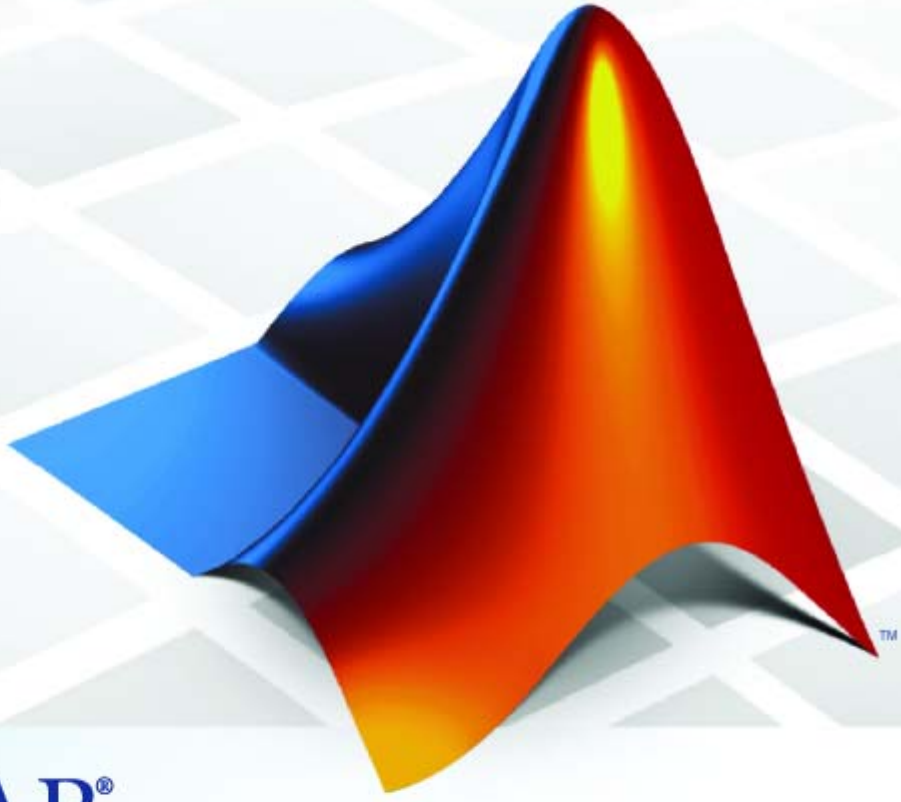
Real-Time Workshop® Embedded Coder™ 5

Developing Embedded Targets



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

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

Real-Time Workshop® Embedded Coder™ Developing Embedded Targets

Trademarks

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

Patents

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

Revision History

December 2002	Online only	Version 3.0 (Release 13)
June 2004	Online only	Revised for Version 4.0 (Release 14)
October 2004	Online only	Revised for Version 4.1 (Release 14SP1)
September 2005	Online only	Revised for Version 4.3 (Release 14SP3)
March 2006	Online only	Revised for Version 4.4 (Release 2006a)
September 2006	Online only	Revised for Version 4.5 (Release 2006b)
March 2007	Online only	Revised for Version 4.6 (Release 2007a)
September 2007	Online only	Revised for Version 5.0 (Release 2007b)
March 2008	Online only	Revised for Version 5.1 (Release 2008a)
October 2008	Online only	Revised for Version 5.2 (Release 2008b)
March 2009	Online only	Revised for Version 5.3 (Release 2009a)
September 2009	Online only	Revised for Version 5.4 (Release 2009b)

Contents

1 Introduction

2 Overview of Embedded Target Development

3 Target Development Mechanics

4 Target Directories, Paths, and Files

5 System Target Files

6 Template Makefiles

7 Supporting Optional Features

8 Using Target Preferences

9 Interfacing to Development Tools

10 Developing Device Drivers for Embedded Targets

1

Introduction

The purpose of this document is to guide you in the development of a custom embedded target for use with Real-Time Workshop® Embedded Coder™ software. This document identifies requirements, implementation tasks, and implementation details for target creation.

The following sections summarize the prerequisites for using this document and list sources of additional information related to embedded target development:

Prerequisites

Custom target creation is a topic for advanced Real-Time Workshop® and Real-Time Workshop Embedded Coder users. This document assumes you have MATLAB®, Simulink®, Real-Time Workshop, and Real-Time Workshop Embedded Coder experience.

This document assumes that you will be developing a target based on the Embedded Real-Time (ERT) target that is included in the Real-Time Workshop Embedded Coder software. The target features and technologies described in this document are subject to change in future Real-Time Workshop Embedded Coder releases.

You should be familiar with the following products and their documentation before reading this document:

- MATLAB and M-file programming
- Simulink
- Real-Time Workshop and its code generation and build process
- Real-Time Workshop Embedded Coder
- Real-Time Workshop Target Language Compiler (TLC)

Familiarity with the Stateflow® product may be helpful, but is not required.

Related Documentation

This document supplements information contained in other Real-Time Workshop Embedded Coder, Real-Time Workshop, and Simulink documents. The following documents provide additional information:

- Real-Time Workshop Embedded Coder documentation: You should be thoroughly familiar with this detailed Real-Time Workshop Embedded Coder documentation and the ERT target. Important topics covered include ERT model execution, timing, and task management; how to interface to and call model code; and default ERT code generation options.
- Real-Time Workshop Getting Started Guide document: General introduction to Real-Time Workshop features. The "Getting Started with Real-Time Workshop Technology" and "How to Develop an Application Using Real-Time Workshop Software" chapters include high-level overviews of target files and the build process.
- Real-Time Workshop documentation: The detailed Real-Time Workshop documentation covers several topics of interest to some target developers:
  - Inlining and code generation issues relevant to device drivers and other S-functions
  - Interfacing signals and parameters within generated code to your own code
  - Combining code generated from multiple models into a single system
  - Implementing external mode communication via your own low-level protocol layer
- Real-Time Workshop Target Language Compiler document: A working knowledge of TLC is needed if you intend to make nontrivial modifications to your system target file, use TLC hooks into the build process, utilize information from the *model*`.rtw` file, implement inlined device drivers, or pass information into or out of the TLC phase of the build process. Minimally, you should work through the introductory sections, including "Getting Started".
- Simulink Writing S-Functions document: Familiarity with writing fully inlined S-functions is required if you intend to develop device driver blocks for your target. "Building S-Functions Automatically" documents the S-Function Builder.

Embedded Target Implementations to Study

You should also consider becoming familiar with the documentation for the Target Support Package™ product. If you do not have a license for the product, you can access the documentation from the MathWorks™ Web site.

2

Overview of Embedded Target Development

Introduction

The targets bundled with the Real-Time Workshop product are suitable for many different applications and development environments. Third-party targets provide additional versatility. However, you might want to implement a custom target for any of the following reasons:

- To enable end users to generate executable production code for a specific CPU or development board, using a specific development environment (compiler/linker/debugger).
- To support I/O devices on the target hardware by incorporating custom device driver blocks into your models.
- To configure the build process for a special compiler (such as a cross-compiler for an embedded microcontroller or DSP board) or development/debugging environment.

The Real-Time Workshop Embedded Coder product provides a point of departure for the creation of custom embedded targets, for the basic purposes above. This manual covers the tasks and techniques you need to implement a custom embedded target.

Types of Targets

In this section...

Introduction

Before considering the specific components, features, and capabilities that should be included in an embedded target, let's consider several types of targets intended for different use cases.

The target types discussed below are not mutually exclusive. A given embedded target can support more than one of these use cases, or additional uses not outlined here. Also, there is a progression of capabilities from the first (baseline) to second (turnkey production) target types; you may want to implement an initial baseline target and a following, more full-featured turnkey version of a target.

The discussion of target types is followed by "Recommended Features for Embedded Targets" on page 2-6, which contains a suggested list of target features and general guidelines for embedded target development.

Baseline Targets

A *baseline target* offers a starting point for targeting a production processor. A baseline target integrates Real-Time Workshop Embedded Coder software with one or more popular cross-development environments (compiler/linker/debugger tool chains). A baseline target provides a starting point from which you can customize the target for application needs.

Target files provided for this type of target should be readable, easy to understand, and fully commented and documented. Specific attention should be paid to the interface to the intended cross-development environment.

This interface should be implemented using the preferred approach for that particular development system. For example, some development environments use traditional make utilities, while others are based on project-file builds that can be automated under control of the Real-Time Workshop software.

When you use a baseline target, you need to include your own device driver and legacy code and modify linker memory maps to suit your needs. You should be familiar with the targeted development system.

Turnkey Production Targets

A *turnkey production target* also targets a production processor, but includes the capability to create target executables that interact immediately with the external world. In general, ease of use is more important than simplicity or readability of the target files, because it is assumed that you do not want or need to modify these files.

Desirable features for a turnkey production target include

- Significant I/O driver support provided out of the box
- Easy downloading of generated standalone executables with third-party debuggers
- User-controlled placement of an executable in FLASH or RAM memory
- Support for target visibility and tuning

HIL Simulation Targets

A specialized use case is the generation of executables intended for use in *Hardware-In-the-Loop* (HIL) simulations. In a HIL simulation, parts of a pure simulation are gradually replaced with hardware components as components are refined and fabricated. HIL simulation offers an efficient design process that eliminates costly iterations of part fabrication.

PIL Cosimulation Targets

Another specialized use case is the generation of executables intended for use in *Processor-In-the-Loop* (PIL) cosimulation. In a PIL cosimulation, a subsystem runs on target hardware, but within the context of a Simulink

simulation. Cosimulation can be useful for validation of generated code and in validating the target compiler/processor environment at the subsystem unit level.

Recommended Features for Embedded Targets

In this section...

Basic Target Features

- Targets should be based on the Embedded Real-Time (ERT) target that is included in the Real-Time Workshop Embedded Coder product. The features documented in this guide are available in the corresponding release of the Real-Time Workshop Embedded Coder product (release versions are listed in the Revision History found in the front matter of the PDF version of the guide).

  Since your target is based on the ERT target, it should use that target's Embedded-C code format, and should inherit the options defined in the ERT target's system target file. By following these recommendations, you ensure that your target has all the production code generation capabilities of the ERT target.

  See Chapter 5, "System Target Files" for further details on the inheritance mechanism, setting the code format, and other details.

- The most fundamental requirement for an embedded target is that it generate a real-time executable from a model or subsystem. Typically, an embedded target generates a timer interrupt-based, bareboard executable (although targets can be developed for an operating system environment as well).

  Your target should support the Real-Time Workshop concepts of singletasking and multitasking solver modes for model execution. Tasking support comes almost "for free" with the ERT target, but you should thoroughly understand how it works before implementing an ERT-based target.

Implementation of timer interrupt-based execution is documented in the "Developing Models for Code Generation" chapter of the Real-Time Workshop Embedded Coder documentation.

- You should generate the target executable's main program module, rather than using a static main module (such as the static `ert_main.c` module provided with the Real-Time Workshop Embedded Coder product). A generated `main.c` or `.cpp` can be made much more readable and more efficient, since it omits preprocessor checks and other extra code.

  See the Real-Time Workshop Embedded Coder documentation for information on generated and static main program modules.

- You should use the target preferences mechanism (see Chapter 8, "Using Target Preferences") to store and configure information about the development environment a user selects and other persistent data associated with your target.

- Follow the guidelines in Chapter 4, "Target Directories, Paths, and Files" to set up a file and directory structure that is consistent with other targets. Consistency between different targets is important and reduces the effort required to create and understand a target.

Integration with Target Development Environments

- Most cross-development systems run under a Microsoft® Windows® PC host. Your target should support the Windows XP operating system as the host environment.

  Some cross-development systems support one or more versions of The Open Group UNIX® platforms, allowing for UNIX host support as well.

- Your embedded target must support at least one embedded development environment. The interface to a development environment can take one of several forms. The most common approach is to use a template makefile to generate standard makefiles with the make utility provided with your development environment. Chapter 6, "Template Makefiles" describes the structure of template makefiles.

  Another approach with IDE-based tools is project file creation and/or Microsoft Windows Component Object Model (COM) automation.

It is important to consider the license requirements and restrictions of the development environment vendor. You may need to modify files provided by the vendor and ship them as part of the embedded target.

See Chapter 9, "Interfacing to Development Tools" for further information.

Observing Execution of Target Code

- Your target should support a mechanism you can use to observe the target code as it runs in real time (outside of a debugger).

  One industry-standard approach is to use the CAN bus, with an ASAP2 file and CAN Calibration Protocol (CCP). There are several host-based graphical front-end tools available that connect to a CCP-enabled target and provide data viewing and parameter tuning. Supporting these tools requires implementation of CAN hardware drivers and CCP protocol for the target, as well as ASAP2 file generation. Your target can leverage the ASAP2 support provided with the Real-Time Workshop Embedded Coder product.

  Another option is to support Simulink External Mode over a serial interface (RS-232). See the Real-Time Workshop documentation for information on using the external mode API.

Deployment and Hardware Issues

- Device driver support is an important issue in the design of an embedded target. Device drivers are Simulink blocks that support either hardware I/O capabilities of the target CPU, or I/O features of the development board.

  If you are developing a baseline target, consider providing minimal driver support, on the assumption that end users develop their own drivers. If you are developing a turnkey production target, you should provide full driver support. See Chapter 10, "Developing Device Drivers for Embedded Targets" for a detailed discussion of device drivers.

- Automatic download of generated code to the target hardware makes a target easier to use. Typically a debugger utility is used; if the chosen debugger supports command script files, this can be straightforward to implement. "STF_make_rtw_hook.m" on page 4-12 describes a mechanism to execute M-code from the build process. You can use this mechanism to

make `system()` calls to invoke utilities such as a debugger. You can invoke other simple downloading utilities in a similar fashion.

If your development system supports COM automation, you can control the download process by that mechanism. Using COM automation is discussed in Chapter 9, “Interfacing to Development Tools”.

- Executables that are mapped to RAM memory are typical. You can provide optional support for FLASH or RAM placement of the executable by using your target’s code generation options. To support this capability, you might need multiple linker command files, multiple debugger scripts, and possibly multiple makefiles or project files. The ability to automatically switch between these files, depending on the RAM/FLASH option value, is also needed.
- Select a popular, widely available evaluation or prototype board for your target processor. Consider enclosed and ruggedized versions of the target board. Also consider board level support for the various on-chip I/O capabilities of the target CPU, and the availability of development systems that support the selected board.

3

Target Development Mechanics

Components of a Custom Target

In this section...

Overview

The components of a custom target are files located in a hierarchy of directories. The top-level directory in this structure is called the *target root directory*. The target root directory and its contents are named, organized, and located on the MATLAB path according to conventions described in Chapter 4, "Target Directories, Paths, and Files".

The components of a custom target include

- Code components: C source code that supervises and supports execution of generated model code.
- Control files:
  - A system target file (STF) to control the code generation process.
  - File(s) to control the building of an executable from the generated code. In a traditional make-based environment, a template makefile (TMF) generates a makefile for this purpose. Another approach is to generate project files in support of a modern integrated development environment (IDE) such as the Freescale™ Semiconductor CodeWarrior® IDE.
  - Hook files: Optional TLC and M-files that can be invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.
- Target preferences files: These files define a *target preferences class* associated with your target. Your target preference class lets you create data objects that define and store properties associated with your target. For example, you may want to store a user-defined path to a cross-compiler that is invoked by the build process. The target preferences mechanism is described in Chapter 8, "Using Target Preferences".

- Other target files: Files that let you integrate your target into the MATLAB environment. For example, you can provide an `info.xml` file to make your target block libraries, demos, and target preferences available from the MATLAB **Start** menu.

The next sections introduce key concepts and terminology you need to know to develop each component. References to more detailed information sources are provided.

Code Components

A Real-Time Workshop program containing code generated from a Simulink model consists of a number of code modules and data structures. These fall into two categories.

Application Components

Application components are those which are specific to a particular model; they implement the functions represented by the blocks in the model. Application components are not specific to the target. Application components include

- Modules generated from the model
- User-written blocks (S-functions)
- Parameters of the model that are visible, and can be interfaced to, external code

Run-Time Interface Components

A number of code modules and data structures, referred to collectively as the *run-time interface*, are responsible for managing and supporting the execution of the generated program. The run-time interface modules are not automatically generated. Depending on the requirements of your target, you must implement certain parts of the run-time interface. Run-Time Interface Components on page 3-4 summarizes the run-time interface components.

Run-Time Interface Components

You Provide...	The Real-Time Workshop Software Provides...
Customized main program	Generic main program
Timer interrupt handler to run model	Execution engine and integration solver (called by timer interrupt handler)
Other interrupt handlers	Example interrupt handlers (Asynchronous Interrupt blocks)
Device drivers	Example device drivers
Data logging, parameter tuning, signal monitoring, and external mode support	Data logging, parameter tuning, signal monitoring, and external mode APIs

User-Written Run-Time Interface Code

The Real-Time Workshop software provides most of the run-time interface. Depending on the requirements of your target, you must implement some or all of the following elements:

- A timer *interrupt service routine* (ISR). The timer runs at the program's base sample rate. The timer ISR is responsible for operations that must be completed within a single clock period, such as computing the current output sample. The timer ISR usually calls the Real-Time Workshop `rt_OneStep` function.

  If you are targeting a real-time operating system (RTOS), your generated code usually executes under control of the timing and task management mechanisms provided by the RTOS. In this case, you may not have to implement a timer ISR.

- The *main program*. Your main program initializes the blocks in the model, installs the timer ISR, and executes a background task or loop. The timer periodically interrupts the main loop. If the main program is designed to run for a finite amount of time, it is also responsible for cleanup operations — such as memory deallocation and masking the timer interrupt — before terminating the program.

If you are targeting a real-time operating system (RTOS), your main program most likely spawns tasks (corresponding to the sample rates used in the model) whose execution is timed and controlled by the RTOS.

Your main program typically is based on the Real-Time Workshop Embedded Coder main program, `ert_main.c`. The Real-Time Workshop Embedded Coder documentation details the structure of the Real-Time Workshop Embedded Coder run-time interface and the execution of Real-Time Workshop Embedded Coder code, and provides guidelines for customizing `ert_main.c`.

- *Device drivers.* Drivers communicate with I/O devices on your target hardware. In production code, device drivers are normally implemented as inlined S-functions.
- *Other interrupt handlers.* If your models need to support asynchronous events, such as hardware generated interrupts and asynchronous read and write operations, you must supply interrupt handlers. The Real-Time Workshop Interrupt Templates library provides examples.
- *Data logging, parameter tuning, signal monitoring, and external mode support.* It is atypical to implement rapid prototyping features such as external mode support in an embedded target. However, it is possible to support these features by using standard Real-Time Workshop APIs. See the Real-Time Workshop documentation for details.

Control Files

The code generation and build process is directed by a number of TLC and M-files collectively called *control files*. This section introduces and summarizes the main control files.

Top-Level Control File (make_rtw)

The build process is initiated when you click **Build** (or type **Ctrl+B**). At this point, Real-Time Workshop build process parses the **Make command** field of the **Real-Time Workshop** target configuration pane, expecting to find the name of a top-level M-file command that controls the build process (as well as optional arguments to that command). The default top-level control file for the build process is `make_rtw.m`.

Normally, target developers do not need detailed knowledge of how `make_rtw` works. (The details that are necessary to target developers are described in

"Understanding and Using the Build Process" on page 3-8.) You should not customize `make_rtw.m`. The `make_rtw.m` file contains all the logic required to execute your target-specific control files, including a number of hook points for execution of your custom code.

`make_rtw` does the following:

- Passes optional arguments in to the build process
- Performs any required preprocessing before code generation
- Executes the STF to perform code generation (and optional HTML report generation)
- Processes the TMF to generate a makefile
- Invokes a make utility to execute the makefile and build an executable
- Performs any required post-processing (such as generating calibration data files or downloading the generated executable to the target)

System Target File (STF)

The Target Language Compiler (TLC) generates target-specific C or C++ code from an intermediate description of your Simulink block diagram (*`model`*`.rtw`). The Target Language Compiler reads *`model`*`.rtw` and executes a program consisting of several target files (`.tlc` files.) The STF, at the top level of this program, controls the code generation process. The output of this process is a number of source files, which are fed to your development system's make utility.

You need to create a customized STF to set code generation parameters for your target. You should copy, rename, and modify the standard ERT system target file (*`matlabroot`*`/rtw/c/ert/ert.tlc`).

The detailed structure of the STF is described in Chapter 5, "System Target Files".

Template Makefile (TMF)

A TMF provides information about your model and your development system. The Real-Time Workshop build process uses this information to create an appropriate makefile (`.mk` file) to build an executable program.

Some targets implement more than one TMF, in order to support multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable vs. generating a project file for your compiler).

The Real-Time Workshop Embedded Coder software provides a large number of TMFs suitable for different types of host-based development systems. These TMFs are located in *`matlabroot`*`/rtw/c/ert`. The standard TMFs are described in the "Template Makefiles and Make Options" section of the Real-Time Workshop documentation.

The detailed structure of the TMF is described in Chapter 6, "Template Makefiles".

Hook Files

Real-Time Workshop build process allows you to supply optional *hook files* that are executed at specified points in the code generation and make process. You can use hook files to add target-specific actions to the build process.

To ensure that hook files are called correctly by the build process, they must follow well-defined naming and location requirements. Chapter 4, "Target Directories, Paths, and Files" describes these requirements.

Understanding and Using the Build Process

In this section...

Introduction

To develop an embedded target, you need a thorough understanding of the Real-Time Workshop build process. Your embedded target uses the build process and may require you to modify or customize the process. A general overview of the build process is given in the "How to Develop an Application Using Real-Time Workshop Software" chapter of the Real-Time Workshop Getting Started Guide.

This section supplements that overview with a detailed flowchart of the build process as customized by the Real-Time Workshop Embedded Coder software. The emphasis is on points in the process where customization hooks are available and on passing information between different phases of the process.

This section concludes with "Additional Information Passing Techniques" on page 3-15, describing assorted tips and tricks for passing information during the build process.

Build Process Phases and Information Passing

It is important to understand where (and when) the build process obtains required information. Sources of information include

- The *model*`.rtw` file, which provides information about the generating model. All information in *model*`.rtw` is available to target TLC files.
- The Real-Time Workshop related panes of the Configuration Parameters dialog box. Options (both general and target-specific) are provided through check boxes, menus, and edit fields. You can associate options with TLC variables and makefile tokens in the `rtwoptions` data structure.

- The target preferences data. Target preferences provide persistent information about the target, such as the location of your development tools.
- The template makefile (TMF), which generates the model-specific makefile.
- Environment variables on the host computer. Environment variables provide additional information about installed development tools.
- Other target-specific files such as target-related TLC files, linker command files, or project files.

It is also important to understand the several phases of the build process and how to pass information between the phases. The build process comprises several high-level phases:

- Execution of the top-level M-file (`slbuild.m` or `rtwbuild.m`) to sequence through the build process for a target
- Conversion of the model into the TLC input file (*`model`*`.rtw`)
- Generation of the target code by the TLC compiler
- Compilation of the generated code with `make` or other utilities
- Transmission of the final generated executable to the target hardware with a debugger or download utility

It is helpful to think of each phase of the process as a different "environment" that maintains its own data. These environments include

- M-code execution environment (MATLAB)
- Simulink
- Target Language Compiler execution environment
- makefile
- Development environments such as and IDE or debugger

In each environment, information may be needed from the various sources mentioned above. For example, during the TLC phase, it may be necessary to execute an M-file to obtain information from the MATLAB environment. Also, a given phase may generate information that is needed in a subsequent phase.

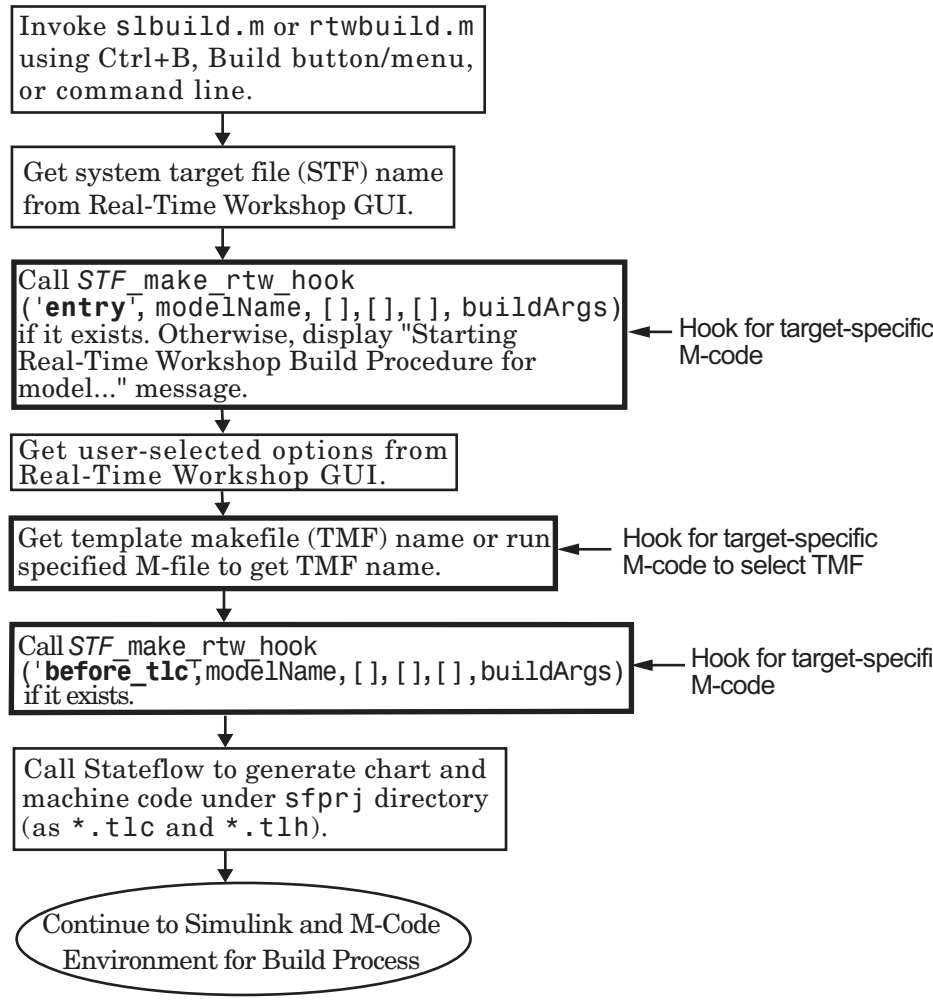
Build Process Flowchart

The following flowcharts detail the build process as a sequence of actions that execute within several environments:

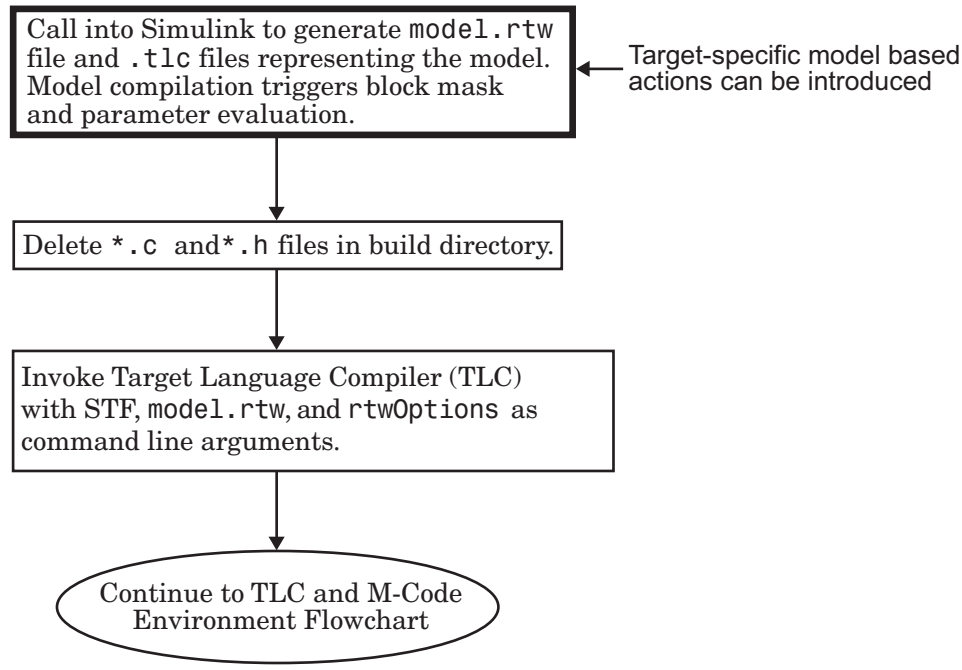
- "MATLAB Environment for Build Process" on page 3-11 depicts the initial M-code execution phase.
- "Simulink and M-Code Environment for Build Process" on page 3-12 depicts the Simulink model compilation phase and M-code execution following it.
- "TLC and M-Code Environment Flowchart" on page 3-13 depicts the main TLC code generation phase and M-code execution following it.
- "M-Code, model.bat, and Makefile Environment Flowchart" on page 3-14 depicts the final M-code, *`model`*`.bat`, and make phase.

In the flowcharts, bold rectangles and oval balloons indicate points where different environments can interact by using hooks or other mechanisms for information passing. See "Key Files in the Target Directory (mytarget/mytarget)" on page 4-10 for details on the available M-file and TLC hooks, with code examples.

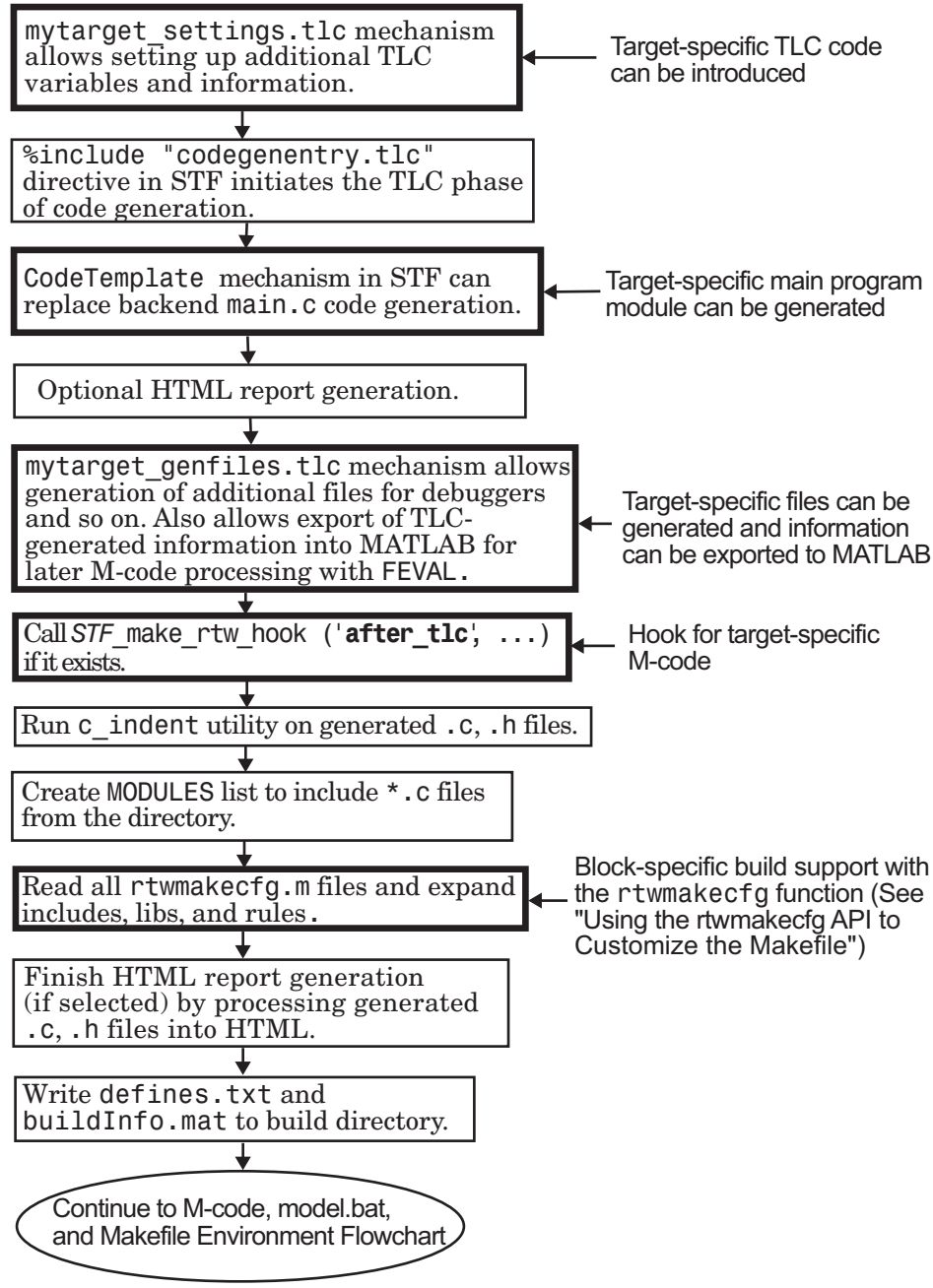
MATLAB Environment for Build Process



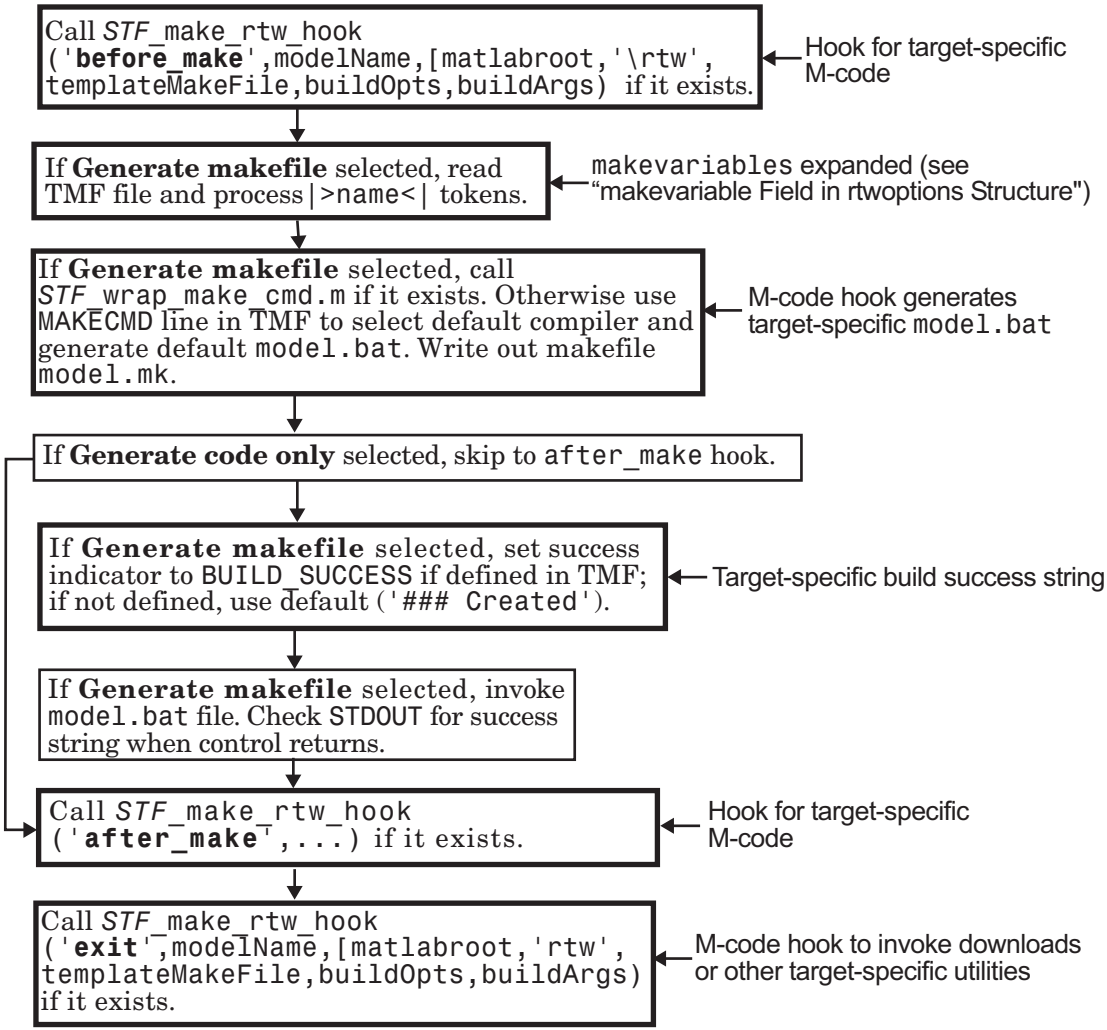
Simulink and M-Code Environment for Build Process



TLC and M-Code Environment Flowchart



M-Code, model.bat, and Makefile Environment Flowchart



Additional Information Passing Techniques

This section describes a number of useful techniques for passing information among different phases of the build process.

tlcvariable Field in rtwoptions Structure

Options on the Real-Time Workshop related panes of the Configuration Parameters dialog box can be associated with a TLC variable, and specified in the `tlcvariable` field of the option's entry in the `rtwoptions` structure. The variable value is passed on the command line when TLC is invoked. This provides a way to make Real-Time Workshop options and their values available in the TLC phase.

See "System Target File Structure" on page 5-4 for further information.

makevariable Field in rtwoptions Structure

Similarly, Real-Time Workshop options can be associated with a template makefile token, specified in the `makevariable` field of the option's entry in the `rtwoptions` structure. If a token of the same name as the `makevariable` name exists in the TMF, the token is updated with the option value when the final makefile is created. If the token does not exist in the TMF, the `makevariable` is passed in on the command line when make is invoked. Thus, in either case, the `makevariable` is available to the makefile.

See "System Target File Structure" on page 5-4 for further information.

Accessing Host Environment Variables

You can access host shell environment variables at the MATLAB command line by entering the `getenv` command. For example:

```
getenv ('MSDEVDIR')

ans =

D:\Applications\Microsoft Visual Studio\Common\MSDev98
```

To access the same information from TLC, use the `FEVAL` directive to invoke `getenv`.

```
%assign eVar = FEVAL("getenv", "<varname>").
```

Supplying Development Environment Information to Your Template Makefile

An embedded target must tie the build process to target-specific development tools installed on a host computer. For the make process to run these tools correctly, the TMF must be able to determine the name of the tools, the path to the compiler, linker, and other utilities, and possibly the host operating system environment variable settings. This section describes two techniques for supplying this information.

The simpler, more traditional approach is to require the end user to modify the target TMF. The user enters path information (such as the location of a compiler executable), and possibly host operating system environment variables, as make variables. This allows the TMF to be tailored to specific needs.

This approach is not satisfactory in an environment where the MATLAB installation is on a network and multiple users share read-only TMFs. Another possible drawback to this approach is that the tool information is only available during the makefile processing phase of the build process.

A second approach is to use the target preferences feature (see Chapter 8, "Using Target Preferences") together with the *STF*_wrap_make_cmd_hook mechanism (see "STF_wrap_make_cmd_hook Mechanism" on page 4-13). In this approach, compiler and other tool path information is stored as preferences data, which is obtained by the *STF*_wrap_make_cmd_hook.m file. This allows tool path information to be saved separately for each user.

Another advantage to the second approach is that target preferences data is available to all phases of the build process, including the TLC phase. This information may be required to support features such as RAM/ROM profiling.

Using MATLAB Application Data

Application data provides a way for applications to save and retrieve data stored with the GUI. This technique enables you to create what is essentially a user-defined property for an object, and use this property to store data for use in the build process. If you are unfamiliar with this technique, see

the “Application Data” section of the MATLAB Creating Graphical User Interfaces document.

The following code examples illustrates the use of application data to pass information to TLC.

This M-file, `tlc2appdata.m`, stores the `data` passed in as application data under the name passed in (`appDataName`).

```
function k = tlc2appdata(appDataName, data)
  disp([mfilename,': ',appDataName,' ', data]);
  setappdata(0,appDataName,data);
  k = 0;  % TLC expects a return value for FEVAL.
```

The following sample TLC file uses the `FEVAL` directive to invoke `tlc2appdata.m` to store arbitrary application data, under the name `z80`.

```
%% test.tlc
%%
%assign myApp = "z80"
%assign myData = "314159"
%assign dummy = FEVAL("tlc2appdata",myApp,myData)
```

To test this technique:

1. Create the `tlc2appdata.m` M-file as shown. Make sure that `tlc2appdata.m` is stored in a directory on the MATLAB path.
2. Create the TLC file as shown. Save it as `test.tlc`.
3. Enter the following command at the MATLAB prompt to execute the TLC file:

   ```
   tlc test.tlc
   ```

4. Get the application data at the MATLAB prompt:

   ```
   k = getappdata(0,'z80')
   ```

   The function returns the value 314159.

5 Enter the following command.

```
who
```

Note that application data is not stored in the MATLAB workspace. Also observe that the z80 data is not visible. Using application data in this way has the advantage that it does not clutter the MATLAB workspace. Also, it helps prevent you from accidently deleting your data, since it is not stored directly in the your workspace.

A real-world use of application data might be to collect information from the *model*`.rtw` file and store it for use later in the build process.

Adding Block-Specific Information to the Makefile

The `rtwmakecfg` mechanism provides a method for inlined S-functions such as driver blocks to add information to the makefile. This mechanism is described in "Using the rtwmakecfg.m API to Customize Generated Makefiles" on page 6-17.

4

Target Directories, Paths, and Files

Introduction

Your initial tasks in setting up an embedded target are

- Create a target directory structure
- Include desired directories in the MATLAB path
- Create the required target files and locate them in your target directories. In some cases you modify files provided by the Real-Time Workshop Embedded Coder software.

The following sections explain how to organize your target directories and files and add them to the your MATLAB path. They also provide high-level descriptions of the files to be stored in each directory of the structure.

You should follow the conventions described. By doing so, you can make your embedded targets consistent, easy to understand, and efficient. The conventions in this section provide guidelines for the root target directory and key directories immediately under it. You can, of course, define further subdirectories if your target is complex or if you need a more modular structure.

Directory and File Naming Conventions

For an actual target implementation, the recommended directory and file naming conventions are

- Use the name of the target processor (for example, `c166`).
- For subdirectories containing files associated with specific development environments or tools, use the name of the tool (for example, `codewarrior`).
- Use *only* lowercase in all directory names, filenames, and extensions.
- Do not embed spaces in directory names. Spaces in directory names cause errors with many third-party development environments.

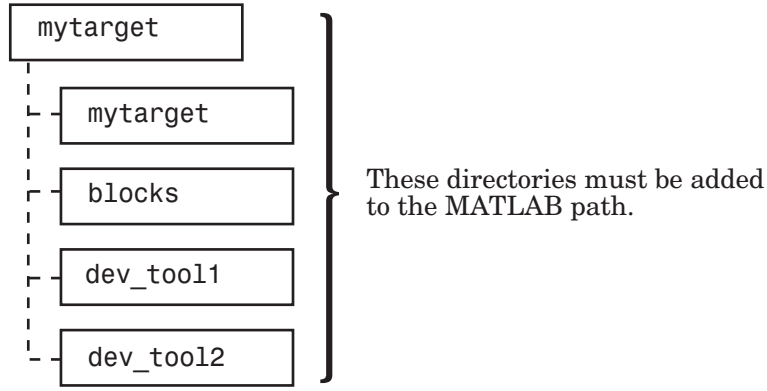
In this document, `mytarget` is a placeholder name that represents directories and files that use the target's name. The names `dev_tool1`, `dev_tool2`, and so on represent subdirectories containing files associated with development environments or tools.

Target Directory Structure and MATLAB Path

In this section...

Overview

You should create a directory structure like that shown in Recommended Target Directory Structure on page 4-4 for your target files. The top-level directory in this structure, `mytarget`, is the *target root directory*.



Recommended Target Directory Structure

The contents of the target root directory and its subdirectories (as well as optional additional directories) are discussed in "Key Directories Under the Target Root (mytarget)" on page 4-6.

Adding Target Directories to the MATLAB Path

The directories shown in Recommended Target Directory Structure on page 4-4 must be added to the MATLAB path.

The directories labeled `dev_tool1` and `dev_tool2` in Recommended Target Directory Structure on page 4-4 contain files associated with specific development environments or tools (`dev_tool1`, `dev_tool2`, etc.) that are supported by your target.

Location of Target Directories

All directory names and filenames must meet the requirements listed in "Directory and File Naming Conventions" on page 4-3. Note carefully the following rules for locating your target directories:

- For embedded targets developed by The MathWorks™ that are included in the MATLAB installation, the target root directory should be located under *`matlabroot`*`/toolbox/rtw/targets/`.
- For embedded targets *not* developed by The MathWorks, the target root directory should *not* be located anywhere in the MATLAB directory tree (that is, in or under the *`matlabroot`* directory). This restriction exists because installing a new MATLAB version (or reinstalling the current version) recreates the MATLAB directories, which deletes any custom target directories existing within the MATLAB tree.

Key Directories Under the Target Root (mytarget)

In this section...

Target Root Directory (mytarget)

This directory contains the key subdirectories for the target (see Recommended Target Directory Structure on page 4-4). You can also locate miscellaneous files (such as a `readme` file) in the target root directory. The following sections describe required and optional subdirectories and their contents.

Target Directory (mytarget/mytarget)

This directory contains files that are central to the target, such as the system target file (STF) and template makefile (TMF). "Key Files in the Target Directory (mytarget/mytarget)" on page 4-10 summarizes the files that should be stored in `mytarget/mytarget`, and provides pointers to detailed information about these files.

Note `mytarget/mytarget` should be on the MATLAB path.

Target Block Directory (mytarget/blocks)

If your target includes device drivers or other blocks, locate the block implementation files in this directory. `mytarget/blocks` contains

- Compiled block MEX-files

- Source code for the blocks
- TLC inlining files for the blocks
- Library models for the blocks (if you provide your blocks in one or more libraries)

Note `mytarget/blocks` should be on the MATLAB path.

You can also store demo models and any supporting M-files in `mytarget/blocks`. Alternatively, you can create a `mytarget/mytargetdemos` directory, which should also be on the MATLAB path.

To display your blocks in the standard Simulink Library Browser and/or integrate your demo models into the standard **Demos** in the Help contents and **Start** button, you can create the files described below and store them in `mytarget/blocks`.

mytarget/blocks/slblocks.m

This file allows a group of blocks to be integrated into the Simulink Library and Simulink Library Browser.

Example slblocks.m File

```
function blkStruct = slblocks
% Information for "Blocksets and Toolboxes" subsystem
blkStruct.Name = sprintf('Embedded Target\n for MYTARGET');
blkStruct.OpenFcn = 'mytargetlib';
blkStruct.MaskDisplay = 'disp(''MYTARGET'')';

% Information for Simulink Library Browser
Browser(1).Library = 'mytargetlib';
Browser(1).Name    = 'Embedded Target for MYTARGET';
Browser(1).IsFlat  = 1;% Is this library "flat" (i.e. no subsystems)?

blkStruct.Browser = Browser;
```

mytarget/blocks/demos.xml

This file provides information about the components, organization, and location of demo models. MATLAB software uses this information to place the demo in the appropriate place in the Help contents and **Start** button.

Example demos.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<demos>
   <name>Embedded Target for MYTARGET</name>
   <type>simulink</type>
   <icon>$toolbox/matlab/icons/boardicon.gif</icon>
   <description source = "file">mytarget_overview.html</description>

   <demosection>
      <label>Multirate model</label>
      <demoitem>
         <label>MYTARGET demo</label>
         <file>mytarget_overview.html</file>
         <callback>mytarget_model</callback>
      </demoitem>
   </demosection>

</demos>
```

Development Tools Directory (mytarget/dev_tool1, mytarget/dev_tool2)

These directories contain files associated with specific development environments or tools (`dev_tool1`, `dev_tool2`, etc.). Normally, your target supports at least one such development environment and invokes its compiler, linker, and other utilities during the build process. `mytarget/dev_tool1` includes linker command files, startup code, hook functions, and any other files required to support this process.

For each development environment, you should provide a separate directory.

You should use the target preferences mechanism (see Chapter 8, "Using Target Preferences") to store information about a user's choice of development environment or tool, paths to the installed development tools, and so on.

Using target preferences data in this way lets your build process code select the appropriate development environment and invoke the appropriate compiler and other utilities. See the code excerpt in "mytarget_default_tmf.m Example Code" on page 6-14 for an example of how to use target preferences data for this purpose.

Target Preferences Directory (mytarget/mytarget/@mytarget)

If you create a target preferences class to store information about user preferences, you should store data class definition files and other files that support your target-specific preferences in `mytarget/mytarget/@mytarget`. The Simulink Data Class Designer creates the `@mytarget` directory automatically within the parent directory. See Chapter 8, "Using Target Preferences" for further information.

Target Source Code Directory (mytarget/src)

This directory is optional. If the complexity of your target requires it, you can use `mytarget/src` to store any common source code and configuration code (such as boot and startup code).

Key Files in the Target Directory (mytarget/mytarget)

In this section...

Introduction

The target directory `mytarget/mytarget` contains key files in your target implementation. These include the system target file, template makefile, main program module, and optional M and TLC hook files that let you add target-specific actions to the build process. The following sections describe the key target directory files.

mytarget.tlc

`mytarget.tlc` is the system target file (STF). Functions of the STF include

- Making the target visible in the System Target File Browser
- Definition of code generation options for the target (inherited and target-specific)
- Providing an entry point for the top-level control of the TLC code generation process

You should base your STF on `ert.tlc`, the STF provided by the Real-Time Workshop Embedded Coder software.

Chapter 5, "System Target Files" gives detailed information on the structure of the STF, and also gives instructions on how to customize an STF to

- Display your target in the System Target File Browser
- Add your own target options to the Configuration Parameters dialog box
- Tailor the code generation and build process to the requirements of your target

mytarget.tmf

`mytarget.tmf` is the template makefile for building an executable for your target.

For basic information on the structure and operation of template makefiles, see Chapter 6, "Template Makefiles".

If your target development environment requires automation of a modern integrated development environment (IDE) rather than use of a traditional make utility, see Chapter 9, "Interfacing to Development Tools".

It is often necessary to create multiple template makefiles to support different development environments. See "Supporting Multiple Development Environments" on page 5-37 and "mytarget_default_tmf.m Example Code" on page 6-14 for information.

mytarget_default_tmf.m

This file is optional. You can implement a `mytarget_default_tmf.m` file to select the correct template makefile, based on user preferences. See "Setting Up a Template Makefile" on page 6-13.

mytarget_settings.tlc

This file is optional. Its purpose is to centralize global settings in the code generation environment. See "Using mytarget_settings.tlc" on page 5-31 for details.

mytarget_genfiles.tlc

This file is optional. `mytarget_genfiles.tlc` is useful as a central file from which to invoke any target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads. See "Using mytarget_genfiles.tlc" on page 5-33 for details.

mytarget_main.c

A main program module is required for your target. To provide a main module, you can either

- Modify the `ert_main.c` module provided by the Real-Time Workshop Embedded Coder software
- Generate `mytarget_main.c` or `.cpp` during the build process

The "Developing Models for Code Generation" chapter of the Real-Time Workshop Embedded Coder documentation contains a detailed description of the operation of `ert_main.c`. The chapter also contains guidelines for generating and modifying a main program module.

The "Optimizing Generated Code" chapter of the Real-Time Workshop Embedded Coder documentation describes how you can generate a customized main program module.

STF_make_rtw_hook.m

`STF``_make_rtw_hook.m` is an optional hook file that you can use to invoke target-specific functions or executables at specified points in the build process. *`STF`*`_make_rtw_hook.m` implements a function that dispatches to a specific action depending on the `method` argument that is passed into it.

The "Optimizing Generated Code" section of the Real-Time Workshop Embedded Coder documentation describes the operation of the *`STF`*`_make_rtw_hook.m` hook file in detail.

STF_wrap_make_cmd_hook.m

Use this file to override the default Real-Time Workshop behavior for selecting the appropriate compiler tool to be used in the build process.

By default, the Real-Time Workshop build process is based on makefiles. On PC hosts, the build process creates *model*.bat, an MS-DOS batch file. *model*.bat sets up the appropriate environment variables for the compiler, linker and other utilities, and invokes a `make` utility. The batch file, *model*.bat, obtains the required environment variable settings from the `MAKECMD` field in the template makefile. The standard Real-Time Workshop template makefiles support only standard compilers that build executables on the host system.

When developing an embedded target, you often need to override these defaults. Typically, you need to support one or more target-specific cross-development systems, rather than supporting compilers for the host system. The *STF*_wrap_make_cmd_hook mechanism provides a way to set up an environment specific to an embedded development tool.

Note that the naming convention for this file is *not* based on the target name. It is based on the concatenation of the system target filename, *STF*, with the string `'_wrap_make_cmd_hook'`.

Stub makefiles

Many modern cross-development systems, such as the Freescale Semiconductor CodeWarrior development environment, are based on project files rather than makefiles. If the interface to the embedded development system is not makefile based, one recommended approach is to create a stub makefile. When the build process invokes the stub makefile, no action takes place.

STF_wrap_make_cmd_hook Mechanism

A recommended approach to supporting non-host-based development systems is to provide a hook file that is called instead of the default host-based compiler selection.

To do this, create a *STF*_wrap_make_cmd_hook.m file. If this file exists, the build process calls it instead of the default compiler selection process. Make sure that:

- The file is on the MATLAB path.
- The filename is the name of your STF, prepended to the string '__wrap_make_cmd_hook.m'.
- The hook function implemented in the file follows the function prototype shown in the code example below.

A typical approach would be to write a *STF*_wrap_make_cmd_hook.m file that creates a MS-DOS batch file (*model*.bat). The batch file first sets up environment variables for the embedded target development system. Then, it invokes the embedded target's make utility on the generated makefile. The *STF*_wrap_make_cmd_hook function should return a system command that invokes *model*.bat.

This approach is shown in "Example STF_wrap_make_cmd_hook Function" on page 4-15.

Alternatively, any MS-DOS batch file can be created by *STF*_wrap_make_cmd_hook, and the function can return any command; it is not limited to *model*.bat. Like the exit case of the *STF*_make_rtw_hook mechanism, this provides the flexibility to invoke other utilities or applications.

Note that on a PC host, the Real-Time Workshop build process checks the standard output (STDOUT) for an appropriate build success string. By default, the string is

```
"### Created"
```

You can change this specifying a different BUILD_SUCCESS variable in the template makefile.

Example STF_wrap_make_cmd_hook Function

```
function makeCmdOut = stfname_wrap_make_cmd_hook(args)
  makeCmd        = args.makeCmd;
  modelName      = args.modelName;
  verbose        = args.verbose;

  % args.compilerEnvVal not used
  cmdFile = ['.\',modelName, '.bat'];
  cmdFileFid = fopen(cmdFile,'wt');
  if ~verbose
    fprintf(cmdFileFid, '@echo off\n');
  end

  try
    prefs = RTW.TargetPrefs.load('mytarget.prefs');
    catch exception
    rethrow(exception);
  end

  fprintf(cmdFileFid, '@set TOOL_VAR1=%s\n', prefs.ImpPath);
  fprintf(cmdFileFid, '@set TOOL_VAR2=x86-win32\n');
  toolRoot = fullfile(prefs.ImpPath,'host','tool','4.4b');
  fprintf(cmdFileFid, '@set TOOL_VAR3=%s\n', toolRoot);
  path = getenv('Path');
  path1 = fullfile(prefs.ImpPath,'host','license;');
  if ~isempty(strfind(path,path1)) path1 = ''; end
  fprintf(cmdFileFid, '@set Path=%s%s%s\n', path1, path);
  fullMakeCmd = fullfile(prefs.ImpPath,'host','tool',...
                         'bin', makeCmd);
  fprintf(cmdFileFid, '%s\n', fullMakeCmd);
  fclose(cmdFileFid);
  makeCmdOut = cmdFile;
```

STF_rtw_info_hook.m (obsolete)

Prior to Release 14, custom targets supplied target-specific information with a hook file (referred to as *STF*_rtw_info_hook.m). The *STF*_rtw_info_hook specified properties such as word sizes for integer data types (for example, `char`, `short`, `int`, and `long`), and C implementation-specific properties of the custom target.

The *STF*_rtw_info_hook mechanism has been replaced by the **Hardware Implementation** pane of the Configuration Parameters dialog box. Using this dialog box, you can specify all properties that were formerly specified in your *STF*_rtw_info_hook file.

For backward compatibility, existing *STF*_rtw_info_hook files continue to operate correctly. However, you should convert your target and models to use the **Hardware Implementation** pane. See the "Configuring the Hardware Implementation" section of the Real-Time Workshop User's Guide.

info.xml

This file provides information to MATLAB software that specifies where to display the target toolbox on the MATLAB **Start** button menu.

Example info.xml File

This example shows you how to set up access to a target's demo page and target preferences GUI from the MATLAB **Start** button. See also "Making Target Preferences Available to the End User" on page 8-9.

```
<productinfo>

<matlabrelease>13</matlabrelease>
<name>Embedded Target for MYTARGET</name>
<type>simulink</type>
<icon>$toolbox/simulink/simulink/simulinkicon.gif</icon>

<list>

<listitem>
<label>Demos</label>
<callback>demo simulink 'Embedded Target for MYTARGET'</callback>
<icon>$toolbox/matlab/icons/demoicon.gif</icon>
</listitem>

<listitem>
<label>MYTARGET Target Preferences</label>
<callback>mytargetTargetPrefs =
RTW.TargetPrefs.load('mytarget.prefs');
```

```
gui(mytargetTargetPrefs); </callback>
<icon>$toolbox/simulink/simulink/simulinkicon.gif</icon>
</listitem>

</list>
</productinfo>
```

mytarget_overview.html

By convention, this file serves as home page for the target demos.

The `<description>` field in `demos.xml` should point to `mytarget_overview.html` (see "mytarget/blocks/demos.xml" on page 4-8).

Example mytarget_overview.html File

```
<html>
<head><title>Embedded Target for MYTARGET</title></head><body>
<p style="color:#990000; font-weight:bold; font-size:x-large">Embedded Target
for MYTARGET Demonstration Model</p>

<p>This demo provides a simple model that allows you to generate an executable
for a supported target board. You can then download and run the executable and
set breakpoints to study and monitor the execution behavior.</p>

</body>
</html>
```

Additional Directories and Files for Externally Developed Targets

In this section...

Introduction

If you are developing an embedded target that is not installed into the MATLAB tree, you should provide a target setup script and target documentation within `mytarget/mytarget`, for the convenience of your users. The following sections describe the required materials and where to place them.

mytarget/mytarget/mytarget_setup.m

This M-file script adds the necessary paths for your target to the MATLAB path. Your documentation should instruct users to run the script when installing the target.

You should include a call to the MATLAB function `savepath` in your `mytarget_setup.m` script. This function saves the added paths, so users need to run `mytarget_setup.m` only once.

The following code is an example `mytarget_setup.m` file.

```
function mytarget_setup()
curpath = pwd;
tgtpath = curpath(1:end-length('\mytarget'));
addpath(fullfile(tgtpath, 'mytarget'));
addpath(fullfile(tgtpath, 'dev_tool1'));
addpath(fullfile(tgtpath, 'blocks'));
addpath(fullfile(tgtpath, 'mytargetdemos'));
savepath;
disp('MYTARGET Target Path Setup Complete.');
```

mytarget/mytarget/doc

You should put all documentation related to your target in the directory `mytarget/mytarget/doc`.

5

System Target Files

Introduction

The system target file (STF) exerts overall control of the code generation stage of the build process. The STF also lets you control the presentation of your target to the end user. The STF provides

- Definitions of variables that are fundamental to the build process, such as code format to be generated
- The main entry point to the top-level TLC program that generates code
- Target information for display in the System Target File Browser
- A mechanism for defining target-specific code generation options (and other parameters affecting the build process) and for displaying them in the Configuration Parameters dialog box
- A mechanism for inheriting options from another target (such as the Embedded Real-Time (ERT) target)

This chapter provides information on the structure of the STF, guidelines for customizing an STF, and a basic tutorial that helps you get a skeletal STF up and running.

Note that, although the STF is a Target Language Compiler (TLC) file, it contains embedded M-code. Before creating or modifying an STF, you should acquire a working knowledge of TLC and of the M language. The Real-Time Workshop Target Language Compiler document and the M-File Programming section of the MATLAB documentation describe the features and syntax of both the TLC and MATLAB languages.

While reading this chapter, you may want to refer to the STFs provided with the Real-Time Workshop product. Most of these files are stored in the target-specific directories under *matlabroot*`/rtw/c`. Additional STFs are stored under *matlabroot*`/toolbox/rtw/targets`.

System Target File Naming and Location Conventions

An STF must be located in a directory on the MATLAB path for the target to be properly displayed in the System Target File Browser and invoked in the build process. Follow the location and naming conventions for STFs and related target files given in Chapter 4, "Target Directories, Paths, and Files". Note particularly the "Directory and File Naming Conventions" on page 4-3.

The rules for the location of target files differ, depending upon whether the target is internally developed at The MathWorks or not:

- For embedded targets developed by The MathWorks that are included in your MATLAB installation, the target root directory should be located under *`matlabroot`*`/toolbox/rtw/targets/`.
- For embedded targets *not* developed by The MathWorks, the target root directory should *not* be located anywhere in the MATLAB directory tree (that is, in or under the *`matlabroot`* directory). This restriction exists because installing a new MATLAB version (or reinstalling the current version) recreates the MATLAB directories, which deletes any custom target directories existing within the MATLAB tree.

System Target File Structure

In this section...

Overview

This section is a guide to the structure and contents of an STF. The following listing shows the general structure of an STF. Note that this is not a complete code listing of an STF. The listing consists of excerpts from each of the sections that make up an STF.

```
%%----------------------------
%% Header Comments Section
%%----------------------------
%% SYSTLC: Example Real-Time Target
%%    TMF: my_target.tmf MAKE: make_rtw EXTMODE: ext_comm
%% Inital comments contain directives for STF Browser.
%% Documentation, date, copyright, and other info may follow.
          ...
%selectfile NULL_FILE
         ...
%%----------------------------
%% TLC Configuration Variables Section
%%----------------------------
%% Assign code format, language, target type.
%%
%assign CodeFormat = "Embedded-C"
%assign TargetType = "RT"
```

```
%assign Language   = "C"
%%
%%----------------------------
%% (OPTIONAL) Import Target Settings
%%----------------------------
%include "mytarget_settings.tlc"
%%
%%----------------------------
%% TLC Program Entry Point
%%----------------------------
%% Call entry point function.
%include "codegenentry.tlc"
%%
%%----------------------------
%% (OPTIONAL) Generate Files for Build Process
%%----------------------------
%include "mytarget_genfiles.tlc"
%%----------------------------
%% RTW_OPTIONS Section
%%----------------------------
/%
  BEGIN_RTW_OPTIONS
  %% Define rtwoptions structure array. This array defines target-specific
  %% code generation variables, and controls how they are displayed.
  rtwoptions(1).prompt = 'example code generation options';
        ...
  rtwoptions(6).prompt = 'Show eliminated blocks';
  rtwoptions(6).type = 'Checkbox';
         ...
  %---------------------------------------%
  % Configure RTW code generation settings %
  %---------------------------------------%
         ...
  %%----------------------------
  %% rtwgensettings Structure
  %%----------------------------
  %% Define suffix string for naming build directory here.
  rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
  %% (OPTIONAL) target inheritance declaration
  rtwgensettings.DerivedFrom = 'ert.tlc';
```

```
  %% (OPTIONAL) r14 callback compatibility declaration
  rtwgensettings.Version = '1';
  %% (OPTIONAL) other rtwGenSettings fields...
         ...
  END_RTW_OPTIONS
%/
%%-----------------------------
%% targetComponentClass - MATHWORKS INTERNAL USE ONLY
%% REMOVE NEXT SECTION FROM USER_DEFINED CUSTOM TARGETS
%%-----------------------------
/%
  BEGIN_CONFIGSET_TARGET_COMPONENT
  targetComponentClass = 'Simulink.ERTTargetCC';
  END_CONFIGSET_TARGET_COMPONENT
%/
```

If you are creating a custom target based on an existing STF, you must remove the `targetComponentClass` section (bounded by the directives `BEGIN_CONFIGSET_TARGET_COMPONENT` and `END_CONFIGSET_TARGET_COMPONENT`). This section is reserved for the use of targets developed internally by The MathWorks.

Header Comments

These lines at the head of the file are formatted as TLC comments. They provide required information to the System Target File Browser and to the build process. Note that you must place the browser comments at the head of the file, before any other comments or TLC statements.

The presence of the comments enables the Real-Time Workshop software to detect STFs. When the System Target File Browser is opened, the Real-Time Workshop software scans the MATLAB path for TLC files that have correctly formatted header comments. The comments contain the following directives:

- SYSTLC: This string is a descriptor that appears in the browser.
- TMF: Name of the template makefile (TMF) to use during build process. When the target is selected, this filename is displayed in the **Template makefile** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

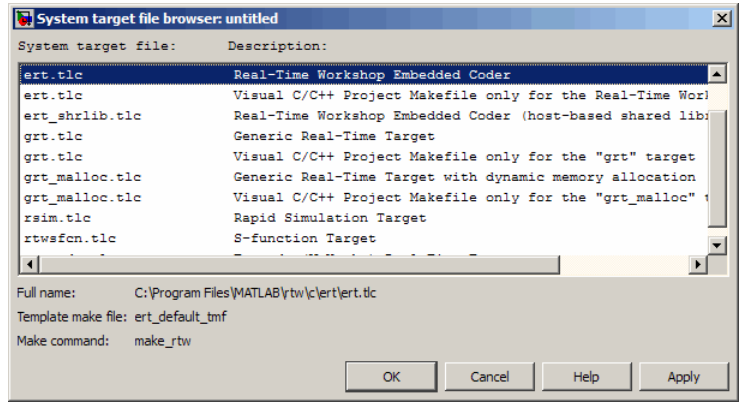
- MAKE: `make` command to use during build process. When the target is selected, this command is displayed in the **Make command** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog box.
- EXTMODE: Name of external mode interface file (if any) associated with your target. If your target does not support external mode, use `no_ext_comm`.

The following header comments are from *`matlabroot`*`/rtw/c/ert/ert.tlc`.

```
%% SYSTLC: Real-Time Workshop Embedded Coder (no auto configuration) \
%%    TMF: ert_default_tmf MAKE: make_rtw EXTMODE: ext_comm
.
.
.
%% SYSTLC: Visual C/C++ Project Makefile only for the Real-Time Workshop Embedded Coder\
%%    TMF: ert_msvc.tmf MAKE: make_rtw EXTMODE: ext_comm
```

Note Limitation: Each comment can only contain a maximum of two lines, as shown in the preceding example.

Note that you can specify more than one group of directives in the header comments. Each such group is displayed as a different target configuration in the System Target File Browser. In the above example, the first two lines of code specify the default configuration of the ERT target. The last two lines specify a configuration that generates a Microsoft® Visual C++® project makefile, using the template makefile `ert_msvc.tmf`. The figure below shows how these configurations appear in the System Target File Browser.



See "Tutorial: Creating a Custom Target Configuration" on page 5-39 for an example of customized header comments.

TLC Configuration Variables

This section of the STF assigns global TLC variables that affect the overall code generation process.

For an embedded target, in almost all cases you should simply use the global TLC variable settings used by the ERT target (`ert.tlc`). It is especially important that your STF select the Embedded-C code format. Make sure values are assigned to the following variables:

- `CodeFormat`: The `CodeFormat` variable selects one of the available code formats. The `Embedded-C` format is used by the ERT target. Your ERT-based target should specify `Embedded-C` format. `Embedded-C` format is designed for production code, minimal memory usage, static memory allocation, and a simplified interface to generated code.

  For information on other code formats, see the "Selecting and Configuring a Target " chapter of the Real-Time Workshop documentation.

- `Language`: The only valid value is `C`, which enables support for `C` or `C++` code generation as specified by the configuration parameter `TargetLang` (see the `TargetLang` entry in "Parameter Command-Line Information Summary" in the Real-Time Workshop documentation for more information).

- `TargetType`: The Real-Time Workshop software defines the preprocessor symbols `RT` and `NRT` to distinguish simulation code from real-time code. These symbols are used in conditional compilation. The `TargetType` variable determines whether `RT` or `NRT` is defined.

  Most targets are intended to generate real-time code. They assign `TargetType` as follows.

  ```
  %assign TargetType = "RT"
  ```

  Some targets, such as the model reference simulation target, accelerated simulation target, RSim target, and S-function target, generate code for use in non-real-time only. Such targets assign `TargetType` as follows.

  ```
  %assign TargetType = "NRT"
  ```

  See "Other Preprocessor Symbols" on page 10-8 for further information on the use of these symbols.

TLC Program Entry Point and Related %includes

The code generation process normally begins with `codegenentry.tlc`. The STF invokes `codegenentry.tlc` as follows.

```
%include "codegenentry.tlc"
```

Note `codegenentry.tlc` and the lower-level TLC files assume that `CodeFormat`, `TargetType`, and `Language` have been correctly assigned. Set these variables before including `codegenentry.tlc`.

If you need to implement target-specific code generation features, you should include the TLC files `mytarget_settings.tlc` and `mytarget_genfiles.tlc in` your STF. These files provide a mechanism for executing custom TLC code before and after invoking `codegenentry.tlc`. For information on these mechanisms, see

- "Using mytarget_settings.tlc" on page 5-31 for an example of custom TLC code for execution before the main code generation entry point.
- "Using mytarget_genfiles.tlc" on page 5-33 for an example of custom TLC code for execution after the main code generation entry point.

- "Understanding and Using the Build Process" on page 3-8 for general information on the build process, and for information on other build process customization hooks.

Another way to customize the code generation process is to call lower-level functions (normally invoked by `codegenentry.tlc`) directly, and include your own TLC functions at each stage of the process. This approach should be taken with caution. See the *Real-Time Workshop Target Language Compiler* document for guidelines.

The lower-level functions called by `codegenentry.tlc` are

- `genmap.tlc`: maps block names to corresponding language-specific block target files.
- `commonsetup.tlc`: sets up global variables.
- `commonentry.tlc`: starts the process of generating code in the format specified by `CodeFormat`.

RTW_OPTIONS Section

The `RTW_OPTIONS` section is bounded by the directives:

```
/%
  BEGIN_RTW_OPTIONS
.
.
.
  END_RTW_OPTIONS
%/
```

The first part of the `RTW_OPTIONS` section defines an array of `rtwoptions` structures. This structure is discussed in "rtwoptions Structure" on page 5-11.

The second part of the `RTW_OPTIONS` section defines `rtwgensettings`, a structure defining the build directory name and other settings for the code generation process. See "rtwgensettings Structure" on page 5-18 for information about `rtwgensettings`.

Note Release 14 introduced significant changes in the way that target options are defined, displayed, and operated. If you have developed a target for an earlier release or are developing a new target for Release 14 or later, see "Defining and Displaying Custom Target Options" on page 5-22. This is particularly important if your STF uses `rtwoptions` callbacks.

rtwoptions Structure

The fields of the `rtwoptions` structure define variables and associated user interface elements to be displayed in the **Real-Time Workshop** pane of the Configuration Parameters dialog box. Using the `rtwoptions` structure array, you can define target-specific options displayed in the dialog box and organize options into categories. You can also write callback functions to specify how these options are processed.

When the **Real-Time Workshop** pane opens, the `rtwoptions` structure array is scanned and the listed options are displayed. Each option is represented by an assigned user interface element (check box, edit field, menu, or push button), which displays the current option value.

The user interface elements can be in an enabled or disabled (grayed-out) state. If an option is enabled, the user can change the option value.

You can also use the `rtwoptions` structure array to define special NonUI elements that cause callback functions to be executed, but that are not displayed in the **Real-Time Workshop** pane. See "NonUI Elements" on page 5-17 for details.

The elements of the `rtwoptions` structure array are organized into groups. Each group of items begins with a header element of type `Category`. The default field of a `Category` header must contain a count of the remaining elements in the category.

The `Category` header is followed by options to be displayed on the **Real-Time Workshop** pane. The header in each category is followed by one or more option definition elements.

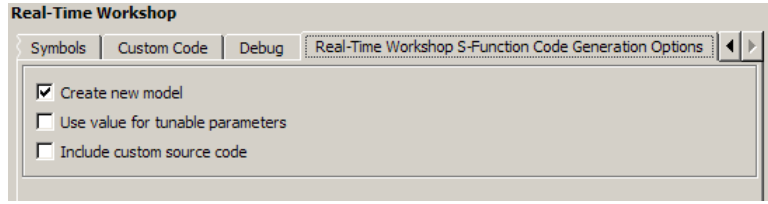
The way in which target option groups are displayed depends on whether or not the STF has been converted for compatibility with Release 14 or later. In Release 14 or later compatible targets, each category of options corresponds to options listed under **Real-Time Workshop** in the Configuration Parameters dialog box. (See "Target Options Display in Release 14 or Later" on page 5-27.)

The table rtwoptions Structure Fields Summary on page 5-14 summarizes the fields of the `rtwoptions` structure.

Example rtwoptions Structure. The following example is excerpted from *`matlabroot`*`/rtw/c/rtwsfcn/rtwsfcn.tlc`, the STF for the S-function target. The code defines an `rtwoptions` structure array of three elements. The default field of the first (header) element is set to `4`, indicating the number of elements that follow the header.

```
rtwoptions(1).prompt        = 'Real-Time Workshop S-Function Code Generation Options';
rtwoptions(1).type          = 'Category';
rtwoptions(1).enable        = 'on';
rtwoptions(1).default       = 4;   % number of items under this category
                                   % excluding this one.
rtwoptions(1).popupstrings  = '';
rtwoptions(1).tlcvariable   = '';
rtwoptions(1).tooltip       = '';
rtwoptions(1).callback      = '';
rtwoptions(1).makevariable  = '';

rtwoptions(2).prompt         = 'Create new model';
rtwoptions(2).type           = 'Checkbox';
rtwoptions(2).default        = 'on';
rtwoptions(2).tlcvariable    = 'CreateModel';
rtwoptions(2).makevariable   = 'CREATEMODEL';
rtwoptions(2).tooltip        = ...
  ['Create a new model containing the generated Real-Time Workshop S-Function
  block inside it'];

rtwoptions(3).prompt         = 'Use value for tunable parameters';
rtwoptions(3).type           = 'Checkbox';
rtwoptions(3).default        = 'off';
rtwoptions(3).tlcvariable    = 'UseParamValues';
rtwoptions(3).makevariable   = 'USEPARAMVALUES';
rtwoptions(3).tooltip        = ...
  ['Use value for variable instead of variable name in generated block mask
  edit fields'];

% Override the default setting for model name prefixing because
% the generated S-function is typically used in multiple models.
rtwoptions(4).default       = 'on';
rtwoptions(4).tlcvariable   = 'PrefixModelToSubsysFcnNames';

rtwoptions(5).prompt         = 'Include custom source code';
rtwoptions(5).type           = 'Checkbox';
rtwoptions(5).default        = 'off';
rtwoptions(5).tlcvariable    = 'AlwaysIncludeCustomSrc';
rtwoptions(5).tooltip        = ...
  ['Always include provided custom source code in the generated code'];
```

The first element adds **S-function target options** under **Real-Time Workshop** in the Configuration Parameters dialog box. The options defined in `rtwoptions(2)`, `rtwoptions(3)`, and `rtwoptions(5)` display as shown in the next figure.



If you want to define a large number of options, you can define multiple `Category` groups within a single system target file.

Note the `rtwoptions` structure and callbacks are written in M-code, although they are embedded in a TLC file. To verify the syntax of your `rtwoptions` structure definitions and code, you can execute the commands at the MATLAB prompt by copying and pasting them to the MATLAB Command Window.

For further examples of target-specific `rtwoptions` definitions, see "Using rtwoptions: Real-Time Workshop Options Example Target" on page 5-17.

rtwoptions Structure Fields Summary on page 5-14 lists the fields of the `rtwoptions` structure.

rtwoptions Structure Fields Summary

Field Name	Description
`callback`	See "Defining and Displaying Custom Target Options" on page 5-22 for information on converting callbacks for Release 14 or later compatibility. For examples of callback usage, see also "Using rtwoptions: Real-Time Workshop Options Example Target" on page 5-17.

rtwoptions Structure Fields Summary (Continued)

Field Name	Description
`closecallback (obsolete)`	If your target uses `closecallback`, convert to `rtwgensettings.PostApplyCallback` instead (see "rtwgensettings Structure" on page 5-18). See "Defining and Displaying Custom Target Options" on page 5-22 for information on converting callbacks for Release 14 or later compatibility. For examples of callback usage, see also "Using rtwoptions: Real-Time Workshop Options Example Target" on page 5-17. `closecallback` is ignored in Release 14 or later. Prior to Release 14, `closecallback` specified an M-code function to be executed when the target options dialog box closes.
`default`	Default value of the option (empty if the `type` is `Pushbutton`).
`enable`	Must be `'on'` or `'off'`. If `'on'`, the option is displayed as an enabled item; otherwise, as a disabled item.
`makevariable`	Template makefile token (if any) associated with the option. The `makevariable` is expanded during processing of the template makefile. See "Template Makefile Tokens" on page 6-2.
`modelReferenceParameterCheck`	Specifies whether the option must have the same value in a referenced model and its parent model. If this field is unspecified or has the value `'on'` the option values must be same. If the field is specified and has the value `'off'` the option values can differ. See "Controlling Configuration Option Value Agreement" on page 7-9.
`NonUI`	Element that is not displayed, but is used to invoke a close or open callback. See "NonUI Elements" on page 5-17.

rtwoptions Structure Fields Summary (Continued)

Field Name	Description
`opencallback (obsolete)`	If your target uses `opencallback`, we strongly recommend that you use `rtwgensettings.SelectCallback` instead (see "rtwgensettings Structure" on page 5-18). If you must maintain use of `opencallback`, see "Defining and Displaying Custom Target Options" on page 5-22 for information on converting callbacks for Release 14 or later compatibility. For examples of callback usage, see also "Using rtwoptions: Real-Time Workshop Options Example Target" on page 5-17. Prior to Release 14, `opencallback` specified M-code to be executed when the selected the target from the System Target File Browser, or during model loading. The purpose of `opencallback` is to synchronize the displayed value of the option with its previous setting.
`popupstrings`	If `type` is `Popup`, `popupstrings` defines the items in the menu. Items are delimited by the `"|"` (vertical bar) character. The following example defines the items of the **MAT-file variable name modifier** menu used by the GRT target. `'rt_|_rt|none'`
`prompt`	Label for the option.
`tlcvariable`	Name of TLC variable associated with the option.
`tooltip`	Help string displayed when mouse is over the item.
`type`	Type of element: `Checkbox`, `Edit`, `NonUI`, `Popup`, `Pushbutton`, or `Category`.

NonUI Elements. Elements of the `rtwoptions` array that have type `NonUI` exist solely to invoke callbacks. A `NonUI` element is not displayed in the Configuration Parameters dialog box. You can use a `NonUI` element if you want to execute a callback that is not associated with any user interface element, when the dialog box opens or closes. Only the `opencallback` and `closecallback` fields of a `NonUI` element have significance. See the next section, "Using rtwoptions: Real-Time Workshop Options Example Target" on page 5-17 for an example.

Using rtwoptions: Real-Time Workshop Options Example Target

A working system target file, with M-file callback functions, has been provided as an example of how to use the `rtwoptions` structure to display and process custom options on the **Real-Time Workshop** pane. The examples are compatible with the Release 14 or later callback API (described in "Defining and Displaying Custom Target Options" on page 5-22).

The example target files are in the directory:

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

The example target files are

- `usertarget.tlc`: The example system target file. This file defines several menus, check boxes, an edit field, and a nonUI item. The file demonstrates the use of callbacks, open callbacks, and closed callbacks.
- `usertargetcallback.m`: An M-file callback invoked by a menu.
- `usertargetclosecallback.m`: An M-file callback invoked by an edit field.

Refer to the example files while reading this section. The example system target file, `usertarget.tlc`: demonstrates the use of callbacks associated with the following UI elements:

- The **Execution Mode** menu executes an open callback that is coded inline within the STF. This callback displays a message and sets a model property with a `set_param()`.
- The **Real-Time Interrupt Source** menu executes a callback defined in an external M-file, `usertargetcallback.m`. The TLC variable associated

with the menu is passed in to the callback, which displays the menu's current value.

- The edit field **Signal Logging Buffer Size in Doubles** executes a close callback defined in an external M-file, `usertargetclosecallback.m`. The TLC variable associated with the edit field is passed in to the callback.
- The **External Mode** check box executes an open callback that is coded inline within the STF.
- The `NonUi` item defined in `rtwoptions(8)` executes open and close callbacks that are coded inline within the STF. Each callback simply prints a status message.

We suggest that you study the example code while interacting with the example target options in the Configuration Parameters dialog box. To interact with the example target file,

1. Make *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwoptions_demo` your working directory.
2. Open any model of your choice.
3. Open the Configuration Parameters dialog box or Model Explorer and select the **Real-Time Workshop** pane.
4. Click **Browse**. The System Target File Browser opens. Select `usertarget.tlc Real-Time Workshop Options Example Target`. Then click **OK**.
5. Observe that the **Real-Time Workshop** pane contains a custom sub-tab: **userPreferred target options (I)**.
6. As you interact with the options in this category and open and close the Configuration Parameters dialog box, observe the messages displayed in the MATLAB Command Window. These messages are printed from code in the STF, or from callbacks invoked from the STF.

rtwgensettings Structure

The final part of the STF defines the `rtwgensettings` structure. This structure stores information that is written to the *`model`*`.rtw` file and used

by the build process. The `rtwgensettings` fields of most interest to target developers are

- `rtwgensettings.Version`: This version compatibility property identifies targets use Release 14 or later compatible `rtwoptions` callbacks. Do not use this field unless you have converted your callbacks, as described in "Using rtwoptions Callbacks in Release 14 or Later" on page 5-22.
- `rtwgensettings.DerivedFrom`: This string property defines the system target file from which options are to be inherited. See "Target Options Inheritance in Release 14 or Later" on page 5-26.
- `rtwgensettings.SelectCallback`: this property specifies a `SelectCallback` function. `SelectCallback` is associated with the target rather than with any of its individual options. The `SelectCallback` function is triggered when the user selects a target with the System Target File browser. When a model created prior to Release 14 is opened, the `SelectCallback` function is also triggered during model loading.

  The `SelectCallback` function is useful for setting up (or disabling) configuration parameters specific to the target.

  The following code installs a `SelectCallback` function:

  ```
  rtwgensettings.SelectCallback = ['my_select_callback_handler(hDlg, hSrc)'];
  ```

  The arguments to the `SelectCallback` function `(hDlg, hSrc)` are handles to private data used by the callback API functions, as described in "Using rtwoptions Callbacks in Release 14 or Later" on page 5-22.

Note If you have developed a custom target and you want it to be compatible with model referencing, you must implement a `SelectCallback` function to declare model reference compatibility. See Chapter 7, "Supporting Optional Features".

- `rtwgensettings.ActivateCallback`: this property specifies an `ActivateCallback` function. The `ActivateCallback` function is triggered when the active configuration set of the model changes. This could happen during model loading, and also when the user changes the active configuration set.

The following code installs an `ActivateCallback` function:

```
rtwgensettings.ActivateCallback = ['my_activate_callback_handler(hDlg, hSrc)'];
```

The arguments to the `ActivateCallback` function `(hDlg, hSrc)` are handles to private data used by the callback API functions, as described in "Using rtwoptions Callbacks in Release 14 or Later" on page 5-22.

- `rtwgensettings.PostApplyCallback`: this property specifies a `PostApplyCallback` function. The `PostApplyCallback` function is triggered when the user clicks the **Apply** or **OK** button after editing options in the Configuration Parameters dialog box. The `PostApplyCallback` function is called after the changes have been applied to the configuration set.

  The following code installs an `PostApplyCallback` function:

  ```
  rtwgensettings.PostApplyCallback = ['my_postapply_callback_handler(hDlg, hSrc)'];
  ```

  The arguments to the `PostApplyCallback` function `(hDlg, hSrc)` are handles to private data used by the callback API functions, as described in "Using rtwoptions Callbacks in Release 14 or Later" on page 5-22.

- `rtwgensettings.BuildDirSuffix`: Most targets define a string that identifies build directories created by the target. The build process appends the string defined in the `rtwgensettings.BuildDirSuffix` field to the model name to form the name of the build directory. For example, if you define `rtwgensettings.BuildDirSuffix` as follows

  ```
  rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
  ```

  the build directories are named *`model`*`_mytarget_rtw`.

Additional Code Generation Options

"Configuring Generated Code with TLC" in the Real-Time Workshop documentation describes additional TLC code generation variables. End users of any target can assign these variables by entering statements of the form

```
-aVariable=val
```

in the **TLC options** field of the **Real-Time Workshop** pane.

However, the preferred approach is to assign these variables in the STF using statements of the form:

```
%assign Variable = val
```

For readability, we recommend that you add such assignments in the section of the STF after the comment `Configure RTW code generation settings`.

Model Reference Considerations

See Chapter 7, "Supporting Optional Features" for important information on STF and other modifications you may need to make to support the Real-Time Workshop model referencing features.

Defining and Displaying Custom Target Options

In this section...

Upgrading Custom Targets to Release 14 or Later

In Release 14 or later, you view the Simulink model options defined in the active configuration set using the Configuration Parameters dialog box or Model Explorer. These views, which replaced the Simulation Parameters dialog box used in releases before Release 14, feature extensive changes in the appearance and layout of code generation options and other target-specific options for Real-Time Workshop targets. For customers upgrading from a release before Release 14, this section describes the following compatibility issues related to the definition and display of target-specific options for custom targets:

- `rtwoptions` callbacks: If the `rtwoptions` array in your custom system target file contains callbacks, convert your callbacks to use the callback compatibility API provided in Release 14 or later.
- Target options inheritance: If your custom target is derived from another target and inherits options, change your system target file to use the inheritance mechanism provided in Release 14 or later.
- Display of target options: Your target options will display differently in Release 14 or later, and you may want to reorganize them.

Using rtwoptions Callbacks in Release 14 or Later

In releases before Release 14, the `callback`, `opencallback`, and `closecallback` fields of the `rtwoptions` structure array (see "rtwoptions Structure" on page 5-11) specify optional M-code functions that are called when the value of an option changes or when the Simulation Parameters dialog box opens or closes. If your custom system target file does *not* specify any such callbacks, your target operates transparently in the Configuration

Parameters dialog box. However, your target options are displayed differently, as described in "Target Options Display in Release 14 or Later" on page 5-27.

If your custom target does specify callbacks, compatibility issues arise, because many callbacks depend upon features of the old-style (pre-Release 14) Simulation Parameters dialog box. For example, a change in the state of one GUI element (such as a check box) may invoke a callback that attempts to get a handle to another GUI element in order to enable or disable it.

In Release 14 or later, the Real-Time Workshop software supports a callback compatibility API that lets your existing `rtwoptions` callbacks operate under the Configuration Parameters dialog box. This is described in the next section, "How to Convert Your rtwOptions Callbacks" on page 5-23. We strongly recommend that you convert your callbacks for Release 14 or later compatibility. If you do not want to do so, see "Operation of Targets with Unconverted Callbacks" on page 5-25 to understand how your custom target runs in the Release 14 or later environment.

How to Convert Your rtwOptions Callbacks

The callback conversion API provides variables and accessor functions that allow your callbacks to access graphical elements associated with target options. Also, a version compatibility property, `rtwgensettings.Version`, has been added to the `rtwgensettings` structure in the system target file. This property identifies targets that have been converted to use Release 14 or later compatible callbacks.

The callback API variables are

- `model`: Handle of the current Simulink model. `model` can be used as an argument to `get_param` and `set_param` calls. If you use such calls, you do not need to change them.
- `hSrc`: This variable is restricted to use in the callback API functions described below. `hSrc` provides a handle to private data used by the callback API functions. Do not set this variable or use it for any other purpose.
- `hDlg`: This variable is restricted to use in the callback API functions described below. `hDlg` provides a handle to private data used by the

callback API functions. Do not set this variable or use it for any other purpose.

The callback API provides accessor functions that let you read and set target option values, and enable or disable options. In the function descriptions below, the `tlc_var_name` argument is the name of the `tlcvariable` defined for the option in the `rtwoptions` struct. The callback API accessor functions are

- `slConfigUIGetVal(hDlg, hSrc, 'tlc_var_name')`: Returns the current value of the option specified by the `'tlc_var_name'` argument. The data type of the return value depends on the data type of the option.
- `slConfigUISetVal(hDlg, hSrc, 'tlc_var_name', value)`: Sets the option specified by the `'tlc_var_name'` argument to the value passed in the `value` argument.
- `slConfigUISetEnabled(hDlg, hSrc, 'tlc_var_name', flag)`: Enables or disables the option specified by the `'tlc_var_name'` argument. The value passed in `flag` should be either 1 (to enable the option) or 0 (to disable the option).

To convert your `rtwOptions` callbacks,

1. Identify all references to the old Simulation Parameters dialog box handle (such as `dialogFig` or objects accessed through `dialogFig`) in your callbacks.
2. Replace such references with equivalent calls to the callback API functions. Your code should use only the API calls and variables described above to reference options. See the files described in "Example Callback Code" on page 5-25.
3. If your target inherits options from an existing target, you should also convert your target to use the new inheritance mechanism. To learn how to do this, see "Target Options Inheritance in Release 14 or Later" on page 5-26.
4. Declare that your system target file is compliant with the callback API by adding the following statement in the `Configure RTW code generation settings` section of the system target file.

```
rtwgensettings.Version = '1';
```

`rtwoptions` callbacks are executed only if `rtwgensettings.Version` is set as shown.

Note If your target defines `opencallback` functions, open callbacks are called during model loading, as well as when the you select the target from the System Target File Browser.

Example Callback Code

An example system target file and callback handlers are available in the directory *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwoptions_demo`. The example files illustrate how to use Release 14 or later compatible callbacks with different types of GUI elements. The files are

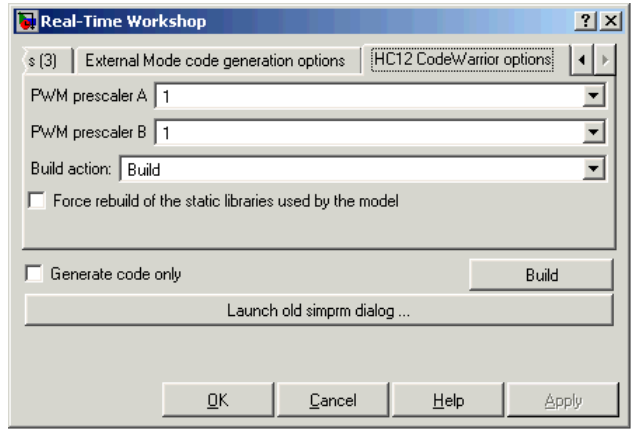
- `usertarget.tlc`: The example system target file. This file defines several menus, check boxes, an edit field, and a nonUI item. The file demonstrates the use of callbacks, open callbacks, and close callbacks.
- `usertargetcallback.m`: An M-file callback invoked by a popup.
- `usertargetclosecallback.m`: An M-file callback invoked by an edit field.

Operation of Targets with Unconverted Callbacks

Callback conversion is recommended, but not required. If you do not want to convert your callbacks, your target operates as follows:

- When the target is selected with the System Target File Browser, the target options are displayed in the Configuration Parameters dialog box, as described in "Target Options Display in Release 14 or Later" on page 5-27. However, any callbacks specified in the `rtwoptions` array are ignored.
- An additional button labeled **Launch old simprm dialog** is displayed at the bottom of all target-specific pages of the Configuration Parameters dialog box. When the user clicks this button, the old Simulation Parameters dialog box opens. As the user interacts with the dialog box, existing callbacks are executed.

The figure below shows the Model Explorer view.



Note If your custom target uses unconverted callbacks, you should inform end users of your target that they should open and use the old Simulation Parameters dialog box when setting target options. If they do not do so, options are not set correctly.

Target Options Inheritance in Release 14 or Later

In releases before Release 14, many custom targets used the technique of merging `rtwoptions` structures in order to derive or inherit options from an existing target. For example, the following code, from a Release 13 target, creates an `rtwoptions` structure and inherits the `rtwoptions` of the ERT target merging them into the structure.

```
/%
  BEGIN_RTW_OPTIONS
  rtwoption_index = 0;

  rtwoption_index = rtwoption_index + 1;
  rtwoptions(rtwoption_index).prompt        = 'mytargets Options';
  rtwoptions(rtwoption_index).type          = 'Category';
  rtwoptions(rtwoption_index).enable        = 'on';
  rtwoptions(rtwoption_index).default       = 5;   % number of items under mytargets
```

```
rtwoptions(rtwoption_index).popupstrings  = '';
rtwoptions(rtwoption_index).tlcvariable   = '';
rtwoptions(rtwoption_index).tooltip       = '';
rtwoptions(rtwoption_index).callback      = '';
rtwoptions(rtwoption_index).makevariable  = '';
%other rtwoptions elements not shown here
...
% Inherit ERT options
file     = fullfile(matlabroot, 'rtw', 'c', 'ert', 'ert.tlc');
propsObj = tlc.rtwoptions(file);
props    = propsObj.getOptions;
rtwoptions = propsObj.combineCategories(props,rtwoptions);
```

Releases 14 and later support a new, simplified inheritance mechanism. The string property `rtwgensettings.DerivedFrom` has been added to the `rtwgensettings` structure. This property defines the system target file from which options are to be inherited. You should convert your custom target to use this mechanism as follows:

1. Remove old inheritance code (such as the four line after the `%Inherit ERT options` comment in the example above).

2. Set the `rtwgensettings.DerivedFrom` property as in the following example

   ```
   rtwgensettings.DerivedFrom = 'stf.tlc';
   ```

   where `stf` is the name of the system target file from which options are to be inherited. For example:

   ```
   rtwgensettings.DerivedFrom = 'ert.tlc';
   ```

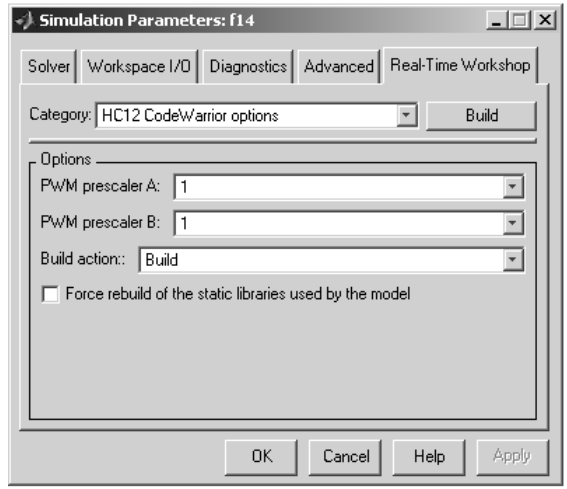
When the Configuration Parameters dialog box executes this line of code, it includes the options from `stf.tlc` automatically. If `stf.tlc` is a MathWorks internal system target file that has been converted to a new layout, the dialog box displays the inherited options using the new layout.

Target Options Display in Release 14 or Later

In the Simulation Parameters dialog box present in releases before Release 14, target options are organized into functional groups, displayed under control of the **Category** menu in the **Real-Time Workshop** pane. The

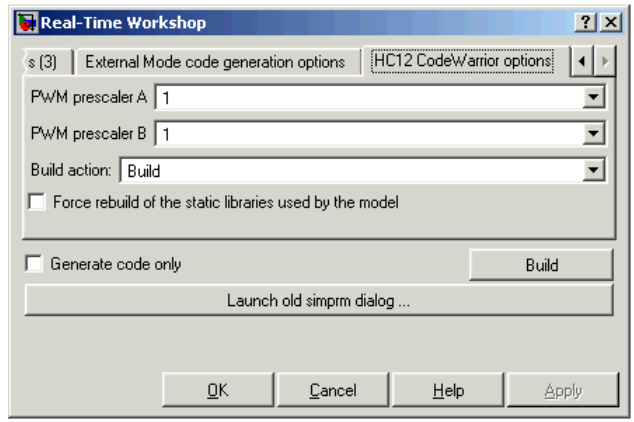
items in the **Category** menu correspond to the elements of the `rtwoptions` structure array. Each group of `rtwoptions` elements is delimited by a header element of type `Category`.

The following figure shows a typical group of target options as displayed in the old-style Simulation Parameters dialog box.

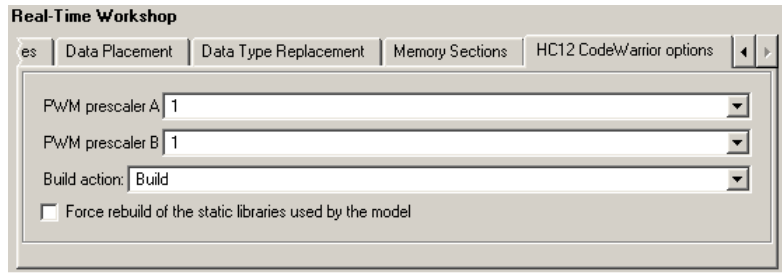


The new Configuration Parameters dialog box preserves the organization of your custom target's `rtwoptions` structure array. However, the **Category** menu has been replaced by a tabbed selection mechanism. In the Model Explorer view, each category of options corresponds to a tab. In the Configuration Parameters dialog box view, each category of options corresponds to an element of the list on the left pane. The spacing and layout of options within each group of options is controlled by the Real-Time Workshop software.

The figure below shows the same target options, as organized and displayed in the Model Explorer view. This figure shows how the target options appear before any Release 14 or later compatibility conversions are made.



After converting the above target to use Release 14 or later compatible callbacks and inheritance options, the target's inherited options are displayed in a more compact form (under categories such as **Interface**, **Templates**, and so on) and the **Launch old simprm dialog** button is removed, as shown in this figure.



The Real-Time Workshop product provides the organization of options described above as a default layout. This lets you continue to use your custom targets with minimal change. This default differs considerably from many of the targets developed internally at The MathWorks (such as the ERT and GRT targets). These MathWorks targets have been converted to use technologies and features that are currently available only to MathWorks developers. In a future release, The MathWorks plans to provide information and APIs that let you convert your custom targets to take full advantage of these technologies and features.

Tips and Techniques for Customizing Your STF

In this section...

Introduction

The following sections include information on techniques for customizing your STF, including

- How to invoke custom TLC code from your STF
- How to inherit target options from another STF
- Approaches to supporting multiple development environments with single or multiple STFs

Required and Recommended %includes

If you need to implement target-specific code generation features, we recommend that your STF include the TLC files `mytarget_settings.tlc` and `mytarget_genfiles.tlc`.

`mytarget_settings.tlc` provides a mechanism for executing custom TLC code before the main code generation entry point. See "Using mytarget_settings.tlc" on page 5-31.

Once your STF has set up any required TLC environment, you must include `codegenentry.tlc` to start the standard code generation process.

`mytarget_genfiles.tlc` provides a mechanism for executing custom TLC code after the main code generation entry point. See "Using mytarget_genfiles.tlc" on page 5-33.

Using mytarget_settings.tlc

This file is optional. Its purpose is to centralize global settings in the code generation environment. Use `mytarget_settings.tlc` to

- Define required TLC paths with `%addincludepath` directives. You may need to do this if you create target-specific TLC function libraries.
- Create records that store target-specific path information and preference settings in the `CompiledModel` general record. This provides a clean mechanism for passing this information into the TLC code generation environment.
- Check user settings for code generation options. If incorrect or unsupported option settings are found, issue the appropriate error or warning and abort the build process if necessary.

mytarget_settings.tlc Example Code. In the TLC code example below, the structure Settings is added to the `CompiledModel` record. The Settings structure is loaded from the stored target preferences (see "Accessing Target Preference Data from the MATLAB Prompt" on page 8-11). The Settings structure stores target preferences data fields `Implementation` and `ImpPath`.

After `Settings` is added to the `CompiledModel` record, the example code handles inherited options. In this example, the target is assumed to have inherited options from the ERT target. The code examines the settings of inherited ERT code generation options. If the user has selected unsupported options, warning or error messages are displayed. In some cases, selecting an unsupported option causes the build process to terminate.

Conditional code at the end of the function allows display of the `Implementation` and `ImpPath` fields in the MATLAB Command Window.

```
%selectfile NULL_FILE

%% Read user preferences for the target and add to CompiledModel
%assign prefs = FEVAL("RTW.TargetPrefs.load","mytarget.prefs","structure")
%addtorecord CompiledModel Settings prefs

%% Check for unsupported Embedded Coder options and error/warn appropriately
%if SuppressErrorStatus == 0
  %assign SuppressErrorStatus = 1
```

```
  %assign msg = "Suppressing Error Status as it is not used by this target."
  %warning %<msg>
%endif
%if GenerateSampleERTMain == 1
  %assign msg = "Generating an example main is not supported as the proper main
function is inherently generated. Unselect the \"Generate an example main program\"
checkbox under ERT code generation options."
  %exit %<msg>
%endif

%if GenerateErtSFunction == 1
  %assign msg = "Generating a Simulink S-Function is not supported. Unselect the
\"Create Simulink(S-Function) block\" checkbox under ERT code generation options."
  %exit %<msg>
%endif

%if ExtMode == 1
  %assign msg = "External Mode is not currently supported. Unselect the \"External
mode\" checkbox under ERT code generation options."
  %exit %<msg>
%endif

%if MatFileLogging == 1
  %assign msg = "MAT-file logging is not currently supported. Unselect the
\"MAT-file logging\" checkbox under ERT code generation options."
  %exit %<msg>
%endif

%if MultiInstanceERTCode == 1
  %assign msg = "Generate reuseable code is not currently supported. Unselect the
\"Generate reuseable code\" checkbox under ERT code generation options."
  %exit %<msg>
%endif

%if GenFloatMathFcnCalls == "ISO_C"
  %assign msg = "Target function libraries other than ANSI-C are not
currently supported. Select ANSI-C for the \"Target function library\" option
under ERT code generation options."
  %exit %<msg>
%endif
```

```
%% To display added TLC settings for debugging purposes, set EchoConfigSettings to
1.
%assign EchoConfigSettings = 0
%if EchoConfigSettings
  %selectfile STDOUT
  ################################################################

  IMPLEMENTATION is:
  %<CompiledModel.Settings.Implementation>

  IMPLEMENTATION path is:
  %<CompiledModel.Settings.ImpPath>

  ################################################################
  %selectfile NULL_FILE
%endif
```

Using mytarget_genfiles.tlc

`mytarget_genfiles.tlc` (optional) is useful as a central file from which to invoke any target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads.

The build process can then invoke these generated files either directly from the make process, or after the executable is created. This is done with the *STF*`_make_rtw_hook.m` mechanism, as described in "Customizing the Target Build Process with the STF_make_rtw Hook File" in the Real-Time Workshop Embedded Coder User's Guide.

The following TLC code shows an example `mytarget_genfiles.tlc` file.

```
%selectfile NULL_FILE

%assign ModelName = CompiledModel.Name

%% Create Debugger script
%assign model_script_file = "%<ModelName>.cfg"
```

```
%assign script_file = "debugger_script_template.tlc"

%if RTWVerbose
   %selectfile STDOUT
   ### Creating %<model_script_file>
   %selectfile NULL_FILE
%endif

%include "%<script_file>"
%openfile bld_file = "%<model_script_file>"
%<CreateDebuggerScript()>
%closefile bld_file
```

Inherited Target Options

`ert.tlc` provides a basic set of Real-Time Workshop Embedded Coder code generation options. If your target is based on `ert.tlc`, your STF should normally inherit the options defined in ERT.

Note The inheritance mechanism described in this section is available in Release 14 or later. Targets developed prior to Release 14 should be converted to use this mechanism as described in "Target Options Inheritance in Release 14 or Later" on page 5-26.

To make options inheritance simple, the Real-Time Workshop software provides the `rtwgensettings.DerivedFrom` property. This string property defines the system target file from which options are to be inherited. Set this property as in the following example

```
rtwgensettings.DerivedFrom = 'stf.tlc';
```

where `stf` is the name of the system target file from which options are to be inherited. For example, to inherit options from the ERT target.

```
rtwgensettings.DerivedFrom = 'ert.tlc';
```

Handling Unsupported Options

If your target does not support all options inherited from `ert.tlc`, you should detect unsupported option settings and display a warning or error message. In some cases, if a user has selected an option your target does not support, you may need to abort the build process. For example, if your target does not support the **Generate an example main program** option, the build process should not be allowed to proceed if that option is selected.

We recommend that you handle these options in `mytarget_settings.tlc`. See the example in "Using mytarget_settings.tlc" on page 5-31.

Even though your target may not support all inherited ERT options, it is required that the ERT options are retained in the **Real-Time Workshop** pane of the GUI. Do not simply remove unsupported options from the `rtwoptions` structure in the STF. Options must be in the GUI to be scanned by the Real-Time Workshop software when it performs optimizations.

For example, you may want to prevent users from turning off the **Single output/update function** option. It may seem safe to remove this option from the GUI and simply assign the TLC variable `CombineOutputUpdateFcns` to `on`. However, if the option is not included in the GUI, the Real-Time Workshop software assumes that output and update functions are *not* to be combined. Less efficient code is generated as a result.

Handling Aliases for Target Option Values

This section describes utility functions that can be used to detect and resolve alias values or legacy values when testing user-specified values for the target device type (`ProdHWDeviceType`) and the target function library (`GenFloatMathFcnCalls`).

RTW.isHWDeviceTypeEq

To test if two target device type strings represent the same hardware device, invoke the following function:

```
result = RTW.isHWDeviceTypeEq(type1, type2)
```

where *`type1`* and *`type2`* are strings containing target device type values or aliases.

The RTW.isHWDeviceTypeEq function returns true if *type1* and *type2* are strings representing the same hardware device. For example, the following call returns true:

```
RTW.isHWDeviceTypeEq('Specified', 'Generic->Custom')
```

For a description of the target device type option ProdHWDeviceType, see the command-line information for the **Hardware Implementation** pane options "Device vendor" and "Device type" in the Simulink reference documentation.

RTW.resolveHWDeviceType

To return the device type value for a hardware device, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveHWDeviceType(type)
```

where *type* is a string containing a target device type value or alias.

The RTW.resolveHWDeviceType function returns the device type value of the device. For example, the following calls both return 'Generic->Custom':

```
RTW.resolveHWDeviceType('Specified')
RTW.resolveHWDeviceType('Generic->Custom')
```

For a description of the target device type option ProdHWDeviceType, see the command-line information for the **Hardware Implementation** pane options "Device vendor" and "Device type" in the Simulink reference documentation.

RTW.isTflEq

To test if two target function library (TFL) strings represent the same TFL, invoke the following function:

```
result = RTW.isTflEq(name1, name2)
```

where *name1* and *name2* are strings containing TFL values or aliases.

The RTW.isTflEq function returns true if *name1* and *name2* are strings representing the same TFL. For example, the following call returns true:

```
RTW.isTflEq('ANSI_C', 'C89/C90 (ANSI)')
```

For a description of the TFL option `GenFloatMathFcnCalls`, see the command-line information for the **Interface** pane option "Target function library" in the Real-Time Workshop reference documentation.

RTW.resolveTflName

To return the TFL value for a target function library, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveTflName(name)
```

where *name* is a string containing a TFL value or alias.

The `RTW.resolveTflName` function returns the TFL value of the referenced TFL. For example, the following calls both return `'C89/C90 (ANSI)'`:

```
RTW.resolveTflName('ANSI_C')
RTW.resolveTflName('C89/C90 (ANSI)')
```

For a description of the TFL option `GenFloatMathFcnCalls`, see the command-line information for the **Interface** pane option "Target function library" in the Real-Time Workshop reference documentation.

Supporting Multiple Development Environments

Your target may require support for multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable vs. generating a project file for your compiler).

One approach to this requirement is to implement multiple STFs; each STF invokes an appropriate template makefile for the development environment. This amounts to providing two separate targets.

Another approach is to use a single STF that specifies multiple configurations in its comment header. The code within the STF then checks the target preferences to determine which template makefile to invoke. See "mytarget_default_tmf.m Example Code" on page 6-14 for an example of how to check target preferences for this information.

One drawback of using a single STF in this way is that the rtwoptions need conditional sections if the target options are not the same for all of the configurations the STF supports. The following example (from a hypothetical example target) defines an rtwoptions menu element differently, depending on the whether or not the MATLAB software is running on a PC (Microsoft Windows platform). This is determined by calling the MATLAB function `ispc`. On the PC, the menu displays a choice of USB or serial ports to be used in communicating with a target device. Otherwise, the menu displays a choice of UNIX[1] logical devices.

```
if ispc
  rtwoptions(rtwoption_index).default      = 'USB';
  rtwoptions(rtwoption_index).popupstrings =
'USB|COM1|COM2|COM3|COM4';
else
  rtwoptions(rtwoption_index).default      = '/dev/ttyS0';
  rtwoptions(rtwoption_index).popupstrings =
'/dev/ttyS0|/dev/ttyS1|/dev/ttyS2|/dev/ttyS3';
end
```

1. UNIX® is a registered trademark of The Open Group in the United States and other countries.

Tutorial: Creating a Custom Target Configuration

In this section...

Introduction

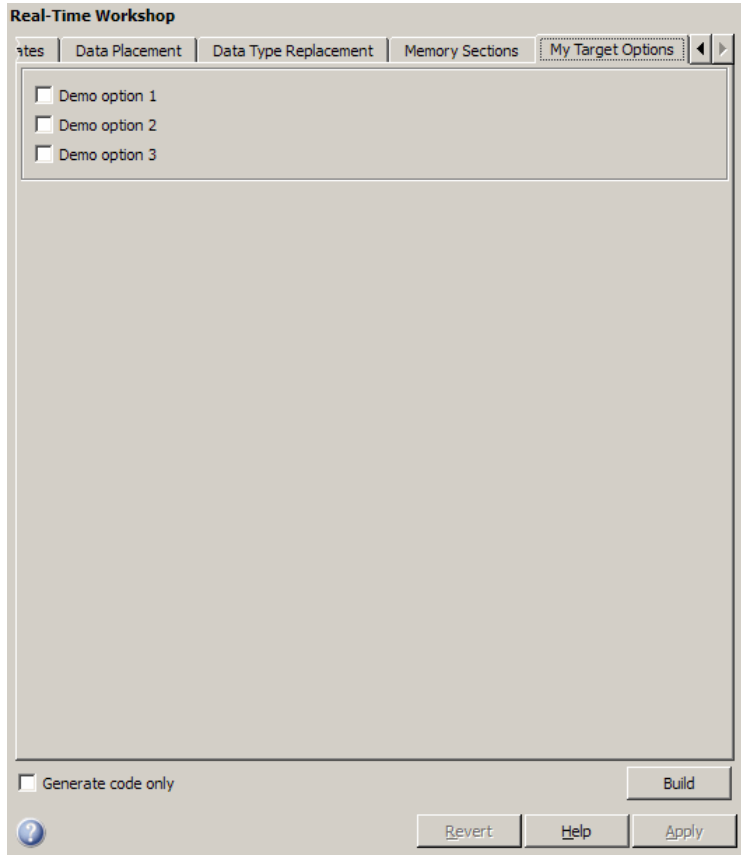
The purpose of this tutorial is to guide you through the process of creating an ERT-based target, `my_ert_target`. This exercise illustrates several tasks that are usually required when creating a custom target:

- Setting up target directories and modifying the MATLAB path.
- Making modifications to a standard STF and TMF such that the custom target is visible in the System Target File Browser, inherits ERT options, displays target-specific options, and generates code with the default host-based compiler.
- Testing the build process with the custom target, using a simple model that incorporates an inlined S-function.

During this exercise you implement an operational, but skeletal, ERT-based target. This target may be useful as a starting point in a complete implementation of a custom embedded target.

my_ert_target Overview

In the following sections you create a skeletal target, `my_ert_target`. The target inherits and supports the standard options of the ERT target, and displays additional target-specific options in the Configuration Parameters dialog box (see Target-Specific Options for my_ert_target on page 5-40).



Target-Specific Options for my_ert_target

`my_ert_target` supports a makefile-based build, generating code and executables that run on the host system. `my_ert_target` uses the LCC compiler on a Microsoft Windows platform. This compiler was chosen because it is readily available and is distributed with the Real-Time Workshop product. If you use a different compiler, you can set up LCC temporarily as your default compiler by typing the MATLAB command

```
mex -setup
```

Follow the prompts and select LCC.

Note On UNIX[2] systems, make sure that you have a C compiler installed. You can then do this exercise, substituting appropriate UNIX directory syntax.

You can test `my_ert_target` with any model that is compatible with the ERT target. (See the section of the Real-Time Workshop Embedded Coder documentation.) Generated programs operate identically to ERT generated programs.

However, to simplify the testing of your target, we recommend testing with `targetmodel.mdl`, a very simple fixed-step model (see "Create Test Model and S-Function" on page 5-49). The S-Function block in `targetmodel.mdl` uses the source code from the `timestwo` example, and generates fully inlined code. See the Simulink Writing S-Functions document and the *Real-Time Workshop Target Language Compiler* document for a complete discussion of the `timestwo` example S-function.

Creating Target Directories

In this section, you create directories to store the target files and add them to the MATLAB path, following the recommended conventions (see "Directory and File Naming Conventions" on page 4-3). You also create a directory to store the test model, S-function, and generated code.

This example assumes that your target and model directories are located within the directory `c:/work`. Note that your target and model directories should not be located anywhere in the MATLAB directory tree (that is, in or under the *`matlabroot`* directory).

To create the necessary directories and make them accessible,

1. Create a target root directory, `my_ert_target`. To do this from the MATLAB Command Window on a Windows platform, enter:

```
cd c:/work
mkdir my_ert_target
```

2. UNIX® is a registered trademark of The Open Group in the United States and other countries.

2 Within the target root directory, create a subdirectory to store your target files.

```
mkdir my_ert_target/my_ert_target
```

3 Add these directories to your MATLAB path.

```
addpath c:/work/my_ert_target
addpath c:/work/my_ert_target/my_ert_target
```

4 Create a directory, `my_targetmodel`, to store the test model, S-function, and generated code.

```
mkdir my_targetmodel
```

Create ERT-Based STF

In this section, you create an STF for your target by copying and modifying the standard STF for the ERT target. Then you validate the STF by viewing the new target in the System Target File Browser and the Configuration Parameters dialog box.

Editing the STF

To edit the STF,

1 Change your working directory to the directory you created in "Creating Target Directories" on page 5-41.

```
cd c:/work/my_ert_target/my_ert_target
```

2 Place a copy of *`matlabroot`*`/rtw/c/ert/ert.tlc` in `c:/work/my_ert_target/my_ert_target` and rename it to `my_ert_target.tlc`. The file `ert.tlc` is the STF for the ERT target.

3 Open `my_ert_target.tlc` in a text editor of your choice.

4 Generally, the first step in customizing an STF is to replace the header comment lines with directives that make your STF visible in the System Target File Browser and define the associated TMF (that you create shortly), `make` command, and external mode interface file (if any). See "Header Comments" on page 5-6 for a detailed explanation of these directives.

Replace the header comments in `my_ert_target.tlc` with the following header comments.

```
%% SYSTLC: My ERT-based Target TMF: my_ert_target_lcc.tmf MAKE: make_rtw \
%%    EXTMODE: no_ext_comm
```

5. The file `my_ert_target.tlc` inherits the standard ERT options, using the mechanism described in "Inherited Target Options" on page 5-34. Therefore, the existing `rtwoptions` structure definition is superfluous. Edit the RTW_OPTIONS section such that it includes only the following code.

```
/%
  BEGIN_RTW_OPTIONS

  %---------------------------------------%
  % Configure RTW code generation settings %
  %---------------------------------------%

  rtwgensettings.BuildDirSuffix = '_ert_rtw';

  END_RTW_OPTIONS
 %/
```

6. Delete the code after the end of the `RTW_OPTIONS` section, which is delimited by the directives `BEGIN_CONFIGSET_TARGET_COMPONENT` and `END_CONFIGSET_TARGET_COMPONENT`. This code is for use only by internal MathWorks developers.

7. Modify the build directory suffix in the `rtwgenSettings` structure in accordance with the conventions described in "rtwgensettings Structure" on page 5-18.

   To set the suffix to a string appropriate to the `_my_ert_target` custom target, change the line

   ```
   rtwgensettings.BuildDirSuffix = '_ert_rtw'
   ```

   to

   ```
   rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw'
   ```

8 Modify the `rtwgenSettings` structure to inherit options from the ERT target and declare Release 14 or later compatibility as described in "rtwgensettings Structure" on page 5-18. Add the following code to the `rtwgenSettings` definition:

```
rtwgensettings.DerivedFrom = 'ert.tlc';
rtwgensettings.Version = '1';
```

9 Add an `rtwoptions` structure that defines a target-specific options category with three check boxes just after the `BEGIN_RTW_OPTIONS` directive. The following code shows the complete `RTW_OPTIONS` section, including the `rtwgenSettings` changes made in previous steps.

```
/%
  BEGIN_RTW_OPTIONS

  rtwoptions(1).prompt         = 'My Target Options';
  rtwoptions(1).type           = 'Category';
  rtwoptions(1).enable         = 'on';
  rtwoptions(1).default        = 3;   % number of items under this category
                                      % excluding this one.
  rtwoptions(1).popupstrings  = '';
  rtwoptions(1).tlcvariable   = '';
  rtwoptions(1).tooltip       = '';
  rtwoptions(1).callback      = '';
  rtwoptions(1).makevariable  = '';

  rtwoptions(2).prompt         = 'Demo option 1';
  rtwoptions(2).type           = 'Checkbox';
  rtwoptions(2).default        = 'off';
  rtwoptions(2).tlcvariable    = 'DummyOpt1';
  rtwoptions(2).makevariable   = '';
  rtwoptions(2).tooltip        = ['Demo option1 (non-functional)'];
  rtwoptions(2).callback       = '';

  rtwoptions(3).prompt         = 'Demo option 2';
  rtwoptions(3).type           = 'Checkbox';
  rtwoptions(3).default        = 'off';
  rtwoptions(3).tlcvariable    = 'DummyOpt2';
  rtwoptions(3).makevariable   = '';
  rtwoptions(3).tooltip        = ['Demo option2 (non-functional)'];
```

```
  rtwoptions(3).callback       = '';

  rtwoptions(4).prompt         = 'Demo option 3';
  rtwoptions(4).type           = 'Checkbox';
  rtwoptions(4).default        = 'off';
  rtwoptions(4).tlcvariable    = 'DummyOpt3';
  rtwoptions(4).makevariable   = '';
  rtwoptions(4).tooltip        = ['Demo option3 (non-functional)'];
  rtwoptions(4).callback       = '';

  %------------------------------------%
  % Configure RTW code generation settings %
  %------------------------------------%

  rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw';
  rtwgensettings.DerivedFrom = 'ert.tlc';
  rtwgensettings.Version = '1';

  END_RTW_OPTIONS
%/
```

10 Save your changes to `my_ert_target.tlc` and close the file.

Viewing the STF

At this point, you can verify that the target inherits and displays ERT options correctly as follows:

1 Create a new model.

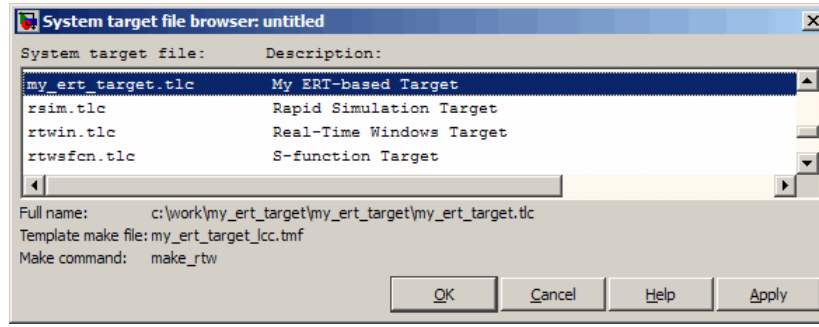
2 Open the Model Explorer or the Configuration Parameters dialog box.

3 Select the **Real-Time Workshop** pane.

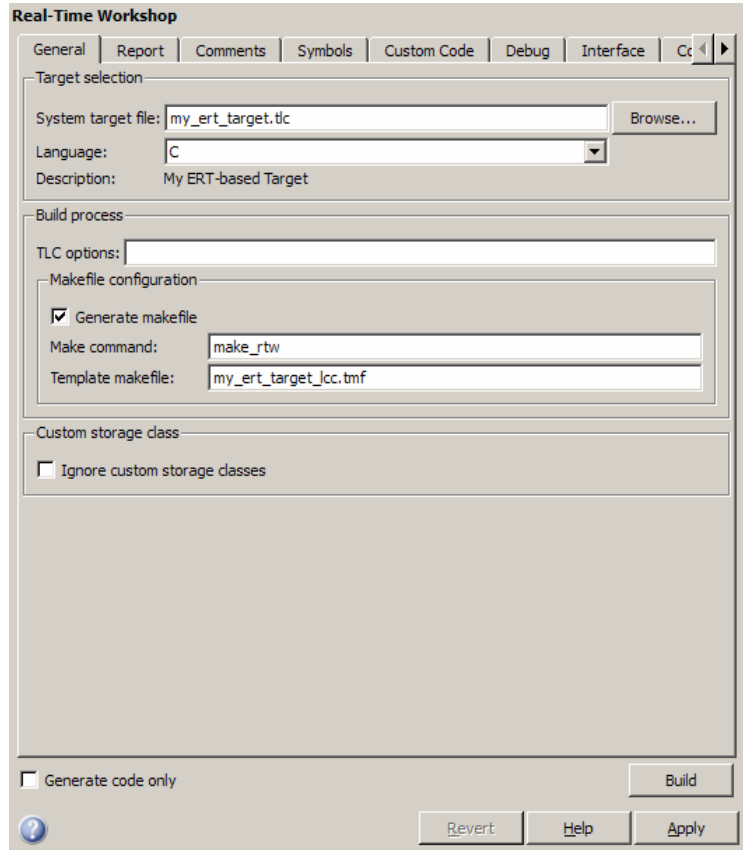
4 Click **Browse** to open the System Target File Browser.

5 In the Browser, scroll through the list of targets to find the new target, `my_ert_target.tlc`. (This step assumes that your MATLAB path contains `c:/work/my_ert_target/my_ert_target`, as previously set in "Creating Target Directories" on page 5-41.)

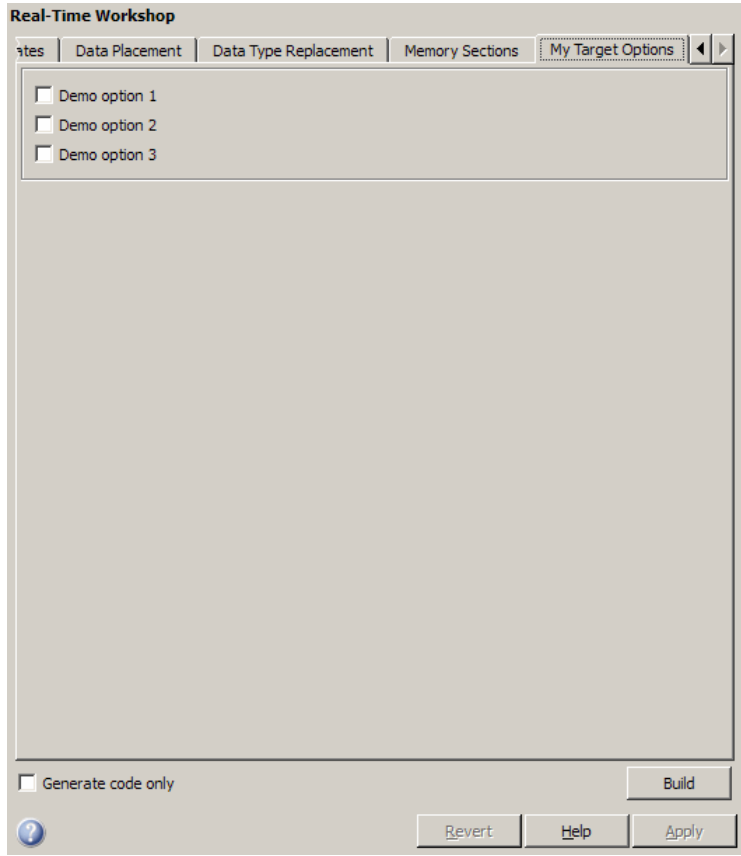
6 Select `My ERT-based Target` as shown below, and click **OK.**



7 The **Real-Time Workshop** pane now shows that the model is configured for the `my_ert_target.tlc` target. The **System target file**, **Make command**, and **Template makefile** fields should appear as follows:



8 Select the **My Target Options** pane and observe that the target displays the three check box options defined in the `rtwoptions` structure, as shown in the following figure.



9 Select the **Real-Time Workshop** pane and reopen the System Target File Browser.

10 Select the Real-Time Workshop Embedded Coder target (`ert.tlc`) and observe that the target displays the standard ERT options.

11 Close the model. You do not need to save it.

At this point, the STF for the skeletal target is complete. Note, however, that the STF header comments reference a TMF, `my_ert_target_lcc.tmf`. You are not able to invoke the build process for your target until the TMF file is in place. In the next section, you create `my_ert_target_lcc.tmf`.

Create ERT-Based TMF

In this section, you create a TMF for your target by copying and modifying the standard ERT TMF for the LCC compiler:

1. Check that your working directory is still set to the target file directory you created previously in "Creating Target Directories" on page 5-41.

   c:/work/my_ert_target/my_ert_target

2. Place a copy of *matlabroot*/rtw/c/ert/ert_lcc.tmf in c:/work/my_ert_target/my_ert_target and rename it to my_ert_target_lcc.tmf. The file ert_lcc.tmf is the ERT compiler-specific template makefile for the LCC compiler.

3. Open my_ert_target_lcc.tmf in a text editor of your choice.

4. Change the SYS_TARGET_FILE parameter so that the correct file reference is generated in the make file. Change the line

   ```
   SYS_TARGET_FILE = any
   ```

   to

   ```
   SYS_TARGET_FILE = my_ert_target.tlc
   ```

5. Save changes to my_ert_target_lcc.tmf and close the file.

Your target can now generate code and build a host-based executable. In the next sections, you create a test model and test the build process using my_ert_target.

Create Test Model and S-Function

In this section, you build a simple test model for later use in code generation:

1. Set your working directory to c:/work/my_targetmodel.

   ```
   cd c:/work/my_targetmodel
   ```

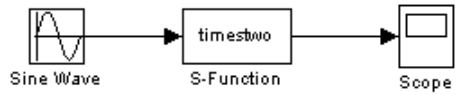
   For the remainder of this tutorial, my_targetmodel is assumed to be the working directory. Your target writes the output files of the code generation

process into a build directory within the working directory. When inlined code is generated for the `timestwo` S-function, the build process looks for the TLC implementation of the S-function in the working directory.

2. Copy the following C and TLC files for the `timestwo` S-function to your working directory:
   - *`matlabroot`*`/simulink/src/timestwo.c`
   - *`matlabroot`*`/toolbox/simulink/blocks/tlc_c/timestwo.tlc`
3. Build the `timestwo` MEX-file in `c:/work/my_targetmodel`.

   ```
   mex timestwo.c
   ```

4. Create the following model, using an S-Function block from the Simulink User-Defined Functions library. Save the model in your working directory as `targetmodel.mdl`.



5. Double-click the S-Function block to open the Block Parameters dialog box. Enter the S-function name `timestwo`. Click **OK**. The block is now bound to the `timestwo` MEX-file.
6. Open Model Explorer or the Configuration Parameters dialog box and select the **Solver** pane.
7. Set the solver **Type** to `fixed-step` and click **Apply**.
8. Save the model.
9. Open the scope and run a simulation. Verify that the `timestwo` S-function multiplies its input by 2.0.

Keep the `targetmodel` model open for use in the next section, in which you generate code using the test model.

Verify Target Operation

In this section you configure `targetmodel` for the `my_ert_target` custom target, and use the target to generate code and build an executable:

1. Open Model Explorer or the Configuration Parameters dialog box and select the **Real-Time Workshop** pane.
2. Click **Browse** to open the System Target File Browser.
3. In the Browser, select `My ERT-based Target` and click **OK**.
4. The Configuration Parameters dialog box now displays the **Real-Time Workshop** pane for `my_ert_target`.
5. Select the **Real-Time Workshop > Report** pane and select the **Create code generation report** option.
6. Click **Apply** and save the model. The model is configured for `my_ert_target`.
7. Build the model. If the build is successful, the MATLAB Command Window displays the message below.

   ```
   ### Created executable: ../targetmodel.exe
   ### Successful completion of Real-Time Workshop build procedure for model:
   targetmodel
   ```

   Your working directory contains the `targetmodel.exe` file and the build directory, `targetmodel_my_ert_target_rtw`, which contains generated code and other files. The working directory also contains an `slprj` directory, used internally by the build process.

   The code generator also creates and displays a code generation report.
8. To view the generated model code, go to the code generation report window. In the **Contents** pane, click the `targetmodel.c` link.

9 In `targetmodel.c`, locate the model step function, `targetmodel_step`. Observe the following code.

```
/* S-Function Block: <Root>/S-Function */
/* Multiply input by two */
targetmodel_B.SFunction = targetmodel_B.SineWave * 2.0;
```

The presence of this code confirms that the `my_ert_target` custom target has generated a correct inlined output computation for the S-Function block in the model.

6

Template Makefiles

Template Makefiles and Tokens

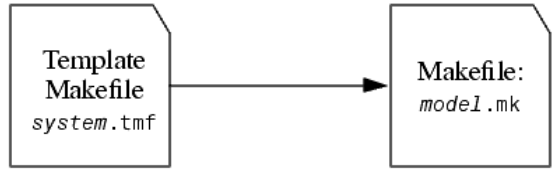
In this section...

Prerequisites

To configure or customize a template makefile (TMF), you should be familiar with how the `make` command works and how it processes makefiles. You should also understand makefile build rules. For information on these topics, refer to the documentation provided with the make utility you use.

Template Makefile Role In Makefile Creation

TMFs are made up of statements containing tokens. The Real-Time Workshop build process expands tokens and creates a makefile, *`model`*`.mk`. TMFs are designed to generate makefiles for specific compilers on specific platforms. The generated *`model`*`.mk` file is tailored to compile and link code generated from your model, using commands specific to your development system.



Creation of model.mk

Template Makefile Tokens

The `make_rtw` M-file command (or a different command provided with some targets) directs the process of generating *`model`*`.mk`. The `make_rtw` command processes the TMF specified on the **General** options section of the **Real-Time Workshop** pane of the Configuration Parameters dialog box. `make_rtw` copies the TMF, line by line, expanding each token encountered. Template

Makefile Tokens Expanded by make_rtw on page 6-3 lists the tokens and their expansions.

These tokens are used in several ways by the expanded makefile:

- To control the conditional behavior in the makefile. The conditionals are used to control the source file lists, library names, target to be built, and other build-related information.
- To provide the appropriate macro definitions needed for the compilation of the files, for example, `-DINTEGER_CODE=1`.

Template Makefile Tokens Expanded by make_rtw

Token	Expansion
`|>ALT_MATLAB_BIN<|`	Alternate full pathname for the MATLAB executable; value is different than value for `MATLAB_BIN` token when the full pathname contains spaces.
`|>ALT_MATLAB_ROOT<|`	Alternate full pathname for the MATLAB installation; value is different than value for `MATLAB_ROOT` token when the full pathname contains spaces.
`|>BUILDARGS<|`	Options passed to `make_rtw`. This token is provided so that the contents of your *`model`*`.mk` file changes when you change the build arguments, thus forcing an update of all modules when your build options change.
`|>COMBINE_OUTPUT_UPDATE_FCNS<|`	True (1) when **Single output/update function** is selected, otherwise False (0). Used for the macro definition `-DONESTEPFCN=1`.
`|>COMPUTER<|`	Computer type. See the MATLAB `computer` command.

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
`|>EXT_MODE<|`	True (1) to enable generation of external mode support code, otherwise False (0).
`|>EXTMODE_TRANSPORT<|`	Index of transport mechanism (for example, `tcpip`, `serial`) for external mode.
`|>EXTMODE_STATIC<|`	True (1) if static memory allocation is selected for external mode. False (0) if dynamic memory allocation is selected.
`|>EXTMODE_STATIC_SIZE<|`	Size of static memory allocation buffer (if any) for external mode.
`|>GENERATE_ERT_S_FUNCTION<|`	True (1) when **Create Simulink (S-Function) block** is selected, otherwise False (0). Used for control of the makefile target of the build.
`|>INCLUDE_MDL_TERMINATE_FCN<|`	True (1) when **Terminate function required** is selected, otherwise False (0). Used for the macro definition `-DTERMFCN==1`.
`|>INTEGER_CODE<|`	True (1) when **Support floating-point numbers** is not selected, otherwise False (0). `INTEGER_CODE` is a required macro definition when compiling the source code and is used when selecting precompiled libraries to link against.
`|>MAKEFILE_NAME<|`	*`model`*`.mk` — The name of the makefile that was created from the TMF.

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
`|>MAT_FILE<|`	True (1) when **MAT-file logging** is selected, otherwise False (0). `MAT_FILE` is a required macro definition when compiling the source code and also is used to include logging code in the build process.
`|>MATLAB_BIN<|`	Location of the MATLAB executable.
`|>MATLAB_ROOT<|`	Path to where MATLAB is installed.
`|>MEM_ALLOC<|`	Either `RT_MALLOC` or `RT_STATIC`. Indicates how memory is to be allocated.
`|>MEXEXT<|`	MEX-file extension. See the MATLAB `mexext` command.
`|>MODEL_MODULES<|`	Any additional generated source 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 modules.
`|>MODEL_NAME<|`	Name of the Simulink block diagram currently being built.
`|>MULTITASKING<|`	True (1) if solver mode is multitasking, otherwise False (0).
`|>NCSTATES<|`	Number of continuous states.
`|>NUMST<|`	Number of sample times in the model.

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
`|>PORTABLE_WORDSIZES<|`	True (1) when **Enable portable word sizes** is selected, otherwise False (0).
`|>RELEASE_VERSION<|`	The MATLAB release version.
`|>S_FUNCTIONS<|`	List of noninlined S-function 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, for example, `ode3.c`.
`|>SOLVER_OBJ<|`	Solver object (`.obj`) filename, for example, `ode3.obj`.
`|>TARGET_LANG_EXT<|`	`c` when the Real-Time Workshop **Language** selection is C, `cpp` when the **Language** selection is C++. Used in the makefile to control the extension on generated source files.
`|>TGT_FCN_LIB<|`	Specifies compiler command line options. The line in the makefile is `TGT_FCN_LIB = |>TGT_FCN_LIB<|`. By default, the Real-Time Workshop build process expands the `|>TGT_FCN_LIB<|` token to match the setting of the **Target function library** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box. Possible values for this option include `ANSI_C`, `C99 (ISO)`, and `GNU99 (GNU)`. You can use this token in a makefile conditional statement to specify compiler options to be

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
	used. For example, if you set the token to `C99 (ISO)`, the compiler might need an additional option set to support C99 library functions.
`|>TID01EQ<|`	True (1) if sampling rates of the continuous task and the first discrete task are equal, otherwise False (0).

These tokens are expanded by substitution of parameter values known to the build process. For example, if the source model contains blocks with two different sample times, the TMF statement

```
NUMST = |>NUMST<|
```

expands to the following in *`model`*`.mk`.

```
NUMST = 2
```

In addition to the above, `make_rtw` expands tokens from other sources:

- Target-specific tokens defined in the target options of the Configuration Parameters dialog box
- Structures in the `rtwoptions` section of the system target file. Any structures in the `rtwoptions` structure array that contain the field `makevariable` are expanded.

The following example is extracted from *`matlabroot`*`/rtw/c/grt/grt.tlc`. The section starting with `BEGIN_RTW_OPTIONS` contains M-file code that sets up `rtwoptions`. The following directive causes the `|>EXT_MODE<|` token to be expanded to `1` (on) or `0` (off), depending on how you set the **External mode** options.

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

Invoking the make Utility

In this section...

make Command

After creating *model*.mk from your TMF, the Real-Time Workshop build process invokes a make command. To invoke make, the build process issues this command.

```
makecommand -f model.mk
```

makecommand is defined by the MAKECMD macro in your target's TMF (see "Structure of the Template Makefile" on page 6-10). You can specify additional options to make in the **Make command** field of the **Real-Time Workshop** pane. (See the sections "Specifying a Make Command" and "Template Makefiles and Make Options" in the Real-Time Workshop documentation.)

For example, specifying OPT_OPTS=-O2 in the **Make command** field causes make_rtw to generate the following make command.

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

A comment at the top of the TMF specifies the available make command options. If these options do not provide you with enough flexibility, you can configure your own TMF.

make Utility Versions

The make utility lets you control nearly every aspect of building your real-time program. There are several different versions of make available. The Real-Time Workshop software provides the Free Software Foundation GNU® make for both UNIX[3] and PC platforms in platform-specific subdirectories under

3. UNIX® is a registered trademark of The Open Group in the United States and other countries.

```
matlabroot/bin
```

It is possible to use other versions of `make` with the Real-Time Workshop software, although GNU Make is recommended. To ensure compatibility with the Real-Time Workshop software, make sure that your version of `make` supports the following command format.

```
makecommand -f model.mk
```

Structure of the Template Makefile

A TMF has four sections:

- The first section contains initial comments that describe what this makefile targets.
- The second section defines macros that tell `make_rtw` how to process the TMF. The macros are
  - `MAKECMD` — This is the command used to invoke the make utility. For example, if `MAKECMD` = `mymake`, then the `make` command invoked is

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

  - `HOST` — The target platform for this TMF 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 it should invoke `make` from the Real-Time Workshop build procedure. Specify `BUILD=yes` or `no`.
  - `SYS_TARGET_FILE` — Name of the system target file or the value `all`. This is used for consistency checking by `make_rtw` to verify that the correct system target file was specified in the **Target selection** panel of the **Real-Time Workshop** pane of the Configuration Parameters dialog box. If you specify `all`, you can use the TMF with any system target file.
  - `BUILD_SUCCESS` — An optional macro that specifies the build success string to be displayed on successful `make` completion on the PC. For example,

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

    The `BUILD_SUCCESS` macro, if used, replaces the standard build success string found in the TMFs distributed with the bundled Real-Time Workshop targets (such as GRT):

    ```
    @echo ### Created executable $(MODEL).exe
    ```

    Your TMF must include either the standard build success string, or use the `BUILD_SUCCESS` macro. For an example of the use of `BUILD_SUCCESS`, see

    ```
    matlabroot/rtw/c/grt/grt_lcc.tmf
    ```

- `BUILD_ERROR` — An optional macro that specifies the build error message to be displayed when an error is encountered during the `make` procedure. For example,

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

- `VERBOSE_BUILD_OFF_TREATMENT = PRINT_OUTPUT_ALWAYS` — add this command if you want the makefile output to be displayed always (regardless of the setting of the **Verbose build** option in the **Real-Time Workshop > Debugging** pane).

The following `DOWNLOAD` options apply only to the Wind River® Tornado® target:

- `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 (see Template Makefile Tokens Expanded by make_rtw on page 6-3).
- 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 next figure shows the general structure of a Template Make File.

```
#--Section 1: Comments ------------------------------------
#
# Description of target type and version of make for which
# this template makefile is intended.
# Also documents any optional build arguments.
#--Section 2: Macros read by make_rtw ----------------------
#
# The following macros are read by the Real-Time Workshop build procedure:
#
#  MAKECMD         - This is the command used to invoke the make utility.
#  HOST            - Platform this template makefile is designed 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.

MAKECMD         = 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 ----------------------------------
#
# The build rules are specific to your target and version of make.
```

Comments

make_rtw macros

make_rtw tokens

Build rules

Customizing and Creating Template Makefiles

In this section...

Introduction

This section describes the mechanics of setting up a custom template makefile (TMF) and incorporating it into the build process. It also discusses techniques for modifying a TMF and M-file mechanisms associated with the TMF.

Before creating a custom TMF, you should read Chapter 4, "Target Directories, Paths, and Files" to understand the directory structure and MATLAB path requirements for custom targets.

Setting Up a Template Makefile

To customize or create a new TMF, you should copy an existing GRT or ERT TMF from one of the following locations:

```
matlabroot/rtw/c/grt
matlabroot/rtw/c/ert
```

Place the copy in the same directory as the associated system target file (STF). Usually, this is the `mytarget/mytarget` directory within the target directory structure. Then, rename your TMF appropriately (for example, `mytarget.tmf`) and modify it.

To ensure that the build process locates and selects your TMF correctly, you must provide information in the STF file header (see "System Target File Structure" on page 5-4). For a target that implements a single TMF, the standard way to specify the TMF to be used in the build process is to use the TMF directive of the STF file header.

```
TMF: mytarget.tmf
```

If your target must support multiple development environments, you can specify an M-file script that selects the correct TMF, based on user preferences (see Chapter 8, "Using Target Preferences"). To do this, you must

- Create the M-file script in your `mytarget/mytarget` directory. The naming convention for this file is `mytarget_default_tmf.m`.
- Specify this M-file in the TMF directive of the STF file header.

  ```
  TMF: mytarget_default_tmf
  ```

The build process then invokes your `mytarget_default_tmf.m` file, which then selects the correct TMF, based on target preference settings. "mytarget_default_tmf.m Example Code" on page 6-14 illustrates this technique.

Another useful technique is to store a path to the user's installed development environment in your target preferences. You can then locate the template makefiles under the appropriate tool directory. This allows several tool-specific template makefiles files to be located under the specific tool directory.

mytarget_default_tmf.m Example Code

The code example below implements an M-function, `mytarget_default_tmf`. The function loads target preferences into a structure from preferences data stored on disk. The code verifies that the target preferences information is consistent with the STF name, and extracts the associated TMF name. The TMF name is returned as the string `tmf`.

```
function [tmf,envVal] = mytarget_default_tmf
  try
    prefs = RTW.TargetPrefs.load('mytarget.prefs','structure');
  catch exception
```

```
rethrow(exception);
end

% Get the desired MYTARGET implementation and ensure it is supported
if ~isfield(prefs, 'Implementation')
  error('MYTARGET preferences not set correctly, update Target Preferences.');
end
imp = deblank(lower(prefs.Implementation));
stfname = deblank(lower(get_param(bdroot, 'RTWSystemTargetFile')));

if ~strncmp(imp, stfname, length(stfname) - length('.tlc'))
  msg = ['System Target filename: ', stfname,
         ' does not match Implementation specified in Target Preferences: ', imp];
  error(msg);
end

% Return the desired template make file.
tmf = [imp, '.tmf'];

% This argument is unused
envVal = '';
```

Using Macros and Pattern Matching Expressions in a Template Makefile

This section shows, through an example, how to use macros and file-pattern-matching expressions in a TMF to generate commands in the *model*`.mk` file.

The make utility processes the *model*`.mk` makefile and generates a set of commands based upon dependency rules defined in *model*`.mk`. After `make` generates the set of commands needed to build or rebuild `test`, `make` executes them.

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 source code. Thus there is a dependency between source and object files.

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, a UNIX[4] environment is assumed. In a PC environment the file extensions and compile and link commands are different.

In processing the first rule

```
test: file1.o file2.o
```

`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
```

If `file1.o` doesn't exist, or if `file1.o` is older than `file1.c`, `make` compiles `file1.c`.

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

```
MACRO_NAME = value
```

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

4. UNIX® is a registered trademark of The Open Group in the United States and other countries.

You can use pattern matching expressions to make the dependency rules more general. For example, using GNU[5] Make, you could replace the two "`file1.o: file1.c`" and "`file2.o: file2.c`" rules with the single rule

```
%.o : %.c
        cc -c $<
```

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

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

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

%.o : %.c
        cc -c $<
```

Note that the `$@` macro above is another special macro that equates to the name of the current dependency target, in this case `test`.

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 (for example, `.c`) with the object file extension (`.o`). This example also generalized the build rule for the program, `test`, to use the special "`$@`" macro.

Using the rtwmakecfg.m API to Customize Generated Makefiles

5. GNU® is a registered trademark of the Free Software Foundation.

Overview

Real-Time Workshop TMFs provide tokens that let you add the following items to generated makefiles:

- Source directories
- Include directories
- Run-time library names
- Run-time module objects

S-functions can add this information to the makefile by using an `rtwmakecfg.m` M-file function. This function is particularly useful when building a model that contains one or more of your S-Function blocks, such as device driver blocks.

To add information pertaining to an S-function to the makefile,

1. Create the M-file function `rtwmakecfg` in a file `rtwmakecfg.m`. The Real-Time Workshop software associates this file with your S-function based on its directory location. "Creating the rtwmakecfg.m M-File Function" on page 6-18 discusses the requirements for the `rtwmakecfg` function and the data it should return.
2. Modify your target's TMF such that it supports macro expansion for the information returned by `rtwmakecfg` functions. "Modifying the Template Makefile" on page 6-21 discusses the required modifications.

After the TLC phase of the build process, when generating a makefile from the TMF, the Real-Time Workshop build process searches for an `rtwmakecfg.m` file in the directory that contains the S-function component. If it finds the file, the build process calls the `rtwmakecfg` function.

Creating the rtwmakecfg.m M-File Function

Create the `rtwmakecfg.m` M-file function in the same directory as your S-function component (a MEX-file with a platform-dependent extension, such as `.mexw32` on 32-bit Microsoft Windows platforms). The function must return a structured array that contains the following fields:

Field	Description
`makeInfo.includePath`	A cell array that specifies additional include directory names, organized as a row vector. The Real-Time Workshop build process expands the directory names into include instructions in the generated makefile.
`makeInfo.sourcePath`	A cell array that specifies additional source directory names, organized as a row vector. The Real-Time Workshop build process expands the directory names into make rules in the generated makefile.
`makeInfo.sources`	A cell array that specifies additional source filenames (C or C++), organized as a row vector. The Real-Time Workshop build process expands the filenames into make variables that contain the source files. You should specify only filenames (with extension). Specify path information with the `sourcePath` field.
`makeInfo.linkLibsObjs`	A cell array that specifies additional, fully qualified paths to object or library files against which Real-Time Workshop generated code should link. The Real-Time Workshop build process does not compile the specified objects and libraries. However, it includes them when linking the final executable. This can be useful for incorporating libraries that you do not want the Real-Time Workshop build process to recompile or for which the source files are not available. You might also use this element to incorporate source files from languages other than C and C++. This is possible if you first create a C compatible object file or library outside of the Real-Time Workshop build process.
`makeInfo.precompile`	A Boolean flag that indicates whether the libraries specified in the `rtwmakecfg.m` file exist in a specified location (`precompile==1`) or if the libraries need to be created in the build directory during the Real-Time Workshop build process (`precompile==0`).
`makeInfo.library`	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The Real-Time Workshop build process expands the information into make rules in the generated makefile. See the next table for a list of the library fields.

The `makeInfo.library` field consists of the following elements:

Element	Description
`makeInfo.library(n).Name`	A character array that specifies the name of the library (without an extension).
`makeInfo.library(n).Location`	A character array that specifies the directory in which the library is located when precompiled. See the description of `makeInfo.precompile` in the preceding table for more information. A target can use the `TargetPreCompLibLocation` parameter to override this value. See "Specifying the Location of Precompiled Libraries" in the Real-Time Workshop documentation for details.
`makeInfo.library(n).Modules`	A cell array that specifies the C or C++ source file base names (without an extension) that comprise the library. Do not include the file extension. The makefile appends the appropriate object extension.

Note The `makeInfo.library` field must fully specify each library and how to build it. The modules list in the `makeInfo.library(n).Modules` element cannot be empty. If you need to specify a link-only library, use the `makeInfo.linkLibsObjs` field instead.

Example:

```
disp(['Running rtwmakecfg from directory: ',pwd]);
   makeInfo.includePath = { fullfile(pwd, 'somedir2') };
   makeInfo.sourcePath = {fullfile(pwd, 'somedir2'), fullfile(pwd, 'somedir3')};
   makeInfo.sources  = { 'src1.c', 'src2.cpp'};
   makeInfo.linkLibsObjs = { fullfile(pwd, 'somedir3', 'src3.object'),...
                             fullfile(pwd, 'somedir4', 'mylib.library')};
   makeInfo.precompile = 1;
   makeInfo.library(1).Name     = 'myprecompiledlib';
   makeInfo.library(1).Location = fullfile(pwd,'somdir2','lib');
   makeInfo.library(1).Modules  = {'srcfile1' 'srcfile2' 'srcfile3' };
```

Note If a path that you specify in the `rtwmakecfg.m` API contains spaces, the Real-Time Workshop software does not automatically convert the path to its non-space equivalent. If the build environments you intend to support do not support spaces in paths, refer to "Enabling the Real-Time Workshop Software to Build When Path Names Contain Spaces" in the Real-Time Workshop documentation.

Modifying the Template Makefile

To expand the information generated by an `rtwmakecfg` function, you can modify the following sections of your target's TMF:

- `Include Path`
- `C Flags` and/or `Additional Libraries`
- `Rules`

The TMF code examples below may not be appropriate for your make utility. For additional examples, see the GRT or ERT TMFs located in *`matlabroot`*`/rtw/c/grt/*.tmf` or *`matlabroot`*`/rtw/c/ert/*.tmf`.

Example — Adding Directory Names to the Makefile Include Path. The following TMF code example adds directory names to the include path in the generated makefile:

```
ADD_INCLUDES = \
|>START_EXPAND_INCLUDES<|   -I|>EXPAND_DIR_NAME<| \
|>END_EXPAND_INCLUDES<|
```

Additionally, the `ADD_INCLUDES` macro must be added to the `INCLUDES` line, as shown below.

```
INCLUDES = -I. -I.. $(MATLAB_INCLUDES) $(ADD_INCLUDES) $(USER_INCLUDES)
```

Example — Adding Library Names to the Makefile. The following TMF code example adds library names to the generated makefile.

```
LIBS =
|>START_PRECOMP_LIBRARIES<|
```

```
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_PRECOMP_LIBRARIES<|
|>START_EXPAND_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_EXPAND_LIBRARIES<|
```

For more information on how to use configuration parameters to control library names and location during the build process, see "Controlling the Location and Naming of Libraries During the Build Process" in the Real-Time Workshop documentation.

Example — Adding Rules to the Makefile. The following TMF code example adds rules to the generated makefile.

```
|>START_EXPAND_RULES<|
$(BLD)/%.o: |>EXPAND_DIR_NAME<|/%.c $(SRC)/$(MAKEFILE) rtw_proj.tmw
    @$(BLANK)
    @echo ### "|>EXPAND_DIR_NAME<|\$*.c"
    $(CC) $(CFLAGS) $(APP_CFLAGS) -o $(BLD)$(DIRCHAR)$*.o \
    |>EXPAND_DIR_NAME<|$(DIRCHAR)$*.c > $(BLD)$(DIRCHAR)$*.lst
|>END_EXPAND_RULES<|

|>START_EXPAND_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|    |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
|>END_EXPAND_LIBRARIES<|

|>START_PRECOMP_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|    |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
```

```
|>END_PRECOMP_LIBRARIES<|
```

Supporting Continuous Time in Custom Targets

If you want your custom ERT-based target to support continuous time, you must update your template makefile (TMF) and the static main program module (for example, `mytarget_main.c`) for your target.

Template Makefile Modifications

Add the `NCSTATES` token expansion after the `NUMST` token expansion, as follows:

```
NUMST = |>NUMST<|
NCSTATES = |>NCSTATES<|
```

In addition, add `NCSTATES` to the `CPP_REQ_DEFINES` macro, as in the following example:

```
CPP_REQ_DEFINES = -DMODEL=$(MODEL) -DNUMST=$(NUMST) -DNCSTATES=$(NCSTATES) \
-DMAT_FILE=$(MAT_FILE)
-DINTEGER_CODE=$(INTEGER_CODE) \
-DONESTEPFCN=$(ONESTEPFCN) -DTERMFCN=$(TERMFCN) \
-DHAVESTDIO
-DMULTI_INSTANCE_CODE=$(MULTI_INSTANCE_CODE) \
```

Modifications to Main Program Module

The main program module defines a static main function that manages task scheduling for all supported tasking modes of single- and multiple-rate models. `NUMST` (the number of sample times in the model) determines whether the main function calls multirate or single-rate code. However, when a model uses continuous time, it is incorrect to rely on `NUMST` directly.

When the model has continuous time and the flag `TID01EQ` is true, both continuous time and the fastest discrete time are treated as one rate in generated code. The code associated with the fastest discrete rate is guarded by a major time step check. When the model has only two rates, and `TID01EQ` is true, the generated code has a single-rate call interface.

To support models that have continuous time, update the static main module to take TID01EQ into account, as follows:

1. Before NUMST is referenced in the file, add the following code:

```
#if defined(TID01EQ) && TID01EQ == 1 && NCSTATES == 0
#define DISC_NUMST (NUMST - 1)
#else
#define DISC_NUMST NUMST
#endif
```

2. Replace all instances of NUMST in the file by DISC_NUMST.

Model Reference Considerations

See Chapter 7, "Supporting Optional Features" for important information on TMF modifications you may need to make to support the Real-Time Workshop model referencing features.

Generating Make Commands for Nondefault Compilers

Custom targets may need a target-specific hook file to generate an appropriate make command when a nondefault compiler is used. This file can be used to override the default Real-Time Workshop behavior for selecting the appropriate compiler tool to be used in the build process. See "STF_wrap_make_cmd_hook.m" on page 4-13 for further details.

7

Supporting Optional Features

Overview

This chapter describes how to configure a custom embedded target to support any of the following optional features:

Optional Feature	Target Configuration Parameters
Building a model that includes referenced models	`ModelReferenceCompliant` `ParMdlRefBuildCompliant` (parallel build support)
Controlling the compiler optimization level for building generated code	`CompOptLevelCompliant`
Controlling inclusion of the `firstTime` argument in the *`model`*`_initialize` function generated for a Simulink model.	`ERTFirstTimeCompliant` (ERT only)
Controlling the C function prototypes of initialize and step functions that are generated for a Simulink model	`ModelStepFunctionPrototypeControl-Compliant` (ERT only)
Generating and configuring C++ encapsulation interfaces to model code	`CPPClassGenCompliant` (ERT only)

The required configuration changes are modifications to your system target file (STF), and in some cases also modifications to your template makefile (TMF) or your custom static main program.

The API for STF callbacks provides a function `SelectCallback` for use in STFs. `SelectCallback` is associated with the target rather than with any of its individual options. If you implement a `SelectCallback` function for a target, it is triggered whenever the user selects the target in the System Target File Browser.

Additionally, the API provides the functions `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` for controlling custom target configuration options from a user-written `SelectCallback` function.

(For function descriptions and examples, see "System Target File Callback Interface" in the Real-Time Workshop Embedded Coder reference documentation.)

The general requirements for supporting one of the optional features include:

- To support model referencing or compiler optimization level control, the target must be derived from the GRT or the ERT target. To support `firstTime` argument control, C function prototype control, or C++ encapsulation interface control, the target must be derived from the ERT target.
- The system target file (STF) must declare feature compliance by including one of the target configuration parameters listed above in a `SelectCallback` function call.
- Additional changes such as TMF modifications or static main program modifications may be required, depending on the feature. See the detailed steps in the subsections for individual features.

Supporting Model Referencing

In this section...

Overview

This section describes how to configure a custom embedded target to support model referencing. Without the described modifications, you will not be able to use the custom target when building a model that includes referenced models. If you do not intend to use referenced models with your target, you can skip this section. If you later find that you need to use referenced models, you can upgrade your target then.

The requirements for supporting model referencing are as follows:

- The target must be derived from the GRT target or the ERT target.
- The system target file (STF) must declare model reference compliance, as described in "Declaring Model Referencing Compliance" on page 7-5.
- The template makefile (TMF) must define some entities that support model referencing, as described in "Providing Model Referencing Support in the TMF" on page 7-6.
- The TMF must support using the Shared Utilities directory, as described in "Supporting the Shared Utilities Directory" on page 7-10.
- To optionally configure a target to support parallel builds for large model reference hierarchies (see "Reducing Build Time for Referenced Models" in the Real-Time Workshop documentation), the STF and TMF must be modified for parallel builds as described in "Declaring Model Referencing

Compliance" on page 7-5 and "Providing Model Referencing Support in the TMF" on page 7-6.

- You can optionally define additional capabilities that support model referencing, as described in "Preventing Resource Conflicts (Optional)" on page 7-14.

See "Referencing a Model" for information about model referencing in Simulink models, and "Creating Model Components" for information about model referencing in Real-Time Workshop generated code.

Declaring Model Referencing Compliance

To declare model reference compliance for your target, you must implement a callback function that sets the `ModelReferenceCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = ['custom_select_callback_handler(hDlg, hSrc)'];
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ModelReferenceCompliant` flag as follows:

```
slConfigUISetVal(hDlg, hSrc, 'ModelReferenceCompliant', 'on');
slConfigUISetEnabled(hDlg, hSrc, 'ModelReferenceCompliant', false);
```

If you might use the target to build models containing large model reference hierarchies, consider configuring the target to support parallel builds, as discussed in "Reducing Build Time for Referenced Models" in the Real-Time Workshop documentation.

To configure a target for parallel builds, your callback function must also set the `ParMdlRefBuildCompliant` flag as follows:

```
slConfigUISetVal(hDlg, hSrc, 'ParMdlRefBuildCompliant', 'on');
slConfigUISetEnabled(hDlg, hSrc, 'ParMdlRefBuildCompliant', false);
```

For more information about the STF callback API, see "System Target File Callback Interface" in the Real-Time Workshop Embedded Coder reference documentation.

Providing Model Referencing Support in the TMF

Do the following to configure the template makefile (TMF) to support model referencing:

1. Add the following make variables and tokens to be expanded when the makefile is generated:

```
MODELREFS                  = |>MODELREFS<|
MODELLIB                   = |>MODELLIB<|
MODELREF_LINK_LIBS         = |>MODELREF_LINK_LIBS<|
MODELREF_LINK_RSPFILE      = |>MODELREF_LINK_RSPFILE_NAME<|
MODELREF_INC_PATH        = |>START_MDLREFINC_EXPAND_INCLUDES<|\
    -I|>MODELREF_INC_PATH<| |>END_MDLREFINC_EXPAND_INCLUDES<|
RELATIVE_PATH_TO_ANCHOR    = |>RELATIVE_PATH_TO_ANCHOR<|
MODELREF_TARGET_TYPE       = |>MODELREF_TARGET_TYPE<|
```

   The following code excerpt shows how makefile tokens are expanded for a referenced model.

```
MODELREFS                =
MODELLIB                 = engine3200cc_rtwlib.a
MODELREF_LINK_LIBS       =
MODELREF_LINK_RSPFILE    =
MODELREF_INC_PATH        =
RELATIVE_PATH_TO_ANCHOR  = ../../..
MODELREF_TARGET_TYPE     = RTW
```

   The following code excerpt shows how makefile tokens are expanded for the top model that references the referenced model.

```
MODELREFS                = engine3200cc transmission
MODELLIB                 = archlib.a
MODELREF_LINK_LIBS       = engine3200cc_rtwlib.a transmission_rtwlib.a
```

```
MODELREF_LINK_RSPFILE     =
MODELREF_INC_PATH      = -I../slprj/ert/engine3200cc -I../slprj/ert/transmission
RELATIVE_PATH_TO_ANCHOR   = ..
MODELREF_TARGET_TYPE      = NONE
```

Token	Expands to
MODELREFS for the top model	List of referenced model names.
MODELLIB	Name of the library generated for the model.
MODELREF_LINK_LIBS token for the top model	List of referenced model libraries that the top model links against.
MODELREF_LINK_RSPFILE token for the top model	Name of a response file that the top model links against. This token is valid only for build environments that support linker response files. For an example of its use, see *matlabroot*/rtw/c/grt/grt_vc.tmf.
MODELREF_INC_PATH token for the top model	Include path to the referenced models.
RELATIVE_PATH_TO_ANCHOR	Relative path, from the location of the generated makefile, to the MATLAB working directory.
MODELREF_TARGET_TYPE	Signifies the type of target being built. Possible values are • NONE: Standalone model or top model referencing other models • RTW: Model reference Real-Time Workshop target build • SIM: Model reference simulation target build

If you are configuring your target to support parallel builds, as discussed in "Reducing Build Time for Referenced Models" in the Real-Time Workshop documentation, you must also add the following token definitions to your TMF:

```
START_DIR = |>START_DIR<|
MASTER_ANCHOR_DIR = |>MASTER_ANCHOR_DIR<|
```

Token	Expands to
START_DIR	Current work directory (pwd) at the time the build started.
MASTER_ANCHOR_DIR	Current work directory (pwd) at the time the build started.

2 Add RELATIVE_PATH_TO_ANCHOR and MODELREF_INC_PATH include paths to the overall INCLUDES variable.

```
INCLUDES = -I. -I$(RELATIVE_PATH_TO_ANCHOR) $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
    $(USER_INCLUDES) $(MODELREF_INC_PATH) $(SHARED_INCLUDES)
```

3 Change the SRCS variable in your TMF so that it initially lists only common modules. Additional modules are then appended conditionally, as described in the next step. For example, change

```
SRCS  = $(MODEL).c $(MODULES) ert_main.c $(ADD_SRCS) $(EXT_SRC)
```

to

```
SRCS = $(MODULES) $(S_FUNCTIONS)
```

4 Create variables to define the final target of the makefile. You can remove any variables that may have existed for defining the final target. For example, remove

```
PROGRAM    = ../$(MODEL)
```

and replace it with

```
ifeq ($(MODELREF_TARGET_TYPE), NONE)
  # Top model for RTW
  PRODUCT            = $(RELATIVE_PATH_TO_ANCHOR)/$(MODEL)
  BIN_SETTING        = $(LD) $(LDFLAGS) -o $(PRODUCT) $(SYSLIBS)
  BUILD_PRODUCT_TYPE = "executable"
  # ERT based targets
  SRCS               += $(MODEL).c ert_main.c $(EXT_SRC)
  # GRT based targets
```

```
  # SRCS             += $(MODEL).c grt_main.c rt_sim.c $(EXT_SRC) $(SOLVER)

else
  # sub-model for RTW
  PRODUCT            = $(MODELLIB)
  BUILD_PRODUCT_TYPE = "library"
endif
```

5 Create rules for the final target of the makefile (replace any existing final target rule). For example:

```
ifeq ($(MODELREF_TARGET_TYPE),NONE)
  # Top model for RTW
  $(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS) $(MODELREF_LINK_LIBS)
               $(BIN_SETTING) $(LINK_OBJS) $(MODELREF_LINK_LIBS)
               $(SHARED_LIB) $(LIBS)
               @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
else
  # sub-model for RTW
  $(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS)
               @rm -f $(MODELLIB)
               $(ar) ruvs $(MODELLIB) $(LINK_OBJS)
               @echo "### Created $(MODELLIB)"
               @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
endif
```

6 Create a rule to allow submodels to compile files that reside in the MATLAB working directory (`pwd`).

```
%.o : $(RELATIVE_PATH_TO_ANCHOR)/%.c
 $(CC) -c $(CFLAGS) $<
```

Controlling Configuration Option Value Agreement

By default, the value of any configuration option defined in the system target file for a TLC-based custom target must be the same in any referenced model and its parent model. To relax this requirement, include the `modelReferenceParameterCheck` field in the `rtwoptions` structure element that defines the configuration option, and set the value of the field to `'off'`. For example:

```
rtwoptions(2).prompt         = 'My Custom Parameter';
```

```
rtwoptions(2).type           = 'Checkbox';
rtwoptions(2).default        = 'on';
rtwoptions(2).modelReferenceParameterCheck  =  on ;
rtwoptions(2).tlcvariable    = 'mytlcvariable';
...
```

The configuration option **My Custom Parameter** can differ in a referenced model and its parent model. See Chapter 5, "System Target Files" for information about TLC-based system target files, and rtwoptions Structure Fields Summary on page 5-14for a list of all `rtwoptions` fields.

Supporting the Shared Utilities Directory

Overview

The makefile used by the Real-Time Workshop build process must support compiling and creating libraries, and so on, from the locations in which the code is generated. Therefore, you need to update your makefile and the model reference build process to support the shared utilities location. The **Utility function generation** options have the following requirements:

- `Auto`
  - Standalone model build — All files go to the build directory; no makefile updates needed.
  - Referenced model or top model build — Use shared utilities directory; makefile requires full model reference support.
- `Shared location`
  - Standalone model build — Use shared utility directory; makefile requires shared location support.
  - Referenced model or top model build — Use shared utilities directory; makefile requires full model reference support.

The shared utilities directory (`slprj/`*`target`*`/_sharedutils`) typically stores generated utility code that is common between a top model and the models

it references. You can also force the build process to use a shared utilities directory for a standalone model. See "Project Directory Structure for Model Reference Targets" in the Real-Time Workshop documentation for details.

If you want your target to support compilation of code generated in the shared utilities directory, several updates to your template makefile (TMF) are required. Note that support for the shared utilities directory is a necessary, but not sufficient, condition for supporting Model Reference builds. See the preceding sections of this chapter to learn about additional updates that are needed for supporting Model Reference builds.

The exact syntax of the changes can vary due to differences in the make utility and compiler/archiver tools used by your target. The examples below are based on the GNU[6] make utility. You can find the following updated TMF examples for GNU and Microsoft Visual C++ make utilities in the GRT and ERT target directories:

- GRT: *matlabroot*/rtw/c/grt/
  - grt_lcc.tmf
  - grt_vc.tmf
  - grt_unix.tmf
- ERT: *matlabroot*/rtw/c/ert/
  - ert_lcc.tmf
  - ert_vc.tmf
  - ert_unix.tmf

Use the GRT or ERT examples as a guide to the location, within the TMF, of the changes and additions described below.

Note The ERT-based TMFs contain extra code to handle generation of ERT S-functions and Model Reference simulation targets. Your target does not need to handle these cases.

---

6. GNU® is a registered trademark of the Free Software Foundation.

Implementing Shared Utilities Directory Support

Make the following changes to your TMF to support the shared utilities directory:

1. Add the following make variables and tokens to be expanded when the makefile is generated:

```
SHARED_SRC      = |>SHARED_SRC<|
SHARED_SRC_DIR  = |>SHARED_SRC_DIR<|
SHARED_BIN_DIR  = |>SHARED_BIN_DIR<|
SHARED_LIB      = |>SHARED_LIB<|
```

   `SHARED_SRC` specifies the shared utilities directory location and the source files in it. A typical expansion in a makefile is

```
SHARED_SRC      = ../slprj/ert/_sharedutils/*.c
```

   `SHARED_LIB` specifies the library file built from the shared source files, as in the following expansion.

```
SHARED_LIB      = ../slprj/ert/_sharedutils/rtwshared.lib
```

   `SHARED_SRC_DIR` and `SHARED_BIN_DIR` allow specification of separate directories for shared source files and the library compiled from the source files. In the current release, all TMFs use the same path, as in the following expansions.

```
SHARED_SRC_DIR  = ../slprj/ert/_sharedutils
SHARED_BIN_DIR  = ../slprj/ert/_sharedutils
```

2. Set the `SHARED_INCLUDES` variable according to whether shared utilities are in use. Then append it to the overall `INCLUDES` variable.

```
SHARED_INCLUDES =
ifneq ($(SHARED_SRC_DIR),)
SHARED_INCLUDES = -I$(SHARED_SRC_DIR)
endif

INCLUDES = -I. $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
           $(USER_INCLUDES) $(SHARED_INCLUDES)
```

3. Update the `SHARED_SRC` variable to list all shared files explicitly.

```
SHARED_SRC := $(wildcard $(SHARED_SRC))
```

4 Create a `SHARED_OBJS` variable based on `SHARED_SRC`.

```
SHARED_OBJS = $(addsuffix .o, $(basename $(SHARED_SRC)))
```

5 Create an `OPTS` (options) variable for compilation of shared utilities.

```
SHARED_OUTPUT_OPTS = -o $@
```

6 Provide a rule to compile the shared utility source files.

```
$(SHARED_OBJS) : $(SHARED_BIN_DIR)/%.o : $(SHARED_SRC_DIR)/%.c
 $(CC) -c $(CFLAGS) $(SHARED_OUTPUT_OPTS) $<
```

7 Provide a rule to create a library of the shared utilities. The following example is based on UNIX[7].

```
$(SHARED_LIB) : $(SHARED_OBJS)
   @echo "### Creating $@ "
   ar r $@ $(SHARED_OBJS)
   @echo "### Created  $@ "
```

Note Depending on your make utility, you may be able to combine Steps 6 and 7 into one rule. For example, `gmake` (used with `ert_unix.tmf`) uses:

```
$(SHARED_LIB) : $(SHARED_SRC)
    @echo "### Creating $@ "
    cd $(SHARED_BIN_DIR); $(CC) -c $(CFLAGS) $(GCC_WALL_FLAG_MAX) $(notdir $?)
    ar ruvs $@ $(SHARED_OBJS)
    @echo "### $@ Created "
```

See this and other examples in the files `ert_vc.tmf`, `ert_lcc.tmf`, and `ert_unix.tmf` located at *`matlabroot`*`/rtw/c/ert`.

---

8 Add `SHARED_LIB` to the rule that creates the final executable.

```
$(PROGRAM) : $(OBJS) $(LIBS) $(SHARED_LIB)
```

7. UNIX® is a registered trademark of The Open Group in the United States and other countries.

```
 $(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(LIBS)
$(SHARED_LIB) $(SYSLIBS)
 @echo "### Created executable: $(MODEL)"
```

9 Remove any explicit reference to `rt_nonfinite.c` from your TMF. For example, change

```
ADD_SRCS = $(RTWLOG) rt_nonfinite.c
```

to

```
ADD_SRCS = $(RTWLOG)
```

Note If your target interfaces to a development environment that is not makefile based, you must make equivalent changes to provide the needed information to your target compilation environment.

Preventing Resource Conflicts (Optional)

Hook files are optional M-files and TLC files that are invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.

If you are adapting your custom target for code generation compatibility with model reference features, consider adding checks to your hook files for handling referenced models differently than top models to prevent resource conflicts.

For example, consider adding the following check to your *STF*`_make_rtw_hook.m file`:

```
% Check if this is a referenced model
mdlRefTargetType = get_param(codeGenModelName,`ModelReferenceTargetType');
isNotModelRefTarget = strcmp(mdlRefTargetType, `NONE'); % NONE, SIM, or RTW
if isNotModelRefTarget
  % code that is specific to the top model
else
  % code that is specific to the referenced model
end
```

You may need to do a similar check in your TLC code.

```
%if !IsModelReferenceTarget()
  %% code that is specific to the top model
%else
  %% code that is specific to the referenced model
%endif
```

Supporting Compiler Optimization Level Control

In this section...

Overview

This chapter describes how to configure a custom embedded target to support compiler optimization level control. Without the described modifications, you will not be able to use the **Compiler optimization level** parameter on the **Real-Time Workshop** pane of the Configuration Parameters dialog box to control the compiler optimization level for building generated code. (For more information about compiler optimization level control, see "Compiler optimization level" in the Real-Time Workshop reference documentation.)

The requirements for supporting compiler optimization level control are as follows:

- The target must be derived from the GRT target or the ERT target.
- The system target file (STF) must declare compiler optimization level control compliance, as described in "Declaring Compiler Optimization Level Control Compliance" on page 7-16.
- The target makefile must honor the setting for **Compiler optimization level**, as described in "Providing Compiler Optimization Level Control Support in the Target Makefile" on page 7-17.

Declaring Compiler Optimization Level Control Compliance

To declare compiler optimization level control compliance for your target, you must implement a callback function that sets the `CompOptLevelCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For

example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = ['custom_select_callback_handler(hDlg, hSrc)'];
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `CompOptLevelCompliant` flag as follows:

```
slConfigUISetVal(hDlg, hSrc, 'CompOptLevelCompliant', 'on');
slConfigUISetEnabled(hDlg, hSrc, 'CompOptLevelCompliant', false);
```

For more information about the STF callback API, see "System Target File Callback Interface" in the Real-Time Workshop Embedded Coder reference documentation.

When the `CompOptLevelCompliant` target configuration parameter is set to `on`, the **Compiler optimization level** parameter is displayed in the **Real-Time Workshop** pane of the Configuration Parameters dialog box for your model.

Providing Compiler Optimization Level Control Support in the Target Makefile

As part of supporting compiler optimization level control for your target, you must modify the target makefile to honor the setting for **Compiler optimization level**. Use a GRT or ERT target provided by The MathWorks as a model for making the modifications.

Supporting firstTime Argument Control

In this section...

Overview

This chapter describes how to configure a custom embedded target to support `firstTime` argument control. Without the described modifications, you will not be able to use the `IncludeERTFirstTime` model configuration parameter to control inclusion of the `firstTime` argument in the *`model`*`_initialize` function generated for your model. (For more information about `firstTime` argument control, see model_initialize in the Real-Time Workshop Embedded Coder reference documentation.)

The requirements for supporting `firstTime` argument control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare `firstTime` argument control compliance, as described in "Declaring firstTime Argument Control Compliance" on page 7-18.
- If your target uses a custom static main program, the main program must handle the inclusion and suppression of the `firstTime` argument for a given model, as described in "Providing firstTime Argument Control Support in the Custom Static Main Program" on page 7-19.

Declaring firstTime Argument Control Compliance

To declare `firstTime` argument control compliance for your target, you must implement a callback function that sets the `ERTFirstTimeCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For

example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = ['custom_select_callback_handler(hDlg, hSrc)'];
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ERTFirstTimeCompliant` flag as follows:

```
slConfigUISetVal(hDlg, hSrc, 'ERTFirstTimeCompliant', 'on');
slConfigUISetEnabled(hDlg, hSrc, 'ERTFirstTimeCompliant', false);
```

For more information about the STF callback API, see "System Target File Callback Interface" in the Real-Time Workshop Embedded Coder reference documentation.

When the `ERTFirstTimeCompliant` target configuration parameter is set to `on`, you can use the `IncludeERTFirstTime` model configuration parameter to control inclusion of the `firstTime` argument in the *`model`*`_initialize` function generated for your model.

Note If the `ERTFirstTimeCompliant` parameter is set to `off` for your selected target, you cannot control inclusion of the `firstTime` argument in the *`model`*`_initialize` function. If you attempt to include or suppress the `firstTime` argument using the `IncludeERTFirstTime` model configuration parameter, Real-Time Workshop Embedded Coder software ignores the request, and your generated *`model`*`_initialize` function will not reflect the requested change.

Providing firstTime Argument Control Support in the Custom Static Main Program

If your target uses a custom static main program, you must update the static main program to handle the inclusion and suppression of the `firstTime` argument for a given model. One way to do this is to

1. Make sure the target TLC file assigns 1 to `AutoBuildProcedure` when using a static main program. For example,

   ```
   %assign AutoBuildProcedure = !GenerateSampleERTMain
   ```

2. In the generated header file `autobuild.h`, the macro INCLUDE_FIRST_TIME_ARG will be set to 0 if the `IncludeERTFirstTime` parameter is set to `off` or 1 if the parameter is set to `on`.

3. Inside the static main program, make sure to `#include autobuild.h` and then conditionally compile declarations and calls to the *`model`*`_initialize` function, based on the value of the INCLUDE_FIRST_TIME_ARG macro.

Supporting C Function Prototype Control

In this section...

Overview

This chapter describes how to configure a custom embedded target to support C function prototype control. Without the described modifications, you will not be able to use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the function prototypes of initialize and step functions that are generated for your model. (For more information about C function prototype control, see "Controlling Generation of Function Prototypes" in the Real-Time Workshop Embedded Coder documentation.)

The requirements for supporting C function prototype control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare C function prototype control compliance, as described in "Declaring C Function Prototype Control Compliance" on page 7-21.
- If your target uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, the static main program must call the function prototype controlled initialize and step functions, as described in "Providing C Function Prototype Control Support in the Custom Static Main Program" on page 7-22.

Declaring C Function Prototype Control Compliance

To declare C function prototype control compliance for your target, you must implement a callback function that sets the `ModelStepFunctionPrototypeControlCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings`

structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = ['custom_select_callback_handler(hDlg, hSrc)'];
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ModelStepFunctionPrototypeControlCompliant` flag as follows:

```
slConfigUISetVal(hDlg, hSrc, 'ModelStepFunctionPrototypeControlCompliant', 'on');
slConfigUISetEnabled(hDlg, hSrc, 'ModelStepFunctionPrototypeControlCompliant', false);
```

For more information about the STF callback API, see "System Target File Callback Interface" in the Real-Time Workshop Embedded Coder reference documentation.

When the `ModelStepFunctionPrototypeControlCompliant` target configuration parameter is set to `on`, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the function prototypes of initialize and step functions that are generated for your model.

Providing C Function Prototype Control Support in the Custom Static Main Program

If your target uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, you must update the static main program to call the function prototype controlled initialize and step functions. You can do this in either of the following ways:

1. Manually adapt your static main program to declare appropriate model data and call the function prototype controlled initialize and step functions.
2. Generate your main program using **Generate an example main program** on the **Templates** pane of the Configuration Parameters dialog

box. The generated main program declares model data and calls the function prototype controlled initialize and step function appropriately.

Supporting C++ Encapsulation Interface Control

In this section...

Overview

This chapter describes how to configure a custom embedded target to support C++ encapsulation interface control. Without the described modifications, you will not be able to use the `C++ (Encapsulated)` language option and the **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ encapsulation interfaces to model code. (For more information about C++ encapsulation interface control, see "Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation.)

The requirements for supporting C++ encapsulation interface control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare C++ encapsulation interface control compliance, as described in "Declaring C++ Encapsulation Interface Control Compliance" on page 7-24.

Declaring C++ Encapsulation Interface Control Compliance

To declare C++ encapsulation interface control compliance for your target, you must implement a callback function that sets the `CPPClassGenCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = ['custom_select_callback_handler(hDlg, hSrc)'];
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `CPPClassGenCompliant` flag as follows:

```
slConfigUISetVal(hDlg, hSrc, 'CPPClassGenCompliant', 'on');
slConfigUISetEnabled(hDlg, hSrc, 'CPPClassGenCompliant', false);
```

For more information about the STF callback API, see "System Target File Callback Interface" in the Real-Time Workshop Embedded Coder reference documentation.

When the `CPPClassGenCompliant` target configuration parameter is set to `on`, you can use the `C++ (Encapsulated)` language option and the **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ encapsulation interfaces to model code.

Note Selecting the `C++ (Encapsulated)` language value for your model turns on the model option **Generate an example main program**. With this option on, code generation generates an example main program, `ert_main.cpp`. The generated example main program declares model data and calls the C++ encapsulation interface configured model step method appropriately, and demonstrates how the generated code can be deployed.

8

Using Target Preferences

Introduction to Target Preferences

In this section...

Prerequisites

The target preferences mechanism discussed in this section is based on Simulink data classes and data objects. This document assumes that you are familiar with Simulink data classes, packages, and objects, and with the use of the Simulink Data Class Designer.

If you are not familiar with these topics, read "Working with Data Objects" in the Simulink documentation.

Target Preference Classes and Properties

Target developers have found that it is often desirable to associate certain types of data with the target. For example, an embedded target may offer users a choice of several supported development systems (cross-compilers, debuggers, and so on). To invoke the correct development tool during the build process, the target needs information such as the user's choice of development tool, and the location on the host system where the user has installed the compiler and debugger executables. Other data associated with a target might specify host/target communications parameters, such as the communications port and baud rate to be used.

Target developers need a mechanism to define and store the properties they want to associate with their target. End users need a simple mechanism to set target property values. The *target preferences* feature meets these needs. Target preferences let you

- Structure the data associated with your target.
- Store data associated with your target persistently, across multiple models and across multiple MATLAB sessions.

- Provide end users with a simple GUI for changing, saving and loading their preferences. The target preferences feature also lets users perform the same functions from the MATLAB command line, or in M-files, with a simple set of commands.

To structure the data associated with your target, you define a *target preferences class* by specifying target properties and property types. The Simulink Data Class Designer simplifies this task.

Your target preferences class inherits methods from a base class (`RTW.TargetPrefs`) provided with the Real-Time Workshop Embedded Coder product. Inherited methods let you do the following with minimal effort:

- Manage persistent storage of preference data. The target preferences class stores such information to a MAT-file that can be easily retrieved, edited, and stored once again.
- Present a **Property Inspector** window to the end user, allowing for easy editing of preference property values.

You can also access target preferences through M-file utilities (for an example, see "Using Target Preferences in the Build Process" on page 8-11). You can use target preferences data during the build process by invoking such utilities from your TLC code. You can use the preference information in makefiles to invoke the user's preferred compiler or perform other target-specific tasks.

Creating Your Target Preferences Class

This section demonstrates the creation of a simple target preferences class using the Simulink Data Class Designer, and summarizes the methods inherited by this class.

This example assumes the skeletal target directory structure (as described in "Target Directory Structure and MATLAB Path" on page 4-4) has been created for an embedded target called `z80`.

The following naming convention is recommended for target preferences classes and packages:

- The package name should be in the form

  ```
  targetname
  ```

  where `targetname` is the name of the target.
- The recommended class name is `prefs`.

Thus the recommended `package.class` naming convention is

```
targetname.prefs
```

In this example, you define target preferences for a hypothetical embedded target for the Z80 microprocessor. The example defines a containing package `z80`, and a class `prefs`. The `prefs` class is a subclass of the `RTW.TargetPrefs` base class. The `z80` package is stored in the directory `z80\z80\@z80`.

1. Set your working directory to a directory that is *not* located anywhere in the MATLAB directory tree (that is, in or under the *`matlabroot`* directory). By the convention described in "Target Directory Structure and MATLAB Path" on page 4-4, enter

   ```
   cd z80\z80
   ```

2. Open the Simulink Data Class Designer by typing the following command at the MATLAB prompt.

   ```
   sldataclassdesigner('Create', 'ShowRTWTargetPrefs')
   ```

3 To define the package, click the **New** button next to the **Package name** field of the Data Class Designer. Enter the package name, `z80`.

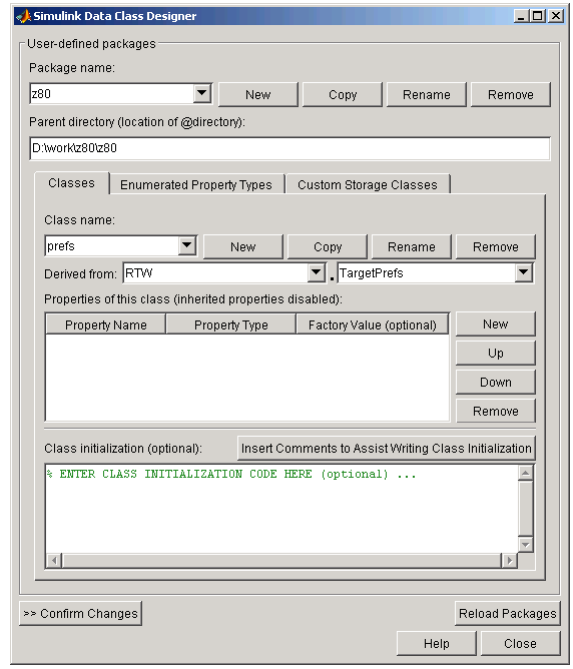
4 Click **OK** to create the new package in memory.

5 In the package **Parent directory** field, enter the path of the directory where you want to create the new package (for example, `D:\work\z80\z80`).

Note that the Data Class Designer creates the specified directory, if it does not already exist, when you save the package to your file system.

6 To define the target preferences class, click the **New** button on the **Classes** pane of the Data Class Designer dialog box. Enter the name of the new class, `prefs`, in the **Class name** field on the **Classes** pane.

7 Click **OK** to create the new class in memory.

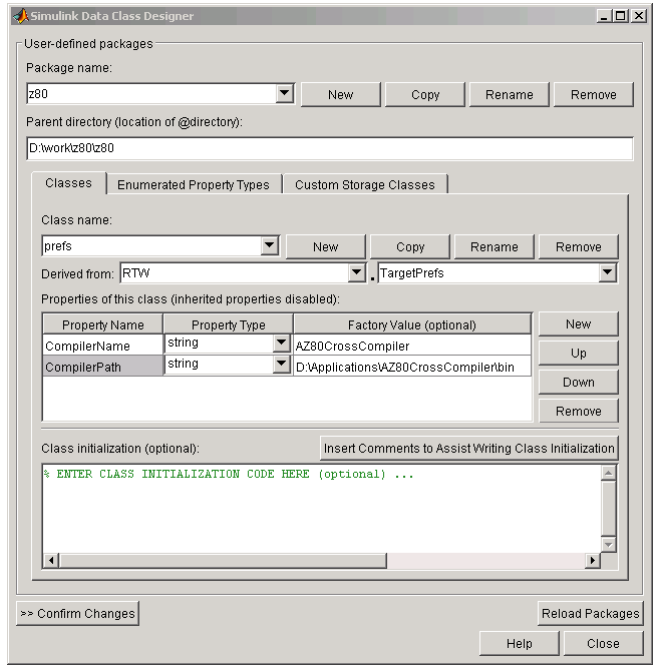


8 Select `RTW.TargetPrefs` as the parent class for the new class. To do this, first select the package name `RTW` from the left **Derived from** list box.

Then, select the class name `TargetPrefs` from the right **Derived from** list box.

Note that the list of properties in the **Properties of this class** field is empty. This is because the `RTW.TargetPrefs` parent class provides only methods, not properties.

9 Populate the list of properties by entering several property names and assigning data types and factory (default) values to them. (See "Defining Class Properties" in the Simulink documentation.) The figure below shows the **Properties of this class** field with two sample properties defined.



10 Click **Confirm changes**. the Data Class Designer displays the **Confirm changes** pane (not shown).

11 Select the package containing the new class definition and click **Write Selected** to save the new class definition.

The directory `z80\z80` now contains the package subdirectory,`\@z80`. The package subdirectory contains the class subdirectory, `@prefs`.

Target Preferences Class Methods

This section describes the methods that your target preferences class inherits from `RTW.TargetPrefs`.

To invoke these methods, instantiate an object of your target preferences class and use the syntax

```
method(objectname)
```

Note that to instantiate the target preferences object, you must use a static method, `load`, of the parent class `RTW.TargetPrefs`. For example:

```
z = RTW.TargetPrefs.load('z80.prefs');
disp(z)
    CompilerName: 'AZ80CrossCompiler'
    CompilerPath: 'D:\Applications\AZ80CrossCompiler\bin'
```

The inherited methods are summarized in this table.

Inherited Target Preferences Class Methods

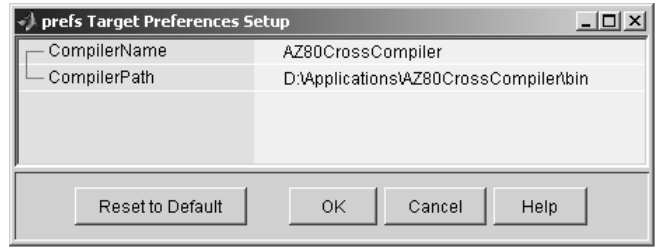
Method	Description
`disp`	Display the current property values of an object of the target preferences class in the MATLAB Command Window.
`reset`	Reset the current property values of an object of the target preferences class to the default (factory) values.
`getclassname`	Return the name of the class as a string.
`gui`	Using an existing object of the target preferences class, load the current property values in memory, and display a **Target Preferences Setup** window. Target Preferences Setup Window on page 8-9 shows an example of such a window.
`load('package.class'[, 'structure]')`	`load` is a static method of the parent class. Load the stored property values into an object of the package and class specified by the first argument. If the second argument is present, the return type is structure instead of object.
`save`	Write out the current property values of an object of the target preferences class.

Making Target Preferences Available to the End User

End users of your target will not normally need to invoke the methods described in "Target Preferences Class Methods" on page 8-8 (with the possible exception of the `gui` method). They only need to know how to open the **Target Preferences Setup** window to set the target properties.

The **Target Preferences Setup** window (shown below) allows the user to

- View and edit the property values.
- Save the property values.
- Reset the property values to their default (factory) values.
- Cancel the edit session.



Target Preferences Setup Window

The simplest way for users to access the **Target Preferences Setup** window is to invoke the `gui` method. This does not require you to provide any additional code.

A better approach, from the standpoint of usability, is to let the user open the **Target Preferences Setup** window from an icon under your target's toolbox in the MATLAB **Start** button. To make your target visible in the **Start** button, you must provide an `info.xml` file in the `mytarget/mytarget` directory (see "info.xml" on page 4-16).

To open the **Target Preferences Setup** window from the **Start** button, your `info.xml` file should also contain a section similar to the following example. This code provides a callback that executes when the user clicks on a standard

icon in the **Start** button. The callback instantiates a `z80` target preferences object and calls the `gui` method of that object.

```
<listitem>
<label>Z80 Target Preferences</label>
<callback>z80TargetPrefs = RTW.TargetPrefs.load( z80.prefs );
gui(z80TargetPrefs); </callback>
<icon>$toolbox/simulink/simulink/simulinkicon.gif</icon>
</listitem>
```

Only the text shown above in bold should be modified.

Once you have added the preceding section to your `info.xml` file, your customized target preferences appear in the **Start** button menu.

Note It is your responsibility to document the user-settable properties of your target. You should also document how users should access your target's preferences.

Using Target Preferences in the Build Process

In this section...

Introduction

This section discusses how to access your target preference data for use in the build process, including

- Two ways to access your target preference data in M-code
- How to return target preference data to a TLC variable

Accessing Target Preference Data from the MATLAB Prompt

Accessing target preference data from the MATLAB prompt or from an M-file is simpler than obtaining the same data in TLC. The following code instantiates a `z80` target preferences object in the MATLAB workspace, and loads the saved preferences data into the object. The `CompilerName` property is then directly accessed and assigned to a variable.

```
tp = RTW.TargetPrefs.load('z80.prefs');
targetName = tp.CompilerName;
```

The next section illustrates how to use the `load` method to return target preferences information to a TLC program.

Accessing Target Preference Data from TLC

You should create a `mytarget_settings.tlc` file to obtain target preferences data for use in the build process. The `mytarget_settings.tlc` file is invoked during the build process by a `%include` statement in the system target file. The `mytarget_settings.tlc` file is also useful for checking user code generation option settings, and other global settings affecting the code generation/build process.

As an example, consider the preferences for the z80 target defined in "Using Target Preferences in the Build Process" on page 8-11. A package/class z80.prefs is defined with properties CompilerName and CompilerPath, as shown in "Creating Your Target Preferences Class" on page 8-4.

The following TLC code examples from z80_settings.tlc show how to obtain the property values from the z80 target preferences and add them to the CompiledModel structure used in the build process.

This example performs a MATLAB evaluation of the load method (see previous section) that returns the property values to an intermediate TLC variable.

```
%assign Z80PREFS = FEVAL("RTW.TargetPrefs.load","z80.prefs","structure")
```

The next example creates a structure (Settings) for the property values within the CompiledModel record and populates the fields in CompiledModel.Settings with the data from the z80 target preferences.

```
%addtorecord CompiledModel Settings Z80PREFS;
```

CompiledModel.Settings can now be used as required by subsequently executing TLC code.

Now, consider an example where the target property values could be used in the build process. Suppose that a requirement for the Z80 target is to support two compilers. The decision as to which compiler is to be invoked during the build process is based on the CompilerName property, as set by the user.

The default value of CompilerName is 'AZ80CrossCompiler'. The AZ80CrossCompiler compiler tool chain is well suited for use with makefiles. If this compiler is specified, it is invoked using gmake and a template makefile, as is the case with most compilers invoked by Real-Time Workshop targets. Normally, a template makefile uses the variable CPP_REQ_DEFINES to contain a list of all the arguments specific to settings made to the model.

The alternative supported compiler, CodeSamurai, uses project files and COM automation, rather than a template makefile. If this compiler is specified, a different action should be taken to create a list of model settings and a list of files to be included in the project file.

The example code below invokes two TLC utilities (not shown) to generate a special header file (`cpp_req_defines.h`) and a list of files.

```
%if CompiledModel.Settings.CompilerName == "CodeSamurai"
%%
%% Generate cpp_req_defines.h and the list of RTW files resulting
%%from code generation.
%%
%include "gen_cpp_req_defines_h.tlc"
%include "gen_rtw_file_list.tlc"
%%
%else
 ... do something else for the the AZ80CrossCompiler compiler
%endif
```

Note that this code does not do any validation of the `CompilerName` setting. A more rigorous approach would be to define `CompilerName` as an enumerated type taking only two values. This would limit the user to a choice of two compiler names and avoid typing errors. Other validation could be done using the `CompilerPath` property. For example, the `CompilerPath` information could be used to access files located in the directories of the specified compiler, to detect that the proper compiler (or a specific required version of the compiler) was installed.

9

Interfacing to Development Tools

Introduction

Unless you are developing a target purely for code generation purposes, you will want your embedded target to support a complete build process. A full post-code generation build process includes

- Compilation of generated code
- Linking of compiled code and runtime libraries into an executable program module (or some intermediate representation of the executable code, such as S-Rec format)
- Downloading the executable to target hardware with a debugger or other utility
- Initiating execution of the downloaded program

Supporting a complete build process is inherently a complex task, because it involves interfacing to cross-development tools and utilities that are external to the Real-Time Workshop software.

If your development tools can be controlled with traditional makefiles and a make utility such as `gmake`, it may be relatively simple for you to adapt existing target files (such as the `ert.tlc` and `ert.tmf` files provided by the Real-Time Workshop Embedded Coder software) to your requirements. This approach is discussed in "Makefile Approach" on page 9-3.

Automating your build process through a modern integrated development environment (IDE) presents a different set of challenges. Each IDE has its own way of representing the set of source files and libraries for a project and for specifying build arguments. Interfacing to an IDE may require generation of specialized file formats required by the IDE (for example, project files) and, and also may require the use of inter-application communication (IAC) techniques to run the IDE. One such approach to build automation is discussed in "Interfacing to an Integrated Development Environment" on page 9-4.

Makefile Approach

A template makefile provides information about your model and your development system. The Real-Time Workshop build process uses this information to create an appropriate makefile (`.mk` file) to build an executable program. The Real-Time Workshop Embedded Coder product provides a number of template makefiles suitable for host-based compilers such as LCC (`ert_lcc.tmf`) and Microsoft Visual C++ (`ert_vc.tmf`).

Adapting one of the existing template makefiles to your cross-compiler's make utility may require little more than copying and renaming the template makefile in accordance with the conventions of your project.

If you need to make more extensive modifications, you need to understand template makefiles in detail. For a detailed description of the structure of template makefiles and of the tokens used in template makefiles, see Chapter 6, "Template Makefiles".

The following sections of this document supplement the basic template makefile information in the Real-Time Workshop documentation:

- "Supporting Multiple Development Environments" on page 5-37
- "Supplying Development Environment Information to Your Template Makefile" on page 3-16
- "mytarget_default_tmf.m" on page 4-11

Interfacing to an Integrated Development Environment

In this section...

Introduction

This section describes techniques that have been used to integrate embedded targets with integrated development environment (IDEs), including

- How to generate a header file containing directives to define variables (and their values) required by a non-makefile based build.
- Some problems and solutions specific to interfacing embedded targets with the Freescale Semiconductor CodeWarrior IDE. The examples provided should help you to deal with similar interfacing problems with your particular IDE.

Generating a CPP_REQ_DEFINES Header File

In Real-Time Workshop template makefiles, the token `CPP_REQ_DEFINES` is expanded and replaced with a list of parameter settings entered with various dialog boxes. This variable often contains information such as `MODEL` (name of generating model), `NUMST` (number of sample times in the model), `MT` (model is multitasking or not), and numerous other parameters (see "Template Makefiles and Tokens" on page 6-2).

The Real-Time Workshop makefile mechanism handles the `CPP_REQ_DEFINES` token automatically. If your target requires use of a project file, rather than the traditional makefile approach, you can generate a header file containing directives to define these variables and provide their values.

The following TLC file, `gen_rtw_req_defines.tlc`, provides an example. The code generates a C header file, `cpp_req_defines.h`. The information required to generate each`#define` directive is derived either from information in the *`model`*`.rtw` file (e.g., `CompiledModel.NumSynchronousSampleTimes`), or from make variables from the `rtwoptions` structure (e.g., `PurelyIntegerCode`).

```
%% File: gen_rtw_req_defines_h.tlc
%openfile CPP_DEFINES = "cpp_req_defines.h"
#ifndef _CPP_REQ_DEFINES_
#define _CPP_REQ_DEFINES_
#define MODEL %<CompiledModel.Name>
#define ERT  1
#define NUMST %<CompiledModel.NumSynchronousSampleTimes>
#define TID01EQ %<CompiledModel.FixedStepOpts.TID01EQ>
%%
%if CompiledModel.FixedStepOpts.SolverMode == "MultiTasking"
#define MT  1
#define MULTITASKING  1
%else
#define MT  0
#define MULTITASKING  0
%endif
%%
#define MAT_FILE  0
#define INTEGER_CODE %<PurelyIntegerCode>
#define ONESTEPFCN %<CombineOutputUpdateFcns>
#define TERMFCN %<IncludeMdlTerminateFcn>
%%
#define MULTI_INSTANCE_CODE  0
#define HAVESTDIO  0
#endif
%closefile CPP_DEFINES
```

Interfacing to the Freescale CodeWarrior IDE

Interfacing an embedded target's build process to the CodeWarrior IDE requires that two problems must be dealt with:

- The build process must generate a CodeWarrior compatible project file. This problem, and a solution, is discussed in "XML Project Import" on page 9-6. The solution described is applicable to any ASCII project file format.
- During code generation, the target must automate a CodeWarrior session that opens a project file and builds an executable. This task is described in "Build Process Automation" on page 9-10. The solution described is applicable to any IDE that can be controlled with Microsoft Component Object Model (COM) automation.

XML Project Import

This section illustrates how to use the Target Language Compiler (TLC) to generate an eXtensible Markup Language (XML) file, suitable for import into the CodeWarrior IDE, that contains all the necessary information about the source code generated by an embedded target.

The choice of XML format is dictated by the fact that the CodeWarrior IDE supports project export and import with XML files. As of this writing, native CodeWarrior project files are in a proprietary binary format.

Note that if your target needs to support some other compiler's project file format, you can apply the techniques shown here to virtually any ASCII file format (see "Generating a CPP_REQ_DEFINES Header File" on page 9-4).

To illustrate the basic concept, consider a hypothetical XML file exported from a CodeWarrior stationery project. The following is a partial listing:

```
<target>
   <settings>
    ...
   <\settings>
   <file><name>foo.c<\name>
   <\file>
   ...
   <file><name>foobar.c<\name>
   <\file>
   <fileref><name>foo.c<\name>
   <\fileref>
   ...
   <fileref><name>foobar.c<\name>
   <\fileref>
<\target>
```

Insert this XML code into an `%openfile/%closefile` block within a TLC file, `test.tlc`, as shown below.

```
%% test.tlc
%% This code will generate a file model_project.xml,
%% where model is the generating model name specified in
%% the CompiledModel.Name field of the model.rtw file.
%openfile XMLFileContents = %<CompiledModel.Name>_project.xml
<target>
   <settings>
    ...
   <\settings>
   <file><name>%<CompiledModel.Name>.c<\name>
   <\file>
   ...
   <file><name>foobar.c<\name>
   <\file>
   <fileref><name>%<CompiledModel.Name>.c<\name>
   <\fileref>
   ...
   <fileref><name>foobar.c<\name>
   <\fileref>
<\target>
%closefile XMLFileContents
%selectfile NULL_FILE
```

Note the use of the TLC token `CompiledModel.Name`. The token is resolved and the resulting filename is included in the output stream. You can specify other information, such as paths and libraries, in the output stream by specifying other tokens defined in *`model`*`.rtw`. For example, `System.Name` may be defined as `<Root>/Susbsystem1`.

Now suppose that `test.tlc` is invoked during a target's build process, where the generating model is `mymodel.mdl`. This should be done after the `codegenentry` statement. For example, `test.tlc` could be included directly in the system target file:

```
%include "codegenentry.tlc"
%include "test.tlc"
```

Alternatively, the `%include "test.tlc"` directive could be inserted into the `mytarget_genfiles.tlc` hook file, if present.

TLC tokens such as

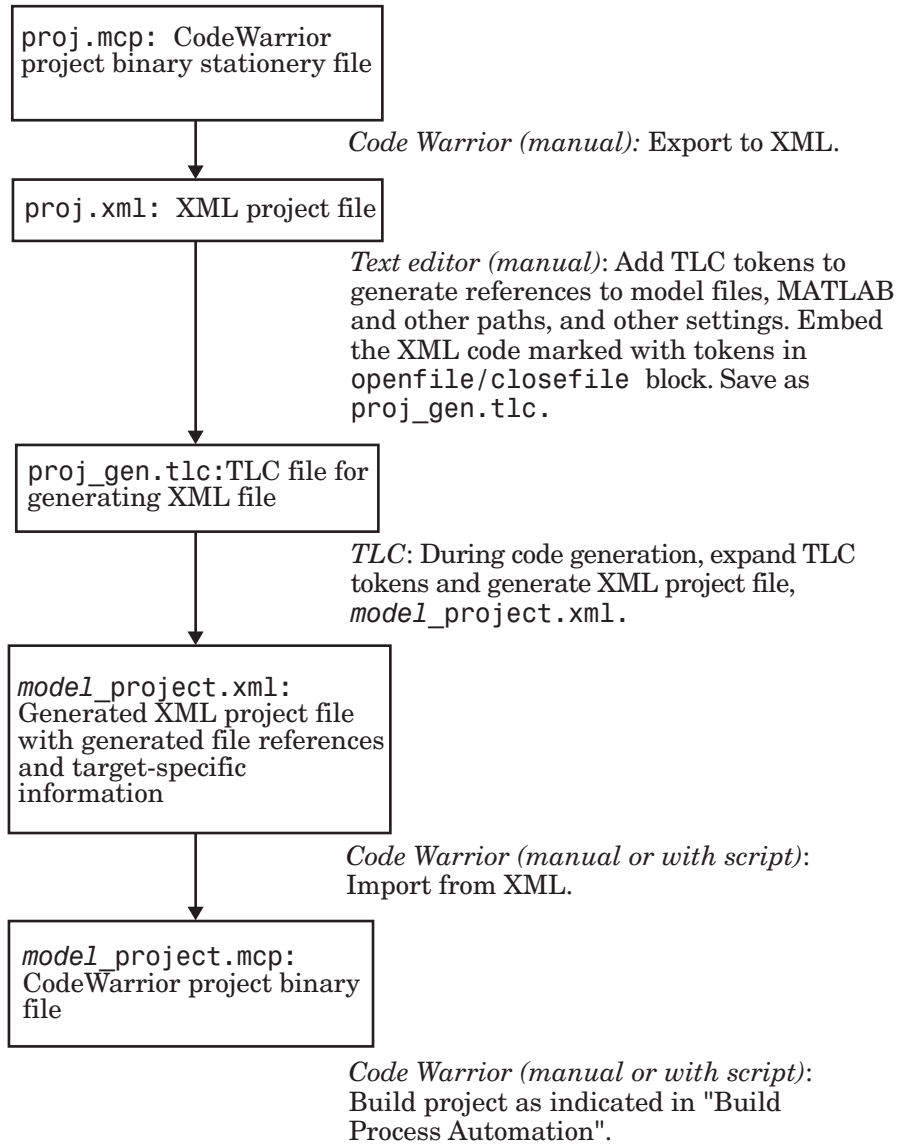
```
<file><name>%<CompiledModel.Name>.c<\name>
```

are expanded, with the `CompiledModel` record in the `mymodel.rtw` file, as in

```
<file><name>mymodel.c<\name>
```

`test.tlc` generates an XML file, file *`model`*`_project.xml`, from any model. *`model`*`_project.xml` contains references to generated code files. *`model`*`_project.xml` can be imported into the CodeWarrior IDE as a project.

The following flowchart summarizes this process.


proj.mcp: CodeWarrior project binary stationery file
Code Warrior (manual): Export to XML.
proj.xml: XML project file
Text editor (manual): Add TLC tokens to generate references to model files, MATLAB and other paths, and other settings. Embed the XML code marked with tokens in openfile/closefile block. Save as proj_gen.tlc.
proj_gen.tlc:TLC file for generating XML file
TLC: During code generation, expand TLC tokens and generate XML project file, model_project.xml.
model_project.xml: Generated XML project file with generated file references and target-specific information
Code Warrior (manual or with script): Import from XML.
model_project.mcp: CodeWarrior project binary file
Code Warrior (manual or with script): Build project as indicated in "Build Process Automation".

Note This process has drawbacks. First, manually editing an XML file exported from a CodeWarrior stationery project can be a laborious task, involving modification of a few dozen lines embedded within several thousand lines of XML code. Second, if you make changes to the CodeWarrior project after importing the generated XML file, the XML file must be exported and manually edited once again.

Build Process Automation

An application that supports COM automation can control any other application that includes a COM interface. Using MATLAB COM automation functions, an M-file can command a COM-compatible development system to execute tasks required by the build process.

The MATLAB COM automation functions described in this section are documented in the COM chapters of the *MATLAB External Interfaces* document.

For information about automation commands supported by the CodeWarrior IDE, see your CodeWarrior documentation.

COM automation is used by some embedded targets (for example, the Target Support Package product) to automate the CodeWarrior IDE to execute tasks such as:

- Opening a new CodeWarrior session
- Configure a project
- Loading a CodeWarrior project file
- Removing object code from the project
- Building or rebuilding the project
- Debug an application

COM technology automates certain repetitive tasks and allows the user to interact directly with the external application. For example, when the end user of the Target Support Package product initiates a build, the target

quickly invokes the necessary CodeWarrior actions and leaves a project built and ready to run with the IDE.

Example COM Automation Functions. The functions below use the MATLAB `actxserver` command to invoke COM functions for controlling the CodeWarrior IDE from a MATLAB M-file:

- `CreateCWComObject`: Create a COM connection to the CodeWarrior IDE.
- `OpenCW`: Open the CodeWarrior IDE without opening a project.
- `OpenMCP`: Open the CodeWarrior project file (`.mcp` file) specified by the input argument.
- `BuildCW`: Open the specified `.mcp` file, remove object code, and build project.

These functions are examples; they do not constitute a full implementation of a COM automation interface. If your target creates the project file during code generation, the top-level `BuildCW` function should be called after the code generation process is completed. Normally `BuildCW` would be called from the `exit` method of your *`STF`*`_make_rtw_hook.m` file (see "STF_make_rtw_hook.m" on page 4-12).

In the code examples, the variable `in_qualifiedMCP` is assumed to store a fully qualified path to a CodeWarrior project file (for example, path, filename, and extension). For example:

```
in_qualifiedMCP = 'd:\work\myproject.mcp';
```

In actual practice, your code is responsible for determining the conventions used for the project filename and location. One simple convention would be to default to a project file *`model`*`.mcp`, located in your target's build directory. Another approach would be to let the user specify the location of project files with the target preferences.

```
%=============================================================================
% Function: CreateCWComObject
% Abstract: Creates the COM connection to CodeWarrior
%
function ICodeWarriorApp = CreateCWComObject
  vprint([mfilename ': creating CW com object']);
  try
```

```
    ICodeWarriorApp = actxserver('CodeWarrior.CodeWarriorApp');
  catch
    error(['Error creating COM connection to ' ComObj ...
       '. Verify that CodeWarrior is installed correctly. Verify COM access to
CodeWarrior outside of MATLAB.']);
  end
  return;


%=========================================================================
% Function: OpenCW
% Abstract: Opens CodeWarrior without opening a project. Returns the
%           handle ICodeWarriorApp.
%
function ICodeWarriorApp = OpenCW()
  ICodeWarriorApp = CreateCWComObject;
  CloseAll;
  OpenMCP(in_qualifiedMCP);


%=========================================================================
% Function: OpenMCP
% Abstract: open an MCP project file
%
function OpenMCP(in_qualifiedMCP)
  % Argument checking. This method requires valid project file.
  if ~exist(in_qualifiedMCP)
      error([mfilename ': Missing or empty project file argument']);
  end
  if isempty(in_qualifiedMCP)
      error([mfilename ': Missing or empty project file argument']);
  end
  ICodeWarriorApp = CreateCWComObject;
  vprint([mfilename ': Importing']);
  try
    ICodeWarriorProject = ...
      invoke(ICodeWarriorApp.Application,...
      'OpenProject', in_qualifiedMCP,...
      1,0,0);
  catch
```

```
    error(['Error using COM connection to import project. ' ...
       ' Verify that CodeWarrior is installed correctly. Verify COM access to
CodeWarrior outside of MATLAB.']);
  end


%=========================================================================
% Function: BuildCW
% Abstract: Opens CodeWarrior.
%           Opens the specified CodeWarrior project.
%           Deletes objects.
%           Builds.
%
function ICodeWarriorApp = BuildCW(in_qualifiedMCP)
    % ICodeWarriorApp = BuildCW;
    ICodeWarriorApp = CreateCWComObject;
    CloseAll;
    OpenMCP(in_qualifiedMCP);
    try
      invoke(ICodeWarriorApp.DefaultProject,'RemoveObjectCode', 0, 1);
    catch
      error(['Error using COM connection to remove objects of current project. ' ...
       'Verify that CodeWarrior is installed correctly. Verify COM access to
CodeWarrior outside of MATLAB.']);
    end
    try
      invoke(ICodeWarriorApp.DefaultProject,'BuildAndWaitToComplete');
    catch
      error(['Error using COM connection to build current project. ' ...
       'Verify that CodeWarrior is installed correctly. Verify COM access to
CodeWarrior outside of MATLAB.']);
    end
```

10

Developing Device Drivers for Embedded Targets

Device Drivers Overview

In this section...

Introduction

Device drivers that communicate with target hardware are essential to many real-time development projects. This chapter discusses issues and solutions in the creation of device drivers specifically for embedded targets. This process includes incorporating drivers into your Simulink model and into the code generated from that model.

This chapter describes techniques for implementing device drivers as fully inlined S-functions. Like other inlined S-functions, fully inlined device drivers have a dual implementation:

- A C MEX S-function is implemented, primarily for use in simulation.
- A TLC implementation is created for use in code generation.

This chapter does not discuss the implementation of noninlined device drivers in detail. Although the Real-Time Workshop Embedded Coder product supports noninlined S-functions, you should use inlined device drivers for embedded applications, for reasons of efficiency. See "Inlined vs. Noninlined Drivers" on page 10-4 for a discussion of the tradeoffs.

Related Documentation

To implement device drivers, you should be familiar with the Simulink C MEX S-function format and API, and with the Target Language Compiler (TLC). These topics are covered in the following documents:

- The Simulink Writing S-Functions document describes C MEX S-functions and the S-function API in general. Writing S-Functions also describes how to access parameters from a masked S-function.
- The "Integrating External Code With Generated C and C++ Code" chapter of the Real-Time Workshop documentation is particularly important. It describes inlining, and how to use the special `mdlRTW` function to parameterize an inlined S-function.
- "Creating a Block Mask" in the Simulink User Guide describes how to mask S-function blocks (instructions are similar to masking subsystem or model blocks).
- The *MATLAB External Interfaces* document explains how to write C and other programs that interact with the MATLAB environment through the MEX API. The Simulink S-function API is built on top of this API. To pass parameters to your device driver block from the MATLAB and/or Simulink environments, you must use the MEX API. The *MATLAB C and Fortran API Reference* contains reference descriptions for the required MATLAB `mx*` routines.
- The *Real-Time Workshop Target Language Compiler* document describes how to customize code generation for blocks and targets. Knowledge of the Target Language Compiler is required in order to inline S-functions. The *Real-Time Workshop Target Language Compiler* document also describes the structure of the *`model`*`.rtw` file.

Tradeoffs in Device Driver Development

Hand Coding vs. S-Function Builder

Part of the task of device driver creation is to create a C MEX-file, primarily for use in simulation. Traditionally, C MEX-files are written manually, often using S-function template provided by the Real-Time Workshop product as a starting point. Most of this chapter is concerned with manually written device driver code.

If you have little experience in writing S-functions, you can simplify the process of implementing your C MEX-file by using the Simulink S-Function Builder. This alternative is described in "Creating Device Drivers with the S-Function Builder" on page 10-36.

Note that use of the S-Function Builder does not completely eliminate the need to write code. You must still write TLC code to generate inlined code from your driver. Furthermore, the S-Function Builder only imports a subset of the S-Function API. Consequently, it may be necessary to modify the C MEX-files created by the S-Function Builder.

Inlined vs. Noninlined Drivers

You can use inlined or non-inlined S-functions with the Real-Time Workshop Embedded Coder software. A benefit of non-inlined S-functions is that you do not have to write TLC code. However, for embedded systems development, fully inlined device drivers have numerous advantages. Inlined device drivers are an appropriate design choice when:

- You need production code generated from the S-function to behave differently than code used during simulation. This is almost always the case when developing device drivers. For example, an output device block may write to a hard device address in generated code, but during simulation, this address may be illegal. The driver should therefore perform no output during simulation.

  This dual behavior can be achieved in a noninlined S-function, but only by use of awkward compiler conditionals.

- You want to avoid overhead associated with calling the S-function API.
- You want to avoid writing stub routines (to satisfy the S-function API) that have no purpose in your generated code.
- You want to reduce memory usage. Note that each noninlined S-function creates its own `Simstruct` structure. Each `Simstruct` structure uses over 1K of memory. Inlined S-functions do not allocate any `Simstruct` structures.
- You want to take advantage of the `mdlRTW` function. Implementing a `mdlRTW` function gives you maximum flexibility in communicating parameter data from the model to the *`model`*`.rtw` file during code generation. The `mdlRTW` mechanism is only available to inlined S-functions.

In device driver development, achieving minimal memory usage and maximum code performance are usually the most important considerations. From this standpoint, there are no compelling reasons for creating noninlined drivers.

Example Device Driver

The Real-Time Workshop Embedded Coder product provides an example of a manually written and fully inlined input device driver, `ADC_examp`, to accompany the discussions in later sections. This driver supports the analog-to-digital converter (ADC) device on the Freescale HC12 microcontroller. A complete driver implementation is available in the directory

```
matlabroot/toolbox/rtw/targets/common/examples/ADC_driver_example
```

The driver files include

- `ADC_examp.c`: Source code for simulation driver S-function
- `ADC_examp.mexext`: C MEX-file built from `ADC_examp.c` (*`mexext`* is a platform-dependent file extension, such as `.mexw32` on 32-bit Microsoft Windows platforms)
- `ADC_examp.tlc`: TLC implementation for inlined code generation
- `ADC_library.mdl`: Simulink library containing masked S-function driver block for use in simulation
- `ADC_examp_model.mdl`: Simple example model that uses the block. This model is configured for ERT code generation only.

If you have a host-target development environment for targeting the HC12 embedded processor, including the required compiler and development boards, you can use the `ADC_examp` driver in simulation and to generate, download, and run an executable with inlined driver code.

Otherwise, you can select the ERT target and use the `ADC_examp` driver in simulation and to generate code.

Writing a Device Driver C MEX S-Function

In this section...

Overview

This discussion assumes that you are implementing a driver as a fully inlined S-function. For use in simulation, you must provide a C MEX S-function. Since this S-function is used only in simulation, it is relatively simple to implement. The S-function may contain functions that:

- Initialize the `SimStruct` structure.
- Display information in the MATLAB Command Window during simulation.
- Validate block parameter data input by the user.
- Implement a `mdlRTW` function for passing data to the *`model`*`.rtw` file.

You should use the Real-Time Workshop S-function template as a starting point for developing your simulation driver S-function. The template file is

```
matlabroot/simulink/src/sfuntmpl_basic.c
```

An extensively commented version of the S-function template is also available. See *`matlabroot`*`/simulink/src/sfuntmpl_doc.c`.

Alternatively, you can use the `ADC_examp` driver (see "Example Device Driver" on page 10-5) as a starting point for your driver.

Your S-function must implement certain specific functions required by the S-function API. These are described in "Functions Required by the S-Function API" on page 10-8. Since these functions are private to the source file, you can incorporate multiple instances of the same S-function into a model.

Note Device driver S-functions used in simulation should not contain code that is intended to operate in real time on the target hardware, or that accesses actual target hardware addresses. Since your target I/O hardware is not present during simulation, writing to addresses in the target environment can result in illegal memory references, overwriting system memory, and other severe errors. Similarly, read operations from nonexistent hardware registers can cause model execution errors.

Required Defines and Include Files

Your driver S-function must begin with the following three statements, in the following order:

1. `#define S_FUNCTION_NAME` *`name`*

   This defines the name of the entry point for the S-function code. *`name`* must be the name of the S-function source file, without the file extension. For example, if the S-function source file is `example_hc12_sfcn_adc_v.c`:

   ```
   #define S_FUNCTION_NAME example_hc12_sfcn_adc_v
   ```

2. `#define S_FUNCTION_LEVEL 2`

   This statement defines the file as a level 2 S-function. This allows you to take advantage of the full feature set included with S-functions. Level-1 S-functions are currently used only to maintain backwards compatibility.

3. `#include "simstruc.h"`

   The file `simstruc.h` defines the `SimStruct` (Simulink data structure) and associated accessor macros. It also defines access methods for the `mx*` functions from the MATLAB MEX API.

The final statement in your S-function is equally critical. Assuming that your S-function contains only simulation code, your code *must* end with the following.

```
#include "simulink.c"
```

`simulink.c` provides required functions interfacing to the Simulink engine.

Other Preprocessor Symbols

Real-Time Workshop software defines several preprocessor symbols that affect how S-functions are built. The conventions for use of these symbols are as follows:

- `MATLAB_MEX_FILE`

  When you build your S-function as a MEX-file with the `mex` command, `MATLAB_MEX_FILE` is automatically defined.

  A test on `MATLAB_MEX_FILE`, such as the following, is useful in drivers that contain only simulation code intended for use in an S-function. This test ensures that the driver S-function is compiled only as a C MEX-file.

  ```
  #ifndef MATLAB_MEX_FILE
  #error "Fatal Error: ADC_examp.c can only be used to create C-MEX S-Function"
  #endif
  ```

- `MDL_START`

  The model execution loop calls `mdlStart` only if the symbol `MDL_START` is declared with a `#define` statement. If you write a `mdlStart` function without defining `MDL_START`, an "unreferenced function" compile-time warning occurs when you build your S-function, and the `mdlStart` code is never be called during simulation. See "mdlStart" on page 10-16 for an example.

Functions Required by the S-Function API

The S-function API requires you to implement several functions in your simulation driver:

- `mdlInitializeSizes`: This function specifies the sizes of various parameters in the `SimStruct` structure, such as the number of output ports for the block.
- `mdlInitializeSampleTimes`: This function specifies the sample time(s) of the block.

  If your device driver block is masked, your initialization functions can obtain the sample time and other parameters entered by the user in the block's dialog box.

- `mdlOutputs`: For an input device, this function usually outputs a nominal value (such as zero) on all channels during simulation. Another approach is to replicate the block's inputs at the outputs. For an output device, `mdlOutputs` can be implemented as a stub.
- `mdlTerminate`: This function can be implemented as a stub.

In addition to the above, you may want to implement the `mdlStart` function. `mdlStart` is called once at the start of model execution.

This following sections provide guidelines for implementing these functions. Code examples are taken from the example input device driver, `ADC_examp`, which is available in the directory

```
matlabroot/toolbox/rtw/targets/common/examples/ADC_driver_example
```

Macro and Symbol Definitions for ADC_examp.c

`ADC_examp.c` defines the following symbols and macros, referenced throughout the code examples below. Note how the example optimizes storage space by using an `enum` statement to define a set of masks that correspond to bit positions in a single word representing the device data.

```
#define TRUE    1
#define FALSE   0

/* Total number of block parameters */
#define N_PAR   5

/*
 *  CHANNELARRAY_ARG  - Array of ADC channels (one or more values between 0 and 7)
 *                      Signal width is also determined from this list
 *  SAMPLETIME(S)     - Sample time
 %  ATDBANK(S)        - Bank 0, or Bank 1. Each bank provides 8 channels.
 *  USE10BITS(S)      - If (USE10BITS_ARGC==1), use 10-bits of ADC resolution
 *                      otherwise, use 8-bits ADC resolution
 *  LEFTJUSTIFY(S)    - If (LEFTJUSTIFY_ARGC==1), left justify the result in
 *                      16-bit word. Else, use right justification (default)
 */

/* Define a set of masks that correspond to bit positions in a single word
```

```
 * representing device data.
 */

enum {ATDBANK_ARGC=0, CHANNELARRAY_ARGC, USE10BITS_ARGC, LEFTJUSTIFY_ARGC,
SAMPLETIME_ARGC};

#define ATDBANK(S)          (mxGetScalar(ssGetSFcnParam(S,ATDBANK_ARGC)))
#define CHANNELARRAY_ARG(S) (ssGetSFcnParam(S,CHANNELARRAY_ARGC))
#define USE10BITS(S)        (mxGetScalar(ssGetSFcnParam(S,USE10BITS_ARGC)))
#define LEFTJUSTIFY(S)      (mxGetScalar(ssGetSFcnParam(S,LEFTJUSTIFY_ARGC)))
#define SAMPLETIME(S)       (mxGetScalar(ssGetSFcnParam(S,SAMPLETIME_ARGC)))
```

mdlInitializeSizes

The `mdlInitializeSizes` function specifies the sizes of various parameters in the `SimStruct` structure. In the example below, this information partially depends upon the parameters passed to the S-function. See "Creating a User Interface for Your Driver" on page 10-18 for information on how to access parameter values specified in S-function dialog boxes.

The `mdlInitializeSizes` function for the `ADC_examp` example input device driver is listed below.

```
static void mdlInitializeSizes(SimStruct *S)
{
    const unsigned int *paramPtr = mxGetData( CHANNELARRAY_ARG(S) );
    int nChannels;

    /* Set and Check parameter count  */

    ssSetNumSFcnParams(S, N_PAR);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) return;
    ssSetSFcnParamTunable(S, 0, SS_PRM_NOT_TUNABLE);
    ssSetSFcnParamTunable(S, 1, SS_PRM_NOT_TUNABLE);
    ssSetSFcnParamTunable(S, 2, SS_PRM_NOT_TUNABLE);
    ssSetSFcnParamTunable(S, 3, SS_PRM_NOT_TUNABLE);
    ssSetSFcnParamTunable(S, 4, SS_PRM_NOT_TUNABLE);

    nChannels  = mxGetNumberOfElements( CHANNELARRAY_ARG(S) );
```

```
/* Single input port of width equal to nChannels */
if ( !ssSetNumInputPorts(  S, 1 ) ) return;
ssSetInputPortWidth(       S, 0, nChannels );

/* Single output port of width equal to nChannels */
if ( !ssSetNumOutputPorts( S, 1 ) ) return;
ssSetOutputPortWidth(      S, 0, nChannels );

/* Set datatypes on input and output ports relative
 * to users choice of 8-, or, 10-bit resolution.
 */
if (USE10BITS(S))
{
   /*
    * Input and output datatypes are uint16
    * when using 10-bit ADC resolution
    */
    ssSetInputPortDataType(  S, 0, SS_UINT16 );
    ssSetOutputPortDataType( S, 0, SS_UINT16 );
} else {
   /*
    * Input and output datatypes are uint8
    * when using 8-bit ADC resolution
    */
    ssSetInputPortDataType(  S, 0, SS_UINT8 );
    ssSetOutputPortDataType( S, 0, SS_UINT8 );
}

ssSetInputPortDirectFeedThrough( S, 0, TRUE );

/* sample times */
ssSetNumSampleTimes( S, 1 );

/* options */
ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
} /* end mdlInitializeSizes */
```

The above `mdlInitializeSizes` function does the following, in order:

- Validates that the number of input parameters is equal to the expected number of parameters in the block's dialog box (`N_PARS`).
- Sets the parameters to be nontunable. This improves efficiency and provides error handling in the event that an attempt is made to change the parameter.
- Obtains `nChannels`, the number of ADC channels (specified as a vector in the **Channels** parameter of the block dialog box). The widths of the input and output ports are set equal to `nChannels`. Notice that the code ensures that the block has exactly one input port and one output port.
- Obtains the user-selected resolution value (returned by `USE10BITS`) and sets the port data types for the block.
- Sets the direct feedthrough property of the block to `TRUE`. (In simulation, the `ADC_examp` output is replicated from the block input you would normally connect the `ADC_examp` input to a Ground.)

  Note that in many cases, input driver blocks do not have input ports. (Input ports can be used, however, to provide pass-through capability to a driver during simulation. See "Device Drivers in Simulation" on page 10-48 for further information.) If your input driver block has no input ports, set the number of input ports to `0`.

  ```
  ssSetNumInputPorts(S, 0);
  ```

- Calls `ssSetNumSampleTimes` to set the number of sample times to 1. This is correct for a driver where all ADC channels run at the same rate. Note that the actual sample period for the block is set in `mdlInitializeSampleTimes`.
- Specifies the S-function option `SS_OPTION_EXCEPTION_FREE_CODE`. This option declares that the block does not throw exceptions. Use this option with care. See "Exception Free Code" in the Simulink Writing S-Functions document.

mdlInitializeSizes for Output Drivers

Initializing size information for an output device, such as a DAC, differs in important ways from initializing sizes for an ADC:

- A DAC is a sink block. That is, it has input ports but typically has no output ports. (Output ports can be used, however, to provide pass-through capability to a driver during simulation. See "Device Drivers in Simulation"

on page 10-48 for further information.) If your output driver block has no output ports, set the number of output ports to `0`.

```
ssSetNumOutputPorts(S, 0);
```

- Since a DAC obtains its inputs from other blocks, the number of channels is equal to the number of inputs.
- A DAC block has direct feedthrough. The DAC block cannot execute until the block feeding it updates its outputs.

mdlInitializeSampleTimes

Device driver blocks are discrete blocks, requiring you to set a sample time. The procedure for setting sample times is the same for both input and output device drivers. Assuming that all channels of the device run at the same rate, the S-function has only one sample time.

The following implementation of `mdlInitializeSampleTimes` (from `ADC_examp`) obtains the sample time from the block's dialog box. The sample time offset is set to `0`.

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime( S, 0, SAMPLETIME(S) );

} /* end mdlInitializeSampleTimes */
```

mdlOutputs

All S-functions implement a `mdlOutputs` function to calculate block outputs. For many simulation drivers, this is a simple task. In the simplest case, the `mdlOutputs` function for an input simulation driver generates a nominal value (usually 0), on all channels. The following code fragment, from a hypothetical simulation driver for an ADC with a fixed number of channels, illustrates this approach.

```
for (i = 0; i < NUM_CHANNELS; i++){
   y[i] = 0.0;
}
```

An output simulation driver, which is a sink, can often be implemented as a stub.

The `ADC_examp` driver implements a more complex `mdlOutputs` function, listed below.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    /*
     * Get "uPtrs" for input port 0 and 1.
     * uPtrs is essentially a vector of pointers because the input signal may
     * not be contiguous.
     */

    DTypeId   yODataType;   /* SS_UINT8 or SS_UINT16 */
    int_T     yOWidth    = ssGetOutputPortWidth(S, 0);
    InputPtrsType uOPtrs = ssGetInputPortSignalPtrs(S,0);

    /*
     * Get data type Identifier for output port 0.
     * This matches the data type ID for input port 0.
     */

    yODataType = ssGetOutputPortDataType(S, 0);

    /*
     * Set output signals equal to input signals
     * for either 16 bit, or 8 bit signals.
     */

    switch (yODataType)
    {
        case SS_UINT8:
        {
            uint8_T           *pY0 = (uint8_T *)ssGetOutputPortSignal(S,0);
            InputUInt8PtrsType pU0 = (InputUInt8PtrsType)uOPtrs;
            int     i;
            /* Set all outputs equal to inputs */
            for( i = 0; i < yOWidth; ++i){
                pY0[i] = *pU0[i];
```

```
                /* For 8-bit ADC results, left-justify is ignored. */
            }
            break;
        }
        case SS_UINT16:
        {
            uint16_T           *pY0 = (uint16_T *)ssGetOutputPortSignal(S,0);
            InputUInt16PtrsType pU0 = (InputUInt16PtrsType)u0Ptrs;
            int     i;

            for( i = 0; i < y0Width; ++i){
                /* Set all outputs equal to inputs */
                if (LEFTJUSTIFY(S)) {
                    /* Shift left for left justify */
                    pY0[i] = *pU0[i]<<6;
                } else {
                    /* No shift required for right justify */
                    pY0[i] = *pU0[i];
                }
            }
            break;
        }
    } /* end switch (y0DataType) */

} /* end mdlOutputs */
```

This `mdlOutputs` function is designed to handle the following requirements:

- Rather than simply generating zeroes, the block passes through an input signal for use in simulation by simply setting outputs equal to inputs.
- I/O ports are variably typed to be either `uint8` or `uint16`, depending on the user's choice of **ADC resolution**. The port data type is obtained with the call

  ```
  y0DataType = ssGetOutputPortDataType(S, 0);
  ```

  A `switch(y0DataType)` statement then determines how the input signal is passed to the output. In the 16-bit case, the data may be right-shifted (justified).

- I/O port widths are variable, in accordance with the number of ADC channels (specified as a vector in the **Channels** parameter of the block dialog box). The port width is obtained with the call

  ```
  int_T yOWidth = ssGetOutputPortWidth(S, 0);
  ```

  `yOWidth` is then used to control iteration over the I/O signals:

  ```
  for( i = 0; i < yOWidth; ++i){
     pY0[i] = *pU0[i];
  }
  ```

mdlTerminate

In `ADC_examp`, the `mdlTerminate` function is provided as a stub, to satisfy the requirements of the S-function API.

```
static void mdlTerminate(SimStruct *S)
{
} /* end mdlTerminate */
```

mdlStart

`mdlStart` is an optional function. It is called once at the start of model execution. In `ADC_examp`, `mdlStart` simply displays a message in the MATLAB Command Window:

```
#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
static void mdlStart(SimStruct *S)
{
/* During simulation, just print a message */
  if (ssGetSimMode(S) == SS_SIMMODE_NORMAL) {
    mexPrintf("\n ADC_examp driver: Simulating initialization\n");
    }
}
#endif /*  MDL_START */
```

Note The model execution loop calls `mdlStart` only if the symbol `MDL_START` is declared as shown above. If you write a `mdlStart` function without defining `MDL_START`, an "unreferenced function" compile-time warning occurs when you build your S-function, and the `mdlStart` code is never called during simulation.

Creating a User Interface for Your Driver

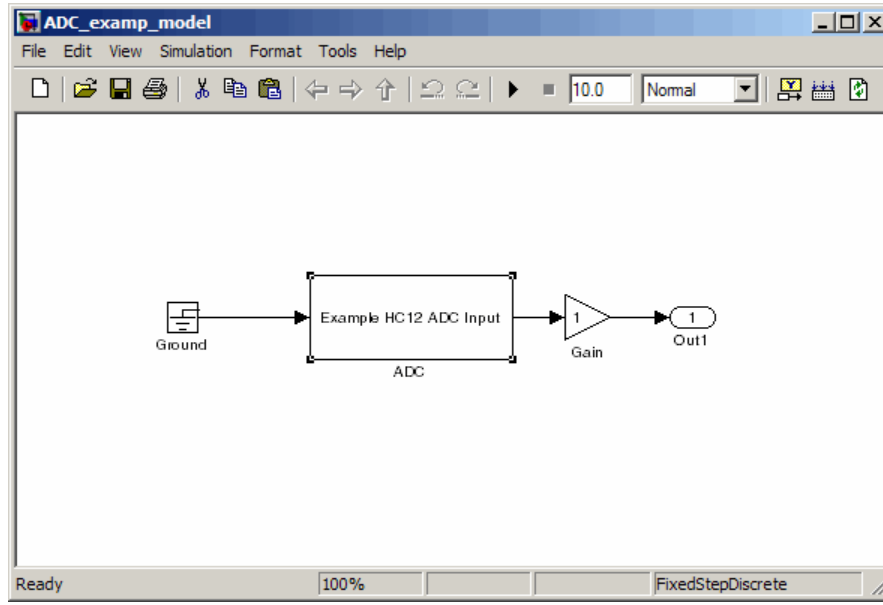
In this section...

Using a Masked Device Driver Block

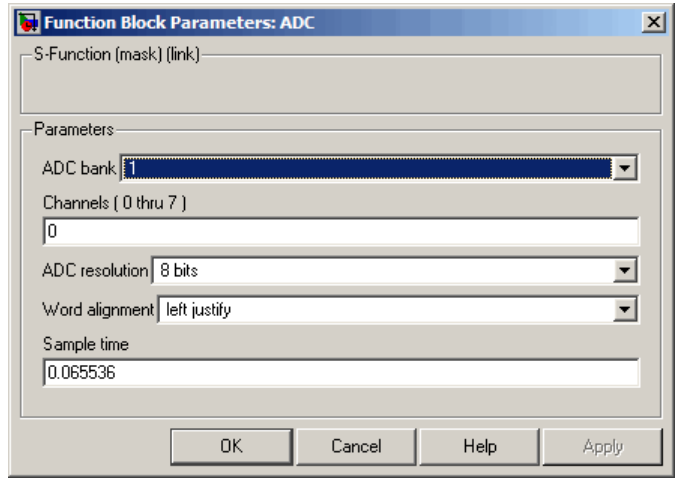
You can add a custom icon, dialog box, and initialization commands to an S-Function block by masking it. This provides an easy-to-use graphical user interface for your device driver in the Simulink environment.

This section uses examples drawn from an actual masked device driver block. You should have basic familiarity with the creation and use of masked blocks. These topics are discussed in the Using Simulink and Simulink Writing S-Functions documents.

The example driver, `ADC_examp`, is an input device driver. To follow the examples in this section, launch `ADC_examp_model.mdl` from *`matlabroot`*`/toolbox/rtw/targets/common/examples/ADC_driver_example`.



`ADC_examp` illustrates a number of techniques for parameterizing a driver by letting the user enter hardware-related variables. Dialog Box for ADC_examp Driver Block on page 10-20 shows the dialog box that `ADC_examp` presents to the user. To launch the dialog box, right-click the ADC S-Function block in the model and select **Mask Parameters**. Parameter values are shown at their default values.

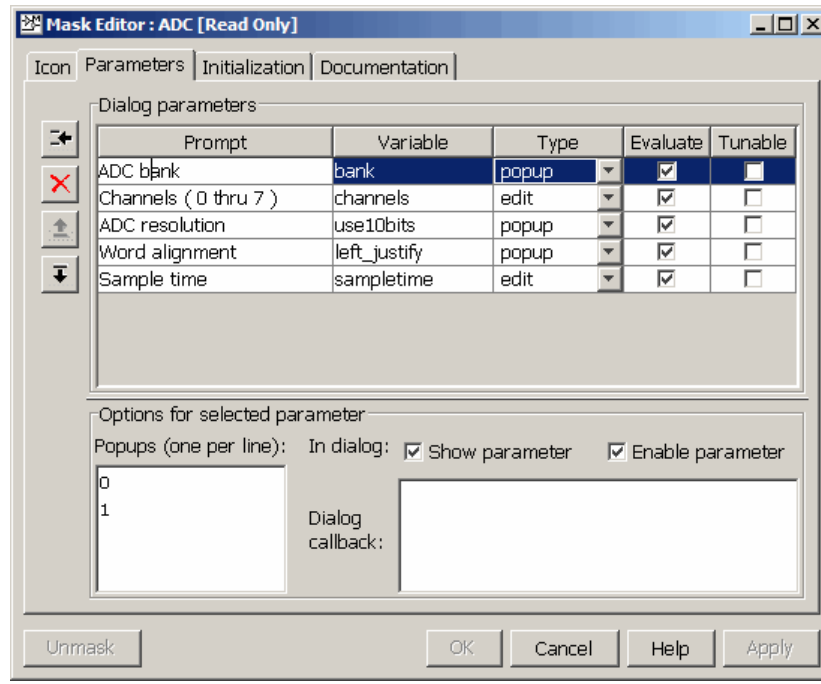


Dialog Box for ADC_examp Driver Block

The Simulink user can enter the following parameters:

- **ADC bank** (menu): Selects one of two 8-channel ADC banks (either bank `0` or `1`).
- **Channels** (edit field): Specifies input channel(s) to be read. Channels are numbered in the range `0-7`. Selected channels are represented as a vector.
- **ADC resolution** (menu): Selects either 8 bits or 10 bits of resolution. If 10 bit resolution is selected, the input signal data is stored in 16 bits.
- **Word alignment** (menu): If **ADC resolution** is set to 10 bits, the user can select either right or left justification of input data within a 16-bit word. If **ADC resolution** is set to 8 bits, input data is stored as a `uint8`, and **Word alignment** is ignored.
- **Sample time** (edit field): Specifies sample time for the block.

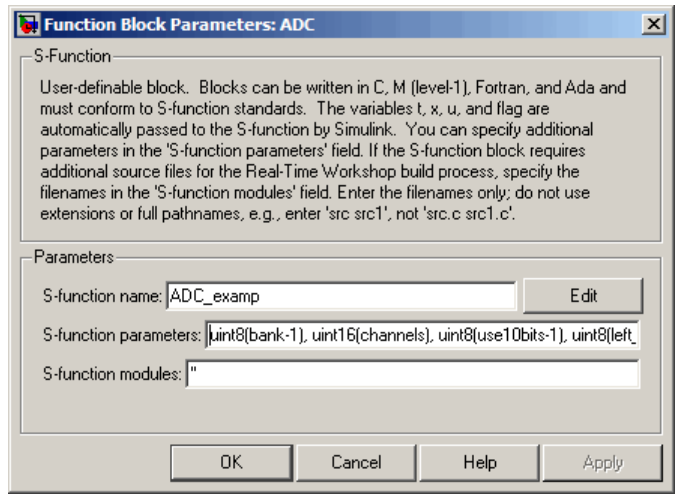
You specify block parameters in the **Parameters** pane of the Simulink Mask Editor. To launch the Mask Editor, right-click the `ADC_examp` S-Function block and select **View Mask**. You can then select the **Parameters** pane.



The preceding figure shows how the parameter section of the mask is defined for the `ADC_examp` driver. In the `ADC_examp` driver, block parameters are declared nontunable in the block mask. If you do not do this, you can declare parameters nontunable by using the `ssSetSFcnParamNotTunable` macro in the `mdlInitializeSizes` routine. Nontunable S-function parameters become constants in the generated code, improving performance.

In certain cases, you may want your driver block to be self-modifying. For example, the block may have a parameter that lets the user set the number of input or output ports on the block. In such cases, you should select the **Allow library block to modify its contents** option in the **Initialization** pane of the Mask Editor (see "Simulink Mask Editor" in the Simulink documentation).

To view the block parameters underlying the mask, right-click the ADC S-Function block and select **Look Under Mask**.



The block parameters underlying the mask, as shown in the preceding figure, provide a binding to the C MEX S-function for use in simulation, and a list of parameter variables corresponding to the **S-function parameters** field. Note that:

- Values returned from menus are offset by `-1` (because menus are 1-based).
- Parameter variables, except `sampletime`, are explicitly cast to unsigned integer data types. The **S-function parameters** field contains the following list of expressions.

  ```
  uint8(bank-1), uint16(channels), uint8(use10bits-1), uint8(left_justify-1),
  sampletime
  ```

  During the build process, parameter expressions are evaluated and the resultant values are written to `Parameter` records in the *`model`*`.rtw` file. These records are used when code is generated by the TLC implementation of the block (see “Inlining the S-Function Device Driver” on page 10-27).

It is typical for a device driver block to read and validate input parameters in its `mdlInitializeSizes` function. A masked S-Function block obtains parameter data from its dialog box using macros and functions provided for the purpose. Let us examine some cases from the `mdlInitializeSizes` function of `ADC_examp.c`.

Obtaining and Using a Scalar Parameter

In the following code excerpt from `ADC_examp.c`, the macro `USE10BITS` is defined. When invoked, `USE10BITS` returns the value obtained from the **ADC resolution** menu.

```
enum {ATDBANK_ARGC=0, CHANNELARRAY_ARGC, USE10BITS_ARGC, LEFTJUSTIFY_ARGC,
SAMPLETIME_ARGC};
...
#define USE10BITS(S)        (mxGetScalar(ssGetSFcnParam(S,USE10BITS_ARGC)))
...
/* Set datatypes on input and output ports relative
    * to users choice of 8-, or, 10-bit resolution.
    */
    if (USE10BITS(S))
    {
       /*
        * Input and output datatypes are uint16
        * when using 10-bit ADC resolution
        */
        ssSetInputPortDataType(  S, 0, SS_UINT16 );
        ssSetOutputPortDataType( S, 0, SS_UINT16 );
    } else {
       /*
        * Input and output datatypes are uint8
        * when using 8-bit ADC resolution
        */
        ssSetInputPortDataType(  S, 0, SS_UINT8 );
        ssSetOutputPortDataType( S, 0, SS_UINT8 );
    }
```

The parameter from the dialog box is accessed with the `ssGetSFcnParam` macro. The arguments to `ssGetSFcnParam` are a pointer to the block's `Simstruct` structure, and the index (0-based) to the desired parameter.

Parameters are stored in arrays of type `mxArray`, even if there is only a single value. In the above code, the value of the first element of the `mxArray` returned by `ssGetSFcnParam` is obtained with the `mxGetScalar` function.

The value returned by `USE10BITS` is used to set the port data types for the block, in accordance with the user-selected resolution. The larger (`uint16`) data type is used only when necessary.

Obtaining and Using a Vector Parameter

This section shows another code excerpt from `ADC_examp.c` that illustrates the use of a vector parameter. You enter the **Channels** parameter as a vector of channels in the range 0..7. The macro `CHANNELARRAY_ARG` returns this vector, and the `mxGetNumberOfElements` function is called to obtain the number of vector elements. The port widths for the block are set accordingly.

```
enum {ATDBANK_ARGC=0, CHANNELARRAY_ARGC, USE10BITS_ARGC, LEFTJUSTIFY_ARGC,
SAMPLETIME_ARGC};
...
#define CHANNELARRAY_ARG(S) (ssGetSFcnParam(S,CHANNELARRAY_ARGC))
...
nChannels  = mxGetNumberOfElements( CHANNELARRAY_ARG(S) );

    /* Single input port of width equal to nChannels */
    if ( !ssSetNumInputPorts(  S, 1 ) ) return;
    ssSetInputPortWidth(       S, 0, nChannels );

    /* Single output port of width equal to nChannels */
    if ( !ssSetNumOutputPorts( S, 1 ) ) return;
    ssSetOutputPortWidth(      S, 0, nChannels );
```

The MathWorks recommends that you study the entire `mdlInitializeSizes` function of `ADC_examp.c` for further examples of the use of masked block parameters in the context of a device driver.

Creating the Device Driver Block

In this section...

Building the MEX-File and the Driver Block

This section outlines how to build a MEX-file from your driver source code for use in a Simulink model. For full details on how to use `mex` to compile an executable MEX-file, see the *MATLAB External Interfaces* document.

1. Your C S-function source code should be in your working directory. To build a MEX-file from `mydriver.c`, type

   ```
   mex mydriver.c
   ```

   `mex` builds `mydriver.`*`mexext`*, where *`mexext`* is a platform-dependent file extension such as `mexw32` (32-bit Microsoft Windows platforms).

2. Add an S-Function block (from the Simulink User-Defined Functions library in the Library Browser) to your model.

3. Double-click the S-Function block to open the Block Parameters dialog box. Enter the S-function name `mydriver`. The block is now bound to the `mydriver` MEX-file.

4. Create a mask for the block if you want to use a custom icon or dialog box (see "Creating a User Interface for Your Driver" on page 10-18).

5. You should create a block library and add your driver to it, or add your driver to an existing block library. See "Working with Block Libraries" in the Using Simulink document to learn how to do this.

Making Your Driver Available to Simulink Users

Your driver implementation files should be stored in a directory that is on the MATLAB path. You should create a `blocks` directory under your target root directory (for example, `mytarget/blocks`). The `blocks` directory should contain

- Compiled block MEX-files
- C source code for the blocks
- TLC inlining files for the blocks
- Library models for the blocks. You should place your blocks in one or more libraries.

Inlining the S-Function Device Driver

In this section...

Code Components

To create a fully inlined device driver, you must provide the following components:

- *driver*.c: C MEX S-function source code, implementing the functions required by the S-function API for a simulation driver. (See "Writing a Device Driver C MEX S-Function" on page 10-6.) For these functions, only the code for simulation by the Simulink engine is required.

  Optionally, *driver*.c may implement a mdlRTW function. The sole purpose of this function is to evaluate and format parameter data during code generation. The parameter data is output to the *model*.rtw file. See "Passing and Obtaining Block Parameter Values with mdlRTW" on page 10-30.

  It is important to ensure that *driver*.c does not attempt to read or write memory locations that are intended to be used in the target hardware environment. The real-time driver implementation, generated with a *driver*.tlc file, should access the target hardware.

- *driver.mexext* : MEX-file built from your C MEX S-function source code. The filename extension *mexext* varies depending on the platform. For example, on 32-bit Microsoft Windows platforms, the extension is .mexw32.

  This component is used:

  - In simulation: The Simulink engine calls the simulation versions of the required functions
  - During code generation: If a mdlRTW function exists in the MEX-file, the code generator executes it to write parameter data to the *model*.rtw file.

- *driver*.tlc: TLC functions that generate real-time implementations of the functions required by the S-function API.
- Hardware support files: Header files, macro definitions, or code libraries that may be provided with your I/O devices or cross-development system. It may be necessary to generate `#include` statements or other directives required for using such support files. See "Generating Target-Specific Compiler Directives" on page 10-29 for information on how to generate these directives.

Inlined Device Driver Operations

Typical operations performed by an inlined device driver include

- Initializing the I/O device. For example, the driver may need to write specific values to one or more control registers to set the device into a desired mode of operation.
- Calculating the block outputs. How this is done depends upon the type of driver being implemented:
  - An input driver for a device such as an ADC usually reads values from an I/O device and assigns these values to the block's output vector `y`.
  - An output driver for a device such as a DAC usually writes values from the block's input vector `u` to an I/O device.
- Terminating the program. This may require setting hardware to a "neutral" state; for example, zeroing DAC outputs.

In generated code, these operations are usually executed within the standard model functions, such as *model*_initialize, *model*_step, and *model*_terminate.

Inlining the Example ADC Driver

As an aid to understanding the process of inlining a device driver, this section describes the TLC implementation of the `ADC_examp` driver. Full TLC source code for `ADC_examp.tlc` is provided in the directory

matlabroot/toolbox/rtw/targets/common/examples/ADC_driver_example

The TLC implementation of the `ADC_examp` driver is somewhat simpler than the simulation code. It contains two TLC functions:

- The `Start` function generates code that is inlined into the *`model`*`_initialize` function. The code initializes several control registers of the Freescale HC12 ADC device.
- The `Outputs` function generates code that is inlined into the *`model`*`_step` function. The code reads data from one or more ADC channels. The data is assigned to the block outputs.

Generating Target-Specific Compiler Directives

Device driver code often references target-specific symbols that are defined externally to the generated code. These symbols represent specific hardware registers, memory addresses, or operating system functions. For example, the `Start` and `Outputs` functions in the TLC implementation of the `ADC_examp` driver generate code to read and write various HC12 ADC registers. These are typically defined in header files provided by the vendor of the target hardware or the cross-development system that compiles the generated code.

Such references are resolved by generating compiler directives (such as `#include` or `#define` statements). These directives can be generated:

- By the device driver block itself. This is often done in a `BlockTypeSetup` function in the driver TLC implementation.
- By a "master" device driver block. Some targets (such as the Target Support Package product) implement a master block that manages hardware resources for multiple drivers. Such targets require inclusion of the master block in the model. Accordingly, the `BlockTypeSetup` function for the master block can generate the includes required by all the other blocks.

To generate `#include` or `#define` statements in *`model`*`.h` (or *`model`*`_private.h`), you can invoke the following functions inside a `BlockTypeSetup` function:

- `LibAddToCommonIncludes(incFileName)`
- `LibCacheDefine(buffer)`

For an example of customizing a `BlockTypeSetup` function, see "Customizing Driver Code Generation" on page 10-41. For more information, see the following sections in the Real-Time Workshop Target Language Compiler documentation:

- In the "Inlining S-Functions" chapter, "Block Target File Methods" provides a Block Functions Overview and describes the `BlockTypeSetup(block, system)` function.
- In the "TLC Function Library Reference" chapter, "Code Configuration Functions" describes the `LibAddToCommonIncludes` and `LibCacheDefine` functions.

Passing and Obtaining Block Parameter Values with mdlRTW

The driver S-function (`ADC_examp.c`) implements a `mdlRTW` function to pass user-entered parameter values (**ADC bank**, **Channels**, **ADC resolution**, and **Word alignment**) to the *`model`*`.rtw` file.

The `mdlRTW` function is a mechanism by which a C MEX S-function can generate and write data structures to the *`model`*`.rtw` file. The Target Language Compiler, in turn, uses these data structures when generating code. The simplest application of `mdlRTW` is to pass block parameter data into the *`model`*`.rtw` file. However, `mdlRTW` also lets you compute virtually any useful data and pass it into the *`model`*`.rtw` file.

Unlike the other functions in a simulation driver, `mdlRTW` executes at code generation time. The `mdlRTW` mechanism is fully described in the "Integrating External Code With Generated C and C++ Code" chapter of the Real-Time Workshop documentation. This section shows the use of `mdlRTW` in the `ADC_examp` device driver.

The `mdlRTW` function in `ADC_examp.c` obtains user-entered parameter values using the symbol and macro definitions described in "Macro and Symbol Definitions for ADC_examp.c" on page 10-9. It then generates a structure that contains these values in the *`model`*`.rtw` file. Macros (such as `SSWRITE_VALUE_DTYPE_NUM`) are defined for this purpose. These macros are described in the Simulink Writing S-Functions document.

The `mdlRTW` function from `ADC_examp.c` is listed below.

```
static void mdlRTW(SimStruct *S)
{
    uint8_T   atdbank     = (uint8_T) ATDBANK(S);
    uint16_T *channels    = (uint16_T *) mxGetData(CHANNELARRAY_ARG(S));
    uint8_T   use10BitRes = (uint8_T) USE10BITS(S);
    uint8_T   leftjustify = (uint8_T) LEFTJUSTIFY(S);

    /* Write out parameters for this block.*/
    if (!ssWriteRTWParamSettings(S, 4,
                                 SSWRITE_VALUE_DTYPE_NUM,"ATDBank",
                                 &atdbank,DTINFO(SS_UINT8, COMPLEX_NO),

                                 SSWRITE_VALUE_DTYPE_VECT, "Channels",
                                 channels,
                                 mxGetNumberOfElements(CHANNELARRAY_ARG(S)),
                                 DTINFO(SS_UINT16, COMPLEX_NO),

                                 SSWRITE_VALUE_DTYPE_NUM,"Use10BitRes",
                                 &use10BitRes,DTINFO(SS_UINT8, COMPLEX_NO),

                                 SSWRITE_VALUE_DTYPE_NUM,"LeftJustify",
                                 &leftjustify,DTINFO(SS_UINT8, COMPLEX_NO)
                                 )) {
        return; /* An error occurred which will be reported by SL */
    }
}
```

A typical *model*`.rtw` structure generated by this `mdlRTW` function is

```
SFcnParamSettings {
ATDBank    1U
Channels    [0U]
Use10BitRes    0U
LeftJustify    1U
}
```

The field values of `SFcnParamSettings` derive from data that you enter.

Values stored in the `SFcnParamSettings` structure are referenced in the TLC block implementation, as in the following code excerpt:

```
%assign Use10BitResolution = CAST("Number",SFcnParamSettings.Use10BitRes)
%assign LeftJustify        = CAST("Number",SFcnParamSettings.LeftJustify)
```

See "Start Function" on page 10-32, and the `ADC_examp.tlc` code, for further examples of how the `SFcnParamSettings` structure is used to generate code for the driver block.

Note During code generation, the Real-Time Workshop build process writes run-time parameters automatically to the *`model`*`.rtw` file, eliminating the need for an S-function to perform this task with a `mdlRTW` method. However, these run-time parameters are always tunable. Generally, it is not appropriate for device driver parameters to be tunable. To control tunability, you must use the more lengthy approach of using the S-function parameter settings for device drivers. See the discussion of run-time parameters in the Simulink Writing S-Functions document for further information.

Start Function

The purpose of the `Start` function in the file `ADC_examp.tlc` is to generate code that initializes several 8-bit control registers of the HC12 ADC device. Each ADC bank (0 or 1) has a separate set of control registers. The bank number is the only variable. Regardless of which bank is selected, the same set of registers is initialized to the same set of bit values.

The symbolic naming convention for these registers is

```
ATDbCTLr
```

where `b` is the user-selected **ADC bank** and `r` is a register number. For example, ATD0CTL1 represents bank 0, control register 1.

The `Start` function obtains the value for `b` from the `SFcnParamSettings` structure (see "Passing and Obtaining Block Parameter Values with mdlRTW" on page 10-30) and uses the returned value in a string substitution, as in the following TLC code excerpt.

```
%assign atdBank = CAST( "Number",SFcnParamSettings.ATDBank)
  ...
ATD%<atdBank>CTL2 = 0x80;
```

For bank 1, this would generate the following statement in the *model*_initialize function.

```
ATD1CTL2 = 0x80;
```

Note also that the Start function generates extensive comments in the code, documenting each register bit setting. A block comment is also generated. You should follow this practice.

Outputs Function

The Outputs function in the file ADC_examp.tlc generates code that repeats the same operations (as inlined code) for all selected ADC channels on the selected ADC bank. For each channel (channelIdx):

- A data conversion is initiated by setting the appropriate channel bits (channelIdx) on ADC control register 5. Bank and channel values derived from the SFcnParamSettings structure are used in string substitutions, as in the following TLC code excerpt.

  ```
  %assign atdBank = CAST( "Number",SFcnParamSettings.ATDBank)
  %%
  %assign nPars = SIZE(SFcnParamSettings.Channels,1)
    ...
  /* Start conversions on selected ADC channels */
  %foreach idx=nPars
      %assign channelIdx = CAST("Number",SFcnParamSettings.Channels[idx])
      ATD%<atdBank>CTL5 = 0x8%<channelIdx>;
    ...
  %endforeach
  ```

  The resulting code for bank 1, channel 0, is

  ```
  /* Start conversions on selected ADC channels */
  ATD1CTL5 = 0x80;
  ```

- The driver continually checks a status register until a conversion completion flag is asserted. The status register symbol is generated by concatenating the current channelIdx and bank parameters, as in the following TLC code excerpt.

  ```
  while (CCF%<channelIdx>_%<atdBank> & 0) {
  ```

```
   /* Wait for Conversion Complete Flag (CCFx)
    * for a conversion on this channel.
    */
}
```

The resulting code for bank 1, channel 0, is

```
while (CCFO_1 & 0) {
    /* Wait for Conversion Complete Flag (CCFx)
     * for a conversion on this channel.
    /*
}
```

- When conversion completes, data is read from a data register for the current bank and channel. The data read from the register is cast to the required data size and left-shifted (justified) if required. The result is assigned to the block output. Again the register symbol is formed by string substitution of the current `channelIdx` and bank parameters, as in the following TLC code excerpt. (Not shown: The index variable `nextChannel` is initialized to 0 and increments by 1 after each channel iteration.)

```
%assign y = LibBlockOutputSignal(0, "", "", %<nextChannel>)
%assign Use10BitResolution = CAST("Number",SFcnParamSettings.Use10BitRes)
%assign LeftJustify        = CAST("Number",SFcnParamSettings.LeftJustify)
%if (%<Use10BitResolution>)
    /* 10-bit resolution */
    %if (%<LeftJustify>)
        /* Left-justified ADC result */
        %<y> = (uint16_T) ATD%<atdBank>DR%<channelIdx> << 6;
    %else
        /* Right-justified ADC result */
        %<y> = (uint16_T) ATD%<atdBank>DR%<channelIdx>;
    %endif
%else
    /* 8-bit resolution */
    %<y> = (uint8_T) ATD%<atdBank>>DR%<channelIdx>;
%endif
```

The code generated for each channel consists of a single line. For example, for the case where 10 bit resolution with left justification is selected for bank 1, channel 0, the following code is generated.

```
/* 10-bit resolution */
/* Left-justified ADC result */
ADC_examp_model_B.ADC_out = (uint16_T) ATD1DR0 << 6;
```

Creating Device Drivers with the S-Function Builder

In this section...

Overview

Traditionally, device drivers used with Simulink software and Real-Time Workshop Embedded Coder software have relied on a dual implementation. For simulation use, you write a device driver block as a Simulink S-function. You also must write a TLC file for inlined code generation purposes.

During simulation, the Simulink engine requires a MEX-file (for example, a `.mexw32` file on 32-bit Microsoft Windows platforms) for an S-function. This MEX-file must provide information such as:

- Number of input signals
- Data types of input signals
- Number of output signals
- Data types of output signals
- Number of parameters for the block
- Data types of parameters

During simulation, the block should provide outputs even if the value is trivial (such as 0 or 1). Assuming the output device is designed so that it has an output signal (in simulation), the appropriate output signal should be provided by the S-function MEX-file.

Defining the correct simulation output for a device driver block is beyond the scope of this discussion. The focus of this discussion is how to create driver blocks for the purpose of generating code with Real-Time Workshop Embedded Coder software.

To create a MEX-file for your S-function, you can

- Write the S-function manually. The Simulink Writing S-Functions document covers this topic.
- Use the Simulink S-Function Builder as a shortcut. If you have little experience in writing S-functions, you should use the S-Function Builder.

This documentation describes the S-Function Builder in sufficient detail for you to get started building device drivers. For a full description of the S-Function Builder, see "Building S-Functions Automatically" in the Simulink documentation.

Currently, the S-Function Builder does not support a device driver mode. Consequently, device driver code resulting from its use may be less optimized than S-function driver code written by hand. Also, since the S-Function Builder supports only a subset of the S-function API, driver code that you produce with the S-Function Builder may lack some desired features.

In the following sections, you create a simple device driver S-function using the S-Function Builder.

Example Device Driver Specification

The driver you will create, `mypwm`, has the following specification:

- The driver supports one channel of pulse width modulation (PWM) output.
- The period of the output signal is fixed.
- The block has one input, which accepts an 8-bit (type `uint8`) modulator signal.
- The duty cycle of the PWM output signal is proportional to the input signal.
- The hardware address of the input port is `0x18h` and is to be symbolically defined in generated code as PORTA.

Building the MEX-File

The first task is to specify the signals and other properties of the driver, and to generate a MEX-file component:

1. Create a new Simulink model.
2. Open the Simulink Library Browser. Copy an instance of the S-Function Builder block from the User-Defined Functions library into the new model.
3. Double-click the block to open the S-Function Builder dialog box.
4. Enter the name of the S-function, `mypwm`, in the **S-function name** field.
5. Select the **Initialization** pane. Make sure that all numeric parameters are set to their defaults (zero) and that **Sample mode** is set to `Inherited`.
6. Select the **Data Properties** pane.
7. In the **Port and Parameter properties** subpane, select **Input ports**. Specify the input (PWM modulator) port as follows:
   - Port name: `u0`
   - Other properties: use defaults
8. Still in the **Port and Parameter properties** subpane, select **Output ports**. Specify the output (PWM signal) port as follows:
   - Port name: `y0`
   - Other properties: use defaults

   Note By default, the **Port and Parameter properties** subpane specifies one input and one output port. However, many device drivers require only an input port or only an output port. For example, an input driver for an analog-to-digital converter requires only an output. In such cases, you should select the port that is not needed in the **Port and Parameter properties** subpane and delete it.
9. Still in the **Port and Parameter properties** subpane, select **Data type attributes**. Specify the port data types as follows:

- `u0`: `uint8`
- `y0`: `uint8`
- Other properties: use defaults

10 Leave all fields under the **Parameters** tab blank. In a real-world driver, you might parameterize hardware settings or other options and add them to your block's mask. For simplicity, this example assumes no parameters are used.

11 Leave the **Libraries** pane unchanged. The driver does not refer to any external source or object files.

12 Select the **Outputs** pane and insert a line of C code.

```
y0[0] = u0[0];
```

This allows the input signal to pass through this block unchanged during simulation.

13 Do not place any additional code under the **Continuous Derivatives** or **Discrete Update** panes.

14 Select the **Build Info** pane. Make sure the **Generate wrapper TLC** option is selected. All other options should be deselected.

15 Click **Build**. The S-function Builder generates several files in your working directory. The names of the generated files are displayed in the **Build Info** pane. Two of them are of interest to us later on:

- `mypwm.`*`mexext`*: MEX-file component for use in simulation (*`mexext`* is a platform-dependent file extension, such as `.mexw32` on 32-bit Windows).
- `mypwm.tlc`: TLC code for generating wrapper S-function.

16 Deselect the **Generate wrapper TLC** option. You edit the generated TLC file, and do not regenerate the TLC file and overwrite edited code.

17 Close the S-Function Builder.

18 Save your model.

Binding the MEX-File to an S-Function Block

In this section you create a binding between the previously created MEX-file and a standard Simulink S-Function block:

1. Copy an instance of the S-Function block from the Simulink User-Defined Functions library into your model.
2. Double-click on the S-Function block to open its dialog box. Enter `mypwm` as the **S-function name** property.
3. Click **Apply** and close the dialog box.
4. Label the S-Function block `pwm driver`.
5. Save the model.

In developing a real-world driver, you would place the `pwm driver` S-Function block into your own drivers library. It is also good practice to keep S-Function blocks that link to generated MEX-files (such as `pwm driver`) separate from the S-Function Builder blocks that generated them. This avoids the possibility that an end user could modify the behavior of this block and generate code unintentionally.

Generated driver MEX-files should be stored in a directory on the MATLAB path along with your other target files.

Masking the Block

In this section you embed the `pwm driver` S-Function block in a masked subsystem. This is useful if simulation and/or code generation parameters are to be added to the driver later:

1. Click on the `pwm driver` block to select it.
2. Select **Create subsystem** from the **Edit** menu in the model window. `pwm driver` is now encapsulated in a subsystem. Label the Subsystem block `pwm driver`.
3. Right-click on the subsystem and select **Mask Subsystem** from the context menu. The Mask Editor window opens.

4 In the **Icon** pane, add the following drawing commands.

```
disp('MYPWM')
port_label('input',1,'Duty cycle')
```

Click **Apply** and close the window.

5 Right-click on the subsystem and select **Look Under Mask** from the context menu. You now apply a mask to the underlying S-Function block.

6 Right-click on the S-Function block and select **Mask S-Function** from the context menu. The Mask Editor window opens.

7 In the **Initialization** pane, add the following code.

```
s = struct('port','PORTA');
set_param(gcb,'RTWdata',s);
```

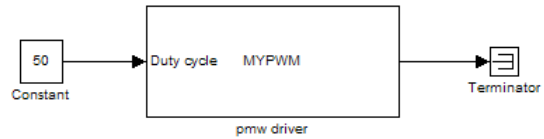
This code extracts mask data (the symbolic port name, PORTA) into a structure that is written into the `RTWdata` structure of the *`model`*`.rtw` file during code generation. This data is then available for use by the TLC file that generates code for the driver block. (See the "Integrating External Code With Generated C and C++ Code" chapter in the Real-Time Workshop documentation for further information on using `RTWdata`.)

8 Click **Apply** and close the Mask Editor window. Save the model.

Customizing Driver Code Generation

When the `mypwm` S-function was built, the **Generate wrapper TLC** option was selected. In this section, you generate code using the TLC file (`mypwm.tlc`) generated by the S-Function Builder. You also examine the TLC file and the C code it produces, and make changes. To exercise the underlying TLC file and inspect code generation as it progresses, you create a test model `test_mypwm`. You then modify the TLC code to generate C code that would be appropriate to an actual hardware PWM driver:

1 Create a new model containing the PWM driver subsystem, with a Constant block and a terminator, as shown in the figure below. In the Constant block parameters, set **Constant value** to `50`, set **Output data type** to `uint8`, and click **OK**. In an actual PWM driver, this would generate a pulse signal with a duty cycle of 50%.



2 Save the model as `test_mypwm`.

3 In the **Solver** pane of the Configuration Parameters dialog box, set **Solver** options to

- **Type**: `Fixed-step`
- **Solver**: `Discrete (no continuous states)`
- **Fixed-step size**: `0.01`

4 In the **Real-Time Workshop** pane of the Configuration Parameters dialog box:

- Select the Real-Time Workshop Embedded Coder target (`ert.tlc`).
- Select the **Generate code only** option.

5 In the **Debug** pane of the Configuration Parameters dialog box, select the **Retain .rtw file** option.

6 Click **Apply** and save the model.

7 Return to the **Real-Time Workshop** pane and click the **Generate code** button.

Real-Time Workshop software generates C code for the model, as well as the `.rtw` file. You now examine information related to the `mypwm` device driver in the `test_mypwm.rtw` file.

8 The `test_mypwm.rtw` file is stored in the build directory. Open `test_mypwm.rtw` into the MATLAB editor.

9 Search for `rtwdata`. The `RTWdata` section containing PORTA is shown below.

```
Block {
      Type        "S-Function"
```

```
        InMask        yes
        SkipBlockFcn       1
        DeletedInIR       1
        MaskType        ""
        BlockIdx        [0, 0, 1]
        SL_BlockIdx       1
        GrSrc        [1, 0]
        ExprCommentInfo {
        }
        ExprCommentSrcIdx {
        }
        RTWdata {
    port   "PORTA"
        }
        Name        "<S1>/pwm driver"
        Identifier       "pwmdriver"
        TID        constant
        RollRegions       [0]
        NumDataInputPorts       1
        DataInputPort {
    SignalSrc  [C0]
    CGTypeIdx  3
    RollRegions  [0]
        }
   ...
```

You can access the RTWdata information from the block as follows:

```
%assign someData =  %<Block.RTWdata.port>
```

With this information, focus on the `mypwm.tlc` file that was generated by the S-Function Builder. The code excerpt below lists the entire file, except for header comments.

```
%implements  mypwm "C"
%% Function: BlockTypeSetup ==========================
%%
%% Purpose:
%%      Set up external references for wrapper functions in the
%%      generated code.
%%
```

```
%function BlockTypeSetup(block, system) Output
  %openfile externs

  extern void mypwm_Outputs_wrapper(const uint8_T *u0,
                           uint8_T *y0);
  %closefile externs
  %<LibCacheExtern(externs)>
  %%
%endfunction


%% Function: Outputs ================================
%%
%% Purpose:
%%      Code generation rules for mdlOutputs function.
%%
%function Outputs(block, system) Output
   /* S-Function "mypwm_wrapper" Block: %<Name> */

  %assign pu0 = LibBlockInputSignalAddr(0, "", "", 0)
  %assign py0 = LibBlockOutputSignalAddr(0, "", "", 0)
  %assign py_width = LibBlockOutputSignalWidth(0)
  %assign pu_width = LibBlockInputSignalWidth(0)
  mypwm_Outputs_wrapper(%<pu0>, %<py0> );

  %%
%endfunction

%% [EOF] mypwm.tlc
```

For device drivers, this `BlockTypeSetup` section is inadequate. Replace the `BlockTypeSetup` section with the following `BlockTypeSetup` function, which contains a port address from the hypothetical target hardware.

```
%function BlockTypeSetup(block, system) Output
  %openfile defines

   #ifndef _MYPWM_
     /* This is a dummy address that you will replace with a
      * meaningful address or declaration suitable for your
```

```
        * hardware.
        */
       #define %<block.RTWdata.port> 0x18h
     #define _MYPWM_
     #endif

   %closefile defines
   %<LibCacheDefine(defines)>
   %%
%endfunction
```

Here you do not import an external C file as the original "wrapper" style TLC code was doing. Instead, you introduce a `#define` relevant to our particular hardware. Of course, this is an optional statement and could be placed elsewhere. Another likely usage would be to modify the above code to include a header file that defines a number of registers or ports by a variety of PWM devices.

If you regenerate code using the modified `mypwm.tlc`, the following code is generated into the file `test_mypwm_private.h`.

```
#ifndef _MYPWM_
/* This is a dummy address that you will replace with a
 * meaningful address or declaration suitable for your
 * hardware
 */
#define PORTA                           0x18h
#define _MYPWM_
#endif
```

Note that the generated TLC file does not include a Start section. You can add your own start section. The following Start section would generate the code `PORTA = 0x00h;` into the model initialize function in the file `test_mypwm.c`.

```
%% Function: Start =======================================
%function Start(block, system) Output
/* Here you would introduce any additional lines of
code needed to initialize this device for your hardware.
For example, you could initialize the period of the PWM
device, its initial output, polarity, and so on.
```

```
One obvious illustration could be just setting the initial
duty to zero as shown below:
*/

%<block.RTWdata.port> = 0x00h;

%endfunction
```

Now, look at the Outputs section. The portion of this code generated by S-Function Builder is

```
%function Outputs(block, system) Output
   /* S-Function "mypwm_wrapper" Block: %<Name> */

  %assign pu0 = LibBlockInputSignalAddr(0, "", "", 0)
  %assign py0 = LibBlockOutputSignalAddr(0, "", "", 0)
  %assign py_width = LibBlockOutputSignalWidth(0)
  %assign pu_width = LibBlockOutputSignalWidth(0)
  mypwm_Outputs_wrapper(%<pu0>, %<py0> );

  %%
%endfunction
```

Rather than calling a function named `mypwm_Outputs_wrapper`, you want your driver code to directly inline the code that implements our PWM driver. During the model outputs computation, this code only needs to translate the input signal `u` to the PWM duty cycle. In this case, change the TLC code to

```
%% Function: Outputs ================================
%%
%% Purpose:
%%      Code generation rules for mdlOutputs function.
%%
%function Outputs(block, system) Output
   /* S-Function PWM Block: %<Name> */

  %assign u = LibBlockInputSignal(0, "", "", 0)
  %<block.RTWdata.port> = %<u>;

  %%
%endfunction
```

The resulting generated code is shown in the model step function of `test_mypwm.c` as follows.

```
/* Model step function */
void test_mypwm_step(void)
{

  /* S-Function PWM Block: <S1>/pwm driver */

  PORTA = test_mypwm_P.Constant_Value;

  /* (no update code required) */

}
```

Device Drivers in Simulation

In this section...

Introduction

When designing device driver blocks, it is important to consider the role of your drivers in both simulation and code generation. This section discusses two approaches to the use of device drivers in simulation and code generation.

If you intend to use your drivers only in the code generation and deployment stages of your development process, you can use separate models for simulation and code generation. This *multiple-model* approach has a number of advantages. For reasons discussed in "Multiple-Model Approach" on page 10-48, this is the recommended approach.

If your driver blocks are used in simulation as well as in code generation, you may want to use a *single-model* approach, which may require that your driver blocks implement special behaviors (such as passing through their input signals) during simulation. This approach is discussed in "Single-Model Approach" on page 10-52.

Multiple-Model Approach

In many applications, it is possible to separate target-specific functions (for example, device drivers or signal conditioning) from the algorithm embodied by the model (for example, a controller). If the algorithmic part of the model can be encapsulated in a common subsystem, it becomes relatively simple to implement two separate models for simulation and code generation. Each model contains the common subsystem, but only the code generation model contains target-specific functions.

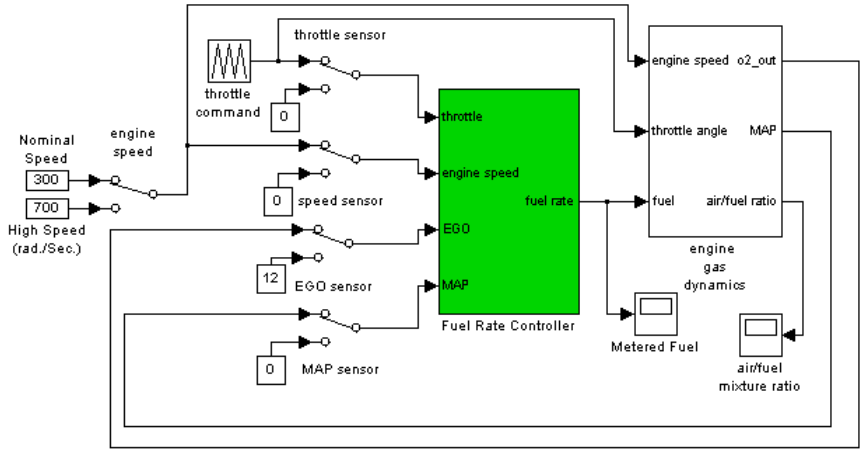
Advantages of the multiple-model approach include:

- There is no need to implement special simulation behaviors (such as use of simulation-only pass-through ports) in the device driver blocks. Real-world scaling and signal conditioning functions can be confined to the code generation model, and omitted from the simulation model.
- Conceptual clarity: Each model operates in a single mode (either simulation or code generation), but reuses components. The purpose of each model is clear to users. In addition, since device driver blocks are not instrumented with pass-through ports, their input/output functions are easier for users to understand.
- Any existing driver can be used without modification in the code generation model.
- Users are free to develop their plant and controller algorithms, without concern over hard-coded pass-through behavior of driver blocks.
- Increased flexibility for the end user: Code generation can be retargeted to different processors by replacing the driver blocks.
- Optimal code generation: Avoids inefficiencies that can occur in code generation when using a single-model approach.

For example, consider a multiple-model approach to a plant/controller system. One model performs a closed-loop simulation of a plant and controller. A second model, used for code generation only, includes the same controller and the I/O device drivers. Code generated from the second model allows the controller to be used in real time on a particular hardware target.

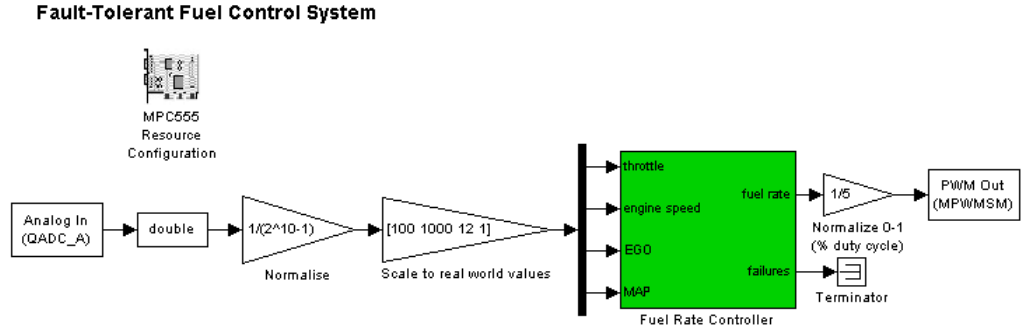
The models shown in this section illustrate this approach. These models were adapted from the Fault-Tolerant Fuel Control System demo.

Multiple-Model Approach: Plant Model for Simulation



The preceding figure shows the simulation version of this model. The controller algorithm (Fuel Rate Controller subsystem) is implemented as a library block. Simulated inputs and outputs to and from the controller are entirely independent of any hardware target to which the model might eventually be deployed.

Multiple-Model Approach: Code Generation Model



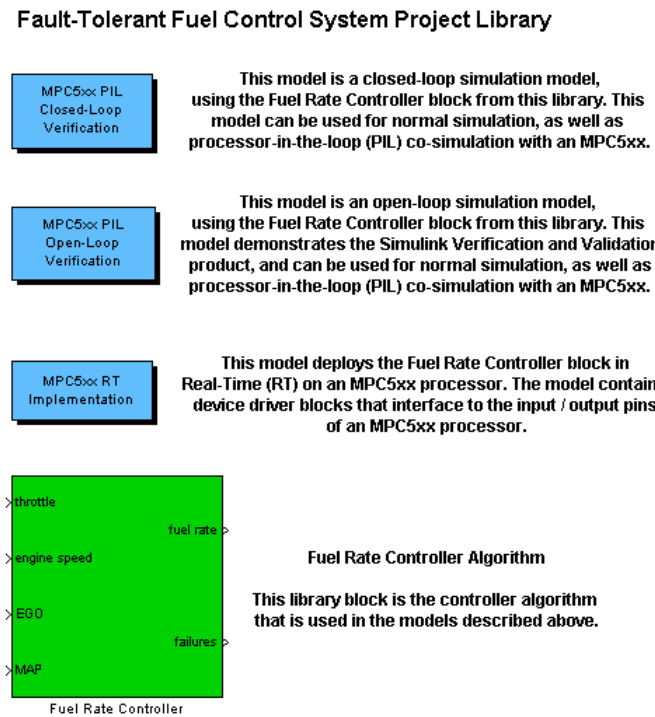
The preceding figure shows a separate version of the model that is specifically targeted for code generation for the Freescale MPC555 processor. This model contains the same controller block, but the controller is connected to MPC555

I/O device drivers (Analog In and PWM Out). The model also contains blocks required for correct operation on the target hardware. These include data type conversion, scaling, and normalization blocks, and an MPC555 Resource Configuration block.

The drivers shown are supplied with Target Support Package product. Code generation could be retargeted to another processor relatively simply by replacing the driver blocks, for example with drivers from the Target Support Package product.

Multiple-Model Approach: Project Library

The multiple-model approach can become problematic if changes are introduced in one model without changing the other. In the example shown, this problem is minimized because the controller algorithm has been extracted into a library block that is used in both models, as shown in the next figure.



The simulation and code generation models have been bundled into a project library, together with the common controller, An alternative would be to implement the controller as a separate model, and reference it with a Model block.

Single-Model Approach

The single-model approach employs the same model for simulation and for code generation. Traditional input simulation drivers generate a nominal value (usually 0), or simply do nothing. Traditional output simulation drivers act as sinks and can often be implemented as stubs.

If you need your drivers to play an active role in a closed-loop simulation, you can implement *pass-through* behavior in your simulation drivers. Pass-through is an option that lets you provide an output signal from your drivers during simulation. In the simplest case, a pass-through device driver block behaves like a "wire," passing its input signal straight through to the output, without any processing. It is also possible to apply scaling or saturation or dynamics processing to the signal as it passes through the block.

Pass-through device drivers resemble traditional device drivers in that the driver behaves differently in simulation than it does when executed on target hardware. However, unlike a traditional simulation driver, a pass-through driver receives and outputs a signal that is significant during simulation.

The following sections describe several approaches to implementation of pass-through behavior device drivers, including possible inefficiencies that may occur in generated code.

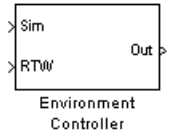
It is assumed that the device drivers discussed below are functioning within a subsystem (for example, a controller subsystem in a plant/controller model) and that subsystem code is generated with the right-click **Build Subsystem** menu option.

Coding Pass-Through Behavior in mdlOutputs

A "traditional" approach implementing pass-through behavior in a simulation driver is to code the pass-through functionality directly into the `mdlOutputs` function of the driver S-function. This is the approach taken in the `ADC_examp` driver. See "mdlOutputs" on page 10-13 for a listing and discussion of the code.

Using the Environment Controller Block for Pass-Through

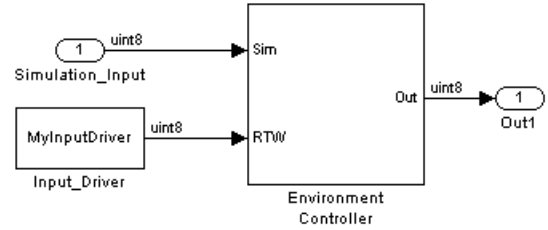
The Environment Controller block (included in the Simulink Signal Routing block library) provides a simple way to implement pass-through drivers. The Environment Controller has two inputs, labeled `Sim` and `RTW`, and a single output.



Environment Controller Block

When a simulation is running, the Environment Controller block routes the `Sim` input signal to the output. During code generation, the block generates code that effectively routes the `RTW` input signal to the output.

You can implement a pass-through driver by creating a subsystem like that shown in Subsystem Implements Pass-Through Logic with Environment Controller on page 10-53. The subsystem contains an S-function device driver block (for an input device such as an ADC), and an Environment Controller block that implements pass-through behavior.



Subsystem Implements Pass-Through Logic with Environment Controller

When the model containing this subsystem is in code generation state, the device driver block connected to the `RTW` input is active, and the path connecting the `Sim` input to the Environment Controller block output port is effectively dead. This path is removed from the generated code by the Real-Time Workshop dead-path elimination optimization.

When the model is in simulation state, the path from the `RTW` input is turned off. The path from the `Sim` input to the output becomes active. This bypasses the device driver block. In this case, the subsystem behaves as though it is a unity gain, passing signals through without change.

Disadvantages of the Environment Controller Block for Pass-Through. When using the Environment Controller block approach to pass-through, a number of inefficiencies can arise in generated code:

- A Switch block underlies the Environment Controller block. In code generation, it is desirable to optimize the Switch block (and any blocks on the unused Switch input) out of the code. This optimization requires that you turn on both the **Block Reduction** and **Inline Parameters** options. These options may not be suitable for your application (for example, if you require all parameters to be tunable).
- If the driver subsystem is built with the right-click **Build Subsystem** menu option, storage for inputs and outputs to and from the subsystem is declared in the containing model's external input (`rtU`) and output (`rtY`) structures.

  For example, in the subsystem shown in Subsystem Implements Pass-Through Logic with Environment Controller on page 10-53, storage would be allocated for the port labeled `Simulation_Input`.
- Output (`rtY`) assignments are generated in the *`model`*`_step` function.

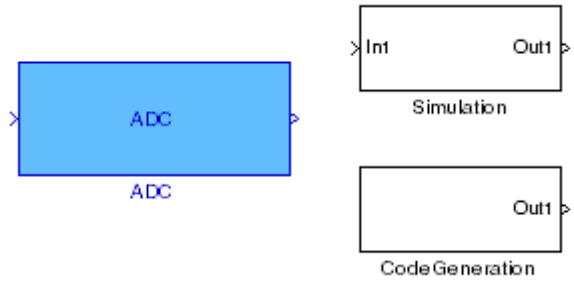
Using a Configurable Subsystem Block for Pass-Through

Another way to implement a pass-through feature is to use a Configurable Subsystem block that includes logic to select either a simulation or code generation version of a device driver.

To do this, a library is constructed, containing both versions of the driver and a master Configurable Subsystem block. The figure below shows a library containing two versions of an ADC driver block:

- The `Simulation` block has both an input and an output port; its `mdlOutputs` function simply copies the input to the output.
- The `CodeGeneration` block has only an output port.

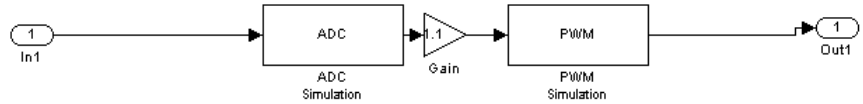
The block labeled `ADC` is a Configurable Subsystem block that is configured to select either `Simulation` or `CodeGeneration`.



Rather than using the conventional manual selection method (the Configurable Subsystem's **Block Choice** context menu), the `ADC` Configurable Subsystem block has mask initialization code that makes the selection automatically, depending on whether the model is in simulation or code generation mode. The mask initialization code is listed below.

```
path = rtwenvironmentmode(bdroot);
cssblk = gcb;
if path
    disp('Taking simulation path')
    set_param(cssblk,'BlockChoice','Simulation');
else
    disp('Taking rtw path')
    set_param(cssblk,'BlockChoice','CodeGeneration');
end
disp(get_param(cssblk,'BlockChoice'))
```

The following block diagram shows a subsystem that includes both the ADC Configurable Subsystem block functioning as an input driver, and a similar Configurable Subsystem block (`PWM`) functioning as an output driver.



Disadvantages of the Configurable Subsystem Block for Pass-Through. When using the Configurable Subsystem block approach to pass-through, a number of inefficiencies can arise in generated code:

- If the driver subsystem is built with the right-click **Build Subsystem** menu option, storage for inputs and outputs to and from the subsystem is declared in the model's external input (`rtU`) and output (`rtY`) structures.
- Output (`rtY`) assignments are generated in the *`model`*`_step` function. These can be eliminated by turning on the **Inline Parameters** option, but inlining parameters may not be suitable for your application.

Index

R

S

T