

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
comp.soft-sys.matlab

Web
Newsgroup



support@mathworks.com
suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Technical support
Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000

Phone



508-647-7001

Fax



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

Mail

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.

Diab and SingleStep are registered trademarks of WindRiver Systems.

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)
	December 2002	Online only	Version 1.1 (Release 13+)

Preface

Installing the Embedded Target for Motorola MPC555 . . .	viii
Using This Guide	ix
Required and Related Products	x
Typographical Conventions	xii

Product Overview

1

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	1-10
Setting Target Preferences	1-11
Configuring the Embedded Target for Motorola MPC555 for Your Cross-Development Toolchain	1-11

Run Test Program	1-15
Download Boot Code to Flash Memory	1-15

Demos

2

Embedded Target for Motorola MPC555 Demos	2-2
Overview of Demos	2-4
Overview of Block Libraries	2-4
MPC555 Real-Time LED Demo	2-5
MPC555 Real-Time I/O Demo	2-6
MPC555 CAN Communication Protocol Demo	2-6
MPC555 Real-Time CAN Messaging Demo	2-7
MPC555 I/O Host Demo	2-7
MPC555 CANdb Host Demo	2-8
MPC555 PIL Fuelsys Demo	2-9
MPC555 Fuelsys Demo with Pass Through Drivers	2-10

Generating Stand-Alone Real-Time Applications

3

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	3-11
Downloading the Application to RAM via CAN	3-13

Downloading the Application to RAM via BDM	3-17
Downloading Boot and Application Code	3-20
RAM vs. Flash Memory	3-20
Overview of Memory Organization and the Boot Process	3-21
Downloading Boot Code	3-23
Downloading Application Code	3-25
Downloading Boot or Application Code via CAN Without Manual CPU Reset	3-32
Boot Code Parameters for CAN Download	3-33
Generating ASAP2 Files	3-36
ASAP2 File Generation Procedure	3-37
Data Acquisition (DAQ) List Configuration	3-38
Summary of the Real-Time Target	3-40
Code Generation Options	3-40
Requirements and Restrictions	3-43

PIL Cosimulation

4

Overview of PIL Cosimulation	4-2
Why Use Cosimulation?	4-2
How Cosimulation Works	4-3
Tutorial 1: Building and Running a PIL Cosimulation	4-5
Before You Begin	4-5
Hardware Connections	4-5
The Demo Model	4-5
Setting Up the Model	4-8
Building PIL and Simulation Components	4-11
Using the Demo Model In a PIL Cosimulation	4-14
Tutorial 2: Modifying and Rebuilding the Controller	4-17

Modifying the Controller	4-17
Rebuilding the Controller and Cosimulating	4-19

Tutorial 3: Using the Demo Model In Simulation 4-21

PIL Target Summary	4-22
Code Generation Options	4-22
Build Process Files and Directories	4-24
Restrictions	4-25

Algorithm Export and Code Analysis Reporting

5

Algorithm Export Target	5-2
Code Analysis Reporting	5-3
Algorithm Export Target Summary	5-5
Code Generation Options	5-5
Restrictions	5-5

Block Reference

6

The Embedded Target for	
Motorola MPC555 Block Libraries	6-2
Using Block Reference Pages	6-3
Blocks Organized by Libraries	6-4
MPC555 Driver Library	6-4
Data Type Support and Scaling for	
Device Driver Blocks	6-7
Configuration Class Blocks	6-11
CAN Message Blocks and CAN Drivers Libraries	6-12

Alphabetical List of Blocks 13

Toolchains and Hardware

A

Setting Up Your Toolchain A-2

**Setting Up Your Installation with
Diab Cross-Compiler and SingleStep Debugger A-3**

**Setting Up Your Installation with
Metrowerks CodeWarrior A-6**

Setting Up Your Target Hardware A-9

CAN Hardware and Drivers A-12

Configuration for Nondefault Hardware A-13
 Hardware Clock Configuration A-13

Preface

This section includes the following topics:

Installing the Embedded Target for
Motorola MPC555 (p. viii)

Installation of the product.

Using This Guide (p. ix)

Suggested path through this document to get you up and running quickly with the Embedded Target for Motorola® MPC555.

Required and 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)

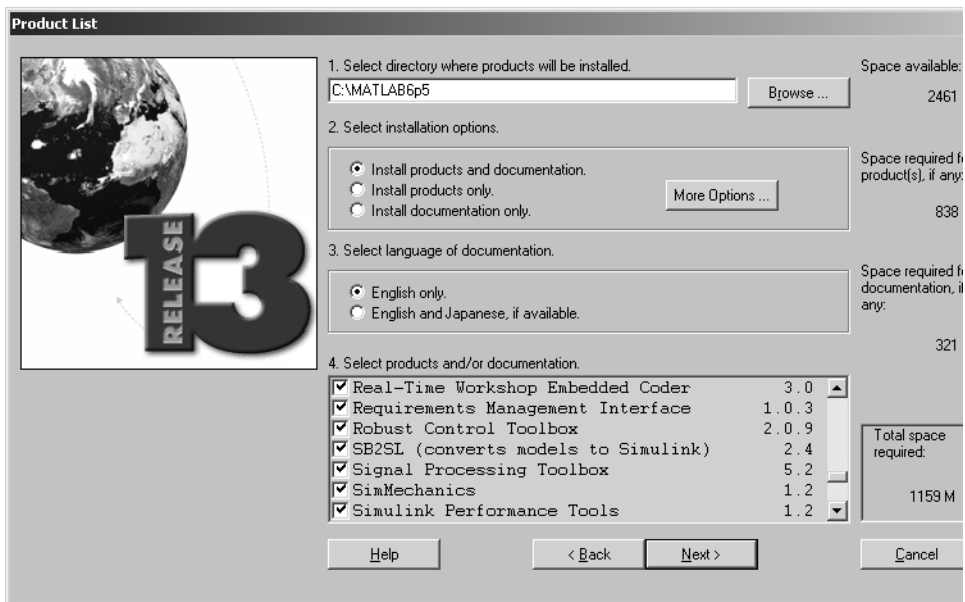
Formatting conventions used in this document.

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 1-10.
- 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.
- If you are interested in processor-in-the-loop (PIL) cosimulation, read Chapter 4, “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 4-5.
- Then, for in-depth information about the device drivers and other blocks supplied with Embedded Target for Motorola MPC555, see Chapter 6, “Block Reference.” It is particularly important to read “MPC555 Resource Configuration” on page 6-49, as the MPC555 Resource Configuration block is required to use most of the device driver blocks.
- See also “Embedded Target for Motorola MPC555 Demos” on page 2-2.

Required and Related Products

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

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

Optional — if you wish to implement the CAN Calibration Protocol (for example, for downloading without manual processor reset) by using the CAN Calibration Protocol block, you also need

- Stateflow® 5.0(Release 13) and Stateflow Coder

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

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.

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB. The blockset includes 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
Simulink	Design and simulate continuous- and discrete-time systems
Stateflow	Design and simulate event-driven systems
Stateflow Coder	Generate C code from Stateflow charts

Operating System Requirements

The Embedded target for Motorola MPC555 is a PC-Windows only product. The product has been tested on Microsoft Windows NT, 2000, and XP. The product is not supported for Windows 98.

You can see the system requirements for MATLAB online at

<http://www.mathworks.com/products/system.shtml/Windows>

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 <code>A = 5</code>
Function names, syntax, filenames, directory/folder names, and user input	Monospace font	The <code>cos</code> function finds the cosine of each array element. Syntax line example is <code>MLGetVar ML_var_name</code>
Buttons and keys	Boldface with book title caps	Press the Enter key.
Literal strings (in syntax descriptions in reference chapters)	Monospace bold for literals	<code>f = freqspace(n, 'whole')</code>
Mathematical expressions	<i>Italics</i> 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 <code>A =</code> <code>5</code>
Menu and dialog box titles	Boldface with book title caps	Choose the File Options menu.
New terms and for emphasis	<i>Italics</i>	An <i>array</i> is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	<code>[c,ia,ib] = union(...)</code>
String variables (from a finite list)	<i>Monospace italics</i>	<code>sysc = d2c(sysd, 'method')</code>

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. 1-10)	Overview of setting up your development tools and hardware to work with the Embedded Target for Motorola MPC555, and verifying correct operation.
Setting Target Preferences (p. 1-11)	Configuring environmental settings and preferences associated with the Embedded Target for Motorola MPC555 for use with specific 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

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

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 or the Time Processor Unit 3 (TPU) modules
 - Analog input via the Queued Analog-to-Digital Converter (QADC64)
 - Digital input and output via the MIOS or TPU
 - 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
 - Driver blocks to support other functions of the TPU modules—Fast Quadrature Decode, New Input Capture/Input Transition Counter, and Programmable Time Accumulator
 - Serial transmit and receive
 - 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.

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 singlerate 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

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.

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. These can be used in simulation to connect to a real CAN bus.
- 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 can automatically produce 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.

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

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:

- Examining the demo model with a plant model and 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
 - 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 5-3 for details.

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 PIL 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 4-5. The tutorial covers the following topics:

- Opening the demo model and examining the plant model and controller
- Selecting the PIL target
- Generating the Embedded Real-Time (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 5-3 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 5-2.

Like the PIL and RT targets, the AE target supports generation of code analysis information. See “Code Analysis Reporting” on page 5-3 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 Phytex phyCORE-MPC555 development board, and we document specific settings and procedures for use with the Phytex 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 “Required and Related Products” in the Preface 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 (version 4.4b), and Wind River Systems SingleStep debugger (version 7.6.2)
- Metrowerks CodeWarrior for Embedded PowerPC (version 6.0)
The full feature set (PIL, RT, and AE targets) is supported for both toolchains.

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 1-10.

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:

- You must set up your development environment and your target hardware. Information on these settings can be found in “Toolchains and Hardware” on page A-1:
 - “Setting Up Your Target Hardware” on page A-9
 - “Setting Up Your Toolchain” on page A-2
- You must configure Embedded Target for Motorola MPC555 to work with your toolchain by specifying the locations of your compiler and debugger. This is described in the section “Setting Target Preferences” on page 1–11.
- We supply a test program to verify your installation. This confirms you have correctly set up your toolchain, target preferences and development board. See “Run Test Program” on page 1–15.
- The next step is to download boot code to the flash memory of your MPC555. See “Download Boot Code to Flash Memory” on page 1–15.

Note You must download the new boot code if you have used a previous release of Embedded Target for Motorola MPC555 with your hardware. See “Downloading Boot Code” on page 3-23.

Once you have completed these steps we suggest you run the tutorials in subsequent sections to get started with the Embedded Target for Motorola MPC555.

Setting Target Preferences

This section describes how to set target preferences associated with the Embedded Target for Motorola MPC555. These settings persist across MATLAB sessions and different models. 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.

Configuring the Embedded Target for Motorola MPC555 for Your Cross-Development Toolchain

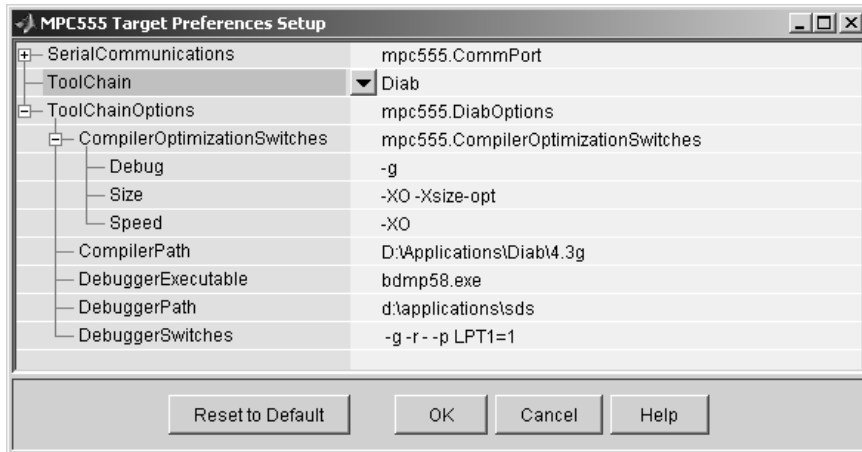
When you set up the Embedded Target for Motorola MPC555, you must make sure you localize the settings to suit your PC and cross-development toolchain. It is important that you set the correct path to your compiler and debugger using the **MPC555 Target Preferences** dialog box.

Instructions for setting up specific third-party toolchains for use with the Embedded Target for Motorola MPC555 are in “Toolchains and Hardware” on page A-1.

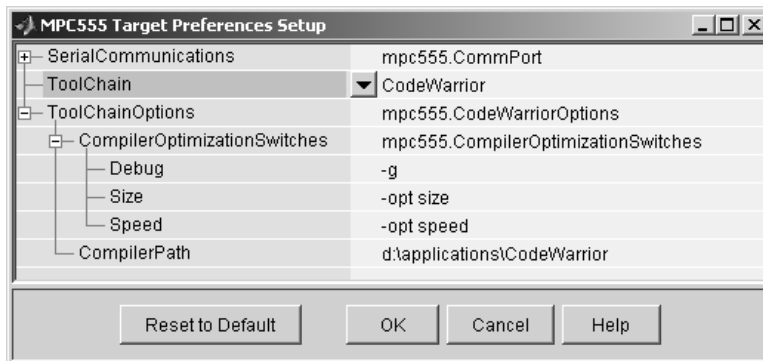
You can modify target preference objects via the new **MPC555 Target Preferences** dialog box:

- 1 Select **Start -> Simulink -> Embedded Target for Motorola MPC555 -> Target Preferences**.

This opens the **MPC555 Target Preferences** dialog box where you can edit the settings for your cross-development environment.



- 2 Select Diab or CodeWarrior from the drop-down **Toolchain** menu
- 3 Type the correct path into **CompilerPath**.
- 4 For SingleStep you must also type the correct path into **Debugger Path**. This is not necessary for CodeWarrior as the compiler and debugger are integrated. The example below shows the CodeWarrior preferences.



DebuggerSwitches Setting

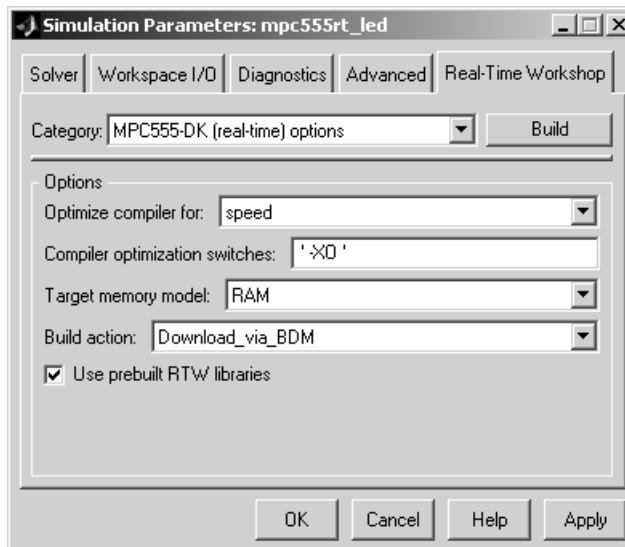
This setting is specific to SingleStep. See “Setting Target Preferences for Diab and SingleStep” on page A-4.

Compiler Optimization Switches Settings

For both toolchains these settings configure optimizations for speed, size, and debug. The settings are compiler specific. These properties can be edited from the **MPC555 Target Preferences** dialog box or from the **Simulation Parameters** dialog box, described below. The defaults should be adequate for most rapid prototyping purposes.

If you want to alter these settings, consult your compiler documentation for specific optimizations. To edit the settings,

- If you want your changes to apply to many models, edit them within the **MPC555 Target Preferences** dialog box. Your settings will appear within the **Simulation Parameters** dialog box in the **Compiler optimization switches** field when you select speed, size or debug from the **Optimize compiler for** options in the drop-down menu. You must choose MPC555-DK (real-time) options from the **Category** menu on the **Real-Time Workshop** tab to reach these settings, as shown in this example.



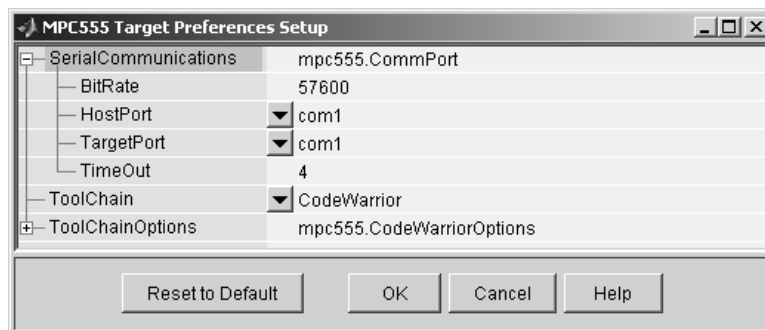
- If you want to customize these settings for a single model, edit them from the **Simulation Parameters** dialog box. **Optimize compiler for** will change to custom and the defaults for these settings will remain unchanged in the **MPC555 Target Preferences** dialog box. When you edit these settings, you

must place single quotation marks at either end of the string. These settings are then applied to model code.

Use Prebuilt RTW Libraries. This check box option (selected by default) saves a considerable amount of time during the build process, as the libraries do not need to be recompiled every time. However, note this uses the defaults we have chosen for compiler optimization switches. These defaults are designed for rapid prototyping mode. If you are going to switch to production code development and want to fine tune the settings, you should clear this check box option. Then the custom optimization switches you set in the **Real-Time Workshop Simulation Parameters** dialog box will be applied to the library code as well as the model code.

Serial Communications

These target preferences relate to Processor-in-the-Loop (PIL) cosimulation only.



- **BitRate** — Bit rate (in bps) for host/target communications. The default is 57600.
- **HostPort** — Host serial port for host/target communications. Select from com1 to com8; the default is com1.
- **TargetPort** — Target board serial port for host/target communications. Select from com1 to com8; the default is com1.
- **TimeOut** — Time-out value (in seconds) for the serial communications port. The default is 4.

Run Test Program

To verify your setup, you can download and run a simple test program on the phyCORE-MPC555 board:

- 1 Select **Start -> Simulink -> Embedded Target for Motorola MPC555 -> Run Embedded Target Test Application.**
- 2 To answer the question Do you want to run the application? Type **y** at the command line.

If you have not set up your target preferences properly the process will stop and ask you to do this now.

Watch as your toolchain downloads and runs the application on your phyCORE board. Successful execution results in a blinking LED.

You have now verified your installation and are ready to begin working with the Embedded Target for Motorola MPC555.

Download Boot Code to Flash Memory

The next step is to download the boot code to flash memory, if you have not already done so. 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. Note if you are upgrading from a previous release of Embedded Target for Motorola MPC555, you must download the new 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. See “Overview of Memory Organization and the Boot Process” on page 3-21 for more information.

The process is very simple. Follow these two steps:

- 1 Connect the BDM cable to the target.
- 2 Do one of the following:

- Select **Start -> Simulink -> Embedded Target for Motorola MPC555 -> Install MPC555 Bootcode.**
- Alternatively, at the command line type
`mpc555_bdm_bootcode_download`

Your development tools will start and execute a command to install the boot code. Wait till the process stops, then exit the debugger. The boot code should now be installed.

Once you have successfully downloaded boot code to your target, you have completed your installation and are ready to use all the features of the Embedded Target for Motorola MPC555. If necessary, please consult your toolchain documentation.

We suggest you now turn to Chapter 3, “Generating Stand-Alone Real-Time Applications” to get hands-on experience with using the Embedded Target for Motorola MPC555 and your toolchain to generate, download, and execute application code on your phyCORE-MPC555 board. You can then also work through the tutorials in Chapter 4, “PIL Cosimulation” to start using processor-in-the-loop simulation for development via the Embedded Target for Motorola MPC555.

Demos

This section includes the following topic:

Embedded Target for Motorola
MPC555 Demos (p. 2-2)

Hyperlinks to demo models that illustrate product
features and how to use them.

Overview of Demos (p. 2-4)

Descriptions of demos and the features they illustrate.

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 start the demos by clicking on the hyperlinks 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
mpc555rt_led	Simple model demonstrating use of device driver blocks provided with the Embedded Target for Motorola MPC555 real-time target. See “MPC555 Real-Time CAN Messaging Demo” on page 2–7 and “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 can be run on target hardware or you can run the model in simulation. The mpc555rt_io model (on hardware or in simulation) can communicate via a CAN channel with the mpc555rt_iohost model, running in Simulink. This demo requires Vector-Informatik CAN hardware and drivers. 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. See “MPC555 Real-Time I/O Demo” on page 2–6 and “MPC555 I/O Host Demo” on page 2–7.
mpc555rt_ccp	Demonstrates use of the CAN Calibration Protocol. See “MPC555 CAN Communication Protocol Demo” on page 2–6 and the block reference page “CAN Calibration Protocol” on page 6-14.

Embedded Target for Motorola MPC555 Demos

Command	Demo Topic
mpc555rt_candb and mpc555rt_candbhost	This model illustrates how to use the CANdb Packing and Unpacking blocks to construct and inspect CAN messages. You can run the mpc555rt_candb model in simulation or generate code from a subsystem to run on MPC555 target hardware. If you have Vector-Informatik CAN hardware and drivers installed, you can use the companion model mpc555rt_candbhost to exchange CAN messages with the mpc555rt_candb model (running either in Simulink simulation or on hardware). See “MPC555 Real-Time CAN Messaging Demo” on page 2–7 and “MPC555 CANdb Host Demo” on page 2–8.
mpc555pil_fuelsys	Model configured for building a configurable subsystem for use in PIL cosimulation. See “MPC555 PIL Fuelsys Demo” on page 2-9 and “Tutorial 1: Building and Running a PIL Cosimulation” on page 4-5.
mpc555dd_fuelsys	This demo is an example of a single model deployment paradigm. With the Embedded Target for Motorola MPC555 you can use the same model for development from simulation to code deployment. This model demonstrates the use of a single closed-loop simulation model for simulation, software-in-the-loop (SIL) simulation, PIL cosimulation, and real-time deployment. The MPC555 I/O device driver blocks provided have a pass through feature that allows you to work in any of these modes without any modifications to your model. Input and output signals are conditioned (with respect to data type and scaling) appropriately for use with the device driver blocks. See the instructions in “MPC555 Fuelsys Demo with Pass Through Drivers” on page 2-10.

Overview of Demos

We provide demos to illustrate how to use the features of Embedded Target for Motorola MPC555. They cover the following categories

- Real-time execution on an MPC555
 - “MPC555 Real-Time LED Demo” on page 2-5
 - “MPC555 Real-Time I/O Demo” on page 2-6
 - “MPC555 CAN Communication Protocol Demo” on page 2-6
 - “MPC555 Real-Time CAN Messaging Demo” on page 2-7
- Non-real-time execution in Simulink
 - “MPC555 I/O Host Demo” on page 2-7
 - “MPC555 CANdb Host Demo” on page 2-8
- Processor in the loop cosimulation
 - “MPC555 PIL Fuelsys Demo” on page 2-9
- A single model deployment paradigm illustrating pass through I/O device drivers.
 - “MPC555 Fuelsys Demo with Pass Through Drivers” on page 2-10

Overview of Block Libraries

The blocks used in the demos to illustrate the functionality of the Embedded Target for Motorola MPC555 are contained in the following block libraries:

- MPC555 I/O driver library

This library provides I/O driver blocks for you to add to your controller model for interfacing to hardware. Supported I/O includes: MIOS, QADC, TouCAN, TPU3 and Serial, with a link to all the demos. To see this library, at the command line type

```
mpc555drivers
```
- CAN Message packing library

You can add CAN Message Blocks to your controller, enabling you to send CAN messages containing signal data from your controller subsystem using

either standard or extended CAN identifiers. To see this library, at the command line type

```
canblocks
```

- **Vector-Informatik host CAN library**

A set of CAN blocks supports Vector CAN drivers on your host computer. This lets you communicate between your host machine and the MPC555 target to capture CAN messages and display signals using Simulink Scope blocks. Alternatively, you can export Simulink signals via the CAN blocks to send signal data to your MPC555 target via CAN. To see this library, at the command line type

```
vector_candrivs
```

For complete information on all the blocks provided with Embedded Target for Motorola MPC555, see Chapter 6, “Block Reference.”

MPC555 Real-Time LED Demo

The real-time LED demo is a simple model for use in generating code for real-time execution. During execution on the Phytex phyCORE 555, digital I/O channels from the MPC555 are connected to LEDs. This model is useful to confirm successful downloading and execution of generated code on the MPC555 using your toolchain.

To open this model, click the link below or at the command line type

```
mpc555rt_led
```

Right-click on the Target_LED block, and select **Real-Time Workshop** -> **Build Subsystem**. Then use the CAN Download Control Panel to download the generated Target_LED_RAM.s19 file to your Phytex phyCORE-555 board and observe the blinking LEDs.

There is a tutorial describing how to use this model to generate and download a stand-alone real-time application. See “Tutorial: Creating a New Application” on page 3-4

MPC555 Real-Time I/O Demo

The real-time I/O demo illustrates a variety of I/O blocks configured as inputs and outputs within a single model. This model, as the name implies, is used for code generation and running in real time. This model serves as a reference when using any I/O device driver blocks provided with the Embedded Target for Motorola MPC555.

This model illustrates how to use MPC555 I/O blocks for digital input/output, Pulse Width Modulation (PWM), waveform measurement, analog input, and Controller Area Network (CAN). PWM and digital I/O are demonstrated via Modular I/O System (MIOS) and Time Processor Unit (TPU3).

You can run this model in simulation, or generate code for the subsystem Target_IO and run it on MPC555 hardware.

To open this model, click the link below or at the command line type

```
mpc555rt_io
```

If you have a CAN board and Vector-Informatik CAN drivers installed, you can exchange CAN messages with the companion model `mpc555rt_iohost`. The subsystem code communicates, via a CAN channel, with the `mpc555rt_iohost` model, running in Simulink. See the instructions in “MPC555 I/O Host Demo” on page 2–7.

MPC555 CAN Communication Protocol Demo

During code generation for the CCP demo model, an ASAP2 file is generated. Using the ASAP2 file, you can use CAN calibration tools to view data from the target MPC555 collected via CAN. When an external tool such as CANape is used to modify parameter values, the CCP protocol provides a means of monitoring signals and altering the parameter values in the controller code running on the target MPC555.

The primary model contains a variety of signals that can be monitored including the output signal of a simple counter, with tunable step size. The generated ASAP2 file allows a calibration tool to tune STEP_PARAM and monitor signals such as COUNTER_SIGNAL.

To open this model, click the link below or at the command line type

```
mpc555rt_ccp
```

MPC555 Real-Time CAN Messaging Demo

This model illustrates how to use the CANdb Packing and Unpacking blocks to construct and inspect CAN messages. You can run the `mpc555rt_candb` model in simulation or generate code from a subsystem to run on MPC555 target hardware.

To open this model, click the link below or at the command line type

```
mpc555rt_candb
```

If you have Vector-Informatik CAN hardware and drivers installed, you can use the companion model `mpc555rt_candbhost` to exchange CAN messages with the `mpc555rt_candb` model (running either in Simulink simulation or on hardware). See the instructions in “MPC555 CANdb Host Demo” on page 2-8

You can examine the settings in these models to see the effect of mode signals and how to pack signals into messages depending on the mode signal, and also how to map more than one signal to the same location in the message. See the block reference page “CAN Message Unpacking (CANdb)” on page –30 for more information about signal types and modes.

MPC555 I/O Host Demo

The MPC555 I/O Host Demo is run as a Simulink simulation which is non-real-time. During simulation, this model can be used to view signals from the MPC555 when running the `mpc555rt_io` model subsystem code on the Phytex board, or in simulation.

The I/O Host demo model is a companion model that provides signal viewing using Simulink Scope blocks. The subsystem code communicates, via a CAN channel, with the `mpc555rt_iohost` model, running in Simulink.

To open this model, click the link below or at the command line type

```
mpc555rt_iohost
```

The two options are

- 1 Generate code for the `mpc555rt_io` demo subsystem and download it to an MPC555. Configure the `mpc555rt_iohost` demo to look at a physical CAN channel by locating and double-clicking the Vector Configuration block and selecting a CAN channel under **Channel parameter**. Run the

`mpc555rt_iohost` model in simulation and view the signals from the code running on the MPC555.

- 2 Configure the `mpc555rt_io` demo to use a virtual CAN channel by locating and double-clicking the Vector Configuration block and choosing `Virtual 1` for the **Channel parameter**. Run the `mpc555rt_io` demo in simulation, configure the `mpc555rt_iohost` demo to use a virtual CAN channel in the same way (choose `Virtual 2`), and run the `mpc555rt_iohost` in simulation. Then the two models in simulation will talk to each other by exchanging messages via the virtual CAN channel.

See “MPC555 Real-Time I/O Demo” on page 2-6.

MPC555 CANdb Host Demo

The demo `mpc555rt_candbhost` is intended for use as a companion model to `mpc555rt_candb` which can run on the MPC555 target hardware or in simulation. You can run both these models to exchange CAN messages (in the same way as the two I/O demos — see also “MPC555 I/O Host Demo” on page 2-7).

To use the model `mpc555rt_candbhost`, you must have suitable CAN hardware and CAN drivers from Vector-Informatik installed on your PC. If your PC CAN hardware has two channels available, you can run both models in simulation using a physical loopback connection between the two CAN channels on your PC. See “MPC555 Real-Time CAN Messaging Demo” on page 2-7.

To open this model, click the link below or at the command line type

```
mpc555rt_candbhost
```

Start the model `mpc555rt_candb` running on the MPC555 target hardware, or in simulation.

Open the `CANdb_Host` subsystem and double-click on the block Vector CAN Configuration. Set the **Channel parameter** according to your installed CAN hardware. Make sure you also set this parameter in the companion model. For example, to run both models in simulation, choose `Virtual 1` and `Virtual 2`.

Run `mpc555rt_candbhost` in Simulink on your host PC.

CAN messages are sent between this model and the companion model (whether in simulation or on the MPC555). Unpacked messages are displayed in the

numeric display blocks, and on the scopes. The messages transmitted by this model are used to control MIOS Digital Outputs on the MPC555.

MPC555 PIL Fuelsys Demo

Processor-in-the-loop (PIL) allows you to validate the production code generated by Real-Time Workshop Embedded Coder from within the framework of a closed-loop plant model simulation (in non real-time).

This demo is based on the original Stateflow Fuelsys demo. The model is altered to allow the plant to run in Simulink, and export the plant output signals via RS232 to the MPC555 target. Upon receiving signals via RS232, the MPC555 runs a single sample interval of the controller code on the MPC555. Outputs of the controller are then sent back to the host via RS232. This enables you to run a closed-loop simulation using your host PC to run the plant model and your MPC555 target to run the controller code. This cosimulation runs “non-real-time.”

To open this model, click the link below (if you are reading online) or at the command line type

```
mpc555pil_fuelsys
```

There is a tutorial with step-by-step instructions on the use of this model in PIL cosimulation. See “Tutorial 1: Building and Running a PIL Cosimulation” on page 4-5.

MPC555 Fuelsys Demo with Pass Through Drivers

This demo is an example of a single model deployment paradigm. With the Embedded Target for Motorola MPC555 you can use the same model for development from simulation to code deployment. This model demonstrates how to prepare a single closed-loop simulation model for

- normal simulation
- processor-in-the-loop (PIL) co-simulation
- real-time deployment

Pass through drivers allow you to work in any of the 3 modes described above without any modifications to your model.

This variation of the Fuelsys demo illustrates the use of pass-through I/O device drivers. These drivers have two modes of operation. When used for Simulink closed-loop simulation, driver inputs are passed through to the controller. Driver outputs from the controller are passed through to the plant model.

When the controller subsystem containing the pass through drivers is used for generating real-time code, the drivers operate in the second mode of operation; they function as I/O device drivers connected to hardware. Physical signals such as voltage are converted to a digital value which is then used by the control algorithm.

The primary advantage of pass through driver technology is that it enables you to maintain a single model that can be used for closed-loop simulation in Simulink as well as for code generation where physical interfaces to external hardware require device drivers. This removes the need to maintain a simulation model separately from a controller/real-time model.

The model `mpc555dd_fuelsys` contains an example of a pass through ADC. Open the controller subsystem named `fuel rate controller`, then open the block `Device Inputs`. During closed-loop simulation, the block `Analog In` behaves as a wire and passes signals through with only scaling and data type alterations. During code generation, this is replaced with device driver code allowing the physical ADC to be used by the controller code. Similarly, the model illustrates a pass through driver for a Pulse Width Modulation signal.

To open this model, click the link below or at the command line type

```
mpc555dd_fuelsys
```


1 Select Tools -> Real-Time Workshop -> Simulation Parameters.

Notice that the selected **System target file** is `mpc555pil.tlc`. The model is preconfigured for PIL.

2 Select MPC555-DK (PIL) options from the Category menu.**3 Select Download_and_run from the Build action drop-down menu.****4 Right-click on the fuel rate controller subsystem and select Real-Time Workshop->Build Subsystem. Click Build in the following dialog to continue the code generation process.**

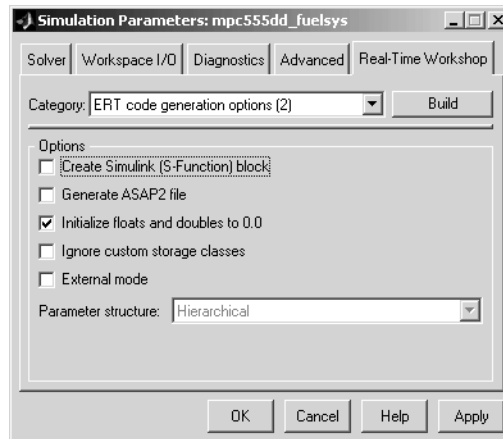
- Code is generated in the `fuel_mpc555pil` directory.
- An S-Function "wrapper" block is created and presented in an Untitled model.
- Your cross compiler (Diab or CodeWarrior) is invoked to create a binary file for the MPC555.
- Your debugger (SingleStep or CodeWarrior) is invoked and downloads the code to the target. The model code starts running on the target.
- You can configure the ERT options (in Simulation Parameters) to generate and display an HTML code generation report.

5 In the newly created library called `fuel_lib` use the double-click convenience button ("==>") to replace the original fuel rate controller subsystem in the model with the generated configurable subsystem.**6 Right click on the fuel rate controller subsystem in your model and select **Block choice**. You have three choices, corresponding to the original subsystem, SIL, or PIL. For PIL choose `fuel rate controller(PIL)`.****7 Start the Simulink simulation of the model `mpc555dd_fuel_sys` by clicking the play button. View processor-in-the-loop simulation results in the Simulink scope blocks.**

Note you can also run this model in software-in-the-loop (SIL) mode by choosing `fuel rate controller (SIL)` from the **Block choice** 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.

To change configuration between PIL and Real-Time:

- 1 Select **Tools** -> **Real-Time Workshop** -> **Simulation Parameters**.
- 2 The **System target file** for PIL is `mpc555pil.tlc`. To switch to real-time deployment, click **Browse** and select `mpc555rt.tlc`.
- 3 For real-time ensure **Create Simulink (S-Function) block** is not selected, as shown below. This option is in the **Category** ERT code generation options (2).



- 4 Right click on the fuel rate controller subsystem in your model and select **Block choice**. You have three choices, corresponding to the original subsystem, SIL, or PIL. For real-time choose the original fuel rate controller.
- 5 Right-click on the fuel rate controller subsystem and choose **Real-Time Workshop** -> **Build Subsystem**. Click **Build** in the following dialog to continue the code generation process.

Code is generated in the `fuel_mpc555rt` directory.

For more information about using the Embedded Target for Motorola MPC555 for PIL, SIL and real-time deployment, see the tutorials contained in “PIL Cosimulation” on page 4-1 and “Generating Stand-Alone Real-Time Applications” on page 3-1.

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. 3-20)

A detailed discussion of the process of downloading code to the MPC555 RAM and flash memory.

Generating ASAP2 Files (p. 3-36)

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 and 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 together with your toolchain, 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.

See “Before You Begin” on page 3–4 for information on supported hardware and toolchains.

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 6, “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 drivers are analog input, digital I/O, PWM, serial, CAN using the MIOS, TPU, and TouCAN modules on the MPC555. See Chapter 6, “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 and ATI Vision. See “CAN Calibration Protocol” on page 6-14 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-through 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
- 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 3-20 for that information.

Before You Begin

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

- Phytex phyCORE-MPC555 development board
 - The tutorial model utilizes two LEDs on the phyCORE-MPC555 board. These LEDs are connected to pins MPI032B0 and MPI032B1 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.

- A supported toolchain for compiling and debugging. Currently supported toolchains are
 - Diab and SingleStep from Wind River Systems
 - CodeWarrior from MetrowerksSee “Setting Up Your Toolchain” on page A-2.
- If you want to download generated code to the target board via CAN, you will need a supported CAN card and drivers from Vector-Informatik. See “CAN Hardware and Drivers” on page A-12.

Configuring Embedded Target for Motorola MPC555

- Make sure that your target preferences are set correctly for your development tools:
 - Select **Start** -> **Simulink** -> **Embedded Target for Motorola MPC555** -> **MPC555 Target Preferences**.
This opens the Target Preferences GUI where you can edit the settings for your cross-development environment.
 - Select Diab or CodeWarrior from the drop-down **Toolchain** menu
 - Click the plus sign to open the **ToolChainOptions**, and type the correct path into **CompilerPath**.
 - For SingleStep you must also type the correct path into **Debugger Path**. This is not necessary for CodeWarrior as the compiler and debugger are integrated.

The first time you use Embedded Target for Motorola MPC555 you must use a toolchain 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 “Downloading Boot Code” on page 3-23.

Note If you are upgrading from a previous release of Embedded Target for Motorola MPC555 you must download the new boot code.

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. The path `matlabroot` is the location where MATLAB is installed.

- 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.

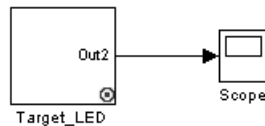


Figure 3-1: mpc555rt_led_demo Model, Root Level

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

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

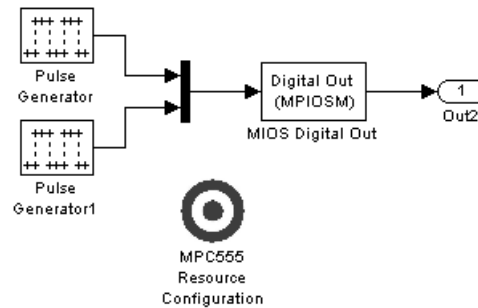
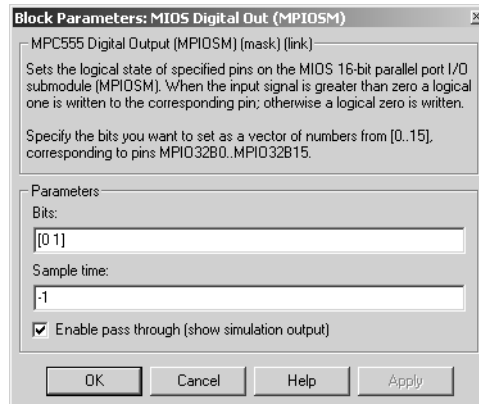


Figure 3-2: Target_LED Subsystem

In the Target_LED 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 (MPIO5M) 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 figure 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.

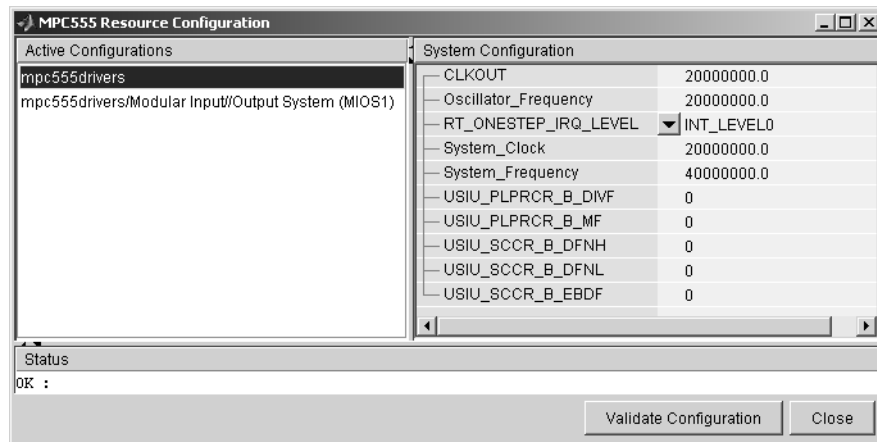
During simulation, the MIOS Digital Out block simply passes its input signal through to its output, and the square waves can be viewed on the Scope block. We will come back to this in the next section, “Using the Pass-Through Option in Simulation” on page 3–9.

In addition to the Pulse Wave, Mux, MIOS Digital Out, and Output blocks, the Target_LED 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 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 Target_LED 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 6-49.

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.

Pass through drivers allow you to use the same model for

- Normal simulation
- Processor-in-the-loop (PIL) co-simulation
- Real-time deployment

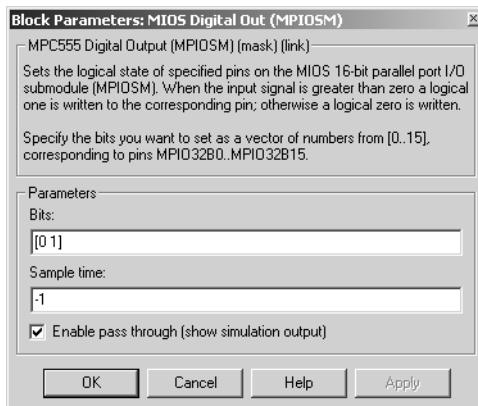
Using pass-through allows you to work in any of the 3 modes described above without any modifications to your model.

Pass-through I/O device drivers have two modes of operation. When used for Simulink closed-loop simulation, driver inputs are passed through to the controller. Driver outputs from the controller are passed through to the plant model, so you can create a feedback loop.

When the controller subsystem containing the pass through drivers is used for generating real-time code, the drivers operate in the second mode of operation; they function as I/O device drivers connected to hardware. Physical signals such as voltage are converted to a digital value, which is then used by the control algorithm.

The primary advantage of pass through driver technology is that it enables you to maintain a single model that can be used for closed-loop simulation in Simulink as well as for code generation where physical interfaces to external hardware require device drivers. This removes the need to maintain a simulation model separately from a controller/real-time model. You can examine an example of a pass-through driver block in this tutorial model.

Open the MIOS Digital Out block from the Target_LED 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 output port 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 Target_LED subsystem.

- 3 Stop the simulation manually (the simulation time is set to inf).

Note If you build a model with device driver blocks in pass-through mode, unnecessary extra code will be generated unless **Inline Parameters** is selected. This check box can be found in the **Simulation Parameters** dialog box on the **Advanced** tab.

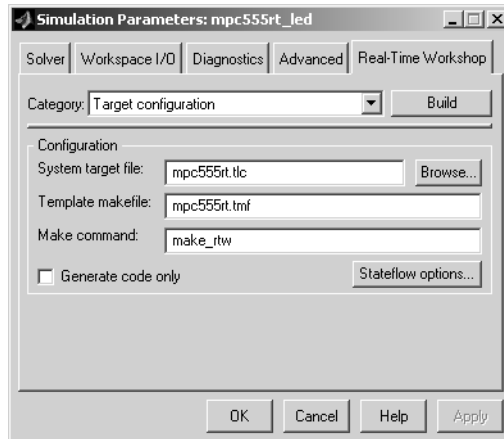
For more information about how to use pass-through for single model simulation and deployment, we provide a demo. See “MPC555 Fuelsys Demo with Pass Through Drivers” on page 2–10.

The next step in this tutorial is generating code.

Generating Code

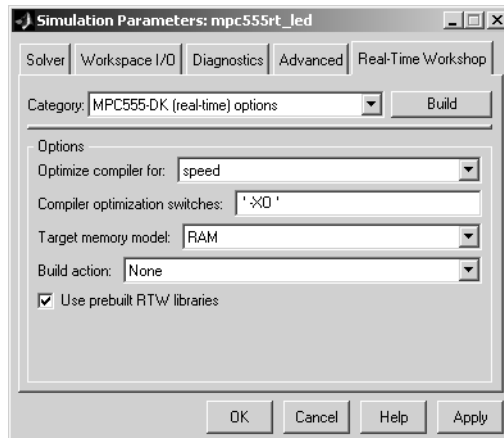
We will now generate application code:

- 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 right-clicking on the Target_LED subsystem and selecting **Real-Time Workshop -> Build subsystem**. Then click the **Build** button in the following dialog.
- 6 On successful completion of the build process, two files are created in the working directory:
 - a Target_LED_ram.s19: This file is for CAN download. It is code only, without symbols, suitable for execution on the target system.
 - b Target_LED_ram.elf: This file is for BDM download.

If debug is selected in the compiler optimization settings, the elf file will contain debugging symbols as well as code. These symbols are suitable for use with a symbolic debugger such as Wind River SingleStep or Metrowerks CodeWarrior. The default optimization setting is speed, so no symbols are included. Symbols are only generated for a debug build. See “CompilerOptimizationSwitches Settings” on page 1–13.

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 3-13.
- Via the BDM port, as described in “Downloading the Application to RAM via BDM” on page 3-17.

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, you will use the Download Control Panel utility to download the generated Target_LED_ram.s19 file to RAM on the target system. The s19 file is for download over CAN.

Target_LED_ram.elf is for BDM download, as described in the next section, “Downloading the Application to RAM via BDM” on page 3–17. Recall you can perform a debug build to include debugging symbols in the elf file.

Do the following before you begin:

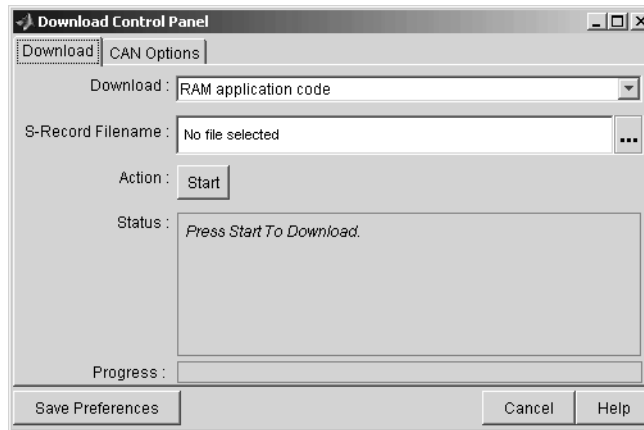
- Make sure that your Vector-Informatik CAN card and drivers are installed and configured properly. See “CAN Hardware and Drivers” on page A-12.

Make sure that the desired CAN port on the PC card is connected to the CAN A port on the target hardware.

- Make sure that you have set up your toolchain as described in “Toolchains and Hardware” on page A-1, and downloaded boot code to the flash memory of the MPC555 as described in “Downloading Boot Code” on page 3–23.
- 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 A-9.
- Cycle the power (or perform a hard reset) on your development board, to clear the RAM.

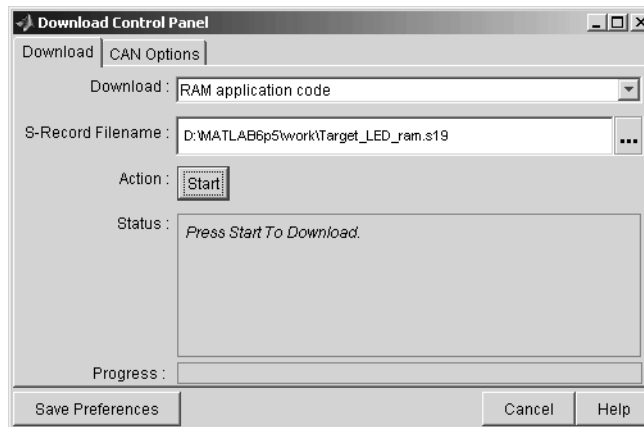
To download the generated `Target_LED_ram.s19` file to RAM:

- 1 Start the Download Control Panel in one of the following ways:
 - Use the MATLAB Start menu. Select **Start** -> **Simulink** -> **Embedded Target for Motorola MPC555** -> **Launch CAN Download**.
 - Type `candownload` at the MATLAB command prompt.
 - You can also start CAN download automatically at the end of the build process. Before you click **Build** in the **Simulation Parameters** dialog box, you can select **Launch CAN Download** from the **Build action** options. You can see an illustration of this dialog in “Generating Code” on page 3-11.
- 2 After using any of these three options, the **Download Control Panel** window opens.



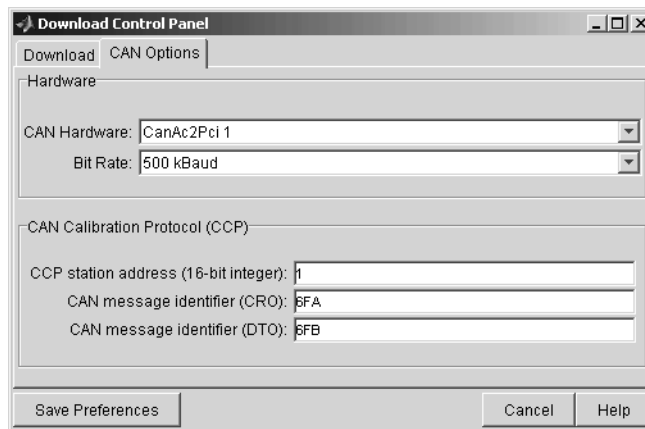
Note RAM application code is automatically selected in the **Download** menu.

- 3 Enter the name of the file to be downloaded into the **Filename** field. Alternatively, you can use the browse button (right of the edit box) to navigate to the desired file. The **Download Control Panel** should now appear as shown in this picture.



- 4 Click on the **CAN Options** tab. If necessary, select an appropriate card and port from the **CAN hardware** drop-down menu. The default settings for the

other parameters are appropriate for most cases. You can save your preferences by clicking the **Save Preferences** button. This picture shows the **CAN Options** configured for a Vector-Informatik CANAC2pci card, channel 1.



- 5 Click the **Download** tab. Then click the **Start** button.

When you click **Start**, the **Download Control Panel's Status** box changes to read Press reset or power-cycle your development board to start download.

- 6 Press the Reset button on your PhyCORE-MPC555 board (or cycle the power). The **Download Control Panel** changes its **Status** box to read CCP Connection OK. Please wait till completion or press Stop to terminate the download.

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

- 7 While downloading proceeds, progress messages are displayed in the **Download Control Panel** and the MATLAB Command Window. A successful download ends with a sequence similar the following Command Window messages:

```
Loading S-Record D:\MATLAB\work\Target_LED_ram.s19
-----
CCP Download with the following settings
-----
```

```
CRO : 1786
DTO : 1787
Baud : 500 kBaud
Bit Timing : presc = 1 sjw = 1 tseg1 = 8 tseg2 = 7 sam = 0
Channel : CanAc2Pci 1 : 3
File : D:\MATLAB\work\Target_LED_ram.s19
Station ID : 1
Download Type : RAM application code
```

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

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 CAN 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 bit rate of 50000 (500 kbaud):

```
btest32 50000
```

Downloading the Application to RAM via BDM

You can choose to automatically download to the target over BDM on completion of the build process. Follow these steps to generate, download and execute the `Target_LED_ram.elf` file to RAM on the target system. `Target_LED_ram.elf` can contain both code and symbols for use with the debugger if you perform a debug build. You will not perform a debug build in this tutorial, so the file will contain code only.

You can use Embedded Target for Motorola MPC555 to download application code via BDM to MPC555 RAM only. If you want to download application code

to MPC555 flash you can use CAN (as described in “Downloading the Application to RAM via CAN” on page 3-13). If you want to download the application to flash memory over BDM manually using your own tools, then the file you need to download is the S-record file *.s19.

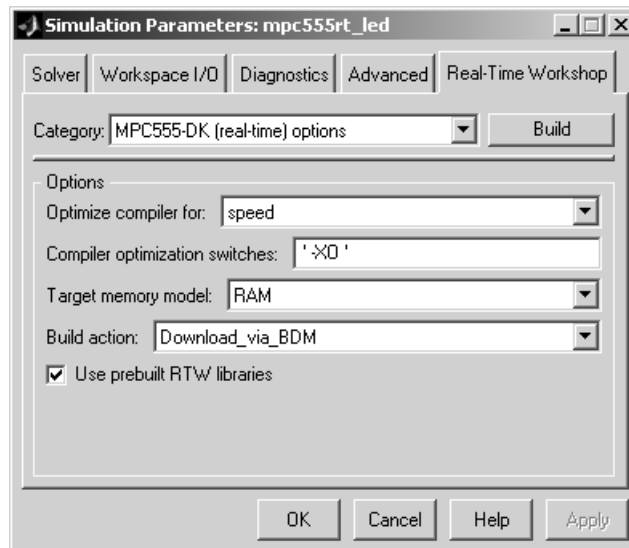
Do the following before you begin:

- Make sure that you have downloaded boot code to the flash memory of the MPC555. See “Downloading Boot Code” on page 3–23.
- Connect the BDM port of your development board to parallel port LPT1 of your host PC (or the port specified for your toolchain if different, see “Setting Up Your Toolchain” on page A-2).
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in “Phytec Jumper Settings” on page A-9.
- Cycle the power (or perform a hard reset) on your development board to clear the RAM.

To generate and download the Target_LED_ram.elf file to RAM over BDM:

- 1 Select **Simulation** -> **Simulation Parameters**.
- 2 On the **Real-Time Workshop** tab, select MPC555-DK (real-time) options from the **Category** drop-down menu.
- 3 Select Run_via_BDM or Debug_via_BDM from the **Build action** drop-down menu.
- 4 Ensure the **Target memory model** selected is RAM (not FLASH).

Notice the default **Optimize compiler for** setting is speed. If you change this setting to debug, the generated elf file will contain both code and symbols for use with a symbolic debugger. See “CompilerOptimizationSwitches Settings” on page 1–13 for more information on these settings. For this tutorial, leave this setting at the default.



- 5** Right click on the Target_LED subsystem and select **Real-Time Workshop** -> **Build Subsystem**.

Note that in the case of this simple example model you could just click **Build** in the **Simulation Parameters** dialog (because there are only Scope blocks outside this subsystem which do not generate code). However for plant/controller models you need to generate code from subsystems only, so you use that method in this tutorial.

You will see progress messages in the MATLAB Command Window as code is generated. Your debugger will be automatically started and will download the code to the target.

Downloading Boot and Application Code

RAM vs. Flash Memory

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

The format of the code and symbol files is the same for both RAM and flash memory targets, suitable for downloading into RAM or on-chip flash memory. The naming convention for these files is

- *model_flash.s19* (for CAN download)
- *model_flash.elf* (for BDM download, can contain debugging symbols).

You can download code to RAM or flash memory via CAN download, 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. The boot code is always required even for RAM applications.

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 3-21. 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 3-23.

Overview of Memory Organization and the Boot Process

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.

The boot code manages the exception handling for the MPC555. RAM applications don't directly handle exceptions but receive them from the boot code. If the boot code is not installed, then RAM applications will not work correctly.

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.

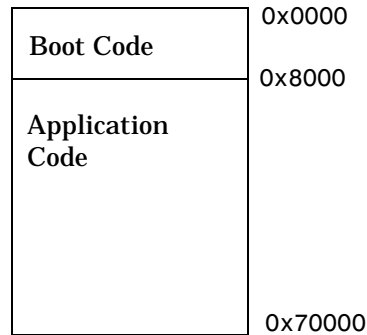


Figure 3-3: Organization of Flash Memory

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 3-26 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 block in your Simulink model. This block is the

CAN Calibration Protocol block. 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.s19` (for CAN download) or `bootcode.elf` (for BDM download) located in the directory

```
matlabroot\toolbox\rtw\targets\mpc555dk\drivers\src\applications  
  \bootcode
```

The path *matlabroot* is the location where MATLAB is installed.

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. Note if you are upgrading from a previous release of Embedded Target for Motorola MPC555 you must download the new 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. Supported toolchains are covered in “Toolchains and Hardware” on page A-1:

- 1 You must first follow the instructions to set up your toolchain. Currently the Embedded Target for Motorola MPC555 supports two toolchains, Wind River Diab and Metrowerks CodeWarrior. In “Setting Up Your Toolchain” on page A-2, we describe how to configure both these toolchains for use with the Embedded Target for Motorola MPC555.
- 2 Connect the BDM cable to the target.

3 Do one of the following:

- Select **Start -> Simulink -> Embedded Target for Motorola MPC555 -> Install MPC555 Bootcode.**
- Alternatively, at the command line type

```
mpc555_bdm_bootcode_download
```

The Embedded Target for Motorola MPC555 automatically starts your debugger (SingleStep or Codewarrior) for you. Your debugger then executes a command to install the boot code. Wait till the process stops then exit the debugger. The boot code should now be installed.

Your PhyCORE-MPC555 board is now ready to receive application code via CAN download. Once the boot code is loaded into flash memory, 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 BDM” on page 3-17.

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 (from Version 1.1 of this product or later) 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 once you have installed the upgrade to the boot code supplied with Version 1.1 of this product. CAN boot code download will only replace boot code for version 1.1 or later; previous versions of the boot code are incompatible.

To replace existing boot code,

1 Open the Download Control Panel in one of the following ways:

- Use the MATLAB Start menu. Select **Start -> Simulink -> Embedded Target for Motorola MPC555 -> Launch CAN Download**
- Type `candownload` at the MATLAB command prompt

The **Download Control Panel** window opens.

2 Select Flash boot code from the **Download type** menu.

- 3 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.
- 4 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.
- 5 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 condition, you do *not* have to reset the target processor manually:

There is an application currently running on the target that implements the CAN Calibration Protocol (as described in “Downloading Boot or Application Code via CAN Without Manual CPU Reset” on page 3-32). In other words, the application contains a CAN Calibration Protocol (CCP) kernel:

- If this condition is met, click the **Download** tab, and then click the **Start** button. Then skip to Step 6.
- If the CCP condition is not met, you must download by the default method. Click on the **Download** tab. Then click on the **Start** button, and immediately press the Reset button on your PhyCORE-MPC555 board.

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

- 6 When downloading is complete, the **Stop** button changes back to **Start**. Close the **Download Control Panel** window.

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.s19*: For CAN download. Code only, without symbols, suitable for execution on the target system.
 - *model_ram.elf*: For BDM download. Can also contain symbols if you perform a debug build, 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 3-13.
 - You can also download to RAM via BDM, as described in “Downloading the Application to RAM via BDM” on page 3-17.

Downloading the Application Code to Flash Memory

To download application code to flash memory, you must generate a code file 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.s19* which contains an image of the executable code, in the working directory.

You can download the file to flash memory via CAN, using the Download Control Panel utility with Vector-Informatik hardware, as described in the following section. Note you cannot use BDM to automatically download application code to flash memory. If you want to download the application to flash memory over BDM manually using your own tools, then the file you need to download is the S-Record file **.s19*.

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.

Do the following before you begin:

- Make sure that your Vector-Informatik CAN card and drivers are installed, and are configured properly. See “CAN Hardware and Drivers” on page A-12. Make sure that the desired CAN port on the PC card is connected to the CAN A port on the target hardware.

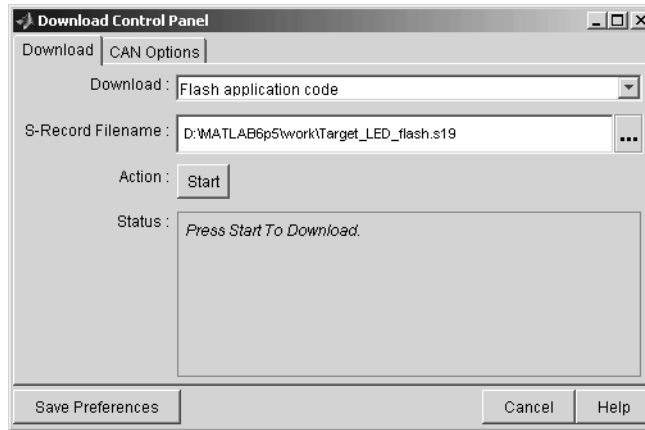
- Make sure that you have set up your toolchain and downloaded boot code to the flash memory of the MPC555, as described in “Downloading Boot Code” on page 3–23.
- Make sure that nothing is connected to the BDM port of your development board.
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in “Phytec Jumper Settings” on page –9.

To download the generated `model_flash.s19` file to flash:

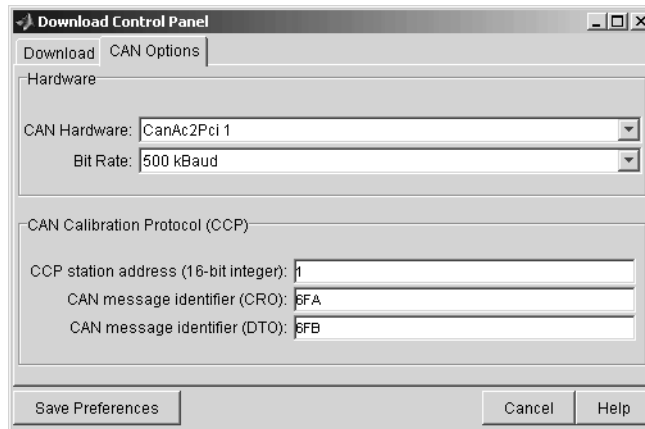
- 1 Open the Download Control Panel in one of the following ways:
 - Use the MATLAB **Start** menu. Select **Start** -> **Simulink** -> **Embedded Target for Motorola MPC555** -> **Launch CAN Download**.
 - Type `candownload` at the MATLAB command prompt.
 - You can also open CAN download automatically at the end of the build process. Before you click **Build** in the **Simulation Parameters** dialog, you can select **Launch CAN Download** from the **Build action** options. You can see an illustration of this dialog in “Generating Code” on page 3–11.

After using any of these three options, the **Download Control Panel** window opens.

- 2 Select Flash application code from the **Download** type menu.
- 3 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. Remember the `.s19` files are for CAN download. The **Download Control Panel** should now appear as shown in this picture.



- 4 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. You can save your preferences by clicking the **Save Preferences** button. This picture shows the **CAN Options** configured for a Vector-Informatik CAN-AC2-PCI card, channel 1.



- 5 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 condition, you do *not* have to reset the target processor:

There is an application currently running on the target that implements the CAN Calibration Protocol (as described in “Downloading Boot or Application Code via CAN Without Manual CPU Reset” on page 3-32). In other words, if the application contains a CAN Calibration Protocol (CCP) kernel, you do not need to reset:

- If this condition is met, click the **Download** tab, and then click the **Start** button.
- If the CCP condition is not met, you must download by the default method. Click on the **Download** tab. Then click on the **Start** button, and immediately press the reset button on your PhyCORE-MPC555 board.

If the condition is met, the CCP kernel running on the board allows automatic download to commence. The MATLAB window displays information like the following:

```
Commencing download to FLASH
-----
CCP Download with the following settings
-----
CRO : 1786
DTO : 1787
Baud : 500 kBaud
Bit Timing : presc = 1 sjw = 1 tseg1 = 8 tseg2 = 7 sam = 0
Channel : CanAc2Pci 1 : 3
File : D:\MATLAB\work\Target_LED_flash.s19
Station ID : 1
```

```
Download Type : RAM application code
-----
```

After a few seconds the download proceeds in the same way as the default method (though in this case there is no need for you to reset your board) and the same successful download messages should appear as in Step 6.

Without CCP implementation, the default download process proceeds as follows:

When you click **Start**, the **Download Control Panel's Status** box changes to read Press reset or power-cycle your development board to start download

When you press the Reset button on your PhyCORE-MPC555 board (or cycle the power), the **Download Control Panel** changes its **Status** box to read CCP Connection OK. Please wait till completion or press Stop to terminate the download.

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

- 6 While downloading proceeds, progress messages are displayed in the **Download Control Panel** and the MATLAB Command Window. A successful download ends with command window messages about the CCP download settings, as shown in the following example:

```
Loading S-Record D:\MATLAB\work\Target_LED_flash.s19
-----
CCP Download with the following settings
-----
CRO : 1786
DTO : 1787
Baud : 500 kBaud
Bit Timing : presc = 1 sjw = 1 tseg1 = 8 tseg2 = 7 sam = 0
Channel : CanAc2Pci 1 : 3
File : D:\MATLAB\work\Target_LED_flash.s19
Station ID : 1
Download Type : Flash application code
```

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

- 7** If the download does not succeed, reset the board and return to step 5.

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 bit rate of 500000 (500kbaud):

```
btest32 500000
```

- 8** 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 6-14).

Note To use the CAN Calibration Protocol block you need Stateflow 5.0(Release 13) and Stateflow Coder.

When the currently running application includes the CAN Calibration Protocol block, the download process begins immediately when you click on the **Start** 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 for signal monitoring, parameter tuning and automatic download, see the demo model `mpc555rt_ccp`. There are instructions for this demo in “MPC555 CAN Communication Protocol Demo” on page 2-6.

Boot Code Parameters for CAN Download

The boot code parameters for download over CAN determine

- CAN bit 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.

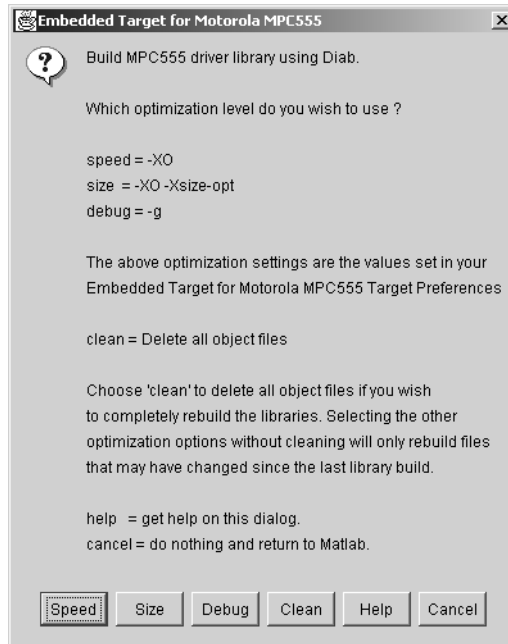
Table 3-1: Default Boot Code Parameters

Parameter	Default Value
CAN bit 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 the default boot code parameter 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 Select **Start -> Simulink -> Embedded Target for Motorola MPC555 -> Build MPC555 Driver Library**.

The **Build Driver Libraries** dialog opens.



- 2 Select the compiler optimization setting you want to use for the build (from speed, size, debug, or clean).
 - See “CompilerOptimizationSwitches Settings” on page 1–13 for more information on the speed, size and debug settings, which are compiler-specific. You can edit these settings in the **Target Preferences** dialog.
 - The clean option deletes all object files. Note that to ensure a rebuild of all files you should run a clean build followed by a build using your required optimization setting. Otherwise only files which have changed since last library build will be rebuilt.

Embedded Target for Motorola MPC555 automatically recompiles the code, using your settings in target preferences.

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.

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

How the Process Works

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 both supported toolchains (Diab and CodeWarrior).

To use the Embedded Target for Motorola MPC555 ASAP2 file generation feature, you simply need to select the **Generate ASAP2** file option in Real-Time Workshop. If it is appropriate to back propagate addresses from the MAP file into the ASAP2 file, then this will also be done automatically. No other steps are necessary to 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.

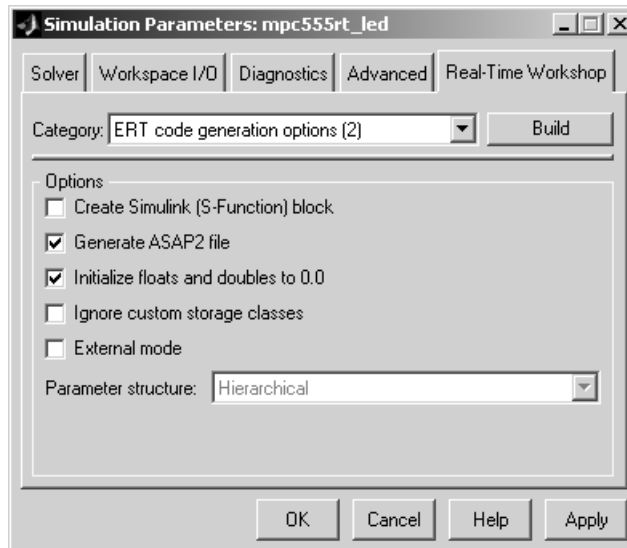
ASAP2 File Generation Procedure

- 1 Create the desired model. Use appropriate parameter names and signal labels to refer to ASAP2 CHARACTERISTICS and MEASUREMENTS respectively.
- 2 Define the corresponding ASAP2.Parameter and ASAP2.Signal objects in the MATLAB workspace.
- 3 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 for each object:
 - ExportedGlobal
 - ImportedExtern
 - ImportedExternPointerExportedGlobal is the default storage class.
- 4 Configure the other data object properties such as LongID_ASAP2, PhysicalMin_ASAP2, etc., for each object.
- 5 In your model window, select the menu item **Simulation -> Simulation Parameters**.
- 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, as shown below.



- 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 6-14 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 6-19) 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

The following sections summarize the features of the Real-Time Target:

- “Code Generation Options” on page 3–40
- “Requirements and Restrictions” on page 3–43

See “MPC555 Fuelsys Demo with Pass Through Drivers” on page 2–10 for an example of a single model deployment paradigm. The pass through device driver blocks provided with the Embedded Target for Motorola MPC555 allow you to use the same model for PIL cosimulation and real-time deployment.

Code Generation Options

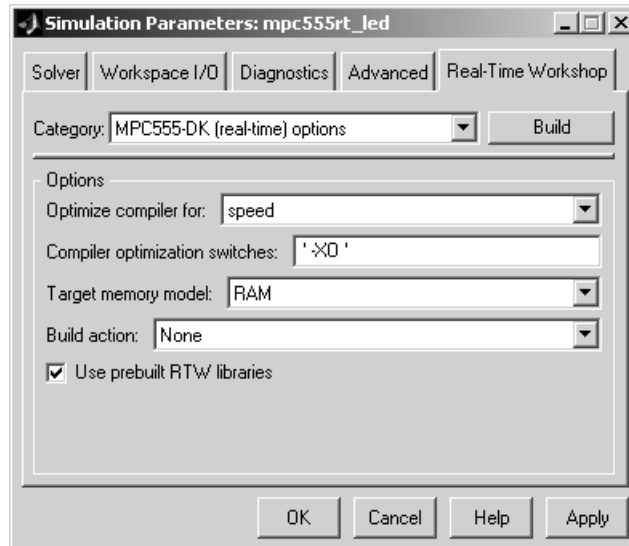
The Real-Time Target is an extension of the Real-Time Workshop Embedded Coder embedded real-time (ERT) target configuration. The real-time target inherits the code generation options of the ERT target, as well as the general code generation options of the Real-Time Workshop. These options are available 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. See “Requirements and Restrictions” on page 3-43 for details.

Note Do not attempt to build code in directories with spaces in the name. This may cause the build to fail as we cannot guarantee that third party toolchains will accept this.

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 parameters for **Options** are

- **Optimize compiler for** — Select speed, size, debug, or custom.

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 by choosing the speed, size, or debug options, or define your own (the custom option). You can edit these preferences here in the **Compiler optimization switches** edit box if you want to apply changes to the current model (**Optimize compiler for:** will change to custom). You can also edit the defaults for these settings in the **Target Preferences** dialog if you want to apply these changes to several models. See “CompilerOptimizationSwitches Settings” on page 1-13 for more information.

- **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. If you select the RAM option, files in a format suitable for downloading into RAM are generated.

In both cases these two files are generated, with this naming convention:

- *model_flash.s19* — code only, for CAN download
- *model_flash.elf* — for BDM download, containing code and optional debugging symbols if you choose a debug build in the **Optimize compiler for settings**

- **Build action**

- None— code generation only.
- Launch_CAN_Download—on completion of code generation the Download Control Panel utility is opened.
- Run_Via_BDM—on completion of code generation download over BDM connection automatically starts and on completion the code is run.
- Debug_Via_BDM—on completion of code generation download over BDM connection automatically starts. When download is complete the code stops at the first line in debug mode, so you can step through the code.

- **Use prebuilt RTW libraries**

This check box option (selected by default) saves a considerable amount of time during the build process, as the libraries do not need to be recompiled every time. However, note this uses the defaults we have chosen for compiler optimization switches. These defaults are designed for rapid prototyping mode. If you are going to switch to production code development and want to fine tune the settings, you should clear this option. Then the custom optimization switches you set in the **Real-Time Workshop Simulation Parameters** dialog box will be applied to the library code as well as the model code.

Note If you build a model with device driver blocks in pass-through mode unnecessary extra code will be generated unless **Inline Parameters** is switched on. This check box can be found in the **Simulation Parameters** dialog box on the **Advanced** tab.

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 6-49 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
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, <code>mpc555dk_main.c</code> .
Generate reusable code	Error if selected; build process terminates

PIL Cosimulation

This section includes the following topics:

- | | |
|---|--|
| Overview of PIL Cosimulation (p. 4-2) | Basic concepts you will need to know to use cosimulation effectively in your design process. |
| Tutorial 1: Building and Running a PIL Cosimulation (p. 4-5) | 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. 4-17) | 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. 4-21) | 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. 4-22) | Summary of code generation options of the PIL target; restrictions and limitations of the PIL target. |

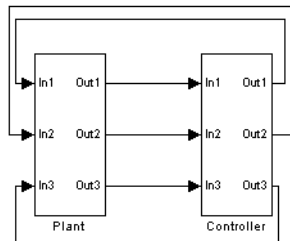
Overview of PIL Cosimulation

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.

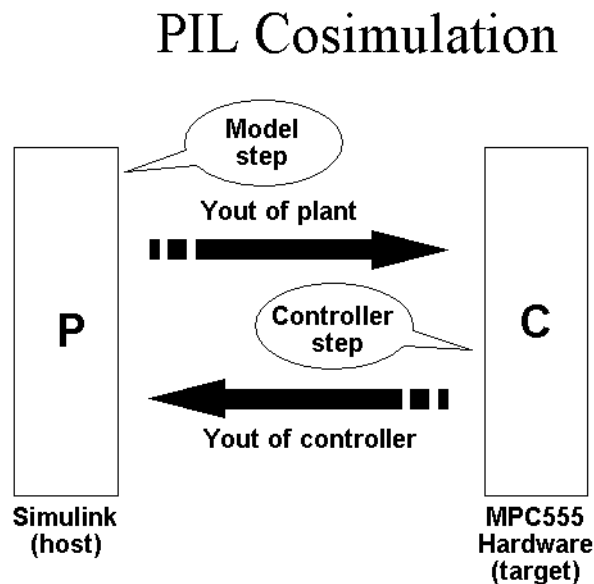
However, these simulation techniques do not help you to account for restrictions and requirements imposed by the hardware. When you finally reach the stage of deploying controller code on the target hardware, you may need to make extensive adjustments to the controller system. Once these

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 (Y_{out} 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 (Y_{out} 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 through hands-on experience, see “Tutorial 1: Building and Running a PIL Cosimulation” on page 4-5.

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 1-10. 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 1-11

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 via BDM. 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 4-1, consists of a plant model with a controller subsystem, the fuel rate controller subsystem.

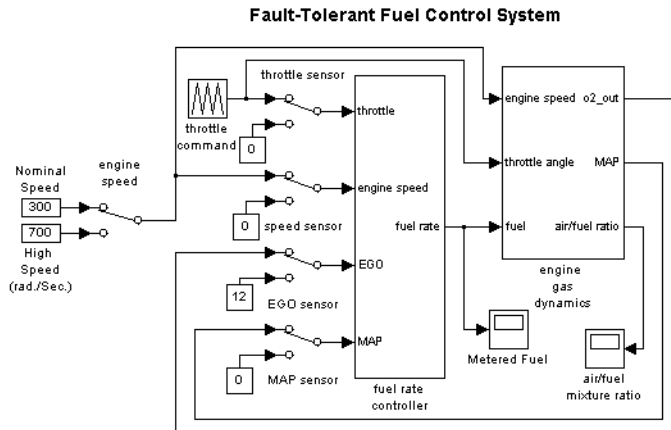
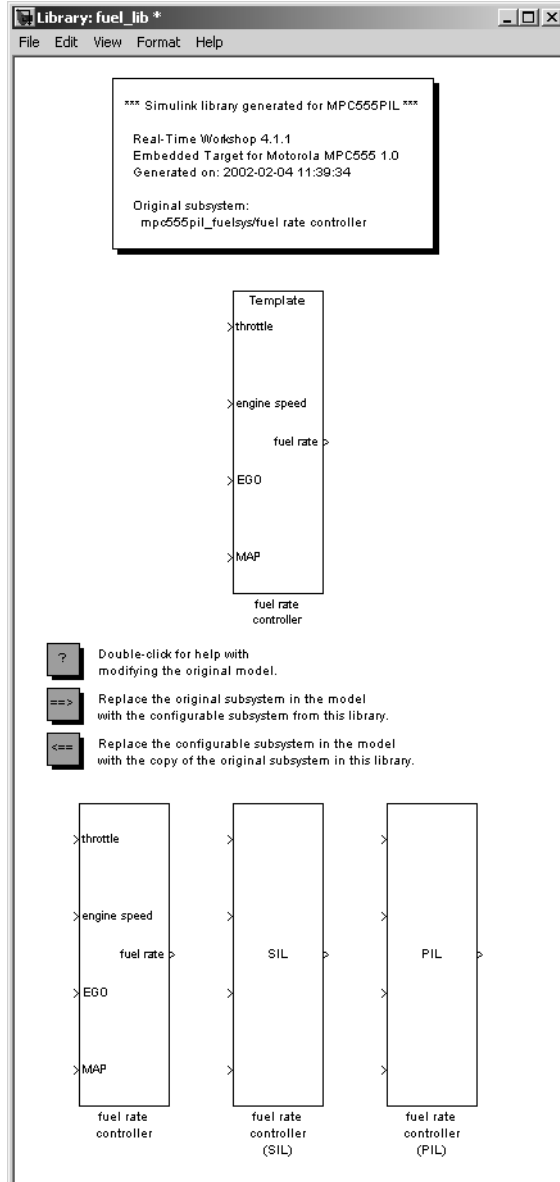


Figure 4-1: Fault-Tolerant Fuel Control System Model

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`.

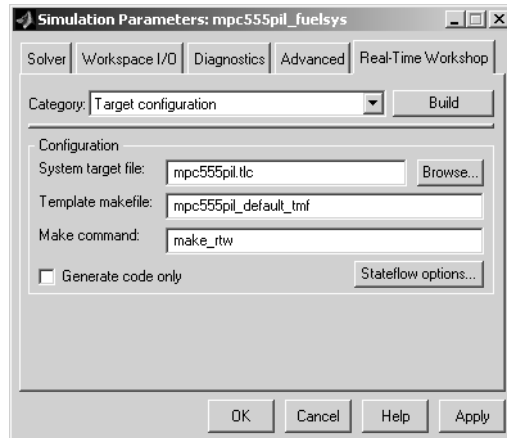
The model is located in the directory `matlabroot/toolbox/rtw/targets/mpc555dk/mpc555demos`. Open `mpc555pil_fuelsys.mdl` and save a copy of the model to your working directory.

The path `matlabroot` should be the location where MATLAB is installed.

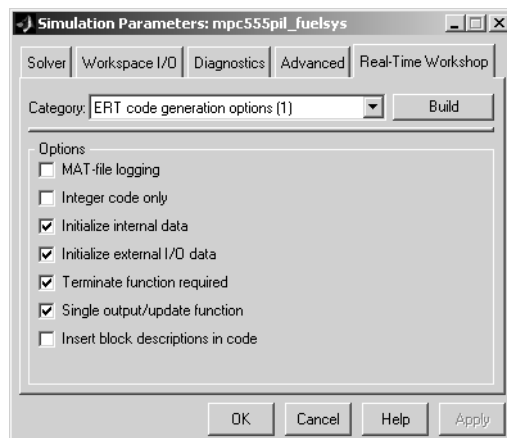
Next, check that the model is correctly configured for use with the Embedded Target for Motorola MPC555.

- 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.

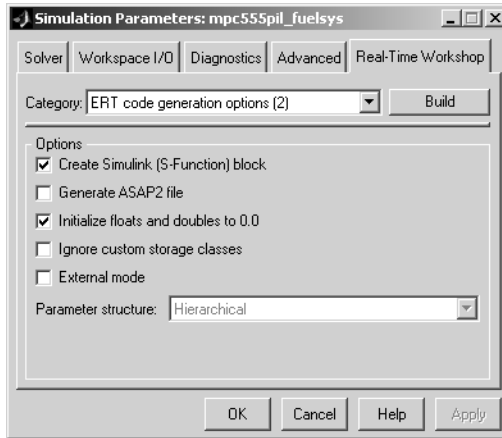
- 3 Select Target configuration from the **Category** menu of the Real-Time Workshop pane. The target configuration should be as shown in this figure.



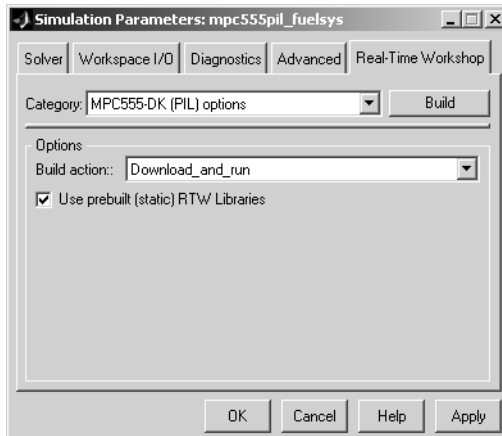
- 4 If the target configuration settings are not as shown, click the **Browse** button to open the System Target File Browser, and select the **Embedded Target for Motorola MPC555 (processor-in-the-loop)** target. Then click **OK** to close the Browser and return to the Real-Time Workshop pane.
- 5 Select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

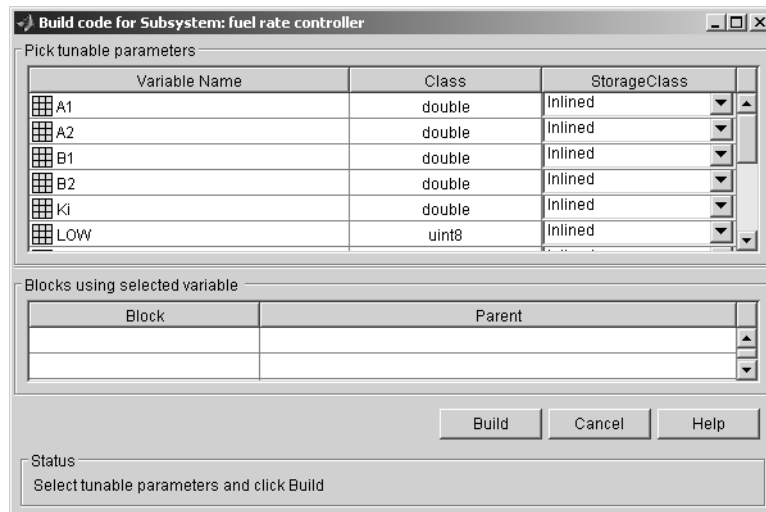


- 8 Select `Download_and_run` from the **Build action** list. This option automatically invokes the appropriate downloading/debugging utility for your development environment, as specified in your target preferences.
- 9 Click **Apply** if you have changed any parameters. Then click **OK** to close the **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 block 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 cosimulation:

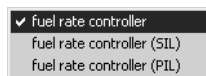
- 1 Click on the fuel_lib library window to activate it. Double-click on the button labeled **Replace the original subsystem in the model with the configurable subsystem from this library**.
- 2 The mpc555pil_fuelsys model window is now the active window. The original fuel rate controller subsystem has 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_fuel_sys`, 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.

- 9 When the simulation completes, the signals displayed on the scopes should appear as shown in Figure 4-2.

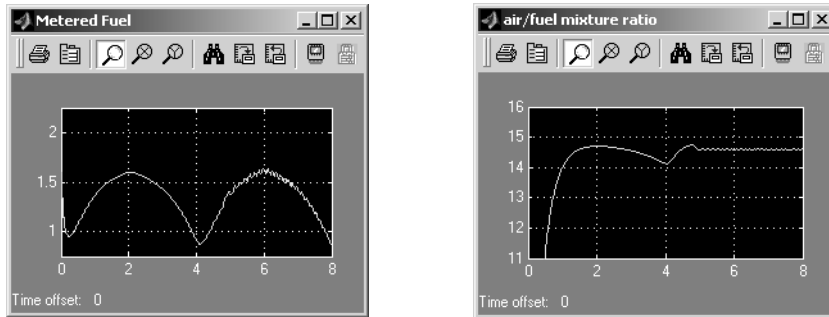


Figure 4-2: Signals Displayed at End of Simulation or Cosimulation

- 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.

You can reload and run the target code created by the PIL target for your cross-development system, and run another cosimulation by typing the following at the command line:

```
run(mpc555_tgtaction, md1 , gcb);
```

This command will redownload and run the executable associated with the last subsystem you clicked on.

See “Build Process Files and Directories” on page 4-24 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 4-5.

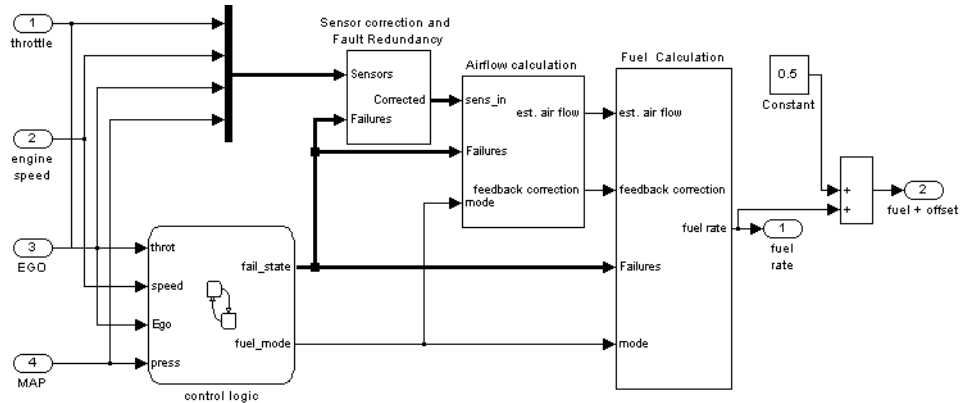
In this tutorial, we will make a simple change to the original `fuel_rate_controller` subsystem in our generated library, `fuel_lib`. We will then rebuild the library components, and run another cosimulation, observing the behavior of our modified controller. All of these steps will be accomplished within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

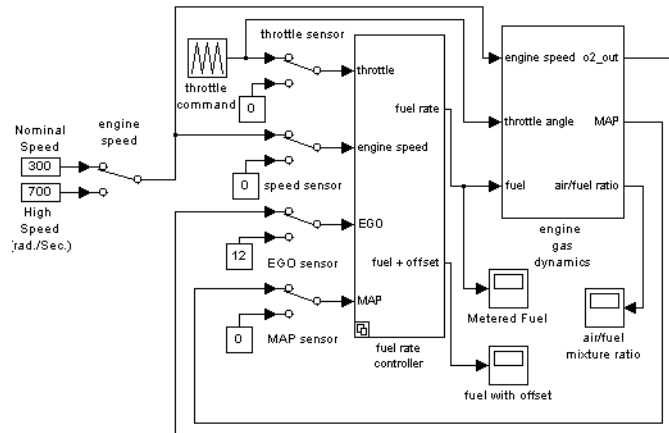
- 1 Activate the `fuel_lib` library, and double-click on the original `fuel_rate_controller` subsystem.
- 2 Add a Sum block, a Constant block, and an output to the `fuel_rate_controller` subsystem. Configure them such that an offset of 0.5 is summed with the fuel rate signal.
- 3 Route the Sum block output to the new output, and label the output `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 outputs.
- 6 Add a Scope block to the model and connect it to the new fuel + offset output of the configurable subsystem. Label the scope fuel with offset. The model should now resemble this block diagram.

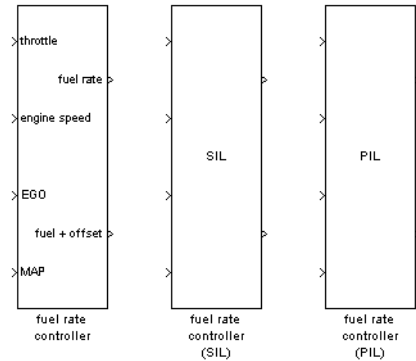


Rebuilding the Controller and Cosimulating

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 4-11 if necessary). At the end of the build process, the `fuel_1.lib` library is again activated. Observe that the rebuilt SIL and PIL components now have two output ports, 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 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 4-2 on page 4-16).

SIL Simulation

- 1 Right-click on the configurable subsystem and select fuel rate controller (SIL) from the **Block choice** submenu of the 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 4-2 on page 4-16).

PIL Target Summary

The following sections summarize the features of the PIL target:

- “Code Generation Options” on page 4-22
- “Build Process Files and Directories” on page 4-24
- “Restrictions” on page 4-25

Code Generation Options

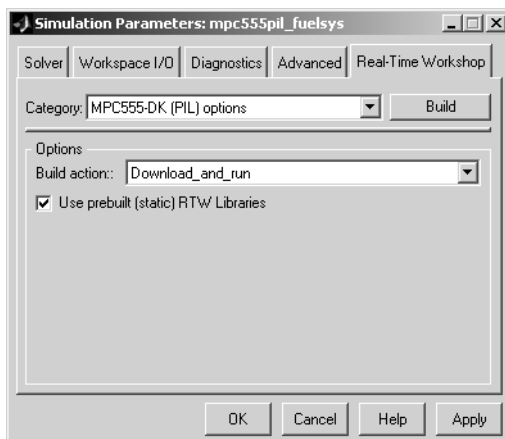
The PIL target is an extension of the Real-Time Workshop Embedded Coder embedded real-time (ERT) target configuration. The PIL target inherits the code generation options of the ERT target, as well as the general code generation options of the Real-Time Workshop. These options are available via the **Category** 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 4-25 for details.

Note Do not attempt to build code in directories with spaces in the name. This may cause the build to fail as we cannot guarantee that third-party toolchains will accept this.

Target-Specific Options

The PIL target has two target-specific code generation options: **Build action** and **Use prebuilt (static) RTW libraries**. To view or change the setting of these options, 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 target preferences (Compiler and Debugger paths) 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.
- **Use prebuilt (static) RTW libraries**
This check box option (selected by default) saves a considerable amount of time during the build process, as the libraries do not need to be recompiled every time.

Manual Download

Once a subsystem has been built using the PIL target, it is possible to manually download the generated code to the target without repeating the entire build process. To do this, use the following procedure:

1 Click on the subsystem that has just been built.

2 From the command line type

```
run(mpc555_tgtaction, mdl , gcb);
```

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 5-3).

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*... *fueln_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 can manually download and run the generated executable using your development system. To do this, use the following procedure:

1 Click on the subsystem that has just been built.

2 From the command line type

```
run(mpc555_tgtaction, mdl , gcb);
```

Restrictions

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

- The PIL target does not support code generation from device driver blocks from the Embedded Target for Motorola MPC555 block libraries. If your model includes any such blocks, they will not cause the build to fail, but they will not function. Device driver blocks do not behave in the way they do in real-time mode. You can use them in pass-through mode during simulation or cosimulation only. If pass-through mode is disabled on the blocks, then no code will be generated for them. If you enable pass-through mode on these blocks, code will be generated for the pass-through inputs/outputs. In PIL mode the state of the input/output pins will not be driven by the model; it is purely for simulation that you can run on the target hardware for cosimulation.
- If you build a model with device driver blocks in pass-through mode, unnecessary extra code will be generated unless **Inline Parameters** is selected. This check box can be found in the **Simulation Parameters** dialog box on the **Advanced** tab.
- 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, the 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.
- If you change the cross-compiler you use with the PIL target (from Diab to CodeWarrior or vice versa), you should rebuild your PIL models in a clean directory, or delete all files from the models' code generation directories. The PIL build process expects to start with a clean directory, or a directory created in the process of building with the same compiler. Leftover components built by a different compiler cause errors.
- 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 4-1 summarizes these restricted options.

Table 4-1: 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, <code>mpc555dk_main.c</code> .
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).

Algorithm Export and Code Analysis Reporting

This section discusses useful tools for code analysis:

Algorithm Export Target (p. 5-2)	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. 5-3)	This section describes the extended HTML code generation report.
Algorithm Export Target Summary (p. 5-5)	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 1-11).

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. Click **OK** to close the browser and return to the Real-Time Workshop pane.
- 3 Select ERT code generation options (3) from the **Category** menu and make sure **Generate an example main program** is selected.
- 4 Follow the usual procedure for generating code from your model or subsystem.

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 5-3). If you select the **Generate HTML report** option as described in the next section, 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.

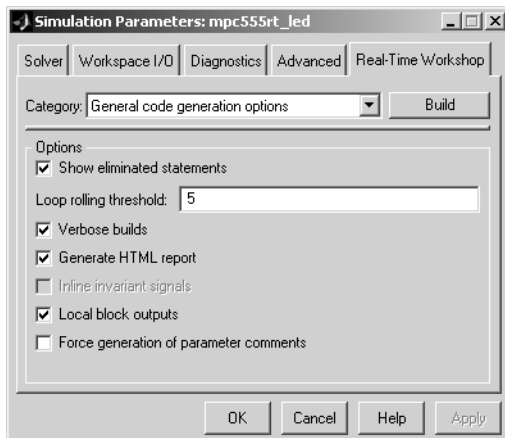
To generate a code generation report and view the profiling report,

- 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

The following sections summarize the features of the Algorithm Export (AE) target:

- “Code Generation Options” on page 5-5
- “Restrictions” on page 5-5

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.

Note Do not attempt to build code in directories with spaces in the name. This may cause the build to fail as we cannot guarantee that third party toolchains will accept this.

The only target-specific option for AE target is **Use prebuilt (static) RTW libraries**. This check box option (selected by default) saves a considerable amount of time during the build process, as the libraries do not need to be recompiled every time.

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 5-1 summarizes these restricted options.

Table 5-1: 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
External mode	Error if selected; build process terminates

Do not include driver blocks in your model for Algorithm Export. The AE target is designed to generate 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, so you should not be including driver blocks.

Block Reference

This section contains the following topics:

The Embedded Target for Motorola
MPC555 Block Libraries (p. 6-2)

Overview of the block libraries provided by the Embedded
Target for Motorola MPC555.

Blocks Organized by Libraries (p. 6-4)

Block summaries and links to the block reference
documentation, grouped by block library.

Alphabetical List of Blocks (p. 6-13)

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.md1`) 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 or the Time Processor Unit 3 (TPU) modules
 - Analog input via the Queued Analog-to-Digital Converter (QADC64)
 - Digital input and output via the MIOS or TPU
 - Digital input via the QADC
 - Frequency and pulse width measurement via the MIOS Double Action Submodule (MDASM)
 - Driver blocks to support other functions of the TPU modules – Fast Quadrature Decode, New Input Capture/Input Transition Counter, and Programmable Time Accumulator
 - Serial transmit and receive
 - Transmission or reception of Controller Area Network (CAN) messages via the MPC555 TouCAN modules
- The CAN Message Blocks library (`canblks.md1`) 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_candivers.md1`) 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 box 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.

MPC555 Driver 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 6-49 for further information.

MPC555 Demos library - see “Embedded Target for Motorola MPC555 Demos” on page 2-2 for links and descriptions. Individual block reference pages also include links to relevant demos.

Modular Input/Output System (MIO51) Sublibrary

Block Name	Purpose
MIOS Digital In	Input driver for MIOS 16-bit Parallel Port I/O Submodule (MPIO5M).
MIOS Digital Out	Output driver for MIOS 16-bit Parallel Port I/O Submodule (MPIO5M).
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.

Time Processor Unit (TPU3) Sublibrary

Block Name	Purpose
TPU3 Digital In	Input driver for TPU3 channel.
TPU3 Digital Out	Output driver for TPU3 channel.
TPU3 Fast Quadrature Decode	Input driver for a pair of TPU3 channels for Fast Quadrature Decode (FQD)

Time Processor Unit (TPU3) Sublibrary (Continued)

Block Name	Purpose
TPU3 New Input Capture/Input Transition Counter	Input driver for TPU3 channel New Input Capture/Input Transition Counter (NITC)
TPU3 Programmable Time Accumulator	Input driver for TPU3 channel Programmable Time Accumulator (PTA)
TPU3 Pulse Width Modulation Out	Output driver for TPU3 channel Pulse Width Modulation.

Serial Communications Interface (SCI)

Block Name	Purpose
Serial Transmit	Configure serial output.
Serial Receive	Configures serial input.

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 data type	logic 1 if input > 0, logic 0 if input <= 0	Boolean	0 or 1 only
MIOS Digital Out	Any Simulink supported data type	logic 1 if input > 0, logic 0 if input <= 0	Any Simulink supported data type	0 or 1 only
MIOS Digital Out (MPWMSM)	Any Simulink supported data type	logic 1 if input > 0, logic 0 if input <= 0	Any Simulink supported data type	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 data type	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 (Continued)

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
TPU3 Digital In	Any Simulink supported data type	Logic 1 if input > 0, logic 0 if input <= 0	Boolean	0 or 1 only
TPU3 Digital Out	Any Simulink supported data type	Logic 1 if input > 0, logic 0 if input <= 0	Any Simulink supported data type	0 or 1 only

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

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
TPU3 Fast Quadrature Decode	Pass through input uint16 Fast Mode input Boolean	N/A	uint16	N/A
TPU3 New Input Capture/Input Transition Counter	uint16	N/A	uint16	N/A
TPU3 Programmable Time Accumulator	Time Accumulation uint32 Period Count uint8	N/A	Time Accumulation uint32 Period Count uint8	N/A
TPU3 Pulse Width Modulation Out	Duty cycle input (top if 2 inputs): double or single	0 to 1	Double or single (must be same as input data type)	0 to 1
	Pulse period register input — uint16	Saturated to be in the range 0 to 32768	uint16	$0 \leq x \leq 32768$

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

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
Serial Transmit	Data: uint8 (vector or scalar) Byte number: uint32 (scalar)	N/A	Number of bytes: uint32	0-16 (for SCI1); 0 or 1 (for SCI2)
Serial Receive	Byte number: uint32 Reset: Boolean	N/A 0 or 1	Data: uint8 Actual byte number: uint32 Framing and parity error: Boolean Overrun flag: Boolean	N/A N/A 0 or 1 0 or 1

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.
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

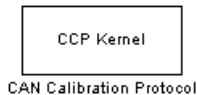
CAN Calibration Protocol	6-14
CAN Calibration Protocol (MPC555)	6-15
CAN Message Filter	6-21
CAN Message Packing	6-23
CAN Message Unpacking	6-25
CAN Message Packing (CANdb)	6-27
CAN Message Unpacking (CANdb)	6-30
MIOS Digital In	6-37
MIOS Digital Out	6-39
MIOS Digital Out (MPWMSM)	6-41
MIOS Pulse Width Modulation Out	6-43
MIOS Waveform Measurement	6-46
MPC555 Resource Configuration	6-49
QADC Analog In	6-65
QADC Digital In	6-70
Serial Transmit	6-73
Serial Receive	6-76
TouCAN Error Count	6-79
TouCAN Fault Confinement State	6-80
TouCAN Interrupt Generator	6-82
TouCAN Receive	6-84
TouCAN Soft Reset	6-87
TouCAN Transmit	6-88
TouCAN Warnings	6-89
TPU3 Digital In	6-90
TPU3 Digital Out	6-92
TPU3 Fast Quadrature Decode	6-94
TPU3 New Input Capture/Input Transition Counter	6-97
TPU3 Programmable Time Accumulator	6-101
TPU3 Pulse Width Modulation Out	6-104
Vector CAN Configuration	6-109
Vector CAN Receive	6-114
Vector CAN Transmit	6-117
Watchdog	6-119

CAN Calibration Protocol

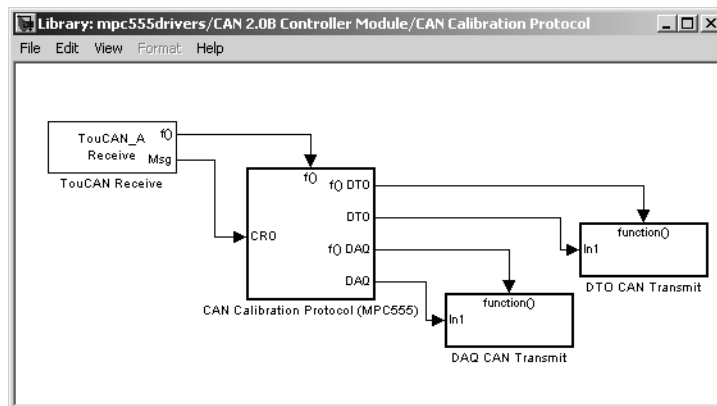
Purpose Provide a configurable template for the CAN Calibration Protocol (MPC555) block

Library Embedded Target for Motorola MPC555

Description The CAN Calibration Protocol block provides a configurable template containing a CAN Calibration Protocol (MPC555) block and CAN Transmit and Receive blocks for I/O. When copied into your model, this block will behave as a standard Simulink subsystem. You can modify the contents and set up the parameters of the blocks inside appropriately for your model.



The CAN Calibration Protocol block inside the template subsystem is set up in a default configuration that you may need to change depending on your requirements. For information about the block parameters, see “CAN Calibration Protocol (MPC555)” on page 6-15. You can see the default template in the figure below.

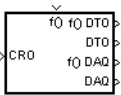


CAN Calibration Protocol (MPC555)

Purpose Implement the CAN Calibration Protocol (CCP) standard

Library This block is embedded in a configurable template, the CAN Calibration Protocol template, within the Embedded Target for Motorola MPC555 block library. See “CAN Calibration Protocol” on page 6-14 for details.

Description



CAN Calibration Protocol (MPC555)

The CAN Calibration Protocol (MPC555) 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 6-19) 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 (DAQ) messages.

Note To use the CAN Calibration Protocol block, you need Stateflow 5.0(Release 13) and Stateflow Coder

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 at <http://www.asam.de>.

You can see an example illustrating how to use the CAN Calibration Protocol (MPC555) block in the `mpc555rt_ccp` demo.

Note this block is entirely CAN triggered, and so is only designed for the Real-Time Target (CAN is disabled during PIL and SIL cosimulation).

Block Inputs and Outputs

The CAN Calibration Protocol (MPC555) 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.

CAN Calibration Protocol (MPC555)

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()` DTO 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 (DAQ) 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 DAQ lists.
- Assign signals in your model to a CCP event channel automatically (see “Generating ASAP2 Files” on page 3-36).

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 6-19) 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:

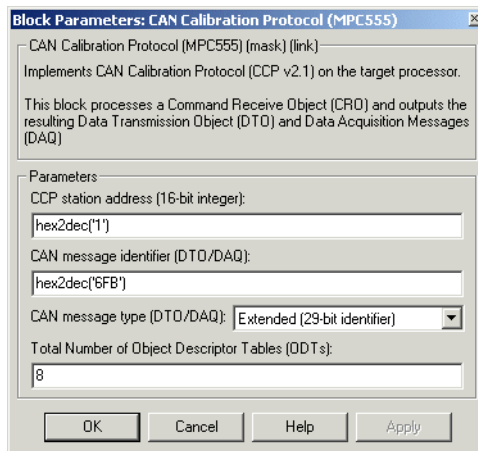
- There is no need for the host to poll for the data. Network traffic is halved.
- The data is transmitted at the correct update rate for the signal. Therefore there is no unnecessary network traffic generated.

CAN Calibration Protocol (MPC555)

- 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” on page 6-19.

CAN message type (DTO/DAQ)

CAN Calibration Protocol (MPC555)

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)

The default number of Object Descriptor Tables (ODTs) is 8. These ODTs are shared equally between all available DAQ lists. You can choose a value between 0 and 254, depending on how many signals you wish to log simultaneously. You must make sure you allocate at least 1 ODT per DAQ list, or your build will fail. The calibration tool will give an error message if there are too few ODTs for the number of signals you specify for monitoring. Be aware that too many ODTs can make the sample time overrun. If you choose more than the maximum number of ODTs (254), the build will fail.

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. To conserve memory on the target the default number is low, allowing DAQ list signal monitoring with reduced memory overhead and processing power.

As an example, if you have five different rates in a model, and you are using three rates for DAQ, then this will create three DAQ lists and you must make sure you have at least three ODTs. ODTs are shared equally among DAQ lists, and therefore you will end up with one ODT per DAQ list. With less than three ODTs you get zero ODTs per DAQ list and the behavior is undefined.

Taking this example further, say you have three DAQ lists with one ODT each, and start trying to monitor signals in a calibration tool. If you try to assign too many signals to a particular DAQ list (that is, signals requiring more space than seven bytes (one ODT) in this case), then the calibration tool will report this as an error.

For more information on DAQ lists, see “Data Acquisition (DAQ) List Configuration” on page 3-38.

Supported CCP Commands

The following CCP commands are supported by the CAN Calibration Protocol (MPC555) 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 (MPC555) 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. This removes the requirement for a manual reset of the processor. 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

CAN Calibration Protocol (MPC555)

memory, or download new boot code to flash memory. See “Downloading Boot or Application Code via CAN Without Manual CPU Reset” on page 3-32.

Note The CAN message identifier of the CCP messages incoming to the target (Command Receive Object (CRO) messages) are set in the mask of the CAN Receive block. The message identifiers for those messages outgoing from the target (Data Transmission Object (DTO) or DAQ) are specified in the block mask for the CAN Calibration Protocol (MPC555) block. These message identifiers 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

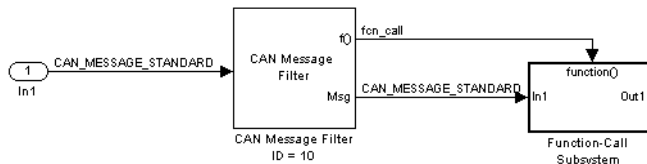
Description



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 Filter

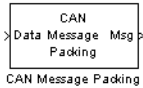
CAN message identifiers to match

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



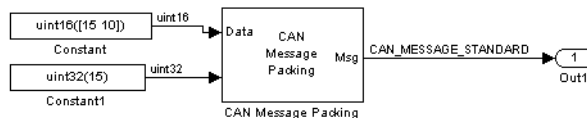
The CAN Message Packing block builds a CAN message. The CAN message type can be either Extended or Standard. You can set the message identifier statically or dynamically. The input port is dynamically typed and will accept any standard Simulink data types as input, as long as the total size does not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

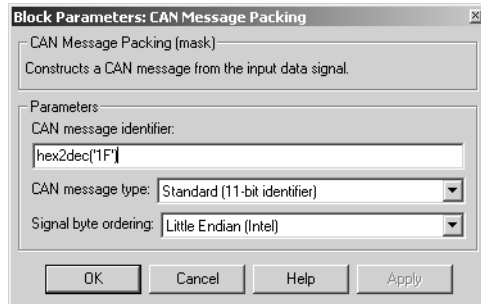
The output data type is either `CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED`.

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).



CAN Message Packing

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.

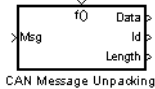
The ordering can be either Little Endian or Big Endian.

CAN Message Unpacking

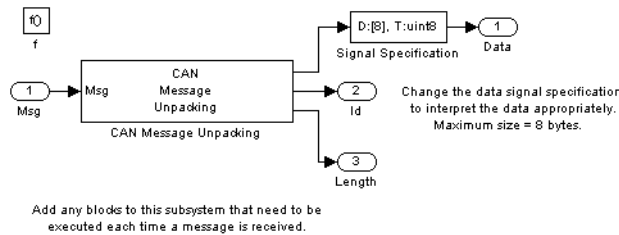
Purpose Inspect and unpack the individual fields in a CAN message

Library CAN Message Blocks

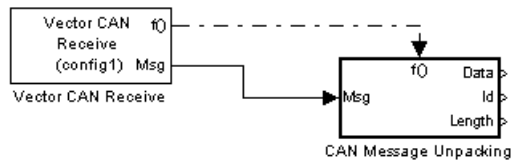
Description The CAN Message Unpacking block receives a CAN message at its input and (by default) outputs the ID, length, and data contained in the message.



Note that the CAN Message Blocks library provides the CAN Message Unpacking block embedded in a Fcn Call subsystem, as shown in this figure.



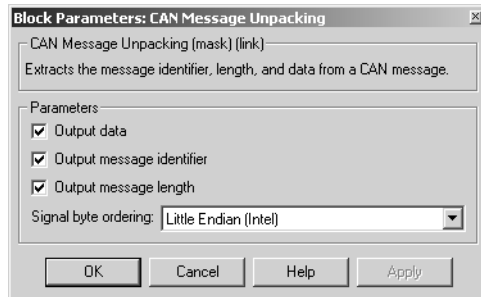
When a message is received, a trigger should be provided to initiate unpacking of the message. The most common way to do this is to connect a CAN message receiving block (such as a Vector CAN Receive block or a TouCAN Receive block) to the CAN Message unpacking template subsystem, as shown in the example below. The function call connection (the dotted line) between these two blocks only calls the CAN Message Unpacking subsystem when a message is received. This saves unnecessary CPU processing time.



CAN Message Unpacking

By default, the data output port outputs a `uint8` signal of width 8. To read the signal as a different data type with a different vector width, modify the Signal Specification block parameters. The help for this block can be found in the Simulink documentation, or right-click on the block itself and select **Help**.

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.

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.

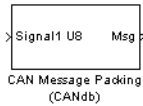
The ordering can be either `Little Endian` or `Big Endian`.

CAN Message Packing (CANdb)

Purpose Pack Simulink signals into CAN messages

Library CAN Message Blocks

Description



The CAN Message Packing (CANdb) block gives you control over the packing of any standard Simulink 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.

You can see an example showing how to use this block in the `mpc555rt_candb` demo.

CAN Message Packing (CANdb)

Dialog Box

Block Parameters: CAN Message Packing (CANdb)

CAN Message Packing (CANdb)
Constructs a CAN message from the input data signals.

The packing of signals within the message can be defined by hand, or by reading in a .DBF file that has been exported from a Vector Informatik CANdb database.

Data Source

Define signals by hand

Use file exported from CANdb .DBF file:

CAN message identifier:

Message

Name: Identifier (hex): Length (bytes):

CAN message type:

Message Signals

List signals for mode:

Port #	Name	Start Bit	Length	Type	Mode	Data Type	Byte Order	Factor	Offset	Units	
1	Signal1	0	8	S	0	U	LE	1	0		<input type="button" value="New"/> <input type="button" value="Edit"/> <input type="button" value="Delete"/>

Signal Editor

Name: Data type:

Start bit: Byte order:

Length (bits): Factor:

Type: Offset:

Mode: Units:

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.

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)

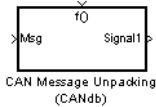
The hexadecimal number that identifies the message on the CAN bus

CAN Message Unpacking (CANdb)

Purpose Decompose a CAN message into its constituent signals

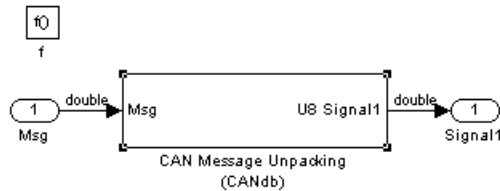
Library CAN Message Blocks

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.

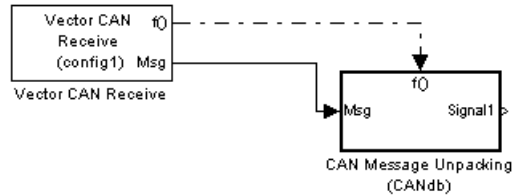
Note that the CAN Message Blocks blocks library provides the CAN Message Unpacking (CANdb) block embedded in a Fcn Call subsystem, as shown in this figure.



Add any blocks to this subsystem that need to be executed each time a message is received.

When a message is received, a trigger should be provided to initiate unpacking of the message. The most common way to do this is to connect a CAN message receiving block (such as a Vector CAN Receive block or a TouCAN Receive block) to the CAN Message Unpacking template subsystem, as shown in the example following. The function call connection (the dotted line) between these two blocks only calls the CAN Message Unpacking subsystem when a message is received. This saves unnecessary CPU processing time.

CAN Message Unpacking (CANdb)



You can see an example showing how to use this block in the `mpc555rt_candb` demo.

You can run the `mpc555rt_candb` model in simulation or generate code from a subsystem to run on MPC555 target hardware.

If you have Vector-Informatik CAN hardware and drivers installed, you can use the companion model `mpc555rt_candbhost` to exchange CAN messages with the `mpc555rt_candb` model (running either in Simulink simulation using virtual CAN channels, or on hardware). You can see the effects of different message packing settings on various signals in these models, for example mode dependant signals. See “Type and Mode” on page 6-35 for more information.

See the instructions in the demo models.

CAN Message Unpacking (CANdb)

Dialog Box

Block Parameters: CAN Message Unpacking (CANdb)

CAN Message Unpacking (CANdb)/CAN Message Unpacking (CANdb)
Constructs a CAN message from the input data signals.

The packing of signals within the message can be defined by hand, or by reading in a .DBF file that has been exported from a Vector Informatik CANdb database.

Data Source

Define signals by hand

Use file exported from CANdb

DBF file:

CAN message identifier:

Message

Name: Identifier (hex): Length (bytes):

CAN message type:

Message Signals

List signals for mode:

Port #	Name	Start Bit	Length	Type	Mode	Data Type	Byte Order	Factor	Offset	Units	
1	Signal1	0	8	S	0	U	LE	1	0		<input type="button" value="New"/> <input type="button" value="Edit"/> <input type="button" value="Delete"/>

Signal Editor

Name: Data type:

Start bit: Byte order:

Length (bits): Factor:

Type: Offset:

Mode: Units:

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.

CAN Message Unpacking (CANdb)

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.

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 hexadecimal 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:

CAN Message Unpacking (CANdb)

- 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 6-35). 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 6-34.

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 signal is actually mapped, you can define at most one of the signals to be the *mode signal*. The mode signal is an integer valued input. Click the required input signal from the list in the **Message Signals** pane, then click **Edit**. You can select Mode signal from the **Type** drop-down menu. Note the value in the **Mode** edit box does not matter for the Mode signal, it is the value of the input signal that drives the packing of the mode dependant signals.

You can then define as many *mode-dependant signals* as you want. Edit the signals in the same way, then choose Mode dependant signal from the **Type** drop-down menu. A mode-dependant signal also makes use of the **Mode** value. Enter the required value for your mode dependant signal in the **Mode** edit box. This value is matched against the value of the defined mode signal input. 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.

If you choose Standard from the **Type** drop-down menu the mode signal has no effect on that signal, and that signal is always packed into the message.

You can see an example showing the effects of all these settings by running the demo models `mpc555rt_candb` and `mpc555rt_candbhost` in simulation.

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.

CAN Message Unpacking (CANdb)

- 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 use *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 (MPIOISM)

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 (MPIOISM) of the MPC555. The **Bits** field specifies a vector of numbers in the range 0..15, corresponding to pins MPI032B0..MPI032B15 on the MPIOISM.

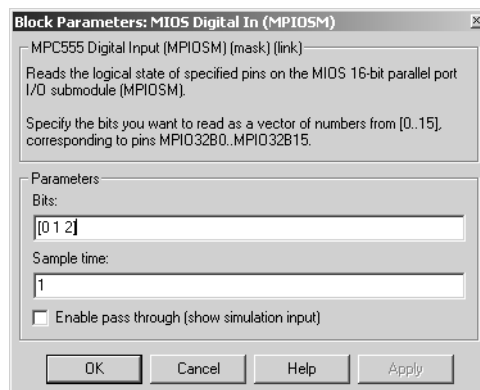


The output of the block is a wide vector representing the logic state of the pins referenced in the **Bits** field. When the signal on a given pin is a logical 1, the block output element will be equal to 1; otherwise the block output element will equal zero.

Refer to section 15.13, “MIOS 16-bit Parallel Port I/O Sub module (MPIOISM),” 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 (MPI032B0 . . MPI032B15) on the MPIO SM.

Sample time

Sample time of the block.

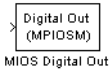
Enable pass through (show simulation input)

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 6-7 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 (MPIOISM)

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 (MPIOISM) of the MPC555. The **Bits** field specifies a vector of numbers in the range 0 . . 15, corresponding to pins MPIO32B0 . . MPIO32B15 on the MPIOISM.



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.

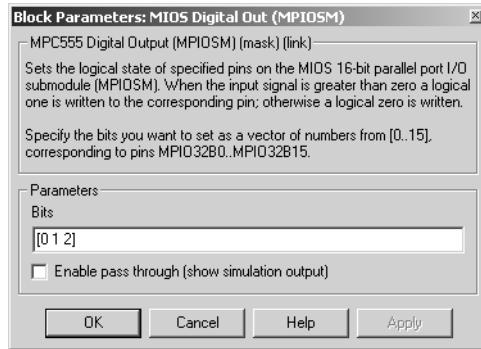
If you want to write to several digital output pins at the same sample rate, using a single MIOS Digital Out block with a vector input signal will result in more efficient code. However, if you want to update different output pins at different sample rates, you must use a separate MIOS Digital Out block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

Refer to section 15.13, “MIOS 16-bit Parallel Port I/O Sub module (MPIOISM),” 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.

MIOS Digital Out

Dialog Box



Bits

A vector of numbers in the range 0 . . 15. Each number corresponds to a pin (MPI032B0 . . MPI032B15) 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 output 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 6-7 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)

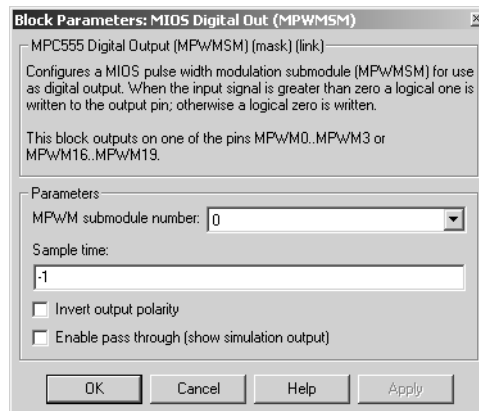
Library Embedded Target for Motorola MPC555

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
Select a PWM submodule for output.

Sample time
Sample time of the block.

Invert output polarity
Switches the output level for logic one and zero.

MIOS Digital Out (MPWMSM)

Enable pass through (show simulation input)

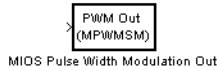
Lets you provide a signal from this block for use in simulation. When this option is enabled, an outport appears on the block. The block input is passed through to the output during simulation. (See “Data Type Support and Scaling for Device Driver Blocks” on page 6-7 for information on supported input/output data types and scaling of input/output signals.) This option affects simulation only.

MIOS Pulse Width Modulation Out

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%.

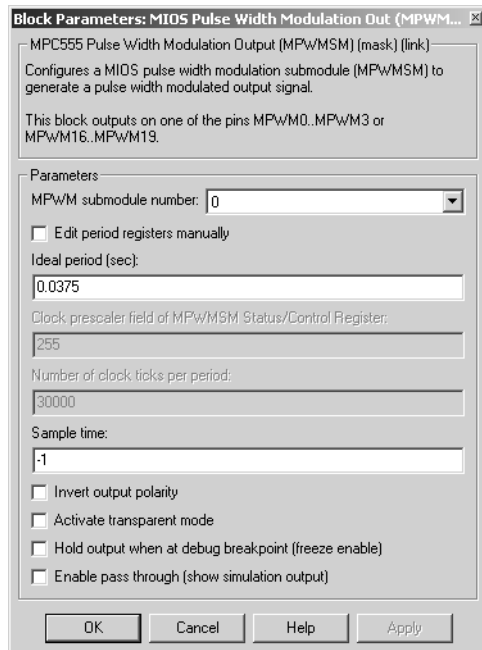
The MIOS Pulse Width Modulation block input signal acts as the modulator, controlling the duty cycle of the signal on the output pin. The input signal is multiplied by the period register value, and saturates if outside 0-1. When the input signal value is 0, the output signal's duty cycle is 0%. When 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.

MIOS Pulse Width Modulation Out

Dialog Box



MPWM submodule number

Select 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 6-49).

Number of clock ticks per period

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

Sample time

Sample time of the block.

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 output 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 6-7 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. The pass-through output is not the PWM waveform, rather it is the value of the PWM duty cycle. 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 Waveform measurement is a feature of the MIOS Double Action Submodule (MDASM) on the MPC555. The MIOS Waveform Measurement block currently implements the following features of the MDASM:



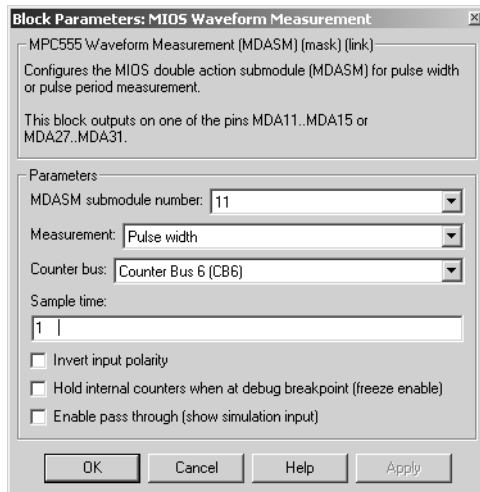
- *Pulse width measurement*: the MIOS Waveform Measurement block outputs the time from the leading edge of a pulse to the trailing edge of the same pulse.
- *Pulse period measurement*: the MIOS Waveform Measurement block outputs the time from the leading edge of a pulse to the next leading edge of a 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 10 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

Select one of the 10 MIOS Double Action Submodules (MDASM) in the MPC555.

Measurement

Select 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 6-58.

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

MIOS Waveform Measurement

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 6-7 for information on supported input/output data types and scaling of input/output signals.) This option affects simulation only.

MPC555 Resource Configuration

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 6-52 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

MPC555 Resource Configuration

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

Types of Configurations

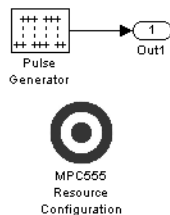
A *configuration* is a collection of parameter values affecting the operation of a group of device driver blocks in one of the Embedded Target for Motorola MPC555 libraries, such as the MIOS1, QADC64 or TouCAN libraries. The MPC555 Resource Configuration object currently supports the following types 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).
- TPU3 Configuration: parameters related to the Time Processor Unit module.
- TouCAN Configuration: parameters related to the CAN 2.0B Controller Module (TouCAN).

Active and Inactive Configurations

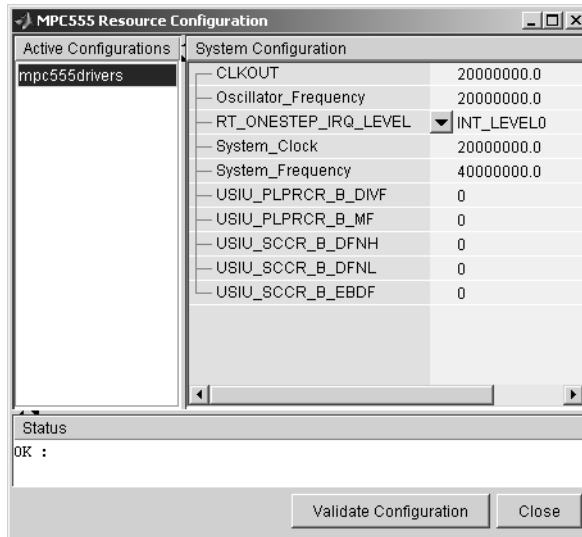
An *active* configuration is a configuration associated with blocks of the model or subsystem in which the MPC555 Resource Configuration object is installed. There is always an active MPC555 configuration. For any other configuration type (e.g., QADC, MIOS, or TouCAN), there is at most one active configuration.

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

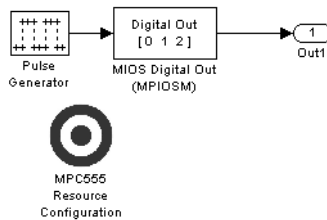


MPC555 Resource Configuration

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

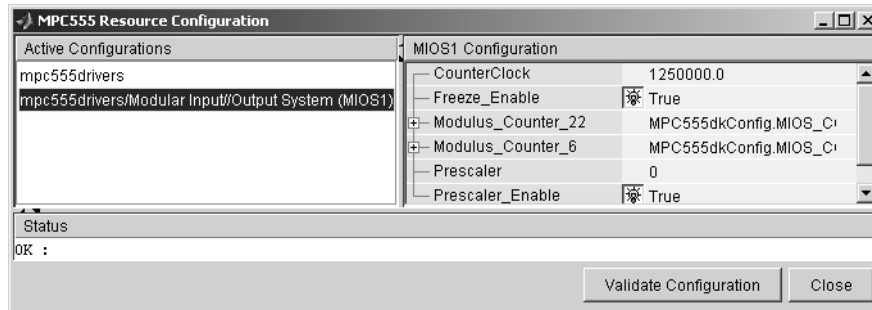


When a device driver block is added to the model, an appropriate configuration is created and activated. This figure shows an MIOS Digital Out block added to the model.



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.

MPC555 Resource Configuration



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.

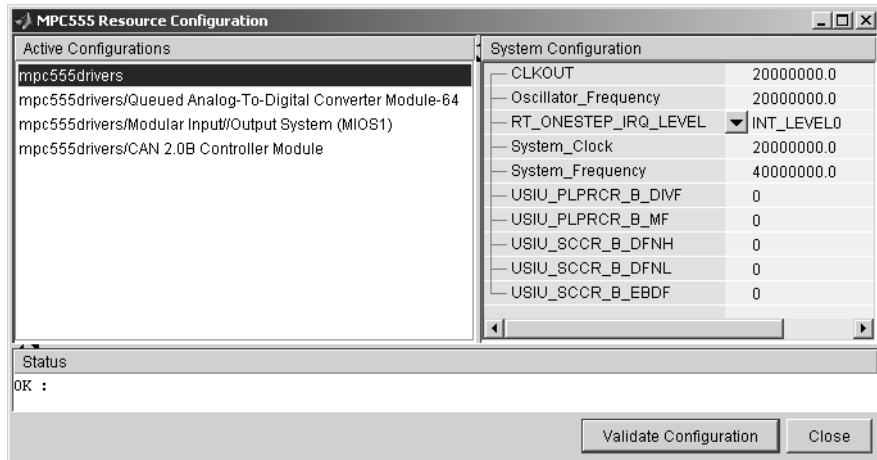
Note When using device driver blocks from the Embedded Target for Motorola MPC555 libraries in conjunction with the MPC555 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the MPC555 Resource Configuration block will operate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

MPC555 Resource Configuration



MPC555 Resource Configuration Window

This figure 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 pop-up 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 pop-up 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 6-54.

MPC555 Resource Configuration

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.

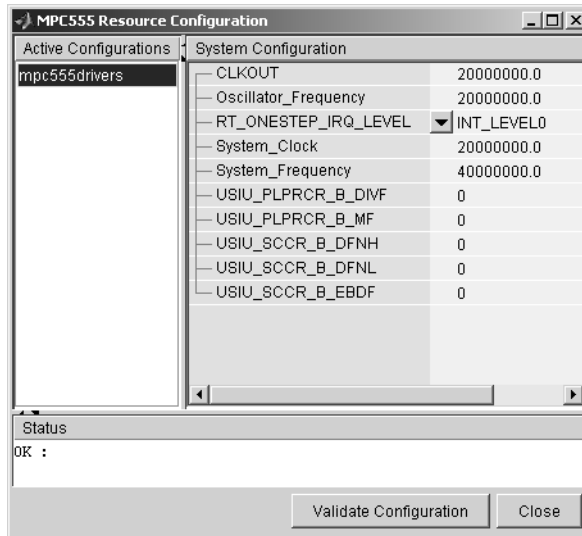
Note that the **Validate Configuration** button does not validate the entire model; it only checks for resource allocation conflicts related to the selected configuration. To detect problems related to the model as a whole, select **Update diagram (Ctrl+D)** from the Simulink Edit menu.

- **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 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..7). Do not select Interrupts Disabled, or the model will not work.

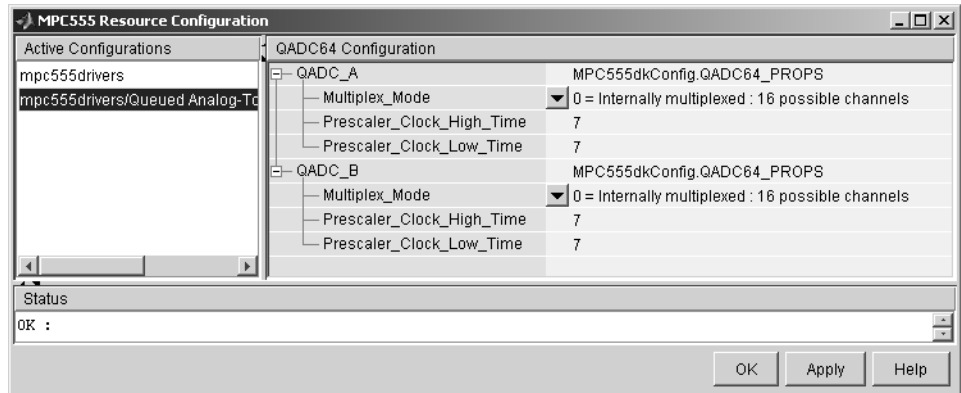
See the “Data Structures and Program Execution” section in the Real-Time Workshop Embedded Coder documentation for a detailed description of the `rt_OneStep` function.

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.

MPC555 Resource Configuration

QADC64 Configuration Parameters



The Queued Analog-To-Digital Converter Module 64 (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

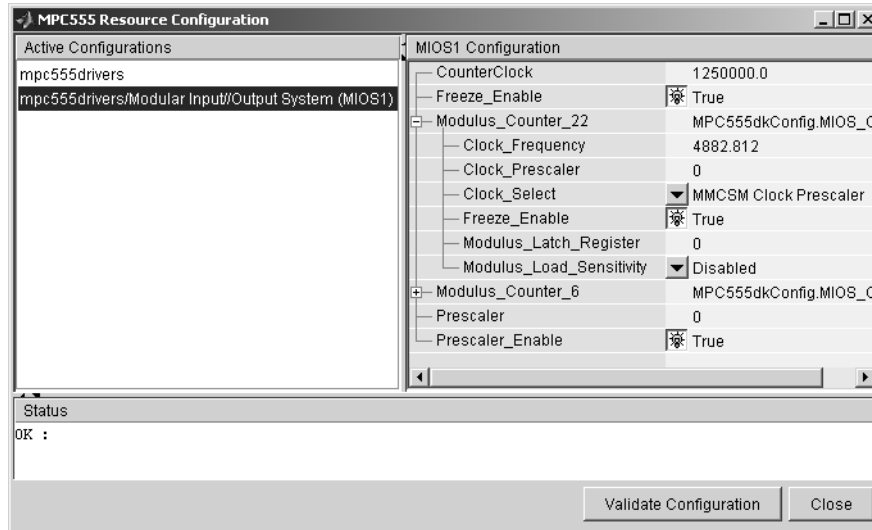
Prescaler clock high (PSH) time. The default is 7. The PSH field selects the QCLK high time in the prescaler. PSH value plus 1 represents the high time in IMB clocks.

Prescaler Clock Low Time

Prescaler clock low (PSL) time. The default is 7. The PSL field selects the QCLK low time in the prescaler. PSL value plus 1 represents the low time in IMB clocks.

MPC555 Resource Configuration

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 Pulse Width Modulation block or MIOS Waveform Measurement block).

Modulus Counter 6 and 22

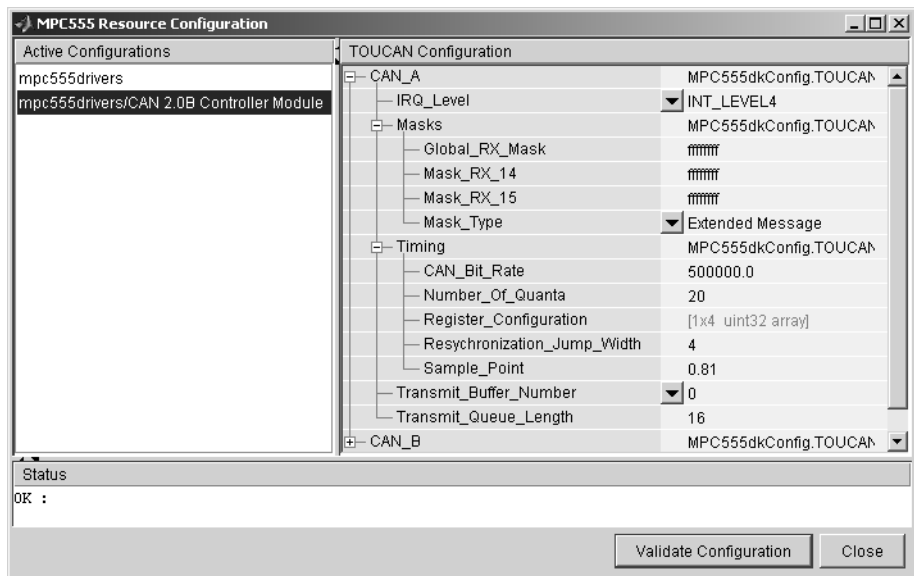
These two counters provide reference clocks to submodules such as the MIOS Pulse Width Modulation Submodule and the MIOS Double Action Submodule (Frequency / Period measurement) subsystems. If you change the **Clock select** to anything other than MMCSM Clock Prescaler, the

MPC555 Resource Configuration

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. Do not disable

MPC555 Resource Configuration

interrupts, this will stop the TouCAN Transmit block from working correctly.

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.

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.

MPC555 Resource Configuration

Transmission Configuration Parameters

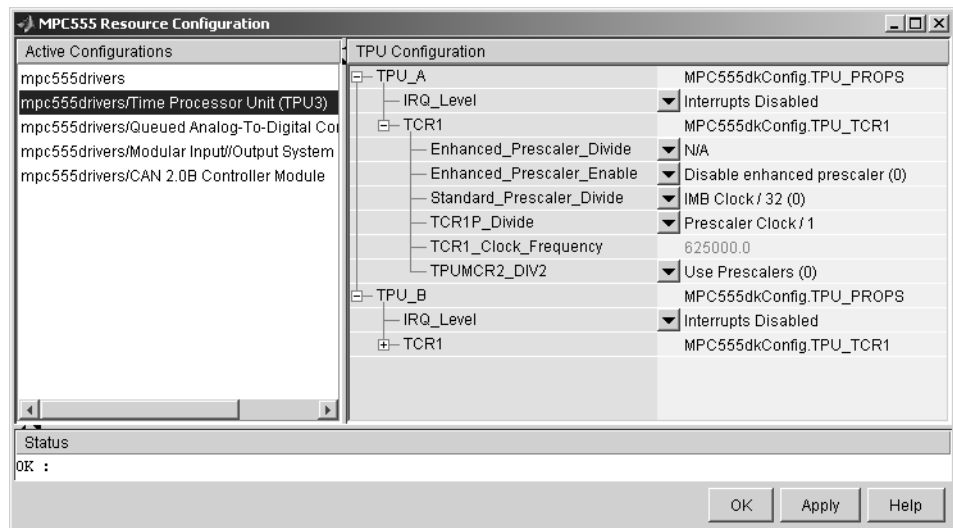
Transmit Buffer Number

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.

Time Processor Unit (TPU3) Configuration Parameters



The TCR1 timebase is configurable for both TPU Channel A and TPU Channel B. The parameters under the TCR1 tree allow you full control to specify the clock speed of the TCR1 timebase. Consult Section 17 of the *MPC555 User's Manual* before editing the TPU configuration parameter defaults. The parameters listed below are the same for TPU modules A and B.

TPUMCR2_DIV2

TPUMCR2_DIV2 (the last setting under the tree) allows you to choose to use a set of prescalers to divide the IMB clock down further (Use Prescalers (0)), or to just divide the IMB clock by two (IMB Clock / 2 (1)). If you choose the divide by two option then none of the other settings

MPC555 Resource Configuration

are applicable and are marked N/A. Note this is the last setting purely because the parameters are laid out in alphabetical order.

Enhanced_Prescaler_Enable

Here you can choose whether you use the Standard Prescaler (set by `Standard_Prescaler_Divide`) or the Enhanced Prescaler (set by `Enhanced_Prescaler_Divide`) to derive the Prescaler Clock.

Standard_Prescaler_Divide

If you choose to use the `Standard_Prescaler_Divide` then you can choose to divide the IMB clock down by either 32 or 4.

Enhanced_Prescaler_Divide

If you choose to use the `Enhanced_Prescaler_Divide`, then you can choose to divide the IMB clock down by either 2, 4, 6, 8, .. , 60, 62, 64.

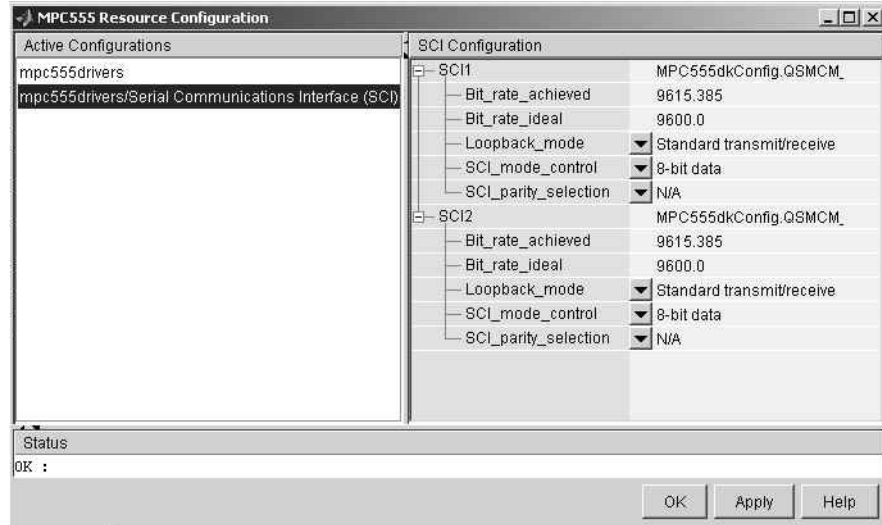
TCR1P_Divide

Whichever type of prescaler you choose (standard or extended), there is a further prescaler that is applied to the clock. `TCR1P_Divide` divides the Prescaler Clock by 1, 2, 4, or 8. The resulting clock is the TCR1 timebase.

IRQ_Level

This enables TPU interrupts. The default is disabled. If your model contains any TPU3 Programmable Time Accumulator blocks, you will need to choose an interrupt level.

Serial Communications Interface (SCI) Configuration Parameters



Bit_rate_achieved

This read-only field shows the achieved serial interface bit rate. In general this value differs slightly from the requested bit rate, but is the closest value that can be achieved by setting allowed values in the MPC555 registers SCC1R0 and SCC2R0 for QSMCM submodules SCI1 and SCI2 respectively.

Bit_rate_ideal

Enter the desired bit rate for serial communications in this field. Appropriate register settings will be calculated automatically. You can check the actual bit rate in the **Bit_rate_achieved** field.

Loopback_mode_enable

Select either Standard transmit/receive or Loopback mode enabled. The loopback mode may be useful for test purposes where the serial interface is required to receive data that it transmitted itself.

SCI_mode_control

Select the desired combination of word length and parity/no parity.

MPC555 Resource Configuration

Parity_selection

If parity is enabled, you must select Odd parity or Even parity.

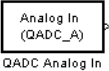
Purpose

Input driver enables use of Queued Analog-Digital Converter (QADC64) in continuous scan mode

Library

Embedded Target for Motorola MPC555

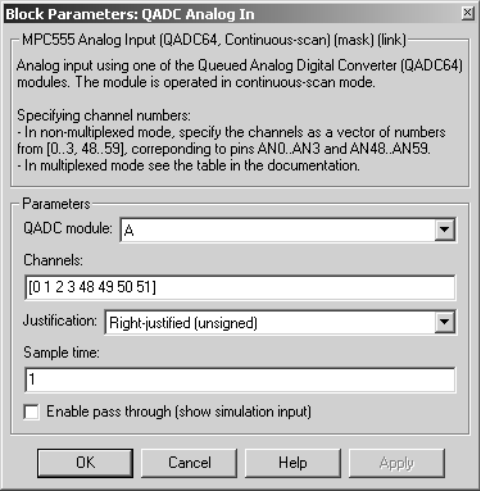
Description



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

Select 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 6-7.

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.

The following two tables 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]

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
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

QADC Analog In

Nonmultiplexed Scan Mode (Continued)

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
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

Multiplexed Scan Mode

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
PQB0	A_AD0 / AN _w	-	I	0-14 even
PQB1	A_AD1 / AN _x	-	I	1-15 odd
PQB2	A_AD2 / AN _y	-	I	16-30 even
PQB3	A_AD3 / AN _z	-	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
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

Multiplexed Scan Mode (Continued)

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
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

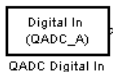
In this table, PQA0, PQA1 and PQA2 are used as output pins to drive an external demultiplexer.

QADC Digital In

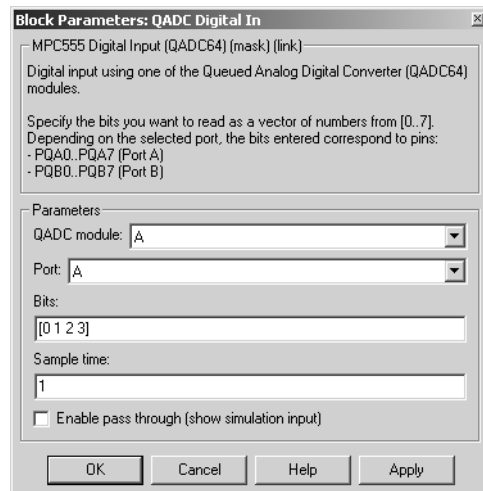
Purpose Input driver enables use of Queued Analog-Digital Converter (QADC64) pins as digital inputs

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

Select module A or B.

Port

Select 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 this table to work out how the block ports and bits map to processor pins on the MPC555.

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

QADC Digital In

Relationship of Port/Bit Parameters to Hardware Pins (Continued)

Port	Bit	Hardware Pin
A	5	A_AD13 / PQA5
A	6	A_AD14 / PQA6
A	7	A_AD15 / PQA7

Purpose Configure MPC555 for serial transmit, using one of the QSMCM submodules SCI1 or SCI2

Library Serial Communications Interface (SCI)

Description



The Serial Transmit block transmits bytes via either of the MPC555 QSMCM submodules SCI1 or SCI2. You can use it either to transmit a fixed number of bytes, or, by enabling the second input, transmit a variable number of bytes each time this block is called. With SCI1, a hardware buffer is used that allows up to 16 bytes to be queued for transmission. With SCI2, the buffer allows only up to one byte to be queued each time the block is called. Once bytes are queued for transmit, they will be sent as fast as possible by the serial interface hardware with no further intervention required by the rest of the application.

If the hardware buffer is not empty when the block is called, i.e., the previous transmission is not yet complete, then no new bytes will be queued for transmit. This condition can be identified from the “actual number of bytes” block output; if no bytes were queued for transmit, this output returns zero.

To configure the serial interface bit rate and data format, see “Serial Communications Interface (SCI) Configuration Parameters” on page 6-63.

The device driver used for the Serial Transmit block does not require the use of CPU interrupts.

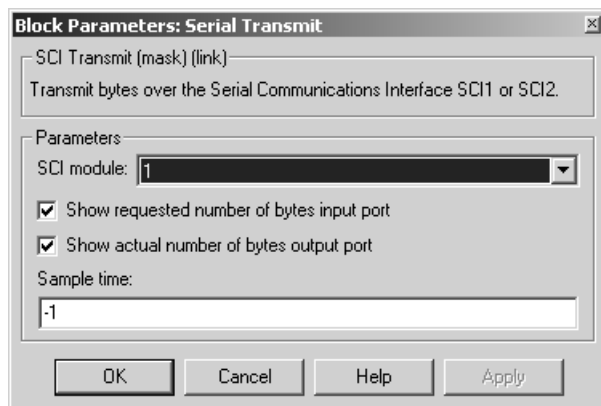
Serial Transmit

Block Inputs and Outputs

The first input contains the data to be transmitted; this input signal may be either a vector or scalar with data type `uint8`. The optional second input must be a scalar and may be used to control the number of bytes transmitted. The number of bytes to transmit should not be greater than the width of the first input signal.

The block output port “actual number of bytes output” gives the number of bytes queued for transmit. If the previous transmission was complete, this number will be equal to the requested number of bytes to transmit, provided that this was less or equal to 16 in the case of SCI1, or 1 in the case of SCI2. See “Data Type Support and Scaling for Device Driver Blocks” on page 6-7 for information on supported input/output data types and scaling of input/output signals.

Dialog Box



SCI module

Select either 1 or 2 (to choose module SCI1 or SCI2).

Show requested number of bytes input port

Enable/disable the input for number of bytes to send. If cleared, the number of bytes sent is just the width of the first inport; if selected, the second input is enabled, which controls the number of bytes to send.

Show number of bytes output port

Enable/disable the output port for number of bytes actually sent. If selected, this value is available from the first output.

Sample time

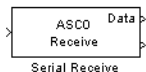
The time interval between samples. To inherit the sample time, leave this parameter at the default -1. See “Specifying Sample Time” in the Simulink documentation for more information.

Serial Receive

Purpose Configure MPC555 for serial receive on either of the QSMCM submodules SCI1 or SCI2.

Library Serial Communications Interface (SCI)

Description



The Serial Receive block receives bytes via either of the MPC555 QSMCM submodules SCI1 or SCI2. It requests either a fixed number of bytes to be received, or, by enabling the first input, a variable number of bytes can be requested each time this block is called. When the block is called, the requested number of bytes are retrieved from a hardware buffer provided by the submodule SCI1 or SCI2. On SCI1, the size of this buffer is 16 bytes. On SCI2, it is one byte.

If the buffer contains fewer bytes than the number requested, these bytes are pulled from the buffer and made available at the block output. The number of bytes actually retrieved from the buffer is made available at the second output. This block will only retrieve bytes that have already been received and placed in the hardware buffer; it will never wait for additional data to be received.

To configure the serial interface bit rate and data format, see “Serial Communications Interface (SCI) Configuration Parameters” on page 6-63.

The device driver used for the Serial Receive block does not require the use of CPU interrupts.

Block Inputs and Outputs

The first input can be enabled so a variable number of bytes can be requested each time.

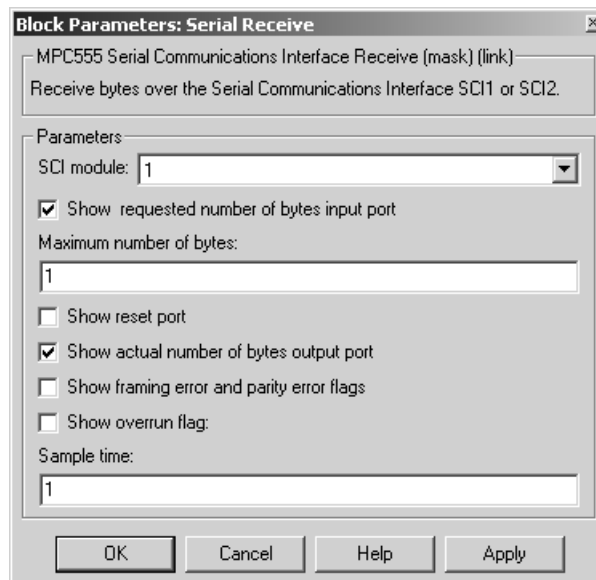
The second input, if enabled, is a reset signal, which must have a Boolean data type. You must reset the SCI1 module if an overrun error or framing or parity error occurs. No reset is required for SCI2.

The first output (marked Data) pulls bytes from the buffer — either the number requested or the number available, whichever is the lower. Note that the number requested is the value of the first input signal if supplied, or the width of output signal otherwise.

The second output (if enabled) is the number of bytes actually retrieved from the buffer. Up to four outputs can be enabled — the third showing framing error and parity error flags, and the fourth showing overrun flags.

See “Data Type Support and Scaling for Device Driver Blocks” on page 6-7 for information on supported input/output data types and scaling of input/output signals.

Dialog Box



Serial Receive

SCI module

Select either 1 or 2 (to choose module SCI1 or SCI2).

Show requested number of bytes input port

Enables an inport (the top one if there are two) where you can input the number of bytes to request.

Maximum number of bytes

Maximum number of bytes to receive (this is only visible if the requested number of bytes input port is enabled). This sets an upper limit on the number of bytes that will be read each time the block is called.

Show reset port

Enables the reset input (the lower inport).

Show actual number of bytes output port

Enables another output that shows the number of bytes actually read from the SCI buffer.

Show framing error and parity error flags

Enables another output. This output is zero if no framing or parity error occurred during the current read; it is true (1) otherwise. Note that for SCI1 only, a reset is required once a data overrun has occurred.

Show overrun flag

Enables another output. This output is true (1) if a data overrun occurred. Note that for SCI1 only, a reset is required once a data overrun has occurred.

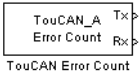
Sample time

The time interval between samples. The default is 1. To inherit the sample time, set this parameter to -1. See “Specifying Sample Time” in the Simulink documentation for more information.

Purpose Count transmit and/or receive errors detected on selected TouCAN modules

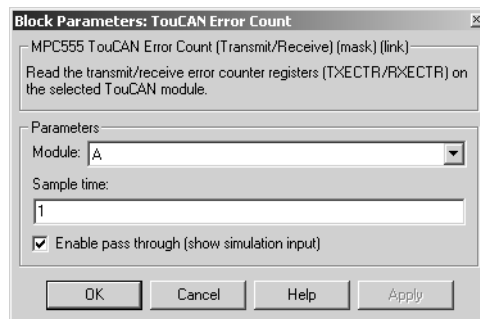
Library Embedded Target for Motorola MPC555

Description 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 6-89.)

Dialog Box



Module

Select TouCAN module A or B.

Sample time

Sample time of the block.

Enable pass through (show simulation input)

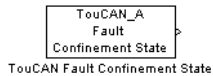
Lets you provide a signal to this block for use in simulation. When this option is enabled, inports appear on the block. The block inputs are passed through, unaltered, to the outputs during simulation. This option affects simulation only.

TouCAN Fault Confinement State

Purpose Indicate the state of a TouCAN module

Library Embedded Target for Motorola MPC555

Description



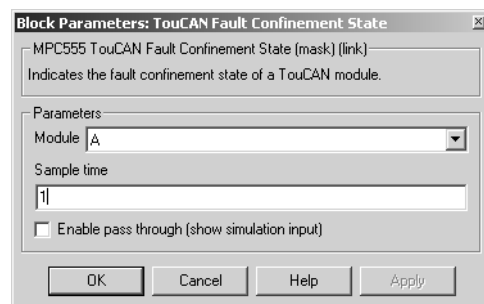
The TouCAN Fault Confinement State block provides an indicator for the state of the selected TouCAN module. The block obtains and outputs a field of two bits from the TouCAN module's Error and Status (ESTAT) register. The possible states are shown in the table below.

Refer to section 16, "CAN 2.0B Controller Module," in the *MPC555 Users Manual* for further information.

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)

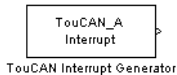
Lets you provide a signal to this block for use in simulation. When this option is enabled, an inport appears on the block. The block input is passed through, unaltered, to the output during simulation. This option affects simulation only.

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.

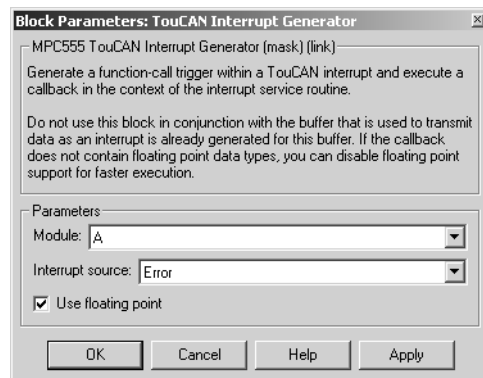


This block may be used to execute a callback on occurrence of Bus Off, Error, or Wake interrupts.

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 run-time.

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

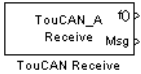
Embedded Target for Motorola MPC555

Description

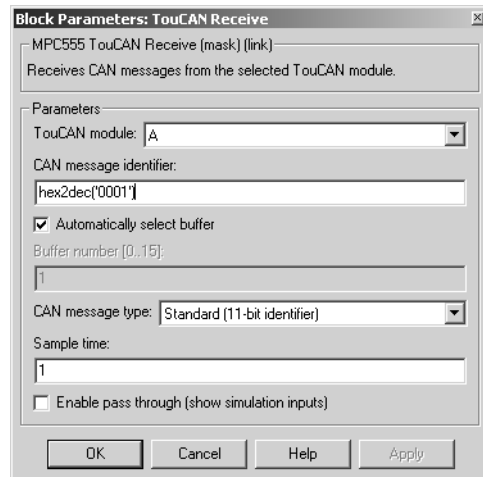
The TouCAN Receive block receives CAN messages from the TouCAN module.

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 block data output.



Dialog Box



TouCAN module

Select 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 6-49) 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 6-59.

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 6-59.

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.

TouCAN Receive

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

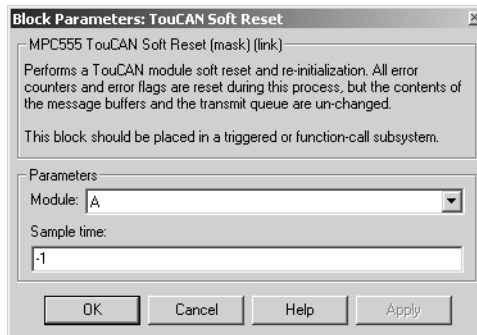
Library Embedded Target for Motorola MPC555

Description 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. See “TouCAN Fault Confinement State” on page 6-80.



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

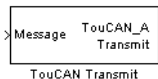
Sample time of the block.

TouCAN Transmit

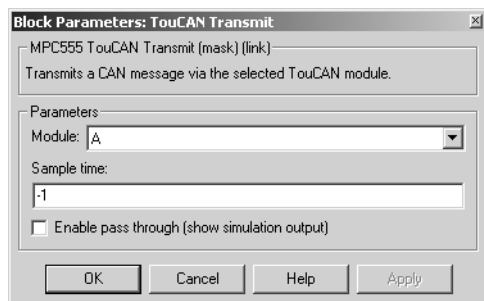
Purpose Transmit a CAN message via a TouCAN module on the MPC555

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 6-49). The block should be connected to CAN Message Packing/Unpacking blocks. Do not ground the block or leave it unconnected. See the demo `mpc555rt_candb` for an example.



Dialog Box



Module

Select one of the two TouCAN modules (A or B).

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 output 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

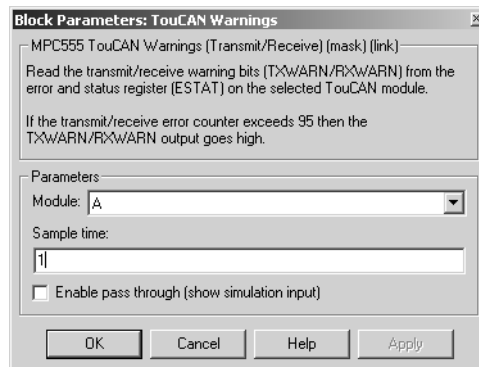
Library Embedded Target for Motorola MPC555

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.

TPU3 Digital In

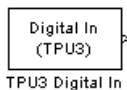
Purpose

Configure a Time Processor Unit (TPU3) channel for digital input

Library

Time Processor Unit (TPU3)

Description



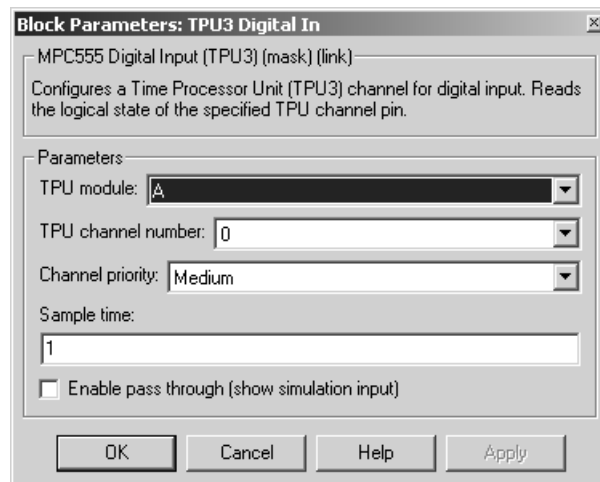
The TPU3 Digital In block reads the logical state of the selected pin (channel) on the TPU3 submodules of the MPC555. You can use this block in the same way as the MIOS Digital In block. You might need to use this block instead of the MIOS Digital In block, for example, if TPU is available but not MIOS. The Channel priority field specifies a number in the range 0..15, corresponding to 16 independent timer channels on each of the two modules of the TPU3. The output of the block represents the logic state of the pin referenced in the module and channels fields. When the signal on a given pin is a logical 1, the block output signal will be equal to 1; otherwise the block output element will equal zero.

The TPU has 16 channels on each module A and B. You can use each of these channels independently, so you could use up to 32 of these blocks, specifying different channels, at once.

Refer to Section 17, “Time Processor Unit 3,” in the *MPC555 Users Manual* for further information.

For an example showing how to use this block see the `mpc555rt_io` demo.

Dialog Box



TPU module

Choose A or B; each has 16 channels.

TPU channel number

Choose 0-15.

Channel priority

Choose Low, Medium or High

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Sample time

The default is always 1 for input driver blocks, but you will need to change this to suit the frequency of your input signals.

Enable pass through (show simulation input)

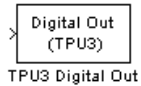
Lets you provide signals to this block for use in simulation. When this option is enabled, an inport appears on the block. The block input is passed through to the output during simulation. (See “Data Type Support and Scaling for Device Driver Blocks” on page 6-7 for information on supported input/output data types and scaling of input/output signals.). This option affects simulation only.

TPU3 Digital Out

Purpose Configure a Time Processor Unit (TPU3) channel for digital output

Library Time Processor Unit (TPU3)

Description



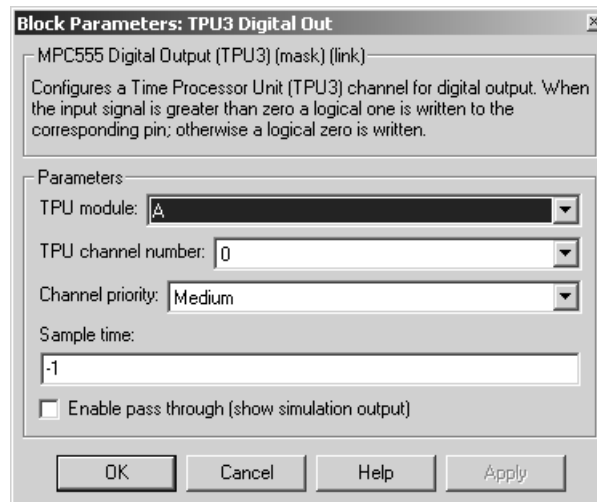
The TPU3 Digital Out block sets the state of the selected pin (channel) on the TPU3 submodule of the MPC555. The Channel priority field specifies a number in the range 0..15, corresponding to the 16 independent channels on each TPU3 module, A and B. You can use each of these channels independently, so you could use up to 32 of these blocks specifying different channels at once.

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 channel.

Refer to Section 17, “Time Processor Unit 3”, in the *MPC555 Users Manual* for further information about the TPU3.

For an example showing how to use this block see the `mpc555rt_io` demo.

Dialog Box



TPU Module

Choose A or B; each has 16 channels.

TPU channel number

Choose 0-15.

Channel priority

Choose Low, Medium or High.

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest first).

Sample time

Default - 1: this setting specifies that the block inherits its sample time from the block connected to its input (inheritance) (unless it is in a triggered subsystem). It makes no sense to sample faster than your input is changing, so normally you leave this at the default.

TPU Digital Out doesn't use a timebase. The output pin is written to at the rate specified by the block sample time. See "Time Processor Unit (TPU3) Configuration Parameters" on page 6-61 for details on settings for the TCR1 clock.

Enable pass through (show simulation input)

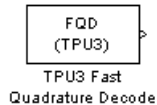
Lets you provide signals from this block for use in simulation. When this option is enabled, an outport appears on the block. (See "Data Type Support and Scaling for Device Driver Blocks" on page 6-7 for information on supported input/output data types and scaling of input/output signals.). This option affects simulation only.

TPU3 Fast Quadrature Decode

Purpose Configure a pair of TPU3 channels for Fast Quadrature Decode (FQD)

Library Time Processor Unit (TPU3)

Description



The TPU3 Fast Quadrature Decode block decodes position information from quadrature encoder hardware. The relative phase of a pair of input signals is used to determine direction of movement. The signals are decoded to increment or decrement the position counter (block output). You can derive a speed from the position information. It is particularly useful for decoding position and direction information from a slotted encoder in motion control systems.

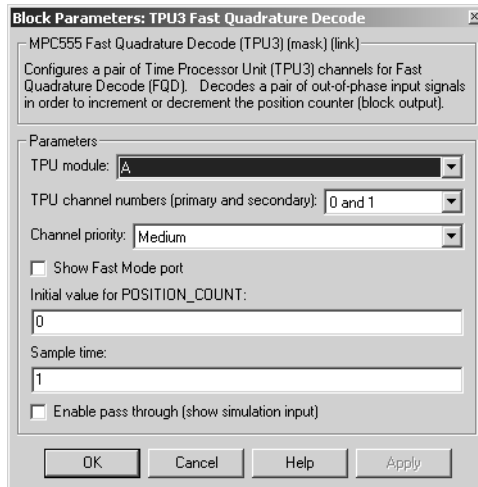
In normal mode (the default), the position counter is incremented or decremented for each valid transition on either channel. The counter increments when the primary channel is ahead and decrements when the primary channel lags. A switch in the phase relationship indicates a change of direction.

At certain speeds you may want to switch to fast mode. You can supply an input to tell the block to switch to fast mode under specified conditions. In fast mode only one of the two input signals is read. The position counter increments or decrements by 4 for each rising transition on the primary channel only (instead of once for each transition in each signal). This reduces the TPU processing load; you can also decode at more than four times the maximum count rate of normal mode.

The counter is 16 bit and free flowing (that is, it overflows to 0, and underflows to 0xFFFF). You must take care when calculating speed derived from the counter, as it may be necessary to use two's complement arithmetic. A useful document is the *TPU Fast Quadrature Decode Programming Note*—search for “*TPUPN02/D*.”

It is possible to overload the TPU processor; if you observe unexpected behavior you should consult the TPU documentation. Refer to Section 17, “Time Processor Unit 3,” in the *MPC555 Users Manual* for further information.

Dialog Box



TPU module

Choose A or B; each has 16 channels.

TPU channel numbers (primary and secondary)

Select a pair of consecutive channels from (0 and 1) to (14 and 15). The primary channel is always the lower channel number.

Channel priority:

Choose Low, Medium, or High

The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Initial value for POSITION_COUNT

Set an initial value. Range checking is applied (must be 16 bit).

Show Fast Mode port

This option is unselected by default. Left unselected, the block always operates in Normal mode. If you select this option, an inport appears where you can input a Boolean signal to control the mode of operation (for example, from a Stateflow subsystem): 0 or false =Normal Mode; 1 or true =Fast Mode.

TPU3 Fast Quadrature Decode

Fast mode conserves TPU activity by only reading one of the two signals. This also allows you to decode at more than four times the maximum count rate of Normal mode. This may be appropriate at some speeds where you can assume the behavior of the second sign—instantaneous direction change is assumed to be impossible. The counter is updated in the same direction as when the last transition was serviced in Normal Mode. The position counter is incremented or decremented by 4 for every rising transition read on the primary channel, instead of having to read all four transitions in the two signals. The Fast Mode port is always the top inport if there are two (if pass through is enabled).

Sample time

The default is always 1 for input driver blocks, but you will need to change this to suit the frequency of your input signals.

This block uses TCR1 as a timebase, but the functionality of the TPU Fast Quadrature Decode (FQD) function used by the block is not changed by changing the speed of the TCR1 clock. The Position Count output is incremented at a rate entirely controlled by the rising and falling edges of the pair of input waveforms, (and the Fast mode input). See “Time Processor Unit (TPU3) Configuration Parameters” on page 6-61 for more information on the TCR1 timebase settings.

Enable pass through (show simulation input)

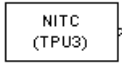
Selecting this option produces an inport, which is a uint16 pass through input for POSITION_COUNT.

TPU3 New Input Capture/Input Transition Counter

Purpose Configure a Time Processor Unit (TPU3) channel for New Input Capture/Input Transition Counter (NITC)

Library Time Processor Unit (TPU3)

Description



TPU3 New Input Capture/
Input Transition Counter

The TPU3 New Input Capture/Input Transition Counter block counts transitions on the input pin and/or captures a TCR timebase value or a TPU parameter RAM value after a certain number of transitions. You can select the number of transitions and whether to capture on rising or falling transitions or both.

You can select up to three outputs to display. Each will have a separate output:

- `FINAL_TRANS_TIME` shows the captured value each time the maximum number of transitions (`MAX_COUNT`) is reached
- `TRANS_COUNT` shows the number of transitions counted (resets each time `MAX_COUNT` is reached)
- `LAST_TRANS_TIME` shows the captured value at the most recent transition, updated at every transition (except final transitions). At the final transition `LAST_TRANS_TIME` shows the captured value at the previous transition.

You can choose whether to capture the TCR1 timebase value each time the `MAX_COUNT` number of transitions is reached, or you can specify the address of a TPU parameter in RAM to capture at that moment. Note this block always operates in continuous mode, not single-shot—transitions are counted up to `MAX_COUNT` and then the block resets and continues counting from zero.

We cannot guarantee that the three outputs are read coherently. They are read one after another, and it is possible that while the memory is accessed for one parameter the next to be read may have changed value. This depends on the speed of your input signal. This should not be important for most purposes because only `TRANS_COUNT` or `FINAL_TRANS_TIME` will be the outputs of interest.

As an example, you could use this block in conjunction with the TPU3 Fast Quadrature Decode block for calibration purposes. Quadrature encoders often generate an index signal in addition to the pair of signals whose relative phase contains the position information. You could put this index signal into an NITC input to count pulses in order to calibrate the position of the encoder.

TPU3 New Input Capture/Input Transition Counter

Refer to Section 17, “Time Processor Unit 3,” in the *MPC555 Users Manual* for further information. A particularly useful document is the *TPU New Input Capture/Input Transition Capture Programming Note*—search for “*TPUPN08/D*.” Look in the appropriate TPU programming note to look up parameter addresses if you want to capture TPU Parameters instead of TCR1 clock ticks.

As an example of using TPU parameters, if you wanted to use this block to capture the position count from a TPU Fast Quadrature Decode block, you need to set the correct channel number and parameter address. You must set the channel number to the primary FQD channel (FQD blocks use a pair of channels, the first is primary). Each TPU channel can have up to eight parameters (0 through 7), in this case you must choose parameter 1 (POSITION_COUNT).

TPU3 New Input Capture/Input Transition Counter

Dialog Box

Block Parameters: TPU3 New Input Capture/ Input Transition ...

MPC555 New Input Capture/Input Transition Counter (TPU3) (mask) (link)

Configures a Time Processor Unit (TPU3) channel for New Input Capture/Input Transition Counter (NITC). Counts individual transitions on the input pin, and allows the capture of a TCR or TPU parameter RAM value after a selectable number of pin transitions.

Parameters:

TPU module: A

TPU channel number: 0

Channel priority: Medium

Show FINAL_TRANS_TIME port

Show TRANS_COUNT port

Show LAST_TRANS_TIME port

Detect transition on: Rising Edge

Capture: TCR1 Value

Number of transitions before capture and reset (MAX_COUNT): 1

Sample time: 1

Enable pass through (show simulation input)

OK Cancel Help Apply

TPU module

Choose A or B; each has 16 channels.

TPU channel number

Choose 0-15.

Channel priority:

Choose Low, Medium, or High

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

TPU3 New Input Capture/Input Transition Counter

Show FINAL_TRANS_TIME port

Outputs the value captured each time the maximum number of transitions (MAX_COUNT) is reached. This value is only captured when MAX_COUNT is reached.

Show TRANS_COUNT port

Outputs the number of transitions counted. Resets to zero each time MAX_COUNT is reached.

Show LAST_TRANS_TIME port

Outputs the captured value at the latest transition. This is updated at every transition except the final one.

Detect transition on:

Choose from Rising Edge, Falling Edge or Either Edge.

Capture:

TCR1 Value — captures the value of the TCR1 timebase. See “Time Processor Unit (TPU3) Configuration Parameters” on page 6-61 for information on setting the TCR1 timebase.

Parameter RAM Value — captures the value of a TPU parameter in RAM. If you select this option you must choose the TPU channel number and parameter address (out of up to eight parameters per TPU channel).

Number of transitions before capture and reset (MAX_COUNT)

This must be a 16-bit number specifying how many transitions to count before capturing and then resetting. A zero will be equivalent to 1 (you cannot count zero transitions) and you must not exceed the maximum of a uint16 number. The range of an unsigned 16-bit number is 0-65535 (because $65535 = (2^{16}) - 1$).

Range checking is applied; you will receive a warning if you input an unsuitable number.

Sample time

Be sure to set sample time fast enough not to miss any transitions. This will depend on the frequency of your input signal.

Enable pass through (show simulation input)

Produces an inport where you can supply a signal for simulation purposes.

TPU3 Programmable Time Accumulator

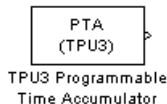
Purpose

Configure a Time Processor Unit (TPU3) channel for Programmable Time Accumulator (PTA).

Library

Time Processor Unit (TPU3)

Description



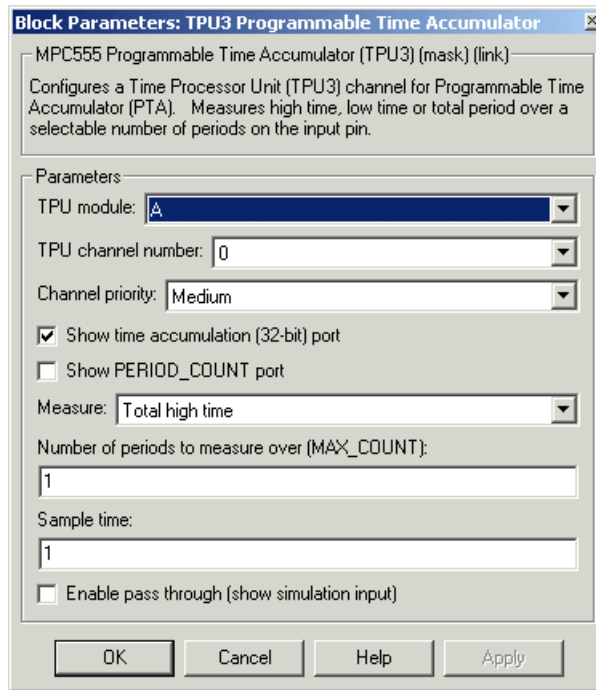
The TPU3 Programmable Time Accumulator block reads an input pin and measures an accumulation of time over a specified number of periods - either high time, low time, or the total time. You can output the accumulated time, or the number of periods or both. You can choose whether to start counting total period on a rising or falling edge.

The accumulated time value will be read at most once between any two model steps. TPU interrupts are used to ensure the 32-bit output is updated only when an accumulation is complete. This ensures that the values of the parameters `HW` and `LW` combined to create the 32-bit output are coherent. This block is under MPC555 Resource Configuration object control, and you will receive a warning if you have not enabled TPU interrupts. If your model contains any PTA blocks, you must change the TPU IRQ settings to enable interrupts. See “Time Processor Unit (TPU3) Configuration Parameters” on page 6-61.

Refer to Section 17, “Time Processor Unit 3,” in the *MPC555 Users Manual* for further information. A particularly useful document is the *Programmable Time Accumulator TPU Function (PTA) Programming Note*—search for “TPUPN06/D.”

TPU3 Programmable Time Accumulator

Dialog Box



TPU module

Choose A or B; each has 16 channels.

TPU channel number

Choose 0-15

Channel priority:

Choose Low, Medium, or High

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

TPU3 Programmable Time Accumulator

Show time accumulation (32-bit) port

Outputs the 32-bit time accumulation value (in TCR1 clock ticks) each time MAX_COUNT is reached. Whether the accumulation measures high time, low time or total time depends on the **Measure** setting.

Show PERIOD_COUNT port

Outputs the number of periods counted.

Measure:

Choose from Total high time, Total low time, Total period (starting on rising edge), Total period (starting on falling edge)

Number of periods to measure over (MAX_COUNT):

Set the number of periods to accumulate time over, up to a maximum of 255. The value is read each time MAX_COUNT is reached. Note that MAX_COUNT is 8-bit here (it is 16-bit in the TPU3 New Input Capture/Input Transition Counter block).

Sample time:

Make sure you set a sample time fast enough not to miss any periods, depending on the frequency of your input signal.

Enable pass through (show simulation input)

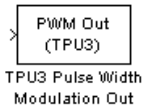
Produces an inport where you can supply a signal for simulation purposes.

TPU3 Pulse Width Modulation Out

Purpose Configure a Time Processor Unit (TPU3) channel for pulse width modulation (PWM) output

Library Time Processor Unit (TPU3)

Description



The TPU3 Pulse Width Modulation Out block is used for Pulse Width Modulation (PWM) output from the TPU3 modules. You can use this block in the same way as the MIOS PWM Out block, and with the TPU block you can also vary the period dynamically using a block inport. You can modulate up to 16 of these for each module (A and B) using any of the independent TPU channels.

A PWM signal is a rectangular waveform whose period may or may not be constant, and whose duty cycle can be varied, under control of a modulator signal, between 0% and 100%. You can either control the period register directly, or enter the desired (ideal) period and the mask will solve for the best values for the period register. Note for the MIOS Pulse Width Modulation Out block the period is constant, but with the TPU Pulse Width Modulation Out block you can also vary the period of the PWM signal (using the Show PWMPER port option you can supply the period as an input).

The TPU3 Pulse Width Modulation Out block acts as the modulator, controlling the duty cycle and period of the signal on the output channel. There can be one or two inputs. Input one (top) is always the duty cycle. Here an input signal in the range 0 to 1 generates a PWM output with corresponding duty cycle. Input signals outside this range cause the duty cycle to saturate at 0% or 100%.

You can specify the period register manually in the mask. If you select the Show PWMPER port option, input two appears. Here you can supply the period as an input, instead of specifying the period in the mask. PWMPER input (either block input or specified as a mask variable) must be 16 bit values with saturation applied to be in the range $0 \leq \text{PWM Period Register Value} \leq 32768$ (0x8000).

This saturation (including pass through) means that the block will not allow you to enter a value for PWMPER $> 0x8000$, or a value for ideal period that makes the PWMPER register go outside this range.

TPU3 Pulse Width Modulation Out

There can be one or two outputs if you select Enable pass through to show simulation input. Output one (top) is the duty cycle input (saturated if input is outside the range of 0-1) passed through to the output. If you supply the period as an input using the PWMPER port, a second output also appears. This period pass through output is saturated to be in the range $0 \leq \text{PWM Period Register Value} \leq 32768$ (0x8000). Note the pass through output is not the PWM waveform, rather it is the value of the PWM duty cycle.

The TPU Pulse Width Modulation Out block uses TCR1 as a timebase for creating the output waveform. By changing the speed of the TCR1 clock, the range of available PWM periods changes. See “Time Processor Unit (TPU3) Configuration Parameters” on page 6-61 for more information on settings for the TCR1 clock.

Refer to Section 17, “Time Processor Unit 3,” in the *MPC555 Users Manual* for further information.

For an example showing both ways to use this block (specifying the period, and using the PWMPER port to input the period), see the `mpc555rt_io` demo.

TPU3 Pulse Width Modulation Out

Dialog Box

Block Parameters: TPU3 Pulse Width Modulation Out

MPC555 Pulse Width Modulation Output (TPU3) (mask) (link)

Configures a Time Processor Unit (TPU3) channel for pulse width modulation (PWM) output. An input signal in the range 0 to 1 generates a PWM output with corresponding duty cycle; input signals above (below) this range cause the duty cycle to saturate at 100% (0%).

Parameters

TPU module: A

TPU channel number: 0

Channel priority: Medium

Show PWMPER port

Edit period register manually

Waveform actual period:

Waveform ideal period: 0.02

Pulse period register (PWMPER):

Sample time: -1

Enable pass through (show simulation output)

OK Cancel Help Apply

TPU Module

Choose A or B; each has 16 channels.

TPU channel number

Choose 0-15

Channel priority

Choose Low, Medium, or High

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are

serviced is determined by assigned priority first, followed by channel number (lowest number first).

Show PWMPER port

If you select this box, the parameters relating to setting the period register disappear because they are no longer used.

A new input appears on the block when you select this option. Here you can input the period register value. Saturation is applied: $0 \leq x \leq 32768$ (0x8000). If **Enable pass through** is also checked you also gain a new output, resulting in a block with two inputs and two outputs. You can see an example of the block in this state in the demo model `mpc555rt_io`.

Edit period register manually

If you select this check box, you can set the **Pulse period register** parameter.

Pulse period register (PWMPER)

The default is 12500. You can enter a value for the period register here ($0 \leq x \leq 32768$ (0x8000)). The actual waveform period is calculated and displayed in the edit box. If **Edit period register manually** is not selected, this edit box is gray.

Waveform ideal period

The default is 0.02. You can enter the waveform period you want by typing in this edit box. From this the period register is calculated and appears in the **Pulse period register (PWMPER)** edit box. The actual waveform period is also calculated and displayed, see below.

Waveform actual period

You can never enter anything in this box (so it is always gray)—it is there purely to inform you, and does not affect the model code. You might find this information useful because actual and ideal waveform period are not always the same—the ideal period you enter may not always be possible.

Sample time

The default is -1: This setting specifies that the block inherits its sample time from the block connected to its input (inheritance) (unless it is in a triggered subsystem). It makes no sense to sample faster than your input is changing, so normally you leave this at the default.

TPU3 Pulse Width Modulation Out

Enable pass through (show simulation input)

Lets you provide a signal from this block for use in simulation. When this option is selected, an outputport appears on the block, or two outputports if **Show PWMPER port** is also selected.

The duty cycle input (top) is passed through to the output (saturated if outside 0-1) during simulation, while the period register pass through output (the lower one) is saturated to be in the range $0 \leq x \leq 32768$ (0x8000). Note the pass-through output is not the PWM waveform, rather it is the value of the PWM duty cycle.

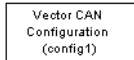
(See “Data Type Support and Scaling for Device Driver Blocks” on page 6-7 for information on supported input/output data types and scaling of I/O signals.). Note that the top input signal is duty cycle value, while the lower input is the period register value.

This option affects simulation only.

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

Library Can Drivers (Vector)

Description The Vector CAN Configuration block configures a CAN channel on the host PC, using the Vector CAN Driver software. A CAN channel can be



Vector CAN Configuration

- 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 Bit 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 bit rate** settings in the **Block Parameters** dialog box. We recommend using the precalculated bit rates wherever possible.

If the precalculated bit rates do not meet your requirements, you can select the **Set bit timing parameters manually** option. You can then set the bit rate by configuring the **Bit rate prescaler**, **Synchronization jump width**, **Time segment 1**, **Time segment 2**, and **Sample mode** parameters as described later in this section.

Vector CAN Configuration

The following variables are defined in calculating the bit rate for the Vector CAN Configuration block:

- f : The Vector hardware oscillator frequency f is a constant, 16MHz.
- *prescaler*: The bit 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 bit 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 Bit rate = (number of bit times per second) = qps / qpb .

Next, select values for the following parameters:

- **Synchronization Jump Width (SJW)** — 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:

- **Bit 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

- **Bit rate prescaler:** 1
- **Synchronization jump width:** 1
- **Time segment 1:** 8.
- **Time segment 2:** 7.
- **Sample mode:** 1 sample per bit

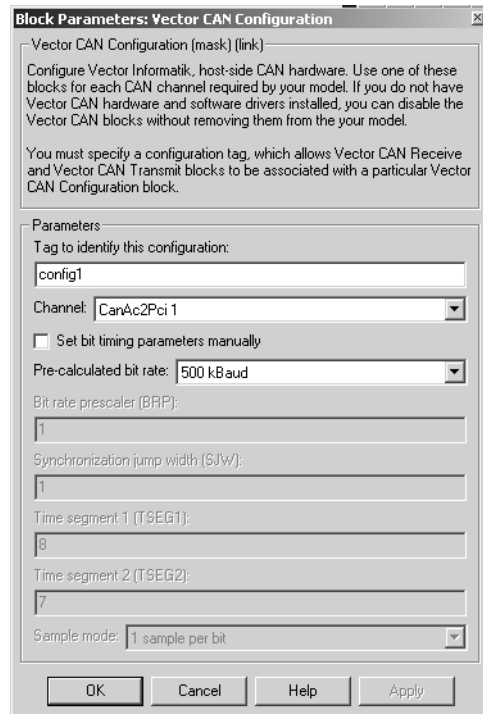
For further information on the CAN, and on CAN bit timing, see the following documents, available at the CAN in Automation (CiA) Web site:

<http://www.can-cia.de>:

- *CAN General Introduction*
- The “Bit Timing” section of the *CAN Physical Layer* document

Vector CAN Configuration

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 bit rate

This parameter sets the speed of your channel connection. If none of the precalculated bit 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 Bit Rate” on page 6-109 for details.

Bit rate prescaler, Synchronization jump width, Time segment 1, Time segment 2, and Sample mode

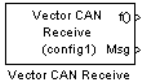
These parameters are enabled only if **Set bit timing parameters manually** is selected. See “Setting the Bit Rate” on page 6-109.

Vector CAN Receive

Purpose Read CAN messages from a Vector CAN channel

Library Can Drivers (Vector)

Description



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 6-116.

Due to a limitation of the Vector Informatik CAN hardware/software package, a maximum of 30 Vector CAN Receive Blocks, on a given channel, can be included in a model. You can specify a vector of CAN message identifiers for each block, so you are not limited to 30 message identifiers on a channel.

Note that this limitation does not apply to Vector CAN Transmit Blocks. An unlimited number of Vector CAN Transmit Blocks can be included in a model.

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.

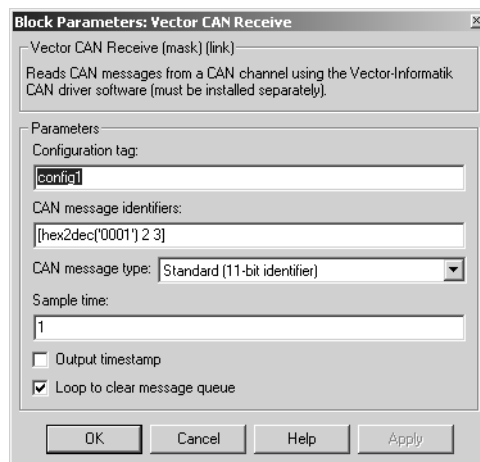
When the Vector CAN Receive block executes, it polls the queue to see how long it is. If **Loop to clear message queue** is selected, then the block will read all messages from the queue and process them sequentially. If more messages arrive during this process, they are added to the end of the queue 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 when the 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.

Vector CAN Receive

CAN message identifiers

Specify a vector of CAN message identifiers for the messages 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

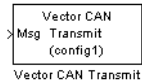
(On by default.) Controls how the block handles cases where multiple messages are queued up between executions of the block. When **Loop to clear message queue** is selected, all messages in the queue for the block at the time of execution are processed. Any connected function call subsystem executes 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.

Purpose Transmit CAN messages on a Vector CAN channel

Library Can Drivers (Vector)

Description



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.

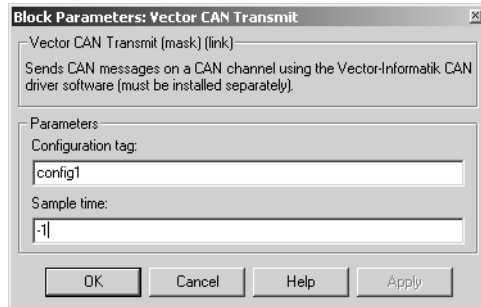
The Vector CAN Transmit block is designed to be placed in a triggered subsystem in order to transmit a message upon an event received. In this case you should specify an inherited sample time by entering -1 in the **Sample time** parameter.

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.

Vector CAN Transmit

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.

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 time-out 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 time-out 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.

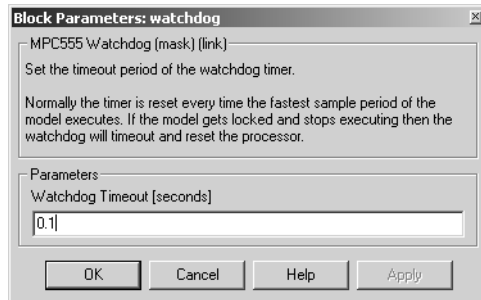
In the event of a software failure that causes the application to lock up, the watchdog timer will not be serviced. Therefore, it will time out when the counter reaches zero. This in turn causes a processor reset, which restarts the application.

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 time-out period is set to the largest possible value, which is several seconds.

Note that the Watchdog block has neither input nor output connections.

Watchdog

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.

Toolchains and Hardware

This section discusses specific settings for different cross-development environments:

Setting Up Your Installation with Diab Cross-Compiler and SingleStep Debugger (p. A-3)	Configuring the Embedded Target for Motorola MPC555 for use with the Diab development tools.
Setting Up Your Installation with Metrowerks CodeWarrior (p. A-6)	Configuring the Embedded Target for Motorola MPC555 for use with the Metrowerks CodeWarrior development tools
Setting Up Your Target Hardware (p. A-9)	Configuring the required connections and jumper settings for the Phytex phyCORE-MPC555 development board
CAN Hardware and Drivers (p. A-12)	Configuring supported CAN hardware and software.
Configuration for Nondefault Hardware (p. A-13)	Manual configuration for different MPC555 hardware, including altering boot code and tool configurations for different hardware clock speeds, ports, and boards.

Setting Up Your Toolchain

The currently supported toolchains are WindRiver (Diab and SingleStep) and CodeWarrior. You must first install and configure your toolchain to work with the Embedded Target for Motorola MPC555. The necessary steps are described in the following sections:

- “Setting Up Your Installation with Diab Cross-Compiler and SingleStep Debugger” on page A-3
- “Setting Up Your Installation with Metrowerks CodeWarrior” on page A-6

Note Do not install your toolchain in a directory with spaces in the name. This may cause build failures.

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:

- An MPC555 development board (such as the phyCORE-MPC555 development board, or the Axiom board) and a debugger connector (such as the BDM Wiggler from Macraigor Systems). Note the phyCORE-MPC555 board comes with built-in debugger connector which you can plug a parallel port connector into directly, in which case you may not require a BDM connector.
- Wind River Systems Diab cross-compiler (version 4.4b or later)
- 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 “Setting Target Preferences for Diab and SingleStep” on page A-4).

Install SingleStep Debugger

The SingleStep debugger, in conjunction with the Embedded Target for Motorola MPC555, lets you download, run and debug generated code.

Follow the instructions of the SingleStep installer. During installation you should select the SStep Professional Suite (MPC5xx) option.

For SingleStep 7.6.2, you may want to 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.

This document describes use of SingleStep version 7.6.2 and this may differ from your installed version of SingleStep, or with future versions of SingleStep. To resolve questions or difficulties with SingleStep, refer to the SingleStep documentation, or contact Wind River Systems.

You will need to note the path to the installed SingleStep debugger in order to configure your target preferences (see “Setting Target Preferences for Diab and SingleStep” on page A-4).

Setting 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 1-11, if you have not yet done so.)

- 1 Select Start -> Simulink -> Embedded Target for Motorola MPC555 -> Target Preferences.**

This opens the Target Preferences GUI where you can edit the settings for your cross-development environment.

- 2 Select Diab from the drop-down Toolchain menu**
- 3 Expand the ToolChainOptions by clicking the plus sign, and type the correct path into CompilerPath.**
- 4 For SingleStep you must also type the correct path into DebuggerPath.** This is not necessary for CodeWarrior as the compiler and debugger are integrated.

Note that the path to the SingleStep debugger, specified in DebuggerPath in the Target Preference GUI, is the root directory of your SingleStep installation, on either an actual hard drive on your PC, or a mapped drive. Do not use a Universal Naming Convention (UNC) path. For most purposes, the other target preferences fields can be left at their defaults. Once you have set these

target preferences, the build process will automatically invoke your compiler and debugger when required for downloading code to flash memory and RAM.

The **DebuggerSwitches** target preference is specific to SingleStep. If you want to change the default debug settings, type

```
help debug
```

at the SingleStep command line to see the options available. For example you can change parallel port here. The default is `-p LPT1=1` which specifies port 1 on your host PC at speed 1. You could change it to `-p LPT2=2` to specify port 2 at speed 2.

The default **DebuggerExecutable** `bdmp58.exe` and the default **DebuggerSwitches** are set up for Wiggler Background Debug Mode (BDM) port connectors. Other executables are supplied with SingleStep — if you want to change the defaults to use a different connection device and different debug settings, consult the SingleStep documentation.

Configure phyCORE-MPC555 Jumpers

Make sure that the jumpers on the phyCORE-MPC555 board are set as described in “Phytec Jumper Settings” on page –9. 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.

The next step is to verify your installation:

- 1 You can download and run the test program supplied. See “Run Test Program” on page 1–15.
- 2 You must then follow the instructions to download boot code (“Download Boot Code to Flash Memory” on page 1–15). Once you have completed these steps, you can begin working with Embedded Target for Motorola MPC555.

Setting Up Your Installation with Metrowerks CodeWarrior

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

- An MPC555 development board (such as the phyCORE-MPC555 development board) and a debugger connector (such as the BDM Wiggler from Macraigor Systems). Note the phyCORE-MPC555 board comes with built-in debugger connector which you can plug a parallel port connector into directly, in which case you may not require a BDM connector.
- Metrowerks CodeWarrior for Embedded PowerPC CD (or network access to the Metrowerks CodeWarrior for Embedded PowerPC installer).
 - Version 1.1 of Embedded Target for Motorola MPC555 requires version 6.0 of Metrowerks CodeWarrior For PowerPC Embedded Systems.
 - Version 1.11 of Embedded Target for Motorola MPC555 requires version 6.5 of Metrowerks CodeWarrior For PowerPC Embedded Systems.

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.
- 3** Open CodeWarrior IDE. You can use the Windows Start menu (**Start -> Programs -> CodeWarrior -> CodeWarrior IDE**).
- 4** Select **Edit -> Preferences -> Build Settings -> Build Before Running**
- 5** Select the option **Never**.

It is vital you set this to avoid errors when building and automatically downloading code with Embedded Target for Motorola MPC555.

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 0x3f9800.
 - 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 Select **Edit -> Preferences -> General -> Build Settings -> Build Before Running** and set to Never.
- 6 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 1-11, if you have not yet done so.). Follow these steps:

- 1** Select **Start -> Simulink -> Embedded Target for Motorola MPC555 -> Target Preferences.**

This opens the new Target Preferences GUI where you can edit the settings for your cross-development environment.

- 2** Select CodeWarrior from the drop-down **Toolchain** menu.
- 3** Expand the ToolChainOptions by clicking the plus sign, and type the correct path into **CompilerPath.**

Note that when using CodeWarrior, you do not also have to specify the DebuggerPath, as the compiler and debugger are integrated. 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.

The next step is to verify your installation.

- 1** You can download and run the test program supplied. See “Run Test Program” on page 1-15.
- 2** You must then follow the instructions to download boot code (“Download Boot Code to Flash Memory” on page 1-15). Once you have completed these steps, you can begin working with Embedded Target for Motorola MPC555.

Setting Up Your Target Hardware

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

If you are not using the phyCORE-MPC555 development board see “Configuration for Nondefault Hardware” on page A-13.

Phytex 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 Phytex 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*.

Phytex Jumper Settings

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

For jumper locations and pin numbers, see Jumper Layout section of the *Development Board for phyCORE-MPC555 Hardware Manual* found at

http://www.phytex.de/pdf/docu_eng/L-525E.pdf

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
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

Jumper	Description	Raven or Blackbird	Wiggler	On-Board BDM
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
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.

CAN Hardware and Drivers

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

The blocks in the CAN Drivers (Vector) blockset, and the `candownload` utility, require correct installation of a CAN card and drivers from Vector-Informatik. See your Vector-Informatik documentation for instructions on installation and verification. This product has been tested with the following hardware / driver combinations:

- CAN-AC2-PCI - Plug & Play Driver V3.4 for Windows 98 / ME / 2000 / XP
- CANcardX - Plug & Play Driver V3.4 for Windows 98 / ME / 2000 / XP

Older Vector CAN drivers should also work without any major problems, however we recommend you install the most recent drivers so that you have the latest improvements and bug fixes.

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).

In addition, after installing the drivers for your hardware, you must also download the CANdriver Library (Programming Library V3.2) from Vector-Informatik, and make sure that the library, `vcand32.dll` is placed in a location on the Windows system path. We recommend placing `vcand32.dll` in the Windows `system32` directory, which will save you having to change the path environment variable itself. If these configuration steps are not followed then errors in the use of the CAN Drivers (Vector) blockset and the `candownload` utility will occur.

Configuration for Nondefault Hardware

The Embedded Target for Motorola MPC555 has been developed and fully tested using Phytex phyCORE-MPC555 development board. We strongly recommend the use of this board for getting started with the Embedded Target for Motorola MPC555. If you are using different MPC555 hardware, it may be necessary to perform some additional manual configuration.

The next section describes how to configure the Embedded Target for Motorola MPC555 real-time target for use on hardware with crystal frequencies other than 20 MHz.

The following sections give additional information about where to make changes for other hardware-specific configurations.

Hardware Clock Configuration

The Embedded Target for Motorola MPC555 uses the Periodic Interrupt Timer (PIT) to support a range of sample times. Note that the PIT is driven by the crystal frequency. This results in the following possible sample time ranges:

For a crystal frequency of 20Mhz:

- Fastest sample time = $1.28e-5$ seconds.
- Slowest sample time = 0.8388 s.

For a crystal frequency of 4 Mhz:

- Fastest sample time = $6.4e-5$ s.
- Slowest sample time = 4.1942 s.

Note that if you select a sample time slower than the slowest possible for your clock frequency, Simulink issues a warning message.

Also note that the fastest sample time may not be achievable because timer overruns may occur, depending on your model.

Configuring the Embedded Target for Motorola MPC555 for a crystal frequency other than 20 MHz.

The MPC555 can operate with a crystal frequency of either 4 MHz or 20 MHz. By default, the Embedded Target for Motorola MPC555 is configured for a crystal frequency of 20 MHz.

For the real-time target, you must change a number of configuration settings as detailed below.

It is necessary to use flash driver files that are compatible with the crystal frequency on your hardware. Pre-built driver files for System Clock 20 MHz and crystal frequency of 4 MHz or 20 MHz are provided in

```
matlabroot\toolbox\rtw\targets\mpc555dk\drivers\src\libsrc\extensions\cmf_flash\General_Market_CMF_Driver_V3.0.3\MPC555\data_generator\prebuiltdata
```

Note that the boot code runs with System Clock of 20 MHz irrespective of whether the crystal frequency is 4 MHz or 20 MHz.

To configure Embedded Target for Motorola MPC555 for a 4 MHz crystal frequency:

- 1 The required changes can be made automatically by typing the following at the MATLAB command line

```
oscillator_frequency_configure(4e6)
```

Running this command makes small modifications to the following files:

```
matlabroot\toolbox\rtw\targets\mpc555dk\drivers\src\libsrc\extensions\cmf_flash\makefile
matlabroot\toolbox\rtw\targets\mpc555dk\drivers\src\libsrc\standard\include\clocks_common.h
matlabroot\toolbox\rtw\targets\mpc555dk\mpc555dk\@codewarrior_tgtaction\mpc555.cfg
matlabroot\toolbox\rtw\targets\mpc555dk\mpc555dk\@diab_tgtaction\mpc555.cfg
matlabroot\toolbox\rtw\targets\mpc555dk\mpc555dk\@diab_tgtaction\phycore-555.wsp
matlabroot\toolbox\rtw\targets\mpc555dk\mpc555dk\@codewarrior_tgtaction\mpc555.cfg
```

Note Note that you can change back to the original configuration for 20 MHz by running the same command with an argument of 20e6 oscillator_frequency_configure(20e6).

After running this command you must recompile the boot code and program it onto your target hardware as follows:

- 2 Recompile the boot code as described in “Boot Code Parameters for CAN Download” on page 3-33. Recompile produces a new bootcode.elf file.
- 3 Program the updated bootcode.elf file onto the target hardware, as described in “Downloading Boot Code” on page 3-23.
- 4 For each model that you create, you must edit the system configuration parameters of the MPC555 Resource Configuration block. Change the **Oscillator Frequency** parameter to 4 MHz, and change the **USIU_PLPRCR_B_MF** parameter to 4. This will result in a system clock of 20 MHz.

Note that the boot code uses pre-built flash driver files that should be suitable for most hardware, however, it is possible to generate driver files for other configurations. To do this you can run the **GMD Data Generator** located in

```
matlabroot\toolbox\rtw\targets\mpc555dk\drivers\src\libsrc\extensions\cmf_flash\General_Market_CMF_Driver_V3.0.3\MPC555\data_generator\.
```

Refer to the GMD documentation found in

```
matlabroot\toolbox\toolbox\rtw\targets\mpc555dk\drivers\src\libsrc\extensions\cmf_flash\General_Market_CMF_Driver_V3.0.3\MPC555\*.pdf
```

Other Configuration Changes for Nondefault Hardware

Depending on your target hardware, it may be necessary to make changes to configure settings such as the size and type of external memory.

If you are downloading using the Metrowerks CodeWarrior development environment, the relevant hardware configuration settings are contained in

```
matlabroot\toolbox\rtw\targets\mpc555dk\mpc555dk\@codewarrior_tg  
taction\mpc555.cfg
```

If you are downloading using the Diab and SingleStep development environment, the configuration settings are contained in

```
matlabroot\toolbox\rtw\targets\mpc555dk\mpc555dk\@diab_tgtaction  
\mpc555.cfg
```

and

```
matlabroot\toolbox\rtw\targets\mpc555dk\mpc555dk\@diab_tgtaction  
\phycore-555.wsp
```

Note that there is now only one SingleStep workspace file for RAM and flash memory.

The necessary changes to these files depend on the hardware that you are using. Depending on your hardware, you may also need to configure switches and jumper settings. Consult the documentation for your development board.

If you are generating standalone real-time applications, you may also need to make changes to settings that are contained in the startup code. These are contained in

```
matlabroot\toolbox\rtw\targets\mpc555dk\drivers\src\applications  
\bootcode\bootcode_init.s
```

Note that after making any changes to `bootloader_init.s`, you must recompile the boot code as described in “Boot Code Parameters for CAN Download” on page 3-33.

A

- algorithm export 5-2
- ASAP2 files, generating 3-36

B

blocks

- CAN Calibration Protocol 6-14, 6-15
- CAN Message Filter 6-21
- CAN Message Packing 6-23
- CAN Message Packing (CANdb) 6-27
- CAN Message Unpacking 6-25
- CAN Message Unpacking (CANdb) 6-30
- MIOS Digital In 6-37
- MIOS Digital Out 6-39
- MIOS Digital Out (MPWMSM) 6-41
- MIOS Pulse Width Modulation Out 6-43
- MIOS Waveform Measurement 6-46
- MPC555 Resource Configuration 6-49
- QADC Analog In 6-65
- QADC Digital In 6-70
- Serial Receive 6-76
- Serial Transmit 6-73
- TouCAN Error Count 6-79
- TouCAN Fault Confinement State 6-80
- TouCAN Interrupt Generator 6-82
- TouCAN Receive 6-84
- TouCAN Soft Reset 6-87
- TouCAN Transmit 6-88
- TouCAN Warnings 6-89
- TPU Digital In 6-90
- TPU Digital Out 6-92
- TPU Fast Quadrature Decode 6-94
- TPU New Input Capture/Input Transition Counter 6-97
- TPU Programmable Time Accumulator 6-101
- TPU Pulse Width Modulation Out 6-104

- Vector CAN Configuration 6-109
- Vector CAN Receive 6-114
- Vector CAN Transmit 6-117
- Watchdog 6-119

C

- CAN Calibration Protocol (CCP) 6-14, 6-15
- CAN Calibration Protocol block 6-14, 6-15
- CAN Message Filter block 6-21
- CAN Message Packing block 6-23
- CAN Message Unpacking block 6-25
- CANdb
 - Message Packing block 6-27
 - Message Unpacking block 6-30
- code analysis report 5-3
- Configuration Class blocks 6-11
- cosimulation 4-2

D

- demos for Embedded Target for Motorola MPC555
 - 2-2
- device driver blocks
 - input data types 6-7
 - input scaling 6-7
 - MPC555 Serial Receive 6-76
 - MPC555 Serial Transmit 6-73
 - output data types 6-7
 - output scaling 6-7
- downloading code to target 3-20
 - application code 3-25
 - to flash memory 3-26
 - to RAM 3-25
 - boot code 3-23
 - via BDM port 3-23

via CAN 3-24

E

Embedded Target for Motorola MPC555
demos 2-2
feature summary 1-6

I

installation of Embedded Target for Motorola
MPC555 xii

M

MIOS Digital In block 6-37
MIOS Digital Out (MPWMSM) block 6-41
MIOS Digital Out block 6-39
MIOS Pulse Width Modulation Out block 6-43
MIOS Waveform Measurement block 6-46
MPC555 Resource Configuration object 6-49

O

ODBC
using with CANdb message packing block
6-29
using with CANdb message unpacking block
6-33

P

PIL (processor-in-the-loop) cosimulation 4-2
benefits of 4-2
getting started tutorial 4-5
hardware connections for 4-5
in plant/controller simulation 4-3
preparation for 4-5

technical overview of 4-3

PIL (processor-in-the-loop) target 4-2
files and directories created by 4-24
in cosimulation 4-14
in SIL simulation 4-21
using in closed-loop simulation 4-21

Q

QADC Analog In block 6-65
QADC Digital In block 6-70

R

real-time target
introduction 3-2
tutorial 3-4
code generation 3-11
example model for 3-6
prerequisites for 3-4
using pass-through for device drivers 3-9

S

Serial Receive block 6-76
Serial Transmit block 6-73
software-in-the-loop (SIL) simulation 4-21

T

target hardware setup
communications ports A-9
jumper settings A-9
TouCAN Error Count block 6-79
TouCAN Fault Confinement State block 6-80
TouCAN Interrupt Generator block 6-82
TouCAN Receive block 6-84

TouCAN Soft Reset block 6-87
TouCAN Transmit block 6-88
TouCAN Warnings block 6-89
TPU Digital In block 6-90
TPU Digital Out block 6-92
TPU Fast Quadrature Decode block 6-94
TPU New Input Capture/Input Transition
Counter block 6-97
TPU Programmable Time Accumulator block
6-101
TPU Pulse Width Modulation Out block 6-104
typographical conventions (table) xii

V

Vector CAN Configuration block 6-109
Vector CAN Receive block 6-114
Vector CAN Transmit block 6-117

W

Watchdog block 6-119
watchdog timer 6-119