox — all Simulink blocks in the toolbox
- Fixed Point Blockset

Getting Started with the Real-Time Workshop

Introduction

This chapter begins a discussion of basic concepts used in real-time development and relates them to concepts in the Real-Time Workshop. You can use the first section as a road map to the rest of the documentation.

After defining a map from real-time development tasks to sections of this book, the chapter discusses generic real-time in more detail and includes an example of how to generate stand-alone C code from your Simulink block diagram.

The chapter continues with a description of what to do based on target type, including a brief discussion of some supported hardware and concludes with some information about the dSPACE target.

Where to Find Information in This Manual

This section discusses typical requirements for producing real-time software and provides a road map to areas in this documentation that address key real-time issues. The Real-Time Workshop has support for all the following tasks that real-time engineers need to do. Each item below has a corresponding subsection that tells you where to find the information necessary to accomplish your tasks using the Real-Time Workshop.

With the Real-Time Workshop, you have the ability to:

1. Develop single- and multitasking code
2. Customize the generated code
3. Optimize the generated code
4. Validate the generated code against the Simulink simulation
5. Merge generated code into an existing code base. This means interfacing signals, parameters, and functions with handwritten legacy code.
6. Merge legacy code into the Real-Time Workshop generated code
7. Communicate with hardware device drivers (e.g., A/D converter)
8. Trace the generated code back to your Simulink diagram
9. Create an automatic build procedure to generate, (cross) compile, and optionally download a model to a real-time target
10. Tune parameters on-the-fly without recompiling the source code
11. Monitor signals and data within the generated code
12. Interface signals and parameters from Real-Time Workshop generated code into hand-written code

The MathWorks has created software for both rapid prototyping and deploying an embedded real-time control system. The Real-Time Workshop generates callable procedures for single and multirate systems, suitable for single- or multitasking environments. To execute the generated code you must

- Write a harness program (i.e., main)
- Install a C compiler

A bundled harness program is supplied for the Tornado and DOS operating systems. That is, these harnesses can be used without modification. More generic harnesses are supplied as starting points for targeting a different operating system. The *Real-Time Workshop User's Guide* addresses each of the items identified in this overview to assist you in making the generated code work for your engineering application.

The following sections map real-time development tasks to corresponding sections of this book.

1. Single- and Multitasking Code Generation

The Real-Time Workshop fully supports single- and multitasking code generation. See Chapter 7, "Models with Multiple Sample Rates," for a complete description.

2. Customizing Generated Code

The Real-Time Workshop provides a Custom Code library that supports customization of the generated code. See "Custom Code Library" on page 11-4 for a description of this library.

An alternative approach to customizing generated code is to modify Target Language Compiler^{TM} (TLC) files. The Target Language Compiler is a interpreted language that translates Simulink models into C code. Using the Target Language Compiler, you can customize the generated code.

There are two TLC files, `hookslib.tlc` and `cachelib.tlc`, that contain functions you can use to customize Real-Time Workshop generated code. See Chapters 3 and 4 of the *Target Language Reference Guide f*or more information about TLC files. The following functions are available in `hookslib.tlc`:

- `LibCacheDefine`, `LibCacheExtern`, `LibCacheFunctionPrototype`, `LibCacheIncludes`, `LibCacheNonFiniteAssignment`, `LibCacheTypeDefs`

The following functions are available in `cachelib.tlc`:

- `LibHeaderFileCustomCode`, `LibPrmFileCustomCode`, `LibRegFileCustomCode`, `LibSourceFileCustomCode`, `LibMdlStartCustomCode`, `LibMdlTerminateCustomCode`, `LibMdlRegCustomCode`, `LibSystemInitializeCustomCode`, `LibSystemOutputCustomCode`, `LibSystemUpdateCustomCode`, `LibSystemDerivativeCustomCode`, `LibSystemEnableCustomCode`, `LibSystemDisableCustomCode`, `LibSystemUserCodeIsEmpty`

Alternatively, you can look at `hookslib.tlc` and `cachelib.tlc` for brief descriptions of these functions. The TLC files are located in *`matlabroot`*`/rtw/c/tlc`.

3. Optimizing Generated Code

The default settings are generic for flexible rapid prototyping systems. The penalty for this flexibility is code that is less than optimal. There are several optimization techniques that you can use to minimize the source code size and memory usage once you have a model that meets your requirements.

For the embedded-C target, see "Embedded-C Code Format" on page 12-14. This section contains information about optimization specifically for the Embedded-C code format.

See "Optimizations Common to All Code Formats" on page 12-21 for a complete discussion of code optimization techniques available for all the code formats.

4. Validating Generated Code

Using the generic real-time (`grt`) environment provided by The MathWorks, you can create an executable that runs on your workstation and creates a *`model`*`.mat` file. You can then compare the results of the generic real-time simulation with the results of running a Simulink simulation.

For more information on how to validate Real-Time Workshop generated code, see "Blocks That Depend on Absolute Time" on page 2-21.

5. Incorporating Generated Code into Legacy Code

If your Real-Time Workshop generated code is intended to be a piece of an existing code base (for example, if you want to use the generated code as a plug-in function), you should use the embedded-C code format. See "Embedded-C Code Format" on page 12-14 for a discussion of this form of code generation.

6. Incorporating Legacy Code into Generated Code

Legacy code, or hand-written code, is existing code that you want to interface with Real-Time Workshop generated code. To interface legacy code with the Real-Time Workshop, you can use an S-function wrapper to do this; see Chapter 4 of *Writing S-Functions* for more information.

7. Communicating With Device Drivers

S-functions provide a flexible method for communicating with device drivers. See “Implementing Device Drivers” on page 10-7 for a description of how to build device drivers. Also, for a complete discussion of S-functions, see *Writing S-Functions*.

8. Code Tracing

The Real-Time Workshop includes special tags throughout the generated code that make it easy to trace the code back to your Simulink model. See “Tracing Generated Code Back to Your Simulink Diagram” on page 2-26 for more information about this feature.

9. Automatic Build Procedure

The Real-Time Workshop is designed so that you can generate code with the push of a button. The automatic build procedure generates code, a template make file, and optionally (cross-) compiles and downloads a program all via the **Build** button on the Real-Time Workshop page of the **Simulation Parameters** dialog box. See “Automatic Program Building” on page 3-3 for a complete description.

10. Parameter Tuning

Parameter tuning is the ability to change parameters on-the-fly without recompiling the generated code. The Real-Time Workshop supports parameter tuning in four different environments:

- External mode — you can tune parameters from Simulink while running the generated code on a target processor. See Chapter 4, "External Mode," for information on this mode.
- External C application program interface (API) — you can write your own C API interface using support files provided by The MathWorks. See "C API for Parameter Tuning" on page 3-19 for more information.
- Rapid Simulation — you can use the Rapid Simulation Target (`rsim`) in batch mode to provide fast simulations for performing parametric studies. Although this is not an on-the-fly application of parameter tuning, it is nevertheless a useful way to evaluate a model. This mode is also useful for Monte Carlo simulation.
- Simulink — prior to generating real-time code, you can tune parameters on-the-fly in your Simulink model.

11. Monitoring Signals and Logging Data

There are several ways to monitor signals and data in the Real-Time Workshop:

- External mode — you can monitor signals by selecting Scope blocks. See "Signals & Triggering" on page 4-7 for a discussion of this method.
- External C API — you can write your own C API using support files provide by the MathWorks. See "C API for Signal Monitoring" on page 5-4 for more information.
- MAT-file logging — you can use MAT-file to log data in the generated executable. See "MAT-File Data Logging" on page 5-2 for more information.
- Simulink — you can use any of Simulink's data logging capabilities.

12. Interfacing Signals and Parameters

You can interface signals and parameters in your model to hand-written code by changing the storage class of signals in your model. For more information, see:

- "Signal Properties" on page 3-19
- "Tunable Parameters" on page 3-18

Basic Concepts in the Real-Time Workshop

Before going through a step-by-step example of how to target specific hardware using the Real-Time Workshop, it is important to understand concepts basic to the product. These include:

- Generic real-time
- Targeting
- Target Language Compiler files
- The Build process
- Template makefiles
- Configuring the template makefile
- Specifying model parameters

Generic Real-Time

The Real-Time Workshop provides a generic real-time development target. Generic real-time is:

- An environment for simulating fixed-step models in single or multitasking mode on your workstation
- A means by which you can perform code validation
- A starting point for targeting "in-house" or custom hardware

Targeting

To use the Real-Time Workshop, you must decide on the environment where you want to place the generated code. This is known as *targeting*, the environment itself is called the t*arget*. The *host* is where you run MATLAB, Simulink, and the Real-Time Workshop. Using the build tools on the host, you create code and an executable that runs on your target system.

The system target file and the template makefile, which are files that you must specify on the Real-Time Workshop page of the **Simulation Parameters** dialog box, define what type of target you have. The table below lists examples of target environments and the system target files and template makefiles associated with each example environment:

Table 2-1: Target Systems and Associated Support Files

Target	System Target File	Template Makefile
Generic real-time	`grt.tlc`	`grt_default.tmf`
Embedded-C	`ert.tlc`	`ert_default.tmf`
DOS (4GW) real-time	`drt.tlc`	`drt_watc.tmf`
Tornado (VxWorks) real-time	`tornado.tlc`	`tornado.tmf`
Rapid Simulation	`rsim.tlc`	`rsim_default.tmf`

For a complete list of available targets, see “Targets Available from the System Target File Browser” on page 3-13. For a comparison of various targets, see Chapter 12, “Configuring Real-Time Workshop for Your Application.”

Target Language Compiler Files

Target Language Compiler files or *TLC files*, which are files that the Target Language Compiler compiles and executes, describe how to create code for your target. The Real-Time Workshop uses TLC files to translate your Simulink model into code.

The system target file is the entry point for the TLC program that creates the executable. The block target files define how the code looks for each of the Simulink blocks in your model.

The Build Process

The Real-Time Workshop build process is controlled by `make_rtw`, which is invoked when you click on the **Build** button of the Real-Time Workshop page of the **Simulation Parameters** dialog box. First, `make_rtw` compiles the block diagram and generates a *`model`*`.rtw` file. Next, `make_rtw` invokes the Target Language Compiler to generate the code. You must specify the system target file on the Real-Time Workshop page for the Target Language Compiler. Then `make_rtw` creates a makefile, *`model`*`.mk`, from the template makefile specified in the Real-Time Workshop page. Finally, if the host on which you are running matches the `HOST` macro specified in the template makefile, `make` is invoked to create a program from the generated code. See Chapter 3, "Code Generation and the Build Process," for more information on the build process.

Template Makefiles

The Real-Time Workshop uses template makefiles to build an executable from the generated code. By convention, a template makefile has an extension of `.tmf` and a name corresponding to your target. For example `grt_unix.tmf` is the generic real-time template makefile for UNIX.

A makefile is created from the template makefile by copying each line from the template makefile, expanding tokens into the makefile. See "Template Makefiles" on page 3-23 for more information about template makefiles and expandable tokens. The name of the makefile created is *`model`*`.mk`. The *`model`*`.mk` file is then passed to a `make` utility, which is a utility that builds an executable from a set of files. The `make` utility performs date checking on the dependencies between the object and C files and rebuilds the object files if needed.

Template Makefile Configuration

You can configure the build process by modifying the template makefile. You can do this by copying it to your local working directory and editing it. Alternatively, you can configure the template makefile's operation, by specifying build options to the `make_rtw` command.

For example, in the Real-Time Workshop page of the **Simulation Parameters** dialog box, you can turn debugging symbols on for the `grt_unix.tmf` by specifying the build command as:

```
make_rtw OPT_OPTS=–g
```

The comment section of the template makefiles distributed by The MathWorks includes some helpful hints about common build options.

Specifying Model Parameters

You modify the model parameters that control aspects of the simulation, such as start and stop time, by altering the fields of the **Simulation Parameters** dialog box. The simulation parameters are used directly for code generation and program building. Therefore, before you generate code and build a program, you must verify that the model parameters have been set correctly in the **Simulation Parameters** dialog box.

Building Generic Real-Time Programs

This example describes how to generate C code from a Simulink model and create a generic real-time program. This program:

- Executes as a stand-alone program, independent of external timing and events
- Saves data in a MATLAB MAT-file for later analysis
- Is built in the UNIX and PC environments

It also demonstrates how to use data logging to validate the generated code. Data logging allows you to compare system outputs from the Simulink block diagram to the data stored by the program created from the generated code.

The Simulink Model

This example uses the `f14` Simulink model from the `simdemos` directory (note that, by default, this directory is on your MATLAB path). `f14` is a model of a flight controller for the longitudinal motion of a Grumman Aerospace F-14 aircraft.

Naming Conventions This example is based on a model of the F-14 aircraft. When typed at the MATLAB prompt, `f14` opens the model. When passed to the operating system, `!f14` executes the file named `f14`. When used with the MATLAB load command, `load f14` reads from the file `f14.mat`. Finally, the phrase F-14 refers to the aircraft.

To display the model with Simulink, at the MATLAB command line, enter:

```
f14
```

This is the Simulink model that appears:

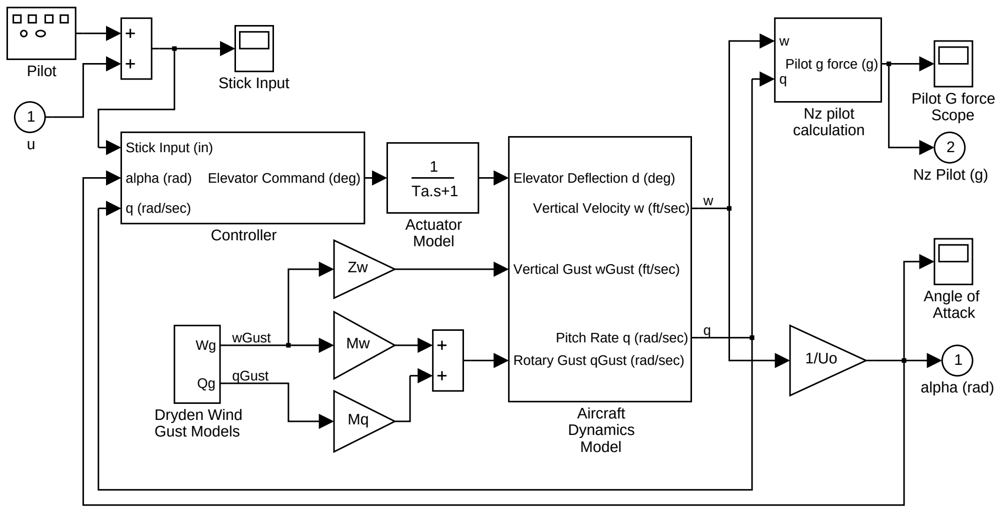


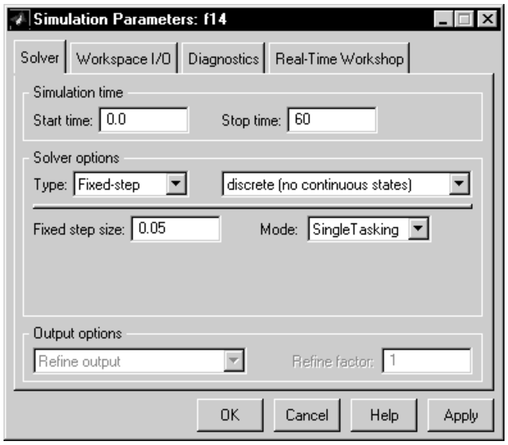
Figure 2-1: Simulink Model of an F14 Aircraft

The model employs a Signal Generator block to simulate the pilot's stick input, which is monitored by a Scope block. The system outputs are the aircraft angle of attack and the G forces experienced by the pilot. The outputs are also monitored by Scope blocks. These Scope blocks provide a means to monitor the operation of the Simulink model.

The design and internal workings of the `f14` model are not discussed in this example; such a discussion is beyond the scope of this manual. However, the procedure you must follow to use the Real-Time Workshop is generally independent of model content and complexity.

Setting Program Parameters

After displaying the `f14` model, select **Parameters** from the **Simulation** pull-down menu. This displays the **Simulation Parameters** dialog box with the Solver page selected by default:



Start time: 0.0 Stop time: 60

Type: Fixed-step discrete (no continuous states)

Fixed step size: 0.05 Mode: SingleTasking

Refine output Refine factor: 1

The **Simulation Parameters** dialog box allows you to set options, select a template makefile, generate code, and build the program.

When initially displayed, the **Simulation Parameters** dialog box contains default values that do not necessarily match the simulation parameters required to create a generic real-time executable. You should, therefore, change the default parameters to match the simulation parameters that are needed to use your model with the Real-Time Workshop. To configure the options for the `f14` model, change the following parameters:

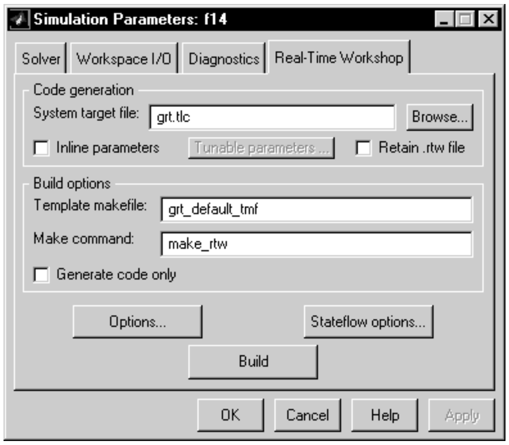
1. Under the Solver page, set the **Solver options Type** to `Fixed-step` and select the `ode5 (Dormand-Prince)` solver.

2. Set the **Fixed Step Size** to `0.05`

Note Alternatively, you can select **RTW Options** from the **Tools** pull-down menu. This brings you directly to the Real-Time Workshop page of the **Simulation Parameters** dialog box.

Building the Program

To build the program, select the Real-Time Workshop page in the **Simulation Parameters** dialog box:


Simulation Parameters: f14

Solver | Workspace I/O | Diagnostics | Real-Time Workshop

Code generation
System target file: grt.tlc
Inline parameters [ ] Retain .rtw file [ ]

Build options
Template makefile: grt_default_tmf
Make command: make_rtw
Generate code only [ ]

Click the **Build** button to generate C code from the `f14` model. The build command calls a system target file, `grt.tlc`, that uses the specified template makefile, `grt_default_tmf`, which in turn selects the template makefile for your system (e.g., `grt_vc.tmf`) to create the makefile. The Real-Time Workshop uses the created makefile to build the program.

Note Alternatively, you can select **RTW Build** from the **Tools** pull-down menu. This directs the Real-Time Workshop to generate code for the current model and is equivalent to clicking the **Build** button on the Real-Time Workshop page.

When you click on the **Build** button in the **Simulation Parameters** dialog box, the Real-Time Workshop invokes the `make` command, which in turn:

- Compiles the block diagram to produce the *`model`*`.rtw` file (in this example, `f14.rtw`)
- Invokes the Target Language Compiler, which in turn compiles the TLC program, starting with `grt.tlc`, and operates on *`model`*`.rtw` to produce the generated code
- Creates a makefile called *`model`*`.mk` (e.g., `f14.mk`) from the template makefile (e.g., `grt_vc.tmf`)
- If Simulink is running on the same host as that specified in the template makefile, then the program is built. Otherwise, processing stops after creating the model code and the makefile unless you define the `HOST` in the template makefile to match your target.

  The variable `HOST` is defined in the template makefile to identify what system you are targeting. The three options are `HOST=PC`, `UNIX`, or `ANY`. If you want to target a system other than the one you're running on, set `HOST=ANY` in the template makefile (for example, select `ANY` if you are targeting a DSP or microcontroller).

Once you have executed the **Build** command, the Real-Time Workshop creates these files by default:

- `f14.c` — the stand-alone C code
- `f14.h` — an include header file containing information about the state variables
- `f14_export.h` — an include header file containing information about exported signals and parameters
- `f14.reg` — an include file that contains the model registration function responsible for initializing data structures in the generated code
- `f14.prm` — an include file that holds information about the parameters used in the f14 model
- `f14` on UNIX and `f14.exe` on PC — the generic real-time executable

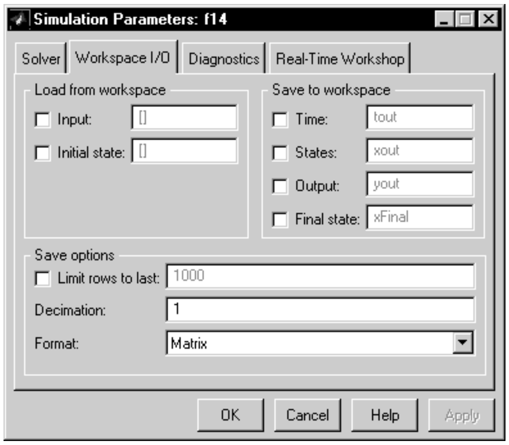
Customizing the Build Process

You can choose to inline parameters by clicking the **Inline parameters** check box on the Real-Time Workshop page. This directs the Real-Time Workshop to substitute the numerical values of the model parameters in place of the variable names. It also instructs Simulink to propagate constant sample times, which improves performance by placing constant operations into the start-up code. You can also customize the template makefile and the `make` commands; these options are discussed later in this chapter.

Clicking **Generate code only** tells the Real-Time Workshop to generate the C code without compiling, meaning that object and executable files are not created. Clicking **Retain .rtw file** directs the Real-Time Workshop to save the *`model`*`.rtw` file. By default, this file is deleted during the build process. For more information about customizing the build process, see “Automatic Program Building” on page 3-3.

Data Logging

This example uses the Real-Time Workshop’s MAT-file Data Logging facility to save system states and outputs, as well as the simulation time. To do this, select the Workspace I/O page of the Simulation Parameters dialog box. Checking the Time, States, and Outputs options directs the Real-Time Workshop to log the simulation time, any discrete or continuous states, and any root Outport blocks. This is a picture of the Workspace I/O page:



Load from workspace
☐ Input: []
☐ Initial state: []

Save to workspace
☐ Time: tout
☐ States: xout
☐ Output: yout
☐ Final state: xFinal

Save options
☐ Limit rows to last: 1000
Decimation: 1
Format: Matrix

For example, if you checked the Time and Outputs logging options, the Real-Time Workshop saves the inputs to all Scope blocks that are configured to save data to the workspace, the simulation time, and the root Output blocks in a MATLAB MAT-file. You can load this data into MATLAB for analysis. See "Blocks That Depend on Absolute Time" on page 2-21 for more information.

Inputs to models cannot be saved using the Workspace I/O page. To save inputs, you can use Scope block and click the **Properties** button. Alternatively, you can use a To Workspace block. See *Using Simulink* for more information about saving data to the MATLAB workspace using either of these methods.

Run-Time Interface for Real-Time

Building the real-time program requires a number of source files in addition to the generated code. These source files contain:

- A main program
- Code to drive execution of the model code
- Code to implement an integration algorithm
- Code to carry out data logging
- Code to create the `SimStruct` data structure, which is used to manage execution of your model

Configuring the Template Makefile

This example uses two different template makefiles: `grt_unix.tmf` for the UNIX environment and `grt_vc.tmf`, `grt_watc.tmf`, or `grt_bc.tmf` (depending on which compiler you have selected) for the PC environment. These makefile templates are automatically processed into makefiles properly configured to build the program. You can alter the build process by copying the template makefile to your local directory and modifying it.

These files are located in the `matlab/rtw/c/grt` directory. Use the MATLAB `matlabroot` command to determine the MATLAB path on your system.

The UNIX Template Makefile

The `grt_unix.tmf` template specifies `cc` or GNU's `gcc` as the default compiler. It also specifies the necessary source files, compiler flag, include paths, etc., that are required to build the program. You can, in general, use this makefile template without modification.

`grt_unix.tmf` is designed to be used by GNU Make, which is located in *`matlabroot`*`/bin/arch/make` (except for Linux systems, which provide GNU Make by default).

PC Template Makefiles

The template makefiles `grt_vc.tmf` and `grt_msvc.tmf` are designed to be used with the Microsoft Visual C/C++ compiler. To use either of these template makefiles, you must verify that the Visual C/C++ environmental variables are defined and that `nmake` is on your search path. To use `grt_vc.tmf` or `grt_msvc.tmf`, you must verify:

- That the `MsDevDir` environmental variable is defined
- That the Visual C/C++ compiler is correctly installed. See “Microsoft Visual C/C++” in the “Forward” section of this book for more information.

The template makefile `grt_watc.tmf` is designed to be used with the Watcom C/C++ Compiler and the Watcom make utility, `wmake`. To use `grt_watc.tmf`, you must verify:

- That the `WATCOM` environment variable is defined.
- That the Watcom compiler is correctly installed. See “Watcom” in “Forward” for more information.

The template makefile `grt_bc.tmf` is designed for use with the Borland C/C++ compiler. To use `grt_bc.tmf`, make sure:

- That the `BORLAND` environment variable is correctly defined.
- That the Borland compiler is correctly installed. See “Borland” in the “Foreword” for more information.

Generic Real-Time Modules

These source modules are automatically linked by the makefile, *`model`*`.mk`. The modules used to build this example are:

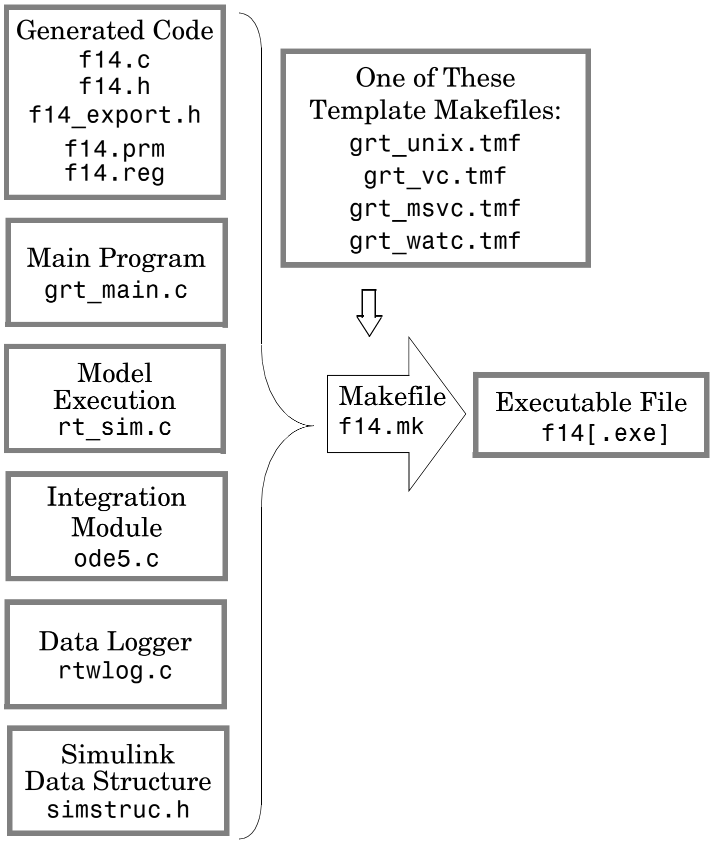


Figure 2-2: Source Modules Used to Build the Program

This diagram illustrates the code modules that are used to build a generic real-time program.

Blocks That Depend on Absolute Time

Some Simulink blocks use the value of absolute time (i.e., the time from the beginning of the program to the present time) to calculate their outputs. If you are designing a program that is intended to run essentially forever, then you cannot use blocks that have a dependency on absolute time.

The problem arises when the value of time reaches the largest value that can be represented by a double precision number. At that point, time is no longer incremented and the output of the block is no longer correct.

The tables below list all the Simulink blocks that depend on absolute time.

Continuous Blocks	
Derivative	Variable Transport Delay
Transport Delay	

Discrete Blocks	
Discrete-Time Integrator — when in triggered subsystems	

Nonlinear Blocks	
Rate Limiter	

Sinks	
Scope	To Workspace — only if logging in structure with time format
To File	

Sources	
Chirp Signal	Pulse Generator
Clock	Ramp
Digital Clock	Repeating Sequence
Discrete Pulse Generator	Signal Generator
From Workspace	Sine Wave
From File	Step

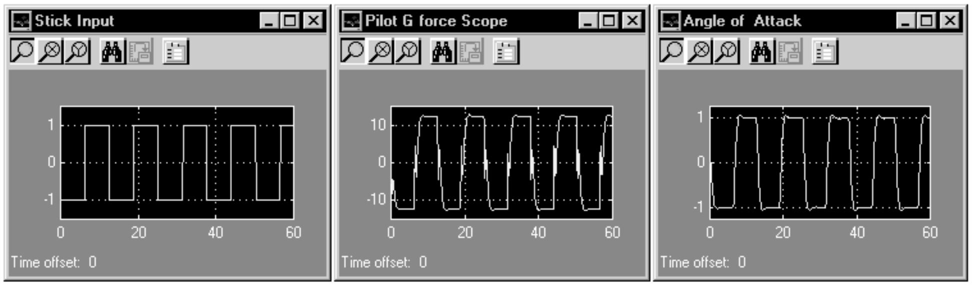
In addition, data logging `time` (in the Workspace I/O page of the **Simulation Parameters** dialog box) also requires absolute time.

Code Validation

After completing the build process, the stand-alone version of the `f14` model is ready for comparison with the Simulink model. The MAT-file data logging options selected with the Workspace I/O page of the **Simulation Parameters** dialog box cause the program to save the pilot G forces, aircraft angle of attack, and simulation time. You can save the Stick Input signal to the MATLAB workspace by setting options in the Scope block. You can now use MATLAB to produce plots of the same data that you see on the three Simulink scopes.

In both the Simulink and the generic real-time stand-alone executable version of the `f14` model, the stick input is simulated with a square wave having a frequency of 0.5 (rad/sec) and an amplitude of plus and minus 1, centered around zero.

Opening the Stick Input, Pilot G Force, and Angle of Attack scopes and running the Simulink simulation from `T = 0` to `T = 60` produces:



Now run the stand-alone program from MATLAB:

```
!f14
```

The “!” character passes the command that follows it to the operating system. This command, therefore, runs the stand-alone version of `f14` (not the M-file).

To obtain the data from the stand-alone program, load the file `f14.mat`:

```
clear
load f14
```

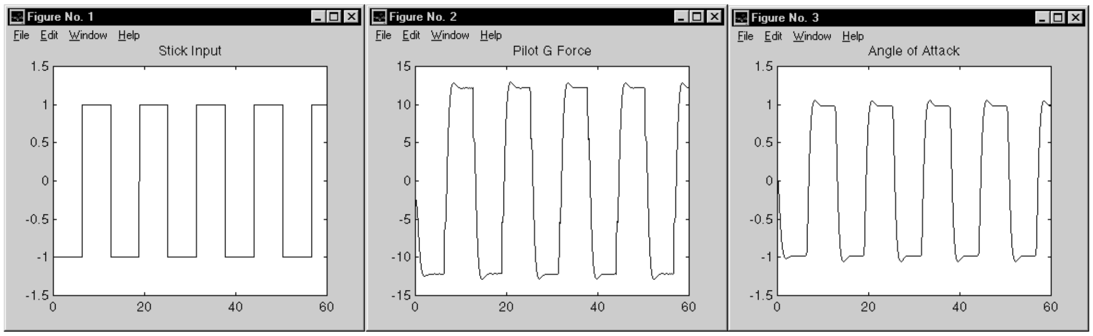
Then look at the workspace variables:

```
who
Your variables are:
rt_Pilot_G_force    rt_tout
rt_Angle_of_attack  rt_xout
rt_Stick_input      rt_yout
```

The variables `rt_tout`, `rt_xout`, and `rt_yout` were logged because the appropriate time, states, and outputs buttons were clicked on the Workspace I/O page. The variables `rt_Pilot_G_force`, `rt_Angle_of_attack`, and `rt_Stick_input` were saved using the Properties page of the Scope blocks in the `f14` model. If you are unfamiliar with how to save variables using Scope blocks, refer to the Simulink documentation. The variables are named according to the Simulink block that produced them, with spaces changed to underscores and an `rt_` prefix added to the variable names to identify them as real-time data.

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 Simulation and Generic Real-Time Results

Your Simulink simulations and generic real-time code should produce nearly identical output. Let's compare the `f14` model output from Simulink to the results achieved by the Real-Time Workshop. Follow these steps to do a correct comparison:

- First, make sure that you've selected the same (fixed-step) integration scheme for both the Simulink run and the Real-Time Workshop build process (for example, `ode5` (Dormand-Prince). Also, set the Fixed step size to the same number (for example, 0.05).
- Set the Scope blocks to log the input and outputs of the `f14` simulation (for example, you can set the Angle of Attack Scope block output to `Angle_of_attack`, the Pilot G Force Scope block output to `Pilot_G_Force`, and the Stick Input Scope block output to `Stick_input`).
- Run the `f14` simulation.
- Build the `f14` generic real-time system, run it, and load the MAT-file.

You should now have two sets of data, one generated by Simulink and one generated by the Real-Time Workshop. At the MATLAB prompt, type:

```
who
```

You should see among the variables in your workspace:

```
Angle_of_attack                rt_Stick_input
Pilot_G_force                  Stick_input
rt_Angle_of_attack
rt_Pilot_G_force
```

Comparing `Angle_of_attack` to `rt_Angle_of_attack` produces:

```
max(abs(rt_Angle_of_attack-Angle_of_attack))
ans =
  1.0e-015 *
         0    0.5551
```

Comparing `Pilot_G_force` to `rt_Pilot_G_force` produces:

```
max(abs(rt_Pilot_G_force-Pilot_G_force))
ans =
  1.0e-012 *
         0    0.1007
```

So overall agreement is within 10^{-12}. This slight error is caused by many factors, including:

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

For example, `sin(2.0)` may differ depending on which C library you are using.

Analyzing Data with MATLAB

For a more detailed analysis, you can add To Workspace blocks to the Simulink model. These blocks allow you to access any data generated within the block diagram and save it in MATLAB workspace variables. Alternatively, you can connect scope blocks configured to log data to the model.

Tracing Generated Code Back to Your Simulink Diagram

When you generate code, the Real-Time Workshop creates tags throughout the code that you can use to trace back to your Simulink model. In particular, the Real-Time Workshop generates a header file, called *`model`*`.h`, that documents how to use the tags with the MATLAB `open_system` command.

The basic format is `<system>/block_name`, where `system` is the system number (uniquely assigned by Simulink) and `block_name` is the name of the block.

Use the MATLAB `open_system` command to trace the generated code back to the model. For example,

```
open_system('<S3>')
```

opens `system 3` and

```
open_system('<S3>/Kp')
```

opens and selects Gain block `Kp` which resides in `S3`.

Targeting dSPACE

dSPACE, a company that specializes in real-time hardware and software products, markets a complete set of tools for use with the Real-Time Workshop. These tools are primarily intended for rapid prototyping of control systems and hardware-in-the-loop applications. This section provides a brief example of how to use dSPACE's RTI (real-time interface) with the Real-Time Workshop.

The dSPACE hardware line includes DSP boards based on the Texas Instruments TMS 320 C31 and C40 DSP's. dSPACE boards include:

- A single processor board, the DS1102, that includes on-board I/O (A/D and D/A conversion as well as digital I/O and encoder inputs).
- The DS1003 board based on the TI C40 DSP.
- The DS1004 based on the DEC Alpha, used for maximum throughput.

For a current summary and more information on dSPACE products and their relationship to the Real-Time Workshop, refer to the documentation provided by dSPACE or contact your dSPACE distributor.

At present, all dSPACE boards conform to the half length or full length ISA slot size. The ISA slot enables the host PC to download the executable to a dSPACE target processor board at a higher bandwidth than is possible over the ISA backplane. dSPACE boards utilize the proprietary PHS bus; this bus lets the dSPACE processor board(s) communicate with other dSPACE I/O boards. Multiprocessor and multi-I/O boards are common. One exception is that the DS1102 is for use as a single board solution with its own on-board I/O. The ISA bus is also used for downloading new block diagram parameters and for passing collected data back to the host computer.

The dSPACE product line includes a variety of A/D and D/A boards, encoder boards, and digital I/O boards. A hardware prototyping board gives you the ability to interface your own hardware directly with the dSPACE hardware. You can use all these products in combination with code generated by the Real-Time Workshop.

The following two figures show a PC setup that includes target hardware in the PC chassis and the dSPACE PHS bus:

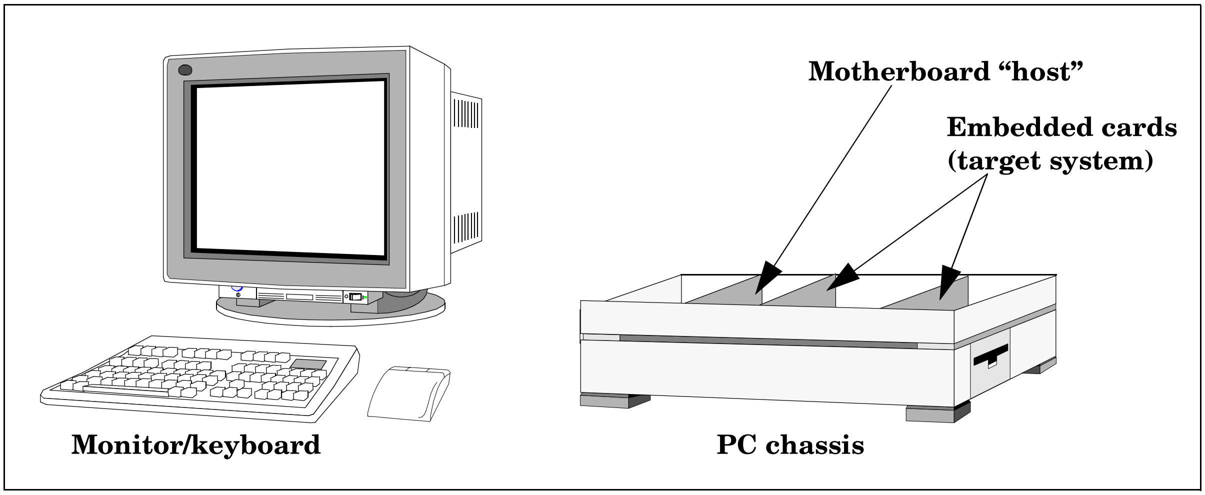


Figure 2-3: A PC Setup Including a Target System

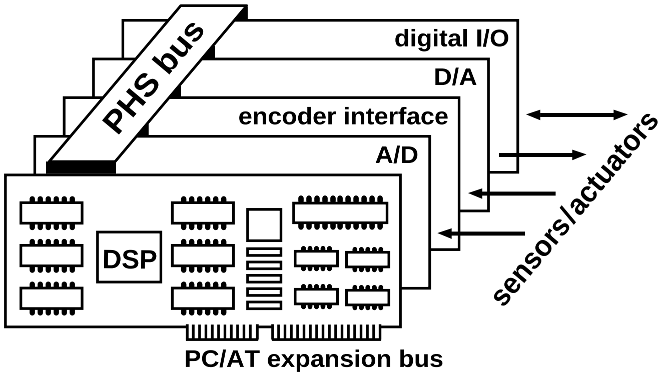


Figure 2-4: The dSPACE PHS Bus

Real-Time Interface (RTI)

When using the Real-Time Workshop with dSPACE hardware, you must generate target-specific software. The key software component for this configuration is the dSPACE Real-Time Interface to Simulink. The RTI, with special versions available for various dSPACE processor boards (C31, C40, DEC Alpha, etc.), is available directly from dSPACE and dSPACE distributors.

Executing code generated from the Real-Time Workshop on a particular target in real-time requires target-specific code. Target-specific code includes I/O device drivers and an interrupt service routine. Other components, such as a communication link with Simulink, are required if you need the ability to download parameters on-the-fly to your target hardware. The dSPACE RTI provides this support. Since these components are specific to particular hardware targets (in this case dSPACE hardware), you must ensure that these target-specific components are compatible with the target hardware. To allow you to build an executable, dSPACE also provides a target makefile specific to a particular dSPACE processor board. This target makefile invokes the crosscompiler, which is also available from dSPACE.

When used in combination with the Real-Time Workshop, dSPACE products provide an integrated environment that, once installed, needs no additional coding.

Building a Real-Time Executable with dSPACE's RTI

dSPACE provides a set of preconfigured files for use with their hardware and RTI. These files simplify the process of building a real-time executable and downloading it to the dSPACE target microprocessor. For operation with dSPACE, refer to the dSPACE documentation.

Assuming you have already installed dSPACE products on your PC, from the MATLAB command line, type `dslib`. This opens a Simulink block diagram that includes a set of preconfigured blocks for dSPACE
I/O device drivers. These devices are associated with particular dSPACE hardware. Add the I/O device drivers as appropriate for your model and for your dSPACE hardware.

Select the pull-down menu for the **Simulink Parameters** dialog box. From this dialog box, select the Real-Time Workshop page. In the following example, it is assumed that you are using the dSPACE DS1102 C31 board. When using other dSPACE processor boards, you must specify the appropriate versions of the system target file and template makefile. For the current example, in the Real-Time Workshop page of the dialog box, specify:

- System target file: `rti1102.tlc`
- Template makefile: `rti1102.tmf`

With this configuration, you can now generate a real-time executable and download it to the dSPACE processor board. You can do this by clicking the **Build** button on the Real-Time Workshop page. The Real-Time Workshop automatically generates C code and inserts the I/O device drivers as specified in your block diagram. These device drivers are inserted in the generated C code as inline S-functions. Inlined S-functions offer advantages in speed and simplify the generated code. For more information about inlining S-functions, refer to the *Target Language Compiler Reference Guide*. For a complete discussion of S-functions, see *Writing S-Functions*.

During the same build operation, the template makefile and dialog entries are combined to form a target makefile for your dSPACE setup. This makefile invokes the TI cross-compiler and builds an executable that is automatically downloaded via the ISA bus onto the dSPACE processor board. With additional support tools from dSPACE, such as Control Desk and COCKPIT, you can view signals, collect data, and log data in a MATLAB MAT-file format.

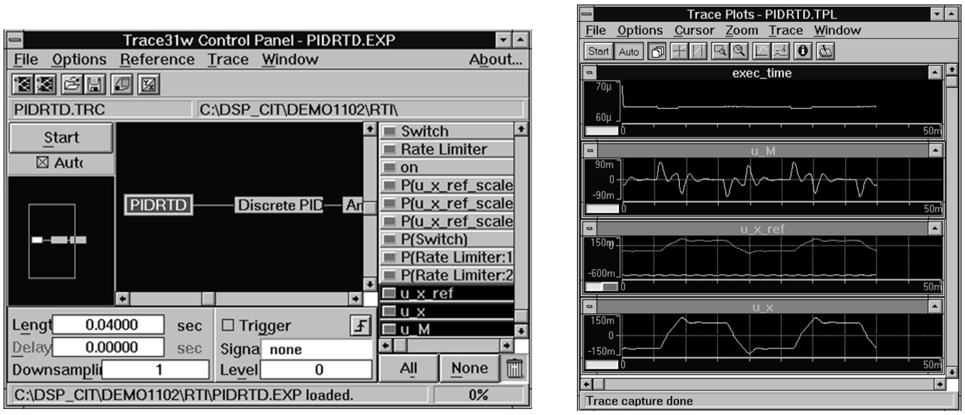
You can also change model parameters while the model runs on the target processor. There are two ways to do this:

- Using Simulink's external mode
- Using dSPACE'S COCKPIT software

dSPACE Control Desk

Control Desk enables you to capture and view signals using names that appear in the Simulink block diagram. Multiple signals can be acquired and displayed from a convenient GUI. Control Desk also includes a signal browser that helps users scan the model hierarchy and that selects signals to monitor from a particular subsystem. Options such as data decimation and data acquisition triggering are available from Control Desk. Once data is collected, you can save it in a MATLAB MAT-file format for easy import into MATLAB.

This picture shows the Control Desk Control Panel and a sample of Control Desk plots:



PIDRTD.TRC C:\DSP_CIT\DEMO1102\RTI\

Lengt 0.04000 sec
Delay 0.00000 sec
Downsampli 1
Signa none
Level 0

C:\DSP_CIT\DEMO1102\RTI\PIDRTD.EXP loaded. 0%

Figure 2-5: The Control Desk Control Panel and Sample Control Desk Plots.

dSPACE COCKPIT

COCKPIT provides a virtual instrument panel on the host PC. It provides an interface for editing parameters while the generated code executes on the dSPACE processor boards. The communications link between COCKPIT and the processor boards lets you display signals with dials and alert displays or change parameters using pushbuttons, sliders, and other input blocks on the COCKPIT instrument panel.

Code Generation and the Build Process

Introduction

The Real-Time Workshop simplifies the process of building application programs. One of the Real-Time Workshop's features is *automatic program building*, which provides a standard means to create programs for real-time applications in a variety of host environments. It does this in a uniform and controlled manner, yet is customizable to different applications.

Automatic program building uses the `make` utility to control how the program is built and an M-file to create a customized makefile (the description file referenced by the `make` utility) from a customizable template.

Chapter 2 introduced the build process and template makefiles. This chapter discusses these concepts in more detail. Topics included are:

- Automatic program building
- The Real-Time Workshop user interface
- Configuring generated code
- Customizing template makefiles
- Generic real-time template makefiles

Automatic Program Building

The Real-Time Workshop automates the task of building a stand-alone program from your Simulink model. When you click on the **Build** button on the Real-Time Workshop page of the **Simulation Parameters** dialog box, the `make` command, which starts with `make_rtw`, is invoked. The build process consists of three main steps that are controlled by an M-file, `make_rtw.m`:

1. Generating the model code.
2. Generating a makefile that is customized for a given build.
3. Invoking the `make` utility with the customized makefile.

The shaded box in the figure below outlines these steps:

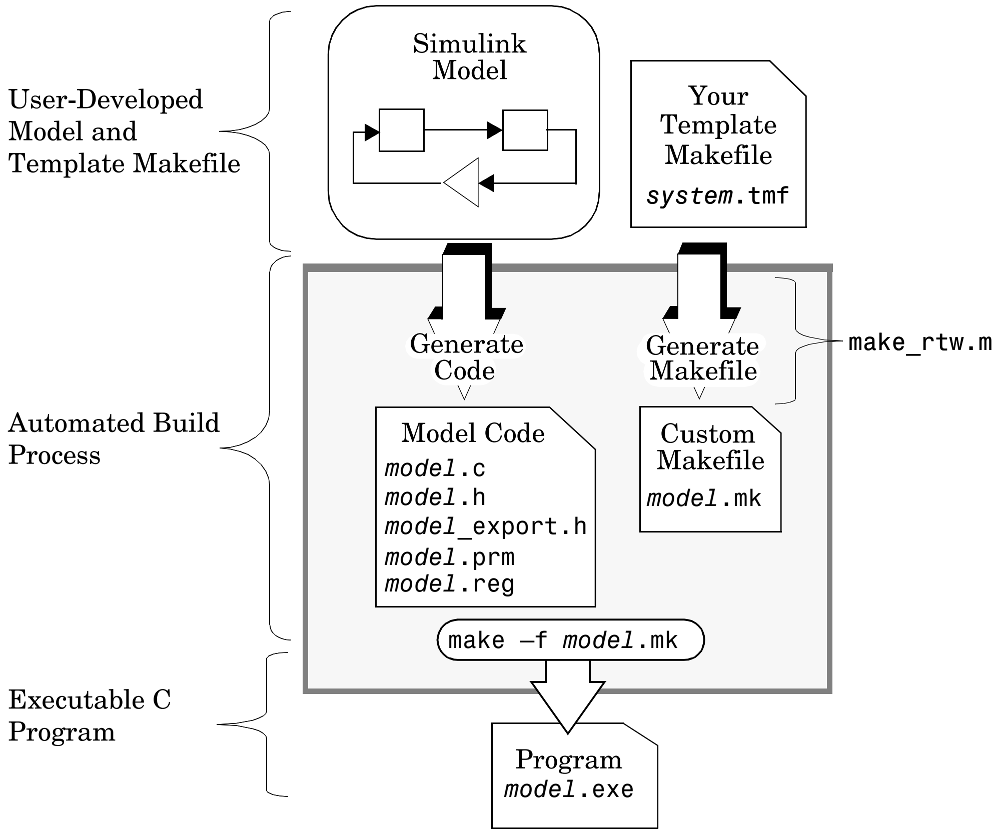


Figure 3-1: The Build Process

The task of creating a stand-alone program from the generated code is simplified by automated program building.

When you click the **Build** button on the Real-Time Workshop page on the **Simulation Parameters** dialog box, these steps are carried out automatically. This diagram illustrates the logic that controls this process:

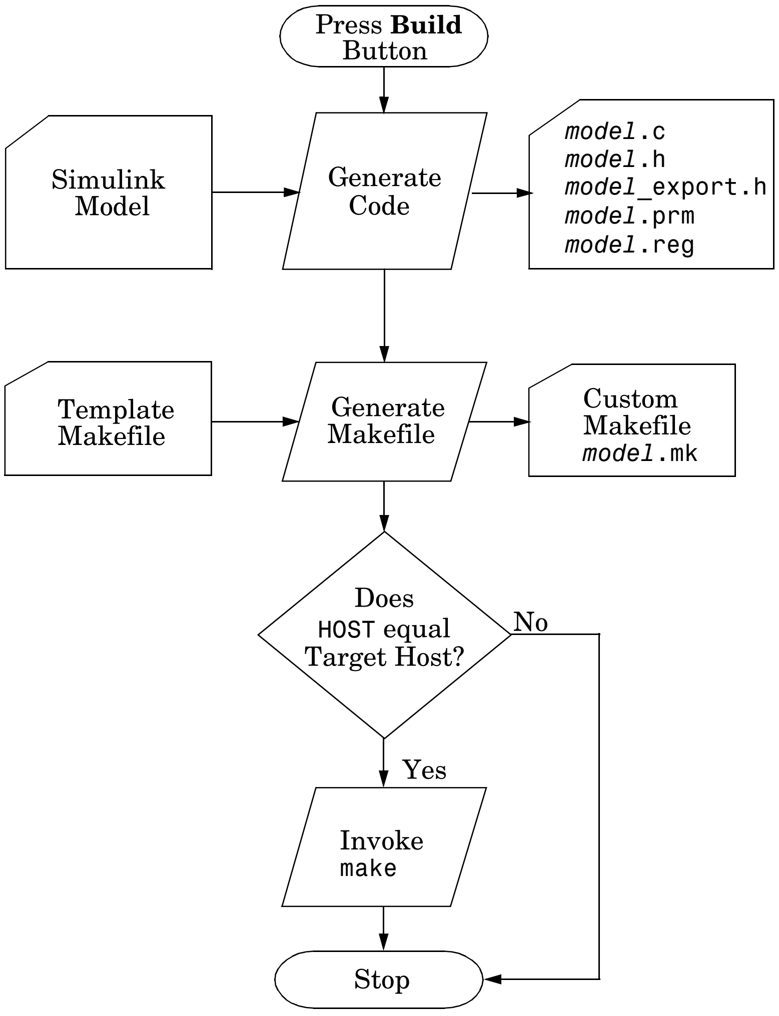


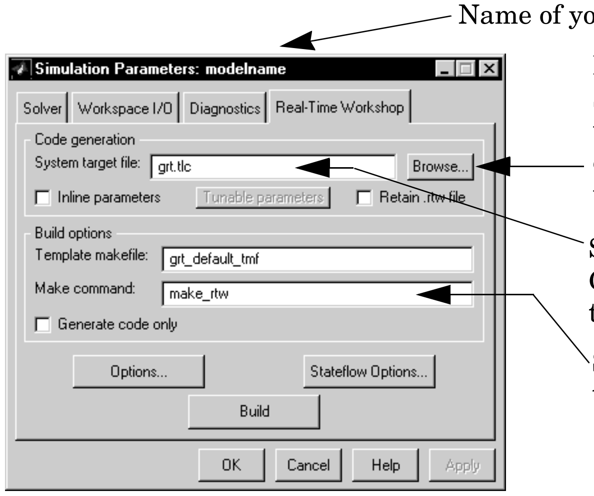
Figure 3-2: The Logic That Controls Automatic Program Building

The Real-Time Workshop User Interface

You work with the Real-Time Workshop by interacting with and modifying fields of the **Simulation Parameters** dialog box of your Simulink model. One page of this dialog box is exclusively for the Real-Time Workshop. The other pages apply to both Simulink simulations and the Real-Time Workshop. To access the **Simulation Parameters** dialog box, you can select the **Parameters** item of the **Simulation** menu. Alternatively, you can select the **RTW Options** from the **Tools** menu, which opens the **Simulation Parameters** dialog box to the Real-Time Workshop page.

All pages of the **Simulation Parameters** dialog box affect Real-Time Workshop code generation. On the Solver page, you specify the type of solver, and its options including start and stop time. When using the Real-Time Workshop you must specify a fixed-step solver. To specify data logging options, you use the Workspace I/O page (providing you target has support for these options). On the Diagnostics page you specify options that affect whether or not various model conditions such as unconnected ports are ignored, treated as a warning, or cause an error condition when the build process is invoked.

This chapter discusses in detail the Real-Time Workshop page of the **Simulation Parameters** dialog box, including the fields on the page and their optional arguments.



System target file: grt.tlc

☐ Inline parameters ☐ Retain .rtw file

Template makefile: grt_default_tmf

Make command: make_rtw

☐ Generate code only

Name of your model

Press the **Browse** button to specify the system target file, template makefile, and `make` command for your desired target.

Specify Target Language Compiler options after the system target filename.

Specify `make` options after the `make` command name.

Figure 3-3: Description of the Real-Time Workshop Page

System Target File

Use the **System target file** field to specify the type of code and target for which you are generating code. For a complete table of available targets, see “Targets Available from the System Target File Browser” on page 3-13. After you have selected the system target file, you can specify options for the Target Language Compiler (TLC). The common options are shown in the table below.

Table 3-1: Target Language Compiler Options

Option	Description
`–I`*`path`*	Adds *`path`* to the list of paths in which to search for target files (`.tlc` files).
`–m[`*`N`*`|a]`	Maximum number of errors to report when an error is encountered (default is 5). For example, `–m3` specifies that at most three errors will be reported. To report all errors, specify `–ma`.
`–d[g|n|o]`	Specifies debug mode (generate, normal, or off). Default is off. When `–dg` is specified, a `.log` file is create for each of your TLC files. When debug mode is enabled (i.e., generate or normal), the Target Language Compiler displays the number of times each line in a target file is encountered.
`–a`*`Variable=expr`*	Assigns a variable to a specified value (i.e., creates a parameter value pair) for use by the target files during compilation.

After you have selected the system target file, you can specify code generation options for the Target Language Compiler (TLC). The most common options are available through the **Code Generation Options** dialog box; see “Options Button” on page 3-15 for a description of this feature. You can configure additional options, however, by appending `-aVariableName=value` in the **System target file** field or including the line

```
%assign VariableName = value
```

in the system target file itself.

This table lists the options available.

Table 3-2: Target Language Compiler Optional Variables

Variable	Description
`–aMaxStackSize=`*`N`*	When local block outputs is enabled, this limits the number of local variables that are declared in a function to not exceed the `MaxStackSize` (in bytes). `N` can be any positive integer.
`–aMaxStackVariableSize=`*`N`*	When local block outputs is enabled, this limits the size that any given local variable that is declared in a function can be (in bytes). `N` can be any positive integer.
`–aFunctionInlineMode=` `"`*`mode`*`"`	Controls function inlining. There are four modes: • `Automatic` • `Manual` • `Either` • `None` `Automatic` uses `AutoInlineThreshold` to determine whether or not to inline. `Manual` uses `RTWData` attached to the Subsystem to determine whether or not to inline. See the *Target Language Compiler Reference Guide* for information about `RTWData`.

Table 3-2: Target Language Compiler Optional Variables (Continued)

Variable	Description
`–aFunctionInlineType="`*`mode`*`"`	Controls how functions are inlined. There are two modes: • `CodeInsertion` • `PragmaInline` Using `CodeInsertion`, the code is actually inserted where the function call would have be made. `PragmaInline` directs the Target Language Compiler to declare the function when the appropriate compiler directive occurs.
`–PragmaInlineString="`*`string`*`"`	If `FunctionInlineType` is set to `PragmaInline`, this should be set to the directive that your compiler uses for inlining a function (for example, for Microsoft VC++, `"__inline"`)
`–aAutoInlineThreshold=`*`N`*	When `FunctionInlineMode` is `Automatic` or `Either`, this variable sets what the threshold for the number of lines should be. For function-call subsystem functions, the number of line is multiplied by the number of callers. If the count is less then this threshold, the function is inlined. `N` is any positive integer.

Table 3-2: Target Language Compiler Optional Variables (Continued)

Variable	Description
`–aWarnNonSaturatedBlocks = `*`value`*	Flag to control display of overflow warnings for blocks that have saturation capability, but have it turned off (unchecked) in their dialog. These are the options: • `0` — no warning is displayed • `1` — displays one warning for the model during code generation • `2` — displays one warning which contains a list of all offending blocks
`–aBlockIOSignals=`*`value`*	Block IO signals C API for monitoring signals in a running model. Setting the variable causes the *`model`*`.bio` file to be generated. These are the options: • `0` — deactivates this feature • `1` — creates *`model`*`.bio`
`–aParameterTuning=`*`value`*	Parameter tuning C API for changing parameter values in a running model Setting the variable causes the *`model`*`.pt` file to be generated. These are the options: • `0` — deactivates this feature • `1` — creates *`model`*`.pt`

Inline Parameters

Inlining parameters refers to a mode where blocks with a constant sample time are removed from the run-time model execution. The output signals of these blocks are set up once during model start up. When you select this option, Simulink automatically sets all the inlined parameters to “invariant constants.” See the *Using Simulink* for information on invariant constants.

Retaining the model.rtw File

When modifying the target files, you will need to look at the `*model*.rtw` file. To prevent the `*model*.rtw` file from being deleted after the build process is completed, select the **Retain .rtw file** check box.

Template Makefile

The template makefile is used when the **Generate code only** check box is not selected. The template makefile uniquely identifies the target for which you are creating an executable.

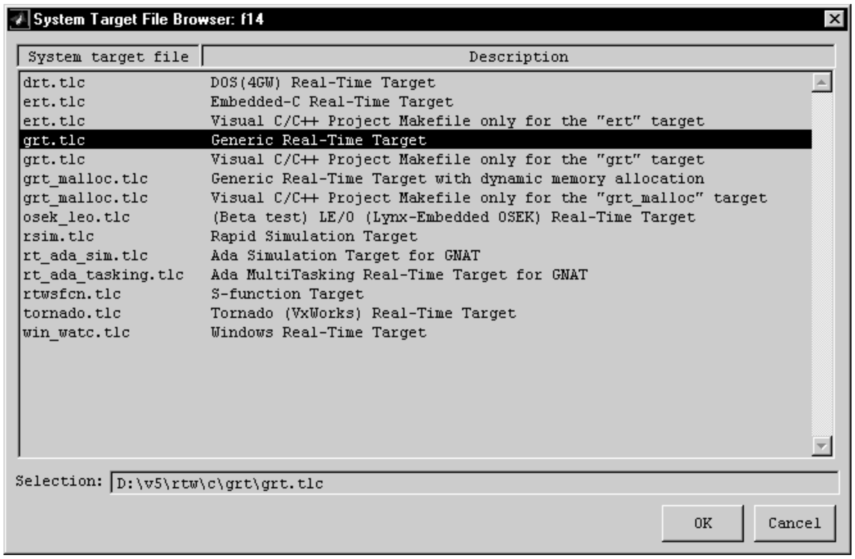
Make Command

The `make` command starts with `make_rtw` (unless this has been replaced with a third-party `make` command, in which case you should see their documentation). This command is invoked when you click the **Build** button. You can supply `make` arguments as additional arguments to `make_rtw`. For example, if you are using one of the `grt*.tmf` files, you can alter the optimization options by specifying:

```
make_rtw OPT_OPTS="compiler_specific_setting"
```

The System Target File Browser

The Real-Time Workshop supports numerous targets and code formats. For your convenience, the Real-Time Workshop includes a browser that allows you select the target appropriate to your application. This is a picture of the browser with the generic real-time target selected.



System target file	Description
drt.tlc	DOS(4GW) Real-Time Target
ert.tlc	Embedded-C Real-Time Target
ert.tlc	Visual C/C++ Project Makefile only for the "ert" target
grt.tlc	Generic Real-Time Target
grt.tlc	Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc	Generic Real-Time Target with dynamic memory allocation
grt_malloc.tlc	Visual C/C++ Project Makefile only for the "grt_malloc" target
osek_leo.tlc	(Beta test) LE/O (Lynx-Embedded OSEK) Real-Time Target
rsim.tlc	Rapid Simulation Target
rt_ada_sim.tlc	Ada Simulation Target for GNAT
rt_ada_tasking.tlc	Ada MultiTasking Real-Time Target for GNAT
rtwsfcn.tlc	S-function Target
tornado.tlc	Tornado (VxWorks) Real-Time Target
win_watc.tlc	Windows Real-Time Target

Selection: D:\v5\rtw\c\grt\grt.tlc

Selecting a system target file and clicking **OK** tells the Real-Time Workshop to use the selected target. The Real-Time Workshop automatically chooses the system target file, template makefile, and `make` command for the selected target.

The Real-Time Workshop supports many hardware targets, including DOS, UNIX, Tornado, and DSPACE; most hardware targets have chapters devoted to them in this book. Targets from third-party vendors come with their own documentation.

The Real-Time Workshop also supports various code formats; see Chapter 12, "Configuring Real-Time Workshop for Your Application," for a comparison of various code formats.

This table lists all the supported system target files, the code format and target associated with it, and the relevant chapter in this book.

Table 3-3: Targets Available from the System Target File Browser

Target/Code Format	System Target File	Template Makefile	Make Command	Relevant Chapters
DOS (4GW)	`drt.tlc`	`drt_watc.tmf`	`make_rtw`	9 and 12
Embedded-C for PC or UNIX	`ert.tlc`	`ert_default_tmf`	`make_rtw`	12
Embedded-C for Watcom	`ert.tlc`	`ert_watc.tmf`	`make_rtw`	12
Embedded-C for Visual C/C++	`ert.tlc`	`ert_vc.tmf`	`make_rtw`	12
Embedded-C for Visual C/C++ Project Makefile	`ert.tlc`	`ert_msvc.tmf`	`make_rtw`	12
Embedded-C for Borland	`ert.tlc`	`ert_bc.tmf`	`make_rtw`	12
Embedded-C for UNIX	`ert.tlc`	`ert_unix.tmf`	`make_rtw`	12
Generic Real-Time for PC/UNIX	`grt.tlc`	`grt_default_tmf`	`make_rtw`	3 and 12
Generic Real-Time for Watcom	`grt.tlc`	`grt_watc`	`make_rtw`	3 and 12
Generic Real-Time for Visual C/C++	`grt.tlc`	`grt_vc.tmf`	`make_rtw`	3 and 12
Generic Real-Time for Visual C/C++ Project Makefile	`grt.tlc`	`grt_msvc.tmf`	`make_rtw`	3 and 12
Generic Real-Time for Borland	`grt.tlc`	`grt_bc.tmf`	`make_rtw`	12

Table 3-3: Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile	Make Command	Relevant Chapters
Generic Real-Time for UNIX	`grt.tlc`	`grt_unix.tmf`	`make_rtw`	3 and 12
Generic Real-Time (dynamic) for PC/ UNIX	`grt_malloc.tlc`	`grt_malloc_default_ tmf`	`make_rtw`	3 and 12
Generic Real-Time (dynamic) for Watcom	`grt_malloc.tlc`	`grt_malloc_watc.tmf`	`make_rtw`	3 and 12
Generic Real-Time (dynamic) for Visual C/C++	`grt_malloc.tlc`	`grt_malloc_vc.tmf`	`make_rtw`	3 and 12
Generic Real-Time (dynamic) for Visual C/C++ Project Makefile	`grt_malloc.tlc`	`grt_malloc_msvc.tmf`	`make_rtw`	3 and 12
Generic Real-Time (dynamic) for Borland	`grt_malloc.tlc`	`grt_malloc_bc.tmf`	`make_rtw`	12
Generic Real-Time (dynamic) for UNIX	`grt_malloc.tlc`	`grt_malloc_unix.tmf`	`make_rtw`	3 and 12
LE/O (Lynx embedded OSEK) Real-Time Target	`osek_leo.tlc`	`osek_leo.tmf`	`make_rtw MAT_FILE=1 RUN=1 LEO_NODE=osek`	Readme file in *`matlabroot`*`/rtw/c/osek_leo`
Rapid Simulation Target (default for PC or UNIX)	`rsim.tlc`	`rsim_default_tmf`	`make_rtw`	12

Table 3-3: Targets Available from the System Target File Browser (Continued)

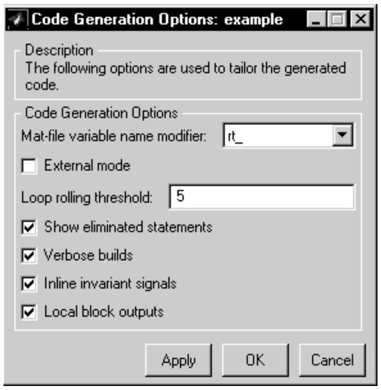
Target/Code Format	System Target File	Template Makefile	Make Command	Relevant Chapters
Rapid Simulation Target for Watcom	`rsim.tlc`	`rsim_watc.tmf`	`make_rtw`	12
Rapid Simulation Target for Visual C/C++	`rsim.tlc`	`rsim_vc.tmf`	`make_rtw`	12
Rapid Simulation Target for Borland	`rsim.tlc`	`rsim_bc.tmf`	`make_rtw`	12
Rapid Simulation Target for UNIX	`rsim.tlc`	`rsim_unix.tmf`	`make_rtw`	12
Ada Simulation Target for GNAT	`rt_ada_sim.tlc`	`gnat_sim.tmf`	`make_rtw -ada`	14
Ada Real-Time Multitasking Target for GNAT	`rt_ada_tasking .tlc`	`gnat_tasking.tmf`	`make_rtw -ada`	14
S-Function Target for PC or UNIX	`rtwsfcn.tlc`	`rtwsfcn_default_tmf`	`make_rtw`	12
S-Function Target for Watcom	`rtwsfcn.tlc`	`rtwsfcn_watc.tmf`	`make_rtw`	12
S-Function Target for Visual C/C++	`rtwsfcn.tlc`	`rtwsfcn_vc.tmf`	`make_rtw`	12
Tornado (VxWorks) Real-Time Target	`tornado.tlc`	`tornado.tmf`	`make_rtw`	8
Windows 95/98/NT Real-Time Target	`win_watc.tlc`	`win_watc.tmf`	`make_rtw`	12

Options Button

There is an **Options** button on the Real-Time Workshop page of the **Simulation Parameters** dialog box. Pressing this button opens a **Code**

Generation Options dialog box that varies depending on which system target file configuration you selected in the **System Target File Browser**.

This picture shows an example for `grt.tlc`.



Description
The following options are used to tailor the generated code.

Code Generation Options
Mat-file variable name modifier: rt_
[ ] External mode
Loop rolling threshold: 5
[✓] Show eliminated statements
[✓] Verbose builds
[✓] Inline invariant signals
[✓] Local block outputs

Figure 3-4: Code Generation Options Dialog Box for the Generic Real-Time Target

The features supported by this dialog box depend on the target, but typically these functions are available:

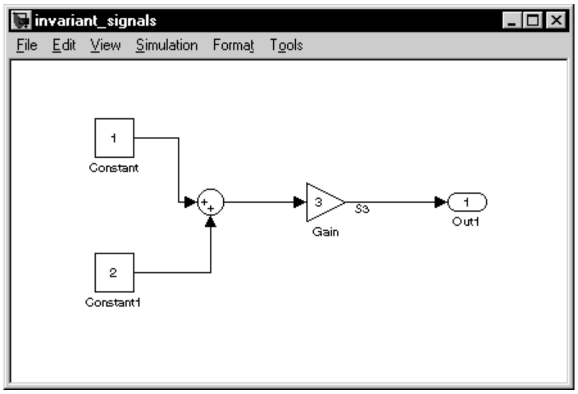
- Loop rolling threshold
- Show eliminated statements
- Verbose builds
- Inline invariant signals
- Local block outputs

Loop rolling threshold. You can set a threshold (a positive integer) for when loop rolling occurs. For example, setting this threshold to five means that if your code has an algorithm that requires more than five increments, then the entire algorithm is rolled into a `for` loop.

Show eliminated statements. Eliminated statements that are the result of optimizations (such as parameter inlining) show up in the generated code as comments.

Verbose builds. This feature forces command line display of code generation stages and compiler output.

Invariant signals. Note that "invariant constants" are not the same things as "invariant signals." Invariant signals are block output signals that do not change during Simulink simulation. For example, the signal `S3` in this block diagram is an invariant signal.

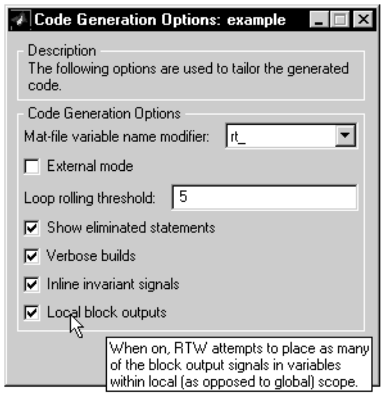


If you select **Inline invariant signals** in the **Code Generation Options** dialog box, the Real-time Workshop inlines the invariant signal `S3` in the generated code. This is different from the **Inline parameters** option on the Real-Time Workshop page, which applies to invariant constants. In this example, the Real-Time Workshop inlines the two constants and the gain if you select **Inline parameters**.

Note that to inline invariant signals, you must have previously chosen to inline parameters. If the **Inline parameters** check box is not selected, the **Inline invariant** signals option will not work. It is possible, however, to inline parameters but leave invariant signals non-inlined.

Local block outputs. When this check box is active, the Real-Time Workshop attempts to place as many of the block output signals in variables within local (as opposed to global) scope.

Online Help. Placing your mouse over any of these field names or check boxes activates a Help facility that explains what the feature is. This picture shows the Help facility in use.


Code Generation Options: example

Description
The following options are used to tailor the generated code.

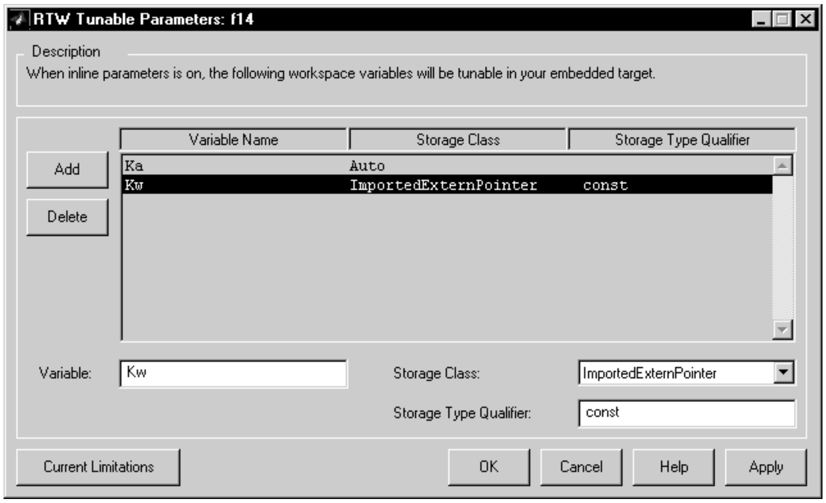
Code Generation Options
Mat-file variable name modifier: rt_
☐ External mode
Loop rolling threshold: 5
☑ Show eliminated statements
☑ Verbose builds
☑ Inline invariant signals
☑ Local block outputs

When on, RTW attempts to place as many of the block output signals in variables within local (as opposed to global) scope.

Figure 3-5: Using the Online Help on the Code Generation Options Dialog Box

Tunable Parameters

If you select **Inline parameters** on the Real-Time Workshop page, the **Tunable parameters** button activates. Clicking this button opens the **RTW Tunable Parameters** dialog box. This picture shows this dialog box for the `f14` model.


RTW Tunable Parameters: f14

Description
When inline parameters is on, the following workspace variables will be tunable in your embedded target.

Variable Name	Storage Class	Storage Type Qualifier
Ka	Auto	
Kw	ImportedExternPointer	const

Variable: Kw
Storage Class: ImportedExternPointer
Storage Type Qualifier: const

The **RTW Tunable Parameters** dialog box supports the following features:

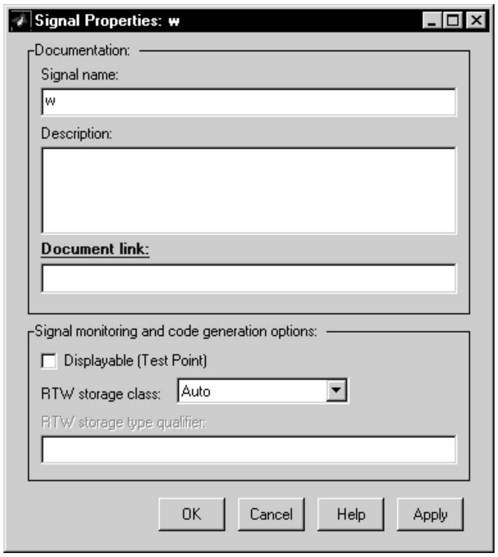
- Parameter tuning — de-inline any model parameter by placing its name in the **Variable** field and clicking **Add**. This effectively negates the **Inline parameters** check box for the variables you select. All other model parameters remain inlined.
- **Storage Class** — you can change the storage class of the selected variable. The options in the pull-down menu are as follows:
  - Auto — directs the Real-Time Workshop to store the variable in a persistent data structure. This is the default storage class option for the Real-Time Workshop.
  - Exported Global — declares the variable as a global variable that can be accessed from outside the generated code.
  - Imported Extern — declares the variable as `extern`. It must then be declared from outside the generated code.
  - Imported Extern Pointer — declares the variable as an `extern` pointer. It must then be declared from outside the generated code.

  These cases are useful if you want to link Real-Time Workshop generated code to other C code (that is, code that the Real-Time Workshop did not generate). Note that you are responsible for correctly linking code modules together.
- **Storage Type Qualifier** — prepend any string (for example, `const`) to the variable type declaration by typing it in the **Storage Type Qualifier** field. Note that the Real-Time Workshop does not check this string for errors; whatever you enter is directly prepended to the variable declaration.

C API for Parameter Tuning. The Real-Time Workshop has support for developing a C application program interface (API) for tuning parameters independent of external mode. See the readme located in *`matlabroot`*`/rtw/src/pt_readme.txt` for more information.

Signal Properties

The Real-Time Workshop supports the same storage class options for signals as it does for parameters. To change the storage class of a signal, select it in your Simulink model; then select **Signal Properties** under the **Edit** menu of your model. This opens the **Signal Properties dialog** box.



Signal name: w

Displayable (Test Point): ☐

RTW storage class: Auto

RTW storage type qualifier:

Refer to *Using Simulink* for information about these options.

The options relevant to the Real-Time Workshop are located in the Signal monitoring and code generation options panel in the bottom half of the dialog box. The supported features are as follows:

- **Displayable (Test Point)** — clicking this check box directs the Real-Time Workshop to place the signal in a unique global memory location. This is useful for testing purposes since it eliminates the possibility of overwriting the signal data. Note that selecting this option forces the **RTW storage class** to be auto.
- **RTW storage class —** you can change the storage class of the selected signal. The options in the pull-down menu are as follows:
  - Auto — directs the Real-Time Workshop to store the signal in whichever storage class it deems as appropriate. This is the default storage class option for the Real-Time Workshop.
  - Exported Global — declares the signal as a global variable that can be accessed from outside the generated code.

- Imported Extern — declares the signal as `extern`. It must then be declared from outside the generated code.
- Imported Extern Pointer — declares the signal as an `extern` pointer. It must then be declared from outside the generated code.

These cases are useful if you want to link Real-Time Workshop generated code to other C code (that is, code that the Real-Time Workshop did not generate). Note that you are responsible for correctly linking code modules together.

C API for Signal Monitoring. The Real-Time Workshop provides support for developing a C API for signal monitoring independent of external mode. See "C API for Signal Monitoring" on page 5-4 for more information.

Configuring the Generated Code

As an alternative to using the **Code Generation Options** dialog box, you can configure the generated code by appending Target Language Compiler options to the system target filename in the System Target file field on the Real-Time Workshop page of the **Simulation Parameters** dialog box (see "System Target File" on page 3-7). Your target configuration may have additional options. Options that can be altered are generally described in your system target file. If you are using a system target file provided by a third-party vendor, please consult their documentation.

In addition to Target Language Compiler options, you can customize all aspects of the generated code by modifying the target `(.tlc)` files. These files are located in
`matlabroot/rtw/c/tlc`. Modification of the target files is described in detail in the *Target Language Compiler Reference Guide*.

Template Makefiles

This section contains a description of how to work with and modify the template makefiles that are used with the Real-Time Workshop. Template makefiles are essentially makefiles in which certain tokens are expanded to create a makefile for your model (`model.mk`). `model.mk` is created from the template makefile specified in the Real-Time Workshop page of the **Simulation Parameters** dialog box by copying each line from the template makefile and expanding the tokens of Table 3-4.

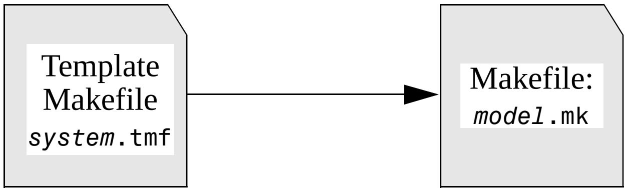


Figure 3-6: Creation of model.mk

`model.mk` is created from your target systems template makefile by copying line for line the contents of it and expanding the Real-Time Workshop tokens. This table lists the Real-Time Workshop tokens and their expansions.

Table 3-4: Template Makefile Tokens Expanded by make_rtw

Token	Expansion
`|>COMPUTER<|`	Computer type. See the MATLAB `computer` command.
`|>MAKEFILE_NAME<|`	`model.mk` — The name of the makefile that was created from the template makefile.
`|>MATLAB_ROOT<|`	Path to where MATLAB is installed.
`|>MATLAB_BIN<|`	Location of the MATLAB executable.
`|>MEM_ALLOC<|`	Either `RT_MALLOC` or `RT_STATIC` — indicates how memory is to be allocated.

Table 3-4: Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
`|>MEXEXT<|`	MEX-file extension. See the MATLAB `mexext` command.
`|>MODEL_NAME<|`	Name of the Simulink block diagram currently being built.
`|>MODEL_MODULES<|`	Any additional generated source (`.c`) modules. For example, you can split a large model into two files, *`model`*`.c` and *`model`*`1.c`. In this case, this token expands to *`model`*`1.c`.
`|>MODEL_MODULES_OBJ<|`	Object filenames (`.obj`) corresponding to any additional generated source (`.c`) modules.
`|>MULTITASKING<|`	Yes (1) or No (0) — Is the solver mode multitasking?
`|>NUMST<|`	Number of sample times in the model.
`|>RELEASE_VERSION<|`	The release version of MATLAB, for example, 11.0.
`|>S_FUNCTIONS<|`	List of noninlined S-function (`.c`) sources.
`|>S_FUNCTIONS_LIB<|`	List of S-function libraries available for linking.
`|>S_FUNCTIONS_OBJ<|`	Object (`.obj`) file list corresponding to noninlined S-function sources.
`|>SOLVER<|`	Solver source filename, e.g., `ode3.c`.
`|>SOLVER_OBJ<|`	Solver object (`.obj`) filename, e.g., `ode3.obj`.

Table 3-4: Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
`|>TID01EQ<|`	Either `1` or `0`: Are the sampling rates of the continuous task and the first discrete task equal?
`|>NCSTATES<|`	Number of continuous states.
`|>BUILDARGS<|`	Options passed to `make_rtw.` This token is provided so that the contents of your *`model`*`.mk` file will change when you change the build arguments, thus forcing an update of all modules when your build options change.

In addition target specific tokens defined via the Real-Time Workshop page of the **Simulation Parameters** dialog box are expanded. In the **RTW Options** section of the system target file, any structures in the `rtwoptions` array that contain the field `makevariable` will be expanded. For example, in *`matlabroot`*`/rtw/c/grt/grt.tlc`, you will see a Target Language Compiler comment containing M-file code that sets up `rtwoptions`. Inside `grt.tlc`, the

```
rtwoptions(2).makevariable = 'EXT_MODE'
```

directive causes the `|>EXT_MODE<|` token to be expanded into `1` (on) or `0` (off).

After creating *`model.mk`* from your template makefile, the Real-Time Workshop invokes a `make` command. Make is a utility designed to create an executable from a set of source files. To invoke `make`, the Real-Time Workshop issues this command:

```
makecommand –f model.mk
```

`makecommand` is defined by the `MAKE` macro in your systems template makefile (see Figure 3-7 on page 3-29). You can specify additional options to `make` as described in the “Make Command” on page 3-11. For example, specifying `OPT_OPTS=-O2` to `make_rtw` generates the following `make` command:

```
makecommand –f model.mk OPT_OPTS=–O2
```

Typically, build options are specified as a comment at the top of the template makefile you are using.

You need to configure your template makefile if the options that can be passed to `make` do not provide you with enough flexibility. The Real-Time Workshop uses `make` since it is a very flexible tool. It lets you control nearly every aspect of building the generated code and run-time interface modules into your real-time program.

Make Utilities

To configure your template makefile, you need to understand how `make` works and how `make` processes makefiles. There are several good books on `make`. Consult your local bookstore or refer to the documentation provided with the `make` utility you are using.

There are several different versions of `make` available. Perhaps the most flexible and powerful `make` utility is GNU Make, which is provided by the Free Software Foundation. We provide GNU Make for both UNIX and PC platforms; it can be found in:

```
matlabroot/rtw/bin/arch
```

It is possible to use other versions of `make` with the Real-Time Workshop, but they do not have as rich a set of features. To work with the Real-Time Workshop, any version of `make` must allow this command format:

```
makecommand –f model.mk
```

Structure of the Template Makefiles

Before configuring or creating a Real-Time Workshop template makefile, you should become familiar with its structure. A template makefile has four sections:

- The first section contains an initial comment section that describes what this makefile targets.
- The second section defines macros that tell `make_rtw` how to process the template makefile. The macros are

  `MAKE` — This is the command used to invoke the make utility. For example, if

  `MAKE = mymake`

then the make command that will be invoked is:

```
mymake -f model.mk
```

HOST — What platform this template makefile is targeted for. This can be HOST=PC, UNIX, computer_name (see the MATLAB computer command), or ANY.

BUILD — This tells make_rtw whether or not (BUILD=yes or no) it should invoke make from the Real-Time Workshop build procedure.

SYS_TARGET_FILE — Name of the system target file. This is used for consistency checking by make_rtw to verify that the correct system target file was specified in the Real-Time Workshop page of the **Simulation Parameters** dialog box.

BUILD_SUCCESS — An optional macro that you can use to specify the build success string to be used when looking for successful completion on the PC. For example:

```
BUILD_SUCCESS = ### Successful creation of
```

BUILD_ERROR — An optional macro that you can use to specify the build error message to be displayed when an error is encountered during the make procedure. For example,

```
BUILD_ERROR = ['Error while building ', modelName]
```

DOWNLOAD — An optional macro that you can specify as yes or no. If specified as yes (and BUILD=yes), then make is invoked a second time with the download target:

```
make -f model.mk download
```

DOWNLOAD_SUCCESS — An optional macro that you can use to specify the download success string to be used when looking for a successful download. For example:

```
DOWNLOAD_SUCCESS = ### Downloaded
```

DOWNLOAD_ERROR — An optional macro that you can use to specify the download error message to be displayed when an error is encountered during the download. For example,

```
DOWNLOAD_ERROR = ['Error while downloading ', modelName]
```

- The third section defines the tokens `make_rtw` expands.
- The fourth section contains the `make` rules used in building an executable from the generated source code. The build rules are typically specific to your version of `make`.

The general structure of a template makefile is shown in Figure 3-7 on page 3-29.

```
#-- Section 1: Comments -----------------------------------------------------
#
# Description of what type of target and version of make this template makefile
# is for and any optional build arguments.
#
#-- Section 2: Macros read by make_rtw ----------------------------------------
#
# The following macros are read by the Real-Time Workshop build procedure:
#
#  MAKE            - This is the command used to invoke the make utility.
#  HOST            - What platform this template makefile is targeted for
#                    (i.e., PC or UNIX)
#  BUILD           - Invoke make from the Real-Time Workshop build procedure
#                    (yes/no)?
#  SYS_TARGET_FILE - Name of system target file.

MAKE            = make
HOST            = UNIX
BUILD           = yes
SYS_TARGET_FILE = system.tlc
#-- Section 3: Tokens expanded by make_rtw ------------------------------------
#

MODEL           = |>MODEL_NAME<|
MODULES         = |>MODEL_MODULES<|
MAKEFILE        = |>MAKEFILE_NAME<|
MATLAB_ROOT     = |>MATLAB_ROOT<|
...
COMPUTER        = |>COMPUTER<|
BUILDARGS       = |>BUILDARGS<|

#-- Section 4: Build rules ----------------------------------------------------
#
```

Figure 3-7: Structure of a Template Makefile

Customizing and Creating Template Makefiles

To customize or create a new template makefile, you can copy an existing template makefile to your local working directory and modify it.

The `make` utility processes the *`model`*`.mk` makefile and generates a set of commands based upon dependencies defined in *`model`*`.mk`. For example, to build a program called `test`, `make` must link the object files. However, if the object files don't exist or are out of date, `make` must compile the C code. After `make` generates the set of commands needed to build or rebuild `test`, `make` executes them.

Each version of `make` differs slightly in its features and how rules are defined. For example, consider a program called `test` that gets created from two sources, `file1.c` and `file2.c`. Using most versions of `make`, the dependency rules would be:

```
test: file1.o file2.o
        cc –o test file1.o file2.o

file1.o: file1.c
        cc –c file1.c

file2.o: file2.c
        cc –c file2.c
```

In this example, we assumed a UNIX environment. In a PC environment the file extensions and compile and link commands will be different. The first rule, `test: file1.o file2.o` encountered by `make` will be built. In processing this rule, `make` sees that to build `test`, it needs to build `file1.o` and `file2.o`. To build `file1.o`, `make` processes the rule `file1.o: file1.c` and will compile `file1.c` if `file1.o` doesn't exist or is older than `file1.c`.

The format of Real-Time Workshop template makefiles follows the above example. Our template makefiles use additional features of `make` such as macros and file-pattern-matching expressions. In most versions of `make`, a macro is defined via

```
MACRO_NAME = value
```

References to macros are made via `$(MACRO_NAME)`. When `make` sees this form of expression, it substitutes `$(MACRO_NAME)` with *`value`*.

You can use pattern matching expressions to `make` the dependency rules more general. For example, using GNU Make you could have replaced the two `"file1.o: file1.c"` and `"file2.o: file2.c"` rules with the single rule:

```
%.o : %.c
        cc –c $<
```

The `$<` is a special macro that equates to the dependency file (i.e., `file1.c` or `file2.c`). Thus, using macros and the "`%`" pattern matching character, the above example can be reduced to:

```
SRCS = file1.c file2.c
OBJS = $(SRCS:.c=.o)

test: $(OBJS)
        cc –o $@ $(OBJS)

%.o : %.c
        cc –c $<
```

This example generates the list of objects (`OBJS`) from the list of sources (`SRCS`) by using the string substitution feature for macro expansion. It replaces the source file extension (`.c`) with the object file extension (`.o`). This example also generalized the build rule for the program, `test`, to use the "`$@`" special macro that equates to the name of the current dependency target, in this case `test`.

Generic Real-Time Templates

The Real-Time Workshop includes a set of built-in template makefiles that are set up for generic real-time code generation. These template makefiles allow you to simulate your fixed-step models on your workstation. This section discusses these template makefiles:

- `grt_unix.tmf` — targets UNIX platforms using any ANSI C compiler (`cc` is the default on all platforms except SunOS, where `gcc` is the default).
- `grt_vc.tmf` — creates a generic real-time executable for Windows 95, Windows 98, or Windows NT using Microsoft Visual C/C++.
- `grt_msvc.tmf` — creates a generic real-time project makefile for Windows 95, Windows 98, or Windows NT for use with Microsoft Visual C/C++.
- `grt_watc.tmf` — creates a generic real-time executable for Windows 95, Windows 98, or Windows NT using the Watcom C/C++ compiler.
- `grt_bc.tmf` — creates a generic real-time executable for Windows 95, Windows 98, or Windows NT using the Borland C/C++ compiler.

If you set the template makefile to `grt_default_tmf`, then the Real-Time Workshop will use the correct makefile for your system. For example, it will use `grt_vc.tmf` if you have your PC configured for Visual C/C++. The exception to this is when you want to create a real-time project makefile for Windows. In that case, you must select `grt_msvc.tmf` instead of `grt_default_tmf`.

grt_unix.tmf

The generic real-time template makefile for UNIX platforms is designed to be used with GNU Make. This makefile is set up to conform to the guidelines specified in the IEEE Std 1003.2-1992 (POSIX) standard.

You can supply the following options to `grt_unix.tmf` via arguments to the `make` command, `make_rtw`:

- `OPTS` — User-specific options, for example:
  ```
  make_rtw OPTS="-DMYDEFINE=1"
  ```
- `OPT_OPTS` — Optimization options. The default optimization option is `-O`. To turn off optimization and add debugging symbols, specify the `-g` compiler switch in the `make` command:
  ```
  make_rtw OPT_OPTS="-g"
  ```

- USER_SRCS — Additional user sources, such as files needed by S-functions. For example, suppose you have an S-function called my_sfcn.c, which is built with sfcn_lib1.c, and sfcn_lib2.c. (library routines for use with many S-functions). You can build my_sfcn.c using:

  ```
  mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
  ```

  In this case, the make command for the Real-Time Workshop should be specified as:

  ```
  make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
  ```

  You can avoid using USER_SRCS by specifying

  ```
  set_param('block','SFunctionModules','sfcn_lib1 sfcn_lib2')
  ```

  See “Using S-Functions with the Real-Time Workshop” in Chapter 3 of *Writing S-Functions*.
- USER_INCLUDES — Additional include paths. For example, suppose you have two include paths (/appl/inc and /support/inc) that are required for the sfcn_lib1.c and sfcn_lib2.c files in the above example. The make command would be specified as:

  ```
  make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
           USER_INCLUDES="-I/appl/inc -I/support/inc"
  ```

grt_vc.tmf

The Real-Time Workshop provides a generic real-time template makefile (grt_vc.tmf) to create an executable for Windows 95, Windows 98, and Windows NT using Microsoft Visual C/C++. This template makefile is designed to be used with nmake, which is bundled with Microsoft’s Visual C/C++.

You can supply these options to grt_vc.tmf via arguments to the make command, make_rtw:

- OPTS — User-specific options, for example,

  ```
  make_rtw OPTS="-DMYDEFINE=1"
  ```

- OPT_OPTS — grt_vc.tmf optimization options. The default optimization option is -Ot. To turn off optimization and add debugging symbols, specify the -Zd compiler switch in the make command:

  ```
  make_rtw OPT_OPTS="-Zd"
  ```

- `USER_SRCS` — Additional user sources, such as files needed by S-functions. For example, suppose you have an S-function called `my_sfcn.c`, which is built with `sfcn_lib1.c`, and `sfcn_lib2.c`.(library routines for use with many S-functions). You can build `my_sfcn.c` using:

  ```
  mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
  ```

  In this case, the `make` command for the Real-Time Workshop should be specified as:

  ```
  make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
  ```

  You can avoid using `USER_SRCS` by specifying

  ```
  set_param('block','SFunctionModules','sfcn_lib1 sfcn_lib2')
  ```

  See "Using S-Functions with the Real-Time Workshop" in Chapter 3 of *Writing S-Functions*.
- `USER_INCLUDES` — Additional include paths for example suppose you have two include paths (`c:\appl\inc` and `c:\support\inc`) that are required for the `sfcn_lib1.c` and `sfcn_lib2.c` files in the above example. The `make` command would be specified as:

  ```
  make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
           USER_INCLUDES="–Ic:\appl\inc –Ic:\support\inc"
  ```

grt_msvc.tmf

The Real-Time Workshop provides a generic real-time template makefile (`grt_msvc.tmf`) to create a Microsoft Visual C/C++ project makefile called *`model`*`.mak` for Windows 95, Windows 98, and Windows NT. `grt_msvc.tmf` is designed to be used with `nmake`, which is bundled with MicroSoft's Visual C/C++.

You can supply the following options to `grt_msvc.tmf` via arguments to the `make` command, `make_rtw`:

- `OPTS` — User-specific options, for example,

  ```
  make_rtw OPTS="/D MYDEFINE=1"
  ```

- `USER_SRCS` — Additional user sources, such as files needed by S-functions. For example, suppose you have an S-function called `my_sfcn.c`, which is built with `sfcn_lib1.c`, and `sfcn_lib2.c`. (library routines for use with many S-functions). You can build, `my_sfcn.c` using:

```
mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
```

In this case, the `make` command for the Real-Time Workshop should be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
```

You can avoid using `USER_SRCS` by specifying

```
set_param('block','SFunctionModules','sfcn_lib1 sfcn_lib2')
```

See "Using S-Functions with the Real-Time Workshop" in Chapter 3 of *Writing S-Functions*.

- `USER_INCLUDES` — Additional include paths. For example suppose you have two include paths (`c:\appl\inc` and `c:\support\inc`) that are required for the `sfcn_lib1.c` and `sfcn_lib2.c` files in the above example. The `make` command would be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
         USER_INCLUDES="/I c:\appl\inc /I c:\support\inc"
```

grt_watc.tmf

The Real-Time Workshop provides a generic real-time template makefile (`grt_watc.tmf`) to create an executable for Windows 95, Windows 98, and Windows NT using Watcom C/C++. This template makefile is designed to be used with `nmake`, which is bundled with Watcom C/C++.

The following options can be supplied to `grt_watc.tmf` via arguments to the `make` command, `make_rtw`. Note that the location of the quotes are different from the other compilers and make utilities discussed in this chapter:

- `OPTS` — User specific options, for example,

```
make_rtw "OPTS=-DMYDEFINE=1"
```

- `OPT_OPTS` — The default optimization option is `-oxat`. To turn off optimization and add debugging symbols, specify the `-d2` compiler switch in the `make` command:

```
make_rtw "OPT_OPTS=-d2"
```

- USER_OBJS — Additional object (.obj) files, which are to be created from user sources, such as files needed by S-functions.For example, suppose you have an S-function called my_sfcn.c, which is built with sfcn_lib1.c, and sfcn_lib2.c. (library routines for use with many S-functions). You can build, my_sfcn.c using:

  ```
  mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
  ```

  In this case, the make command for the Real-Time Workshop should be specified as:

  ```
  make_rtw "USER_OBJS=sfcn_lib1.obj sfcn_lib2.obj"
  ```

  You can avoid using USER_OBJS by specifying

  ```
  set_param('block','SFunctionModules','sfcn_lib1 sfcn_lib2)
  ```

  See “Using S-Functions with the Real-Time Workshop” in Chapter 3 of *Writing S-Functions*.
- USER_PATH — The directory path to the source (.c) files, which are used to create any .obj files specified in USER_OBJS. Multiple paths must be separated with a semicolon (e.g., USER_PATH="*path1*;*path2*"). For example, suppose sfcn_lib1.c exists in your local directory and sfcn_lib2.c exists in c:\appl\src. Then the make command would be specified as:

  ```
  make_rtw "USER_OBJS=sfcn_lib1.obj sfcn_lib2.obj"
           "USER_PATH=c:\appl\src"
  ```

- USER_INCLUDES — Additional include paths. For example, suppose you have two include paths (c:\appl\inc and c:\support\inc) that are required for the sfcn_lib1.c and sfcn_lib2.c files in the above example. The make command would be specified as:

  ```
  make_rtw "USER_OBJS=sfcn_lib1.obj sfcn_lib2.obj"
           "USER_PATH=c:\appl\src"
           "USER_INCLUDES= –Ic:\appl\inc –Ic:\support\inc"
  ```

grt_bc.tmf

The Real-Time Workshop provides a generic real-time template makefile (grt_bc.tmf) to create an executable for Windows 95, Windows 98, and Windows NT using Borland C/C++.

You can supply these options to grt_bc.tmf via arguments to the make command, make_rtw:

- `OPTS` — User-specific options, for example,

  ```
  make_rtw OPTS="-DMYDEFINE=1"
  ```

- `OPT_OPTS` — `grt_bc.tmf` optimization options. To turn off optimization and add debugging symbols, specify the `-v` compiler switch in the `make` command:

  ```
  make_rtw OPT_OPTS="-v"
  ```

- `USER_SRCS` — Additional user sources, such as files needed by S-functions. For example, suppose you have an S-function called `my_sfcn.c`, which is built with `sfcn_lib1.c`, and `sfcn_lib2.c`.(library routines for use with many S-functions). You can build `my_sfcn.c` using:

  ```
  mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
  ```

  In this case, the `make` command for the Real-Time Workshop should be specified as:

  ```
  make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
  ```

  You can avoid using `USER_SRCS` by specifying

  ```
  set_param('block','SFunctionModules','sfcn_lib1','sfcn_lib2)
  ```

  See "Using S-Functions with the Real-Time Workshop" in Chapter 3 of *Writing S-Functions*.

- `USER_INCLUDES` — Additional include paths. For example, suppose you have two include paths (`c:\appl\inc` and `c:\support\inc`) that are required for the `sfcn_lib1.c` and `sfcn_lib2.c` files in the above example. The `make` command would be specified as:

  ```
  make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
           USER_INCLUDES="-Ic:\appl\inc -Ic:\support\inc"
  ```

File and Function Splitting

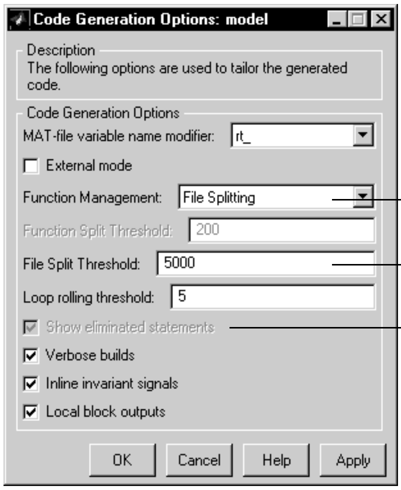
To support compilers with file size limitations, the Real-Time Workshop provides support for splitting the source code whenever the code size exceeds a specified threshold. This option is available with the generic real-time (`grt`), embedded-C (`ert`), and rapid simulation (`rsim`) targets. These targets support:

- File splitting
- Function splitting
- File and function splitting

Both file and function splitting require at least Perl v5.000. The MathWorks ships Perl with MATLAB on PC platforms. For UNIX platforms, you must provide the Perl binary; that is, you must place Perl on your path. Perl is available from `http://www.perl.com`.

File Splitting

Generated source files may have too many lines of code. File splitting provides a way to reduce file size. You can control file splitting by specifying a function size threshold on the **Code Generation Options** dialog box.


Code Generation Options: model
Description
The following options are used to tailor the generated code.
Code Generation Options
MAT-file variable name modifier: rt_
☐ External mode
Function Management: File Splitting
Function Split Threshold: 200
File Split Threshold: 5000
Loop rolling threshold: 5
☑ Show eliminated statements
☑ Verbose builds
☑ Inline invariant signals
☑ Local block outputs

Select File Splitting from the **Function Management** pull-down menu.

Specify the **File Split Threshold**. This can be any positive integer.

The **Show eliminated statements** check box is disabled when file splitting occurs.

File splitting is controlled by the TLC variable `FileSizeThreshold`, which by default is set to 5000 lines.

You must turn off the TLC variable `ShowEliminatedStatements` for file splitting to work. This is required because the `#if ... #endif` constructs that surround eliminated statements do not cross file boundaries. Files will only be split at function boundaries. If a source file contains a few long functions, file splitting by itself may not be effective. In this case, you can choose to use both function and file splitting. Function splitting will produce smaller functions; file splitting can then split at the function boundaries to produce smaller files.

A Perl scripts performs file splitting by text-based post-processing of the generated source code.

To illustrate how the source code splits, assume the model's output section is being generated when the file size threshold is exceeded:

Inside *`model`*`.c`:

```
void MdlOutputs(int_T tid)
{
  :
  :
  /* File size threshold exceeded. Splitting outputs into
     source file model1.c */
  {
      extern void MdlOutputs_split1(int_T tid);
      MdlOutputs_split1(tid);
  }
}
/* EOF model.c */
```

Inside *`model`*`1.c`

```
void MdlOutputs_split1(int_T tid)
{
  :
  : continues where model.c left off
  :
}
:
:
```

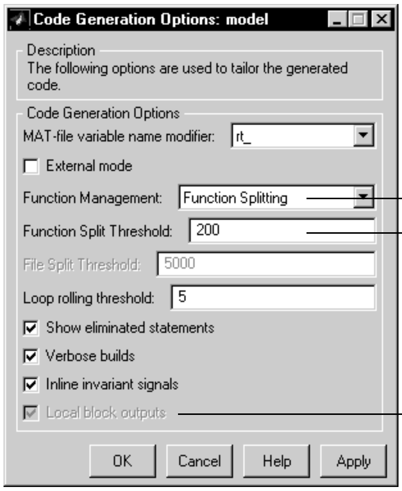
The file *`model1`*`.c` includes *`model`*`.h`, but not *`model`*`.prm`.

Note that if *`model1`*`.c` (or any of the other split files) already exists, it will be overwritten without warning.

Function Splitting

Sometimes a file may not be too large, but a single function contained in the file might be too long. Because of this, your compiler may not be able to perform code optimizations. To solve this problem, you can use function splitting to create smaller functions.

You can control function splitting by using the **Code Generation Options** dialog box.



Code Generation Options: model

Description
The following options are used to tailor the generated code.

Code Generation Options
MAT-file variable name modifier: rt_
☐ External mode
Function Management: Function Splitting
Function Split Threshold: 200
File Split Threshold: 5000
Loop rolling threshold: 5
☑ Show eliminated statements
☑ Verbose builds
☑ Inline invariant signals
☑ Local block outputs

Select Function Splitting from the **Function Management** pull-down menu.

Specify the **Function Split Threshold**. This can be any positive integer.

The **Local block outputs** check box is disabled when file splitting occurs. Local variable declarations in a function are not copied over to split functions.

The function size threshold is specified by the TLC variable `FunctionSizeThreshold`, which by default is set to zero.

Functions will only be split at block boundaries or at `ssIsSampleHit` calls at the base scope level. If `ssIsSampleHit` or a block boundary is not found at the function split threshold, the next line that satisfies either of these conditions becomes the location where the function is split.

A Perl script performs function splitting by text-based post-processing of the generated source code.

Function splitting will only split functions that begin with the characters `Mdl`, `Sys`, or the name of the model. The naming convention for split functions takes the original function name and appends an integer, starting with 1 and incrementing by one each time the function is split. For example, if your original function is `MdlOutput`, the split functions will be

```
MdlOuput1
MdlOutput2
```

and so on.

File and Function Splitting

You can perform function and file splitting together. To do this, select Function and File Splitting from the **Function Management** pull-down menu. The rest of the steps are the same as described in the function and file splitting sections respectively.

External Mode

Introduction

External mode is a simulation mode provided by the Real-Time Workshop that supports on-the-fly parameter tuning in a real-time environment. “External” refers to the fact that this mode involves two separate computing environments, a host and a target. The *host* is the computer where MATLAB and Simulink are executing, and the *target* is the computer where the executable that you create with the Real-Time Workshop runs. Using external mode, you can modify the block parameters and view and log block outputs in Scope blocks.

External mode works by establishing a communications channel between Simulink and the Real-Time Workshop generated code. This communication channel can be a networking protocol such as the Transmission Control Protocol (TCP) or shared memory.

In external mode, Simulink waits for parameter changes. Once Simulink receives the parameter changes, it provides the target with the new parameters.

The external mode implementation included with the Real-Time Workshop relies on the TCP to provide a communications layer between the target (where the executable that you create runs) and the host (where your model resides).

The external mode implementation included with the Real-Time Workshop for use with the Real-Time Windows Target uses shared memory as the communication channel. Note that the Real-Time Windows Target is a separate product from the Real-Time Workshop.

This chapter discusses these topics:

- How to set up external mode on your computer (self-targeting your machine)
- How to use the TCP socket-based external mode implementation that is included in Simulink
- Parameter tuning
- Signal viewing and logging by using Simulink Scope blocks
- How to create your own external mode communication channel for your custom target

Before reading this chapter, you should have read Chapter 3, “Code Generation and the Build Process.”

External Mode

Simulink external mode is a mechanism that manages communication between Simulink and stand-alone programs built with the Real-Time Workshop. This mechanism allows you to use a Simulink block diagram as a graphical front end to the corresponding program (i.e., the program built from code generated for that block diagram).

In external mode, whenever you change parameters in the block diagram, Simulink automatically downloads them to the executing program. This feature lets you perform parameter tuning in your program without recompiling.

External mode makes use of the client/server model of computing, where Simulink is a client that sends a request to the server (the external program) to install new parameter values.

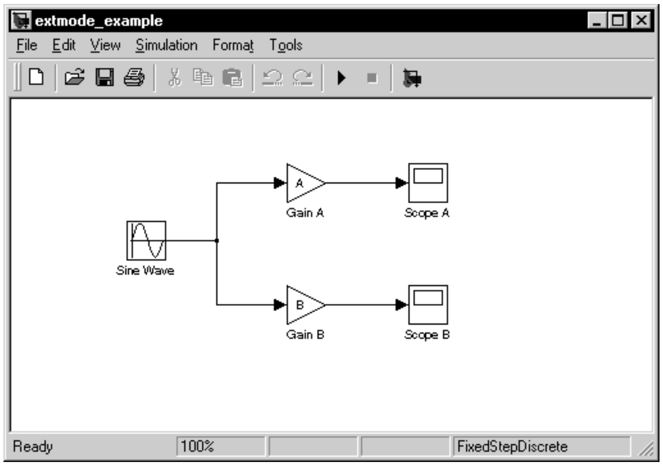
Getting Started with External Mode Using grt

This section provides a step-by-step set of instructions for getting started with external mode. Since the example presented is shown using the generic real-time (`grt`) target, you don't need any hardware other than your basic computer setup to get the example running. This process, running the generated executable on the host computer, is known as *self-targeting*. If you are running UNIX instead of Windows 95, Windows 98, or Windows NT, the steps presented here are the same, except that you should open an xterm for your target instead of a DOS command prompt.

For a more thorough description of external mode, including a discussion of all the options available to it, see "External Mode GUI" on page 4-10.

Setting Up the External Mode Host Environment

First, 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 should look like this.



Setting up external mode for this model involves the following steps:

1 Set the simulation to **External** under the **Simulation** menu.

2 Open the **Simulation Parameters** dialog box. On the Solver page, set the **Solver Options Type** to `Fixed-step`, and **Fixed-step Size** to `0.01`. Leave

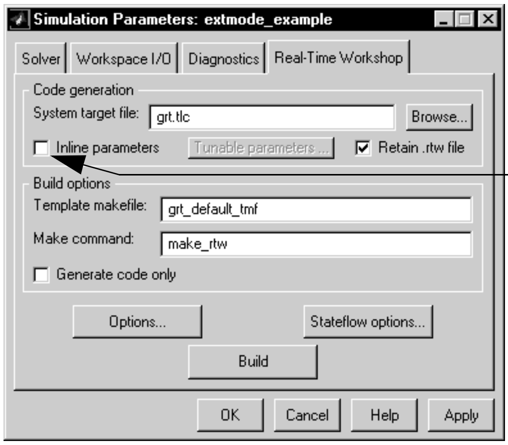
the **Stop Time** set to its default value. Set the **Decimation** to 1 (no decimation).

3 On the Workspace I/O page, select **Time** and **Output**. By default the variable names should be `tout` and `yout`.

4 On the Real-Time Workshop page, click the **Browse** button and select the generic real-time target (default for PC and UNIX). For a description of the System Target File Browser, see “The System Target File Browser” on page 3-11.

5 Click the **Options** button and click the **External mode** check box

The finished Real-Time Workshop page should look like this picture.



Simulation Parameters: extmode_example

Real-Time Workshop

Code generation

System target file: grt.tlc

[ ] Inline parameters

[✓] Retain .rtw file

Build options

Template makefile: grt_default_tmf

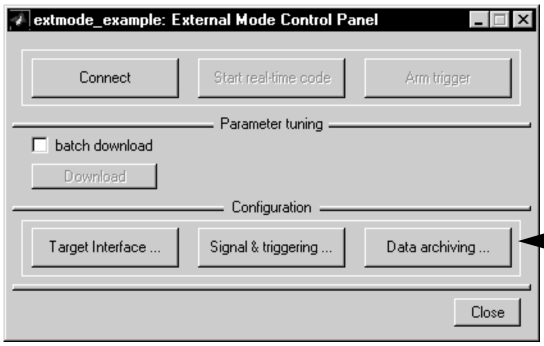
Make command: make_rtw

[ ] Generate code only

Note: do not click the **Inline parameters** check box. This option does not work with external mode.

Setting External Mode Features

The External Mode Control Panel allows convenient navigation of all the features available to external mode. To open the External Mode Control Panel, select **External Mode Control Panel** under the **Tools** menu. This dialog box opens.



These three buttons open three separate dialog boxes.

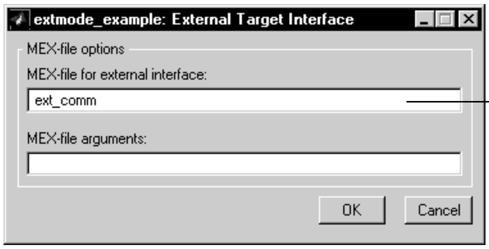
The top four buttons for use after you have started your real-time program on your target. The three lower buttons open three separate dialog boxes:

- 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 they are triggered. You must select signals in this dialog box before starting the real-time program. The term *triggering* refers to when signals should be acquired and displayed.
- The **Data archiving** button opens the **External Data Archiving** dialog box. See "Data Archiving" on page 4-15 for a discussion of this feature.

Target Interface

Before you can run your model in external mode, you must first configure the target interface and set various triggering and data archiving options. By default, the **MEX-file for external interface** should be set to `ext_comm`. This file supports the TCP communications protocol that external mode uses to communicate between the host and target. **MEX-file arguments** can be left blank.

To open the **External Target Interface** dialog box, click the **Target Interface** button. The dialog box should look like this.


extmode_example: External Target Interface
MEX-file options
MEX-file for external interface:
ext_comm
MEX-file arguments:
OK
Cancel

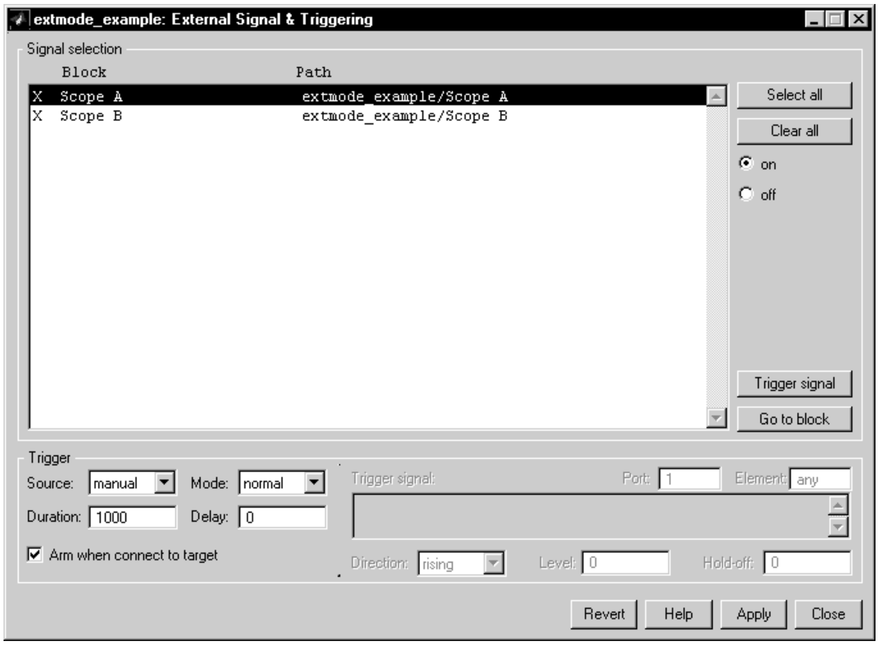
Specify `ext_comm` here. This is the MEX-file that provides TCP communication for the executable created by the `grt` target.

Signals & Triggering

The model is nearly ready to build. Click the **Signal & triggering** button on the External Mode Control Panel. This opens the **External Data Logging Configuration** dialog box. In this dialog box, specify the following:

- Click **Select All**. This configures both Scope blocks in your model so that you will be able to see the output signals in real-time.
- Under the **Trigger** panel, set **Source** to `manual`, **Mode** to `normal`, **Duration** to `1000`, and **Delay** to `0` (no delay).
- Click the **Arm when connect to target** check box.

The dialog box should look like this when you're finished.



```
   Block                   Path
X  Scope A                 extmode_example/Scope A
X  Scope B                 extmode_example/Scope B
```

[✓ on] off

Source: manual   Mode: normal
Duration: 1000   Delay: 0
[✓] Arm when connect to target

Port: 1   Element: any
Direction: rising   Level: 0   Hold-off: 0

Prior to building your model, you must select values for gains `A` and `B`. The example discussed here uses `A=2` and `B=3`. Once you've selected two values, click the **Build** button to generate code in external mode.

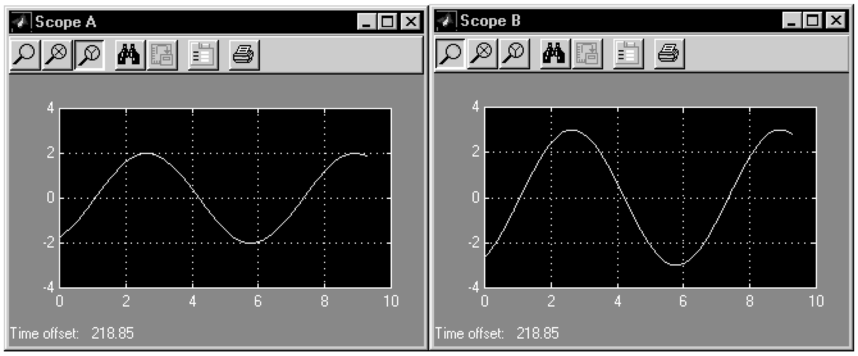
Note: You do not need to specify any data archiving options for this example. See "External Mode GUI" on page 4-10 for a complete description of all available options for external model.

Running External Mode on Your Target

To run external mode, you must open an MS-DOS command prompt (on UNIX systems, an Xterm window). At the MS-DOS command prompt, type

```
modelname -tf 0
```

and the model that you've created will execute. The `-tf` switch overrides the stop time set for the model in Simulink. The `0` value directs the model to run indefinitely. Under the **Simulation** menu of your model, select **Connect to target**. You should see your output signals `A` and `B` on the two scopes in your model. You may have to adjust the parameters of your scopes to fit your signal, depending on what values you selected for `A` and `B`. With `A=2` and `B=3`, the output looks like this:



You can change signals on-the-fly by opening the dialog boxes for the Gain blocks in the model and entering a new numerical value for the gain or by setting the gain value in the MATLAB workspace. In the latter case, you must type **Crtl-D** or select **Update Diagram** under the **Edit** menu of your model after changing the value of the MATLAB variable. Changing parameters in your model in this fashion is called *downloading*.

The following sections discuss the options available for external mode.

External Mode GUI

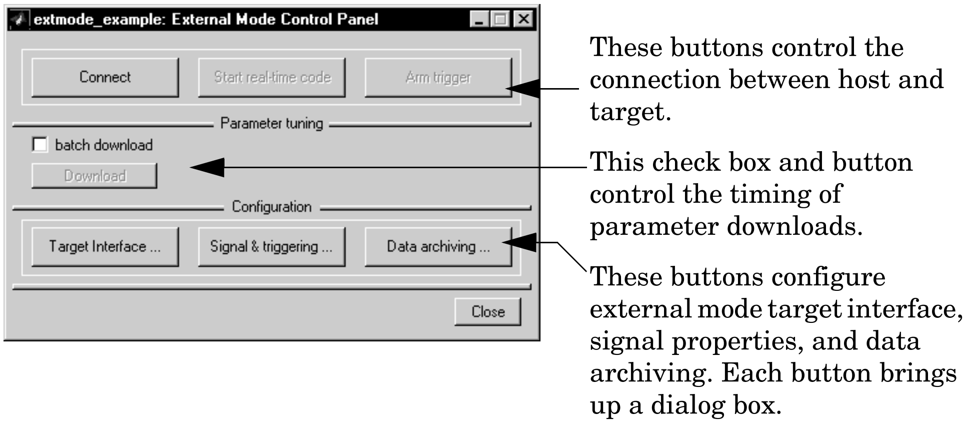
The Real-Time Workshop provides an extensive graphical user interface (GUI), which includes four separate windows, that supports a wide variety of features in external mode. The four windows include the following:

- External Mode Control Panel
- External Target Interface dialog box
- External Signal & Triggering dialog box
- External Data Archiving dialog box

The next sections discuss each of these in turn. Although some of these features were discussed in “Getting Started with External Mode Using grt” on page 4-4, the following sections go into much greater detail.

External Mode Control Panel

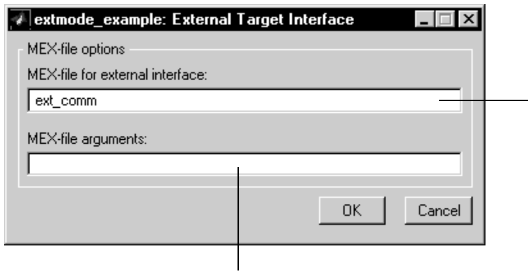
You can use external mode with your target, providing an external interface MEX-file exists that communicates with your target system. To configure external mode, open the **External Mode Control Panel**.


extmode_example: External Mode Control Panel
Connect
Start real-time code
Arm trigger
Parameter tuning
batch download
Download
Configuration
Target Interface ...
Signal & triggering ...
Data archiving ...
Close
These buttons control the connection between host and target.
This check box and button control the timing of parameter downloads.
These buttons configure external mode target interface, signal properties, and data archiving. Each button brings up a dialog box.

The **External Mode Control Panel** allows you to navigate through the various features supported by external mode.

Target Interface

Pressing the **Target Interface** button activates the **External Target Interface** dialog box.



MEX-file for external interface: ext_comm

MEX-file arguments:

Specify `ext_comm` here for `grt` and Tornado targets, `win_tgt` for the Real-Time Windows Target, or the name of a third party external mode link.

By default, leave this blank. Alternatively, for `ext_comm` you can type in your local target name, verbosity level, and port number 17725:
`'localtargetname', 1, 17725`

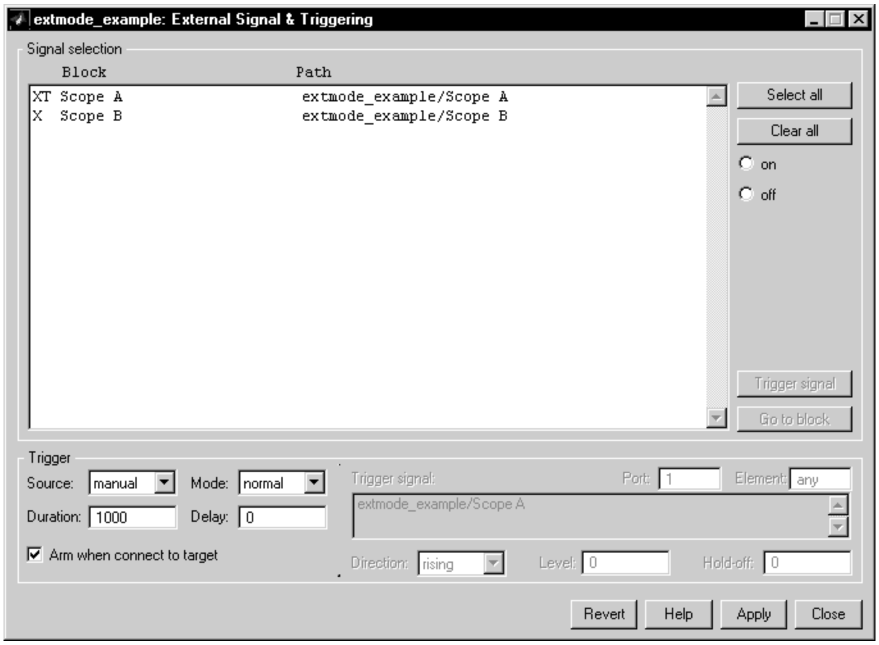
You must set the MEX-file options:

- MEX-file for external interface — name of the external interface MEX-file
- MEX-file arguments — any arguments for the external interface MEX-file

For example, the TCP-based `ext_comm` external interface MEX-file described in the following sections of this chapter contains three optional arguments: the network name of your target, the verbosity level, and a TCP server port number.

External Signal & Triggering

Pressing the **Signal & Triggering** button activates the **External Signal & Triggering** dialog box.


extmode_example: External Signal & Triggering

Signal selection

```
    Block                        Path
XT  Scope A                      extmode_example/Scope A
X   Scope B                      extmode_example/Scope B
```

Trigger

Source: manual Mode: normal

Duration: 1000 Delay: 0

☑ Arm when connect to target

Trigger signal: extmode_example/Scope A Port: 1 Element: any

Direction: rising Level: 0 Hold-off: 0

This windows provides support for:

- Signal selection — specify which Scope blocks are active in external mode
- Trigger selection — select the signal that defines when to display data in Scope blocks.
- Trigger options — configure how and when to display data in Scope blocks

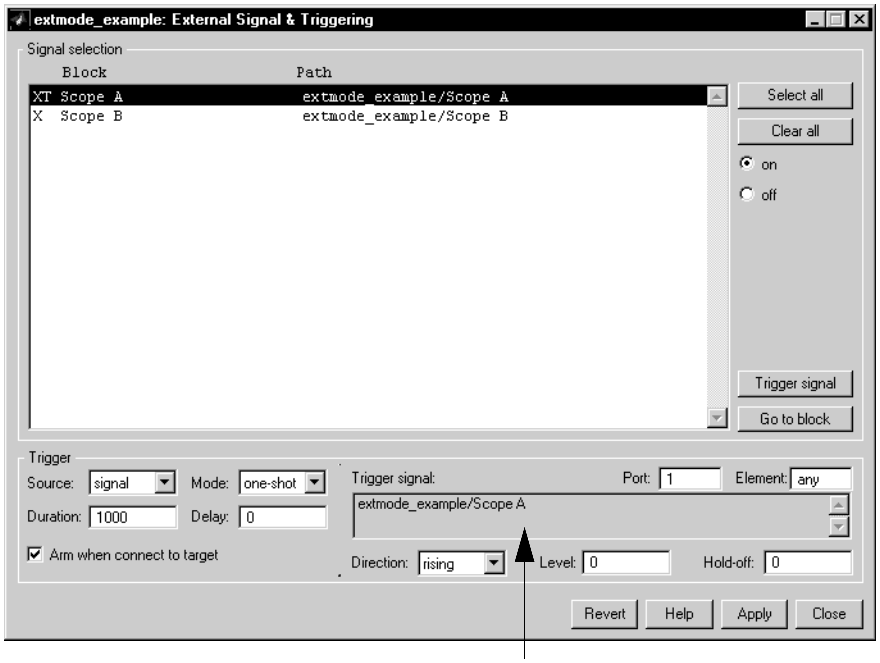
Signal Selection. External mode currently supports Scope blocks for signal selection. Any Scope block in your model will appear in the **External Signal & Triggering** dialog box in the Signal selection list. The **Select all** and **Clear all** buttons toggle between selecting and deselecting all listed signals. A signal is selected if an "X" appears to the left of the block's name. Also, when a signal is selected, the radio button (on and off) is activated if you click on the block's name with your mouse. Alternatively, you can double-click a signal to toggle between selection and deselection.

Trigger Signal Selection. You select a trigger signal by choosing it from the Signal selection list and pressing the **Trigger signal** button. A 'T' appears to the left of the block's name if you have chosen it as a trigger. Note that this setting only applies when the Trigger source has been set to `signal`.

Trigger Options. There are various options available under the Trigger panel located at the bottom of the **External Signal & Triggering** dialog box:

- **Source** — manual or signal. Selecting `manual` directs external mode to start logging data once the **Arm trigger** button on the **External Mode Control Panel** is clicked. Selecting `signal` tells external mode to start logging data when the selected trigger signal satisfies the trigger conditions (that is, crosses the trigger level in the specified direction).
- **Duration** — the number of base rate steps for which external mode logs data after a trigger event. For example, if the fastest rate in the model is 1 second and a 1 Hz signal is being logged for a duration of 10 seconds, then external mode will collect 10 samples. If a 2 Hz signal is logged, only 5 samples will be collected.
- **Mode** — normal or one-shot. In normal mode, external mode automatically rearms the trigger after each trigger event. In one-shot mode, external mode collects only one buffer of data each time you arm the trigger.
- **Delay** — you can specify a delay in your trigger. Delays, expressed in base rate steps, can be positive or negative and represent the amount of time that elapses between the trigger occurrences and the start of data collections. A negative delay corresponds to pretriggering.
- **Arm when connect to target** — once your trigger is armed, external mode monitors the trigger signal for the specified trigger condition. This check box tells external mode to arm automatically once you've connected to the target.

If you have selected a trigger, the Trigger signal panel activates.



extmode_example: External Signal & Triggering

Signal selection

```
    Block                       Path
XT  Scope A                     extmode_example/Scope A
X   Scope B                     extmode_example/Scope B
```

Trigger

Source: signal Mode: one-shot
Duration: 1000 Delay: 0
☑ Arm when connect to target

Trigger signal: extmode_example/Scope A
Port: 1 Element: any
Direction: rising Level: 0 Hold-off: 0

The Trigger signal panel

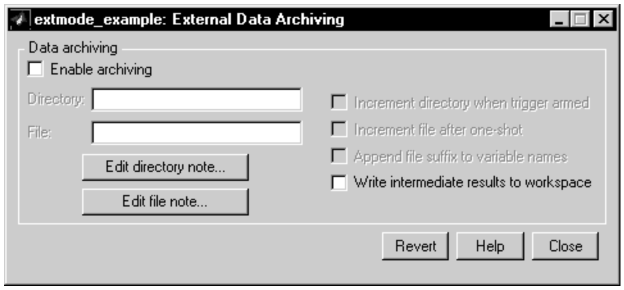
By default, any element of the first input port of the specified block can cause the trigger to fire (i.e., Port 1, any element). You can modify this behavior by adjusting the **Port** and **Element** fields located on the right side of the Trigger panel. The **Port** field accepts a number or the keyword `last`. The **Element** field accepts a number or the keywords `any` and `last`.

These are the options that you can specify under the Trigger signal panel:

- Direction — rising, falling, or either. This specifies the triggering event.
- Level — specifies a bias in the trigger signal. By default, the level is 0. Setting the level to 1, for example, directs the trigger to fire once the signal has crossed 1 in the specified direction.
- Hold-off — only applies to normal mode. Expressed in base rate steps, hold-off is the time between the termination of one trigger event and the rearming of the trigger.

Data Archiving

Pressing the **Data Archiving** button opens the **External Data Archiving** dialog box.


extmode_example: External Data Archiving
Data archiving
Enable archiving
Directory:
File:
Edit directory note...
Edit file note...
Increment directory when trigger armed
Increment file after one-shot
Append file suffix to variable names
Write intermediate results to workspace
Revert
Help
Close

This panel supports various features in external mode, including:

- Directory notes — directory level annotation
- File notes — file level annotation
- Data archiving — Automated data logging and storage

Directory Notes. Pressing the **Edit directory note** button opens the MATLAB editor. Place comments in this window that you want saved in the specified directory.

File Notes. Pressing **Edit file note** opens a file finder window that is, by default, set to the last file to which you have written. Selecting any MAT-file opens an edit window. Add or edit comments in this window that you want saved with your individual MAT-file.

Data Archiving. Clicking the **Enable Archiving** check box activates the automated data archiving features of external mode. To understand how the archiving features work, it is necessary to consider the handling of data when archiving is not enabled. There are two cases, one-shot and normal mode.

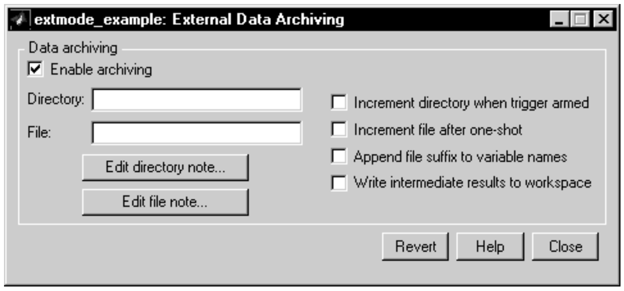
In one-shot mode, after a trigger event occurs, each Scope block writes its data to the workspace just as it would at the end of a simulation. If another one-shot is triggered, the existing workspace data will be overwritten.

In normal mode, external mode automatically rearms the trigger after each trigger event. Consequently, you can think of normal mode as a series of one-shots. Each one-shot in this series, except for the last, is referred to as an *intermediate result*. Since the trigger can fire at any time, writing intermediate results to the workspace generally results in unpredictable overwriting of the workspace variables. For this reason, the default behavior is to write only the results from the final one-shot to the workspace. The intermediate results are discarded. If you know that sufficient time exists between triggers for inspections of the intermediate results, then you can override the default behavior by checking the **Write to intermediate workspace** check box. Note that this option does not protect the workspace data from being overwritten by subsequent triggers.

The options in the **External Data Archiving** dialog box support automatic writing of logging results, including intermediate results, to disk. Data archiving provides the following settings:

- **Directory** — specifies the directory in which data is saved. External mode appends a suffix if you select **Increment directory when trigger armed**.
- **File** — specifies the filename in which data is saved. External mode appends a suffix if you select **Increment file after one-shot**.
- **Increment directory when trigger armed** — external mode uses a different directory for writing log files each time that you press the **Arm trigger** button. The directories are named incrementally; for example: `dirname1`, `dirname2`, and so on.
- **Increment file after one-shot** — new data buffers are saved in incremental files: `filename1`, `filename2`, etc. Note that this happens automatically in normal mode.
- **Append file suffix to variable names** — whenever external mode increments filenames, each file contains variables with identical names. Choosing **Append file suffix to variable name** results in each file containing unique variable names. For example, external mode will save a variable named `xdata` in incremental files (`file_1`, `file_2`, etc.) as `xdata_1`, `xdata_2`, and so on. This is useful if you want to compare variables in MATLAB. Without the unique names, each instance of `xdata` would overwrite the previous one in the MATLAB workspace.

This picture shows the **External Data Archiving** dialog box with enabled archiving.


extmode_example: External Data Archiving
Data archiving
Enable archiving
Directory:
File:
Edit directory note...
Edit file note...
Increment directory when trigger armed
Increment file after one-shot
Append file suffix to variable names
Write intermediate results to workspace
Revert
Help
Close

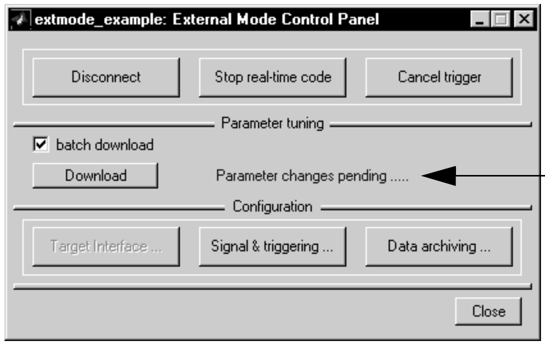
Unless you select **Enable archiving**, entries for the **Directory** and **File** fields are not accepted.

Parameter Tuning

You can set parameter tuning options on the **External Mode Control Panel**. The **batch download** check box enables/disables batch parameters changes. When batch mode is enabled, changes made to block parameters are stored locally and sent only after you have pressed the **Download** button. External mode indicates unsent changes by a “changes pending” status message that appears to the right of the download button. The changes pending status indicator disappears after Simulink receives notification from the target that the new parameters have been installed into the Real-Time Workshop parameter vector.

If batch mode is not enabled, changes made to block parameters are sent immediately. You can change block parameters that are MATLAB workspace variables by modifying the variable in the MATLAB workspace and clicking the **Download parameters** button.

This picture shows the **External Mode Control Panel** with the batch download option activated:



"Parameter changes pending" appears here if you have unsent parameter value changes.

External Mode Operation

When external mode is enabled, Simulink does not simulate the system represented by the block diagram. Instead, it performs an **Update Diagram** and downloads current values of all parameters as soon as you start external mode. After the initial download, Simulink remains in a waiting mode until you change parameters in the block diagram or until it receives data from the target.

The Download Mechanism

When you change a parameter in the block diagram, Simulink calls the external interface MEX-file, passing new parameter values (along with other information) as arguments. (MEX-files are subroutines that are dynamically linked to Simulink.)

The external interface MEX-file contains code that implements one side of the interprocess communication (IPC) channel. This channel connects the Simulink process (where the MEX-file executes) to the process that is executing the external program.

The MEX-file transfers the new parameter values via this channel to the external program. The other side of the communication channel is implemented within the external program. This side writes the new parameter values into the program's `SimStruct` (the data structure containing all data relating to the model code).

The Simulink side initiates the parameter download operation by calling a procedure on the external program side. In the general terminology of client/server computing, this means the Simulink side is the client and the external program is the server. The two processes can be remote, in which case a communication protocol is used to transfer data, or they can be local and employ shared memory to transfer data.

The following diagram illustrates this relationship.

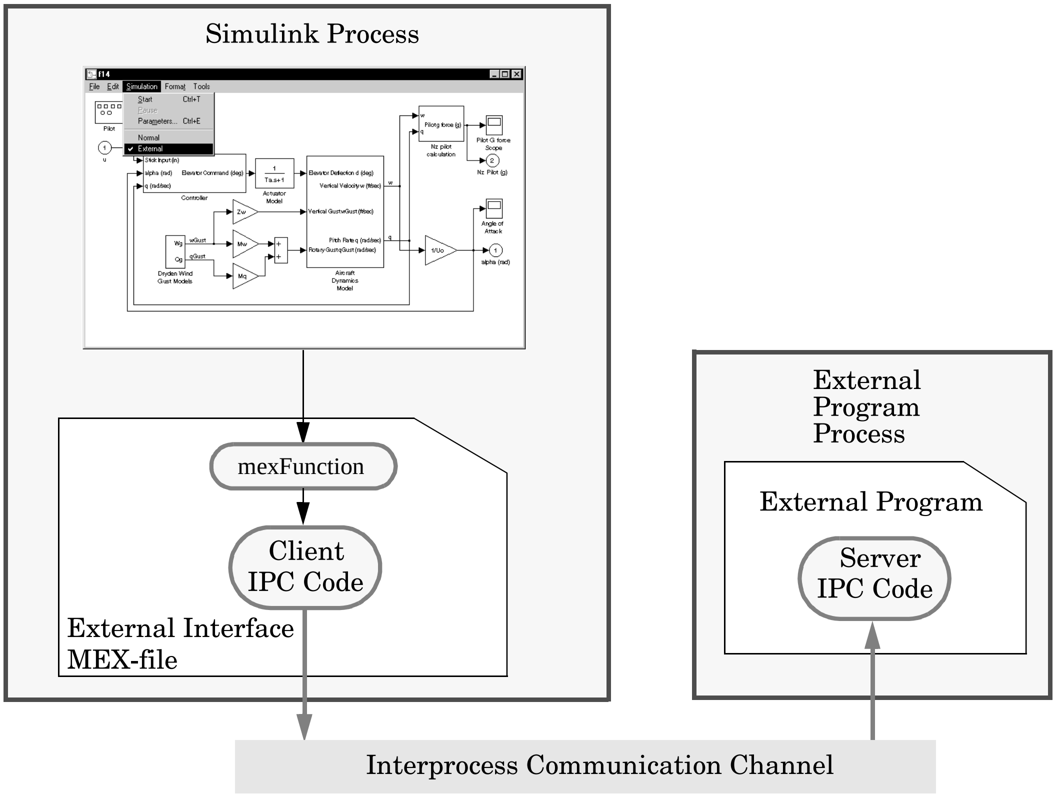


Figure 4-1: External Mode Architecture

Simulink calls the external interface MEX-file whenever you change parameters in the block diagram. The MEX-file then downloads the parameters to the external program via the communication channel.

Limitations

In general, you cannot change a parameter if doing so results in a change in the structure of the model. For example, you cannot change:

- The number of states, inputs, or outputs of any block
- The sample time or the number of sample times
- The integration algorithm for continuous systems
- The name of the model or of any block
- The parameters to the Fcn block

If you cause any of these changes to the block diagram, then you must rebuild the program with newly generated code.

However, parameters in transfer function and state space representation blocks *can* be changed in specific ways:

- The parameters (numerator and denominator polynomials) for the Transfer Fcn (continuous and discrete) and Discrete Filter blocks can be changed (as long as the number of states does not change).
- Zero entries in the State Space and Zero Pole (both continuous and discrete) blocks in the user-specified or computed parameters (i.e., the A, B, C, and D matrices obtained by a zero-pole to state-space transformation) cannot be changed once external simulation is started.
- In the State Space blocks, if you specify the matrices in the controllable canonical realization, then all changes to the A, B, C, D matrices that preserve this realization and the dimensions of the matrices are allowed. For a discussion of controllable canonical realizations, see *Linear Systems Theory* by C.T. Chen.

TCP Implementation

The Real-Time Workshop provides code to implement both the client and server side based on TCP. You can use socket-based external mode implementation provided by the Real-Time Workshop with the generated code, provided that your target system has an external mode server. For example, the Tornado environment and generic real-time targets contain an external server.

The following section discusses how to use external mode with real-time programs on a UNIX or PC system. Chapter 8, “Targeting Tornado for Real-Time Applications,” illustrates the use of external mode in the Tornado environment.

Using the TCP Implementation

This section describes how to use the TCP-based client/server implementation provided with the Real-Time Workshop. In order to use Simulink external mode, you must:

- Specify the name of the external interface MEX-file in the **External Target Interface** dialog box.
- Configure the template makefile so that it links the proper source files for the TCP server code and defines the necessary compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set Simulink to external mode and start the simulation.

This figure shows the structure of the TCP-based implementation:

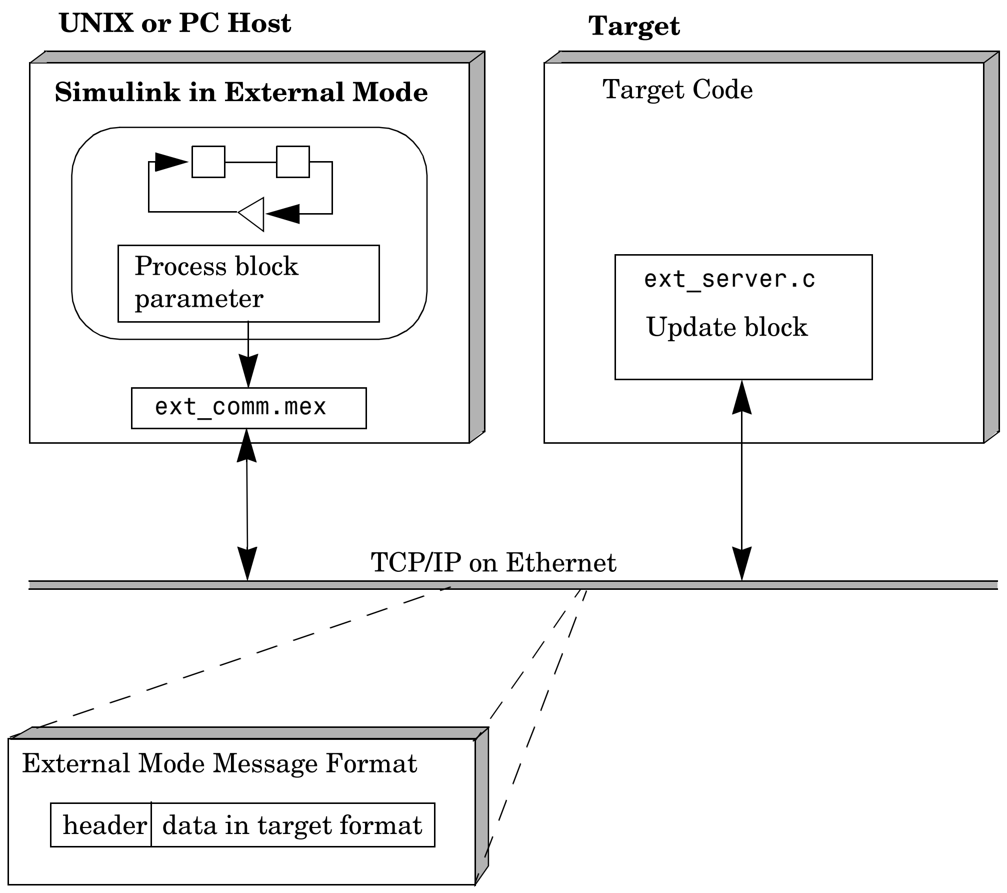
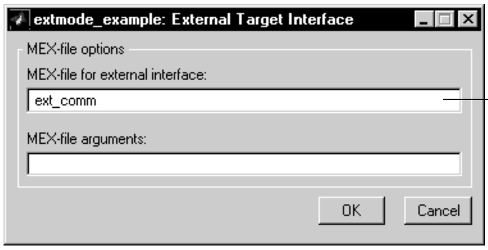


Figure 4-2: TCP-based Client/Server Implementation for External Mode

The following sections discuss the details of how to use the external mode of Simulink.

The External Interface MEX-File

You must specify the name of the external interface MEX-file in the **External Target Interface** dialog box:


extmode_example: External Target Interface
MEX-file options
MEX-file for external interface:
ext_comm
MEX-file arguments:
OK
Cancel

Enter the name of the external interface MEX-file in the box (you do not need to enter the `.mex` extension). This file must be in the current directory or in a directory that is on your MATLAB path.

You may be required to provide arguments to the MEX-file. In particular, `ext_comm` has three optional arguments: the network name of your target, the verbosity level, and the TCP port number. The implementation of the TCP-based MEX-file is called `ext_comm`, which is the default for this dialog box.

MEX-File Optional Arguments

You can specify optional arguments that are passed to the MEX-file in the **External Target Interface** dialog box. These include the network name of the target host, the verbosity level, and the TCP server port number:

1. Target network name — The network name of the computer running the external program. By default, this is the computer on which Simulink is running.

2. Verbosity level — Controls the level of detail of the information printed out during the actual data transfer. The value is either 0 or 1 and has the following meaning:

   0 — no information
   1 — detailed information

3. TCP server port number — The default value is 17725. You can change the port number to a value between 245 and 65535 to avoid a port conflict if necessary.

You must specify these options in order. For example, if you want to specify the verbosity level (the second argument), then you must also specify the target host name (the first argument).

Note that you can also specify verbosity level and port number as options in the external program. See "Running the External Program" on page 4-25 for more information.

External Mode Support

The generic real-time and Tornado targets have support for external mode. To enable external mode, check **External mode** in the **Code Generation Options** dialog box. You can open this dialog box by clicking **Options** on the Real-Time Workshop page of the **Simulation Parameters** dialog box.

Building the External Program

Click the **Build** button to build the program. You can use the MAT-file data logging options to observe the effects of changing parameters in a real-time program. Tornado applications can also use StethoScope with signal monitoring to observe the effect of changing parameters.

See Chapter 2 for an example of how to build the program and save data that you can display with MATLAB. See Chapter 8 for information on building programs for Tornado and using StethoScope.

Running the External Program

The external program must be running before you can use Simulink in external mode.

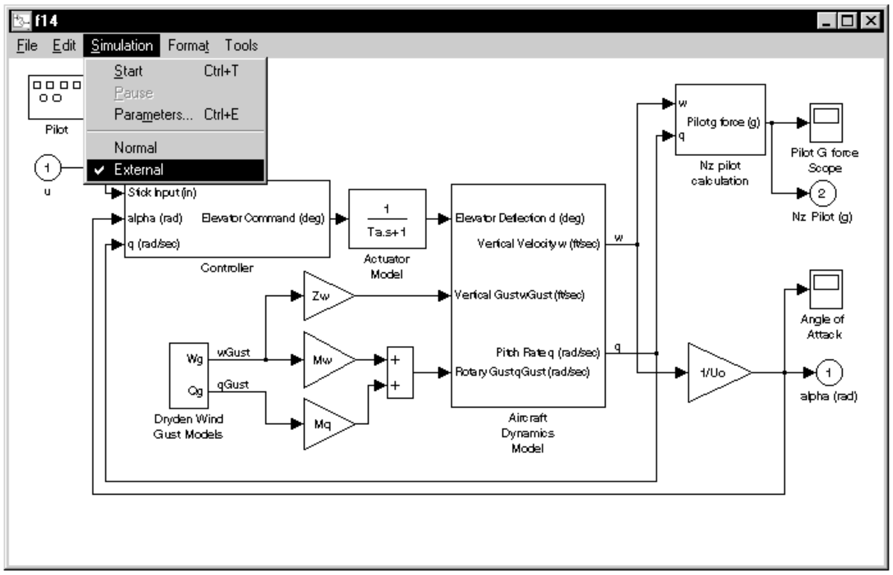
In the UNIX environment, if you start the external program from MATLAB, you must run it in the background so that you can still access Simulink. For example, the command

```
!model &
```

runs the executable file `model` from MATLAB by spawning another process to run it.

Enabling External Mode

To enable external mode, display the Simulink block diagram and select **External** from the **Simulation** menu:



Next, select **Start** from the **Simulation** menu, as you would to begin a simulation and change parameters as desired.

Possible Error Conditions

If the Simulink block diagram does not match the external program, Simulink displays an error box informing you that the checksums do not match (i.e., the model has changed since you generated code). This means you must rebuild the program from the new block diagram (or reload the correct one) in order to use external mode.

If the external program is not running, Simulink displays an error informing you that it cannot connect to the external program.

Creating an External Mode Communication Channel

The Real-Time Workshop provides support for TCP/IP through ext_comm. When creating your own custom target for a new board, you may choose to use another communications protocol.

To create your own external mode communication channel, you must:

- Create a MEX-file for the external interface
- Add a communications protocol to your target

The best way to do this is to use the TCP channel provided with the Real-Time Workshop as an example. Start by copying the source files in `matlabroot/rtw/ext_comm` and looking at the generic real-time target (`grt`) source files in `matlabroot/rtw/c/src` and `matlabroot/rtw/c/grt`.

Rebuilding the ext_comm MEX-file

The Real-Time Workshop provides compiled versions of `ext_comm`, which are created from `ext_comm.c` for all supported hosts. If you want to rebuild the MEX-file, you can regenerate it by using MATLAB's `mex` command, provided that you have an installed compiler supported by the MATLAB API. See the *MATLAB Application Program Interface Guide* for more information.

This table lists the form of the commands on PC and UNIX platforms:

Table 4-1: Commands Needed to Rebuild MEX-files

Platform	**Commands**

PC	`cd matlabroot\toolbox\rtw`
	`mex matlabroot\rtw\ext_mode\ext_comm.c`
	`    matlabroot\rtw\ext_mode\ext_convert.c`
	`    –Imatlab\rtw\c\src –DWIN32`
	`    compiler_library_path\wsock32.lib`

UNIX	`cd matlabroot/toolbox/rtw`
	`mex matlabroot/rtw/ext_mode/ext_comm.c`
	`    matlabroot\rtw\ext_mode\ext_convert.c`
	`    –Imatlab/rtw/c/src`

Data Logging and Signal Monitoring

MAT-File Data Logging

For the Real-Time Workshop targets that have access to a disk, you can use MAT-file logging to examine the results of the real-time program.

Real-Time Workshop targets can create a MAT-file that logs system states and outputs at each model execution time step. Some provide more options than others; for example, the rapid simulation target (`rsim`) provides enhanced logging configurations. See Chapter 13, “Real-Time Workshop Rapid Simulation Target,” for more information.

By using the Workspace I/O page of the **Simulation parameters** dialog box, you can select time, state, and outputs to log as well as the decimation desired. The default variable names used in the MAT-file are `rt_tout`, `rt_xout`, and `rt_yout` respectively. The Real-Time Workshop logs outputs for:

- All root Outport blocks (`rt_yout`)
- All Scopes that have `save data to workspace` selected
- All To Workspace blocks in the model

For Scope and To Workspace blocks, you must specify variable names in each block’s dialog box. You can log states for all continuous and discrete states in the model. The sort order of the `rt_yout` array is based on the port number of the Outport block, starting with 1. You can determine the sort order of the `rt_xout` array by using this MATLAB command

```
[a,b,c,d] = model([],[],[],0)
```

and inspecting the variable `c`. The filename defaults to *`model`*`.mat` but can be changed by specifying `OPTS="-DSAVEFILE=filename"` in the **Make command** field on the Real-Time Workshop page of the **Simulation parameters** dialog box.

To File Block MAT-Files

Aside from the Workspace I/O MAT-file logging described above, MAT-file logging can be done using the To File block. A separate MAT-file containing time and input variable(s) is created for every To File block in the model. You must specify the filename, variable name, decimation, and sample time in the To File block’s dialog box. The To File block cannot be used in DOS real-time targets because of limitations of the DOS target.

Singletasking versus Multitasking MAT-File Logging

When logging data you will notice differences in the logging of:

- Noncontinuous root outports
- Discrete states

in singletasking vs. multitasking environments. In multitasking mode, the logging of states and outputs is done after the first task execution (and not at the end of the first time step). In singletasking mode, the Real-Time Workshop logs states and outputs after the first time step. See Chapter 7, “Models with Multiple Sample Rates,” for more details on the differences between single- and multitasking data logging.

C API for Signal Monitoring

Signal monitoring provides a second method for accessing block outputs in an externally running program. All block output data is written to the `SimStruct` with each time step in the model code. However, to access the output of any given block in the `SimStruct`, you must know the address of the `BlockIO` vector where the data is stored, how many output ports the block has, and the width of each output. All of this information is contained in the `BlockIOSignals` data structure. This is the mechanism that the StethoScope Graphical Monitoring/ Data Analysis Tool uses to collect signal information in Tornado targets. See "Targeting Tornado for Real-Time Applications" in Chapter 8 for more information on using StethoScope.

The `BlockIOSignals` data structure is created during code generation by appending

```
-aBlockIOSignals=1
```

to the system target file field on the Real-Time Workshop page of the **Simulation Parameters** dialog box. Otherwise, `BlockIOSignals` is not created. If included, the TLC file *`matlabroot`*`/rtw/c/tlc/biosig.tlc` creates the file *`model`*`.bio` that contains the information on all block outputs in the model in an array of structures. This array can then be used to monitor the value of any output while the program is running. The structure definition for the elements of the array is in *`matlabroot`*`/rtw/c/src/bio_sig.h`. This file is included by *`model`*`.bio` and should also be included by any source file that uses the array (for example, `rt_main.c`). Note that, depending on the size of your model, the `BlockIOSignals` array can consume a considerable amount of memory.

Using BlockIOSignals to Obtain Block Outputs

The `BlockIOSignals` data structure is declared as follows:

```
typedef struct BlockIOSignals_tag {
  char_T *blockName;  /* Block's full pathname
                         (mangled by the Real-Time Workshop) */
  char_T *signalName; /* Signal label (unmangled) */
  uint_T portNumber;  /* Block output port number (start at 0) */
  uint_T signalWidth; /* Signal's width */
  void *signalAddr;   /* Signal's address in the blockIO vector */
  char_T *dtName;     /* The C language data type name */
  uint_T dtSize;      /* The size (# of bytes) for the data type*/
} BlockIOSignals;
```

The model code file *model*.bio defines an array of BlockIOSignals structures, for example:

```
#include "bio_sig.h"
/* Block output signal information */
const BlockIOSignals rtBIOSignals[] =
  {
  /* blockName,
     signalName, portNumber, signalWidth, signalAddr,
     dtName, dtSize */
  {
    "simple/Constant",
    NULL, 0, 1, &rtB.Constant,
    "double", sizeof(real_T)
  },
  {
    "simple/Constant1",
    NULL, 0, 1, &rtB.Constant1,
    "double", sizeof(real_T)
  },
  {
    "simple/Gain",
    "accel", 0, 2, &rtB.accel[0],
    "double", sizeof(real_T)
  },
  {
    NULL, NULL, 0, 0, 0, NULL, 0
  }
};
```

Each structure element describes one output port for a block. Thus, a given block will have as many entries as it has output ports. In the above example, the `simple/Gain` structure has a signal named `accel` on block output port 0. The width of the signal is 2.

The array is accessed via the name `rtBIOSignals` by whatever code would like to use it. To avoid overstepping array bounds, you can do either of the following:

- Use the `SimStruct` access macro `ssGetNumBlockIO` to determine the number of elements in the array
- Use the fact that last element has a `blockName` of `NULL`.

You must then write code that walks through the `rtBIOSignals` array and chooses the signals to be monitored based on the `blockName` and `signalName` or `portNumber`. How the signals are monitored is up to you. For example, you could collect the signals at every time step or just sample them asynchronously by a separate, lower priority task.

For example, the Tornado source file, `rt_main.c`, defines the following function that selectively installs signals from the `BlockIOSignals` array into the StethoScope Graphical Monitoring/Data Analysis Tool by calling `ScopeInstallSignal`. The signals are then collected in the main simulation task by calling `ScopeCollectSignals`.

The code below is an example routine that installs signals from the `BlockIOSignals` array:

```
static int_T rtInstallRemoveSignals(SimStruct *S,
char_T *installStr, int_T fullNames, int_T install)
{
  uint_T                  i, w;
  char_T                 *blockName;
  char_T                 name[1024];
  extern BlockIOSignals  rtBIOSignals[];
  int_T                  ret = -1;

  if (installStr == NULL) {
    return -1;
  }

  i = 0;
  while(rtBIOSignals[i].blockName != NULL) {
    BlockIOSignals *blockInfo = &rtBIOSignals[i++];

    if (fullNames) {
      blockName = blockInfo->blockName;
    } else {
      blockName = strrchr(blockInfo->blockName, '/');
      if (blockName == NULL) {
        blockName = blockInfo->blockName;
      } else {
        blockName++;
      }
    }

    if ((*installStr) == '*') {
    } else if (strcmp("[A-Z]*", installStr) == 0) {
      if (!isupper(*blockName)) {
        continue;
      }
    } else {
      if (strncmp(blockName, installStr, strlen(installStr)) != 0) {
```

```
    continue;
      }
    }
    /*install/remove the signals*/
    for (w = 0; w < blockInfo->signalWidth; w++) {
      sprintf(name, "%s_%d_%s_%d", blockName, blockInfo->portNumber,
          (blockInfo->signalName==NULL)?"":blockInfo->signalName, w);
      if (install) { /*install*/
        if (!ScopeInstallSignal(name, "units",
                               (void *)((int)blockInfo->signalAddr +
                                           w*blockInfo->dtSize),
                                  blockInfo->dtName, 0)) {
              fprintf(stderr,"rtInstallRemoveSignals: ScopeInstallSignal "
                      "possible error: over 256 signals.\n");
              return -1;
          } else {
              ret =0;
          }
      } else { /*remove*/
        if (!ScopeRemoveSignal(name, 0)) {
          ifprintf(stderr,"rtInstallRemoveSignals: ScopeRemoveSignal\n"
                          "%s not found.\n",name);
            return -1;
          } else {
            ret =0;
        }
      }
    }
  }
  return ret;
}
```

Below is an excerpt from an example routine that collects signals taken from the main simulation loop:

```
/*******************************************
 * Step the model for the base sample time *
 *******************************************/
MdlOutputs(FIRST_TID);

#ifdef MAT_FILE
  if (rt_UpdateTXYLogVars(S) != NULL) {
    fprintf(stderr,"rt_UpdateTXYLogVars() failed\n");
    return(1);
  }
#endif

#ifdef STETHOSCOPE
  ScopeCollectSignals(0);
#endif

MdlUpdate(FIRST_TID);
<code continues ...>
```

The Real-Time Workshop provides a mechanism that allows your application program to monitor block outputs during program execution.

All block output data is written to the `SimStruct` at each step through the model code. However, to access the output of any given block in the `SimStruct`, you must know the index into the `BlockIO` vector where the data is stored, as well as how many output ports the block has and the width of each output. All of this information is contained in the `ModelBlockInfo` data structure.

The `ModelBlockInfo` data structure is created by the generated code only if you define the flag `USE_MDLBLOCKINFO` when you build your program. Otherwise, `ModelBlockInfo` is not created.

Note that, depending on the size of your model, `ModelBlockInfo` can consume a considerable amount of memory.

Program Architecture

Introduction

The Real-Time Workshop generates two styles of code. One code style is suitable for rapid prototyping (and simulation via code generation). The other style is suitable for embedded applications. This chapter discusses the program architecture, that is, the structure of the Real-Time Workshop generated code, associated with these two styles of code. The table below classifies the targets shipped with the Real-Time Workshop.

Table 6-1: Code Styles Listed By Target

Target	Code Style (using C unless noted)
DOS Real-Time Target	Rapid prototyping — runs model in real-time at ISR level under DOS.
Embedded-C Real-Time Target	Embedded — useful as a starting point when using the generated C code in an embedded application.
Generic Real-Time Target	Rapid prototyping — nonreal-time simulation on your workstation. Useful as a starting point for creating a rapid prototyping real-time target that does not use real-time operating system tasking primitives. Also useful for validating the generated code on your workstation.
Generic Real-Time Target with dynamic memory allocation	Rapid prototyping — very similar to the generic real-time target except that this target declares all “model” working memory using dynamically allocated memory as opposed to statically declaring it in advance.
Rapid Simulation Target	Rapid prototyping — nonreal-time simulation of your model on your workstation. Useful as a high-speed or batch simulation tool.

Table 6-1: Code Styles Listed By Target

Target	Code Style (using C unless noted)
S-function Target	Rapid prototyping — creates a C-MEX S-function for simulation of your model within another Simulink model.
Tornado (VxWorks) Real-Time Target	Rapid prototyping — runs model in real-time using the VxWorks real-time operating system tasking primitives. Also useful as a starting point for targeting a real-time operating system.
Real-Time Windows Target	Rapid prototyping — runs model in real-time at ISR level while your PC is running Microsoft Windows in the background.
Ada Simulation Target	Embedded — nonreal-time simulation on your workstation using Ada. Useful for validating the generated code on your workstation.
Ada Multitasking Real-Time Target	Embedded — uses Ada tasking primitives to run your model in real-time. Useful as a starting point when using the generated Ada code in an embedded application.

Third-party vendors supply additional targets for the Real-Time Workshop. Generally, these can be classified as rapid prototyping targets. Consult The MathWorks Web site (`http:/mathworks.com`) or our *Connections Catalog* for more information about third-party products.

You can identify the rapid prototyping style of generated code by using the `SimStruct` data structure (i.e., `#include "simstruc.h"`). In contrast, the embedded code style does not have a `SimStruct`.

This chapter is divided into three sections. The first section discusses model execution; the second section discusses the rapid prototyping style of code; and the third section discusses the embedded style of code.

Model Execution

Before looking at the two styles of generated code, you need to have a high-level understanding of how the generated model code is executed. The Real-Time Workshop generates algorithmic code as defined by your model. You may include your own code into your model via S-functions. S-functions can range from high-level signal manipulation algorithms to low level device drivers.

The Real-Time Workshop also provides a run-time interface that executes the generated model code. The run-time interface and model code are compiled together to create the model executable. The diagram below shows a high-level object-oriented view of the executable.

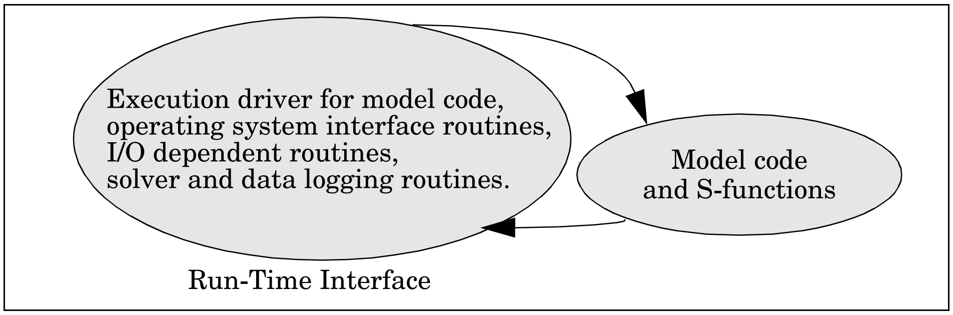

Execution driver for model code, operating system interface routines, I/O dependent routines, solver and data logging routines.
Model code and S-functions
Run-Time Interface

Figure 6-1: The Object-Oriented View of a Real-time Program

In general, the model execution driver does not change in concept between the rapid prototyping and embedded style of generated code. The following sections describe model execution for singletasking and multitasking environments both for simulation (nonreal-time) and for real-time. For most models, the multitasking environment will provide the most efficient model execution (i.e., fastest sample rate). See Chapter 7, "Models with Multiple Sample Rates," for more information on singletasking and multitasking execution.

The following concepts are useful in describing how models execute:

- Initialization — Initializing the run-time interface code and the model code.
- ModelOutputs — Calling all blocks in your model that have a time hit at the current point in time and having them produce their output. ModelOutputs can be done in major or minor time steps. In major time steps, the output is

a given simulation time step. In minor time steps, the run-time interface integrates the derivatives to update the continuous states.

- ModelUpdate — Calling all blocks in your model that have a sample hit at the current point in time and having them update their discrete states or similar type objects.
- ModelDerivatives — Calling all blocks in your model that have continuous states and having them update their derivatives. ModelDerivatives is only called in minor time steps.

The pseudocode below shows the execution of a model for a singletasking simulation (nonreal-time).

```
main()
{
  Initialization
  While (time < final time)
    ModelOutputs     -- Major time step.
    LogTXY           -- Log time, states and root outports.
    ModelUpdate      -- Major time step.
    Integrate:       -- Integration in minor time step for models
                     -- with continuous states.
      ModelDerivatives
      Do 0 or more:
        ModelOutputs
        ModelDerivatives
      EndDo (Number of iterations depends upon the solver.)
      Integrate derivatives to update continuous states.
    EndIntegrate
  EndWhile
  Shutdown
}
```

The initialization phase begins first. This consists of initializing model states and setting up the execution engine. The model then executes, one step at a time. First ModelOutputs executes at time t, then the workspace I/O data is logged, and then ModelUpdate updates the discrete states. Next, if your model has any continuous states, ModelDerivatives integrates the continuous states' derivatives to generate the states for time $t_{new} = t + h$, where h is the step size. Time then moves forward to t_{new} and the process repeats.

During the ModelOutputs and ModelUpdate phases of model execution, only blocks that have hit the current point in time execute. They determine if they have hit by using a macro (`ssIsSampleHit`, or `ssIsSpecialSampleHit`) that checks for a sample hit.

The pseudocode below shows the execution of a model for a multitasking simulation (nonreal-time).

```
main()
{
  Initialization
  While (time < final time)
    ModelOutputs(tid=0)   -- Major time step.
    LogTXY                -- Log time, states, and root outports.
    ModelUpdate(tid=1)    -- Major time step.
    For i=1:NumTids
      ModelOutputs(tid=i) -- Major time step.
      ModelUpdate(tid=i)  -- Major time step.
    EndFor
    Integrate       -- Integration in minor time step for models
                    -- with continuous states.
      ModelDerivatives
      Do 0 or more:
        ModelOutputs(tid=0)
        ModelDerivatives
      EndDo (Number of iterations depends upon the solver.)
      Integrate derivatives to update continuous states.
    EndIntegrate
  EndWhile
  Shutdown
  }
```

The multitasking operation is more complex when compared with the singletasking execution because the output and update functions are subdivided by the task identifier that is passed into these functions. This allows for multiple invocations of these functions with different task identifiers using overlapped interrupts, or for multiple tasks when using a real-time operating system. In simulation, multiple tasks are emulated by executing the code in the order that would occur if there were no preemption in a real-time system.

Note that the multitasking execution assumes that all tasks are multiples of the base rate. Simulink enforces this when you have created a fixed-step multitasking model.

The multitasking execution loop is very similar to that of singletasking, except for the use of the task identifier (tid) argument to ModelOutputs and ModelUpdate. The `ssIsSampleHit` or `ssIsSpecialSampleHit` macros use the tid to determine when blocks have a hit. For example, ModelOutputs (tid=5) will execute only the blocks that have a sample time corresponding to task identifier 5.

The pseudocode below shows the execution of a model in a real-time singletasking system where the model is run at interrupt level.

```
rtOneStep()
{
  Check for interrupt overflow
  Enable "rtOneStep" interrupt
  ModelOutputs    -- Major time step.
  LogTXY          -- Log time, states and root outports.
  ModelUpdate     -- Major time step.
  Integrate       -- Integration in minor time step for models
                  -- with continuous states.
     ModelDerivatives
     Do 0 or more
       ModelOutputs
       ModelDerivatives
     EndDo (Number of iterations depends upon the solver.)
     Integrate derivatives to update continuous states.
  EndIntegrate
}

main()
{
  Initialization (including installation of rtOneStep as an
    interrupt service routine, ISR, for a real-time clock).
  While(time < final time)
    Background task.
  EndWhile
  Mask interrupts (Disable rtOneStep from executing.)
  Complete any background tasks.
  Shutdown
}
```

Real-time singletasking execution is very similar to the nonreal-time single tasking execution, except that the execution of the model code is done at interrupt level.

At the interval specified by the program's base sample rate, the interrupt service routine (ISR) preempts the background task to execute the model code. The base sample rate is the fastest rate in the model. If the model has continuous blocks, then the integration step size determines the base sample rate.

For example, if the model code is a controller operating at 100 Hz, then every 0.01 seconds the background task is interrupted. During this interrupt, the controller reads its inputs from the analog-to-digital converter (ADC), calculates its outputs, writes these outputs to the digital-to-analog converter (DAC), and updates its states. Program control then returns to the background task. All of these steps must occur before the next interrupt.

This code shows how a model executes in a real-time multitasking system (where the model is run at interrupt level).

```
rtOneStep()
{
  Check for interrupt overflow
  Enable "rtOneStep" interrupt
  ModelOutputs(tid=0)     -- Major time step.
  LogTXY                  -- Log time, states and root outports.
  ModelUpdate(tid=0)      -- Major time step.
  Integrate               -- Integration in minor time step for
                          -- models with continuous states.
     ModelDerivatives
     Do 0 or more:
       ModelOutputs(tid=0)
       ModelDerivatives
     EndDo (Number of iterations depends upon the solver.)
     Integrate derivatives and update continuous states.
  EndIntegrate
  For i=1:NumTasks
    If (hit in task i)
      ModelOutputs(tid=i)
      ModelUpdate(tid=i)
    EndIf
  EndFor
}

main()
{
  Initialization (including installation of rtOneStep as an
    interrupt service routine, ISR, for a real-time clock).
  While(time < final time)
    Background task.
  EndWhile
  Mask interrupts (Disable rtOneStep from executing.)
  Complete any background tasks.
  Shutdown
}
```

Running models at interrupt level in real-time multitasking environment is very similar to the previous singletasking environment, except that overlapped interrupts are employed for concurrent execution of the tasks.

The execution of a model in a singletasking or multitasking environment when using real-time operating system tasking primitives is very similar to the interrupt-level examples discussed above. The pseudocode below is for a singletasking model using real-time tasking primitives.

```
tSingleRate()
{
  MainLoop:
    If clockSem already "given", then error out due to overflow.
    Wait on clockSem
    ModelOutputs            -- Major time step.
    LogTXY                  -- Log time, states and root outports
    ModelUpdate             -- Major time step
    Integrate               -- Integration in minor time step for
                            -- models with continuous states.
      ModelDeriviatives
      Do 0 or more:
        ModelOutputs
        ModelDerivatives
      EndDo (Number of iterations depends upon the solver.)
      Integrate derivatives to update continuous states.
    EndIntegrate
  EndMainLoop
}

main()
{
  Initialization
  Start/spawn task "tSingleRate".
  Start clock that does a "semGive" on a clockSem semaphore.
  Wait on "model-running" semaphore.
  Shutdown
}
```

In this singletasking environment, the model is executed using real-time operating system tasking primitives. In this environment, we create a single task (tSingleRate) to run the model code. This task is invoked when a clock tick occurs. The clock tick gives a clockSem (clock semaphore) to the model task (tSingleRate). The model task will wait for the semaphore before executing. The clock ticks are configured to occur at the fundamental step size (base rate) for your model.

The pseudocode below is for a multitasking model using real-time tasking primitives.

```
tSubRate(subTaskSem,i)
{
  Loop:
    Wait on semaphore subTaskSem.
    ModelOutputs(tid=i)
    ModelUpdate(tid=i)
  EndLoop
}

tBaseRate()
{
  MainLoop:
    If clockSem already "given", then error out due to overflow.
    Wait on clockSem
    For i=1:NumTasks
      If (hit in task i)
        If task i is currently executing, then error out due to
          overflow.
        Do a "semGive" on subTaskSem for task i.
      EndIf
    EndFor
    ModelOutputs(tid=0)    -- major time step.
    LogTXY                 -- Log time, states and root outports.
    ModelUpdate(tid=0)     -- major time step.
    Loop:                  -- Integration in minor time step for
                           -- models with continuous states.
      ModelDeriviatives
      Do 0 or more:
        ModelOutputs(tid=0)
        ModelDerivatives
      EndDo (number of iterations depends upon the solver).
      Integrate derivatives to update continuous states.
    EndLoop
    EndMainLoop
  }

  main()
  {
    Initialization
    Start/spawn task "tSingleRate".
    Start clock that does a "semGive" on a clockSem semaphore.
    Wait on "model-running" semaphore.
    Shutdown
  }
```

In this multitasking environment, the model is executed using real-time operating system tasking primitives. In this environment, it is necessary to create several model tasks (tBaseRate and several tSubRate tasks) to run the model code. The base rate task (tBaseRate) has a higher priority than the subrate tasks. The subrate task for tid=1 has a higher priority than the subrate task for tid=2, and so on. The base rate task is invoked when a clock tick occurs. The clock tick gives a clockSem to tBaseRate. The first thing tBaseRate does is give semaphores to the subtasks that have a hit at the current point in time. Since the base rate task has a higher priority, it continues to execute. Next it executes the fastest task (tid=0) consisting of blocks in your model that have the fastest sample time. After this execution, it resumes waiting for the clock semaphore. The clock ticks are configured to occur at executing at the fundamental step size for your model.

Program Timing

Real-time programs require careful timing of the task invocations (either via an interrupt or a real-time operating system tasking primitive) to ensure that the model code executes to completion before another task invocation occurs. This includes time to read and write data to and from external hardware.

The following diagram illustrates interrupt timing.

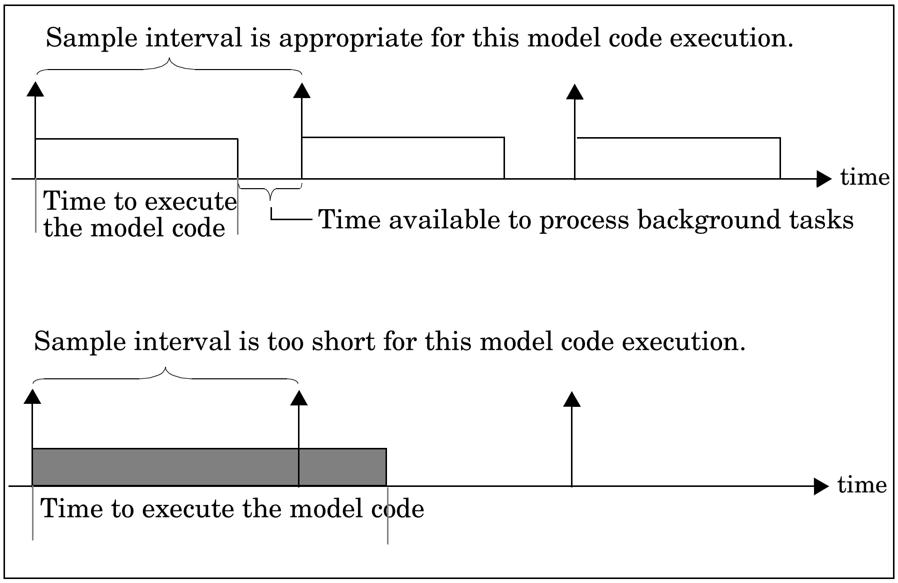

Sample interval is appropriate for this model code execution.
time
Time to execute the model code
Time available to process background tasks
Sample interval is too short for this model code execution.
time
Time to execute the model code

Figure 6-2: Task Timing

The sample interval must be long enough to allow model code execution between task invocations.

In the figure above, the time between two adjacent vertical arrows is the sample interval. The empty boxes in the upper diagram show an example of a program that can complete one step within the interval and still allow time for the background task. The gray box in the lower diagram indicates what happens if the sample interval is too short. Another task invocation occurs before the task is complete. Such timing results in an execution error.

Note also that, if the real-time program is designed to run forever (i.e., the final time is 0 or infinite so the while loop never exits), then the shutdown code never executes.

Program Execution

As the previous section indicates, a real-time program may not require 100% of the CPU's time. This provides an opportunity to run background tasks during the free time.

Background tasks include operations like writing data to a buffer or file, allowing access to program data by third-party data monitoring tools, or using Simulink external mode to update program parameters.

It is important, however, that the program be able to preempt the background task at the appropriate time to ensure real-time execution of the model code.

The way the program manages tasks depends on capabilities of the environment in which it operates.

External Mode Communication

External mode allows communication between the Simulink block diagram and the stand-alone program that is built from the generated code. In this mode, the real-time program functions as an interprocess communication server, responding to requests from Simulink. See Chapter 4, "External Mode," for information on external mode.

Data Logging

You can use the built-in data logging capabilities provided by the Real-Time Workshop or you can use third-party data logging tools with your program (see Chapter 5, "Data Logging and Signal Monitoring," for more information).

The MathWorks also provides data logging facilities for creating a `model.mat` file at the completion of the model execution. This data logging facility is useful for code validation and high-speed simulations. See the generic real-time target, `grt`, described in "Building Generic Real-Time Programs" on page 2-12.

Logging Differences Between Singletasking and Multitasking Model Execution

If you examine how LogTXY is called in the singletasking and multitasking environments, you will notice that for singletasking LogTXY is called after ModelOutputs. During this ModelOutputs call, all blocks that have a hit at time *t* are executed, whereas in multitasking, LogTXY is called after ModelOutputs(tid=0) that executes only the blocks that have a hit at time *t* and that have a task identifier of 0. This results in differences in the logged values between singletasking and multitasking logging. Specifically, consider a model with two sample times, the faster sample time having a period of 1.0 second and the slower sample time having a period of 10.0 seconds. At time t = k*10, k=0,1,2... both the fast (tid=0) and slow (tid=1) blocks have a hit. When executing in multitasking mode, when LogTXY is called, the slow blocks will have a hit, but the previous value will be logged, whereas in singletasking the current value will be logged.

Another difference occurs when logging data in an enabled subsystem. Consider an enabled subsystem that has a slow signal driving the enable port and fast blocks within the enabled subsystem. In this case, the evaluation of the enable signal occurs in a slow task and the fast blocks will see a delay of one sample period, thus the logged values will show these differences.

To summarize differences in logged data between singletasking and multitasking, differences will be seen when:

- Any root outport block that has a sample time that is slower than the fastest sample time
- Any block with states that has a sample time that is slower than the fastest sample time
- Any block in an enabled subsystem where the signal driving the enable port is slower than the rate of the blocks in the enabled subsystem

For the first two cases, even though the logged values are different between singletasking and multitasking, the model results are not different. The only real differences is where (at what point in time) the logging is done. The third (enabled subsystem) case results in a delay that can be seen in a real-time environment.

Differences Between Rapid Prototyping and Embedded Model Execution

The rapid prototyping program framework provides a common application programming interface (API) that does not change between model definitions. The embedded program framework, in contrast, provides a optimized API that is tailored to your model. It is intended that when you use the embedded style of generated code that you are modeling how you would like your code to execute in your embedded system. Therefore, the definitions defined in your model should be specific to your embedded targets. Items such as the model name, parameter and signal storage class are included as part of the API for the embedded style of code.

The single largest difference between the rapid prototyping and embedded style of generated code is that the embedded code does not contain the `SimStruct` data structure. The `SimStruct` defines a common API that the rapid prototyping style of generated code relies heavily on. However, the `SimStruct` data structure supports many options and therefore consumes memory that is not needed in an embedded application.

The next major difference between the rapid prototyping and embedded style of generated code is that the latter contains fewer entry-point functions. The embedded style of code can be configured to have only one run-time function *`model`*`_step`. You can define a single run-time function because the embedded target:

- Can only be used with models that do not have continuous sample time (and therefore no continuous states)
- Requires that all S-functions must be inlined with the Target Language Compiler, which means that they do not access the `SimStruct` data structure

Thus, when looking at the model execution pseudo-code presented earlier in this chapter, you can eliminate the `Loop...EndLoop` statements, and group the ModelOutputs, LogTXY, and ModelUpdate into a single statement, *`model`*`_step`.

Rapid Prototyping Model Functions

The rapid prototyping code defines the following functions that interface with the run-time interface:

- `Model()` — The model registration function. This is responsible for initializing the work areas (e.g. allocating and setting pointers to various data structures) needed by the model. The model registration function calls the `MdlInitializeSizes` and `MdlInitializeSampleTimes` functions. These two functions are very similar to the S-function `mdlInitializeSizes` and `mdlInitializeSampleTimes` methods.
- `MdlStart(void)` — After the model registration functions, `MdlInitializeSizes` and `MdlInitializeSampleTimes` (located in

model.reg), execute, the run-time interface starts execution by calling MdlStart. This routine is called once at start-up.

The function MdlStart has four basic sections:

- Code to initialize the states for each block in the root model that has states. A subroutine call is made to the "initialize states" routine of conditionally executed subsystems.
- Code generated by the one-time initialization (start) function for each block in the model.
- Code to enable: 1) the blocks in the root model that have enable methods, 2) the blocks inside triggered or function-call subsystems residing in the root model. Simulink blocks can have enable and disable methods. An enable method is called just before a block starts executing, and the disable method is called just after the block stops executing.
- Code for each block in the model that has a constant sample time.

• MdlOutputs(int_T tid) — MdlOutput is responsible for updating the output of blocks at appropriate times. The tid (task identifier) parameter identifies the task that in turn maps when to execute blocks based upon their sample time. This routine is invoked by the run-time interface during major and minor time steps. The major time steps are when the run-time interface is taking an actual time step (i.e., it is time to execute a specific task). If your model contains continuous states, the minor time steps will be taken. The minor time steps are when the solver is generating integration stages, which are points between major outputs. These integration stages are used to compute the derivatives used in advancing the continuous states.

• MdlUpdate(int_T tid) — MdlUpdate is responsible for updating the discrete states and work vector state information (i.e., states that are neither continuous nor discrete) saved in work vectors. The tid (task identifier) parameter identifies the task that in turn indicates which sample times are active allowing you to conditionally update states of only active blocks. This routine is invoked by the run-time interface after the major MdlOutput has been executed.

• MdlDerivatives(void) — MdlDerivatives is responsible for returning the block derivatives. This routine is called in minor steps by the solver during its integration stages. All blocks that have continuous states have an identical number of derivatives. These blocks are required to compute the derivatives so that the solvers can integrate the states.

- `MdlTerminate(void)` — `MdlTerminate` contains any block shutdown code. `MdlTerminate` is called by the run-time interface, as part of the termination of the real-time program.

The contents of the above functions are directly related to the blocks in your model. A Simulink block can be generalized to the following set of equations:

$$y = f_0(t, x_c, x_d, u)$$

Output, y, is a function of continuous state, x_c, discrete state, x_d, and input, u. Each block writes its specific equation in the appropriate section of `MdlOutput`.

$$x_{d+1} = f_u(t, x_d, u)$$

The discrete states, x_d, are a function of the current state and input. Each block that has a discrete state updates its state in `MdlUpdate`.

$$\dot{x} = f_d(t, x_c, u)$$

The derivatives, x, are a function of the current input. Each block that has continuous states provides its derivatives to the solver (e.g., `ode5`) in `MdlDerivatives`. The derivatives are used by the solver to integrate the continuous state to produce the next value.

The output, y, is generally written to the block I/O structure. Root-level Outport blocks write to the external outputs structure. The continuous and discrete states are stored in the states structure. The input, u, can originate from another block's output, which is located in the block I/O structure, an external input (located in the external inputs structure), or a state. These structures are defined in the *`model`*`.h` file that the Real-Time Workshop generates.

The general content of the rapid prototyping style of code for C is shown in the following figure.

```
/*
 * Version, Model options, TLC options,
 * and code generation information are placed here.
 */

<includes>

void MdlStart(void)
{
  /*
   * State initialization code.
   * Model start-up code - one time initialization code.
   * Execute any block enable methods.
   * Initialize output of any blocks with constant sample times.
   */
}

void MdlOutputs(int_T tid)
{
  /* Compute: y = f0(t,xc,xd,u) for each block as needed. */
}

void MdlUpdate(int_T tid)
{
  /* Compute: xd+1 = fu(t,xd,u) for each block as needed. */
}

void MdlDerivatives(void)
{
  /* Compute: dxc = fd(t,xc,u) for each block as needed. */
}

void MdlTerminate(void)
{
  /* Perform shutdown code for any blocks that
     have a termination action */
}

/* Model registration */
#include "model.reg"
```

Figure 6-3: Content of Model.c for the Rapid Prototyping Code Style

A flow chart describing the execution of the rapid prototyping generated code is shown below.

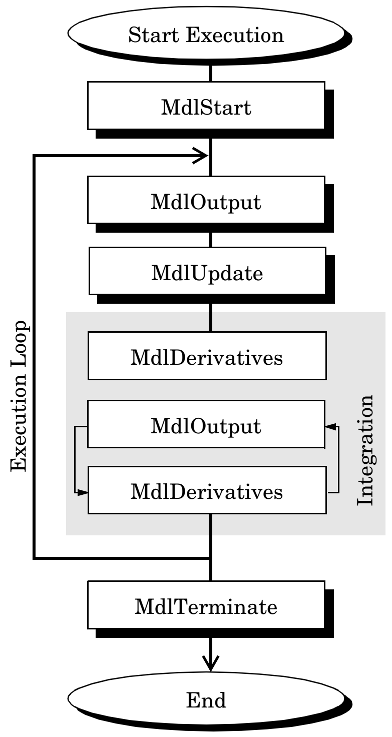


Figure 6-4: Rapid Prototyping Execution Flow Chart

Each block places code into specific `Mdl` routines according to the algorithm that it is implementing. Blocks have input, output, parameters, states, plus other general items. For example, in general, block inputs and outputs are written to a block I/O structure (`rtB`). Block inputs can also come from the external input structure (`rtU`) or the state structure when connected to a state port of an integrator (`rtX`), or ground (`rtGround`) if unconnected or grounded.

Block outputs can also go to the external output structure (`rtY`). The following figure shows the general mapping between these items.

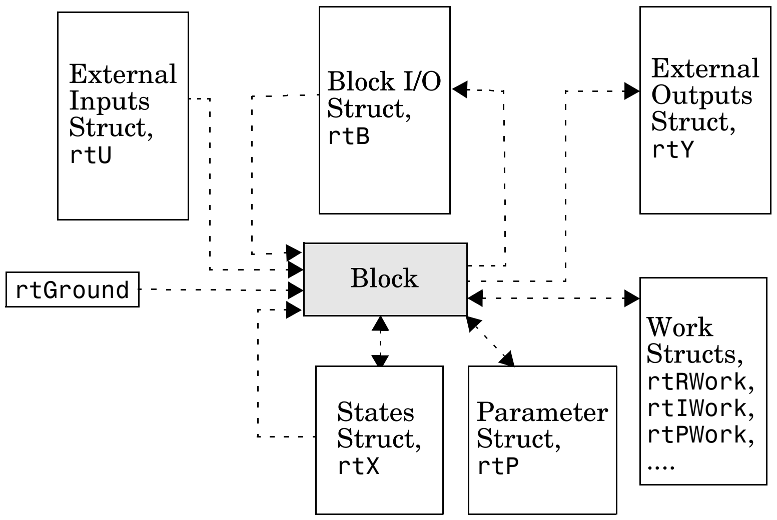

External Inputs Struct, rtU
Block I/O Struct, rtB
External Outputs Struct, rtY
rtGround
Block
Work Structs, rtRWork, rtIWork, rtPWork, ...
States Struct, rtX
Parameter Struct, rtP

Figure 6-5: Data View of the Generated Code

Structure definitions:

- Block I/O Structure (`rtB`) — This structure consists of all block output signals. The number of block output signals is the sum of the widths of the data output ports of all nonvirtual blocks in your model. If you activate block I/O optimizations by unchecking the **Disable block I/O optimizations** check box on the Diagnostics page of the **Simulation Parameters** dialog box, Simulink and the Real-Time Workshop reduce the size of the `rtB` structure by:
  - Reusing the entries in the `rtB` structure
  - Making other entries local variables

  Structure field names are determined by either the block's output signal name (when present) or by the block name and port number when the output signal is left unlabeled.
- Block States Structure (`rtX`) — The states structure contains the continuous and discrete state information for any blocks in your model that have states.

The states structure has two sections: the first is for the continuous states; the second is for the discrete states.

- Block Parameters Structure (`rtP`) — The parameters structure contains all block parameters that can be changed during execution (e.g., the parameter of a Gain block).
- External Inputs Structure (`rtU`) —The external inputs structure consists of all root-level Inport block signals. Field names are determined by either the block's output signal name, when present, or by the Inport block's name when the output signal is left unlabeled.
- External Outputs Structure (`rtY`) —The external outputs structure consists of all root-level Outport blocks. Field names are determined by the root-level Outport block names in your model.
- Real Work, Integer Work, and Pointer Work Structures (`rtRWork`, `rtIWork`, `rtPWork`) — Blocks may have a need for real, integer, or pointer work areas. For example, the Memory block uses a real work element for each signal. These areas are used to save internal states or similar information.

Embedded Model Functions

The embedded-C and Ada code targets generate the following functions:

- *`model`*`_intialize` — Performs all model initialization and should be called once before you start executing your model.
- If the Real-Time Workshop option "Single output/update function" is selected, then you will see:
  - *`model`*`_step(int_T tid)` — Contains the "output" and "update" code for all blocks in your model.

  Otherwise you will see:
  - *`model`*`_output(int_T tid)` — Contains the "output" code for all blocks in your model.
  - *`model`*`_update(int_T tid)` — This contains the "update" code for all blocks in your model.
- If the Real-Time Workshop option "Terminate function required" is selected, then you will see:
  - *`model`*_terminate — This contains all model "shutdown" code and should be called as part of system shutdown.

When compared with the rapid prototyping environment, these functions are very similar.

Rapid Prototyping Program Framework

Generating code for a Simulink model results in at least four files — *`model`*`.c`, *`model`*`.h`, *`model`*`.prm`, and *`model`*`.reg`, where *`model`* is the name of the Simulink model. This code implements the model's system equations, contains block parameters, and performs initialization.

The Real-Time Workshop's program framework provides the additional source code necessary to build the model code into a complete, stand-alone program. The program framework consists of *application modules* (files containing source code to implement required functions) designed for a number of different programming environments.

The automatic program builder ensures the program is created with the proper modules once you have configured your template makefile.

The application modules and the code generated for a Simulink model are implemented using a common API (application program interface). This API defines a data structure (called a `SimStruct`) that encapsulates all data for your model. Note that the embedded-C target does not have a `SimStruct`, but does have a common calling syntax for model execution.

This API is similar to that of S-functions, with one major exception: the API assumes that there is only one instance of the model, whereas S-functions can have multiple instances. The function prototypes also differ from S-functions.

Rapid Prototyping Program Architecture

The structure of a real-time program consists of three *components*. Each component has a dependency on a different part of the environment in which the program executes. The following diagram illustrates this structure.

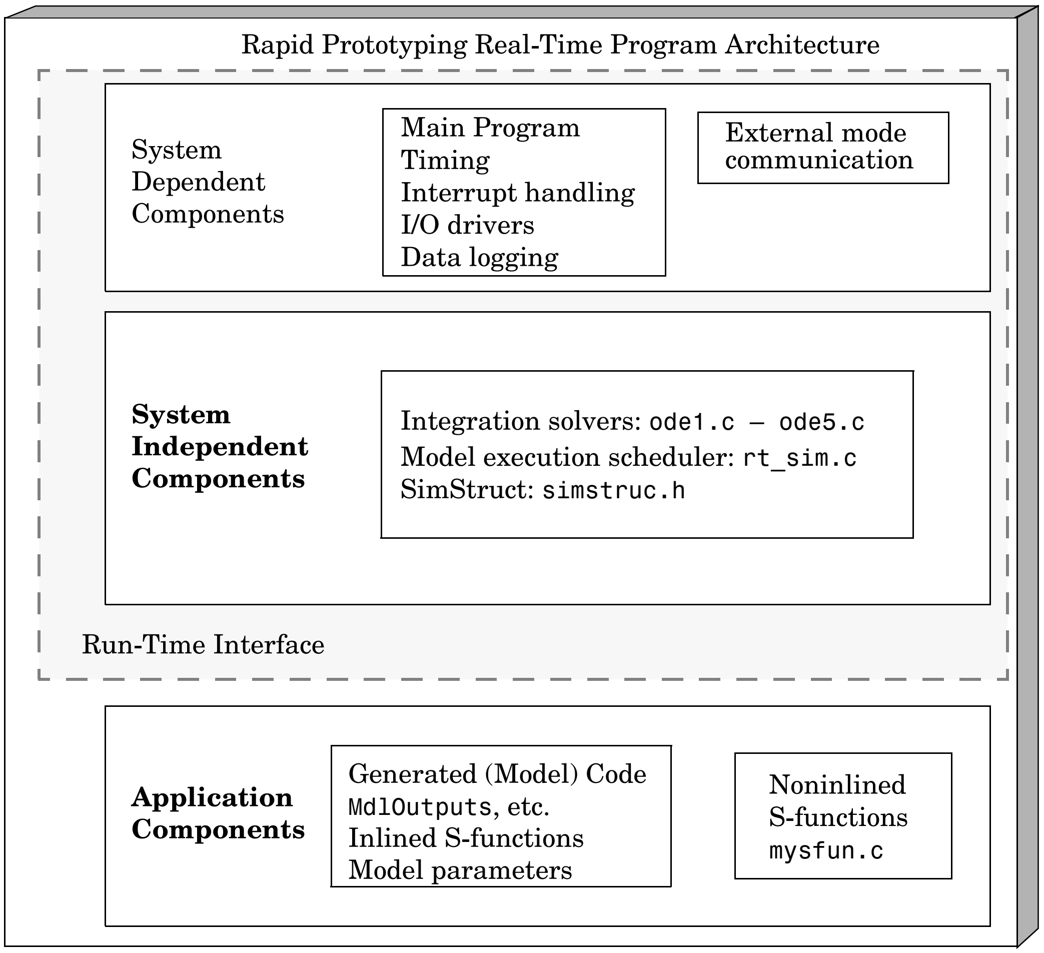
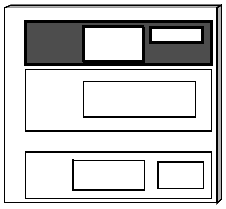


Figure 6-6: The Rapid Prototyping Program Architecture

The Real-Time Workshop architecture consists of three parts. The first two components, system dependent and independent, together form the run-time interface. The system dependent and independent components can be viewed together collectively as the *run-time interface*.

This architecture readily adapts to a wide variety of environments by isolating the dependencies of each program component. The following sections discuss each component in more detail and include descriptions of the application modules that implement the functions carried out by the system dependent, system independent, and application components.

Rapid Prototyping System Dependent Components



These components contain the program's main function, which controls program timing, creates tasks, installs interrupt handlers, enables data logging, and performs error checking.

The way in which application modules implement these operations depends on the type of computer. This means that, for example, the components used for a DOS-based program perform the same operations, but differ in method of implementation from components designed to run under Tornado on a VME target.

The main Function

The `main` function in a C program is the point where execution begins. In Real-Time Workshop application programs, the `main` function must perform certain operations. These operations can be grouped into three categories — initialization, model execution, and program termination.

Initialization

- Initialize special numeric parameters: `rtInf`, `rtMinusInf`, and `rtNaN`. These are variables that the model code can use.
- Call the model registration function to get a pointer to the `SimStruct`. The model registration function has the same name as your model. It is responsible for initializing `SimStruct` fields and any S-functions in your model.
- Initialize the model size information in the `SimStruct`. This is done by calling `MdlInitializeSizes`.
- Initialize a vector of sample times and offsets (for systems with multiple sample rates). This is done by calling `MdlInitializeSampleTimes`.
- Get the model ready for execution by calling `MdlStart`, which initializes states and similar items.
- Set up the timer to control execution of the model.
- Define background tasks and enable data logging, if selected.

Model Execution

- Execute a background task, for example, communicate with the host during external mode simulation or introduce a wait state until the next sample interval.
- Execute model (initiated by interrupt).
- Log data to buffer (if data logging is used).
- Return from interrupt.

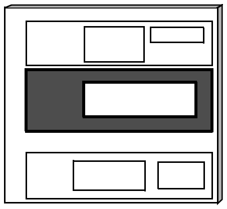
Program Termination

- Call a function to terminate the program if it is designed to run for a finite time — destroy the `SimStruct`, deallocate memory, and write data to a file.

Rapid Prototyping Application Modules for System Dependent Components

The application modules contained in the system dependent components generally include a main module such as `rt_main.c` containing the main entry point for C. There may also be additional application modules for such things as I/O support and timer handling.

Rapid Prototyping System Independent Components



These components are collectively called system independent because all environments use the same application modules to implement these operations. This section steps through the model code (and if the model has continuous states, calls one of the numerical integration routines). This section also includes the code that defines, creates, and destroys the Simulink data structure (`SimStruct`). The model code and all S-functions included in the program define their own `SimStruct`.

The model code execution driver calls the functions in the model code to compute the model outputs, update the discrete states, integrate the continuous states (if applicable), and update time. These functions then write their calculated data to the `SimStruct`.

Model Execution

At each sample interval, the main program passes control to the model execution function, which executes one step though the model. This step reads inputs from the external hardware, calculates the model outputs, writes outputs to the external hardware, and then updates the states.

The following diagram illustrates these steps.

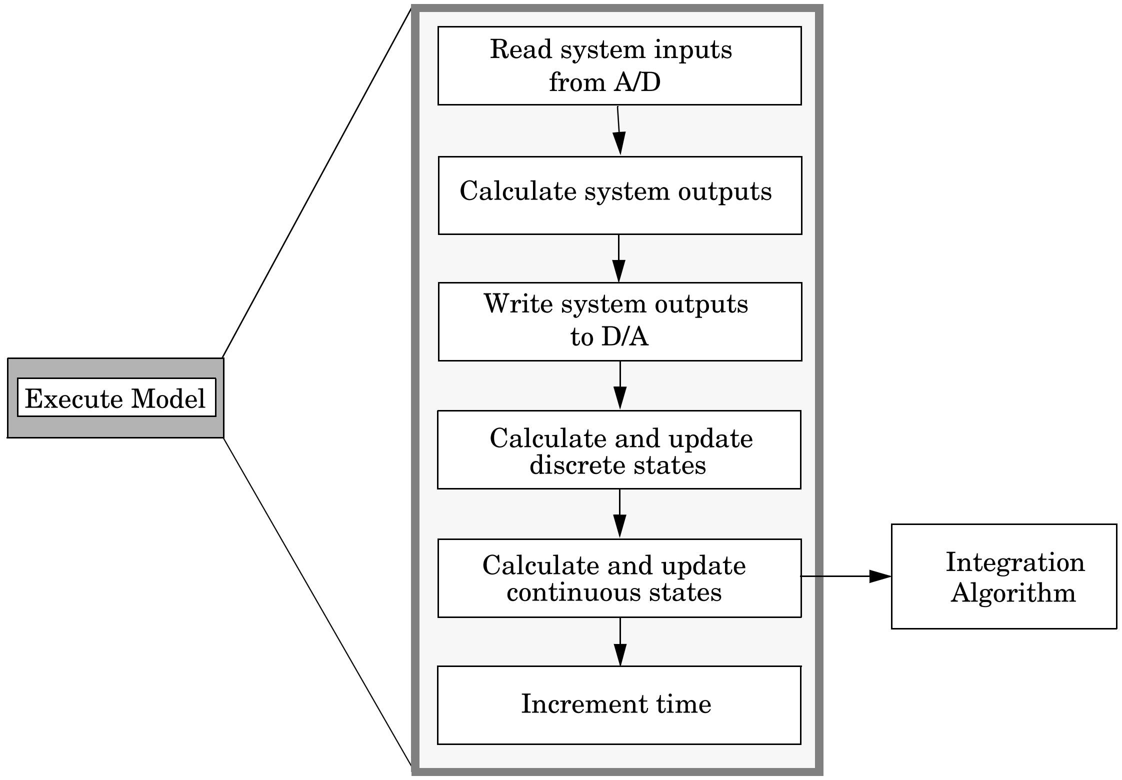

Execute Model
Read system inputs from A/D
Calculate system outputs
Write system outputs to D/A
Calculate and update discrete states
Calculate and update continuous states
Increment time
Integration Algorithm

Figure 6-7: Executing the Model

Note that this scheme writes the system outputs to the hardware before the states are updated. Separating the state update from the output calculation minimizes the time between the input and output operations.

Integration of Continuous States

The real-time program calculates the next values for the continuous states based on the derivative vector, dx/dt, for the current values of the inputs and the state vector.

These derivatives are then used to calculate the next value of the states using a state-update equation. This is the state-update equation for the first order Euler method (`ode1`).

$$x = x + \frac{dx}{dt}h$$

where h is the step size of the simulation, x represents the state vector, and dx/dt is the vector of derivatives. Other algorithms may make several calls to the output and derivative routines to produce more accurate estimates.

Note, however, that real-time programs use a fixed-step size since it is necessary to guarantee the completion of all tasks within a given amount of time. This means that, while you should use higher order integration methods for models with widely varying dynamics, the higher order methods require additional computation time. In turn, the additional computation time may force you to use a larger step size, which can diminish the accuracy increase initially sought from the higher order integration method.

Generally, the stiffer the equations, (i.e., the more dynamics in the system with widely varying time constants), the higher the order of the method that you must use.

In practice, the simulation of very stiff equations is impractical for real-time purposes except at very low sample rates. You should test fixed-step size integration in Simulink to check stability and accuracy before implementing the model for use in real-time programs.

For linear systems, it is more practical to convert the model that you are simulating to a discrete time version, for instance, using the `c2d` function in the Control System Toolbox.

Application Modules for System Independent Components

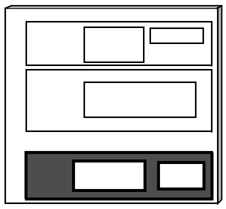
The system independent components include these modules:

- `rt_sim.c` — Performs the activities necessary for one time step of the model. It calls the model function to calculate system outputs and then updates the discrete and continuous states.
- `ode1.c`, `ode2.c`, `ode3.c`, `ode4.c`, `ode5.c` — These modules implement the integration algorithms supported for real-time applications. See the Simulink documentation for more information about these fixed-step solvers.
- `simstruc.h` — Contains actual definition of the Simulink data structure and the definition of the `SimStruct` access macros.
- `simstruc_types.h` — Contains definitions of various events, including subsystem enable/disable and zero crossings. It also defines data logging variables.

The system independent components also include code that defines, creates, and destroys the Simulink data structure (`SimStruct`). The model code and all S-functions included in the program define their own `SimStruct`.

The `SimStruct` data structure encapsulates all the data relating to the model or S-function, including block parameters and outputs. See *Writing S-Functions* for more information about the `SimStruct`.

Rapid Prototyping Application Components



The application components contain the generated code for the Simulink model, including the code for any S-functions in the model. This code is referred to as the model code because these functions implement the Simulink model.

However, the generated code contains more than just functions to execute the model (as described in the previous section). There are also functions to perform initialization, facilitate data access, and complete tasks before program termination. To perform these operations, the generated code must define functions that:

- Create the `SimStruct`.
- Initialize model size information in the `SimStruct`.

- Initialize a vector of sample times and sample time offsets and store this vector in the `SimStruct`.
- Store the values of the block initial conditions and program parameters in the `SimStruct`.
- Compute the block and system outputs.
- Update the discrete state vector.
- Compute derivatives for continuous models.
- Perform an orderly termination at the end of the program (when the current time equals the final time, if a final time is specified).
- Collect block and scope data for data logging (either with the Real-Time Workshop or third-party tools).

The SimStruct Data Structure

The generated code includes the file `simstruc.h`, which contains the definition of the `SimStruct` data structure. Each instance of a model (or an S-function) in the program creates its own `SimStruct`, which it uses for reading and writing data.

All functions in the generated code are public. For this reason, there can be only one instance of a model in a real-time program. This function, which always has the same name as the model, is called during program initialization to return a pointer to the `SimStruct` and initialize any S-functions.

Rapid Prototyping Model Code Functions

The functions defined by the model code are called at various stages of program execution (i.e., initialization, model execution, or program termination).

The following diagram illustrates the functions defined in the generated code and shows what part of the program executes each function.

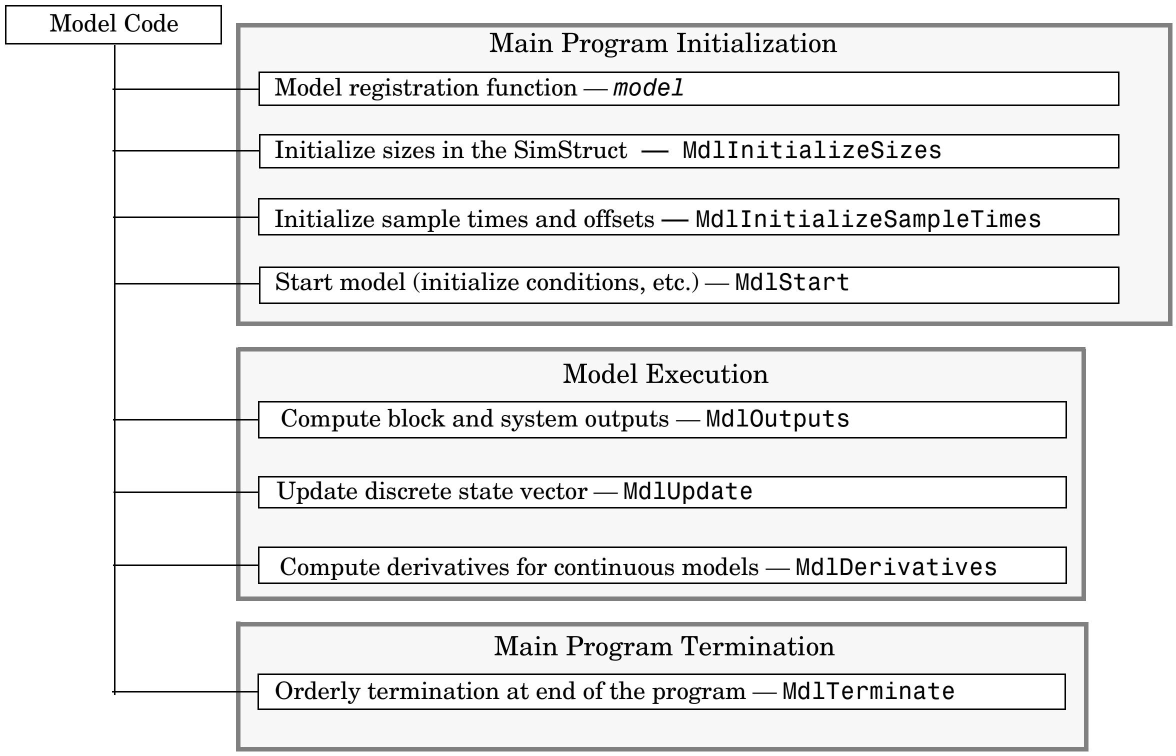

Model Code
Main Program Initialization
Model registration function — model
Initialize sizes in the SimStruct — MdlInitializeSizes
Initialize sample times and offsets — MdlInitializeSampleTimes
Start model (initialize conditions, etc.) — MdlStart
Model Execution
Compute block and system outputs — MdlOutputs
Update discrete state vector — MdlUpdate
Compute derivatives for continuous models — MdlDerivatives
Main Program Termination
Orderly termination at end of the program — MdlTerminate

Figure 6-8: Execution of the Model Code

This diagram shows what functions are defined in the generated code and where in the program these functions are called.

The Model Registration Function

The model registration function has the same name as the Simulink model from which it is generated. It is called directly by the main program during initialization. Its purpose is to initialize and return a pointer to the `SimStruct`.

Models Containing S-Functions

A *noninlined S-function* is any C MEX S-function that is not implemented using a customized TLC file. If you create a C MEX S-function as part of a Simulink model, it is by default noninlined unless you write your own TLC file that inlines it within the body of the *`model`*`.c` code. The Real-Time Workshop automatically incorporates your non-inlined C code S-functions into the program if they adhere to the S-function API described in the Simulink documentation.

This format defines functions and a `SimStruct` that are local to the S-function. This allows you to have multiple instances of the S-function in the model. The model's `SimStruct` contains a pointer to each S-function's `SimStruct`.

Code Generation and S-Functions

If a model contains S-functions, the source code for the S-function must be on the search path the `make` utility uses to find other source files. The directories that are searched are specified in the Rules section of the template makefile that is used to build the program.

See Chapter 3, "Code Generation and the Build Process," for more information on how S-function source code is located

S-functions are implemented in a way that is directly analogous to the model code. They contain their own public registration function (which is called by the top-level model code) that initializes static function pointers in its `SimStruct`. When the top-level model needs to execute the S-function, it does so via the function pointers in the S-function's `SimStruct`. The S-functions use the same `SimStruct` data structure as the generated code; however, there can be more than one S-function with the same name in your model. This is accomplished by having function pointers to static functions.

Inlining S-Functions

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to inline the S-function, thus improving performance by eliminating function calls to the S-function itself. For more information on inlining S-functions, see "Inlining an S-Function," in Chapter 3 of the *Target Language Compiler Reference Guide*.

Application Modules for Application Components

When the Real-Time Workshop generates code, it produces the following files:

- *`model`*`.c` — The C code generated from the Simulink block diagram. This code implements the block diagram's system equations as well as performing initialization and updating outputs.
- *`model`*`.h` — Header file containing the block diagram's simulation parameters, I/O structures, work structures, etc.
- *`model`*`_export.h` — Header file containing declarations of exported signals and parameters.
- *`model`*`.prm` — Header file containing parameter and other structure definitions.
- *`model`*`.reg` — Header file included at the bottom of `model.c`. This file contains the model registration, `MdlInitializeSizes`, and `ModelInitializeSampleTimes`.

These files are named for the Simulink model from which they are generated; *`model`* is replaced with the actual model name.

If you have created custom blocks using C MEX S-functions, you need the source code for these S-functions available during the build process.

Embedded Program Framework

The embedded program framework and architecture is outlined by the following figure.

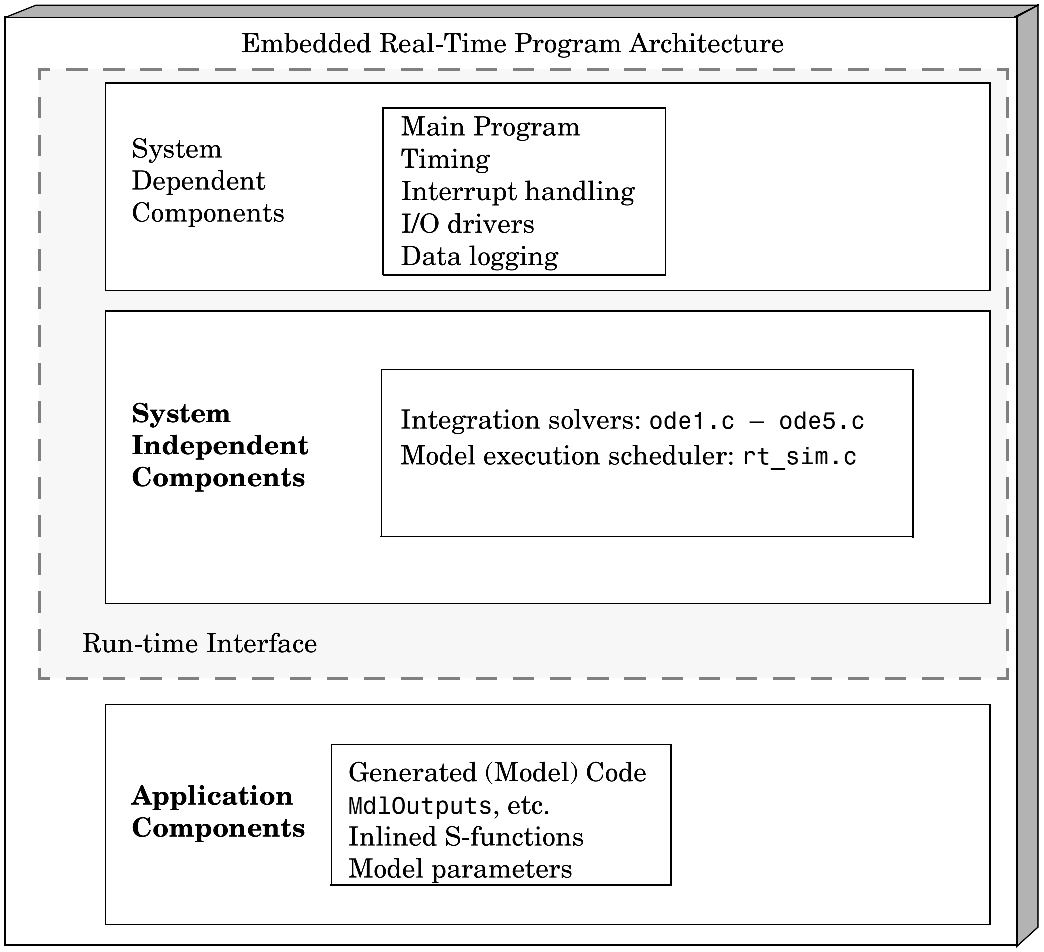


Figure 6-9: The Embedded Program Architecture

Note the similarity between this architecture and the rapid prototyping architecture on page 6-26. The main difference is the lack of the `SimStruct` data structure and the removal of the noninlined S-functions.

Using this figure, you can compare the embedded style of generated code with the rapid prototyping style of generated code of the previous section. Most of the rapid prototyping explanations in the previous section hold for the embedded target. The embedded target simplifies the process of using the generated code in your custom-embedded applications by providing a model-specific API and eliminating the `SimStruct`. The embedded target contains the same conceptual layering as the rapid prototyping target, however each layer has been simplified.

For a discussion of the embedded-C code, and all the other available code formats, see Chapter 12, "Configuring Real-Time Workshop for Your Application."

Models with Multiple Sample Rates

Introduction

Every Simulink block can be classified according to its sample time as constant, continuous-time, discrete-time, inherited, or variable. Examples of each type include:

- Constant — Constant block, Width
- Continuous-time — Integrator, Derivative, Transfer Function
- Discrete-time — Unit Delay, Digital Filter
- Inherited — Gain, Sum, Lookup Table
- Variable — These are S-functions blocks that set their time of next hit based upon current information. These blocks work only with variable step solvers.

Blocks in the inherited category assume the sample time of the blocks that are driving them. Every Simulink block therefore has a sample time, whether it is explicit, as in the case of continuous or discrete blocks (continuous blocks have a sample time of zero), or implicit, as in the case of inherited blocks.

Simulink allows you to create models without any restrictions on the connections between blocks with disparate sample times. It is therefore possible to have blocks with differing sample times in a model (a mixed-rate system). A possible advantage of employing multiple sample times is improved efficiency when executing in a multitasking real-time environment.

Simulink provides considerable flexibility in building these mixed-rate systems. However, the same flexibility also allows you to construct models for which the code generator cannot generate correct real-time code for execution in a multitasking environment. But to make these models operate correctly in real-time (i.e., give the right answers), you must modify your model. In general, the modifications involve placing Unit Delay and Zero Order Hold blocks between blocks that have unequal sample rates. The sections that follow discuss the issues you must address to use a mixed-rate model successfully in a multitasking environment.

Single- Versus Multitasking Environments

There are two basic ways in which you can execute a fixed-step Simulink model — singletasking and multitasking. You use the **Solver options** pull-down menu on the Solver page of the **Simulation Parameters** dialog box to specify how to execute your model. The default is auto, which specifies that your model will use multitasking if your model contains two or more different rates. Otherwise, it will use singletasking.

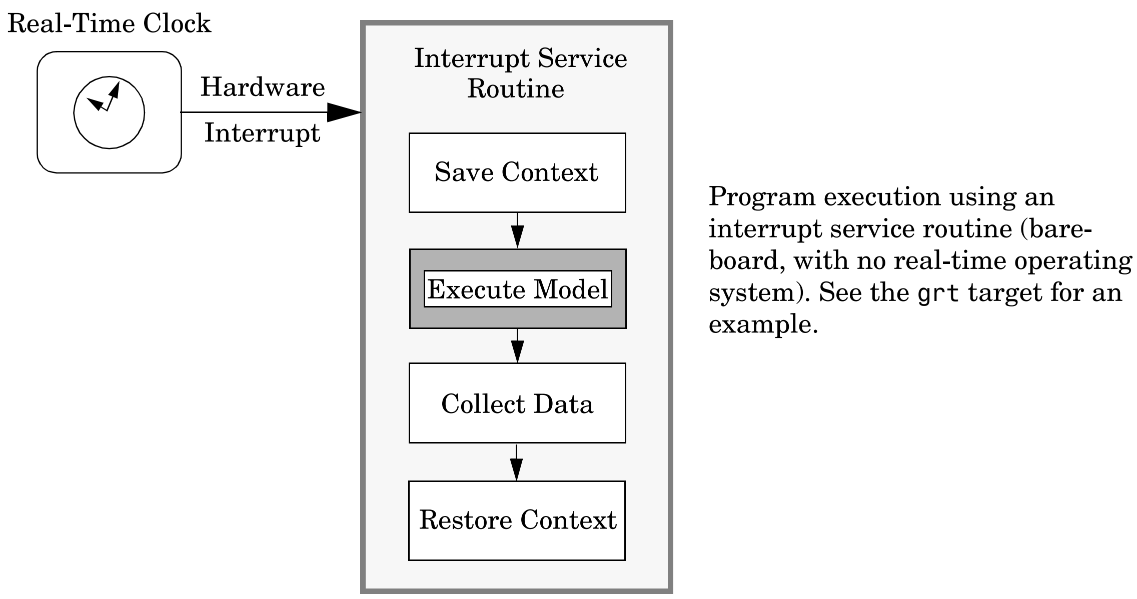
Execution of models in a real-time system can be done with the aid of a real-time operating system, or it can be done on a *bare-board* target, where the model runs in the context of an interrupt service routine (ISR).

Note that simply being a multitasking system, such as UNIX or MS-Windows, does not guarantee that the program can execute in real-time because you cannot guarantee that the program can preempt other processes when you want it to.

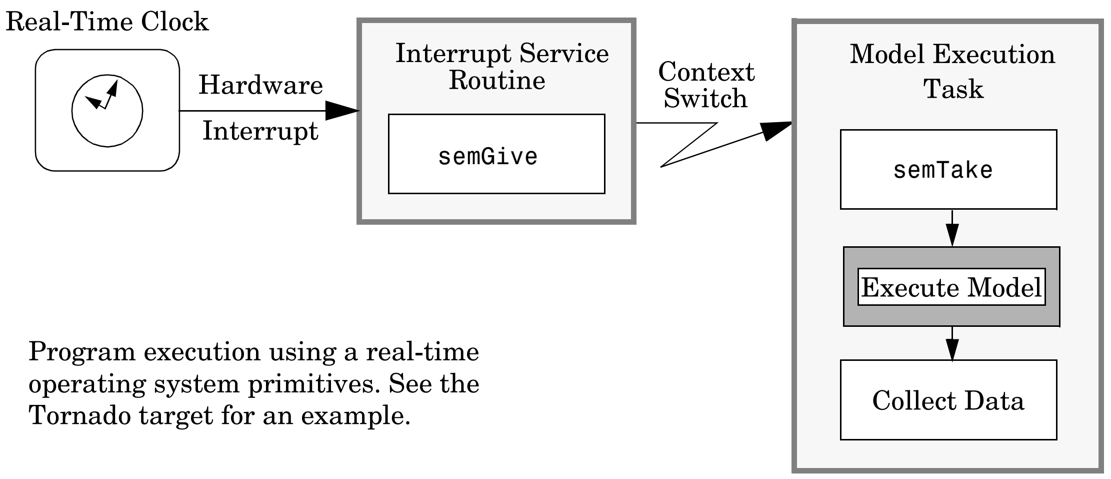
In DOS, where only one process can exist at any given time, the interrupt service routine (ISR) must perform the steps of saving the processor context, executing the model code, collecting data, and restoring the processor context.

Tornado, on the other hand, provides automatic context switching and task scheduling. This simplifies the operations performed by the ISR. In this case, the ISR simply enables the model execution task, which is normally blocked.

The following diagram illustrates this difference.



Program execution using an interrupt service routine (bare-board, with no real-time operating system). See the `grt` target for an example.



Program execution using a real-time operating system primitives. See the Tornado target for an example.

Figure 7-1: Real-Time Program Execution

This chapter focuses on when and how the run-time interface executes your model. See “Program Execution” on page 6-15 for a description of what happens during model execution.

Executing Multitasking Models

In cases where the continuous part of a model executes at a rate that is different from the discrete part, or a model has blocks with different sample rates, the code assigns each block a *task identifier* (tid) to associate it with the task that executes at its sample rate.

Certain restrictions apply to the sample rates that you can use:

- The sample rate of any block must be an integer multiple of the base (i.e., the fastest) sample rate. The base sample rate is determined by the fixed step size specified on the Solver page of the **Simulation parameters** dialog box (if a model has continuous blocks) or by the fastest sample time specified in the model (if the model is purely discrete). Continuous blocks always execute via an integration algorithm that runs at the base sample rate.
- The continuous and discrete parts of the model can execute at different rates only if the discrete part is executed at the same or a slower rate than the continuous part (and is an integer multiple of the base sample rate).

Multitasking and Pseudomultitasking

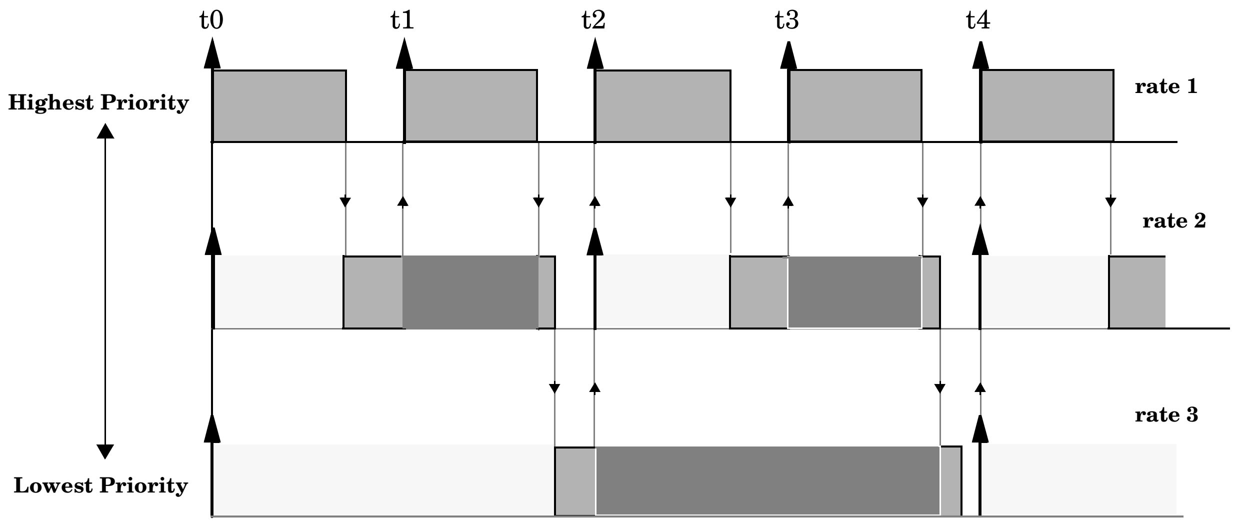
In a multitasking environment, the blocks with the fastest sample rates are executed by the task with the highest priority, the next slowest blocks are executed by a task with the next lower priority, and so on. Time available in between the processing of high priority tasks is used for processing lower priority tasks. This results in efficient program execution.

See “Multitasking System Execution” on page 7-6 for a graphical representation of task timing.

In multitasking environments (i.e., a real-time operating system), you can define separate tasks and assign them priorities. In a bare-board target (i.e., no real-time operating system present), you cannot create separate tasks. However, the Real-Time Workshop application modules implement what is effectively a multitasking execution scheme using overlapped interrupts, accompanied by manual context switching.

This means an interrupt can occur while another interrupt is currently in progress. When this happens, the current interrupt is preempted, the floating-point unit (FPU) context is saved, and the higher priority interrupt executes its higher priority (i.e., faster sample rate) code. Once complete, control is returned to the preempted ISR.

The following diagrams illustrate how mixed-rate systems are handled by the Real-Time Workshop in these two environments.



↑ Vertical arrows indicate sample times.

↓ Dotted lines with downward pointing arrows indicate the release of control to a lower priority task.

▢ Dark gray areas indicate task execution.

▨ Hashed areas indicate task preemption by a higher priority task.

↑ Dotted lines with upward pointing arrows indicate preemption by a higher priority task.

□ Light gray areas indicate task execution is pending.

Figure 7-2: Multitasking System Execution

.

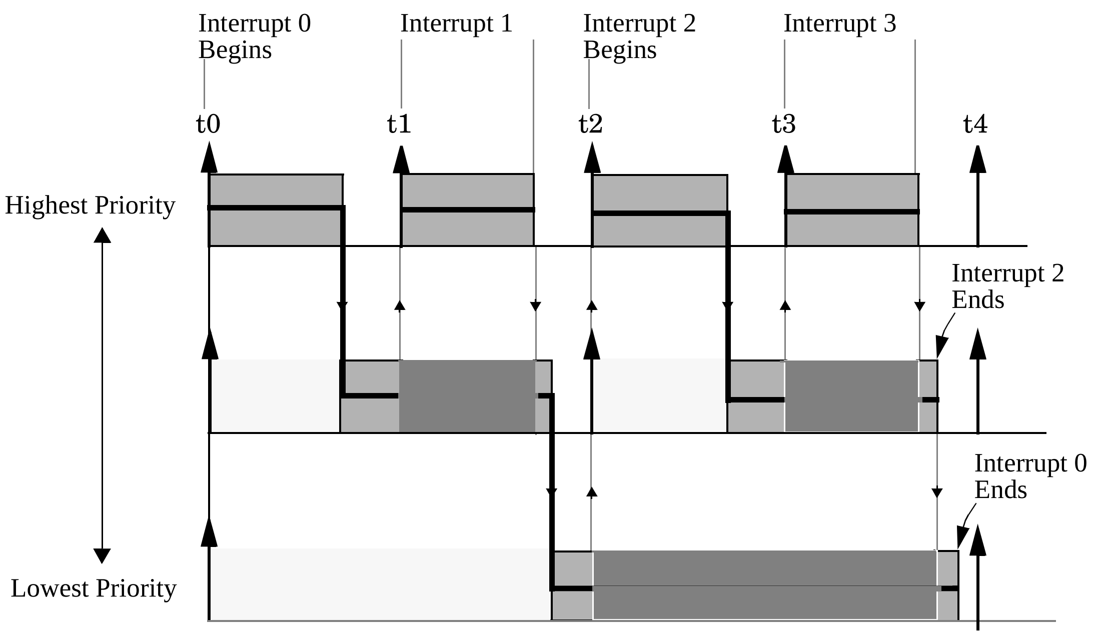


Figure 7-3: Pseudomultitasking Using Overlapped Interrupts

This diagram illustrates how overlapped interrupts are used to implement pseudomultitasking. Note that in this case, Interrupt 0 does not return until after Interrupts 1, 2, and 3.

Building the Program for Multitasking Execution

To use multitasking execution, select auto (the default) or multitasking as the mode on the Solver page of the **Simulation Parameters** dialog box. The Mode pull-down menu is only active if you have selected fixed-step as the Solver options type. Auto solver mode will result in a multitasking environment if your model has more than two sample times or it has two different sample times. In particular, a model with a continuous and a discrete sample time will run in singletasking mode if the fixed-step size is equal to the discrete sample time.

Singletasking

It is possible to execute the model code in a strictly singletasking manner. While this method is less efficient with regard to execution speed, in certain situations it may allow you to simplify your model.

In a singletasking environment, the base sample rate must define a time interval that is long enough to allow the execution of all blocks within that interval.

The following diagram illustrates the inefficiency inherent in singletasking execution:

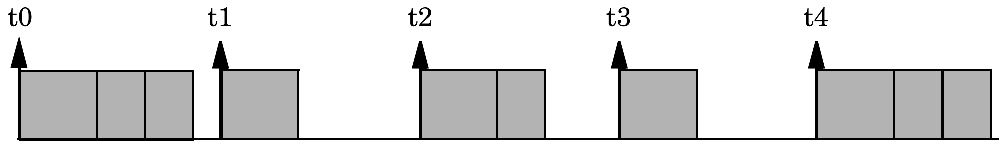


Figure 7-4: Singletasking System Execution

Singletasking system execution requires a sample interval that is long enough to execute one step through the entire model.

Building the Program for Singletasking Execution

To use singletasking execution, select the singletasking mode on the Solver page of the **Simulation Parameters** dialog box. If the solver mode is auto, singletasking will be used if your model:

- Contains one sample time, or
- Contains a continuous and a discrete sample time and the fixed step size is equal to the discrete sample time

Model Execution

To generate code that executes correctly in real-time, you may need to modify sample rate transitions within the model before generating code. To understand this process, first consider how Simulink simulations differ from real-time programs.

Simulating Models with Simulink

Before Simulink simulates a model, it orders all of the blocks based upon their topological dependencies. This includes expanding subsystems into the individual blocks they contain and flattening the entire model into a single list. Once this step is complete, each block is executed in order.

The key to this process is the proper ordering of blocks. Any block whose output is directly dependent on its input (i.e., any block with direct feedthrough) cannot execute until the block driving its input has executed.

Some blocks set their outputs based on values acquired in a previous time step or from initial conditions specified as a block parameter. The output of such a block is determined by a value stored in memory, which can be updated independently of its input. During simulation, all necessary computations are performed prior to advancing the variable corresponding to time. In essence, this results in all computations occurring instantaneously (i.e., no computational delay).

Executing Models in Real-Time

A real-time program differs from a Simulink simulation in that the program must execute the model code synchronously with real-time. Every calculation results in some computational delay. This means the sample intervals cannot be shortened or lengthened (as they can be in Simulink), which leads to less efficient execution:

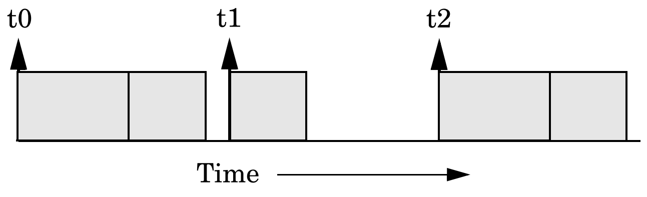


Figure 7-5: Unused Time in Sample Interval

Sample interval t1 cannot be compressed to increase execution speed because sample times must stay in sync with real-time.

Real-Time Workshop application programs are designed to circumvent this potential inefficiency by using a multitasking scheme. This technique defines tasks with different priorities to execute parts of the model code that have different sample rates.

See “Multitasking and Pseudomultitasking” on page 7–5 for a description of how this works. It is important to understand that section before proceeding here.

Multitasking Operation

The use of multitasking can improve the efficiency of your program if the model is large and has many blocks executing at each rate. It can also degrade performance if your model is dominated by a single rate, and only a few blocks execute at a slower rate. In this situation, the overhead incurred in task switching can be greater than the time required to execute the slower blocks. It is more efficient to execute all blocks at the dominant rate.

If you have a model that can benefit from multitasking execution, you may need to modify your Simulink model for this scheme to generate correct results.

Singletasking Operation

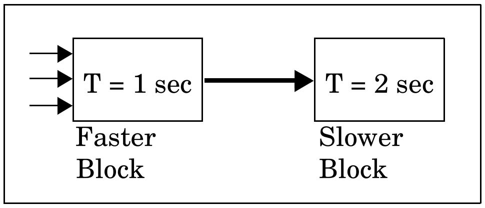
Alternatively, you can run your real-time program in singletasking mode. Singletasking programs require longer sample intervals due to the inherent inefficiency of that mode of execution.

Sample Rate Transitions

There are two possible sample rate transitions that can exist within a model:

- A faster block driving a slower block
- A slower block driving a faster block

In singletasking systems, there are no issues involved with multiple sample rates. In multitasking and pseudomultitasking systems, however, differing sample rates can cause problems. To prevent possible errors in calculated data, you must control model execution at these transitions. In transitioning from faster to slower blocks, you must add Zero-Order Hold blocks between fast to slow transitions and set the sample rate of the Zero-Order Hold to that of the slower block:



becomes

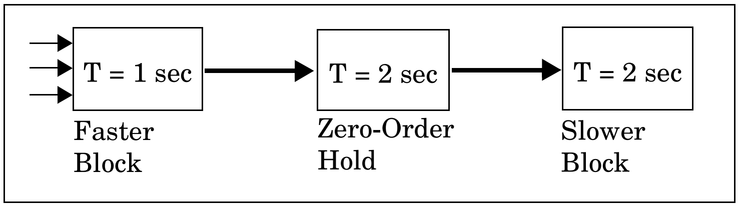
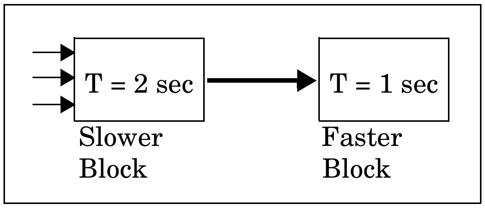


Figure 7-6: Transitioning from Faster to Slower Blocks (T = sample period)

In transitioning from slower to faster blocks, you must add Unit Delay blocks between slow to fast transitions and set the sample rate of the Unit Delay to that of the slower block:



becomes

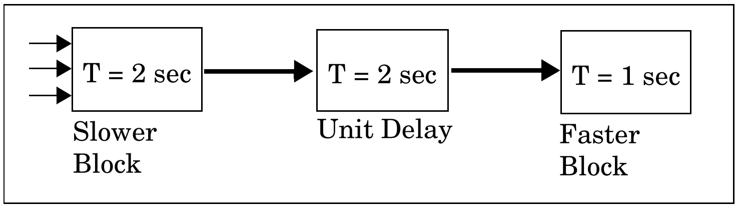
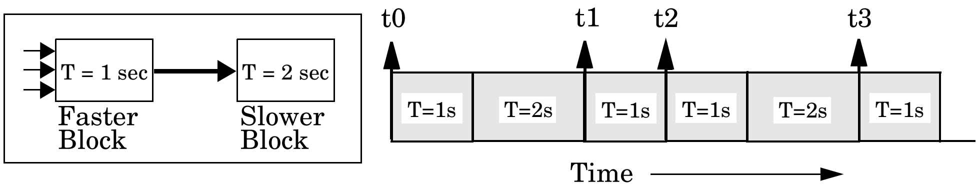


Figure 7-7: Transitioning from Slower to Faster Blocks (T = Sample Period)

The next four sections describe the theory and reasons why Unit Delay and Zero-Order Hold blocks are necessary for sample time transitions.

Faster to Slower Transitions in Simulink

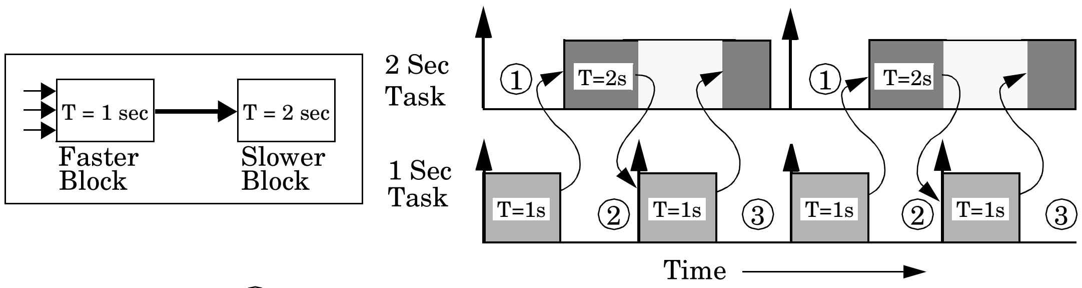
In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are always computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute. The following diagram illustrates this situation:



Simulink does not execute in real-time, which means that it is not bound by real-time constraints. Simulink waits for, or moves ahead to, whatever tasks are necessary to complete simulation flow. The actual time interval between sample time steps can vary.

Faster to Slower Transitions in Real-Time

In models where a faster block drives a slower block, you must compensate for the fact that execution of the slower block may span more than one execution period of the faster block. This means that the outputs of the faster block may change before the slower block has finished computing its outputs. The following diagram illustrates a situation where this problem arises. The hashed area indicates times when tasks are preempted by higher priority before completion:



① The faster task (T=1s) completes.

② Higher priority preemption occurs.

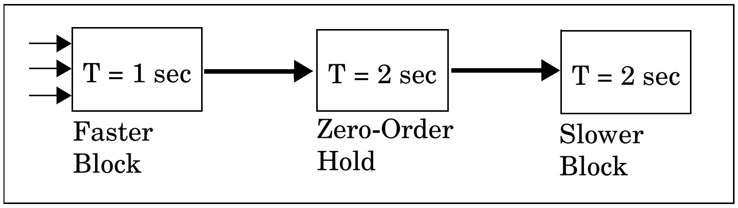
③ The slower task (T=2s) resumes and its inputs have changed. This leads to unpredictable results.

Figure 7-8: Time Overlaps in Faster to Slower Transitions (T=Sample Time)

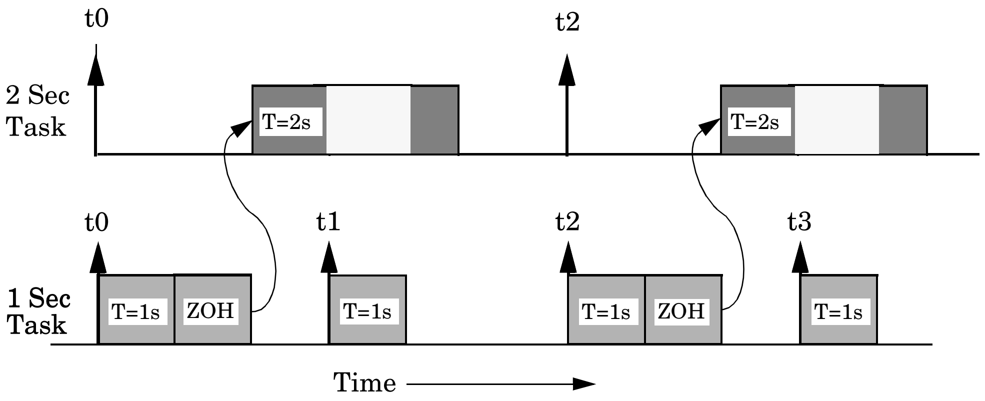
In this figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing.

To avoid this situation, you must hold the outputs of the 1 sec (faster) block until the 2 sec (slower) block finishes executing. The way to accomplish this is by inserting a Zero-Order Hold (ZOH) block between the 1 sec and 2 sec blocks.

The sample time of the Zero Order Hold block must be set to 2 sec (i.e., the sample time of the slower block).



The Zero Order Hold block executes at the sample rate of the slower block, but with the priority of the faster block.

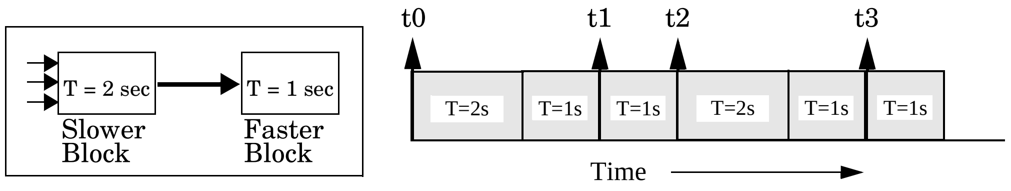


This ensures that the Zero Order Hold block executes before the 1 sec block (its priority is higher) and that its output value is held constant while the 2 sec block executes (it executes at the slower sample rate).

Slower to Faster Transitions in Simulink

In a model where a slower block drives a faster block, Simulink again computes the output of the driving block first. During sample intervals where only the faster block executes, the simulation progresses more rapidly.

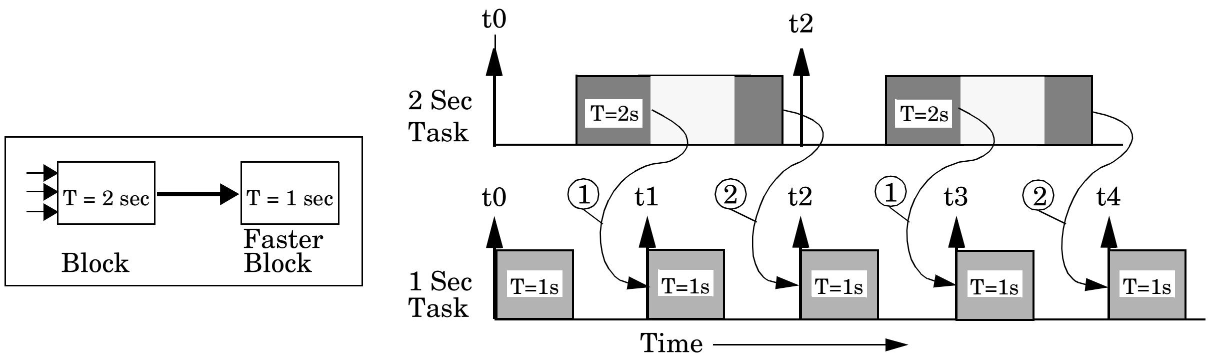
The following diagram illustrates the execution sequence:



As you can see from the preceding diagrams, Simulink can simulate models with multiple sample rates in an efficient manner. However, Simulink does not operate in real-time.

Slower to Faster Transitions in Real-Time

In models where a slower block drives a faster block, the generated code assigns the faster block a higher priority than the slower block. This means the faster block is executed before the slower block, which requires special care to avoid incorrect results.



(1) The faster block executes a second time prior to the completion of the slower block.

(2) The faster block executes before the slower block.

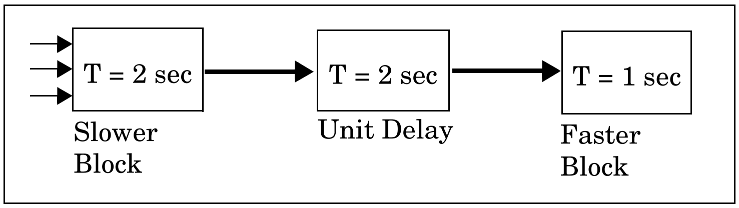
Figure 7-9: Time Overlaps in Slower to Faster transitions

This timing diagram illustrates two problems:

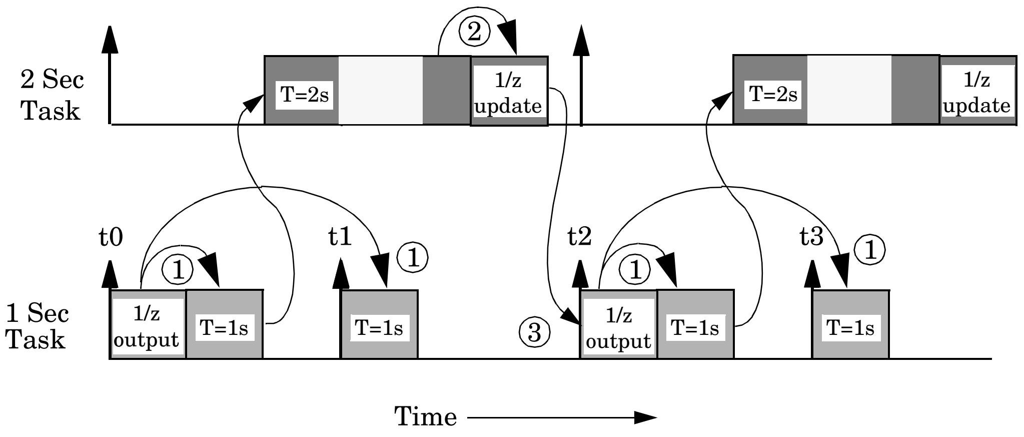
1 Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed execution. This means the inputs to the slower task can change, causing unpredictable results.

2 The faster block executes before the slower block (which is backwards from the way Simulink operates). In this case, the 1 sec block executes first; but the inputs to the faster task have not been computed. This can cause unpredictable results.

To eliminate these problems, you must insert a Unit Delay block between the slower and faster blocks. The sample rate for a Unit Delay block must be set to that of the block that is driving it (i.e., the slower block):



This pictures shows the timing sequence that results with the added Unit Delay block:



Three key points about this diagram:

1 Unit delay output runs in 1 sec task, but only at its rate (2 sec). The output of the unit delay block feeds the 1 sec task blocks.

2 The unit delay update uses the output of the 2 sec task in its update of its internal state.

3 The unit delay update uses the state of the unit delay in the 1 sec task.

The output portion of a Unit Delay block is executed at the sample rate of the slower block, but with the priority of the faster block. Since a Unit Delay block drives the faster block and has effectively the same priority, it is executed before the faster block. This solves the first problem.

The second problem is alleviated because the Unit Delay block executes at a slower rate and its output does not change during the computation of the faster block it is driving.

Note Inserting a Unit Delay block changes the model. The output of the slower block is now delayed by one time step compared to the output without a Unit Delay.

Targeting Tornado for Real-Time Applications

Introduction

This chapter describes how to create real-time programs for execution under VxWorks, which is part of the Tornado environment.

The VxWorks real-time operating system is available from Wind River Systems, Inc. It provides many UNIX-like features and comes bundled with a complete set of development tools.

Note Tornado is an integrated environment consisting of VxWorks (a high-performance real-time operating system), application building tools (compiler, linker, `make`, and archiver utilities), and interactive development tools (editor, debugger, configuration tool, command shell, and browser).

This chapter discusses the run-time architecture of VxWorks-based real-time programs generated by the Real-Time Workshop and provides specific information on program implementation. Topics covered include:

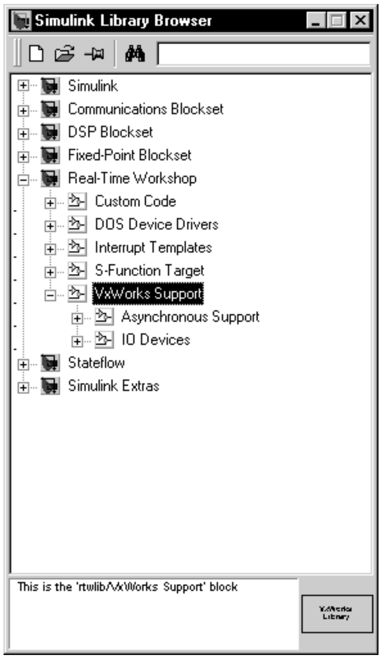
- Configuring device driver blocks and makefile templates
- Building the program
- Downloading the object file to the VxWorks target
- Executing the program on the VxWorks target
- Using Simulink external mode to change model parameters and download them to the executing program on the VxWorks target
- Using the StethoScope data acquisition and graphical monitoring tool, which is available as an option with VxWorks. It allows you to access the output of any block in the model (in the real-time program) and display the data on the host.

Confirming Your Tornado Setup Is Operational

Before beginning, you must install and configure Tornado on your host and target hardware, as discussed in the Tornado documentation. You should then run one of the VxWorks demonstration programs to ensure you can boot your VxWorks target and download object files to it. See the *Tornado User's Guide* for additional information about installation and operation of VxWorks and Tornado products.

VxWorks Library

Selecting **VxWorks Support** under the **Real-Time Workshop** library in the Simulink Library Browser opens the **VxWorks Support** library.



This is the 'rtwlib/VxWorks Support' block

The blocks discussed in this chapter are located in the Asynchronous Support library, which is a sublibrary of the VxWorks Support:

- Interrupt Control
- Rate Transition
- Read Side
- Task Synchronization
- Write Side

There is a second sublibrary, the IO Devices library, that contains support for these drivers:

- Matrix MS-AD12
- Matrix MS-DA12
- VME Microsystems VMIVM-3115-1
- Xycom XVME-500/590
- Xycom XVME-505/595

Each of these blocks has online help available through the **Help** button on the block's dialog box. Refer to the manufacturer's documentation for detailed information on these blocks.

Run-time Architecture Overview

In a typical VxWorks-based real-time system, the hardware consists of a UNIX or PC host running Simulink connected to a VxWorks target CPU via Ethernet. In addition, the target chassis may contain I/O boards with A/D and D/A converters to communicate with external hardware. The following diagram shows the arrangement:

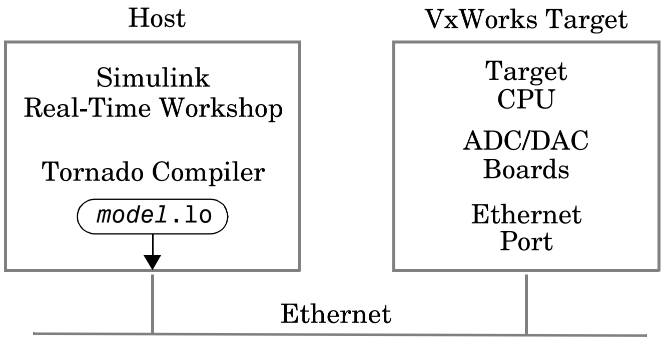


Figure 8-1: Typical Hardware Setup for a VxWorks Application

The real-time code is compiled on the UNIX or PC host using the cross compiler supplied with the VxWorks package. The object file (`model.lo`) output from the Real-Time Workshop program builder is downloaded, using `WindSh` (the command shell) in Tornado, to the VxWorks target CPU via an Ethernet connection.

The real-time program executes on the VxWorks target and interfaces with external hardware via the I/O devices installed on the target.

Parameter Tuning and Monitoring

You can change program parameters from the host and monitor data with Scope blocks while the program executes using Simulink external mode. You can also monitor program outputs using the StethoScope data analysis tool.

Using Simulink external mode and/or StethoScope in combination allows you to change model parameters in your program "on the fly" and to analyze the results of these changes in real-time.

External Mode

Simulink external mode provides a mechanism to download new parameter values to the executing program and to monitor signals in your model. In this mode, the external link MEX-file sends a vector of new parameter values to the real-time program via the network connection. These new parameter values are sent to the program whenever you make a parameter change without requiring a new code generation or build iteration.

Using VxWorks for signal monitoring is explained in the Chapter 5, “Data Logging and Signal Monitoring.” See “C API for Signal Monitoring” on page 5-4 for more information.

The real-time program (executing on the VxWorks target) runs a low priority task that communicates with the external link MEX-file and accepts the new parameters as they are passed into the program.

Communication between Simulink and the real-time program is accomplished using the sockets network API. This implementation requires an ethernet network that supports TCP/IP. See Chapter 4, “External Mode,” for more information on external mode.

Changes to the block diagram structure (for example, adding or removing blocks) require generation of model and execution of the build process.

Configuring VxWorks to Use Sockets

If you want to use Simulink external mode with your VxWorks program, you must configure your VxWorks kernel to support sockets by including the `INCLUDE_NET_INIT`, `INCLUDE_NET_SHOW`, and `INCLUDE_NETWORK` options in your VxWorks image. For more information on configuring your kernel, see the *VxWorks Programmer’s Guide*.

Before using external mode, you must ensure that VxWorks can properly respond to your host over the network. You can test this by using the host command

```
ping <target_name>
```

Note You may need to enter a routing table entry into VxWorks if your host is not on the same local network (subnet) as the VxWorks system. See `routeAdd` in the *VxWorks Reference Guide* for more information.

Configuring Simulink to Use Sockets

Simulink external mode uses a MEX-file to communicate with the VxWorks system. The MEX-file is

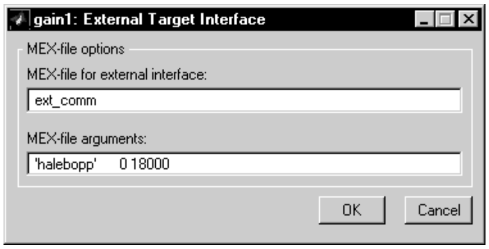
```
matlabroot/toolbox/rtw/ext_comm.*
```

where `*` is a host-dependent MEX-file extension. See Chapter 4, "External Mode," for more information.

To use external mode with VxWorks, specify `ext_comm` as the **MEX-file for external interface** in the **External Target Interface** dialog box (accessed from the **External Mode Control Panel**). In the **MEX-file arguments** field you must specify the name of the VxWorks target system and, optionally, the verbosity and TCP port number. Verbosity can be 0 (the default) or 1 if extra information is desired. The TCP port number ranges from 256 to 65535 (the default is 17725). If there is a conflict with other software using TCP port 17725, you can change the port that you use by editing the third argument of the **MEX-file for external interface** on the **External Target Interface** dialog box. The format for the MEX-file arguments field is

```
'target_network_name' [verbosity] [TCP port number]
```

For example, this picture shows the **External Target Interface** dialog box configured for a target system called `halebopp` with default verbosity and the port assigned to 18000:


gain1: External Target Interface

MEX-file options

MEX-file for external interface: ext_comm

MEX-file arguments: 'halebopp' 0 18000

StethoScope

With StethoScope, you can access the output of any block in the model (in the real-time program) and display this data on a host. Signals are installed in StethoScope by the real-time program using the `BlockIOSignals` data structure or, interactively from the `WindSh` while the real-time program is running. To use StethoScope interactively, see the *StethoScope User's Manual*.

To use StethoScope you must specify the proper options with the build command. See "Build Command Options" on page 8-17 for information on these options.

Run-Time Structure

The real-time program executes on the VxWorks target while Simulink and StethoScope execute on the same or different host workstations. Both Simulink and StethoScope require tasks on the VxWorks target to handle communication.

This diagram illustrates the structure of a VxWorks application using Simulink's external mode and StethoScope.

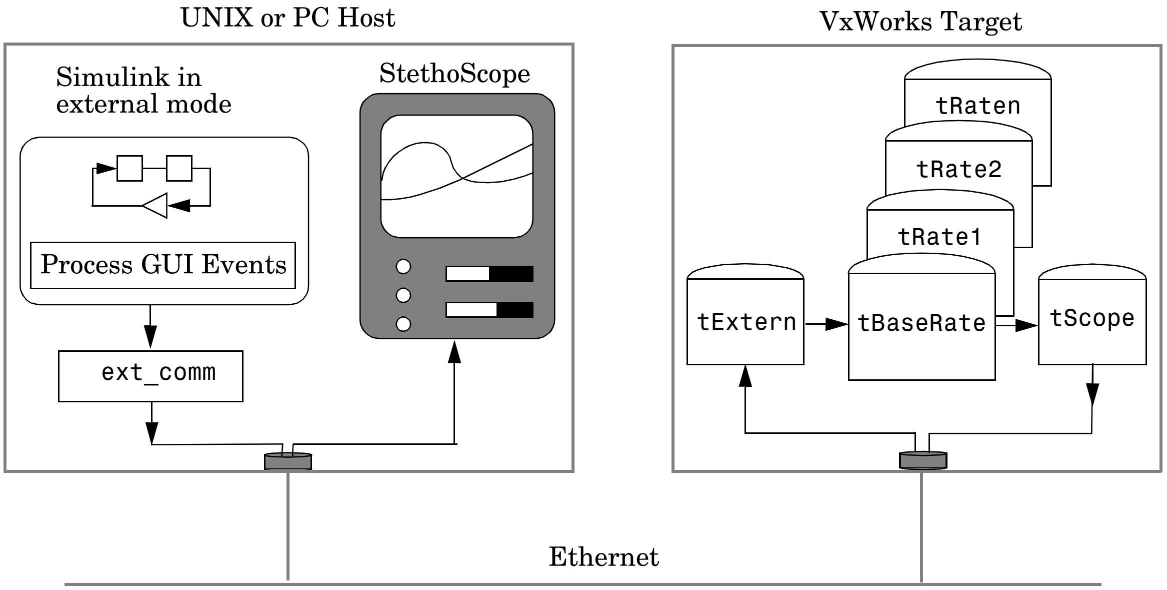


Figure 8-2: The Run-Time Structure

The program creates VxWorks tasks to run on the real-time system: one communicates with Simulink, the others execute the model. StethoScope creates its own tasks to collect data.

Host Processes

There are two processes running on the host side that communicate with the real-time program:

- Simulink running in external mode. Whenever you change a parameter in the block diagram, Simulink calls the external link MEX-file to download any new parameter values to the VxWorks target.
- The StethoScope user interface module. This program communicates with the StethoScope real-time module running on the VxWorks target to retrieve model data and plot time histories.

VxWorks Tasks

There are two modes in which the real-time program can be run—singletasking and multitasking. The code for both modes is located in:

```
matlabroot/rtw/c/tornado/rt_main.c
```

The Real-Time Workshop compiles and links `rt_main.c` with the model code during the build process.

Singletasking. By default, the model is run as one task, `tSingleRate`. This may actually provide the best performance (highest base sample rate) depending on the model. See Chapter 7, "Models with Multiple Sample Rates," for more information on singletasking vs. multitasking.

- `tSingleRate` — Runs at the base rate of the model and executes all necessary code for the slower sample rates. Execution of the `tSingleRate` task is normally blocked by a call to the VxWorks `semTake` routine. When a clock interrupt occurs, the interrupt service routine calls the `semGive` routine, which causes the `semTake` call to return. Once enabled, the `tSingleRate` task executes the model code for one time step. The loop then waits at the top by again calling `semTake`. For more information about the `semTake` and `semGive` routines, refer to the *VxWorks Reference Manual*. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.

Multitasking. Optionally, the model can run as multiple tasks, one for each sample rate in the model.

- `tBaseRate` — This task executes the components of the model code run at the base (highest) sample rate. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.
- `tRate`*`n`* — The program also spawns a separate task for each additional sample rate in the system. These additional tasks are named `tRate1`, `tRate2`, ..., `tRaten`, where `n` is slowest sample rate in the system. The priority of each additional task is one lower than its predecessor (`tRate1` has a lower priority than `tBaseRate`).

Supporting Tasks. If you select external mode and/or StethoScope during the build process, these tasks will also be created:

- `tExtern` — This task implements the server side of a socket stream connection that accepts data transferred from Simulink to the real-time program. In this implementation, `tExtern` waits for a message to arrive from Simulink. When a message arrives, `tExtern` retrieves it and modifies the specified parameters accordingly.

  `tExtern` runs at a lower priority than `tRaten`, the lowest priority model task. The source code for `tExtern` is located in *`matlabroot`*`/rtw/c/src/ext_svr.c`.
- `tScopeDaemon` and `tScopeLink` — StethoScope provides its own VxWorks tasks to enable real-time data collection and display. In singletasking mode, tSingleRate collects signals; in multitasking mode, tBaseRate collects them. Both perform the collection on every base time step. The StethoScope tasks then send the data to the host for display when there is idle time, that is, when the model is waiting for the next time step to occur. `rt_main.c` starts these tasks if they are not already running.

Implementation Overview

To implement and run a VxWorks-based real-time program using the Real-Time Workshop, you must:

- Design a Simulink model for your particular application
- Add the appropriate device driver blocks to the Simulink model, if desired
- Configure the `tornado.tmf` template makefile for your particular setup
- Establish a connection between the host running Simulink and the VxWorks target via ethernet
- Use the automatic program builder to generate the code and the custom makefile, invoke the `make` utility to compile and link the generated code, and load and activate the tasks required

The figure below shows the Real-Time Workshop Tornado run-time interface modules and the generated code for the F-14 example from Chapter 2:

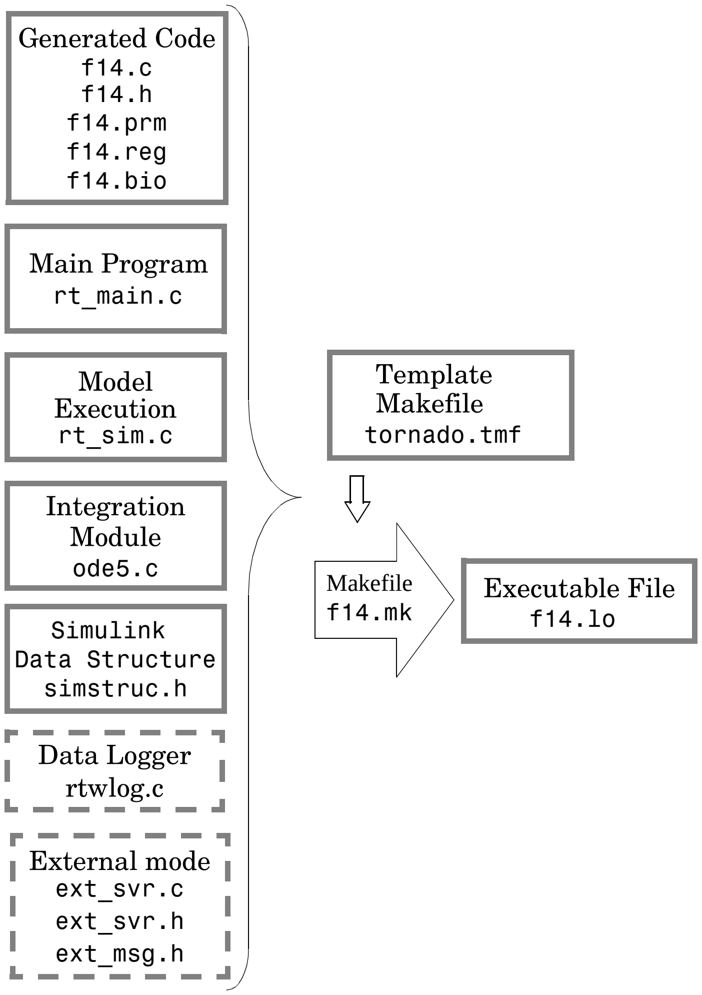


Figure 8-3: Source Modules Used to Build the VxWorks Real-Time Program

This diagram illustrates the code modules used to build a VxWorks real-time program. Dashed boxes indicate optional modules.

Adding Device Driver Blocks

The real-time program communicates with the I/O devices installed in the VxWorks target chassis via a set of device drivers. These device drivers contain the necessary code that runs on the target processor for interfacing to specific I/O devices.

To make device drivers easy to use, they are implemented as Simulink S-functions using C code MEX-files. This means you can connect them to your model like any other block and the code generator automatically includes a call to the block's C code in the generated code.

You can also inline S-functions via the Target Language Compiler. Inlining allows you to restrict function calls to only those that are necessary for the S-function. This can greatly increase the efficiency of the S-function. For more information about inlining S-functions, see *Writing S-Functions* and the *Target Language Compiler Reference Guide*.

You can have multiple instances of device driver blocks in your model. See Chapter 10, "Targeting Custom Hardware," for more information about creating device drivers.

Configuring the Template Makefile

To configure the VxWorks template, `tornado.tmf`, you must specify information about the environment in which you are using VxWorks. This section lists the lines in the file that you must edit.

VxWorks Configuration

To provide information used by VxWorks, you must specify the type of target and the specific CPU on the target. The target type is then used to locate the correct cross compiler and linker for your system.

The CPU type is used to define the `CPU` macro which is in turn used by many of the VxWorks header files. Refer to the *VxWorks Programmer's Guide* for information on the appropriate values to use.

This information is in the section labeled

```
#------------- VxWorks Configuration -------------
```

Edit the following lines to reflect your setup.

```
VX_TARGET_TYPE = 68k
CPU_TYPE = MC68040
```

Downloading Configuration

In order to perform automatic downloading during the build process, the target name and host name that the Tornado target server will run on must be specified. Modify these macros to reflect your setup.

```
#------------- Macros for Downloading to Target-------------
TARGET = targetname
TGTSVR_HOST = hostname
```

Tool Locations

In order to locate the Tornado tools used in the build process, the following three macros must either be defined in the environment or specified in the template makefile. Modify these macros to reflect your setup.

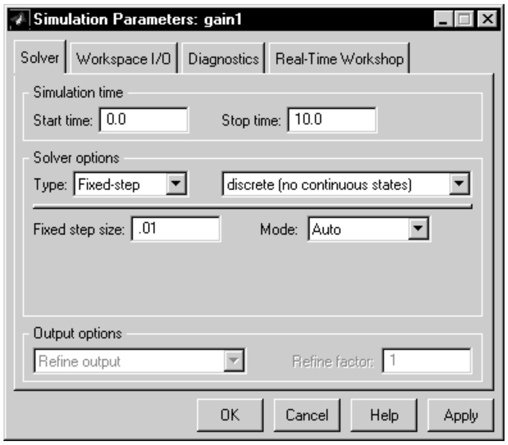
```
#------------- Tool Locations -------------
WIND_BASE = c:/Tornado
WIND_HOST_TYPE = x86–win32
WIND_REGISTRY = $(COMPUTERNAME)
```

Building the Program

Once you have created the Simulink block diagram, added the device drivers, and configured the makefile template, you are ready to set the build options and initiate the build process.

Specifying the Real-Time Build Options

Set the real-time build options using the Solver and Real-Time Workshop pages of the **Simulation Parameters** dialog box. To access this dialog box, select **Parameters** from the Simulink **Simulation** menu:



Solver Page

Simulation Parameters: gain1

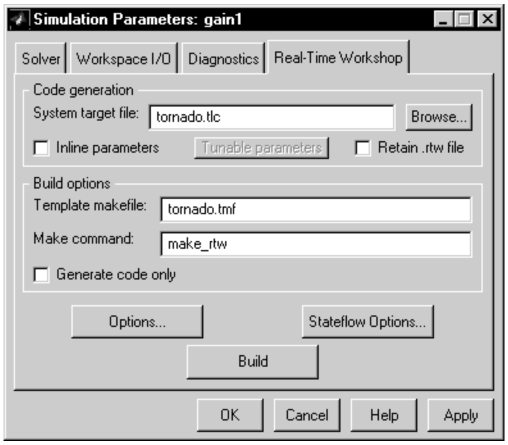
Solver | Workspace I/O | Diagnostics | Real-Time Workshop

Simulation time
Start time: 0.0   Stop time: 10.0

Solver options
Type: Fixed-step   discrete (no continuous states)
Fixed step size: .01   Mode: Auto

Output options
Refine output   Refine factor: 1

OK | Cancel | Help | Apply



Real-Time Workshop Page

Simulation Parameters: gain1

Solver | Workspace I/O | Diagnostics | Real-Time Workshop

Code generation
System target file: tornado.tlc   Browse...
☐ Inline parameters   Tunable parameters   ☐ Retain .rtw file

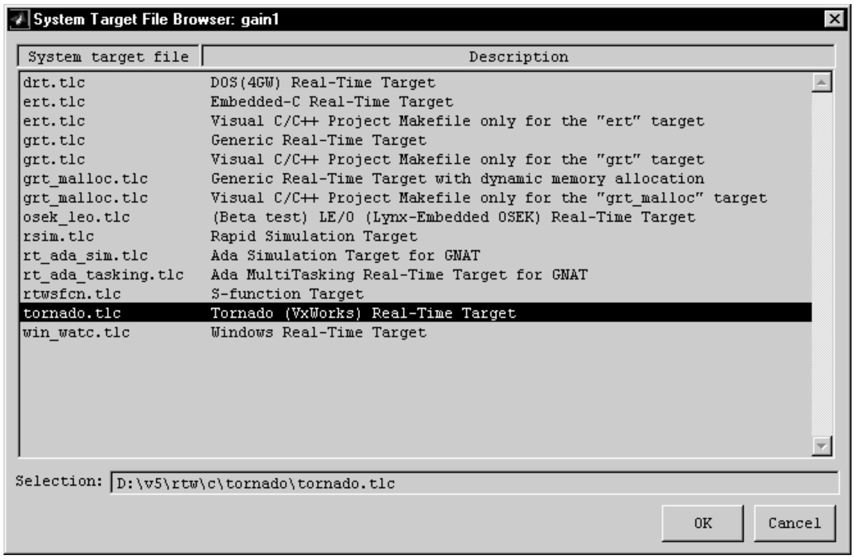
Build options
Template makefile: tornado.tmf
Make command: make_rtw
☐ Generate code only

Options... | Stateflow Options... | Build

OK | Cancel | Help | Apply

For models with continuous blocks, set the **Type** to **Fixed-step**, the **Step Size** to the desired integration step size, and select the integration algorithm. For models that are purely discrete, set the integration algorithm to **discrete**.

Next, use the **System Target File Browser** to select the correct Real-Time Workshop page settings for Tornado. You can open the browser by pressing the **Browse** button on the Real-Time Workshop page of the **Simulation Parameters** dialog box.



System target file	Description
drt.tlc	DOS(4GW) Real-Time Target
ert.tlc	Embedded-C Real-Time Target
ert.tlc	Visual C/C++ Project Makefile only for the "ert" target
grt.tlc	Generic Real-Time Target
grt.tlc	Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc	Generic Real-Time Target with dynamic memory allocation
grt_malloc.tlc	Visual C/C++ Project Makefile only for the "grt_malloc" target
osek_leo.tlc	(Beta test) LE/O (Lynx-Embedded OSEK) Real-Time Target
rsim.tlc	Rapid Simulation Target
rt_ada_sim.tlc	Ada Simulation Target for GNAT
rt_ada_tasking.tlc	Ada MultiTasking Real-Time Target for GNAT
rtwsfcn.tlc	S-function Target
tornado.tlc	Tornado (VxWorks) Real-Time Target
win_watc.tlc	Windows Real-Time Target

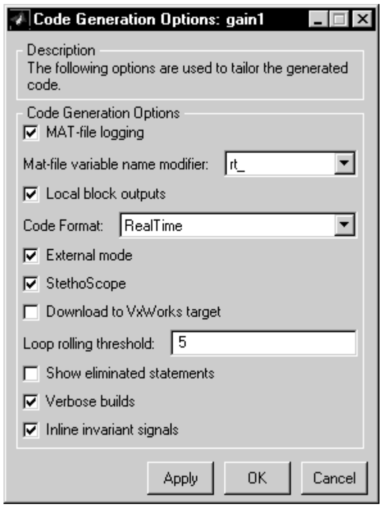
Selection: D:\v5\rtw\c\tornado\tornado.tlc

Selecting “Tornado (VxWorks) Real-Time Target” sets the following:

- **System target file** — `tornado.tlc`
- **Template makefile** — `tornado.tmf` template, which you must configure according to the instructions in “Configuring the Template Makefile” on page 8-13. (You can rename this file; simply change the dialog box accordingly.)
- **Make command** — `make_rtw`

You can optionally inline parameters for the blocks in the C code, which can improve performance. Inlining parameters is not allowed when using external mode. See Chapter 4, “External Mode,” for more information.

Build Command Options. You can specify build command options on the **Code Generation Options** dialog box. Click the **Options** button on the Real-Time Workshop to activate this dialog box.



The following options are used to tailor the generated code.

- [✓] MAT-file logging
- Mat-file variable name modifier: rt_
- [✓] Local block outputs
- Code Format: RealTime
- [✓] External mode
- [✓] StethoScope
- [ ] Download to VxWorks target
- Loop rolling threshold: 5
- [ ] Show eliminated statements
- [✓] Verbose builds
- [✓] Inline invariant signals

The Real-Time Workshop provides flags that set the appropriate macros in the template makefile, causing any necessary additional steps to be performed during the build process.

The flags and switches are as follows:

External mode — to enable the use of external mode in the generated executable, check **External Mode**. You can optionally enable a verbose mode of external mode by appending `-DVERBOSE` to the `OPTS` flag in the make command. For example,

```
make_rtw OPTS="-DVERBOSE"
```

causes parameter download information to be printed to the console of the VxWorks system.

- StethoScope — to enable the use of StethoScope with the generated executable, check **StethoScope**. There are two command line arguments, when starting `rt_main`, that control the block names used by StethoScope; you can use them when starting the program on VxWorks. See the section, "Running the Program" on page 8-19 for more information on these arguments.

- MATLAB MAT-file — to enable data logging during program execution, check **MAT-file logging**. The program will create a file named `MODEL.mat` at the end of program execution; this file will contain the variables that you specified in the Solver page of the **Simulation Parameters** dialog box. By default, the MAT-file is created in the root directory of the current default device in VxWorks. This is typically the host file system that VxWorks was booted from. Other remote file systems can be used as a destination for the MAT-file using rsh or ftp network devices or NFS. See the *VxWorks Programmer's Guide* for more information. If a device or filename other than the default is desired, add `"-DSAVEFILE=filename"` to the `OPTS` flag to the make command. For example,

  ```
  make_rtw OPTS="-DSAVEFILE=filename"
  ```

- Additional options are available on the **Code Generation Options** dialog box. To see descriptions of them, place your mouse over any field name. An online help description of the selected option will appear on your screen.

Initiating the Build

Click on the **Build** button in the Real-Time Workshop page of the **Simulation parameters** dialog to build the program. The resulting object file is named with the `.lo` extension (which stands for "loadable object"). This file has been compiled for the target processor using the cross compiler specified in the makefile. If automatic downloading (**Download to VxWorks target**) is enabled in the **Code Generation Options** dialog box, the target server is started and the object file is downloaded and started on the target. If **StethoScope** was checked on the **Code Generation Options** dialog box, you can now start StethoScope on the host. The StethoScope object files, `libxdr.so`, `libutilstssip.so`, and `libscope.so`, will be loaded on the VxWorks target by the automatic download. See the *StethoScope User's Manual* for more information.

Downloading and Running the Executable Interactively

If automatic downloading is disabled, you must use the Tornado tools to complete the process. This involves three steps:

1 Establishing a communication link to transfer files between the host and the VxWorks target

2 Transferring the object file from the host to the VxWorks target

3 Running the program

Connecting to the VxWorks Target

After completing the build process, you are ready to connect the host workstation to the VxWorks target. The first step is starting the target server that is used for communication between the Tornado tools on the host and the target agent on the target. This is done either from the DOS command line or from within the Tornado development environment. From the DOS command line use:

```
tgtsvr target_network_name
```

Downloading the Real-Time Program

To download the real-time program, use the VxWorks `ld` routine from within `WindSh`. `WindSh` (wind shell) can also be run from the command line or from within the Tornado development environment. (For example, if you want to download the file `vx_equal.lo`, which is in the `/home/my_working_dir` directory, use the following commands at the `WindSh` prompt:

```
cd "/home/my_working_dir"
ld <vx_equal.lo
```

You will also need to load the StethoScope libraries if the StethoScope option was selected during the build. The *Tornado User's Guide* describes the `ld` library routine.

Running the Program

The real-time program defines a function, `rt_main()`, that spawns the tasks to execute the model code and communicate with Simulink (if you selected external mode during the build procedure.) It also initializes StethoScope if you selected this option during the build procedure.

The `rt_main` function is defined in the `rt_main.c` application module. This module is located in the *`matlabroot`*`/rtw/c/tornado` directory.

The `rt_main` function takes six arguments, and is defined by the following ANSI C function prototype:

```
SimStruct *rt_main(void (*model)(SimStruct *),
                   char *optStr,
                   char *scopeInstallString,
                   int scopeFullNames,
                   int priority,
                   int TCPport);
```

The following table lists the arguments to this `SimStruct`.

Table 8-1: Arguments to the rt_main SimStruct

`model`	A pointer to the entry point function in the generated code. This function has the same name as the Simulink model. It registers the local functions that implement the model code by adding function pointers to the model's `SimStruct`. See Chapter 6, "Program Architecture," for more information.
`optStr`	The options string used to specify a stop time (`-tf`) and whether to wait (`-w`) in external mode for a message from Simulink before starting the simulation. An example options string: `"-tf 20 -w"`
`scopeInstallString`	A character string that determines which signals are installed to StethoScope. Possible values are: `NULL` (the default) — Install no signals. `"*"` — Install all signals. `"[A-Z]*"` — Install signals from blocks whose names start with a capital letter. Specifying any other string installs signals from blocks whose names start with that string.
`scopeFullNames`	This argument determines whether StethoScope uses full hierarchical block names for the signals it accesses or just the individual block name. Possible values are: `1` Use full block names; `0` Use individual block names (the default). It is important to use full block names if your program has multiple instances of a model or S-function.

Table 8-1: Arguments to the rt_main SimStruct (Continued)

`model`	A pointer to the entry point function in the generated code. This function has the same name as the Simulink model. It registers the local functions that implement the model code by adding function pointers to the model's `SimStruct`. See Chapter 6, "Program Architecture," for more information.
`priority`	The priority of the program's highest priority task (`tBaseRate`). Not specifying any value (or specifying a value of zero) causes `tBaseRate` to have the default priority of 30.
`TCPport`	The port number that the external mode sockets connection should use. The valid range is 256 to 65535 with 17725 as the default when nothing is specified.

Calling rt_main. To begin program execution, call `rt_main` from `WindSh`. For example,

```
sp(rt_main, vx_equal, "-tf 20 -w", "*", 0, 30, 17725)
```

- Begins execution of the `vx_equal` model
- Specifies no stop time so the program runs indefinitely
- Provides access to all signals (block outputs) in the model by StethoScope
- Uses only individual block names for signal access (instead of the potentially lengthy hierarchical name)
- Uses the default priority (30) for the `tBaseRate` task
- Uses TCP port 17725, the default

Targeting DOS for Real-Time Applications

Introduction

This chapter provides information that pertains specifically to using the Real-Time Workshop in a DOS environment.

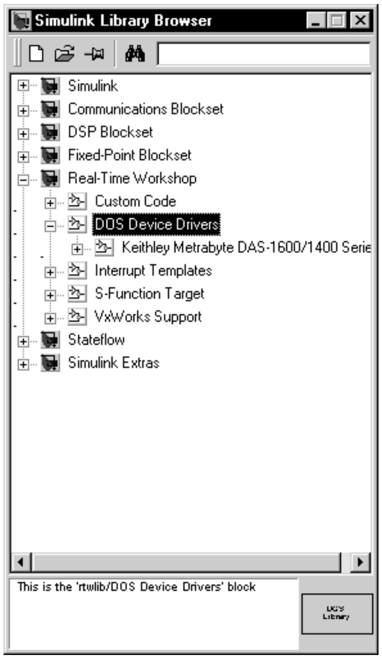
This chapter includes a discussion of:

- DOS-based Real-Time Workshop applications
- Supported compilers and development tools
- Device driver blocks — adding them to your model and configuring them for use with your hardware
- Building the program

The DOS target creates an executable using Watcom for DOS (i.e., a computer running the DOS operating system and not a Microsoft Windows DOS command prompt). This executable installs interrupt service routines and effectively takes over the computer, which allows the generated code to run in real-time. If you want to run the generated code in real-time under Microsoft Windows, you should use the Real-Time Windows Target, which is a separate product from the Real-Time Workshop. See the *Real-Time Windows Target User's Guide* for more information about this product.

DOS Device Drivers Library

Selecting **DOS Device Drivers** under the Real-Time Workshop library in the Simulink Library Browser opens the DOS Device Drivers library.



This is the 'rtwlib/DOS Device Drivers' block

The device drivers discussed in this chapter are located in this library.

Implementation Overview

The Real-Time Workshop includes DOS run-time interface modules designed to implement programs that execute in real-time under DOS. These modules, when linked with the code generated from a Simulink model, build a complete program that is capable of executing the model in real time. The DOS run-time interface files can be found in the *matlabroot*/rtw/c/dos/rti directory.

Real-Time Workshop DOS run-time interface modules and the generated code for the F-14 example from Chapter 2 are shown in the figure below.

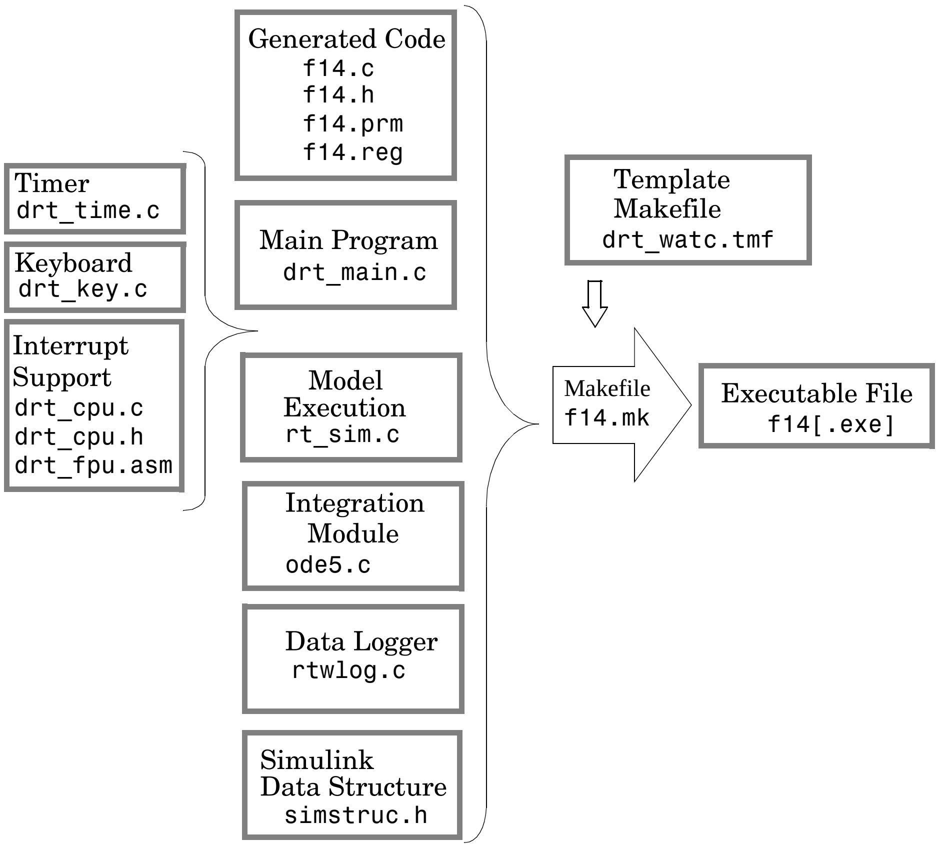


Figure 9-1: Source Modules Used to Build the DOS Real-Time Program

This diagram illustrates the code modules that are used to build a DOS real-time program.

To execute the code in real time, the program runs under the control of an interrupt driven timing mechanism. The program installs its own interrupt service routine (ISR) to execute the model code periodically at predefined sample intervals. The PC-AT's 8254 Programmable Interval Timer is used to time these intervals.

In addition to the modules shown in Figure 10-1, the DOS run-time interface also consists of device driver modules to read from and write to I/O devices installed on the DOS target.

Figure 9-2 shows the recommended hardware setup for designing control systems using Simulink, and then building them into DOS Realtime applications using the Real-Time Workshop. The figure shows a robotic arm being controlled by a program (i.e., the controller) executing on the Target-PC. The controller senses the arm position and applies inputs to the motors accordingly, via the I/O devices on the target PC. The controller code executes on the PC and communicates with the apparatus it controls via I/O hardware.

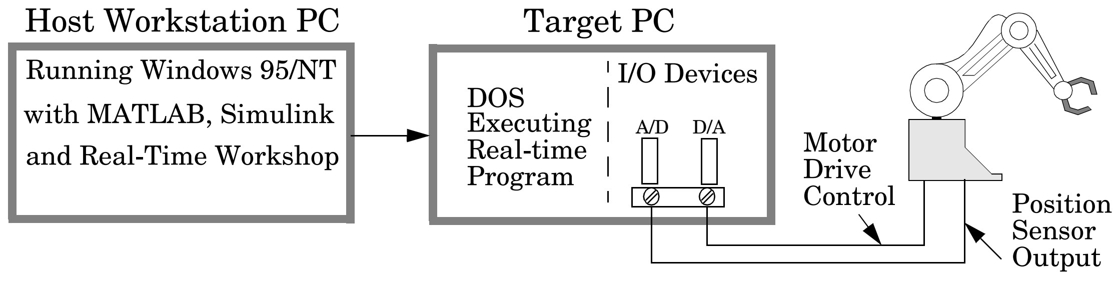

Host Workstation PC
Running Windows 95/NT with MATLAB, Simulink and Real-Time Workshop
Target PC
DOS Executing Real-time Program
I/O Devices
A/D
D/A
Motor Drive Control
Position Sensor Output

Figure 9-2: Typical Hardware Setup

System Configuration

You can use the Real-Time Workshop with a variety of system configurations, as long as these systems meet the following hardware and software requirements.

Hardware Requirements

The hardware needed to develop and run a real-time program includes:

- A workstation running Windows 95, Windows 98, or Windows NT and capable of running MATLAB/Simulink. This workstation is the *host* where the real-time program is built.
- A PC-AT (386 or higher) running DOS. This system is the *target*, where the real-time program executes.
- I/O boards, which include analog to digital converter and digital to analog converters (collectively referred to as I/O devices), on the target.
- Electrical connections from the I/O devices to the apparatus you want to control (or to use as inputs and outputs to the program in the case of hardware-in-the-loop simulations).

Once built, you can run the executable on the target hardware as a stand-alone program that is independent of Simulink.

Software Requirements

The development host must have the following software:

- MATLAB, Simulink to develop the model, and Real-Time Workshop to create the code for the model. You also need the run-time interface modules included with the Real-Time Workshop. These modules contain the code that handles timing, interrupts, data logging, and background tasks.
- Watcom C/C++ compiler, Version 10.6.

The target PC must have the following software:

- DOS4GW extender `dos4gw.exe`, included with your Watcom compiler package) must be on the search path on the DOS-targeted PC.

You can compile the generated code (i.e., the files *`model`*`.c`, *`model`*`.h`, etc.) along with user-written code using other compilers. However, the use of 16-bit compilers is not recommended for any application.

Device Drivers

If your application needs to access its I/O devices on the target, then the real-time program must contain device driver code to handle communication with the I/O boards. The Real-Time Workshop DOS run-time interface includes source code of the device drivers for the Keithley Metrabyte DAS 1600/1400 Series I/O boards. See the “Device Driver Blocks” section for information on how to use these blocks.

Simulink Host

The development host must have Windows 95, Windows 98, or Windows NT to run Simulink. However, the real-time target requires only DOS, since the executable built from the generated code is not a Windows application. The real-time target will not run in a “DOS box” (i.e., a DOS window on Windows 95/98/NT).

Although it is possible to reboot the host PC under DOS for real-time execution, the computer would need to be rebooted under Windows 95/NT for any subsequent changes to the block diagram in Simulink. Since this process of repeated rebooting the computer is inconvenient, we recommend a second PC running only DOS as the real-time target.

Sample Rate Limits

Program timing is controlled by installing an interrupt service routine that executes the model code. The target PC’s CPU is then interrupted at the specified rate (this rate is determined from the step size).

The rate at which interrupts occur is controlled by application code supplied with the Real-Time Workshop. This code uses the PC-AT’s 8254 Counter/Timer to determine when to generate interrupts.

The code that sets up the 8254 Timer is in `drt_time.c`, which is in the `matlabroot\rtw\c\dos\rti` directory. It is automatically linked in when you build the program using the DOS real-time template makefile.

The 8254 chip is a 16-bit counter that operates at a frequency of 1.193 MHz. However, the timing module, drt_time.c in the DOS run-time interface can extend the range by an additional 16 bits in software, effectively yielding a 32-bit counter. This means that the slowest base sample rate your model can have is

$$1.193\times10^{6} \div (2^{32} - 1) \approx \frac{1}{3600} Hz$$

This corresponds to a maximum base step size of approximately one hour.

The fastest sample rate you can define is determined by the minimum value from which the counter can count down. This value is 3, hence the fastest sample rate that the 8254 is capable of achieving is:

$$1.193 \times 10^6 \div 3 \approx 4 \times 10^5 \text{Hz}$$

This corresponds to a minimum base step size of

$$1 \div 4 \times 10^5 \approx 2.5 \times 10^{-6} \text{seconds}$$

However, bear in mind that the above number corresponds to the fastest rate the timer can generate interrupts. It does not account for execution time for the model code, which would substantially reduce the fastest sample rate possible for the model to execute in real time. Execution speed is machine dependent and varies with the type of processor and the clock rate of the processor on the target PC.

The slowest and fastest rates computed above refer to the base sample times in the model. In a model with more than one sample time, you can define blocks that execute at slower rates as long as the sample times are an integer multiple of the base sample time.

Modifying Program Timing

If you have access to an alternate timer (e.g., some I/O boards include their own clock devices), you can replace the file `drt_time.c` with an equivalent file that makes use of the separate clock source. See the comments in `drt_time.c` to understand how the code works.

You can use your version of the timer module by redefining the `TIMER_OBJS` macros with the build command. For example, in the Real-Time Workshop page of the **Simulation parameters** dialog box, changing the build command to,

```
make_rtw TIMER_OBJS=my_timer.obj
```

replaces the file `drt_time.c` with `my_timer.c` in the list of source files used to build the program.

Device Driver Blocks

The real-time program communicates with external hardware via a set of device drivers. These device drivers contain the necessary code for interfacing to specific I/O devices.

The Real-Time Workshop includes device drivers for commercially available Keithley Metrabyte 1600/1400 Series I/O boards. These device drivers are implemented as C-coded, S-functions to interface with Simulink. This means you can add them to your model like any other block.

In addition, each of these S-function device drivers has a corresponding target file to inline the device driver in the model code. See Chapter 10, “Targeting Custom Hardware,” for information on implementing your own device drivers.

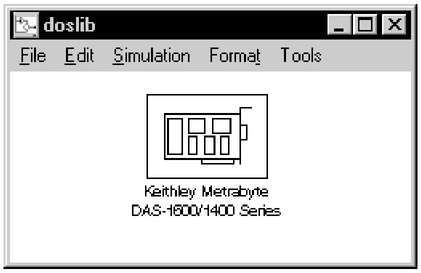
Since the device drivers are provided as source code, you can use these device drivers as a template to serve a a starting point for creating custom device drivers for other I/O boards.

Device Driver Block Library

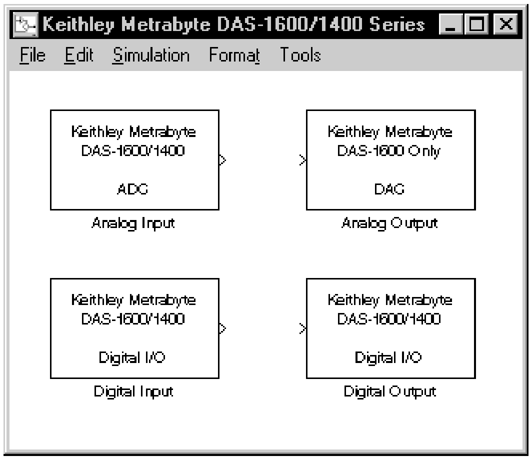
The device driver blocks for the Keithley Metrabyte 1600/1400 Series I/O boards designed for use with DOS applications are contained in a block library called `doslib` (`matlabroot\toolbox\rtw\doslib.mdl`). To display this library, type

```
doslib
```

at the MATLAB prompt. This window will appear:



To access the device driver blocks, double-click on the sublibrary icon.


Keithley Metrabyte DAS-1600/1400 Series
File Edit Simulation Format Tools
Keithley Metrabyte DAS-1600/1400 ADC
Analog Input
Keithley Metrabyte DAS-1600 Only DAC
Analog Output
Keithley Metrabyte DAS-1600/1400 Digital I/O
Digital Input
Keithley Metrabyte DAS-1600/1400 Digital I/O
Digital Output

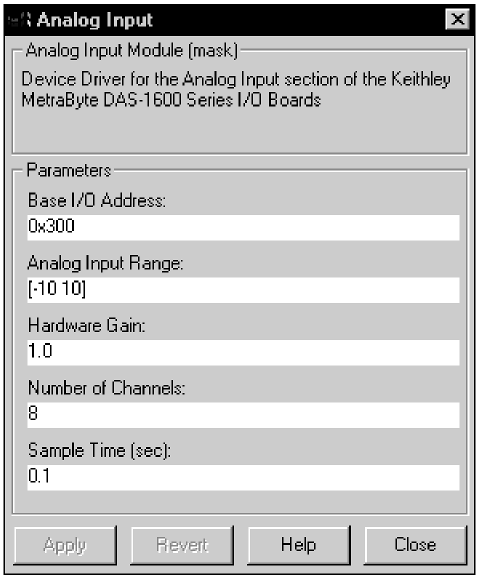
The blocks in the library contain device drivers that can be used for the DAS-1600/1400 Series I/O boards. The DAS-1601/1602 boards have 16 analog input (ADC) channels, two 12-bit analog output (DAC) channels and 4-bits of digital I/O. The DAS-1401/1402 boards do not have DAC channels. The DAS-1601/1401 boards have high programmable gains (1, 10, 100 and 500), while the DAS-1602/1402 boards offer low programmable gains (1, 2, 4 and 8). For more information, refer to the manufacturer's documentation for the I/O board.[1]

Configuring Device Driver Blocks

Each device driver block has a dialog box that you use to set configuration parameters. As with all Simulink blocks, double-clicking on the block displays the dialog box. Some of the device driver block parameters (such as Base I/O Address) are hardware specific and are set either at the factory or configured via DIP switches at the time of installation.

1. *DAS-1600/1400 Series User's Guide, Revision B* - August, 1996. Part Number: 80940, Keithley Metrabyte Division, Keithley Instruments, Inc., 440 Myles Standish Blvd., Taunton, MA 02780.
Web site: `www.metrabyte.com`

Analog Input (ADC) Block Parameters



Device Driver for the Analog Input section of the Keithley MetraByte DAS-1600 Series I/O Boards

Base I/O Address:
0x300

Analog Input Range:
[-10 10]

Hardware Gain:
1.0

Number of Channels:
8

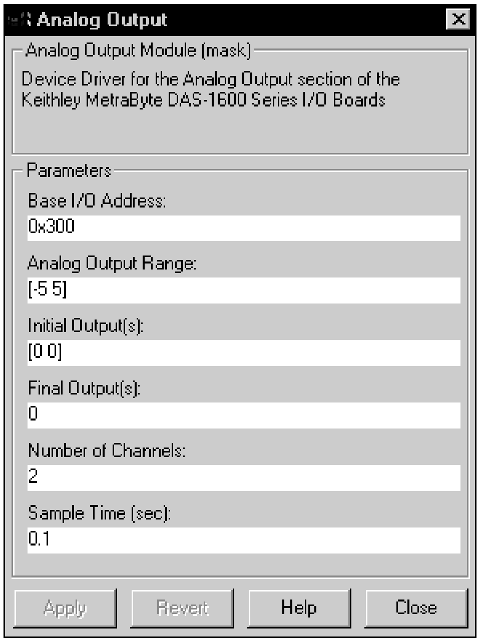
Sample Time (sec):
0.1

- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Analog Input Range** — This two-element vector specifies the range of values supported by the Analog to Digital Converter. The specified range must match the I/O board's settings. Specifically, the DAS 1600/1400 Series boards can be switch configured to either `[0 10]` for unipolar or `[-10 10]` for bipolar input signals.
- **Hardware Gain** — This parameter specifies the programmable gain that is applied to the input signal before presenting it to the ADC. Specifically, the DAS-1601/1401 boards have programmable gains of 1, 10, 100, and 500. The DAS-1602/1402 boards have programmable gains of 1, 2, 4, and 8. Configure the Analog Input Range and the Hardware Gain depending on the type and range of the input signal being measured. For example, a DAS-1601 board in bipolar configuration with a programmable gain of 100 is best suited to measure input signals in the range between [±10v] ÷ 100 = ±0.1v.

  Voltage levels beyond this range will saturate the block output form the ADC block. Please adhere to manufacturers' electrical specifications to avoid damage to the board.
- **Number of Channels** — The number of analog input channels enabled on the I/O board. The DAS-1600/1400 Series boards offer up to 16 ADC channels when configured in unipolar mode (8 ADC channels if you select differential mode). The output port width of the ADC block is equal to the number of channels enabled.

- **Sample Time (sec)** — Device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are executed at the specified rate. Specifically, when the ADC block is executed, it causes the ADC to perform a single conversion on the enabled channels, and the converted values are written to the block output vector.

Analog Output (DAC) Block Parameters



Analog Output Module (mask)

Device Driver for the Analog Output section of the Keithley MetraByte DAS-1600 Series I/O Boards

Parameters

Base I/O Address: 0x300

Analog Output Range: [-5 5]

Initial Output(s): [0 0]

Final Output(s): 0

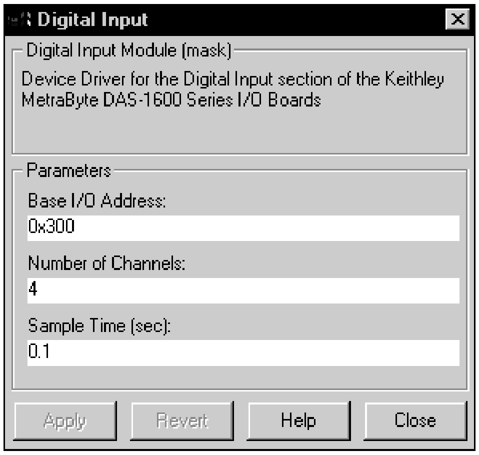
Number of Channels: 2

Sample Time (sec): 0.1

- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Analog Output Range** — This parameter specifies the output range settings of the DAC section of the I/O board. Typically, unipolar ranges are between `[0 10]` volts and bipolar ranges are between `[-10 10]` volts. Refer to the DAS-1600 documentation for other supported output ranges.
- **Initial Output(s)** — This parameter can be specified either as a scalar or as an N element vector, where N is the number of channels. If a single scalar value is entered, the same scalar is applied to output. The specified initial output(s) is written to the DAC channels in the `mdlInitializeConditions` function.
- **Final Output(s)** — This parameter is specified in a manner similar to the Initial Output(s) parameter except that the specified final output values are written out to the DAC channels in the `mdlTerminate` function. Once the generated code completes execution, the code sets the final output values prior to terminating execution.

- **Number of Channels** — Number of DAC channels enabled. The DAS-1600 Series I/O boards have two 12-bit DAC channels. The DAS-1400 Series I/O boards do not have any DAC channels. The input port width of this block is equal to the number of channels enabled.
- **Sample Time (sec)** — DAC device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are executed at the specified rate. Specifically, when the DAC block is executed, it causes the DAC to convert a single value on each of the enabled DAC channels, which produces a corresponding voltage on the DAC output pin(s).

Digital Input Block Parameters


Digital Input

Digital Input Module (mask)

Device Driver for the Digital Input section of the Keithley MetraByte DAS-1600 Series I/O Boards

Parameters

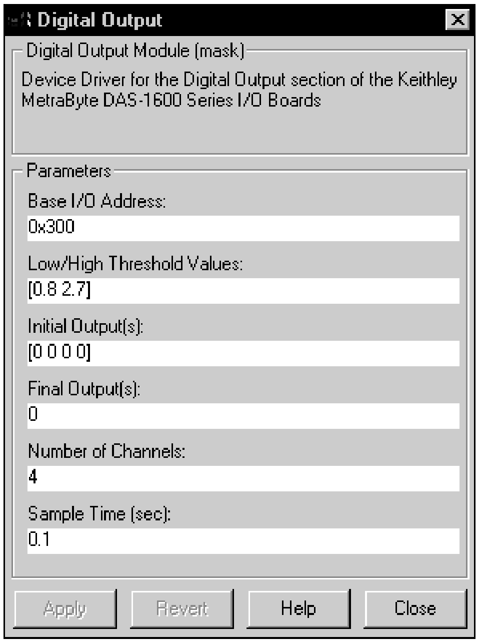
Base I/O Address:
0x300

Number of Channels:
4

Sample Time (sec):
0.1

- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Number of Channels** — This parameter specifies the number of 1-bit digital input channels being enabled. This parameter also determines the output port width of the block in Simulink. Specifically, the DAS-1600/1400 Series boards provide four bits (i.e., channels) for digital I/O.
- **Sample Time (sec)** — Digital input device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are executed at the specified rate. Specifically, when the digital input block is executed, it reads a boolean value from the enabled digital input channels. The corresponding input values are written to the block output vector.

Digital Output Block Parameters



Digital Output Module (mask)

Device Driver for the Digital Output section of the Keithley MetraByte DAS-1600 Series I/O Boards

Parameters

Base I/O Address:
0x300

Low/High Threshold Values:
[0.8 2.7]

Initial Output(s):
[0 0 0 0]

Final Output(s):
0

Number of Channels:
4

Sample Time (sec):
0.1

- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Low/High Threshold Values** — This parameter specifies the threshold levels, `[lo hi]`, for converting the block inputs into 0/1 digital values. The signal in the block diagram connected to the block input should rise above the high threshold level for a 0 to 1 transition in the corresponding Digital Output Channel on the I/O board. Similarly, the input should fall below the low threshold level for a 1 to 0 transition.
- **Initial Output(s)** — Same as the Analog Output block, except the specified values are converted to 0 or 1 based on the lower threshold value before they are written to the corresponding digital output channel.
- **Final Output(s)** — Same as the Analog Output block, except the specified values are converted to 0 or 1 based on the lower threshold value before they are written to the corresponding digital output channel on the I/O board.
- **Number of Channels** — This parameter specifies the number of 1-bit digital I/O channels being enabled. This parameter also determines the output port width of the block. Specifically, the DAS-1600/1400 Series boards provide four bits (i.e., channels) for digital I/O.
- **Sample Time (sec)** — Digital output device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are

executed at the specified rate. Specifically, when the digital output block is executed, it causes corresponding boolean values to be output from the board's digital I/O channels.

Adding Device Driver Blocks to the Model

Add device driver blocks to the Simulink block diagram as you would any other block — simply drag the block from the block library and insert it into the model. Connect the ADC or Digital Input module to the model's inputs and connect the DAC or Digital Output module to the model's outputs.

Including Device Driver Code

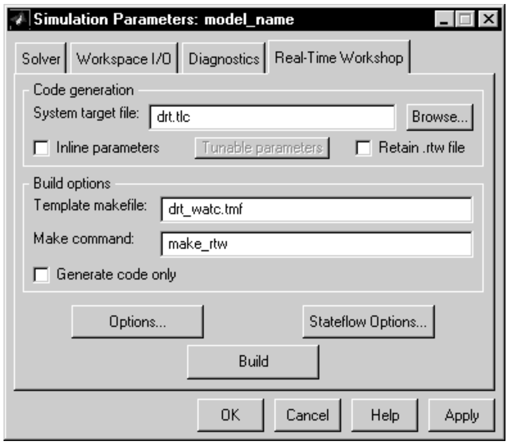
Device driver blocks are implemented as S-functions written in C. The C code for a device driver block is compiled as a MEX-file so that it can be called by Simulink. See the *MATLAB Application Program Interface Guide* for information on MEX-files.

The same C code can also be compiled and linked to the generated code just like any other C-coded, S-function. However, by using the target (`.tlc`) file that corresponds to each of the C file S-functions, the device driver code is inlined in the generated code.

The `matlabroot\rtw\c\dos\devices` directory contains the MEX-files, C files, and target files (`.tlc`) for the device driver blocks included in `doslib`. This directory is automatically added to your MATLAB path when you include any of the blocks from `doslib` in your model.

Building the Program

Once you have created your Simulink model and added the appropriate device driver blocks, you are ready to build a DOS target application. To do this, select **Parameters** from the **Simulation** menu of your Simulink model and display the Real-Time Workshop page of the **Simulink parameters** dialog box.



System target file: drt.tlc
☐ Inline parameters
☐ Retain .rtw file
Template makefile: drt_watc.tmf
Make command: make_rtw
☐ Generate code only

On the Real-Time Workshop page, specify:

- `drt.tlc` as the **System target file**
- `drt_watc.tmf` as the **Template makefile**. This is used with the Watcom compiler, assembler, linker, and `WMAKE` utility.
- `make_rtw` as the **Make command**

Alternatively, you can open the system target file browser by clicking **Browse** and select `drt.tlc`. This automatically fills in the correct files.

You can specify Target Language Compiler options in the system target file field following `drt.tlc`, and make options in the Make command field following `make_rtw`. Chapters 2 and 3 provide detailed descriptions of the available Target Language Compiler and make options. The DOS system target file, `drt.tlc`, and the template makefile, `drt_watc.tmf`, are located in the `matlab\rtw\c\dos` directory.

1 The template makefile assumes that the Watcom C/386 Compiler, Assembler, and Linker have been correctly installed on the host

workstation. You can verify this by checking the environment variable, `WATCOM`, which correctly points to the directory where the Watcom files are installed.

2. The program builder invokes the Watcom `wmake` utility on the generated makefile, so the directory where `wmake` is installed must be on your path.

Running the Program

The result of the build process is a DOS 32-bit protected-mode executable. The default name of this is `model.exe`, where `model` is the name of your Simulink model. You must run this executable in DOS; you cannot run the executable in Windows 95/98/NT.

Targeting Custom Hardware

Introduction

This chapter contains information on targeting custom hardware and implementing device driver blocks. By implementing your own blocks, you can create a library to include blocks for your particular I/O devices.

Typically, to target custom hardware, you must:

- Create a main program for your target system to execute the generated code.
- Create a system target file. This the entry point for the TLC program used to transform the models into generated code. The system target file can use the block target files and TLC function library provided by The MathWorks. Or you can create your own block target files and TLC function library.
- Create a template makefile to build your real-time executable.
- Write device drivers. Device drivers consist of a Simulink C MEX S-function and, optionally, a corresponding target file to inline the device instructions.

This chapter provides information on:

- Targeting custom hardware
- Creating a device driver block
- The basic structure of a device driver S-function
- How to implement operations performed by a device driver
- Compiling a device driver as a MEX-file
- Masking a device driver block

Device driver blocks can be implemented as S-functions. This chapter assumes that you are familiar with the Simulink C-MEX S-function format and API. See the Simulink documentation for more information on S-functions.

You can implement device driver blocks in two ways:

1. As C language S-functions (with no TLC files provided). This is referred to as a *noninlined* S-function.

2. As inlined S-functions (using TLC files).

Note For examples of device drivers, see the S-functions supplied in `*matlabroot*/rtw/c/tornado/devices` and `*matlabroot*/rtw/c/dos/devices`.

Run-Time Interface

There are two ways to target custom hardware:

- Rapid prototyping
- Embedded real-time

Rapid prototyping targets have an environment similar to the generic real-time or Tornado target. This section describes how to create a Run-time interface for a rapid prototyping target.

To create a run-time interface (i.e., a main program), you can begin with the generic real-time target (`grt`). The run-time interface for `grt` consists of:

- `grt_main.c` — located in *`matlabroot`*`/rtw/c/grt`
- `rt_sim.c` — located in *`matlabroot`*`/rtw/c/src`
- `ode1.c` - `ode5.c` — you must use one of these solvers if your model has continuous states. These are located in *`matlabroot`*`/rtw/c/src`.
- library routines for the model code — located in *`matlabroot`*`/rtw/c/libsrc`

For your run-time interface, you will need to copy and modify `grt_main.c`. The other modules do not require modification.

To create a new target called `mytarget`, start by creating a directory. For example, create

```
/applications/mytarget
```

and add it to your MATLAB path.

Copy `grt_main.c` to this directory and rename it `rt_main.c`. You will then need to modify `rt_main.c` to execute your model code. By default, `grt_main.c` is set up to execute your code in pseudomultitasking mode. If your target has a real-time operating system, you will need to modify `rt_main.c` to include calls for task creation and management. See Chapter 6, “Program Architecture,” and Chapter 7, “Models with Multiple Sample Rates,” for more details on the model code and how it is executed. The Tornado target is an example of a system that has a real-time operating system (see *`matlabroot`*`/rtw/c/tornado/rt_main.c` for the Tornado main module). Refer to Chapter 8, “Targeting Tornado for Real-Time Applications,” for detailed information about targeting Tornado.

Creating System Target Files and Template Makefiles

Assuming that you've created the directory

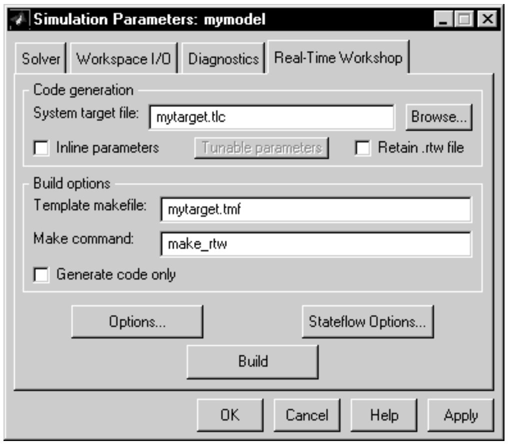
```
/applications/mytarget
```

you should copy *matlabroot*/rtw/c/grt/grt.tlc into it and rename grt.tlc to mytarget.tlc. You can then modify your system target file as needed, based on the requirements of your real-time target. You should modify or remove the first comment lines that define browser information.

Chapter 3, "Code Generation and the Build Process," provides information on how makefiles work. If your compiler and linker come with a make utility, then you can use it providing that it conforms to the "general" structure of most make utilities. If a make utility isn't provided with your compiler, you can use GNU make, which is located in rtw/bin/arch/make. When working with GNU make, you should use grt_unix.tmf as a starting point — even on PC platforms.

You should copy one of the *matlabroot*/rtw/c/grt/*.tmf template makefiles to /applications/mytarget and rename it mytarget.tlc. You will need to modify the template makefile to support your compiler, linker, and your version of make.

You can exercise mytarget.tlc and mytarget.tmf by creating a small Simulink fixed-step model. In the Real-Time Workshop page of the **Simulation Parameters** dialog box, you must set the **System target file** to mytarget.tlc and the **Template makefile** to mytarget.tmf:



System target file: mytarget.tlc

Inline parameters

Retain .rtw file

Template makefile: mytarget.tmf

Make command: make_rtw

Generate code only

Clicking the **Build** button causes the Real-Time Workshop to generate code as dictated by `mytarget.tlc` and compile it as dictated by `mytarget.tmf`.

Implementing Device Drivers

S-functions can be built into MEX-files or compiled and linked with other code to become part of a stand-alone program. This dual nature is exploited by the device driver blocks. These blocks are implemented as C code S-functions. They are compiled as MEX-files so you can place them in your Simulink block diagram. When you use the same model to build a real-time program, the source code for the device driver blocks is automatically compiled and linked with your program.

Additionally, the Real-Time Workshop uses Target Language Compiler technology that allows you to *inline* your S-function. By providing a customized TLC file containing the definition of your block, you can directly generate the code for your device driver. The advantage of inlining your S-function in this fashion is the elimination of function call overhead associated with the S-function interface.

It is helpful to examine existing device driver code in conjunction with this chapter. The source code for the blocks resides in the following directories:

- `matlabroot/rtw/c/dos/devices` directory for the source code to blocks in `doslib`.
- `matlabroot/rtw/c/tornado/devices` directory for the source code to blocks in `vxlib`.

There is also an S-function template that provides a useful starting point for the creation of any S-function. The file is

```
matlabroot/simulink/src/sfuntmpl.c
```

or

```
matlabroot/simulink/src/sfuntmpl.doc
```

Note This chapter does not discuss inlining of S-functions. Refer to “Inlining an S-function” in the *Target Language Compiler Reference Guide*.

Device drivers are implemented as C MEX S-functions. When you insert these blocks into your Simulink model, the code generator compiles and links the block’s source along with the source files used to build your program.

You can add your own blocks to a device driver library by writing S-functions that implement device drivers for your particular I/O board. However, in order for the code generator to generate code that can call your blocks, they must be implemented using the API defined by Simulink.

Before beginning the task of creating your own device driver block, you should consult the Simulink documentation for a description of how to write S-functions using the API.

Device Driver Blocks

A device driver block is:

- A device driver — software that handles communication between a real-time program and an I/O device. See your particular hardware documentation for information on its requirements.
- An S-function — a system function that interacts with Simulink during the simulation and is linked with the generated code. See the Simulink documentation for more information.
- A C MEX-file — a C subroutine that is dynamically linked to MATLAB. See the *MATLAB Application Program Interface Guide* for more information on MEX-files.
- A masked Simulink block — masked blocks allow you to define your own dialog box, icon, and initialization commands for the block. See the *Using Simulink* manual for more information on masking blocks.

The S-Function Format

The S-function API requires you to define specific functions and a `SimStruct` (a Simulink data structure). Like the generated code, the functions that implement the S-function are private to the source file. This means you can generate code for models having multiple instances of the same S-function.

Device driver S-functions are relatively simple to implement because they perform only a few operations. These operations include:

- Initializing the `SimStruct`
- Initializing the I/O device
- Calculating the block outputs according to the type of driver being implemented:

  - Reading values from an I/O device and assigning these values to the block's output vector `y` (if it is an ADC)
  - Writing values from the block's input vector `u`, to an I/O device (if it is a DAC)
- Terminating the program (e.g., zeroing the DAC outputs)

Required Functions

The S-function API requires you to define these functions:

- Functions called during initialization

  ```
  mdlInitializeSizes
  mdlInitializeSampleTimes
  mdlInitializeConditions
  ```

- Function called to calculate block outputs

  ```
  mdlOutputs
  ```

- Function called to reset the hardware when the program terminates

  ```
  mdlTerminate
  ```

The sections that follow describe how to implement these functions.

S-Function File Format

S-functions also require certain defined values and include files. The diagram on the next page illustrates the format of a device driver S-function.

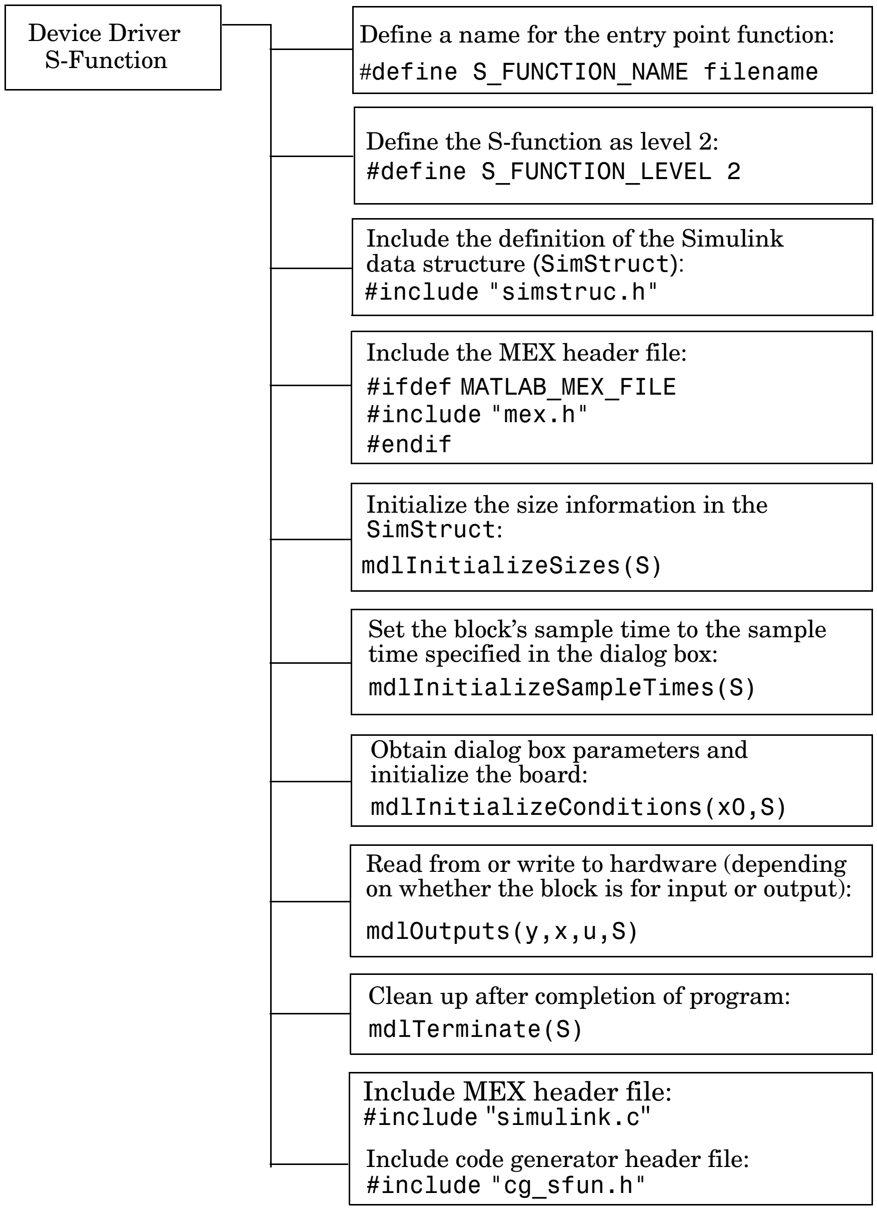

Device Driver S-Function
Define a name for the entry point function:
#define S_FUNCTION_NAME filename
Define the S-function as level 2:
#define S_FUNCTION_LEVEL 2
Include the definition of the Simulink data structure (SimStruct):
#include "simstruc.h"
Include the MEX header file:
#ifdef MATLAB_MEX_FILE
#include "mex.h"
#endif
Initialize the size information in the SimStruct:
mdlInitializeSizes(S)
Set the block's sample time to the sample time specified in the dialog box:
mdlInitializeSampleTimes(S)
Obtain dialog box parameters and initialize the board:
mdlInitializeConditions(x0,S)
Read from or write to hardware (depending on whether the block is for input or output):
mdlOutputs(y,x,u,S)
Clean up after completion of program:
mdlTerminate(S)
Include MEX header file:
#include "simulink.c"
Include code generator header file:
#include "cg_sfun.h"

Figure 10-1: Format of a Device Driver S-Function

S_FUNCTION_NAME Definition

The statement

```
#define S_FUNCTION_NAME name
```

defines the name of the function that is the entry point for the S-function code. This function is defined in `cg_sfun.h`, which is included at the end of the S-function. Note that `name` must be the name of the S-function file without the `.c` extension (e.g., `dt2811ad` for the Data Translation DT2811 A/D block). Also, you must specify the `S_FUNCTION_NAME` before you include `simstruc.h`.

Level 2 S-Function Definition

The statement

```
#define S_FUNCTION_LEVEL 2
```

defines the S-function as level 2. This allows you to take advantage of the full feature set included with S-functions, while maintaining backwards compatibility with older S-functions.

Defining the SimStruct

You must include the file `simstruc.h` to define the `SimStruct` (the Simulink data structure) and the `SimStruct` access macros:

```
#include "simstruc.h"
```

`simstruc.h` itself includes `rt_matrx.h`, which contains definitions for `mxGetPr` (and other matrix access macros). These `mx` macros are equivalent to those with the same name defined in MATLAB's Application Program Interface Library. They are used to obtain the parameters specified in the device driver's dialog box. See the *Application Program Interface Guide* for more information on matrix access macros.

This parallel definition of matrix access macros allows the same source code to be used as a MEX-file or as a source module that is compiled and linked with the generated code.

Conditional Compilations

In general, you can use an S-function in these environments:

- Simulink
- Real-Time

When you use your S-function in Simulink, you do so by creating a MEX-file from it. In this case, the macro `MATLAB_MEX_FILE` is defined.

When you use your S-function in real-time (for example, with a fixed-step solver and the Real-Time Workshop), the macro `RT` is defined.

Initialization

Initialization is performed in three separate steps. The S-function API requires you to implement three functions for initialization:

- `mdlInitializeSizes` — specifies the sizes of various parameters in the `SimStruct`.
- `mdlInitializeSampleTimes` — obtains the board's sample time from the dialog box and sets this value in the `SimStruct`.
- `mdlInitializeConditions` — reads values from the dialog box, initializes the board, and saves parameter values in the `SimStruct`.

Obtaining Values from Dialog Boxes

Obtaining the values of parameters from the block's dialog box requires you to:

1. Get the S-function input parameter (in this case the dialog box entry) using the `ssGetSFcnParam` macro.

2. Get the particular value from the input `mxArray` (since all input arguments to MEX-files are of type `Mxarray *`) using the `mxGetPr` function.

For example, suppose you want the value of the third item in the block's dialog box, which happens to be an integer. For convenience, you can define a macro that obtains the desired argument:

```
#define THIRD_ARGUMENT ssGetSFcnParam(S, 2)
```

Next define a variable used to store the integer (which is the number of channels in this example):

```
uint_T num_channels;
```

Finally, extract the value from the argument (which is of type `mxArray` *), and assign it to a type `uint_T` (unsigned integer). Since the parameter is a single integer, you need only the first element in the parameter:

```
num_channels = mxGetPr(THIRD_ARGUMENT)[0];
```

The macro `ssGetSFcnParam` is part of the S-function API. It is defined in `simstruc.h` and described in the *Using Simulink* manual.

The function `mxGetPr` is part of the External Interface Library. Its function prototype is in `mex.h` and rt_matrx.h. It is described in the *Application Program Interface Guide*.

Initializing Sizes — Input Devices

The `mdlInitializeSizes` function sets size information in the `SimStruct`. For example, the following definition of `mdlInitializeSizes` initializes a typical ADC (analog to digital converter) board:

```
static void mdlInitializeSizes(SimStruct *S)
{
  uint_T num_channels = mxGetPr(THIRD_ARGUMENT)[0];
  ssSetNumSFcnParams(S, 3);  /* Number of expected parameters */
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
  /* Return if number of expected != number of actual param's */
    return;
  }
  ssSetNumInputPorts(S, 0);
  ssSetNumOutputPorts(S, num_channels);
  ssSetNumSampleTimes(S, 1);
  {
    uint_T i;
    for(i=0; i<num_channels;i++){
      ssSetInputPortdirectfeedthrough(S,i,0);
    }
  }
}
```

This list describes the function of each macro in `mdlInitializeSizes`:

- `ssSetNumSFcnParams` — The number of input parameters is equal to the number of parameters in the block's dialog box.
- `ssSetNumInputPorts` — The ADC block has no inputs because it reads data from the I/O board. An ADC is a *source* block (i.e., the block has only output ports).
- `ssSetNumOutputPorts` — The number of outputs equals the number of I/O channels. The code above obtains the number of channels from the dialog box, which is the first argument passed to the S-function (i.e., the first item in the dialog box).
- `ssSetInputPortDirectFeedThrough` — It is important to note that the ADC has no direct feedthrough. The ADC's output is calculated based on values obtained external to the Simulink model, not from the input of another block. In fact, the ADC has no inputs; it calculates its output based on values obtained from the I/O board. This information affects the organization of the generated code.
- `ssSetNumSampleTimes` — Assuming that all channels run at the same rate, there is only one sample time (the board's sample time specified in its dialog box).

Initializing Sizes — Output Devices

Initializing size information for an output device, such as a DAC (digital to analog converter) has two important differences from the ADC:

- Since the DAC obtains its inputs from other blocks, the number of channels is equal to the number of inputs (instead of number of outputs as is the case with the ADC). This is because the DAC is a *sink* block (i.e., the block has only input ports and it writes data to something that is external to the Simulink model — the I/O device).
- Also notice that this block has direct feedthrough (i.e., the DAC cannot execute until the block feeding it updates its outputs). This information affects the organization of the generated code.

The following example illustrates the definition of `mdlInitializeSizes` for a DAC:

```
static void mdlInitializeSizes(SimStruct *S)
{
  uint_T num_channels = mxGetPr(ssGetSFcnParam(S,0)[0]);
  uint_T num_sample_times = mxGetPr(ssGetSFcnParams(S,1))[0])
  ssSetNumSFcnParams(S, 3);  /* Number of expected parameters */
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
    /* Return if number of expected != number of actual param's */
    return;
  }
  ssSetNumInputPorts(S, num_channels);  /*Number of inputs is now
                                          the number of
                                          channels. */
  ssSetNumOutputPorts(S, 0);
  {
    uint_T i;
    for(i=0, i < num_channels, i++){
      ssSetInputPortDirectFeedThrough(S,i,1);
    }
  }
  ssSetNumSampleTimes(S, num_sample_times);
}
```

Initializing Sample Times

Device driver blocks are discrete blocks that require you to set a sample time. Assuming that all channels run at the same rate, the S-function has only one sample time — the sample time specified in the dialog box.

The following definition of `mdlInitializeSampleTimes` reads the sample time from the block's dialog box. In this case, sample time is the second parameter in the dialog box:

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
  ssSetSampleTime(S, 0, mxGetPr(ssGetSFcnParams(S,1))[0]);
  ssSetOffsetTime(S, 0, 0.0);
}
```

Initializing the I/O Device

Device driver S-functions use the `mdlInitializeConditions` function to:

- Read parameters from the block's dialog box
- Save parameter values in the `SimStruct`
- Initialize the I/O device

When reading parameter values, you must be careful to handle data types correctly. The next two examples illustrate how to read two different types of data from the dialog box.

Reading the Base Address. The following code reads the base address parameter from the dialog box. While this parameter is actually a hexadecimal number, it is specified as a character string and is ultimately stored as an integer:

```
static void mdlInitializeConditions(SimStruct *S)
{
  int base_addr_str_len = 128;
  char base_addr_str[128];
  mxGetString(ssGetSFcnParams(S, 0), base_addr_str,
  base_addr_str_len);
  sscanf(base_addr_str, "%lx", &base_addr);
}
```

In this example, the base address is the first argument in the dialog box. You obtain the parameter with `ssGetSFcnParam` and then use `mxGetString` to convert the argument (which, like all MEX-file arguments, is actually a type `mxArray *`) to a string. The parameter is passed as a string because you cannot specify hexadecimal numbers in the dialog box.

The conversion from string to hexadecimal is accomplished by `sscanf`, which returns the integer equivalent of the hexadecimal number.

Calculating Block Outputs

The basic purpose of a device driver block is to allow your program to communicate with I/O hardware. Typically, you can accomplish this using low level hardware calls that are part of your compiler's C library or using C-callable functions provided with your I/O hardware. This section provides examples of both techniques.

All S-functions call the `mdlOutputs` function to calculate block outputs. For a device driver block, this function contains the code that reads from or writes to the hardware.

Accessing the Hardware

The mechanism you use to access the I/O device depends on your particular implementation. One possibility is to use low level hardware calls that are part of your compiler's C library. For example, on the PC a device driver could use the following technique. These S-functions define access routines for both of the supported compilers:

```
/* compiler dependent low level hardware calls */
#include <conio.h>
#define hw_outportb(portid, value) _outp(portid, value)
#define hw_inportb(portid)         _inp(portid)
#define hw_outport(portid, value)  _outpw(portid, value)
#define hw_inport(portid )         _inpw(portid)
```

Another technique is to use C callable functions provided with the I/O hardware.

ADC Outputs

In the case of an ADC, `mdlOutputs` must:

- Initiate a conversion for each channel.
- Read the board's A/D converter output for each channel (and perhaps apply scaling to the values read).
- Set these values in the output vector `y` for use by the model.

DAC Outputs

In the case of a DAC, `mdlOutputs` must:

- Read the input `u` from the upstream block.
- Set the board's D/A converter output for each channel (and apply scaling to the input values if necessary).
- Initiate a conversion for each channel.

For examples of device drivers, see the S-functions supplied in *`matlabroot`*`/rtw/c/tornado/devices` and *`matlabroot`*`/rtw/c/dos/devices`.

The Termination Function

The final required function is typically used only in DACs to zero the output at the end of the program. For example:

```
static void mdlTerminate(SimStruct *S)
{
uint_T num_channels = (uint_t)mxGetPr(ssGetSFcnParams(S,0)[0]);
uint_T i;
for (i = 0; i < num_channels; i++) {
  ds1102_da(i + 1, 0.0);
  }
}
```

This `for` loop simply sets the output of each channel to 0.0. ADCs typically implement this function as an empty stub.

Additional Include Files

The code implementing a device driver block must serve as both MEX-file and as stand-alone code module that can be compiled and linked with the generated code. This code is used as a MEX-file to enable Simulink to obtain information for performing consistency checks during the code generation process.

Each of these applications of the device driver S-function requires its own include file. The two files are `simulink.c` for MEX-files and `cg_sfun.h` for the generated code.

MEX-files are built with the `mex` command. `mex` defines the string `MATLAB_MEX_FILE` to provide a mechanism that allows you to include certain parts of your code in the MEX-file and other parts in the generated code. To include the proper files, place the following statements at the end of your S-function:

```
#ifdef MATLAB_MEX_FILE /* Is this a MEX-file? */
#include "simulink.c" /* MEX-file include file */
#else
#include "cg_sfun.h" /* Generated code include file */
#endif
```

The S-function template file `*matlabroot*/simulink/src/sfuntmpl.c` contains these statements.

The Public Registration Function

The include file `cg_sfun.h` defines a function that is the entry point for the S-function code. This function is named by the

```
#define S_FUNCTION_NAME name
```

macro that you specified earlier in your code.

This function registers the other local functions by installing pointers to them in the `SimStruct`. This is the only public function in the S-function and is called by the real-time program's public registration function during program startup. The remainder of the S-function code is then accessed via these pointers. This scheme produces re-entrant code allowing multiple instances of an S-function within a single program.

The `S_FUNCTION_NAME` function always has the same definition and therefore is most simply implemented by including `cg_sfun.h` at the bottom of your S-function.

Compiling the MEX-File

See the *MATLAB Application Program Interface Guide* for information on how to use `mex` to compile the device driver S-function into an executable MEX-file.

Note that many I/O boards supply include files as part of their software support package. If your device driver code includes such files, you must ensure that these files are available when you build your real-time program, as well as when you compile the device drivers as MEX-files.

Converting the S-Function to a Block

Once you have compiled the S-function into an executable MEX-file, you can convert it to a block and then mask the block to create its own dialog box. To do this, follow these two steps:

- Place the S-function block into a subsystem block
- Mask the Subsystem block

See "Creating Subsystems" and "Using Masks to Customize Blocks" in *Using Simulink* for more information about subsystems and masking.

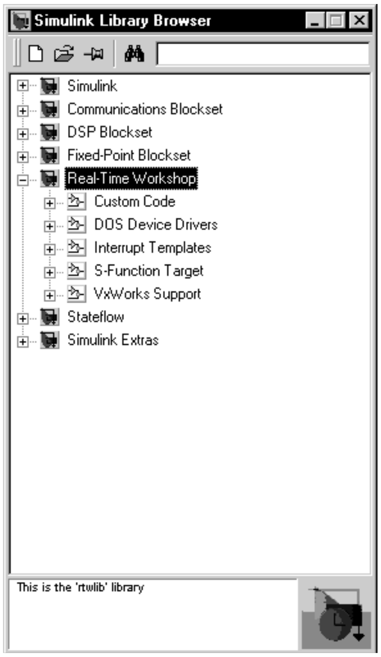
Setting the MATLAB Path

The device driver blocks in a library can automatically change your MATLAB path to ensure that Simulink can find the MEX-file. This is accomplished by calling the `addpath` M-file as part of the masked block's initialization command. This command is executed whenever you start a simulation or generate code.

Real-Time Workshop Libraries

Introduction

The Real-Time Workshop provides a library of functions that allow you great flexibility in constructing real-time system models and generated code. The Real-Time Workshop Library is a collection of sublibraries located in the Simulink Library Browser.



Note that, depending on which MathWorks products you have installed, your browser may show a different collection of libraries.

There are five sublibraries in the Real-Time Workshop library:

- Custom Code Library — A collection of blocks that allow you to insert custom code into the generated source code files and/or functions associated with your model.
- DOS Device Drivers — Blocks for use with DOS. See “Targeting DOS for Real-Time Applications” in Chapter 9 for a discussion of targeting DOS.
- Interrupt Templates — A collection of blocks that you can use as templates for building your own asynchronous interrupts.

- S-Function Target — This block is intended for use in conjunction with the Real Time W S-function code format. See “S-Function Code Format” on page 12-18 for more information.
- VxWorks Support — A collection of blocks that support VxWorks (Tornado). See “Targeting Tornado for Real-Time Applications” in Chapter 8 for information on VxWorks.

Note This chapter discusses asynchronous interrupt handling. For information about device drivers, refer to Chapters 8, 9, and 10, which discuss VxWorks, DOS, and custom device drivers respectively.

Custom Code Library

The Real-Time Workshop also provides a Custom Code library containing blocks that allow you to place your own code, in C or in Ada, inside the code generated by the Real-Time Workshop. There are two Custom Code sublibraries in both the C and Ada Custom Code libraries:

- Custom Model Code
- Custom Subsystem Code

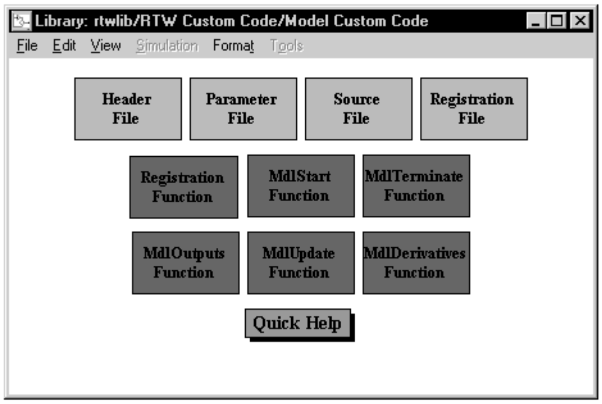
Both sublibraries contain blocks that target specific files and subsystems within which you can place your code.

Custom Model Code

The Custom Model Code sublibrary contains 10 blocks that insert custom code into the generated model files and functions. You can view the blocks either by:

- Expanding the Model Code library (under C Custom Code) in the Simulink Browser
- Right clicking your mouse on the Model Code library in the Simulink browser

The latter method opens this window.



The four blocks on the top row contain texts fields to insert custom code at the top and bottom of the following files:

- `model.h` — Header File block
- `model.prm` — Parameter File block
- `model.c` — Source File block
- `model.reg` — Registration File block

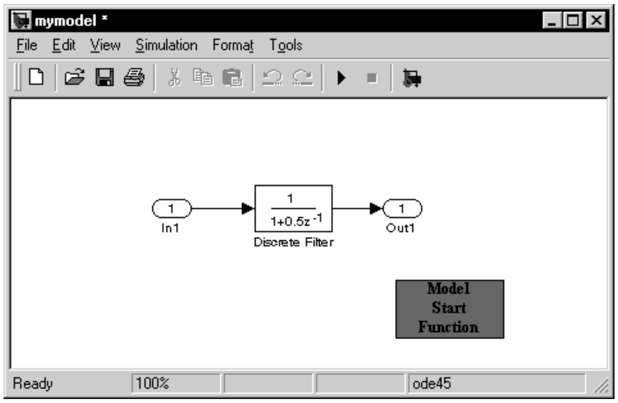
The six function blocks in the second and third rows contain text fields to insert critical code sections at the top and bottom of these designated model functions:

- `Registration function` — Registration Function block
- `MdlStart` — MdlStart Function block
- `MdlTerminate` — MdlTerminate Function block
- `MdlOutputs` — MdlOutputs Function block
- `MdlUpdate` — MdlUpdate Function block
- `MdlDerivatives` — MdlDerivatives Function block

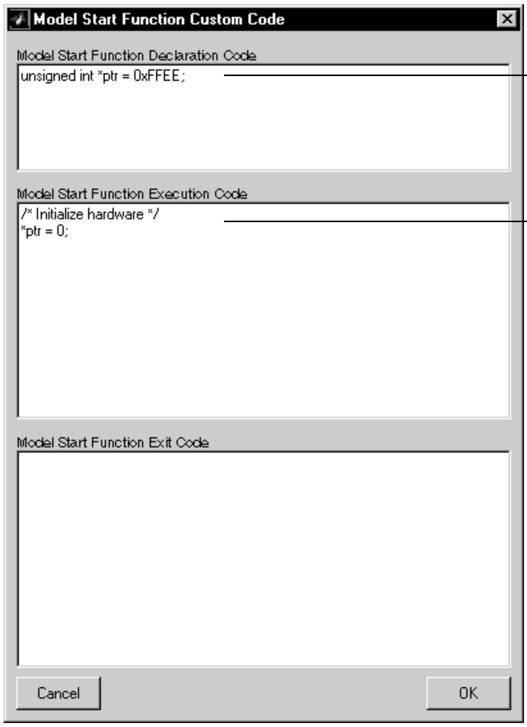
Each block provides a dialog box that contains three fields.

Example of How to Use a Custom Code Block

The following example uses an MdlStart Function block to introduce code into the `MdlStart` function. The diagram below shows a simple model with the Custom Code block inserted.



Double clicking the Model Start Function block opens a the **MdlStart Function Custom Code** dialog box.


Model Start Function Custom Code
Model Start Function Declaration Code
Model Start Function Execution Code
Model Start Function Exit Code
Cancel
OK

```
unsigned int *ptr = 0xFFEE;
```

```
/* Initialize hardware */
*ptr = 0;
```

The Real-Time Workshop inserts the code entered here into the `MdlStart` function in the generated code.

You can insert custom code into any or all of the available text fields.

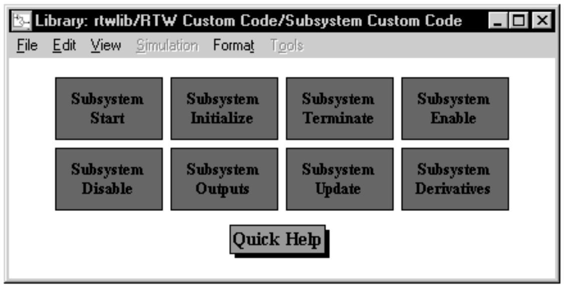
The code below is the `MdlStart` function for this example (`mymodel`).

```
void MdlStart(void)
{
  /* user code (Start function Header) */
  /* System: <Root> */
  unsigned int *ptr = 0xFFEE;

  /* user code (Start function Body) */
  /* System: <Root> */
  /* Initialize hardware */
  *ptr = 0;

  /* state initialization */
  /* DiscreteFilter Block: <Root>/Discrete Filter */
  rtX.d.Discrete_Filter = 0.0;
}
```

The custom code entered in the MdlStart Function Custom Code dialog box is embedded directly in the generated code.

Custom Subsystem Code

The Custom Subsystem Code sublibrary contains eight blocks to insert critical code sections into system functions:



Each of these blocks has a dialog box containing two text fields that allow you to place data at the top and the bottom of system functions. The eight blocks are:

- Subsystem Start
- Subsystem Initialize
- Subsystem Terminate

- Subsystem Enable
- Subsystem Disable
- Subsystem Outputs
- Subsystem Update
- Subsystem Derivatives

The location of the block in your model determines the location of the custom code. In other words, the code is local to the subsystem that you select. For example, the Subsystem Outputs block places code in `mdlOutputs` when the code block resides in the root model, but the code is placed in the system's outputs function when it resides in an enabled subsystem.

The ordering for a triggered system is:

1. Output entry
2. Output exit
3. Update entry
4. Update exit code

Interrupt Handling

The Real-Time Workshop provides blocks in the Interrupt Template library that allow you to model synchronous/asynchronous event handling, including interrupt service routines (ISRs). These blocks include:

- Asynchronous Interrupt block
- Task Synchronization block
- Asynchronous Buffer block (read)
- Asynchronous Buffer block (write)
- Asynchronous Rate Transition block

Using these blocks, you can create models that handle asynchronous events, such as hardware generated interrupts and asynchronous read and write operations. This chapter discusses each of these blocks in the context of VxWorks Tornado operating system.

Asynchronous Interrupt Block

Interrupt service routines (ISR) are realized by connecting the outputs of the VxWorks Asynchronous Interrupt block to the control input of a function-call subsystem, the input of a VxWorks Task Synchronization block, or the input to a Stateflow chart configured for a function-call input event.

The Asynchronous Interrupt block installs the downstream (destination) function-call subsystem as an ISR and enables the specified interrupt level. The current implementation of the VxWorks Asynchronous Interrupt block supports VME interrupts 1-7 and uses the VxWorks system calls `sysIntEnable`, `sysIntDisable`, `intConnect`, `intLock` and `intUnlock`. Ensure that your target architecture (BSP) for VxWorks supports these functions.

When a function-call subsystem is connected to an Asynchronous Interrupt block output, the generated code for that subsystem becomes the ISR. For large subsystems, this can have a large impact on interrupt response time for interrupts of equal and lower priority in the system. As a general rule, it is best to keep ISRs as short as possible, and thus you should only connect function-call subsystems that contain few blocks. A better solution for large systems is to use the Task Synchronization block to synchronize the execution of the function-call subsystem to an event. The Task Synchronization block is placed between the Asynchronous Interrupt block and the function-call subsystem (or Stateflow Chart). The Asynchronous Interrupt block then installs the Task Synchronization block as the ISR, which releases a synchronization semaphore (performs a `semGive`) to the function-call subsystem and returns. See the VxWorks Task Synchronization block for more information.

Using the Asynchronous Interrupt Block

The Asynchronous Interrupt block has two modes: RTW and Simulation;

- In RTW mode, the Asynchronous Interrupt block configures the downstream system as an ISR and enables interrupts during model startup. You can select this mode using the Asynchronous Interrupt block dialog box when generating code.
- In Simulation mode, simulated Interrupt Request (IRQ) signals are routed through the Asynchronous Interrupt block's trigger port. Upon receiving a simulated interrupt, the block calls the associated system.

  You should select this mode when simulating, in Simulink, the effects of an interrupt signal. Note that there can only be one VxWorks Asynchronous Interrupt block in a model and all desired interrupts should be configured by it.

In both RTW and Simulation mode, in the event that two IRQ signals occur simultaneously, the Asynchronous Interrupt block executes the downstream systems according to their priority interrupt level.

The Asynchronous Interrupt block provides these two modes to make the development and implementation of real-time systems that include ISRs easier and quicker. You can develop two models, one that includes a plant and a controller for simulation, and one that only includes the controller for code generation. Using the Library feature of Simulink, you can implement changes to both models simultaneously. This picture illustrates the concept:

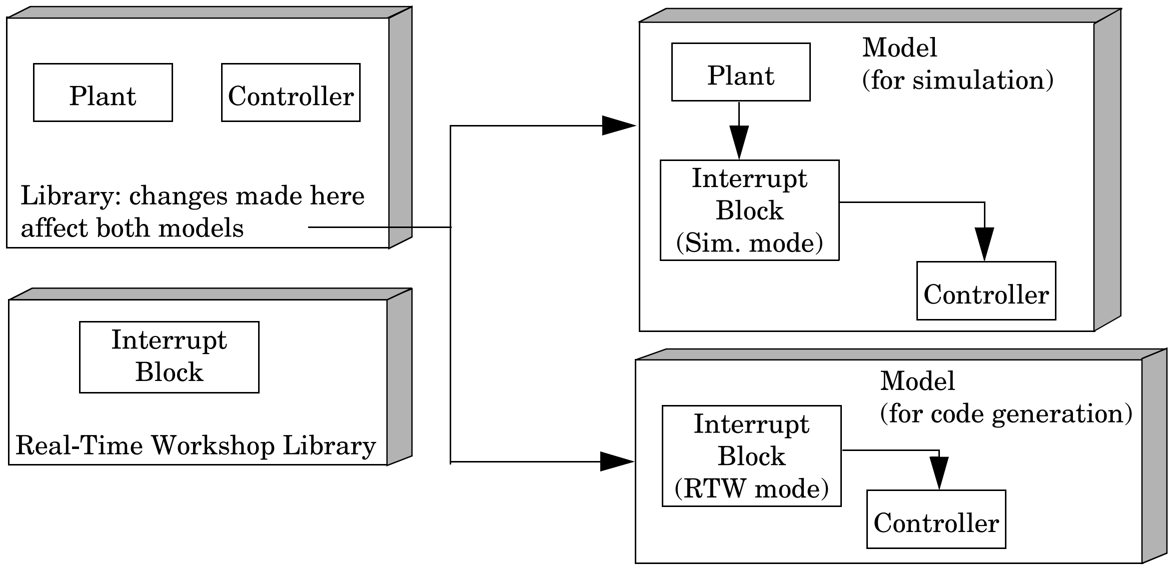

Plant
Controller
Library: changes made here affect both models
Interrupt Block
Real-Time Workshop Library
Model (for simulation)
Plant
Interrupt Block (Sim. mode)
Controller
Model (for code generation)
Interrupt Block (RTW mode)
Controller

Figure 11-1: Using the Asynchronous Interrupt Block with Simulink's Library Feature in the Rapid Prototyping Process

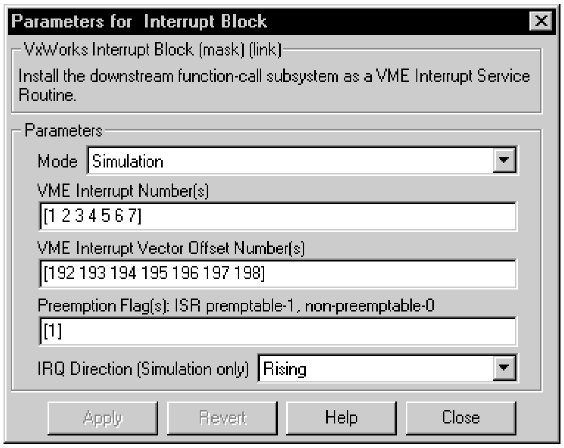
By supporting two modes in the Asynchronous Interrupt block, the Real-Time Workshop provides another tool for use in rapid prototyping. For more information on rapid prototyping, see Chapter 1. To learn more about the Library feature in Simulink, see *Using Simulink*.

Real-Time Workshop models normally run from a periodic interrupt. All blocks in a model run at their desired rate by executing them in multiples of the timer interrupt rate. Asynchronous blocks, on the other hand, execute based on other interrupt(s) that may or may not be periodic.

The hardware that generates the interrupt is not configured by the Asynchronous Interrupt block. Typically, the interrupt source is a VME I/O board, which generates interrupts for specific events (e.g., end of A/D conversion). The VME interrupt level and vector are set up in registers or by using jumpers on the board. The `mdlStart` routine of a user-written device driver (S-function) can be used to set up the registers and enable interrupt generation on the board. You must match the interrupt level and vector specified in the Asynchronous Interrupt block dialog to the level and vector setup on the I/O board.

Asynchronous Interrupt Block Parameters

The picture below shows the VxWorks Asynchronous Interrupt block dialog box:


Parameters for Interrupt Block

VxWorks Interrupt Block (mask) (link)

Install the downstream function-call subsystem as a VME Interrupt Service Routine.

Parameters

Mode: Simulation

VME Interrupt Number(s): [1 2 3 4 5 6 7]

VME Interrupt Vector Offset Number(s): [192 193 194 195 196 197 198]

Preemption Flag(s): ISR premptable-1, non-preemptable-0: [1]

IRQ Direction (Simulation only): Rising

Parameters associated with the Asynchronous Interrupt block are:

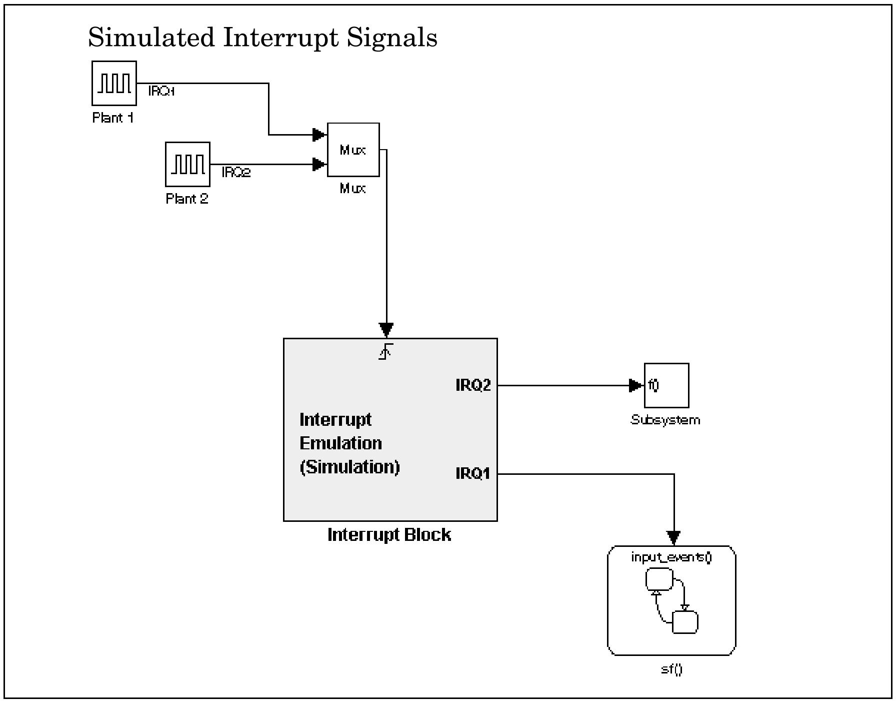
- Mode — in Simulation mode, the ISRs are executed nonpreemptively. If they occur simultaneously, signals are executed in the order specified by their number (1 being the highest priority). Interrupt mapping during simulation is left to right, top to bottom. That is, the first control input signal maps to the topmost ISR. The last control input signal maps to the bottom most ISR.

  In RTW mode, the Real-Time Workshop uses `vxinterrupt.tlc` to realize asynchronous interrupts in the generated code. The ISR is passed one argument, the root `SimStruct`, and the Simulink definition of the function-call subsystem is remapped to conform with the information in the `SimStruct`.
- VME Interrupt Number(s) — Specify the VME interrupt numbers for the interrupts to be installed. The valid range is 1-7; for example: `[4 2 5]`).

- VME Interrupt Offset Number(s) — the Real-Time Workshop uses this number in the call to `intConnect(INUM_TO_IVEC(#),...)`. You should specify a unique vector offset number for each interrupt number.
- Preemption Flag(s) — By default, higher priority interrupts can preempt lower priority interrupts in VxWorks. If desired, you can lock out interrupts during the execution of a ISR by setting the preemption flag to 0. This causes `intLock()` and `intUnlock()` calls to be inserted at the beginning and end of the ISR respectively. This should be used carefully since it increases the systems interrupt response time for all interrupts at the `intLockLevelSet()` and below.
- IRQ Direction — In simulation mode, a scalar IRQ direction is applied to all control inputs, and is specified as 1 (rising), -1 (falling), or 0 (either). Configuring inputs separately in simulation is done prior to the control input. For example, a Gain block set to -1 prior to a specific IRQ input will change the behavior of one control input relative to another. In RTW mode the IRQ direction parameter is ignored.

Asynchronous Interrupt Block Example - Simulation Mode

This example shows how the Asynchronous Interrupt block works in simulation mode:

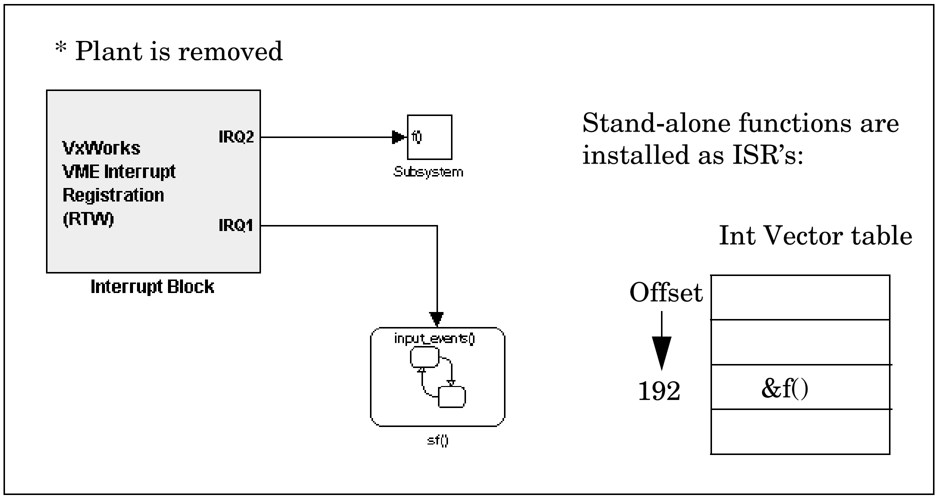


The Asynchronous Interrupt block works as a “handler” that routes signals and sets priority. If two interrupts occur simultaneously, the rule for handling which signal is sent to which port is left/right and top/bottom. This means that IRQ2 receives the signal from plant 1 and IRQ1 receives the signal from plant 2 simultaneously. IRQ1 still has priority over IRQ2 in this situation.

Note that the Asynchronous Interrupt block executes during simulation by processing incoming signals and executing downstream functions. Also, interrupt preemption cannot be simulated.

Asynchronous Interrupt Block Example - RTW Mode

This example shows the Asynchronous Interrupt block in RTW mode:



Note that the simulated plant signals that were included in the previous example have been omitted. In RTW mode, the Asynchronous Interrupt block receives interrupts directly from the hardware.

During the Target Language Compiler (TLC) phase of code generation, the Asynchronous Interrupt block installs the code in the Stateflow Chart and the Subsystem block as interrupt service routines. Configuring a function-call subsystem as an ISR requires two function calls, `int_connect` and `int_enable`. For example, the function `f(u)` in the Function block requires this procedure:

- In the `mdlStart` function, the Asynchronous Interrupt block inserts a call to `int_connect` and `sysIntEnable`:

```
/* model start function */
MdlStart()
{
  . . .
  int_connect(f,192,1);
  . . .
  sysIntEnable(1);
  . . .

}
```

Locking and Unlocking ISRs. It is possible to lock ISRs so that they are not preempted by a higher priority interrupt. Configuring the interrupt as nonpreemptive has this effect. This code shows where the Real-Time Workshop places the `int_lock` and `int_unlock` functions to configure the interrupt as nonpreemptive:

```
f()

{
  lock = int_lock();    }
  . . .                 }  Real-Time Workshop code
  . . .                 }
  . . .
  int_unlock(lock);
}
```

Finally, the model's terminate function disables the interrupt:

```
/* model terminate function */
MdlTerminate()
{
  ...
  int_disable(1);
  ...
}
```

Task Synchronization Block

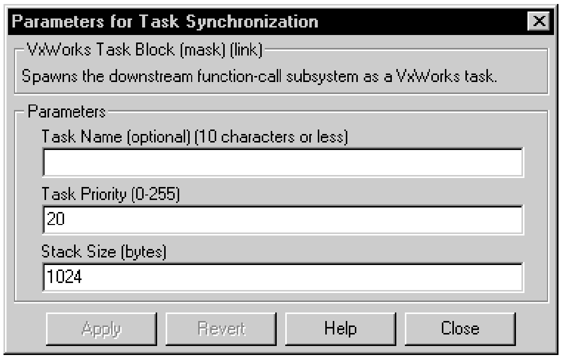
The VxWorks Task Synchronization block is a function-call subsystem that spawns, as an independent VxWorks task, the function-call subsystem connected to its output. It is meant to be placed between two other function-call subsytems. Typically it would be placed between the VxWorks Asynchronous Interrupt block and a function-call subsystem block or a Stateflow chart. Another example would be to place the Task Synchronization block at the output of a Stateflow diagram that has an Event "Output to Simulink" configured as a function-call. The VxWorks Task Synchronization block performs the following functions:

- The downstream function-call subsystem is spawned as an independent task using the VxWorks system call `taskSpawn()`. The task is deleted using `taskDelete()` during model termination.

- A semaphore is created to synchronize the downstream system to the execution of the Task Synchronization block.
- Code is added to this spawned function-call subsystem to wrap it in an infinite while loop.
- Code is added to the top of the infinite while loop of the spawned task to wait on a the semaphore, using `semTake()`. `semTake()` is first called with `NO_WAIT` specified. This allows the task to determine if a second `semGive()` has occurred prior to the completion of the function-call subsystem. This would indicate the interrupt rate is too fast or the task priority is too low.
- Synchronization code, i.e., `semgive()`, is generated for the Task Synchronization block (a masked function-call subsystem). This allows the output function-call subsystem to run. As an example, if you connect the Task Synchronization block to the output of a VxWorks Asynchronous Interrupt block, the `semGive()` would occur inside an ISR.

Task Synchronization Parameters

The picture below shows the VxWorks Task Synchronization block dialog box:



Parameters for Task Synchronization

VxWorks Task Block (mask) (link)

Spawns the downstream function-call subsystem as a VxWorks task.

Parameters

Task Name (optional) (10 characters or less):

Task Priority (0-255): 20

Stack Size (bytes): 1024

Parameters associated with the Task Synchronization block are:

- Task Name — An optional name, which if provided, is used as the first argument to the `taskSpawn()` system call. This name is used by VxWork routines to identify the task they are called from to aid in debugging.
- Task Priority — The task priority is the VxWorks priority that the function-call subsystem task is given when it is spawned. The priority can be a very important consideration in relation to other tasks priorities in the

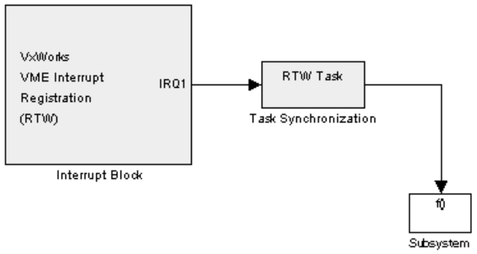
VxWorks system. In particular, the default priority of the model code is 30 and, when multitasking is enabled, the priority of the each subrate task increases by one from the default model base rate. Other task priorities in the system should also be considered when choosing a task priority. VxWorks priorities range from 0 to 255 where a lower number is a higher priority.

- Stack Size — The function-call subsystem is spawned with the stack size specified. This is maximum size to which the task's stack can grow. The value should be chosen based on the number of local variables in the task.

  By default, Real-Time Workshop limits the number of bytes for local variables in all of the generated code to 8192 bytes (see `matlabroot/rtw/c/tornado/tornado.tlc`). As a rule, providing twice 8192 bytes, 16384, for the one function that is being spawned as a task should be sufficient.

Task Synchronization Block Example

This example shows a Task Synchronization block as a simple ISR:



The Task Synchronization block inserts this code during the TLC phase of code generation:

- In `MdlStart`, the Task Synchronization block is registered by the Asynchronous Interrupt block as an ISR. The Task Synchronization block creates and initializes the synchronization semaphore. It also spawns the function-call subsystem as an independent task.

```
/* Create and spawn task: <Root>/Faster Rate(.015) */
if ((*(SEM_ID *)rtPWork.s6_S_Function.SemID =
  semBCreate(SEM_Q_PRIORITY, SEM_EMPTY)) == NULL)
  ssSetErrorStatus(rtS,"semBCreate call failed "
                   "for block <Root>/Faster Rate(.015).\n ");
}
if ((rtIWork.s6_S_Function.TaskID = taskSpawn("root_Faster_", 20, VX_FP_TASK,
    1024, (FUNCPTR)Sys_root_Faster__OutputUpdate,
    (int_T)rtS, 0, 0, 0, 0, 0, 0, 0, 0, 0)) == ERROR) {
      ssSetErrorStatus(rtS,"taskSpawn call failed for block <Root>/ Faster Rate "
                            "(.015).\n");
  }
```

- The Task Synchronization block modifies the downstream function-call subsystem by wrapping it inside an infinite loop and adding semaphore synchronization code:

```
/* Output and update for function-call system: <Root>/Faster    Rate(.015) */
void Sys_root_Faster__OutputUpdate(void *reserved, int_T
                                   controlPortIdx, int_T tid)
{
  /* Wait for semaphore to be released by system: <Root>/Task Synchronization */
  for(;;) {
    if (semTake(*(SEM_ID *)rtPWork.s6_S_Function.SemID,NO_WAIT) != ERROR) {
      logMsg("Rate for function-call subsystem"
             "Sys_root_Faster__OutputUpdate() fast.\n",0,0,0,0,0,0);
#if STOPONOVERRUN
      logMsg("Aborting real-time simulation.\n",0,0,0,0,0,0);
      semGive(stopSem);
      return(ERROR);
#endif
    } else {

      semTake(*(SEM_ID *)rtPWork.s6_S_Function.SemID, WAIT_FOREVER);
    }
    /* UniformRandomNumber Block: <S3>/Uniform Random Number */
    rtB.s3_Uniform_Random_Number =
    rtRWork.s3_Uniform_Random_Number.NextOutput;
   .
   .
   .
}
```

Asynchronous Buffer Block

The VxWorks Double Buffer blocks are meant to be used to interface signals to asynchronous function-call subsystems in a model. This is needed whenever a function-call subsystem has input or output signals and its control input ultimately connects (sources) to the VxWorks Asynchronous Interrupt block or Task Synchronization block.

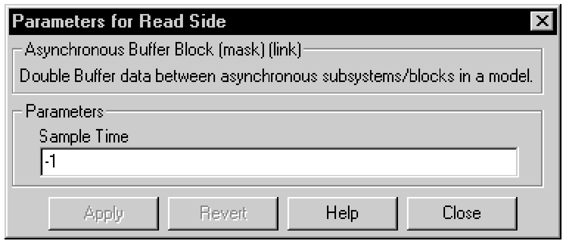
Because an asynchronous function-call subsystem can preempt or be preempted by other model code, an inconsistency arises when more than one signal element is connected to it. The issue is that signals passed to and/or from the function-call subsystem can be in the process of being written or read when the preemption occurs. Thus, partial old and partial new data will be used. The Double Buffer blocks can be used to guarantee the data passed to and/or from the function-call subsystem is all from the same iteration.

The Double Buffer blocks are used in pairs, with a write side driving the read side. To ensure the data integrity, no other connections are allowed between the two Double Buffer blocks. The pair works by using two buffers ("double buffering") to pass the signal and, by using mutually exclusive control, allow only exclusive access to each buffer. For example, if the write side is currently writing into one buffer, the read side can only read from the other buffer.

The initial buffer is filled with zeros so that if the read side executes before the write side has had time to fill the other buffer, the read side will collect zeros from the initial buffer.

Asynchronous Buffer Block Parameters

There are two kinds of Asynchronous Buffer blocks, a reader and a writer. The picture below shows the Asynchronous Buffer block's dialog boxes (reader and writer):



Parameters for Read Side

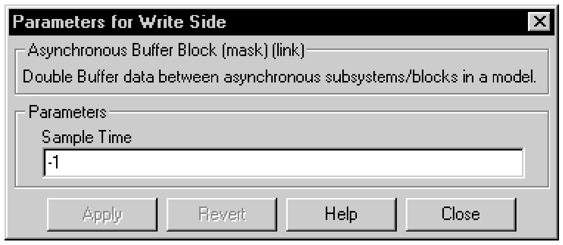
Asynchronous Buffer Block (mask) (link)

Double Buffer data between asynchronous subsystems/blocks in a model.

Parameters

Sample Time

-1



Parameters for Write Side

Asynchronous Buffer Block (mask) (link)

Double Buffer data between asynchronous subsystems/blocks in a model.

Parameters

Sample Time

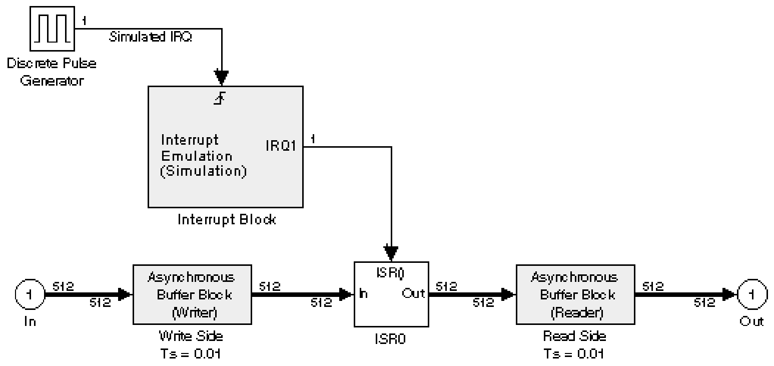
-1

Both blocks require the same parameter:

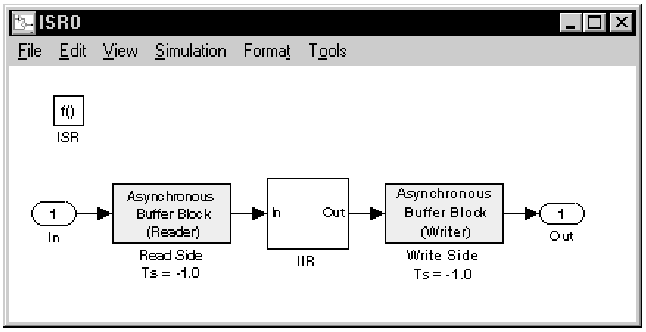
- Sample Time — The sample time should be set to -1 inside a function call and to the desired time otherwise.

Asynchronous Buffer Block Example

This example shows how you might use the Asynchronous Buffer block to control the data flow in an interrupt service routine:



The ISR() subsystem block, which is configured as a function-call subsystem, contains another set of Asynchronous Buffer blocks:


ISR0
File Edit View Simulation Format Tools
f()
ISR
In
Asynchronous Buffer Block (Reader)
Read Side Ts = -1.0
In
Out
IIR
Asynchronous Buffer Block (Writer)
Write Side Ts = -1.0
Out

Rate Transition Block

The VxWorks Rate Transition block is used to provide a sample time for blocks connected to an asynchronous function-call subsystem when double buffering is not required. There are three options for connecting I/O to the Asynchronous block:

- Use the Rate Transition block, or some other block that requires a sample time to be set, at the output of the asynchronous function-call subsystem.

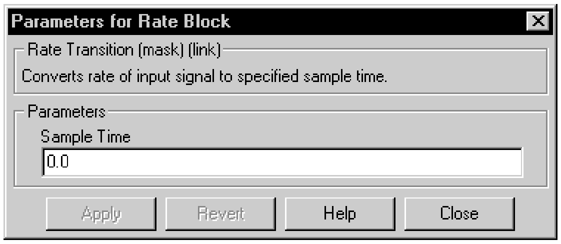
This will cause blocks up- or downstream from it, which would otherwise inherit from the function-call subsystem, to use the sample time specified. Note that if the signal width is greater than 1, data consistency is not guaranteed, which may or may not an issue, see next option.

The Rate Transition block does not introduce any system delay. It only specifies the sample time of the downstream blocks. This block is typically used for scalar signals that do not require double buffering.

- Use the Double Buffer block pair. This not only will set the sample time of the blocks up or downstream that would otherwise inherit from the function-call subsystem, and also guarantees consistency of the data on the signal. See the Double Buffer block for more information on data consistency.
- The third option is to allow the blocks to inherit their sample time. The inherited sample time can be seen by turning on sample time colors in Simulink. If you achieve the required sample times, no further action is required.

Rate Transition Block Parameters

This picture shows the VxWorks Rate Transition block's dialog box:


Parameters for Rate Block

Rate Transition (mask) (link)

Converts rate of input signal to specified sample time.

Parameters

Sample Time

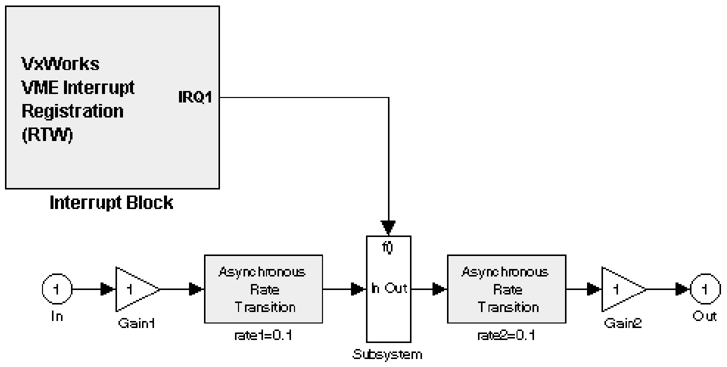
0.0

There is only one parameter:

- Sample time — set the sample time to the desired rate.

Rate Transition Block Example

This picture shows a sample application of the Rate Transition block in an ISR:



In this example, the Rate Transition block on the input to the function-call subsystem causes both the In and Gain1 blocks to run at the 0.1 second rate. The Rate Transition block on the output of the function-call subsystem causes both the Gain2 and Out blocks to run at the 0.2 second rate. Using this scheme ensures strict adherence to the sample-time setting in an ISR.

Creating a Customized Asynchronous Library for Your System

You can use the Real-Time Workshop's VxWorks Asynchronous blocks as templates that provide a starting point for creating your own asynchronous blocks. Templates are provided for these blocks:

- Asynchronous Interrupt block
- Task Synchronization block
- Rate Transition block
- Asynchronous Buffer block

You can customize each of these blocks by implementing a set of modifications to files associated with each template. These files are:

- The block's underlying S-function C MEX-file
- The block's mask and the associated mask M-file
- The TLC files that control code generation of the block

At a minimum, you must rename the system calls generated by the TLC files to the correct names for the new real-time operating system (RTOS) and supply the correct arguments for each file. There is a collection of files that you must copy (and rename) from `7/rtw/c/tornado/devices` into a new directory, for example, `*matlabroot*/rtw/c/*my_os*/devices`. These files are:

- Asynchronous Interrupt block — `vxinterrupt.tlc`, `vxinterrupt.c`, `vxintbuild.m`
- Asynchronous Buffer block — `vxdbuffer.tlc`, `vxdbuffer.c`
- Task Synchronization block — `vxtask.tlc`, `vxtask.c`
- O/S include file — `vxlib.tlc`

Configuring Real-Time Workshop for Your Application

Introduction

This chapter discusses how Simulink interacts with the Real-Time Workshop and also describes how to customize the code generated by the Real-Time Workshop for your application. The last section of this chapter describes several optimization techniques that will improve the performance of the generated code.

The Real-Time Workshop provides five different *code formats*. Each code format specifies a framework for code generation suited for specific applications.

The five code formats and corresponding application areas are:

- Real-time — Rapid prototyping
- Real-time malloc — Rapid prototyping
- Embedded-C — Deeply embedded systems
- S-function — Creating proprietary S-function `.dll` or MEX-file objects, code reuse, and speeding up your simulation
- Ada (the Real-Time Workshop Ada Extension is a separate product. See Chapter 14, "Real-Time Workshop Ada Coder," for more information.)

Apart from specifying the code format, you can also specify a *target*. A target is a hardware or operating system on which the generated code will run. Some examples of targets are Tornado, rapid simulation, DOS, etc. You can compile these targets for different operating systems by using the different template make files provided. See Chapter 3, "Code Generation and the Build Process", for more details about the build process.

Interaction between Simulink and the Real-Time Workshop

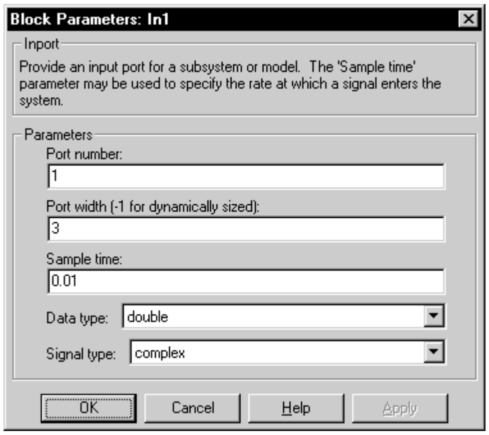
The Real-Time Workshop provides much of the functionality that Simulink does and often removes the need for users to hand-write code.

One of the key components of Simulink is its engine, which propagates data from one block to the next along signal lines. The data propagated are:

- Data type
- Line widths
- Sample times

When generating code, Simulink first compiles the block diagram. This compile stage is analogous to that of a C program. The C compiler carries out type checking and pre-processing. Similarly, Simulink verifies that input/output data types of block ports are consistent, line widths between blocks are of the correct thicknesses, and the sample times of connecting blocks are consistent.

The Simulink engine typically derives signal attributes from a source block. For example, the Inport block's parameters dialog box specifies the signal attributes for the block.


Block Parameters: In1

Inport

Provide an input port for a subsystem or model. The 'Sample time' parameter may be used to specify the rate at which a signal enters the system.

Parameters

Port number: 1

Port width (-1 for dynamically sized): 3

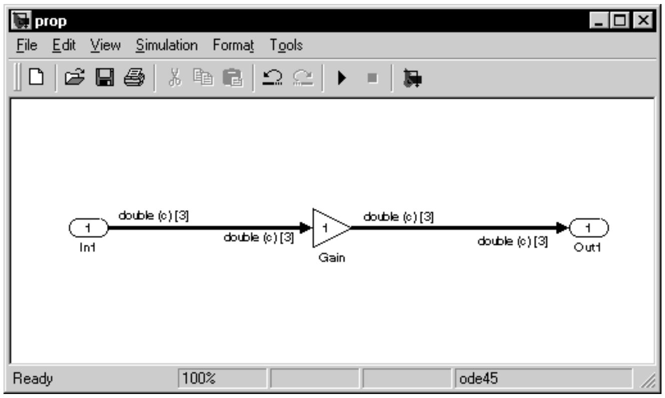
Sample time: 0.01

Data type: double

Signal type: complex

In this example, the Inport block has a port width of 3, a sample time of .01 seconds, the data type is double, and the signal is complex.

This figure shows the propagation of the signal attributes associated with the Inport block through a simple block diagram.



In this example, the Gain and Outport blocks inherit the attributes specified for the Inport block.

Inherited sample times in source blocks (e.g., a root inport) can sometimes lead to unexpected and unintended sample time assignments. Since a block may specify an inherited sample time, often there is not enough information at the outset to completely compile a block diagram. In such cases, the Simulink engine propagates the known or assigned sample times to blocks with inherited sample times which have not yet been assigned a sample time. Thus, it continues to fill in the blanks, the unknown sample times, until as many blocks as possible have been assigned a sample time. Blocks which still do not have a sample time are assigned a default sample time according to the following rule:

1 If the current system has at least one rate in it, the block is assigned the fastest rate.

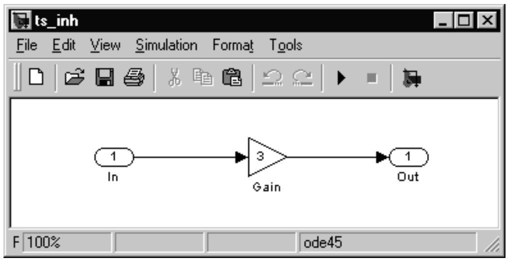
2 If no rate exists and the model is configured for a variable-step solver, the block is assigned a continuous sample time (but fixed in minor time steps). Note that the Real-Time Workshop does not currently support variable-step solvers.

3 If no rate exists and the model is configured for a fixed-step solver, the is assigned a discrete sample time of $(T_f - T_i)/50$ where T_i is the simulation

start time and T_f is the simulation stop time. If T_f is infinity, the default sample time is set to 0.2.

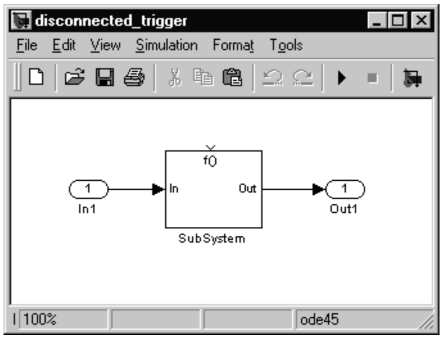
To ensure a completely deterministic model (i.e, where no sample times are set using the above rules), you should explicitly specify the sample time of all your source blocks. Source blocks include root inport blocks and any blocks without input ports. You do not have to set subsystem input port sample times. You may want to do so, however, when creating modular systems.

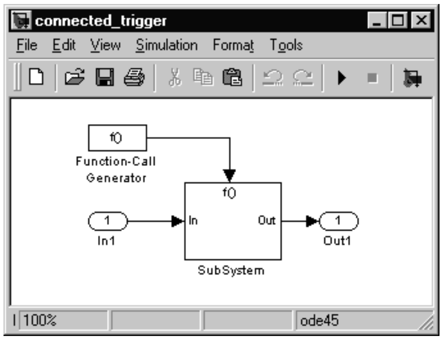
An unconnected input implicitly sources ground. For ground blocks and ground connections, the default sample time is derived from destination blocks or the default rule.



All blocks have an inherited sample time (T_s = -1). They will all be assigned a sample time of (T_f - T_i)/50.

Once Simulink compiles the block diagram, it creates a `model.rtw` file (analogous to an object file generated from a C file). The `model.rtw` file contains all the connection information of the model as well as the necessary signal attributes. Thus, the timing engine in the Real-Time Workshop knows when blocks with different rates should be executed. You cannot override this execution order by directly calling a block (in hand-written code) in a model. For example, the `disconnected_trigger` model below will have its trigger port source to ground which may lead to all blocks inheriting a constant sample time. Calling the trigger function, `f()`, directly from hand-code will not work correctly and should never be done. Instead, you should use a function-call generator to properly specify the rate at which `f()` should be executed as shown in the `connected_trigger` model below.





Instead of the function-call generator, you could have any other block that can drive the trigger port. Then, you should call the model's main entry point to execute the trigger function.

For multirate models, a common use of the Real-Time Workshop is to build individual models separately and then hand-code the I/O between the models. This approach places the burden of data consistency between models on the developer. Another approach is to let Simulink and the Real-Time Workshop ensure data consistency between rates and generate multirate code for use in a multitasking environment. The Real-Time Workshop Interrupt Template and VxWorks Support libraries provide blocks which allow synchronous and asynchronous data flow. For a description of all the Real-Time Workshop libraries, see Chapter 11, "Real-Time Workshop Libraries." For more information on multi-rate code generation, see Chapter 7, "Models with Multiple Sample Rates."

Choosing a Code Format for Your Application

There are many options you can choose on the path from a Simulink model to generated code. Depending on your specific application, settings for these options significantly influence code generation. The most important option is the code format, which specifies the overall framework of the generated code and also determines its style.

Choose the real-time or real-time `malloc` code format for rapid prototyping. If your application does not have significant restrictions in code size, memory usage, or stack usage, you may want to continue using the generic real-time target throughout development. The real-time format is the most comprehensive code format and supports almost all the built-in blocks.

If, however, your application demands that you limit source code size, memory usage, or maintain a simple call structure, then you should choose the embedded-C format.

Finally, you should choose the S-function format if you are not concerned about RAM and ROM usage and want to:

- Componentize a model for scalability
- Create a proprietary S-function `.dll` or MEX-file object
- Interface the generated code using the S-function C API
- Speed up your simulation

More details for each code format appear in the following sections.

Choosing a Target

To choose a target, you must specify these files:

- System target file
- Template makefile
- Makefile

Every target has all three of these files and is represented in the System Target File browser entry associated with it. See "The System Target File Browser" on page 3-11 for more information on the browser.

Typically, the system target file will specify the code format. If the system target file does not specify one, the default is real-time.

Table 12-1 shows the various options available for each code format/target available in the Real-Time Workshop.

Table 12-1: Features Supported by Real-Time Workshop Targets and Code Formats

	Real-Time	Real-Time malloc	Embedded-C	DOS	OSEK LE/O	Ada	Tornado	S-Fcn	RSim	Real-Time Windows
Static memory allocation	X		X	X	X	X	X	X		X
Dynamic memory allocation		X					X	X	X	
Continuous time	X	X		X	X		X	X	X	X
C MEX S-functions (not inlined with TLC)	X	X		X	X		X	X	X	X
Any S-function (inlined with TLC)	X	X	X	X	X	X	X	X	X	X
Optimized for minimum RAM and ROM Usage			X			X				
Supports external mode	X	X					X			X
Intended for rapid prototyping	X	X		X	X		X			X
Intended for embedded applications			X			X				
Batch parameter tuning and Monte Carlo methods									X	

Table 12-1: Features Supported by Real-Time Workshop Targets and Code Formats (Continued)

	Real-Time	Real-Time malloc	Embedded-C	DOS	OSEK LE/O	Ada	Tornado	S-Fcn	RSim	Real-Time Windows
Executes in hard real-time				X	X		X			X
Non real-time executable included	X	X				X			X	
Multiple instantiation of one model (provided no Stateflow blocks are in your model)		X						X		

Real-Time Code Format

The real-time code format (corresponding to the generic real-time target) is useful for rapid prototyping applications. If you want to generate real-time code while iterating model parameters rapidly, you should begin the design process with a generic real-time target. The real-time code format supports:

- Continuous time
- Continuous states
- C MEX S-functions (Non-inlined and inlined with TLC)

For more information on inlining S-functions, see Chapter 3 of the *Target Language Compiler Reference Guide*.

The real-time code format declares memory statically, that is, at compile time.

Unsupported Blocks

The real-time format does not support the following built-in blocks:

- Functions & Tables
  - MATLAB Fcn
  - S-Functions — M-file and Fortran S-functions, C MEX S-functions that call into MATLAB.

System Target Files

- `drt.tlc` — DOS real-time target
- `grt.tlc` — generic real-time target
- `osek_leo.tlc` — Lynx-Embedded OSEK target
- `rsim.tlc` — rapid simulation target
- `tornado.tlc` — Tornado (VxWorks) real-time target
- `win_watc.tlc` — Windows 95/98/NT real-time target (Watcom compiler only)

Template Makefiles

- `drt.tmf`
- `grt`
  - `grt_bc.tmf` — Borland C
  - `grt_vc.tmf` — Visual C
  - `grt_watc.tmf` — Watcom C
  - `grt_unix.tmf` — UNIX host
- `osek_leo.tmf`
- `rsim`
  - `rsim_bc.tmf` — Borland C
  - `rsim_vc.tmf` — Visual C
  - `rsim_watc.tmf` — Watcom C
  - `rsim_unix.tmf` — UNIX host
- `tornado.tmf`
- `win_watc.tmf`

Real-Time malloc Code Format

The real-time `malloc` code format (corresponding to the generic real-time malloc target) is very similar to the real-time code format. The differences are:

- Real-time `malloc` declares memory dynamically.
- Real-time `malloc` allows you to multiply instance the same model with each instance maintaining its own unique data.
- Real-time `malloc` allows you to combine multiple models together in one executable. For example, to integrate two models into one larger executable, real-time `malloc` maintains a unique instance of each of the two models. If you do not use the real-time `malloc` format, the Real-Time Workshop will not necessarily create uniquely named data structures for each model, potentially resulting in name clashes.

  `grt_malloc_main.c`, the main routine for the generic real-time `malloc` (`grt_malloc`) target, supports one model by default. You must modify it to fit your particular multimodel scenario. `grt_malloc_main.c` is located in the directory `matlabroot/rtw/c/grt_malloc`.

Unsupported Blocks

The real-time malloc format does not support the following built-in blocks:

- Functions & Tables
  - MATLAB Fcn
  - S-Functions — M-file and Fortran S-functions, C MEX S-functions that call into MATLAB.

System Target Files

- `grt_malloc.tlc`
- `tornado.tlc` — Tornado (VxWorks) real-time target

Template Makefiles

- `grt_malloc`
  - `grt_malloc_bc.tmf` — Borland C
  - `grt_malloc_vc.tmf` — Visual C
  - `grt_malloc_watc.tmf` — Watcom C
  - `grt_malloc_unix.tmf` — UNIX host
- `tornado.tmf`

Embedded-C Code Format

The embedded-C code format (corresponding to the embedded real-time target) produces code that is optimized for speed, memory usage, and simplicity. It is intended for use in deeply embedded systems, as opposed to in the rapid prototyping process. It uses static memory allocation and produces only two or three entry points that you can call from the main routine:

```
model_initialize
model_step
model_terminate (optional)
```

The embedded-C code format uses a much smaller version of the `SimStruct` called the *real-time object*. Defined in `model_export.h`, this data structure contains the necessary model and timing information.

Optimizations

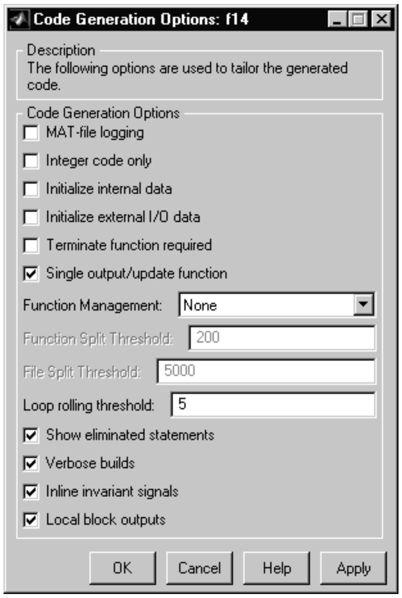
This section contains suggestions that will help you optimize the code generated by the embedded-C code format. All the options are located on the **Code Generation Options** dialog box that you can access by pressing the **Options** button on the Real-Time Workshop page of the **Simulation Parameters** dialog box.

Each of the following actions will result in more highly optimized code:

- Deselect the **MAT-file logging** check box.
- Deselect the **Initialize internal data** check box.
- Deselect the **Initialize external I/O data** check box.

  Initializing the internal and external data is a precaution and may not be necessary for your application. Many embedded application environments initialize all RAM at startup. Therefore, reinitializing the memory is redundant.
- Deselect the **Terminate function required** check box if you do not require a terminate function for your model.

- Select the **Single output/update function** check box. Combining the output and update functions allows the Real-Time Workshop to use more local variables in the step function of the model.
- Select **None** or **File Splitting** in the **Function Management** pull-down menu. Selecting function splitting disables the **Local block outputs** optimization.


Code Generation Options: f14

Description
The following options are used to tailor the generated code.

Code Generation Options
- [ ] MAT-file logging
- [ ] Integer code only
- [ ] Initialize internal data
- [ ] Initialize external I/O data
- [ ] Terminate function required
- [x] Single output/update function

Function Management: None
Function Split Threshold: 200
File Split Threshold: 5000
Loop rolling threshold: 5

- [x] Show eliminated statements
- [x] Verbose builds
- [x] Inline invariant signals
- [x] Local block outputs

Restrictions

- You must inline all S-functions with a corresponding TLC file (see Chapter 3 of the *Target Language Compiler Reference Guide* for more information about inlining S-functions).
- You cannot have any continuous time blocks in your model.
- You must select the multitasking solver mode when the model is multirate. Singletasking multirate models are not supported.

Unsupported Blocks

The embedded-C format does not support the following built-in blocks:

- Continuous
  - No blocks in this library are supported
- Discrete
  - First-Order Hold
- Functions & Tables
  - MATLAB Fcn
  - S-Functions — M-file and Fortran S-functions, C MEX S-functions that call into MATLAB.
- Math
  - Algebraic Constraint
  - Matrix Gain
- Nonlinear
  - Rate Limiter
  - Manual Switch
- Sinks
  - XY Graph
  - Display
- Sources
  - Clock
  - Chirp Signal
  - Pulse Generator
  - Ramp
  - Repeating Sequence
  - Signal Generator

System Target File

- `ert.tlc`

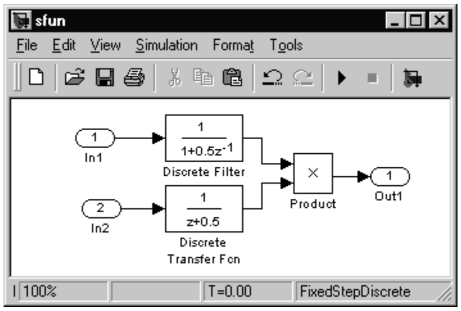
Template Makefiles

- `ert_bc.tmf` — Borland C
- `ert_vc.tmf` — Visual C
- `ert_watc.tmf` — Watcom C
- `ert_unix.tmf` — UNIX host

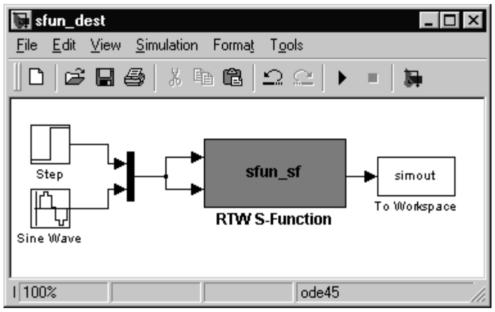
S-Function Code Format

The S-function code format (corresponding to the RTW S-function target) generates code so that you can use the model as a (C MEX) S-function block in another model. This format has these applications:

- Componentize a model — You can generate an object file for a model, `m1`, in the S-function format. Then, you can place the generated S-function block in another model, `m2`. Regenerating code for `m2` will not require regenerating code for `m1`.
- Speeding up simulation — You can generate an S-function for your model and then use different inputs to test the model. For example, the model `sfun` is pictured below.



Generating code and compiling it for an S-function target results in the block `sfun_sf`, which is an RTW S-Function block that contains all the functionality of the `sfun` model. The picture below shows the sfun_sf block embedded in a new model.



After you have placed the RTW S-Function block inside a model, you can try out several different inputs and measure the response of the original model. The speed at which the S-Function block executes is faster than the original model. This difference in speed will become more pronounced for larger and more complicated models

- Sharing the model without providing the source code — To protect your proprietary models or algorithms, you can generate an S-function from them and then only provide the binary `.dll` or MEX-file object.
- Reusing code - You can incorporate multiple instances of one model inside another without replicating the code for each instance. Each instance will continue to maintain its own unique data.

You can generate an executable by using the real-time or real-time malloc code formats for a model that contains S-functions generated by the Real-Time Workshop. You cannot use the embedded-C format since it requires inlined S-functions. Also, you can place a generated S-Function block in another model which you can then be generated with the S-function format again, effectively allowing any level of nested S-functions.

Note that sample times propagation for the S-function code format is slightly different from the other code formats. An RTW S-Function block will inherit its sample time from the model in which it is placed if no blocks in the original model specify their sample times.

Restrictions

- Hand-written S-functions without corresponding TLC files must contain exception-free code. For more information on exception-free code, refer to "Exception-Free Code" in Chapter 3 of *Writing S-Functions*.
- The parameter values of the blocks in the source model are hard-coded into the generated S-function. There is no mechanism to modify parameters in a Real-Time Workshop generated S-function.
- If you modify the source model that generated an S-Function block, the Real-Time Workshop will not automatically rebuild the model containing this S-function block.

Unsupported Blocks

The S-function format does not support the following built-in blocks:

- Functions & Tables
  - MATLAB Fcn
  - S-Functions — M-file and Fortran S-functions, C MEX S-functions that call into MATLAB.
- Sinks
  - Scope
  - To Workspace

System Target File

- `rtwsfcn.tlc`

Template Makefiles

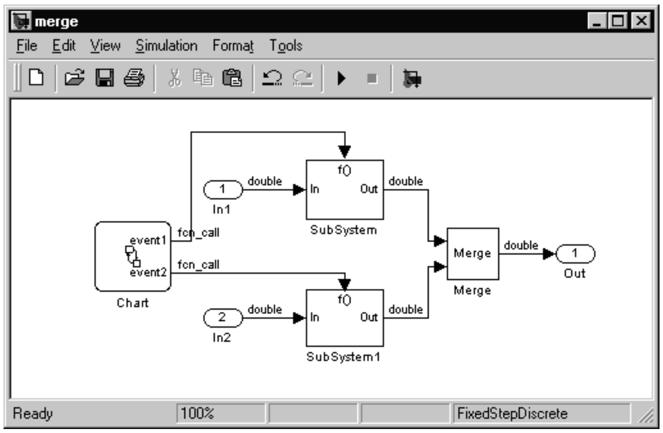
- `rtwsfcn_bc.tmf` — Borland C
- `rtwsfcn_vc.tmf` — Visual C
- `rtwsfcn_watc.tmf` — Watcom C
- `rtwsfc_unix.tmf` — UNIX host

Optimizations Common to All Code Formats

General Modeling Techniques

The following are techniques that you can use with any code format:

- Run `slupdate` on old models to automatically convert them to use the newest features.
- Directly inline C MEX S-functions into the generated code by including a TLC file for the S-function. See Chapter 3 of the *Target Language Compiler Reference Guide* for more information on writing a TLC file for inlining an S-function.
- Use a Simulink data type other than double when possible. The available data types are Boolean, signed and unsigned 8-, 16-, and 32-bit integers, and 32- and 64-bit floats. A double is a 64-bit float. See Chapter 3 of *Using Simulink* for more information on data types.
- Remove repeated values in lookup table data.
- Use the Merge block to merge the output of function-call subsystems. This block is particularly helpful when controlling the execution of function-call subsystems with Stateflow. This diagram is an example of how to use the Merge block.



Stateflow Optimizations

If your model contains Stateflow blocks, select the **Use Strong Data Typing with Simulink I/O** check box (on the **Chart Properties** dialog box) on a chart-by-chart basis.

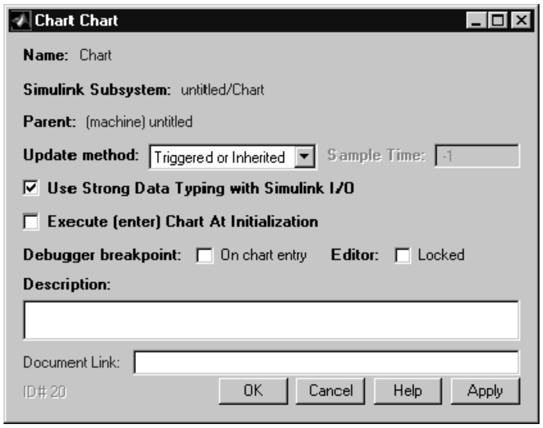


Chart Chart

Name: Chart

Simulink Subsystem: untitled/Chart

Parent: (machine) untitled

Update method: Triggered or Inherited   Sample Time: -1

[✓] Use Strong Data Typing with Simulink I/O

[ ] Execute (enter) Chart At Initialization

Debugger breakpoint: [ ] On chart entry   Editor: [ ] Locked

Description:

Document Link:

ID# 20

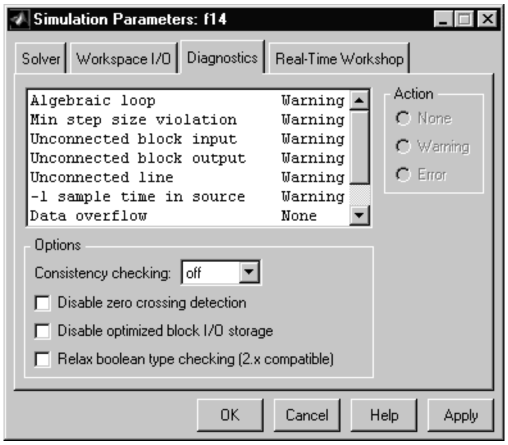
See the *Stateflow User's Guide* for more information about the **Chart Properties** dialog box.

Simulation Parameters

Options on each page of the **Simulation Parameters Dialog Box** affect the generated code.

Diagnostic Page

- Deselect the **Disable optimized block I/O storage** check box. Disabling optimized block I/O storage makes all block outputs global and unique, which in many cases significantly increases RAM and ROM usage.



```
Algebraic loop                Warning
Min step size violation       Warning
Unconnected block input       Warning
Unconnected block output      Warning
Unconnected line              Warning
-1 sample time in source      Warning
Data overflow                 None
```

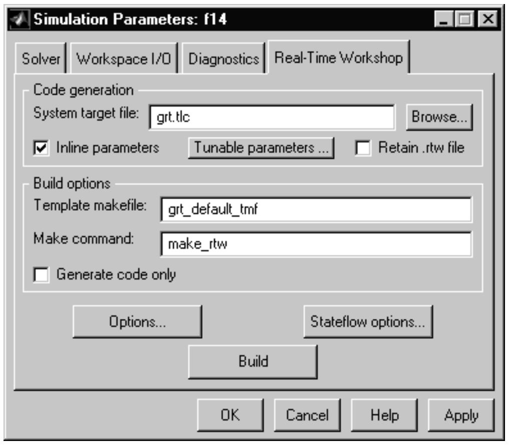
Consistency checking: off

- [ ] Disable zero crossing detection
- [ ] Disable optimized block I/O storage
- [ ] Relax boolean type checking (2.x compatible)

- Deselect the **Relax boolean type checking (2.x compatible)** check box. A boolean signal typically requires one byte of storage while a double signal requires eight bytes of storage.

Real-Time Workshop Page

- Select the **Inline parameters** check box. Inlining parameters reduces global RAM usage since parameters are not declared in the global parameters vector. Note that you can override the inlining of individual parameter by using the **Tunable Parameters** dialog box.



Code generation
System target file: grt.tlc
[✓] Inline parameters
[ ] Retain .rtw file

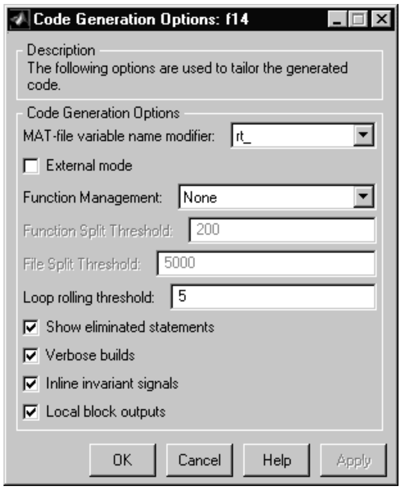
Build options
Template makefile: grt_default_tmf
Make command: make_rtw
[ ] Generate code only

Code Generation Options

You can access the **Code Generation Options** by pressing the **Options** button on the Real-Time Workshop page.

- Set an appropriate **Loop rolling threshold**. The loop rolling threshold determines when a wide signal should be wrapped into a `for` loop and when it should be generated as a separate statement for each element of the signal (see Chapter 2 of the *Target Language Compiler Reference Guide* for more information on loop rolling).

- Select the **Inline invariant signals** check box. The Real-Time Workshop will not generate code for blocks with a constant (invariant) sample time.


Code Generation Options: f14

Description
The following options are used to tailor the generated code.

Code Generation Options
MAT-file variable name modifier: rt_
☐ External mode
Function Management: None
Function Split Threshold: 200
File Split Threshold: 5000
Loop rolling threshold: 5
☑ Show eliminated statements
☑ Verbose builds
☑ Inline invariant signals
☑ Local block outputs

- Select the **Local block outputs** check box. Blocks signals will be declared locally in functions instead of being declared globally (when possible). This check box is ignored when the **Disable optimized block I/O storage** check box is selected.

Compiler Options

- If you do not require double precision for your application, define `real_T` as `float` in your template make file, or you can simply specify `-DREAL_T=float` after `make_rtw` in the **Make command** field.
- Turn on the optimizations for the compiler (e.g., `-O2` for `gcc`, `-Ot` for Microsoft Visual C).

Real-Time Workshop Rapid Simulation Target

Introduction

The Real-Time Workshop rapid simulation target (`rsim`) consists of a set of target files for nonreal-time execution on your host computer. You can use the Real-Time Workshop to generate fast, stand-alone simulations that allow batch parameter tuning and loading of new simulation data (signals) from a standard MATLAB MAT-file without needing to recompile your model.

C code generated from Real-Time Workshop is highly optimized to provide fast execution of discrete-time systems or systems that use a fixed-step solver. The speed of the generated code also makes it ideal for batch or Monte Carlo simulation. The run-time interface for the rapid simulation target enables the generated code to read and write data to standard MATLAB MAT-files. Using these support files, `rsim` reads new signals and parameters from MAT-files at the start of the simulation.

After building an `rsim` executable with Real-Time Workshop and an appropriate C compiler for your host computer, you can perform any combination of the following by using command line options. Without recompiling, the rapid simulation target allows you to:

- Specify a new file(s) that provides input signals for From File blocks
- Specify a new file that provides input signals with any Simulink data type (`double`, `float`, `int32`, `uint32`, `int16`, `uint16`, `int8`, `uint8`, and complex data types) by using the From Workspace block
- Replace the entire block diagram parameter vector (restricted to data of type double) and run a simulation
- Specify a new stop time for ending the stand-alone simulation
- Specify a new name of the MAT-file used to save model output data
- Specify name(s) of the MAT-files used to save data connected to To File blocks

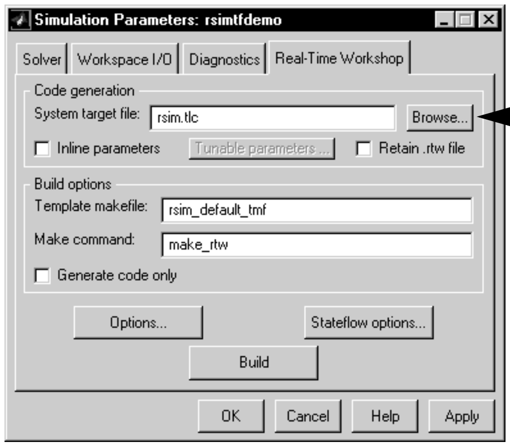
Since it is possible to run these options:

- Directly from your operating system command line (for example, DOS box or UNIX shell) or
- By using the bang (!) command with a command string at the MATLAB prompt

you can easily write simple scripts that will run a set of simulations in sequence while using new data sets. These scripts can be written to provide unique filenames for both input parameters and input signals, as well as output filenames for the entire model or for To File blocks.

Building for the Rapid Simulation Target

By specifying the system target file (`rsim.tlc`) and the template makefile (`rsim_default_tmf`) on the Real-Time Workshop page of the **Simulation Parameters** dialog box, you can use the Real-Time Workshop to generate and build an `rsim` executable. This picture shows the dialog box settings for the rapid simulation target



Simulation Parameters: rsimtfdemo

Code generation
System target file: rsim.tlc
Inline parameters: off
Retain .rtw file: off

Build options
Template makefile: rsim_default_tmf
Make command: make_rtw
Generate code only: off

Press the **Browse** button and select the rapid simulation target from the **System Target File Browser**. This automatically selects the correct settings for the system target file, the template makefile, and the make command.

Figure 13-1: Specifying Target and Make Files for rsim.

After specifying system target and make files as noted above, select any desired Workspace I/O settings, and press **Build**. The Real-Time Workshop will automatically generate C code and build the executable for your host machine using your host machine C compiler. See "Target Systems and Associated Support Files" on page 2-9 for additional information on compilers that are compatible with Simulink and the Real-Time Workshop.

Running a Rapid Simulation

The rapid simulation target lets you run a simulation similar to the generic real-time target (`grt`) provided by the Real-Time Workshop. This simulation does not use timer interrupts, and therefore is a nonreal-time simulation environment. The difference between `grt` and `rsim` simulations is that `rsim` allows you to change parameter values or input signals at the start of a simulation without the need to generate code or recompile. The generic real-time target, on the other hand, is a starting point for targeting a new processor.

A single build of your model can be used to study effects from varying parameters or input signals. Command line arguments provide the necessary mechanism to specify new data for your simulation. This table lists all available command line options.

Table 13-1: rsim Command Line Options

Command Line Option	Description
`model -f old.mat=new.mat`	Read From File block input signal data from a replacement MAT-file.
`model -o newlogfile.mat`	Write MAT-file logging data to a file named `newlogfile.mat`.
`model -p filename.mat`	Read a new (replacement) parameter vector from a file named `filename.mat`.
`model -s <stoptime>`	Run the simulation until the time value `<stoptime>` is reached.
`model -t old.mat=new.mat`	The original model specified saving signals to the output file `old.mat`. For this run use the file `new.mat` for saving signal data.
`model -v`	Run in verbose mode.
`model -w workspace.mat`	After you have saved all workspace signal data to a MAT-file (as structures), this option allows you to select a new MAT-file containing a different set of signal data. This includes all supported data types.

<-f> Specifying a New Signal Data File for a From File Block

To understand how to specify new signal data for a From File block, create a working directory and cd to that directory. Open the model `rsimtfdemo` by typing

```
  rsimtfdemo
```

at the MATLAB prompt. Type

```
w = 100;
zeta = 0.5;
```

to set parameters. Copy a test data file by typing

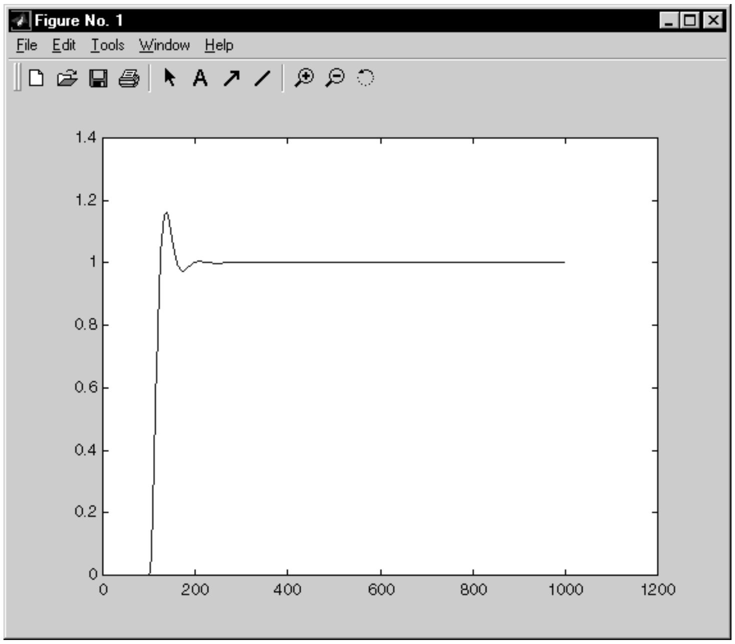
```
!matlabroot\toolbox\rtw\rtwdemos\rsim_tfdata.mat
```

at the MATLAB prompt.

Be sure to specify `rsim.tlc` as the system target file and `rsim_default_tmf` as the template makefile. Then press the **Build** button on the Real-Time Workshop page to create the `rsim` executable.

```
!rsimtfdemo
load rsimtfdemo
plot(rt_yout)
```

The resulting plot shows simulation results using the default input data.



Replacing Input Signal Data. New data for a From File block can be placed in a standard MATLAB MAT-file. As in Simulink, the From File block data must be stored in a matrix with the first row containing the time vector while subsequent rows contain u vectors as input signals. After generating and compiling your code, you can type the model name rsimtfdemo at a DOS prompt to run the simulation. In this case, the file rsm_tfdata.mat provides the input data for your simulation.

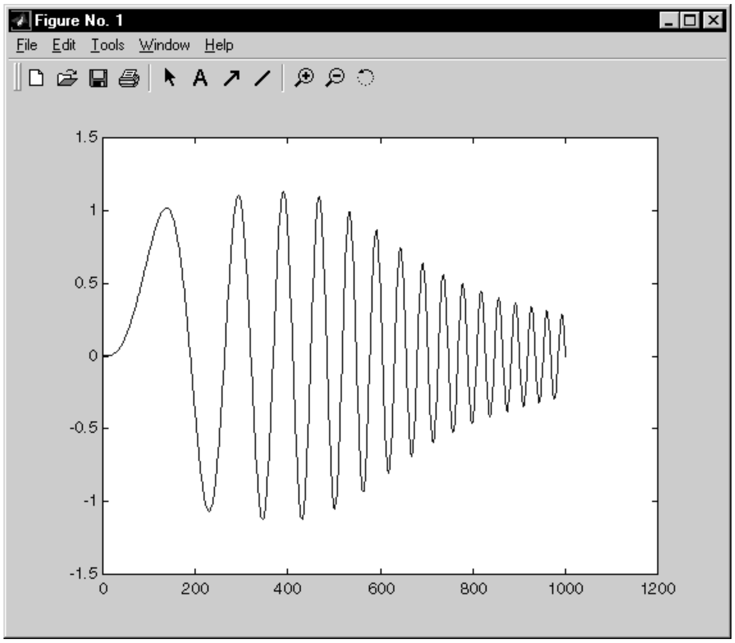
For the next simulation, create a new data file called newfrom.mat and use this to replace the original file (rsim_tfdat.mat) and run an rsim simulation with this new data. This is done by typing

```
t=[0:.001:1];
u=sin(100*t.*t);
tu=[t;u];
save newfrom.mat tu;
!rsimtfdemo -f rsim_tfdata.mat=newfrom.mat
```

at the MATLAB prompt. Now you can load the data and plot the new results by typing

```
load rsimtfdemo
plot(rt_yout)
```

This picture shows the resulting plot.



As a result the new data file is read and the simulation progresses to the stop time as specified in the Solver page of the **Simulation Parameters** dialog box. It is possible to have multiple instances of From File blocks in your Simulink model.

Since `rsim` does not place signal data into generated code, it reduces code size and compile time for systems with large numbers of data points that originate in From File blocks. The From File block requires the time vector and signals to be data of type double. If you need to use data types other than double, use a From Workspace block with the data specified as a structure. The workspace data must be in this format:

```
variable.time
variable.signals.values
```

If you have more than one signal, the format must be:

```
variable.time
variable.signals(1).values
variable.signals(2).values
```

<-o> Specifying a New Output Filename for the Simulation

If you have specified **Save to Workspace** options (that is, checked **Time**, **States**, **Outputs**, or **Final States** checkboxes on the Workspace I/O page of the **Simulation Parameters** dialog box under the **Simulation** menu in Simulink), the default is to save simulation logging results to the file `*modelname*.mat`. You can now specify a replacement filename for subsequent simulations. In the case of the model `rsimtfdemo`, by typing `!rsimtfdemo` at the MATLAB prompt, a simulation runs and data is normally saved to `rsimtfdemo.mat`.

```
!rsimtfdemo
created rsimtfdemo.mat
```

You can specify a new output filename for data logging by typing:

```
!rsimtfdemo -o sim1.mat
```

In this case, the set of parameters provided at the time of code generation, including any From File block data, is run. You can combine a variety of `rsim` flags to provide new data, parameters, and output files to your simulation. Note that the MAT-file containing data for the From File blocks is required. This differs from the `grt` operation, which inserts MAT-file data directly into the generated C code that is then compiled and linked as an executable. In contrast, `rsim` allows you to provide new or replacement data sets for each successive simulation. A MAT-file containing From File or From Workspace data must be present, however, if any of these blocks (From File or From Workspace blocks) exist in your model.

<-p> Changing Block Parameters for an rsim Simulation

Once you have altered one or more parameter in the Simulink block diagram, you can extract the parameter vector, `rtP`, for the entire model. The `rtP` vector, along with a model checksum, can then be saved to a MATLAB MAT-file. This MAT-file can be read in directly by the stand-alone `rsim` executable allowing you to replace the entire parameter vector quickly, for running studies of variations of parameter values.

The model checksum provides a safety check to ensure that any parameter changes are only applied to `rsim` models that have the same model structure. If any block is deleted, or a new block added, then when generating a new `rtP` vector, the new checksum will no longer match the original checksum. `rsim` will detect this incompatibility in parameter vectors and exit to avoid returning incorrect simulation results. In this case, where model structure has changed, you must regenerate the code for the model.

The `rsim` target allows you to alter any model parameter (of type double) including parameters that include *side-effects* functions. An example of a side-effects function is a simple Gain block that includes the following parameter entry in a dialog box:

```
gain value:   2 * a
```

In general, the Real-Time Workshop evaluates side-effects functions prior to generating code. The generated code for this example retains only one memory location entry, and the dependence on parameter `a` is no longer visible in the generated code. The `rsim` target overcomes the problem of handling side-effects functions by replacing the entire parameter structure, `rtP`. You must create this new structure by using `rsimgetrtp.m`. and then save it in a MAT-file. For the `rsimtfdemo` example, type

```
zeta = .2;
myrtp = rsimgetrtp('modelname');
save myparamfile myrtp;
```

at the MATLAB prompt.

In turn, `rsim` can read the MAT-file and replace the entire `rtP` structure whenever you need to change one or more parameters — without recompiling the entire model.

For example, assume that you have changed one or more parameters in your model, generated the new `rtP` vector, and saved `rtP` to a new MAT-file called `myparamfile.mat`. In order to run the same `rsimtfdemo` model and use these new parameter values, execute the model by typing:

```
!rsimtfdemo -p myparamfile.mat
load rsimtfdemo
plot(rt_yout)
```

Note that the `p` is lower-case and represents “Parameter file.”

<-s> Specifying a New Stop Time for an rsim Simulation

If a new stop time is not provided, the simulation will run until reaching the value specified in the Solver page at the time of code generation. You can specify a new stop time value as follows:

```
!rsimtfdemo -s 6.0
```

In this case, the simulation will run until it reaches 6.0 seconds. At this point it will stop and log the data according to the MAT-file data logging rules as described above.

If your model includes From File blocks that also include a time vector in the first row of the time and signal matrix, the end of the simulation is still regulated by the original setting in the Solver page of the **Simulation Parameters** dialog box or from the `-s` option as described above. However, if the simulation time exceeds the end points of the time and signal matrix (that is, if the final time is greater than the final time value of the data matrix), then the signal data will be extrapolated out to the final time value as specified above.

<-t> Specifying New Output Filenames for To File BLocks

In much the same way as you can specify a new system output filename, you can also provide new output filenames for data saved from one or more To File blocks. This is done by specifying the original filename at the time of code generation with a new name as follows:

```
!mymodel -t original.mat=replacement.mat
```

In this case, assume that the original model wrote data to the output file called `original.mat`. Specifying a new filename forces `rsim` to write to the file `replacement.mat`. This technique allows you to avoid over-writing an existing simulation run.

<-w> Specifying a New MAT-file with From Workspace Data

The `-w` option allows you to specify the name of a new MAT-file containing From Workspace signal data. The From Workspace block supports signals with data types including: int8, uint8, int16, uint16, int32, uint32, float, double, and the complex signals of these data types. MATLAB consists of one workspace in which you can have a selection of variables containing signal data. `rsim`, by default, expects to find a MAT-file called `from_workspace.mat` that contains the same variables (in the structure format) including time and input signal).

From Workspace data must be contained in a structure variable with the following fields:

```
var.time
var.signals.values
```

The field `var.time` must be of type double. The field `var.signals.values` can be of any data type that Simulink supports. The length of `var.time` must equal the length of `var.signals.values`.

By using the From Workspace block with `rsim`, you can input new signal data in any of the supported data types without the need to regenerate or recompile your code.

You can also run a simulation using substitute signal data if each signal data type is consistent with the signal data type at the time of code generation. For example, if your model uses a signal contained in a MATLAB workspace structure that is called

```
voice.signal.values
```

and this signal is of type int32, the replacement MAT-file data must also contain a variable called `voice` with the field `voice.signals.values` of type int32. The variable must also contain the same number of channels (number of signals). If desired, you can use time and signal data sets that are much larger than the data used at the time of code generation. We recommend this for simulations with very large sets of signal data, since it greatly reduces the amount of unnecessary data that the Real-Time Workshop generates in the `.rtw` file.

The `rsim` target permits only one MAT-file which must contain all signal data. If you do not intend to replace the original signal data, you can store the original data in the default file, `from_workspace.mat`. In this case, you can run the `rsim` simulation without any arguments:

```
model
```

This has the same effect as running:

```
model -w from_workspace.mat
```

If you have specified data logging options prior to code generation, or, if you are using To File or To Workspace blocks, data will be saved to the appropriate MAT-file.

Simulation Performance

It is not possible to predict accurately the simulation speed-up of an `rsim` simulation compared to a standard Simulink simulation. Performance will vary. Larger simulations have achieved speed improvements of up to 10 times faster than standard Simulink simulations. Some models may not show any noticeable improvement in simulation speed. The only way to determine speed-up is to time your standard Simulink simulation and then compare its speed with the associated `rsim` simulation.

Batch and Monte Carlo Simulations

The `rsim` target is intended to be used for batch simulations in which parameters and/or input signals are varied for each new simulation. New output filenames allow you run new simulations without over-writing prior simulation results. A simple example of such a set of batch simulations can be run by creating a `.bat` file for use under Microsoft Windows 95 or Windows NT.

This simple file (for Windows 95 or Windows NT) is created with any text editor and executed by typing the filename, for example, `mybatch`, where the name of the text file is `mybatch.bat`:

```
rsimtfdemo -f rsimtfdemo.mat=run1.mat -o results1.mat -s 10.0
rsimtfdemo -f rsimtfdemo.mat=run2.mat -o results2.mat -s 10.0
rsimtfdemo -f rsimtfdemo.mat=run3.mat -o results3.mat -s 10.0
rsimtfdemo -f rsimtfdemo.mat=run4.mat -o results4.mat -s 10.0
```

In this case, batch simulations are run using the four sets of input data in files `run1.mat`, `run2.mat`, and so on. `rsim` saves the data to the corresponding files specified after the `-o` option.

The variable names containing simulation results in each of these files are identical. Therefore, loading consecutive sets of data without renaming the data once it is in the MATLAB workspace will result in over-writing the prior workspace variable with new data. If you want to avoid over-writing, you can copy the result to a new MATLAB variable prior to loading the next set of data.

You can also write M-file scripts to create new signals, and new parameter structures, as well as to save data and perform batch runs using the bang command (`!`).

For additional insight into the rapid simulation target, explore `rsimdemo1` and `rsimdemo2`, located in `*matlabroot*/toolbox/rtw/rtwdemos`. These examples demonstrate how `rsim` can be called repeatedly within an M-file for Monte Carlo simulations.

Real-Time Workshop Ada Coder

Introduction

This chapter presents an introduction to the Real-Time Workshop Ada Coder. It compares and contrasts the Real-Time Workshop Ada Coder with the Real-Time Workshop, shows you how to use the product by presenting an example, and concludes with a discussion of code validation.

Note: The Real-Time Workshop Ada Coder is a separate product from the Real-Time Workshop.

Like the Real-Time Workshop, the Real-Time Workshop Ada Coder provides a real-time development environment that features:

- A rapid and direct path from system design to hardware implementation
- Seamless integration with MATLAB and Simulink
- A simple, easy-to-use interface
- An open and extensible architecture

The package includes application modules that allow you to build complete programs targeting a wide variety of environments. Program building is fully automated. Automatic program building provides a standard means to create programs for real-time applications. This chapter contains examples of automatic program building on DOS and UNIX platforms.

The Real-Time Workshop Ada Coder is an automatic Ada language code generator. It produces Ada95 code directly from Simulink models and automatically builds programs that can be run in real time in a variety of environments. The Real-Time Workshop Ada Coder is an extension of the Real-Time Workshop.

With the Real-Time Workshop Ada Coder, you can run your Simulink model in real time on a remote processor. You can run accelerated, stand-alone simulations on your host machine or on an external computer.

Real-Time Workshop Ada Coder Applications

Like the Real-Time Workshop, the Real-Time Workshop Ada Coder supports a variety of real-time applications:

- Real-Time Control – You can design your control system using MATLAB and Simulink and generate Ada code from your block diagram model. You can then compile and download the Ada code directly to your target hardware.
- Hardware-in-the-Loop Simulation – You can use Simulink to model real-life measurement and actuation signals. You can use the code generated from the model on special-purpose hardware to provide a real-time representation of the physical system. Applications include control system validation, training simulation, and fatigue testing using simulated load variations.

Supported Compilers

The Real-Time Workshop Ada Coder supports the GNAT Ada Compiler version 3.11.

Supported Targets

The Real-Time Workshop Ada Coder supports the following targets:

- Ada Simulation Target — useful for validating generated code. This does not use Ada tasking primitives.
- Ada Multitasking Real-Time Target — useful as a starting point for targeting real-time systems. This uses Ada tasking primitives.

You can also add your own target by creating a system target file, make process, and run-time interface files along with any device drivers using Ada inlined C MEX S-functions.

The Generated Code

The generated code (i.e., the model code) is highly optimized, fully commented, and can be generated from any discrete-time Simulink model — linear or nonlinear.

All Simulink blocks are automatically converted to code, with the exception of:

- MATLAB function blocks
- Any continuous sample time blocks
- S-functions that are not inlined using the Target Language Compiler

See “Supported Blocks” on page 14–21 for a list of supported blocks.

Types of Output

The Real-Time Workshop Ada Coder’s interface supports two forms of output:

- Ada code – Generate code that contains system equations and initialization functions for the Simulink model. You can use this code in real-time applications.
- A real-time program – Transform the generated code into a real-time program suitable for use with dedicated real-time hardware. The resulting code is designed to interface with an external clock source and hence runs at a fixed, user-specified sample rate.

Supported Blocks

See “Supported Blocks” on page 14-21 for a complete list of the Simulink blocks supported by the Real-Time Workshop Ada Coder.

Restrictions

The Real-Time Workshop Ada Coder has the same constraints imposed upon it as the embedded-C target. The code generator does not produce code that solves algebraic loops, and Simulink blocks that are dependent on absolute time can be used only if the program is not intended to run for an indefinite period of time.

There are additional constraints for the Ada code generation. The Real-Time Workshop Ada Coder does not provide:

- Nonreal-time variable step integration models
- Continuous-time integration
- Since Ada does not support implicit upcasting of data types, all numerical operations in your model must be of homogeneous data types for Ada code

generation. You can, however, perform explicit upcasting using the Data Type Conversion block in Simulink.

S-Functions

S-functions provide a mechanism for extending the capabilities of Simulink. They allow you to write your own Ada code and incorporate it into the Simulink model as an S-function block in the generated code.

Ada MEX-files are not supported. You can use MEX-files in the Real-Time Workshop Ada Coder by first developing S-functions as C MEX-files or M-files for use with Simulink simulations. After the S-function has been tested with Simulink, you must create a TLC file that directs the Target Language Compiler to create Ada code that performs the algorithms defined in your S-function. See the *Target Language Compiler Reference Guide* for more information. Also, see examples located in `matlabroot/simulink/src` and the TLC files in `matlabroot/toolbox/simulink/blocks/tlc_ada`.

The Real-Time Workshop Ada Coder generates code for a Simulink model and compiles the generated code (along with the appropriate application modules) to produce a stand-alone program that executes independently of Simulink.

Getting Started

This section illustrates, through a simple example, how to transform a Simulink model into a stand-alone executable program. This program runs independently of Simulink, allowing accelerated execution on the development host or a different target computer.

Generating Ada code from a Simulink model is very similar to generating C code. Begin by typing

```
countersdemo
```

at the MATLAB prompt. This block diagram appears.

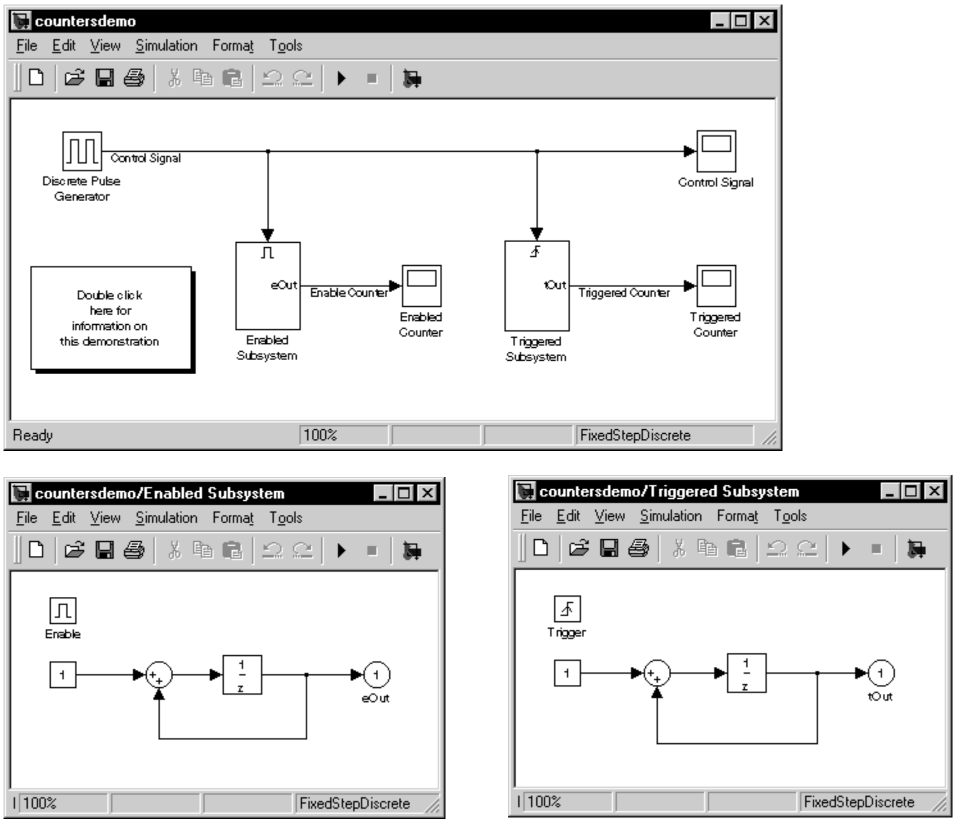


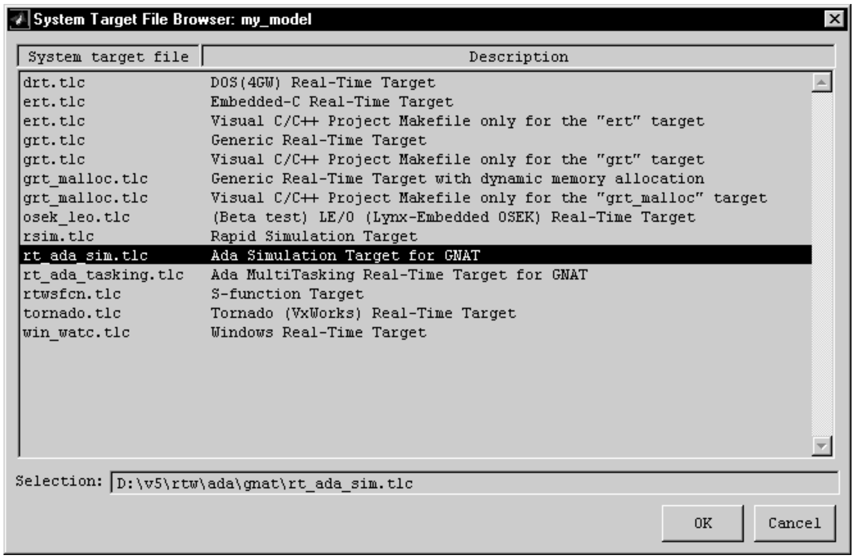
Figure 14-1: Counter Demonstration with Subsystems Open

Setting Options for Ada Code Generation

You must specify the correct options before you generate Ada code from this model. These are the steps:

1. Select **Parameters** under the **Simulation** menu. This opens the **Simulation Parameters** dialog box.
2. On the Solver page, set the **Solver options** to **Fixed-step discrete (no continuous states)**
3. Click the **Browse** button on the Real-Time Workshop page. This opens the **System Target File Browser**.
4. Select **Ada Simulation Target for GNAT** and click **OK**. This automatically sets the correct **System target file**, **Template makefile**, and **Make command** fields for Ada code generation.

This picture shows the System Target File Browser with the correct selection for Ada code generation:



System target file	Description
drt.tlc	DOS(4GW) Real-Time Target
ert.tlc	Embedded-C Real-Time Target
ert.tlc	Visual C/C++ Project Makefile only for the "ert" target
grt.tlc	Generic Real-Time Target
grt.tlc	Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc	Generic Real-Time Target with dynamic memory allocation
grt_malloc.tlc	Visual C/C++ Project Makefile only for the "grt_malloc" target
osek_leo.tlc	(Beta test) LE/O (Lynx-Embedded OSEK) Real-Time Target
rsim.tlc	Rapid Simulation Target
rt_ada_sim.tlc	Ada Simulation Target for GNAT
rt_ada_tasking.tlc	Ada MultiTasking Real-Time Target for GNAT
rtwsfcn.tlc	S-function Target
tornado.tlc	Tornado (VxWorks) Real-Time Target
win_watc.tlc	Windows Real-Time Target

Selection: D:\v5\rtw\ada\gnat\rt_ada_sim.tlc

Figure 14-2: The System Target File Browser

Alternatively, you can specify the settings on the Real-Time Workshop page manually by following these steps:

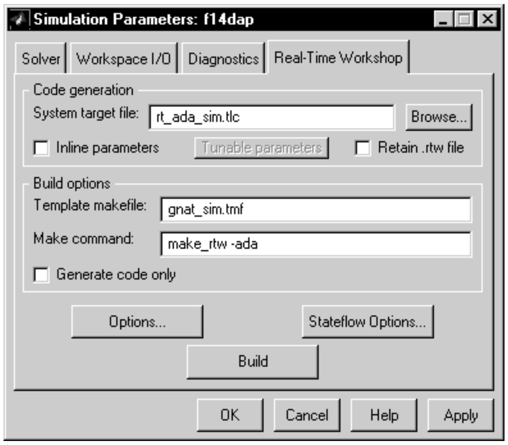
1 Select **RTW Options** under the **Tools** menu. This opens the Real-Time Workshop page of the **Simulation Parameters** dialog box.

2 Specify `rt_ada_sim.tlc` as the **System target file**.

3 Specify `gnat_sim.tmf` as the **Template makefile**.

4 Specify `make_rtw -ada` as the **Make command**.

This picture shows the Real-Time Workshop page with the correct settings.


Simulation Parameters: f14dap

Solver | Workspace I/O | Diagnostics | Real-Time Workshop

Code generation
System target file: rt_ada_sim.tlc
☐ Inline parameters
☐ Retain .rtw file

Build options
Template makefile: gnat_sim.tmf
Make command: make_rtw -ada
☐ Generate code only

Figure 14-3: The Real-Time Workshop Page of the Simulation Parameters Dialog Box

In addition, you can use the `make` command to pass compiler switches to the code compilation phase. For example, if you want to compile with debugging symbols, add a `-g` after the `-ada` switch in the **Make command** field (there must be a space between each switch). This switch is applied on a model basis; for more permanent changes, see "Configuring the Template Makefile" on page 14-12.

Generating Ada Code

To generate Ada code and build an Ada executable, press **Build** on the Real-Time Workshop page or select **RTW Build** under the **Tools** menu.

Generated Files

This table lists the Ada files generated by the Real-Time Workshop Ada Coder from the counter demonstration (`countersdemo`).

Table 14-1: Ada Files Generated by the Real-Time Workshop Ada Coder

Filename	Description
`countersdemo.adb`	Package body with the implementation details of the model.
`countersdemo.ads`	Package specification that defines the callable procedures of the model and any external inputs and outputs to the model.
`countersdemo_types.ads`	Package specification of data types used by the model. The Real-Time Ada Coder derives the data types from the block name and signal width.
`register.ads`	Package specification that defines model rate information and renames program entry points in `countersdemo.ads`.
`register2.ads`	Package specification that contains the subset of `register.ads` required for elimination of circular compilation dependencies.
`rt_engine-rto_data.ads`	Package specification that contains the timing information for executing the model encapsulated in the real-time object.

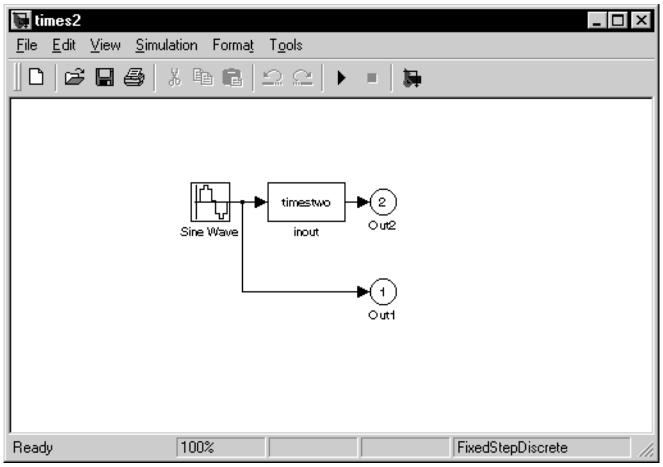
Models with S-Functions

Non-inlined Ada S-functions are currently not supported by the Real-Time Workshop Ada Coder. It is possible, however, to generate Ada code for models with C MEX S-functions. To do so, you must create a TLC file that incorporates the algorithm from your S-function. The example presented in this section shows how to write a TLC for a simple C MEX S-function. For more information on writing TLC files, see the *Target Language Compiler Reference Guide*.

Create model `times2` using these blocks:

- Sine Wave (sample time = 0.1)
- `timestwo` S-function (provided in the *`matlabroot`*`/toolbox/simulink/blocks`) with no parameters set (i.e., leave the **Parameters** field blank)
- Two Outport blocks

Your model should look like this picture.



The `times2` model contains a simple S-function, called `timestwo`, that takes the input sine wave signal and doubles its amplitude. The TLC file corresponding to the S-function is shown below:

```
%% Copyright (c) 1994-98 by The MathWorks, Inc.
%%
%% Abstract:
%%      TLC file for timestwo.c used in Real-Time Workshop
%%      S-Function test.

%implements "timestwo" "Ada"

%% Function: Outputs
==========================================================
%function Outputs(block, system) Output
  -- %<Type> Block: %<Name>
  -- Multiply input by two
  %<LibBlockOutputSignal(0, "", "", 0)> := ...
    %<LibBlockInputSignal(0, "", "", 0) > * 2.0;

%endfunction

%% [EOF] timestwo.tlc
```

The key line in this TLC file is the assignment of two times the `LibBlockInputSignal` to the `LibBlockOutputSignal`. This line directs the Target Language Compiler to place the algorithm directly into the generated Ada code.

This TLC file is located in
`matlabroot``/toolbox/simulink/blocks/tlc_ada/timestwo.tlc`.

Generating the Ada Code

The build process is similar to the case where your model does not contain any S-functions. The only additional requirement for generating Ada code for models containing S-functions is to provide a TLC file with the same name as the S-function. Otherwise, the build procedure is exactly the same.

Configuring the Template Makefile

Template makefiles specify the compiler, link, and make directives native to the target computer's operating system and compiler you are using. Two examples of template makefiles are provided in the directory `matlab/rtw/ada/gnat`. File `gnat_sim.tmf` is the template makefile that builds that real-time Ada simulation program. The automatic build process expands the macros defined at the top of the `.tmf` file to create the call to `gnatmake`. `gnatmake` then compiles and links the program.

There are two ways to make permanent changes to the template makefile if you need to make modifications either to target a different Ada95 compiler or to make minor adjustments to the gnat make directive:

- Copy the template makefile into the directory where the model is located
- Copy the template makefile to a unique name in `matlab/rtw/ada/gnat` and make modifications to the new file.

Data Logging

You can use the Ada real-time simulation target (`rt_ada_sim`) to perform the same data logging as a Simulink simulation. To enable MAT-file logging, select the **MAT-file logging** check box in the **Code Generation Options** dialog box. When this option is selected, the Ada program executes for the duration specified by the **Stop time** field on the Solver page of the **Simulation Parameters** dialog box.

When the Ada Coder finishes its run, a *`model`*`.mat` file is created that contains all workspace variables that would have been created by running a Simulink simulation. The names of these workspace variables are the same as the names that would have been created by Simulink, except that an `rt_` prefix is attached. See Chapter 5, "Data Logging and Signal Monitoring" for more information about data logging.

Note If you do not select MAT-file logging, the stop time is ignored and the Ada program runs without stopping.

Application Modules Required for the Real-Time Program

Building the real-time program requires a number of support files in addition to the generated code. These support files contain:

- A main program
- Code to drive execution of the model code
- Code to carry out data logging

The makefile automatically compiles and links these source modules. This diagram shows the modules used to build the countersdemo example.

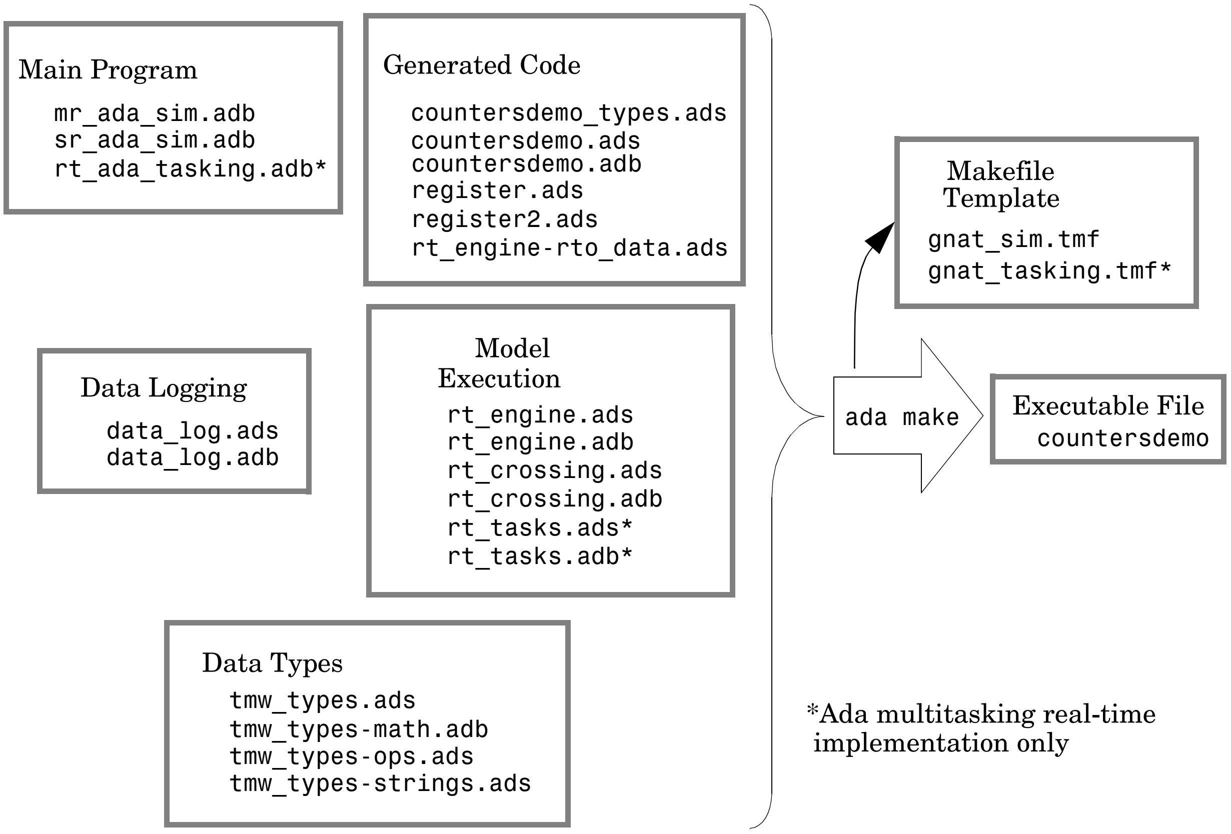
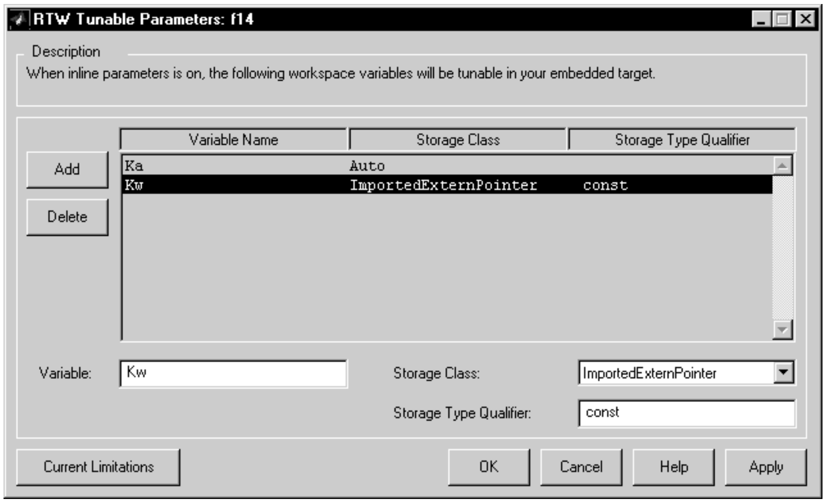


Figure 14-4: Source Modules Used to Build the countersdemo Program

Tunable Parameters

If you select **Inline parameters** on the Real-Time Workshop page, the **Tunable parameters** button activates. Clicking this button opens the **RTW Tunable Parameters** dialog box. This picture shows this dialog box for the `f14` model.



Description
When inline parameters is on, the following workspace variables will be tunable in your embedded target.

Variable Name	Storage Class	Storage Type Qualifier
Ka	Auto	
Kw	ImportedExternPointer	const

Variable: Kw
Storage Class: ImportedExternPointer
Storage Type Qualifier: const

The **RTW Tunable Parameters** dialog box supports the following features:

- Parameter tuning — de-inline any model parameter by placing its name in the **Variable** field and clicking **Add**. This effectively negates the Inline parameters check box for the variables you select. All other model parameters remain inlined.
- **Storage Class** — you can change the storage class of the selected variable. The options available in the **RTW Tunable Parameters** dialog box have C code names that map to Ada concepts. The options in the pull-down menu are as follows:
  - Auto — directs the Real-Time Workshop Ada Coder to store the variable in a persistent data structure.
  - Exported Global — declares the variable as a global variable that can be accessed from outside the generated code.

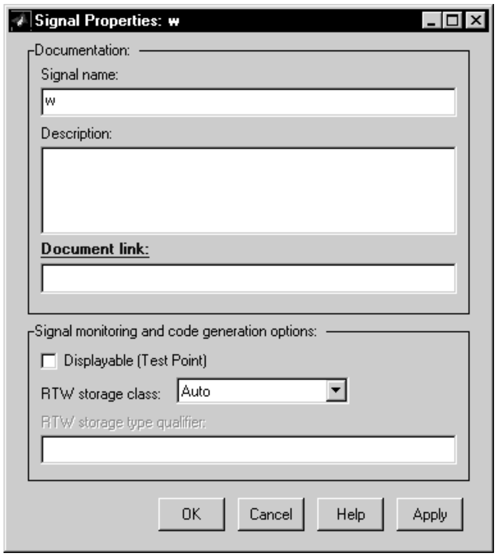
    - ImportedExtern — the variable is assumed to be declared in the package specification entered in the **Storage Type Qualifier** field. The generated code accesses this variable as `Your_Package.Your_variable`.
    - ImportedExternPointer — this is not permitted in Ada.
- **Storage Type Qualifier** — this is only used when specifying the package specification to qualify fully the variable name for the ImportedExtern option. This field is ignored in all other cases.

  These cases are useful if you want to link Real-Time Workshop generated code to other Ada code (that is, code that the Real-Time Workshop did not generate)

Signal Properties

The Real-Time Workshop Ada Coder supports the same storage class options for signals as it does for parameters. It has heuristics for supporting Simulink signal labels. These heuristics automatically map signal labels to Simulink blocks based on the block name, signal name, and connectivity. You can override the default behavior and either specify an external declaration for the signal name or direct the Real-Time Workshop Ada Coder to declare a unique declaration of the signal that is visible in the generated model package specification. The heuristics are implemented on a signal basis as specified by the **Signal name**, **RTW Storage Class**, and **RTW Storage Type Qualifier** (externally declared signals only).

To change the storage class of a signal, select it in your Simulink model; then select **Signal Properties** under the **Edit** menu of your model. This opens the **Signal Properties** dialog box.



Refer to *Using Simulink* for information about these options.

The options relevant to the Real-Time Workshop Ada Coder are located in the Signal monitoring and code generation options panel in the bottom half of the dialog box. The supported features are as follows:

- **Displayable (Test Point)** — clicking this check box directs the Real-Time Workshop Ada Coder to place the signal in a unique global memory location (`Block_IO` structure). This is useful for testing purposes since it eliminates the possibility of overwriting the signal data. Note that selecting this option forces the **RTW storage class** to be auto.
- **RTW storage class —** you can change the storage class of the selected signal. The options in the pull-down menu are as follows:
  - Auto — directs the Real-Time Workshop to store the signal in a persistent data structure. Specifically, an element called *`Signal_Name`* is declared in the `External_Inputs` structure defined in the *`Model_Types`* package

  specification. The generated code accesses this signal as `RT_U.`*`Signal_Name`*.
  - ExportedGlobal — declares the signal as a global variable that can be accessed from outside the generated code. The signal is declared in the model package specification but not in the `External_Inputs` structure. The generated code accesses this signal as *`Signal_Name`*. The signal will be globally visible as *`Model`*`.`*`Signal_Name`*.
  - ImportedExtern — the signal is assumed to be declared in the package specification entered in the **RTW Storage Type Qualifier** field. The generated code accesses this signal as *`Your_Package`*`.`*`Signal_Name`*.
  - ImportedExternPointer — this is not permitted in Ada.
- **RTW Storage Type Qualifier** — this is only used when specifying the package specification to qualify fully the signal name for the ImportedExtern option. This field is ignored in all other cases.

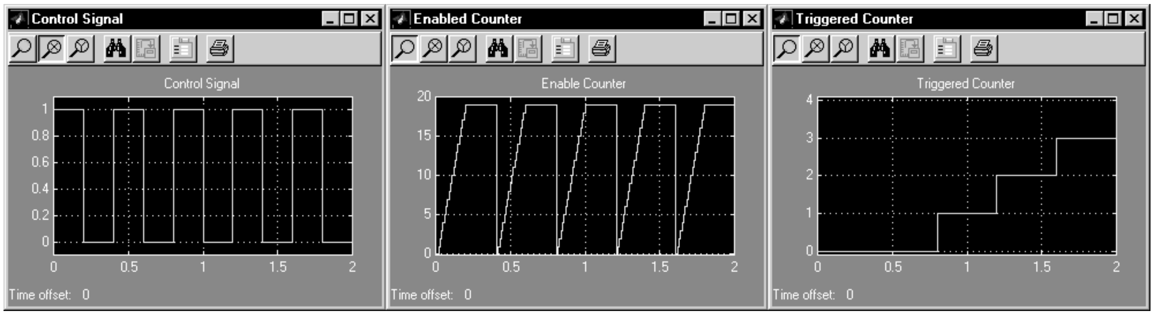
These cases are useful if you want to link Real-Time Workshop Ada Coder generated code to other Ada code (i.e., code that the Real-Time Workshop Ada Coder did not generate).

Code Validation

After completing the build process, the stand-alone version of the countersdemo model is ready for comparison with the Simulink model. The data logging options selected with the **Real-time Options** dialog box cause the program to save the control signal, enabled counter, triggered counter, and simulation time. You can now use MATLAB to produce plots of the same data that you see on the three Simulink scopes.

In both the Simulink and the stand-alone executable version of the countersdemo model, the control input is simulated with a discrete-pulse generator producing a 10 Hz, fifty percent duty cycle waveform.

Opening the control signal, enabled counter, and triggered counter scopes and running the Simulink simulation from T=0 to T=2 produces these outputs.



Type who at the MATLAB prompt to view the variable names from Simulink simulation:

```
who

Your variables are:
Enable_Signal       Triggered_Counter
Enabled_Counter     tout
```

Now run the stand-alone program from MATLAB:

```
!countersdemo
```

The “!” character passes the command that follows it to the operating system. This command, therefore, runs the stand-alone version of `countersdemo` (not the M-file).

To obtain the data from the stand-alone program, load the file `countersdemo.mat`:

```
load countersdemo
```

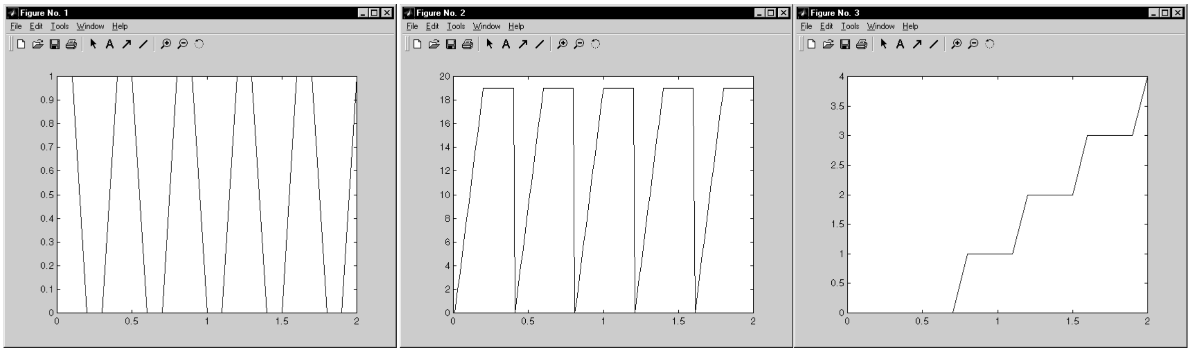
Then look at the workspace variables:

```
who
Your variables are:
Enable_Signal                rt_Triggered_Counter
Enabled_Counter              rt_tout
Triggered_Counter            tout
rt_Enable_Signal
rt_Enabled_Counter
```

The stand-alone Ada program prepends `rt_` to the logged variable names to distinguish them from the variables Simulink logged.

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

```
plot(rt_Enable_Signal(:,1),rt_Enable_Signal(:,2))
figure
plot(rt_Enabled_Counter(:,1),rt_Enabled_Counter(:,2))
figure
plot(rt_Triggered_Counter(:,1),rt_Triggered_Counter(:,2))
```



Analyzing Data with MATLAB

Points to consider when data logging:

- The Ada Coder only supports data logging to a matrix.
- To Workspace blocks log data at the frequency of the driving block and do not log time.
- Scope blocks log data at the frequency of the driving block and log time in the first column of the matrix.
- Root Outport blocks are updated at the frequency of the driving block but are logged at the base rate of the model.

Supported Blocks

The Real-Time Workshop Ada Coder supports the following Simulink blocks.

Discrete Blocks	
Discrete-Time Integrator	Discrete Zero-Pole
Discrete Filter	Unit Delay
Discrete State-Space	Zero-Order Hold
Discrete Transfer Fcn	

Functions & Tables	
Fcn	Lookup2D
Lookup	S-Function — only Target Language Compiler inlined S-functions are supported

Math Blocks	
Abs	MinMax
Combinatorial Logic	Product
Complex to Magnitude-Angle	Real-Imag to Complex
Complex to Real-Imag	Relational Operator
Dot Product	Rounding Function
Gain	Sign
Logic Operator	Slider Gain
Magnitude-Angle to Complex	Sum
Math Function	Trigonometric Function

Nonlinear Blocks	
Backlash	Quantizer
Coulomb & Viscous Friction	Relay
DeadZone	Saturation
Manual Switch (must Break Library Link and use discrete sample time)	Switch
Multiport Switch	

Signals & Systems Blocks	
Bus Selector	Hit Crossing
Configurable Subsystem	Initial Condition (IC)
DataStore Memory	Inport
DataStore Read	Merge
DataStore Write	ModelInfo
Data Type Conversion	Outport
Demux	Probe
Enable	Selector
From	Subsystem
Goto Tag Visibility	Terminator
Goto	Trigger Width
Ground	

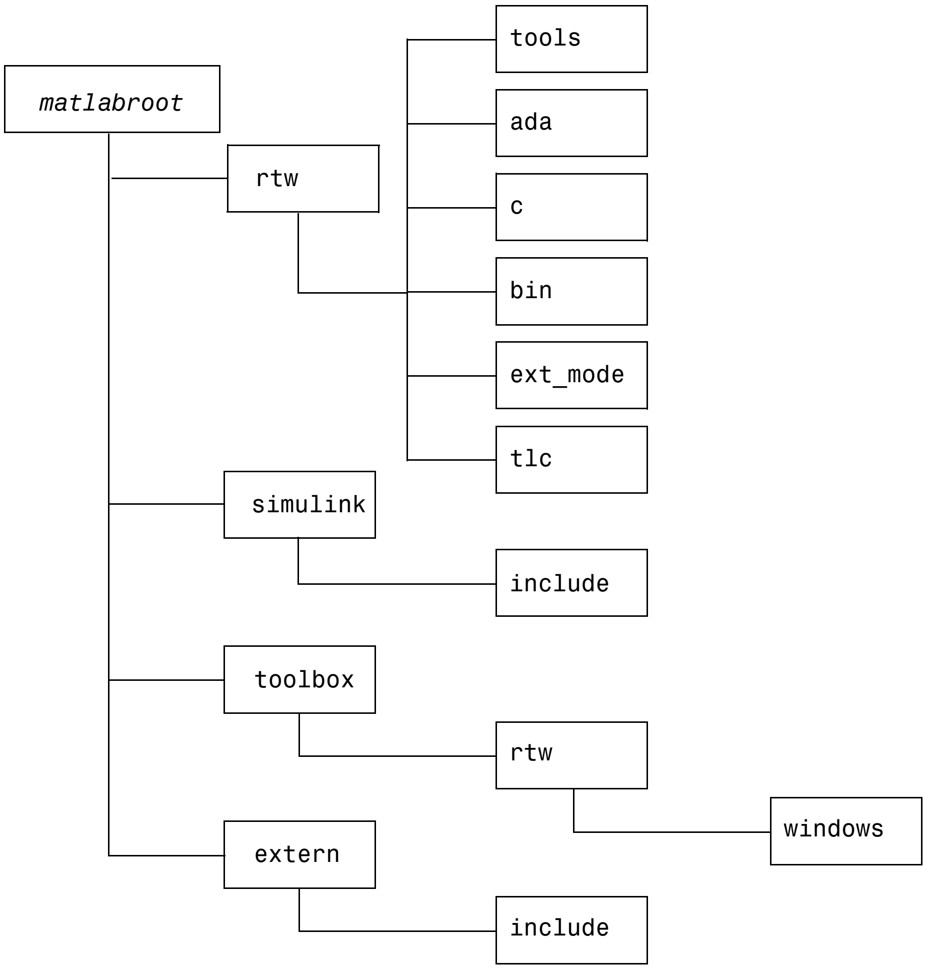
Sinks	
Display — no code is generated for this block	To File
Scope — matrix data logging only (double arrays)	To Workspace — matrix data logging only (double arrays)
Stop Simulation	

Sources	
Band-Limited White Noise	Ramp — (you must break the library link and replace the clock with a discrete clock and manually set the sample time step to match the discrete clock).
Chirp Signal — (you must break the library link and use a discrete clock)	Random Number
Constant	Sine Wave
Digital Clock	Repeating Sequence - (you must break the library link and replace the clock with a discrete clock and manually set the sample time step to match the discrete clock).
Discrete Pulse Generator	Step
From File	Uniform Random Number

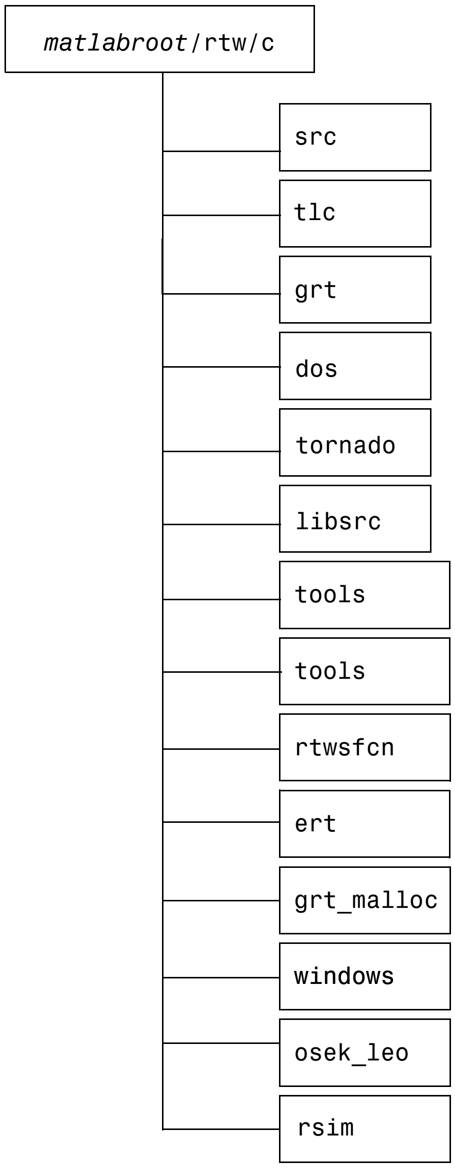
A

Real-Time Workshop Directory Tree

The files provided to implement real-time and generic real-time applications reside in the *`matlabroot`* tree, where *`matlabroot`* is the directory in which you installed MATLAB. The following diagram illustrates the basic directory structure.

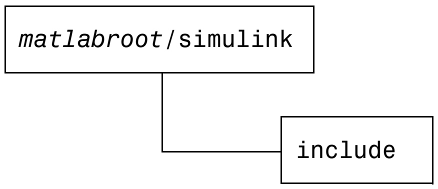


The matlabroot/rtw/c Directory



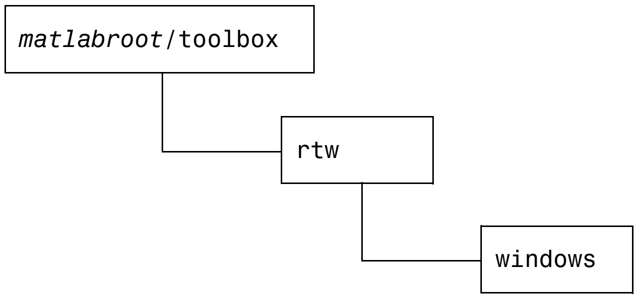
Directory	Purpose
matlabroot/rtw/c/dos	Files for targeting DOS
matlabroot/rtw/c/ert	Files for targeting embedded C code
matlabroot/rtw/c/grt	Files for targeting generic real-time
matlabroot/rtw/c/libsrc	Files used to implement functionality in certain blocks and for matrix support
matlabroot/rtw/c/grt_malloc	Files targeting generic real-time using dynamic memory allocation
matlabroot/rtw/c/osek_leo	Files for targeting the OSEK operating system
matlabroot/rtw/c/rsim	Rapid Simulation target files.
matlabroot/rtw/c/rtwsfcn	Files for targeting an S-function
matlabroot/rtw/c/src	Files for executing simulation steps, logging data, and using external mode
matlabroot/rtw/c/tlc	Target Language Compiler files for blocks and model
matlabroot/rtw/c/tools	Helper files for building targets
matlabroot/rtw/c/tornado	Files for targeting Tornado
matlabroot/rtw/c/windows	Files for targeting real-time Windows 95/98/NT

The matlabroot/simulink Directory



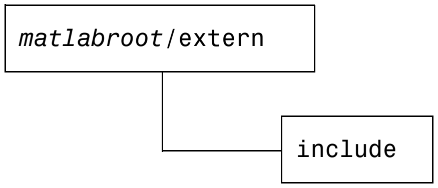
Directory	Purpose
matlabroot/simulink/ include	Include files used to build the target

The matlabroot/toolbox Directory



Directory	Purpose
matlabroot/toolbox/rtw	M-files that implement the RTW build procedure and device driver libraries
matlabroot/toolbox/rtw windows	Real-Time Windows Target I/O device drivers, device driver GUI's, and utility files

The matlabroot/extern Directory



Directory	Purpose
matlabroot/extern/include	Include files used to build the target

B

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.

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.

File Extensions – Below is a table that lists the file extensions associated with Simulink, the Target Language Compiler, and the Real-Time Workshop.

Extension	Created by	Description
`.mdl`	Simulink	Contains structures associated with Simulink block diagrams
`.rtw`	Real-Time Workshop	A translation of the `.mdl` file into an intermediate file prior to generating C code
`.tlc`	Target Language Compiler	Script files that Real-Time Workshop uses to translate
`.c`	Target Language Compiler	The generated C code
`.h`	Target Language Compiler	A C include header file used by the `.c` program
`.prm`	Target Language Compiler	A C file that contains parameter information
`.reg`	Target Language Compiler	A C include file that contains the model registration function responsible for initializing fields within the `SimStruct` (C structure)

Extension	Created by	Description
`.tmf`	Supplied with Real-Time Workshop	A template makefile
`.mk`	Real-Time Workshop	A makefile specific to your model that is derived from the template makefile

Generic Real-Time – An environment where model code is generated for a real-time system, and the resulting code is simulated on your workstation. (Note that execution is not tied to a real-time clock.) You can use generic real-time as a starting point for targeting custom hardware.

Host System – The computer system on which you create 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 the 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-Functions – S-functions can be inlined into the generated code by implementing them as a `.tlc` file. The implementation of this S-function would then be in the generated code itself. In contrast, noninlined S-functions require a function call to the S-function code from which you created your 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 block 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 the `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 the 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.

Nonvirtual Block – Any block that performs some algorithm, such as a Gain block.

Pseudomultitasking – In processors that do not offer multitasking support, you can perform pseudomultitasking by scheduling events on a fixed time-sharing basis.

Real-Time System – A system that uses actual hardware to implement algorithms, for example, digital signal processing or control applications.

Run-time Interface or Run-Time Support Files – A wrapper around the generated code that can be built into a stand-alone executable. These support files consist 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 or M-code. S-functions can be inlined in the Real-Time Workshop.

Singletasking – A mode in which a mode is run 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 Language Compiler – A compiler that compiles and executes system and target files.

Target File – A file that is compiled and executed by the Target Language Compiler. A combination of these files describes how to transform the Real-Time Workshop file (`model.rtw`) into target-specific code.

Target System – The computer system on which you execute your real-time application.

Targeting – The process of creating an executable for your target system.

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.

Target Language Compiler Program – A set of TLC files that describe how to convert a `model.rtw` file into generated code.

TID (**task id**) – Each sample time in your model is assigned a task id. The task id is passed to the model output and update routines to decide which portion of your model should be executed at a given time.

Virtual Block – A connection or graphical block, for example, a Mux block.

Index