Embedded Target for Motorola® HC12

For Use with Real-Time Workshop®

- Modeling
- Simulation
- Implementation

User's Guide

Version 1

How to Contact The MathWorks:

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

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

508-647-7000 Phone

508-647-7001 Fax

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

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

Embedded Target for Motorola HC12 User's Guide

Revision History

February 2003	Online only	Version 1.0 (Release 13+)
June 2004	Online only	Version 1.1 (Release 14)
October 2004	Online only	Version 1.1.1 (Release 14SP1)
March 2005	Online only	Minor update for version 1.1.2 (Release 14SP2)
September 2005	Online only	Minor update for version 1.1.3 (Release 14SP3)

Contents

1 Getting Started

2 Configuring the Target

3 Generating Real-Time HC12 Programs

4 Setting HC12 Timing Parameters

5 Code Generation and Code Generation Reports

6 Creating Device Drivers

7 Creating Custom Project Stationery

1

Getting Started

What Is the Embedded Target for Motorola HC12?

The Embedded Target for Motorola HC12 is an add-on product for use with the Real-Time Workshop® Embedded Coder. It provides a unified set of tools for developing real-time applications for the Motorola HC12/HS12 processor. This product was developed and tested with the Motorola EVB912DP256 board, which employs an MC68HC9S12DP256 microcontroller.

The Embedded Target for Motorola HC12 generates code for the Motorola HC12 processor from Simulink block diagrams. Code generation uses the highly efficient Real-Time Workshop Embedded Coder format. Used in conjunction with Simulink®, Stateflow®, and the Real-Time Workshop Embedded Coder, the Embedded Target for Motorola HC12 lets you

- Design and model your system and algorithms.
- Compile, download, run, and debug generated code on the target hardware, seamlessly integrating with the CodeWarrior for Motorola HC12 development environment.

The next section outlines the major features of the Embedded Target for Motorola HC12.

Feature Summary

Support for RAM and Flash Memory Models

The EVB912DP256 provides 12K RAM and 256K flash memories. The 16-bit addressing architecture allows direct access to only 64K of contiguous memory. The memory segmentation model of the EVB912DP256 uses a `PPAGE` (byte) register to page in 16k memory pages.

The Embedded Target for Motorola HC12 supports several memory models by providing special CodeWarrior project stationery. You can select the desired memory model from an option menu before generating code. The following memory models are supported:

- *Small memory model for RAM* allows access to the entire 12K of RAM.

- *Small memory model for flash* allows direct access of up to 64K bytes of flash memory.
- *Banked memory model* supports access of the entire 256K bytes of flash memory on the EVB912DP256 board. The banked memory model can extend beyond 256K bytes for boards that include additional memory.

Modifiable Project Stationery

The Embedded Target for Motorola HC12 uses special CodeWarrior project stationery. This project stationery can be modified, extended, or replaced with your own CodeWarrior project.

Using the CodeWarrior IDE, you can modify settings of the project stationery. Project settings allow you to customize instructions provided to the assembler, compiler, linker, and debugger. You can add your own custom C code to the project stationery.

During code generation, project stationery is replicated and placed beneath your current working directory. The build process automatically invokes CodeWarrior and opens the project stationery. After the project is compiled and linked, a click on the CodeWarrior IDE start button lets you download and run your application to the EVB912DP256, via the ICD12 Background Debugging Module (BDM).

Extensible Device Driver Library

The Embedded Target for Motorola HC12 provides a library of sample I/O device drivers. You can use these device drivers as is, or modify, extend, or replace them for your needs. The drivers are fully documented to help you modify them to your requirements.

The library includes the following blocks:

- ADC Input (analog-to-digital converter)
- Digital Input
- Digital Output
- Master Block
- PWM Output (Pulse Width Modulation)

The driver blocks are based on a driver paradigm that is easy to understand and replicate. With minimum effort, you can modify device drivers and extend the device driver block library to support drivers that you may have already implemented in production applications.

The Master block is a special block that is required in every model used for code generation with the Embedded Target for Motorola HC12. The functions of the Master block are summarized in the next section.

Resource Management via the Master Block

A *resource collision* occurs in a model when two or more device driver blocks require the same hardware resource. For example, suppose that model contains a Digital Output block, configured to use PORTB as the output channel. If copies of this block are added to the model, each copy would be contending for use of PORTB, resulting in a resource collision.

The Master block maintains a *resource database* that helps guard against resource collisions. As device driver blocks are added to a model, they register resource usage with the Master block. If a block requests a resource that is already in use, a resource collision error is reported via a dialog box, and Simulink highlights the conflicting blocks in the model. You can then correct this error by selecting an alternate resource for the block, or by eliminating one of the conflicting blocks. See “Hardware Resource Management” on page 9-6 for details.

Computation of Accurate, Hardware-Achievable Sample Time

The `hc12_closest_st` function helps you set a sample time for your model that is achievable via the on-chip Clock Reset Generator (CRG). This feature lets you simulate your model using the same step size that will be used when generated code is deployed on the target hardware. See Chapter 4, “Setting HC12 Timing Parameters” for details.

Report Generation

The Embedded Target for Motorola HC12 generates an extended version of the HTML code generation report supported by the Real-Time Workshop Embedded Coder. The HTML report includes detailed information on code size for RAM and ROM. This information is obtained by post-processing the map file generated by CodeWarrior during compilation.

What You Need to Know to Use This Product

This document assumes you are experienced with MATLAB®, Simulink, Real-Time Workshop, and the Real-Time Workshop Embedded Coder.

Minimally, you should read the following from the "Basic Concepts and Tutorials" section of the Real-Time Workshop documentation:

- "Basic Real-Time Workshop Concepts." This section introduces general concepts and terminology related to Real Time Workshop.
- "Quick Start Tutorials." This section provides several hands-on exercises that demonstrate the Real-Time Workshop user interface, code generation and build process, and other essential features.

You should also familiarize yourself with the Real-Time Workshop Embedded Coder documentation. In particular, you should read the following sections:

- "Data Structures and Code Modules"
- "Code Generation Options and Optimizations"

You should also be familiar with Metrowerks CodeWarrior for HC12 (including the IDE, assembler, linker, and debugger). This is the supported HC12 cross-development environment. Familiarity with CodeWarrior project stationery is also helpful.

Familiarity with the target board and processor is also helpful. The Embedded Target for Motorola HC12 has been developed and fully tested with the Motorola EVB912DP256 board, which employs an MC68HC9S12DP256 microcontroller.

Both the EVB912DP256 board and the Metrowerks CodeWarrior for HC12 cross-development environment are included in the Motorola M68KIT912DP256 HCS12 Development Kit. You can find information about this kit on the Motorola Web site: `http://e-www.motorola.com`.

Required Products

The Embedded Target for Motorola HC12 *requires* these products:

- MATLAB 7.0 (Release 14)
- Simulink 6.0 (Release 14)
- Real-Time Workshop 6.0 (Release 14)
- Real-Time Workshop Embedded Coder 4.0 (Release 14)

For more information about any of these products, see either

- The online documentation for that product
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section.

Product Limitations

This section describes limitations on the Embedded Target for Motorola HC12. In the current release, these are

- The Embedded Target for Motorola HC12 does not support any of the Simulink blockset products.
- The Embedded Target for Motorola HC12 does not support the Simulink model referencing feature.
- Certain ERT code generation options are not supported (or are restricted) by the Embedded Target for Motorola HC12. The restricted options are described in “Restrictions on Code Generation Options” on page 5-6.

Installing the Product

Your platform-specific Installation guide provides all of the information you need to install the Embedded Target for Motorola HC12.

Prior to installing the Embedded Target for Motorola HC12, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you are permitted to install and use.

After the installation process completes, verify the installation by clicking the MATLAB **Start** button. Follow the **Start > Simulink > Embedded Target for Motorola HC12** links. You should now see the Embedded Target for Motorola HC12 submenu.

After installing Embedded Target for Motorola HC12, proceed to the next section, “Using This Guide” on page 1-9 where you will find a quick summary of this document and a quick path to information about the hardware and software required for use with the product, how to set up your target hardware, cross-development tools, and target preferences properties, and how to generate and download applications.

Using This Guide

We suggest the following path to get acquainted with the Embedded Target for Motorola HC12 and gain hands-on experience with the features most relevant to your interests:

- Read "What Is the Embedded Target for Motorola HC12?" on page 1-2 to learn about the general features of the product.
- Read "What You Need to Know to Use This Product" on page 1-5 to understand the prerequisite knowledge required to use the Embedded Target for Motorola HC12, and to learn about related documentation you may need to read.
- Read Chapter 2, "Configuring the Target" to learn how to set up your development environment and configure the Embedded Target for Motorola HC12.
- Read Chapter 3, "Generating Real-Time HC12 Programs" to work through a step-by-step tutorial on generating and deploying an application on target hardware.
- Read Chapter 4, "Setting HC12 Timing Parameters" for important information on how to set a correct and hardware-achievable sample time for your model.
- Read Chapter 5, "Code Generation and Code Generation Reports" to learn more about code generation options, report generation features, and current limitations of the Embedded Target for Motorola HC12.
- The Embedded Target for Motorola HC12 provides a library of device driver blocks. The Master block (required for all models) is particularly important, as it provides facilities for precise control of the HC12 real-time clock, and also manages requests for hardware resources from blocks. For device driver information see
  - Chapter 8, "Blocks — Categorical List", and Chapter 9, "Blocks — Alphabetical List" for descriptions of the device driver blocks provided in the Embedded Target for HC12 block library.
  - "Hardware Resource Management" on page 9-6 for details on how the Master block manages requests for hardware resources from blocks, and how it deals with potential conflicts in resource usage.

- Finally, advanced information on customization of the target is available in
  - Chapter 6, “Creating Device Drivers” This chapter contains information on how to create your own device drivers or customize existing drivers.
  - Chapter 7, “Creating Custom Project Stationery” This chapter describes how to create customized CodeWarrior project stationery for use with processors in the HC12 family.
- See also “Demos and Test Models” on page 1-11 in the next section.

Demos and Test Models

The MathWorks has provided a number of demos and test models to help you become familiar with features of the Embedded Target for Motorola HC12.

If you are reading this document online in the MATLAB Help browser, you can run the demos by clicking on the links in the **Command** column of the following table.

Alternatively, you can access the demo suite from the MATLAB **Start** button. Follow the **Start > Simulink > Embedded Target for Motorola HC12 > Demos** links. You can also type demo commands from the MATLAB command prompt, as in this example:

```
hc12_adc
```

Note that these demos assume use of a Motorola EVB912DP256 evaluation board. The EVB912DP256 features a block of LEDs that are wired to digital output port B. Most of the demos use the LEDs to provide a simple form of visual feedback from the generated programs.

Embedded Target for Motorola HC12 Demos and Test Models

Command	Demo Topic
`hc12_adc`	Demonstrates the use of input and output device drivers. Reads a value from an analog-to-digital converter (ADC) channel and writes an 8 bit result to digital output port B. The 8 bit value is visible on the LEDs of the Motorola EVB912DP256 evaluation board.
`hc12_led`	Generates a continually shifting bit pattern and writes it to digital output port B. The bit pattern is visible on the LEDs of the Motorola EVB912DP256 evaluation board.
`hc12_pwm`	Demonstrates the use of the PWM Output block to generate pulse width modulation (PWM) signals with different duty cycles on several output channels of the Motorola EVB912DP256 evaluation board. You can view the output signals on an oscilloscope.

Embedded Target for Motorola HC12 Demos and Test Models (Continued)

Command	Demo Topic
`hc12_di`	Demonstrates the use of digital input. Reads a signal from the digital input port A input pins on the Motorola EVB912DP256 evaluation board. The signal is delayed and then written to port B, driving the LED display. Requires connection of a signal generator to the port A input pins on the EVB912DP256.
`hc12_fuelsys`	This version of the `fuel_sys` model has been converted to fixed point to allow efficient code generation for the Motorola HC12 microcontroller. To generate code for this example, right-click on the `fuel rate controller` subsystem. Then, select **Real Time Workshop** > **Build Subsystem** from the pop-up menu. Note that this demo requires Stateflow, Stateflow® Coder and the Fixed-Point Blockset.

2

Configuring the Target

Hardware and Software Requirements

This section describes the hardware components and software tools required by the Embedded Target for Motorola HC12. Before you begin to work with the target, make sure that you have all the required components.

Host Platform

The Embedded Target for Motorola HC12 supports only the PC as the host platform, running Windows 2000 or Windows XP.

Hardware Requirements

The Motorola M68KIT912DP256 HCS12 Development Kit includes all required hardware. The hardware includes the EVB912DP256 evaluation board and +12 Vdc power supply. The board includes the MC68HC9S12DP256 microcontroller. The evaluation board and microcontroller specifications include

- 12K RAM
- 256K flash memory
- RS232C interface
- DBM connectors
- Header access to all MCU pins
- 16-MHz oscillator
- CAN connectors
- Reset button
- ICD12 Background Debugging Module (BDM) device with cables

Software Requirements

- The MathWorks products required to use the Embedded Target for Motorola HC12 are specified in “Required Products” on page 1-6.

- Metrowerks CodeWarrior for Motorola HC12 (Version 3.0, or 3.1), including the IDE, assembler, linker, and debugger. The Motorola M68KIT912DP256 HCS12 Development Kit includes this software.

Setting Up Your Installation

The next sections describe how to configure your hardware, development environment, and target preferences.

Note In this document, we assume that you are working with the Motorola M68KIT912DP256 HCS12 Development Kit. This kit contains the EVB912DP256 evaluation board, all required cables, and Metrowerks CodeWarrior for Motorola HC12.

Before You Begin

Make sure that you have installed the Embedded Target for Motorola HC12 correctly, as described in "Installing the Product" on page 1-8.

Setup Overview

You should proceed through the following sections, in order:

- "Setting Up Your Target Hardware" on page 2-5 describes how to connect and configure your development board.
- "Setting Up Metrowerks CodeWarrior for HC12" on page 2-6 describes how to install the CodeWarrior for HC12 IDE.
- "Setting Target Preferences" on page 2-7 describes how to specify target preferences properties such as the location of the CodeWarrior project stationery. Be sure to localize these properties appropriately for your installation.

Verifying Correct Operation

After you have completed the above setup steps, you should verify that all components are working properly by generating a test application from a Simulink model. provides a tutorial that will take you through the process of building, downloading and running a program based on a demo model.

Setting Up Your Target Hardware

To set up the EVB912DP256 board for use with the Embedded Target for Motorola HC12, simply follow the instructions on the Quick Start card provided with the Motorola M68KIT912DP256 HCS12 Development Kit.

In particular, follow the instructions in Section A of the Quick Start card carefully, to ensure that

- The BDM device is properly connected to the parallel port (LPT1) of your host PC.
- Jumper J17 is in the correct (factory default) position.

Setting Up Metrowerks CodeWarrior for HC12

Setting up the Metrowerks CodeWarrior for HC12 environment for use with the Embedded Target for Motorola HC12 requires a few steps, described in the next sections.

Installing CodeWarrior for HC12

To install CodeWarrior for HC12, simply follow the instructions in sections B-D of the Quick Start card provided with the Motorola M68KIT912DP256 HCS12 Development Kit.

We also strongly recommend that you test your hardware and compiler environment by building, downloading, and running a simple project, as described in section E of theQuick Start card.

Setting Target Preferences

This section describes environmental settings associated with the Embedded Target for Motorola HC12. These settings, which persist across MATLAB sessions and different models, are referred to as *target preferences*. Target preferences let you specify properties such as the location of project stationery files and other parameters affecting the generation, building, and downloading of code. Most of the target preference properties for the Embedded Target for Motorola HC12 are related to the selection of the memory model (banked, flash, or RAM) for which the generated program is to be built.

This section is divided into three parts:

- "Target Preference Properties" on page 2-7 summarizes the target preference properties.
- "Editing Target Preferences" on page 2-9 describes how to use the Target Preferences Setup window.
- "Configuring the Memory Model via Target Preference Properties" on page 2-10 describes how to specify the memory model for which your generated programs will be built.

Target Preference Properties

Embedded Target for Motorola HC12 Preferences Summary on page 2-8 summarizes the preference properties, and their defaults, for the Embedded Target for Motorola HC12.

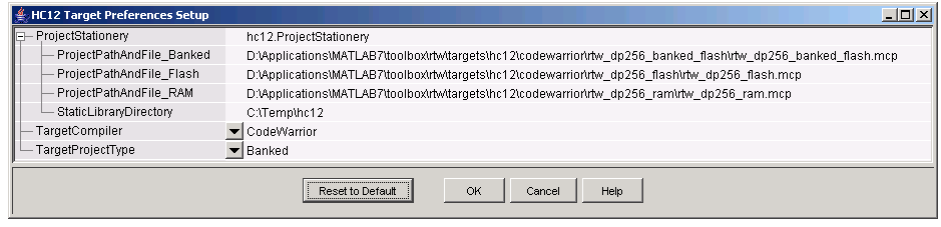
Embedded Target for Motorola HC12 Preferences Summary

Preference Name	Description	Default Value
`ProjectStationery`	Container for preferences data related to project stationery and libraries. Click + sign to view `ProjectPathandFile_Banked`, `ProjectPathandFile_Flash`, `ProjectPathandFile_RAM`, and `StaticLibraryDirectory` properties.	N/A
`ProjectPathandFile_Banked`	Path to CodeWarrior project stationery for Banked memory model	See "Configuring the Memory Model via Target Preference Properties" on page 2-10.
`ProjectPathandFile_Flash`	Path to CodeWarrior project stationery for Flash memory model	See "Configuring the Memory Model via Target Preference Properties" on page 2-10.
`ProjectPathandFile_RAM`	Path to CodeWarrior project stationery for RAM memory model	See "Configuring the Memory Model via Target Preference Properties" on page 2-10.
`StaticLibraryDirectory`	Directory where static object libraries (such as `rtwlib.lib`) are stored (see "Code Generation Options" on page 5-4)	Defaults to the `/hc12` subdirectory within the system temporary directory. (See `tempdir` in the MATLAB Function Reference.)
`TargetCompiler`	Name of installed cross-compiler	`'CodeWarrior'`
`TargetProjectType`	Memory model: select one of: `RAM`, `Flash`, `Banked`. See "Configuring the Memory Model via Target Preference Properties" on page 2-10.	`'Banked'`

Note By default the `ProjectStationery` target preferences properties point to project stationery installed with the Embedded Target for Motorola HC12. This stationery is suitable for use with the demos and examples that you use to familiarize yourself with the product. If you want to modify or customize any of the installed project stationery, we suggest copying the entire project stationery directory to another folder outside the MATLAB directory tree. Then, modify your `ProjectStationery` target preferences properties to point to this stationery. This approach will let you revert back to the original project stationery installed with the Embedded Target for Motorola HC12 (if necessary). For information on this topic, see Chapter 7, “Creating Custom Project Stationery”.

Editing Target Preferences

To configure the target preferences, you use the **Target Preferences Setup** window. This window lets you view, edit, and save the preferences, or reset the preferences to their default (factory) values.



To open the **Target Preferences Setup** window and edit target preferences:

1. Click on the MATLAB **Start** button. Follow the **Start ->Simulink ->Embedded Target for Motorola HC12** links.
2. Select **HC12 Target Preferences** from the **Embedded Target for Motorola HC12** submenu. The **Target Preferences Setup** window opens.
3. Modify the properties you want to change.
4. Click **OK** to close the window and make your changes persistent.

Alternatively, you can open the **Target Preferences Setup** window by typing the command

```
hc12editprefs
```

Configuring the Memory Model via Target Preference Properties

The MathWorks provides CodeWarrior project stationery configured for use with the Embedded Target for Motorola HC12. The project stationery supports several memory models.

The default project stationery files for each memory model are located in separate directories under `matlabroot\toolbox\rtw\targets\hc12\codewarrior`. The following memory models are supported:

- Small memory model for RAM allows access to the entire 12K of RAM. The default project stationery file for this memory model is

  ```
  rtw_dp256_ram\rtw_dp256_ram.mcp
  ```

- Small memory model for flash allows direct access of up to 64K bytes of flash memory. The default project stationery file for this memory model is

  ```
  rtw_dp256_flash\rtw_dp256_flash.mcp
  ```

- Banked memory model supports access of the entire 256K bytes of flash on the EVB912DP256board. The banked memory model can extend beyond 256K bytes for boards that include additional memory. The default project stationery file for this memory model is

  ```
  rtw_dp256_banked_flash\rtw_dp256_banked_flash.mcp
  ```

The `TargetProjectType` target preference property selects which project stationery is used. To set the memory model that will be used to build your generated applications, select `RAM`, `Flash`, or `Banked` from the **TargetProjectType** menu.

During the build process, the Embedded Target for Motorola HC12 replicates the selected project stationery and places a copy under the MATLAB working

directory. Generated code is then placed into the `sources` directory within the project. Once code generation is complete, the Embedded Target for Motorola HC12 invokes CodeWarrior, which compiles the project.

Custom Project Stationery

You can create customized project stationery and use it instead of the default project stationery. CodeWarrior provides project stationery for a number of HC12 and HCS12 derivatives. If you need to generate code for one of the HC12 or HCS12 derivatives, you can modify the project stationery and use it instead of the default project stationery provided by the Embedded Target for Motorola HC12. For information on this topic, see Chapter 7, “Creating Custom Project Stationery”.

3

Generating Real-Time HC12 Programs

Introduction

This chapter is a tutorial describing how use the Embedded Target for Motorola HC12 to generate, download, and run real-time Motorola HC12 applications on a target development board.

This tutorial introduces you to the basic operation of the Embedded Target for Motorola HC12. After you understand the basic feature set, read "Code Generation Options" on page 5-4 for a complete list of all the options available.

Before You Begin

This tutorial requires specific hardware and properly configured software. Please be sure that you have

- Installed the Embedded Target for Motorola HC12 as described in "Installing the Product" on page 1-8.
- Set up your target hardware, cross-development tools, and target preferences properties as described in Chapter 2, "Configuring the Target".

The Example Model

This tutorial uses a simple demo model, `hc12_led`. This demo is provided with the Embedded Target for Motorola HC12.

`hc12_led` (see) is a single-rate model that uses a Digital Output block to write to port B on the HC12. On the EVB912DP256, port B is wired to the LEDs, so writing to port B can illuminate the LEDs.

In the `hc12_led` model, one set of blocks is used to shift a bit left. The result of this operation is summed with the output of a similar set of blocks that shifts a separate bit right. These shift operations are phased so that only one LED is illuminated at any instant in time. As a result, the LEDs first show a bit shifting consecutively to the left (8 times). Then, a bit is seen shifting consecutively to the right (8 times).

This tutorial uses the model to provide a visual indication that the generated program has successfully downloaded and started in target RAM. We will not be concerned with the detailed operation of the model.

Begin by making a local copy of the model:

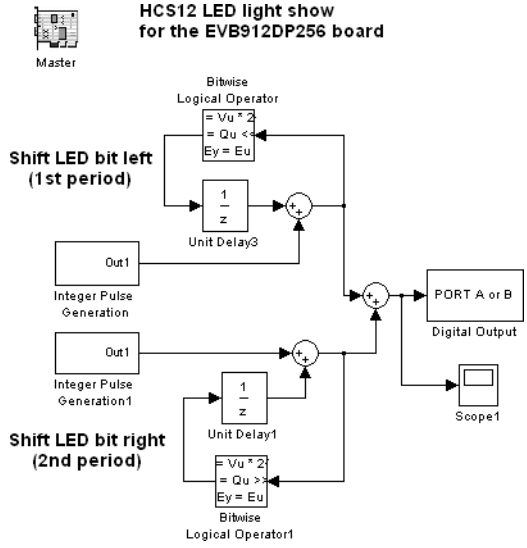
1. Open the model. If you are reading this document online in the MATLAB Help browser, you can open the model by clicking on this link `hc12_led`.

   Alternatively, type the model name at the MATLAB command line:

   ```
   hc12_led
   ```

2. Create a directory, `hc12_tut`, that is outside the MATLAB directory structure. Make `hc12_tut` your working directory.

3. Save a local copy of the `hc12_led` model to your working directory. We work with this copy throughout this exercise.



hc12_led Model

4 Notice the Master block in the upper left corner of the model. One (and only one) Master block is required in every model used for code generation with the Embedded Target for Motorola HC12. Unlike conventional blocks, the Master block is not connected to other blocks via signal lines. The Master block provides the following core functionality for the model:

- Sets the real-time clock period on the HC12. See Chapter 4, “Setting HC12 Timing Parameters” for details.
- Manages a *resource database* to arbitrate potentially conflicting requests for HC12 hardware resources (such as ports) by device driver blocks. See “Hardware Resource Management” on page 9-6 for details.

To learn how to configure the model and generate a program for your target board, continue to “Tutorial: Creating an Application with the Embedded Target for Motorola HC12” on page 3-5.

Tutorial: Creating an Application with the Embedded Target for Motorola HC12

In this tutorial, we build a real-time application from the demo model. We assume that you are already familiar with Simulink and with the Real-Time Workshop code generation and build process. You should also be familiar with the CodeWarrior IDE.

In the following sections, we will

- Select the small memory model for RAM for code generation.
- Configure the model.
- Generate code and build a CodeWarrior project and an executable program.
- Manually download the executable code to a target board via the CodeWarrior debugger.
- Observe the execution of the program on the target board.

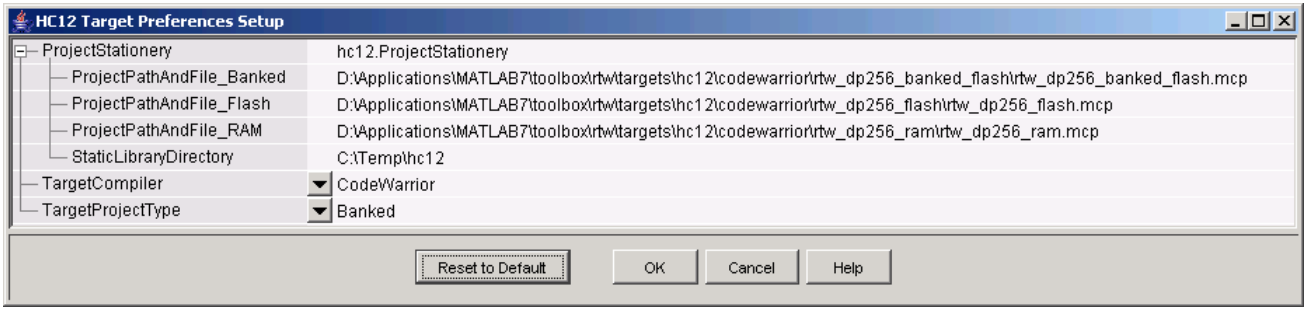
Selecting the Memory Model

In this tutorial, we will generate code for the small memory model for RAM. You select the memory model by editing target preferences in the **Target Preferences Setup** window, as follows:

1 At the MATLAB command line, type

```
hc12editprefs
```

2 The **HC12 Target Preferences Setup** window opens.



3 The `ProjectStationery.ProjectPathandFile_RAM` preference property shows the path to the project stationery file for the small memory model for RAM. By default, this property is set to

```
matlabroot\toolbox\rtw\targets\hc12\codewarrior\rtw_dp256_ram\rtw_dp256_ram.mcp
```

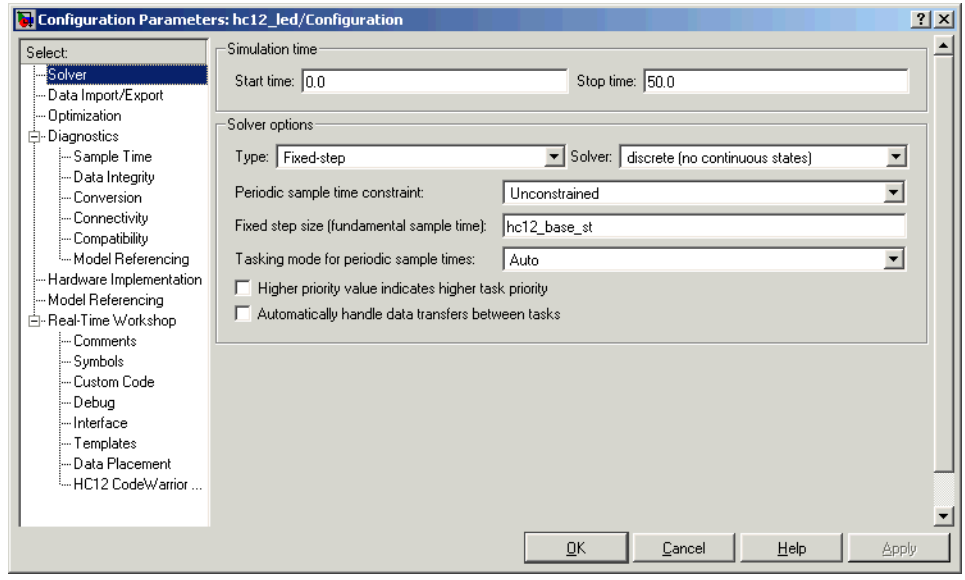
Make sure that `ProjectStationery.ProjectPathandFile_RAM` is set to the default.

4 Select the `RAM` option from the `TargetProjectType` pull-down menu.

The Embedded Target for Motorola HC12 is now configured to generate code for the small memory model for RAM. Note that the memory model is not a property of the model; it is a property of the target. All models built with this target will use this memory model, until you reconfigure the target preferences properties. For further information, see “Configuring the Memory Model via Target Preference Properties” on page 2-10.

Setting the Model Parameters for Code Generation

1 Select **Configuration Parameters** from the **Simulation** menu. When the **Configuration Parameters** dialog box opens, select the **Solver** pane. Make sure that the solver options are set as shown in the figure below.



Observe that the solver **Fixed step size** is set to the workspace variable `hc12_base_st`. For this demonstration model, the variable `hc12_base_st` specifies the sample time for the model and for the Master block. To simplify this tutorial, `hc12_base_st` has been preconfigured (using a preload function) to a sample time that is achievable on the target hardware.

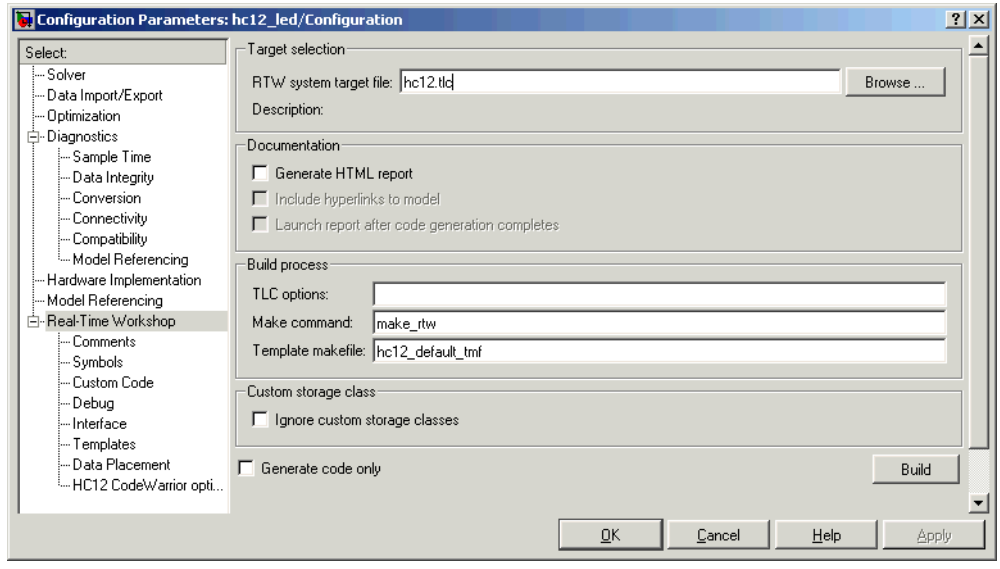
In an actual application, you would normally want to use the `hc12_closest_st` function to help select a sample time for your model. This function computes a sample time that is achievable on the actual hardware (i.e., the on-chip Clock Reset Generator (CRG)). You would then store the sample time value computed by `hc12_closest_st` in a workspace variable, and use the workspace variable as the sample time parameter as required in your model. See Chapter 4, “Setting HC12 Timing Parameters” for further details.

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

3 Click on the **Browse** button to open the **System Target File Browser**. In the browser, select `Embedded Target for Motorola HC12 and CodeWarrior (real-time)`.

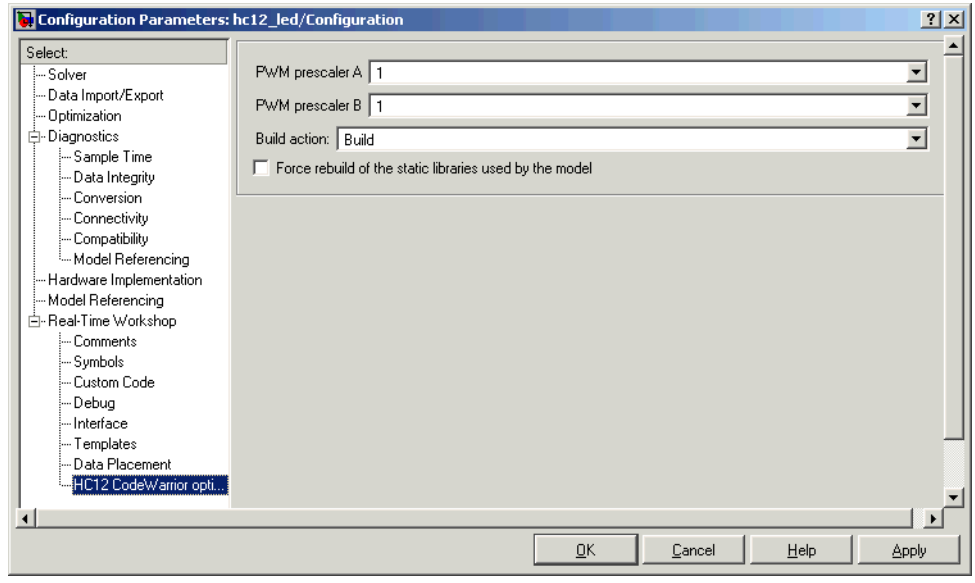
4 Click **OK** to close the browser and return to the **Real-Time Workshop** pane.

5 Deselect the **Generate HTML Report** option. Then click **Apply**. The **Real-Time Workshop** pane settings should now be as shown in this figure.



To simplify this tutorial, we have turned off generation of the HTML code generation report. Later, you can learn about the HTML code generation report in “Code Generation Reports” on page 5-8.

6 Select the **HC12 CodeWarrior options** pane. Make sure that the options are configured to their default settings, as shown in this figure.



Currently, the **Build action** menu has only one option.When `Build` is selected, a CodeWarrior project is created and opened. Future versions may support extended **Build action** capabilities.

7 The model is now configured for code generation. Save the model.

Building the Application

In this section, we will generate code and build both a CodeWarrior project and an executable program:

1 Select the **Real-Time Workshop** pane. Click the **Build** button to initiate the build process. The build process starts CodeWarrior, and begins to display status messages in the MATLAB Command Window.

2 On successful completion of the build process, the following message is displayed in the MATLAB Command Window:

```
### Completed target action: build
```

3 Observe that the build process has created a build directory, `hc12_led_hc12rt`, in your working directory. Use the `dir` command to view the contents of the build directory.

```
dir hc12_led_hc12rt
```

For this model, the CodeWarrior project file `rtw_dp256_ram.mcp` has been generated in the build directory. The build process has loaded this project file into CodeWarrior, and instructed CodeWarrior to make the project.

The generated source code for the project is stored in the `hc12_led_hc12rt\sources` subdirectory. The build process also creates a number of other directories and files. For now, we are only concerned with the CodeWarrior project file that has been generated. See "Build Directories and Files" on page 5-2 for information on the detailed contents of the build directory.

4 The executable code has been built by CodeWarrior and is ready to run. Proceed to "Downloading and Running the Application" on page 3-10.

Downloading and Running the Application

In this final section of the tutorial, we will use CodeWarrior to download the generated executable to target RAM and observe it running on the target board:

1 Activate CodeWarrior and select the **Project** window. The project file `rtw_dp256_ram.mcp` is displayed.

2 To download code and initiate a debugging session, click on the green Debug arrow in the toolbar at the top of the **Project** window.

3 Code is downloaded to the target RAM. During downloading a progress indicator is displayed. When downloading completes, the CodeWarrior debugger window is opened.

4 To start program execution on the target, click on the green Start/Continue arrow in the toolbar at the top of the **Debugger** window.

5 Observe the LEDs on the target board. As the program executes, the LED illumination changes in a pattern, as described in "The Example Model" on page 3-3.

6 If you want, use the debugger controls to halt or continue the program, set breakpoints, observe variables, etc. We recommend that you terminate the debugging session and close CodeWarrior when done.

Using Generated CodeWarrior Projects

The build process has generated a complete CodeWarrior project (in this case, `rtw_dp256_ram.mcp`) in the directory `hc12_led_hc12rt`. You can manually open such a generated project into CodeWarrior and interact with it just as you would any other CodeWarrior project.

You should exercise caution if you make any modifications to a generated project or its source files. If you rebuild code from your model into the same directory, the previous project will be overwritten.

Where to Go Next

An understanding of issues related to setting the HC12 hardware clock period and the sample rate of the model is critical to correct use of the Embedded Target for Motorola HC12. We strongly recommend that you read Chapter 4, “Setting HC12 Timing Parameters” before starting to develop your own applications.

4

Setting HC12 Timing Parameters

Introduction

A common (but less than optimal) method of setting a model's sample time is to specify a theoretically ideal value (even though this ideal sample time may not be achievable on the target hardware). The code generator is then allowed to round off to the nearest achievable value. This can result in a significant discrepancy between the ideal sample rate (used in simulation) and the actual hardware-achievable rate used when the generated code is deployed on the target hardware.

The Clock Rate Generator (CRG) module functions as the HC12 system clock. The Master block provided by the Embedded Target for Motorola HC12 generates the code that sets the CRG frequency by programming the oscillator clock frequency and the bit fields of the 7-bit CRG RTI Control Register (RTICTL):

- RTR Interrupt Modulus Ctr RTR[0:3]
- RTR Interrupt Prescale Rate Select RTR[4:6]

To help you configure the Master block timing parameters, the Embedded Target for Motorola HC12 provides the `hc12_closest_st` function. This function lets you determine a precise sample time value that is

- Achievable by the hardware
- As close as possible to a specified ideal sample time

We recommend that you use the sample time computed by `hc12_closest_st` to set the Master block parameters, and also use it as the step size for your model. This can increase the accuracy of your simulation, avoiding discrepancies between the simulation behavior and real-time behavior of your model. The goal is to ensure that the sample rate you select is applied with the greatest possible accuracy. This is particularly valuable for calculations that require a sample rate when computing coefficients for transfer functions or state-space systems, since the sample rate could have significant effects on the overall stability of the system.

The following sections describe how to use the `hc12_closest_st` function and the Master block to set HC12 timing parameters. When reading the discussion, refer to Table 3-2, "RTI Frequency Divide Rates" in the Motorola

document *CRG Block User Guide*. This document is available in PDF format (`S12CRGV2.pdf`) on the Motorola Web site: `http://e-www.motorola.com`.

The “RTI Frequency Divide Rates” table indicates the frequency divide rates achievable by setting the bit fields of the RTI control register.

The hc12_closest_st Function

`hc12_closest_st` is an M-file command. The full syntax is

```
[st,rtr30,rtr64,percentError] = hc12_closest_st(desiredST,osc)
```

where the input arguments are

- `desiredST`: The desired, or ideal, sample time for your model, in seconds. This sample time may not be achievable by the CRG.
- `osc`: The CRG oscillator clock frequency, in Hz.

and the outputs are

- `st` : The hardware-achievable sample time that is closest to `desiredST`, given the specified clock frequency (`osc`).
- `rtr30` : The value for bits 0-3 of the RTR register that is required to achieve `st`.
- `rtr64` : The value for bits 4-6 of the RTR register that is required to achieve `st`.
- `percentError` : The discrepancy (expressed as a percentage) between the desired sample time (`desiredST`) and the achievable sample time (`st`).

The outputs `rtr30`, `rtr64`, and `percentError` may be omitted. The following calls are legal.

```
[st] = hc12_closest_st(desiredST,osc)
[st, pctErr] = hc12_closest_st(desiredST,osc)
[st, rtr30, rtr64, pctErr] = hc12_closest_st(desiredST,osc)
```

In the following example, the ideal sample time is `0.01` second and the clock frequency is 16 MHz.

```
[st,rtr30,rtr64,percentError]  = hc12_closest_st(0.01,16000000)
### Closest achievable sample time: 0.01024

st =
```

```
0.0102

rtr30 =

     4

rtr64 =

     4

percentError =

    2.4000
```

In the above example, the ideal sample time is not achievable at the given clock rate. Changing the clock rate may not be feasible due to hardware considerations. If so, the best choice in the above case would be to use a sample time of `0.0102` in the model. The next section provides guidelines for consistent use of the sample time computed by `hc12_closest_st` throughout a model.

Computing the Sample Time for Your Model

When building a model, we recommend that you use `hc12_closest_st` to compute the hardware-achievable sample time (`st`) that is closest (preferably with a `percentError` less than 2%) to your ideal sample time. When you have determined the `st` value, use `st` as a parameter throughout the model so that it can be easily modified. Use `st` for the following purposes:

- Set the **Sample time** parameter of the Master block to `st`. Set the **Clock frequency** parameter of the Master block to the value of the `osc` argument you passed in to the `hc12_closest_st` function.
- Set the correct sample time for your model by using `st` as the **Fixed step size** solver parameter for the model. Simply enter the string `st` into the **Fixed step size** field of the **Solver** pane of the **Configuration parameters** dialog box.
- Similarly, you can use `st` as the sample time parameter for blocks that require sample times (such as Delay or ADC Input blocks).
- In a multirate model, each subrate must run at some integer submultiple of the base rate. You can specify the sample rate of a block to run at 1/4 of the base rate by specifying the block sample time as the expression `4*st`.

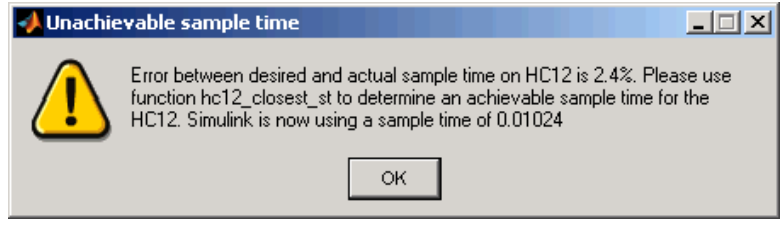
Sample Time Computation and the Master Block

The Master block calls the `hc12_closest_st` function whenever

- The Master block **Clock frequency** or **Sample time** parameters are changed.
- Model compilation is performed (for example, when a Master block is added to a model, when a simulation or code generation is initiated, or when an **Update Diagram** is executed.).

The Master block passes in its **Clock frequency** and **Sample time** parameters as the `osc` and `desiredST` arguments to `hc12_closest_st`. If the `percentError` returned is less than or equal to 2%, the Master block deems the requested **Sample time** to be within tolerance. In generated code, the Master block will set the CRG using the values (`rtr30`, `rtr64`) returned by `hc12_closest_st`.

If the `percentError` returned is greater than 2%, the Master block displays a warning message. For example, where the **Sample time** is 0.01 seconds and the **Clock frequency** is 16 MHz, the returned `percentError` is 2.4. This figure shows the warning message.



The warning message reports

- The `percentError` value
- The hardware-achievable sample time (0.01024 seconds) that is closest to the requested **Sample time** parameter of the block. This hardware-achievable sample time (*not* the **Sample time** parameter of the Master block) will be used in code generation. The recommended corrective action would be to change the **Sample time** of the Master block to 0.01024.

It is good practice, whenever possible, to use `hc12_closest_st` to compute a hardware-achievable sample time when (or even before) you add a Master block to your model. By setting the **Sample time** of the Master block to the computed value, you can minimize the occurrence of such warning messages.

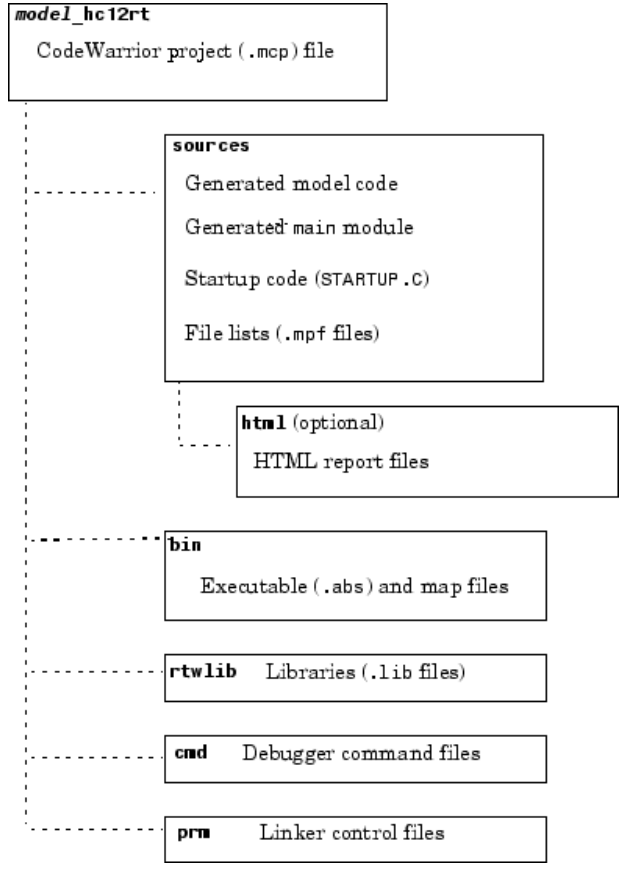
Setting a correct, hardware-achievable sample time for the Master block does not affect the sample time of the model itself, or of any other block in the model. To achieve a consistent sample time throughout the model, follow the recommendations in “Computing the Sample Time for Your Model” on page 4-6.

5

Code Generation and Code Generation Reports

Build Directories and Files

The build directory structure and files created by the Embedded Target for Motorola HC12 are based on the structure of the CodeWarrior project stationery that is used in the build process. This structure differs somewhat from the standard Real-Time Workshop Embedded Coder build directories. summarizes the main directories and files created during the build process.



Directories and Files Created by Build Process

The top-level build directory is created in your working directory, using the naming convention *model*_hc12rt, where *model* is the name of the generating model.

The build directory contains the top-level CodeWarrior project files (`.mcp` files) used during the build process. The files of interest to most users are contained in the following subdirectories of the build directory:

- `sources`: Standard generated files including source code (*`model`*`.c`, *`model`*`.h`) and generated main program module.

  This subdirectory also contains startup code and files lists (`.mpf` files), which are usually of interest only to advanced users (see "Creating Custom CodeWarrior Project Stationery" on page 7-2).

- `sources/html` (optional): This directory is created if the **Generate HTML report** option is selected (see "Code Generation Reports" on page 5-8). It contains the HTML code generation report files.

- `bin`: Generated executable (`.abs`) file suitable for downloading and execution on the target; also linker map file.

Other directories and files within the build directories are mainly of interest to those who want to customize or extend the Embedded Target for Motorola HC12. These are detailed in "Creating Custom CodeWarrior Project Stationery" on page 7-2.

Code Generation Options

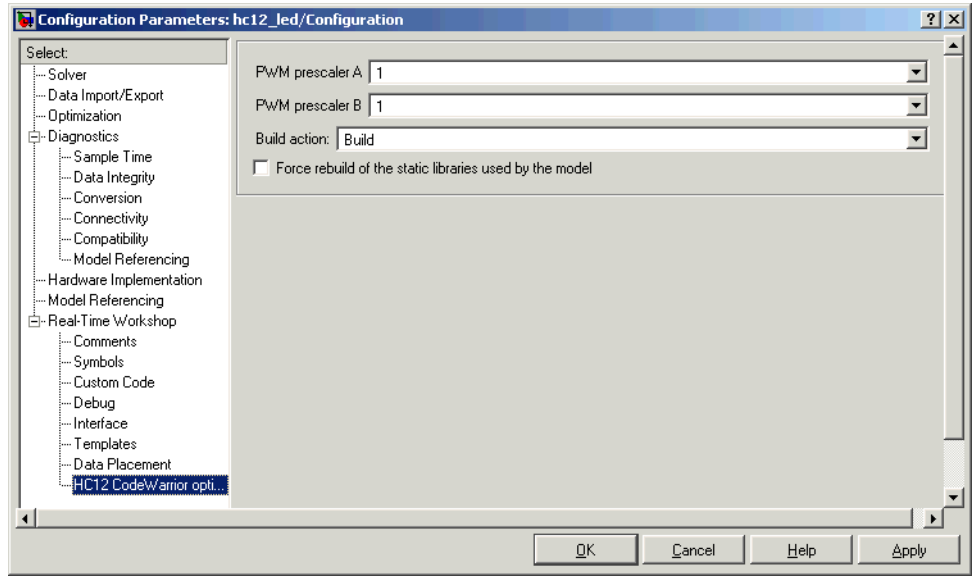
The Embedded Target for Motorola HC12 is an extension of the Real-Time Workshop Embedded Coder embedded real-time (ERT) target configuration. The Embedded Target for Motorola HC12 inherits the code generation options of the ERT target, as well as the general code generation options of the Real-Time Workshop. These options are available via the **Configuration Parameters** dialog box. The options are described in the Real-Time Workshop and Real-Time Workshop Embedded Coder documentation.

Some code generation options of the ERT target are not relevant to the Embedded Target for Motorola HC12, and are either unsupported, or restricted in their operation. See “Restrictions on Code Generation Options” on page 5-6 for details.

Target-Specific Options

The Embedded Target for Motorola HC12 has reserved an option category for target-specific code generation options. You can set these options via the **HC12 CodeWarrior options** pane in the **Real-Time Workshop** pane of the **Configuration parameters** dialog box.

This figure shows the **HC12 CodeWarrior options** pane, with the options configured to their defaults.



The options are

- **Build action:** Currently, this menu supports only one option, `Build`. The build process generates code, creates a CodeWarrior project that includes the generated code, and makes the project. You must download and run or debug the code manually. Future releases may support extended capabilities such as automated downloading.

  Note also that if the Real-Time Workshop **Generate code only** option is selected, a CodeWarrior project is created but a make is not performed.

- **Force rebuild of the static libraries used by the model:** This option controls whether or not object file libraries (such as `rtwlib.lib`) referenced by the model are rebuilt during the build process.

  By default, this option is deselected. Since rebuilding object libraries may involve compiling a large number of source files, we recommend the use of the default.

  In the default case, the build process checks to see if object libraries should be rebuilt. It is sometimes necessary to rebuild `rtwlib.lib` due to changes in the model, even if **Force rebuild** is deselected. For example, if floating-point operations are introduced into a model that previously

generated integer-only code, `rtwlib.lib` must include floating-point code and must be rebuilt.

When an object library is rebuilt, a copy of the library is retained in a directory specified by the `StaticLibraryDirectory` target preferences property (see "Target Preference Properties" on page 2-7). Subsequently, in cases where no rebuild is required, the build process makes a copy of the library from this directory.

If this option is selected, object-file libraries are always rebuilt.

Generating Integer-Only Code

The HC12 is a fixed-point processor. It is possible to use floating-point libraries to support floating-point operations on the HC12. However, we recommend using purely integer code whenever possible, for increased efficiency.

If your application uses only integer arithmetic, deselect the **Support floating point numbers** option to ensure that generated code contains no floating-point data or operations. When this option is deselected, an error is raised if any noninteger data or expressions are encountered during code generation. The error message reports the offending blocks and parameters.

The **Support floating point numbers** option is located on the **Interface** pane of the **Configuration Parameters** dialog box. By default, this option is deselected.

If you decide to use floating-point operations, you will need to make sure **Support floating point numbers** is selected. Note that generation of floating-point code increases the number of files required to build the Real-Time Workshop library (`rtwlib.lib`). This results in significantly longer build times.

Restrictions on Code Generation Options

Certain ERT code generation options are not supported (or are restricted) by the Embedded Target for Motorola HC12. summarizes these restricted options.

Embedded Target for Motorola HC12 Restricted Code Generation Options

Option	Restriction
Hardware implementation pane	The options on the **Hardware implementation** pane are preconfigured to the correct values for the Motorola HC12. These options should not be changed, and are therefore disabled.
MAT-file logging	Option disabled.
Create Simulink (S-Function) block	Option disabled.
External mode	Not supported. An error message is displayed during the build process if **External mode** is selected.
Generate ASAP2 file	A generic ASAP2 file is generated. Note that address values in the file are not updated from the linker MAP file.
Generate an example main program	Option disabled. Note that the Embedded Target for Motorola HC12 generates a target-specific main program, `hc12_main.c`.
File customization template	Option disabled.
Suppress error status in real-time model data structure	This option is always selected. The option is disabled so that it cannot be deselected.
GRT compatible call interface	Option disabled.
Custom code options	The options in the lower half of the Custom Code pane (Labeled **Include list of additional**) is not supported by the Embedded Target for Motorola HC12. This pane includes the fields `Include directores`, `Source files`, and `Libraries`. Code entered into these fields is ignored.

Code Generation Reports

The Embedded Target for Motorola HC12 supports an extended version of the Real-Time Workshop Embedded Coder HTML code generation report.

The extended code generation report consists of several sections:

- The **Generated Source Files** section of the **Contents** pane contains a table of source code files generated from your model. You can view the source code in the MATLAB Help browser. Hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view the relevant blocks or subsystems in a Simulink model window.
- The **Summary** section lists version and date information, options used in code generation, and Simulink model settings.
- The **Optimizations** section lists the optimizations used during the build, and also those that are available. If you chose options that generated less than optimal code, they are marked in red. This section can help you select options that will better optimize your code.
- The report also includes information on other code generation options, code dependencies, and links to relevant documentation.
- The code profile report section includes a detailed itemization of RAM and ROM usage for all code and data sections, and a complete memory map of the generated code.

To generate a code generation report and view the profiling report,

1. Open the **Configuration Parameters** dialog box and select the **Real-Time Workshop** pane.
2. In the **Documentation** subpane, select **Generate HTML report**. By default, **Include hyperlinks to model** and **Launch report after code generation completes** are also selected.

   You can deselect either or both these options if desired.
3. Follow the usual procedure for generating code from your model or subsystem.

4. Real-Time Workshop writes the code generation report files in the `html` subdirectory of the build directory. The top-level HTML report file is named *`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`.

5. If you selected **Launch report after code generation completes**, Real-Time Workshop automatically opens a MATLAB Web browser window and displays the code generation report.

   If you did not select **Launch report after code generation completes**, you can open the code generation report (*`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`) manually into a MATLAB Web browser window, or into another Web browser.

6. If you selected **Include hyperlinks to model**, hyperlinks to blocks in the generating model are created in the report files. When you view the report files in MATLAB, clicking on these hyperlinks will display and highlight the referenced blocks in the model.

Note You can also view the HTML report files, as well as the generated code files, in the Simulink Model Explorer. See the Real-Time Workshop documentation for details.

6

Creating Device Drivers

Creating Device Drivers for the Embedded Target for Motorola HC12

The Embedded Target for Motorola HC12 provides a library of ready-to-run device driver blocks that perform many useful functions (see Chapter 8, “Blocks — Categorical List” or Chapter 9, “Blocks — Alphabetical List”for a description of these drivers). Your application may require you to augment or even replace this set of device drivers. For example, you may want to target an HC12 derivative processor other than the MC9S12DP256. This chapter discusses techniques for development of device driver blocks for use with the Embedded Target for Motorola HC12.

Device drivers for the Embedded Target for Motorola HC12 can (and should) take advantage of the resource management mechanism provided by the Master block. This mechanism is described in “Hardware Resource Management Overview” on page 6-5.

Device driver blocks are implemented as fully inlined C-MEX S-functions. “Implementing Resource Management Compatible Device Drivers” on page 6-13 describes how to create drivers using the Simulink S-Function Builder, and on how to implement resource management callbacks.

Related Documentation

In the following discussion, we describe a technique for creating drivers using the Simulink S-Function Builder. This technique reduces some of the effort of device driver development. However, use of the Simulink S-Function Builder does not obviate the need to understand the code generation process and write or modify a small amount of Target Language Compiler (TLC) code. Basic familiarity with the following topics is required:

- Development of inlined C-MEX S-functions, including use of TLC
- Development of device driver blocks in general
- Use of the Simulink S-Function Builder
- Simulink data objects and the Simulink Data Class Designer

You can find relevant information on these topics in the following documentation:

- Writing S-Functions documentation: Familiarity with writing fully inlined S-functions is helpful in development of device driver blocks. The section "Writing S-Functions for Real-Time Workshop" documents inlining and code generation issues, including the `RTWdata` structure. The section "Building S-Functions Automatically" documents the S-Function Builder.
- "Developing Embedded Targets for Real-Time Workshop Embedded Coder" includes a general chapter on "Creating Device Drivers."
- The Using Simulink documentation provides information on Simulink data objects, the Simulink Data Class Designer, and the Simulink S-Function Builder.
- Target Language Compiler Reference Guide documentation: Working knowledge of TLC is needed to implement inlined device drivers, and/or pass information into or out of the TLC phase of the build process. We suggest that you work through the introductory sections, including "A TLC Tutorial."

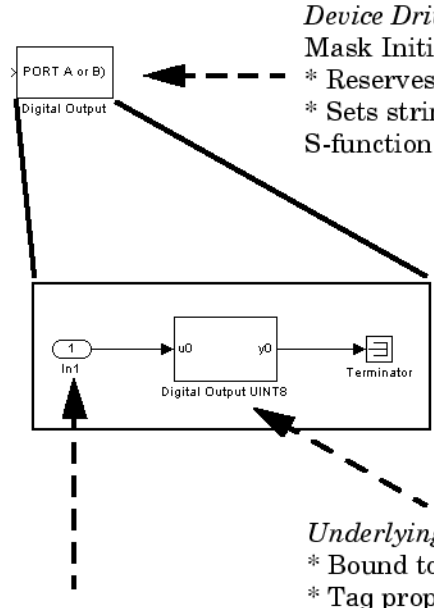
Overview of Device Driver Development Methodology

The methodology described in this chapter was used to develop most of the device drivers provided with the Embedded Target for Motorola HC12. In this methodology, a device driver block is implemented as a masked subsystem that contains an S-Function block. The underlying S-function is fully inlined and is responsible for the device driver code.

The driver subsystem block is assumed to operate in the context of a model that contains a Master block. This is a fundamental requirement of the Embedded Target for Motorola HC12.

The driver subsystem block mask includes a mask initialization callback. This callback interacts with the Master block's resource management database by reserving resource keywords that represent HC12 hardware resources, such as `PORTA` or `PORTB`.

The figure Relationship of Device Driver Subsystem, S-Function, and Hardware Resource on page 6-4 shows the relationship between a driver subsystem block (in this case, the Digital Output driver) its underlying S-function, and a hardware resource.



Device Driver Subsystem Block (masked)
Mask Initialization callback:
* Reserves resource keyword (e.g. `'PORTA'`)
* Sets strings into `RTWdata` struct of underlying S-function

Underlying S-Function Block
* Bound to DLL generated by S-Function Builder
* Tag property set to `"HC12DriverData"`
* TLC implementation is hand-written for code generation.

Hardware Resource
(input) referenced in `RTWdata` struct:

```
doport   "PORTA"
```

Relationship of Device Driver Subsystem, S-Function, and Hardware Resource

The mask initialization callback executes whenever an **Update Diagram** (**Ctrl+D**) command is executed. The subsystem block passes parameters or strings to the mask initialization callback. The callback requests that the Master block reserve the specified resource keywords for the block. If the resources are available, they are reserved for the block (without any visible indication in the block diagram). Otherwise, blocks attempting to claim the same resource(s) are highlighted, and an error message is displayed.

The underlying S-Function block must contain the tag `HC12DriverData`. The mask initialization callback searches all blocks below the level of the driver subsystem block for an underlying S-function that contains the tag `HC12DriverData`. (Each driver subsystem block should contain only one S-Function block containing this tag.)

When the callback finds the underlying S-Function block, it places the requested keywords into a MATLAB structure. At code generation time, this structure is written into the `RTWdata` structure of the S-Function, within the `model.rtw` file. In most cases, this approach greatly simplifies the effort of writing TLC code, because the TLC code for the driver can simply use the field name from the mask to extract the associated keyword.

In the discussion below, an S-function underlying a driver subsystem is created by the Simulink S-Function Builder. The S-Function Builder is actually used only to generate the DLL for the S-function. Everything else generated from the S-Function Builder (such as the wrapper C file and TLC file) is discarded. The generating S-Function Builder block is replaced with a standard S-Function block, bound to the generated DLL. (This prevents end users from accidently overwriting any hand-written driver code).

Note that the Digital Input, Digital Output, and PWM device driver blocks in the Embedded Target for Motorola HC12 library were all created using the S-Function Builder to generate the S-Function DLL.

However, the S-Function underlying the ADC Input block was written manually. This block takes advantage of improvements that are currently possible. For example, the ADC Input block automatically switches between 8-bit and 16-bit signals based on parameter settings.

Regardless of how the underlying S-Function block was created, its TLC implementation must be written manually.

Hardware Resource Management Overview

Each model used for code generation with the Embedded Target for Motorola HC12 must contain a Master block. The Master block provides a mechanism to guard against conflicts among multiple blocks in contention for an HC12 hardware resource, such as a port. The user view of this mechanism is described in “Hardware Resource Management” on page 9-6.

Your device drivers, like the blocks provided in the Embedded Target for Motorola HC12 block library, should use the Master block for resource management.

The Master block monitors resource usage by maintaining a database of resource *keywords*. Keywords are strings that represent HC12 I/O ports, registers (or even individual bits within ports or registers), and other hardware resources.

To define keywords, we provide a user-extensible package of Simulink data objects, the `hc12regs` package. Keywords are defined as properties of classes within the `hc12regs` package. The `hc12regs` package includes classes related to existing drivers (such as `pwm` or `adc`). You use the Simulink Data Class Designer to add keywords to existing classes, or to create your own classes.

A driver block that needs access to a device resource registers a keyword representing that resource in the Master block database. A block registers a keyword via an M-file callback function. Registration takes place at the time the block is added to the model. If no other block in the model has registered the keyword, access to the resource is granted to the requesting block. Otherwise, a resource collision is reported, and the offending blocks are highlighted in the block diagram.

During code generation, the same callback function that registers keywords can be used to propagate keywords (strings) to the `model.rtw` file in order to minimize the TLC code needed for your device drivers.

The first step in creating a device driver that is compliant with the resource management mechanism is to extend the `hc12regs` package by adding keywords. The following section, “Adding Resource Keywords to the hc12regs Package” on page 6-7, describes how to do this.

The second step is to create the driver itself. This step includes implementation of callback functions to register keywords associated with the driver. This step is described in “Implementing Resource Management Compatible Device Drivers” on page 6-13.

Adding Resource Keywords to the hc12regs Package

Note Before editing the `hc12regs` package, we strongly recommend that you read the "Working with Data Objects" section of the Using Simulink documentation, especially the section on the Simulink Data Class Designer.

The process of extending the `hc12regs` package consists of

- Selecting keywords that represent resources required by your driver. "Selecting Resource Keywords" on page 6-7 provides guidelines.
- Using the Simulink Data Class Designer to add properties and/or classes containing these keywords to the `hc12regs` package. This process is described in these subsections:
  - "Setting Up the hc12regs Package for Editing" on page 6-8
  - "Adding Keywords to hc12regs Package" on page 6-11
  - "Testing the Modified hc12regs Package" on page 6-12

Selecting Resource Keywords

This section provides guidelines (rather than hard and fast rules) for selection of resource keywords.

The CodeWarrior header file `6812dp256.h` defines an extensive list of symbols for virtually every I/O device on the HC12 microcontroller. It is good practice to use the symbols defined in this header file as your keywords for a particular resource.

Note that a keyword must be unique within the class that contains it. If this requirement is not met, the behavior of the resource management mechanism is undefined.

However, it is possible (and not unusual) to use the same keyword in multiple classes. For example, either the Digital Input and Digital Output driver blocks can reserve port A or B. Therefore the keywords `PORTA` and `PORTB` are defined for both the `di` and `do` (Digital Input and Digital Output) classes. In such cases, the first block that reserves a keyword gets exclusive access to the

associated resource. For example, if a Digital Input block reserves PORTA, a subsequent request for PORTA from a Digital Output block will be denied.

The level of detail with which you specify resource keywords depends on your application. For example, suppose you want to define keywords associated with a hypothetical 8-bit register, A_REG. The documentation and/or header file may specify that A_REG can be accessed as either

- A full 8-bit byte (defined in a header file as A_REG)
- Individually defined bits within the register (defined in a header file as A_REG0,AREG1..AREG7)

You may want to design your driver to reserve the entire register with one keyword and the individual bits with additional keywords. Accordingly, you would define a class areg with keywords A_REG,A_REG0,AREG1..AREG7.

Your device driver would reserve all of these keywords. This technique guards against other device drivers that may try to reserve just one of the register bits.

You do not always need to specify keywords at this level of detail. For example, the Digital Input and Digital Output driver blocks define keywords PORTA and PORTB, representing locations that can be written to or read from. The related Data Direction Registers (DDR) are also defined by the symbols DDRA and DDRB. However, the Digital Input and Digital Output driver blocks actually reserve only the keywords PORTA and PORTB. (See also the discussion of the Digital Output driver in “Callback Example” on page 6-15.) This is acceptable as long as it is guaranteed that no other drivers will reserve DDRA or DDRB.

Setting Up the hc12regs Package for Editing

The hc12regs package is provided in the directory

```
matlabroot\toolbox\rtw\targets\hc12\blocks\@hc12regs
```

Normally, the hc12regs package is filtered out of the Simulink Data Class Designer display, to avoid the possibility that an end-user will accidentally modify the package. Accordingly, you cannot open the hc12regs package into Simulink Data Class Designer and edit it in place.

Instead, you must circumvent this restriction by making a local copy, modifying it, and replacing the original package. Do this as follows:

1. As a precaution, make a copy of the original `@hc12regs` directory, and rename it. (For example, rename it to `backup_@hc12regs`.)

2. Create a directory in which you will edit the hc12regs package. This directory must *not* be in the MATLAB tree. For the purposes of this discussion, we will call this directory `/mykeywords`.

3. Move the entire `@hc12regs` directory from

   ```
   matlabroot\toolbox\rtw\targets\hc12\blocks\@hc12regs
   ```

   to the `/mykeywords` directory you just created.

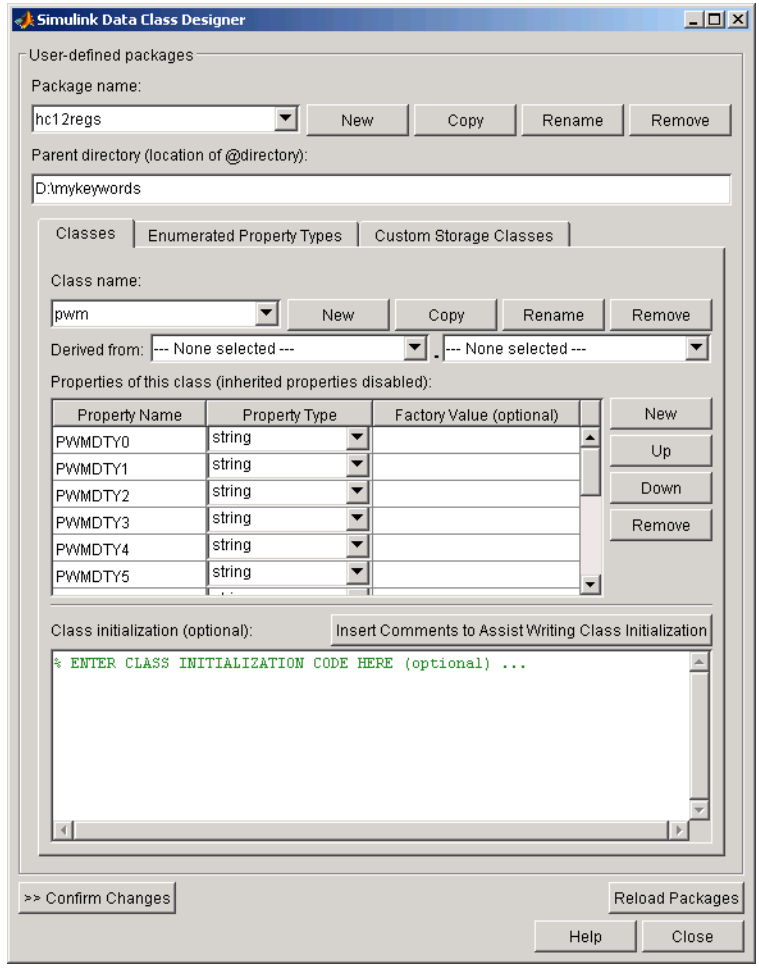
4. Make the `/mykeywords` directory your working directory.

   ```
   cd mykeywords
   ```

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

   ```
   sldataclassdesigner
   ```

6. Your copy (in the `/mykeywords` directory) of the `hc12regs` package should now be visible to the Simulink **Data Class Designer**. Verify that the `hc12regs` appears in the **Package name** field of the Simulink **Data Class Designer**, as shown in hc12regs Package and pwm Class Visible in Simulink Data Class Designer on page 6-10.



hc12regs Package and pwm Class Visible in Simulink Data Class Designer

7 Several preexisting classes are provided within the `hc12regs` package:

- `pwm`: keywords associated with the Pulse Width Modulation device driver
- `adc`: keywords associated with the ADC Input device driver
- `di`: keywords associated with the Digital Input device driver
- `do`: keywords associated with the Digital Output device driver

You can view preexisting classes and their properties by selecting each class in the **Class name** field.

Adding Keywords to hc12regs Package

As shipped, the package files in the `@hc12regs` directory are marked read only. Before adding keywords or otherwise editing the `hc12regs` package, the read-only attribute of the package files must be turned off. Otherwise, changes to the package cannot be written out.

To make the `hc12regs` files writable, first make sure that your working directory is the `/mykeywords` directory that you created in the previous section. Then, type the following at the MATLAB prompt.

```
!attrib -r /S *.*
```

To add keywords to the `hc12regs` package:

1. In the Simulink **Data Class Designer**, select `hc12regs` in the **Package name** field.
2. Select the **Class name** to which you want to add keywords. Alternately, create a new class by clicking on the **New** button.
3. Enter each keyword in the **Property Name** field of the class.
4. Set the **Property Type** for each keyword to `string`.
5. Make sure that the **Factory Value** for each keyword is empty. If this field is not empty, it will not be possible for any block to use the resource.

   Note also that

   - Keyword strings are case sensitive.
   - If you intend to use any of the existing device drivers, you should avoid deletion or modification of any keywords from the associated classes (`pwm`, `adc`, `di`, and `do`). Deletion or modification of such keywords would make it impossible for the existing device drivers to register their keywords in the database.
6. When your edits are completed, click the **Confirm Changes** button.

7 In the **Confirm Changes** pane, click either **Write All** or **Write Selected** to save your definitions.

8 Close the Simulink **Data Class Designer** window.

9 Exit MATLAB.

10 Move the entire `@hc12regs` directory from the `/mykeywords` directory back to

```
matlabroot\toolbox\rtw\targets\hc12\blocks\@hc12regs
```

11 Your modified version of the hc12regs package replaces the original version. The package will now be filtered out of the Simulink **Data Class Designer**. However, the package is accessible to the Embedded Target for Motorola HC12 for resource management purposes. You can also examine the package via MATLAB commands.

Note If you decide to make further modifications to the hc12regs package, you must repeat the whole procedure of copying, editing, and replacing the package, starting from "Setting Up the hc12regs Package for Editing" on page 6-8.

Testing the Modified hc12regs Package

Note Before testing your modifications, make sure you have exited MATLAB as instructed in the previous section. Then, start a new MATLAB session. This is necessary to ensure that any prior package or class information is ignored and only the newly modified package is used.

You can test your changes via the `findpackage` and `get` commands. For example, suppose you added keywords to the `do` (Digital Output) class. The following command verifies the existence of the package.

```
findpackage(`hc12regs')

ans =
```

```
 schema.package
```

In the following example, an object `x` of class `hc12regs` is instantiated and the `get` method is invoked to report its properties. The list of the `do` properties should include any that were added or changed.

```
x = hc12regs.do

x =
 hc12regs.do

x.get

ans =
    PORTA: ''
    PORTB: ''
     DDRA: ''
     DDRB: ''
...
```

If you have added a new class to `hc12regs`, confirm that your new class is valid. For example, the following displays the properties of the class `myclass`.

```
get(hc12regs.myclass)
```

Implementing Resource Management Compatible Device Drivers

This section describes how to generate a device driver S-function via the Simulink S-Function Builder, add a resource management callback, and modify the TLC code generated for the inlined implementation of your block.

The process consists of the following steps:

- Use the Simulink S-Function Builder to generate a C-MEX S-function (DLL) for your device driver.
- Create a library for your driver blocks and add a subsystem block and an S-Function block to your library. Bind the S-Function block to the DLL generated in the previous step.

- Add a mask and mask initialization callback to your driver subsystem. The callback registers your driver's keywords with the resource database. The callback may also write data into the `RTWdata` structure of the underlying S-Function block. The validity of this structure should be tested by generating and examining a `model.rtw` file.

  Before writing your callback code, it is essential to read "Writing Mask Initialization Code for Device Drivers" on page 6-14 (below), as there are a number of specific requirements for mask initialization callbacks written for the Embedded Target for Motorola HC12.

- Customize the TLC code generated by the S-Function Builder, enabling your block to generate C code.

These steps are expanded in the following sections.

Writing Mask Initialization Code for Device Drivers

This discussion assumes familiarity with

- Masked subsystems and with the Simulink Mask Editor. See "Creating Masked Subsystems" in the Using Simulink documentation if you are not familiar with masked subsystems.
- The use of the `RTWdata` structure for passing parameter information to the code generation process. See "Writing S-Functions for Real-Time Workshop" in the Writing S-Functions documentation for information on `RTWdata`.

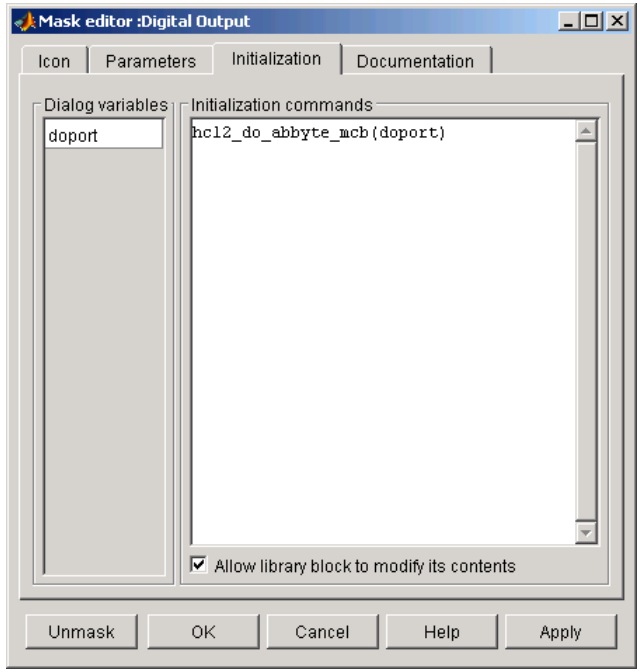
To utilize the resource management mechanism of the Embedded Target for Motorola HC12, device drivers must implement a mask initialization callback that registers the driver's keywords with the resource database. Note that this mask initialization code belongs to the driver subsystem block in the library, *not* the underlying S-Function block.

Your mask initialization callback must call the `reservationmanager` function to register keywords. In addition, you may want your callback to pass data (such as the names and values of block parameters) directly into the `RTWdata` structure of the underlying S-function. For this purpose, we provide a special function, `hc12_setsfunrtwdata`. By using the `RTWdata` structure in this way, you can maintain consistent variable naming throughout the code generation process, from the subsystem mask down to the device driver TLC code.

Note that your callback M-files must be located on the MATLAB path. We recommend that you locate your callback M-files in the same directory as your other driver-related files, such as DLLs.

Callback Example. The subsystem blocks in the Embedded Target for Motorola HC12 block library contain many useful examples of mask initialization callback code. In this section, we will examine some of the callback code for the Digital Output block. While reading this discussion, it will be helpful to view the mask of the Digital Output block in the Mask Editor.

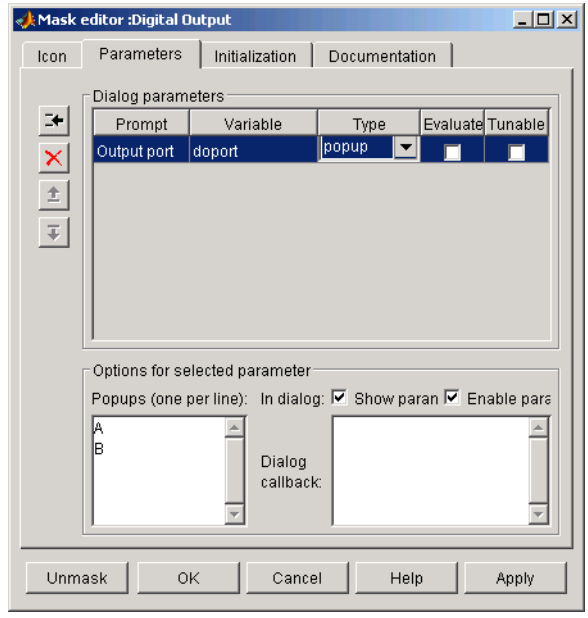
Callbacks are invoked from the **Initialization** section of a the driver block mask. Typically, the callback code in the mask consists of a single line that invokes the body of the callback function, which is implemented in an M-file. The M-file, in turn, makes the appropriate calls to `reservationmanager` and related functions. The callbacks for the Embedded Target for Motorola HC12 blocks are implemented in the M-files named `xxx_mcb.m` under the `matlabroot\toolbox\rtw\targets\hc12\blocks` directory. For the Digital Output driver, the initialization code invokes the M-file callback `hc12_do_abbyte.m`, as shown in this figure.



Mask Initialization Callback of Digital Output Driver

The function `hc12_do_abbyte` is named by the convention that this HC12 device driver writes to a digital output (`do`); writing a single byte to either port A or B (`abbyte`).

The block parameter definition in the mask is shown by this figure.



Output Port Parameter Properties for Digital Output Driver

The user-selected **Output port** parameter is represented by the mask variable `doport`. Note that in order for the `hc12_do_abbyte_mcb` function to correctly retrieve the parameter value, the **Evaluate** option must be deselected.

The following code is a full listing of the `hc12_do_abbyte_mcb` function from `hc12_do_abbyte_mcb.m`.

```
function hc12_do_abbyte_mcb(DOPortChoice)
    %
    % HC12 Digital Output (ports) A or B, Mask Initialization Callback.
    %
    % $Revision: 1.1.4.2 $  $Date: 2005/07/11 20:57:05 $
    % Copyright 2002 The MathWorks, Inc.

    % Create resource keyword to be reserved in resource database
    DOPortChoiceStr = strcat('PORT',DOPortChoice);

    % Try reserving 'PORTA' or 'PORTB' for this block instance
```

```
% If the resource is not available, it will error out immediately.
reservationmanager('do', {DOPortChoiceStr} );

modelRTWFields = struct( ...\
          'doport', DOPortChoiceStr, ...\
          'ddrStr', strcat('DDR',DOPortChoice) );

% Insert modelRTWFields in the I/O block's S-function
% containing the Tag 'HC12DriverData'
hc12_setsfunrtwdata(modelRTWFields);
```

The `hc12_do_abbyte_mcb` function has two purposes:

- To register resource keywords and reserve the associated resources for the block
- To pass parameter data to an `RTWdata` structure that will be created in the `model.rtw` file at code generation time

Resource Reservation. The Digital Output driver block call to `hc12_do_abbyte_mcb` under the **Initialization** tab is

```
hc12_do_abbyte_mcb(doport)
```

The argument passed in (`doport`) is the value of the user-selected **Output port** menu (which is either `'A'` or `'B'`). This string value is concatenated with `'PORT'` to obtain the keyword string `'PORTA'` or `'PORTB'`.

The resultant keyword is passed in to the `reservationmanager` function. The syntax of the `reservationmanager` function is

```
reservationmanager('className', {resources} );
```

where `className` is the name of a driver class defined in the `hc12regs` package, and `resources` is a cell array of resource keywords (strings).

In `hc12_do_abbyte_mcb` the call requests that the chosen port (`'PORTA'` or `'PORTB'`) be reserved for the block, which is associated with the `do` (Digital Output) class.

```
reservationmanager('do', {DOPortChoiceStr} );
```

If the reservationmanager call fails, execution of the callback is terminated. Otherwise, reservationmanager grants access to the port.

Note that hc12_do_abbyte_mcb does not reserve Data Direction Registers 'DDRA' or 'DDRB' since these are implied by the use of either PORTA or PORTB.

Creating RTWdata. The next section of the code builds a structure, modelRTWFields, that contains information that is to be written (at code generation time) to an RTWdata structure in the model.rtw file. For the Digital Output driver block, this structure contains variable/value pairs representing the user-selected port (doport) and Data Direction Register (ddrStr) associated with the port.

```
modelRTWFields = struct( ...\
              'doport', DOPortChoiceStr, ...\
              'ddrStr', strcat('DDR',DOPortChoice) );
```

The final function call

```
hc12_setsfunrtwdata(modelRTWFields);
```

uses the set_param command to add this modelRTWFields structure into the RTWdata of the S-function that underlies the Digital Output driver subsystem.

Note For hc12_setsfunrtwdata to work properly, the S-function receiving the RTWdata must contain the tag HC12DriverData. Make sure that you have added this tag to your device driver S-function, as described in "Create Device Driver Library and Add Your Driver Block" on page 6-22. Also, refer to the hc12_setsfunrtwdata.m source code, located in *matlabroot*\work\r12\toolbox\rtw\targets\hc12\blocks.

As an example of the resultant RTWdata, assume the user has selected output port 'A'. The generated model.rtw file will contain an RTWdata structure containing the following fields and their string values.

```
Tag      "HC12DriverData"
      RTWdata {
```

```
doport  "PORTA"
ddrStr  "DDRA"
      }
```

The keywords in the `RTWdata` can be accessed easily in the TLC implementation of the block. For example, given the `RTWdata` structure above, the following generates code to initialize `PORTA`.

```
%<block.RTWdata.doport> = 0xFF;
```

Since this data is placed directly into the S-function's `RTWdata` structure, it is not necessary to write TLC code to scan through the entire `model.rtw` file to locate the data.

Generate Driver C-MEX S-Function with Simulink S-Function Builder

---

Note This section assumes basic familiarity with the Simulink S-Function Builder. If you are unfamiliar with the S-Function Builder, we recommend that you read the S-Function Builder section of the Using Simulink documentation before proceeding.

To generate your driver S-function:

1. Create a new model.
2. Open the **Simulink Library Browser**. Locate the S-Function Builder block in the **User-Defined Functions** sublibrary. Copy the S-Function Builder block to your new model.
3. Double-click on the S-Function Builder block to open the **S-Function Builder** dialog box.
4. Enter a newname into the **S-function name** field. The S-function name should not duplicate the name of any existing driver DLL provided with the Embedded Target for Motorola HC12. Check the directory where these DLLs are stored:

```
matlabroot\toolbox\rtw\targets\hc12\blocks
```

5 Select the **Data Properties** tab. Then select the **Input Ports** tab. Specify the data type for your driver's input signal. Leave the other properties at their default values.

6 Select the **Output Ports** tab and specify the data type of your driver's output. We recommend specifying the same data type for both the output and input signals, if possible. Leave the other properties at their default values.

The current version of the S-Function Builder requires at least one input and one output. However, a device driver block normally has either an output signal or an input signal, but not both. You can work around this problem by placing your S-Function block into a subsystem, and leaving the unused input or output disconnected. (See "Create Device Driver Library and Add Your Driver Block" on page 6-22.)

7 Select the **Parameters** tab and specify block parameters (if any). Leave the other properties at their default values.

8 Select the **Build Info** tab. Make sure that the **Generate wrapper TLC** option is selected. Leave the other properties at their default values.

9 Click the **Build** button.

The S-Function Builder now generates

- A C-MEX S-function (DLL): `drivername.dll`
- Source code for the S-function: `drivername.c`
- Source code for a wrapper S-function `drivername_wrapper.c`
- A TLC wrapper implementation of the block for code generation: `drivername.tlc`

On successful completion, hyperlinks to the generated files are displayed on the **Compilation Diagnostics** pane.

10 Deselect the **Generate wrapper TLC** option.

The reason for deselecting this option is that in almost all cases, manual editing of the generated TLC code is required. Deselecting the **Generate wrapper TLC** will avoid accidental overwriting of your edited TLC file if you build the S-function later.

11 Click **Close**. Save the model, to preserve the S-Function Builder information.

Create Device Driver Library and Add Your Driver Block

The next step is to create a block library and add a driver block to the library: We recommend you store your drivers in a library that is separate from the Embedded Target for Motorola HC12 block library.

1 From the Simulink **File** menu, open a new Simulink library that will serve as your device driver library.

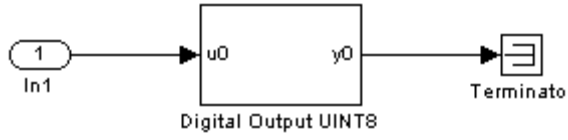
2 Insert a Subsystem block (from the **Ports and Subsystems** sublibrary of the **Simulink Library Browser**) into your library.

3 Open the Subsystem block.

4 Insert an S-Function block (from the **User-Defined Functions** sublibrary of the **Simulink Library Browser**) into the Subsystem. Connect the S-Function block to the input or output ports of the Subsystem as required:

- If your driver is an input driver, connect only its output. Connect a Ground block to the Subsystem input.
- If your driver is an output driver, connect only its input. Connect a Terminator block to the Subsystem input.

As an example, this block diagram shows the Subsystem connections for the Digital Output block.



5 Your block must contain a special tag (`'HC12DriverData'`) that is used by the resource management mechanism. Select the S-Function block (within your library), and use the `set_param` command to insert the tag.

```
set_param(gcb,'Tag','HC12DriverData')
```

The use of the `'HC12DriverData'` tag will be explained later, in the section "Implementing Resource Management Compatible Device Drivers" on page 6-13.

6 Open the S-Function block and enter the name of your newly generated DLL into the **S-function name** field.

7 If you defined any parameters for the driver in the S-Function Builder, enter them into the **S-function parameters** field. Use the same variable names that you used in the S-Function Builder. We strongly recommend explicitly typecasting the variables to the desired data type.

Add Mask Initialization Code to Subsystem Block

This section assumes you have read "Writing Mask Initialization Code for Device Drivers" on page 6-14 and written a callback M-file in accordance with the guidelines given there.

Editing the Block Mask. Once you have written your callback M-file, edit your driver block mask as follows:

1 Right-click on the driver subsystem block and select **Edit mask** from the context menu.

2 Add device driver block drawing commands (if any) under the **Icon** tab. The following drawing command example is from the Digital Output driver.

```
disp('PORT A or B\n(Digital Out)')
```

3 Enter block parameters via the **Parameters** tab. For example, the Digital Output driver defines an **Output port** parameter `doport`, having **Type** `popup` with options `A` and `B` (see Output Port Parameter Properties for Digital Output Driver on page 6-17.)

4 Specify initialization commands via the **Initialization** tab. For the Digital Output driver block, the initialization code invokes the M-file callback `hc12_do_abbyte.m` (see "Callback Example" on page 6-15.)

Test Generation of model.rtw File

To test code generation from your driver, place your driver block into a simple model. Make sure the following Real-Time Workshop options are selected:

- **Retain.rtw file**
- **Generate code only**

Then, click **Generate code**. The generated `model.rtw` file is located in the `sources` subdirectory of the build directory.

Edit the `model.rtw` file and search for the string `RTWdata` in the `model.rtw` file. Depending on your model, the string `RTWdata` will appear in multiple locations. However, you should see your keyword strings precisely as you provided them in the mask initialization callback. For example, from the Digital Output driver:

```
RTWdata {
doport  "PORTA"
ddrStr  "DDRA"
      }
```

Customize the S-Function Block TLC Code

The S-Function Builder generated TLC code for the S-Function block created previously. This code is generic and will require extensive modification to implement the driver functionality you require and make use of any RTWdata information your driver has written to the `model.rtw` file. It is often necessary to use a `blockTypeSetup` function to specify special `#define` or `#include` directives specific to your driver.

Typically, you must also add an initialization or outputs function to the generated TLC file. The example below is the initialization function from the TLC implementation of the Digital Output block.

```
%function Start(block, system) Output

     /* Initialize digital outputs for port
%<block.RTWdata.doport> */
     %% Initialize DDRA or DDRB
     %<block.RTWdata.ddrStr> = 0xFF;
     %% Initialize output on PORTA or PORTB
     %% Note that for PORTB lights off = 0xFF
     %<block.RTWdata.doport> = 0xFF;
     %%
%endfunction
```

The full TLC source code is found in `matlabroot\toolbox\rtw\targets\hc12\blocks\hc12_sfcn_do_abbyte.tlc`.

7

Creating Custom Project Stationery

Creating Custom CodeWarrior Project Stationery

This section describes how to convert and customize CodeWarrior project stationery for use with the Embedded Target for Motorola HC12.

The first three subsections provide basic information you will need to create stationery:

- "Introduction" on page 7-2 provides a conceptual overview of the use and structure of project stationery.
- "Project Stationery Structure Overview" on page 7-3 summarizes the directories and files contained in project stationery.
- "Overview of MathWorks Project Files" on page 7-5 describes special file lists required in project stationery used with the Embedded Target for Motorola HC12.

These sections are followed by a step-by-step description of procedures for creating your own stationery. When creating stationery, work through these sections in order:

- "Setting Up the Project Stationery Directories" on page 7-7: construct a directory structure to contain the files for your stationery
- "Setting Up the rtwlib Subproject" on page 7-8: configure a subproject for building the Real-Time Workshop library
- "Creating an Empty MathWorks Project File" on page 7-10: set up a "placeholder" file list for use in your stationery
- "Modifying the CodeWarrior Project File (.mcp File)" on page 7-10: configure the main CodeWarrior project file for use in the build process
- "Using the New Project Stationery" on page 7-13: reconfigure target preferences to use your stationery in the build process.

Introduction

CodeWarrior provides project stationery and demo projects for a number of processors in the HC12 and HCS12 family. The default project stationery provided by the Embedded Target for Motorola HC12 is based on the CodeWarrior stationery for the MC9S12DP256. (The original stationery is

located in the `Icd-12 Target Interface\Dp256` directory of the CodeWarrior installation.)

Project stationery consists of a number of directories and files. The `ProjectStationery` properties of the target preferences point to the top level of a selected project stationery structure to be used in the build process (see "Configuring the Memory Model via Target Preference Properties" on page 2-10). During the build process, the Embedded Target for Motorola HC12 replicates the entire directory structure of the selected project stationery and places a copy in the build directory, *`model`*`_hc12rt`. Generated code is then placed into the `sources` directory within the project.

If you need to generate code for an HC12 derivative microprocessor other than the MC9S12DP256, you can modify the CodeWarrior project stationery for that derivative and use it instead of the default project stationery provided by the Embedded Target for Motorola HC12. After creating your custom project stationery, you make it visible to the build process by setting the target preferences `ProjectStationery` properties to point to the location of your stationery.

Note that if you intend to support an HC12 derivative microprocessor, there may be other tasks required in addition to creating custom project stationery. In particular, you may need to create device drivers specific to a particular HC12 derivative, and modify or remove the existing device drivers provided with the Embedded Target for Motorola HC12. See "Creating Device Drivers for the Embedded Target for Motorola HC12" on page 6-2 for information on this topic.

Project Stationery Structure Overview

The CodeWarrior project stationery files are located in the `Metrowerks\CodeWarrior MOT_V1.2\Stationery\HC12` subdirectory within your CodeWarrior installation. Stationery for specific HC12 derivatives is located within subdirectories named for the derivative (e.g., `\Icd-12 Target Interface\B32`). For each type of project stationery, CodeWarrior provides a top-level project directory and several subdirectories. Metrowerks provides the following documentation for the CodeWarrior project stationery:

- The `readme.txt` file provided in each project stationery directory

- The CodeWarrior manuals located in the `Metrowerks\CodeWarrior MOT_V1.2\CodeWarrior Manuals` subdirectory within your CodeWarrior installation

To convert existing CodeWarrior project stationery for use with the Embedded Target for Motorola HC12, you copy the stationery structure and modify it by adding and removing certain files and setting project parameters via the CodeWarrior IDE. The following table provides an overview of the contents of a project stationery directory after customizing.

For an example of such a directory, refer to one of the Embedded Target for Motorola HC12 project stationery directories, such as `matlabroot\toolbox\rtw\targets\hc12\codewarrior\rtw_dp256_flash`.

Summary of Project Stationery Structure

Directory	Description
`stationeryname`	Top-level directory of the project. It contains a project file (`stationeryname.mcp`) and one or more debugger configuration files (`.ini` files). The project file is a binary file that is modified via the CodeWarrior IDE, as described in "Modifying the CodeWarrior Project File (.mcp File)" on page 7-10. The `.ini` files are specific to different memory models (RAM, flash, or banked RAM).
`stationeryname/bin`	When a project is compiled, binaries (executables) and map files are written to the `bin` subdirectory. In project stationery, this directory should be empty.
`stationeryname/cmd`	The `cmd` subdirectory contains debugger command files (`.cmd` files). These files are executed at various stages of a debugging session. For example, `startup.cmd` is executed to set up the target system after a host/target the connection has been established. Normally, you can use existing `.cmd` files as provided with the CodeWarrior stationery.

Summary of Project Stationery Structure (Continued)

Directory	Description
`stationeryname/prm`	This directory contains linker parameter (`.prm`) files. These files specify linker settings for a particular memory model. For a description of the syntax of linker parameter files, see the SmartLinker manual (`Manual SmartLinker.pdf`) in the `CodeWarrior Manuals\common` directory. For use with the Embedded Target for Motorola HC12, each `.prm` file must contain a `VECTOR` command to define the timer interrupt vector for the Real-Time Workshop step function (usually `rt_OneStep`). You must obtain the timer interrupt vector address from the documentation for your particular HC12 derivative. You may also need to specify interrupt vectors for other devices you want to support in the linker parameter file.
`stationeryname/sources`	This directory contains generated source code and other files required to build a generated program. These include special file lists ("Overview of MathWorks Project Files" on page 7-5) generated by the build process. These lists specify the generated source code for the project and also the contents of a subproject that builds the `rtwlib.lib` library. Project stationery must contain empty (placeholder) file lists. A startup file, `START12.C`, is also required in the `sources` subdirectory. You can use a `START12.C` file from an existing CodeWarrior project without modification.

Overview of MathWorks Project Files

The build process creates a CodeWarrior project from project stationery and adds generated source code files to the project. To instruct CodeWarrior about which files are to be added to the project, the Embedded Target for Motorola HC12 generates special file lists called *MathWorks Project Files* (`.mpf` files).

The following `.mpf` files are required in the `sources` subdirectory of the project directory:

- `rtw_filelist.mpf` specifies generated code files and header files to be added to the project.
- `rtwlib.mpf` specifies files required to build the Real-Time Workshop library, `rtwlib.lib`. The library is built as a subproject. The library is built differently depending on whether the generating model generated any floating-point code.

Project stationery contains project-relative references to `rtw_filelist.mpf` and `rtwlib.mpf` files. These references act as placeholders. The build process regenerates these files as complete file lists. To use the `.mpf` files properly, the CodeWarrior project must be modified as described in "Modifying the CodeWarrior Project File (.mcp File)" on page 7-10.

Example File Lists

The following example `rtw_filelist.mpf` file was generated from the `hc12_led demo` model.

```
D:\work\r12\extern\include\tmwtypes.h
D:\work\r12\simulink\include\simstruc_types.h
D:\work\r12\rtw\c\libsrc\rtlibsrc.h
hc12_led.c
hc12_led_data.c
hc12_main.c
```

The following example `rtwlib.mpf` was generated for an integer project (that is, the generating model did not contain blocks requiring floating-point code).

```
d:\work\r12\extern\include\tmwtypes.h
d:\work\r12\simulink\include\simstruc_types.h
d:\work\r12\rtw\c\libsrc\rtlibsrc.h
d:\work\r12\rtw\c\libsrc\rt_enab.c
d:\work\r12\rtw\c\libsrc\rt_sat_prod_int8.c
d:\work\r12\rtw\c\libsrc\rt_sat_prod_int16.c
d:\work\r12\rtw\c\libsrc\rt_sat_prod_int32.c
d:\work\r12\rtw\c\libsrc\rt_sat_prod_uint8.c
d:\work\r12\rtw\c\libsrc\rt_sat_prod_uint16.c
```

```
d:\work\r12\rtw\c\libsrc\rt_sat_prod_uint32.c
d:\work\r12\rtw\c\libsrc\rt_sat_div_int8.c
d:\work\r12\rtw\c\libsrc\rt_sat_div_int16.c
d:\work\r12\rtw\c\libsrc\rt_sat_div_int32.c
d:\work\r12\rtw\c\libsrc\rt_sat_div_uint8.c
d:\work\r12\rtw\c\libsrc\rt_sat_div_uint16.c
d:\work\r12\rtw\c\libsrc\rt_sat_div_uint32.c
```

Setting Up the Project Stationery Directories

The first step in converting existing CodeWarrior custom project stationery for use with the Embedded Target for Motorola HC12 is to make a copy of the desired CodeWarrior project stationery. We recommend that you make a separate project stationery structure for each memory model that you want to support.

Your project stationery can be located anywhere outside the MATLAB directory tree.

After copying the project stationery, do the following:

1. Rename the top-level directory of your stationery so that it is distinct from the original CodeWarrior stationery. We suggest that the directory name indicate the targeted HC12 derivative and memory model (e.g., `Dx128_flash`).

2. In the top-level directory:

   - Delete any `.ini` files that are not relevant to your targeted memory model. For example, if you are creating stationery for the flash memory model, delete `banked_flash.ini` and `ram.ini`.
   - Rename the project (`.mcp`) file. Again, we suggest that the name indicate the targeted HC12 derivative and memory model (e.g., `Dx128_flash.mcp`).

3. Delete any files in the `bin` subdirectory.

4. Delete `main.c` from the `sources` subdirectory. Do *not* delete `START12.C`.

5. In the `prm` subdirectory:

- Delete any .prm files that are not relevant to your targeted memory model. For example, if you are creating stationery for the flash memory model, delete banked_flash.prm and ram.prm.
- Rename the remaining .prm file to project.prm. (This is the convention used by the Embedded Target for Motorola HC12.)
- Edit the project.prm file and specify the timer interrupt vector that is to be associated with rt_OneStep. This vector is specific to the HC12 variant CPU that you are targeting and must be obtained from the documentation for that CPU.

  For example, the default project stationery for the Embedded Target for Motorola HC12 specifies

  ```
  VECTOR ADDRESS 0xFFF0  rt_OneStep
  ```

- If required, define any other required interrupt vectors similarly.

Setting Up the rtwlib Subproject

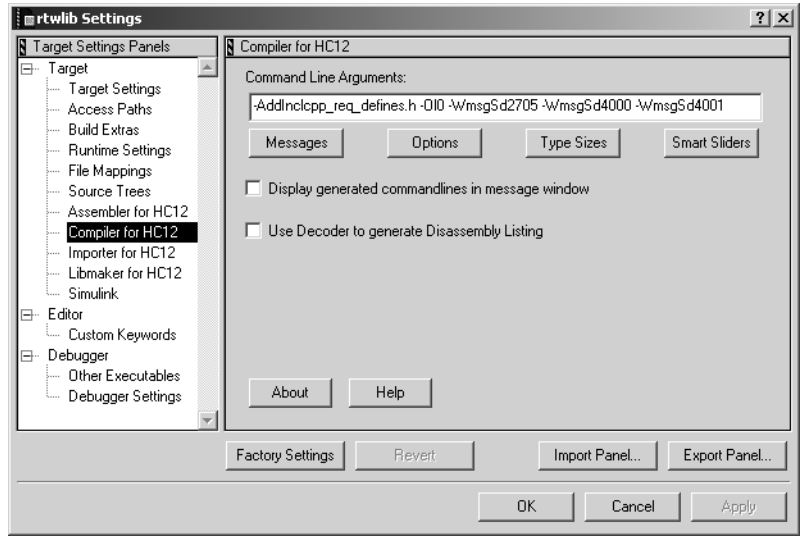
The next step is to configure a subproject for building the Real-Time Workshop library (rtwlib.lib). Within the project stationery for each of the supported memory maps, the Embedded Target for Motorola HC12 provides an rtwlib.mcp subproject. The subproject files are located in the directories:

- matlabroot\toolbox\rtw\targets\hc12\codewarrior\rtw_dp256_banked_flash
- matlabroot\toolbox\rtw\targets\hc12\codewarrior\rtw_dp256_flash
- matlabroot\toolbox\rtw\targets\hc12\codewarrior\rtw_dp256_ram

The rtwlib.mcp subprojects are, for the most part, pre-configured. You need only copy the required file and edit a few settings, as follows:

1. Copy the file rtwlib.mcp from the project stationery directory appropriate for your targeted memory map to the top level of your custom project stationery directory.
2. Right-click on the hc12_rtwlib.mcp icon and select **Properties** from the context menu. Deselect the **Read-only** property.
3. Open rtwlib.mcp into the CodeWarrior IDE.

4 Press **ALT+F7** to open the **Project Settings** dialog box. You will be editing parameters under the headings listed in the **Target** pane.



5 Select **Compiler for HC12**. Add any command switches appropriate to your memory model. For example. if you are creating stationery for a banked memory model, add the following option to the **Command Line Arguments** list.

```
-Mb
```

6 Click **OK** and close the **Project Settings** dialog box.

Note The **File Mappings** options of the `rtwlib.mcp` subprojects are configured to recognize and expand Mathworks Project Files (`.mpf` files) correctly. When the build process is invoked, an `rtwlib.mpf` file is automatically created in the `sources` subdirectory of your top-level project stationery directory. The mapping for `.mpf` files is critical to correct operation of the build process. Do not change or remove this mapping.

---

7 Close `rtwlib.mcp` and exit CodeWarrior IDE.

Creating an Empty MathWorks Project File

Before continuing and editing any project (`.mcp`) files, you must create an empty project file list, `rtw_filelist.mpf`. This Mathworks Project File must exist in the `sources` subdirectory of your top-level project stationery directory. The empty `rtw_filelist.mpf` file is added to your project stationery's main project file as a placeholder. After you have edited your project stationery, you will delete the empty `rtw_filelist.mpf` file, but the project stationery will retain the reference to it. When you generate code using the stationery, the build process regenerates the file as a complete file list.

An easy way to create `rtw_filelist.mpf` from the CodeWarrior IDE is to select the **New...** option from the **File** menu and create an empty text file. Save the file as `rtw_filelist.mpf` in your `sources` subdirectory.

Modifying the CodeWarrior Project File (.mcp File)

This section describes how to customize the CodeWarrior project file (`.mcp` file) for your stationery. Project files are in a binary format and must be edited in the CodeWarrior IDE.

Add rtw_filelist.mpf to Project File

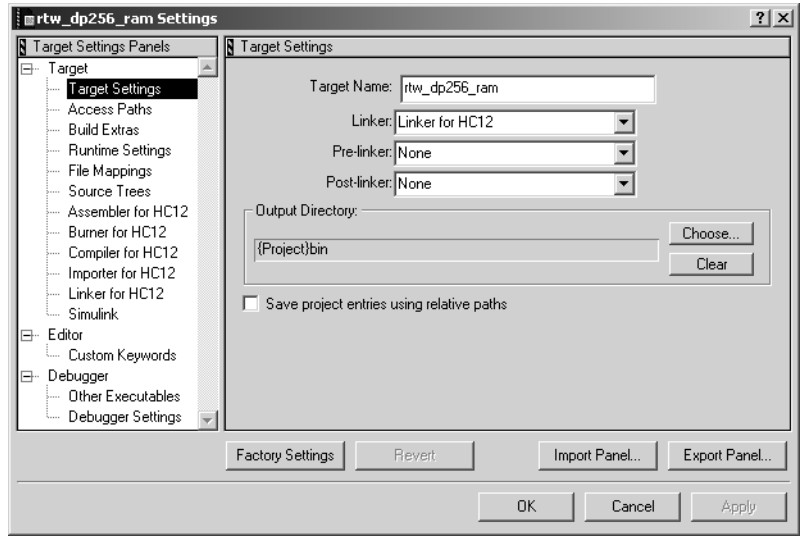
The first step in modifying your project file is to add a reference to the empty `rtw_filelist.mpf` file that you created previously:

1. Open your project stationery file into the CodeWarrior IDE.
2. In the CodeWarrior project window, select **Add File(s)** from the **Project** menu.
3. The **Add File**s dialog box is displayed. Navigate to your `sources` subdirectory and select `rtw_filelist.mpf`. Click **Open**.
4. `rtw_filelist.mpf` is now listed as a file at the top level of your project file.

Specify Project Settings

The next step in modifying your project file is to edit the project settings as follows:

1 Press **Alt+F7** to open the **Project Settings** dialog box. You will be editing parameters under the headings listed in the **Target** pane.



2 Select **Target Settings** from the **Target** pane. Enter the desired name for your custom stationery in the **Target Name** field.

3 If you want to specify access paths to be added to the CodeWarrior search path, select **Access Paths** and enter them.

4 Select **File Mappings** from the **Target** pane. You must define a mapping for `.mpf` files. Otherwise, the build process will not recognize or expand `.mpf` files correctly. Make sure that a mapping for the file extension `.mpf` is defined in the **Extensions** column. If it is not defined, add a mapping by entering the following into the fields of the **Mapping Info** subpane:

- **File type**: `TEXT`
- **Extension**: `.mpf`
- **Edit Language**: `None`
- **Compiler**: `Simulink SysGen`
- **Flags**: `Precompiled`

5 If you added a mapping for `.mpf` files in the previous step, click **Add**. Make sure that the mapping has been added to the stationery by checking the **Extensions** column.

6 Select **Compiler for HC12** from the **Target** pane.We recommend entering the following in the **Command Line Arguments** field.

```
-AddIncl"cpp_req_defines.h" -Ol0 -WmsgSd4000 -WmsgSd4001
```

If you are creating stationery for a banked memory model, add the following option to the **Command Line Arguments** list.

```
-Mb
```

7 Select **Linker for HC12** from the **Target** pane. Specify the following fields:

- **Command Line Arguments**: `-WmsgSd1923`
- **Application Filename**: This property specifies the name of the generated executable (`.abs` file). We recommend `model.abs`.

  Note that, unlike most Real-Time Workshop targets, the Embedded Target for Motorola HC12 does not automatically name the generated application after the source model. Instead, it uses the fixed filename in the **Application Filename** property. This filename is also used by the HTML code generation report.

8 Select **Simulink**. Specify the following fields:

- **Group Name**: `Sources:Simulink`
- **File Name**: `{InputFile}`

9 Click **OK** and close the **Project Settings** dialog box. Leave the project open, as you will need it in the next section.

Add rtwlib Subproject

The next step in modifying your project file is to add the `rtwlib.mpc` subproject to the CodeWarrior project. Before starting the procedure in this section, make sure you have followed the instructions in "Setting Up the rtwlib Subproject" on page 7-8. Add the subproject as follows:

1. In the CodeWarrior project window, select **Add File(s)...** from the **Project** menu.
2. Navigate to your `sources` subdirectory and select `rtwlib.mcp`. file. Click **Open**.
3. The **Add File**s dialog box is displayed. Select the appropriate memory model option (`RamApplication`, `FlashApplication`, or `BankedRam`) and make sure the other options are deselected. Then click **OK**.
4. `rtwlib` is now listed as a subproject at the top level of your project file.
5. Close the project and exit CodeWarrior.

Clean Up Project Directories

After exiting CodeWarrior:

1. Examine your top-level project stationery directory. As a side effect of opening the project files, CodeWarrior has created subdirectories with the suffix `_Data` (e.g., `rtwlib_Data`). These subdirectories are not needed and you should delete them.
2. Delete the empty `rtw_filelist.mpf` from the `sources` subdirectory. Your project stationery retains a file reference to `rtw_filelist.mpf`, and will generate a correct file list during the build process.

Using the New Project Stationery

At this stage, you have completed the steps necessary to modify a project stationery for use with the Embedded Target for Motorola HC12.

Before you can use your stationery in code generation, however, you must make it visible to the build process by

- Specifying the path to your project stationery in the `ProjectStationery` target preferences property that corresponds to the memory model supported by your stationery. (See “Setting Target Preferences” on page 2-7.)
- Selecting the corresponding memory model in the `TargetProjectType` target preferences property.

After setting the target preferences, test your stationery by opening a simple model, such as the `hc12_led` demo, and initiating a build. Your project stationery should be opened automatically by the Embedded Target for Motorola HC12 if you have specified the correct path and stationery name in the target preferences.

8

Blocks — Categorical List

The blocks in the Embedded Target for Motorola HC12 library are organized into categories that support different functions. The tables below reflect that organization.

I/O Device Drivers

Block Name	Purpose
`ADC Input`	Input driver for analog-to-digital converter (ADC) device
`Digital Input`	Input driver for use with digital input Port A or Port B
`Digital Output`	Output driver for use with digital output Port A or Port B
`PWM Output`	Pulse width modulation (PWM) signal generation

Timing and Resource Management

Block Name	Purpose
`Master`	Set HC12 real-time clock rate and manage hardware resources for model

9

Blocks — Alphabetical List

ADC Input

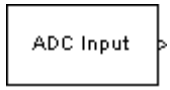
Purpose

Input driver for analog-to-digital converter (ADC) device

Library

Embedded Target for Motorola HC12

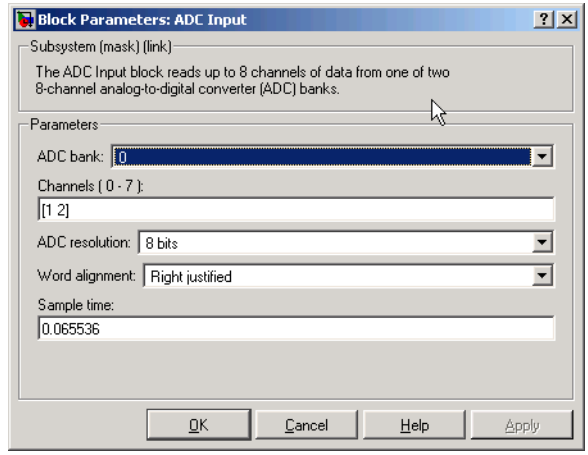
Description



ADC Input

The ADC Input block reads up to 8 channels of data from one of two 8-channel analog-to-digital converter (ADC) banks. You can select either 8 or 10 bits of resolution. When 10-bit resolution is selected, you can specify either right or left justification of data within a 16-bit word.

Dialog Box



ADC bank

Select either bank `0` or `1`. Each bank has 8 channels.

Channels

Specify input channel(s) `0-7`. Specify multiple channels as a vector.

ADC resolution

Select either `8 bits` or `10 bits` of resolution. Default is 8-bit resolution.

Word alignment

If **ADC resolution** is set to `10 bits`, you can select either right or left justification of input data with a 16-bit word. Default is right justification.

If **ADC resolution** is set to `8 bits`, input data is stored as a `uint8`, and **Word alignment** is ignored.

Sample time

Sample time for the block. To set a correct, hardware-achieveable sample time, you should use the `hc12_closest_st` function as described in "Computing the Sample Time for Your Model" on page 4-6.

Digital Input

Purpose

Input driver for use with digital input port A or port B

Library

Embedded Target for Motorola HC12

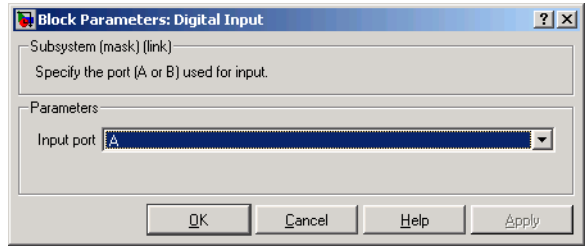
Description

Digital Input

The Digital Input device driver block is configured for use with either port A or port B (represented in generated code as `PORTA` or `PORTB`). Select the desired port from the **Input port** menu in the block dialog box.

The Digital Input driver block produces an input signal of data type `uint8`. During a read, all 8 bits are read from the selected port.

Dialog Box



Input port

Select either port `A` or `B`.

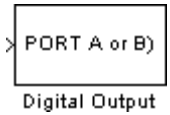
Purpose

Output driver for use with digital input port A or port B

Library

Embedded Target for Motorola HC12

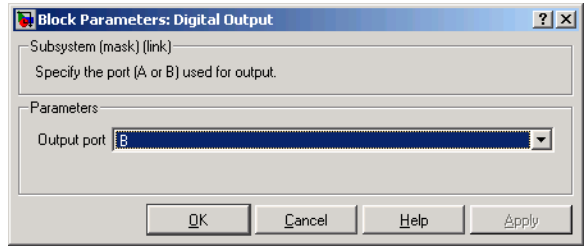
Description



The Digital Output device driver block is configured for use with either port A or port B (represented in generated code as `PORTA` or `PORTB`). Select the desired port from the **Output port** menu in the block dialog box.

The Digital Output driver block produces an output signal of data type `uint8`. During a write, all 8 bits are written to the selected port.

Dialog Box



Output port

Select either port `A` or `B`.

Master

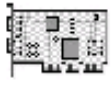
Purpose

Set HC12 real-time clock rate and manage hardware resources for model

Library

Embedded Target for Motorola HC12

Description



One (and only one) Master block is required in every model used for code generation with the Embedded Target for Motorola HC12. The Master block provides the following core functionality for the model:

- Sets the real-time clock period on the HC12. See for further information.
- Manages a *resource database* to arbitrate potentially conflicting requests for HC12 hardware resources (such as ports) by device driver blocks.

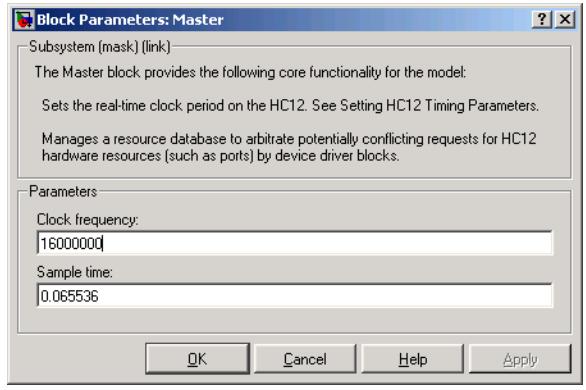
Hardware Resource Management

The Master block provides a mechanism to guard against *resource collisions*. Resource collisions can occur when multiple blocks are contending for an HC12 hardware resource, such as a port. For example, consider a Digital Output device driver block that is configured to use port B as the output channel. If a copy of this block is added to the model, two Digital Output blocks would be contending for use of port B as an output. The function of the Master block is to grant access to port B to one of the contending blocks, report the resource collision condition, and highlight the offending blocks in the block diagram. Depending on the resource, you can then correct the collision. In the case of two Digital Output blocks contending for port B, you could reassign one of the blocks to port A.

The Master block monitors resource usage by maintaining a database of *resource keywords*. Keywords represent HC12 I/O ports, registers (or even individual bits within ports or registers) and other hardware resources. When a driver block needs access to a device resource, it registers a keyword in the Master block database via a callback function. If no other block in the model has registered the keyword, access to the resource is granted to the requesting block. Otherwise, a resource collision is reported.

If you develop your own HC12 device driver blocks, you should make them compatible with the Master block resource monitoring mechanism. To learn how to do this, see “Creating Device Drivers for the Embedded Target for Motorola HC12” on page 6-2.

Dialog Box



Clock frequency

Sets clock frequency in Hz. The default value (16 MHz) is equal to the clock frequency on the EVB912DP256 board.

Sample time

To set a correct, hardware-achieveable sample time, you should use the `hc12_closest_st` function as described in Chapter 4, “Setting HC12 Timing Parameters”

PWM Output

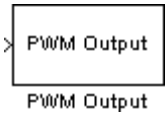
Purpose

Pulse width modulation (PWM) signal generation

Library

Embedded Target for Motorola HC12

Description



The PWM Output block generates a pulse wave with a duty cycle that is determined by the input (modulator) signal. The PWM Output block writes to a single channel (`0-7`).

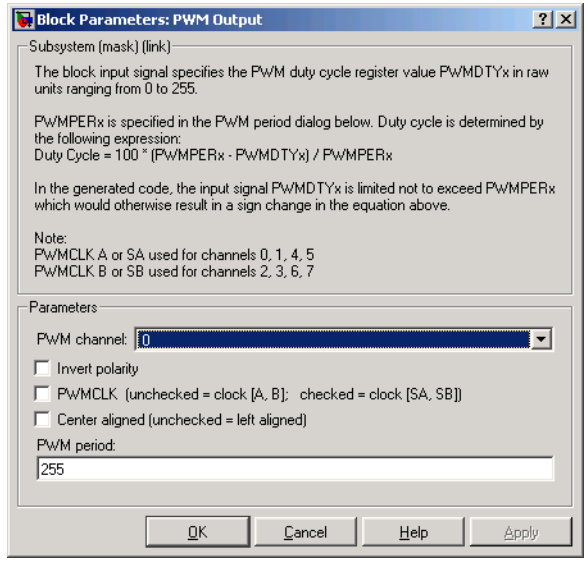
The input signal is of data type `uint8`.

An input signal value of `0` corresponds to `off` or no duty cycle. A signal value of `255` corresponds to `on` or 100% duty cycle. Intermediate values between `0` and `255` correspond to a duty cycle of

```
(x/255) * 100%
```

where `x` is the input signal value.

Dialog Box



PWM channel
Select an output channel 0-7.

Invert polarity
Select this option to invert the polarity of the generated signal. By default, this option is deselected and the signal is not inverted.

Clock B
Select this option to use Clock B as the timing source. By default, this option is deselected and Clock A is used.

Center aligned
When this option is selected, the waveform is center aligned.

PWM period
Period of the generated signal. The PWM period is specified as an 8-bit value with a maximum of 255.

Index