Embedded Target for Motorola® MPC555

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 MPC555 User's Guide

© COPYRIGHT 2002 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

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

Motorola is a registered trademark and MPC555 is a trademark of Motorola, Inc.

Metrowerks and CodeWarrior are registered trademarks of Metrowerks Corporation.

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

Printing History: March 2002 Online only Version 1.0 (Release 12.1+)
July 2002 Online only Version 1.0.1 (Release 13)

Contents

Preface

Installing the Embedded Target for Motorola MPC555 .... **vi**

Using This Guide ..... **vii**

Embedded Target for Motorola MPC555 Demos ..... **viii**

Related Products ..... **x**

Typographical Conventions ..... **xii**

1 Product Overview

Prerequisites **1-2**

Introduction to the Embedded Target for Motorola MPC555 ..... **1-3**
Feature Summary **1-3**
Applications for the Embedded Target for Motorola MPC555 ..... **1-6**

Hardware and Software Requirements **1-9**
Host Platform **1-9**
Hardware Requirements **1-9**
Software Requirements **1-9**

Setting Up and Verifying Your Installation **1-11**

Setting Up Your Target Hardware **1-12**

Setting Target Preferences **1-15**
MATLAB Commands for Working with Target Preferences **1-15**
Editing Target Preferences via the Property Inspector Window **1-19**

Setting Up Your Installation with Diab Cross-Compiler and SingleStep Debugger **1-20**
Installing SingleStep Debugger **1-21**
Configuring SingleStep and Downloading Boot Code **1-22**

Setting Up Your Installation with Metrowerks CodeWarrior **1-31**

2 PIL Cosimulation

Overview of PIL Cosimulation **2-2**
Why Use Cosimulation? **2-2**
How Cosimulation Works **2-3**

Tutorial 1: Building and Running a PIL Cosimulation **2-5**
Before You Begin **2-5**
Hardware Connections **2-5**
The Demo Model **2-5**
Setting Up the Model **2-8**
Building PIL and Simulation Components **2-11**
Using the Demo Model In a PIL Cosimulation **2-14**

Tutorial 2: Modifying and Rebuilding the Controller **2-17**
Modifying the Controller **2-17**
Rebuilding the Controller and Cosimulating **2-19**

Tutorial 3: Using the Demo Model In Simulation **2-21**

PIL Target Summary **2-22**

Code Generation Options 2-22
Build Process Files and Directories 2-23
Restrictions 2-24

3 Generating Stand-Alone Real-Time Applications

Introduction 3-2
Deploying Generated Code 3-2

Tutorial: Creating a New Application 3-4
Before You Begin 3-4
The Example Model 3-6
Using the Pass-Through Option in Simulation 3-9
Generating Code 3-10
Downloading the Application to RAM via SingleStep/BDM 3-13
Downloading the Application to RAM via CAN 3-19

Downloading Boot and Application Code 3-23
RAM vs. Flash Memory 3-23
Overview of Memory Organization and the Boot Process 3-24
Downloading Boot Code 3-25
Downloading Application Code 3-27
Downloading Boot or Application Code via CAN Without Manual CPU Reset 3-32
Boot Code Parameters for CAN Download 3-33

Generating ASAP2 Files 3-35
Requirements and Limitations 3-35
ASAP2 File Generation Procedure 3-36
Data Acquisition (DAQ) List Configuration 3-38

Summary of the Real-Time Target 3-40
Code Generation Options 3-40
Requirements and Restrictions 3-42

4 Algorithm Export and Code Analysis Reporting

Algorithm Export Target **4-2**

Code Analysis Reporting **4-3**

Algorithm Export Target Summary **4-5**
Code Generation Options **4-5**
Restrictions **4-5**

5 Block Reference

The Embedded Target for Motorola MPC555 Block Libraries **5-2**
Using Block Reference Pages **5-2**

Blocks Organized by Libraries **5-4**
Embedded Target for Motorola MPC555 Library **5-4**
Data Type Support and Scaling for Device Driver Blocks **5-6**
Configuration Class Blocks **5-8**
CAN Message Blocks and CAN Drivers Libraries **5-9**

Alphabetical List of Blocks **5-11**

Preface

This section includes the following topics:

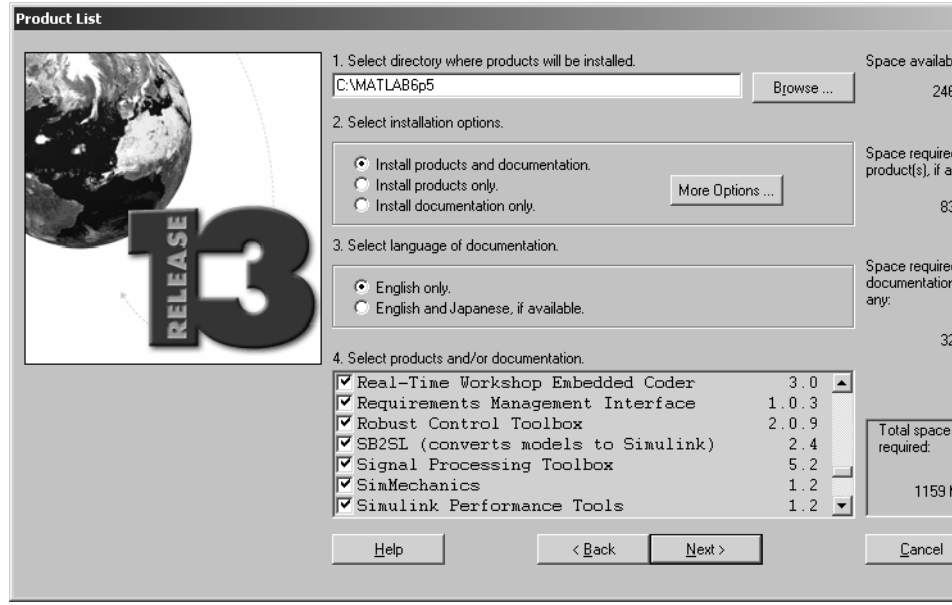
Installing the Embedded Target for Motorola MPC555 (p. vi)	Installation of the product.
Using This Guide (p. vii)	Suggested path through this document to get you up and running quickly with the Embedded Target for Motorola MPC555.
Embedded Target for Motorola MPC555 Demos (p. viii)	Hyperlinks to demo models that illustrate product features and how to use them.
Related Products (p. x)	Products required when using the Embedded Target for Motorola MPC555; also products that are especially relevant to the kinds of tasks you can perform with the Embedded Target for Motorola MPC555.
Typographical Conventions (p. xii)	Formatting conventions used in this document.

Installing the Embedded Target for Motorola MPC555

Your platform-specific MATLAB Installation guide provides all of the information you need to install the Embedded Target for Motorola MPC555.

Prior to installing the Embedded Target for Motorola MPC555, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you are permitted to install and use.

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



Using This Guide

We suggest the following path to get acquainted with the Embedded Target for Motorola MPC555 and gain hands-on experience with the features most relevant to your interests:

- Read Chapter 1, “Product Overview” in its entirety, paying particular attention to “Setting Up and Verifying Your Installation” on page 1-11.
- If you are interested in processor-in-the-loop (PIL) cosimulation, read Chapter 2, “PIL Cosimulation” to learn about the Embedded Target for Motorola MPC555 PIL target. Work through the “Tutorial 1: Building and Running a PIL Cosimulation” on page 2-5.
- If you are interested in using the device driver blocks supplied with Embedded Target for Motorola MPC555 and in deploying stand-alone, real-time applications on the MPC555, read Chapter 3, “Generating Stand-Alone Real-Time Applications.” Work through the “Tutorial: Creating a New Application” on page 3-4.
- Then, for in-depth information about the device drivers and other blocks supplied with Embedded Target for Motorola MPC555, see Chapter 5, “Block Reference.” It is particularly important to read “MPC555 Resource Configuration” on page 5-47, as the MPC555 Resource Configuration block is required to use most of the device driver blocks.
- See also “Embedded Target for Motorola MPC555 Demos” below.

Embedded Target for Motorola MPC555 Demos

We have provided a number of demos to help you become familiar with features of the Embedded Target for Motorola MPC555.

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

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

```
mpc555rt_led
```

Embedded Target for Motorola MPC555 Demos

Command	Demo Topic
`mpc555pil_fuelsys`	Model configured for building a Configurable Subsystem for use in PIL cosimulation. See "Tutorial 1: Building and Running a PIL Cosimulation" on page 2-5.
`mpc555rt_led`	Simple model demonstrating use of device driver blocks provided with the Embedded Target for Motorola MPC555 real-time target. See "Tutorial: Creating a New Application" on page 3-4.
`mpc555rt_io` and `mpc555rt_iohost`	Demonstration of MPC555 I/O device driver blocks and CAN support. A subsystem of the `mpc555rt_io` model runs on target hardware. The subsystem code communicates, via a CAN channel, with the `mpc555rt_iohost` model, running in Simulink®. This demo requires Vector-Informatik CAN hardware and drivers. See the instructions in the demo models Note: you can also use the `mpc555rt_io` model alone in simulation; this demonstrates the use of the pass-through feature of the device driver blocks.

Embedded Target for Motorola MPC555 Demos

Command	Demo Topic
`mpc555rt_ccp`	Demonstrates use of the CAN Calibration Protocol. See "CAN Calibration Protocol" on page 5-12.
`mpc555dd_fuelsys`	This model demonstrates the use of MPC555 I/O device driver blocks in simulation, software-in-the-loop simulation, and PIL cosimulation. Input and output signals are conditioned (with respect to data type and scaling) appropriately for use with the device driver blocks.

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Embedded Target for Motorola MPC555. They are listed in the table below.

The Embedded Target for Motorola MPC555 *requires* these products:

- MATLAB® 6.5 (Release 13)
- Simulink® 5.0 (Release 13)
- Stateflow® 5.0(Release 13) and Stateflow Coder
- Real-Time Workshop® 5.0 (Release 13)
- Real-Time Workshop Embedded Coder 3.0 (Release 13)

For more information about any of these products, see either

- The online documentation for that product, if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

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

Product	Description
Fixed-Point Blockset	Design and simulate fixed-point systems
MATLAB	The Language of Technical Computing
Real-Time Workshop	Generate C code from Simulink models
Real-Time Workshop Embedded Coder	Generate production code for embedded systems

Product	Description
Simulink	Design and simulate continuous- and discrete-time systems
Stateflow	Design and simulate event-driven systems
Stateflow Coder	Generate C code from Stateflow charts

Typographical Conventions

This manual uses some or all of these conventions.

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

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

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

1

Product Overview

This section contains the following topics:

Prerequisites (p. 1-2)	What you need to know before using the Embedded Target for Motorola® MPC555.
Introduction to the Embedded Target for Motorola MPC555 (p. 1-3)	Overview of the product and the use of the Embedded Target for Motorola MPC555 in the development process.
Hardware and Software Requirements (p. 1-9)	Hardware platforms supported by the product; development tools (e.g. compilers, debuggers) required for use with the product.
Setting Up and Verifying Your Installation (p. 1-11)	Overview of setting up your development tools and hardware to work with the Embedded Target for Motorola MPC555, and verifying correct operation.
Setting Up Your Target Hardware (p. 1-12)	Port connections and jumper settings.
Setting Target Preferences (p. 1-15)	Configuring environmental settings and preferences associated with the Embedded Target for Motorola MPC555.
Setting Up Your Installation with Diab Cross-Compiler and SingleStep Debugger (p. 1-20)	Configuring the Embedded Target for Motorola MPC555 for use with the Diab development tools.
Setting Up Your Installation with Metrowerks CodeWarrior (p. 1-31)	Configuring the Embedded Target for Motorola MPC555 for use with the Metrowerks CoderWarrior development tools.

Prerequisites

This document assumes you are experienced with MATLAB®, Simulink®, Stateflow®, 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 addition, if you want to understand and use the device driver blocks in the the Embedded Target for Motorola MPC555 library, you should have at least a basic understanding of the architecture of the MPC555. The Motorola *MPC555 Users Guide* is required reading. We recommend that you read the introduction to the processor and familiarize yourself with all the major subsystems of the MPC555.You can find this document at the following URL: `http://e-www.motorola.com/webapp/sps/library/prod_lib.jsp`.

Introduction to the Embedded Target for Motorola MPC555

The Embedded Target for Motorola MPC555 is an add-on product for use with the Real-Time Workshop Embedded Coder. It provides a complete and unified set of tools for developing embedded applications for the Motorola MPC555 processor.

Used in conjunction with Simulink, Stateflow, and the Real-Time Workshop Embedded Coder, the Embedded Target for Motorola MPC555 lets you

- Design and model your system and algorithms.
- Compile, download, run and debug generated code on the target hardware, seamlessly integrating with industry-standard compilers and development tools for the MPC555.
- Use cosimulation and rapid prototyping techniques to evaluate performance and validate results obtained from generated code running on the target hardware.
- Deploy production code on the target hardware.

Feature Summary

Simulation and Cosimulation

- Automatic S-function generation lets you validate your generated code in software-in-the-loop (SIL) simulation.
- Processor-in-the-loop (PIL) cosimulation lets you integrate generated code, running on the target processor, into your simulation.
- SIL and PIL code components are generated by the Real-Time Workshop Embedded Coder. These simulation components are in the same compact and efficient format as the production code generated for final deployment.

Production Code Generation

- The Real-Time Workshop Embedded Coder generates production code for use on the target MPC555 microcontroller.
- The Real-Time Workshop Embedded Coder generates project or makefiles for popular cross-development systems:
  - Wind River Systems Diab cross-compiler

  - Metrowerks® CodeWarrior®
- Debugger support:
  - Wind River Systems SingleStep debugger
  - Metrowerks CodeWarrior debugger

Code and Performance Analysis

Web-viewable code generation report includes

- Analysis of RAM/ROM usage and other variables
- Analysis of code generation options used, with optimization suggestions
- Hyperlinks to all generated code files
- Hyperlinks from generated code to source model in Simulink

Applications Development and Rapid Prototyping

- Generation of real-time, stand-alone code for MPC555
- Scheduler and time functions for single- rate or multirate real-time operation
- CAN-based loader for download of generated code to RAM or flash memory
- CAN-based host-target communications for non-real-time retrieval of data on host computer

Device Driver Support

- The Embedded Target for Motorola MPC555 Library provides device driver blocks that let your applications access on-chip resources. The I/O blocks support the following features of the MPC555:
  - Pulse width modulation (PWM) generation via the Modular Input/Output Subsystem (MIOS) PWM unit.
  - Analog input via the Queued Analog-to-Digital Converter (QADC64)
  - Digital input and output via the MIOS
  - Digital input via the QADC64
  - Frequency and pulse width measurement via the MIOS Double Action Submodule (MDASM)

  - Transmit or receive Controller Area Network (CAN) messages via the MPC555 TouCAN modules
  - Utility blocks such as a watchdog timer
- Device driver blocks support a *pass-through* option. The pass-through option lets you leave your device driver blocks in your model during simulation. You can then provide a Simulink signal to use in place of the actual device driver signal.

CAN Support

- Transmit or receive CAN messages via the MPC555 TouCAN modules.
- CAN Drivers (Vector) library provides blocks for configuring and connecting to Vector-Informatik CAN hardware and drivers.
- The CAN Message Blocks library includes blocks for transmitting, receiving, decoding, and formatting CAN messages. It also supports message specification via the Vector-Informatik CANdb standard.

Code Validation and Performance Analysis

Code Validation. Since signal data is available to Simulink during each sample interval in a PIL simulation, you can observe signal data on Scope blocks or other Simulink signal viewing blocks. You can also store signal data to MAT-files via To File blocks. To validate the results obtained by the generated code running on the target processor, you can compare these files to results obtained using a normal Simulink plant/controller simulation.

Determining Code Size. In control design it is critical to ensure that the size of the generated code does not exceed physical limitations of RAM and ROM. The Embedded Target for Motorola MPC555 automatically produces a code generation report that displays the RAM usage and ROM size of the generated code.

This capability is useful when selecting which code generation optimizations will be used. After determining the size of the required RAM and ROM, you can consider which code generation optimizations to use, and consider modifications to the modeling style.

Applications for the Embedded Target for Motorola MPC555

The Embedded Target for Motorola MPC555 provides targets that support three application scenarios.

- Processor-in-the-loop cosimulation (PIL) target
- Real-time execution and rapid prototyping (RT) target
- Algorithm export (AE) target

In the sections that follow, we summarize typical applications and the tasks you will need to perform for each; we also provide links to the relevant documentation.

Processor-in-the-Loop

The processor-in-the-loop (PIL) target lets you run a cosimulation of a closed-loop Simulink model for the purpose of code validation and analysis. When running a PI cosimulation, you use a closed-loop model with two major components: a plant model and a controller. The plant model may contain any Simulink blocks including a combination of continuous-time and discrete-time blocks. The controller must not include any continuous-time blocks, since this component is used for code generation in the Embedded-C format of the Real-Time Workshop Embedded Coder.

To get started with the PIL target, see "Tutorial 1: Building and Running a PIL Cosimulation" on page 2-5. The tutorial covers the following topics:

- Target preferences setup
- Creating the controller subsystem
- Connecting the plant model to the controller
- Selecting the PIL target
- Generating the ERT S-function and the corresponding library block
- Inserting the S-function back into the closed-loop model
- Automatic downloading of generated code with
  - SingleStep debugger and a Background Debug Mode (BDM) port connector
  - CodeWarrior and a BDM connector
- Running a PIL cosimulation

You may also be interested in generating code analysis information from your PIL target build. See “Code Analysis Reporting” on page 4-3 for details.

Real-Time Execution and Rapid Prototyping

The Embedded Target for Motorola MPC555 real-time target enables you to use your controller block diagram in real time to perform embedded control. With this target, you can add I/O blocks for the MPC555 to your controller subsystem, generate and build code, download to the target, and run the generated C code.

When you first begin using the RT target, see “Tutorial: Creating a New Application” on page 3-4, which demonstrates the following topics through the use of a simple model with a device driver:

- Toolchain setup using target preferences
- Creating the controller subsystem
- Connecting the plant model to the controller
- Adding the MPC555 Resource Configuration block to your subsystem
- Adding I/O device drivers from the Embedded Target for Motorola MPC555 library
- Selecting the RT target
- Generating code for real-time
- Downloading code with
  - SingleStep debugger and a BDM connector
  - CAN
- Running the generated in real-time code

You may also be interested in generating code analysis information from your RT target build. See “Code Analysis Reporting” on page 4-3 for details.

Algorithm Export

The Embedded Target for Motorola MPC555 algorithm export (AE) target enables you generate code for your controller subsystem and build the code as a stand-alone executable for use on the MPC555. The difference between the AE and the PIL target is that the AE target eliminates all extraneous code (such as serial communications code) used for cosimulation, and also eliminates any real-time interrupts. The AE target therefore generates code

only for the basic controller subsystem (e.g. algorithm code). You can then modify or customize this code for your own special purposes.

In contrast, the RT target provides turnkey code including interrupt service routines, driver code, and underlying initialization code for the MPC555. Depending upon your particular application, you may find it more valuable to begin with the AE target baseline, and extend this environment for your own use.

The AE target is documented in “Algorithm Export Target” on page 4-2.

Like the PIL and RT targets, the AE target supports generation of code analysis information. See “Code Analysis Reporting” on page 4-3 for details.

Hardware and Software Requirements

Host Platform

The Embedded Target for Motorola MPC555 supports only the PC platform.

Hardware Requirements

Programs generated by the Embedded Target for Motorola MPC555 can run on any Electronic Control Unit (ECU) that is based on the MPC555 processor.

In this document, however, we assume that you are working with the Phytec phyCORE-MPC555 development board, and we document specific settings and procedures for use with the Phytec phyCORE-MPC555 board, in conjunction with specific cross-development environments.

If you use a different development board, you may need to adapt these settings and procedures for your development board.

Software Requirements

See "Related Products" for information on MathWorks products required to use Embedded Target for Motorola MPC555.

In addition to the required MathWorks software, a supported cross-development environment is required. The Embedded Target for Motorola MPC555 currently supports the cross-development tools listed below; please read carefully the limitations noted:

- Wind River Systems Diab cross-compiler, and Wind River Systems SingleStep debugger (version 7.6.2)

  The full feature set (PIL, RT, and AE targets) is supported when used with the Diab cross-compiler and SingleStep debugger.
- Metrowerks CodeWarrior for Embedded PowerPC (version 6.0 or later)

  Due to current limitations of the Embedded Target for Motorola MPC555, (not of Metrowerks CodeWarrior), only the PIL and AE targets are supported for use with Metrowerks CodeWarrior. The RT target is not currently supported for use with Metrowerks CodeWarrior.

Before using the Embedded Target for Motorola MPC555 with any of the above cross-development tools, please be sure to read and follow the instructions in “Setting Up and Verifying Your Installation” on page 1-11.

Setting Up and Verifying Your Installation

The next sections describe how to configure your development environment (compiler, debugger, etc.) for use with the Embedded Target for Motorola MPC555 and verify correct operation. The initial configuration steps are described in the following sections:

- "Setting Up Your Target Hardware" on page 1–12
- "Setting Target Preferences" on page 1–15

After completing these steps, proceed to the section appropriate to your development environment:

- If you are using the Diab cross-compiler, see "Setting Up Your Installation with Diab Cross-Compiler and SingleStep Debugger" on page 1–20.
- If you are using the Metrowerks CodeWarrior Integrated Development Environment (IDE), see "Setting Up Your Installation with Metrowerks CodeWarrior" on page 1–31.

Setting Up Your Target Hardware

In this document, we assume that you are working with the Phytec phyCORE-MPC555 development board. This section describes the required connections and jumper settings for the board.

Phytec Communications Ports

Before you begin working with the Embedded Target for Motorola MPC555, you should set up your phyCORE-MPC555 board and connect it to your host computer. The hardware setup is described in the *phyCORE-MPC555 Quickstart Instructions* manual on your Phytec Spectrum CD. See the "Interfacing the phyCORE-MPC555 to a Host PC" section of the "Getting Started" chapter.

In this document, we assume that you have connected your phyCORE-MPC555 board to the same serial (COM1) and parallel (LPT1) ports described in the *phyCORE-MPC555 Quickstart Instructions*.

Phytec Jumper Settings

The Embedded Target for Motorola MPC555 (PIL and RT targets) has been tested by the MathWorks with the Phytec phyCORE-MPC555 board, using the jumper settings indicated in the table below.

For jumper locations and pin numbers, see the *phyCORE-MPC555 Quickstart Instructions* manual.

The following tables summarizes the correct jumper settings to use when your host PC is connected to the on-board BDM port, or via Wiggler, Raven, or Blackbird devices.

Jumper	Description	Raven or Blackbird	Wiggler	On-Board BDM
JP13	CAN A bus termination	Closed (apply 120 Ohm termination)	as Raven	as Raven
JP14	CAN B bus termination	Closed (apply 120 Ohm termination)	as Raven	as Raven

Jumper	Description	Raven or Blackbird	Wiggler	On-Board BDM
JP15	Select boot memory	1+2 (boot from internal flash memory)	as Raven	as Raven
JP3	Connect push button to different reset signals	1+2 (/HRESIN connected to push button)	as Raven	as Raven
JP18	Connect interrupt to push button	Default 1+2	as Raven	as Raven
JP17	Connect /HRESET or /SRESET to external BDM interface logic	1+2 (/HRESET connected to BDM interface logic)	as Raven	as Raven
JP1	On-board BDM reset signal connection	Open	as Raven	3+4 closed
JP5,JP6,JP7, JP8,JP9	Jumpers relating to on-board BDM	Open	as Raven	All closed
JP2	Power supply for external BDM	Open (unless BDM device requires supply voltage from development board)	1+2 closed	1+2 closed
JP10	Connect one of the LEDs to supply voltage	Closed	as Raven	as Raven
JP11	Connect 5V supply voltage	Closed	as Raven	as Raven
JP12	Connect 3V3 supply voltage	Closed	as Raven	as Raven

Jumper	Description	Raven or Blackbird	Wiggler	On-Board BDM
JP4	Programming of Internal MPC555 Flash internal memory enabled	Closed	as Raven	as Raven
JP16	Use J5 as source of Hard-Reset-Configuration	Open	as Raven	as Raven

Note The MPC555 flash memory has a limited lifetime, which is shortened each time the flash memory is programmed. To extend product life, Motorola recommends using flash programming only when necessary.

After setting up your phyCORE-MPC555 board, you must set environment variables associated with the Embedded Target for Motorola, as described in the next section.

Setting Target Preferences

This section describes environmental settings associated with the Embedded Target for Motorola MPC555. These settings, which persist across MATLAB sessions and different models, are referred to as *target preferences*. Target preferences let you specify the location of your MPC555 cross-compiler, the communications port to be used for downloading code, and other parameters affecting the generation, building, and downloading of code.

Target preferences are stored in data objects called *target preference objects*. These objects let you create different named objects for different sets of preferences. We provide a "factory" preference object with a default configuration. Table 1-1, Target Preferences Summary, summarizes the preference parameters and their defaults.

At any time, only one *active* target preference object exists. We will refer to this as the *active target preferences*. You can instantiate as many target preference objects as you require, but only the active target preferences influence the operation of the Embedded Target for Motorola MPC555.

Note Until you have invoked `settargetprefs`, there are no active target preferences. See "settargetprefs Command" on page 1-16.

MATLAB Commands for Working with Target Preferences

You can create, modify, activate, or examine target preference objects via several MATLAB commands.

gettargetprefs Command

The `gettargetprefs` command retrieves the currently active target preferences. Note that until you have invoked `settargetprefs`, there are no active target preferences. If there are no active target preferences, `gettargetprefs` returns the factory default values.

The following example retrieves the active preferences into the target preference object `tpObj`:

```
>> tpObj = gettargetprefs('mpc555dk')
```

```
tpObj =

   target.mpc555dkprefs
```

settargetprefs Command

The `settargetprefs` command lets you create, clear, or modify your target preference settings. Until you have invoked `settargetprefs`, there are no active target preferences.Valid syntax for the `settargetprefs` command is shown below.

To reset to the factory default values for the target preferences:

```
settargetprefs('mpc555dk','factory')
```

To make a target preference object active:

```
settargetprefs('mpc555dk',tpObj)
```

To modify a property value for the target preferences:

```
settargetprefs('mpc555dk','PropertyName','PropertyValue')
```

To clear the currently active target preferences:

```
settargetprefs('mpc555dk',[])
```

In the sample MATLAB session below. the target preferences are set to the factory default values. The target preferences are then returned to an object, `tpObj`:

```
>> settargetprefs('mpc555dk','factory')
>> tpObj = gettargetprefs('mpc555dk')

tpObj =

   target.mpc555dkprefs
```

display Command

The target preference object implements the `display` method. The `display` command allows you to retrieve a target preference structure from a target preference object. When the target preference object is already defined in your workspace, you can use the `display` command to view your current settings, as shown in the following example:

```
>> display(tpObj)

tpObj =

        TargetCompiler: 'CodeWarrior'
    TargetCompilerPath: 'D:\Applications\Metrowerks\CodeWarrior\Bin'
           CommTimeout: '4'
        TargetDebugger: 'SingleStep'
     TargetDebuggerExe: 'D:\Applications\SingleStep\sds762\cmd\bdmp58.exe'
         CommHostPort: 'COM1'
       CommTargetPort: 'COM1'
         CommBaudRate: '57600'
       PlugInIncludes: '  '
           PlugInLibs: '  '
     PlugInTLCIncludes: '  '
```

Table 1-1: Target Preferences Summary

Preference Name	Description	Default Value
`TargetCompiler`	Choice of MPC555 cross-development system: `'Diab'` or `'CodeWarrior'`	'Diab'
`TargetCompilerPath`	Path to your MPC555 cross-development system.	You must localize this to suit your PC.
`CommTimeout`	Timeout value (in seconds) for the serial communications port.	`4`
`CommHostPort`	Host serial port for host/target communications	`'COM1'`
`CommTargetPort`	Target board serial port for host/target communications	`'COM1'`
`CommBaudRate`	Baud rate (in bps) for host/target communications	`57600`
`TargetDebugger`	Debugger to be invoked by PIL target. Currently used only when `TargetCompiler = 'Diab'`; ignored when `TargetCompiler = 'CodeWarrior'`	`'SingleStep'`
`TargetDebuggerExe`	Fully qualified path and name for the debugger executable. The drive designated in the path must be either an actual hard drive on your PC, or a mapped drive. Do not use a UNC. Note: this field is ignored when `TargetCompiler = 'CodeWarrior'`	You must localize this to suit your PC.

Editing Target Preferences via the Property Inspector Window

The **Property Inspector** window provides a graphical user interface for editing Target Preferences objects. The **Property Inspector** window does not have **Apply** or **Save** buttons, so you must still use the commands described in "Setting Target Preferences" on page 1-15 to make changes to target preferences persistent.

To edit the target preferences via the **Property Inspector** window:

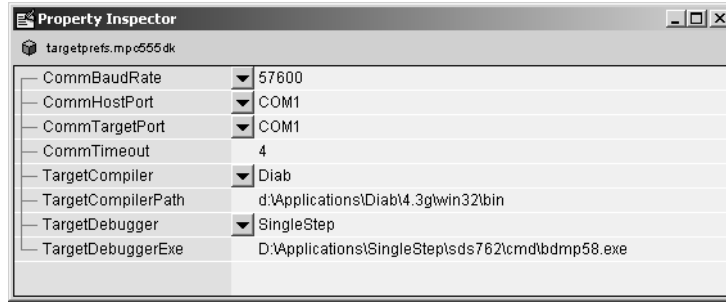
1. Retrieve the target preferences to an object:

```
tpObj = gettargetprefs('mpc555dk')
```

2. Invoke the `edit` method of the object.

```
edit(tpObj)
```

The `edit` method opens the **Property Inspector** window, as shown below.



3. Modify the fields you want to change.

4. Make your changes persistent by using the object to set your active preferences.

```
settargetprefs('mpc555dk',tpObj)
```

Setting Up Your Installation with Diab Cross-Compiler and SingleStep Debugger

To use the Embedded Target for Motorola MPC555 with the Diab cross-compiler, you need the following:

- Configurable Subsystem block phyCORE-MPC555 Rapid Development Target, including the phyCORE-MPC555 development board, DB-25 parallel printer port cable, serial cable, and AC to DC adapter.
- Wind River Systems Diab cross-compiler
- Wind River Systems SingleStep debugger (version 7.6.2).

Install Diab Cross-Compiler

If you have not already done so, install the Diab cross-compiler, following the installation instructions provided by Wind River Systems. When the installer prompts for **Components**, select `Diab C Compiler`. When the installer prompts for a **Target**, select `PowerPC` and all related components.

You do not need to set a default processor or other compiler defaults. During the code generation and build process, the Embedded Target for Motorola MPC555 will generate a makefile that sets the correct options.

You will need to note the path to the installed compiler in order to configure your target preferences (see “Set Target Preferences for Diab and SingleStep” on page 1-21).

Install SingleStep Debugger

The SingleStep debugger, in conjunction with the Embedded Target for Motorola MPC555, lets you download, run and debug generated code. Installing and configuring SingleStep for this purpose is a somewhat elaborate procedure, so we have documented it in a separate section, “Installing SingleStep Debugger” on page 1-21.

You will need to note the path to the installed SingleStep debugger in order to configure your target preferences (see “Set Target Preferences for Diab and SingleStep” on page 1-21).

Set Target Preferences for Diab and SingleStep

After installing your development tools, the next step is to configure your target preferences for the Diab cross-compiler and SingleStep debugger. (Please read "Setting Target Preferences" on page 1–15, if you have not yet done so.) The default target preferences are configured for Diab and SingleStep, so set your preferences to the "factory" values as follows:

```
settargetprefs('mpc555dk','factory')
```

Then, change `TargetCompilerPath` and `TargetDebuggerExe` to the appropriate paths to your Diab and SingleStep installations. For example:

```
settargetprefs('mpc555dk','TargetCompilerPath','D:\Apps\Diab\4.3g\win32\bin')
settargetprefs('mpc555dk','TargetDebuggerExe',...
'D:\Apps\singlestep\sds762\cmd\bdmp58.exe')
```

For most purposes, the other target preferences fields can be left at their defaults.

Installing SingleStep Debugger

The next sections describe how to install and configure the SingleStep for use with the Embedded Target for Motorola MPC555, and how to use SingleStep to download boot load code into flash memory on a Phytec phyCORE-MPC555 board.

SingleStep Installation

If you have not already done so, you should install the SingleStep debugger and confirm its operation with your phyCORE-MPC555 board before proceeding with this section. You should select the `SStep Professional Suite (MPC5xx)` option during installation. If necessary, please consult your SingleStep documentation.

For SingleStep 7.6.2, you must obtain the following files from Wind River Systems and apply the updates they contain:

- `pcflash11_29_00.zip`

  Apply this update first. See the accompanying file, `pcflash11_29_00.txt`.
- `pcflash3_15_01.zip`

  Apply this update second. See the accompanying file, `pcflash3_15_01.txt`.

Configuring SingleStep and Downloading Boot Code

The following sections explain how to configure the SingleStep debugger and download boot load code into flash memory. After configuring SingleStep, you will also be able to use the debugger directly to debug generated programs.

Purpose of Flash Memory Boot Code

When reading this section, you may want to refer to the internal memory map of the MPC555 in section 1.3 of the *MPC555 Users Guide*. You can find this document at the following URL.
`http://e-www.motorola.com/webapp/sps/library/prod_lib.jsp`

To run generated code from the flash memory, you must load the first 32K flash sector with boot code. The primary purpose of the boot code is to load and start application code when the board is powered on or reset. The boot code also acts as a download agent that downloads generated code into flash memory via CAN.

You will use SingleStep debugger to load the boot code. Once the boot code is loaded into flash memory, you can download code to the processor entirely over a CAN network without needing to use SingleStep or a BDM connection.

For more information on downloading generated code into flash memory via CAN see “Downloading Boot and Application Code” on page 3-23.

Note The SingleStep options and user interface screens discussed below are based on SingleStep version 7.6.2 and may differ from your installed version of SingleStep, or with future versions of SingleStep. The MathWorks provides the configuration information below only as a convenience. To resolve questions or difficulties with SingleStep, refer to the SingleStep documentation, or contact Wind River Systems.

Configure Shortcuts to SingleStep

In order to start SingleStep with the proper options, we recommend that you configure two shortcuts to SingleStep as follows.

1 The SingleStep installer will have installed a shortcut named `SingleStep On Chip (MPC5xx)` in your system’s **Start/Programs/SingleStep 7.6.2**

menu. Locate this shortcut file and make a copy of it on your desktop. Rename the copy to `SingleStep On Chip (MPC5xx) for Flash`.

2 Right-click on the `SingleStep On Chip (MPC5xx) for Flash` shortcut file and edit its **Target** property to read as follows

```
ssteproot\cmd\bdmp58.exe -P -r
matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\app_startup\memcfg.dbg -S
matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\app_startup\mw_flash.wsp
```

where *`ssteproot`* is the installed SingleStep directory and *`matlabroot`* is the MATLAB root directory.

You will use the `SingleStep On Chip (MPC5xx) for Flash` shortcut when downloading boot or application code to flash memory via SingleStep.

3 Make a copy of the `SingleStep On Chip (MPC5xx) for Flash` shortcut file you created in step 2 and rename it to `SingleStep On Chip (MPC5xx) for RAM`. Edit its **Target** property to read as follows

```
ssteproot\cmd\bdmp58.exe -P -S
matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\app_startup\mw_ram.wsp
```

where *`ssteproot`* is the installed SingleStep directory and *`matlabroot`* is the MATLAB root directory.

You will use this `SingleStep On Chip (MPC5xx) for RAM` shortcut when downloading code to RAM via SingleStep.

Configure phyCORE-MPC555 Jumpers

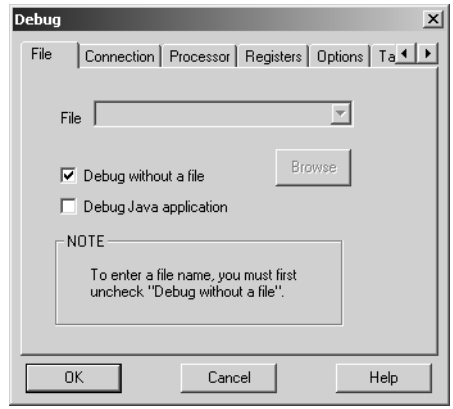
Make sure that the jumpers on the phyCORE-MPC555 board are set as described in “Phytec Jumper Settings” on page 1–12. The correct jumper configuration is required when downloading to flash memory via the BDM port. Any other jumper settings may cause downloading to flash memory to fail, or cause other problems when operating with the Embedded Target for Motorola MPC555. For additional information on jumper settings, consult the phyCORE-MPC555 documentation and the SingleStep manual.

Configure SingleStep Parameters

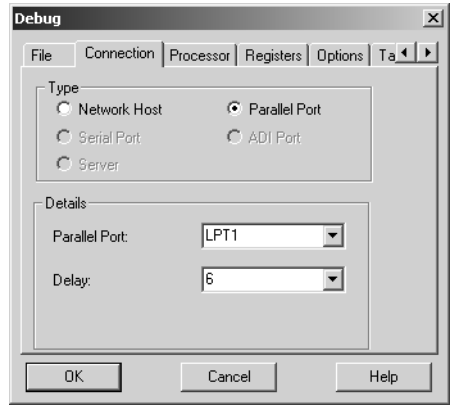
The next step is to configure the SingleStep options as follows:

1 Start SingleStep using the `SingleStep On Chip (MPC5xx) for Flash` shortcut you created previously. The **Debug** dialog box opens.

2 Click on the **File** tab. Select the **Debug without a file** check box, as shown.

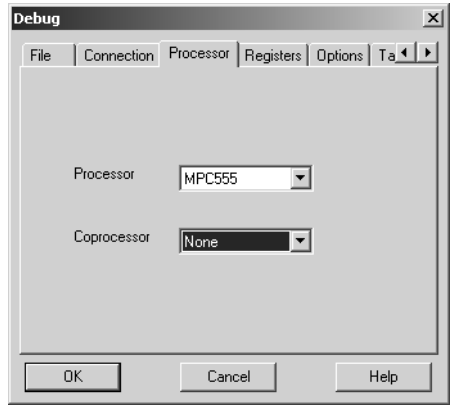


3 Click on the **Connection** tab. Choose parallel port or network settings appropriate to the physical connection you will be using between your PC and PhyCORE-MPC555 board. In the figure below, connection options are configured for the parallel port `LPT1`, with a **Delay** setting of `6`.

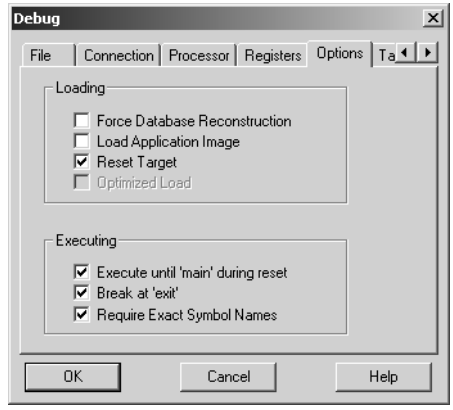


Note Depending on the BDM probe you are using, you may need to experiment with the **Delay** setting.

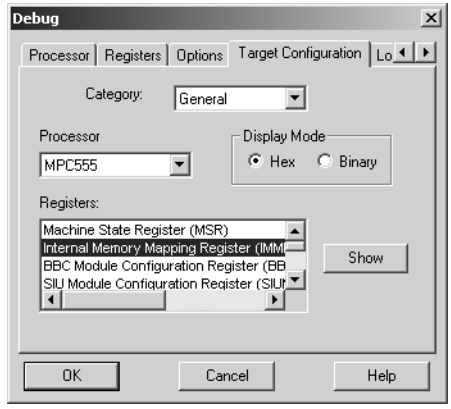
4 Click on the **Processor** tab. Confirm that the MPC555 is selected in the **Processor** menu, as shown.



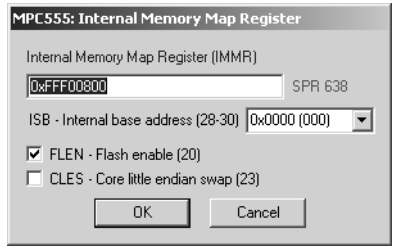
5 Click on the **Options** tab. Make sure that the **Reset Target** check box is selected, as shown.



6 Click on the **Target Configuration** tab and select `General` from the **Category** menu. Then select `Internal Memory Mapping Register` from the **Registers** menu, as shown.



7 Click on the **Show** button to open the **Internal Memory Mapping Register** dialog box. Confirm that the **FLEN-flash enable** check box is selected as shown below.

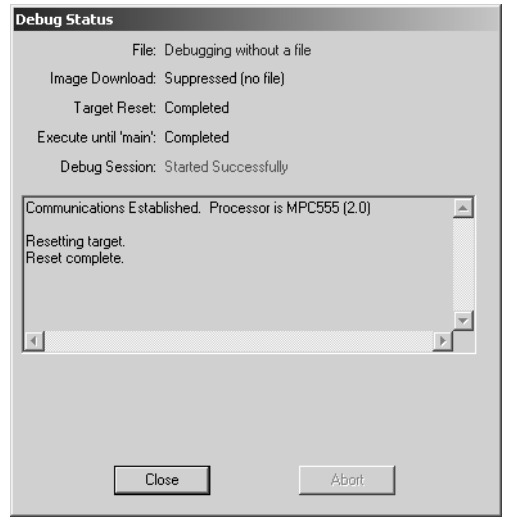


8 Then click **OK** to close the **Internal Memory Mapping Register** dialog box and return to the **Debug** dialog box.

Note Make sure to click **OK**, not **Cancel**, or SingleStep may use settings other than those shown in the dialog box.

---

9 Click **OK**. SingleStep attempts to connect to the processor, and displays a **Debug Status** window. This figure shows the **Debug Status** window after a successful connection.



If you see error messages, consult the SingleStep documentation to troubleshoot the connection, or contact Wind River Systems for technical support.

10 Click **Close** to dismiss the **Debug Status** window.

Download Boot Code to Flash Memory

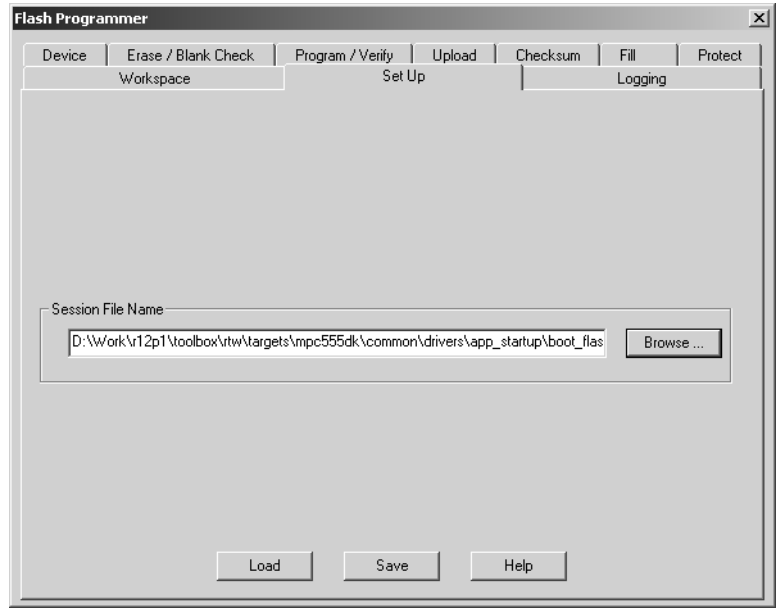
The next step is to download the boot code to flash memory, using the SingleStep **Flash Programmer** dialog box.

Note Be sure to apply the SingleStep updates described in "SingleStep Installation" on page 1–21. If you do not update SingleStep properly, the **Flash Programmer** dialog box will not operate correctly.

---

1 Activate the main SingleStep window. If you do not see a **Flash** button in the toolbar, select **Tools** from the **ToolBars** menu.

2 Click on the **Flash** button on the toolbar. The **Flash Programmer** dialog box opens. Click on the **Set Up** tab. This figure shows the **Set Up** pane.



3 Enter the **Session File Name** as

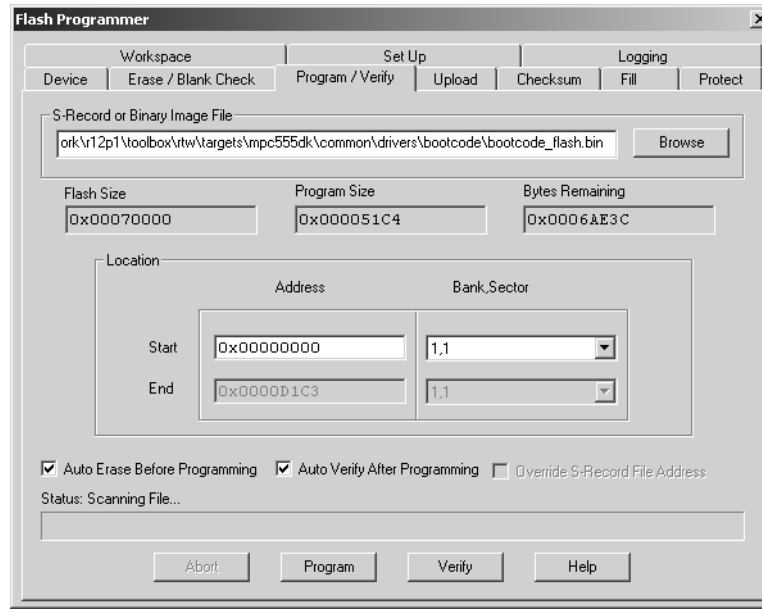
```
matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\app_startup
\boot_flash.pts
```

4 Click the **Load** button. After a brief interval, a message box appears, indicating that the load has completed. Click **OK** to dismiss the message box.

5 Click on the **Program/Verify** tab. On the **Program/Verify** pane of the dialog, enter the **S-Record or Binary Image File** as

```
matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\bootcode\
bootcode_flash.bin
```

Also select the **Auto Erase Before Programming** and **Auto Verify After Programming** options.



6 Click the **Program** button. The boot code is downloaded. During downloading, a number of progress messages are displayed in the **Status** panel at the bottom of the dialog box.

7 Upon completing the download process, SingleStep displays a message box indicating successful completion. Click **OK** to dismiss the message box. Then, close the **Flash Programmer** dialog box. Do not save changes to the **Flash Programmer** when the **Save** dialog appears.

8 If flash programming fails, you should

- Check that all jumpers are set correctly as described in “Configure phyCORE-MPC555 Jumpers” on page 1-23.
- Quit SingleStep and repeat the entire procedure “Configure SingleStep Parameters” on page 1-23. Make sure, in step 8, that you click **OK** (not **Cancel**) when closing the **Internal Memory Mapping Register** dialog box. Otherwise, SingleStep may use settings other than those shown in the dialog box.

Your PhyCORE-MPC555 board is now ready to receive application code via CAN download. It is no longer necessary to download the code via the BDM device. However, if you prefer to download the application via BDM, you can do so as described in “Downloading the Application to RAM via SingleStep/BDM” on page 3-13.

Setting Up Your Installation with Metrowerks CodeWarrior

To use the Embedded Target for Motorola MPC555 with Metrowerks CodeWarrior, you need the following:

- Configurable Subsystem block phyCORE-MPC555 Rapid Development Target, including the phyCORE-MPC555 development board, DB-25 parallel printer port cable, serial cable, AC to DC power adapter, and Configurable Subsystem block Spectrum CD.

  You do not need to install anything from the Configurable Subsystem block Spectrum CD except the `Blinky` test program that you will use to verify your installation.
- Metrowerks CodeWarrior for Embedded PowerPC (version 6.0 or later) CD (or network access to the Metrowerks CodeWarrior for Embedded PowerPC installer). Do not use a version of Metrowerks CodeWarrior for Embedded PowerPC that is earlier than version 6.0.

Install Metrowerks CodeWarrior IDE

The first step is to install the Metrowerks CodeWarrior IDE:

1. If you have previously installed a version of Metrowerks CodeWarrior for Embedded PowerPC that is earlier than version 6.0, uninstall it.

2. Install CodeWarrior for Embedded PowerPC version 6.0 using the setup program provided on your Metrowerks CodeWarrior CD (or on your network). Run `Setup.exe` and follow the prompts.

Configure Metrowerks CodeWarrior Debugger

The next step is to configure the CodeWarrior debugger to communicate with the phyCORE-MPC555 board over the parallel port:

1. Start the Metrowerks CodeWarrior IDE. From the **Edit** menu, open the **IDE Preferences** dialog box. In the **IDE Preference Panels** pane, click on the plus sign next to **Debugger**.

2. A list of choices opens below **Debugger**. Select `Remote Connections`. The **Remote Connections** panel is displayed on the right.

3 Select `MPC555DK Wiggler` from the list in the **Remote Connections** panel.

   If no `MPC555DK Wiggler` configuration exists, create one as follows:

   a Click the **Add...** button. The **New Connection** configuration dialog box opens.

   b Set the **Name** property to `MPC555DK Wiggler`.

   c Set the **Debugger** property to `EPPC MSI Wiggler`.

   d Set the **Connection Type** property to `Parallel`.

   e Set the **Connection Port** property to match the port to which you have connected your phyCORE-MPC555 board (the default is `LPT1`).

   f Set the **Speed** property to `1`.

   g Set the **FPU Buffer Address** property to `0xFFF00000`.

   h Click **OK** and skip to step 5.

4 If a `MPC555DK Wiggler` exists, click the **Change...** button. The `MPC555DK Wiggler` configuration dialog box opens. By default, the **Parallel Port** property is set to `LPT1`. If you have connected your phyCORE-MPC555 board to a different port, change the **Parallel Port** setting accordingly. Then click **OK** to close the **MSI Wiggler** configuration dialog box.

5 Click **Apply** and close the **IDE Preferences** dialog box.

Set Target Preferences

The next step is to configure your target preferences for Metrowerks CodeWarrior. (Please read "Setting Target Preferences" on page 1–15, if you have not yet done so.) The relevant preferences are

- `TargetCompiler`: set to `'CodeWarrior'`.
- `TargetCompilerPath`: change this to the appropriate path to your CodeWarrior installation.
- Note that when using CodeWarrior, you do not have to specify the `TargetDebugger` preference. When required, the build process will automatically invoke the CodeWarrior debugger.

For most purposes, the other target preferences fields can be left at their defaults.

Run Test Program (Optional)

Optionally, to verify your CodeWarrior setup, you may want to download and run a simple test program on the phyCORE-MPC555 board. We will use the `Blinky` test program provided on the Configurable Subsystem block Spectrum CD. The `Blinky` test program is described in the "Getting Started" chapter of the *phyCORE-MPC555 Quickstart Instructions* manual.

1. Copy the directory `/PhyBasic/pcMPC555/Demos/Blinky` to your hard drive. Open the `/Blinky` directory and open the project file `Blinky.mcp`.

2. CodeWarrior will prompt that the project is read-only and needs to be converted to the current format. Click **OK** to allow conversion to proceed.

3. To set the debugger configuration of the project, select **ram Settings...** from the **Edit** menu. A dialog box titled **ram Settings** is displayed. In the left pane, click on the minus sign next to **Debugger**.

4. A list of choices opens below **Debugger**. Select `Remote Debugging`.The **Remote Debugging** panel is displayed on the right.

5. Select `MPC555DK Wiggler` from the **Connection** pop-up menu in the right pane.

6. Select **EPPC Debugger Settings**. The **EPPC Debugger Settings** panel is displayed on the right.

7. Select the **Use Target Initialization File** check box in the right pane. Then use the **Browse** button to select the following initialization file:

   ```
   phytecCD\PhyBasic\pcMPC555\Tools\PowerPC_EABI_Support\Initialization_Files\Bdm\
   phyCORE-MPC555_BDM_init.cfg.
   ```

8. Click **Apply** and then **OK** to close the dialog box.

9. The project code is already compiled and linked. To download and execute it on the target system, select **Debug** from the **Project** menu. After the code downloads, you should observe two LEDs blinking alternately on the phyCORE-MPC555 board.

   You can now break and resume execution, set breakpoints, single-step through code, and perform other debugger functions.

10 When finished, select **Kill** from the **Debug** menu to terminate the program on the target hardware.

You have now verified your installation and are ready to begin working with the Embedded Target for Motorola MPC555. We suggest you now turn to Chapter 2, “PIL Cosimulation” to get hands-on experience with processor-in-the-loop simulation via the Embedded Target for Motorola MPC555 and Metrowerks CodeWarrior.

2

PIL Cosimulation

This section includes the following topics:

Overview of PIL Cosimulation (p. 2-2)	Basic concepts you will need to know to use cosimulation effectively in your design process.
Tutorial 1: Building and Running a PIL Cosimulation (p. 2-5)	A hands-on, step-by-step introduction to cosimulation with the PIL target, using a plant/controller demonstration model.
Tutorial 2: Modifying and Rebuilding the Controller (p. 2-17)	This tutorial shows you how to use the PIL target's single-model development methodology to make iterative changes to a controller subsystem.
Tutorial 3: Using the Demo Model In Simulation (p. 2-21)	In addition to building code suitable for cosimulation, the PIL target builds components you can use in closed-loop and software-in-the-loop (SIL) simulations. This tutorial shows you how to use these components.
PIL Target Summary (p. 2-22)	Summary of code generation options of the PIL target; restrictions and limitations of the PIL target.

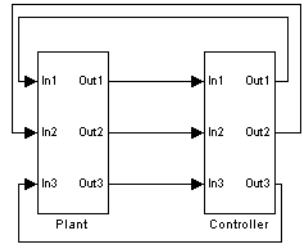
Overview of PIL Cosimulation

The Embedded Target for Motorola MPC555 supports *processor-in-the-loop* (PIL) *cosimulation*, a technique that is designed to help you evaluate how well a candidate control system operates on the actual target processor selected for the application.

The Embedded Target for Motorola MPC555 (processor-in-the-loop) target is an extended version of the embedded real-time (ERT) target configuration, designed specifically for PIL cosimulation. We will refer to this target as the *PIL target*.

Why Use Cosimulation?

PIL cosimulation is particularly useful for simulating, testing and validating a controller algorithm in a system comprising a *plant* and a *controller*. In classic closed-loop simulation, Simulink and Stateflow model such a system as two subsystems and the signals transmitted between them, as shown in this block diagram.



Your starting point in developing a plant/controller system is to model the system as two subsystems in closed-loop simulation. As your design progresses, you can use Simulink external mode with standard Real-Time Workshop targets (such as GRT or ERT) to help you model the control system separately from the plant.

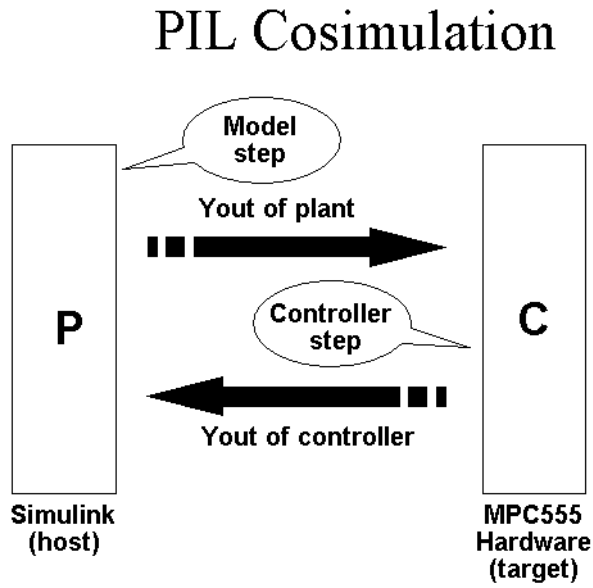
However, these simulation techniques do not help you to account for restrictions and requirements imposed by the hardware. When you finally reach the stage of deploying controller code on the target hardware, you may need to make extensive adjustments to the controller system. Once these

adjustments are made, your deployed system may diverge significantly from the original model. Such discrepancies can create difficulties if you need to return to the original model and change it.

PIL cosimulation addresses these issues by providing an intermediate stage between simulation and deployment. The term “cosimulation” reflects a division of labor in which Simulink models the plant, while code generated from the controller subsystem runs on the actual target hardware. In a PIL cosimulation, the target processor participates fully in the simulation loop — hence the term “processor-in-the-loop.”

How Cosimulation Works

This figure illustrates how the plant (P) and controller (C) components interact in a PIL cosimulation.



In a PIL cosimulation, the Real-Time Workshop Embedded Coder generates efficient code for the control system. This code runs (in simulated time) on a target board using the intended microcontroller. The plant model remains in Simulink without the use of code generation.

During PIL cosimulation, Simulink simulates the plant model for one sample interval and exports the output signals (`Yout` of the plant) to the target board via a communications link. When the target processor receives signals from the plant model, it executes the controller code for one sample step. The controller returns its output signals (`Yout` of the controller) computed during this step to Simulink, via the same communications link. At this point one sample cycle of the simulation is complete and the plant model proceeds to the next sample interval. The process repeats and the simulation progresses.

To learn about PIL cosimulation though hands-on experience, see “Tutorial 1: Building and Running a PIL Cosimulation” on page 2-5.

Tutorial 1: Building and Running a PIL Cosimulation

In this tutorial, you will use a subsystem in a Simulink model as a component in simulations on your host computer, and also in a PIL cosimulation running on your phyCORE-MPC555 board.

Before You Begin

Before working with this tutorial, you should read and follow the procedures in "Setting Up and Verifying Your Installation" on page 1-11. Make sure that the target preferences are set up appropriately for your development system (CodeWarrior or Diab) as described in "Setting Target Preferences" on page 1-15

Hardware Connections

The PIL target requires both parallel and serial connections between your PC and the phyCORE-MPC555 board. The parallel connection is required for downloading code. The serial connection is required for host/target communications during cosimulation.

We assume that you have made the following connections, as described in the "Interfacing the phyCORE-MPC555 to a Host PC" section of the *phyCORE-MPC555 Quickstart Instructions* manual:

- Host PC parallel (LPT1) port to the DB-25 connector (P1) on the phyCORE-MPC555. (If you are using a Wiggler, connect LPT1 to the Wiggler DB-25 connector, and connect the Wiggler to the internal BDM connector on the phyCORE-MPC555 board.)
- Host PC serial (COM1) port to the RS232-1 (P2) connector on the phyCORE-MPC555 board.

The Demo Model

We have provided a demo model for your use. The Fault-Tolerant Fuel Control System model, shown in Figure 2-1, consists of a plant model with a controller subsystem, the `fuel rate controller` subsystem.

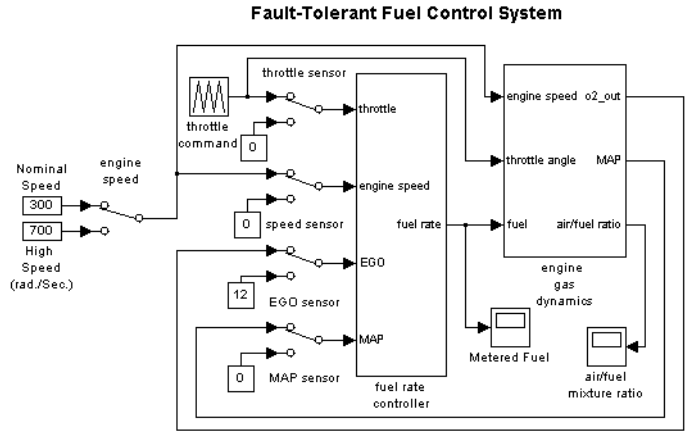
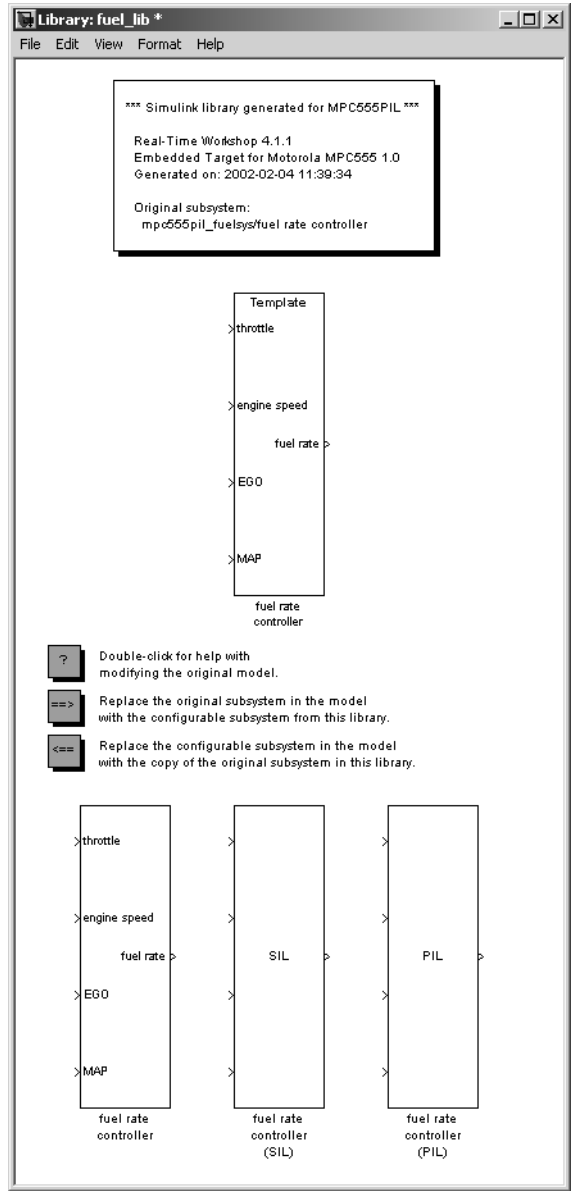


Figure 2-1: Fault-Tolerant Fuel Control System Model

In the following sections, you will use the demo model and the PIL target to generate the following:

- PIL code to run on the target board. The PIL target automatically invokes the appropriate cross-development tools to compile, link, and (optionally) download and run a target executable.
- A library containing
  - The original `fuel rate controller` subsystem block for use in simulation.
  - An S-function wrapper block, generated by the Real-Time Workshop Embedded Coder, that implements the `fuel rate controller` subsystem for use in software-in-the-loop (SIL) simulation.
  - A subsystem block that implements the `fuel rate controller` subsystem on the host side during cosimulation. This subsystem communicates with generated PIL code running on the target board.
  - A master Configurable Subsystem block that represents the above three components. You will plug this block into a plant model and select each of the three components in turn for use in a simulation.

This figure shows a library generated by the PIL target.



Once you start the build process, there is almost no manual intervention required to build all these components.

After building the components, you will use them in normal simulation, SIL simulation, and PIL cosimulation. You will monitor the results of each simulation via the Scope blocks in the model.

Setting Up the Model

In this section you will make a local copy of the demo model and configure the model as required by this exercise:

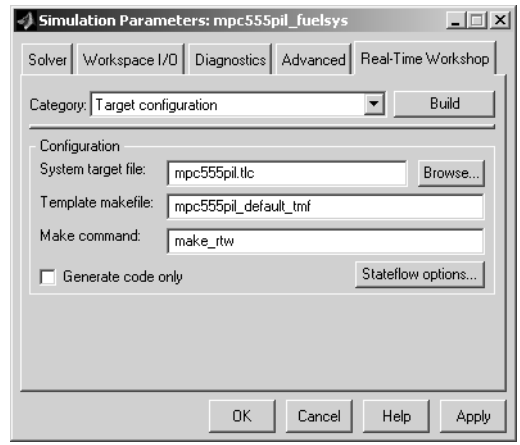
1 Make a local copy of the demo model, `mpc555pil_fuelsys.mdl`.

   The model is located in the directory *`matlabroot`*`/toolbox/rtw/targets/mpc555dk/mpc555demos`. Open `mpc555pil_fuelsys.mdl` and save a copy of the model to your working directory.

Next, check that the model is correctly configured for use with the Embedded Target for Motorola MPC555.

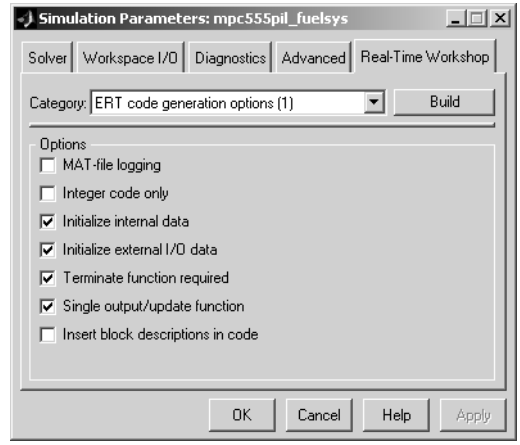
2 From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog box opens. Click on the **Real-Time Workshop** tab to activate the Real-Time Workshop pane.

3 Select `Target configuration` from the **Category** menu of the Real-Time Workshop pane. The target configuration should be as shown in this figure.



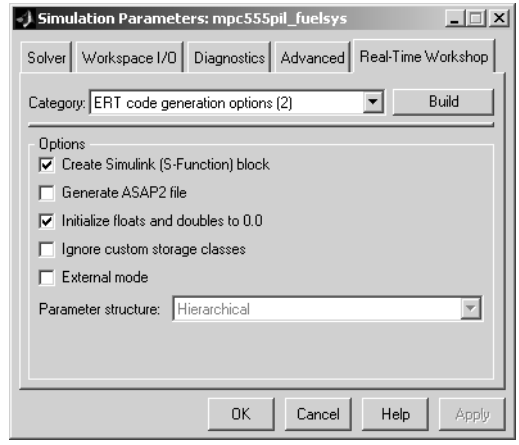
4 If the target configuration settings are not as shown, click the **Browse...** button to open the System Target File Browser, and select the **Embedded Target for Motorola MPC555 (processor-in-the-loop)** target. Then click **OK** to close the Browser and return to the Real-Time Workshop pane.

5 Select `ERT code generation options(1)` from the **Category** menu. Make sure that the options are set to their defaults, as shown in this figure.

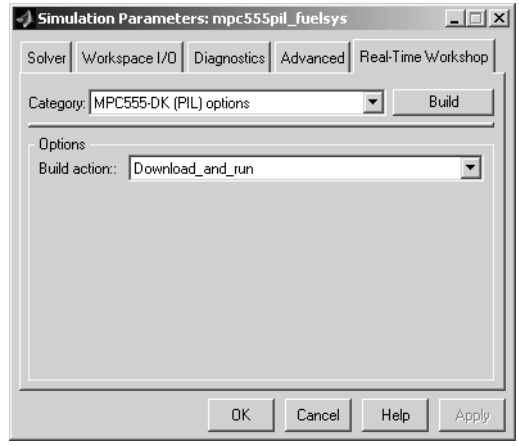


6 Select `ERT code generation options (2)` from the **Category** menu. Make sure that the options are set as shown in this figure. Note that the **Create Simulink (S-Function) block** option is selected. This is required to generate a Real-Time Workshop Embedded Coder S-function wrapper block.

.



7 Select `MPC555-DK (PIL) options` from the **Category** menu.



8 Select `Download_and_run` from the **Build action** list. This option automatically invokes the appropriate downloading/debugging utility for your development environment, as specified in your target preferences.

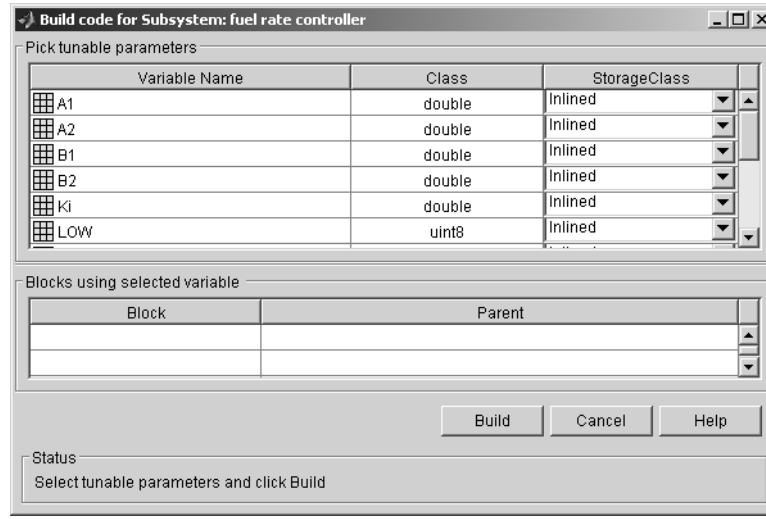
9 Click **Apply** if you have changed any parameters. Then click **OK** to close the **Simulation Parameters** dialog box. If needed, save the model to preserve any changes you have made.

Building PIL and Simulation Components

In this section, you will build a library of simulation, SIL, and PIL components from the `fuel rate controller` subsystem:

1 Right-click on the `fuel rate controller` subsystem. A context menu appears. Select `Build Subsystem` from the **Real-Time Workshop** submenu of the context menu.

2 The **Build code for Subsystem** window opens. This window displays information about each variable (or data object) that is referenced as a block parameter in the subsystem. The window lets you inline or set the storage class of individual parameters. We will not be concerned with these features in this exercise. Click the **Build** button to continue the code generation and build process.

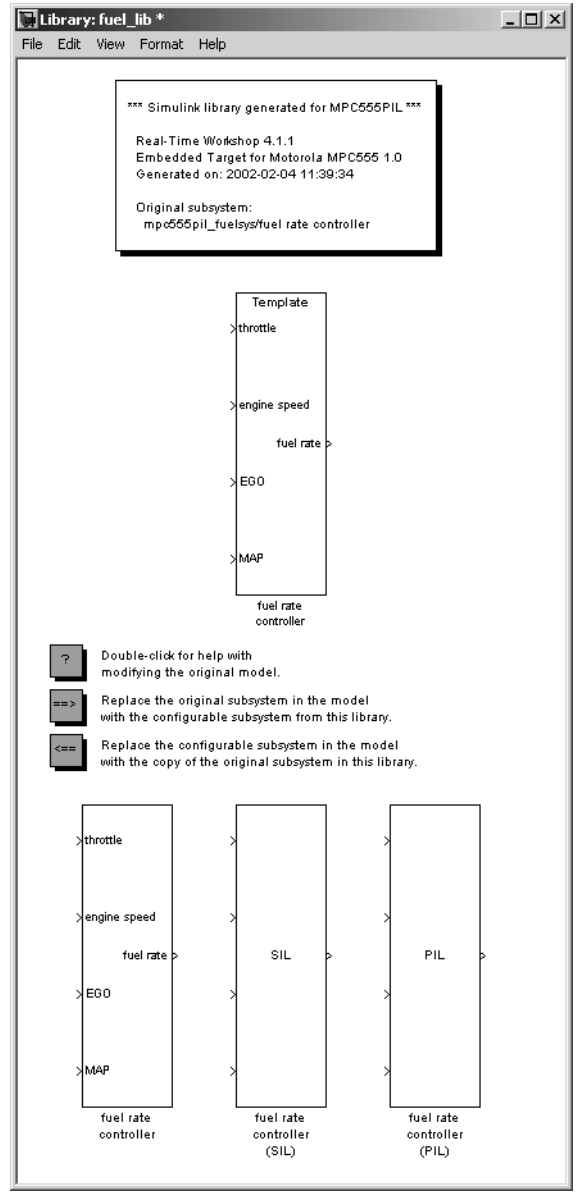


3 The build process displays status messages in the MATLAB command window. Intermediate Simulink windows are displayed as the build process creates various components.

4 When the code generation process completes, the PIL target automates the process of compiling, downloading, and executing the generated PIL code that is to run on the target hardware. To accomplish this, the PIL target launches your cross-development system (Diab or CodeWarrior), compiles and makes the executable, and invokes the appropriate downloading and debugging utility (SingleStep or CodeWarrior debugger). You do not need to intervene in this process.

5 At this point, the generated program is running on the target hardware and waiting for communication to be established with Simulink on the host PC.

6 The build process has created and opened a library named `fuel_lib`, as shown in this figure.


Library: fuel_lib *
File Edit View Format Help
*** Simulink library generated for MPC555PIL ***
Real-Time Workshop 4.1.1
Embedded Target for Motorola MPC555 1.0
Generated on: 2002-02-04 11:39:34
Original subsystem:
mpc555pil_fuelsys/fuel rate controller
Template
throttle
engine speed
fuel rate
EGO
MAP
fuel rate controller
?
Double-click for help with modifying the original model.
==>
Replace the original subsystem in the model with the configurable subsystem from this library.
<==
Replace the configurable subsystem in the model with the copy of the original subsystem in this library.
throttle
engine speed
fuel rate
EGO
MAP
fuel rate controller
SIL
fuel rate controller (SIL)
PIL
fuel rate controller (PIL)

The library contains

- A copy of the original `fuel rate controller` subsystem.
- A Real-Time Workshop Embedded Coder generated S-function, labeled `fuel rate controller (SIL)`.
- A subsystem block that communicates with generated PIL code running on the target board during cosimulation, labeled `fuel rate controller (PIL)`.
- A master Configurable Subsystem referencing the other three blocks. The default block choice for this subsystem is the original `fuel rate controller` subsystem.

The Configurable Subsystem, when plugged into the model, lets you choose which of the three library components will perform the controller functions in the model. We will use the Configurable Subsystem in the following sections.

The library window also contains the following controls:

- A **Help** button that displays PIL target documentation in the MATLAB help browser.
- A button that lets you replace the original (generating) subsystem in the model with the generated Configurable Subsystem.
- A button that lets you do the inverse, i.e., remove the Configurable Subsystem from the model from the original model and replace it with the original (generating) subsystem from the library.

The library window documents the name of the original model/subsystem from which the library was generated,

Using the Demo Model In a PIL Cosimulation

In this section, we will plug the Configurable Subsystem into the demo model, select the PIL component, and use it in a PIL co- simulation:

1. Click on the `fuel_lib` library window to activate it. Double-click on the button labeled **Replace the original subsystem in the model with the configurable subsystem from this library**.

2. The `mpc555pil_fuelsys` model window is now the active window. The original `fuel rate controller` subsystem has been deleted from the model. It has been replaced by the Configurable Subsystem from the `fuel_lib`

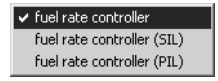
library. The Configurable Subsystem is automatically connected to the same signals that the original `fuel rate controller` subsystem was connected to.

Note It is important to be aware that the insertion of the Configurable Subsystem into the containing model establishes a link between the model, `mpc555pil_fuelsys`, and the library, `fuel_lib`. The library has information about the model and subsystem from which it was generated. The model, in turn, has information about the library from which the Configurable Subsystem comes. This linkage is based on the names of the library and the model, and will be broken if either is renamed. To avoid errors, treat the model and library as a single unit, and do not rename either.

---

3. Save the model.

4. Right-click on the Configurable Subsystem in the model. A context menu appears. Select the **Block choice** menu item and observe the block choice submenu. This figure shows the default block choice selection.



5. From the **Block choice** submenu of the context menu, select `fuel rate controller (PIL)`.

6. Open the model's two Scope blocks, if they are not already opened.

7. Make sure that Simulink is in Normal mode.

8. You are now ready to run the cosimulation. To start the cosimulation, click the **Start simulation** button in the Simulink toolbar.

   The target system now starts executing the controller code. Observe that the output signals computed on the target are displayed on the scopes. The

updating of the Scope blocks is slow, relative to a normal simulation, because data is transmitted over the serial line on every model step.

Notice also that one of the green LEDs on the PhyCORE-MPC555 board blinks while the cosimulation runs.

9 When the simulation completes, the LED stops blinking. The signals displayed on the scopes should appear as shown in Figure 2-2.

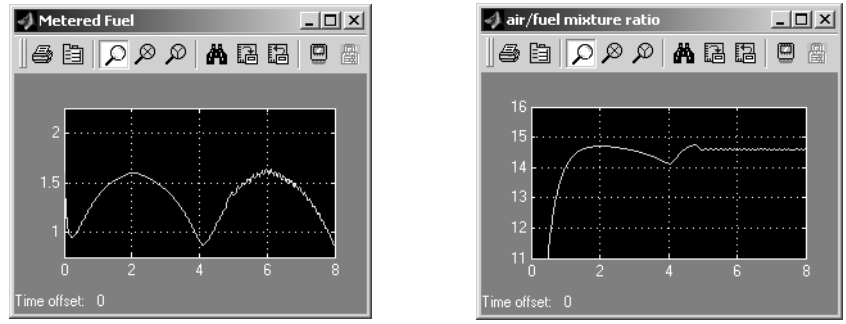


Figure 2-2: Signals Displayed at End of Simulation or Cosimulation

10 When the cosimulation has completed, or has stopped or paused, the target code enters a wait state until it receives a command to start (or resume) from the host. Restart the cosimulation by clicking the **Start simulation** button again. You can start, stop, restart, pause, or continue a cosimulation exactly as you would a normal simulation. Try each of these operations a few times.

11 Stop the cosimulation (or let it complete) and activate your cross-development system. Terminate the program on the target system, and exit your cross-development system.

Note that you can open the project and executable files, created by the PIL target for your cross-development system, at any time. You can then reload and run the target code, and run another cosimulation. See “Build Process Files and Directories” on page 2-23 for information on the files and directories created by the build process.

Tutorial 2: Modifying and Rebuilding the Controller

In this section, we will continue to use the Configurable Subsystem we built in "Tutorial 1: Building and Running a PIL Cosimulation" on page 2-5.

In this tutorial, we will make a simple change to the original `fuel rate controller` subsystem in our generated library, `fuel_lib`. We will then rebuild the library components, and run another cosimulation, observing the behavior of our modified controller. All of these steps will be accomplished within the same model/library pair.

Note Before you begin the procedure below, make sure that you have stopped the target program and exited your cross-development system.

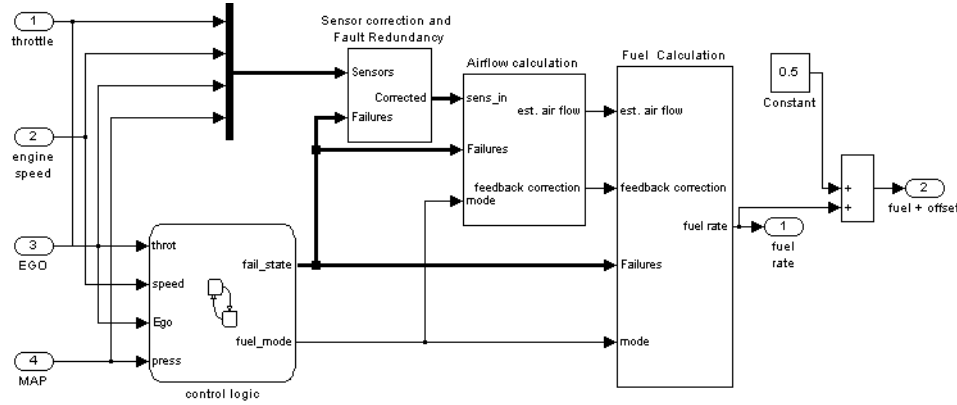
Modifying the Controller

In this section, we add an output signal and port to the controller subsystem. The changes we make to the controller subsystem in this section are for demonstration purposes; they do not add useful functionality to the model:

1. Activate the `fuel_lib` library, and double-click on the original `fuel rate controller` subsystem.

2. Add a Sum block, a Constant block, and an outport to the `fuel rate controller` subsystem. Configure them such that an offset of 0.5 is summed with the fuel rate signal.

3 Route the Sum block output to the new outport, and label the outport `fuel + offset`. The `fuel rate controller` subsystem should now resemble this block diagram.



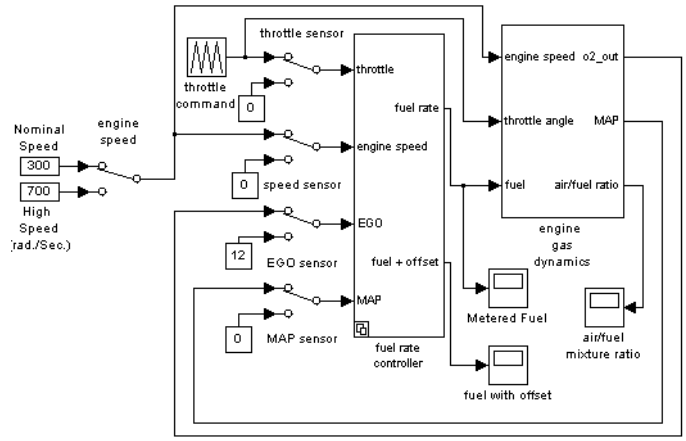
4 Close the `fuel rate controller` subsystem. Observe that, in the library window, the `fuel rate controller` subsystem now has two outputs, but the SIL and PIL blocks in the library do not. These components will not be updated until the library is rebuilt.

Note You do not need to remove the Configurable Subsystem from the model to rebuild the code for the SIL and PIL components.

---

5 Activate the `mpc555pil_fuelsys` model. Right-click on the Configurable Subsystem in the model. A context menu appears. From the **Block choice** submenu of the context menu, select `fuel rate controller`. Observe that the Configurable Subsystem reflects the change in the corresponding component of the library, showing two outports.

6 Add a Scope block to the model and connect it to the new `fuel + offset` outport of the Configurable Subsystem. Label the scope `fuel with offset`. The model should now resemble this block diagram.



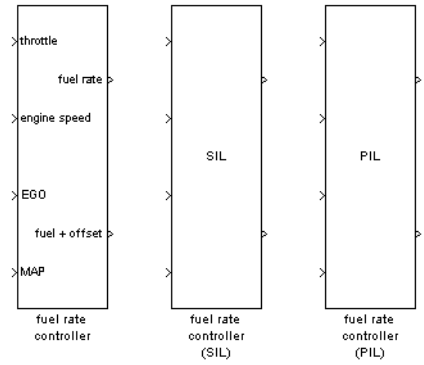
Rebuilding the Controller and Cosimulating

You are now ready to rebuild the PIL code and library components. This time, however, you will build from the Configurable Subsystem, which must be linked back to the `fuel rate controller` subsystem in the library. Before continuing, right-click on the Configurable Subsystem and make sure that, in the **Block choice** submenu of the context menu, `fuel rate controller` is selected. *Do not select* `fuel rate controller (SIL)` *or* `fuel rate controller (PIL)`.

To rebuild the PIL code and library components:

1 Right-click on the Configurable Subsystem, and select **Build Subsystem** from the **Real-Time Workshop** submenu of the context menu.

2 The build process proceeds as described in the previous tutorial (see "Building PIL and Simulation Components" on page 2-11 if necessary). At the end of the build process, the `fuel_lib` library is again activated. Observe that the rebuilt SIL and PIL components now have two outports, like the original subsystem from which they were generated, as shown in this figure.



3 The PIL code has been downloaded to the target; you can now cosimulate again with the rebuilt PIL code. As before, right-click on the Configurable Subsystem in the model, and select `fuel rate controller (PIL)` from the **Block choice** submenu of the context menu.

4 Open all the model's Scope blocks, if they are not already opened.

5 Make sure that Simulink is in Normal mode.

6 Click the **Start simulation** button in the Simulink toolbar.

Observe the signals displayed on the scopes. The `fuel with offset` scope and the `Metered Fuel` scope should display signals that are identical except for their offsets. Otherwise, all signals should be identical to the signals generated by the previous cosimulation.

7 Clean up by terminating the program on the target system, and exiting your cross-development system. Save the model if desired.

In the next section, you will use the other components of the `fuel_lib` library in simulations.

Tutorial 3: Using the Demo Model In Simulation

In this section, we will continue to use the Configurable Subsystem in the demo model, using it first in a normal closed-loop simulation and then in a SIL simulation.

Closed-Loop Simulation

1. Right-click on the Configurable Subsystem and select `fuel rate controller` from the **Block choice** submenu of the subsystem block's context menu. This selects the controller subsystem that was used in the original model.

2. Open the Scope blocks and start the simulation. When the simulation completes (simulation time is set to 8 seconds), the signals displayed on the scopes should appear identical to those displayed during the previous cosimulation (see Figure 2-2 on page 2-16).

SIL Simulation

1. Right-click on the Configurable Subsystem and select `fuel rate controller (SIL)` from the **Block choice** submenu of the subsystem block's context menu.

   Selecting this option directs Simulink to call a generated wrapper S-function that implements the controller algorithm in highly efficient Real-Time Workshop Embedded Coder generated code. You can now run a SIL simulation.

2. Start the simulation. You will notice that the simulation completes much more quickly, due to the efficiency of the generated code. Also, observe that the generated code displays results, on the scopes, that are identical to the previous simulation and cosimulation (see Figure 2-2).

PIL Target Summary

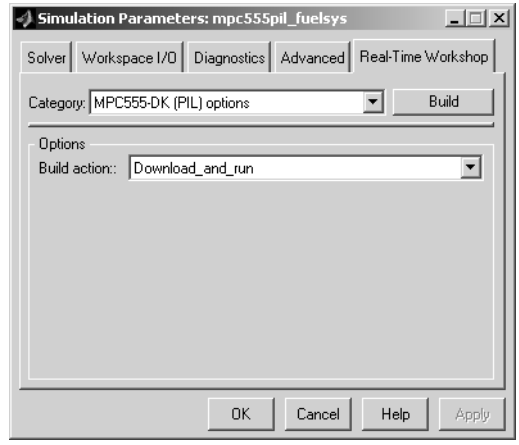
Code Generation Options

The PIL target is an extension of the Real-Time Workshop Embedded Coder embedded real-time (ERT) target configuration. The PIL target 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 **Category** menu of the Real-Time Workshop pane of the **Simulation Parameters** dialog box; they are documented in the Real-Time Workshop documentation and the Real-Time Workshop Embedded Coder documentation.

Some code generation options of the ERT target are not relevant to the PIL target, and are either unsupported, or restricted in their operation, by the PIL target. See “Restrictions” on page 2-24 for details.

Target-Specific Option

The the PIL target has one target-specific code generation option: **Build action**. To view or change the setting of this option, select `MPC555-DK (PIL) options` from the **Category** menu of the Real-Time Workshop pane of the **Simulation Parameters** dialog box.



The **Build action** menu has two options that control what action the PIL target takes after completing the code generation process:

- `Download_and_run`: When this option is selected, the PIL target automatically invokes the appropriate downloading/debugging utility for your development environment, as specified in your target preferences. The PIL target downloads the generated code to the target board and begins execution of the code.

  Before using this option, make sure that the `TargetCompiler`, `TargetCompilerPath`, `TargetDebugger`, and `TargetDebuggerExe` target preferences are set correctly.
- `None`: When this option is selected, the PIL target does not take any action after code generation completes. To download and run your application, you must do so manually, using your development tools.

Build Process Files and Directories

The PIL target creates the following in your working directory:

- A build directory, containing generated source code, object files, and a makefile and other control files. The build directory also may contain subdirectories used by Stateflow and by the HTML code generation report generator (see "Code Analysis Reporting" on page 4-3).

  The naming convention for the build directory is *`source`*`_mpc555pil`, where *`source`* is the first word of the generating subsystem or model. For example, the `fuel rate controller` subsystem used in the PIL tutorials generates the build directory `fuel_mpc555pil`.
- The generated library, *`source`*`_lib.mdl`, and the `.dll` components that are bound to the generated PIL and SIL blocks in the library. Note that if you rebuild *`source`*`_lib.mdl` in the same working directory, a revision number is appended to the *`source`* string. For example, building from the `fuel rate controller` subsystem used in the PIL tutorials generates `fuel_lib.mdl`, `fuel1_lib.mdl`, `fuel2_lib.mdl`... `fuel`*`n`*`_lib.mdl`.
- Executable PIL code in a format suitable for downloading to the target and execution by your development system (Diab or Metrowerks).
- Project files, debugging symbol files, link maps, and other files specific to your development system (Diab and Metrowerks).

If you do not select the `Download_and_run` option when you generate code (or if you want to rerun PIL code after it is built), you must manually download and

run the generated executable using your development system. The procedure for doing this varies with your development system:

- Metrowerks CodeWarrior: The PIL target generates *source*.mcp, a Metrowerks CodeWarrior project file, in the build directory.
- Diab compiler/SingleStep debugger: The PIL target generates *source*_ram.scr, a control file to be used with SingleStep, in the build directory. To invoke SingleStep and download and run PIL code, type the following command at the MATLAB prompt (as one continuous line)

```
!start ssteproot\bdmp58.exe -P -S
matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\app_startup\mw_ram.wsp -r
source_mpc555pil\source_ram.scr
```

  where *ssteproot* is the installed SingleStep directory, and *matlabroot* is the MATLAB root directory.

Restrictions

Please note the following restrictions on the use of the PIL target:

- The PIL target does not currently support code generation from device driver blocks from the Embedded Target for Motorola MPC555 block libraries. If your model includes any such blocks, you can use them in pass-through mode during simulation or cosimulation only.
- In a plant/controller simulation where the controller is built via the PIL target, the plant model can contain any Simulink blocks, including a combination of continuous-time and discrete-time blocks. However, controller subsystem must not include any continuous-time blocks,. This component is used for code generation in the Embedded-C format of the Real-Time Workshop Embedded Coder; the Embedded-C format does not support continuous blocks.
- Certain ERT code generation options are not supported by the PIL target. If these options are selected, the PIL target either ignores the option or issues an error message during the build process. Table 2-1 summarizes these restricted options.

Table 2-1: PIL Target Restricted Code Generation Options

Option	Restriction
MAT-file logging	Ignored if selected; build process proceeds
Generate ASAP2 file	Ignored if selected; build process proceeds
External mode	Error if selected; build process terminates
Generate an example main program	This option should not be selected for the PIL target. The PIL target supplies a target-specific main program, `mpc555dk_main.c`.
Generate reusable code	Error if selected; build process terminates
Target floating-point math environment	Error if `ISO_C` menu option is selected. Use only the `ANSI_C` option (default).

3

Generating Stand-Alone Real-Time Applications

This section includes the following topics:

Introduction (p. 3-2)	An overview of the Embedded Target for Motorola MPC555 real-time target, other components required to generate stand-alone real-time applications, and the process of deploying generated code on target hardware.
Tutorial: Creating a New Application (p. 3-4)	A hands-on exercise in building an application from a demo model, including downloading and executing generated code on a target board.
Downloading Boot and Application Code (p. 3-23)	A detailed discussion of the process of downloading code to the MPC555 RAM and flash memory.
Generating ASAP2 Files (p. 3-35)	How to generate ASAP2 files from your model.
Summary of the Real-Time Target (p. 3-40)	Summary of the code generation options specific to the real-time target, and requirements am restrictions that apply to the current release.

Introduction

This section describes how to generate a stand-alone real-time application for the MPC555. The components required to generate stand-alone code are

- The Embedded Target for Motorola MPC555 real-time target
- The MPC555 Resource Configuration object provided in the Embedded Target for Motorola MPC555 library
- I/O driver blocks provided in the Embedded Target for Motorola MPC555 library
- Utilities for downloading generated code to the target hardware

Using these, you can build a complete application. You do not need to hand-write any C code to integrate the generated code into a final application.

The tutorial “Tutorial: Creating a New Application” on page 3-4 uses two blocks from the Embedded Target for Motorola MPC555 library. For complete information on the Embedded Target for Motorola MPC555 library blocks, see Chapter 5, “Block Reference.”

Before reading this section and using the Embedded Target for Motorola MPC555 library, you should have at least a basic understanding of the architecture of the MPC555. To learn about the MPC555, we suggest that you study the *MPC555 Users Manual*. We recommend that you read the introduction to the processor and familiarize yourself with all the major subsystems of the MPC555.You can find this document at the following URL: `http://e-www.motorola.com/webby/asps/library/prod_lib.jsp`.

Deploying Generated Code

You can load a generated program into the MPC555 flash memory for permanent deployment. You can also load your code into external RAM (if available on your development hardware).

Alternatively, you can use the automatic code generation process for rapid prototyping and investigate a range of different design alternatives before making a deployment decision.

Your generated program can run on any Electronic Control Unit (ECU) that is based on the MPC555 processor. Your application can use any of the supported MPC555 on-chip I/O devices. The supported devices include analog input,

digital I/O, CAN and PWM input or output (see Chapter 5, “Block Reference” for further information on the device driver blocks in the Embedded Target for Motorola MPC555 library).

In addition to on-chip I/O resources, an ECU typically provides additional I/O devices. If you want to access such custom I/O devices, you must write device drivers and integrate them with the automatically generated code. See the following documentation for details:

- *Real-Time Workshop User’s Guide*
- *Real-Time Workshop Embedded Coder User’s Guide*
- *Writing S-Functions*

Once the application has been programmed into memory on the target system, you may need to monitor signals or tune parameters. The Embedded Target for Motorola MPC555 supports signal monitoring and parameter tuning via the CAN Calibration Protocol (CCP). To enable CCP, you must include a CAN Calibration Protocol block in your model. The CAN Calibration Protocol block implementation of CCP has been tested against CANape from Vector-Informatik. See “CAN Calibration Protocol” on page 5-12 for further information.

Tutorial: Creating a New Application

In this tutorial, we will build a stand-alone real-time application from a model incorporating blocks from the Embedded Target for Motorola MPC555 library. We assume that you are already familiar with Simulink and with the Real-Time Workshop code generation and build process.

In the following sections, we will

- Configure the model
- Use the pass-though feature in simulation
- Generate code from a subsystem
- Download code by one of the following methods:
  - Download to target RAM via a CAN connection, using the Download Control Panel utility (provided with the Embedded Target for Motorola MPC555).
  - Download to target RAM via a BDM connection, using Wind River Systems SingleStep debugger.
- Execute the code on the target.

After you complete this tutorial, you may want to learn how to deploy generated code into the MPC555 flash memory. See “Downloading Boot and Application Code” on page 3-23 for that information.

Before You Begin

This tutorial requires the following specific hardware and software in addition to the Embedded Target for Motorola MPC555:

- Phytec phyCORE-MPC555 development board

  The tutorial model utilizes two LEDs on the phyCORE-MPC555 board. These LEDs are connected to pins `MPIO32B0` and `MPIO32B1` on the MPC555 MIOS digital output pins. If you are using a different development board, you may be able to obtain the same functionality by making similar connections.
- If you want to download generated code to the target board via CAN, you will need one of the following supported CAN cards, and the drivers supplied by the manufacturer:

- Vector-Informatik CanAC2PCI
- Vector-Informatik CanAC2
- Vector-Informatik CanCardX
- Vector-Informatik CanPari

If you have not yet done so, install your Vector-Informatik CAN card and drivers. See your Vector-Informatik documentation for instructions on installation and verification. In addition, after installing the drivers from Vector-Informatik, you must locate the `vcand32.dll` library, and add a path to the location where `vcand32.dll` was installed to the Windows system path.

Note Please check the Vector Informatik Web site at `http://www.vector-informatik.com/` to make sure you have drivers suitable for your PC operating system version. Note that serious system problems can arise if you use drivers for the wrong PC operating system version (e.g., installing drivers for Windows NT on a Windows 2000 system).

- Wind River Systems Diab Data D-CC cross-compiler

  Make sure that your target preferences are set correctly for the Diab development tools:

  - Set the `TargetCompiler` preference to `'Diab'`:

    ```
    settargetprefs('mpc555dk','TargetCompiler','Diab')
    ```

  - Set the `TargetCompilerPath` preference to the location of your Diab cross compiler. For example:

    ```
    settargetprefs('mpc555dk','TargetCompilerPath',...
    'D:\Applications\diab\4.3g\WIN32\bin')
    ```

- Wind River Systems SingleStep debugger (version 7.6.2)

  If you have not yet done so, set the `TargetDebuggerExe` preference to the location of SingleStep. For example:

```
settargetprefs('mpc555dk','TargetCompilerPath','D:\Apps\Diab\4.3g\win32\bin')
settargetprefs('mpc555dk','TargetDebuggerExe',...
'D:\Apps\singlestep\sds762\cmd\bdmp58.exe')
```

  SingleStep is required to download boot code to the MPC555 flash memory. Once the boot code is loaded into flash memory, you can download code to the processor entirely over the CAN network as described later in this tutorial. If you have not yet downloaded boot code to the MPC555 flash memory, please read and follow the instructions in "Configuring SingleStep and Downloading Boot Code" on page 1-22.

  You can also use SingleStep to download and execute generated code in MPC555 RAM, as described later in this tutorial.

The Example Model

In this tutorial we will use a simple example model, `mpc555rt_led`, from the directory *`matlabroot`*`/toolbox/rtw/targets/mpc555dk/mpc555demos`.

This directory is on the default MATLAB path.

1. Open the model.

   ```
   mpc555rt_led
   ```

2. Save a local copy to your working directory. We will work with this copy throughout this exercise.

   Figure 3-1 shows the example model at the root level. We will only use the root model in simulation.

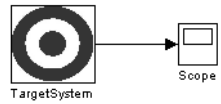


Figure 3-1: mpc555rt_led_demo Model, Root Level

3 Double-click on the `TargetSystem` subsystem block.

Figure 3-2 shows the `TargetSystem` subsystem, from which we will generate code.

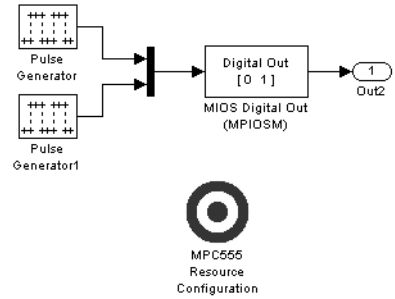
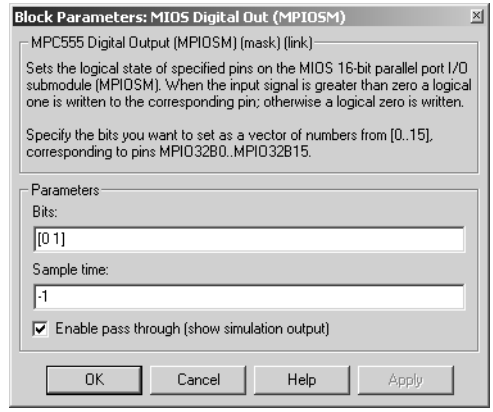


Figure 3-2: TargetSystem Subsystem

In the `TargetSystem` subsystem, two square wave signals are multiplexed and routed to the MIOS Digital Out block. The MIOS Digital Out block accepts a vector of numbers representing pins 0-15 on the MIOS 16-bit Parallel Port I/O Submodule (MPIOSM) on the MPC555. As the square wave signals oscillate between 0 and 1, the MIOS Digital Out block writes corresponding logic values to the appropriate pin on the port.

This picture shows the parameters of the MIOS Digital Out block.



The **Bits** field is set to the vector `[0 1]`. The block maps this vector to the MPC555 MIOS digital output pins `MPIO32B0` and `MPIO32B1`. When the application runs, it will send a pulse signal to these output pins. On the PhyCORE-MPC555 board, these signals are connected to two of the LEDs, which will switch on and off at the frequency set in the respective pulse generator blocks.

During simulation, the MIOS Digital Out block simply passes its input signal through to its output, and the square waves can be viewed on the Scope block.

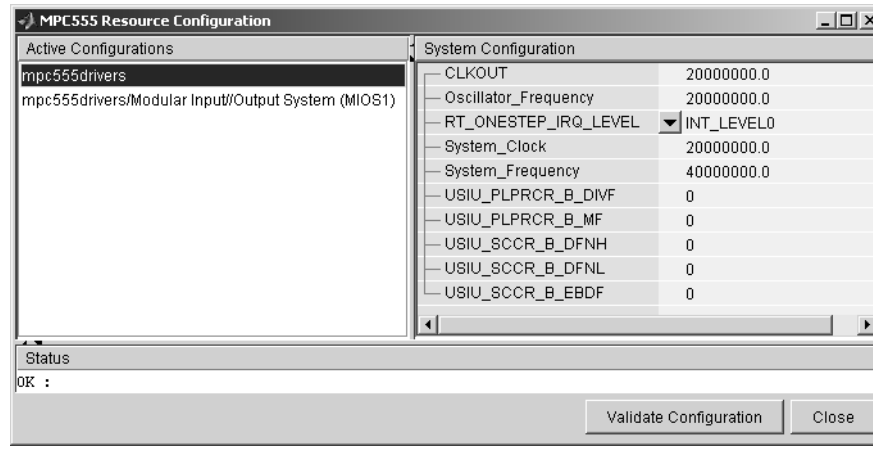
In addition to the Pulse Wave, Mux, MIOS Digital Out, and Output blocks, the `TargetSystem` subsystem contains a MPC555 Resource Configuration object. When building a model with driver blocks from the Embedded Target for Motorola MPC555 library, you must always place a MPC555 Resource Configuration object into the model (or the subsystem from which you want to generate code) first.

The purpose of the MPC555 Resource Configuration object is to provide information to other blocks in the model. Unlike conventional blocks, the MPC555 Resource Configuration object is not connected to other blocks via input or output ports. Instead, driver blocks (such as the MIOS Digital Out block in the example model) query the MPC555 Resource Configuration object for required information.

For example, a driver block may need to find the system clock speed that is configured in the MPC555 Resource Configuration object. The MPC555 has a

number of clocked subsystems; to generate correct code, driver blocks need to know the speeds at which the these clock busses will run.

The MPC555 Resource Configuration window lets you examine and edit the MPC555 Resource Configuration settings. To open the MPC555 Resource Configuration window, double-click on the MPC555 Resource Configuration icon. This picture shows the MPC555 Resource Configuration window for the `TargetSystem` subsystem.



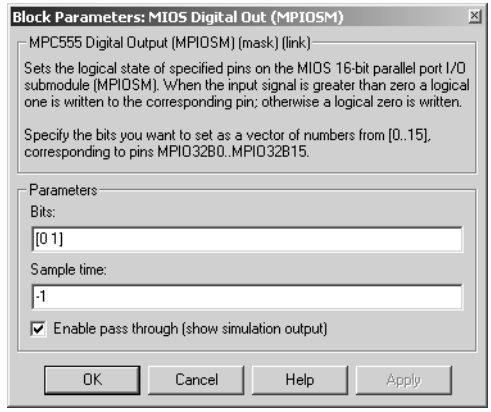
In this tutorial, we will use the default MPC555 Resource Configuration settings. Observe, but do not change, the parameters in the MPC555 Resource Configuration window. To learn more about the MPC555 Resource Configuration object, see “MPC555 Resource Configuration” on page 5-47.

Close the MPC555 Resource Configuration window before proceeding.

Using the Pass-Through Option in Simulation

Device driver blocks in the Embedded Target for Motorola MPC555 library have a unique pass-through option. This option lets you provide a signal from a device driver block for use in simulation.

Open the MIOS Digital Out block from the `TargetSystem` subsystem. Make sure that the **Enable pass through (show simulation input)** option is selected, as shown in this picture.



When this option is enabled, an outport appears on the block. The block input is passed through to the output during simulation. This option affects simulation only.

To see the effect of pass-through in simulation:

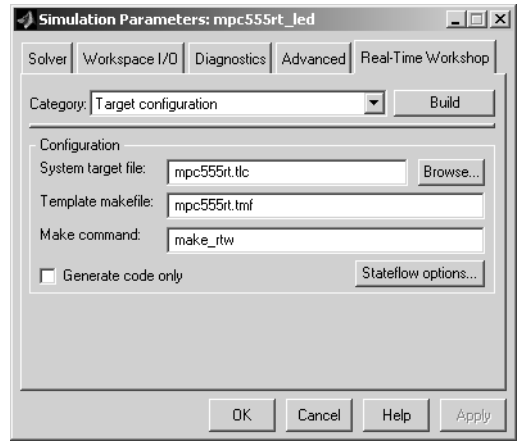
1. Open the Scope block in the root model.
2. Start the simulation; observe the output signals from the `TargetSystem` subsystem.
3. Stop the simulation manually (the simulation time is set to `inf`).

Generating Code

We will now generate application code:

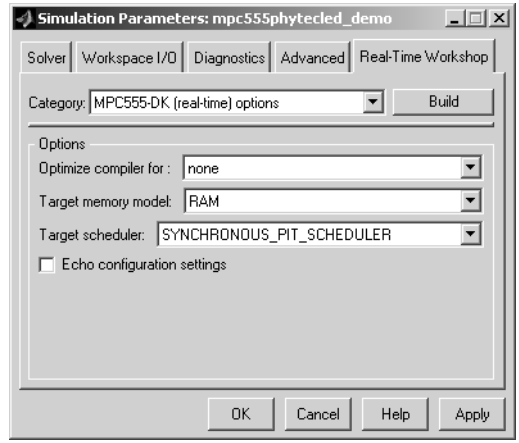
1. Open the **Simulation parameters** dialog box and select the Real-Time Workshop pane. Select `Target configuration` from the **Category** menu.
2. Click on the **Browse** button to open the **System Target File Browser**. In the browser, select `Embedded Target for Motorola MPC555 (real-time target)`. Then click **OK** to close the browser and return to the Real-Time Workshop pane.

3 The **Target configuration** settings should now be as shown below.



4 From the **Category** menu of the Real-Time Workshop pane, select `MPC555-DK (real-time) options`. The `RAM` option should be selected from the **Target memory model** menu. This option directs the Real-Time Workshop to generate a code file in Motorola S-record format, which is suitable for downloading and execution in RAM.

Leave the other options set to their defaults. The code generation options should appear as shown below.



5 You are now ready to build the application. Do this by clicking on the **Build** button.

6 On successful completion of the build process, two files are created in the working directory:

   a `Target_ram.srec`: Code only, without symbols, suitable for execution on the target system.

   b `Target_ram.out`: Code and symbols, suitable for use with a symbolic debugger such as Wind River SingleStep.

   You can download to RAM:

   - Via the BDM port, using the Wind River SingleStep debugger, as described in "Downloading the Application to RAM via SingleStep/BDM" on page 3-13.
   - Via CAN, using the Download Control Panel utility with Vector-Informatik hardware, as described in "Downloading the Application to RAM via CAN" on page 3-19.

Downloading the Application to RAM via SingleStep/BDM

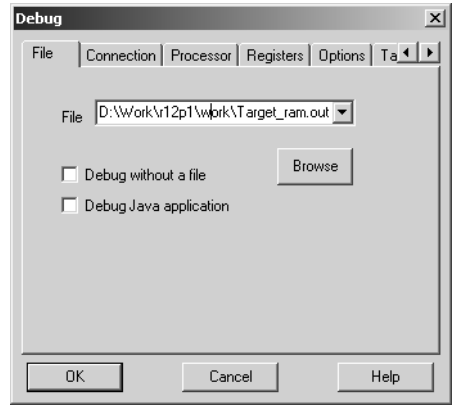
In this section, we will use the Wind River Systems SingleStep debugger to download and start the generated `Target_ram.out` file to RAM on the target system. `Target_ram.out` contains both code and symbols for use with the debugger.

Do the following before you begin:

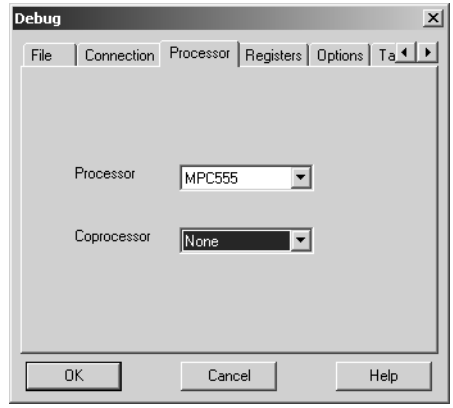
- Make sure that you have downloaded boot code to the flash memory of the MPC555. See "Configuring SingleStep and Downloading Boot Code" on page 1–22.
- Connect the BDM port of your development board to parallel port LPT1 of your host PC.
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in "Phytec Jumper Settings" on page 1–12.
- Cycle the power (or perform a hard reset) on your development board to clear the RAM.

To download the generated `Target_ram.out` file to RAM:

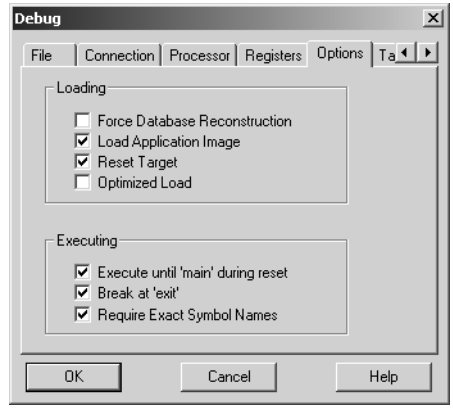
1 Start SingleStep using the `SingleStep On Chip (MPC5xx) for RAM` shortcut you created previously (see "Configure Shortcuts to SingleStep" on page 1–22). The **Debug** dialog opens. Clear the **Debug without a file** option. Click on the **File** tab. Then, use the **Browse** button to locate the `Target_ram.out` file.



2 Click on the **Processor** tab. Confirm that the `MPC555` is selected in the **Processor** list, as shown.

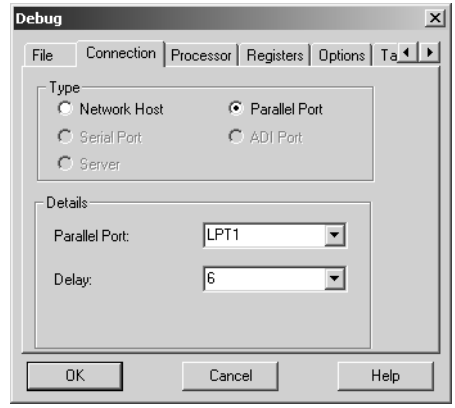


3 Click on the **Options** tab. Make sure that the **Reset Target** and **Load Application Image** options are selected, as shown.

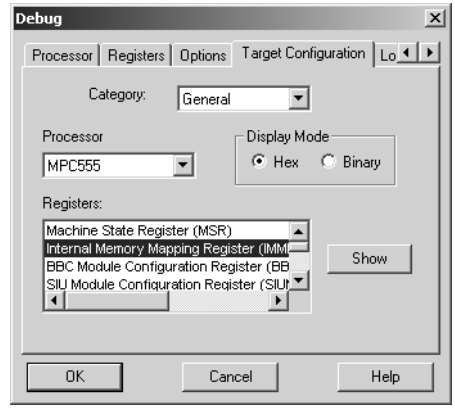


4 Click on the **Connection** tab. Choose parallel port or network settings appropriate to the physical connection you will be using between your PC and PhyCORE-MPC555 board. In the picture below, connection options are configured for the parallel port `LPT1`, with a **Delay** setting of `6`.

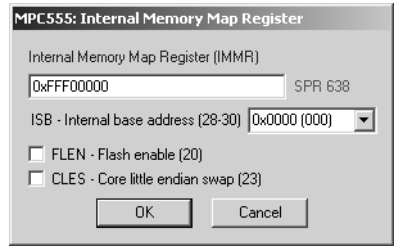
Note We have found that a **Delay** setting of at least 6 (shown) works best.



5 Click on the **Target Configuration** tab and select `General` from the **Category** menu. Then select `Internal Memory Mapping Register` from the **Registers** menu, as shown.

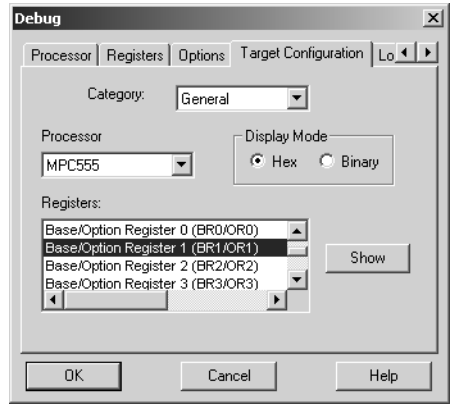


6 Click on the **Show** button to open the **Internal Memory Mapping Register** dialog. Clear the **FLEN-flash enable** option, as shown below.

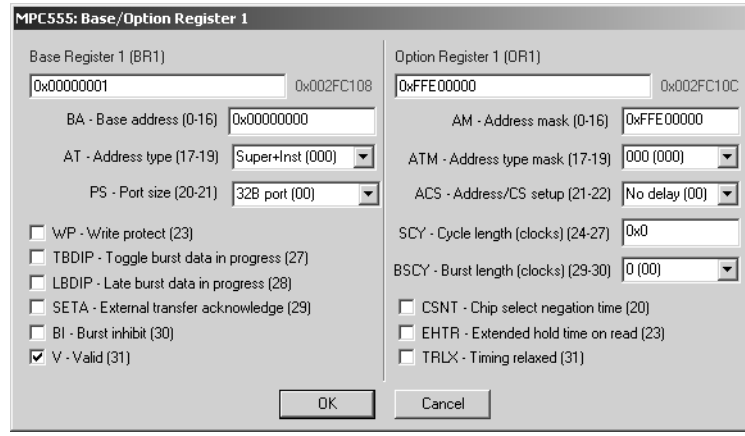


7 Click **OK** to return to the Target Configuration pane, and select `Base/Option Register 1` from the **Registers** menu, as shown.

Note Be sure to click **OK**, not **Cancel**, when closing the **Internal Memory Mapping Register** dialog box. Otherwise, SingleStep may use different parameter values than those displayed in the dialog box.

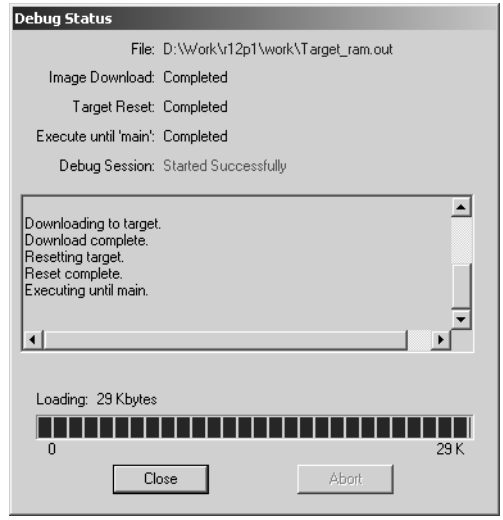


8 Click on the **Show** button to open the **Base/Option Register 1** dialog. Set the **BA-Base address** field to `0x00000000`, as shown below.



9 Click **OK** to return to the **Target Configuration** pane.

10 In the **Target Configuration** pane, click **OK.** SingleStep attempts to connect to the processor, and displays a **Debug Status** window. This picture shows the **Debug Status** window after a successful connection and download.



If you see error messages, consult the SingleStep documentation to troubleshoot the connection, or contact Wind River Systems for technical support.

11 Click **Close** to dismiss the **Debug Status** window.

12 At this point, the application code is in RAM, and SingleStep has established a debugging session. To execute the program, click on the green **Go** button in the SingleStep Debug panel. Observe that the red and green LEDs on the target board are blinking, indicating that the program is running.

13 You can now stop, restart, and otherwise exercise the target program via SingleStep.

Note Do not reset or cycle power on the board while the target program is running; doing so will hang SingleStep. You must stop the target program before exiting SingleStep.

Downloading the Application to RAM via CAN

The Download Control Panel utility can be used to download application code to MPC555 RAM or to MPC555 flash memory.

In this section, we will use the Download Control Panel utility to download the generated `Target_ram.srec` file to RAM on the target system.

Do the following before you begin:

- Make sure that your Vector-Informatik CAN card and drivers are installed and are operating properly. Make sure that the desired CAN port on the PC card is connected to the CAN A port on the target hardware. See your Vector-Informatik documentation for instructions on installation and verification.

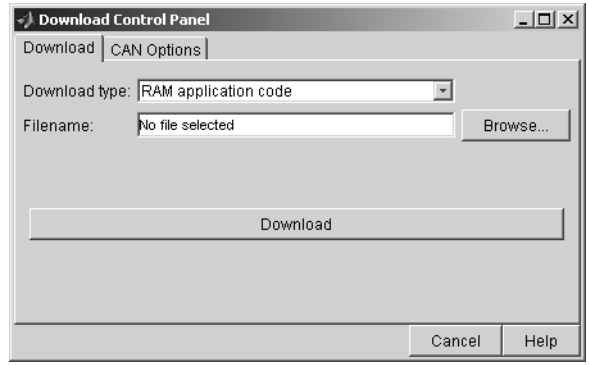
  In addition, after installing the drivers from Vector-Informatik, you must locate the `vcand32.dll` library, and add a path to the location where `vcand32.dll` was installed to the Windows system path.
- Make sure that you have set up SingleStep and downloaded boot code to the flash memory of the MPC555, as described in "Configuring SingleStep and Downloading Boot Code" on page 1–22.
- Make sure that nothing is connected to the BDM port of your development board.
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in "Phytec Jumper Settings" on page 1–12.
- Cycle the power (or perform a hard reset) on your development board, to clear the RAM.

To download the generated `Target_ram.srec` file to RAM:

1. Type the following command at the MATLAB command prompt:

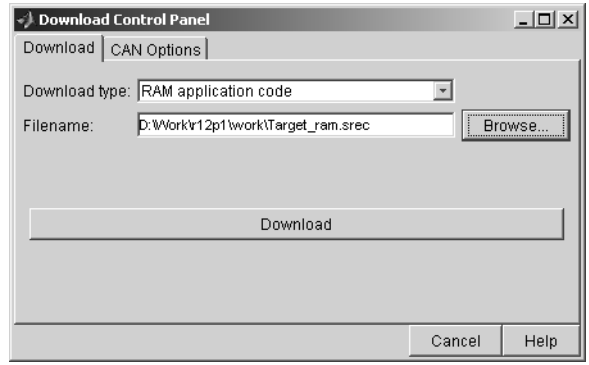
   ```
   candnload
   ```

2. The **Download Control Panel** window opens.

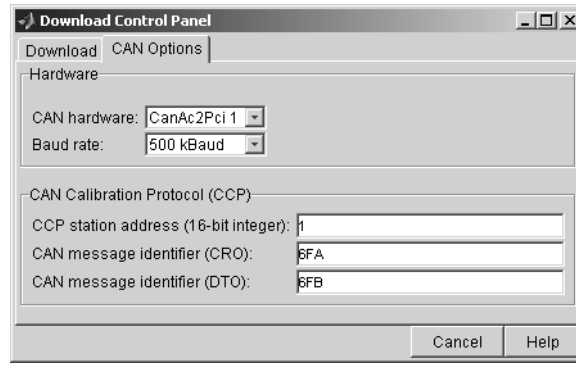


3 Select `RAM application code` from the **Download type** menu.

4 Enter the name of the file to be downloaded into the **Filename** field. Alternatively, you can use the **Browse...** button to navigate to the desired file. The **Download Control Panel** should now appear as shown in this picture.



5 Click on the **CAN Options** tab. If necessary, select an appropriate card and port from the **CAN hardware** pop-up menu. The default settings for the other parameters are appropriate for most cases. This picture shows the **CAN Options** configured for a Vector-Informatik CANAC2pci card, channel 1.



6 Click on the **Download** tab. Then click on the **Download** button, and immediately press the reset button on your PhyCORE-MPC555 board.

Downloading commences, and the **Download** button caption changes to **Stop**.

7 While downloading proceeds, numerous messages are displayed in the MATLAB command window. A successful download ends with the following messages:

```
send_connect()
send_program_prepare()
send_dnload(0)
STOP
```

After the download, the **Stop** button caption changes back to **Download**.

If the download does not succeed, reset your development board and return to step 6.

8 Close the **Download Control Panel** window.

9 A few seconds after a successful download, the boot code transfers control to the application program. At this point, you should see two LEDs (red and green) blinking on the target board. This indicates that the program is operating correctly.

Note that you can monitor the progress of the flash download using a program such as CANalyzer. Alternatively, you can use the `btest32` utility supplied with the Vector Informatik driver software. You can invoke the `btest32` utility from the PC command prompt. The following example runs `btest32` with a baud rate of 500000:

```
btest32 500000
```

Downloading Boot and Application Code

RAM vs. Flash Memory

The Embedded Target for Motorola MPC555 (real-time target) writes a file containing the application executable code that must be programmed onto the MPC555. It also writes a file of symbolic information suitable for use with a debugger. The files are written to your working directory.

The format of the code and symbol files depends on your selection from the **Target memory model** menu in the **MPC555-DK (real-time) options** category of the Real-Time Workshop pane, as follows:

- If you select the `FLASH` option, files in a format suitable for downloading into on-chip flash memory are generated. The naming convention for these files is *`model`*`_flash.bin` (code only), and *`model`*`_flash.out` (debugging symbols). You can download code to flash memory via the CAN bus, or via the MPC555's BDM port.
- If you select the `RAM` option, files in a format suitable for downloading into RAM are generated. The naming convention for these files is *`model`*`_ram.srec` (code only), and *`model`*`_ram.out` (code plus debugging symbols). You can download code to RAM via the CAN bus, or via the MPC555's BDM port.

There are advantages and disadvantages to each memory model.

Loading the application code into RAM is faster than loading it into flash memory. In addition, by using RAM you can avoid using up the programming cycles of the flash memory; this lengthens the usable lifetime of the flash memory. Running the application from RAM is a good option for initial testing of the application.

To program applications into RAM, your target hardware must have additional RAM external to the MPC555 on-chip RAM. The Embedded Target for Motorola MPC555 does not support downloading of code to the MPC555 on-chip RAM, because the MPC555 has only 26K of on-chip RAM.

For final deployment, or to load code onto a test board for use at a test site, you will generally want to program your code into the nonvolatile flash memory. 416K of flash memory is available for application code. Code programmed into flash memory is persistent and restarts when the board is powered on.

To download code to flash memory, you must first load a binary boot code file into the flash memory. The Embedded Target for Motorola MPC555 provides the boot code file. You must load the boot code into flash memory in order to run application code from flash memory; the boot code is also required if you want to program the MPC555 via CAN.

To understand the download process, it is first necessary to review the memory organization on the MPC555 and the operation of the boot code. This is described in "Overview of Memory Organization and the Boot Process" on page 3-24. If you just want to get started without reading about how the process works, you can jump ahead to the section "Downloading Boot Code" on page 3-25.

Overview of Memory Organization and the Boot Process

Memory Organization

The MPC555 has a total of 448K of on-chip flash memory. This memory is organized into 14 banks of 32K each. The first bank is always used to store the boot code and the remaining 416K is available for application code. When using the Embedded Target for Motorola MPC555, the on-chip flash memory is located at absolute address `0x0000` in the MPC555 address space.

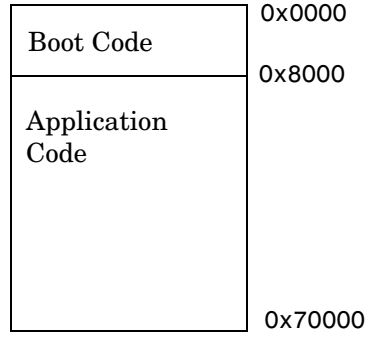


Figure 3-3: Organization of Flash Memory

To run a stand-alone application on the MPC555, it is first necessary to program the boot code into the first bank of flash memory.

The Boot Process

The boot code is executed following power on or reset (except if a probe is connected to the BDM port). Normally, the boot code performs basic hardware initialization and then branches to the application code. Once the application code is running, there is no way to return to the boot code except by performing a reset.

One of the important functions of the boot code is to serve as agent that allows program code to be downloaded over CAN. There are two methods of initiating a program download over CAN:

- The default method for initiating a flash download is to send a special CAN message during a short window of time while the boot code is executing. In the supplied boot code, this window is set to 40ms. If this special message is received during the window while the boot code is executing, a program download sequence commences and a new application can be programmed into flash memory. See “Downloading Application Code to Flash Memory via CAN” on page 3-29 for details.
- Alternatively, it is possible to commence a flash download over CAN while application code is running on the target. To initiate a download over CAN, you must include a special Simulink block in your Simulink model. See “Downloading Boot or Application Code via CAN Without Manual CPU Reset” on page 3-32 for details.

Downloading Boot Code

The Embedded Target for Motorola MPC555 provides the boot code in the file `bootcode_flash.bin`, located in the directory

```
matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\bootcode
```

Normally, you will only need to program the boot code into flash memory once. After this is done, new application code can be downloaded as often as required without any changes to the boot code.

The first time you program the boot code into the target hardware, you must download it via the BDM port. However, if existing boot code is already programmed into flash memory and must be replaced (for example, with a

newer or modified version) it is possible to download over CAN. In this case the boot code actually replaces itself.

Downloading Boot Code via BDM

A variety of proprietary tools are available for flash programming the MPC555 over the BDM port. In the section "Configuring SingleStep and Downloading Boot Code" on page 1-22, we describe how to program the flash memory using the Wind River Systems SingleStep debugger.

Downloading Boot Code via CAN

You can use the Download Control Panel to download new boot code to the MPC555, providing that existing boot code is already programmed onto the target hardware. (Recall that the first time boot code is programmed, it must be downloaded via BDM). Normally, you would not need to replace existing boot code unless there is an upgrade to the boot code supplied with this product.

To replace existing boot code:

1. Type the following command at the MATLAB command prompt:

   ```
   candnload
   ```

2. The **Download Control Panel** window opens.

3. Select `Flash boot code` from the **Download type** menu.

4. In the **Filename** field, enter the name of the boot code image file that you want to download. Alternatively, you can use the **Browse...** button to navigate to the desired file.

5. Click the **CAN Options** tab. If necessary, select an appropriate card and port from the **CAN hardware** pop-up menu. The default settings for the other parameters are appropriate for most cases.

6. The next step is to download code. The default method for download over CAN requires that you manually reset the target processor in order for the download process to begin. However, under the following conditions, you do *not* have to reset the target processor manually:

   a. There is an application currently running on the target.

b The application running on the target implements the CAN Calibration Protocol (as described in "Downloading Boot or Application Code via CAN Without Manual CPU Reset" on page 3-32).

If *both* these conditions are met, click on the **Download** tab, and then click on the **Download** button. Then skip to Step 8.

7 If the conditions a and b given in Step 6 are not met, you must download by the default method. Click on the **Download** tab. Then click on the **Download** button, and immediately press the reset button on your PhyCORE-MPC555 board.

8 Downloading commences, and the **Download** button caption changes to **Stop**.

9 Close the **Download Control Panel** window when downloading is complete.

Downloading Application Code

The following sections describe how to download generated image files and run generated code on the target hardware. They also describe how to download to RAM and to flash memory, via either the BDM port, or via CAN.

Downloading the Application Code to RAM

To download application code to RAM, you must generate a code file in Motorola S-record format, which is suitable for downloading and execution in RAM. To do this, select the `RAM` option from the **Target memory model** menu in the **MPC555-DK (real-time) options** category of the Real-Time Workshop pane. The build process creates two files in the working directory:

- *`model`*`_ram.srec`: Code only, without symbols, suitable for execution on the target system.
- *`model`*`_ram.out`: Code and symbols, suitable for use with a symbolic debugger such as Wind River SingleStep.

You can download to RAM:

- Via CAN, using the Download Control Panel utility with Vector-Informatik hardware, as described in "Downloading the Application to RAM via CAN" on page 3-19

- Via the BDM port, using the Wind River SingleStep debugger, as described in "Downloading the Application to RAM via SingleStep/BDM" on page 3-13

Downloading the Application Code to Flash Memory

To download application code to flash memory, you must generate a code file in flash binary format, which is suitable for downloading and execution in flash memory. To do this, select the `FLASH` option from the **Target memory model** menu in the **MPC555-DK (real-time) options** category of the Real-Time Workshop pane. The build process creates the files *`model`*`_flash.bin` and *`model`*`_flash.out` in the working directory. The *`model`*`_flash.bin` file contains an image of the executable code. The *`model`*`_flash.out` file contains symbols suitable for use with a symbolic debugger such as Wind River SingleStep.

You can download to flash memory:

- Via CAN, using the Download Control Panel utility with Vector-Informatik hardware, as described in "Downloading Application Code to Flash Memory via CAN" on page 3-29.
- Via the BDM port, using the Wind River SingleStep debugger, as described in "Downloading the Application Code to Flash Memory via BDM and SingleStep" on page 3-28.

Downloading the Application Code to Flash Memory via BDM and SingleStep

You can use SingleStep to download application code to flash memory via the BDM port. The procedure is almost identical to that for downloading boot code to flash memory.

If you have not created shortcuts to SingleStep, as described in "Configure Shortcuts to SingleStep" on page 1-22, do so before proceeding. Then, follow the download procedure given in "Configuring SingleStep and Downloading Boot Code" on page 1-22, with the following differences:

1. Start SingleStep using the shortcut file `SingleStep On Chip (MPC5xx) for Flash`.

2. In the **Setup** tab of the **Flash Programmer** dialog box, enter the session file name:

   ```
   matlabroot\toolbox\rtw\targets\mpc555dk\common\drivers\app_startup\app_flash.pts
   ```

3 The application code must be programmed into memory starting at the second bank of flash memory. On the Program/Verify pane of the **Flash Programmer** dialog box, set the **Bank, Sector** field to `1,1` and the **Address** field to `0x00008000`.

4 On the Program/Verify pane of the **Flash Programmer** dialog box, enter the name of your application, *`model`*`_flash.bin`, as the **S-record or Binary Image File**.

Downloading Application Code to Flash Memory via CAN

You can use the Download Control Panel to download generated application code to the MPC555 flash memory.

To download application code to flash memory, you must generate a code file in flash binary format, which is suitable for downloading and execution in flash memory. To do this, select the `FLASH` option from the **Target memory model** menu in the **MPC555-DK (real-time) options** category of the Real-Time Workshop pane. The build process creates the file *`model`*`_flash.bin` in the working directory.

Do the following before you begin:

- Make sure that your Vector-Informatik CAN card and drivers are installed, are operating properly. Make sure that the desired CAN port on the PC card is connected to the CAN A port on the target hardware. See your Vector-Informatik documentation for instructions on installation and verification.

  In addition, after installing the drivers from Vector-Informatik, you must locate the `vcand32.dll` library, and add a path to the location where `vcand32.dll` was installed to the Windows system path.
- Make sure that you have set up SingleStep and downloaded boot code to the flash memory of the MPC555, as described in “Configuring SingleStep and Downloading Boot Code” on page 1–22.
- Make sure that nothing is connected to the BDM port of your development board.
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in “Phytec Jumper Settings” on page 1–12.

To download the generated *`model`*`_flash.bin` file to RAM:

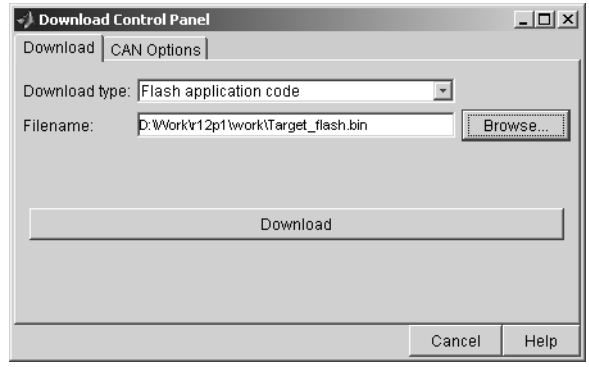
1 Type the following command at the MATLAB command prompt:

```
candnload
```

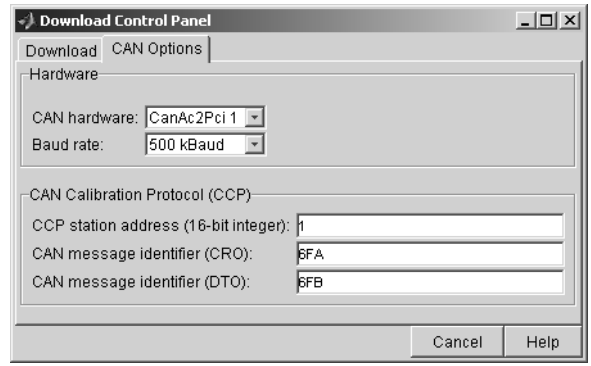
2 The **Download Control Panel** window opens.

3 Select `Flash application code` from the **Download** type menu.

4 Enter the name of the file to be downloaded into the **Filename** field. Alternatively, you can use the **Browse...** button to navigate to the desired file. The **Download Control Panel** should now appear as shown in this picture.



5 Click on the **CAN Options** tab. If necessary, select an appropriate card/port from the **CAN hardware** pop-up menu. The default settings for the other parameters are appropriate for the default boot process. This picture shows the **CAN Options** configured for a Vector-Informatik CAN-AC2-PCI card, channel 1.



6 The next step is to download code. The default method for download over CAN requires that you manually reset the target processor in order for the download process to begin. Under the following conditions, you do *not* have to reset the target processor:

   a There is an application currently running on the target.

   b The application running on the target implements the CAN Calibration Protocol (as described in "Downloading Boot or Application Code via CAN Without Manual CPU Reset" on page 3-32).

   If *both* these conditions are met, click on the **Download** tab, and then click on the **Download** button. Then skip to Step 8.

7 If the conditions a and b given in Step 6 are not met, you must download by the default method. Click on the **Download** tab. Then click on the **Download** button, and immediately press the reset button on your PhyCORE-MPC555 board.

8 Downloading commences, and the **Download** button caption changes to **Stop**.

9 While downloading proceeds, numerous messages are displayed in the MATLAB command window. A successful download ends with the following messages:

```
send_connect()
send_program_prepare()
send_dnload(0)
```

```
STOP
```

After the download, the **Stop** button caption changes back to **Download**.

If the download does not succeed, reset the board and return to step 6.

You can monitor the progress of the flash download using a program such as CANalyzer. Alternatively, you can use the `btest32` utility supplied with the Vector Informatik driver software. You can invoke the `btest32` utility from the PC command prompt. The following example runs `btest32` with a baud rate of 500000:

```
btest32 500000
```

10 Close the **Download Control Panel** window.

Once the download process is complete, the application starts running immediately on the target hardware.

Downloading Boot or Application Code via CAN Without Manual CPU Reset

The default method for download over CAN requires that the target processor be manually reset in order for the download process to begin. This requirement may be problematic if the target hardware is not physically accessible or if it cannot be individually reset or powered down/up.

It is possible to remove this requirement for manual reset if a suitably prepared application is already running on the target. To do this, include a CAN Calibration Protocol block within the model (See “CAN Calibration Protocol” on page 5-12).

When the currently running application includes the CAN Calibration Protocol block, the download process begins immediately when you click on the **Download** button of the Download Control Panel; it is not necessary to manually reset the target hardware to initiate the download.

When using the CAN Calibration Protocol block, you must specify

- CAN message identifier for Command Receive Objects
- CAN message identifier for Data Transmit Objects
- Can Calibration Protocol Station Address

Note that the values specified may differ from the default values for these parameters that are programmed in the boot code. When performing the download procedure using the Download Control Panel, you must ensure that the parameters specified on the **CAN Options** tab match those specified in the currently running application.

For an example of how to use the CAN Calibration Protocol block, see the demo model `mpc555rt_ccp`.

Boot Code Parameters for CAN Download

The boot code parameters for download over CAN determine

- CAN baud rate
- CAN message identifier for Command Receive Objects (CRO)
- CAN message identifier for Data Transmit Objects (DTO)
- Can Calibration Protocol Station Address
- The duration of the window during which the boot code listens for a download command message

Table 3-1 shows the default values for these parameters. These defaults should be suitable for most applications.

Table 3-1: Default Boot Code Parameters

Parameter	Default Value
CAN baud rate	`500,000`
CCP station address	`1`
CAN message identifier (CRO)	`6FA`
CAN message identifier (DTO)	`6FB`
Duration of listening window	40 ms

You cannot change these default values except by modifying and recompiling the boot code. If it is absolutely necessary to do this, you can recompile the boot code as follows:

1. Open a PC Command Prompt window.

2. Check whether you have the environment variables `MATLABROOT` and `DIABROOT` defined by typing

   ```
   set MATLABROOT
   set DIABROOT
   ```

3. If these variables are not defined you must define them by typing the following commands:

   ```
   set MATLABROOT=matlabroot
   set DIABROOT=diabpath
   ```

   The paths *`matlabroot`* and *`diabpath`* specified in the commands should be the location where MATLAB and the Diab compiler are installed.

4. Recompile the boot code by typing the following at the PC Command Prompt:

   ```
   cd matlabroot/toolbox/rtw/targets/mpc555dk
   matlabroot/rtw/bin/win32/gmake -f rt_makefile
   ```

Note You should not make changes to the boot code without fully understanding the effect of your changes. Note also that the boot code may be changed without notice in future releases of this product.

Generating ASAP2 Files

ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a standard description you use for data measurement, calibration, and diagnostic systems. The Embedded Target for Motorola MPC555 real-time target lets you export an ASAP2 file containing information about your model during the code generation process.

Before you begin generating ASAP2 files with the Embedded Target for Motorola MPC555 real-time target, you should read the "Generating ASAP2 Files" section of the Real-Time Workshop Embedded Coder documentation. That section describes how to define the signal and parameter information required by the ASAP2 file generation process.

The process of generating an ASAP2 file from your model with Embedded Target for Motorola MPC555 real-time target is similar to that described in the Real-Time Workshop Embedded Coder documentation. However, there are certain differences and limitations. In the following sections, we describe these differences and limitations and how they affect the procedure for generating ASAP2 files.

The `mpc555rt_ccp` demo provides an example of the Embedded Target for Motorola MPC555 ASAP2 file generation feature.

Requirements and Limitations

Compiler-Specific Post-Processing Requirements

The Embedded Target for Motorola MPC555 generates an initial ASAP2 file during the code generation process. At this point, the addresses of signals and parameters on the target system are unavailable, since the code has not been compiled and linked. The initial ASAP2 file contains placeholders for the unresolved addresses.

To supply the required memory addresses, the generated code must be compiled and a compiler-generated MAP file must be created.

After the build process, if the Embedded Target for Motorola MPC555 real-time target detects the presence of the ASAP2 file and a MAP file in the required format, it performs a post-processing phase. During this phase, the MAP file is used to propagate the required address information back into the ASAP2 file.

MAP file formats differ between compilers, so the post processing phase is compiler-specific. The Embedded Target for Motorola MPC555 provides the post-processing mechanism for the Diab compiler. Therefore, the Embedded Target for Motorola MPC555 real-time target supports generation of ASAP2 files only when used with the Diab cross-compiler.

To use the Embedded Target for Motorola MPC555 ASAP2 file generation feature, you must configure MATLAB via the `mpc555dk_asap2` utility. Before generating ASAP2 files, run the `mpc555dk_asap2` utility (described in the next section). This will ensure that the generated MAP and ASAP2 files are automatically post-processed.

The names of the ASAP2 file and the MAP file derive from the source model. The MAP file is generated in the same directory as the source model. The ASAP2 file is written to the build directory.

mpc555dk_asap2 Utility

The Embedded Target for Motorola MPC555 provides a setup utility, `mpc555dk_asap2`. Before generating an ASAP2 file, you must execute the `mpc555dk_asap2` utility as follows:

```
mpc555dk_asap2 option
```

where *`option`* is either `standard` or `mpc555`. The default is `standard`.

The *`option`* parameter determines how the MAP file and ASAP2 file are post-processed:

- `mpc555`: processes a MAP file generated by the Diab cross-compiler in the Diab format. This is the appropriate option for use with the Embedded Target for Motorola MPC555 real-time target, since the real-time target supports only the Diab compiler.
- `standard`: processes the MAP file with a standard Perl script (`asap2post.pl`) provided with Real-Time Workshop Embedded Coder. This script is *not* suitable for use with Embedded Target for Motorola MPC555.

We recommend that you add a call to `mpc555dk_asap2` in your `startup.m` file.

ASAP2 File Generation Procedure

1. Set up for ASAP2 file generation by running the `mpc555dk_asap2`, as described above.

2. Create the desired model. Use appropriate parameter names and signal labels to refer to ASAP2 `CHARACTERISTICS` and `MEASUREMENTS` respectively.

3. Define the corresponding `ASAP2.Parameter` and `ASAP2.Signal` objects in the MATLAB workspace.

4. Configure the data objects to generate unstructured global storage declarations in the generated code by assigning one of the following storage classes to the `RTWInfo.StorageClass` property:

   - `ExportedGlobal`
   - `ImportedExtern`
   - `ImportedExternPointer`

   `ExportedGlobal` is the default storage class.

5. Configure the other data object properties such as `LongID_ASAP2`, `PhysicalMin_ASAP2`, etc.

6. In the Advanced pane of the **Simulation Parameters** dialog box, select the **Inline parameters** option.

   Note that you should *not* configure the parameters associated with your data objects in the **Model Parameter Configuration** dialog box. If a parameter that resolves to a Simulink data object is configured using the **Model Parameter Configuration** dialog box, the dialog box configuration is ignored. You can, however, use the **Model Parameter Configuration** dialog to configure other parameters in your model.

7. In the Real-Time Workshop pane of the **Simulation Parameters** dialog, select `ERT code generation options(2)` from the **Category** menu. Then select the **Generate ASAP2 file** option.

8. Click **Apply**.

9. Click **Build** (or **Generate code**).

10. The ASAP2 file is generated as part of the build process.

Data Acquisition (DAQ) List Configuration

The Embedded Target for Motorola MPC555 supports the Data Acquisition (DAQ) List feature of the CAN Calibration Protocol (CCP). DAQ lists allow efficient synchronous signal monitoring. The CCP block provided with the Embedded Target for Motorola MPC555 supports DAQ lists (see "CAN Calibration Protocol" on page 5-12 for details).

`ASAP2.Signal` objects are used for monitoring a signal in the CCP polling mode of operation. To monitor a signal in a DAQ list, however, you must configure the signal somewhat differently. The differences are as follows:

- Instead of defining an `ASAP2.Signal` in the MATLAB workspace (and associated signal in the Simulink model), define a `canlib.Signal` object instead.
- There is no need to set the `RTWInfo.StorageClass` property of the `canlib.Signal` object. By default, the storage class is set to `Custom`.
- You should enter data in the other fields of the `canlib.Signal` object in the same way you would do for an `ASAP2.Signal` object.

During code generation, the Embedded Target for Motorola MPC555 automatically determines how to configure the DAQ lists in the generated code. For each distinct sample rate (of the set of `canlib.Signal` objects assigned by the user) one DAQ list in the model is created. The CCP DAQ List Object Descriptor Tables (ODTs) are shared equally between the created DAQ lists.

The sample rates of the `canlib.Signal` objects are mapped to CCP event channels in an extra file, `DAQ_LIST_EVENT_MAPPINGS`, that is generated in the build directory. This shows how to assign event channels to `MEASUREMENT` signals in a calibration package.

The event channels periodically transmit events that are used to trigger the sending of DAQ data to the host. By assigning event channels as defined in `DAQ_LIST_EVENT_MAPPINGS`, consistent and efficient transmission of DAQ data is achieved.

It is the responsibility of the calibration tool (see "Compatibility with Calibration Packages" on page 5-15) to assign an event channel and data to the available DAQ lists using CCP commands, and to interpret the synchronous response.

It is the responsibility of the user to make sure the calibration tool is set up correctly and that the event channels assigned to `MEASUREMENT` signals correspond to those defined in the file `DAQ_LIST_EVENT_MAPPINGS`.

Summary of the Real-Time Target

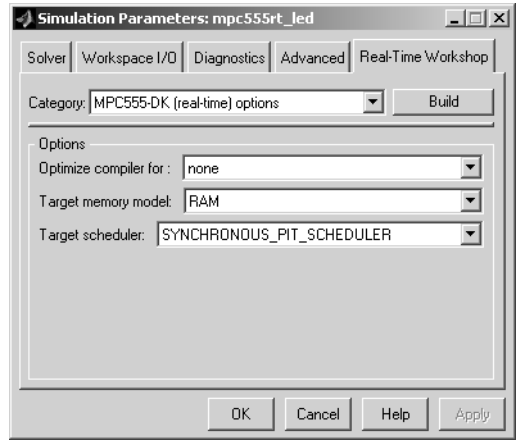
Code Generation Options

The real-time target is an extension of the Real-Time Workshop Embedded Coder embedded real-time (ERT) target configuration. The real-time target 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 **Category** menu of the Real-Time Workshop pane of the **Simulation Parameters** dialog box; they are documented in the Real-Time Workshop documentation and the Real-Time Workshop Embedded Coder documentation.

Some code generation options of the ERT target are not relevant to the real-time target, and are either unsupported, or restricted in their operation, by the PIL target. See “Requirements and Restrictions” on page 3-42 for details.

Target-Specific Options

The real-time target has several target-specific code generation options. To view or change the setting of these options, select **MPC555-DK (real-time) options** from the **Category** menu of the Real-Time Workshop pane of the **Simulation Parameters** dialog box. This picture shows the options at their default settings.



The options are

- **Optimize compiler for**: This option controls compiler optimization switches used during the build process. The exact effect of the optimization switches depends on whether you are using the Diab or CodeWarrior compiler. You can optimize for performance via the `speed` option, or optimize for code size via the `size` option. If you select `none`, no optimization switches are invoked.

- **Target memory model**: Select either `FLASH` or `RAM`.

  If you select the `FLASH` option, files in a format suitable for downloading into the MPC555 on-chip flash memory are written. The naming convention for these files is *`model`*`_flash.bin` (code only), and *`model`*`_flash.out` (debugging symbols).

  If you select the `RAM` option, files in a format suitable for downloading into RAM is generated. The naming convention for these files is *`model`*`_ram.srec` (code only), and *`model`*`_ram.out` (code plus debugging symbols).

- **Target scheduler**: This option controls the scheduling algorithm used by the main program provided with the Embedded Target for Motorola MPC555. To understand this option, refer to

  - The source code:
    `matlabroot/toolbox/rtw/targets/mpc555dk/mpc555dk/mpc555dk_main.c`
  - The "Data Structures and Program Execution" section of the Real-Time Workshop Embedded Coder documentation.

The default option is `SYNCHRONOUS_PIT_SCHEDULER`. When you select this option, the main program operates as described in the "Data Structures and Program Execution" section of the Real-Time Workshop Embedded Coder documentation. The `main()` function installs `rtOneStep` as a timer interrupt service routine (ISR). The IRQ level for `rtOneStep` is specified by the `RT_ONESTEP_IRQ_LEVEL` parameter of the model's MPC555 Resource Configuration block (see "MPC555 Resource Configuration" on page 5-47). The `main()` function also sets the rate of the MPC555 Programmable Interrupt Timer (PIT) and enables timer interrupts.

If you select the `USER_DEFINED_SCHEDULER` option, the `main()` function does not install any ISR, and runs as an idle loop with optional calls to a background task. You must supply your own timer ISR and execution scheduler and modify `mpc555dk_main.c` accordingly.

Requirements and Restrictions

MPC555 Resource Configuration Block Required

To generate code from a model using the Embedded Target for Motorola MPC555 real-time target, an MPC555 Resource Configuration block must be included in the model. The MPC555 Resource Configuration block is required even for models that do not contain any MPC555 device driver blocks.

Note When using device driver blocks from the Embedded Target for Motorola MPC555 libraries in conjunction with the MPC555 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the MPC555 Resource Configuration block will operate incorrectly. See "MPC555 Resource Configuration" on page 5-47 for further information.

Certain ERT code generation options are not supported by the real-time target. If these options are selected, the real-time target either ignores the option or issues an error message during the build process. Table 3-2 summarizes these restricted options.

Table 3-2: Real-Time Target Restricted Code Generation Options

Option	Restriction
MAT-file logging	Ignored if selected; build process proceeds
Create Simulink (S-function) block	Error if selected; build process terminates
External mode	Error if selected; build process terminates

Table 3-2: Real-Time Target Restricted Code Generation Options

Option	Restriction
Generate an example main program	This option should not be selected for the real-time target. The real-time target supplies a target-specific main program, `mpc555dk_main.c`.
Generate reusable code	Error if selected; build process terminates

4

Algorithm Export and Code Analysis Reporting

This section discusses two useful tools for code analysis:

Algorithm Export Target (p. 4-2)	The Algorithm Export (AE) target generates only the code that implements the algorithm of your model or subsystem. This is useful for code analysis and interfacing to hand-written or legacy code.
Code Analysis Reporting (p. 4-3)	This section describes the extended HTML code generation report.
Algorithm Export Target Summary (p. 4-5)	Summary of code generation options and restrictions.

Algorithm Export Target

The Embedded Target for Motorola MPC555 Algorithm Export (AE) target is an aid to code analysis and interfacing. The target generates only the code that implements the algorithm of your model or subsystem, without any overhead for PIL host/target communications or other operations extraneous to the model. Such purely algorithmic code is easier to interface to your hand-written or legacy code than code generated by the PIL or RT targets.

Another application of the AE target is to use it to produce a code generation report. Since only model code is included, you can more easily analyze the code generated from your model.

The AE target supports both the CodeWarrior and Diab cross-compilers, as specified in your target preferences (see "Setting Target Preferences" on page 1-15).

To use the AE target:

1. Open the **Simulation parameters** dialog box and select the **Real-Time Workshop** tab. Select `Target configuration` from the **Category** menu.

2. Click on the **Browse** button to open the **System Target File Browser**. In the browser, select **Embedded Target for Motorola MPC555 (algorithm export)** target. Then click **OK** to close the browser and return to the Real-Time Workshop pane.

3. Follow the usual procedure for generating code from your model or subsystem.

4. The AE target generates a project file appropriate for your cross-compiler. After generating the code and the project, the AE target automatically invokes your cross-compiler and opens the project.

We recommend using the AE target in conjunction with the Embedded Target for Motorola MPC555 HTML code generation report (see "Code Analysis Reporting" on page 4-3). If you select the **Generate HTML report** option, you can view a profiling report that includes detailed itemization of RAM and ROM usage for all code and data sections, and a complete memory map of the generated code. You can also easily examine the generated code via hyperlinks in the code generation report.

Code Analysis Reporting

The Embedded Target for Motorola MPC555 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, TLC 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.

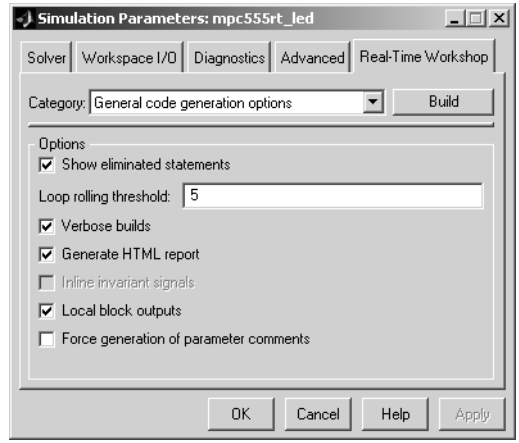
To generate a code generation report and view the profiling report:

1. Select the Real-Time Workshop tab of the **Simulation Parameters** dialog box. Select `Target configuration` from the **Category** menu. Make sure that the **Generate code only** option is not selected.

   The reason for this step is that the Embedded Target for Motorola MPC55 extended code generation report obtains information from MAP files that are created by your cross-compiler during the build process. If the **Generate code only** option is on, these files are not generated, which prevents the generation of the code generation report.

2. Select `General code generation options` from the **Category** menu.

3 Select **Generate HTML report**, as shown in this picture.



4 Follow the usual procedure for generating code from your model or subsystem.

5 The Real-Time Workshop writes the code generation report file in the build directory. The file is named *model*`_codegen_rpt.html` or *subsystem*`_codegen_rpt.html`.

6 The Real-Time Workshop automatically opens the MATLAB Help browser and displays the code generation report.

7 To view the profiling report, click on the **Code profile report** link in the Contents pane of the report.

Alternatively, you can view the code generation report in your Web browser.

Algorithm Export Target Summary

Code Generation Options

The AE target is an extension of the Real-Time Workshop Embedded Coder embedded real-time (ERT) target configuration. The AE target 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 **Category** menu of the Real-Time Workshop pane of the **Simulation Parameters** dialog box; they are documented in the Real-Time Workshop documentation and the Real-Time Workshop Embedded Coder documentation.

Some code generation options of the ERT target are not relevant to the AE target, and are either unsupported, or restricted in their operation, by the AE target. See "Restrictions" below for details.

The AE target has no target-specific options.

Restrictions

Certain ERT code generation options are not supported by the AE target. If these options are selected, the AE target either ignores the option or issues an error message during the build process. Table 4-1 summarizes these restricted options.

Table 4-1: AE Target Restricted Code Generation Options

Option	Restriction
MAT-file logging	Ignored if selected; build process proceeds
Create Simulink (S-function) block	Error if selected; build process terminates
Generate ASAP2 file	Ignored if selected; build process proceeds

Table 4-1: AE Target Restricted Code Generation Options (Continued)

Option	Restriction
External mode	Error if selected; build process terminates
Generate an example main program	This option should not be selected for the AE target. The AE target supplies a target-specific main program, `mpc555dk_main.c`.

5

Block Reference

This section contains the following topics:

The Embedded Target for Motorola MPC555 Block Libraries (p. 5-2)	Overview of the block libraries provided by the Embedded Target for Motorola MPC555.
Blocks Organized by Libraries (p. 5-4)	Block summaries and links to the block reference documentation, grouped by block library.
Alphabetical List of Blocks (p. 5-11)	Block summaries and links to the block reference documentation, in alphabetical order.

The Embedded Target for Motorola MPC555 Block Libraries

The Embedded Target for Motorola MPC555 provides three block libraries:

- The Embedded Target for Motorola MPC555 library (`mpc555drivers.mdl`) provides device driver blocks that let your applications access on-chip resources. The I/O blocks support the following features of the MPC555:
  - Pulse width modulation (PWM) generation or digital output via the Modular Input/Output Subsystem (MIOS) PWM unit.
  - Analog input via the Queued Analog-to-Digital Converter (QADC64).
  - Digital input and output via the MIOS.
  - Digital input via the QADC
  - Frequency and pulse width measurement via the MIOS Double Action Submodule (MDASM).
  - Transmission or reception of Controller Area Network (CAN) messages via the MPC555 TouCAN modules.
- The CAN Message Blocks library (`canblks.mdl`) provides device driver and utility blocks that support the Controller Area Network (CAN) protocol. CAN is an industry standard protocol used in automotive electronics and many other embedded environments where dispersed components require sharing of information. The CAN Message Blocks library includes blocks for transmitting, receiving, decoding, and formatting CAN messages. The CAN Message Blocks library also supports message specification via the Vector-Informatik CANdb standard.
- The CAN Drivers (Vector) library (`vector_candrivers.mdl`) provides blocks for configuring and connecting to Vector-Informatik CAN hardware and drivers.

The following sections provide complete information on each block in the Embedded Target for Motorola MPC555 block libraries, in a structured format. Refer to these pages when you need details about a specific block. Click **Help** on the **Block Parameters** dialog for the block, or access the block reference page through Help.

Using Block Reference Pages

Block reference pages are listed in alphabetical order by the block name. Each entry contains the following information:

- **Purpose** — describes why you use the block or function.
- **Library** — identifies the block library where you find the block.
- **Description** — describes what the block does.
- **Dialog Box** — shows the block parameters dialog and describes the parameters and options contained in the dialog. Each parameter or option appears with the appropriate choices and effects.
- **Examples** — optional section that provides demonstration models to highlight block features.

In addition, block reference pages provide pictures of the Simulink model icon for the blocks.

Blocks Organized by Libraries

The blocks in the Embedded Target for Motorola MPC555 libraries are organized into sublibraries that support different functions. The tables below reflect that organization.

Embedded Target for Motorola MPC555 Library

Top Level Library

Block Name	Purpose
MPC555 Resource Configuration	Support driver configuration for MPC555 and MIOS, QADC, and TouCAN submodules.
Watchdog	In event of application failure, time out and reset processor.

Note To generate code from a model using the Embedded Target for Motorola MPC555 real-time target, an MPC555 Resource Configuration block must be included in the model. The MPC555 Resource Configuration block is required even for models that do not contain any MPC555 device driver blocks.

Note When using device driver blocks from the Embedded Target for Motorola MPC555 libraries in conjunction with the MPC555 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the MPC555 Resource Configuration block will operate incorrectly. See “MPC555 Resource Configuration” on page 5-47 for further information.

Modular Input/Output System (MIOS1) Sublibrary

Block Name	Purpose
MIOS Digital In	Input driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM).
MIOS Digital Out	Output driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM).
MIOS Digital Out (MPWMSM)	Digital output via the MIOS Pulse Width Modulation Submodule (MPWMSM).
MIOS Pulse Width Modulation Out	Output driver for MIOS Pulse Width Modulation Submodule (MPWMSM).
MIOS Waveform Measurement	Support pulse width and pulse period measurement via MIOS Double Action Submodule.

Queued Analog-to-Digital Converter Module-64 Sublibrary

Block Name	Purpose
QADC Analog In	Input driver enables use of Queued Analog-Digital Converter (QADC64) in continuous scan mode.
QADC Digital In	Input driver enables use of QADC64 pins as digital inputs.

CAN 2.0B Controller Module (TouCAN) Sublibrary

Block Name	Purpose
CAN Calibration Protocol	Implement the CAN Calibration Protocol (CCP) standard.
TouCAN Error Count	Count transmit and/or receive errors detected on selected TouCAN modules.
TouCAN Fault Confinement State	Indicate the state of a TouCAN module.
TouCAN Interrupt Generator	Generate an interrupt subsystem for CAN interrupt sources.
TouCAN Receive	Receive CAN messages from a TouCAN module on the MPC555.
TouCAN Soft Reset	Reset a TouCAN module.
TouCAN Transmit	Transmit a CAN message via a TouCAN module on the MPC555.
TouCAN Warnings	Flag excessively high transmit or receive error counts on TouCAN modules.

Data Type Support and Scaling for Device Driver Blocks

The following table summarizes the input and output data types supported by the device driver blocks in the Embedded Target for Motorola MPC555 library, and the scaling applied to block inputs and outputs.

I/O Data Types and Scaling for MPC555 Device Driver Blocks

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
MIOS Digital In	Any Simulink supported datatype	logic 1 if input > 0, logic 0 if input <= 0	`Boolean`	0 or 1 only
MIOS Digital Out	Any Simulink supported datatype	logic 1 if input > 0, logic 0 if input <= 0	Any Simulink supported datatype	0 or 1 only
MIOS Digital Out (MPWMSM)	Any Simulink supported datatype	logic 1 if input > 0, logic 0 if input <= 0	Any Simulink supported datatype	0 or 1 only
MIOS Pulse Width Modulation Out	`double` or `single`	0 to 1	`double` or `single` (must be same as input data type)	0 to 1
MIOS Waveform Measurement	`double` or `single`	seconds	`double` or `single` (must be same as input data type)	Seconds
QADC Analog In	`double` or `single`	0 to 1	`uint16` or `int16` (defined by **Justification** parameter)	(defined by **Justification** parameter)
QADC Digital In	Any Simulink supported datatype	logic 1 if input > 0, logic 0 if input <= 1	`Boolean`	0 or 1 only

I/O Data Types and Scaling for MPC555 Device Driver Blocks

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
TouCAN Receive	`CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED` (must be same as output)	N/A	`CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED`	N/A
TouCAN Transmit	`CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED`	N/A	`CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED` (must be same as input)	N/A
TouCAN Warnings	`Boolean`	N/A	`Boolean`	N/A
TouCAN Error Count	`uint8`	N/A	`uint8`	N/A
TouCAN Fault Confinement State	`uint16`	N/A	`uint16`	N/A

Configuration Class Blocks

Each sublibrary of the Embedded Target for Motorola MPC555 library contains a *configuration class block* that has an icon similar to the one shown in this picture.

Note Configuration class blocks exist only to provide information to other blocks. *Do not copy these objects into a model under any circumstances.*

CAN Message Blocks and CAN Drivers Libraries

CAN Message Blocks

Block Name	Purpose
CAN Message Packing	Map Simulink signals to CAN messages.
CAN Message Packing (CANdb)	Pack Simulink double signals into CAN messages.
CAN Message Filter	Dispatch message processing based on message ID.
CAN Message Unpacking	Inspect and unpack the individual fields in a CAN message.
CAN Message Unpacking (CANdb)	Decompose a CAN frame into its constituent signals.

CAN Drivers (Vector)

Block Name	Purpose
Vector CAN Configuration	Configure a CAN channel (either hardware or virtual) for use with Vector-Informatik drivers.

CAN Drivers (Vector)

Block Name	Purpose
Vector CAN Receive	Read CAN frames from a Vector CAN channel.
Vector CAN Transmit	Transmit CAN frames on a Vector CAN channel.

Alphabetical List of Blocks

CAN Calibration Protocol 5-12
CAN Message Filter 5-17
CAN Message Packing 5-19
CAN Message Packing (CANdb) 5-21
CAN Message Unpacking 5-27
CAN Message Unpacking (CANdb) 5-29
MIOS Digital In 5-35
MIOS Digital Out 5-37
MIOS Digital Out (MPWMSM) 5-39
MIOS Pulse Width Modulation Out 5-41
MIOS Waveform Measurement 5-44
MPC555 Resource Configuration 5-47
QADC Analog In 5-59
QADC Digital In 5-64
TouCAN Error Count 5-67
TouCAN Fault Confinement State 5-68
TouCAN Interrupt Generator 5-70
TouCAN Receive 5-72
TouCAN Soft Reset 5-75
TouCAN Transmit 5-76
TouCAN Warnings 5-77
Vector CAN Configuration 5-78
Vector CAN Receive 5-83
Vector CAN Transmit 5-86
Watchdog 5-88

CAN Calibration Protocol

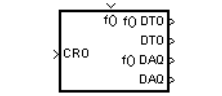
Purpose

Implement the CAN Calibration Protocol (CCP) standard

Library

Embedded Target for Motorola MPC555

Description



The CAN Calibration Protocol block provides an implementation of a subset of the CAN Calibration Protocol (CCP Version 2.1). CCP is a protocol for communicating between the target processor and the host machine over CAN. In particular, a calibration tool (see "Compatibility with Calibration Packages" on page 5-15) running on the host can communicate with the target, allowing remote signal monitoring and parameter tuning.

This block processes a Command Receive Object (CRO) and outputs the resulting Data Transmission Object (DTO) and Data Acquisition Messages (DAQ).

For more information on CCP refer to *ASAM Standards: ASAM MCD: MCD 1a* on the Association for Standardization of Automation and Measuring Systems (ASAM) Web site:

`http://www.asam.de`

Block Inputs and Outputs

The CAN Calibration Protocol block inputs are

- f(): a function call trigger input.
- CRO: a CAN message. The expected data source is a block such as a CAN Receive block. The message received at the CRO input is read when a trigger is received at the Fcn- Call input.

The block outputs are

- f() DTO: a function call trigger output.
- DTO: a CAN message. DTO should be read by the receiving block when it receives the f() DTOl trigger.
- f() DAQ: a function call trigger output.
- DAQ: a CAN message. DAQ should be read by the receiving block when it receives the f() DAQ trigger.

These inputs and outputs can be used to set up generic data I/O for the block.

Using the DAQ Output

The DAQ output is the output for any CCP Data Acquisition Lists that have been set up. You can use the ASAP2 file generation feature of the RT target to

- Set up signals to be transmitted using CCP Data Acquisition Lists.
- Assign signals in your model to a CCP Event Channel automatically (see "Generating ASAP2 Files" on page 3-35).

Once these signals are set up, Event Channels then periodically fire events that trigger the transmission of DAQ data to the host. When this occurs, CAN messages with the appropriate CCP / DAQ data appear on the DAQ output, along with an associated function call trigger.

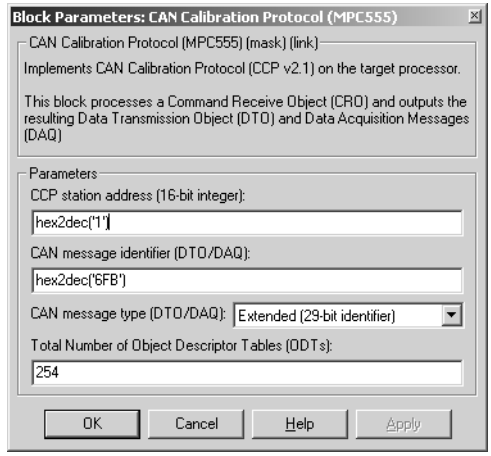
It is the responsibility of the calibration tool (see "Compatibility with Calibration Packages" on page 5-15) to use CCP commands to assign an Event Channel and data to the available DAQ lists, and to interpret the synchronous response.

Using DAQ lists for signal monitoring has the following advantages over the polling method:

- There is no need for the host to poll for the data. Network traffic is halved.
- The data is transmitted at the correct update rate for the signal. Therefore there is no unnecessary network traffic generated.
- Data is guaranteed to be consistent. The transmission takes place after the signals have been updated, so there is no risk of interruptions while sampling the signal.

Note The Embedded Target for Motorola MPC555 does not currently support event channel prescalers.

Dialog Box



CAN station address (16 bit integer)

The station address of the target. The station address is interpreted as a `uint16`. It is used to distinguish between different targets. By assigning unique station addresses to targets sharing the same CAN bus, it is possible for a single host to communicate with multiple targets.

CAN message identifier (DTO/DAQ)

The message identifier is the CAN message ID used for both CAN message block outputs. It is also used for transmitting messages to the host during the software-induced CAN download (soft boot). See "Extended Functionality" below.

CAN message type (DTO/DAQ)

The message type to be transmitted by the DTO and DAQ outputs. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

Total number of Object Descriptor Tables (ODTs)

CCP provides up to 254 Object Descriptor Tables (ODTs) that are shared between all available Data Acquisition Lists. This implementation shares these ODTs equally between the available DAQ Lists.

A single ODT uses 56 bytes of memory. Using all 254 ODTs would require over 14 KB of memory, a large proportion of the available memory on the target. If you need to conserve memory on the target, you can reduce the

total number of ODTs, allowing DAQ List signal monitoring with reduced memory overhead.

Supported CCP Commands

The following CCP commands are supported by the CAN Calibration Protocol block:

- `CONNECT`
- `DISCONNECT`
- `DNLOAD`
- `DNLOAD_6`
- `EXCHANGE_ID`
- `GET_CCP_VERSION`
- `GET_DAQ_SIZE`
- `GET_S_STATUS`
- `SET_DAQ_PTR`
- `SET_MTA`
- `SET_S_STATUS`
- `SHORT_UP`
- `START_STOP`
- `START_STOP_ALL`
- `TEST`
- `UPLOAD`
- `WRITE_DAQ`

Compatibility with Calibration Packages

The above commands support:

- Synchronous signal monitoring via calibration packages that use DAQ Lists
- Asynchronous signal monitoring via calibration packages that poll the target
- Asynchronous parameter tuning via CCP Memory Programming

This CCP implementation has been tested successfully with the Vector-Informatik CANape calibration package running in both DAQ List and Polling mode, and with the Accurate Technologies Inc. Vision calibration package running in DAQ List mode. (Note that Accurate Technologies Inc. Vision does not support the Polling mechanism for signal monitoring).

Extended Functionality

The CAN Calibration Protocol block also supports the PROGRAM_PREPARE command. This command is an extension of CCP that allows the automatic download of new code into the target memory. On receipt of the PROGRAM_PREPARE command, the target will reboot and begin the CAN download process. This lets you download new application code to RAM or flash memory, or download new boot code to flash memory.

Note The CAN message identifier of the incoming CCP messages (CRO), and that specified in the block mask for CAN message identifier (DTO/DAQ) are used as the CAN identifiers for the download process after a PROGRAM_PREPARE reboot. The type of CAN message used for this PROGRAM_PREPARE download process is always Extended (29-bit identifier).

Purpose

Dispatch message processing based on message ID

Library

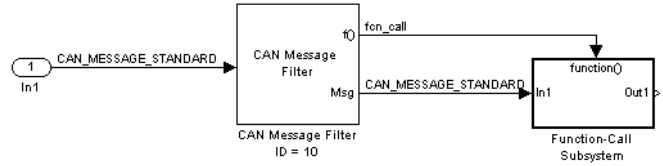
CAN Message Blocks

Description

The CAN Message Filter block lets you process CAN messages selectively, by message identifier (ID). It is possible that you could build a system where the message signal would have different IDs at different times. This may happen if you configure a TouCAN module to receive more than one ID per buffer. To take a different action depending on the ID, program a CAN Message Filter block with the ID you want to match. Then, connect the CAN Message Filter block's `fcn` output to the trigger input of a function call subsystem. The function call subsystem is triggered if the ID is matched.

You must program the function call system to receive and process the CAN message.

In the block diagram below, a CAN Message Filter block dispatches messages with ID 10 to a function call subsystem.



Dialog Box

CAN message identifiers to match

Specify an ID or a vector of IDs. When these IDs are encountered in the input message, the function call is triggered and the message is passed to the output of the block.

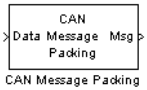
Purpose

Map Simulink signals to CAN messages

Library

CAN Message Blocks

Description



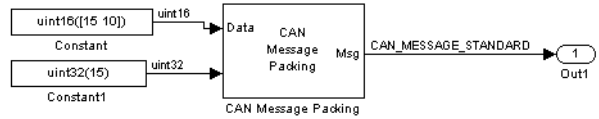
The CAN Message Packing block builds a CAN message. The CAN message type can be either Extended or Standard. You can set the message identifier statically or dynamically. The input port is dynamically typed and will accept any standard Simulink data types as input, as long as the total size does not exceed 8 bytes (64 bits).

The input port can accept

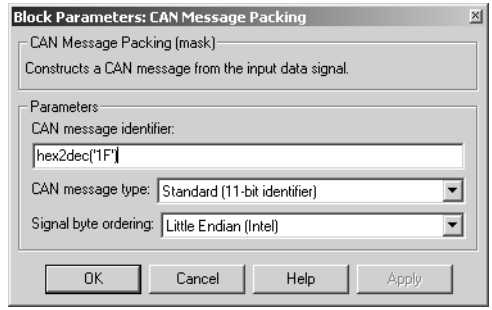
- `int8` or `uint8`: signals of width 1 - 8
- `int16` or `uint16`: signals of width 1 - 4
- `int32` or `uint32`: signals of width 1 - 2
- `single`: signals of width 1 - 2
- `double`: signals of width 1

The output data type is either `CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED`.

In this block diagram, a CAN Message Packing block accepts a `uint16` signal of width 2 (4 bytes). The lower input is a dynamic message ID (see “CAN message identifier” below).



Dialog Box



CAN message identifier

Set the identifier of the message. Note that an extended message has a 27 bit ID and a standard message has a 11 bit ID. If you specify that the value of the identifier is -1, then an extra input port on the block will appear. This lets you set the ID dynamically.

CAN message type

Specify the CAN message type: select either `Standard (11 bit identifier)` or `Extended (29 bit identifier)`.

Signal byte ordering

Signals are packed into the message from left to right. Within each signal, however, the byte order for signals of more than one byte is defined by the signal byte ordering. To use the CAN Message Packing and CAN Message Unpacking blocks correctly, both blocks must use the same signal byte ordering.

The ordering can be either `Little Endian` or `Big Endian`.

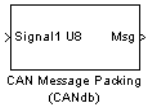
Purpose

Pack Simulink double signals into CAN messages

Library

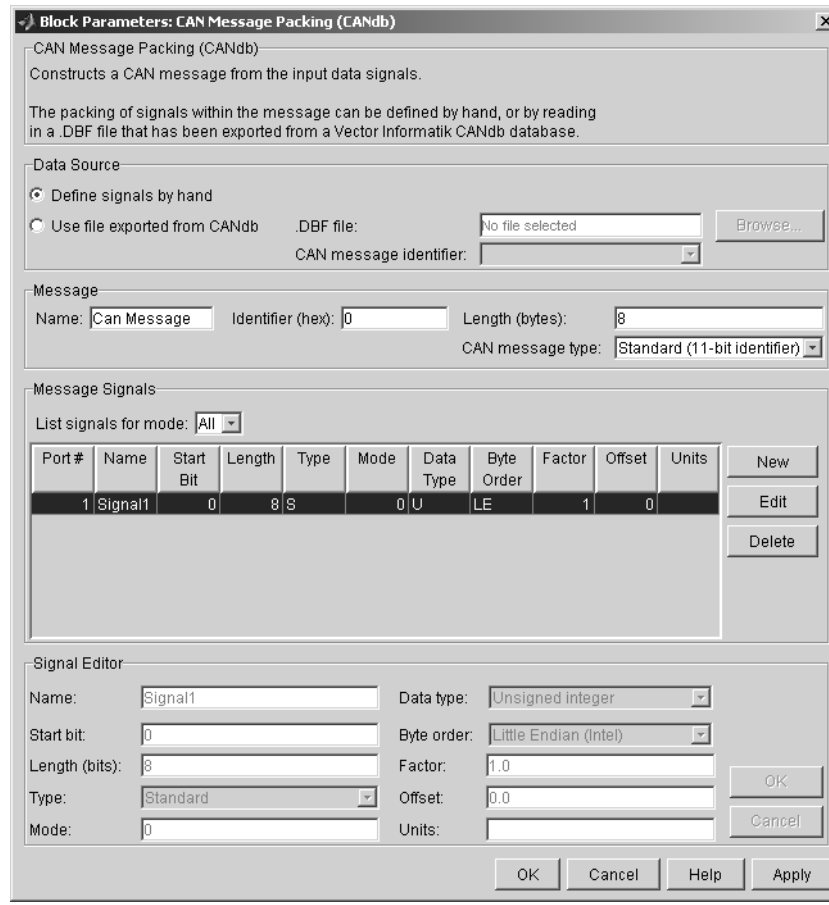
CAN Message Blocks

Description



The CAN Message Packing (CANdb) block gives you control over the packing of Simulink double signals into a CAN message. You can specify the scaling and offset of individual input signals and how many bits the signal will take up in the message.

Dialog Box



There are four panels to the **CAN Message Packing (CANdb)** dialog box.

Data Source Panel

Specify whether you want to define the message composition manually, or retrieve a message composition specification from a CANdb database. CANdb is a standard controlled by Vector-Informatik. To use CANdb you need to purchase a copy of CANdb from Vector-Informatik.

The software cannot read the CANdb format file directly. Instead you must export the file from CANdb as a DBASE (`.dbf`) format file and place it in your working directory.

Define signals by hand

When you select **Define signals by hand**, the **Message**, **Message Signals**, and **Signal Editor** panels are activated. You can then edit the message characteristics manually.

Use file exported from CANdb

Select this option if you want to retrieve a message composition specification from a CANdb database. Before using this feature, you must configure an ODBC data source and locate your database files in the MATLAB work directory. The procedure for doing this is given in “Configuring the ODBC Data Source for CANdb Blocks” on page 5-25.

When you select **Use file exported from CANdb**, the **Browse**... button and **.DBF file** field become active. You can then select a database file by browsing for it or by entering the file name in the **.DBF file** field. After you have selected a database file, the **CAN message identifier** menu displays a list of available messages (by identifier) in the database. Select the desired message.

Message Panel

This panel contains information that applies to the whole message.

Name

Name of the message

Identifier (hex)

The hexidecimal number that identifies the message on the CAN bus.

Length

Length of the message, in bytes

CAN message type

Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

Message Signals Panel

Provides a display-only list of information for each input signal to be packed into the message. The signal parameters displayed in this list are described in the "Signal Editor Panel" section below.

In addition to the list of signals, this panel contains three buttons:

- The **New** button lets you create a signal, which you can then edit in the **Signal Editor**.
- The **Edit** button activates the **Signal Editor** panel. This panel lets you edit the parameters of the selected signal.
- The **Delete** button lets you delete a signal.

The **List signals for mode** menu lets you restrict the list of signals to display only those signals with a given mode value. (See "Type and Mode" on page 5-24). By default, the mode is set to display all signals regardless of mode value. Note that when displaying signals for a specific mode, the **Signal Editor** panel is disabled.

The **Port#** column displays the number of each input signal. The number indicates the order in which the signals enter the Simulink block, from top to bottom.

The other columns display properties described in "Signal Editor Panel" on page 5-23.

Signal Editor Panel

Lets you enter or change parameter values for the selected signal. To edit a signal, select it in the Message Signals Panel, and then press the **Edit** button. The **Signal Editor** panel will become active. After you have made the required changes to the signal click **OK** to commit your changes. Click **Cancel** to finish editing without retaining changes.

Each parameter is described briefly here. Each signal is defined according to the CANdb standard; refer to your CANdb documentation for detailed information.

Name

The name of the signal that you see displayed in the CANdb and on the Simulink block.

Start bit

The bit position, in the data bytes, where the signal is inserted.

Length (bits)

The number of bits to be used in the message

Type and Mode

More than one signal can be mapped to the same location in a message. To determine which message is actually mapped, you can define at most one *mode signal*. The mode signal is an integer valued input. You can then define as many *mode dependent signals* as you want. A mode dependent signal also makes use of the **Mode** value. The **Mode** value is matched against the mode signal. If the **Mode** value equals the value of the mode signal then the signal is packed into the message. If it does not match then the signal is ignored.

Mode

See **Type and Mode** above.

Data type

This menu allows you to specify whether the bits allocated to a signal are interpreted as a signed (two's complement) or unsigned integer. For example, consider a 3-bit signal where all bits are set to 1. If the signal is interpreted as signed, it would represent the value -1 before scaling and offsetting. If the signal is interpreted as unsigned, it would represent the value 7 (`0x7h`) before scaling and offsetting.

Byte order

Select one of the following options:

- `Big Endian (Motorola)`: the start bit is the least significant bit of the least significant byte from the end of the message. Bytes are counted left from here.

- `Little Endian (Intel)`: the start bit is the least significant bit of the least significant byte from the beginning of the message. Bytes are counted right from here.

  The CAN Message Packing (CANdb) block uses *Motorola Backwards* big endian format, which is the default for CANdb. It does not uses *Motorola Forwards* big endian format, which is the default for CANdb++.

  For more detailed descriptions of byte layout, see your CANdb documentation.

Factor and Offset

Define the *Factor* and *Offset* values in the formula for conversion between the physical value (Simulink signal value) and the data stored in the packet. The conversion formula is defined as

Phys_value = (Raw_value - Offset) / Factor

Where *Raw_value* is the value stored in the packet. Note that the *Raw_value* may be signed or unsigned depending on the **Data type**.

Units

This field is informational only. If desired, enter the appropriate units for the signal.

Configuring the ODBC Data Source for CANdb Blocks

The CAN Message Packing (CANdb) and CAN Message Unpacking (CANdb) blocks let you retrieve a message composition specification from a CANdb database. To use this feature, you must configure an ODBC data source, specifying the MATLAB work directory as the location for database files.

To configure the data source:

1. Locate the MATLAB work directory. At the MATLAB prompt, type

   ```
   what (work)
   ```

   The command returns the path to *`matlabroot`*`/work`, for example:

   ```
   files in directory D:\Work\r12p1\work
   ```

   Note this path for use in Step 7.

2. From the **Start** menu, open **Settings/Control Pane/Administrative Tools/Data Sources (ODBC)**.
3. The **ODBC Data Sources Administrator** window opens. Click the **User DSN** tab.
4. Click the **Add...** button. The **Create New Data Source** dialog box opens. Select `Microsoft dBase Driver (*.dbf)` from the list of drivers. Then click **Finish**.
5. The **ODBC dBase Setup** dialog box opens. Enter `dBase Sources for CAN` in the **Data Source Name** field. You must use this *exact* name.
6. Clear the **Use Current Directory** check box.
7. Click the **Select Directory** button and navigate to the MATLAB work directory that you located in Step 1. Click **OK**.
8. The path to the MATLAB work directory should now appear on the **ODBC dBase Setup** dialog box. Click **OK** to dismiss the **ODBC dBase Setup** dialog box.
9. Click **OK** to complete **ODBC dBase Setup**. Then click **OK** to close the **ODBC Data Sources Administrator** window.
10. Place the DBF files you want to work with in the MATLAB work directory. You can then use the **Use file exported from CANdb** option to access the CANdb information by using the CANdb blocks.

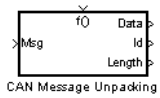
Purpose

Inspect and unpack the individual fields in a CAN message

Library

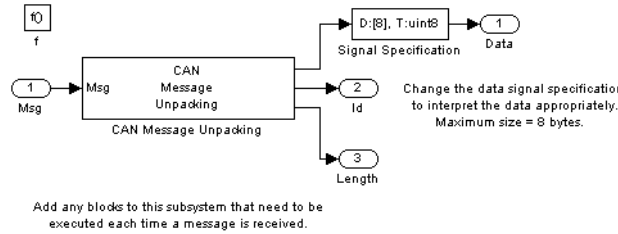
CAN Message Blocks

Description



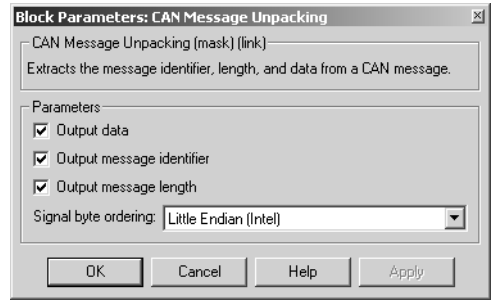
The CAN Message Unpacking block receives a CAN message at its input and (by default) outputs the ID, length, and data contained in the message.

Note that the CAN Message Blocks library provides the CAN Message Unpacking block embedded in a Fcn Call subsystem, as shown in this figure.When a message is received, a trigger should be provided to initiate unpacking of the message.



By default, the data output port outputs a `uint8` signal of width 1. To read the signal as a different data type with a different vector width, modify the Signal Specification block parameters.

Dialog Box



Output data

Extract the data of the CAN message as a signal. See the notes above on mapping the bytes of the CAN message to a Simulink signal.

Output message identifier

Extract the ID of the CAN message as a Simulink signal.

Output message length

Extract the length of the CAN message as a signal.

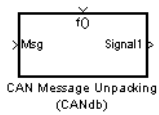
Purpose

Decompose a CAN message into its constituent signals

Library

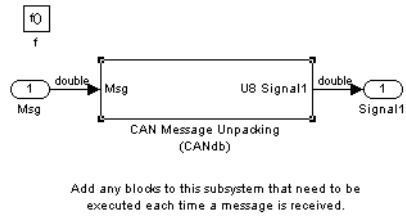
CAN Message Blocks

Description

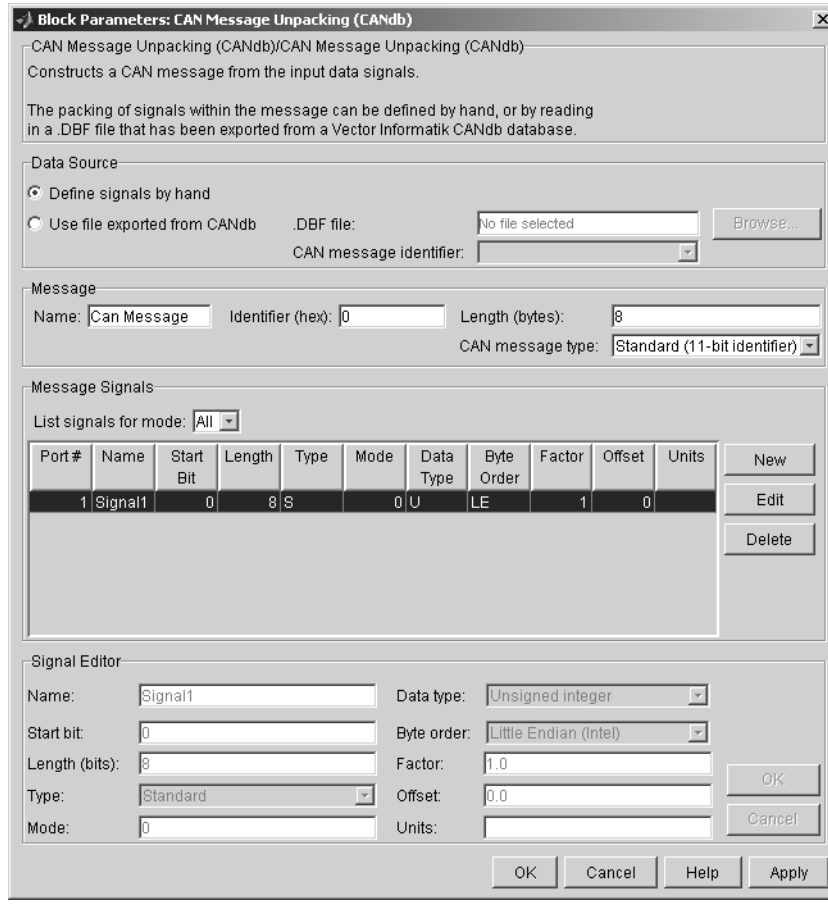


The CAN Message Unpacking (CANdb) block complements the CAN Message Packing (CANdb) block. The user interface for the block is almost identical. The difference is that the CAN Message Unpacking (CANdb) block accepts a CAN message as an input and decomposes it into individual signals.

Note that the CAN Message Blocks blocks library provides the CAN Message Unpacking (CANdb) block embedded in a Fcn Call subsystem, as shown in this figure.When a message is received, a trigger should be provided to initiate unpacking of the message.



Dialog Box



There are four panels to the CAN Message Unpacking (CANdb) dialog box.

Data Source Panel

Specify whether you want to define the message composition by hand or retrieve a message composition specification from a CANdb database. CANdb is a standard controlled by Vector-Informatik. To use CANdb you will need to purchase a copy from Vector-Informatik.

The software cannot read the CANdb format file directly. Instead in CANdb you must export the file as a DBASE (`.dbf`) format file and place it in your working directory.

Define signals by hand

When you select **Define signals by hand**, the **Message**, **Message Signals**, and **Signal Editor** panels are activated. You can then edit the message characteristics manually.

Use file exported from CANdb

Select this option if you want to retrieve a message composition specification from a CANdb database. Before using this feature, you must configure an ODBC data source and locate your database files in the MATLAB work directory. The procedure for doing this is given in "Configuring the ODBC Data Source for CANdb Blocks" on page 5-25.

When you select **Use file exported from CANdb**, the **Browse**... button and **.DBF file** field become active. You can then select a database file by browsing for it or by entering the filename in the **.DBF file** field. After you have selected a database file, the **CAN message identifier** menu displays a list of available messages in the database. Select the desired message.

Message Panel

Contains information that applies to the whole message.

Name

Name of the message

Identifier (hex)

The hexidecimal number that identifies the message on the CAN bus.

Length

Length of the message, in bytes.

CAN message type

Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

Message Signals Panel

Provides a display-only list of information for each input signal to be packed into the message. The signal parameters displayed in this list are described in the "Signal Editor Panel" section below.

In addition to the list of signals, this panel contains three buttons:

- The **New** button lets you create a signal, which you can then edit in the **Signal Editor**.
- The **Edit** button activates the **Signal Editor** panel. This panel lets you edit the parameters of the selected signal.
- The **Delete** button lets you delete a signal.

The **List Signals for Mode** menu lets you restrict the list of signals to display only those signals with a given mode value. (See "Type and Mode" on page 5-24). By default, **List Signals for Mode** is set to display all signals regardless of mode value.

The **Port#** column displays the number of each input signal. The number indicates the order in which the signals enter the Simulink block, from top to bottom.

The other columns display properties described in "Signal Editor Panel" on page 5-32.

Signal Editor Panel

Lets you enter or change parameter values for the selected signal. To edit a signal, select it in the Message Signals Panel, and then click the **Edit** button. The **Signal Editor** panel will become active. After you have made the required changes to the signal, click **Apply** or **OK** to commit your changes. Click **Cancel** to finish editing without retaining changes.

Each parameter is described briefly here. Each signal is defined according to the CANdb standard; refer to your CANdb documentation for detailed information.

Name

The name of the signal that you see displayed in the CANdb and on the Simulink block.

Start bit

Bit position, in the data bytes, where the signal is inserted.

Length (bits)

The number of bits to be used in the message

Type and Mode

More than one signal can be mapped to the same location in a message. To determine which message is actually mapped, you can define at most one *mode signal*. The mode signal is an integer valued input. You can then define as many *mode-dependent signals* as you want. A mode-dependent signal also makes use of the **Mode** value.The **Mode** value is matched against the mode signal. If the **Mode** value equals the value of the mode signal then the signal is packed into the message. If it does not match then the signal is ignored.

Mode Value

See **Type and Mode** above.

Data type

This menu allows you to specify whether the bits allocated to a signal are interpreted as a signed (twos complement) or unsigned integer. For example, consider a 3-bit signal where all bits are set to 1. If the signal is interpreted as signed, it would represent the value -1 before scaling and offsetting. If the signal is interpreted as unsigned, it would represent the value 7 (`0x7h`) before scaling and offsetting.

Byte order

Select one of the following options:

- `Big Endian (Motorola)`: the start bit is the least significant bit of the least significant byte from the end of the message. Bytes are counted left from here.
- `Little Endian (Intel)`: the start bit is the least significant bit of the least significant byte from the beginning of the message. Bytes are counted right from here.

The CAN Message Unpacking (CANdb) block uses *Motorola Backwards* big endian format, which is the default for CANdb. It does not uses *Motorola Forwards* big endian format, which is the default for CANdb++.

For more detailed descriptions of byte layout see your CANdb documentation.

Factor and Offset

Define the *Factor* and *Offset* values in the formula for conversion between the physical value (Simulink signal value) and the data stored in the packet. The conversion formula is defined as

*Phys_value = (Raw_value * Factor + Offset)*

Where *Raw_value* is the value stored in the packet. Note that the *Raw_value* may be signed or unsigned depending on the **Data type**.

Units

This field is informational only. If desired, enter the appropriate units for the signal.

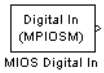
Purpose

Input driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM)

Library

Embedded Target for Motorola MPC555

Description



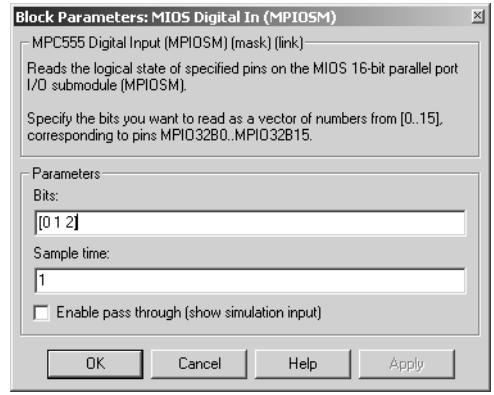
The MIOS Digital In block reads the state of selected pins (bits) on the MIOS 16-bit Parallel Port I/O Submodule (MPIOSM) of the MPC555. The **Bits** field specifies a vector of numbers in the range `0..15`, corresponding to pins `MPIO32B0..MPIO32B15` on the MPIOSM.

The output of the block is a wide vector representing the logic state of the pins referenced in the **Bits** field. When the signal on a given pin is a logical 1, the block output element will be equal to 1; otherwise the block output element will equal zero.

Refer to section 15.13, "MIOS 16-bit Parallel Port I/O Sub module (MPIOSM)," in the *MPC555 Users Manual* for further information.

Note You are responsible for ensuring that pin assignments of MIOS Digital In and MIOS Digital Out blocks in your model do not conflict. No error checking is performed to detect conditions where the same pin is referenced by both an input and an output block. If such a condition occurs, the behavior of the system is undefined.

Dialog Box



Bits

A vector of numbers in the range `0..15`. Each number corresponds to a pin (`MPIO32B0..MPIO32B15`) on the MPIOSM.

Sample time

Sample time of the block.

Enable pass through (show simulation input)

Lets you provide a signal to this block for use in simulation. When this option is enabled, an inport appears on the block. The block input is passed through to the output during simulation. (See “Data Type Support and Scaling for Device Driver Blocks” on page 5-6 for information on supported input/output data types and scaling of input/output signals.) This option affects simulation only.

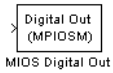
Purpose

Output driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM)

Library

Embedded Target for Motorola MPC555

Description



The MIOS Digital Out block sets the state of selected pins (bits) on the MIOS 16-bit Parallel Port I/O Submodule (MPIOSM) of the MPC555. The **Bits** field specifies a vector of numbers in the range `0..15`, corresponding to pins `MPIO32B0..MPIO32B15` on the MPIOSM.

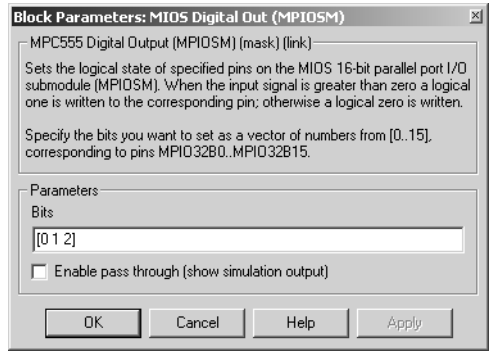
The input to the block is a wide vector with one signal element per pin. When the input signal is greater than zero, a logical 1 is written to the corresponding pin. When the input signal is less than or equal to zero, a logical zero is written to the corresponding pin.

If you want to write to several digital output pins at the same sample rate, using a single MIOS Digital Out block with a vector input signal will result in more efficient code. However, if you want to update different output pins at different sample rates, you must use a separate MIOS Digital Out block for each rate.

Refer to section 15.13, "MIOS 16-bit Parallel Port I/O Sub module (MPIOSM)," in the *MPC555 Users Manual* for further information.

Note You are responsible for ensuring that pin assignments of MIOS Digital In and MIOS Digital Out blocks in your model do not conflict. No error checking is performed to detect conditions where the same pin is referenced by both an input and an output block. If such a condition occurs, the behavior of the system is undefined.

Dialog Box



Bits

A vector of numbers in the range `0..15`. Each number corresponds to a pin (`MPIO32B0..MPIO32B15`) on the MPIOSM.

Enable pass through (show simulation input)

Lets you provide a signal from this block for use in simulation. When this option is enabled, an outport appears on the block. The block input is passed through, to the output during simulation. (See “Data Type Support and Scaling for Device Driver Blocks” on page 5-6 for information on supported input/output data types and scaling of input/output signals.) This option affects simulation only.

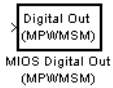
Purpose

Digital output via the MIOS Pulse Width Modulation Submodule (MPWMSM)

Library

Embedded Target for Motorola MPC555

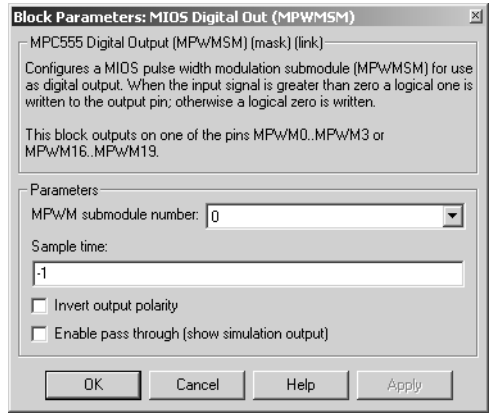
Description



The MIOS Digital Out (MPWMSM) block is a device driver that lets you use the MIOS Pulse Width Modulation Submodule (MPWMSM) in *digital output mode*. In digital output mode, the Pulse Width Modulation (PWM) feature of the MPWMSM is turned off. When the input signal is greater than zero, a logical 1 is written to the output pin; otherwise a logical zero is written.

Refer to section 15.12, "MIOS Pulse Width Modulation Submodule (MPWMSM)," in the *MPC555 Users Manual* for further information on the parameters described below.

Dialog Box



MPWM submodule number

Selects a PWM submodule for output.

Sample time

Sample time of the block.

Invert output polarity

Switches the output level for logic one and zero.

Enable pass through (show simulation input)

Lets you provide a signal from this block for use in simulation. When this option is enabled, an outport appears on the block. The block input is passed through to the output during simulation. (See “Data Type Support and Scaling for Device Driver Blocks” on page 5-6 for information on supported input/output data types and scaling of input/output signals.) This option affects simulation only.

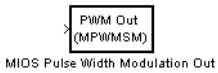
Purpose

Output driver for MIOS Pulse Width Modulation Submodule (MPWMSM)

Library

Embedded Target for Motorola MPC555

Description



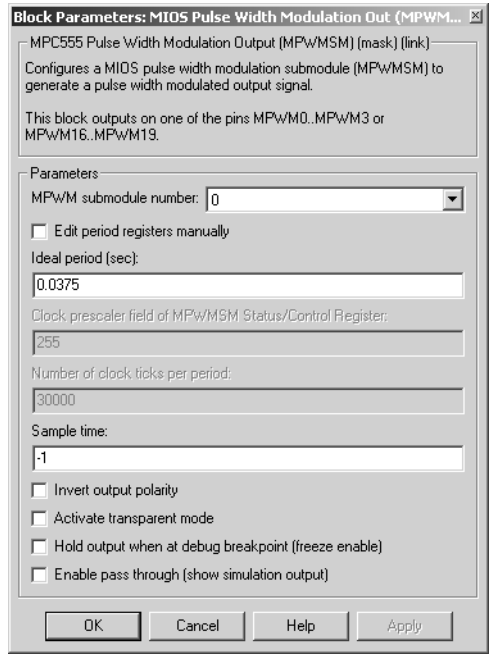
The MIOS Pulse Width Modulation Out block is used for Pulse Width Modulation (PWM) output from the MIOS Pulse Width Modulation Submodule (MPWMSM). A PWM signal is a rectangular waveform whose period is constant but whose duty cycle can be varied, under control of a modulator signal, between 0% and 100%.

The MIOS Pulse Width Modulation block input signal acts as the modulator, controlling the duty cycle of the signal on the output pin. The input signal is constrained to be between 0 and 1. When the input signal value is 0, the output signal's duty cycle is 0%. When the input signal value is 1, the output signal's duty cycle is 100%.

There are two possible methods for calculating the period of the waveform. You can either control the scaling registers directly, or enter the desired (ideal) period and the mask will solve for the best values for the scaling registers.

Refer to section 15.12, "MIOS Pulse Width Modulation Submodule (MPWMSM)," in the *MPC555 Users Manual* for further information on the parameters described below.

Dialog Box



MPWM submodule number

Selects a PWM submodule for output.

Edit period registers manually

When this option is selected, the **Clock prescaler field of MPWM Status/Control Register** and **Number of clock ticks per period** edit fields are activated. You can then set the PWM period by setting these values.

When this option is not selected, use the **Ideal period (sec)** field to set the PWM period parameters.

Ideal period (sec)

Specifies the desired period of the pulse signal. The mask then solves for the clock prescaler and the pulse period.

Clock prescaler field of MPWM Status/Control Register

Divides the counter clock to get the clock signal used to drive the PWM output. Note that the counter clock itself is derived from the MPC555 system clock as configured by the MPC555 Resource Configuration block (see "MPC555 Resource Configuration" on page 5-47).

Number of clock ticks per period

Determines the number of PWM counter ticks per pulse period. Valid values are `1 - 65535`.

Invert output polarity

Switches the output level for logic one and zero.

Activate transparent mode

Bypasses the register double buffers. When transparent mode is active, a software write to the Next Pulse Width Register is immediately transferred to the Pulse Width Register. When transparent mode is inactive, the updated value only takes effect at the start of the next period.

Hold output when at debug break point (freeze enable)

Stops the PWM counters when a breakpoint is hit during debug mode, and holds the current output values.

Enable pass through (show simulation input)

Lets you provide a signal from this block for use in simulation. When this option is enabled, an outport appears on the block. The block input is passed through to the output during simulation. (See "Data Type Support and Scaling for Device Driver Blocks" on page 5-6 for information on supported input/output data types and scaling of input/output signals.) Note that both the input and output signals are duty cycle values. This option affects simulation only.

MIOS Waveform Measurement

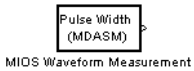
Purpose

Support pulse width and pulse period measurement via MIOS Double Action Submodule (MDASM)

Library

Embedded Target for Motorola MPC555

Description



MIOS Waveform Measurement

Waveform measurement is a feature the MIOS Double Action Submodule (MDASM) on the MPC555. The MIOS Waveform Measurement block currently implements the following features of the MDASM:

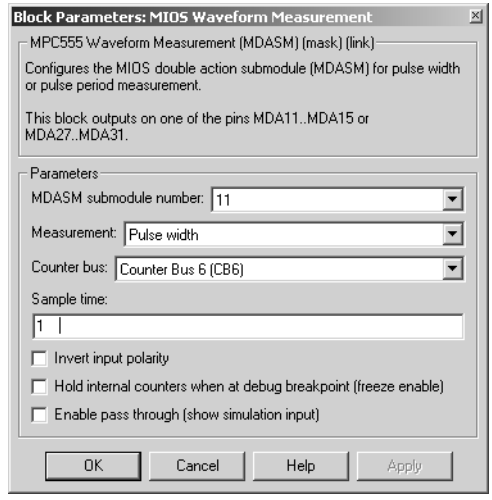
- *Pulse width measurement*: the MIOS Waveform Measurement block outputs the time from the leading edge of a pulse to the trailing edge of the same pulse.
- *Pulse period measurement*: the MIOS Waveform Measurement block outputs the time from the leading edge of a pulse to the next leading edge of a pulse.

Note that the minimum and maximum measurable pulse periods and pulse widths are dependent on the selected clock sources and their configurations.

You must configure the clock sources via the MPC555 Resource Configuration object. There are only two clock sources (assigned via the **Counter bus** parameter) assignable to the ten MDASM modules. More than one MDASM can be assigned to a single clock source.

Refer to section 15.11, “MIOS Double Action Submodule (MDASM) Registers” in the *MPC555 Users Manual* for further information on the parameters described below.

Dialog Box



MDASM submodule number

Selects one of the 10 MIOS Double Action Submodules in the MPC555.

Measurement

Selects the mode of operation of the block: either pulse width measurement or pulse period measurement.

Counter bus

Select one of the two counters that can be used as sources to drive the MDASM module.The counters must be configured via the MPC555 Resource Configuration object. See “MIOS1 Configuration Parameters” on page 5-55.

Sample time

The period at which Simulink reads the pulse width or period. The measurements are performed in hardware so it is not necessary to set the sample time to suit the expected period of the incoming signal.

Invert output polarity

Changes the sense of the leading edge of the pulse. When **Invert output polarity** is selected, the leading edge is rising. Otherwise, the leading edge is falling.

Hold output when at debug break point (freeze enable)

Stops the clocks of the MDASM module when a breakpoint is hit during debug mode.

Enable pass through (show simulation input)

Lets you provide a signal to this block for use in simulation. When this option is enabled, an inport appears on the block. The block input is passed through to the output during simulation. (See “Data Type Support and Scaling for Device Driver Blocks” on page 5-6 for information on supported input/output data types and scaling of input/output signals.) This option affects simulation only.

Purpose

Support device configuration for MPC555 CPU and MIOS, QADC, and TouCAN Submodules

Library

Embedded Target for Motorola MPC555

Description

The MPC555 Resource Configuration block differs in function and behavior from conventional blocks. Therefore, we refer to this block as the MPC555 Resource Configuration *object*.

The MPC555 Resource Configuration object maintains configuration settings that apply to the MPC555 CPU and its MIOS, QADC, and TouCAN subsystems. Although the MPC555 Resource Configuration object resembles a conventional block in appearance, it is not connected to other blocks via input or output ports. This is because the purpose of the MPC555 Resource Configuration object is to provide information to other blocks in the model. MPC555 device driver blocks register their presence with the MPC555 Resource Configuration object when they are added to a model or subsystem; they can then query the MPC555 Resource Configuration object for required information.

To install a MPC555 Resource Configuration object in a model or subsystem, open the top-level Embedded Target for Motorola MPC555 library and select the MPC555 Resource Configuration icon. Then drag and drop it into your model or subsystem, like a conventional block.

Having installed a MPC555 Resource Configuration object into your model or subsystem, you can then select and edit configuration settings in the MPC555 Resource Configuration window. See “Using the MPC555 Resource Configuration Window” on page 5-51 for further information.

Note Any model or subsystem using device driver blocks from the Embedded Target for Motorola MPC555 library *must* contain an MPC555 Resource Configuration object. You should place an MPC555 Resource Configuration object at the top level system for which you are going to generate code. If your whole model is going to run on the target processor, put the MPC555 Resource Configuration object at the root level of the model. If you are going to generate code from separate subsystems (to run specific subsystems on the target), place an MPC555 Resource Configuration object at the top level of each subsystem. You should not have more than one MPC555 Resource

Configuration object in the same branch of the model hierarchy. Errors will result if these conditions are not met.

Types of Configurations

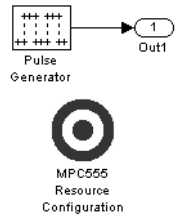
A *configuration* is a collection of parameter values affecting the operation of a group of device driver blocks in one of the Embedded Target for Motorola MPC555 libraries, such as the MIOS1, QADC64 or TouCAN libraries. The MPC555 Resource Configuration object currently supports the following types of configurations:

- System Configuration: MPC555 clocks and other CPU-related parameters.
- MIOS1 Configuration: parameters related to the Modular Input/Output System (MIOS).
- QADC64 Configuration: parameters related to the Queued Analog-to-Digital Converter module (QADC).
- TouCAN Configuration: parameters related to the CAN 2.0B Controller Module (TouCAN).

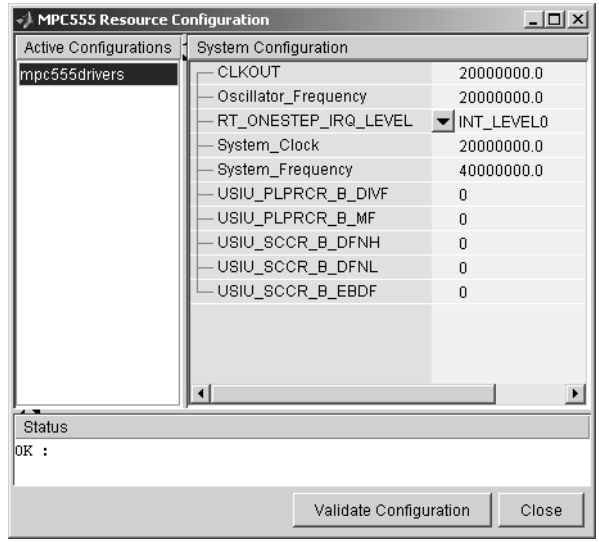
Active and Inactive Configurations

An *active* configuration is a configuration associated with blocks of the model or subsystem in which the MPC555 Resource Configuration object is installed. There is always an active MPC555 configuration. For any other configuration type (e.g., QADC, MIOS, or TouCAN), there is at most one active configuration.

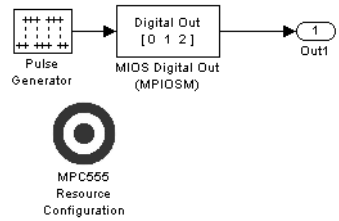
Consider this model, which contains a MPC555 Resource Configuration object but no MPC555 device driver blocks.



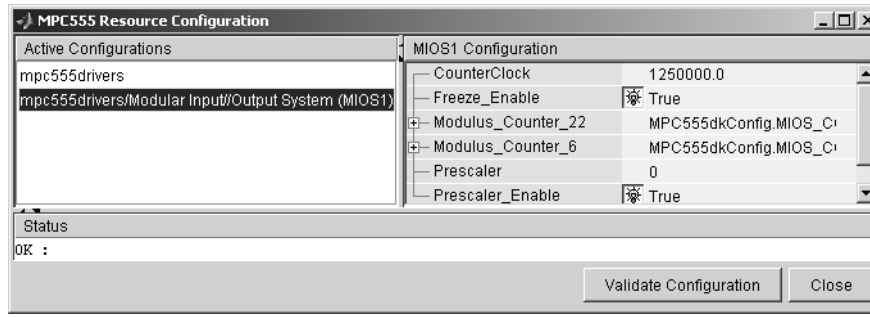
This model has only one active configuration, for the MPC555 itself, as shown in the MPC555 Resource Configuration window.



When a device driver block is added to the model, an appropriate configuration is created and activated. This figure shows an MIOS Digital Out block added to the model.



The addition of the MIOS Digital Out block causes an MIOS configuration to be added to the list of active configurations, as shown in this figure.



A configuration remains active until all blocks associated with it are removed from the model or subsystem. At that point, the configuration is in an *inactive* state. Inactive configurations are not shown in the MPC555 Resource Configuration window. You can reactivate a configuration by simply adding an appropriate block into the model.

Note When using device driver blocks from the Embedded Target for Motorola MPC555 libraries in conjunction with the MPC555 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the MPC555 Resource Configuration block will operate incorrectly.

When you save a model that contains inactive configurations, you have the option to either save inactive configurations with the model, or delete them.

Using the MPC555 Resource Configuration Window

To open the MPC555 Resource Configuration window, install a MPC555 Resource Configuration object in your model or subsystem, and double-click on the MPC555 Resource Configuration icon. The MPC555 Resource Configuration window then opens.

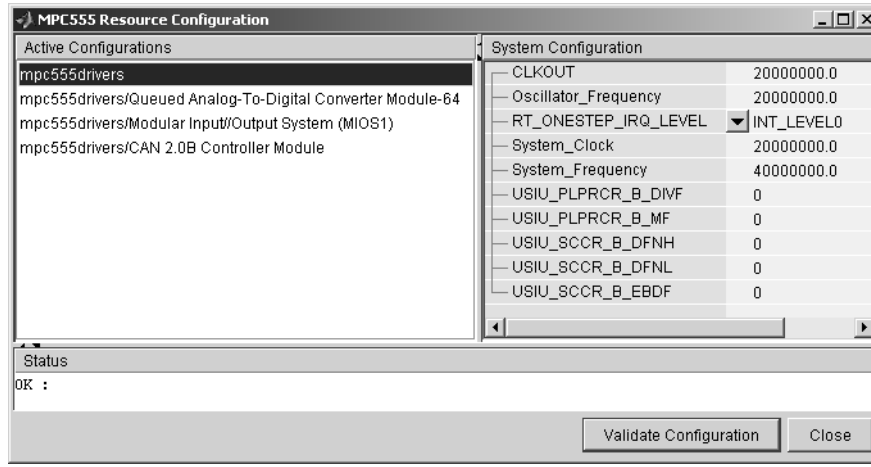


Figure 5-1: MPC555 Resource Configuration Window

Figure 5-1 shows the MPC555 Resource Configuration window for a model that has active configurations for MPC555, MIOS1, QADC, and TouCAN.

The MPC555 Resource Configuration window consists of the following elements:

- **Active Configurations** panel: This panel displays a list of currently active configurations. To edit a configuration, click on its entry in the list. The parameters for the selected configuration then appear in the **System configuration** panel.

  To link back to the library associated with an active configuration, right-click on its entry in the list. From the popup menu that appears, select **Go to library**.

  To see documentation associated with an active configuration, right-click on its entry in the list. From the popup menu that appears, select **Help**.

- **System configuration** panel: This panel lets you edit the parameters of the selected configuration. The parameters of each configuration type are detailed in "MPC555 Resource Configuration Window Parameters" on page 5-52.

Note There is no **Apply** or **Undo** functionality in the **System configuration** panel. All parameter changes are applied immediately.

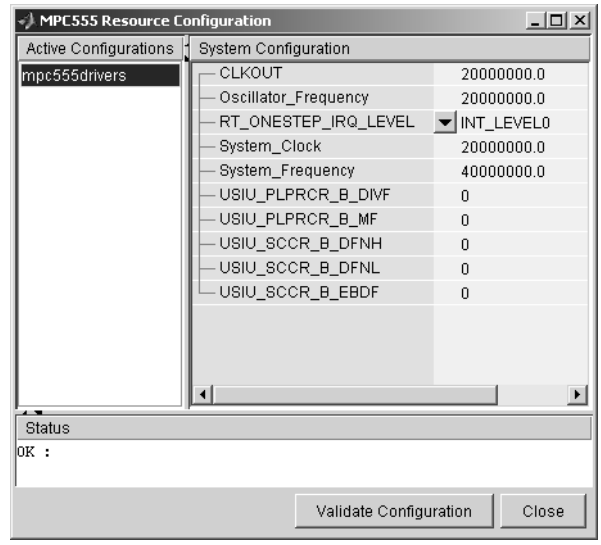
- **Status** panel: The **Status** panel displays error messages that may arise if resource allocation conflicts are detected in the configuration.
- **Validate Configuration** button: After you edit a configuration, you should always click the **Validate Configuration** button to check for resource allocation conflicts. For example, if both TouCAN modules A and B are assigned to interrupt level IRQ 1, the **Validate Configuration** process will detect the conflict and display a warning in the **Status** panel.

  Note that the **Validate Configuration** button does not validate the entire model; it only checks for resource allocation conflicts related to the selected configuration. To detect problems related to the model as a whole, select **Update diagram** (**Ctrl+D**) from the Simulink Edit menu.
- **Close** button: Dismisses the window.

MPC555 Resource Configuration Window Parameters

The sections below describe the parameters for each type of configuration in the MPC555 Resource Configuration window. The default parameter settings are optimal for most purposes. If you want to change the settings, we suggest you read the sections of the *MPC555 Users Manual* referenced below. You can find this document at the following URL.
`http://e-www.motorola.com`.

System Configuration Parameters



RT_ONESTEP_IRQ_LEVEL

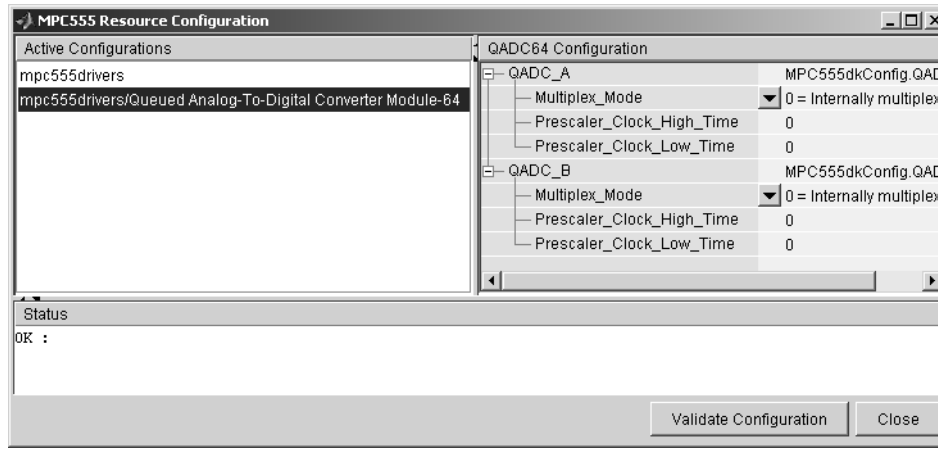
The `rt_OneStep` function is the basic execution driver of all programs generated by the Embedded Target for Motorola MPC555. `rt_OneStep` is installed as a timer interrupt service routine; it sequences calls to the *`model`*`_step` function. The **RT_ONESTEP_IRQ_LEVEL** parameter lets you associate `rt_OneStep` with any of the available IRQ levels (`0..32`). For non-interrupt driven operation, select `Interrupts Disabled`.

See the "Data Structures and Program Execution" section in the Real-Time Workshop Embedded Coder documentation for a detailed description of the `rt_OneStep` function.

System Clock Parameters

The other parameters in the MPC555 group alter the speed of three of the main clocks in the MPC555. Refer to section 8, "Clocks and Power Control," in the *MPC555 Users Manual* for information on these settings.

QADC64 Configuration Parameters



The QADC64 Configuration parameters configure the QADC64 operational mode and supports the blocks in the QADC sublibrary.

The QADC64 performs 10 bit analog to digital conversion on an input signal. Currently the blocks in this library support only the *continuous scan* mode of operation. In continuous scan mode, the QADC64 is set to run, and then continuously acquires data into its result buffer. Input is double buffered, so the model can read the result buffer at any time to get the latest available signal data.

The MPC555 has two QADC modules, QADC_A and QADC_B. You can program these individually. By default each QADC module has 16 input channels. By attaching an external multiplexer to three of the analog input pins, you can increase the number of possible channels to 41. These pins become outputs from the processor and can act as inputs to an analog multiplexer. The **Multiplex Mode** parameter determines whether the QADC64 operates in internally or externally multiplexed mode.

Refer to section 13, “Queued Analog-to-Digital Converter Module-64,” in the *MPC555 Users Manual* for detailed information about the QADC64.

In general, you should not need to change any of the settings of the parameters described below from their defaults. The other parameters are advanced

settings. Refer to section 13, “Queued Analog-to-Digital Converter Module-64,” in the *MPC555 Users Manual* for information on these settings.

Multiplex Mode

Configures the QADC64 for internally or externally multiplexed mode by setting the MUX bit. The MUX bit determines the interpretation of the channel numbers and forces the MA[2:0] pins to be outputs. Valid settings are:

- `0 = Internally multiplexed : 16 possible channels`
- `1 = Externally multiplexed : 41 possible channels`

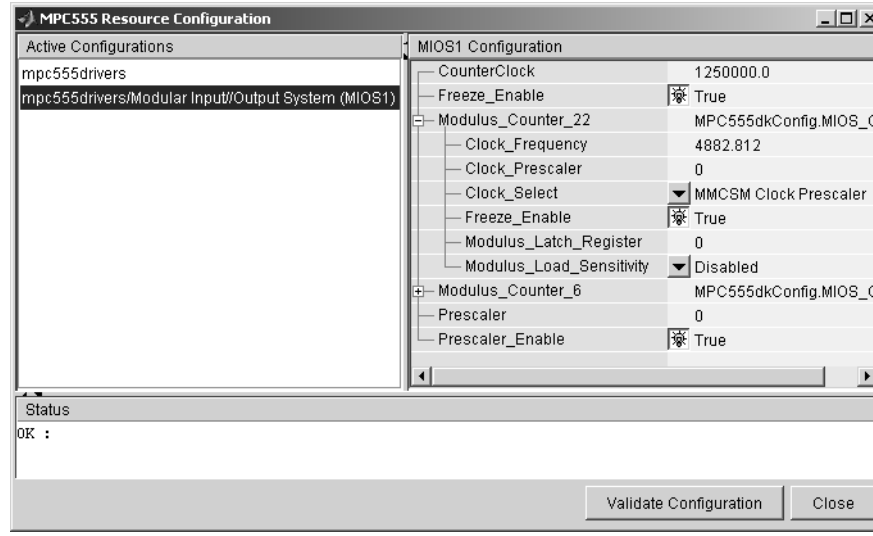
Prescaler Clock High Time

Prescaler clock high (PSH) time. The PSH field selects the QCLK high time in the prescaler. PSH value plus 1 represents the high time in IMB clocks.

Prescaler Clock Low Time

Prescaler clock low (PSL) time. The PSL field selects the QCLK low time in the prescaler. PSL value plus 1 represents the low time in IMB clocks.

MIOS1 Configuration Parameters



CounterClock

The MIOS counter clock is generated by the MIOS counter prescaler submodule. The MIOS counter clock drives the other MIOS1 submodules. The value shown for the counter clock is calculated automatically as the system clock frequency divided by the prescaler value.

Freeze Enable

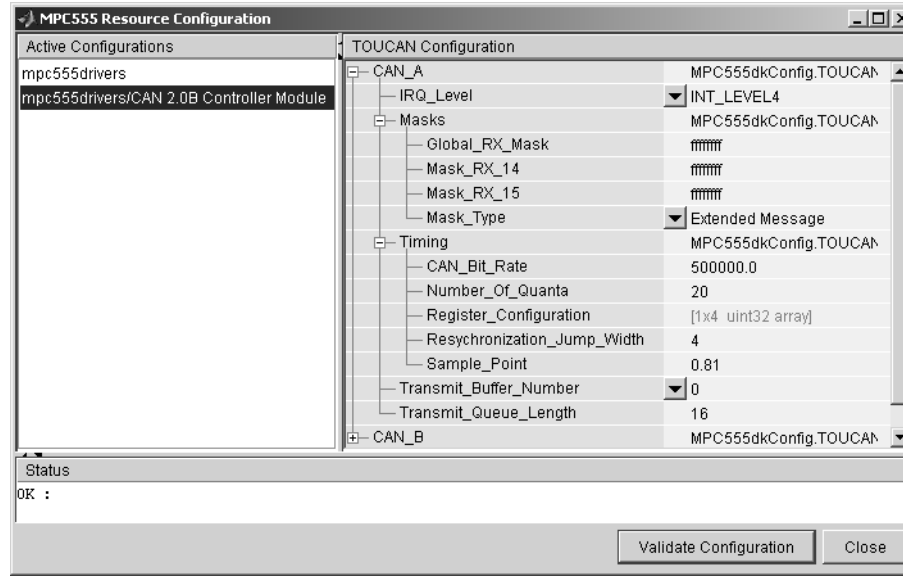
This allows all counters on the MIOS1 to be frozen when the processor is stopped in debug mode. Note that this is in addition to the **Freeze Enable** setting for individual submodules on the MIOS1. To allow the counters on a particular submodule to be stopped, select Freeze enable here, and select **Hold output when at debug break point (freeze enable)** in the block parameters associated with the submodule (e.g., MIOS PWM block or MIOS Waveform Measurement block).

Modulus Counter 6 and 22

These two counters provide reference clocks to submodules such as the MIOS Pulse Width Modulation Submodule and the MIOS Double Action Submodule (Frequency / Period measurement) subsystems. If you change the **Clock select** to anything other than `MMCSM Clock Prescaler`, the MIOS Pulse Width Modulation and MIOS Waveform Measurement blocks will not work as expected. To change the clock frequency and hence the available resolution of pulse width modulation and waveform measurement, change the **Clock Prescaler** to a value between 0 and 255.

Refer to section 15.10, “MIOS Modulus Counter Submodule (MMCSM),” in the *MPC555 Users Manual* for information on these settings.

TouCAN Configuration Parameters



The parameters listed below are the same for TouCAN modules A and B. Consult Section 16 of the *MPC555 User's Manual* before editing the TouCAN configuration parameter defaults.

IRQ Level

The transmit queue for each TouCAN module requires a processor interrupt to run. Select an interrupt level (0-31) that is not used by any other device. Use the **Validate Configuration** button to make sure you do not select an interrupt level that is already in use.

Mask Configuration Parameters

Global RX Mask

Buffers 0-13 use this mask. Setting a bit to 1 in the mask causes the corresponding bits in the message to be masked out (i.e., ignored).

Mask RX 14

Same as **Global RX Mask**, but the mask applies only to buffer 14.

Mask RX 15

Same as **Global RX Mask**, but the mask applies only to buffer 15.

Mask Type

Specify whether the buffer masks are Standard or Extended frame IDs. If you want to receive Extended Frames in your model, you should set the **Mask Type** to **Extended Message**. The mask type option tells the compiler how to map the bits specified in the mask options to the bits in the hardware. The decision as to whether a message is a Standard or Extended frame is defined on a per message buffer basis.

Timing Configuration Parameters

CAN Bit Rate

Enter the desired bit rate. the default bit rate is `500000.0`.

Number of Quanta

The number of TouCAN clock ticks per message bit.

Register Configuration

This field is read-only. It contains intermediate data used for bit-timing calculations.

Resynchronization Jump Width

The maximum number of clock ticks that the TouCAN device can resynchronize over when it detects that it is losing message synchronization.

Sample Point

The point in the message where the TouCAN tries to sample the value of the message bit.

Transmission Configuration Parameters

Transmit Buffer Number

Select one of buffers (0-15) for use as a transmit buffer. A queue is created for that buffer and is used by all TouCAN transmit blocks.

Transmit Queue Length

Length (in bytes) allocated to messages in the transmit queue.

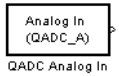
Purpose

Input driver enables use of Queued Analog-Digital Converter (QADC64) in continuous scan mode

Library

Embedded Target for Motorola MPC555

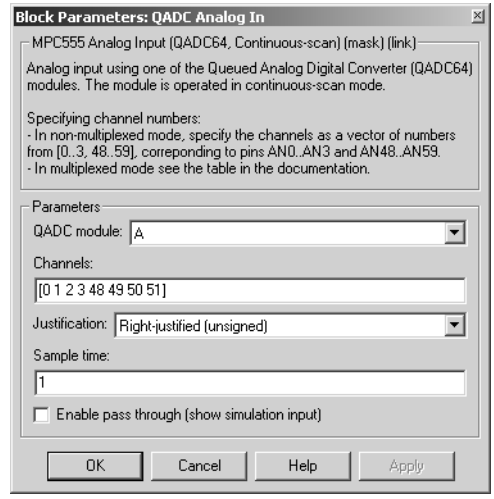
Description



The QADC Analog In block sets the QADC64 into continuous scan mode. It then samples the specified channels at the specified rate. In continuous scan mode, the analog-to-digital converter is scanned as fast as possible, at a rate much faster than the sample rate of the model. Using continuous scan mode ensures that your application will obtain the latest signal value.

The MPC555 has two QADC modules, A and B. You can program these individually. You can place only one instance of the QADC Analog In block per module in your model or subsystem.

Dialog Box



QADC module

Selects module A or B.

Channels

A vector of numbers representing channels to be scanned. See “Channel Number Selection” below.

Justification

Converted data is read from the 10-bit wide QADC64 result word table into a 16-bit word. Data from the result word table can be accessed in three different formats. The **Justification** menu selects from the following formats:

- `Right-justified (unsigned)`: with zeros in the higher order unused bits.
- `Left-justified (signed)`: with the most significant bit inverted to form a sign bit, and zeros in the unused lower order bits. In this mode, zero is treated as the half scale of the input range.
- `Left-justified (unsigned)`: with zeros in the unused lower order bits.

Refer to section 13.13, in the "Queued Analog-to-Digital Converter Module-64" section of the *MPC555 Users Manual* for further information.

Sample time

Block sample time; determines sample rate at which the port is monitored.

Enable pass through (show simulation input)

Lets you provide a signal to this block for use in simulation. When this option is enabled, an inport appears on the block. The input (pass-through) signal must have double or single data type. The input signal is scaled onto the range `0..1` to represent the minimum and maximum voltage input. This input signal is mapped onto output according to the selected **Justification** option for justification. The **Enable pass through** option affects simulation only.

See also "Data Type Support and Scaling for Device Driver Blocks" on page 5-6.

Channel Number Selection

A channel number in the **Channels** vector selects the input channel number corresponding to the analog input pin to be sampled and converted. The analog input pin channel number assignments and the pin definitions vary, depending on whether the QADC64 is operating in multiplexed or nonmultiplexed mode. The queue scan mechanism makes no distinction between an internally or externally multiplexed analog input.

If a reserved channel number (channels 32 to 47) or an invalid channel number (channels 4 to 31 in nonmultiplexed mode), the low reference level (V_{RL}) is converted.

Programming the channel field to channel 63 indicates the end of the queue.

Channels 60 to 62 are special internal channels. When one of these channels is selected, the sample amplifier is not used. Instead, the value of V_{RL}, V_{RH}, or $(V_{RH} - V_{RL})/2$ is placed directly into the converter. Programming the input sample time to any value other than two for one of the internal channels has no benefit except to lengthen the overall conversion time.

Table 5-1 and Table 5-2 show the mapping between the channel numbers and the hardware pins for the two scanning modes (multiplexed and nonmultiplexed).

For example, in nonmultiplexed mode, to scan all 16 channels of the QADC64 you would specify the following vector in the **Channels** field:

```
[ 0 1 2 3 48 49 50 51 52 53 54 55 56 57 58 59 ]
```

Table 5-1: Nonmultiplexed Scan Mode

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
PQB0	A_AD0 / AN0	-	I	0
PQB1	A_AD1 / AN1	-	I	1
PQB2	A_AD2 / AN2	-	I	2
PQB3	A_AD3 /AN3	-	I	3
-	-	Invalid	-	4 to 31
-	-	Reserved	-	32 to 47
PQB4	A_AD4 / AN48	-	I	48
PQB5	A_AD5 / AN49	-	I	49
PQB6	A_AD6 / AN50	-	I I	50
PQB7	A_AD7 / AN51	-	I	51
PQA0	A_AD8 / AN52	-	I/O	52
PQA1	A_AD9 / AN53	-	I/O	53

Table 5-1: Nonmultiplexed Scan Mode (Continued)

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
PQA2	A_AD10 / AN54	-	I/O	54
PQA3	A_AD11 / AN55	-	I/O	55
PQA4	A_AD12 / AN56	-	I/O	56
PQA5	A_AD13 / AN57	-	I/O	57
PQA6	A_AD14 / AN58	-	I/O	58
PQA7	A_AD15 / AN59	-	I/O	59
-	V RL	-	I	60
-	V RH	-	I	61
-	-	$(V_{RH} - V_{RL})/2$	-	62
-	-	End of Queue Code	-	63

Table 5-2: Multiplexed Scan Mode

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
PQB0	A_AD0 / ANw	-	I	0-14 even
PQB1	A_AD1 / ANx	-	I	1-15 odd
PQB2	A_AD2 / ANy	-	I	16-30 even
PQB3	A_AD3 / Anz	-	I	17-31 odd
-	-	Reserved	-	32-47
PQB4	A_AD4 / AN48	-	I	48
PQB5	A_AD5 / AN49	-	I	49
PQB6	A_AD6 / AN50	-	I	50
PQB7	A_AD7 / AN51	-	I I	51
PQA0	-	MA0	I/O	52
PQA1	-	MA1	I/O	53
PQA2	-	MA2	I/O	54

Table 5-2: Multiplexed Scan Mode (Continued)

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
PQA3	A_AD11 / AN55	-	I/O	55
PQA4	A_AD12 / AN56	-	I/O	56
PQA5	A_AD13 / AN57	-	I/O	57
PQA6	A_AD14 / AN58	-	I/O	58
PQA7	A_AD15 / AN59	-	I/O	59
-	V RL	-	I	60
-	V RH	-	I	61
-	-	$(V_{RH} - V_{RL})/2$	-	62
-	-	End of Queue Code	-	63

In Table 5-2, PQA0, PQA1 and PQA2 are used as output pins to drive an external demultiplexer.

QADC Digital In

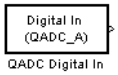
Purpose

Input driver enables use of Queued Analog-Digital Converter (QADC64) pins as digital inputs

Library

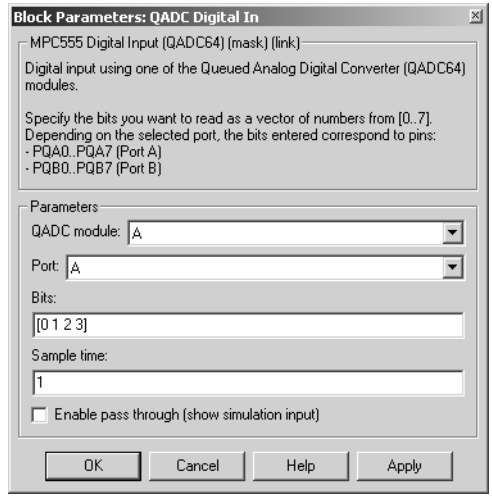
Embedded Target for Motorola MPC555

Description



The QADC Digital In block allows you to treat the QADC64 pins as digital inputs. Each QADC64 module has two 8-bit ports, A and B. You can use any bit on either port as a digital input.

Dialog Box



QADC module

Selects module A or B.

Port

Selects an 8 bit port (A or B) on the module.

Bits

A vector of bits (numbered 0-7) to read. The vector should not be longer than eight elements.

Sample time

Block sample time; determines sample rate at which the port is monitored.

Enable pass through (show simulation input)

Lets you provide a signal to this block for use in simulation. When this option is enabled, an inport appears on the block. The block input is passed through, unaltered, to the output during simulation. This option affects simulation only.

Mapping Bits To Hardware Pins

Use Table 5-3 to work out how the block ports and bits map to processor pins on the MPC555.

Table 5-3: Relationship of Port/Bit Parameters to Hardware Pins

Port	Bit	Hardware Pin
B	0	A_AD0 / PQB0
B	1	A_AD1 / PQB1
B	2	A_AD2 / PQB2
B	3	A_AD3 / PQB3
B	4	A_AD4 / PQB4
B	5	A_AD5 / PQB5
B	6	A_AD6 / PQB6
B	7	A_AD7 / PQB7
A	0	A_AD8 / PQA0
A	1	A_AD9 / PQA1
A	2	A_AD10 / PQA2
A	3	A_AD11 / PQA3
A	4	A_AD12 / PQA4
A	5	A_AD13 / PQA5

Table 5-3: Relationship of Port/Bit Parameters to Hardware Pins (Continued)

Port	Bit	Hardware Pin
A	6	A_AD14 / PQA6
A	7	A_AD15 / PQA7

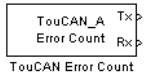
Purpose

Count transmit and/or receive errors detected on selected TouCAN modules

Library

Embedded Target for Motorola MPC555

Description

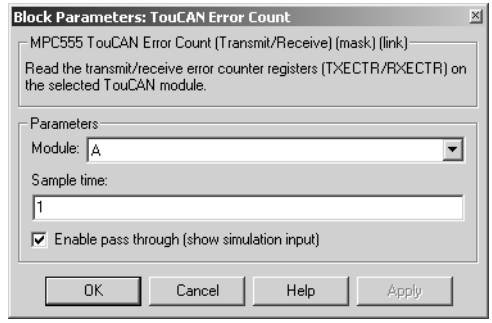


TouCAN Error Count

The TouCAN Error Count block maintains and reports a count of errors detected by the selected TouCAN module during receive and transmit. The receive and transmit error counts are output to the `RX` and `TX` outputs of the block, respectively.

The error counts also drive the TouCAN Warnings block outputs. (See "TouCAN Warnings" on page 5-77.)

Dialog Box



Module

Select TouCAN module A or B.

Sample time

Sample time of the block.

Enable pass through (show simulation input)

Lets you provide a signal to this block for use in simulation. When this option is enabled, inports appear on the block. The block inputs are passed through, unaltered, to the outputs during simulation. This option affects simulation only.

TouCAN Fault Confinement State

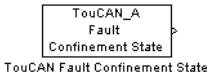
Purpose

Indicate the state of a TouCAN module

Library

Embedded Target for Motorola MPC555

Description



TouCAN Fault Confinement State

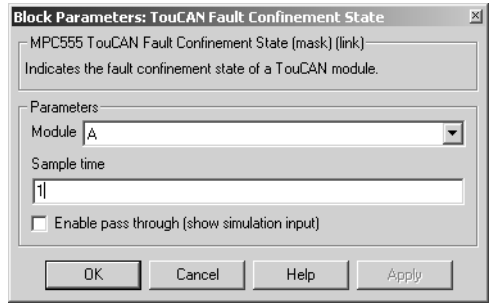
The TouCAN Fault Confinement State block provides an indicator for the state of the selected TouCAN module. The block obtains and outputs a field of two bits from the TouCAN module's Error and Status (ESTAT) register. The possible states are shown in Table 5-4.

Refer to section 16, "CAN 2.0B Controller Module," in the *MPC555 Users Manual* for further information.

Table 5-4: FCS State Values

State	Value	Description
Error Active	00	Normal operation
Error Passive	01	Listening only mode. The device cannot transmit.
Bus Off	1x	The device is not allowed to transmit or receive and is effectively cut off from the bus.

Dialog Box



Module

Select TouCAN module A or B.

Sample time

Sample time of the block.

Enable pass through (show simulation input)

Lets you provide a signal to this block for use in simulation. When this option is enabled, an inport appears on the block. The block input is passed through, unaltered, to the output during simulation. This option affects simulation only.

TouCAN Interrupt Generator

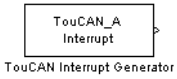
Purpose

Generate an interrupt subsystem for CAN interrupt sources

Library

Embedded Target for Motorola MPC555

Description



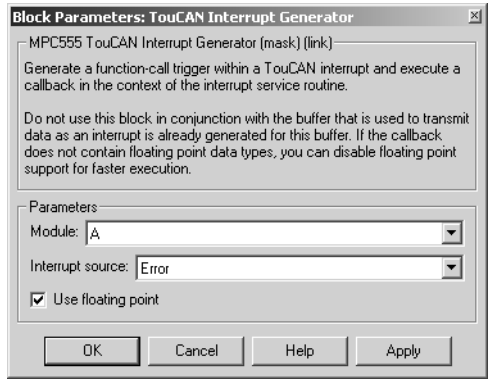
The TouCAN Interrupt Generator block generates a function-call trigger within a TouCAN interrupt and executes a callback in the context of the interrupt service routine.

This block may be used to execute a callback on occurrence of Bus Off, Error, or Wake interrupts.

Do not use this block unless you are aware of the dangers of using asynchronous interrupts in the model. Unpredictable data loss or model behavior may result unless extreme caution is taken.

For faster interrupts, you can disable floating point support via the **Use floating point** option. However, if you disable floating point support, do not use blocks that require floating point operations in your model. Use of such blocks will cause a floating point exception at runtime.

Dialog Box



Module

Select TouCAN module A or B.

Interrupt source

Choose the interrupt source (Bus Off, Error or Wake) for your ISR generator.

Use floating point

Enable or disable floating point support.

TouCAN Receive

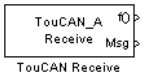
Purpose

Receive CAN messages from the TouCAN module on the MPC555

Library

Embedded Target for Motorola MPC555

Description

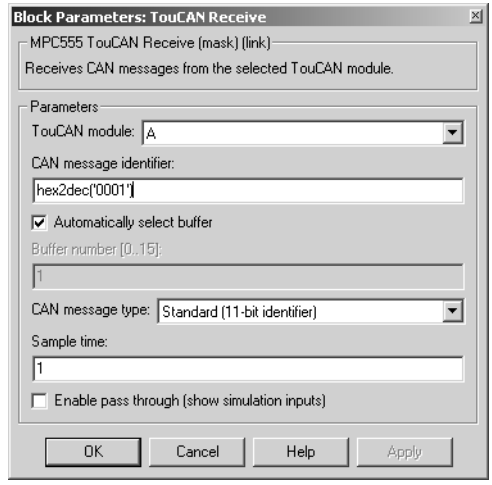


The TouCAN Receive block receives CAN messages from the TouCAN module.

The TouCAN Receive block can reserve any of the 16 buffers on the TouCAN module. Alternatively, you can instruct the TouCAN Receive block to select a hardware buffer automatically from the available buffers.

The TouCAN Receive block has two outputs: a data output and a function call trigger output. The TouCAN Receive block polls its message buffer at a rate determined by the block's sample time. When the TouCAN Receive block detects that a message has arrived, the function call trigger is activated. You should use a function call subsystem, activated by the trigger, to decode the message available at the TouCAN Receive data output.

Dialog Box



TouCAN module

Selects one of the two TouCAN modules (A or B) on the MPC555. The TouCAN modules can receive messages independently.

CAN message type

The type of message you want to receive. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

CAN message identifier

The identifier of the message you want to receive. Note that if you have set the TouCAN configuration parameters (see "MPC555 Resource Configuration" on page 5-47) in your model to mask out certain bits (e.g., the message identifier field) you may receive messages with identifiers other than the identifier specified here.

Automatically select buffer

When this option is selected, the TouCAN Receive block automatically selects a receive buffer from the available buffers. We recommend that you use this automatic buffer selection, unless you want to use buffer 14 or 15 to receive multiple CAN message identifiers in a single buffer. See also "TouCAN Configuration Parameters" on page 5-57.

Buffer number [0..15]

This field is enabled if the **Automatically select buffer** option is cleared. **Buffer number** specifies the identifier of the receive buffer for this block. We recommend that you select **Automatically select buffer** instead of manually specifying the buffer, unless you want to use buffer 14 or 15 to receive multiple CAN message IDs in a single buffer. See also "TouCAN Configuration Parameters" on page 5-57.

Sample time

Determines the rate at which to sample the buffer to see if a new message has arrived.

Note The TouCAN Receive block sample time must be set to a value that is smaller than the minimum time between CAN messages that will be received into the corresponding buffer. If more than one message is received into a buffer during a single sample interval, the older message will be overwritten.

Enable pass through (show simulation input)

Lets you provide a signal to this block for use in simulation. When this option is enabled, inports appear on the block. The block inputs are passed through, unaltered, to the outputs during simulation. This option affects simulation only.

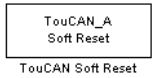
Purpose

Reset a TouCAN module

Library

Embedded Target for Motorola MPC555

Description

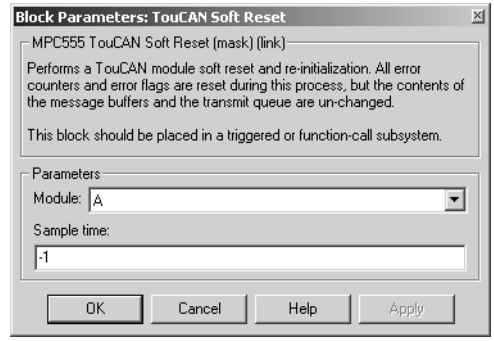


TouCAN Soft Reset

When the TouCAN Soft Reset block executes, the TouCAN module resets its internal state. The TouCAN error counters will be reset. The Fault Confinement State will be reset to the Error Active state, provided the TouCAN module has not reached the Bus Off state.

We recommend that you place this block in a triggered subsystem, with a sample time of -1 (inherited).

Dialog Box



Module

Select TouCAN module A or B.

Sample time

Sample time of the block.

TouCAN Transmit

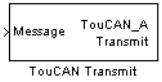
Purpose

Transmit a CAN message via a TouCAN module on the MPC555

Library

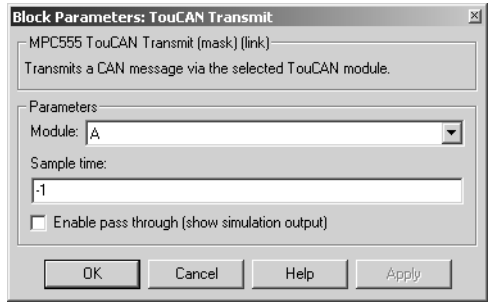
Embedded Target for Motorola MPC555

Description



The TouCAN Transmit block transmits a CAN message onto the CAN bus. The TouCAN Transmit block uses the queue set up by the MPC555 Resource Configuration object (see “MPC555 Resource Configuration” on page 5-47).

Dialog Box



Module

Selects one of the two TouCAN modules (A or B) on the MPC555.

Sample time

Choose -1 to inherit the sample time from the driving blocks. The TouCAN Transmit block does not inherit constant sample times and runs at the base rate of the model if driven by invariant signals.

Enable pass through (show simulation input)

Lets you provide a signal from this block for use in simulation. When this option is enabled, an outport appears on the block. The block input is passed through, unaltered, to the output during simulation. This option affects simulation only.

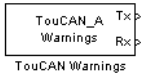
Purpose

Flag excessively high transmit or receive error counts on TouCAN modules

Library

Embedded Target for Motorola MPC555

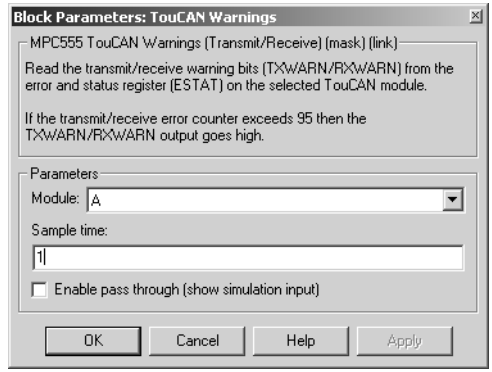
Description



The TouCAN Warnings block has two logical outputs, `RX` and `TX`. If the transmit error counter is over 95, then the `TX` output goes high. If the receive error counter is over 95, then the `RX` output goes high.

Use this block, in conjunction with a TouCAN Error Count block, to monitor error conditions on a selected TouCAN module.

Dialog Box



Module

Select TouCAN module A or B.

Sample time

Sample time of the block.

Enable pass through (show simulation input)

Lets you provide signals to this block for use in simulation. When this option is enabled, inports appear on the block. The block inputs are passed through, unaltered, to the output during simulation. This option affects simulation only.

Vector CAN Configuration

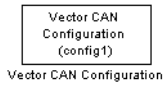
Purpose

Configure a Vector CAN channel (either hardware or virtual) for use with Vector-Informatik drivers

Library

Can Drivers (Vector)

Description



Vector CAN Configuration

The Vector CAN Configuration block configures a CAN channel on the host PC, using the Vector CAN Driver. A CAN channel can be

- A channel associated with a CAN card installed on your PC
- A virtual channel, not requiring any CAN hardware

The Vector CAN Driver software must be installed on your PC, regardless of whether you want to use virtual channels or actual hardware.

Place one Vector CAN Configuration block in the model for each CAN channel required.

You can use virtual channels to communicate between two separate Simulink models in the same MATLAB session, or between a CANalyser session and Simulink, or even between two Simulink models running in different sessions of MATLAB on the same machine. For an example of how this can be done see the mpc555rt_io and mpc555rt_iohost demos.

A Vector CAN Configuration block works in association with Vector CAN Transmit and Vector CAN Receive blocks. The association is formed by assigning the same values to the **Tag** parameter of all the blocks.

Setting the Baud Rate

The Vector CAN Configuration block lets you set the speed of the CAN channel connection.

In many cases you can avoid the complexities of CAN bit timing by selecting one of the **Precalculated baud rate** settings in the **Block Parameters** dialog box. We recommend using the precalculated baud rates wherever possible.

If the precalculated baud rates do not meet your requirements, you can select the **Set bit timing parameters manually** option. You can then set the baud rate by configuring the **Baud rate prescaler**, **Synchronization jump width**, **Time segment 1**, **Time segment 2**, and **Sample mode** parameters as described in this section.

The following variables are defined in calculating the baud rate for the Vector CAN Configuration block:

- *f*: The Vector hardware oscillator frequency *f* is a constant, 16MHz.
- *prescaler*: The baud rate prescaler that defines the length of one time quanta.Valid values are `[1..64]`.
- *tseg1*: Defines the length of time preceding the sample point within a bit time. Valid values are `[1..16]`.
- *tseg2*: Defines the length of time following a sample point within a bit time. Valid values are `[1..8]`.

To set the baud rate, first derive values for *prescaler*, *tseg1*, and *tseg2*, using the following formulas:

1. Number of time quanta per second *qps = f /(2 * prescaler)*.
2. Number of time quanta per bit time *qpb = 1 + tseg1 + tseg2*.
3. Baud rate = (number of bit times per second) = *qps / qpb*.

Next, select values for the following parameters:

- *SJW* (Synchronization Jump Width): For CAN to work successfully, all nodes on the network must be synchronized. However, as time goes by, clocks on different nodes will drift out of sync, and must resynchronize. *SJW* specifies the maximum width (in time quanta) that can be added to *tseg1* (in case of a slower transmitter), or subtracted from *tseg2* (in case of a faster transmitter) in order to regain synchronization during the receipt of a CAN message.

  Valid values for *SJW* are `[1..4]`.
- *Sample mode*: *Sample mode* defines how many samples of the signal to take to determine a valid bit. Select either `1 sample per bit` or `3 samples per bit`, via the drop down menu. `3 samples per bit` is more reliable, and may be necessary on a noisy channel.

Finally, enter the values derived above into the block parameters:

- **Baud rate prescaler**: value of *prescaler*.
- **Time segment 1**: value of *tseg1*.
- **Time segment 2**: value of *tseg2*.
- **Sample mode**: value of *sample mode*.

- **Synchronization jump width**: value of *SJW*.

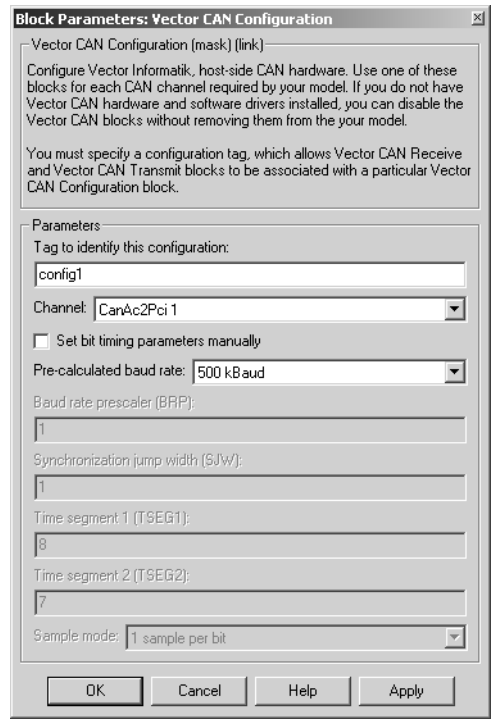
As an example, the parameters used for the precalculated baud rate 500 kBaud were

- **Baud rate prescaler**: *1*
- **Synchronization jump width**: `1`
- **Time segment 1**: `8`.
- **Time segment 2**: `7`.
- **Sample mode**: `1 sample per bit`

For further information on the CAN, and on CAN bit timing, see the following documents, available at the CAN in Automation (CiA) Web site: `http://www.can-cia.de`

- *CAN General Introduction*
- The “Bit Timing” section of the *CAN Physical Layer* document.

Dialog Box



Tag to identify this configuration

A unique identifier for naming a configured CAN channel. This tag is used to associate transmit and receive blocks with the configured channel.

Channel

Lets you select either a virtual channel (`Virtual 1` or `Virtual 2`) or a supported hardware device.

If you have the required drivers installed, but no hardware device, you can use `Virtual 1` or `Virtual 2`.

If you do not have the required drivers installed, select `None`. This allows you to use the block in simulation, even without the required driver or hardware installed. If you select a hardware device and the driver detects

that you do not have the requested hardware installed, the block will report an error during simulation.

Precalculated baud rate

This parameter sets the speed of your channel connection. If none of the precalculated baud rates meets your requirements, you can select the **Set bit timing parameters manually** option.

Set bit timing parameters manually

Select this option if you want to program the CAN bit timing values yourself. See "Setting the Baud Rate" on page 5-78 for details.

Baud rate prescaler

Synchronization jump width

Time segment 1

Time segment 2

Sample mode

These parameters are enabled only if **Set bit timing parameters manually** is selected. See "Setting the Baud Rate" on page 5-78.

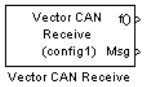
Purpose

Read CAN messages from a Vector CAN channel

Library

Can Drivers (Vector)

Description



Vector CAN Receive

The Vector CAN Receive block reads CAN messages from the channel specified by the **Configuration tag** parameter. The block has the following outputs:

- `f()` (function call trigger): The Vector CAN Receive block polls the CAN channel and outputs a trigger from this port when a message or messages are available. The function call trigger output should be connected to the trigger input of a function call subsystem that will process the message when triggered.
- `Msg`: This port outputs the CAN message received. Connect this port to a Function Call subsystem that will process the message when triggered.
- If the **Output timestamp** option is selected, a third output, labeled `Timestamp`, is added to the block. See “Output timestamp” on page 5-85.

For an example of the use of the Vector CAN Receive and Transmit blocks, see the `mpc555rt_io` and `mpc555rt_iohost` demos.

Message Queuing

The Vector Can driver allocates a message queue to each Vector CAN Receive block. A message queue has a maximum size of 1024 messages. As messages are received from the CAN network, they are put into the appropriate queue.

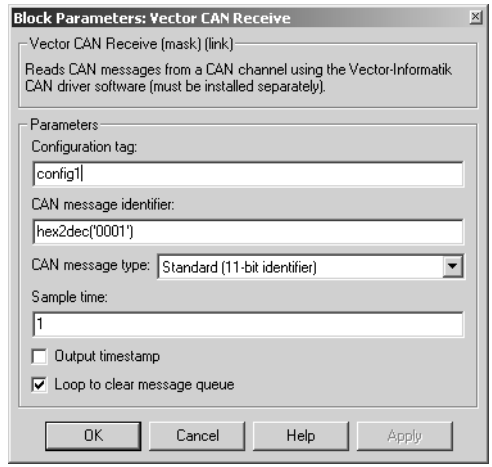
When the Vector CAN Receive block executes, it polls the queue to see how long it is. If **Loop to clear message queue** is selected, then the block will read all messages from the queue and process them sequentially. If more messages arrive during this process, they are added to the end of the queue and processed the next time the Vector CAN Receive block executes. We recommend that you select the **Loop to clear message queue** option, to avoid overflowing the block’s message queue.

If **Loop to clear message queue** is not selected, the Vector CAN Receive block processes only a single message from the queue is processed when the Vector CAN Receive block executes. Any other messages remain in the queue until handled by subsequent executions of the Vector CAN Receive block.

No messages are lost and no errors are reported until the queue size for a particular Vector CAN Receive block exceeds maximum length of 1024 messages.

Note To use this block, you must place a Vector CAN Configuration block in the model. The **Configuration tag** parameter of the Vector CAN Configuration block must be identical to the **Configuration tag** parameter of the Vector CAN Receive block.

Dialog Box



Configuration tag

A unique identifier for naming a configured CAN channel. This tag is used to associate the Vector CAN Receive block with a Vector CAN Configuration block. The Vector CAN Receive block will then receive messages from the channel specified in the Vector CAN Configuration block. You can assign more than one Vector CAN Receive block to the same CAN Configuration block.

CAN message identifier

Specify a single CAN message identifier for the message you want to receive.

CAN message type

Specify the CAN message type: either `Standard (11 bit identifier)` or `Extended (29 bit identifier)`.

Sample time

Specifies how often the block is to poll the CAN driver to see if any messages are available on the specified channel.

Output timestamp

When this option is selected, a third port, labeled `Timestamp`, is added to the Vector CAN Receive block.

The message timestamp output gives the time that the message was received. This timestamp is the clock time at which the message was received, not the Simulink run time. The timestamp is an integer scaled to 10 microsecond resolution. A value of 1 on the timestamp output equals 10 microseconds.

Loop to clear message queue

(On by default.) Controls how the block handles cases where multiple messages are queued up between executions of the block. When **Loop to clear message queue** is selected, all messages in the queue for the block at the time of execution are processed. Any connected Function Call subsystem executes as many times as required.

When **Loop to clear message queue** is not selected, a single message from the queue for the block is processed, leaving any other messages on the queue to be processed in the future.

Vector CAN Transmit

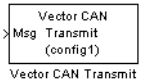
Purpose

Transmit CAN messages on a Vector CAN channel

Library

Can Drivers (Vector)

Description



Vector CAN Transmit

The Vector CAN Transmit block transmits CAN messages at its input on the channel specified by its **Configuration tag** parameter. The block will accept either Standard CAN Message or Extended CAN Message data typed signals as input.

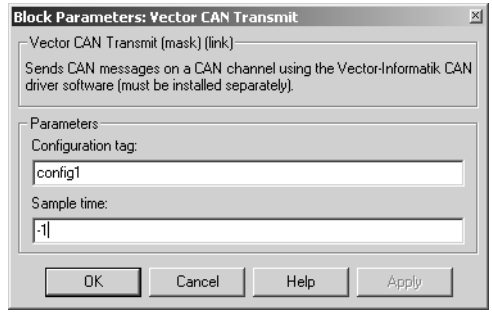
Connect the input of this block to a CAN message signal source, such as a CAN Message Packing block. The Vector CAN Transmit block does not construct or specify any of the information in the messages it transmits.

The Vector CAN Transmit block is designed to be placed in a triggered subsystem in order to transmit a message upon an event received. In this case you should specify an inherited sample time by entering -1 in the **Sample time** parameter.

For an example of the use of the Vector CAN Receive and Transmit blocks, see the `mpc555rt_io` and `mpc555rt_iohost` demos.

Note To use this block, you must place a Vector CAN Configuration block in the model. The **Configuration tag** parameter of the Vector CAN Configuration block must be identical to the **Configuration tag** parameter of the Vector Transmit block.

Dialog Box



Configuration tag

A unique identifier for naming a configured CAN channel. This tag is used to associate the Vector CAN Transmit block with a Vector CAN Configuration block. The Vector CAN Transmit block then transmits messages on the channel specified in the Vector CAN Configuration block.

Sample time

Specify how often this block executes to transmit a CAN message. Note that during simulation, the block's sample time is relative to other blocks in the diagram, not to a real-time clock. We recommend that you enter `-1` (inherited sample time) in the **Sample time** parameter, and use the Vector CAN Transmit block inside a triggered subsystem in order to transmit a message upon an event received.

Watchdog

Purpose

In the event of an application failure, time out and reset processor

Library

Embedded Target for Motorola MPC555

Description

The Watchdog block lets you set the timeout period for the watchdog timer. The watchdog timer is a safety feature that is used to monitor correct behavior of the application. The timer is loaded with an initial value and counts down from this value. If the timer ever reaches zero, a watchdog timeout occurs, forcing a processor reset.

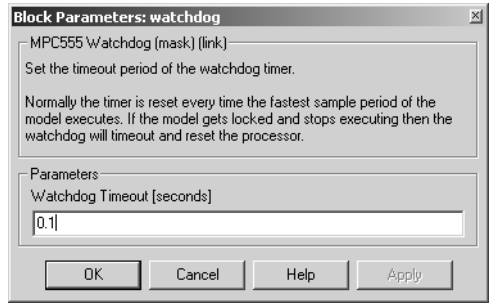
In normal operation, the watchdog timer is reloaded at a regular intervals by the application code; this occurs at a higher frequency than the **Watchdog Timeout** parameter period. Therefore the counter never reaches zero and a processor reset is never triggered.

In the event of a software failure that causes the application to lock up, the watchdog timer will not be serviced. Therefore, it will time out when the counter reaches zero. This in turn causes a processor reset, which restarts the application.

You do not need to include a Watchdog block in your model unless you want to change the **Watchdog Timeout** parameter period to a value other than the default. By default, the watchdog timer is enabled and the timeout period is set to the largest possible value, which is several seconds.

Note that the Watchdog block has neither input nor output connections.

Dialog Box



Watchdog Timeout

The **Watchdog Timeout** period must be set to a value that is larger than the fastest sample rate in the system. To set the **Watchdog Timeout** period, place a Watchdog block anywhere in the model and open its dialog box.

Index

A

algorithm export 4-2
ASAP2 files, generating 3-35

B

blocks
  CAN Calibration Protocol 5-12
  CAN Message Filter 5-17
  CAN Message Packing 5-19
  CAN Message Packing (CANdb) 5-21
  CAN Message Unpacking 5-27
  CAN Message Unpacking (CANdb) 5-29
  MIOS Digital In 5-35
  MIOS Digital Out 5-37
  MIOS Digital Out (MPWMSM) 5-39
  MIOS Pulse Width Modulation Out 5-41
  MIOS Waveform Measurement 5-44
  MPC555 Resource Configuration 5-47
  QADC Analog In 5-59
  QADC Digital In 5-64
  TouCAN Error Count 5-67
  TouCAN Fault Confinement State 5-68
  TouCAN Interrupt Generator 5-70
  TouCAN Receive 5-72
  TouCAN Soft Reset 5-75
  TouCAN Transmit 5-76
  TouCAN Warnings 5-77
  Vector CAN Configuration 5-78
  Vector CAN Receive 5-83
  Vector CAN Transmit 5-86
  Watchdog 5-88

C

CAN Calibration Protocol (CCP) 5-12
CAN Calibration Protocol block 5-12
CAN Message Filter block 5-17
CAN Message Packing block 5-19
CAN Message Unpacking block 5-27
CANdb
  Message Packing block 5-21
  Message Unpacking block 5-29
code analysis report 4-3
Configuration Class blocks 5-8
cosimulation 2-2

D

demos for Embedded Target for Motorola MPC555 viii
device driver blocks
  input data types 5-6
  input scaling 5-6
  output data types 5-6
  output scaling 5-6
downloading code to target 3-23
  application code 3-27
    to flash memory 3-28
    to RAM 3-27
  boot code 3-25
    via BDM port 3-26
    via CAN 3-26

E

Embedded Target for Motorola MPC555
  demos viii
  feature summary 1-6

I

installation of Embedded Target for Motorola MPC555 xii

M

MIOS Digital In block 5-35
MIOS Digital Out (MPWMSM) block 5-39
MIOS Digital Out block 5-37
MIOS Pulse Width Modulation Out block 5-41
MIOS Waveform Measurement block 5-44
MPC555 Resource Configuration object 5-47

O

ODBC
  configuring data source for 5-25
  using with CANdb message packing block 5-22
  using with CANdb message unpacking block 5-31

P

PIL (processor-in-the-loop) cosimulation 2-2
  benefits of 2-2
  getting started tutorial 2-5
  hardware connections for 2-5
  in plant/controller simulation 2-3
  preparation for 2-5
  technical overview of 2-3
PIL (processor-in-the-loop) target 2-2
  files and directories created by 2-23
  in cosimulation 2-14
  in SIL simulation 2-21
  using in closed-loop simulation 2-21

Q

QADC Analog In block 5-59
QADC Digital In block 5-64

R

real-time target
  introduction 3-2
  tutorial 3-4
    code generation 3-10
    example model for 3-6
    prerequisites for 3-4
    using pass-through for device drivers 3-9

S

software-in-the-loop (SIL) simulation 2-21

T

target hardware setup
  communications ports 1-12
  jumper settings 1-12
TouCAN Error Count block 5-67
TouCAN Fault Confinement State block 5-68
TouCAN Interrupt Generator block 5-70
TouCAN Receive block 5-72
TouCAN Soft Reset block 5-75
TouCAN Transmit block 5-76
TouCAN Warnings block 5-77
typographical conventions (table) xii

V

Vector CAN Configuration block 5-78
Vector CAN Receive block 5-83
Vector CAN Transmit block 5-86

W

Watchdog block 5-88
watchdog timer 5-88