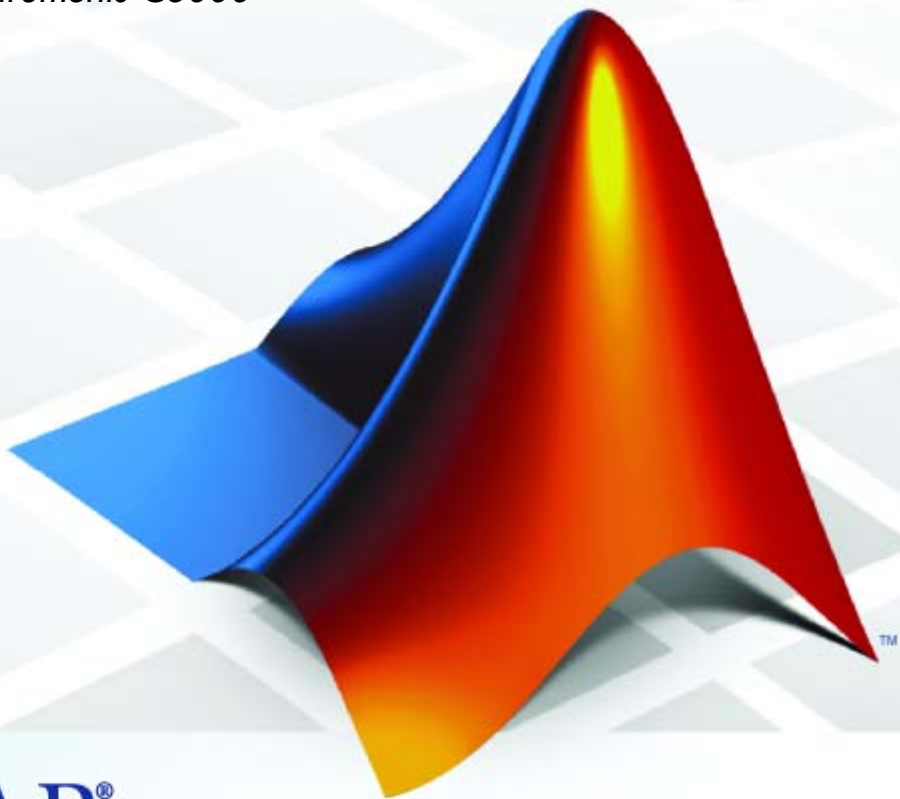


Target Support Package™ 4

User's Guide

For Use with Texas Instruments C6000™



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab@mathworks.com)
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



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

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

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

Target Support Package™ User's Guide

© COPYRIGHT 2002–2010 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, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

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

Revision History

July 2002	Online only	Revised for Version 1.0 (Release 13)
January 2003	Online only	Revised for Version 1.1
September 2003	Online only	Revised for Version 2.0 (Release 13SP1+)
June 2004	Online only	Revised for Version 2.1 (Release 14)
August 2004	Online only	Revised for Version 2.2
October 2004	Online only	Revised for Version 2.2.1 (Release 14SP1)
October 2004	Online only	Revised for Version 2.0 (Release 13SP2)
December 2004	Online only	Revised for Version 2.3 (Release 14SP1+)
March 2005	Online only	Revised for Version 2.3.1 (Release 14SP2)
September 2005	Online only	Revised for Version 2.4 (Release 14SP3)
March 2006	Online only	Revised for Version 3.0 (Release 2006a)
September 2006	Online only	Revised for Version 3.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.4 (Release 2008a)
October 2008	Online only	Revised for Version 3.5 (Release 2008b)
March 2009	Online only	Revised for Version 3.6 (Release 2009a)
September 2009	Online only	Revised for Version 4.0 (Release 2009b)
March 2010	Online only	Revised for Version 4.1 (Release 2010a)

1

Getting Started

Product Overview	1-2
Product Description	1-2
Using This Guide	1-3
Expected Background	1-3
Configuration Information	1-5
Setting Up and Configuring	1-7
System Requirements	1-7
Supported Hardware	1-7
Installing and Configuring Software	1-7

2

Targeting C6000 DSP Hardware

Introduction to Targeting	2-2
Overview	2-2
About the Tutorials	2-2
C6000 and Code Composer Studio IDE	2-4
Using Code Composer Studio with Target Support Package Software	2-4
Supported Boards and Simulators	2-5
Typical Hardware Setup for a Development Board	2-5
Targeting Tutorial — Single Rate Application	2-7
Overview	2-7
Building the Audio Reverberation Model	2-8
Adding C6713 DSK Blocks to Your Model	2-9

Configuring Target Support Package Blocks	2-11
Specifying Configuration Parameters for Your Model	2-15
Using the c6000lib Blockset	2-19
Schedulers and Timing	2-23
Timer-Based Versus Asynchronous Interrupt	
Processing	2-23
Synchronous Scheduling	2-24
Asynchronous Scheduling	2-25
Asynchronous Scheduler Examples	2-26
Uses for Asynchronous Scheduling	2-29
Scheduling Considerations	2-33
Model Reference and Target Support Package	
Software	2-35
Overview	2-35
How Model Reference Works	2-35
Using Model Reference with Target Support Package	
Software	2-37
Configuring Targets to Use Model Reference	2-38
Targeting Supported Boards	2-40
Overview	2-40
Typical Targeting Process	2-41
Targeting the C6713 DSP Starter Kit	2-41
Configuring Your C6713DSK	2-43
Confirming Your C6713DSK Installation	2-44
Simulink Models and Targeting	2-45
Creating Your Simulink Model for Targeting	2-45
Blocks to Avoid in Your Models	2-46
Targeting Tutorial II — A More Complex Application ..	2-48
Overview	2-48
Working and Build folders	2-49
Setting Simulation Program Parameters	2-50
Selecting the Target Configuration	2-51
Building and Running the Program	2-55
Contents of the Build folder	2-56

Targeting Your C6713 DSK and Other Hardware	2-58
Overview	2-58
Configuring Your C6713 DSK	2-59
Confirming Your C6713 DSK Installation	2-59
Running Models on Your C6713 DSK	2-60
 Creating Code Composer Studio Projects Without	
Building	2-63
Introduction	2-63
Creating Projects in CCS IDE Without Loading Files to Your Target	2-63
 Targeting Custom Hardware	2-65
Overview	2-65
Typical Targeting Process	2-67
Targeting a Custom Target	2-69
Sections Pane	2-77
To Create Memory Maps for Targets	2-83
 Using Target Support Package Software with Real-Time	
Workshop® Embedded Coder Software	2-84
Introduction	2-84
To Use the Real-Time Workshop® Embedded Coder Target File	2-84

Targeting with DSP/BIOS Options

3

Introducing DSP/BIOS	3-2
 DSP/BIOS and Targeting Your C6000 DSP	3-4
Introduction	3-4
DSP/BIOS Configuration File	3-5
Memory Mapping	3-6
Hardware Interrupt Vector Table	3-6
Linker Command File	3-6
 Code Generation with DSP/BIOS	3-7

Overview	3-7
Generated Code Without and With DSP/BIOS	3-7
Profiling Generated Code	3-11
Overview	3-11
Profiling Subsystems	3-12
Details About Timing and Profiling	3-13
Profiling Multitasking Systems	3-14
The Profiling Report	3-16
Interrupts and Profiling	3-17
Reading Your Profile Report	3-18
Definitions of Report Entries	3-19
Profiling Your Generated Code	3-21
To Enable Profiling for Your Generated Code	3-22
To Create Atomic Subsystems for Profiling	3-22
Using DSP/BIOS with Your Target Application	3-25
Enabling DSP/BIOS When You Generate Code	3-25
Generating Code for Any C64x+ Processor or Board ..	3-27
Example: Creating a Custom Target Preferences Block for OMAP-L138/C6748 EVM	3-29

Using the C62x and C64x DSP Libraries

4

About the C62x and C64x DSP Libraries	4-2
C62x DSP Library	4-2
C64x DSP Library	4-3
Supported Platforms	4-3
Characteristics Common to C62x and C64x Library Blocks	4-4
Fixed-Point Numbers	4-5
Notation	4-5
Signed Fixed-Point Numbers	4-6
Q Format Notation	4-6

Building Models	4-10
Overview	4-10
Converting Data Types	4-10
Using Sources and Sinks	4-11
Choosing Blocks to Optimize Code	4-11

Configuring Timing Parameters for CAN Blocks

5

Setting Timing Parameters	5-2
Accessing the Timing Parameters	5-2
Determining Timing Parameter Values	5-3
CAN Bit Timing Example	5-4

Block Reference

6

AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)	6-2
C6416 DSK (c6416dsklib)	6-3
C6455 EVM (c6455evmlib)	6-4
C6713 DSK (c6713dsklib)	6-4
C6747 EVM (c6747evmlib)	6-5
TCI6428 DSK (tci6428dsklib)	6-5
C6727 PADK (c6727padklib)	6-5
Custom C6000 (c6000customlib)	6-6

Custom C64x+ (c64xpcustomlib)	6-6
CAN Message Handling Blocks (canmsglib)	6-7
DM642 EVM (dm642evmlib)	6-7
DM6437 EVM (dm6437evmlib)	6-7
DM648 EVM (dm648evmlib)	6-9
DSP/BIOS (dspbioslib)	6-9
C62x DSP Library (tic62dsplib)	6-9
Conversions	6-10
Filters	6-10
Math and Matrices	6-11
Transforms	6-11
C64x DSP Library (tic64dsplib)	6-12
Conversions	6-12
Filters	6-12
Math and Matrices	6-13
Transforms	6-14
Scheduling (c6000dspcorelib)	6-14
Target Communication (targetcommplib)	6-14
Target Preferences (c6000tgtppreflib)	6-15

Hardware Issues

A

Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP	A-2
Requirements for the DM642 EVM	A-3
Identifying Your DM642 EVM Board Version	A-3
Installing Third-party Software	A-3
Configuring the Target Preferences Block for Your DM642 EVM	A-4
Configuring the DM642 EVM Video ADC Block	A-5
Installing and Configuring the Avnet Board Support Library	A-6
Preface	A-6
Installing the Avnet Board Support Library	A-6
Setting the MATLAB Environment	A-6
For Spectrum Digital DM6437EVM Users	A-7
Verifying Your Installation	A-8
Continuing Issues with Target Support Package Software	A-9
Setting the Clock Speed on the C6713 DSK	A-9
Simulink Stop Block Works Differently When Not Using DSP/BIOS Features	A-10
Installing Third-Party Target Support Packages	A-10

Index

Getting Started

- “Product Overview” on page 1-2
- “Using This Guide” on page 1-3
- “Configuration Information” on page 1-5
- “Setting Up and Configuring” on page 1-7

Product Overview

Product Description

Use Target Support Package™ to deploy generated code for real-time execution on embedded microprocessors, microcontrollers, and DSPs. Using Target Support Package, you can integrate peripheral devices with the algorithms created using Embedded MATLAB™, Simulink®, and Stateflow®. You can deploy the resulting executable onto embedded hardware for on-target rapid prototyping, real-time performance analysis, and field production.

Using This Guide

Expected Background

This document introduces you to using Target Support Package software with Real-Time Workshop® software to develop digital signal processing applications for the Texas Instruments™ C6000™ family of DSP development hardware, such as the TI TMS320C6713 DSP Starter Kit. To get the most out of this manual, you should be familiar with MATLAB® software and its associated programs, such as Signal Processing Blockset™ software and Simulink® software. We do not discuss details of digital signal processor operations and applications, except to introduce concepts related to using specific targets. For more information about digital signal processing, you may find one or more of the following books helpful:

- McClellan, J. H., R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*, Prentice Hall, 1998.
- Lapsley, P., J. Bier, A. Sholam, and E. A. Lee, *DSP Processor Fundamentals Architectures and Features*, IEEE® Press, 1997.
- Oppenheim, A.V., R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.
- Mitra, S. K., *Digital Signal Processing — A Computer-Based Approach*, The McGraw-Hill Companies, Inc, 1998.
- Steiglitz, K, *A Digital Signal Processing Primer*, Addison-Wesley Publishing Company, 1996.

Refer to the documentation for your TI boards for information about setting them up and using them.

If You Are a New User

New users should read Chapter 1, “Getting Started”, which introduces the Target Support Package environment—the required software and hardware, installation requirements, and the board configuration settings that you need. You will find descriptions of the blocks associated with the targeting software, and an introduction to the range of digital signal processing applications of which the target support package is capable.

If You Are an Experienced User

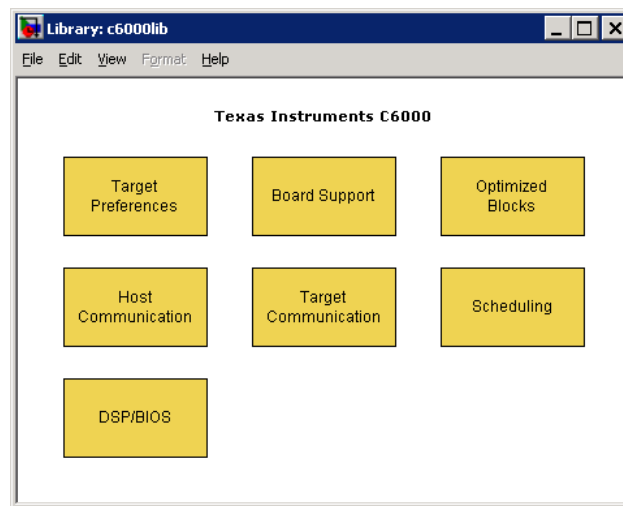
All users should read Chapter 2, “Targeting C6000 DSP Hardware” for information and examples about using the new blocks and build software to target your C6713 DSK. Two example models introduce the targeting software and build files, and give you an idea of the range of applications supported by the target support package. For C6713 DSK users, refer to “Configuring Your C6713 DSK” on page 2-59 for more information about installing and using your C6713 DSK.

Configuration Information

To determine whether Target Support Package software is installed on your system, type this command at the MATLAB prompt.

```
c6000lib
```

Entering that command displays the following C6000 block library:



If you do not see the listed libraries, or MATLAB software does not recognize the command, install the target support package. Without the software, you cannot use Simulink and Real-Time Workshop software to develop applications targeted to the TI boards.

Note For up-to-date system requirements, visit <http://www.mathworks.com/products/target-package/requirements.html> or go to <http://www.mathworks.com> and select Products & Services, Product List, Target Support Package, and System Requirements.

To verify that the CCS IDE is installed on your machine, enter

```
ccsboardinfo
```

at the MATLAB command line. With the CCS IDE installed and configured, the command line returns information about the boards that CCS IDE recognizes on your machine, in a form similar to the following listing.

Board		Processor	
Num	Name	Num	Name
		Type	
-----		---	
0	C6x11 DSK (Texas Instruments)	0	CPU
		TMS320C6x1x	

If the command line does not return information about any boards, revisit your CCS IDE installation and setup in your CCS IDE documentation.

As a final test, launch CCS IDE to ensure that it starts up successfully. For the target support package to operate with this application, the CCS IDE must be able to run on its own.

Setting Up and Configuring

In this section...
“System Requirements” on page 1-7
“Supported Hardware” on page 1-7
“Installing and Configuring Software” on page 1-7

System Requirements

For detailed information about the software and hardware required to use Target Support Package software, refer to the Target Support Package system requirements areas on the MathWorks Web site:

- Requirements for Target Support Package:
www.mathworks.com/products/target-package/requirements.html
- Requirements for use with TI's C6000:
www.mathworks.com/products/target-package/ti-adaptor/

Supported Hardware

For a list of supported hardware, visit
<http://www.mathworks.com/products/target-package/supportedio.html>.

Installing and Configuring Software

Consult the “System Requirements” on page 1-7 for Target Support Package . Only use *supported versions* of the software listed under “Third-Party Target Support Package Requirements”. Uninstall unsupported versions *before* installing supported versions. Doing so prevents errors that occur when the Windows Environment Variables points to the unsupported versions.

The System Requirements describe where you can obtain the additional third-party software, and when available, provide links for downloading that software.

Install the software (only the supported versions!) in the following order:

- 1** If needed, install the required and optional MathWorks software. (The software license you purchase determines which products are available.)
- 2** If needed, install TI Code Composer Studio™ (CCS).
- 3** Install TI Service Release for CCS.
- 4** Install the TI Code Generation Tools for your processor.
- 5** If you are using a Spectrum Digital board, download and install the matching Spectrum Digital Driver.
- 6** Install additional board-specific packages in the order in which they appear on the System Requirements web page.

Configure CCS as follows:

- 1** In CCS, open **Help > About > Component Manager > Build tools**
- 2** Open each target processor you will be using and enable the supported version of **Code Generation Tools**.
- 3** Open **Help > About > Component Manager > Build Tools > Target Content (DSP/BIOS)** .
- 4** Open each target processor you will be using and enable the supported version of **Texas Instruments DSP/BIOS**.
- 5** In Component Manager, select Save the changes. Then exit and restart CCS.
- 6** If you have a Spectrum Digital DM6437EVM board and/or an Avnet S3ADSP DM6437 board, refer to “Installing and Configuring the Avnet Board Support Library” on page A-6.
- 7** Verify the installation by repeating the instructions in “Configuration Information” on page 1-5.

Targeting C6000 DSP Hardware

- “Introduction to Targeting” on page 2-2
- “C6000 and Code Composer Studio IDE” on page 2-4
- “Targeting Tutorial — Single Rate Application” on page 2-7
- “Using the c6000lib Blockset” on page 2-19
- “Schedulers and Timing” on page 2-23
- “Model Reference and Target Support Package Software” on page 2-35
- “Targeting Supported Boards” on page 2-40
- “Simulink Models and Targeting” on page 2-45
- “Targeting Tutorial II — A More Complex Application” on page 2-48
- “Targeting Your C6713 DSK and Other Hardware” on page 2-58
- “Creating Code Composer Studio Projects Without Building” on page 2-63
- “Targeting Custom Hardware” on page 2-65
- “Using Target Support Package Software with Real-Time Workshop® Embedded Coder Software” on page 2-84

Introduction to Targeting

In this section...
“Overview” on page 2-2
“About the Tutorials” on page 2-2

Overview

The Target Support Package software lets you use Real-Time Workshop software to generate a C language real-time implementation of your Simulink model. You can compile, link, download, and execute the generated code on the C6713 DSP Starter Kit (DSK). The target support package is ideal for rapid prototyping and developing embedded systems applications for C6713 digital signal processors. The target support package focuses on developing real-time digital signal processing (DSP) applications for C6000 hardware. Additional hardware that we support is listed in Appendix A, “Hardware Issues”.

Although the tutorials in this chapter focus on the C6713 DSK, the techniques and processes apply to any supported hardware, with minor adjustments for the processor involved.

This chapter describes how to use the target support package to create and execute applications on Texas Instruments C6000 development boards. To use the targeting software, you should be familiar with using Simulink software to create models and with the basic concepts of Real-Time Workshop software automatic code generation. To read more about Real-Time Workshop software, refer to your Real-Time Workshop documentation.

About the Tutorials

In most cases, this chapter deals with the C6713 DSK targets. Fortunately, all members of the C6000 family of processors that we support work in a manner similar to the C6713 DSK. While you review the contents of this chapter, and follow the tutorials, recall that the concepts and techniques or development processes apply, with a few adjustments, to all supported C6000 processors and boards.

Later sections discuss the Real-Time Workshop® Embedded Coder™ software and targeting custom hardware.

Tip To make your figure easier to read, use easily distinguishable colors and line styles.

C6000 and Code Composer Studio IDE

In this section...

“Using Code Composer Studio with Target Support Package Software” on page 2-4

“Supported Boards and Simulators” on page 2-5

“Typical Hardware Setup for a Development Board” on page 2-5

Using Code Composer Studio with Target Support Package Software

Texas Instruments (TI) markets a complete set of software tools to use when you develop applications for your C6000 hardware boards. This section provides a brief example of how Target Support Package software uses Code Composer Studio (CCS) Integrated Development Environment (IDE) with the Real-Time Workshop software and the c6000lib blockset.

Executing code generated from Real-Time Workshop software on a particular target in real time requires that Real-Time Workshop software generate target code that is tailored to the specific hardware target. Target-specific code includes I/O device drivers and an interrupt service routine (ISR). Since these device drivers and ISRs are specific to particular hardware targets, you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, TI C6000 uses the MATLAB links in Embedded IDE Link™ software to invoke the code building process within the CCS IDE. After you download your executable to your target and run it, the code runs wholly on the target; you can access the running process only from the CCS IDE debugging tools. Otherwise the running process is not accessible.

Used in combination with your target support package and Real-Time Workshop software, TI products provide an integrated development environment that, once installed, needs no additional coding.

Supported Boards and Simulators

Using the C6000 target provided by Target Support Package software, you can generate code to run on a range of boards, both evaluation modules and DSP starter kits.

Refer to Appendix A, “Hardware Issues” for the latest information about the hardware supported by the target support package.

About Simulators

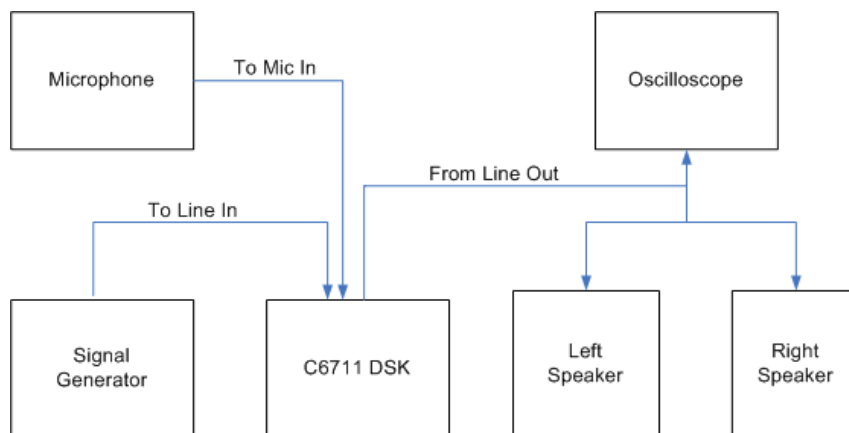
The CCS IDE offers simulators for the C6000 processors in the CCS IDE Setup utility. Much of your model and algorithm development efforts work with the simulators, such as code generation. And, since the target support package provides a software-based scheduler, your models and generated code run on the simulators just as they do on your hardware. For more information about the simulators in CCS IDE, refer to your CCS online help system.

When you set up a simulator, match the processor on your target exactly to simulate your target hardware. For example, to target a C6713DSK board, your simulator must contain a C6713 processor, not just a C6xxx simulator. Simulators must match the target processor because the codecs on the board are not the same and the simulator needs to identify the correct codec. Correctly matching your simulator to your hardware ensures that the memory maps and registers match those of your intended target signal processor.

In general, use the device cycle accurate simulators provided by CCS Setup to simulate your processor.

Typical Hardware Setup for a Development Board

The following block diagram represents typical inputs and output for a C6713 DSK development board.



After installing a supported development board, start MATLAB software. At the command prompt, type `c6000lib`. This opens a Simulink blockset named `c6000lib` that includes libraries that contain blocks predefined for C6000 input and output devices.

The board-based block library for the C6713 DSK contains these blocks:

- ADC block
- DAC block
- DIP Switch block (optional, refer to the reference page for the DIP Switch block for your target)
- LED block
- Reset block

Blocks from these libraries are associated with your boards and hardware. As needed, add the devices to your model. If you choose not to include either an ADC or DAC block in your model (they are available in the target specific libraries), the target support package provides a timer that produces the interrupts required for timing and running your model, either on your hardware target or on a simulator.

Targeting Tutorial — Single Rate Application

In this section...

“Overview” on page 2-7

“Building the Audio Reverberation Model” on page 2-8

“Adding C6713 DSK Blocks to Your Model” on page 2-9

“Configuring Target Support Package Blocks” on page 2-11

“Specifying Configuration Parameters for Your Model” on page 2-15

Overview

In this tutorial you create and build a model that simulates audio reverberation applied to an input signal. Reverberation is similar to the echo effect you can hear when you shout across an open valley or canyon, or in a large empty room.

You can choose to create the Simulink model for this tutorial from blocks in Signal Processing Blockset software and Simulink block libraries, or you can find the model in Target Support Package demos. For this example, you see the model as it appears in the demonstration program. The demonstration model name is `c6713dskafxr.mdl` as shown in the next figure. Open this model by entering `c6713dskafxr` at the MATLAB prompt.

To run this model you need a microphone connected to the **Mic In** connector on your C6713 DSK, and speakers and an oscilloscope connected to the **Line Out** connector on your C6713 DSK. To test the model, speak into the microphone and listen to the output from the speakers. You can observe the output on the oscilloscope as well.

To download and run your model on your C6713 DSK, complete the following tasks:

- 1 Use Simulink blocks, Signal Processing Blockset software blocks, and blocks from other blocksets to create your model application.
- 2 Add Target Support Package blocks that let your signal sources and output devices communicate with your C6713 DSK—the C6713 DSK ADC and

C6713 DSK DAC blocks that you find in Target Support Package c6000lib blockset.

- 3 Add the C6713DSK target preferences block from the Target Preferences library to your model. Verify and set the block parameters for your hardware. In most cases, the default settings work fine.

If you are using a C6713 simulator target, select **Simulator** on the **Board info** pane of the target preferences block.

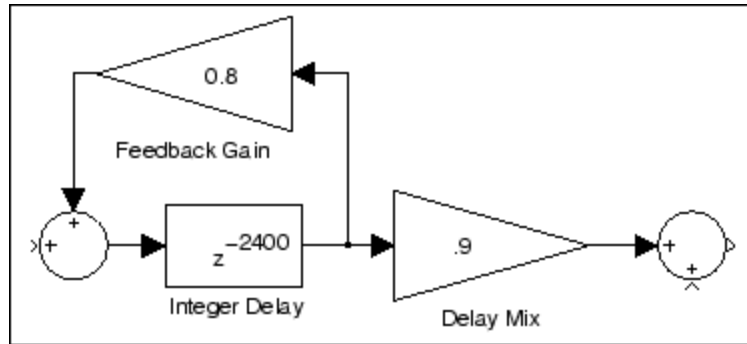
- 4 Set the configuration parameters for your model, including
 - Solver parameters such as simulation start and solver options
 - Real-Time Workshop software options such as target configuration and target compiler selection
- 5 Build your model to the selected target.
- 6 Test your model running on the target by changing the input to the target and observing the output from the target.

Your target for this tutorial is your C6713 DSK installed on your PC. Be sure to configure and test your board as directed in “Configuring Your C6713DSK” on page 2-43 in this guide before continuing this tutorial.

Building the Audio Reverberation Model

To build the model for audio reverberation, follow these steps:

- 1 Start Simulink.
- 2 Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.
- 3 Use Simulink blocks and Signal Processing Blockset software blocks to create the following model.



Look for the Integer Delay block in the Signal Operations library of the Signal Processing Blockset software. You do not need to add the input and output signal lines at this time. When you add the C6713 DSK blocks in the next section, you add the input and output to the sum blocks.

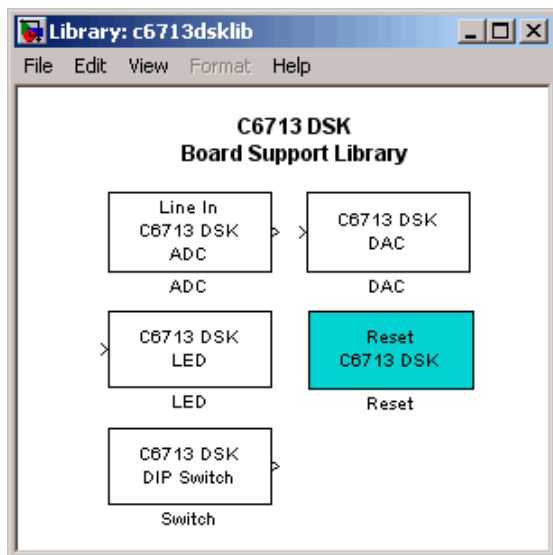
4 Save your model with a suitable name before continuing.

Adding C6713 DSK Blocks to Your Model

So that you can send signals to your C6713 DSK and get signals back from the board, Target Support Package software includes a block library containing five blocks designed to work with the codec on your C6713 DSK:

- Input block (C6713 DSK ADC)
- Output block (C6713 DSK DAC)
- Light emitting diode block (C6713 DSK LED)
- Software reset block (Reset C6713 DSK)
- DIP switch block (C6713 DSK DIP Switch)

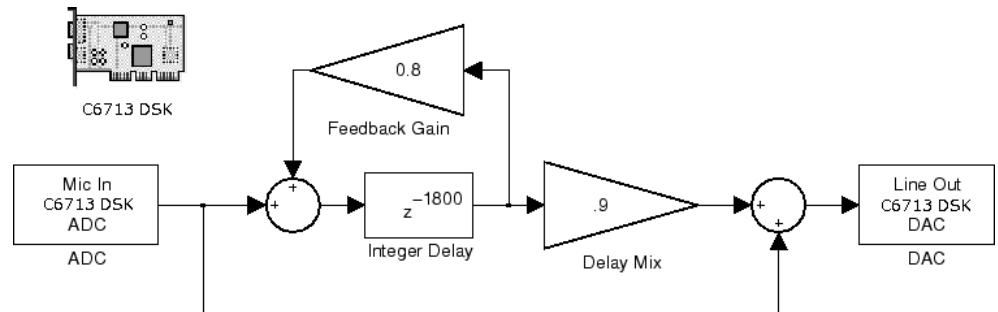
Entering `c6713dsklib` at the MATLAB prompt opens this window showing the library blocks. This block library is included in Target Support Package `c6000lib` blockset in the Simulink Library browser.



The C6713 DSK ADC and C6713 DSK DAC blocks generate code that configures the codec on your C6713 DSK to accept input signals from the input connectors on the board, and send the model output to the output connector on the board. Essentially, the C6713 DSK ADC and C6713 DSK DAC blocks add driver software that controls the behavior of the codec for your model.

To add C6713 DSK target blocks to your model, follow these steps:

- 1** Double-click Target Support Package software in the Simulink Library browser to open the c6000lib blockset.
- 2** Click the library C6713 DSK Board Support to see the blocks available for your C6713 DSK.
- 3** Drag and drop C6713 DSK ADC and C6713 DSK DAC blocks to your model as shown in the figure.



- 4 Connect new signal lines as shown in the figure.
- 5 Finally, from the TI C6000 Target Preferences block library, add the C6713DSK Target Preferences block to the model. Notice that it is not connected to any other block in the model.

Configuring Target Support Package Blocks

To configure Target Support Package blocks in your model, follow these steps:

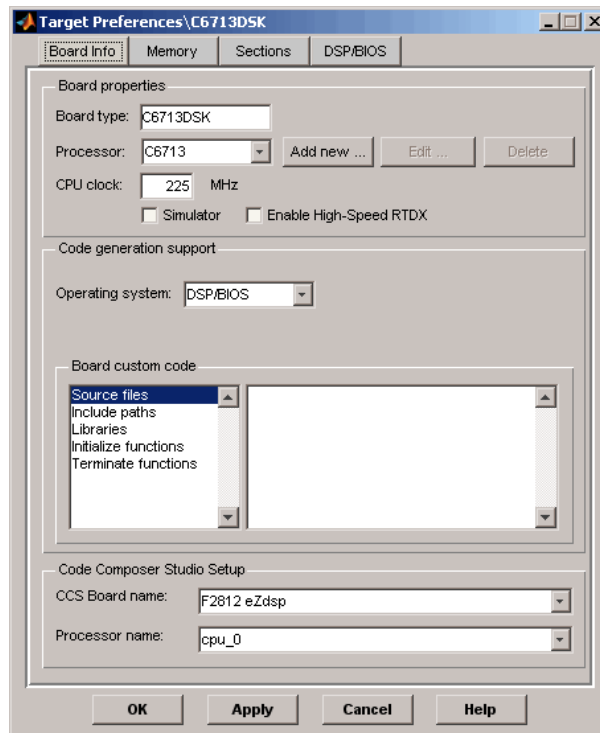
- 1 Click the C6713 DSK ADC block to select it.
- 2 Select **Block Parameters** from the Simulink **Edit** menu.
- 3 Set the following parameters for the block:
 - Clear the **Stereo** check box.
 - Select the **+20 dB mic gain boost** check box.

From the list, set **Sample rate** to 8000.

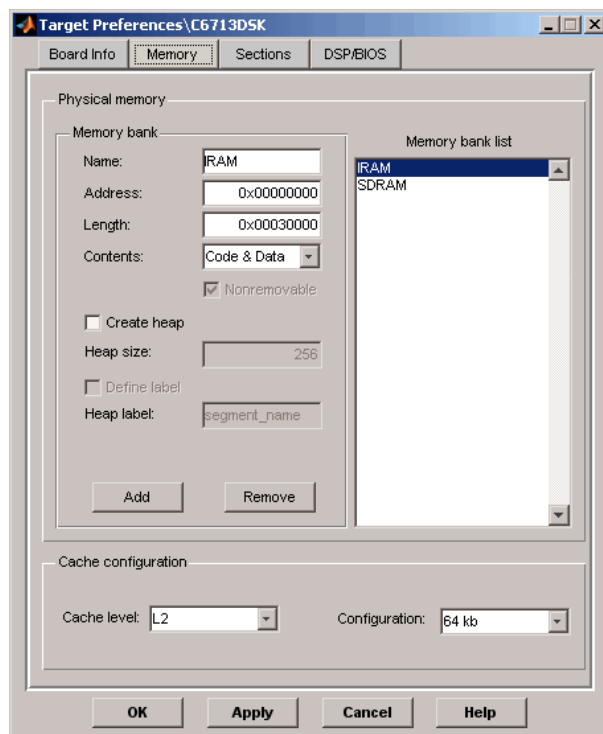
 - Set **Codec data format** to 16-bit linear.
 - For **Output data type**, select Double from the list.
 - Set **Scaling** to Normalize.
 - Set **Source gain** to 0.0.
 - Enter 64 for **Samples per frame**.

Include a signal path directly from the input to the output so you can display both the input signal and the modified output signal on the oscilloscope for comparison.

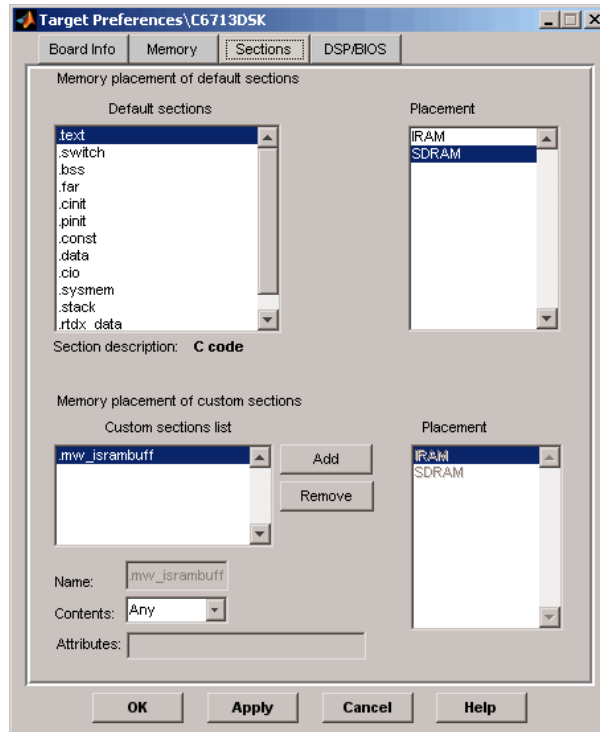
- 4** For **C6713 DSK ADC source**, select **Mic In**.
- 5** Click **OK** to close the C6713 DSK ADC dialog box.
- 6** Now set the options for the C6713 DSK DAC block.
 - Set **Codec data format** to **16-bit linear**.
 - Set **Scaling** to **Normalize**.
 - For **DAC attenuation**, enter **0.0**.
 - Set **Overflow mode** to **Saturate**.
- 7** Click **OK** to close the dialog box.
- 8** Click the C6713DSK Target Preferences block.
- 9** Select **Block Parameters** from the Simulink **Edit** menu.
- 10** Verify the parameter settings for the C6713 DSK target. The figures below show the proper values.



Board info Settings



Memory Settings



Section Settings

You have completed the model. Now configure the Real-Time Workshop software options to build and download your new model to your C6713 DSK.

Specifying Configuration Parameters for Your Model

The following sections describe how to build and run real-time digital signal processing models on your C6713 DSK. Running a model on the target starts with configuring and building your model from the Configuration Parameters dialog box in Simulink software.

Setting Simulink Configuration Parameters

After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the configuration parameters for the model:

- 1 Open the Configuration Parameters dialog box and set the appropriate options on the **Solver** category for your model and for Target Support Package software.
 - Set **Start time** to 0.0 and **Stop time** to `inf` (model runs without stopping). Generated code does not honor this setting if you set a stop time. Set this to `inf` for completeness.
 - Under **Solver options**, select the Fixed-step and Discrete settings from the lists
 - Set the **Fixed step size** to Auto and the **Tasking Mode** to Single Tasking

Note Generated code does not honor Simulink stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, you must put a Stop Simulation block in your model.

Ignore the Data Import/Export, Diagnostics, and Optimization categories in the Configuration Parameters dialog box. The default settings are correct for your new model.

Setting Real-Time Workshop Target Build Options

You can configure Real-Time Workshop software to generate and build code that is appropriate for your hardware target. Follow these steps to set the Real-Time Workshop options to target your C6713 DSK:

- 1 Open the Configuration Parameters dialog box by entering **Ctrl+E** or by selecting the **Simulation** menu item and then **Configuration Parameters**.
- 2 From the **Select** tree, choose **Real-Time Workshop**.
- 3 Verify that the system target file is set to `ccslink_grt.tlc`. If needed, click **Browse** and select `ccslink_grt.tlc`.
- 4 From the **Select** tree, choose **Embedded IDE Link**.

- 5** Among the **Runtime Options**, set **Build action** to **Build_and_execute**, and set **Interrupt overrun notification method** to **Print_message**.
- 6** Among the **Project Options**, keep the default settings.
- 7** Among the **Code Generation** options, clear **Profile real-time execution**.
- 8** Among the **Link Automation** options, verify that **Export IDE link handle to base workspace** is selected and that **IDE link handle name** has a name (e.g., **CCS_Obj**).
- 9** From the **Select** tree, choose **Hardware Implementation**.
- 10** Verify that **Byte ordering** is **Little endian**.

When you have completed these steps, you have configured the Real-Time Workshop options for the C6713 DSK target. Some Real-Time Workshop categories on the **Select** tree, such as **Comments**, **Symbols**, and **Optimization**, do not require configuration. The default values for the options in these categories are already correct for your new model. For other models, you may want to set the options in these categories to provide information during the build and to run TLC debugging when you generate code.

Building and Executing Your Model on Your C6713 DSK

After you set the configuration parameters and configure Real-Time Workshop software to create the files you need, you direct Real-Time Workshop software to build, download, and run your model executable on your target:

- 1** Change the category to Real-Time Workshop on the Configuration Parameters dialog box.
- 2** Clear **Generate code only** and click **Build** to generate and build an executable file targeted to your C6713 DSK.

When you click **Build** with **Build_and_execute** selected for **Build action**, the automatic build process creates an executable file that can be run by the C6713 DSP on your C6713 DSK, and then downloads the executable file to the target and runs the file.

- 3** To stop model execution, click the **Reset C6713 DSK** block or use the **Halt** option in CCS IDE. You could type halt from the MATLAB command prompt as well.

Testing Your Audio Reverb Model

With your model running on your C6713 DSK, speak into the microphone you connected to the board. The model should generate a reverberation effect out of the speakers, delaying and echoing the words you speak into the mike. If you built the model yourself, rather than using the supplied model `c6713dskafxr`, try running the demonstration model to compare the results.

Using the c6000lib Blockset

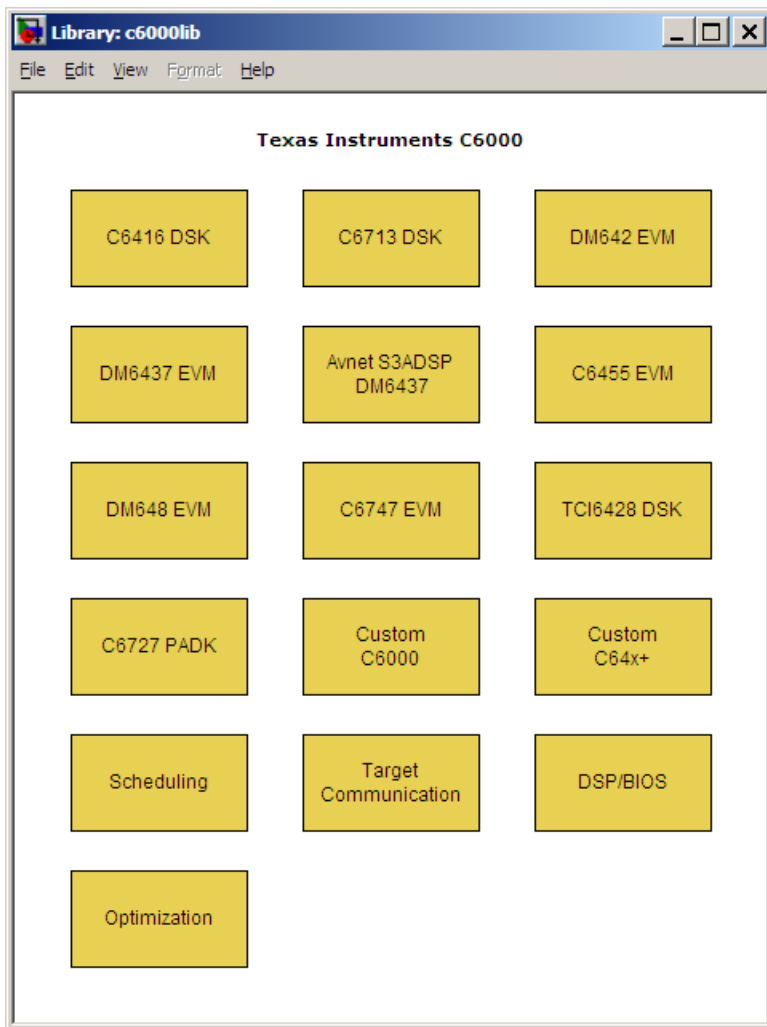
The c6000lib blockset contains the block libraries described in the following table.

Library	Descriptions
Texas Instruments C6000 (c6000lib)	Contains: Target Preferences (c6000tgtppreflib), Board Support (c6000boardsupportlib), Optimized Blocks (c6000optimizedblks), Host Communication (hostcommplib), Target Communication (targetcommplib), Scheduling (c6000dspcorelib), DSP/BIOS (dspbioslib)
“C6416 DSK (c6416dsklib)” on page 6-3	Configures C6416 DSK peripherals.
“C6713 DSK (c6713dsklib)” on page 6-4	Configures C6713 DSK peripherals.
“DM642 EVM (dm642evmlib)” on page 6-7	Configures DM642 EVM peripherals and video capture.
“DM6437 EVM (dm6437evmlib)” on page 6-7	Configures DM6437 EVM peripherals and video capture.
“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2	Configures Avnet Spartan-3A DSP DaVinci Evaluation Platform Board peripherals and video capture.
“C6455 EVM (c6455evmlib)” on page 6-4	Configures SRIO peripherals on the C6455 EVM.
“DM648 EVM (dm648evmlib)” on page 6-9	Configures DM648 EVM peripherals and video capture.
“C6747 EVM (c6747evmlib)” on page 6-5	Configures C6747 EVM peripherals.
“TCI6428 DSK (tci6428dsklib)” on page 6-5	Configures TCI6428 DSK peripherals.
“C6727 PADK (c6727padklib)” on page 6-5	Configures C6727 PADK peripherals.
“Custom C6000 (c6000customlib)” on page 6-6	Configures C6000 peripherals.

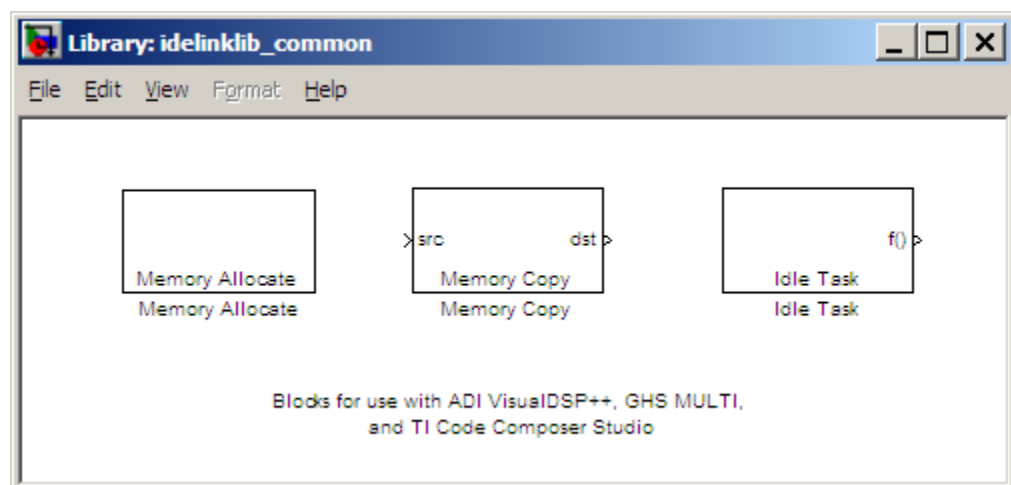
Library	Descriptions
“Custom C64x+ (c64xpcustomlib)” on page 6-6	Configures C64x+ peripherals.
“Scheduling (c6000dspcorelib)” on page 6-14	Manages memory and task scheduling on C6000-based targets.
“Target Communication (targetcommplib)” on page 6-14	Provides UDP and TCP/IP communications capabilities. Includes byte manipulation blocks.
“DSP/BIOS (dspbioslib)” on page 6-9	Provides scheduling management using DSP/BIOS.
Optimized Blocks (c6000optimizedblks)	Contains C62x DSP (tic62dsplib) and C64x DSP Library (tic64dsplib)
“C62x DSP Library (tic62dsplib)” on page 6-9	Provides C62x-optimized algorithms such as filtering and matrix manipulation.
“C64x DSP Library (tic64dsplib)” on page 6-12	Provides C64x-optimized algorithms such as filtering and matrix manipulation.
“Host Communication (hostcommplib)”	Includes blocks for host-side UDP communications and byte manipulation.

Similarities in the C6000 boards result in the ADC, DAC, DIP Switch, LED, and Reset blocks for the C6000-based boards being almost identical. Each section about a block, such as the ADC block, presents all possible options for the block, noting when an option applies only to a board-specific version of the ADC block.

The `c6000lib` blockset, below, displays all of the block libraries in the Target Support Package software. For a comprehensive list of the blocks in each library, consult the Chapter 6, “Block Reference” topic.



In addition to the `c6000lib`, you can also use blocks from the `idelinklib_common` library.



Schedulers and Timing

In this section...

“Timer-Based Versus Asynchronous Interrupt Processing” on page 2-23

“Synchronous Scheduling” on page 2-24

“Asynchronous Scheduling” on page 2-25

“Asynchronous Scheduler Examples” on page 2-26

“Uses for Asynchronous Scheduling” on page 2-29

“Scheduling Considerations” on page 2-33

Timer-Based Versus Asynchronous Interrupt Processing

Code generated for periodic tasks, both single- and multitasking, runs out of the context of a timer interrupt. The generated code that represents model blocks for periodic tasks runs periodically, clocked by the periodic interrupt whose period is equal to the base sample time of the model. This description of scheduling and timing applies both to generated code operation that incorporates DSP/BIOS real-time operating system (RTOS) and basic code generation mode where DSP/BIOS RTOS is not included.

Note In timer-based models, the timer counts through one full base-sample-time before it creates an interrupt. When the model is finally executed, it is for time 0.

This execution scheduling scheme is not flexible enough for some systems, such as control and communication systems that must respond to asynchronous events in real time. Such systems may need to handle a variety of hardware interrupts in an asynchronous, or aperiodic, fashion.

When you plan your project or algorithm, select your scheduling technique based on your application needs.

- If your application processes hardware interrupts asynchronously, add the appropriate asynchronous scheduling blocks from the Target Support Package library to your model, listed here.

Blocks in the DSP/BIOS (dspbioslib) library

- Hardware Interrupt — Create interrupt service routine on C6000 hardware target.
- Task — Create task that runs as separate DSP/BIOS thread.
- Triggered Task — Create asynchronously triggered task.

Blocks in the Scheduling (c6000dspcorelib) library

- Block Processing — Repeat user-specified operation on submatrices of input matrix, using internal memory of DSP for increased efficiency.
- CPU timer — Generate interrupt service routine.
- EDMA — Configure EDMA Controller on C6000 processor.

Blocks in the Embedded IDE Link library for Texas Instruments Code Composer Studio (idelinklib_ticcs)

- C6000 Hardware Interrupt — Generate interrupt service routine. Same as the DSP/BIOS interrupt block.

Blocks in the Embedded IDE Link Common library (idelinklib_common)

- Idle Task — Create free-running background task
- If your application does not service asynchronous interrupts, your model should include only the algorithm and device driver blocks that specify the periodic sample times. Generating code from a model like this automatically enables and manages a timer interrupt. The periodic timer interrupt clocks the entire model.

Synchronous Scheduling

For code that runs synchronously in the context of the timer interrupt, each iteration of the model runs after an interrupt has been posted and serviced by an interrupt service routine (ISR). The code generated for Target Support Package software uses Timer 1 in DSP/BIOS mode and bare-board mode. Timer 1 is configured so that the base rate sample time for the coded process corresponds to the interrupt rate. The target support package calculates and configures the timer period to ensure the desired sample rate.

The minimum achievable base rate sample time depends on the algorithm complexity and the CPU clock speed. The maximum value depends on the maximum timer period value and the CPU clock speed.

If all the blocks in the model inherit their sample time value, and no sample time is defined explicitly, Simulink assigns a default sample time of 0.2 second.

Note In timer-based models, the timer counts through one full base-sample-time before it creates an interrupt. When the model is finally executed, it is for time 0.

Asynchronous Scheduling

Target Support Package software facilitates modeling and automatically generating code for asynchronous systems by using the following scheduling blocks:

- C5000/C6000 Hardware Interrupt and Idle Task blocks for bare-board code generation mode
- DSP/BIOS Hardware Interrupt, DSP/BIOS Task, and DSP/BIOS Triggered Task blocks for DSP/BIOS code generation mode

C6000 Hardware Interrupt blocks enable selected hardware interrupts for the TI TMS320C6000 DSP, generate corresponding ISRs, and connect them to the corresponding interrupt service vector table entries.

When you connect the output of the C6000 Hardware Interrupt block to the control input of a function-call subsystem, the generated subsystem code is called from the ISRs each time the interrupt is raised.

The C6000 Idle Task block specifies one or more functions to execute as background tasks in the code generated for the model. The functions are created from the function-call subsystems to which the Idle Task block is connected.

The DSP/BIOS Hardware Interrupt block (in DSP/BIOS code generation mode) has the same functionality as the bare-board C6000 Hardware

Interrupt block. The configuration and low-level handling of the hardware interrupts is implemented through DSP/BIOS using DSP/BIOS Hardware Interrupt module and DSP/BIOS dispatcher.

DSP/BIOS Task blocks (DSP/BIOS code generation mode) spawn free-running tasks as separate DSP/BIOS threads. The spawned task runs the function-call subsystem connected to its output. Blocks in the subsystem may use various conditions and techniques to control sharing sources with other tasks.

DSP/BIOS Triggered Task blocks (in DSP/BIOS code generation mode) spawn semaphore-controlled tasks as separate DSP/BIOS threads. The semaphore that enables execution of a single instance of the task is posted by an ISR that is created by a DSP/BIOS Hardware Interrupt block. This block is connected to a DSP/BIOS Triggered Task block.

Asynchronous Scheduler Examples

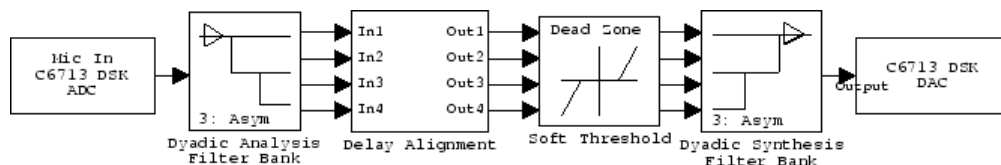
Now you can use an asynchronous (real-time) scheduler for your target application. Earlier versions of Target Support Package software used a synchronous CPU timer interrupt-driven scheduler. With the asynchronous scheduler you can define interrupts and tasks to occur when you want them to using blocks in the following libraries:

- Core Support library (idelinklib_common)
- DSP/BIOS library (dsplib)

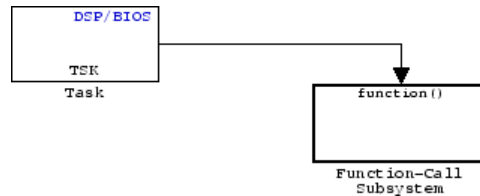
Also, you can schedule multiple tasks for asynchronous execution using those blocks libraries.

The following figures show a model updated to use the asynchronous scheduler rather than the synchronous scheduler.

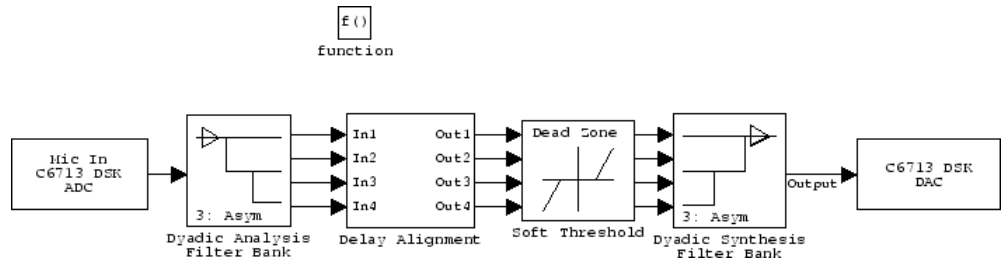
Before



After



Model Inside the Function Call Subsystem Block

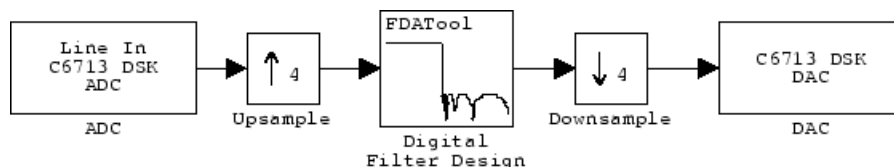


Compatibility Considerations. The V3.0 changes in the real-time scheduler can break some existing multirate models that contain codec blocks such as the ADC and DAC. The models affected contain at least one sample rate that is faster than the codec block rate. You do not run into this problem if all rates in the model are lower than the codec rate.

The new scheduler provides improved control for your processing and improved performance. You should recast all of your models to use the new asynchronous scheduler. To update your models, embed the entire processing algorithm or system in a function-call subsystem driven by a DSP/BIOS Task or Idle Task block from the DSP/BIOS library.

An example of such a model contains a combination of an ADC block and a DAC block, with a processing algorithm between them that executes at the higher rate. If you run code generated for such a model in multitasking or auto solver mode, you might hear occasional audio glitches or your program may overrun. The exact symptom of the problem depends on the run-time overrun action setting in the Embedded IDE Link options.

The following model demonstrates one possible model configuration that can demonstrate the audio problems.



This multirate model uses two interrupts to control real-time execution of the generated code:

- A DMA interrupt to drive the execution of the code for ADC and DAC blocks
- A timer interrupt to drive the execution of the code for the FIR filter at an increased sample rate

In earlier product versions, the generated scheduler constantly synchronized the DMA and timer interrupts to ensure they remained in sync with one another, despite the possible clock drift with interrupts that are recorded by independent clock sources.

With the new real-time scheduler, the product does not synchronize the ADC and timer interrupts.

One interrupt may get out of sync with the other, with the time difference between them (drift) fluctuating with changes in the independent interrupt clocks. When the drift reaches a critical threshold, processing may skip an instance of a lower-priority task.

At that point, the interrupts are back in sync and the process continues. Losing synchronization between the interrupts can corrupt the audio signal or lead to an interrupt overrun.

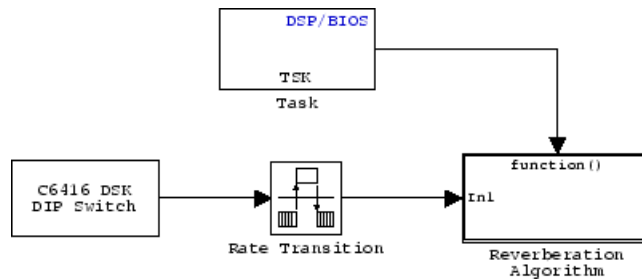
To avoid the audio problems in an existing model that you cannot update to the new scheduler, set the run-time overrun action for the model to either `None` or `Notify_and_continue` to prevent the program from overrunning.

Uses for Asynchronous Scheduling

The following sections present common cases for the scheduling blocks described in the previous sections.

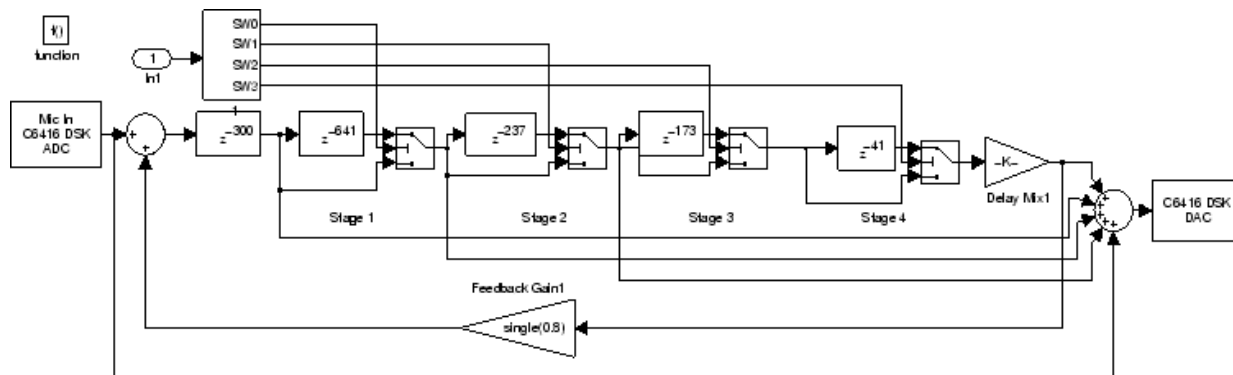
Free-Running DSP/BIOS Task

The following model illustrates a case where a reverberation algorithm runs in the context of a free-running DSP/BIOS task.



Normally, the algorithms in this type of task run in free-running mode, that is, they run repetitively and indefinitely. However, in this function-call subsystem (shown in detail in the following figure), ADC and DAC blocks suspend the execution of the task until the ADC and DAC data is available.

Each instance of the reverberation algorithm is triggered only after the data buffer is available (for both ADC and DAC). An asynchronous ADC/DAC device driver layer separate from the task function manages the triggers condition. This device driver layer uses a direct memory access (DMA) interrupt to signal to the DSP/BIOS task when ADC and DAC data become available for the task function.



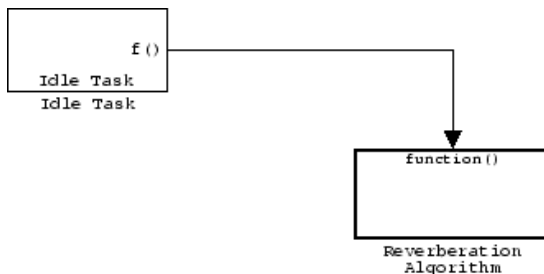
This model also illustrates how synchronous and asynchronous tasks can work together. The code generated for C6416 DSK DIP Switch block runs as a periodic task at the rate of 0.01 s. This is the only periodic task in the model. It runs out of the context of a DSP/BIOS task scheduled via a timer interrupt configured to go off every 0.01 second.

In general, Simulink blocks that specify nonzero sample rates, such as the DIP Switch block, are scheduled by the C6000 synchronous scheduler and executed either from the context of a DSP/BIOS task (if you incorporate DSP/BIOS in your project) or a hardware interrupt (when you do not incorporate DSP/BIOS).

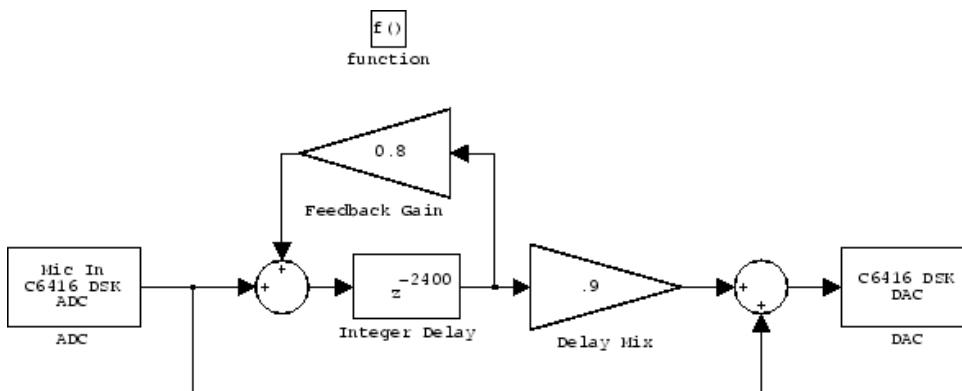
To ensure data integrity, Simulink Rate Transition blocks connect the C6416 DSK DIP Switch block with the reverberation algorithm. This transition is required because the blocks belong to different rate groups. If the synchronous and asynchronous parts of the model do not interact, the Rate Transition blocks are not needed.

Idle Task

The following model illustrates a case where the reverberation algorithm runs in the context of a background task in bare-board code generation mode.

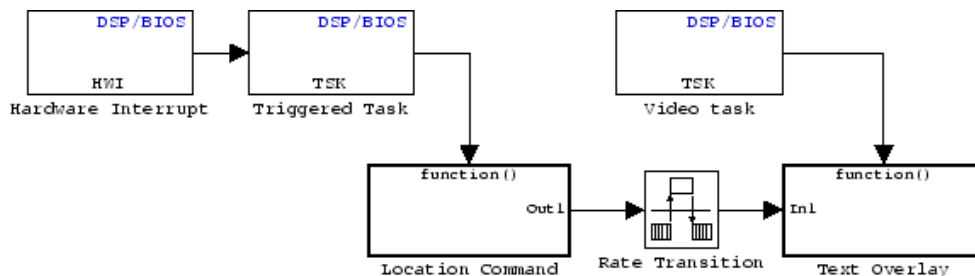


The function generated for this task normally runs in free-running mode—repetitively and indefinitely. However, the ADC and DAC blocks in this subsystem run in blocking mode. As a result, subsystem execution of the reverberation function is the same as the subsystem described for the Free-Running DSP/BIOS Task. It is data driven via a background DMA interrupt-controlled ISR, shown in the following figure.



Hardware Interrupt Triggered DSP/BIOS Task

The next model illustrates a case where a function (Location Command) runs in the context of a hardware interrupt-triggered DSP/BIOS task.

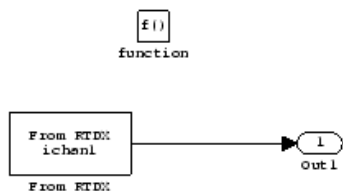


The DSP/BIOS Hardware Interrupt block installs an ISR function that signals a DSP/BIOS task to run when the ISR detects an RTDX™ interrupt. Signaling between the ISR and DSP/BIOS triggered task occurs via semaphores. This task receives an RTDX message carrying the location command for the downstream Text Insert block in the Text Overlay from the host computer.

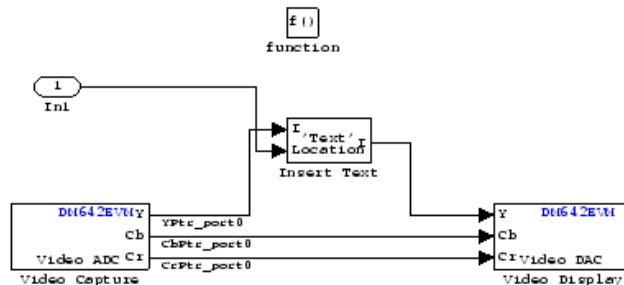
The blocks running inside the Location Command and Text Overlay subsystems are shown in the following figure.

The text overlay subsystem is executed as for the Free-Running DSP/BIOS Task. A Rate Transition block connects the two subsystems that run at two different asynchronous rates to ensure data integrity. The execution of two asynchronous rates is ordered based on the priority settings for the DSP/BIOS Task blocks.

Location Command Subsystem

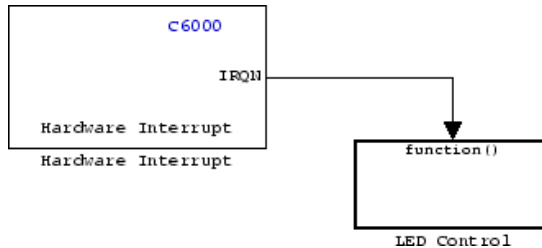


Text Overlay Subsystem

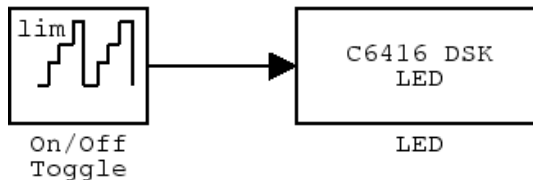


Hardware Interrupt Triggered Task

In the next figure, you see a case where a function (LED Control) runs in the context of a hardware interrupt triggered task.



In this model, the C6000 Hardware Interrupt block installs a task that runs when it detects an external interrupt. This task then toggles an external C6416DSK LED on or off.



Scheduling Considerations

When you use the DSP/BIOS task blocks for scheduling, either the DSP/BIOS Task block or the DSP/BIOS Triggered Task block, you must take care to avoid some common scheduling pitfalls.

First, the DSP/BIOS operating system always executes the task with the highest priority. Contrast this execution scheme with that of some other real-time operating systems (RTOS) where each task gets its fair share of processing time. Therefore, depending on the situation, there may be cases

where lower-priority tasks never execute because a higher priority task is never blocked.

A DSP/BIOS task blocks only when a blocking device driver block is included in the function call subsystem the task is executing, such as ADC/DAC blocks and C6000 UDP Receive blocks. If a particular DSP/BIOS task executes a function call subsystem that does not include any device driver blocks, and this particular task has the highest priority, it never releases the CPU, effectively disabling all other lower priority tasks in the application.

For more information about asynchronous schedulers, refer to the “Handling Asynchronous Events” chapter in your Real-Time Workshop documentation in the online help system.

Model Reference and Target Support Package Software

In this section...

“Overview” on page 2-35

“How Model Reference Works” on page 2-35

“Using Model Reference with Target Support Package Software” on page 2-37

“Configuring Targets to Use Model Reference” on page 2-38

Overview

Model reference lets your model include other models as modular components. This technique provides useful features because it:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your Real-Time Workshop documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- Top model — The root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.

- Referenced models — Blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your Real-Time Workshop documentation in the online help system.

Model Reference in Simulation

When you simulate the top model, Real-Time Workshop software detects that your model contains referenced models. Simulink generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (a MEX file, `.mex`) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these setting through the **Model Reference** pane of the Configuration Parameters dialog box.

Model Reference in Code Generation

Real-Time Workshop software requires executables to generate code from models. If you have not simulated your model at least once, Real-Time Workshop software creates a `.mex` file for simulation.

Now, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, Real-Time Workshop software calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

Using Model Reference with Target Support Package Software

With few limitations or restrictions, the target support package provides full support for generating code from models that use model reference.

Build Action Setting

The most important requirement for using model reference with the TI targets is that you must set the **Build action** (go to **Configuration Parameters > Embedded IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action

- 1** Open your model.
- 2** Select **Simulation > Configuration Parameters** from the model menus.
The Configuration Parameters dialog box opens.
- 3** From the **Select** tree, choose **Embedded IDE Link**.
- 4** In the right pane, under **Runtime**, set **Build action** to `Archive_library`.

If your top model uses a reference model that does not have the build action set to `Archive_library`, the build process automatically changes the build action to `Archive_library` and issues a warning about the change.

As a result of selecting the `Archive_library` setting, other options are disabled:

- DSP/BIOS is disabled for all referenced models. Only the top model supports DSP/BIOS operation.
- **Overrun action**, **Overrun notification method**, **Exporting CCS object to the workspace**, and **Stack size** are all disabled for the referenced models.

Target Preferences Blocks in Reference Models

Each referenced model and the top model must include a Target Preferences block for the correct target. You must configure all the Target Preferences blocks for the same target.

To obtain information about which compiler to use and which archiver to use to build the referenced models, the referenced models require Target Preferences blocks. Without them, the compile and archive processes does not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the Target Preferences block in the model provides.

Other Block Limitations

Model reference with Target Support Package software does not allow you to use certain blocks or S-functions in reference models:

- No blocks from the C62x DSP Library (`tic64dsplib`) (because these are noninlined S-functions)
- No blocks from the C64x DSP Library (`tic62dsplib`) (because these are noninlined S-functions)
- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Target Support Package library

Configuring Targets to Use Model Reference

Targets that you plan to use in Model Referencing must meet some general requirements.

- A model reference compatible target must be derived from the ERT or GRT targets.
- When you generate code from a model that references another model, you need to configure both the top-level model and the referenced models for the same code generation target.

- The External mode option is not supported in model reference Real-Time Workshop target builds. Target Support Package software supports External mode, but not with model reference. If you select this option, it is ignored during code generation. For more information, please see the “Communicating With Code Executing on a Target System Using Simulink External Mode” chapter in the Real-Time Workshop User’s Guide.
- To support model reference builds, your TMF must support use of the shared utilities folder, as described in Supporting Shared Utility folders in the Build Process.

To use an existing target, or a new target, with Model Reference, you set the `ModelReferenceCompliant` flag for the target. For information on how to set this option, refer to `ModelReferenceCompliant` in the online help system.

If you start with a model that was created prior to version 2.4 (R14SP3), to make your model compatible with the model reference target, use the following command to set the `ModelReferenceCompliant` flag to On:

```
set_param(bdroot,'ModelReferenceCompliant','on')
```

Models that you target with the target support package versions 2.4 and later automatically include the model reference capability. You do not need to set the flag.

Targeting Supported Boards

In this section...
“Overview” on page 2-40
“Typical Targeting Process” on page 2-41
“Targeting the C6713 DSP Starter Kit” on page 2-41
“Configuring Your C6713DSK” on page 2-43
“Confirming Your C6713DSK Installation” on page 2-44

Overview

Texas Instruments markets a complete set of tools for you to use with the a range of development boards, such as the C6713 DSK. These tools are primarily intended for rapid prototyping of control systems and hardware-in-the-loop applications. This section provides a brief example of how to use TI development tools with Real-Time Workshop software and the C6713 DSK blocks.

Executing code generated from Real-Time Workshop software on a particular target in real time requires target-specific code. Target-specific code includes I/O device drivers and an interrupt service routine. Other components, such as Embedded IDE Link software, are required if you need the ability to download parameters on the fly to your target hardware.

Since these components are specific to particular hardware targets (in this case, the C6713 DSK), you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, Target Support Package software provides a target makefile specific to the evaluation module. This target makefile invokes the optimizing compiler, provided as part of TI Code Composer Studio software.

Used in combination with Real-Time Workshop software, TI products provide an integrated development environment that, once installed, needs no additional coding.

Typical Targeting Process

Generally, targeting hardware, or a development environment as some call it, requires that you complete a series of processes that starts with building your model and ends with generating code to suit your target.

- 1** Build the Simulink model of your algorithm or process to be converted to code for your target.
- 2** Add target-specific blocks to your model, such as ADC and DAC blocks, and configure the block parameters.
- 3** Add a target preferences block to your model. Select the block that best matches your target—one of the device specific blocks, like C6713 DSK, or the Custom Board C6000 target preferences block when none of the specific blocks is appropriate. All models that you target to a C6000-processor-based hardware must have a target preferences block at the top level of the model.
- 4** Configure the options on the target preferences block to select the target, map memory segments, allocate sections to the memory segments, and configure other target-specific options.
- 5** Set the configuration parameters for your model. Notice that you do this step after you add the target preferences block to your model.
- 6** Build your model to your target.

Targeting the C6713 DSP Starter Kit

After you install the C6713 DSK development board and supporting TI products on your PC, start the MATLAB software. At the MATLAB command prompt, enter `c6713dsklib`. This opens a Simulink block library, `c6713dsklib`, that includes a set of blocks for C6713 DSK I/O devices, as described in the following table.

Block	Description
C6713 DSK ADC	Configure the analog to digital converter
C6713 DSK DAC	Configure the digital to analog converter
C6713 DSK LED	Control the user status LEDs on the C6713 DSK
C6713 DSK Reset	Reset the processor on the C6713 DSK

These blocks are associated with your C6713 DSK board. As needed, add the blocks to your model.

With your model open, select **Simulation > Configuration Parameters**. From this dialog box, select Real-Time Workshop from the **Select** tree. You must specify the appropriate versions of the system target file. For the C6713 DSK, in the Real-Time Workshop pane, specify **System target file** —`ccslink_grt.tlc`

With this configuration, you can generate a real-time executable and download it to the TI C6713 evaluation board. You generate the executable by clicking **Build** on the Real-Time Workshop pane. The Real-Time Workshop software automatically generates C code and inserts the I/O device drivers as specified in your block diagram. These device drivers are inserted in the generated C code as inlined S-functions. Inlined S-functions offer speed advantages and simplify the generated code. For more information about inlining S-functions, refer to Target Language Compiler Reference documentation. For a complete discussion of S-functions, refer to your Writing S-Functions documentation.

During the same build operation, the software invokes the TI compiler to build an executable file. If you select the **Build_and_execute** option, Real-Time Workshop software automatically downloads the executable to the TI evaluation board via the peripheral component interface (PCI) bus. After downloading the executable file to the C6713 DSK, the build process runs the file on the processor.

Starting and Stopping DSP Applications on the C6713 DSK

When you generate code, build the project, and download the code for your Simulink model to your C6713 DSK, you are running actual machine code corresponding to the block diagram you built in Simulink software. To start running your DSP application on the evaluation module, you must open your Simulink model and rebuild the machine executable by clicking **Build** on the **Real-Time Workshop** pane. To start the application on the C6713 DSK, you use Real-Time Workshop software to rebuild the executable from the Simulink model and download the code to the board.

Your model runs until it encounters one of the following actions:

- You select **Debug > Halt** in CCS IDE.

- You shut down the host PC.
- The process encounters a Stop block in the model code.
- The running application encounters an error condition that stops the process.

If you included a Reset C6713 DSK block in your model, clicking the block stops the running application and restores the digital signal processor to its initial state.

Note When you build and execute a model on the C6713 DSK, the Real-Time Workshop build process resets the evaluation module automatically. You do not need to reset the board before building models. To stop processes that are running on the evaluation module, or to return the board to a known state for any reason, use the Reset C6713 DSK block.

Configuring Your C6713DSK

When you install the C6713DSK, set the dual inline pin (DIP) switches as shown in the following table. If you have installed the board with different settings, reconfigure the board. Refer to your *TMS320C6201/6713Evaluation Module User's Guide* for details.

DIP Switch	Name	Setting	Effect
SW2-1	BOOTMODE4	On	Boot mode setting
SW2-2	BOOTMODE3	On	Boot mode setting
SW2-3	BOOTMODE2	Off	Sets memory map = 1 when SW2-5 is off
SW2-4	BOOTMODE1	On	Boot mode setting
SW2-5	BOOTMODE0	Off	Sets memory map =1 when SW2-3 is off
SW2-6	CLKMODE	On	Sets multiply-by-4 mode
SW2-7	CLKSEL	On	Selects oscillator A
SW2-8	ENDIAN	On	Selects little endian mode

DIP Switch	Name	Setting	Effect
SW2-9	JTAGSEL	Off	Selects internal Test Bus Controller (TBC)
SW2-10	USER2	On	User-defined option
SW2-11	USER1	On	User-defined option
SW2-12	USER0	On	User-defined option

Confirming Your C6713DSK Installation

Texas Instruments supplies a test utility to verify the operation of the board and its associated software. For complete information about running the test utility and interpreting the results, refer to your *TMS320C6201/6713 DSP Starter Kit User's Guide*.

To run the C6713 DSK verification test, complete the following steps after you install your board:

- 1 Start CCS IDE.
- 2 Select **Start > Programs > Code Composer Studio > DSK Confidence Test**. As the test runs, the results appear on your display.

By default, the test utility does not create a log file to store the test results. To specify the name and location of a log file to contain the results of the confidence test, use the command line options in CCS IDE to run the confidence test utility. For further information about running the verification test from a DOS window and using the command line options, refer to *TMS320C6201/6713 Evaluation Module User's Guide*.

- 3 Review the test results to verify that everything works. Check that the options settings match the settings listed in the table above.

If your options settings do not match the configuration shown in the preceding table, reconfigure your C6713 DSK. After you change your board configuration, rerun the verification utility to check your new settings.

Simulink Models and Targeting

In this section...
“Creating Your Simulink Model for Targeting” on page 2-45
“Blocks to Avoid in Your Models” on page 2-46

Creating Your Simulink Model for Targeting

You create real-time digital signal processing models the same way you create other Simulink models—by combining standard DSP blocks and C-MEX S-functions.

You add blocks to your model in several ways:

- Use blocks from the Signal Processing Blockset software
- Use blocks from the fixed-point blocks library TI C62x DSPLIB or TI C64x DSPLIB
- Use other Simulink discrete-time blocks
- Use the blocks provided in the C6000 blockset: ADC, DAC, LED and Reset blocks for specific supported target hardware
- Use blocks that provide the functions you need from any blockset installed on your computer
- Create and use custom blocks

Once you have designed and built your model, you generate C code and build the real-time executable by clicking **Build** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box. The automatic build process creates the file `modelName.out` containing a real-time model image in COFF file format that can run on your target.

The file `modelName.out` is an executable whose format is target-specific. You can load the file to your target and execute it in real time. Refer to your Real-Time Workshop documentation for more information about the build process.

Blocks to Avoid in Your Models

Many blocks in the blocksets communicate with your MATLAB workspace. All blocks generate code, but they do not work in the generated code as they do on your desktop.

You avoid using certain blocks, such as the Scope block and some source and sink blocks, in Simulink models that you use on Target Support Package targets. These blocks waste time in the generated code waiting to send or receive data from your MATLAB workspace, slowing your signal processing application without adding instrumentation value.

The following table describes blocks you should *not* use in your target models.

Block Name/Category	Library	Description
Scope	Simulink, Signal Processing Blockset software	Provides oscilloscope view of your output. Do not use the Save data to workspace option on the Data history pane in the Scope Parameters dialog box.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	Signal Processing Blockset	Compute and display the short-time FFT of a signal. It has internal buffering that can slow your process without adding value.
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	Signal Processing Blockset	Send data to your MATLAB workspace.

Block Name/Category	Library	Description
Signal To Workspace	Signal Processing Blockset	Send a signal to your MATLAB workspace.
Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
To Wave device	Signal Processing Blockset	Send data to a .wav device.
From Wave device	Signal Processing Blockset	Get data from a .wav device.

In general, using blocks to add instrumentation to your application is a valuable tool. In most cases, blocks you add to your model to display results or create plots, such as Histogram blocks, add to your generated code without affecting your running application.

Targeting Tutorial II – A More Complex Application

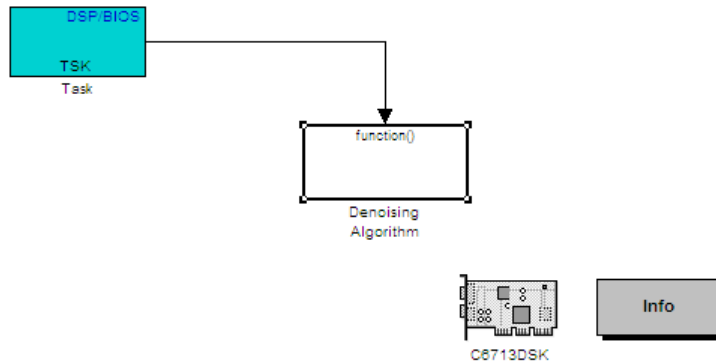
In this section...
“Overview” on page 2-48
“Working and Build folders” on page 2-49
“Setting Simulation Program Parameters” on page 2-50
“Selecting the Target Configuration” on page 2-51
“Building and Running the Program” on page 2-55
“Contents of the Build folder” on page 2-56

Overview

For this tutorial, we demonstrate an application that uses multiple stages—using wavelets to remove noise from a noisy signal. Open the demo model, `c6713dskwdnoisf`. As with any model file, you can run this denoising demonstration by typing `c6713dskwdnoisf` at the MATLAB prompt. The model also appears in the MATLAB demos collection in the Help browser—under Simulink demos, in the Target Support Package category. Here is a picture of the model as it appears in the demonstration library.

Wavelet Denoising

C6713 DSK



Unlike the audio reverberation demo, this model is difficult to build from blocks in Simulink software. It uses complex subsystems for the Delay Alignment block and the Soft Threshold block. For this tutorial, you work with a copy of the demonstration model, rather than creating the model.

This tutorial takes you through generating C code and building an executable program from the demonstration model. The resulting program runs on your C6713 DSK as an executable COFF file.

Working and Build folders

It is convenient to work with a local copy of the `c6713dskwdnoisf` model, stored in its own folder, which you named (something like `c6713dnoisfex`). This discussion assumes that the `c6713dnoisfex` folder resides on drive `d:`. Use a different drive letter if necessary for your machine. Set up your working folder as follows:

- 1 Create the new model folder from the MATLAB command line by typing

```
!mkdir d:\c6713dnoisfex (on PC)
```

- 2 Make `c6713dnoisfex` your working folder.

```
cd d:/c6713dnoisfex
```

- 3 Open the `c6713dskwdnoisf` model.

```
c6713dskwdnoisf
```

The model appears in the Simulink window.

- 4 From the **File** menu, choose **Save As**. Save a copy of the `c6713dskwdnoisf` model as `d:/c6713dnoisfex/dnoisfrtw.mdl`.

During code generation, Real-Time Workshop software creates a build folder within your working folder. The build folder name is `model_target_rtw`, derived from the name of your source model and your chosen target. In the build folder, Real-Time Workshop software stores generated source code and other files created during the build process. You examine the contents of the build folder at the end of this tutorial.

Setting Simulation Program Parameters

To generate code correctly from the `dnoisfrtw` model, you must change some of the configuration parameters. In particular, Real-Time Workshop software uses a fixed-step solver. To set the parameters, use the Configuration Parameters dialog box as follows:

- 1 From the **Simulation** menu, choose **Configuration Parameters**. The Configuration Parameters dialog box opens.
- 2 Click **Solver** and enter the following parameter values on the **Solver** pane. Note that Target Support Package software does not honor a stop time if you set one here.

Start Time: 0.0

Stop Time: `inf`

Solver options: set **Type** to **Fixed-step**. Select the **Discrete** solver algorithm. (Targeting does not work with continuous time solvers.)

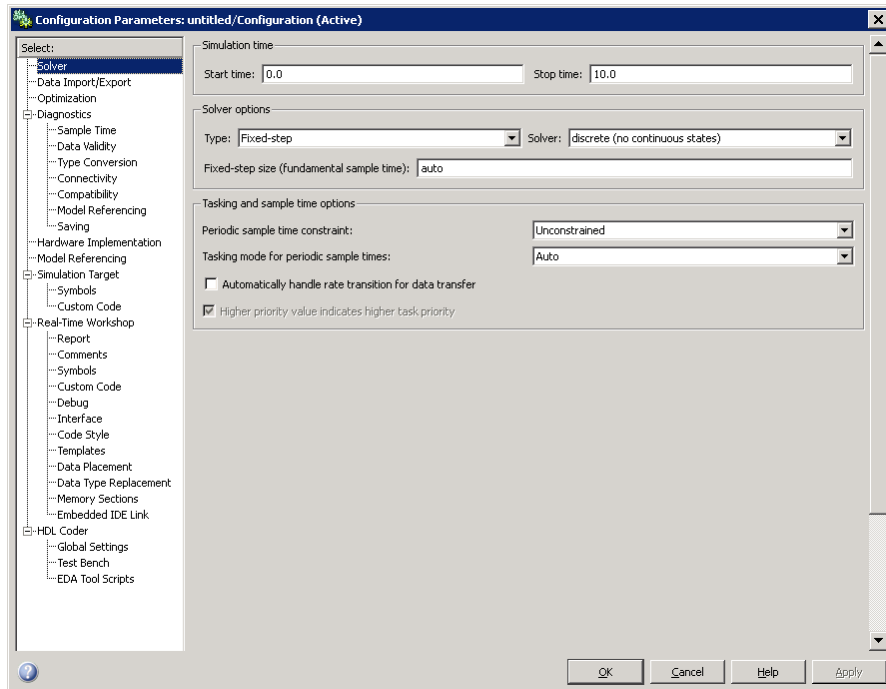
Fixed step size: `auto`

Tasking mode for periodic sample times: `Auto`

3 Click **Apply**, and then click **OK** to close the dialog box.

4 Save the model. Configuration parameters persist with the model (as the model configuration set), for you to use in future sessions.

In the next figure you see the Solver pane with the correct parameter settings.



Selecting the Target Configuration

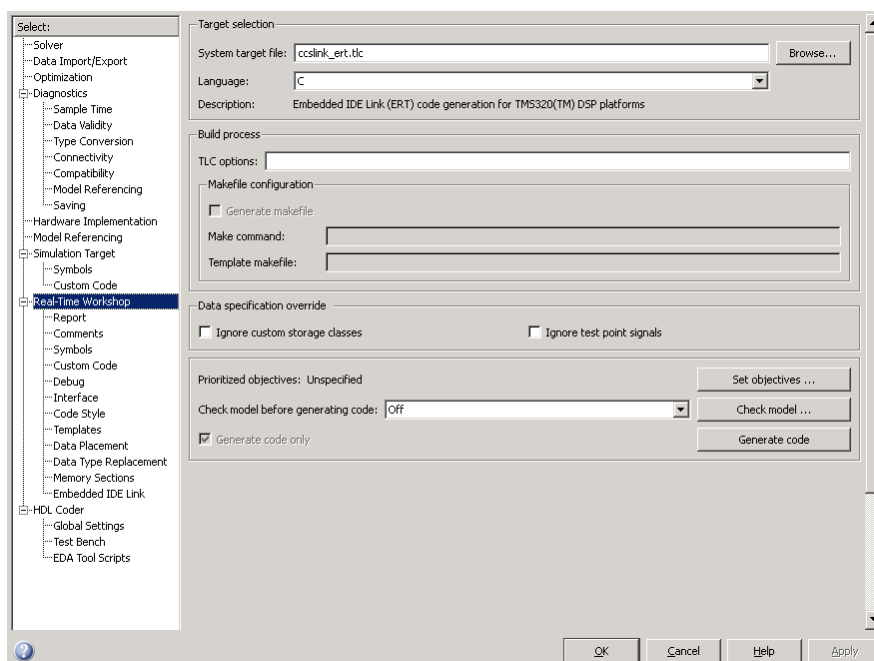
To specify the desired target configuration, choose the **System target file**.

In these tutorials, you do not need to specify these parameters individually. Instead, you use the ready-to-run `ccslink_grt.tlc` target configuration.

Note The Real-Time Workshop category has several subcategories that you select using the **Select** tree in the Configuration Parameters dialog box. During this tutorial you change or review options in just a few of the categories in the tree.

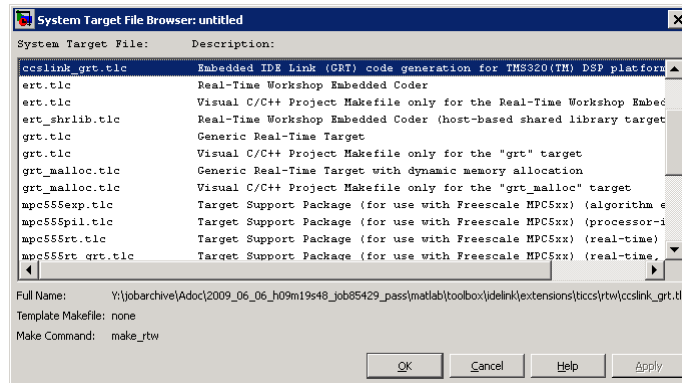
To target your C6713 DSK:

- 1 From the **Simulation** menu, choose **Configuration Parameters**. The Configuration Parameters dialog box opens.
- 2 Click Real-Time Workshop on the **Select** tree. The Real-Time Workshop pane activates.

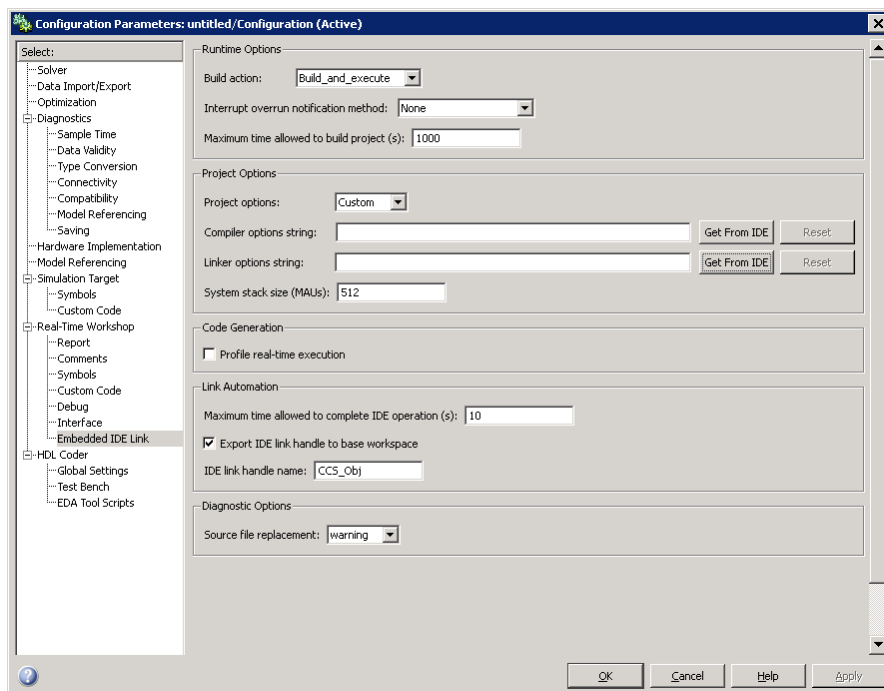


- 3 Click **Browse** next to the **System target file** field. This opens the **System Target File Browser**. The browser displays a list of available target configurations. When you select a target configuration, Real-Time

Workshop software automatically chooses the appropriate system target file.



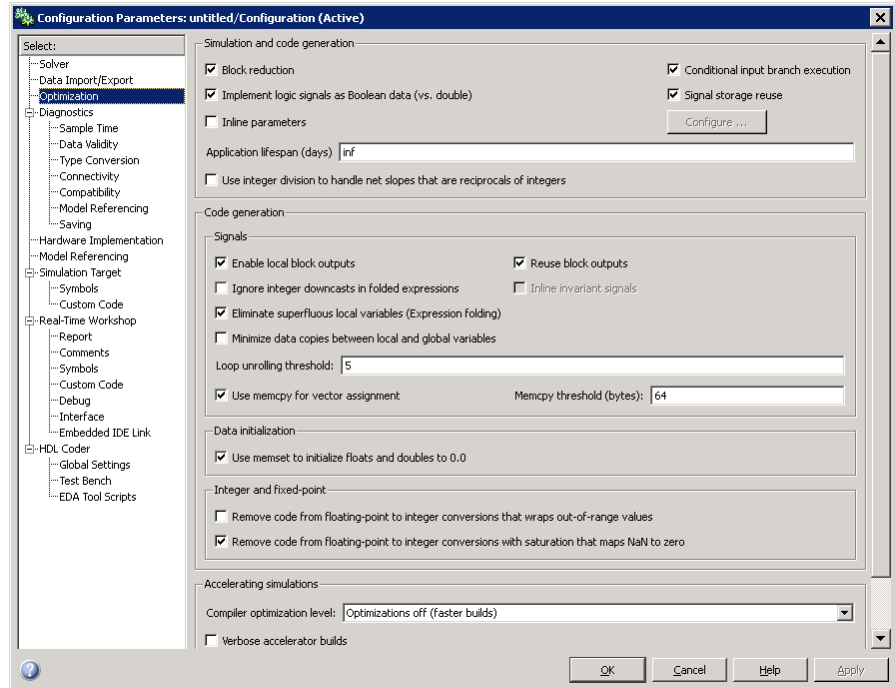
- 4 From the list of available configurations, select `ccslink_grt.tlc`, and click **OK**.
- 5 To decide whether to export a CCS handle to your MATLAB workspace when you generate code, or run your model, select **Embedded IDE Link** from the **Select** tree.



6 Set the **Runtime** and **Project options** as shown in the preceding figure.

7 To export the handle (a variable) that CCS IDE creates when you generate code from your model, select **Export IDE link handle to base workspace**, and enter a name for the handle in **IDE link handle name**.

8 Select **Optimization** from the **Select** tree. A new set of options appears. The options displayed here are common to all target configurations. Make sure that all options are set to their defaults, as shown in the following figure.



- 9 Click **OK** to close the Configuration Parameters dialog box. Save the model to retain your new build settings.

Building and Running the Program

The Real-Time Workshop build process generates C code from your model, and then compiles and links the generated program.

To build and run your program:

- 1 Access the Configuration Parameters dialog box for your model.
- 2 Click **Build** in the Real-Time Workshop pane to start the build process.
- 3 A number of messages concerning code generation and compilation appear in the MATLAB workspace. The initial messages are

```
### Starting Real-Time Workshop build procedure for model:
```

```
dnoisfrtw
### Generating code into build folder: .\dnoisfrtw_c6000_rtw
```

The content of the succeeding messages depends on your compiler and operating system. The final message is

```
### Successful completion of Real-Time Workshop build procedure
for model: dnoisfrtw
```

- 4** The working folder now contains an executable, `dnoisfrtw.exe`. In addition, Real-Time Workshop software created a build folder, `dnoisfrtw_c6000_rtw`.

To review the contents of the working folder after the build, type the `dir` command at the MATLAB command prompt.

```
dir
.          dnoisfrtw.exe      dnoisfrtw_c6000_rtw
..         dnoisfrtw.mdl
```

- 5** To run the executable from the MATLAB command prompt, type

```
!dnoisfrtw
```

The “!” character passes the command that follows it to the operating system, which runs the stand-alone `dnoisfrtw` program.

The program produces one line of output.

```
**starting the model**
```

- 6** To see the contents of the build folder, type

```
dir dnoisfrtw_c6713_rtw
```

Contents of the Build folder

The build process creates a build folder and names it `model_target_rtw`, concatenating the name of your source model and your chosen target. In this example, your build folder is named `dnoisfrtw_c6713_rtw`.

`dnoisfrtw_c6713_rtw` contains these generated source code files:

- `dnoisfrtw.c` — The stand-alone C code that implements the model.
- `dnoisfrtw.h` — An include header file containing information about the state variables
- `dnoisfrtw_export.h` — An include header file containing information about exported signals and parameters

The build folder also contains other files used in the build process, such as the object (`.obj`) files and the generated makefile (`dnoisfrtw.mk`).

Targeting Your C6713 DSK and Other Hardware

In this section...
“Overview” on page 2-58
“Configuring Your C6713 DSK” on page 2-59
“Confirming Your C6713 DSK Installation” on page 2-59
“Running Models on Your C6713 DSK” on page 2-60

Overview

Target Support Package software lets you use Real-Time Workshop software to generate, target, and execute Simulink models on the Texas Instruments (TI) C6713 DSP Starter Kit (C6713 DSK). In combination with the C6713 DSK, your the target support package is the ideal resource for rapidly prototyping and developing embedded systems applications for the TI C6713 Digital Signal Processor. The target support package focuses on developing real-time digital signal processing (DSP) applications for the C6713 DSK.

This chapter describes how to use the target support package to create and execute applications on the C6713 DSK. To use the targeting software, you should be familiar with using Simulink to create models and with the basic concepts of Real-Time Workshop software automatic code generation. To read more about Real-Time Workshop software, refer to your Real-Time Workshop documentation.

In this chapter, you will find sections that detail how to use the target support package to build and download DSP applications in Simulink software to your C6713 DSK and to the Texas Instruments Code Composer Studio IDE:

- Configuring your the target support package, in
- Configuring your Texas Instruments TMS320C6713 DSP Starter Kit, in “Configuring Your C6713 DSK” on page 2-59
- Testing your hardware and software installation to be sure everything works, in “Confirming Your C6713 DSK Installation” on page 2-59

Configuring Your C6713 DSK

After you install and configure your C6713 DSK according to the instructions in the CCS IDE online help, you do not need to configure further your C6713 DSK.

Confirming Your C6713 DSK Installation

Texas Instruments supplies a test utility to verify operation of the board and its associated software. For complete information about running the test utility and interpreting the results, refer to your “TMS320CDISK Help” under TMS320C6000 Code Composer Studio Help in the CCS online help system.

To run the C6713 DSK confidence test, complete the following steps after you install and configure your board.

1 Open a DOS command window.

2 Access the folder `..\ti\c6000\disk6x11\conftest`

CCS IDE creates this folder when you install it. It contains the files to run the C6713 confidence test.

3 Start the confidence test by typing `disk6xtst` at the DOS prompt.

By default, the test utility creates a log file named `disk6xtst.log` where it stores the test results. To specify the name and location of a log file to contain the results of the confidence test, use the CCS IDE command line options to run the confidence utility. For further information about running the confidence test from a DOS window and using the command line options, refer to the "DSK Confidence Test" topic in the CCS IDE online help.

4 Review the test results to verify that everything works.

If your confidence test fails, reconfigure your C6713 DSK. After you change your board configuration, rerun the confidence utility to check your new settings.

Running Models on Your C6713 DSK

Texas Instruments markets a complete set of tools for use with the C6713 DSK. These tools are primarily intended for rapid prototyping of control systems and hardware-in-the-loop applications.

This section provides a brief example of how the TI development tools work with Real-Time Workshop software, the target support package, and the C6713 DSK Board Support block library.

Executing code generated from Real-Time Workshop software on a particular target in real-time requires target-specific code. Target-specific code includes I/O device drivers and an interrupt service routine.

Other components, such as Embedded IDE Link software, are required if you need the ability to download parameters on-the-fly to your target hardware.

Since these components are specific to particular hardware targets (in this case, the C6713 DSK), you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, the target support package provides a target makefile specific to C6000 hardware targets. This target makefile invokes the optimizing compiler provided as part of CCS IDE.

Used in combination with the target support package and Real-Time Workshop software, TI products provide an integrated development environment that, once installed, needs no additional coding.

After you have installed the C6713 DSK development board and supporting TI products on your PC, start the MATLAB software. At the MATLAB command prompt, type `c6713dsklib`. This opens a Simulink block library, `c6713dsklib`, that includes a set of blocks for C6713 DSK I/O devices:

- C6713 DSK ADC — Configure the analog to digital converter
- C6713 DSK DAC — Configure the digital to analog converter
- C6713 DSK LED — Control the user-defined light emitting diodes (LED) on the C6713 DSK

- C6713 DSK DIP Switch — Set the dual inline pin switches on the C6713 DSK
- C6713 DSK Reset — Reset the processor on the C6713 DSK

These devices are associated with your C6713 DSK board.

With your model open, select **Simulation > Configuration Parameters** from the menu bar to open the Configuration Parameters dialog box.

From this dialog box, click **Real-Time Workshop** on the select tree. You must specify the appropriate versions of the system target file. For the C6713 DSK, in the **Real-Time Workshop** pane of the dialog box, specify **System target file** — `ccslink_grt.tlc`

With this configuration, you can generate and download a real-time executable to your TI C6713 DSK. Start the Real-Time Workshop build process by clicking **Build** on the **Real-Time Workshop** pane. Real-Time Workshop software automatically generates C code and inserts the I/O device drivers as specified by the ADC and DAC blocks in your block model.

These device drivers are inserted in the generated C code as inlined S-functions. Inlined S-functions offer speed advantages and simplify the generated code. For more information about inlining S-functions, refer to your Target Language Compiler documentation. For a complete discussion of S-functions, refer to your documentation about writing S-functions.

During the same build operation, the software invokes the TI compiler to build an executable file.

If you select the **Build_and_execute** option, the executable file is automatically downloaded via the peripheral component interface (PCI) bus to the TI evaluation board. After downloading the executable file to the C6713 DSK, the build process runs the file on the digital signal processor.

Starting and Stopping DSP Applications on the C6713 DSK

When you create, build, and download a Simulink model to the C6713 DSK, you are not running a simulation of your DSP application. You are running the actual machine code corresponding to the block diagram you built in Simulink software. To start running your DSP application on the evaluation

module, you must open your Simulink model and rebuild the machine executable by clicking **Build** on the **Real-Time Workshop** pane. Each time you want to start the application on the C6713 DSK, you use Real-Time Workshop software to rebuild the executable from the Simulink model and download the code to the board.

Your model runs until the model encounters one of the following actions:

- Using the **Debug > Halt** option in CCS IDE
- Using halt from the MATLAB command prompt
- Encountering a Stop block in the model.
- Clicking the C6713 DSK Reset block in your model (if you added one) or in the DSK block library

Clicking the Reset block stops the running application and restores the digital signal processor to its initial state.

Creating Code Composer Studio Projects Without Building

In this section...

“Introduction” on page 2-63

“Creating Projects in CCS IDE Without Loading Files to Your Target” on page 2-63

Introduction

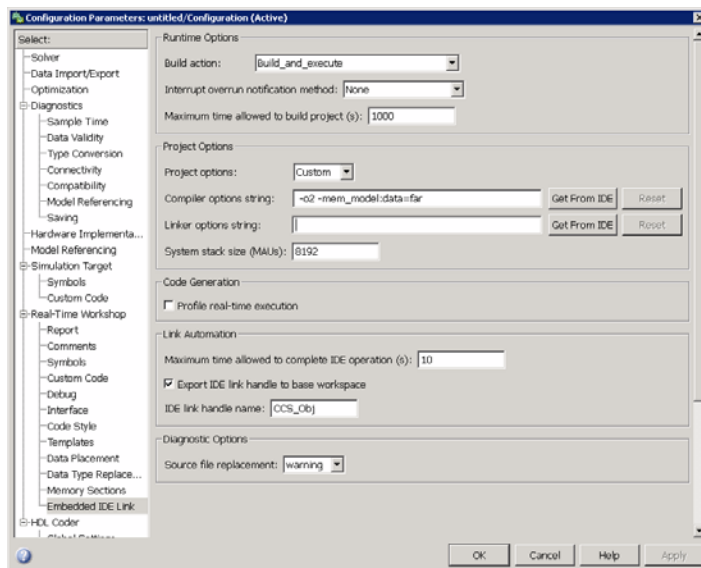
Rather than targeting your C6000 board when you build your signal processing application, you can create Texas Instruments Code Composer Studio (CCS) IDE projects. Creating projects for CCS IDE lets you use the tools provided by the CCS IDE software suite to debug your real-time process.

If you build and download your Simulink model to CCS IDE, Target Support Package software opens Code Composer Studio software, creates a new CCS IDE project named for your model, and populates the new project with all the files it creates during the build process—the object code files, the assembly language files, the map files, and any other necessary files. As a result, you can immediately use CCS IDE to debug your model using the features provided by the CCS IDE.

Creating a project in CCS IDE is the same as targeting C6000 hardware. You configure your target options, select your build action to create a CCS IDE project, and then build the project in CCS IDE by clicking **Make Project**.

Creating Projects in CCS IDE Without Loading Files to Your Target

From the **Select** tree in the Configuration Parameters dialog box, select **Embedded IDE Link**. Select **Create_Project** for the **Build action**, as shown in the next figure. The **Build** and **Build_and_execute** options create CCS IDE projects as well. The **Archive_library** option does not create a CCS IDE project. None of the other options has an effect here. Ignore them when you are creating a project in CCS IDE rather than generating code.



After you select `Create_CCS_Project`, set the options for the **Code Generation** options on the **Embedded IDE Link** category on the **Select** tree.

Return to the **Real-Time Workshop** category, clear **Generate code only** and click **Build** to build your new CCS IDE project.

Real-Time Workshop software and Target Support Package software generate all the files for your project in CCS IDE and create a new project in the IDE. Your new project is named for the model you built, with a custom project build configuration `CustomMW`, not `Release` or `Debug`.

In CCS IDE you see your project with the files in place in the folder tree.

Targeting Custom Hardware

In this section...

“Overview” on page 2-65

“Typical Targeting Process” on page 2-67

“Targeting a Custom Target” on page 2-69

“Sections Pane” on page 2-77

“To Create Memory Maps for Targets” on page 2-83

Overview

As long as the processor on your custom board is from the TI C6000 DSP family, you can use Target Support Package software to generate code for your target.

The blocks for the peripherals in the C6000 DSP Library, such as the C6416 DSK ADC or C6713 DSK DAC blocks, are specific to their hardware and will not work with your custom board. None of the board-specific blocks provided by this toolbox work with custom hardware.

Custom hardware targeting currently supports all C6000 processors through target preferences blocks, either specific to the processor, or a general custom preferences block. These target preferences blocks are described briefly in the following table

Target Preferences Block	Description
Custom Board C6000	Provides access to the hardware set up for targeting any C6000 processor-based board. Note that it does not set any default values. When you add this block to a model, you must set all the options on each available pane—board information, memory mapping, and section layout.

Target Preferences Block	Description
C6416DSK	Sets default values for targeting the C6416 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6455DSK	Sets default values for targeting the C6455 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6713 DSK	Sets default values for targeting the C6713 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6713DSK	Sets default values for targeting the C6713 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6727DSK	Sets default values for targeting the C6727 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
DM642EVM	Sets default values for targeting the DM642 EVM. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.

These target preferences blocks provide a direct way for you to target boards that are not specifically supported. Due to certain features related to memory maps and other processor-specific attributes, custom hardware targeting only works with the C6000 DSPs.

Several guidelines affect your targeting configuration decisions when you decide to use custom targets and the custom target preferences block:

- 1** Specify the memory allocation (memory mapping) using the **Memory** and **Sections** panes on the C6000 Target Preferences dialog box. Set the memory mapping for your target that best matches your hardware. For example, if your custom target uses the C6713 processor, be sure your memory configuration is the same as the one on the supported C6713 DSK, such as has the same memory size, the same EMF settings, the same memory sections, and the same cache organization.
- 2** To use on-chip memory only for your target, choose the `Near_Calls` setting for the **Memory model** in the **TI C6000 compiler** options. To use external memory that is specific to your board, choose the `Far_Calls` setting for the **Memory model**. The other selection in the **Memory model** list offers a combination of near and far allocation for data and aggregate data.
- 3** Do not use the existing ADC, DAC, DIP Switch, or LED blocks unless you are quite sure that your hardware is identical to the appropriate EVM or DSK in all important respects. Generally, the ADC, DAC, and other target-specific blocks are design specifically for their designated targets and can cause problems when you use them on hardware that is not identical.
- 4** Set the **Overrun notification method** in the **TI C6000 runtime** category to `Print_message` when you use the overrun notification feature. If you choose to use the LED notification option, verify that on your specialized target you access the LEDs in exactly the same way, and the LEDs respond in the same way, as the LEDs on the corresponding supported DSK or EVM.

To use one of the custom targets, create your model, add and configure the Custom Board C6000 target preferences block, and then open the Configuration Parameters dialog box for the model.

Typical Targeting Process

Generally, targeting hardware, or a development environment as it is called by some, requires that you complete a series of processes that starts with building your model and ends with generating code to suit your target.

- 1** Build the Simulink model of your algorithm or process to be converted to code for your target.

- 2 Add target-specific blocks to your model, such as ADC and DAC blocks, and configure the block parameters. (Skip this step when you are targeting a processor on a custom board.)
- 3 Add a target preferences block to your model. Select a device-specific block, such as the C6713DSK target preferences block. When none of the specific blocks is appropriate, select the Custom Board C6000 target preferences block. The top level of the model must contain a target preferences block.
- 4 Configure the options on the target preferences block to select the target, map memory segments, allocate code and data sections to the memory segments, and set other target-specific options.
- 5 Set the Simulink configuration parameters for your model. Notice that you do this after you add the target preferences block to your model.
- 6 Build your model to your target.

Memory Maps

Memory maps are an essential part of targeting any processor or board. Without the map, the code generation process cannot determine where various features of the generated code, such as variables, data, and executable code, reside on the target.

To discuss memory maps and configuring memory, a few terms need to be defined:

- **Memory map** — Map of the memory space for a target system. The memory space is partitioned into functional blocks.
- **memory segment** — Memory partition that corresponds to a physical range of memory on the target. The segment is named in some fashion, such as IPRAM or SDRAM.
- **Memory section** — The smallest unit of an object file. This is a block of data or code that, based on the memory map, resides in an area of contiguous memory on the target and in the memory map. Sections of object files are both distinct and separate. Memory sections come in two flavors:

- Uninitialized sections that reserve memory space for uninitialized data. One example of an uninitialized section is `.bss`. The `.bss` section reserves space for variables that are not initialized.
- Initialized sections contain code and data. The `.text` (containing executable code) and `.data` (containing initialized data) sections are initialized.
- Memory management — Process of specifying the memory segments that the various memory sections use for your application. A logical memory map of the hardware memory results from the process of managing memory.

During code generation, the linker and assembler work to allocate your code and data into the memory on your target according to the memory map specifications you provide. For more information about memory utilization and memory management, refer to the CCS IDE online help, using keywords like memory map, memory segment, and section.

The compiler does not interact with the memory map. It makes no assumptions about memory allocation and is not aware of the memory map. As far as the C6000 compiler is concerned, the physical memory on your target is one continuous linear block of memory that is subdivided into smaller blocks containing code, data, or both.

When you configure the block parameters for the Custom Board C6000 target preferences block, you are setting up the memory map for your target. You specify the memory segments that are defined and the contents of each segment. You specify the sections, both named and default, and the segments to which the sections are assigned.

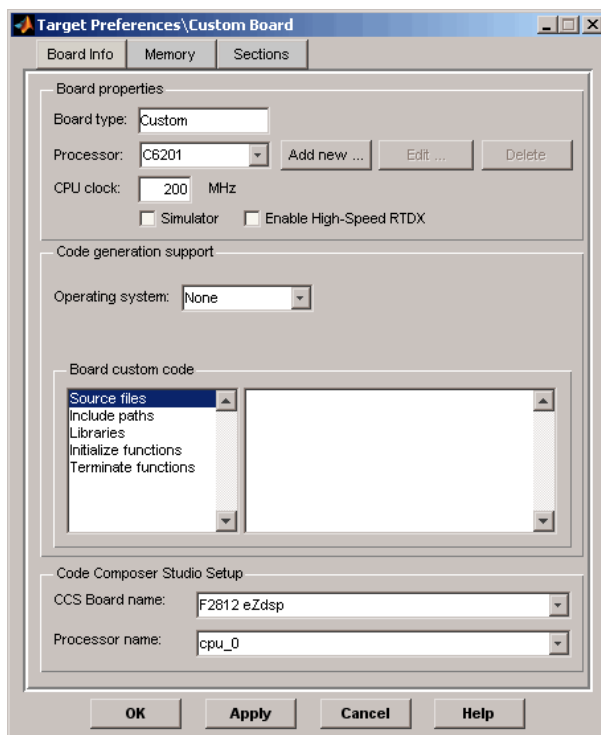
These memory management functions are identical to the ones available in the CCS IDE Configuration Tool.

Targeting a Custom Target

To use a board that has a TI C6000 processor but is not one of the supported boards, use the Custom Board C6000 target preferences block by adding it to your model.

Configuring the block parameters tell Simulink software, Target Support Package software, and Real-Time Workshop software about your target processor and how to generate code that will run on the target.

- 1 After you add the Custom Board C6000 target preferences block to your model, open the block by selecting **Edit > Open Block** from the model menu bar. This step opens the C6000 Target Preferences dialog box, containing default values for all options. In the next steps you change the options to specify features of your target processor and board.
- 2 Click **Board Info** to access the board information pane shown in the following figure.



- 3 For **Board type**, enter Custom to tell the system you are targeting a board that the target support package does not explicitly support.

- 4 Select your target processor from the **Processor** list. Most of the C6000 family of DSP processors are on the list. If the one you need is not listed, pick one that closely matches your target.
- 5 Set the actual CPU clock rate for the CPU on your target in CPU clock speed (MHz). Report the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results might be wrong, and code profiling results will not be correct. You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.
- 6 If your target is a simulator rather than a hardware target, select **Simulator**.
- 7 To enable the target support package to connect to CCS IDE, select your target from the **CCS board name** list. On this list you see the names of the boards you have configured in the CCS Setup Utility. If your target board does not appear on the list, start CCS Setup and add your board to the System Configuration dialog box.
- 8 Select the processor to target from the **CCS processor name** list. For the board you selected in **CCS board name**, **CCS processor name** lists all the processors on the board. The list comes from the processors you added to the board in the CCS Setup Utility.

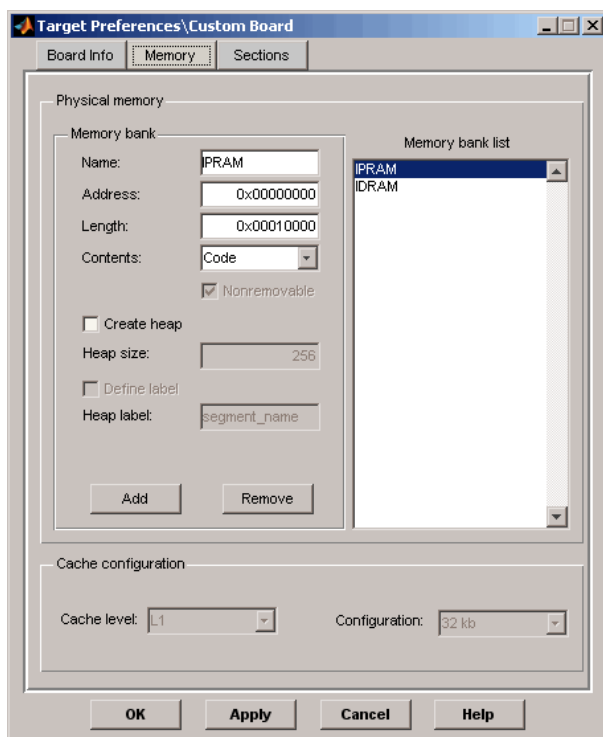
Now you have completed the process of identifying your target to the target support package and Real-Time Workshop software. While this process is necessary, it represents only one small part of enabling you to generate code to run on your custom board.

One very important part of targeting custom hardware is to provide the target memory map configuration to the linker and assembler.

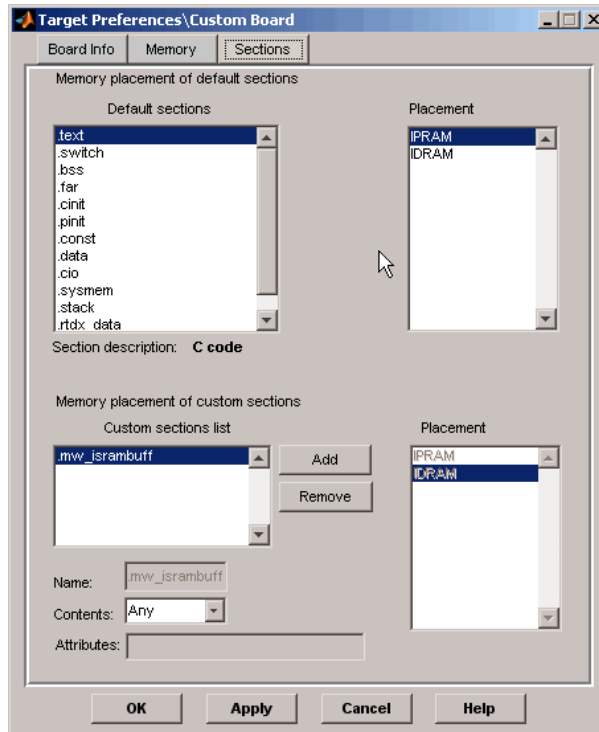
Memory and **Sections** panes on the C6000 Target Preferences dialog box provide the controls required to specify how the linker and assembler arrange the code, data, and variables on your target.

The following figures show the **Memory** and **Sections** panes with the default values for all options.

Memory Pane



Sections Pane



The information that follows describes the options on the panes in detail.

The **Memory** pane contains memory options in three areas:

- **Physical Memory** specifies the mapping for processor memory
- **Heap** specifies whether you use a heap and determines the size in words
- **L2 Cache** enables the L2 cache (where available) and sets the size in kB

Be aware that these options can affect the options on the **Sections** pane. You can make selections here that change how you configure options on the **Sections** pane.

Most of the information about memory segments and memory allocation is available from the Code Composer Studio online help.

Physical Memory Options

This list shows the physical memory segments available on the board and processor. By default, target preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured target preferences blocks shows the memory segments available on the board, but off of the processor. Target preferences blocks set default starting addresses, lengths, and contents of the default memory segments.

The default memory segments for each processor and board are different. For example:

- Custom boards based on C670x processors provide IPRAM and IDRAM memory segments by default.
- C6713 DSK boards provide SDRAM memory segment by default

Name

When you highlight an entry on the **Physical memory** list, the name of the entry appears here. To change the name of the existing memory segment, select it in the Physical memory list and then type the new name here.

Note You cannot change the names of default processor memory segments.

To add a new physical memory segment to the list, click **Add**, replace the temporary label in **Name** with the one to use, and press **Return**. Your new segment appears on the list.

After you add the segment, you can configure the starting address, length, and contents for the new segment. New segments start with code and data as the type of content that can be stored in the segment (refer to the **Contents** option).

Names are case sensitive. NewSegment is not the same as newsegment or newSegment.

Address

Address reports the starting address for the memory segment showing in **Name**. Address entries are in hexadecimal format and limited only by the board or processor memory.

When you are using a processor-specific preferences block, the starting address shown is the default value. You can change the starting value by entering the new value directly in **Address** when you select the memory segment to change.

Length

From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs). For the C6000 processor family, the MADU is 8 bytes, one word.

When you are using a processor-specific preferences block, the length shown is the default value. You can change the value by entering the new value directly in this option.

Contents

Contents describes the kind of program sections that you can store in the memory segment in **Name**. As the processor type for the target preferences block changes, the kinds of information you store in listed memory segments can change. Generally, the **Contents** list contains these strings:

- **Code** — Allow code to be stored in the memory segment in **Name**.
- **Data** — Allow data to be stored in the memory segment in **Name**.
- **Code** and **Data** — Allow code and data to be stored in the memory segment in **Name**. When you add a new memory segment, this is the default setting for the contents of the new element.

You can add or use as many segments of each type as you need, within the limits of the memory on your processor.

Add

Click **Add** to add a new memory segment to the target memory map. When you click **Add**, a new segment name appears, for example NEWMEM1, in **Name** and on the **Physical memory** list. In **Name**, change the temporary name NEWMEM1 by entering the new segment name. Entering the new name, or clicking **Apply** updates the temporary name on the list to the name you enter.

Remove

This option lets you remove a memory segment from the memory map. Select the segment to remove in the **Physical memory** list and click **Remove** to delete the segment.

Create Heap

If your processor supports using a heap, as does the C6713, for example, selecting this option enables creating the heap and enables the **Heap size** option. **Create heap** is not available on processors that either do not provide a heap or do not allow you to configure the heap.

Using this option you can create a heap in any memory segment on the **Physical memory** list. Select the memory segment on the list and then select **Create heap** to create a heap in the select segment. After you create the heap, use the **Heap size** and **Define label** options to configure the heap.

The location of the heap in the memory segment is not under your control. The only way to control the location of the heap in a segment is to make the segment and the heap the same size. Otherwise, the compiler determines the location of the heap in the segment.

Heap Size

After you select **Create heap**, this option lets you specify the size of the heap in words. Enter the number of words in decimal format. When you enter the heap size in decimal words, the system converts the decimal value to hexadecimal format. You can enter the value directly in hexadecimal format as well. Processors may support different maximum heap sizes.

Define Label

Selecting **Create heap** enables this option that allows you to name the heap. Enter your label for the heap in the **Heap label** option.

Heap Label

Selecting **Define label** enables this option. You use **Heap Label** to provide the label for the heap. Any combination of characters is accepted for the label except reserved characters in C/C++ compilers.

Enable L2 Cache

C621x, C671x, and C641x processors support an L2 cache memory structure that you can configure as SRAM and partial cache. Both the data memory and the program share this second-level memory. C620x DSPs do not support L2 cache memory, and this option is not available when you choose one of the C620x processors as your target.

If your processor supports the two-level memory scheme, this option enables the L2 cache on the processor.

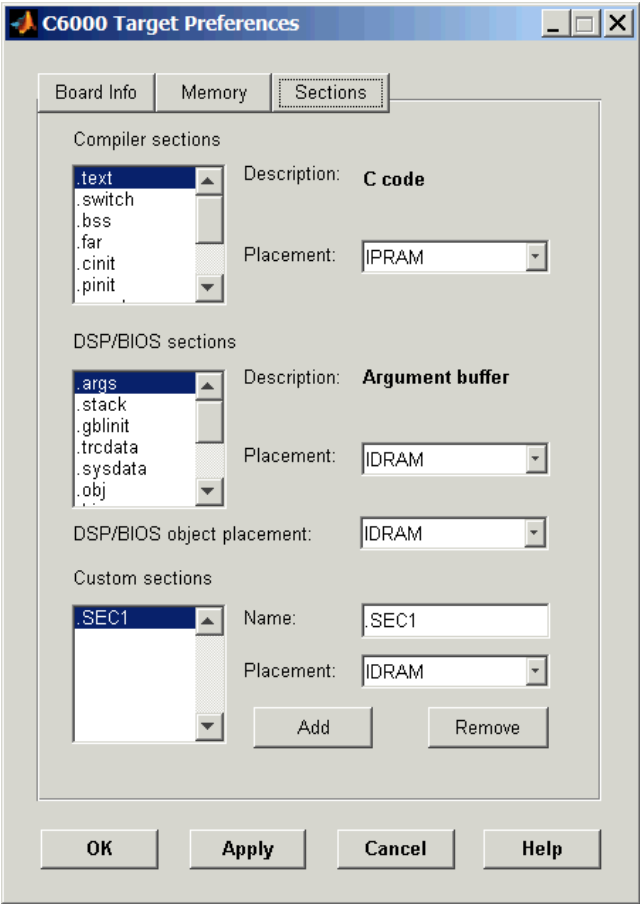
L2 Cache Size

After you enable the L2 cache, select the size of the cache from the list.

Sections Pane

Options on this pane let you specify where various program sections should go in memory. Program sections are distinct from memory segments—sections are portions of the executable code stored in contiguous memory locations. Among the sections used generally are `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler, some to DSP/BIOS, and some can be custom sections as you require.

For more information about program sections and objects, refer to the CCS IDE online help. Most of the definitions and descriptions in this section come from CCS IDE.



In the pane shown in the preceding figure, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

This table provides brief definitions of the various kinds of sections in the **Compiler**, **DSP/BIOS**, and **Custom** lists. All sections do not appear on both lists. The string appears on the list shown in the table.

String	Section List	Description of the Section Contents
.args	DSP/BIOS	Argument buffers
.bss	Compiler	Static and global C variables in the code
.bios	DSP/BIOS	DSP/BIOS code if you are using DSP/BIOS options in your program
.cinit	Compiler	Tables for initializing global and static variables and constants
.cio	Compiler	Standard I/O buffer for C programs
.const	Compiler	Data defined with the C qualifier and string constants
.data	Compiler	Program data for execution
.far	Compiler	Variables, both static and global, defined as far variables
.gblinit	DSP/BIOS	Load allocation of the DSP/BIOS startup initialization tables section
.hwi	DSP/BIOS	Dispatch code for interrupt service routines
.hwi_vec	DSP/BIOS	Interrupt Service Table
.obj	DSP/BIOS	Configuration properties that the target program can read
.pinit	Compiler	Load allocation of the table of global object constructors section.
.stack	Compiler	The global stack
.switch	Compiler	Jump tables for switch statements in the executable code
.sysdata	DSP/BIOS	Data about DSP/BIOS
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code

String	Section List	Description of the Section Contents
.systemem	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help.

Compiler Sections

During program compilation, the C6000 compiler produces both uninitialized and initialized blocks of data and code. These blocks are allocated into memory as required by the configuration of your system. On the **Compiler Sections** list you find both initialized (sections that contain data or executable code) and uninitialized (sections that reserve space in memory) sections. The initialized sections are

- .cinit
- .const
- .switch
- .text (created by the assembler)

These sections are uninitialized:

- .bss (created by the assembler)
- .far
- .stack
- .systemem

Other sections appear on the list as well:

- .data (created by the assembler)
- .cio
- .pinit

Note The C/C++ compiler does not use this section.

When you highlight a section on the list, **Description** shows a brief description of the section. Also, **Placement** shows you where the section is currently allocated in memory.

Description

Provides a brief explanation of the contents of the selected entry in the **Compiler Sections** list.

Placement

Shows you where the selected **Compiler Sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments as defined in the physical memory map on the **Memory** pane. Select one of the listed memory segments to allocate the highlighted compiler section to the segment.

DSP/BIOS Sections

During program compilation, DSP/BIOS produces both uninitialized and initialized blocks of data and code. These blocks get allocated into memory as required by the configuration of your system. On the **DSP/BIOS sections** list you find both initialized (sections that contain data or executable code) and uninitialized (sections that reserve space in memory) sections.

Description

Provides a brief explanation of the contents of the selected **DSP/BIOS Sections** list entry.

Placement

Shows where the selected **DSP/BIOS Sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments available on C6000 processors, and changes based on the processor you are using.

DSP/BIOS Object Placement

Distinct from the entries on the **DSP/BIOS Sections** list, DSP/BIOS objects like STS or LOG, if your project uses them, are placed in the memory segment you select from the **DSP/BIOS Object Placement** list. All DSP/BIOS objects use the same memory segment. You cannot select the locations for individual objects.

Custom Sections

When your program uses code or data sections that are not included in either the **Compiler Sections** or **DSP/BIOS Sections** lists, you add the new sections to this list. Initially, the **Custom Sections** list contains no fixed entries, just a placeholder for a section for you to define.

Name

You enter the name for your new section here. To add a new section, click **Add**. Then replace the temporary name with the name to use. Although the temporary name includes a period at the beginning, you do not need to include the period in your new name. Names are case sensitive. `NewSection` is not the same as `newsection`, or `newSection`.

Placement

With your new section added to the **Name** list, select the memory segment to which to add your new section. Within the restrictions imposed by the hardware and compiler, you can select any segment that appears on the list.

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom Sections** list.

After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog box.

Remove

To remove a section from the **Custom Sections** list, select the section to remove and click **Remove**. The selected section disappears from the list.

To Create Memory Maps for Targets

Although each processor has memory map requirements, the C6000 DSP family of processors share some memory features and not others. Details of the memory sections and segments, as well as memory allocations and limitations for each processor, are provided in your documentation for CCS IDE and from TI.

To manage the memory on your processor, set the options within these panes to specify the memory allocation to use. Recall that the memory map is the result of the settings you provide for the options in the **Memory** and **Sections** panes in the C6000 Target Preferences dialog box.

Unfortunately, each processor has different needs, and the differences make it impossible to provide details about how you set the options for your target. You determine, from your model and code

- What memory segments you require
- Which sections you need and where
- Whether you need custom memory segments and sections
- Where to begin each memory segment and how much memory to allot to each segment
- Any other information that you need to set the options on the **Memory** and **Sections** panes?

After you configure the options in the C6000 Target Preferences dialog box, you are ready to set the Simulink configuration parameters for your model and generate code.

Using Target Support Package Software with Real-Time Workshop Embedded Coder Software

In this section...

“Introduction” on page 2-84

“To Use the Real-Time Workshop® Embedded Coder Target File” on page 2-84

Introduction

To take advantage of Real-Time Workshop Embedded Coder software features, you must migrate your models to a system target file called `ccslink_ert.tlc`. This target is based on the embedded real-time target (ERT) used by Real-Time Workshop Embedded Coder software. Other Embedded IDE Link target files are based on the generic real-time target (GRT).

To use Real-Time Workshop Embedded Coder software with Target Support Package software, you must choose the system target file `ccslink_ert.tlc`, available in the **System Target File Browser**.

If you simply choose the system target file `ccslink_ert.tlc` in the **System Target File Browser** directly to change the target for the model, all the **Embedded IDE Link** options are reset to default values by the switch. The C6000-specific options are the same between the two system target files.

You can set your model to use this system target file the usual way, via the **System Target File Browser**, available from the **Real-Time Workshop** pane in the Configuration Parameters dialog box. However, when you use the system target browser to switch your model between the ERT- and GRT-based TI C6000 system target files, the TI C6000-specific options (the configuration set) for the model are reset to default values.

To Use the Real-Time Workshop Embedded Coder Target File

For setting up a new model to use the ERT-based target `.tlc` file.

- 1** From your model menu bar, select **Simulation > Configuration Parameters**.
- 2** Click **Real-Time Workshop** on the **Select** tree to access the Real-Time Workshop software options.
- 3** Click **Browse** to open the **System Target File Browser**.
- 4** On the **System Target File Browser**, find and select the file `ccslink_ert.tlc`.
- 5** Click **OK**.

Targeting with DSP/BIOS Options

- “Introducing DSP/BIOS” on page 3-2
- “DSP/BIOS and Targeting Your C6000 DSP” on page 3-4
- “Code Generation with DSP/BIOS” on page 3-7
- “Profiling Generated Code” on page 3-11
- “Using DSP/BIOS with Your Target Application” on page 3-25
- “Generating Code for Any C64x+ Processor or Board” on page 3-27

Introducing DSP/BIOS

Target Support Package software supports DSP/BIOS features as options when you generate code for your target. In the following sections, read more about DSP/BIOS, how the target support package incorporates DSP/BIOS features into your generated code, and ways to use the real-time operating system (RTOS) features of DSP/BIOS in your applications. Follow these links for more information on specific areas that interest you, or read on for more details.

- “DSP/BIOS and Targeting Your C6000 DSP” on page 3-4
- “Code Generation with DSP/BIOS” on page 3-7
- “Profiling Generated Code” on page 3-11
- “Using DSP/BIOS with Your Target Application” on page 3-25

As a part of the Texas Instruments eXpressDSP™ technology, TI designed DSP/BIOS to include three components:

- DSP/BIOS Real-Time Analysis Tools — use these tools within Code Composer Studio IDE to view your program as it executes on the target in real time.
- DSP/BIOS Configuration Tool — enables you to add and configure any DSP/BIOS objects that you use to instrument your application. Use this tool to configure interrupt schedules and handlers, set thread priorities, and configure the memory layout on your DSP.
- DSP/BIOS Application Program Interface (API) — use C or assembly language functions to access DSP/BIOS functions by calling over 150 API functions. Target Support Package software uses the API to access DSP/BIOS.

You link these components into your application, directly or indirectly referencing only functions you need for your application to run efficiently and optimally. Only functions that you specifically reference become part of your code base. To avoid adding unused code to your project, the software excludes functions you do not reference. After you add DSP/BIOS functions, the configuration tool helps you disable features you do not need later, optimizing your program for speed and size.

For details about DSP/BIOS and what it can do for your applications, refer to your CCS IDE and DSP/BIOS documentation from Texas Instruments.

DSP/BIOS and Targeting Your C6000 DSP

In this section...
“Introduction” on page 3-4
“DSP/BIOS Configuration File” on page 3-5
“Memory Mapping” on page 3-6
“Hardware Interrupt Vector Table” on page 3-6
“Linker Command File” on page 3-6

Introduction

When you generate code from your DSP model, you can include DSP/BIOS features provided by Target Support Package software.

Including DSP/BIOS in your generated project adds the following files to your project:

- `modelname.tcf` — a DSP/BIOS configuration file
- `modelnamecfg.s62` — contains the DSP/BIOS objects required by your application and the vector table for the hardware interrupts.
- `modelnamecfg.h62` — the header file for `modelnamecfg.s62`.
- `modelnamecfg.h` — model configuration header file.
- `modelnamecfg_c.c` — source code for the model.
- `modelnamecfg.cmd` — the linker command file for the project. Adds the required DSP/BIOS libraries and the library `RTS6201.lib`, or the run-time support library for your target.

The executable code and source code you generate when you use the DSP/BIOS option are not the same as the code generated without DSP/BIOS included.

Instead of incorporating the DSP/BIOS files manually, as you would with CCS IDE, the Target Support Package software starts from your Simulink model and adds the DSP/BIOS files automatically. As it adds the files, the support package:

- Configures the DSP/BIOS configuration file for your model needs
- Sets up the objects you use to analyze your program while it runs on your target
- Handles memory mapping to optimize your code based on the blocks in your model

DSP/BIOS Configuration File

DSP/BIOS projects all have a file with the extension `.tcf`. The file contains the DSP/BIOS configuration information for your project, in the form of objects for instrumenting and scheduling tasks in the program code. A DSP/BIOS project can include the following files:

- Log (LOG) objects for logging events and messages (replace the `*printf` statements, for instance)
- Statistics (STS) objects for tracking the performance of your code
- A clock (CLK) object for configuring the clock on your target, and various memory functions
- Hardware and software interrupt (HWI, SWI) objects that control program execution
- Other objects you use to meet your needs

Your TI DSP/BIOS documentation can provide all the details about the objects and how to use them. In addition, your installed software from TI includes tutorials to introduce you to using DSP/BIOS in projects.

Not all of the DSP/BIOS objects get used by the code you generate from Target Support Package software. In the next sections, you learn about which objects the targeting software uses and how. You can still add more objects to your code through CCS IDE.

If you add DSP/BIOS objects beyond those provided by the target support package, you lose your additions when you regenerate code from your Simulink model.

Memory Mapping

Memory mapping that takes place in the linker command file now appears in the MEM object in the DSP/BIOS configuration file. Your memory sections, such as the DATA_MEM assignments and definitions, move to the MEM object, as do the memory segments. After completing this conversion, the memory assignment portions of your non-DSP/BIOS linker command file are not necessary in the linker command file.

Hardware Interrupt Vector Table

In non-DSP/BIOS project, the assembly language file `vector.asm` in your project defines the hardware interrupt vector table. This file defines which interrupts your project uses and what each one does.

When you use DSP/BIOS, the interrupts defined in the vector table move to the Hardware Interrupt Service Routine Manager in the CCS Configuration Tool. With your interrupts defined as Hardware Interrupts (HWD) in the Configuration Tool, your project does not need `vector.asm`, so the file does not appear in your DSP/BIOS enabled projects.

Linker Command File

After migrating your memory sections, segment, and hardware interrupt vector table to the configuration file, building with the DSP/BIOS option creates a compound linker command file. Because DSP/BIOS allows only one command file per project, and your linker file may comprise command options that did not relocate the DSP/BIOS configuration, Target Support Package software uses *compound* command files. Compound command files work to let your project use more than one command file.

By starting your original linker command file with the statement

```
"-lmodelnamecfg.cmd"
```

added as the first line in the file, your DSP/BIOS enabled project uses both your original linker command file and the DSP/BIOS command file. You get the features provide by DSP/BIOS as well as the custom command directives you need.

Code Generation with DSP/BIOS

In this section...
“Overview” on page 3-7
“Generated Code Without and With DSP/BIOS” on page 3-7

Overview

While generating code that includes the DSP/BIOS options is straightforward, changes occur between code that does not include DSP/BIOS and code that does. Two things change when you generate code with DSP/BIOS—files are added and removed from the project in CCS IDE, and DSP/BIOS objects become part of your generated code. With these in place, you can use the DSP/BIOS features in CCS IDE to debug your project, as well as use the profiling option in Target Support Package software to check the performance of your application running on your target.

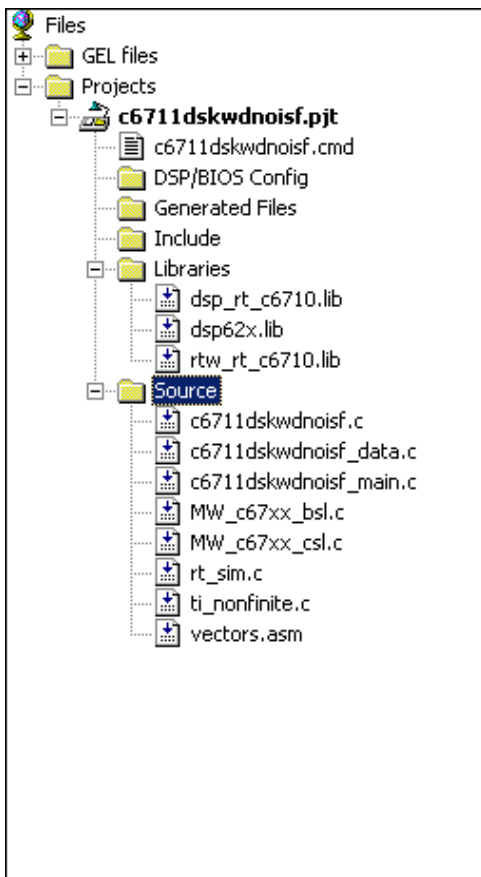
To generate code that includes DSP/BIOS options, open the Target Preferences block and select **DSP/BIOS** from the **Operating system** list on the **Board Info** pane.

Generated Code Without and With DSP/BIOS

The next two figures show the results of generating code without and with the DSP/BIOS option enabled in the **Simulation Parameters** dialog.

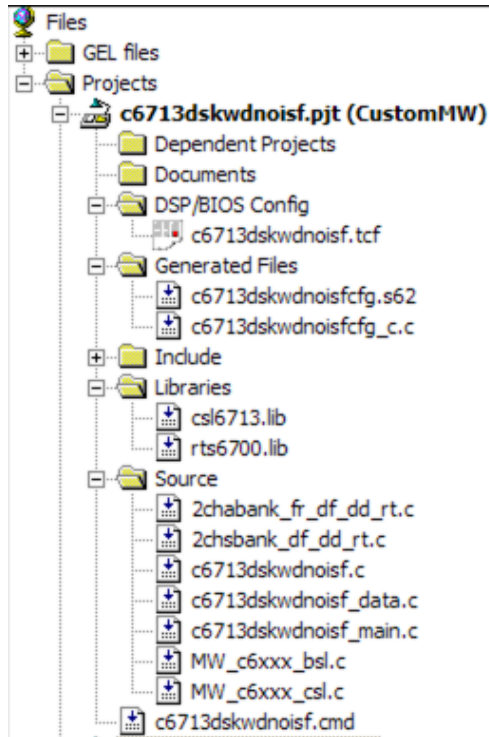
Example — c6713dskwdnoisf.pjt code Generated Without DSP/BIOS

When you create your project in CCS IDE, the folder structure looks like this.



Example — c6713dskwdnoisf.pjt Code Including DSP/BIOS

If you now create a project that includes DSP/BIOS, the folder structure for your project changes to look like the following figure.



Added File	Description
modelName.tcf	Contains the DSP/BIOS objects required by your application, and the vector table for the hardware interrupts
modelNamecfg.s62	Shows all the included files in your project, the variables, the DSP/BIOS objects, and more in this file generated from the .tcf file
modelNamecfg.h62	The header file for modelNamecfg.s62
modelNamecfg.h	Model configuration header file

Added File	Description
modelnamecfg_c.c	Source code for the model
modelnamecfg.cmd	The linker command file for the project. Adds the required DSP/BIOS libraries and the library RTS6201.lib or the run-time support library for your target.

Notice that the new folder includes some new files, shown in the next table.

With DSP/BIOS functions enabled for your project, the following files no longer appear in your project.

Filename	Description
vectors.asm	Defines the hardware interrupts (HWI) used by interrupt service routines on the processor. This file is removed after all of the hardware interrupts appear in the HWI section of the Configuration Tool.
Original linker command file—modelname.cmd	Assigns memory sections on the processor. This file is removed if the SECTION directive is empty because all of the section assignments moved to the configuration file. Otherwise, include call to the DSP/BIOS command file.
Some *.lib files	Provide access to libraries for the processor, and peripherals. These files are removed if their contents have been incorporated in the new compound linker command file.

When you investigate your generated code, notice that the function main portion of modelname_main.c includes different code when you generate DSP/BIOS-enabled source code, and modelname_main.c incorporates one or more new functions.

Profiling Generated Code

In this section...

“Overview” on page 3-11
“Profiling Subsystems” on page 3-12
“Details About Timing and Profiling” on page 3-13
“Profiling Multitasking Systems” on page 3-14
“The Profiling Report” on page 3-16
“Interrupts and Profiling” on page 3-17
“Reading Your Profile Report” on page 3-18
“Definitions of Report Entries” on page 3-19
“Profiling Your Generated Code” on page 3-21
“To Enable Profiling for Your Generated Code” on page 3-22
“To Create Atomic Subsystems for Profiling” on page 3-22

Overview

When you use Target Support Package software to generate code that incorporates the DSP/BIOS options, you can easily profile your generated code to gauge performance and find bottlenecks.

By selecting **Profile real-time execution** in the Real-Time Workshop software options, Real-Time Workshop software inserts statistics (STS) object instrumentation at the beginning and end of the code for each atomic subsystem in your model. (For more about STS objects, refer to your DSP/BIOS documentation from Texas Instruments.)

After your code has been running for a few seconds on your target, you can retrieve the profiling results from your target and display the information in a custom HTML report.

Code profiling works only on atomic subsystems in your model. To allow the target support package to profile your model when you build it in

Real-Time Workshop software, you convert segments of your model into atomic subsystems using **Create subsystem**.

By designating subsystems of your model as atomic, you force each subsystem to execute only when all of its inputs are available. Waiting for all the subsystem inputs to be available before running the subsystem allows the subsystem code to be profiled as a contiguous segment.

To enable the profile feature for your Simulink model, select **Simulation > Configuration Parameters** from the model menu bar. In the Configuration Parameters dialog box, select **Real Time Workshop > Embedded IDE Link**. Under **Code Generation**, enable **Profile real-time execution**.

Profiling Subsystems

Nested subsystems are profiled as part of their parent systems—the execution time reported for the parent subsystem includes the time spent in any profiled child subsystems. You cannot profile child subsystems separately.

For models that include multiple sample times, one or more subsystems in your model might not be included in the profiling process. When your model is configured to use single-tasking mode, all atomic subsystems in your model are profiled and appear in the report. When your model uses multitasking (refer to your Real-Time Workshop documentation for more about multitasking models) profiling applies only to single-rate subsystems that execute at the base rate of your model. This limitation arises because all of the generated code segments must execute contiguously for the profiling timing measurements to be correct. Setting the **Tasking mode for periodic sample times** to Auto in the model configuration parameters does not guarantee contiguous execution for all code segments and subsystems.

Notice two things in your code:

- STS objects are added to the generated code
- A generated DSP/BIOS configuration gets added to the project configuration file

Target Support Package software inserts and configures these objects specifically for profiling your code. You do not have to make changes to the

STS objects. To see the statistics objects in use, download your generated application to your board, select **DSP/BIOS > Statistics View** from the menu bar in CCS IDE, and run the board for a few seconds. You see the statistics being accumulated by the STS objects.

Details About Timing and Profiling

The profiling system in Target Support Package software relies on DSP/BIOS STS objects and the `CLK_gettime()` function. `CLK_gettime()` returns a high resolution timing counter that enables profiling to measure the instruction cycles the CPU spends executing code segments. To understand profiling, you need to understand how `CLK_gettime()` works.

This is how the system determines the value of `CLK_gettime()`:

```
CLK_gettime() return val = CLK_gettime() *PRD0 + CNT0
```

`PRD0` and `CNT0` are timer 0 period and counter registers. In code generation, BIOS allocates timer 0 as a system timer and set the timer to generate a timer interrupt every 1ms. `CLK_gettime()` in turn returns the number of BIOS system timer interrupts. By this logic, `PRD0` is set to the number of CPU clock cycles divided by the number of low resolution clock cycles that is equivalent to 1 millisecond in absolute time (8 low resolution clock cycles for C64x processors, for example).

The key point here is that function `CLK_gettime()` relies on the `CLK_gettime()` function which in turn relies on a timer 0 interrupt. If your process globally disables interrupts during code execution for more than 1 `PRD0` instruction cycle, one or more timer interrupts can be missed, resulting in a situation where both `CLK_gettime()` and `CLK_gettime()` can be inaccurate.

`CLK_gettime()` will be inaccurate because it does not report the correct time value. But it is always positive. The situation is worse for `CLK_gettime()`. It may report negative timing around code segments where interrupts are disabled:

```
A = CLK_gettime();
IRQ_globalDisable();
{
    Code segment;
```

```
}  
IRQ_globalEnable();  
B = CLK_gettime();
```

In this situation, if interrupts are disabled longer than 1ms around the code segment to be profiled, B might be smaller than A since CTN0 might have rolled over. So the count of the instruction cycles computed as (B - A) might be negative.

Correcting Inaccurate Profile Information Due to Timing

One way to correct problems in profiling caused by the disabled interrupts is to set the DSP/BIOS system timer interrupt to occur less frequently. As noted earlier, the timer is set to 1 millisecond by default.

You can change setting manually after you generate code for your project. Here are the steps to use to reset the DSP/BIOS system timer interval.

- 1 Open the .tcf file for the project.
- 2 Select **Scheduling > CLK Clock Manager**.
- 3 Right-click **CLK Clock Manager** to set the properties for the clock manager.
- 4 Change the **Microseconds/Int** value from the default 1000.00 microseconds to something larger, for example, 5000.00 microseconds.
- 5 Save the project.

This timing change reduces the chances of missing a system timer interrupt. If you do this and profile the code again, the profiling results are usually accurate. You can verify that if you reduce the system timer interrupt interval further, to perhaps 100 microseconds, you get less and less accurate profiling results, possibly reporting negative timing values.

Profiling Multitasking Systems

For a multitasking system, DSP/BIOS STS objects cannot reliably measure the time the processor spends in all tasks. When tasks can be preempted by other tasks (a result of multitasking operation), the profile timing measurements may be incorrect. For this reason, Target Support Package

software includes profiling instrumentation for atomic systems that run at the base sample rate only.

When you run the same model in single tasking mode, you can get the timing measurements for all the systems in your model for one iteration:

- 1** Select **Simulation > Configuration Parameters** from the model menu bar.
- 2** Under **Tasking** on the **Solver** pane, select **SingleTasking** for **Tasking mode for periodic sample times**.
- 3** Rebuild and execute your model on your C6000 hardware.

The program will probably overrun immediately since single tasking mode requires that all tasks complete within the base sample time which usually does not happen. However, all systems and subsystems do run once before the program terminates. This allows you to obtain profiling results for all systems.

When the overrun occurs, click **Halt** in CCS IDE to stop DSP/BIOS operation.

Then, enter `CCS_Obj.profile('report')` at the MATLAB prompt to report the statistics measurements.

Now you can view the timing measurements for each subsystem. Keep in mind that the percentages are given relative to the base sample time, so you must do some arithmetic to figure out whether a given system will fit in its available time interval. For instance, if your base sample time is 1 second, subsystem A executes every 3 seconds, the base-rate task takes 0.1 seconds to run, and A takes 2.5 seconds to run, the system should execute without overruns in multitasking mode.

If you change the overrun action option from its default setting of **Notify and halt** to **Notify and continue** or **None**, you can get measurements for multiple iterations of the system. Also, you will be able to request the profile report without first halting the CPU.

The Profiling Report

To help you measure subsystem performance, Target Support Package software provides a custom report that analyzes and displays the profile statistics. The report shows you the amount of time spent computing each subsystem, including **Outputs** and **Update** code segments, and provides links that open the corresponding subsystem in the Simulink model.

To view the profiling report, enter

```
profile(cc, 'report')
```

at the MATLAB prompt, where `cc` is the handle to your target and CCS IDE, and `report` is one of the input arguments for `profile`.

When you generate the report, the target support package stores the report in your code generation working folder, something like `modelName.c6000.rtw`, with the name `profileReport.html`.

If the MATLAB® software cannot find your code generation folder, the profile reports is stored in your temporary folder, `tempdir`. To locate your temporary folder, enter

```
tempdir
```

at the MATLAB command prompt.

Caution Each time you run the profiling process, the target support package replaces your existing report with a newer version. To save earlier reports, rename and save the report before you generate a new one, or change your destination temporary folder in the MATLAB workspace.

You must invoke `profile` after your Real-Time Workshop build, without clearing MATLAB memory between operations, so that stored information about the model is still available to the report generator. If you clear your MATLAB memory, information required for the profile report gets deleted and the report does not work properly. When this occurs, and if you have a CCS IDE project that was previously created with Real-Time Workshop software,

you must repeat the Real-Time Workshop build to see the subsystem-based profile analysis in the report.

Trace each subsystem presented in the profile report back to its corresponding subsystem in your Simulink model by clicking a link in the report. (The mapping from Simulink subsystems to generated system code is complex and thus not detailed here.) Inspect your generated code, particularly `modelname.c`, to determine where and how Simulink and Real-Time Workshop software implemented particular subsystems.

Within the generated code, you see entries like the following that define STS objects used for profiling.

```
STS_set(&stsSys0_Output, CLK_gethtime());
```

or

```
STS_delta(&stsSys0_Output, CLK_gethtime());
```

This pair of code examples perform the profiling of the code section that lies between them in `modelname.c`.

In CCS IDE, STS objects show up in the Statistics Object Manager section under **Instrumentation** in the `modelname.tcf` file. Double-click the file `modelname.tcf` in the CCS IDE tree view to open the file and see the sections.

In some cases, Real-Time Workshop software may have pruned unused data paths, causing related performance measurements to become meaningless. Reusable system code, or code reuse, where a single function is called from multiple places in the generated code, can exhibit extra measurements in the profile statistics, while the duplicate subsystem may not show valid measurements.

Interrupts and Profiling

Although there are STS objects that measure the execution time of the entire `mdlOutputs` and `mdlUpdate` functions, those measurements can be misleading because they do not include other segments of code that execute at each interrupt. Statistics for the SWI are used when calculating the headroom (the difference between the number of CPU cycles your process requires to complete and the number available for the process to complete,

which does not include the small overhead required for each interrupt. Note that profiling of multitasking systems does not measure the headroom. In addition, multitasking profiling does not use the SWI statistics.

To measure most accurately the overall application CPU usage, consider the DSP/BIOS IDL statistics, which measure time spent *not* doing application work. Your DSP/BIOS documentation from TI provides details about the various DSP/BIOS objects in the `tcf` file.

The interrupt rate for a DSP/BIOS application created by Target Support Package software is the fastest block execution rate in the model. The interrupt rate is usually, but not always, the same as the codec frame rate. When there is an upsampling operation or other rate increasing operation in your model, interrupts are triggered by a timer (PRD) object at the faster rate. You can determine the effective interrupt rate of the model by inverting the interrupt interval reported by the profiler.

Profiling subsystems that contain “blocking” device drivers, such as the ADC/DAC blocks and C6000 UDP Receive blocks may produce inaccurate and misleading results, raising values for **Max time spent in this subsystem per interrupt** and **Max percent of base interval** by many orders of magnitude. To avoid this problem, design subsystems to isolate blocking device drivers from algorithmic and other processing functions, and configure profiling appropriately.

Reading Your Profile Report

After you have the report from your generated code, you need to interpret the results. This section provides a link to sample report from a model and explains each entry in the report.

Sample of a Profile Report

When you click Sample Profile Report, the sample report opens in a new Help browser window. This opens the sample report in a new window so you can read the report and the descriptions of the report contents at the same time. Running the model `c6713dskwdnoisf` with DSP/BIOS generates the sample profile report. The next sections explain the headings in the report—what they mean and how they are measured (where that applies).

Report Heading Information

At the beginning of the report, profiling provides the name of the model you profiled, the target you used, and the date of the report. Since the report changes each time you run it, the date can be an important means of tracking model development.

Report Subsections and Contents

Within the body of your profile report, sections report the overall performance of your generated code and the performance of each atomic subsystem.

Report Heading	Description
Timing Constants	Shows you the base sample time in your model ($=1/\text{base rate in Hz}$) and the CPU clock speed used for the analysis.
Profiled Simulink Subsystems	Presents the statistics for each profiled subsystem separately, by subsystem. Each listing includes the STS object name or names that instrument the subsystem.
STS Objects	Lists every STS object in the generated code and the statistics for each. DSP/BIOS uses these objects to determine the CPU load statistics. For more information about STS objects, refer to your DSP/BIOS documentation from TI.

STS objects that are associated with subsystem profiling are configured for host operation at $4 \times x$, reflecting the numerical relationship between CPU clock cycles and high-resolution timer clicks, x . STS Average, Max, and Total measurements return their results in counts of instructions or CPU clock cycles.

Definitions of Report Entries

In the following sections, we provide definitions of the entries in the profile report. These definitions help you decipher the report and better understand how your process is performing.

System name

Provides the name of the profiled model, using the form *targetnameprofile*. *targetname* is the processor or board assigned as the target, via the target preferences block.

Number of Iterations Counted

The number of interrupts that occurred between the start of model execution and the moment the statistics were obtained.

CPU Clock Speed

The instruction cycle speed of your digital signal processor. On the C6713 DSK, you can adjust this speed to one of four values, where 100 MHz is the default—25, 33.25, 100, 133 MHz. If you change the speed to something other than the default setting of 100 MHz, you must specify the new speed in the Real-Time Workshop software options. Use the **Current C6713DSK CPU clock rate** option on the TIC6000 runtime category on the Real-Time Workshop tab.

Set at a fixed 150 MHz, you cannot change the CPU clock rate on the C6713 DSK. You do not need to report the setting in the Real-Time Workshop software options.

Maximum Time Spent in This Subsystem per Interrupt

The amount of time spent in the code segment corresponding to the indicated subsystem in the worst case. Over all the iterations measured, the maximum time that occurs is reported here. Since the profiler only supports single-tasking solver mode, no calculation can be preempted by a new interrupt. All calculations for all subsystems must complete within one interrupt cycle, even for subsystems that execute less often than the fastest rate.

Maximum Percent of Base Interval

The worst-case execution time of the indicated subsystem, reported as a percentage of the time between interrupts.

STS Objects

Profiling uses STS objects to measure the execution time of each atomic subsystem. STS objects are a feature of the DSP/BIOS run-time analysis tools, and one STS object can be used to profile exactly one segment of code. Depending on how Real-Time Workshop software generates code for each subsystem, there may be one or two segments of code for the subsystem; the computation of outputs and the updating of states can be combined or separate. Each subsystem is assigned a unique index, *i*. The name of each STS object helps you determine the correspondence between subsystems and STS objects. Each STS object has a name of the form

`stsSysi_segment`

where *i* is the subsystem index and *segment* is `Output`, `Update`, or `OutputUpdate`. For example, in the sample profile report shown in the next section, the STS objects have the names `stsSys1_OutputUpdate`, and `stsSys2_OutputUpdate`.

Profiling Your Generated Code

Before profiling your generated code, you must configure your model and Real-Time Workshop software to support the profiling features in Target Support Package software. Your model must use DSP/BIOS features for profiling to work fully.

The following tasks compose the process of profiling the code you generate.

- 1 Enable DSP/BIOS for your code.
- 2 Enable profiling in the Real-Time Workshop software.
- 3 Create atomic subsystems to profile in your model.
- 4 Build, download, and run your model.
- 5 Use `profile` to view the MATLAB profile report.

To demonstrate profiling generated code, this procedure uses the wavelet denoising model `c6713dskwdnoisf.mdl` that is included with Target Support Package demo programs. If you are using the C6713 DSK as your target, use

the model `C6713dskwdnoisf` throughout this procedure. Simulators work as well, just choose the appropriate model for your simulator.

Begin by loading the model, entering

```
c6713dskwdnoisf
```

at the MATLAB prompt. The model opens on your desktop.

To Enable Profiling for Your Generated Code

Recall that you must use DSP/BIOS in your code to use profiling.

To enable the profile feature for your Simulink model, select **Simulation > Configuration Parameters** from the model menu bar. In the Configuration Parameters dialog box, select **Real Time Workshop > Embedded IDE Link**. Under **Code Generation**, enable **Profile real-time execution**.

To Create Atomic Subsystems for Profiling

Profiling your generated code depends on two features—DSP/BIOS being enabled and your model having one or more subsystems defined as atomic subsystems. To learn more about subsystems and atomic subsystems, refer to your Simulink documentation in the Help browser.

In this tutorial, you create two atomic subsystems—one from the Analysis Filter Bank block and a second from the Soft Threshold block:

- 1 Select the Analysis Filter Bank block. Select **Edit > Create subsystem** from the model menu bar. The name of the block changes to subsystem. Repeat for the Soft Threshold block.
- 2 To convert your new subsystems to atomic subsystems, right-click on each subsystem and choose **Subsystem parameters...** from the context menu.
- 3 In the **Block Parameters: Subsystem** dialog for each subsystem, select the **Treat as atomic unit** option. Click **OK** to close the dialog. If you look closely you can see that the subsystems now have heavier borders to distinguish them from the other blocks in your model.

To Build and Profile Your Generated Code

You have enabled profiling in your model and configured two atomic subsystems in the model as well. Now, use the profiling feature in the target support package to see how your code runs and check the performance for bottlenecks and slowdowns as the code runs on your target.

Caution Do not click on any other open model while you are profiling your model. Clicking on another open model can cause profiling to fail with an error message like “Invalid Simulink object specifier.”

1 Select **Tools > Real-Time Workshop > Build Model**.

If you did not use the Real-Time Workshop software options to automate model compiling, linking, downloading, and executing, perform these tasks using the **Project** options in CCS IDE.

Allow the application to run for a few seconds or as long as necessary to execute the model segments of interest a few times. Then stop the program.

2 Create a link to CCS IDE by entering

```
cc = ticcs;
```

at the MATLAB prompt.

3 Enter

```
profile(cc, 'report')
```

at the prompt to generate the profile report of your code executing on your target.

The profile report appears in the Help browser. It should look very much like the following sample report; your results may differ based on your target and the settings in the model.

Profile Report

Simulink model: [c6416dskprofile.mdl](#)
Target: C6416DSK

Report of profile data from Code Composer Studio (tm)
XX-XXX-2005 17:27:27

Timing constants

Base sample time	250 ms
CPU Clock speed ¹	720 MHz

Profiled Simulink Subsystems

System name	c6416dskprofile
STS object	stsSys2_OutputUpdate
Max time spent in this subsystem per interrupt	14.93 μ s
Max percent of base interval	0.00597%
Number of iterations counted	144

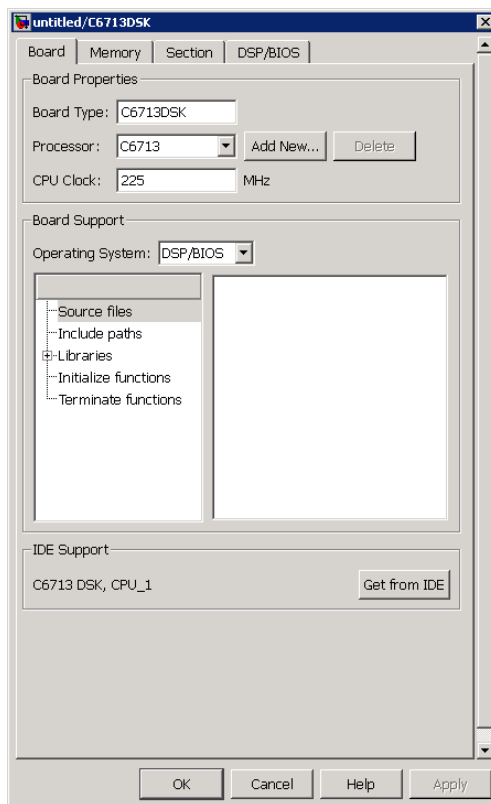
System name	c6416dskprofile/Subsystem1
STS object	stsSys1_OutputUpdate
Max time spent in this subsystem per interrupt	12.8 μ s
Max percent of base interval	0.00512%
Number of iterations counted	144

Using DSP/BIOS with Your Target Application

Enabling DSP/BIOS When You Generate Code

For any code you generate using Real-Time Workshop software and Target Support Package software, you have the option of including DSP/BIOS features automatically when you generate the code. Incorporating the features requires you to select one option in the Target Preferences/Custom Board block—DSP/BIOS for the operating system.

- 1** Open the model to use to generate code.
- 2** Open the Target Preferences block in your model.
- 3** On the **Board Info** pane, select DSP/BIOS for **Operating system** under the Code Generation options.



4 As shown in the figure, select DSP /BIOS for **Operating system**.

Generating Code for Any C64x+ Processor or Board

Unlike other Target Preferences blocks, the C64x+ Target Preferences block imports hardware information directly from DSP/BIOS. This feature enables you to create custom target preferences blocks for any C64x+ CPU core-based processor or board. You create and reuse these target preferences blocks in your models to generate code for your C64x+ processor or board.

To create a custom target preferences block for your C64x+ processor or board:

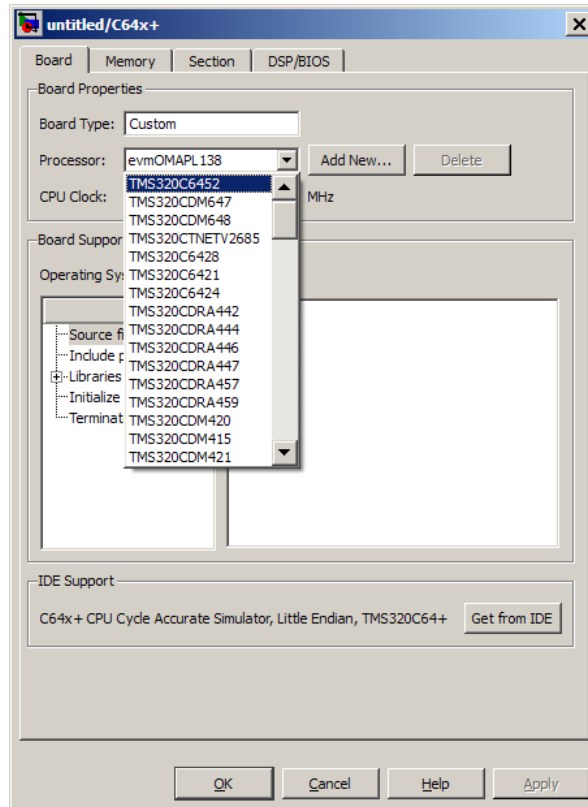
- 1** In MATLAB, enter `c6000tgtpreflib`. This command opens the C6000 Target Preferences library.
- 2** Drag the C64x+ Target Preferences block to your Simulink model.
- 3** Double-click the C64x+ Target Preferences block. This action opens the block.

When you open the C64x+ Target Preferences block, the Embedded IDE Link software uses the `BIOS_INSTALL_DIR` environment variable to locate the DSP/BIOS installation folder. The link software then queries DSP/BIOS for a list of processors and boards with C64x+ cores and displays them in the **Processor** list.

Note If you change DSP/BIOS versions in the CCS Component Manager, reopen the block to display the updated **Processor** list.

- 4** Select your processor from the **Processor** list.

The C64x+ Target Preferences block imports settings from DSP/BIOS such as the memory map, cache settings, and CPU clock rate. The block applies the settings to the **Memory**, **Section** and **DSP/BIOS** tabs.



- 5 Set the CPU Clock rate for your processor.
- 6 To improve the efficiency of your application, you can adjust the L1 and L2 caches values and the compiler sections. The following section provides an example of how to adjust these settings.
- 7 Click **OK**.

To create a library of custom target preferences blocks:

- 1 In Simulink, create a library: **File>New>Library**.
- 2 Copy any custom target preferences blocks from your models to the library.

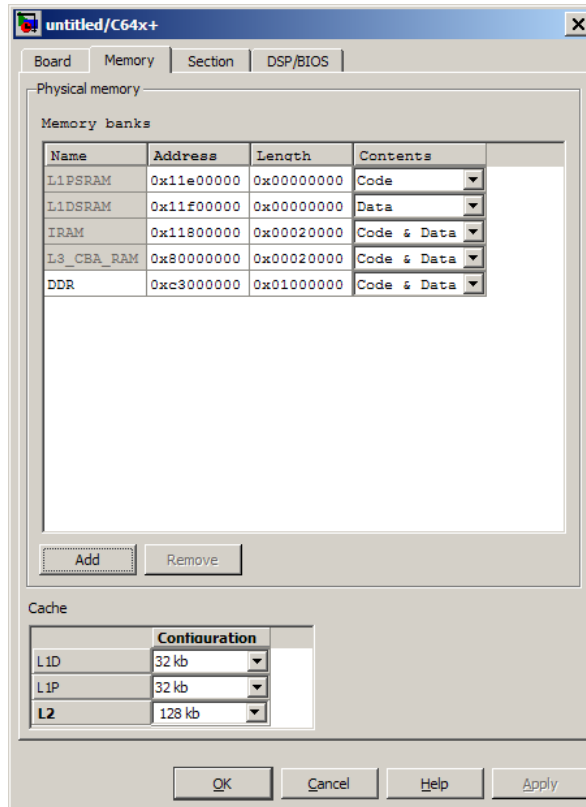
- 3** Relabel individual blocks for the processors they specify: Click the block labels and edit the text.
- 4** Save the library to your default current folder. For example, save it as `c64xcustomtgtpreflib.mdl` in `c:\Program Files\matlab\bin`.

Later, you can reopen the library by entering the library name on the MATLAB command line.

Example: Creating a Custom Target Preferences Block for OMAP-L138/C6748 EVM

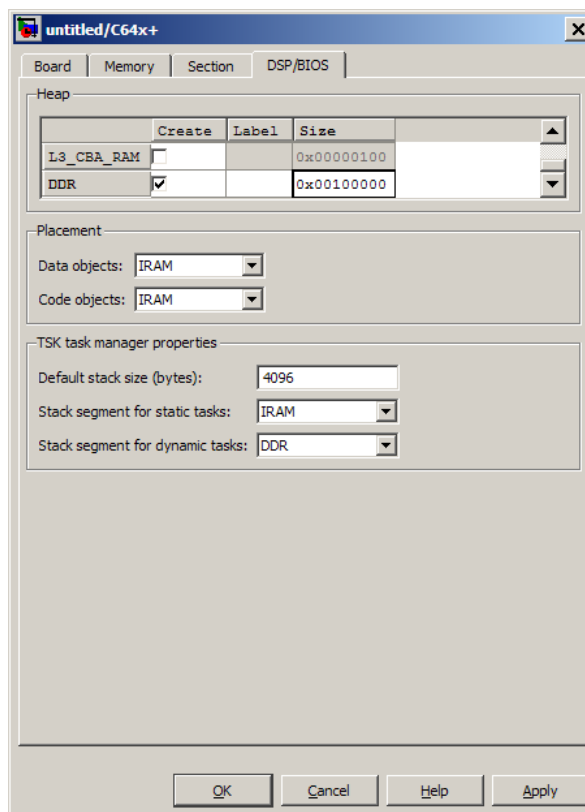
To create a custom Target Preferences Block for OMAP-L138/C6748 EVM:

- 1** On the MATLAB command line, enter `c6000tgtpreflib` to open the C6000 Target Preferences library.
- 2** Copy the C64x+ Target Preferences block to your model.
- 3** Open the C64x+ Target Preferences block.
- 4** For **Processor**, select `evmOMAPL138`. The block populates the **Memory**, **Section** and **DSP/BIOS** tabs with default values for that processor.
- 5** Adjust the cache settings for efficiency. Turn L1 and L2 cache on:
 - a** Click the **Memory** tab. The L1P, L1D, and L2 cache sizes are zero by default.
 - b** Set **L1D** and **L1P** cache sizes to 32 kb.
 - c** When you increase the cache size, decrease the **L1PSRAM** and **L1DSRAM** to accommodate memory taken from the high address range of the corresponding L1 memory segments. Because L1PSRAM and L1DSRAM are 32 kb in the OMAP-L138/C6748 processor, change the **Length** for **L1PSRAM** and **L1DSRAM** to `0x00000000`. This setting allows you to use the entire L1 memory as level one cache.
 - d** Set **L2** cache size to 128 kb. Decrease the **IRAM** length to `0x00020000` (128 kb).
 - e** Confirm the following configuration for the **Memory** tab, and click **Apply**.



- 6 Reassign compiler sections to optimize the efficiency of the generated code. Put `.stack`, `.bios`, `.hwi`, `.hwi_vec` sections into fast internal memory and assign everything else to external memory. This approach avoids linking errors caused by placing excessive code and data in limited internal memory. It also runs critical sections of the application from internal memory.
 - a Click the **Section** tab.
 - b In the **Compiler sections** list, set the **Placement** of every section, except `.stack`, `.bios`, `.hwi`, and `.hwi_vec`, to DDR.
 - c Set the **Placement** of `.stack`, `.bios`, `.hwi`, and `.hwi_vec` to IRAM.

- 7** The final step in configuring the OMAP-L138/C6748 EVM Target Preferences block is to create a heap in external memory. Device drivers from TI use heap to allocate data structures and device driver buffers. Without a heap, integrating device drivers from TI would not be possible.
- a** Click the **DSP/BIOS** tab.
 - b** In the **Heap** list, select **DDR** and set the heap size to 0x00100000 (1 MB).
 - c** Confirm the following configuration for the **DSP/BIOS** tab, and click **OK**.



Using the C62x and C64x DSP Libraries

- “About the C62x and C64x DSP Libraries” on page 4-2
- “Fixed-Point Numbers” on page 4-5
- “Building Models” on page 4-10

About the C62x and C64x DSP Libraries

In this section...
“C62x DSP Library” on page 4-2
“C64x DSP Library” on page 4-3
“Supported Platforms” on page 4-3
“Characteristics Common to C62x and C64x Library Blocks” on page 4-4

C62x DSP Library

Blocks in the C62x DSP library correspond to functions in the Texas Instruments TMS320C62x DSP Library assembly-code library, which target the TI C62x family of digital signal processors. Use these blocks to run simulations by building models in Simulink software before generating code. Once you develop your model, you can invoke Real-Time Workshop software to generate code that is optimized to run on C6713 DSK development platforms or C62x hardware. (Fixed-point processing on C67x hardware is identical to C62x fixed point hardware and processing so you can develop on the C67x for the C62x.) During code generation, each C62x DSP Library block in your model is mapped to its corresponding TMS320C62x DSP Library assembly-code routine to create target-optimized code.

C62x DSP Library blocks generally input and output fixed-point data types. Chapter 6, “Block Reference” discusses the data types accepted and produced by each block in the library. “Fixed-Point Numbers” on page 4-5 gives a brief overview of using fixed-point data types in Simulink software. For an in-depth discussion of fixed-point data types, including issues with scaling and precision when you perform fixed-point operations, refer to your “Fixed-Point Toolbox” documentation.

You can use C62x DSP Library blocks with certain blocks from the Signal Processing Blockset software and Simulink software. To learn more about creating models that include both C62x DSP Library blocks and blocks from other blocksets, refer to “Building Models” on page 4-10.

C64x DSP Library

Blocks in the C64x DSP library correspond to functions in the Texas Instruments TMS320C64x DSP library assembly-code library, which target the TI C64x family of digital signal processors. Use these blocks to run simulations by building models in Simulink software before generating code. Once you develop your model, you can invoke Real-Time Workshop software to generate code that is optimized to run on the C6416 DSK development platform or other C64x hardware. During code generation, each C64x DSP Library block in your model is mapped to its corresponding TMS320C64x DSP Library assembly-code routine to create target-optimized code.

C64x DSP Library blocks generally input and output fixed-point data types. Chapter 6, “Block Reference” discusses the data types accepted and produced by each block in the library. “Fixed-Point Numbers” on page 4-5 gives a brief overview of using fixed-point data types in Simulink software. For an in-depth discussion of fixed-point data types, including issues with scaling and precision when you perform fixed-point operations, refer to your Fixed-Point Toolbox™ documentation.

You can use C64x DSP Library blocks with certain blocks from the Signal Processing Blockset software and Simulink software. To learn more about creating models that include both C64x DSP Library blocks and blocks from other blocksets, refer to “Building Models” on page 4-10.

Note While you can use C62x blocks on C64x targets, the generated code is not optimal for the C64x target. Using the appropriate C64x block creates better optimized code. (Target Support Package software generates a warning message when you try to do this but allows you to use the block.)

Supported Platforms

The C62x and C64x DSP libraries can be used with the platforms listed in the following table:

Library	Supported platforms
C62x	C62x, C67x, C67x+, C64x, C64x+
C64x	C64x, C64x+

Characteristics Common to C62x and C64x Library Blocks

The following characteristics are common to all C62x and C64x DSP Library blocks:

- All blocks inherit sample times from driving blocks.
- The blocks are single rate.
- Block filter weights and coefficients are tunable, but not in real time. Other block parameters are not tunable.
- All blocks support discrete sample times. Individual block reference pages indicate blocks that also support continuous sample times.

To learn more about characteristics particular to each block in the library, refer to Chapter 6, “Block Reference”

Fixed-Point Numbers

In this section...

“Notation” on page 4-5

“Signed Fixed-Point Numbers” on page 4-6

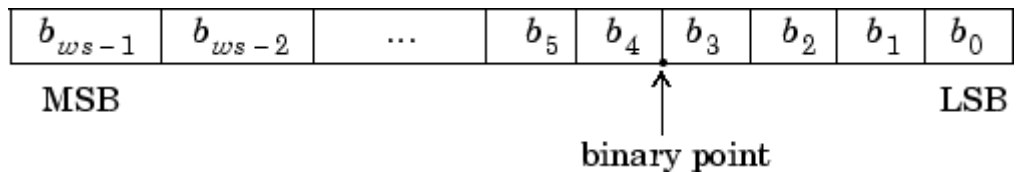
“Q Format Notation” on page 4-6

Notation

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1’s and 0’s). How hardware components or software functions interpret this sequence of 1’s and 0’s is defined by the data type.

Binary numbers are represented as either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a fractional fixed-point number (either signed or unsigned) is shown below.



where

- b_i is the i th binary digit.
- ws is the word size in bits.
- b_{ws-1} is the location of the most significant (highest) bit (MSB).
- b_0 is the location of the least significant (lowest) bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example the number is said to have four fractional bits, or a fraction length of four.

Note For Target Support Package, the results of fixed-point and integer operations in MATLAB/Simulink match the results on the hardware target down to the least significant bit (bit-trueness). The results of floating-point operations in MATLAB/Simulink do not match those on the hardware target, because the libraries used by the third-party compiler may be different from those used by MATLAB/Simulink.

Signed Fixed-Point Numbers

Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude
- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and the one TI digital signal processors use.

Negation using signed two's complement representation consists of a bit inversion (translation into one's complement) followed by the binary addition of a one. For example, the two's complement of 000101 is 111011:

000101 ->111010 (bit inversion) ->111011 (binary addition of 1 to the LSB)

results in the negative of 000101 being 111011.

Q Format Notation

The position of the binary point in a fixed-point number determines how you interpret the scaling of the number. When performing arithmetic such as addition or subtraction, hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits have no knowledge of a binary point. They perform signed or unsigned integer arithmetic—as if the binary point is to the right of the LSB (b_0). Therefore, you determine the binary point in your code.

In the C62x DSP Library, the position of the binary point in signed, fixed-point data types is expressed in and designated by Q format notation. This fixed-point notation takes the form

$$Q_{m.n}$$

where

- Q designates that the number is in Q format notation—the Texas Instruments notation for signed fixed-point numbers.
- m is the number of bits used to designate the two's complement integer portion of the number.
- n is the number of bits used to designate the two's complement fractional portion of the number, or the number of bits to the right of the binary point. Sometimes n is called the scale factor.

Q format always designates the most significant bit of a binary number as the sign bit. Representing a signed fixed-point data type in Q format requires $m+n+1$ bits to account for the sign.

Example — Q.15

For example, a signed 16-bit number with $n = 15$ bits to the right of the binary point is expressed as

$$Q_{0.15}$$

in this notation. This is (1 sign bit) + (0 = m integer bits) + (15 = n fractional bits) = 16 bits total in the data type. In Q format notation the $m = 0$ is often implied, as in

$$Q.15$$

In Fixed-Point Toolbox software, this data type is expressed as

`sfrac16`

or

`sfix16_En15`

Filter Design Toolbox™ software expresses this data type as the vector

[16 15]

meaning the word length is 16 bits and the fraction length is 15 bits.

Example – Q1.30

Multiplying two Q.15 numbers yields a product that is a signed 32-bit data type with 30 bits to the right of the binary point. One bit is the designated sign bit, forcing m to be 1:

$$m+n+1 = 1+30+1 = 32 \text{ bits total}$$

Therefore this number is expressed as

Q1.30

In Fixed-Point Toolbox software, this data type is expressed as

`sfixed32_En30`

In Filter Design Toolbox software, this data type is expressed as

[32 30]

Example – Q-2.17

Consider a signed 16-bit number with a scaling of $2^{(-17)}$. This requires $n = 17$ bits to the right of the binary point, meaning the most significant bit is a *sign-extended* bit.

Sign extension adds bits to the high end (MSB end) of the word and fills the added bits with the value of the MSB. For example, consider a 4-bit two's complement number 1011. Extending the number to 7 bits with sign extension changes the number to 1111011—the value of the number remains the same.

One bit is the designated sign bit, forcing m to be -2.

$$m+n+1 = -2+17+1 = 16 \text{ bits total}$$

Therefore this number is expressed as

Q-2.17

In Fixed-Point Toolbox software, this data type is expressed as

`sfix16_En17`

To express this data type in Filter Design Toolbox software, use

`[16 17]`

Example – Q17.-2

Consider a signed 16-bit number with a scaling of 2^2 or 4. The binary point is implied to be 2 bits to the right of the 16 bits, or that there are $n = -2$ bits to the right of the binary point. One bit must be the sign bit, forcing m to be 17.

$$m+n+1 = 17+(-2)+1 = 16$$

Therefore this number is expressed as

Q17.-2

In Fixed-Point Toolbox software, this data type is expressed as

`sfix16_E2`

In Filter Design Toolbox software, this data type is expressed as

`[16 -2]`

Building Models

In this section...
“Overview” on page 4-10
“Converting Data Types” on page 4-10
“Using Sources and Sinks” on page 4-11
“Choosing Blocks to Optimize Code” on page 4-11

Overview

You can use C62x or C64x DSP Library blocks in models along with certain core Simulink and Signal Processing Blockset software. This section discusses issues you should consider when you build models with blocks from these libraries.

Converting Data Types

Any blocks you connect in a model have compatible input and output data types. In most cases, C62x or C64x DSP Library blocks handle only a limited number of specific data types. Refer to any block reference page in Chapter 6, “Block Reference” for a discussion of the data types that each block accept and produces.

When you connect C62x or C64x DSP Library blocks and Simulink blocks, you often need to set the data type and scaling in the block parameters of the Simulink block to match the data type of the C62x DSP Library block. Many Simulink blocks allow you to set their data type and scaling by inheriting from the driving block, or by back propagating from the next block. This can be a good way to set the data type of a Simulink block to match a connected C62x DSP Library block.

Some Signal Processing Blockset software blocks and Simulink blocks also accept fixed-point data types. Make the appropriate settings in the block parameters when you connect them to a C62x DSP Library block.

To use Signal Processing Blockset software or core Simulink blocks that do not handle fixed-point data types with C62x DSP Library blocks in your model, you must use an appropriate data type conversion block:

- To connect fixed-point and nonfixed-point blocks, use the Data Type Conversion block from the Simulink Data Type library.
- To provide an interface to nonfixed-point blocks, use the C62x Convert Floating-Point to Q.15 and C62x Convert Q.15 to Floating-Point blocks from the C62x DSP Library.
- To connect blocks of varying nonfixed-point data types in your model, use the Data Type Conversion block from the Signals and Systems Simulink library
- To connect blocks of varying fixed-point data types in your model, use the Data Type Conversion Inherited block from the Simulink Data Type library.

Refer to the reference pages for these blocks or invoke the Help system from their block dialogs for more information.

Using Sources and Sinks

The C62x DSP Library does not include source or sink blocks. Use source or sink blocks from the core Simulink library or Signal Processing Blockset software in your models with C62x DSP Library blocks. See “Converting Data Types” on page 4-10 for more information on incorporating blocks from other libraries into your models.

Choosing Blocks to Optimize Code

In some cases, blocks that perform similar functions appear in more than one blockset. For example, the C62x DSP Library, the C64x DSP Library, and the Signal Processing Blockset software all have Autocorrelation blocks. How do you choose which to include in your model? If you are building a model to run on the C6713 DSK or on C62x hardware, choosing the block from the C62x DSP Library always yields better optimized code. You can use a similar block from another library if it provides functionality that the C62x DSP Library block does not support, but you generate less well optimized code.

In the same manner, if you are building a model to run on the C6416 DSK or on C64x hardware, choosing the block from the C64x DSP Library always yields better optimized code. You can use a similar block from another library if it provides functionality that the C64x DSP Library block does not support, but you generate less well optimized code.

Configuring Timing Parameters for CAN Blocks

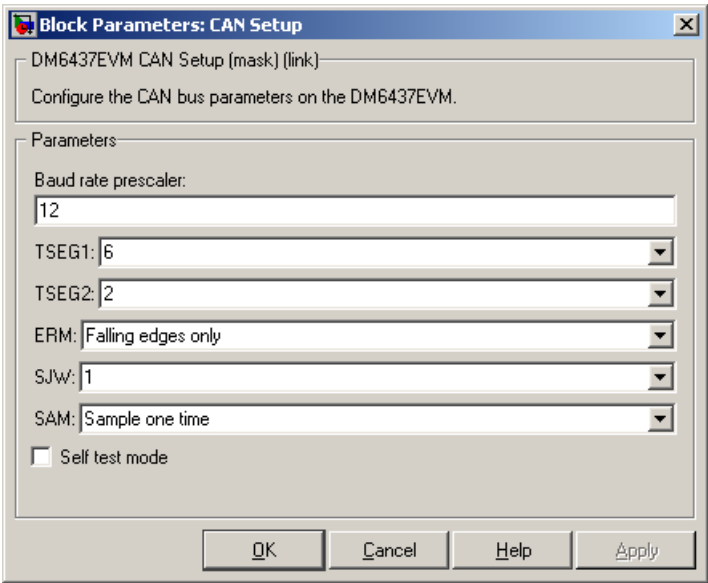
Setting Timing Parameters

In this section...

- “Accessing the Timing Parameters” on page 5-2
- “Determining Timing Parameter Values” on page 5-3
- “CAN Bit Timing Example” on page 5-4

Accessing the Timing Parameters

The timing parameters that control the bit rate for DM643x CAN Receive and DM643x CAN Transmit blocks are **Baud rate prescaler**, **TSEG1**, and **TSEG2** in the DM643x CAN Setup block.



The following sections describe how to set these parameters.

Determining Timing Parameter Values

The following steps show you how to determine the appropriate values to use for the timing parameters.

1 Gather these two values:

- Bit rate of the CAN network
- **SYSCLKOUT** — This is equivalent to the CAN module system clock frequency. The CAN peripheral in the DM6437 is in the CLKIN clock domain, which operates at the same frequency as the primary reference clock to the DSP. In the DM6437EVM board, the primary reference clock operates at 27 MHz.

2 Estimate the value of the **Baud rate prescaler (BRP)** and then solve this equation for BitTime:

$$\text{BitTime} = \text{SYSCLKOUT} / (\text{BRP} * \text{Bit rate})$$

3 Estimate values for **TSEG1** and **TSEG2** that satisfy the following equation:

$$\text{BitTime} = \text{TSEG1} + \text{TSEG2} + 1$$

The estimated values must also satisfy the following constraints:

$$\text{TSEG1} \geq \text{TSEG2}$$

$$\text{IPT (Information Processing Time)} = 3 / \text{BRP}$$

$$\text{IPT} \leq \text{TSEG1} \leq 16 \text{ TQ}$$

$$\text{IPT} \leq \text{TSEG2} \leq 8 \text{ TQ}$$

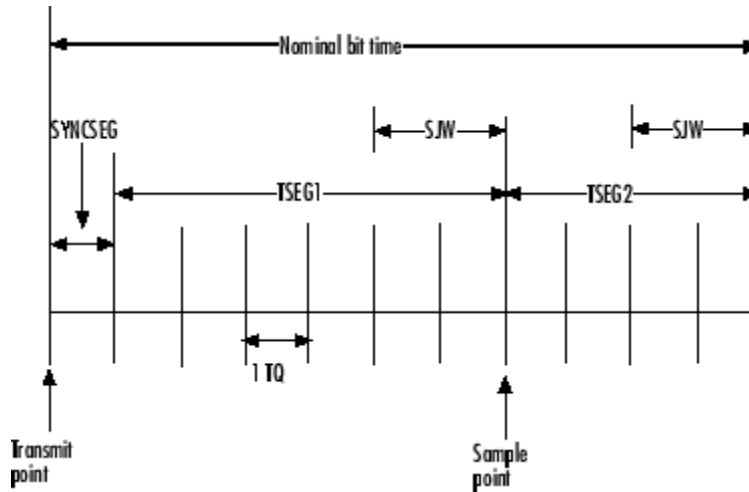
$$1 \text{ TQ} \leq \text{SJW} \leq \min(4 \text{ TQ}, \text{TSEG2})$$

where:

IPT is Information Processing Time, **TQ** is Time Quanta, and **SJW** is Synchronization Jump Width, which can be set in the **CAN Setup** block.

4 Iterate steps two and three until the values selected for **TSEG1**, **TSEG2**, and **BRP** meet all of the criteria.

The following illustration shows the relationship between the parameters:



CAN Bit Timing Example

This example shows how to determine appropriate CAN timing parameters.

Assume that $\text{SYSCLKOUT} = 27 \text{ MHz}$, and a Bit rate of 1 Mbits/s is required.

- 1 With the **Baud rate prescaler (BRP)** set to 12, substitute the values of Bit rate, *BRP*, and *SYSCLKOUT* into the following equation, solving for *BitTime*:

$$\text{BitTime} = \text{SYSCLKOUT} / (\text{BRP} * \text{Bit rate})$$

$$\text{BitTime} = 27\text{MHz} / (12 * 0.25 \text{ Mbits/sec}) = 9\text{TQ}$$

- 2 Set the values of **TSEG1** and **TSEG2** to 6TQ and 2TQ, respectively. Substitute the values of *BitTime* from the previous equation, and the chosen values for *TSEG1* and **TSEG2** into the following equation:

$$\text{BitTime} = \text{TSEG1} + \text{TSEG2} + 1$$

$$9\text{TQ} = 6\text{TQ} + 2\text{TQ} + 1$$

3 Finally, check the selected values against the rules:

$$\text{IPT} = 3/\text{BRP} = 3/12 = .25$$

$$\text{IPT} \leq \text{TSEG1} \leq 16 \text{ TQ True! } .25 \leq 6\text{TQ} \leq 16\text{TQ}$$

$$\text{IPT} \leq \text{TSEG2} \leq 8\text{TQ True! } .25 \leq 2\text{TQ} \leq 8\text{TQ}$$

$$1\text{TQ} \leq \text{SJW} \leq \min(4\text{TQ}, \text{TSEG2}), \text{ as a result of which SJW can be set to either 1 or 2.}$$

4 All chosen values satisfy the criteria, so no further iteration is necessary.

The following table provides common timing parameter settings for typical values of Bit rate and SYSCLKOUT = 27 MHz. This clock frequency is the maximum for the DM6437 EVM blocks.

Bit rate	TSEG1	TSEG2	Bit Time	BRP	SJW
250 Kbits/sec	6	2	3	12	1 or 2
500 Kbits/sec	3	1	6	9	1
1 Mbits/sec*	6	2	9	3	1 or 2
2 Mbits/sec*	1	1	4.5	3	ERROR

* 3-time sampling in the DM643x CAN module is not possible at this Bit rate. In the DM643x CAN Setup block, the **SAM** parameter cannot be set to Sample three times.

References

For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981A, available at the Texas Instruments Web site.

See Also

DM643x CAN Setup, DM643x CAN Transmit

Block Reference

AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437) (p. 6-2)	Work with DM6437 EVM boards
C6416 DSK (c6416dsklib) (p. 6-3)	Work with C6416 DSK boards
C6455 EVM (c6455evmlib) (p. 6-4)	Work with SRIO on C6455 EVM boards
C6713 DSK (c6713dsklib) (p. 6-4)	Work with C6713 DSK boards
C6747 EVM (c6747evmlib) (p. 6-5)	Work with DM648 EVM boards
TCI6428 DSK (tc6428dsklib) (p. 6-5)	Work with DM648 EVM boards
C6727 PADK (c6727padklib) (p. 6-5)	Work with DM648 EVM boards
Custom C6000 (c6000customlib) (p. 6-6)	Work with DM648 EVM boards
Custom C64x+ (c64xpcustomlib) (p. 6-6)	Work with DM648 EVM boards
CAN Message Handling Blocks (canmsglib) (p. 6-7)	Host CAN blocks
DM642 EVM (dm642evmlib) (p. 6-7)	Work with DM642 EVM boards
DM6437 EVM (dm6437evmlib) (p. 6-7)	Work with DM6437 EVM boards
DM648 EVM (dm648evmlib) (p. 6-9)	Work with DM648 EVM boards
DSP/BIOS (dsplib) (p. 6-9)	Work with C6000 models to provide DSP/BIOS tasks and interrupts
C62x DSP Library (tic62dsplib) (p. 6-9)	Work with C62x processors

C64x DSP Library (tic64dsplib) (p. 6-12)	Work with C64x processors
Scheduling (c6000dspcorelib) (p. 6-14)	Work with all C6000 processors
Target Communication (targetcommplib) (p. 6-14)	Work with C6000 processor and board models that communicate with hosts such as xPC Target or host-side models
Target Preferences (c6000tgtppreflib) (p. 6-15)	Configure models for code generation and targeting

AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)

C6000 Deinterleave	Separate interleaved YCbCr 4:2:2 data into Y, Cb, and Cr components
C6000 Interleave	Convert planar YCbCr 4:2:2 data to interleaved YCbCr 4:2:2 data
C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
DM643x CAN Receive	Receive messages from CAN serial communications bus on DM643x
DM643x CAN Setup	Configure CAN serial communications bus parameters on DM643x
DM643x CAN Transmit	Configure CAN mailbox to transmit messages on CAN serial communications bus on DM643x
DM643x Draw Rectangles	Configure Video Processing Back End to draw rectangles using On Screen Display (OSD) module
DM643x OSD	Overlay graphics and text on video

DM643x PWM	Configure DM643x DSP Event Manager to generate PWM waveforms
DM643x UART Config	Configure DM643x UART for serial communication
DM643x UART Receive	Configure receiver element of DM643x UART module for serial communication
DM643x UART Transmit	Configure transmitter element of DM643x UART module for serial communication
DM643x Video Capture	Configure Video Processing Front End (VPFE) to capture REC656 or generic YCbCr 4:2:2 video
DM643x Video Display	Configure Video Processing Back End to display NTSC/PAL video

For information about the Avnet S3ADSP DM6437 Target Preferences block, see the following topic:

Target Preferences/Custom Board	Configure model for Texas Instruments processor.
---------------------------------	--------------------------------------------------

C6416 DSK (c6416dsklib)

C6416 DSK ADC	Digitized output from codec to processor
C6416 DSK DAC	Use codec to convert digital input to analog output
C6416 DSK DIP Switch	Simulate or read DIP switches
C6416 DSK LED	Control LEDs
C6416 DSK Reset	Reset to initial conditions

C6455 EVM (c6455evmlib)

C6455 DSK ADC	Configure AIC23 audio codec to capture audio stream from LINE-IN or MIC
C6455 DSK DAC	Configure AIC23 codec to convert digital signal to audio output on LINE OUT and HP OUT
C6455 DSK DIP	Output state of user-selected DIP switch as Boolean
C6455 DSK LED	Apply Boolean input to user-selected LED
C6455 DSK SRIO Config	Configure generated code for serial RapidI/O peripheral
C6455 DSK SRIO Receive	Configure generated code to receive serial RapidI/O packets
C6455 DSK SRIO Transmit	Configure generated code to transmit serial RapidI/O packets

C6713 DSK (c6713dsklib)

C6713 DSK ADC	Digitized signal output from codec to processor
C6713 DSK DAC	Configure codec to convert digital input to analog output
C6713 DSK DIP Switch	Simulate or read DIP switches
C6713 DSK LED	Control LEDs
C6713 DSK Reset	Reset to initial conditions

C6747 EVM (c6747evmlib)

C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
C6747EVM ADC	Capture audio stream from LINE IN jack
C6747EVM DAC	Output audio on LINE OUT / HP OUT jacks
C6747EVM DIP Switch	Output DIP switch status
C6747EVM LED	Control four on-board LEDs

For information about the C6747 EVM Target Preferences block, see the following topic:

Target Preferences/Custom Board	Configure model for Texas Instruments processor.
---------------------------------	--------------------------------------------------

TCI6428 DSK (tci6428dsklib)

For information about the TCI6428 DSK Target Preferences block, see the following topic:

Target Preferences/Custom Board	Configure model for Texas Instruments processor.
---------------------------------	--------------------------------------------------

C6727 PADK (c6727padklib)

For information about the C6727 PADK Target Preferences block, see the following topic:

Target Preferences/Custom Board	Configure model for Texas Instruments processor.
---------------------------------	--------------------------------------------------

Custom C6000 (c6000customlib)

For information about the C6000 Target Preferences block, see the following topic:

Target Preferences/Custom Board	Configure model for Texas Instruments processor.
---------------------------------	--------------------------------------------------

Custom C64x+ (c64xpcustomlib)

For information about the C64x+ Target Preferences block, see “Generating Code for Any C64x+ Processor or Board” on page 3-27 and the following topic:

Target Preferences/Custom Board	Configure model for Texas Instruments processor.
---------------------------------	--------------------------------------------------

CAN Message Handling Blocks (canmsglib)

DM642 EVM (dm642evmlib)

DM642 EVM Audio ADC	Audio codec and peripherals
DM642 EVM Audio DAC	Configure codec to convert digital audio input to analog audio output
DM642 EVM FPGA GPIO Read	User GPIO registers to read from selected pins
DM642 EVM FPGA GPIO Write	Write to GPIO registers
DM642 EVM LED	Control LEDs
DM642 EVM Reset	Reset to initial conditions
DM642 EVM Video ADC	Video decoders to capture analog video
DM642 EVM Video DAC	Video encoder to display video
DM642 EVM Video Port	Video port to receive video data from video input port

DM6437 EVM (dm6437evmlib)

C6000 Deinterleave	Separate interleaved YCbCr 4:2:2 data into Y, Cb, and Cr components
C6000 Interleave	Convert planar YCbCr 4:2:2 data to interleaved YCbCr 4:2:2 data
C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
DM6437 EVM ADC	Configure AIC33 audio codec to capture audio stream from LINE-IN or MIC

DM6437 EVM DAC	Configure AIC33 codec to convert digital signal to audio output on LINE OUT and HP OUT
DM6437 EVM DIP	Output state of user-selected DIP switch as Boolean
DM6437 EVM LED	Apply Boolean input to user-selected LED
DM6437 EVM Video Capture	Configure video peripherals to capture NTSC/PAL video
DM643x CAN Receive	Receive messages from CAN serial communications bus on DM643x
DM643x CAN Setup	Configure CAN serial communications bus parameters on DM643x
DM643x CAN Transmit	Configure CAN mailbox to transmit messages on CAN serial communications bus on DM643x
DM643x Draw Rectangles	Configure Video Processing Back End to draw rectangles using On Screen Display (OSD) module
DM643x OSD	Overlay graphics and text on video
DM643x PWM	Configure DM643x DSP Event Manager to generate PWM waveforms
DM643x UART Config	Configure DM643x UART for serial communication
DM643x UART Receive	Configure receiver element of DM643x UART module for serial communication

DM643x UART Transmit

Configure transmitter element of
DM643x UART module for serial
communication

DM643x Video Display

Configure Video Processing Back
End to display NTSC/PAL video

DM648 EVM (dm648evmlib)

C6000 IP Config

Configure Internet Protocol on
C6000 targets with Ethernet ports

DM648 EVM Video Capture

Configure DSP peripherals to
capture NTSC/PAL or HD video

DM648 EVM Video Display

Configure DSP peripherals to display
NTSC, PAL, HD, or VESA video

DSP/BIOS (dspbioslib)

DSP/BIOS Hardware Interrupt

Generate Interrupt Service Routine

DSP/BIOS Task

Create task that runs as separate
DSP/BIOS thread

DSP/BIOS Triggered Task

Create asynchronously triggered
task

C62x DSP Library (tic62dsplib)

Conversions (p. 6-10)

Convert data types

Filters (p. 6-10)

Filter input signals

Math and Matrices (p. 6-11)	Perform mathematical operations
Transforms (p. 6-11)	Perform transforms

Conversions

C62x Convert Floating-Point to Q.15	Convert single-precision floating-point input signal to Q.15 fixed-point
C62x Convert Q.15 to Floating-Point	Convert Q.15 fixed-point signal to single-precision floating-point

Filters

C62x Complex FIR	Filter complex input signal using complex FIR filter
C62x General Real FIR	Filter real input signal using real FIR filter
C62x LMS Adaptive FIR	LMS adaptive FIR filtering
C62x Radix-4 Real FIR	Filter real input signal using real FIR filter
C62x Radix-8 Real FIR	Filter real input signal using real FIR filter
C62x Real Forward Lattice All-Pole IIR	Filter real input signal using lattice filter
C62x Real IIR	Filter real input signal using IIR filter
C62x Symmetric Real FIR	Filter real input signal using FIR filter

Math and Matrices

C62x Autocorrelation	Autocorrelate input vector or frame-based matrix
C62x Block Exponent	Minimum number of extra sign bits in each input channel
C62x Matrix Multiply	Matrix multiply two input signals
C62x Matrix Transpose	Matrix transpose input signal
C62x Reciprocal	Fraction and exponent portions of reciprocal of real input signal
C62x Vector Dot Product	Vector dot product of real input signals
C62x Vector Maximum Index	Zero-based index of maximum value element in each input signal channel
C62x Vector Maximum Value	Maximum value for each input signal channel
C62x Vector Minimum Value	Minimum value for each input signal channel
C62x Vector Multiply	Element-wise multiplication on inputs
C62x Vector Negate	Negate each input signal element
C62x Vector Sum of Squares	Sum of squares over each real input channel
C62x Weighted Vector Sum	Weighted sum of input vectors

Transforms

C62x Bit Reverse	Bit-reverse elements of each complex input signal channel
C62x FFT	Decimation-in-frequency forward FFT of complex input vector

C62x Radix-2 FFT	Radix-2 decimation-in-frequency forward FFT of complex input vector
C62x Radix-2 IFFT	Radix-2 inverse FFT of complex input vector

C64x DSP Library (tic64dsplib)

Conversions (p. 6-12)	Data conversion
Filters (p. 6-12)	Filter input signals
Math and Matrices (p. 6-13)	Mathematical operations
Transforms (p. 6-14)	Transforms

Conversions

C64x Convert Floating-Point to Q.15	Convert floating-point signal to Q.15 fixed-point
C64x Convert Q.15 to Floating-Point	Convert Q.15 fixed-point signal to single-precision floating-point

Filters

C64x Complex FIR	Filter complex input signal using complex FIR filter
C64x General Real FIR	Filter real input signal using real FIR filter
C64x LMS Adaptive FIR	LMS adaptive FIR filtering
C64x Radix-4 Real FIR	Filter real input signal using real FIR filter
C64x Radix-8 Real FIR	Filter real input signal using real FIR filter

C64x Real Forward Lattice All-Pole IIR	Filter real input signal using lattice IIR filter
C64x Real IIR	Filter real input signal using IIR filter
C64x Symmetric Real FIR	Filter real input signal using FIR filter

Math and Matrices

C64x Autocorrelation	Autocorrelate input vector or frame-based matrix
C64x Block Exponent	Minimum number of extra sign bits in each input channel
C64x Matrix Multiply	Matrix multiply two input signals
C64x Matrix Transpose	Matrix transpose input signal
C64x Reciprocal	Fraction and exponent of reciprocal of real input signal
C64x Vector Dot Product	Vector dot product of real input signals
C64x Vector Maximum Index	Zero-based index of maximum value element in each input signal channel
C64x Vector Maximum Value	Maximum value for each input signal channel
C64x Vector Minimum Value	Minimum value for each input signal channel
C64x Vector Multiply	Element-wise multiplication on inputs
C64x Vector Negate	Negate each input signal element
C64x Vector Sum of Squares	Sum of squares over each real input channel
C64x Weighted Vector Sum	Weighted sum of input vectors

Transforms

C64x Bit Reverse	Bit-reverse elements of each complex input signal channel
C64x FFT	Decimation-in-frequency forward FFT of complex input vector
C64x Radix-2 FFT	Radix-2 decimation-in-frequency forward FFT of complex input vector
C64x Radix-2 IFFT	Radix-2 inverse FFT of complex input vector

Scheduling (c6000dspcorelib)

C6000 Block Processing	Repeat user-specified operation on submatrices of input matrix, using internal memory of DSP for increased efficiency
C6000 CPU Timer	Select timer and configure periodic interrupt
C6000 EDMA	Configure EDMA Controller on C6000 processor

Target Communication (targetcommlib)

C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
C6000 TCP/IP Receive	Receive message from remote IP interface
C6000 TCP/IP Send	Send message to remote IP interface

C6000 UDP Receive	Receive uint8 vector as UDP message
-------------------	-------------------------------------

C6000 UDP Send	Send UDP message to host
----------------	--------------------------

Also present are the following three blocks from the Target Support Package/Common/Host Communication library:

Byte Reversal	Communicate with target processor that is big-endian.
---------------	-------------------------------------------------------

Byte Pack	Pack input data into single output vector of type uint8.
-----------	----------------------------------------------------------

Byte Unpack	Unpack binary byte vector to extract data.
-------------	--------------------------------------------

Target Preferences (c6000tgtpreflib)

For information about any of the Target Preferences/Custom Board blocks for Texas Instruments' processors, see the following topic:

Target Preferences/Custom Board	Configure model for Texas Instruments processor.
---------------------------------	--------------------------------------------------

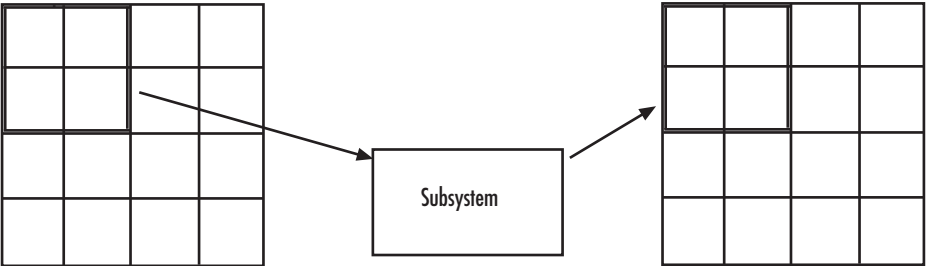
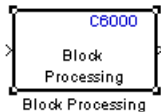
Blocks — Alphabetical List

C6000 Block Processing

Purpose Repeat user-specified operation on submatrices of input matrix, using internal memory of DSP for increased efficiency

Library “Scheduling (c6000dspcorelib)” on page 6-14

Description The Block Processing block extracts submatrices of a user-specified size from each input matrix. It sends each submatrix to a subsystem for processing, and then reassembles each subsystem output into the output matrix, as shown in the following figure. While processing images as matrices, this submatrix capability can greatly improve the throughput.



Note Because you modify the Block Processing block subsystem, the link between this block and the block library is broken when you click-and-drag a Block Processing block into your model. Thus, this block is not automatically updated if you upgrade to a newer version of the target support package. To delete blocks from this subsystem without triggering a warning, right-click on the block and select **Look under mask**. If you search for library blocks in a model, this block is not part of the results.

The blocks inside the subsystem dictate the following block configuration information:

- Frame status of the input and output signals
- Whether the block supports single channel or multichannel signals

- Which data types this block supports

Use the **Number of inputs** and **Number of outputs** parameters to specify the number of input and output ports on the Block Processing block.

Use the **Block size** parameter to specify the size of each submatrix in cell array format. Each vector in the cell array corresponds to one input; the block uses the vectors in the order you enter them. If you have one input port, enter one vector. If you have more than one input port, you can enter one vector that is used for all inputs or you can specify a different vector for each input. For example, to specify each submatrix as a 2-by-3 array, enter `{[2 3]}`. The output matrix size depends on the size of the submatrix at the output of the subsystem and the number of submatrices at the input. For example, if the output submatrix size is 32x16 and the input submatrix sizes are 8x16, the total output matrix size will be 256x256. If the block size specified does not subdivide an input matrix evenly, i.e. there are leftover matrix elements which are not covered by the subdivision, those uncovered elements will be ignored.

Use the **Overlap** parameter to specify the overlap of each submatrix in cell array format. Each vector in the cell array corresponds to the overlap of one input; the block uses the vectors in the order they are specified. If you enter one vector, each overlap is the same size. For example, to specify that each 3-by-3 submatrix overlap by 1 row and 2 columns, enter `{[1 2]}`.

The **Traverse order** parameter determines how the block extracts submatrices from the input matrix. If you select **Row-wise**, the block extracts submatrices by moving across the rows. If you select **Column-wise**, the block extracts submatrices by moving down the columns.

Click **Open Subsystem** to open the block subsystem. Click-and-drag blocks into this subsystem to define the processing operations the block performs on the submatrices. The input to this subsystem are the submatrices defined by the **Block size** parameter.

C6000 Block Processing

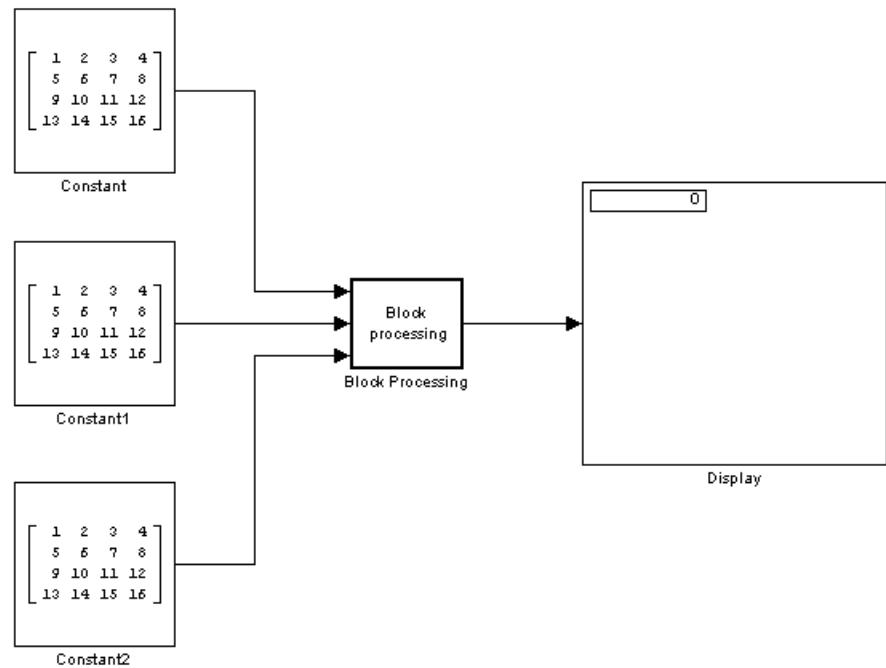
Note When you place an Assignment block inside a Block Processing block subsystem, the Assignment block behaves as though it is inside a For Iterator block. For a description of this behavior, refer to “Iterated Assignment” on the Assignment block reference page. To produce the normal behavior of the Assignment block, use an Overwrite Values block inside the Block Processing block subsystem.

Example

This section provides an example that applies the block processing block to multiply and add submatrices.

Multiple Inputs

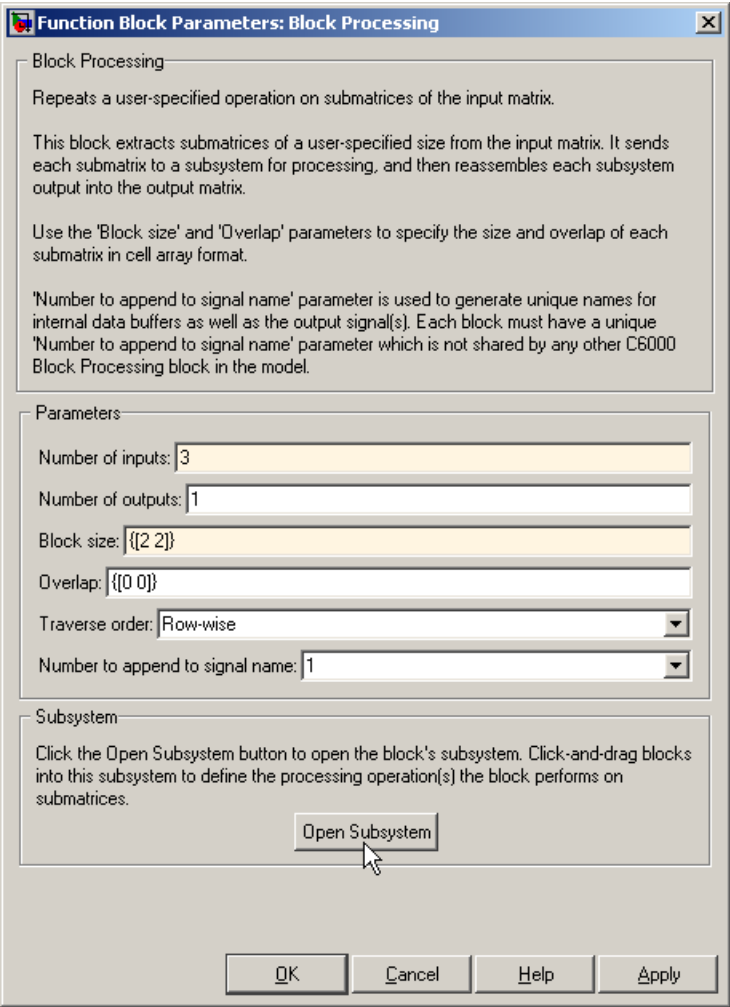
In this example, you multiply each element of three input matrices by two and add the results using the Block Processing block. Suppose you have the following model:



1 Use the Block Processing block to perform the multiplication and addition on submatrices of the three input matrices. Set the block parameters as shown in the following figure:

- **Number of inputs** = 3
- **Number of outputs** = 1
- **Block size** = {[2 2]}

C6000 Block Processing

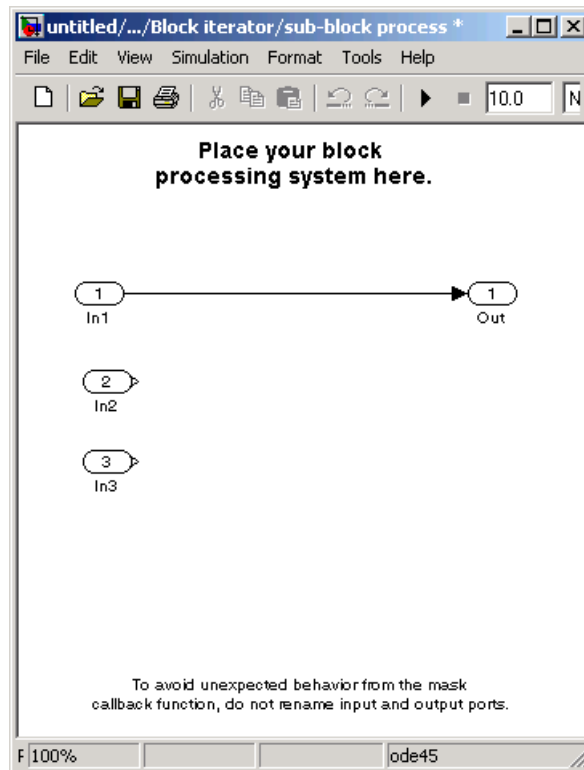


For each iteration, the block sends a 2-by-2 submatrix from each input matrix to the Block Processing block subsystem to be processed. The block calculates its total number of iterations using the dimensions of the matrix connected to the top input port. In this case,

the first input is a 4-by-4 matrix. The block can extract four 2-by-2 submatrices from this input matrix, so the block iterates four times.

2 Click Open Subsystem.

The block subsystem opens.

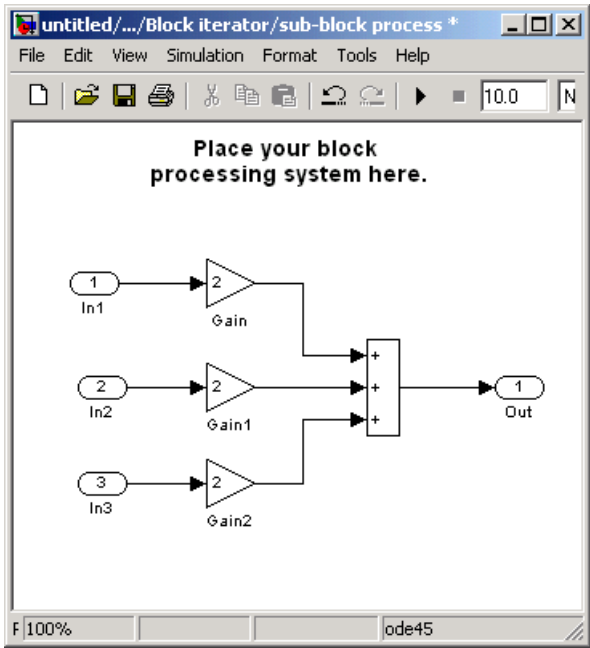


3 Click and drag the blocks shown in the following table into the subsystem.

C6000 Block Processing

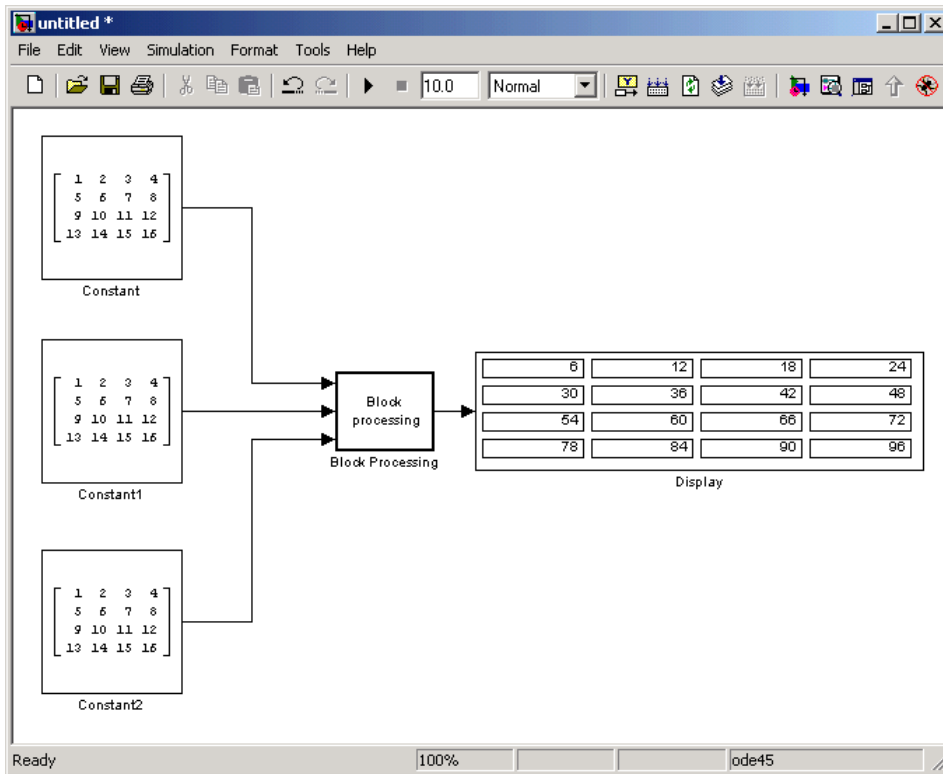
Block	Library	Quantity
Gain	Simulink / Math Operations	3
Sum	Simulink / Math Operations	1

- 4 Use the Gain blocks to multiply the elements of each submatrix by two. Set the **Gain** parameter to 2.
- 5 Use the Sum block to add the values. Set the **Icon shape** parameter to rectangular and the **List of signs** parameter to +++.
- 6 Connect the blocks as shown in the following figure.



- 7 Close the subsystem and click **OK**.
- 8 Run the model.

C6000 Block Processing

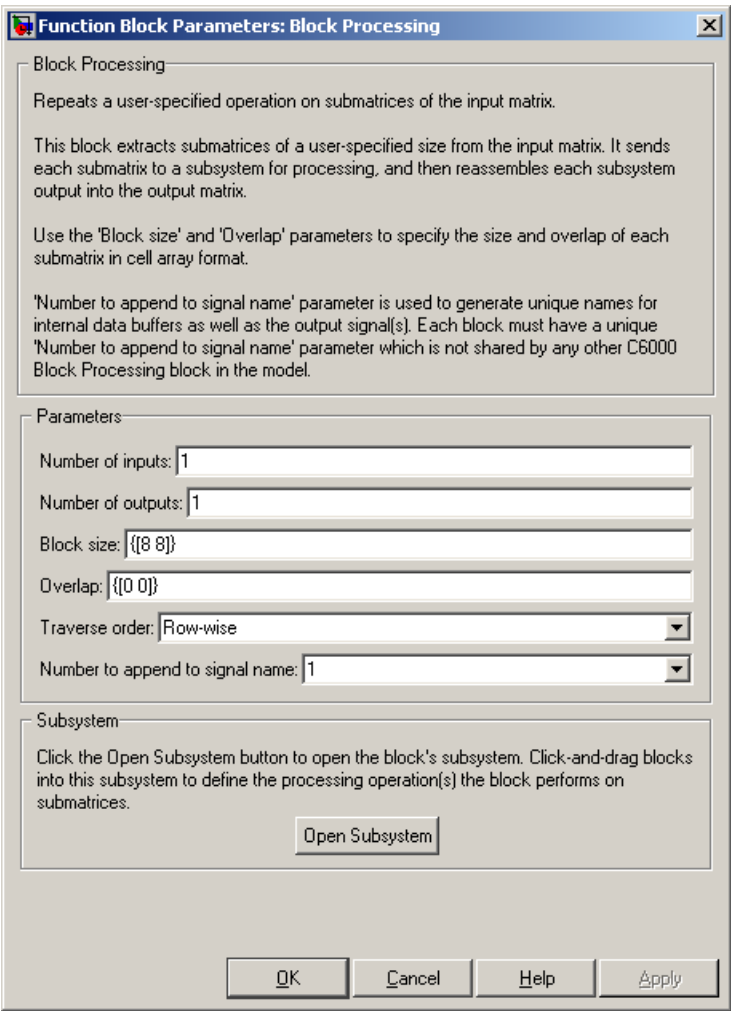


The Block Processing block operates on the submatrices, assembles the results into an output matrix, and then uses the Display block to present the output matrix.

C6000 Block Processing

Dialog Box

The Block Processing dialog box appears as shown in the following figure.



Number of inputs

Enter the number of input ports on the Block Processing block.

Number of outputs

Enter the number of output ports on the Block Processing block.

Block size

Specify the size of each submatrix in cell array format. Each vector in the cell array corresponds to one input.

Overlap

Specify the overlap of each submatrix in cell array format. Each vector in the cell array corresponds to the overlap of one input.

Traverse order

Determines how the block extracts submatrices from the input matrix. If you select **Row-wise**, the block extracts submatrices by moving across the rows. If you select **Column-wise**, the block extracts submatrices by moving down the columns.

Open Subsystem

Click this button to open the block's subsystem. Click and drag blocks into this subsystem to define the processing the block performs on the submatrices.

See Also

Memory Allocate, Memory Copy, C6000 EDMA

C6000 Deinterleave

Purpose Separate interleaved YCbCr 4:2:2 data into Y, Cb, and Cr components

Library “AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2
“DM6437 EVM (dm6437evmlib)” on page 6-7

Description This block separates interleaved YCbCr 4:2:2 data into its luma component (Y), blue-difference chroma component (Cb), and red-difference chroma component (Cr).

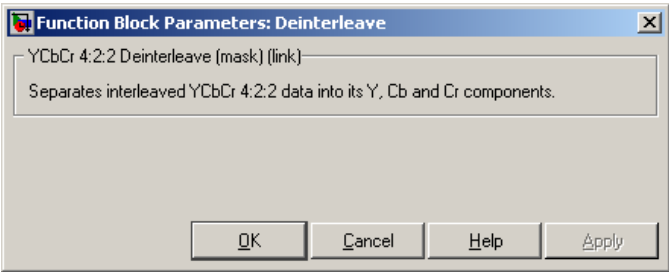
The input, YCbCr, is a $(2*M)*N$ array of 8-bit unsigned values representing an interleaved YCbCr 4:2:2 image where the size of the luma plane, Y, is $M*N$. Input data is assumed to be in row-major format, and the data stored in each row of the input is assumed to be interleaved in the following order:



$Cb(1), Y(1), Cr(1), Y(2), \dots, Cb(M), Y(M), Cr(M), Y(M)$

The deinterleaved outputs are the planar format luma component, Y, and the chroma components, Cb and Cr, of the YCbCr 4:2:2 input. If the input image is a $(2*M)$ by N matrix, then the output dimensions for the Y port is $(M*N)$ and the dimensions for the Cb and Cr ports are $(M/2)$ by N .

Dialog Box



This block does not have settable options.

See Also C6000 Interleave

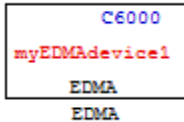
Purpose

Configure EDMA Controller on C6000 processor

Library

“Scheduling (c6000dspcorelib)” on page 6-14

Description



Use this block to configure the Enhanced Direct Memory Access (EDMA) Controller on C6000 processors. The controller manages data transfers between the device peripherals on the C6000 processors and the level two (L2) cache/memory controller. Data transfers handled by the controller include:

- Host accesses to cache
- Accessing noncacheable memory
- Servicing cache
- Transferring data by user programs

EDMA controller handles transfers without involving the processor and can process transfers between any addressable memory spaces, including internal and external memory.

For details about the EDMA controller, refer to *TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Reference Guide*, SPRU234, from the Texas Instruments Web site.

Note The C6000 EDMA block does not support C64x⁺ processors, such as the C6455 or TCI6482.

EDMA blocks provide two operating modes—open an EDMA channel and allocate a table in EDMA parameter RAM (PaRAM).

The open channel mode opens an EDMA channel for the controller. When you open a channel, EDMA sets the transfer parameters for the channel and writes those to a table as PaRAM entries.

In allocate table mode, the block sets the EDMA transfer parameters and places them in a table in EDMA PaRAM without opening a

channel. With this mode, you can use EDMA channels and transfers to develop complex memory structures like sorting, or circular buffers. The allocate table operating mode lets you link multiple EDMA blocks on one EDMA channel. One EDMA block opens an EDMA channel and succeeding blocks link to the open channel and originating EDMA block by the device handle setting.

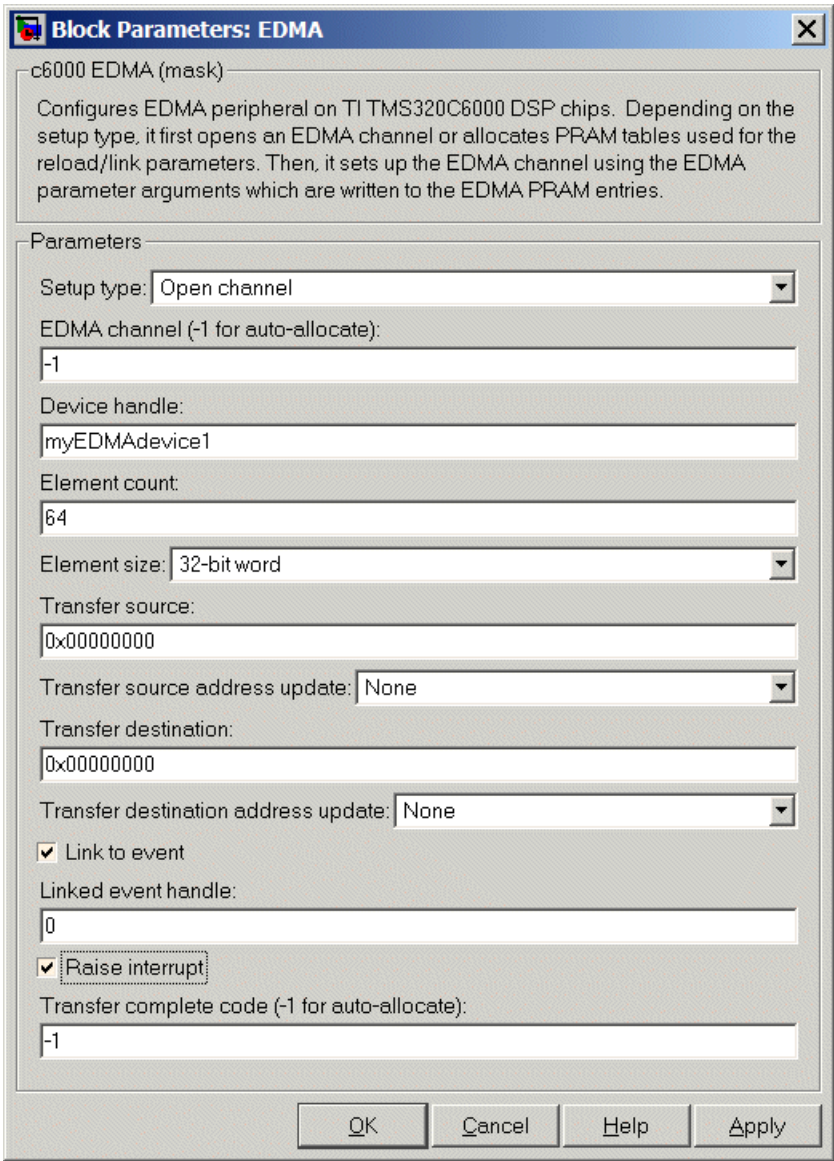
Use the following procedure to link EDMA blocks in a model:

- 1** Add an EDMA block to your model, open the block dialog box, and set **Setup type** to **Open channel**.
- 2** Assign an EDMA channel to use in **EDMA channel (-1 for auto-allocate)** by entering a channel number or entering -1 to let the block choose the channel.
- 3** In **Device handle**, provide a name for this EDMA block. The name you enter becomes the block identifier for other blocks to link to this block. Use any valid C variable string.
- 4** Close the block dialog box.
- 5** Add a second EDMA block to your model, and open the block dialog box to set the block parameters.
- 6** Select **Allocate table** from the **Setup type** list.
- 7** Select the **Link to event** check box.
- 8** Enter the device handle from the earlier block to link to in **Linked event handle** in this block. The two blocks are linked together through the device handle and they use the same channel.
- 9** Close the block dialog box.
- 10** To link more EDMA blocks to this channel, repeat steps 5 through 9 for each new block, entering the same device handle.

For a demonstration of using and linking EDMA blocks, refer to the demo Custom Device Driver via Legacy Code Integration in the Target Support Package demos in the online help system.

--

Dialog Box



The preceding dialog box shown presents all of the parameters available. In some cases, parameters are available only when you select other parameters. The following list of block parameters describes all of the available parameters for the block and when one parameter enables another.

Setup type

Choose either `Open channel` or `Allocate table` from the list. If this is the only EDMA block in your model, choose `Open channel`. If your model includes multiple EDMA blocks, choose `Open channel` when each block should use a different channel. Select `Allocate table` for any block that you plan to link to another EDMA block.

EDMA channel (-1 for auto-allocate)

Enter an integer from 0 to 63 to specify the EDMA channel to use. If you enter -1, the block assigns the channel automatically from the available channels.

Device handle

Provide a name for this block. The name you enter must be a valid C variable. The EDMA controller uses the name as the identifier for this block and open channel. Other EDMA blocks in your model can link to this block and channel by using the device handle you enter.

Element count

Specifies the number of elements in a frame. The value 65355 is the maximum number of elements allowed in one frame. The value defaults to 64 elements.

Element size

EDMA supports 32-bit words, 16-bit half words, and 8-bit bytes. Select one of the list entries according to your needs.

Transfer source

Enter the address of the elements to transfer. Specify the address as a hexadecimal value as shown by the default address 0x.00000000

Transfer source address update

Select whether to enable transfer source update on the EDMA controller. When you select an option from the list, the controller updates the transfer source address according to your choice. Choose one of the list entries shown in the following table.

Option	Effect on Transfer Source Address	Condition Indicated
None	Does not change address after submitting the transfer request.	Indicates that all of the elements to transfer are located at the same address in memory.
Increment	Increases the transfer address by the value in Element count after submitting the transfer request.	Indicates that the elements are contiguous, with each subsequent element located at a higher address than the previous element.
Decrement	Decreases the transfer address by the value in Element count after submitting the transfer request.	Indicates that the elements are contiguous, with each subsequent element located at a lower address than the previous element.

Transfer destination

Enter the destination memory address for the data transfer. Specify the address as a hexadecimal value as shown by the default address 0x.00000000

Transfer destination address update

Select whether to enable transfer destination update on the EDMA controller. When you select an option from the list, the controller updates the transfer destination address according to your choice. Choose one of the list entries shown in the following table.

Option	Effect on Transfer Destination Address	Condition Indicated
None	Does not change address after submitting the transfer request.	Indicates that all of the elements to transfer are located at the same address in memory.
Increment	Increases the transfer address by the value in Element count after submitting the transfer request.	Indicates that the elements are contiguous, with each subsequent element located at a higher address than the previous element.
Decrement	Decreases the transfer address by the value in Element count after	Indicates that the elements are

Option	Effect on Transfer Destination Address	Condition Indicated
	submitting the transfer request.	contiguous, with each subsequent element located at a lower address than the previous element.

Link to event

You can link EDMA transfers together to create more complicated memory applications such as buffers and sorting routines. When you select **Link to event** to enable linking, the EDMA controller link feature reloads the current transfer parameters from PaRAM when the previous transfer is complete.

Linked event handle

To link to another EDMA block to create more complex memory applications, enter the device handle from the EDMA block to link to in **Linked event handle**. This entry is an alphanumeric string and the EDMA controller interprets your entry as a string.

Raise interrupt

Select this check box to direct the EDMA controller to raise an interrupt when the transfer request completes. When you select this parameter, you enable the Transfer complete code (-1 for auto-allocate) option. Clearing Raise interrupt stops the controller from raising the interrupt on TR completion.

Transfer complete code (-1 for auto-allocate)

The transfer code Indicates when the controller has submitted a required number of transfer requests (TR). Provide an integer from 0 and 62. On C67x processors, the code must be from 0 to 15. The default value of -1 lets the controller assign the transfer code for this channel.

When you enable this option, the EDMA controller submits the transfer request with a request that the controller signal completion of the transfer with this code. When the transfer is completed, the transfer controller returns the specified code to the EDMA controller.

After the EDMA controller receives the transfer complete code in response to the TR, the controller uses the code to trigger another TR or to raise an interrupt to the processor when you select **Raise interrupt**.

References

For details about the EDMA controller, refer to *TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Reference Guide*, SPRU234, available from the Texas Instruments Web site.

For an introduction to the EDMA controller, refer to *TMS320C6000 Peripherals Reference Guide*, SPRU190, which provides an overview of the controller, available from the Texas Instruments Web site.

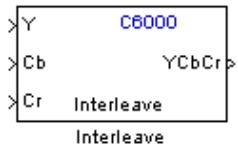
See Also

Memory Allocate, Memory Copy

C6000 Interleave

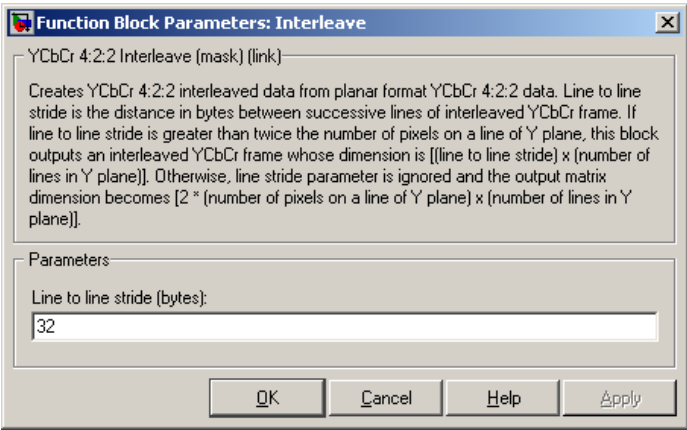
Purpose	Convert planar YCbCr 4:2:2 data to interleaved YCbCr 4:2:2 data
Library	“DM6437 EVM (dm6437evmlib)” on page 6-7 “AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

Description	<p>This block takes planar YCbCr 4:2:2 data on three separate inputs and converts them to a single interleaved YCbCr 4:2:2 data output.</p> <p>The input is a planar, color separated, YCbCr 4:2:2 image represented as a 2-D matrix of 8-bit unsigned integers. There are three input ports, one each for the luma component (Y), blue-difference chroma component (Cb), and red-difference chroma component (Cr). If the input to the Y port has dimensions $M \times N$, the input to the Cb and Cr ports must be $(M/2) \times N$.</p>
--------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



The output is an interleaved YCbCr 4:2:2 image represented as a 2-D matrix of 8-bit unsigned integers. If the dimension of the Y port is $M \times N$ and dimensions of the Cb and Cr ports are $M/2 \times N$, the image dimensions of the YCbCr output dimensions are $2 \times M \times N$ under normal conditions. If you specify a line-to-line stride greater than $2 \times M$ in the block's mask, the output dimensions become (line-to-line stride) $\times N$.

Dialog Box



Line to line stride (bytes)

Use the line-to-line stride parameter to satisfy the input requirements of the DM6437EVM Video Display block. Because of hardware requirements, each line of the input to the DM6437EVM Video Display block must have a size that is multiple of 32 bytes. For example, if the image you want to display is 180 by 120, use a line-to-line stride of 384 to satisfy the hardware requirements. Under normal conditions, the output of the Interleave block would have size 360x120 which would not be accepted by the DM6437EVM Video Display block. By using a line stride of 384, the block outputs a 384 by 120 matrix—of which only the 360x120 portion contains valid data—that is readily accepted by the DM6437EVM Video Display block.

Line-to-line stride is the distance in bytes between successive lines of an interleaved YCbCr frame. If line-to-line stride is greater than twice the number of pixels on a line of Y plane, this block outputs an interleaved YCbCr frame whose dimensions are the line-to-line stride times the number of lines in Y plane. Otherwise, line stride parameter is ignored, and the output matrix dimension becomes $2 \times (\text{number of pixels on a line of Y plane}) \times (\text{the number of lines in Y plane})$.

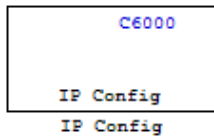
See Also

C6000 Deinterleave

Purpose Configure Internet Protocol on C6000 targets with Ethernet ports

Library Target Communication Library (targetcommmlib)
“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2
“C6747 EVM (c6747evmlib)” on page 6-5
“DM6437 EVM (dm6437evmlib)” on page 6-7
“DM648 EVM (dm648evmlib)” on page 6-9
“Target Communication (targetcommmlib)” on page 6-14

Description Adding this block to your model provides options to configure the IP parameters for your C6000 board. Setting the options for the block sets the address and name for your board and specifies your target and Ethernet daughtercard.



To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements:

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Configure the daughter card. Refer to “Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP” on page A-2 for more information about configuring the card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

The block uses dynamic addressing, getting the address from the local server or static addressing. If you have a dynamic host configuration protocol (DHCP) server available, you can allow the server to provide an IP address for your board. Dynamic IP addresses can be useful but unreliable — they can change.

To use static addressing, create a static IP address by clearing **Use DHCP to allocate an IP address for DM642 EVM (requires DHCP server)**. to enable the manual IP address configuration parameters.

Note When you use the UDP Send and Receive blocks in a model, you must also include this block to set up the IP drivers for the Ethernet parameters for the target networking capability.

Whether you choose to use dynamic addressing, you must set the Host name, and select and set the **Use the following CPU interrupt for Ethernet driver (4-13)** options.

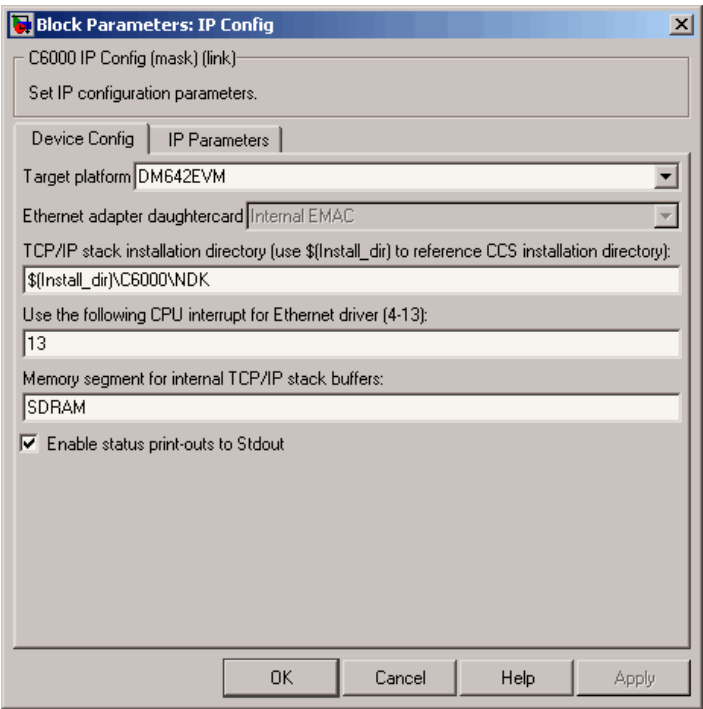
When you build and run your model, this block has no effect. It outputs zeros. When you generate code from your model, this block adds the code that configures IP on your board.

C6000 IP Config

Dialog Box

The block dialog box provides options on two tabs — **Device Config** and **IP Parameters**.

Device Tab Options



Target platform

Specify your C6000 target by selecting the appropriate target board from the list. Changing the target platform changes the entry on the **Ethernet adapter daughtercard** list.

Ethernet adapter daughtercard

After you select you target platform, this option lets you select whatever daughtercard is available to implement Ethernet communications on the target.

TCP/IP stack installation folder

To use the UDP and TCP blocks for the board, you must install the TMS320C6000 TCP/IP Stack from Texas Instruments. Specify the folder where the TMS320C6000 TCP/IP Stack from Texas Instruments is installed.

Use the following CPU interrupt for Ethernet driver (4-13)

The Ethernet driver on the DM642 can respond to any one of the CPU interrupts from 4 to 13. Enter one valid CPU interrupt for the driver to react to. CPU interrupt 13 is the default interrupt.

Memory segment for internal TCP/IP stack buffers

Shows you the segment in memory where the TCP/IP stack buffers reside. For the supported boards, the default setting and location is SDRAM. You can change the location by entering the name of the memory segment to use. TCP/IP stack buffers occupy approximately 130 kB of memory. In most cases you should locate the TCP/IP stack buffers in external memory. Be sure that the segment you specify here agrees with the memory segment allocation in the target preferences block in your model.

Enable status print-outs to Stdout

Select this option to direct the block to send IP status information to the standard output device.

IP Parameters Options

Block Parameters: IP Config

C6000 IP Config (mask)
Set IP configuration parameters.

Device Config | **IP Parameters**

☒ Use DHCP to allocate an IP address (requires a DHCP server):
Use the following IP address:
100.100.100.2
Subnet mask:
255.255.255.0
Gateway IP:
100.100.100.1
Domain name server IP:
0.0.0.0
Domain name (less than 64 characters):
mathworks.net
Host name (less than 64 characters):
dm642evm

OK Cancel Help Apply

Use DHCP to allocate an IP address (requires a DHCP server)

Selecting this parameter configures the board to get an IP address from the local DHCP server on the network. If you select this option and you do not have a DHCP server, the generated code does not run correctly. Clearing this option enables all of the IP configuration options for the block to let you define your IP address manually.

Use the following IP address

Specify an IP address. This value is the address that others use to communicate with the evaluation module over IP. Use the full xxx.xxx.xxx.xxx format.

Subnet mask

Define the subnet mask address, entering the full subnet mask in the format xxx.xxx.xxx.xxx. Subnet masks define how many bits of the IP address are used to identify the network.

By using 1s in all the address bits that identify the network, the subnet mask shows you which bits define the network and which are internal to the network. In the figure, the subnet mask 255.255.255.0 indicates that the first three octets in the address define the network.

Gateway IP

Enter one address for the gateway server or router that maintains a more complete listing of the surrounding networks. Messages that are destined for machines outside the local network are sent to the gateway address for address resolution.

Domain name server IP

Enter the address of the server for the domain in which the target is a member.

Domain name

Enter the name for the domain. Without the correct domain name, the target cannot communicate on the network within the domain.

Host name (less than 64 characters)

Enter the name of the host. Usually this value is the NetBIOS name for the machine if it exists.

See Also

C6000 TCP/IP Receive, C6000 TCP/IP Send,

C6000 TCP/IP Receive

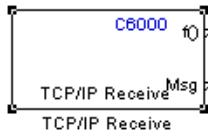
Purpose

Receive message from remote IP interface

Library

“Target Communication (targetcommplib)” on page 6-14

Description



Adding this block to your Simulink model results in generated code that configures TCP/IP on your target to receive messages.

To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements.

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Configure the daughter card. Refer to “Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP” on page A-2 for more information about configuring the card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

The block receives the message from the specified IP address on a host machine and passes it out the Msg port to a downstream block. There is no restriction on message size.

A second block output is a function call port that issues a function call whenever a new message is available on the receive buffer.

In simulations, this block outputs a stream of data (default type `uint8_T`) from the Msg port with the first bytes set to 0xFF and the rest set to 0x00. When the function call port exists, it generates a function call for every sample time hit.

Models that contain this block generate code for the parameters that configure TCP/IP on the target, including the ports, buffers, and message sizes.

Dialog Box

Main Pane

Source Block Parameters: TCP/IP Receive

C6000 TCP/IP Receive (mask) (link)

Configure TCP/IP stack to receive TCP/IP messages from a remote interface identified by a remote IP address and a remote IP port parameter pair. Local port parameter is used to specify the listening port on the target for incoming connections.

Main | Data types

Connection type: Server

Remote IP address and IP port to receive from (format IP address:IP port):
100.100.100.2:0

Local IP port:
49000

TCP/IP receive buffer size:
8192

☐ Enable blocking mode

Sample time:
0.01

OK Cancel Help

Connection type

Connection type specifies the connection initiation method used for the block. This is a read-only parameter — you cannot change it.

A **Server** connection creates a listening socket at the IP address and port in **Local IP port**. The TCP/IP layer uses this socket to accept incoming connection requests. Any external TCP/IP interface that sends TCP/IP data to this block must actively seek the connection to establish communications (the *client* model).

Remote address and IP port to receive from (format IP Address:IP port)

Identifies the remote TCP/IP interface, by IP address and IP port, from which the block expects to receive messages. The input format uses the IP address and IP port identifier, separated by a colon. IP port value ranges from 0 to 65535. Entering a 0 for the IP port when the **Connection type** is **Client** specifies that the TCP/IP stack automatically assigns a port to use to seek connections.

Local IP port

This option identifies the IP port to use when **Connection type** is **Server** and when it is **Client**.

When you choose **Server**, **Local IP port** specifies the well-known port of the target TCP/IP server. Your IP port value must lie between 1 and 65535.

When you specify **Client** for the connection type, **Local IP port** specifies the TCP/IP address for the client socket. The IP port value can range from 0 to 65535, where 0 specifies that the TCP/IP stack assigns an ephemeral port automatically to seek connections.

TCP/IP receive buffer size

Specifies the size of the buffer used for queuing incoming TCP/IP messages. Typically, larger TCP/IP receive buffers provide a cushion for packet drops and can improve efficiency. The compiler allocates the TCP/IP receive buffer on the heap.

All TCP/IP blocks that specify a common local IP port must share a common TCP/IP receive buffer, because the size of the TCP/IP buffer is set only for the listening socket. All active connecting sockets inherit their buffer size value from the listening socket.

Enable blocking mode

Select this option to put the calling TCP/IP task into blocking mode so that the block receives messages completely before

outputting the messages in the buffer to downstream blocks. Blocks connected to the receive block do not execute until the receive process completes. In blocking mode, program execution for receiving data stops until data in the message buffer is received.

Clearing this option puts the block in non blocking mode. The block checks the number of bytes in the TCP/IP receive buffer and returns output data only when the receive buffer contains more data than requested.

The block receives or outputs data at any time. Processes do not wait for data. Disabling blocking activates the **Sample time** parameter and adds an additional function call port to the block that indicates when the data port contains new, valid data.

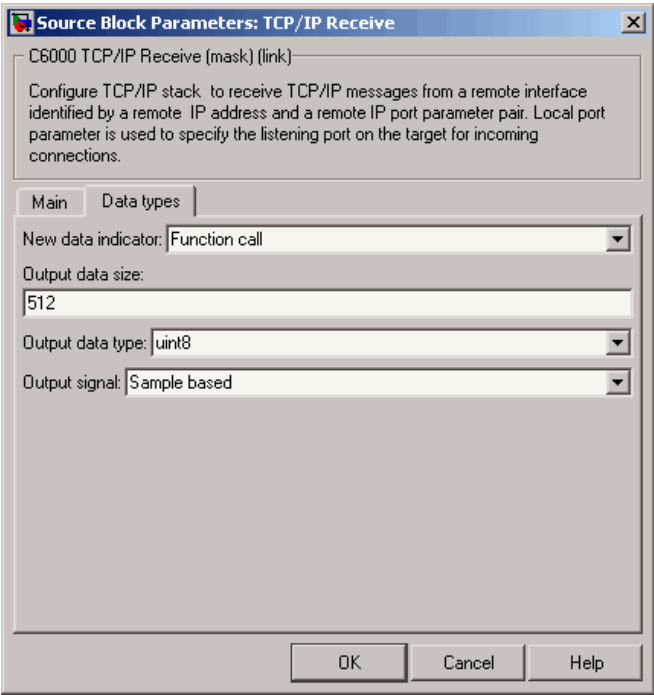
Selecting blocking mode activates the **Timeout** parameter.

Sample Time

Use this option to specify when the block polls for new messages. This parameter value should be positive. Setting this to a specific value, often large, can reduce the chances of TCP/IP messages getting dropped. The default sample time is 0.01 seconds.

C6000 TCP/IP Receive

Data Types Pane



New Data Indicator

Use this option to specify how new data is indicated, either by a function call or a Boolean status.

Output Data Size

Use this option to specify the size of the output data, the units depend on the output data type.

Output Data Type

Use this option to specify the type of the output data. The value selected can be any built-in Simulink data type.

Output Signal

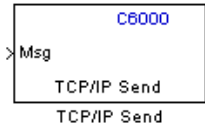
Use this option to specify whether the output signal is to be frame-based or sample-based.

See Also

C6000 TCP/IP Send, C6000 UDP Receive, C6000 UDP Send

C6000 TCP/IP Send

Purpose	Send message to remote IP interface
Library	“Target Communication (targetcommplib)” on page 6-14
Description	



Adding this block to your Simulink model results in generated code that configures TCP/IP on your target to send messages.

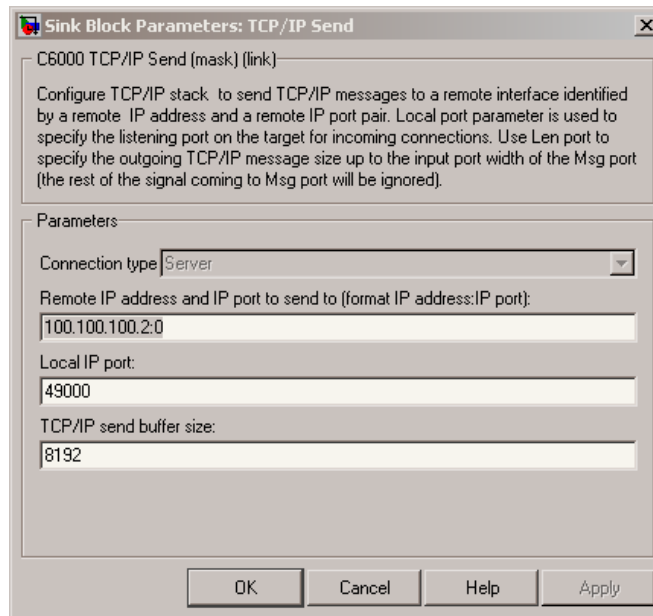
To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements.

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Configure the daughter card. Refer to “Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP” on page A-2 for more information about configuring the card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

The block sends the message to the specified IP address on a host machine. There is no restriction on the data type of the message to be sent, as long as it is a built-in Simulink data type. There is also no restriction on the size of the data to be transmitted.

Models that contain this block generate code for the parameters that configure TCP/IP on the target, including the ports, buffers, and message sizes.

Dialog Box



Connection type

Connection type specifies the connection initiation method used for the block. This is a read-only parameter — you cannot change it.

A **Server** connection creates a listening socket at the IP address and port in **Local IP port**. The TCP/IP layer uses this socket to accept incoming connection requests. For an external TCP/IP interface to receive TCP/IP data from this block, it must actively seek the connection to establish communications (the *client* model).

IP Address:IP port). External interfaces that want to exchange data with this block must be listening at the specified remote IP address and port.

C6000 TCP/IP Send

Remote IP address and IP port to send to (format IP address:IP port)

Identifies the remote TCP/IP interface, by IP address and IP port, to which the block expects to send messages. The input format uses the IP address and IP port identifier, separated by a colon. IP port value ranges from 0 to 65535. Entering a 0 for the IP port when the **Connection type** is **Client** specifies that the TCP/IP stack automatically assigns a port to use to seek connections.

Local IP port

This option identifies the IP port used when **Connection type** is **Server**.

When the connection type is **Server**, **Local IP port** specifies the well-known port of the target TCP/IP server. The IP port value must lie between 1 and 65535.

TCP/IP send buffer size

Specifies the size of the buffer used for queuing outgoing TCP/IP messages. Typically, larger TCP/IP receive buffers provide a cushion for packet drops and can improve efficiency. The compiler allocates the TCP/IP send buffer on the heap.

All TCP/IP blocks that specify a common local IP port must share a common TCP/IP send buffer, because the size of the TCP/IP buffer is set only for the listening socket. All active connecting sockets inherit their buffer size value from the listening socket.

See Also

C6000 TCP/IP Receive, UDP Send, UDP Receive

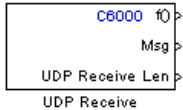
Purpose

Receive uint8 vector as UDP message

Library

“Target Communication (targetcommlib)” on page 6-14

Description



This block configures the Ethernet driver on the target to receive UDP messages. A UDP message comes into this block from the transport layer, usually TCP/IP. The block passes the message to the next downstream block out the Msg port. One block output (Msg) is the data vector from the message. A second output is a flag that indicates when a new UDP message is available. A third output specifies the length of the message for variable length messages.

To use this block with the C6416, or C6713 DSK targets, you must meet the following requirements.

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Configure the daughter card. Refer to “Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP” on page A-2 for more information about configuring the card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

This block reads a single UDP packet every sample hit. It does not attempt to receive multiple UDP packets to fill the output vector. If the UDP packet size is greater than the output port width parameter, UDP messages at the Msg port are truncated. The part for the UDP packet that does not fit into the Msg port is discarded as a result. The missing message content cannot be retrieved. Conversely, if the UDP packet size is smaller than the Msg port width specified, the portion of the output vector that does not fit into the specified size is invalid data.

In non blocking mode, the data in the Msg port is not valid unless the block issues a function call.

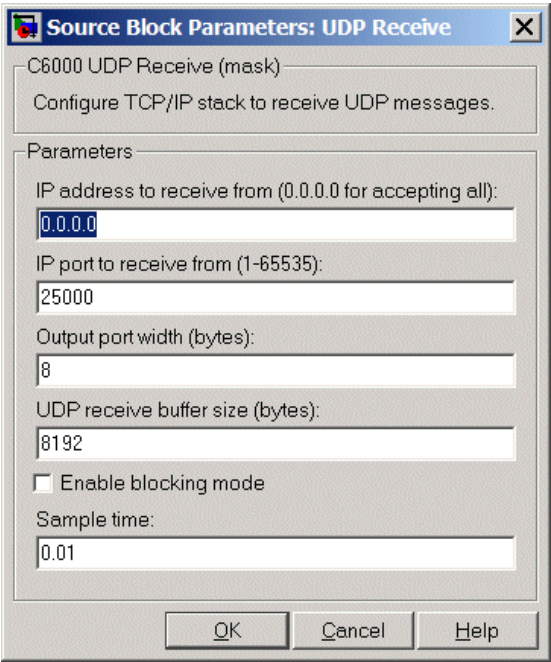
C6000 UDP Receive blocks operate only to generate code for the target Ethernet driver. They do not perform any function in simulation and their simulation outputs are zeros.

C6000 UDP Receive

Note To use the C6000 UDP Send and C6000 UDP Receive blocks, you must include the C6000 IP Config block to configure the Ethernet parameters for the target network. This block sets up the IP drivers for use and must be in the model for network-related processing.

Additional options let you decide whether the UDP messages work in blocking mode and set the sampling time for polling for new messages.

Dialog Box



IP address to receive from (0.0.0.0 to accept all)

Specifies the IP address from which the block accepts messages. Setting the address 0.0.0.0 configures the block to accept messages from any IP address. Setting a specific address, not 0.0.0.0, directs the block to accept messages from the specified address only.

Selecting Enable blocking mode, disables the IP address to receive from parameter. As a result, the block accepts messages from any IP address. You must clear Enable blocking mode to be able to set IP address to receive from to any value except for 0.0.0.0. The block must be in non blocking mode to specify the address to receive messages from via UDP.

IP port to receive from

Specify the port on this machine from which the block accepts messages. The other end of the communication, usually a UDP Send block, sends messages to this port. The value defaults to 25000, but the values can range from 1 to 65535.

Output port width (bytes)

Specifies the width of messages that the block accepts. When you design the transmit end of the UDP communication channel, you decide the message width. Set this parameter to a value as large or larger than any message you expect to receive.

UDP receive buffer size (bytes)

Specify the size of the buffer in which UDP messages are stored when received. 8192 bytes is the default size. You need a buffer large enough to store UDP messages that come in while your process reads a message from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

Enable blocking mode

Select this option to put the UDP receive process in blocking mode meaning the block outputs received messages before accepting input new messages. In blocking mode, program execution for receiving data stops until data in the buffer is sent. In non blocking mode, the block receives data or sends data at any time. Processes do not wait for data.

Sample time (seconds)

Use this option to specify when the block polls for new messages. The value entered here should always be greater than zero. Setting this to a specific value, often large, can reduce the chances

C6000 UDP Receive

of UDP messages getting dropped. The default sample time is 0.01 seconds.

See Also

C6000 TCP/IP Receive, C6000 TCP/IP Send, C6000 UDP Send

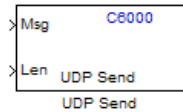
Purpose

Send UDP message to host

Library

“Target Communication (targetcommllib)” on page 6-14

Description



The UDP send block configures the target’s on-board Ethernet driver to receive a `uint8` vector that it sends as a UDP message to the host. Models can contain only one C6000 UDP Send block.

To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements.

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Configure the daughter card. Refer to “Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP” on page A-2 for more information about configuring the card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

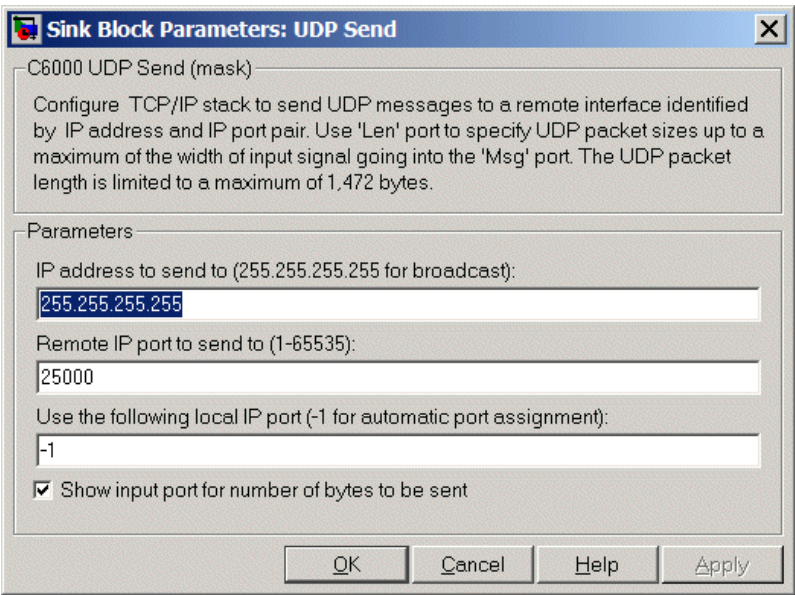
Msg input format must be a `uint8` vector with UDP format. To use variable length messages, supply the message length for each message as input to the Len port. Message length can be any integer value in bytes up to the input width of signal at the Msg port.

C6000 UDP Send blocks operate only to generate code for the target Ethernet driver. They do not perform any function in simulation and they output zero.

Note To use the UDP Send and Receive blocks, for network processing, you must include the C6000 IP Config block to set up the IP drivers for the target Ethernet network.

C6000 UDP Send

Dialog Box



IP address to send to (255.255.255.255 for broadcast)

Specify the IP address to which the block sends the message. If you enter the address 255.255.255.255, the block broadcasts message to any listening IP address. If you enter a specific IP address, you limit the block to sending the message to the specified address.

Remote IP port to send to (1–65535)

Specify the port on the host to which the block sends the message. Port numbers range from 1 to 65535.

Note This port designation must match the port number where you configure the host to receive UDP messages.

Use the following local IP port (-1 for automatic port assignment)

Specify the local IP port the block sends the message from. If you accept the default value of -1, the network automatically selects the local IP port for sending the message.

If the address you are sending to expects the message to come from a specific port, enter that port address in this parameter. If you entered a port number in the UDP Receive block option **Remote IP port to receive from**, enter that port identifier in this parameter also.

Show input port for the number of bytes to be sent

Adds a block input port that lets you specify the number of bytes to send for each UDP message. The maximum allowed value is 1472 bytes. Use the input to dynamically change the length of each message if necessary.

See Also

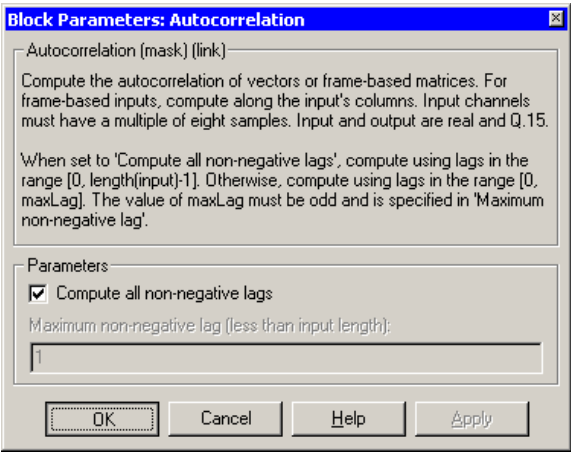
C6000 TCP/IP Receive, C6000 TCP/IP Send, C6000 UDP Receive

C62x Autocorrelation

Purpose	Autocorrelate input vector or frame-based matrix
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11
Description	<p>The Autocorrelation block computes the autocorrelation of an input vector or frame-based matrix. For frame-based inputs, the autocorrelation is computed along each of the input’s columns. The number of samples in the input channels must be an integer multiple of eight. Input and output signals are real and Q.15.</p> <p>Autocorrelation blocks support discrete sample times and little-endian code generation only.</p>



Dialog Box



Compute all non-negative lags

When you select this parameter, the autocorrelation is performed using all nonnegative lags, where the number of lags is one less than the length of the input. The lags produced are therefore in the range $[0, \text{length}(\text{input})-1]$. When this parameter is not selected, you specify the lags used in **Maximum non-negative lag (less than input length)**.

Maximum non-negative lag (less than input length)

Specify the maximum lag (maxLag) the block should use in performing the autocorrelation. The lags used are in the range [0, maxLag]. The maximum lag must be odd. Enable this parameter by clearing the **Compute all non-negative lags** parameter.

Algorithm

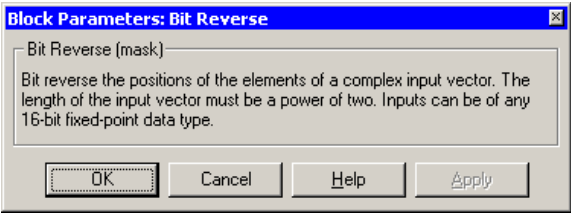
In simulation, the Autocorrelation block is equivalent to the TMS320C62x DSP Library assembly code function DSP_autocor. During code generation, this block calls the DSP_autocor routine to produce optimized code.

C62x Bit Reverse

Purpose	Bit-reverse elements of each complex input signal channel
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Transforms” on page 6-11
Description	<p>The Bit Reverse block bit-reverses the elements of each channel of a complex input signal, X. The Bit Reverse block is primarily used to provide correctly-ordered inputs and outputs to or from blocks that perform FFTs. Inputs to this block must be 16-bit fixed-point data types.</p> <p>The Bit Reverse block supports discrete sample times and little-endian code generation only.</p>



Dialog Box

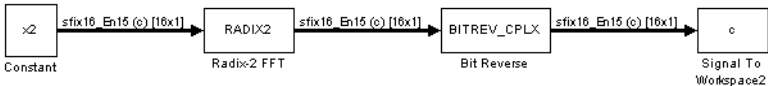


Algorithm

In simulation, the Bit Reverse block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_bitrev_cplx`. During code generation, this block calls the `DSP_bitrev_cplx` routine to produce optimized code.

Examples

The Bit Reverse block reorders the output of the C62xRadix-2 FFT in the model below to natural order.



The following code calculates the same FFT in the workspace. The output from this calculation, `y2`, is displayed side-by-side with the output from the model, `c`. The outputs match, showing that the Bit Reverse block reorders the Radix-2 FFT output to natural order:


```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);
```

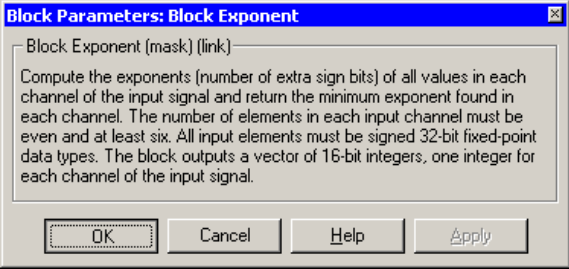
```
[y2, c]
0.5000                0.5000
0.4619 - 0.1913i      0.4619 - 0.1913i
0.3536 - 0.3536i      0.3535 - 0.3535i
0.1913 - 0.4619i      0.1913 - 0.4619i
0 - 0.5000i           0 - 0.5000i
-0.1913 - 0.4619i     -0.1913 - 0.4619i
-0.3536 - 0.3536i     -0.3535 - 0.3535i
-0.4619 - 0.1913i     -0.4619 - 0.1913i
-0.5000                -0.5000
-0.4619 + 0.1913i     -0.4619 + 0.1913i
-0.3536 + 0.3536i     -0.3535 + 0.3535i
-0.1913 + 0.4619i     -0.1913 + 0.4619i
0 + 0.5000i           0 + 0.5000i
0.1913 + 0.4619i      0.1913 + 0.4619i
0.3536 + 0.3536i      0.3535 + 0.3535i
0.4619 + 0.1913i      0.4619 + 0.1913i
```

See Also

C62xRadix-2 FFT, C62xRadix-2 IFFT

C62x Block Exponent

Purpose	Minimum number of extra sign bits in each input channel
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11
Description	<div><div>The icon for the Block Exponent block. It features a black square with a white Texas map outline. Above the map is the text 'TI C62x' in red, and below the map is 'BEXP' in white. The entire icon is enclosed in a white border with input and output ports on the left and right sides. Below the icon is the text 'Block Exponent'.</div><p>The Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be even and at least six. All input elements must be 32-bit signed fixed-point data types. The output is a vector of 16-bit integers — one integer for each channel of the input signal.</p><p>This block is useful for determining whether every sample in a channel is using extra sign bits. If so, you can scale your signal by the minimum number of extra sign bits to eliminate the common extra bits. This increases the representable precision and decreases the representable range of the signal.</p><p>The Block Exponent block supports both continuous and discrete sample times. This block supports little-endian code generation only.</p></div>

Dialog Box	<div>A screenshot of the 'Block Parameters: Block Exponent' dialog box. The title bar is blue with the text 'Block Parameters: Block Exponent'. The main area has a light gray background and contains the text: 'Block Exponent (mask) (link)' followed by a detailed description of the block's function: 'Compute the exponents (number of extra sign bits) of all values in each channel of the input signal and return the minimum exponent found in each channel. The number of elements in each input channel must be even and at least six. All input elements must be signed 32-bit fixed-point data types. The block outputs a vector of 16-bit integers, one integer for each channel of the input signal.' At the bottom, there are four buttons: 'OK', 'Cancel', 'Help', and 'Apply'.</div>
-------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Algorithm	In simulation, the Block Exponent block is equivalent to the TMS320C62x DSP Library assembly code function DSP_bexp. During code generation, this block calls the DSP_bexp routine given to produce optimized code.
------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Purpose

Filter complex input signal using complex FIR filter

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10

Description

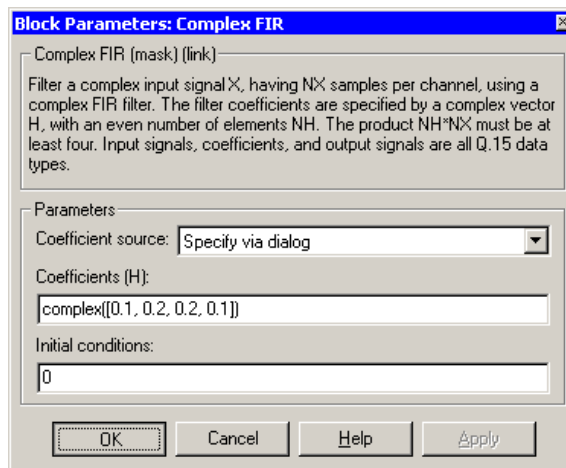


The Complex FIR block filters a complex input signal X using a complex FIR filter. This filter is implemented using a direct form structure.

The number of FIR filter coefficients, which are given as elements of the input vector H , must be even. The product of the number of elements of X and the number of elements of H must be at least four. Inputs, coefficients, and outputs are all $Q.15$ data types.

The Complex FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Coefficients (H)** parameter in the dialog
- **Input port** — Accept the coefficients from port H . This port must have the same rate as the input data port X .

Coefficients (H)

Designate the filter coefficients in vector format. There must be an even number of coefficients. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

You may enter real-valued initial conditions. Zero-valued imaginary parts will be assumed.

Algorithm

In simulation, the Complex FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_cplx`. During code generation, this block calls the `DSP_fir_cplx` routine to produce optimized code.

See Also

C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

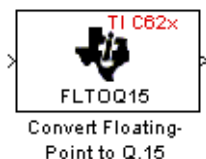
Purpose

Convert single-precision floating-point input signal to Q.15 fixed-point

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Conversions” on page 6-10

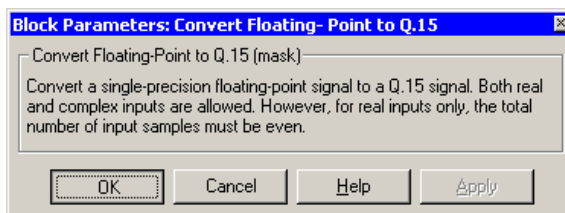
Description



The Convert Floating-Point to Q.15 block converts a single-precision floating-point input signal to a Q.15 output signal. Input can be real or complex. For real inputs, the number of input samples must be even.

The Convert Floating-Point to Q.15 block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



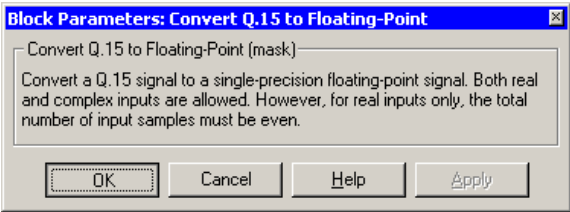
Algorithm

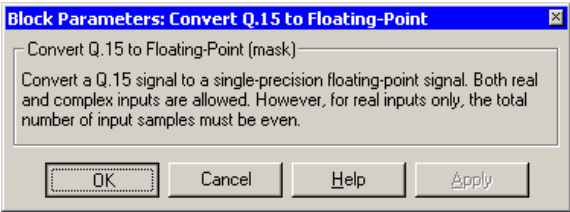
In simulation, the Convert Floating-Point to Q.15 block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fltq15`. During code generation, this block calls the `DSP_fltq15` routine to produce optimized code.

See Also

C62xConvert Q.15 to Floating Point

C62x Convert Q.15 to Floating-Point

Purpose	Convert Q.15 fixed-point signal to single-precision floating-point
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Conversions” on page 6-10
Description	<p>The Convert Q.15 to Floating-Point block converts a Q.15 input signal to a single-precision floating-point output signal. Input can be real or complex. For real inputs, the number of input samples must be even.</p> <p>The Convert Q.15 to Floating-Point block supports both continuous and discrete sample times. This block supports little-endian code generation only.</p>
Dialog Box	
Algorithm	In simulation, the Convert Q.15 to Floating-Point block is equivalent to the TMS320C62x DSP Library assembly code function DSP_q15tof1. During code generation, this block calls the DSP_q15tof1 routine to produce optimized code.
See Also	C62xConvert Floating-Point to Q.15



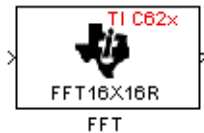
Purpose

Decimation-in-frequency forward FFT of complex input vector

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Transforms” on page 6-11

Description



The FFT block computes the decimation-in-frequency forward FFT, with scaling between stages, of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 8 to 16,384, inclusive. The input must also be in natural (linear) order. The block outputs a complex signal in natural order. Inputs and outputs are signed 16-bit fixed-point data types.

The `fft16x16r` routine used by this block employs butterfly stages to perform the FFT. The number of butterfly stages used, S , depends on the input length $L = 2^k$. If k is even, then $S = k/2$. If k is odd, then $S = (k+1)/2$.

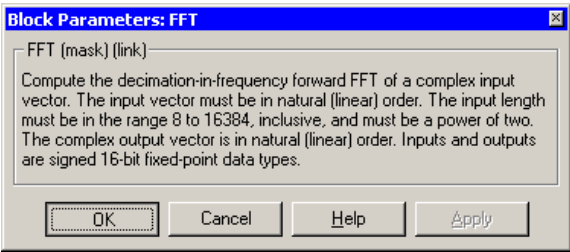
If k is even, then L is a power of two as well as a power of four, and this block performs all S stages with radix-4 butterflies to compute the output. If k is odd, then L is a power of two but not a power of four. In that case this block performs the first $(S-1)$ stages with radix-4 butterflies, followed by a final stage using radix-2 butterflies.

To minimize noise, the FFT block also implements a divide-by-two scaling on the output of each stage except for the last. Therefore, to ensure that the gain of the block matches that of the theoretical FFT, the FFT block offsets the location of the binary point of the output data type by $(S-1)$ bits to the right relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type minus $(S-1)$.

$$\text{OutputFractionalBits} = \text{InputFractionalBits} - (S-1)$$

The FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

**Dialog
Box**



Algorithm

In simulation, the FFT block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fft16x16r`. During code generation, this block calls the `DSP_fft16x16r` routine to produce optimized code.

See Also

C62xRadix-2 FFT, C62xRadix-2 IFFT

Purpose

Filter real input signal using real FIR filter

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10

Description

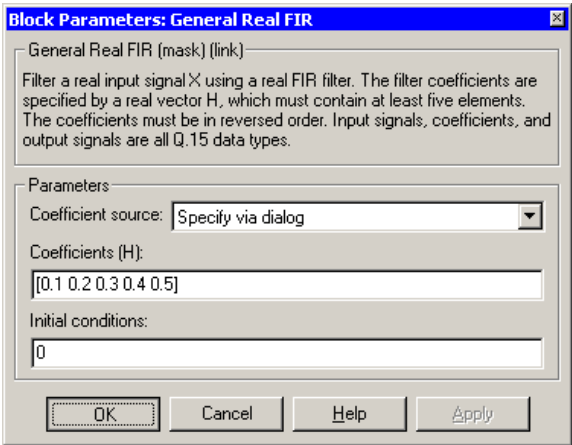


The General Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The filter coefficients are specified by a real vector H , which must contain at least five elements. The coefficients must be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.

The General Real FIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Coefficients (H)** parameter in the dialog
- **Input port** — Accept the coefficients from port H. This port must have the same rate as the input data port X

C62x General Real FIR

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

The initial conditions must be real.

Algorithm

In simulation, the General Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_gen`. During code generation, this block calls the `DSP_fir_gen` routine to produce optimized code.

See Also

C62xComplex FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

Purpose

LMS adaptive FIR filtering

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10

Description



The LMS Adaptive FIR block performs least-mean-square (LMS) adaptive filtering. This filter is implemented using a direct form structure.

The following constraints apply to the inputs and outputs of this block:

- The scalar input X must be a Q.15 data type.
- The scalar input B must be a Q.15 data type.
- The scalar output R is a Q1.30 data type.
- The output \bar{H} has length equal to the number of filter taps and is a Q.15 data type. The number of filter taps must be a positive, even integer.

This block performs LMS adaptive filtering according to the equations

$$e(n+1) = d(n+1) - [\bar{H}(n) \cdot \bar{X}(n+1)]$$

and

$$\bar{H}(n+1) = \bar{H}(n) + [\mu e(n+1) \cdot \bar{X}(n+1)],$$

where

- n designates the time step.
- \bar{X} is a vector composed of the current and last $nH-1$ scalar inputs.
- d is the desired signal. The output R converges to d as the filter converges.
- \bar{H} is a vector composed of the current set of filter taps.

- e is the error, or $d - [\bar{H}(n) \cdot \bar{X}(n+1)]$.
- μ is the step size.

For this block, the input B and the output R are defined by

$$B = \mu e(n+1)$$

and

$$R = \bar{H}(n) \cdot \bar{X}(n+1),$$

which combined with the first two equations, result in the following equations that this block follows:

$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)].$$

d and B must be produced externally to the LMS Adaptive FIR block. Refer to Examples below for a sample model that does this.

The LMS Adaptive FIR block supports discrete sample times and supports little-endian code generation only.

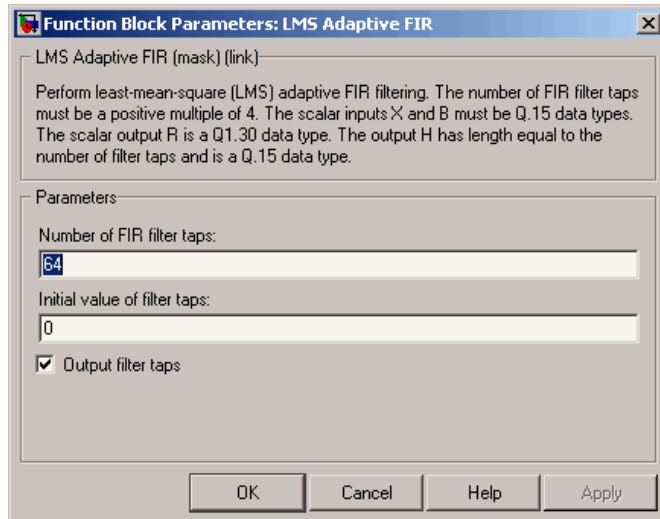
The rounding mode used is *floor*, and the saturation mode is *wrap*. All intermediate products have **s32Q30** data type. The update equation is as follows:

$$H_i = H_i + S16Q15(S32Q30(B) \times S32Q30(X_i))$$
$$R = \sum_N (X_i \times H_i),$$

where N is the number of filter taps.

Note This block does not implement a leaky LMS algorithm, so comparison to the leakage factor of the LMS block of the Signal Processing Blockset software is not appropriate.

Dialog Box



Number of FIR filter taps

Designate the number of filter taps. The number of taps must be a positive, even integer.

Initial value of filter taps

Enter the initial value of the filter taps.

Output filter coefficients H?

If you select this option, the filter taps are produced as output H. If not selected, H is suppressed.

Algorithm

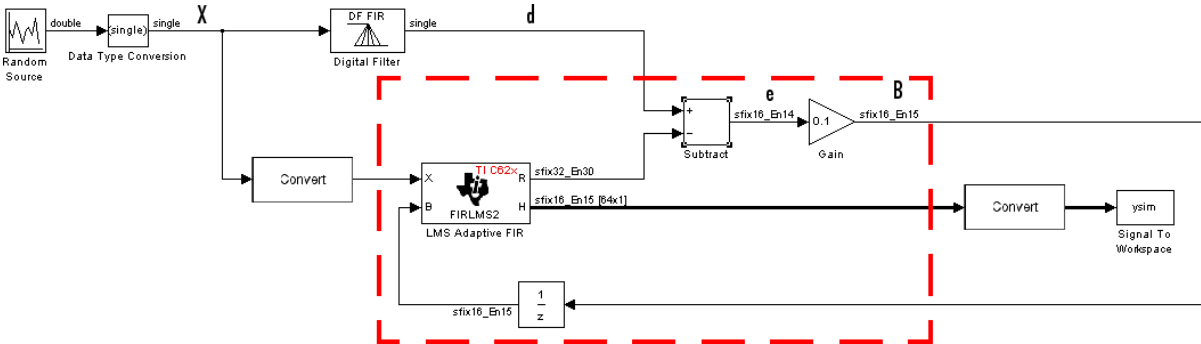
In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_firlms2`.

C62x LMS Adaptive FIR

During code generation, this block calls the DSP_firlms2 routine to produce optimized code.

Examples

The following model uses the LMS Adaptive FIR block.



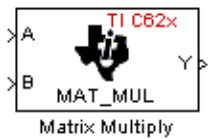
The portion of the model enclosed by the dashed line produces the signal B and feeds it back into the LMS Adaptive FIR block. The inputs to this region are \bar{X} and the desired signal d , and the output of this region is the vector of filter taps \bar{H} . Thus this region of the model acts as a canonical LMS adaptive filter. For example, compare this region to the `adaptfilt.lms` function in Filter Design Toolbox software. `adaptfilt.lms` performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

To use the LMS Adaptive FIR block you must create the input B in some way similar to the one shown here. You must also provide the signals \bar{X} and d . This model simulates the desired signal d by feeding \bar{X} into a digital filter block. You can simulate your desired signal in a similar way, or you may bring d in from the workspace with a From Workspace or codec block.

Purpose Matrix multiply two input signals

Library “C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

Description



The Matrix Multiply block multiplies two input matrices A and B. Inputs and outputs are real, 16-bit, signed fixed-point data types. This block wraps overflows when they occur.

The product of the two 16-bit inputs results in a 32-bit accumulator value. The Matrix Multiply block, however, only outputs 16 bits. You can choose to output the highest or second-highest 16 bits of the accumulator value.

Alternatively, you can choose to output 16 bits according to how many fractional bits you want in the output. The number of fractional bits in the accumulator value is the sum of the fractional bits of the two inputs.

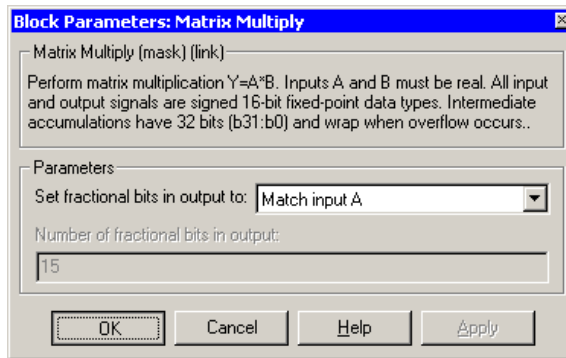
	Input A	Input B	Accumulator Value
Total Bits	16	16	32
Fractional Bits	R	S	$R + S$

Therefore $R+S$ is the location of the binary point in the accumulator value. You can select 16 bits in relation to this fixed position of the accumulator binary point to give the desired number of fractional bits in the output (see Examples below). You can either require the output to have the same number of fractional bits as one of the two inputs, or you can specify the number of output fractional bits in the **Number of fractional bits in output** parameter.

The Matrix Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.

C62x Matrix Multiply

Dialog Box



Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Choose which 16 bits to output from the list:

- **Match input A** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or *R* in the discussion above).
- **Match input B** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or *S* in the discussion above).
- **Match high bits of acc. (b31:b16)** — Output the highest 16 bits of the accumulator value.
- **Match high bits of prod. (b30:b15)** — Output the second-highest 16 bits of the accumulator value.
- **User-defined** — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is enabled only when you select User-defined for **Set fractional bits in output to**.

Algorithm

In simulation, the Matrix Multiply block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mat_mul`. During code generation, this block calls the `DSP_mat_mul` routine to produce optimized code.

Examples

Example 1

Suppose A and B are both Q.15. The data type of the resulting accumulator value is therefore the 32-bit data type Q1.30 ($R + S = 30$). In the accumulator, bits 31:30 are the sign and integer bits, and bits 29:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q.15	b30:b15
Match input B	Q.15	b30:b15
Match high bits of acc.	Q1.14	b31:b16
Match high bits of prod.	Q.15	b30:b15

Example 2

Suppose A is Q12.3 and B is Q10.5. The data type of the resulting accumulator value is therefore Q23.8 ($R + S = 8$). In the accumulator, bits 31:8 are the sign and integer bits, and bits 7:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

C62x Matrix Multiply

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q12.3	b20:b5
Match input B	Q10.5	b18:b3
Match high bits of acc.	Q23.-8	b31:b16
Match high bits of prod.	Q22.-7	b30:b15

See Also

C62xVector Multiply

Purpose

Matrix transpose input signal

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

Description

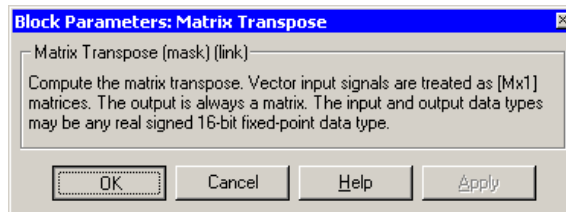


The Matrix Transpose block transposes an input matrix or vector. A 1-D input is treated as a column vector and is transposed to a row vector. Input and output signals are any real, 16-bit, signed fixed-point data type.

The Matrix Transpose block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Note If you use Target Function Library (TFL) technology with this block, the TI compiler generates processor and compiler-specific instructions that improve the performance of the generated code. For more information, consult “Introduction to Target Function Libraries”.

Dialog Box



Algorithm

In simulation, the Matrix Transpose block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mat_trans`. During code generation, this block calls the `DSP_mat_trans` routine to produce optimized code.

C62x Radix-2 FFT

Purpose Radix-2 decimation-in-frequency forward FFT of complex input vector

Library “C62x DSP Library (tic62dsplib)” on page 6-9, “Transforms” on page 6-11

Description The Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types, and the output data type matches the input data type.



You can use the C62x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

The Radix-2 FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box

Block Parameters: Radix-2 FFT

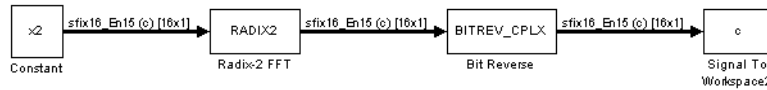
Radix-2 FFT (mask) (link)

Compute the radix-2 decimation-in-frequency forward FFT of a complex input vector. The input vector must be in natural (linear) order. The input length must be in the range 16 to 32768, inclusive, and must be a power of two. The output vector is complex and in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types.

OK Cancel Help Apply

Algorithm In simulation, the Radix-2 FFT block is equivalent to the TMS320C62x DSP Library assembly code function DSP_radix2. During code generation, this block calls the DSP_radix2 routine to produce optimized code.

Examples The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the C62x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.



The following code calculates the same FFT as the above model in the workspace. The output from this calculation, `y2`, is then displayed side-by-side with the output from the model, `c`. The outputs match, showing that the Bit Reverse block does reorder the Radix-2 FFT block output to natural order:

```

k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);
  
```

```

[y2, c]
    0.5000                0.5000
    0.4619 - 0.1913i      0.4619 - 0.1913i
    0.3536 - 0.3536i      0.3535 - 0.3535i
    0.1913 - 0.4619i      0.1913 - 0.4619i
         0 - 0.5000i         0 - 0.5000i
   -0.1913 - 0.4619i   -0.1913 - 0.4619i
   -0.3536 - 0.3536i   -0.3535 - 0.3535i
   -0.4619 - 0.1913i   -0.4619 - 0.1913i
   -0.5000             -0.5000
   -0.4619 + 0.1913i   -0.4619 + 0.1913i
   -0.3536 + 0.3536i   -0.3535 + 0.3535i
   -0.1913 + 0.4619i   -0.1913 + 0.4619i
         0 + 0.5000i         0 + 0.5000i
    0.1913 + 0.4619i    0.1913 + 0.4619i
    0.3536 + 0.3536i    0.3535 + 0.3535i
    0.4619 + 0.1913i    0.4619 + 0.1913i
  
```

See Also

C62x Bit Reverse, C62x FFT, C62x Radix-2 IFFT

C62x Radix-2 IFFT

Purpose	Radix-2 inverse FFT of complex input vector
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Transforms” on page 6-11

Description



The Radix-2 IFFT block computes the radix-2 inverse FFT of each channel of a complex input signal. This block uses a decimation-in-frequency forward FFT algorithm with butterfly weights modified to compute an inverse FFT. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types.

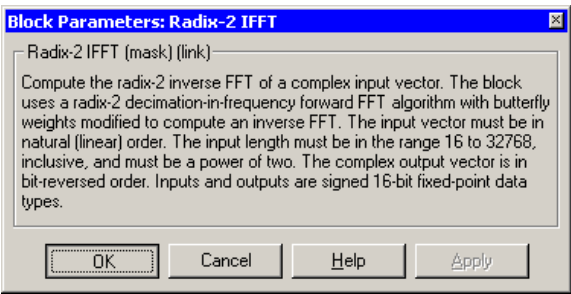
The radix2 routine used by this block employs a radix-2 FFT of length $L=2^k$. To ensure that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by k bits to the left relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type plus k .

$$OutputFractionalBits = InputFractionalBits + (k)$$

You can use the C62x Bit Reverse block to reorder the output of the Radix-2 IFFT block to natural order.

The Radix-2 IFFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box



Algorithm

In simulation, the Radix-2 IFFT block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

See Also

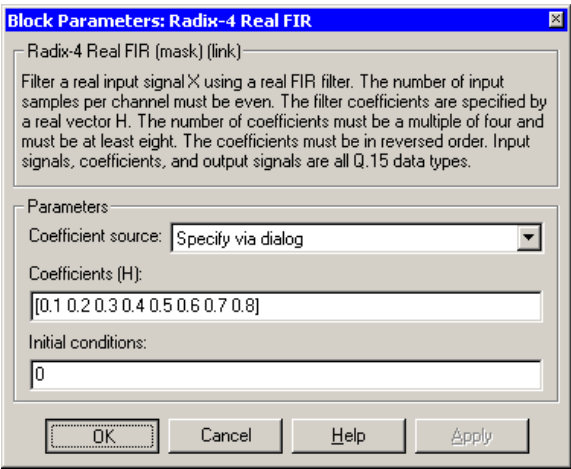
C62x Bit Reverse, C62x FFT, C62x Radix-2 FFT

C62x Radix-4 Real FIR

Purpose	Filter real input signal using real FIR filter
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10
Description	<p>The Radix-4 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.</p> <p>The number of input samples per channel must be even. The filter coefficients are specified by a real vector, H. The number of filter coefficients must be a multiple of four and must be at least eight. The coefficients must also be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.</p> <p>The Radix-4 Real FIR block supports discrete sample times and supports little-endian code generation only.</p>



Dialog Box



- Coefficient source**
- Specify the source of the filter coefficients:
- **Specify via dialog** — Enter the coefficients in the **Coefficients** parameter in the dialog

- **Input port** — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-4 Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_r4`. During code generation, this block calls the `DSP_fir_r4` routine to produce optimized code.

See Also

C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

C62x Radix-8 Real FIR

Purpose Filter real input signal using real FIR filter

Library “C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10

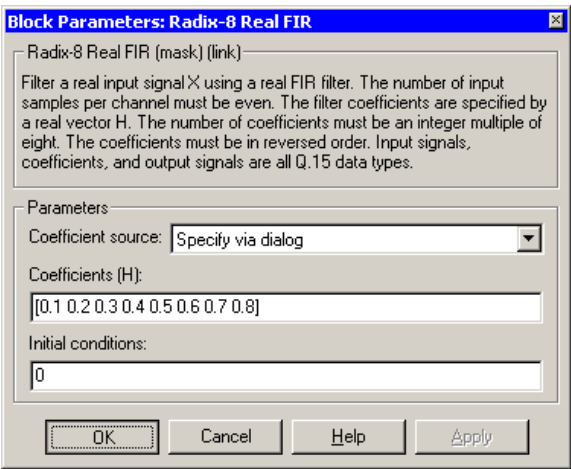
Description The Radix-8 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.



The number of input samples per channel must be even. The filter coefficients are specified by a real vector, H. The number of coefficients must be an integer multiple of eight. The coefficients must be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.

The Radix-8 Real FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Coefficients** parameter in the dialog
- **Input port** — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-8 Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_r8`. During code generation, this block calls the `DSP_fir_r8` routine to produce optimized code.

See Also

C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xSymmetric Real FIR

C62x Real Forward Lattice All-Pole IIR

Purpose Filter real input signal using lattice filter

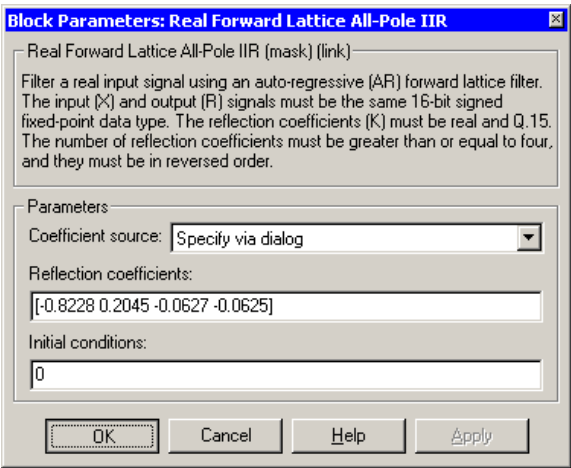
Library “C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10

Description The Real Forward Lattice All-Pole IIR block filters a real input signal using an autoregressive forward lattice filter. The input and output signals must be the same 16-bit signed fixed-point data type. The reflection coefficients must be real and Q.15. The number of reflection coefficients must be greater than or equal to four, and they must be in reversed order. Use an even number of reflection coefficients to maximize the speed of your generated code.



The Real Forward Lattice All-Pole IIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Reflection coefficients** parameter in the dialog
- **Input port** — Accept the coefficients from port K

Reflection coefficients

Designate the reflection coefficients of the filter in vector format. The number of coefficients must be greater than or equal to four, and they must be in reverse order. Using an even number of reflection coefficients maximizes the speed of your generated code. This parameter is visible when you select **Specify via dialog** for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If your block initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length (number of elements) of this vector must be the same as the number of reflection coefficients in your filter.
- Different across channels, enter a matrix containing all initial conditions. The number of rows (initial conditions for one channel) of this matrix must be the same as the number of reflection coefficients, and the number of columns of this matrix must be equal to the number of channels.

Algorithm

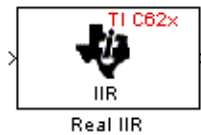
In simulation, the Real Forward Lattice All-Pole IIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_iirlat`. During code generation, this block calls the `DSP_iirlat` routine to produce optimized code.

See Also

C62xReal IIR

C62x Real IIR

Purpose	Filter real input signal using IIR filter
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10
Description	<p>The Real IIR block filters a real input signal X using a real autoregressive moving-average (ARMA) IIR Filter. This filter is implemented using a direct form I structure.</p> <p>There must be five AR coefficients and five MA coefficients. The first AR coefficient is always assumed to be one. Inputs, coefficients, and output are Q.15 data types.</p> <p>The Real IIR block supports discrete sample times and supports little-endian code generation only.</p>



Dialog Box

Block Parameters: Real IIR

Real IIR (mask) (link)

Filter a real input signal X using a real auto-regressive moving-average (ARMA) IIR filter. There must be five AR coefficients and five MA coefficients; however, the first AR coefficient is assumed to be equal to one. Inputs, coefficients, and output are all Q.15 data types.

Parameters

Coefficient sources: Specify via dialog

MA (numerator) coefficients:
[0.1 0.2 0.3 0.4 0.5]

AR (denominator) coefficients:
[1 0.1 0.2 0.3 0.4]

Input state initial conditions:
0

Output state initial conditions:
0

OK

Cancel

Help

Apply

Coefficient sources
Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **MA (numerator) coefficients** and **AR (denominator) coefficients** parameters in the dialog
- **Input ports** — Accept the coefficients from ports MA and AR

MA (numerator) coefficients

Designate the moving-average coefficients of the filter in vector format. There must be five MA coefficients. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

AR (denominator) coefficients

Designate the autoregressive coefficients of the filter in vector format. There must be five AR coefficients, however the first AR coefficient is assumed to be equal to one. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

Input state initial conditions

If the input state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the input state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all input state initial conditions. This matrix must have four rows.

Output state initial conditions

If the output state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the output state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all output state initial conditions. This matrix must have four rows.

C62x Real IIR

Algorithm

In simulation, the Real IIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_iir`. During code generation, this block calls the `DSP_iir` routine to produce optimized code.

See Also

C62xReal Forward Lattice All-Pole IIR

Purpose

Fraction and exponent portions of reciprocal of real input signal

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

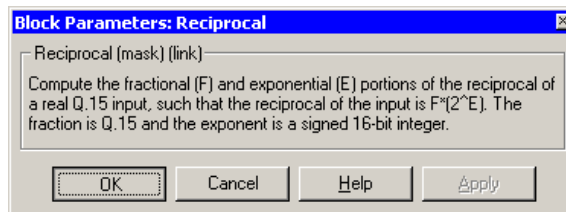
Description



The Reciprocal block computes the fractional (F) and exponential (E) portions of the reciprocal of a real Q.15 input, such that the reciprocal of the input is $F \cdot (2^E)$. The fraction is Q.15 and the exponent is a 16-bit signed integer.

The Reciprocal block supports both continuous and discrete sample times. This block also supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Reciprocal block is equivalent to the TMS320C62x DSP Library assembly code function DSP_recip16. During code generation, this block calls the DSP_recip16 routine to produce optimized code.

C62x Symmetric Real FIR

Purpose Filter real input signal using FIR filter

Library “C62x DSP Library (tic62dsplib)” on page 6-9, “Filters” on page 6-10

Description The Symmetric Real FIR block filters a real input signal using a symmetric real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be even. The filter coefficients are specified by a real vector H, which must be symmetric about its middle element. The number of coefficients must be of the form $16k + 1$, where k is a positive integer. This block wraps overflows that occur. The input, coefficients, and output are 16-bit signed fixed-point data types.



Intermediate multiplies and accumulates performed by this filter result in a 32-bit accumulator value. However, the Symmetric Real FIR block only outputs 16 bits. You can choose to output 16 bits of the accumulator value in one of the following ways.

Match input x	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the input
Match coefficients h	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the coefficients
Match high 16 bits of acc.	Output bits 31 - 16 of the accumulator value

Match high 16 bits of prod.	Output bits 30 - 15 of the accumulator value
User-defined	Output 16 bits of the accumulator value such that the output has the number of fractional bits specified in the Number of fractional bits in output parameter

The Symmetric Real FIR block supports discrete sample times and only little-endian code generation.

Dialog Box

Block Parameters: Symmetric Real FIR

Symmetric Real FIR (mask) (link)

Filter a real input signal X using a symmetric real FIR filter. The number of input samples per channel must be even. The filter coefficients are specified by a real vector H , which must be symmetric about its middle element. The number of elements in H must be of the form $16k+1$ where k is a positive integer. Intermediate accumulations have 32 bits (b31:b0) and use wrap-around arithmetic. All input and output signals are signed 16-bit fixed-point data types.

Parameters

Coefficient source: Specify via dialog

Coefficients:
0.05*(1:17)

Set fractional bits in coefficients to: Best precision

Number of fractional bits in coefficients:
10

Set fractional bits in output to: Match high 16 bits of product (b30:b

Number of fractional bits in output:
10

Initial conditions:
0

OK

Cancel

Help

Apply

C62x Symmetric Real FIR

Coefficient source

Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Coefficients** parameter in the dialog
- **Input port** — Accept the coefficients from port H

Coefficients

Enter the coefficients in vector format. This parameter is visible only when **Specify via dialog** is specified for the **Coefficient source** parameter. This parameter is tunable in simulation.

Set fractional bits in coefficients to

Specify the number of fractional bits in the filter coefficients:

- **Match input X** — Sets the coefficients to have the same number of fractional bits as the input
- **Best precision** — Sets the number of fractional bits of the coefficients such that the coefficients are represented to the best precision possible
- **User-defined** — Sets the number of fractional bits in the coefficients with the **Number of fractional bits in coefficients** parameter

This parameter is visible only when **Specify via dialog** is specified for the **Coefficient source** parameter.

Number of fractional bits in coefficients

Specify the number of bits to the right of the binary point in the filter coefficients. This parameter is visible only when **Specify via dialog** is specified for the **Coefficient source** parameter, and is only enabled if **User-defined** is specified for the **Set fractional bits in coefficients to** parameter.

Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Select which 16 bits to output:

- **Match input X** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input X
- **Match coefficients H** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in coefficients H
- **Match high bits of acc. (b31:b16)** — Output the highest 16 bits of the accumulator value
- **Match high bits of prod. (b30:b15)** — Output the second-highest 16 bits of the accumulator value
- **User-defined** — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter

See Matrix Multiply “Examples” on page 7-65 for demonstrations of these selections.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is only enabled if **User-defined** is selected for the **Set fractional bits in output to** parameter.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less

C62x Symmetric Real FIR

than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Algorithm

In simulation, the Symmetric Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_sym`. During code generation, this block calls the `DSP_fir_sym` routine to produce optimized code.

See Also

C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR

Purpose

Vector dot product of real input signals

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

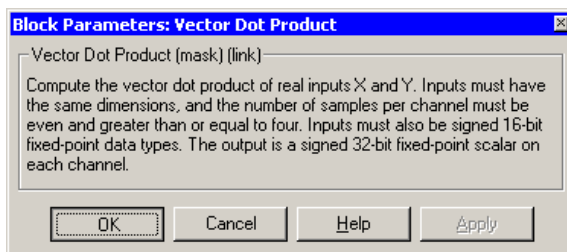
Description



The Vector Dot Product block computes the vector dot product of two real input vectors, X and Y. The input vectors must have the same dimensions and must be signed 16-bit fixed-point data types. The number of samples per channel of the inputs must be even and greater than or equal to four. The output is a signed 32-bit fixed-point scalar on each channel, and the number of fractional bits of the output is equal to the sum of the number of fractional bits of the inputs.

The Vector Dot Product block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

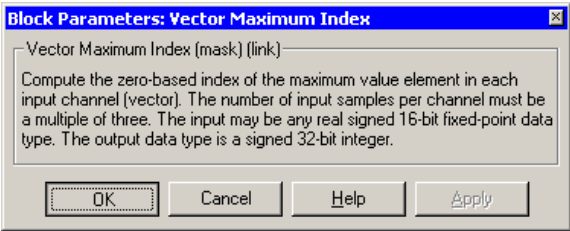
In simulation, the Vector Dot Product block is equivalent to the TMS320C62x DSP Library assembly code function DSP_dotprod. During code generation, this block calls the DSP_dotprod routine to produce optimized code.

C62x Vector Maximum Index

Purpose	Zero-based index of maximum value element in each input signal channel
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11
Description	<p>The Vector Maximum Index block computes the zero-based index of the maximum value element in each channel (vector) of the input signal. The input may be any real, 16-bit, signed fixed-point data type, and the number of samples per input channel must be an integer multiple of three. The output data type is a 32-bit signed integer.</p> <p>The Vector Maximum Index block supports both continuous and discrete sample times. This block supports little-endian code generation only.</p>



Dialog Box



Algorithm	In simulation, the Vector Maximum Index block is equivalent to the TMS320C62x DSP Library assembly code function <code>DSP_maxidx</code> . During code generation, this block calls the <code>DSP_maxidx</code> routine to produce optimized code.
------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Purpose

Maximum value for each input signal channel

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

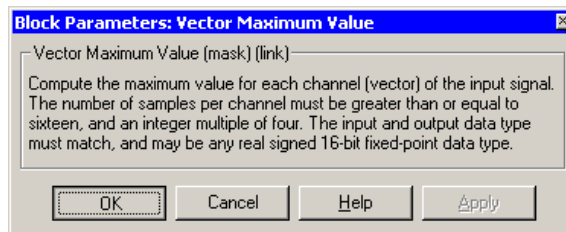
Description



The Vector Maximum Value block returns the maximum value in each channel (vector) of the input signal. The input can be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of four and must be at least 16. The output data type matches the input data type.

The Vector Maximum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Maximum Value block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_maxval`. During code generation, this block calls the `DSP_maxval` routine to produce optimized code.

See Also

C62xVector Minimum Value

C62x Vector Minimum Value

Purpose Minimum value for each input signal channel

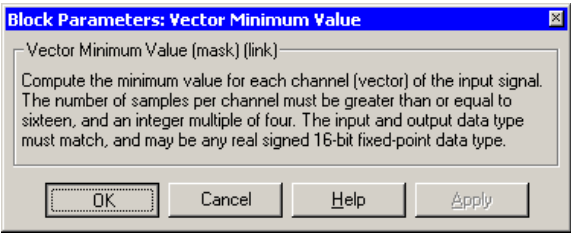
Library “C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

Description The Vector Minimum Value block returns the minimum value in each channel of the input signal. The input may be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of four and must be at least 16. The output data type matches the input data type.



The Vector Minimum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm In simulation, the Vector Minimum Value block is equivalent to the TMS320C62x DSP Library assembly code function DSP_minval. During code generation, this block calls the DSP_minval routine to produce optimized code.

See Also C62xVector Maximum Value

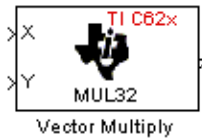
Purpose

Element-wise multiplication on inputs

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

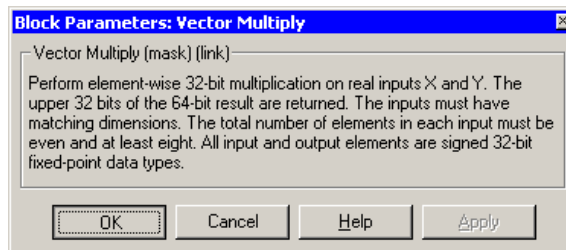
Description



The Vector Multiply block performs element-wise 32-bit multiplication of two inputs X and Y. The total number of elements in each input must be even and at least eight, and the inputs must have matching dimensions. The upper 32 bits of the 64-bit accumulator result are returned. All input and output elements are 32-bit signed fixed-point data types.

The Vector Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Multiply block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mul32`. During code generation, this block calls the `DSP_mul32` routine to produce optimized code.

See Also

C62xMatrix Multiply

C62x Vector Negate

Purpose Negate each input signal element

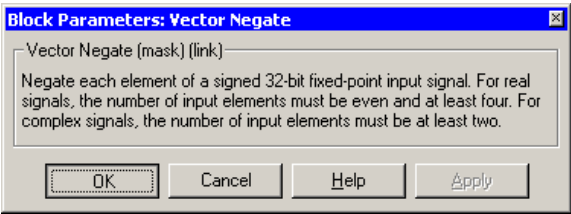
Library “C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

Description The Vector Negate block negates each element of a 32-bit signed fixed-point input signal. For real signals, the number of input elements must be even and at least four. For complex signals, the number of input elements must be at least two. The output is the same data type as the input.



The Vector Negate block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm In simulation, the Vector Negate block is equivalent to the TMS320C62x DSP Library assembly code function DSP_neg32. During code generation, this block calls the DSP_neg32 routine to produce optimized code.

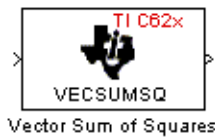
Purpose

Sum of squares over each real input channel

Library

“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11

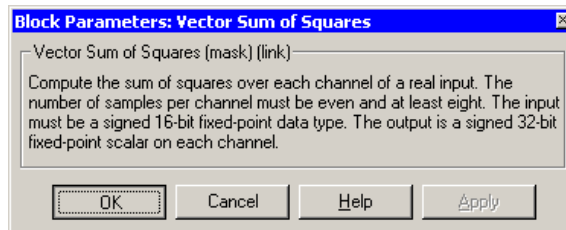
Description



The Vector Sum of Squares block computes the sum of squares over each channel of a real input. The number of samples per input channel must be even and at least eight, and the input must be a 16-bit signed fixed-point data type. The output is a 32-bit signed fixed-point scalar on each channel. The number of fractional bits of the output is twice the number of fractional bits of the input.

The Vector Sum of Squares block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

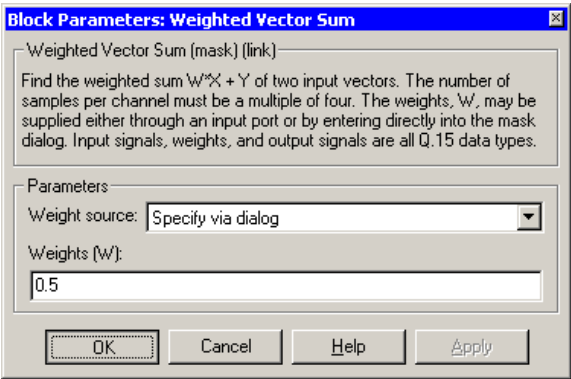
In simulation, the Vector Sum of Squares block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_vecsumsq`. During code generation, this block calls the `DSP_vecsumsq` routine to produce optimized code.

C62x Weighted Vector Sum

Purpose	Weighted sum of input vectors
Library	“C62x DSP Library (tic62dsplib)” on page 6-9, “Math and Matrices” on page 6-11
Description	<p>The Weighted Vector Sum block computes the weighted sum of two inputs, X and Y, according to $(W \cdot X) + Y$. Inputs may be vectors or frame-based matrices. The number of samples per channel must be a multiple of four. Inputs, weights, and output are Q.15 data types, and weights must be in the range $-1 < W < 1$.</p> <p>The Weighted Vector Sum block supports both continuous and discrete sample times. This block supports little-endian code generation only.</p>



Dialog Box



Weight source
Specify the source of the weights:

- **Specify via dialog** — Enter the weights in the **Weights (W)** parameter in the dialog
- **Input port** — Accept the weights from port W

Weights (W)

This parameter is visible only when **Specify via dialog** is specified for the **Weight source** parameter. This parameter is tunable in simulation. When the weights are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be a multiple of four.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be a multiple of four, and the number of columns of this matrix must be equal to the number of channels.

Weights must be in the range $-1 < W < 1$.

Algorithm

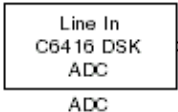
In simulation, the Weighted Vector Sum block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

C6416 DSK ADC

Purpose Digitized output from codec to processor

Library “C6416 DSK (c6416dsklib)” on page 6-3

Description Use the C6416 DSK ADC (analog-to-digital converter) block to capture and digitize analog signals from the analog input jacks on the board. Placing an C6416 DSK ADC block in your Simulink block diagram lets you use the AIC23 coder-decoder module (codec) on the C6416 DSK to convert an analog input signal to a digital signal for the digital signal processor.



Most of the configuration options in the block affect the codec. However, the **Output data type**, **Samples per frame**, and **Scaling** options relate to the model you are using in Simulink software, the signal processor on the board, or direct memory access (DMA) on the board. In the following table, you find each option listed with the C6416 DSK hardware affected.

Option	Affected Hardware
ADC Source	Codec
Mic	Codec
Output data type	TMS320C6416 digital signal processor
Samples per frame	Direct memory access module
Sample Rate	Codec
Scaling	TMS320C6416 digital signal processor
Word Length	Codec

You can select one of two input sources from the **ADC source** list:

- **Line In** — the codec accepts input from the line in connector (LINE IN) on the board’s mounting bracket.

- **Mic** — the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.

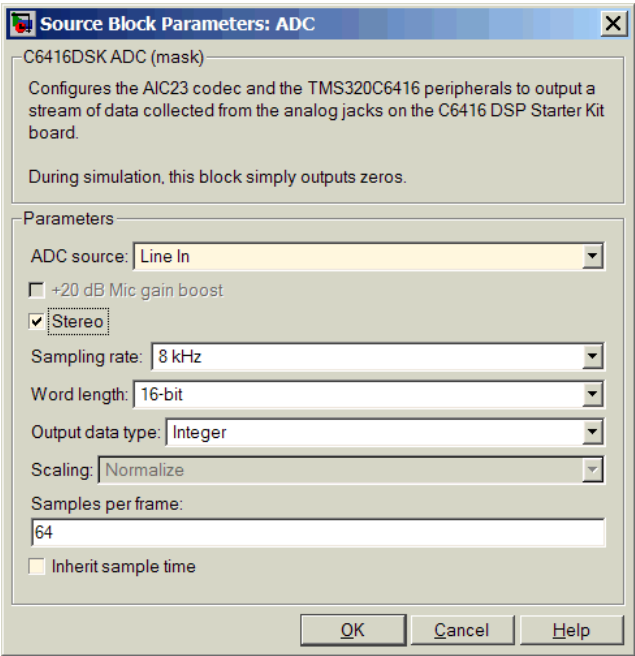
Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels on input and output.

The block uses frame-based processing of inputs, buffering the input data into frames at the specified samples per frame rate. In Simulink software, the block puts monaural data into an N-element column vector. Stereo data input forms an N-by-2 matrix with N data values and two stereo channels (left and right).

When the samples per frame setting is more than one, each frame of data is either the N-element vector (monaural input) or N-by-2 matrix (stereo input). For monaural input, the elements in each frame form the column vector of input audio data. In the stereo format, the frame is the matrix of audio data represented by the matrix rows and columns — the rows are the audio data samples and the columns are the left and right audio channels.

When you select Mic for **ADC source**, you can select the **+20 dB Mic gain boost** check box to add 20 dB to the microphone input signal before the codec digitizes the signal.

Dialog Box



ADC source

The input source to the codec. **Line In** is the default. Selecting **Mic** enables the **+20 dB Mic gain boost** option.

+20 dB Mic gain boost

Boosts the input signal by +20dB when **ADC source** is **Mic**. Gain is applied before analog-to-digital conversion.

Stereo

Indicates whether the input audio data is in monaural or stereo format. Select the check box to enable stereo input. Clear the check box when you input monaural data. By default, stereo is enabled. Monaural data comes from the right channel.

Sample rate

Sets the sample rate for the data output by the codec. Options are 8, 32, 44.1, 48, and 96 kHz, with a default of 8 kHz.

Word length

Sets the resolution with which the ADC samples the analog input. Increasing the word length increases the accuracy of the data in each sample. If your model also contains a DAC block, set its word length match that of the ADC block.

Output data type

Selects the word length and shape of the data from the codec. By default, `double` is selected. Options are `Double`, `Single`, and `Integer`. To process single and double data types, the block uses emulated floating-point instructions on the C6416 processor.

Scaling

Selects whether the codec data is unmodified, or normalized to the output range to ± 1.0 , based on the codec data format. Select either `Normalize` or `Integer` from the list. `Normalize` is the default setting.

Samples per frame

Creates frame-based outputs from sample-based inputs. This parameter specifies the number of samples of the signal the block buffers internally before it sends the digitized signals, as a frame vector, to the next block in the model. This value defaults to 64 samples per frame. Notice that the frame rate depends on the sample rate and frame size. For example, if your input is 8000 samples per second, and you select 32 samples per frame, the frame rate is 250 frames per second. The throughput remains the same at 8000 samples per second.

Inherit sample time

Selects whether the block inherits the sample time from the model base rate or Simulink base rate as determined in the Solver options in Configuration Parameters. Selecting **Inherit sample time** directs the block to use the specified rate in model configuration. Entering -1 configures the block to accept the sample rate from the upstream HWI, Task, or Triggered Task blocks.

See Also

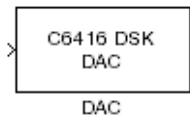
C6416 DSK DAC

C6416 DSK DAC

Purpose Use codec to convert digital input to analog output

Library “C6416 DSK (c6416dsklib)” on page 6-3

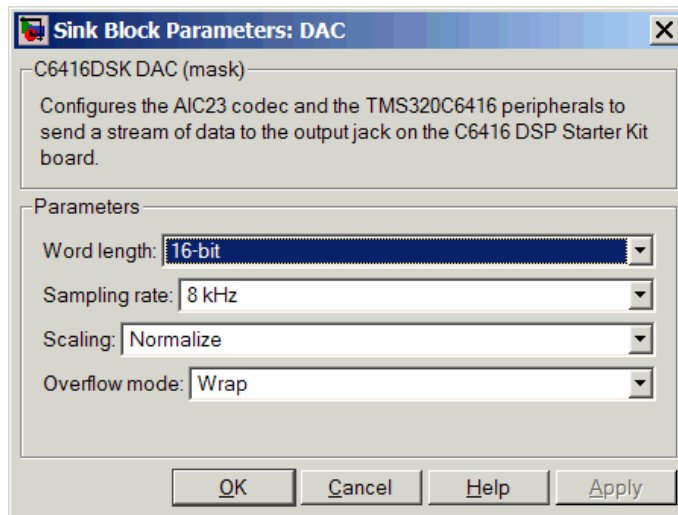
Description Adding the C6416 DSK DAC (digital-to-analog converter) block to your Simulink model lets you output an analog signal to the LINE OUT connection on the C6416 DSK board. When you add the C6416 DSK DAC block, the digital signal received by the codec is converted to an analog signal and sent to the output jack.



Only the **Word length** option in the block affects the codec. The other options relate to the model you are using in Simulink software and the signal processor on the board. Refer to the following table for information.

Option	Affected Hardware
Overflow mode	TMS320C6416 Digital Signal Processor
Scaling	TMS320C6416 Digital Signal Processor
Word length	Codec

Dialog Box



Word length

Sets the DAC to interpret the input data word length. Without this setting, the DAC cannot convert the digital data to analog correctly. The value defaults to 16 bits, with options of 20, 24, and 32 bits. The word length you set here should always match the ADC setting.

Sampling rate

Sets the sampling rate for the block output to the output ports on the target. Select from the list of available rates.

Scaling

Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range ± 1.0 . Matching the setting for the C6416 DSK ADC block is usually appropriate here.

Overflow mode

Determines how the codec responds to data that is outside the range specified by the **Scaling** parameter. You can choose **Wrap** or **Saturate** to handle the result of an overflow in an operation. If efficient operation matters, **Wrap** is the more efficient mode.

C6416 DSK DAC

See Also C6416 DSK ADC

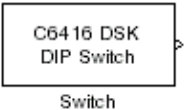
Purpose

Simulate or read DIP switches

Library

“C6416 DSK (c6416dsklib)” on page 6-3

Description



Added to your model, this block behaves differently in simulation than in code generation and targeting.

In Simulation — the options **Switch 0**, **Switch 1**, **Switch 2**, and **Switch 3** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6416 DSK. Each option turns the associated DIP switch on when you select it. The switches are independent of one another.

By defining the switches to represent actions on your target, DIP switches let you modify the operation of your process by reconfiguring the switch settings.

Use the **Data type** to specify whether the DIP switch options output an integer or a logical string of bits to represent the status of the switches. The table that follows presents all the option setting combinations with the result of your **Data type** selection.

Option Settings to Simulate the User DIP Switches on the C6416 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Cleared	Cleared	Cleared	0000	0
Selected	Cleared	Cleared	Cleared	0001	1
Cleared	Selected	Cleared	Cleared	0010	2
Selected	Selected	Cleared	Cleared	0011	3
Cleared	Cleared	Selected	Cleared	0100	4
Selected	Cleared	Selected	Cleared	0101	5
Cleared	Selected	Selected	Cleared	0110	6

C6416 DSK DIP Switch

Option Settings to Simulate the User DIP Switches on the C6416 DSK (Continued)

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Selected	Selected	Selected	Cleared	0111	7
Cleared	Cleared	Cleared	Selected	1000	8
Selected	Cleared	Cleared	Selected	1001	9
Cleared	Selected	Cleared	Selected	1010	10
Selected	Selected	Cleared	Selected	1011	11
Cleared	Cleared	Selected	Selected	1100	12
Selected	Cleared	Selected	Selected	1101	13
Cleared	Selected	Selected	Selected	1110	14
Selected	Selected	Selected	Selected	1111	15

Selecting the Integer data type results in the switch settings generating integers in the range from 0 to 15 (uint8), corresponding to converting the string of individual switch settings to a decimal value. In the Boolean data type, the output string presents the separate switch setting for each switch, with the **Switch 0** status represented by the least significant bit (LSB) and the status of **Switch 3** represented by the most significant bit (MSB).

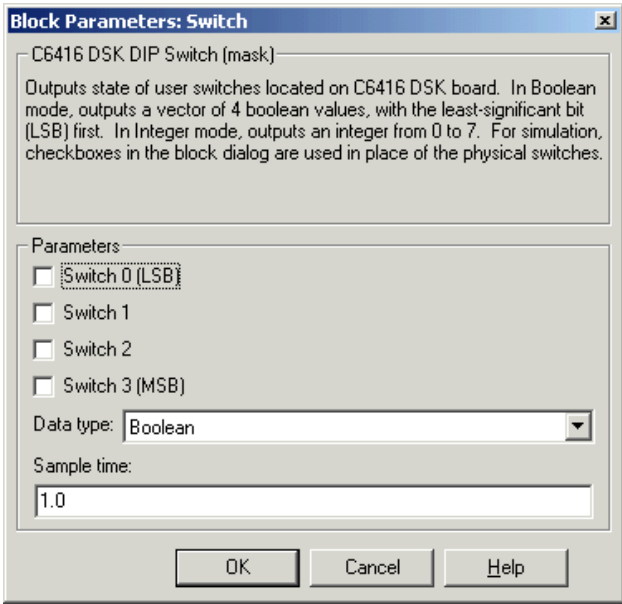
In Code generation and targeting — the code generated by the block reads the physical switch settings of the user switches on the board and reports them as shown in the table above. Your process uses the result in the same way whether in simulation or in code generation. In code generation and when running your application, the block code ignores the settings for **Switch 0**, **Switch 1**, **Switch 2** and **Switch 3** in favor of reading the hardware switch settings. When the block reads the DIP switches, it reports the results as either a Boolean string or an integer value as the following table shows.

Output Values From The User DIP Switches on the C6416 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2
On	On	Off	Off	0011	3
Off	Off	On	Off	0100	4
On	Off	On	Off	0101	5
Off	On	On	Off	0110	6
On	On	On	Off	0111	7
Off	Off	Off	On	1000	8
On	Off	Off	On	1001	9
Off	On	Off	On	1010	10
On	On	Off	On	1011	11
Off	Off	On	On	1100	12
On	Off	On	On	1101	13
Off	On	On	On	1110	14
On	On	On	On	1111	15

C6416 DSK DIP Switch

Dialog Box



Opening this dialog causes a running simulation to pause. Refer to “Changing Source Block Parameters During Simulation” in your online Simulink documentation for details.

Switch 0

Simulate the status of the user-defined DIP switch on the board.

Switch 1

Simulate the status of the user-defined DIP switch on the board.

Switch 2

Simulate the status of the user-defined DIP switch on the board.

Switch 3

Simulate the status of the user-defined DIP switch on the board.

Data type

Determines how the block reports the status of the user-defined DIP switches. **Boolean** is the default, indicating that the output is a vector of four logical values.

Each vector element represents the status of one DIP switch; the first is **Switch 0** and the fourth is **Switch 3**. The data type **Integer** converts the logical string to an equivalent unsigned 8-bit (**uint8**) value. For example, when the logical string generated by the switches is 0101, the conversion yields 5 — the MSB is 0 and the LSB is 1.

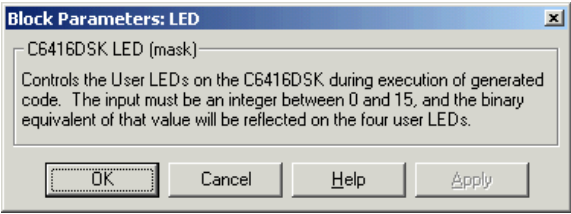
Sample time

Specifies the time between samples of the signal. This value defaults to 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

C6416 DSK LED

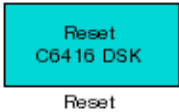
Purpose	Control LEDs
Library	“C6416 DSK (c6416dsklib)” on page 6-3
Description	<p>Adding the C6416 DSK LED block to your Simulink block diagram lets you trigger the user light emitting diodes (LED) on the C6416 DSK. To use the block, send a nonzero real scalar to the block. The C6416 DSK LED block controls all four User LEDs located on the C6416 DSK.</p> <p>When you add this block to a model, and send an integer to the block input, the block sets the LED state based on the input value it receives:</p> <ul style="list-style-type: none">• When the block receives an input value equal to 0, the specified LEDs are turned off (disabled), 0000• When the block receives a nonzero input value, the specified LEDs are turned on (enabled), 0001 to 1111 <p>To activate the block, send it an integer in the range 0 to 15. Vectors do not work to activate LEDs; nor do complex numbers as scalars or vectors.</p> <p>For example, sending the value 6 turns on the diodes to show 0110 (off/on/on/off). 13 turns on the diodes to show 1101.</p> <p>All LEDs maintain their state until the C6416 DSK LED block receives an input value that changes the state. Enabled LEDs stay on until the block receives an input value that turns the LEDs off; disabled LEDs stay off until turned on. Resetting the C6416 DSK turns off all User LEDs. When you start an application, the LEDs are turned off by default.</p>

Dialog Box



This dialog does not have any user-selectable options.

C6416 DSK Reset

Purpose	Reset to initial conditions
Library	“C6416 DSK (c6416dsklib)” on page 6-3
Description	<div>The image shows a cyan rectangular block with a black border. Inside the block, the text "Reset" is on the top line and "C6416 DSK" is on the bottom line, both in black. Below the block, the word "Reset" is written in a smaller black font.</div> <p>Double-clicking this block in a Simulink model window resets the C6416 DSK that is running the executable code built from the model. When you double-click the C6416 DSK Reset block, the block runs the software reset function provided by CCS IDE that resets the processor on your C6416 DSK. Applications running on the board stop and the signal processor returns to the initial conditions you defined.</p> <p>Before you build and download your model, add the block to the model as a stand-alone block. You do not need to connect the block to any block in the model. When you double-click this block in the block library, it resets your C6416 DSK. In other words, any time you double-click a C6416 DSK Reset block, you reset your C6416 DSK.</p>
Dialog Box	This block does not have settable options and does not provide a user interface dialog.

Purpose

Configure AIC23 audio codec to capture audio stream from LINE-IN or MIC

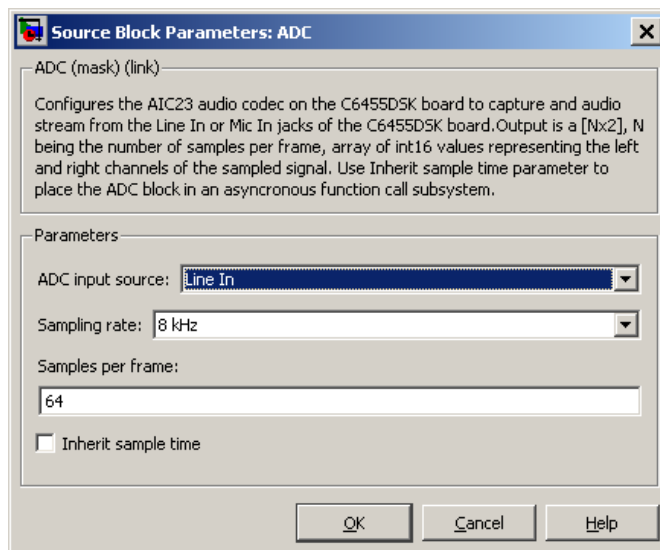
Library

“C6455 EVM (c6455evmlib)” on page 6-4

Description

This block uses the AIC23 audio codec on the C6455 DSK board to capture an analog audio stream from the **Line In** or **Mic** jacks and generate a digital frame-based output. Output is a $[N \times 2]$ array of int16 values representing the left and right channels of the sampled signal, where N is the number of samples per frame. Use the **Inherit sample time** parameter to place the ADC block in an asynchronous function call subsystem.

Dialog Box



ADC input source

Select **Line In** or **Mic In** as the input source.

Sampling Rate

Set the sampling rate of the analog-to-digital converter. Increasing the frequency increases the accuracy of the sampling data over time.

Samples per frame

Set the number of samples the block buffers internally before it sends the digitized signals, as a frame vector, to the next block in the model. This value defaults to 64 samples per frame. The frame rate depends on the sample rate and frame size. For example, if **Sampling Rate** is 8 kHz, and **Samples per frame** is 32, the frame rate is 250 frames per second ($8000/32 = 250$).

Inherit sample time

Select whether the block inherits the sample time from the model base rate or Simulink base rate as determined in the Solver options in Configuration Parameters. Selecting Inherit sample time directs the block to use the specified rate in model configuration. Entering -1 configures the block to accept the sample rate from the upstream HWI, Task, or Triggered Task blocks.

See Also

DM6437 EVM DAC

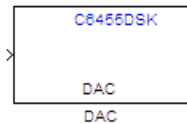
Purpose

Configure AIC23 codec to convert digital signal to audio output on LINE OUT and HP OUT

Library

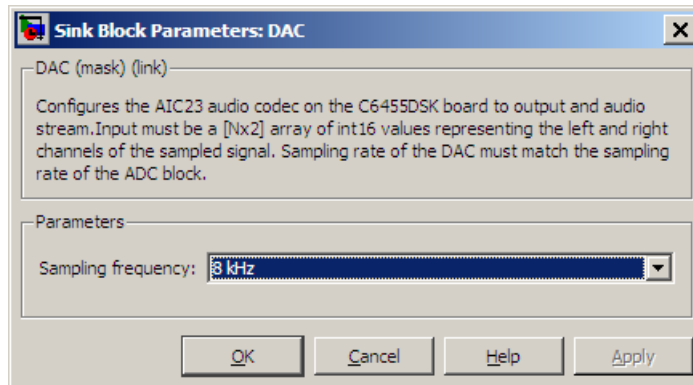
“C6455 EVM (c6455evmlib)” on page 6-4

Description



Configure the AIC23 stereo codec on the C6455 EVM board to convert a digital signal to an analog audio stream on the LINE OUT and HP OUT output jacks. The digital signal input must be an [Nx2] array of int16 values. Column 1 of the array is the left channel and column 2 is the right channel of the sampled signal. The sampling rate of the DAC output must match the sampling rate of the digital signal from the ADC.

Dialog Box



Sampling Frequency

Set the sampling rate of the digital-to-analog converter. The rate defaults to 8 kHz. Options range up to 96 kHz.

See Also

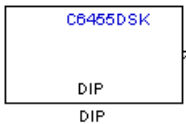
C6455 DSK ADC

C6455 DSK DIP

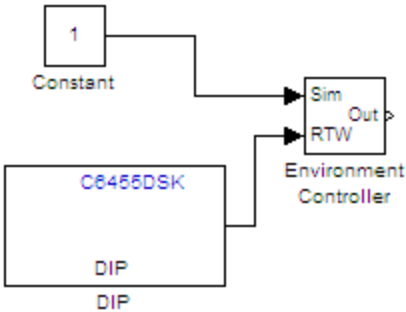
Purpose Output state of user-selected DIP switch as Boolean

Library “C6455 EVM (c6455evmlib)” on page 6-4

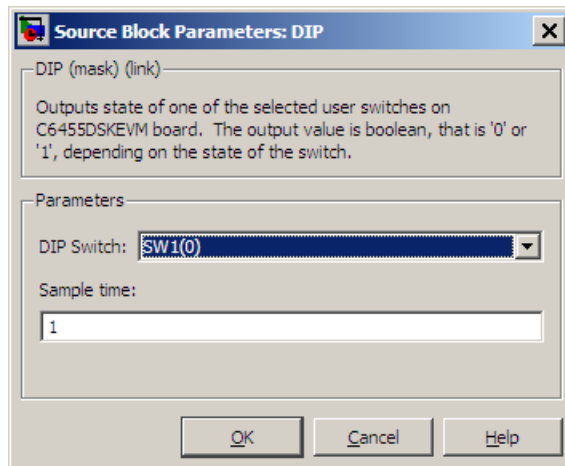
Description Outputs a Boolean that gives the state of a user-selected DIP switch from the SW1 bank of switches on the C6455 DSK/EVM board. Boolean 0 means the switch is open, and Boolean 1 means it is closed. Use multiple blocks to output the state of multiple DIP switches.



For simulations, you may want to use the C6455 DSK DIP block with a Constant block and an Environment Controller block, both from the Simulink block libraries.



Dialog Box



DIP Switch

Select the switch, 0 through 3, from the SW1 bank of switches.

Sample Time

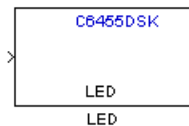
Specifies the time between samples of the signal in seconds. This value defaults to 1 second between samples.

C6455 DSK LED

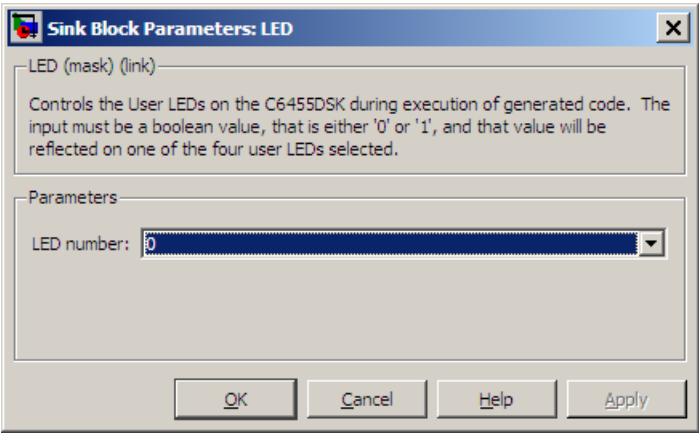
Purpose Apply Boolean input to user-selected LED

Library “C6455 EVM (c6455evmlib)” on page 6-4

Description This block controls an individual LED among the User LEDs on the C6455 DSK during execution of generated code. The block input accepts Boolean values, 0 (off) or 1 (on). Use multiple blocks to control multiple LEDs.



Dialog Box



LED number Specify the number of the User LED that the Boolean input controls.

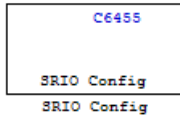
Purpose

Configure generated code for serial RapidI/O peripheral

Library

“C6455 EVM (c6455evmlib)” on page 6-4

Description



The C6455 processor supports the serial RapidI/O (SRIO) peripheral from Texas Instruments for high-speed packet-switched chip-to-chip and board-to-board communications. This block provides the parameters you use to configure the SRIO peripheral on your hardware to communicate between different processors.

The dialog box parameters that you set provide values to initialize the registers on the processor relevant to SRIO processing.

Because SRIO handles communications between two platforms, it requires two models or sets of code—one running on the local device and one running on the remote device. Both models must include the SRIO Config block to configure their SRIO communications capability, and the blocks must have the correct device IDs to refer to one another.

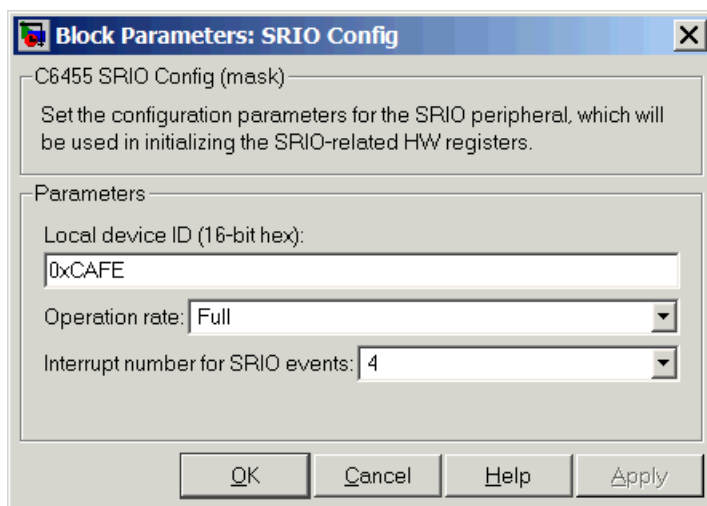
SRIO blocks implement both direct I/O and doorbell interrupt forms of SRIO communications. Direct I/O provides data transfer directly between two processors. With direct I/O you have burst-write and burst-read access with the remote device. The block configures the SRIO peripheral as a 4x SRIO, meaning that all four links of SRIO are bundled together for the fastest link. Direct I/O uses the Load/Store Unit (LSU) and Direct Memory Access (DMA) Engine to control and monitor the data transfer.

Doorbell interrupt enables the local device to initiate CPU interrupts on the remote device if burst-write access is enabled. Such interrupts signal that data is ready to transfer. Both devices, local (source) and remote (destination) include doorbell message queues. The destination device reads its queue to determine the interrupt source and to process the doorbell INFO field.

To see the SRIO blocks in use, refer to the Interprocessor Communications via Serial Rapid I/O (SRIO) demo, located in the online help system demos for Target Support Package software.

C6455 DSK SRIO Config

Dialog Box



Local device ID (16-bit hex)

Enter the ID of the local device to configure the device ID field in the generated code. Use a 16-bit hexadecimal format. When you configure SRIO Transmit and SRIO Receive blocks in models, the local device ID in this field must match the remote device ID for the Transmit and Receive block in each model.

In the generated code, you see the input device ID as a constant mapped to the following program code entry.

```
#define SRIO_LARGE_DEV_ID 0xCAFE
```

Operation rate

Set the operating frequency of the SRIO serializer/deserializer (SERDES). Two variables determine the primary operating frequency of the SERDES, the reference clock frequency and PLL multiplication factor. Select Full, Half, or Quarter from the list.

- Full takes two data samples for each PLL output clock cycle.
- Half takes one data sample for each PLL output clock cycle.

- Quarter takes one data sample and a delay for two PLL output cycles

This value defaults to Full.

Interrupt number for SRIO events

Assigns an interrupt number to initiate for SRIO events. After you select a value from the list, you see a constant similar to the following defined in the generated code

```
#define SRIO_INTR_NUMBER 4
```

References

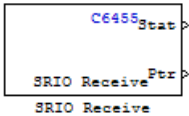
For more information about SRIO, refer to *TMS320TCI648x Serial RapidIO User's Guide*, Literature Number: SPRUE13. Texas Instruments Incorporated.

C6455 DSK SRIO Receive

Purpose Configure generated code to receive serial RapidI/O packets

Library “C6455 EVM (c6455evmlib)” on page 6-4

Description SRIO receive blocks add the ability to receive SRIO packets to the processor that is running the embedded code. Each receive block has two output ports—theStat port that is permanent and the optional Ptr port, that report the status of the block and output a pointer to data.



Writing data between DSPs is more efficient than writing because SRIO write can handle up to 4kB per write request without stalling the processor while SRIO read only handles up to 256 bytes per read request. Thus, the time needed to transfer data by reading from the remote device can be much longer than that required for writing from the remote device. Use the doorbell interrupt options to signal remote devices and to coordinate the data transfer between processors.

The Stat port reports SRIO operating status as shown in the following table.

Value at Stat Port	Description
1	SRIO request is done (success)
0	SRIO request is pending
-1	SRIO request failed
-2	SRIO request was not sent because the SRIO request queue is full

To see the SRIO blocks in use, refer to the Interprocessor Communications via Serial Rapid I/O (SRIO) demo in the online help system demos for Target Support Package software.

Dialog Box

The block dialog box provides parameters on two panes:

- **Main** pane includes parameters that configure the data transfer operation, the doorbell interrupt ID, and various address settings for the remote device and host.
- “Data Types Pane” on page 7-34 parameters configure the data type and size that the block reads.

C6455 DSK SRIO Receive

Main Pane

Source Block Parameters: SRIO Receive

C6455 SRIO Receive (mask)
Configure the SRIO peripheral to accept doorbell interrupt and/or read data from remote device.

Main

Data Properties

Remote device ID (16-bit hex):

0xCAFE

☒ Accept doorbell interrupt from remote device

Doorbell interrupt ID:

0

☒ Read from remote device

Remote address (32-bit hex aligned to an 8-byte boundary):

0x00900000

☒ Show output port for local address pointer

Local address (32-bit hex aligned to an 8-byte boundary):

0x00900500

☒ Enable blocking mode

Sample time:

0.01

Timeout value:

inf

OK

Cancel

Help

Remote device ID (16-bit hex)

Enter the ID of the remote device in 16-bit hexadecimal format to configure the remote ID field in the generated code. When you configure SRIO Receive blocks for this communication link, the remote device ID in this field must match the local device ID for the SRIO Config block in the transmitting model.

Accept doorbell interrupt from remote device

Enables the doorbell interrupt operation for the block. The block always waits until it receives a doorbell interrupt before it reads from the remote device. Selecting this option enables the **Doorbell interrupt ID** parameter so you can set the interrupt ID.

Doorbell interrupt ID

Sets the interrupt ID for the doorbell to determine which SRIO Receive block should be awakened based on the incoming interrupt value. Select a value from the list. If your model contains more than one SRIO receive block, each receive block must use a different ID. IDs range from 0 to 15 with a default value of 0. SRIO Receive and SRIO Transmit blocks are paired together by this ID. Create an SRIO Transmit block with this ID to send the doorbell interrupt.

Read from remote device

Selecting this option tells the block to perform a burst read from the remote device at the address in **Remote address**. If you clear this option, you must select **Accept doorbell interrupt from remote device**.

Remote address (32-bit hex aligned to an 8-byte boundary)

This address specifies where the data is being read from the remote device. The address you enter here should match the local address of the corresponding SRIO Transmit block.

This address should align to an 8-byte boundary in memory.

Show output port for local address pointer

When you select this parameter, the output port Ptr returns the pointer that you specify in **Local address (32-bit hex aligned to an 8 byte boundary)**. Clearing this option removes the Ptr port from the block.

Local address (32-bit hex aligned to an 8 byte boundary)

This address specifies the destination for the data to transfer. This address should match the remote address of the corresponding

SRIO Transmit block. You will need it if the SRIO Transmit block performs burst-write operations.

Enable blocking mode

SRIO receive blocks can operate in either blocking or nonblocking modes.

- Selecting this option puts the block in blocking mode and the block waits for a doorbell interrupt to come or timeout to occur before passing program control to downstream blocks or performing any read operations.
 - Clearing **Enable blocking mode** directs the block to poll the doorbell interrupt status register to determine whether the SRIO Transmit block sent a doorbell packet.
 - Sending the packet indicates that the transmitting block completed a data transfer to this block.
- Clearing this option to put the block in nonblocking mode enables the **Sample time** option. In nonblocking mode, Simulink software uses the sample time to determine the polling period the block uses for polling the interrupt status register.

Enable blocking mode is not available when you clear **Enable doorbell**. Clearing **Accept doorbell interrupt from remote device** also disables this option because blocking mode refers to the doorbell interrupt process.

Sample time

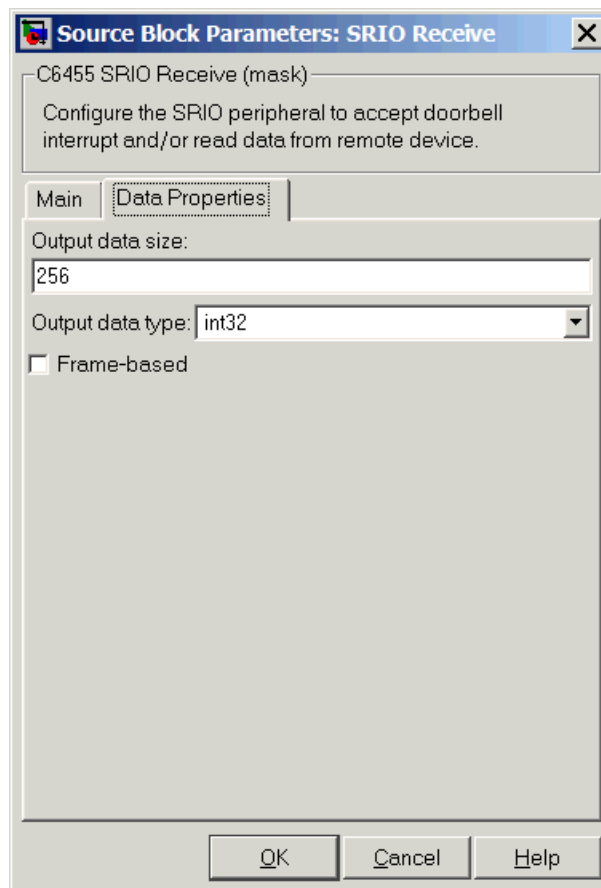
Determines the polling period, in seconds, for the block in nonblocking mode. Enter the time period to wait between polls. To enable this option, clear **Enable blocking mode** and select **Accept doorbell interrupt from remote device**.

Timeout value

In blocking mode, this value determines how long the block waits for a doorbell interrupt before it sets the **Stat** output port to Timeout status. Enter a time in seconds (The value defaults

to inf to block until the block receives a doorbell interrupt). The default time-out value is 1 second. Clearing either **Enable blocking mode** or **Accept doorbell interrupt from remote device** disables this option.

Data Properties Pane



Output data size

Use this to specify the amount of data in bytes to transfer. Enter either a scalar to define a vector of elements or a two-element array. For example, enter 256 to specify a vector of 256 elements. To specify a two-dimensional array of 512 elements, enter [256 2]. The block uses this value to determine the size of the **Ptr** port. If you select the **Frame-based** option, you must enter the vector, or scalar value, as an array. Thus the 256-element vector example entry becomes [256 1].

Output data type

Specify the data type used for the output. With this information, the block calculates the size of the data transfer in bytes using this value and the **Output data size** value.

Frame-based

When you select this option, the block treats the data as frame-based rather than sample-based. If you select **Frame-based**, you must enter your output data size as a two-element array. For example, to specify a vector that contains 256 elements, enter [256 1].

References

For more information about SRIO, refer to *TMS320TCI648x Serial RapidIO User's Guide*, Literature Number: SPRUE13. Texas Instruments Incorporated.

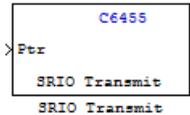
Purpose

Configure generated code to transmit serial RapidI/O packets

Library

“C6455 EVM (c6455evmlib)” on page 6-4

Description



SRIO transmit blocks add the ability to transmit SRIO packets to another processor. Each transmit block has an input **Ptr** port, and an optional **Stat** output port controlled by the **Show output port for status** option.

Writing data between DSPs is more efficient than reading because SRIO write can handle up to 4kB per write request without stalling the processor while SRIO read only handles up to 256 bytes per read request. Thus, the time needed to transfer data by reading from the remote device can be much longer than that required for writing from the remote device. SRIO read may require multiple requests. Use the doorbell interrupt options signal remote devices and to coordinate the data transfer between the processors.

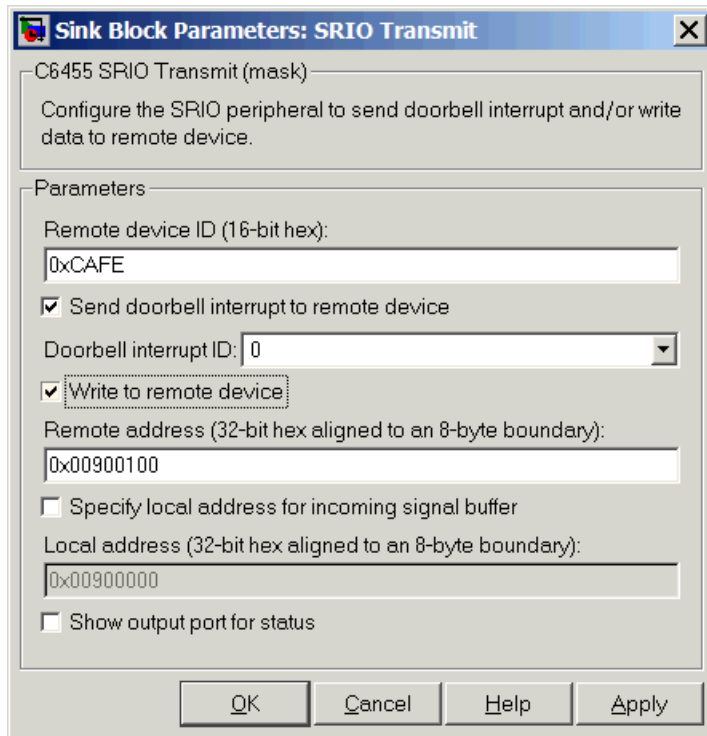
The **Stat** port reports SRIO operating status as shown in the following table.

Value at Stat Port	Description
1	SRIO request is done (success)
0	SRIO request is pending
-1	SRIO request failed
-2	SRIO request was not sent because the SRIO request queue is full

To see the SRIO blocks in use, refer to the Interprocessor Communications via Serial Rapid I/O (SRIO) demo in the online help system demos for Target Support Package software.

C6455 DSK SRIO Transmit

Dialog Box



The dialog box is titled "Sink Block Parameters: SRIO Transmit". It contains a description of the block's function and a section for configuring parameters.

C6455 SRIO Transmit (mask)
Configure the SRIO peripheral to send doorbell interrupt and/or write data to remote device.

Parameters

Remote device ID (16-bit hex):

☒ Send doorbell interrupt to remote device
Doorbell interrupt ID:

☒ Write to remote device
Remote address (32-bit hex aligned to an 8-byte boundary):

☐ Specify local address for incoming signal buffer
Local address (32-bit hex aligned to an 8-byte boundary):

☐ Show output port for status

Buttons: OK, Cancel, Help, Apply

Remote device ID (16-bit hex)

Enter the ID of the remote device in 16-bit hexadecimal format to configure the remote ID field in the generated code. When you configure SRIO Transmit blocks for this communication link, the remote device ID in this field must match the local device ID for the SRIO Config block on the receiving end of the transmission.

Send doorbell interrupt to remote device

Enables the doorbell interrupt operation for the bloc, which sends a doorbell interrupt after writing data to the remote device. Selecting this option enables **Doorbell interrupt ID**.

Doorbell interrupt ID

Sets the interrupt ID for the doorbell to set the doorbell INFO field of the SRIO packet. Select a value from the list. If your model contains more than one SRIO transmit block, each transmit block must use a different ID. IDs range from 0 to 15 with a default value of 0. SRIO Receive and SRIO Transmit blocks are paired together by this ID. Create an SRIO Receive block with this ID to receive the doorbell interrupt. The block uses this value to set the doorbell INFO field in an SRIO packet.

Write to remote device

Selecting this option tells the block to perform a burst write using Direct IO to the device at the address in **Remote device ID**. If you clear this option, you must select **Send doorbell interrupt to remote device**. Selecting this option enables the **Remote address (32-bit hex aligned to an 8-byte boundary)** option.

Remote address (32-bit hex aligned to an 8-byte boundary)

Enter the address to write the output data to at the remote device.

Clearing **Write to remote device** disables this option. It becomes a do not care field.

To ensure efficient data transfers, enter an address that aligns to an 8-byte boundary in memory.

Specify local address for incoming signal buffer

Select this option to enable you to specify the local address for the input data to this block. Select this option if you are pairing this block with an SRIO Receive block that performs burst-read operation. The SRIO Receive block needs to know the specific address to read the data from. When you select this option, you enable **Local address (32-bit hex aligned to an 8 byte boundary)** where you enter the local address.

Local address (32-bit hex aligned to an 8 byte boundary)

This address specifies the location of the incoming data. For burst write operations, this value is a local address that SRIO uses to form the direct I/O packets.

C6455 DSK SRIO Transmit

To ensure efficient data transfers, enter an address that aligns to an 8-byte boundary in memory.

Show output port for status

When you select this parameter, the output port Stat appears on the block. Stat returns the status of the write transmit operation.

References

For more information about SRIO, refer to *TMS320TCI648x Serial RapidIO User's Guide*, Literature Number: SPRUE13. Texas Instruments Incorporated.

Purpose

Autocorrelate input vector or frame-based matrix

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

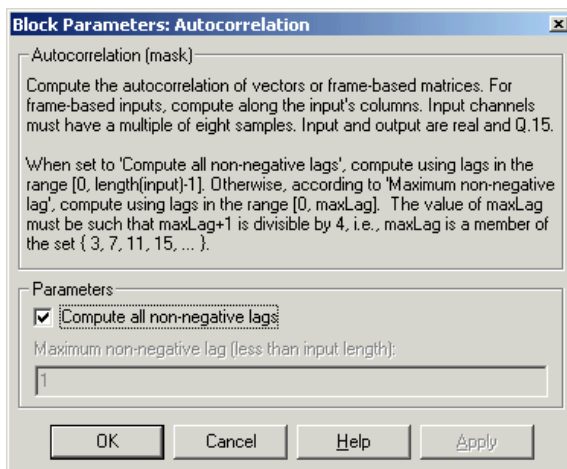
Description



The C64x Autocorrelation block computes the autocorrelation of an input vector or frame-based matrix. For frame-based inputs, the autocorrelation is computed along each of the input's columns. The number of samples in the input channels must be an integer multiple of eight. Input and output signals are real and Q.15.

Autocorrelation blocks support discrete sample times and little-endian code generation only.

Dialog Box



Compute all non-negative lags

When you select this parameter, the autocorrelation is performed using all nonnegative lags, where the number of lags is one less than the length of the input. The lags produced are therefore in the range $[0, \text{length}(\text{input})-1]$. When this parameter is not selected, you specify the lags used in **Maximum non-negative lag (less than input length)**.


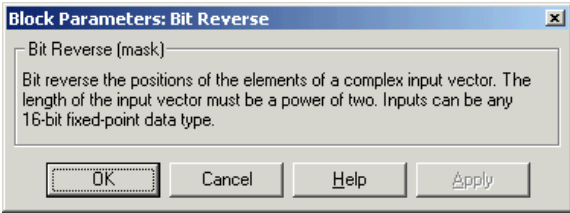
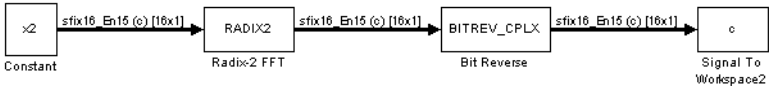
C64x Autocorrelation

Maximum non-negative lag (less than input length)

Specify the maximum lag (maxLag) the block should use in performing the autocorrelation. The lags used are in the range [0, maxLag]. The maximum lag must be odd, and (maxLag+1) must be divisible by 4, such as maxLag equal to 3, 7, or 19. This parameter is enabled when you clear the **Compute all non-negative lags** parameter.

Algorithm

In simulation, the Autocorrelation block is equivalent to the TMS320C64x DSP Library assembly code function DSP_autocor. During code generation, this block calls the DSP_autocor routine to produce optimized code.

Purpose	Bit-reverse elements of each complex input signal channel
Library	“C64x DSP Library (tic64dsplib)” on page 6-12, “Transforms” on page 6-14
Description	<div><div></div><div><p>The C64x Bit Reverse block bit-reverses the elements of each channel of a complex input signal X. The Bit Reverse block is used primarily to provide correctly-ordered inputs and outputs to or from blocks that perform FFTs. Inputs to this block must be 16-bit fixed-point data types. Input vector lengths must be a power of two. Because you use this block with FFT blocks the input vector length must be a power of two.</p><p>The Bit Reverse block supports discrete sample times and little-endian code generation only.</p></div></div>
Dialog Box	
Algorithm	In simulation, the Bit Reverse block is equivalent to the TMS320C64x DSP Library assembly code function DSP_bitrev_cplx. During code generation, this block calls the DSP_bitrev_cplx routine to produce optimized code.
Examples	<p>The Bit Reverse block reorders the output of the C64x Radix-2 FFT in the model below to natural order.</p> 

The following code calculates the same FFT in the workspace. The output from this calculation, y2, is displayed side-by-side with the output from the model, c. The outputs match, showing that the Bit Reverse block reorders the Radix-2 FFT output to natural order:

```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);
```


[y2, c]	
0.5000	0.5000
0.4619 - 0.1913i	0.4619 - 0.1913i
0.3536 - 0.3536i	0.3535 - 0.3535i
0.1913 - 0.4619i	0.1913 - 0.4619i
0 - 0.5000i	0 - 0.5000i
-0.1913 - 0.4619i	-0.1913 - 0.4619i
-0.3536 - 0.3536i	-0.3535 - 0.3535i
-0.4619 - 0.1913i	-0.4619 - 0.1913i
-0.5000	-0.5000
-0.4619 + 0.1913i	-0.4619 + 0.1913i
-0.3536 + 0.3536i	-0.3535 + 0.3535i
-0.1913 + 0.4619i	-0.1913 + 0.4619i
0 + 0.5000i	0 + 0.5000i
0.1913 + 0.4619i	0.1913 + 0.4619i
0.3536 + 0.3536i	0.3535 + 0.3535i
0.4619 + 0.1913i	0.4619 + 0.1913i

See Also

C64x Radix-2 FFT, C64x Radix-2 IFFT

Purpose

Minimum number of extra sign bits in each input channel

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

Description

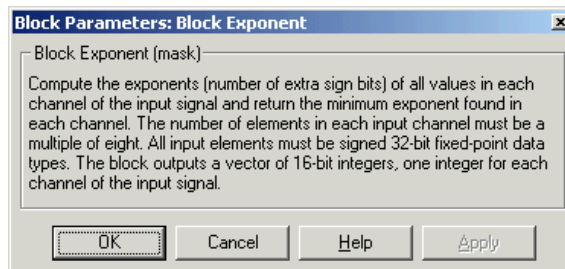


The C64x Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be a multiple of eight. Input elements must be 32-bit signed fixed-point data types. The output is a vector of 16-bit integers — one integer for each channel of the input signal.

This block is useful for determining whether every sample in a channel is using extra sign bits. If so, you can scale your signal by the minimum number of extra sign bits to eliminate the common extra bits. This increases the representable precision and decreases the representable range of the signal.

Block Exponent blocks support both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Block Exponent block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_bexp`. During code generation, this block calls the `DSP_bexp` routine given to produce optimized code.

C64x Complex FIR

Purpose Filter complex input signal using complex FIR filter

Library “C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

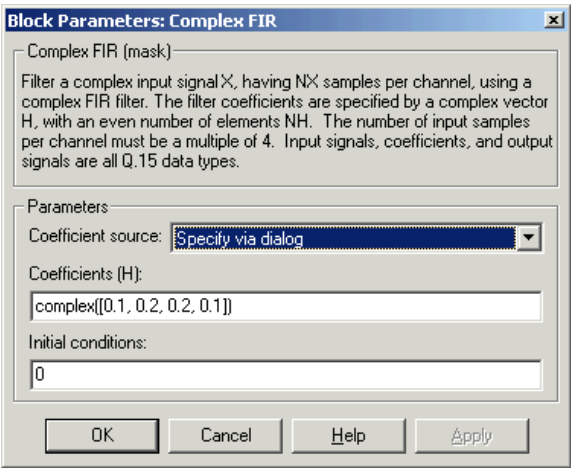
Description The C64x Complex FIR block filters a complex input signal X using a complex FIR filter. This filter is implemented using a direct form structure. Each input channel must contain an integer multiple of four samples, with four samples as the minimum required.



The number of FIR filter coefficients, which are given as elements of the input vector H, must be even. The product of the number of elements of X and the number of elements of H must be at least four. Inputs, coefficients, and outputs are all Q.15 data types. For each channel, the number of input elements must be a multiple of four.

The Complex FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source
Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Coefficients (H)** parameter in the dialog box
- **Input port** — Accept the coefficients from port H. This port must have the same rate as the input data port X. Choosing this option adds an input port to the block.

Coefficients (H)

Designate the filter coefficients in vector format. There must be an even number of coefficients. This parameter is visible only when **Specify via dialog** is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

Lets you provide initial conditions for the filter. If your initial conditions for the channels are

- All the same, enter a scalar that applies to all channels.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. These conditions then apply to all channels. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions for every individual channel. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

You may enter real-valued initial conditions. Zero-valued imaginary parts will be assumed.


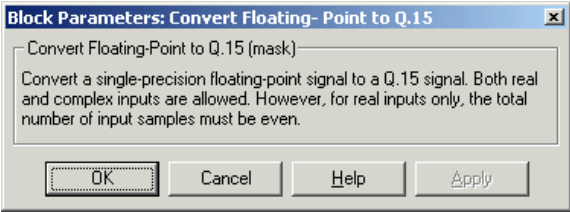
Algorithm

In simulation, the Complex FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_cplx`. During code generation, this block calls the `DSP_fir_cplx` routine to produce optimized code.

See Also

C64x General Real FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

C64x Convert Floating-Point to Q.15

Purpose	Convert floating-point signal to Q.15 fixed-point
Library	“C64x DSP Library (tic64dsplib)” on page 6-12, “Conversions” on page 6-12
Description	<div><div></div><div>The C64x Convert Floating-Point to Q.15 block converts a single-precision floating-point input signal to a Q.15 output signal. Input can be real or complex. For real inputs, the number of input samples must be even.</div><div>The Convert Floating-Point to Q.15 block supports both continuous and discrete sample times. This block supports little-endian code generation only.</div></div>
Dialog Box	<div></div>
Algorithm	In simulation, the Convert Floating-Point to Q.15 block is equivalent to the TMS320C64x DSP Library assembly code function DSP_flttoq15. During code generation, this block calls the DSP_flttoq15 routine to produce optimized code.
See Also	C64x Convert Q.15 to Floating Point

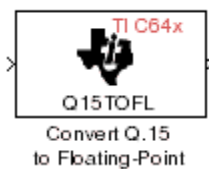
Purpose

Convert Q.15 fixed-point signal to single-precision floating-point

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Conversions” on page 6-12

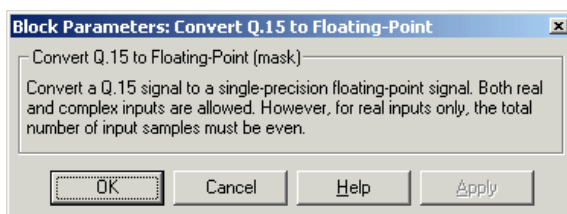
Description



The C64x Convert Q.15 to Floating-Point block converts a Q.15 input signal to a single-precision floating-point output signal. Input can be real or complex. For real inputs, the number of input samples must be even.

The Convert Q.15 to Floating-Point block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Convert Q.15 to Floating-Point block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_q15tof1`. During code generation, this block calls the `DSP_q15tof1` routine to produce optimized code.

See Also

C64x Convert Floating-Point to Q.15

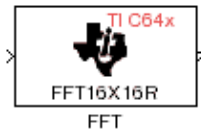
Purpose

Decimation-in-frequency forward FFT of complex input vector

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Transforms” on page 6-14

Description



The C64x FFT block computes the decimation-in-frequency forward FFT, with scaling between stages, of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 8 to 16,384, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in natural order. Inputs and outputs are all signed 16-bit fixed-point data types.

The `fft16x16r` routine used by this block employs butterfly stages to perform the FFT. The number of butterfly stages used, S , depends on the input length $L = 2^k$. If k is even, then $S = k/2$. If k is odd, then $S = (k+1)/2$.

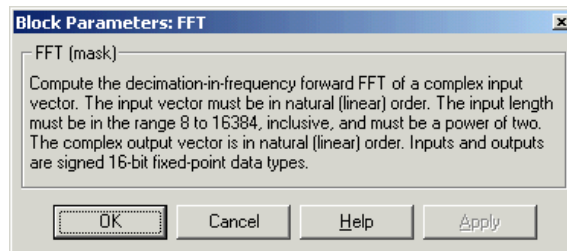
If k is even, then L is a power of two as well as a power of four, and this block performs all S stages with radix-4 butterflies to compute the output. If k is odd, then L is a power of two but not a power of four. In that case this block performs the first $(S-1)$ stages with radix-4 butterflies, followed by a final stage using radix-2 butterflies.

To minimize noise, the FFT block also implements a divide-by-two scaling on the output of each stage except for the last. Therefore, to ensure that the gain of the block matches that of the theoretical FFT, the FFT block offsets the location of the binary point of the output data type by $(S-1)$ bits to the right relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type minus $(S-1)$.

$$\text{OutputFractionalBits} = \text{InputFractionalBits} - (S-1)$$

The FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box



Algorithm

In simulation, the FFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fft16x16r`. During code generation, this block calls the `DSP_fft16x16r` routine to produce optimized code.

See Also

C64x Radix-2 FFT, C64x Radix-2 IFFT

C64x General Real FIR

Purpose Filter real input signal using real FIR filter

Library “C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

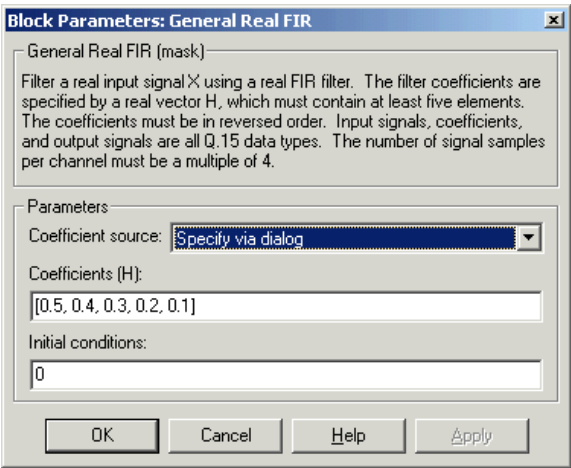
Description The C64x General Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure. Signal X must contain at least four samples per channel and the number of samples must be an integer multiple of four.



The filter coefficients are specified by a real vector H, which must contain at least five elements. The coefficients must be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.

The General Real FIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source
Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Coefficients (H)** parameter in the dialog box

- **Input port** — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

The initial conditions must be real.

Algorithm

In simulation, the General Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_gen`. During code generation, this block calls the `DSP_fir_gen` routine to produce optimized code.

See Also

C64x Complex FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

C64x LMS Adaptive FIR

Purpose

LMS adaptive FIR filtering

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

Description



The C64x LMS Adaptive FIR block performs least-mean-square (LMS) adaptive filtering. This filter is implemented using a direct form structure.

The following constraints apply to the inputs and outputs of this block:

- The scalar input, X must be a Q.15 data type.
- The scalar input B must be a Q.15 data type.
- The scalar output R is a Q1.30 data type.
- The output \bar{H} has length equal to the number of filter taps and is a Q.15 data type. The number of filter taps must be a positive integer that is a multiple of four.

This block performs LMS adaptive filtering according to the equations

$$e(n+1) = d(n+1) - [\bar{H}(n) \cdot \bar{X}(n+1)]$$

and

$$\bar{H}(n+1) = \bar{H}(n) + [\mu e(n+1) \cdot \bar{X}(n+1)],$$

where

- n designates the time step.
- \bar{X} is a vector composed of the current and last $nH-1$ scalar inputs.
- d is the desired signal. The output R converges to d as the filter converges.
- \bar{H} is a vector composed of the current set of filter taps.

- e is the error, or $d - [\bar{H}(n) \cdot \bar{X}(n+1)]$.
- μ is the step size.

For this block, the input B and the output R are defined by

$$B = \mu e(n+1)$$

and

$$R = \bar{H}(n) \cdot \bar{X}(n+1),$$

which combined with the first two equations, result in the following equations that this block follows:

$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)].$$

d and B must be produced externally to the LMS Adaptive FIR block. See “Examples” on page 7-147 below for a sample model where this is done.

The LMS Adaptive FIR block supports discrete sample times and supports little-endian code generation only.

The rounding mode used is *floor*, and the saturation mode is *wrap*. All intermediate products have **s32Q30** data type. The update equation is as follows:

$$H_i = H_i + S16Q15(S32Q30(B) \times S32Q30(X_i))$$

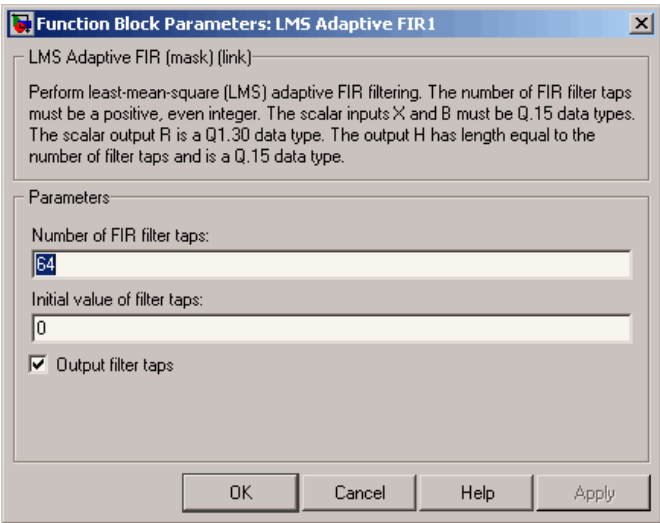
$$R = \sum_N (X_i \times H_i),$$

where N is the number of filter taps.

C64x LMS Adaptive FIR

Note This block does not implement a leaky LMS algorithm, so comparison to the leakage factor of the LMS block of the Signal Processing Blockset software is not appropriate.

Dialog Box



Number of FIR filter taps

Designate the number of filter taps. The number of taps must be a positive integer that is also a multiple of four.

Initial value of filter taps

Enter the initial value of the filter taps.

Output filter coefficients H?

If you select this option, the filter taps are produced as output H. If you do not select this option, H is suppressed.

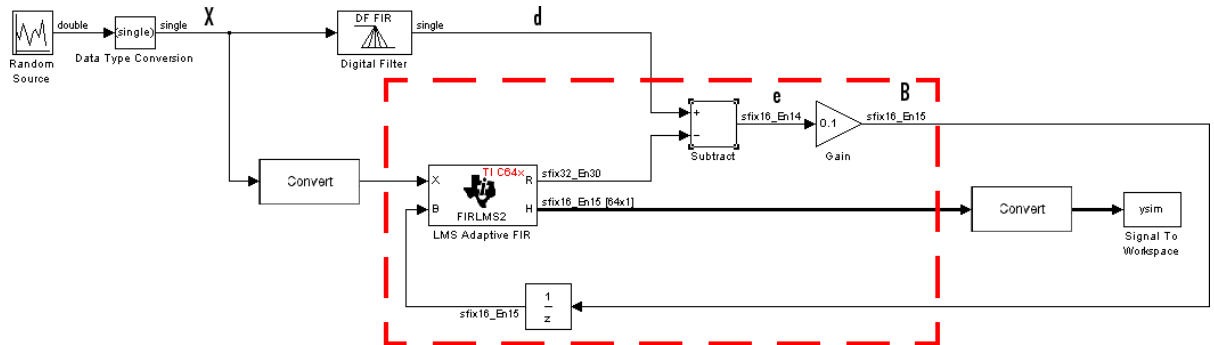
Algorithm

In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir1ms2`.

During code generation, this block calls the DSP_firlms2 routine to produce optimized code.

Examples

The following model uses the LMS Adaptive FIR block.



The portion of the model enclosed by the dashed line produces the signal B and feeds it back into the LMS Adaptive FIR block. The inputs to this region are \bar{X} and the desired signal d , and the output of this region is the vector of filter taps \bar{H} . Thus this region of the model acts as a canonical LMS adaptive filter. For example, compare this region to the `adaptfilt.lms` function in Filter Design Toolbox software. `adaptfilt.lms` performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

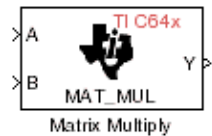
To use the LMS Adaptive FIR block you must create the input B in some way similar to the one shown here. You must also provide the signals \bar{X} and d . This model simulates the desired signal d by feeding \bar{X} into a digital filter block. You can simulate your desired signal in a similar way, or you may bring d in from the workspace with a From Workspace or codec block.

C64x Matrix Multiply

Purpose Matrix multiply two input signals

Library “C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

Description The C64x Matrix Multiply block multiplies two input matrices A and B. Inputs and outputs are real, 16-bit, signed fixed-point data types. This block wraps overflows when they occur.



The product of the two 16-bit inputs results in a 32-bit accumulator value. The Matrix Multiply block, however, only outputs 16 bits. You can choose to output the highest or second-highest 16 bits of the accumulator value.

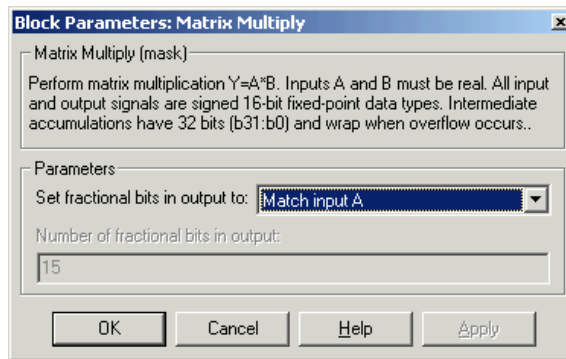
Alternatively, you can choose to output 16 bits according to how many fractional bits you want in the output. The number of fractional bits in the accumulator value is the sum of the fractional bits of the two inputs.

	Input A	Input B	Accumulator Value
Total Bits	16	16	32
Fractional Bits	R	S	$R + S$

Therefore $R+S$ is the location of the binary point in the accumulator value. You can select 16 bits in relation to this fixed position of the accumulator binary point to give the desired number of fractional bits in the output (see “Examples” on page 7-150 below). You can either require the output to have the same number of fractional bits as one of the two inputs, or you can specify the number of output fractional bits in the **Number of fractional bits in output** parameter.

The Matrix Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Choose which 16 bits to output from the list:

- **Match input A** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or *R* in the discussion above).
- **Match input B** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or *S* in the discussion above).
- **Match high bits of acc. (b31:b16)** — Output the highest 16 bits of the accumulator value.
- **Match high bits of prod. (b30:b15)** — Output the second-highest 16 bits of the accumulator value.
- **User-defined** — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

C64x Matrix Multiply

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is enabled only when you select User-defined for **Set fractional bits in output to**.

Algorithm

In simulation, the Matrix Multiply block is equivalent to the TMS320C64x DSP Library assembly code function DSP_mat_mul. During code generation, this block calls the DSP_mat_mul routine to produce optimized code.

Examples

Example 1

Suppose A and B are both Q.15. The data type of the resulting accumulator value is therefore the 32-bit data type Q1.30 ($R + S = 30$). In the accumulator, bits 31:30 are the sign and integer bits, and bits 29:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q.15	b30:b15
Match input B	Q.15	b30:b15
Match high bits of acc.	Q1.14	b31:b16
Match high bits of prod.	Q.15	b30:b15

Example 2

Suppose A is Q12.3 and B is Q10.5. The data type of the resulting accumulator value is therefore Q23.8 ($R + S = 8$). In the accumulator, bits 31:8 are the sign and integer bits, and bits 7:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q12.3	b20:b5
Match input B	Q10.5	b18:b3
Match high bits of acc.	Q23.-8	b31:b16
Match high bits of prod.	Q22.-7	b30:b15

See Also C64x Vector Multiply

C64x Matrix Transpose

Purpose

Matrix transpose input signal

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

Description

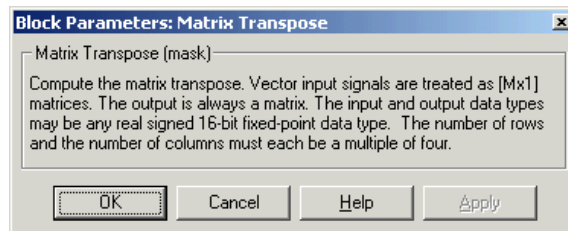


The C64x Matrix Transpose block transposes an input matrix or vector. A 1-D input is treated as a column vector and transposed to a row vector. Input and output signals are any real, 16-bit, signed fixed-point data type. Both the number of rows and the number of columns must be multiples of four.

The Matrix Transpose block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Note If you use Target Function Library (TFL) technology with this block, the TI compiler generates processor and compiler-specific instructions that improve the performance of the generated code. For more information, consult “Introduction to Target Function Libraries”.

Dialog Box



Algorithm

In simulation, the Matrix Transpose block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_mat_trans`. During code generation, this block calls the `DSP_mat_trans` routine to produce optimized code.

Purpose

Radix-2 decimation-in-frequency forward FFT of complex input vector

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Transforms” on page 6-14

Description

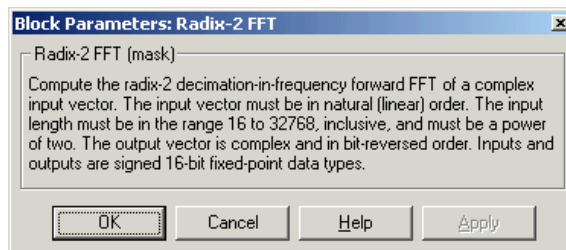


The C64x Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types, and the output data type matches the input data type.

You can use the C64x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

The Radix-2 FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box



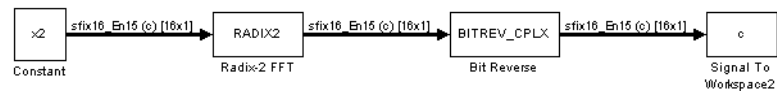
Algorithm

In simulation, the Radix-2 FFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

Examples

The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the C64x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

C64x Radix-2 FFT



The following code calculates the same FFT as the above model in the workspace. The output from this calculation, y2, is then displayed side-by-side with the output from the model, c. The outputs match, showing that the Bit Reverse block does reorder the Radix-2 FFT block output to natural order:

```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);

[y2, c]
```

0.5000	0.5000
0.4619 - 0.1913i	0.4619 - 0.1913i
0.3536 - 0.3536i	0.3535 - 0.3535i
0.1913 - 0.4619i	0.1913 - 0.4619i
0 - 0.5000i	0 - 0.5000i
-0.1913 - 0.4619i	-0.1913 - 0.4619i
-0.3536 - 0.3536i	-0.3535 - 0.3535i
-0.4619 - 0.1913i	-0.4619 - 0.1913i
-0.5000	-0.5000
-0.4619 + 0.1913i	-0.4619 + 0.1913i
-0.3536 + 0.3536i	-0.3535 + 0.3535i
-0.1913 + 0.4619i	-0.1913 + 0.4619i
0 + 0.5000i	0 + 0.5000i
0.1913 + 0.4619i	0.1913 + 0.4619i
0.3536 + 0.3536i	0.3535 + 0.3535i
0.4619 + 0.1913i	0.4619 + 0.1913i

See Also

C64x Bit Reverse, C64x FFT, C64x Radix-2 IFFT

Purpose

Radix-2 inverse FFT of complex input vector

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Transforms” on page 6-14

Description



The C64x Radix-2 IFFT block computes the radix-2 inverse FFT of each channel of a complex input signal. This block uses a decimation-in-frequency forward FFT algorithm with butterfly weights modified to compute an inverse FFT. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types.

The `radix2` routine used by this block employs a radix-2 FFT of length $L=2^k$. To ensure that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by k bits to the left relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type plus k .

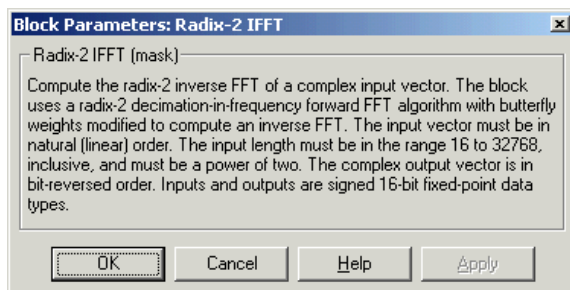
$$\text{OutputFractionalBits} = \text{InputFractionalBits} + (k)$$

You can use the C64x Bit Reverse block to reorder the output of the Radix-2 IFFT block to natural order.

The Radix-2 IFFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

C64x Radix-2 IFFT

Dialog Box



Algorithm

In simulation, the Radix-2 IFFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

See Also

C64x Bit Reverse, C64x FFT, C64x Radix-2 FFT

Purpose

Filter real input signal using real FIR filter

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

Description

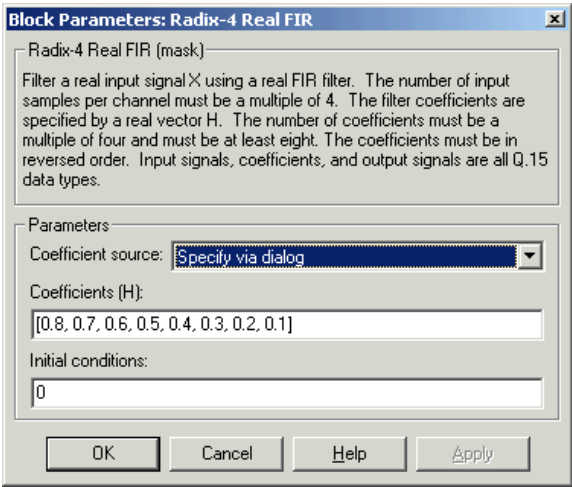


The C64x Radix-4 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be a multiple of four. The filter coefficients are specified by a real vector, H . The number of filter coefficients must be a multiple of four and must be at least eight. The coefficients must also be in reversed order $\{b(n), b(n-1), \dots, b(0)\}$. All inputs, coefficients, and outputs are Q_{15} signals.

The Radix-4 Real FIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Coefficients** parameter in the dialog box

C64x Radix-4 Real FIR

- **Input port** — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient source** parameter. Enter the n coefficients in reversed order — $b(n), b(n-1), \dots, b(0)$. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-4 Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_r4`. During code generation, this block calls the `DSP_fir_r4` routine to produce optimized code.

See Also

C64x Complex FIR, C64x General Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

Purpose

Filter real input signal using real FIR filter

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

Description

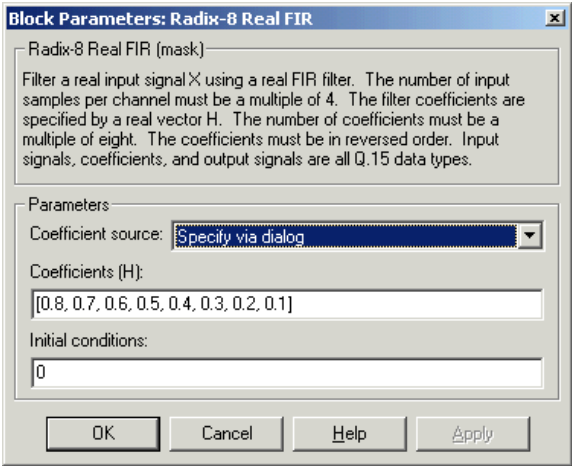


The C64x Radix-8 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be a multiple of four. The filter coefficients are specified by a real vector, H . The number of coefficients must be an integer multiple of eight. The coefficients must be in reversed order — $\{b(n), b(n-1), \dots, b(0)\}$. All inputs, coefficients, and outputs are Q.15 signals.

The Radix-8 Real FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- Specify via dialog — Enter the coefficients in the **Coefficients** parameter in the dialog box

C64x Radix-8 Real FIR

- **Input port** — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format, entering them in reversed order — $b(n), b(n-1), \dots, b(0)$. This parameter is visible when **Specify via dialog** is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-8 Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_r8`. During code generation, this block calls the `DSP_fir_r8` routine to produce optimized code.

See Also

C64x Complex FIR, C64x General Real FIR, C64x Radix-4 Real FIR, C64x Symmetric Real FIR

Purpose

Filter real input signal using lattice IIR filter

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

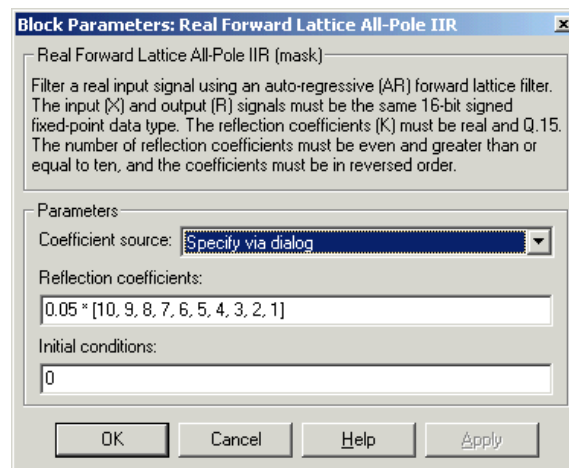
Description



The C64x Real Forward Lattice All-Pole IIR block filters a real input signal using an autoregressive forward lattice filter. The input and output signals must be the same 16-bit signed fixed-point data type. The reflection coefficients must be real and Q.15. The number of reflection coefficients must be greater than or equal to ten; they must be even; and they must be in reversed order — $k(n)$, $k(n-1)$, ..., $k(0)$. Using an even number of reflection coefficients maximizes the speed of your generated code.

The Real Forward Lattice All-Pole IIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **Reflection coefficients** parameter in the dialog box

C64x Real Forward Lattice All-Pole IIR

- **Input port** — Accept the coefficients from port K

Reflection coefficients

Designate the reflection coefficients of the filter in vector format. The number of coefficients must be greater than or equal to ten and be even. Enter the coefficients in reverse order from $k(n)$ to $k(0)$. Using an even number of reflection coefficients maximizes the speed of your generated code. This parameter is visible when you select **Specify via dialog** for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If your block initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length (number of elements) of this vector must be the same as the number of reflection coefficients in your filter.
- Different across channels, enter a matrix containing all initial conditions. The number of rows (initial conditions for one channel) of this matrix must be the same as the number of reflection coefficients, and the number of columns of this matrix must be equal to the number of channels.

Algorithm

In simulation, the Real Forward Lattice All-Pole IIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_iirlat`. During code generation, this block calls the `DSP_iirlat` routine to produce optimized code.

See Also

C64x Real IIR

Purpose

Filter real input signal using IIR filter

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

Description

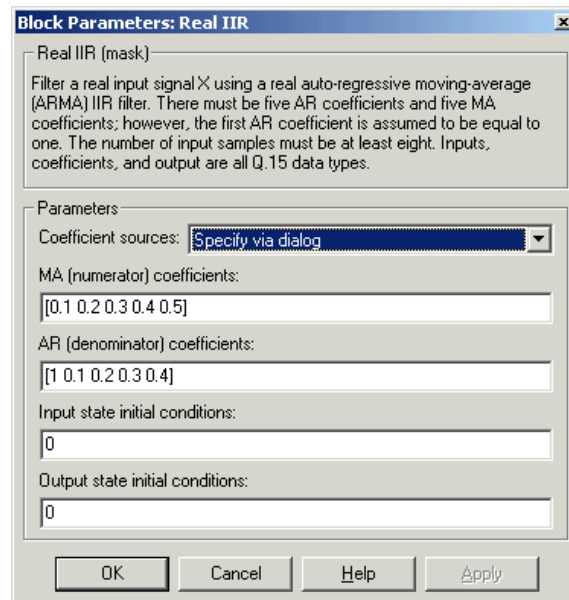


The C64x Real IIR block filters a real input signal X using a real autoregressive moving-average (ARMA) IIR Filter. This filter is implemented using a direct form I structure. You must use at least eight input samples.

There must be five AR coefficients and five MA coefficients. The first AR coefficient is always assumed to be one. Inputs, coefficients, and output are Q.15 data types.

The Real IIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient sources

Specify the source of the filter coefficients:

- **Specify via dialog** — Enter the coefficients in the **MA (numerator) coefficients** and **AR (denominator) coefficients** parameters in the dialog box
- **Input ports** — Accept the coefficients from ports MA and AR

MA (numerator) coefficients

Designate the moving-average coefficients of the filter in vector format. There must be five MA coefficients. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

AR (denominator) coefficients

Designate the autoregressive coefficients of the filter in vector format. There must be five AR coefficients, however the first AR coefficient is assumed to be equal to one. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

Input state initial conditions

If the input state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the input state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all input state initial conditions. This matrix must have four rows.

Output state initial conditions

If the output state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the output state initial conditions for one channel. The length of this vector must be four.

- Different across channels, enter a matrix containing all output state initial conditions. This matrix must have four rows.

Algorithm

In simulation, the Real IIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_iir`. During code generation, this block calls the `DSP_iir` routine to produce optimized code.

See Also

C64x Real Forward Lattice All-Pole IIR

C64x Reciprocal

Purpose

Fraction and exponent of reciprocal of real input signal

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

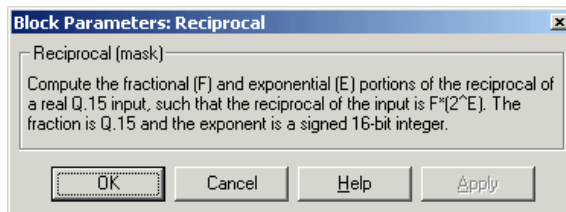
Description



The C64x Reciprocal block computes the fractional (F) and exponential (E) portions of the reciprocal of a real Q.15 input, such that the reciprocal of the input is $F \cdot (2^E)$. The fraction is Q.15 and the exponent is a 16-bit signed integer.

The Reciprocal block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Reciprocal block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_recip16`. During code generation, this block calls the `DSP_recip16` routine to produce optimized code.

Purpose

Filter real input signal using FIR filter

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Filters” on page 6-12

Description



The C64x Symmetric Real FIR block filters a real input signal using a symmetric real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be even. The filter coefficients are specified by a real vector H , which must be symmetric about its middle element. Thus you must use an odd number of coefficients. The number of coefficients must be of the form $16k + 1$, where k is a positive integer. This block wraps overflows that occur. The input, coefficients, and output are 16-bit signed fixed-point data types.

Intermediate multiplies and accumulates performed by this filter result in 32-bit accumulator values. However, the Symmetric Real FIR block only outputs 16 bits. You can choose to output 16 bits of the accumulator value in one of the following ways.

Match input x	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the input
Match coefficients h	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the coefficients
Match high 16 bits of acc.	Output bits 31 - 16 of the accumulator value
Match high 16 bits of prod.	Output bits 30 - 15 of the accumulator value
User-defined	Output 16 bits of the accumulator value such that the output has the number of fractional bits specified in the Number of fractional bits in output parameter

C64x Symmetric Real FIR

The Symmetric Real FIR block supports discrete sample times and only little-endian code generation.

Dialog Box

Block Parameters: Symmetric Real FIR

Symmetric Real FIR (mask)

Filter a real input signal X using a symmetric real FIR filter. The number of input samples per channel must be a multiple of four. The filter coefficients are specified by a real vector H , which must be symmetric about its middle element. The number of elements in H must be of the form $16k+1$ where k is a positive integer. Intermediate accumulations have 32 bits (b31:b0) and use wrap-around arithmetic. All input and output signals are signed 16-bit fixed-point data types.

Parameters

Coefficient source:

Specify via dialog

Coefficients:

0.05 * [1, 2, 3, 4, 5, 6, 7, 8, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Set fractional bits in coefficients to:

Best precision

Number of fractional bits in coefficients:

10

Set fractional bits in output to:

Match high 16 bits of product (b30:b

Number of fractional bits in output:

10

Initial conditions:

0

OK

Cancel

Help

Apply

Coefficient source

Specify the source of the filter coefficients:

- Specify via dialog — Enter the coefficients in the **Coefficients** parameter in the dialog box
- Input port — Accept the coefficients from port H

Coefficients

Enter the coefficients in vector format. Coefficients must be symmetric about the middle element of the vector, so the number

of coefficients must be odd. This parameter is visible when **Specify via dialog** is specified for the **Coefficient source** parameter. This parameter is tunable in simulation.

Set fractional bits in coefficients to

Specify the number of fractional bits in the filter coefficients:

- **Match input X** — Sets the coefficients to have the same number of fractional bits as the input
- **Best precision** — Sets the number of fractional bits of the coefficients such that the coefficients are represented to the best precision possible
- **User-defined** — Sets the number of fractional bits in the coefficients with the **Number of fractional bits in coefficients** parameter

This parameter is visible only when **Specify via dialog** is specified for the **Coefficient source** parameter.

Number of fractional bits in coefficients

Specify the number of bits to the right of the binary point in the filter coefficients. This parameter is visible only when **Specify via dialog** is specified for the **Coefficient source** parameter, and is only enabled if **User-defined** is specified for the **Set fractional bits in coefficients to** parameter.

Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Select which 16 bits to output:

- **Match input X** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input X
- **Match coefficients H** — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in coefficients H

C64x Symmetric Real FIR

- Match high bits of acc. (b31:b16) — Output the highest 16 bits of the accumulator value
- Match high bits of prod. (b30:b15) — Output the second-highest 16 bits of the accumulator value
- User-defined — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter

See Matrix Multiply “Examples” on page 7-150 for demonstrations of these selections.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is only enabled if User-defined is selected for the **Set fractional bits in output to** parameter.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Algorithm

In simulation, the Symmetric Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_sym`. During code generation, this block calls the `DSP_fir_sym` routine to produce optimized code.

See Also

C64x Complex FIR, C64x General Real FIR, C64x Radix-4 Real FIR,
C64x Radix-8 Real FIR

C64x Vector Dot Product

Purpose	Vector dot product of real input signals
Library	“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13
Description	<p>The C64x Vector Dot Product block computes the vector dot product of two real input vectors, X and Y. The input vectors must have the same dimensions and must be signed 16-bit fixed-point data types. The number of samples per channel of the inputs must be a multiple of four. The output is a signed 32-bit fixed-point scalar on each channel, and the number of fractional bits of the output is equal to the sum of the number of fractional bits of the inputs.</p> <p>The Vector Dot Product block supports both continuous and discrete sample times. This block supports little-endian code generation only.</p>



Dialog Box

Block Parameters: Vector Dot Product

Vector Dot Product (mask)
Compute the vector dot product of real inputs X and Y. Inputs must have the same dimensions, and the number of samples per channel must be a multiple of four. Inputs must also be signed 16-bit fixed-point data types. The output is a signed 32-bit fixed-point scalar on each channel.

OK

Cancel

Help

Apply

Algorithm

In simulation, the Vector Dot Product block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_dotprod`. During code generation, this block calls the `DSP_dotprod` routine to produce optimized code.

Purpose

Zero-based index of maximum value element in each input signal channel

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

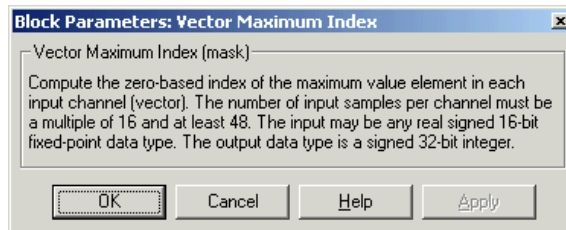
Description



The C64x Vector Maximum Index block computes the zero-based index of the maximum value element in each channel (vector) of the input signal. The input may be any real, 16-bit, signed fixed-point data type. The number of samples per input channel must be an integer multiple of 16 and at least 48. The output data type is 32-bit signed integer.

The Vector Maximum Index block supports both continuous and discrete sample times. This block supports little-endian code generation only.

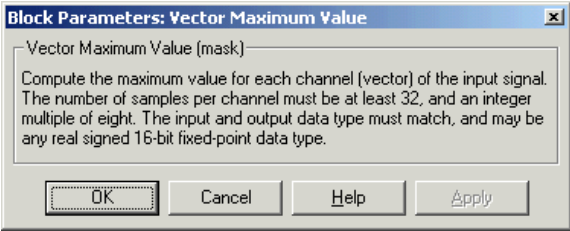
Dialog Box



Algorithm

In simulation, the Vector Maximum Index block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_maxidx`. During code generation, this block calls the `DSP_maxidx` routine to produce optimized code.

C64x Vector Maximum Value

Purpose	Maximum value for each input signal channel
Library	“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13
Description	<p>The C64x Vector Maximum Value block returns the maximum value in each channel (vector) of the input signal. The input can be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of 8 and must be at least 32. The output data type matches the input data type.</p> <p>The Vector Maximum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.</p>
Dialog Box	
Algorithm	In simulation, the Vector Maximum Value block is equivalent to the TMS320C64x DSP Library assembly code function DSP_maxval. During code generation, this block calls the DSP_maxval routine to produce optimized code.
See Also	C64x Vector Minimum Value

Purpose

Minimum value for each input signal channel

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

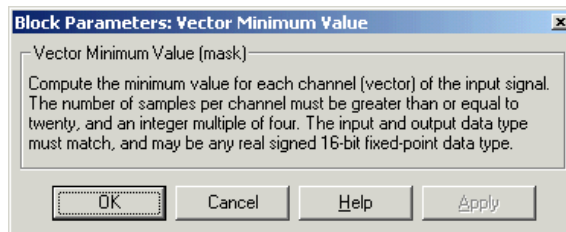
Description



The C64x Vector Minimum Value block returns the minimum value in each channel of the input signal. The input may be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of 4 and must be at least 20. The output data type matches the input data type.

The Vector Minimum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



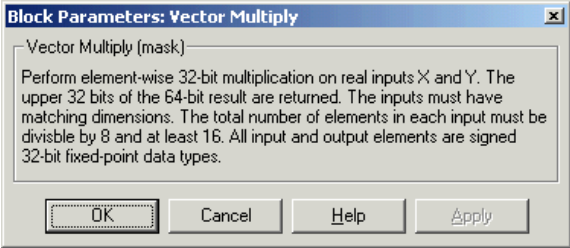
Algorithm

In simulation, the Vector Minimum Value block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_minval`. During code generation, this block calls the `DSP_minval` routine to produce optimized code.

See Also

C64x Vector Maximum Value

C64x Vector Multiply

Purpose	Element-wise multiplication on inputs
Library	“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13
Description	<p>The C64x Vector Multiply block performs element-wise 32-bit multiplication of two inputs X and Y. The total number of elements in each input must be a multiple of 8 and at least 16, and the inputs must have matching dimensions. The upper 32 bits of the 64-bit accumulator result are returned. All input and output elements are 32-bit signed fixed-point data types.</p> <p>The Vector Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.</p>
Dialog Box	
Algorithm	In simulation, the Vector Multiply block is equivalent to the TMS320C64x DSP Library assembly code function DSP_mul32. During code generation, this block calls the DSP_mul32 routine to produce optimized code.
See Also	C64x Matrix Multiply

Purpose

Negate each input signal element

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

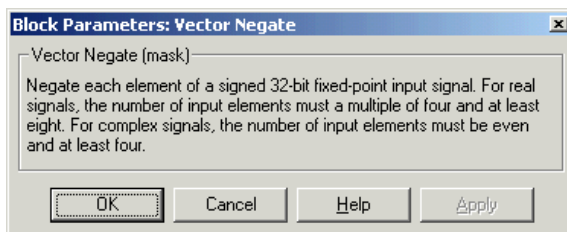
Description



The C64x Vector Negate block negates each element of a 32-bit signed fixed-point input signal. For real signals, the number of input elements must be a multiple of four, and at least eight. For complex signals, the number of input elements must be at least two. The output is the same data type as the input.

The Vector Negate block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Negate block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_neg32`. During code generation, this block calls the `DSP_neg32` routine to produce optimized code.

C64x Vector Sum of Squares

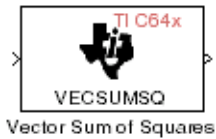
Purpose

Sum of squares over each real input channel

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

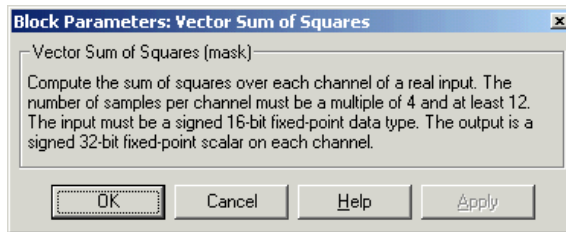
Description



The C64x Vector Sum of Squares block computes the sum of squares over each channel of a real input. The number of samples per input channel must be divisible by 4; equal to or greater than 8; and the input must be a 16-bit signed fixed-point data type. The output is a 32-bit signed fixed-point scalar on each channel. The number of fractional bits of the output is twice the number of fractional bits of the input.

The Vector Sum of Squares block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Sum of Squares block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_vecsumsq`. During code generation, this block calls the `DSP_vecsumsq` routine to produce optimized code.

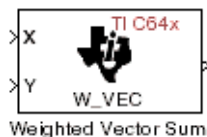
Purpose

Weighted sum of input vectors

Library

“C64x DSP Library (tic64dsplib)” on page 6-12, “Math and Matrices” on page 6-13

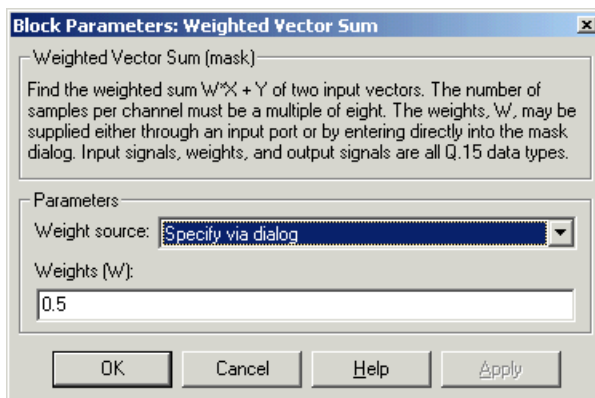
Description



The C64x Weighted Vector Sum block computes the weighted sum of two inputs, X and Y, according to $(W \cdot X) + Y$. Inputs may be vectors or frame-based matrices. The number of samples per channel must be a multiple of eight. Inputs, weights, and output are Q.15 data types, and weights must be in the range $-1 < W < 1$.

The Weighted Vector Sum block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Weight source

Specify the source of the weights:

- **Specify via dialog** — Enter the weights in the **Weights (W)** parameter in the dialog box
- **Input port** — Accept the weights from port W

C64x Weighted Vector Sum

Weights (W)

This parameter is visible only when **Specify via dialog** is specified for the **Weight source** parameter. This parameter is tunable in simulation. When the weights are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be a multiple of four.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be a multiple of four, and the number of columns of this matrix must be equal to the number of channels.

Weights must be in the range $-1 < W < 1$.

Algorithm

In simulation, the Weighted Vector Sum block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

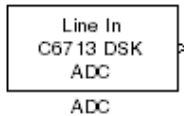
Purpose

Digitized signal output from codec to processor

Library

“C6713 DSK (c6713dsklib)” on page 6-4

Description



Use the C6713 DSK ADC (analog-to-digital converter) block to capture and digitize analog signals from external sources, such as signal generators, frequency generators or audio devices. Placing an C6713 DSK ADC block in your Simulink block diagram lets you use the audio coder-decoder module (codec) on the C6713 DSK to convert an analog input signal to a digital signal for the digital signal processor.

Due to a hardware limitation, there can be only one C6713 DSK ADC block per model. Using two blocks will generate an error message.

Most of the configuration options in the block affect the codec. However, the **Output data type**, **Samples per frame** and **Scaling** options are related to the model you are using in Simulink software, the signal processor on the board, or direct memory access (DMA) on the board. In the following table, you find each option listed with the C6713 DSK hardware affected.

Option	Affected Hardware
ADC source	Codec
Mic	Codec
Output data type	TMS320C6713 digital signal processor
Samples per frame	Direct memory access functions
Scaling	TMS320C6713 digital signal processor
Source gain (dB)	Codec

You can select one of three input sources from the **ADC source** list:

- **Line In** — the codec accepts input from the line in connector (LINE IN) on the board's mounting bracket.
- **Mic** — the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.

Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels on input and output.

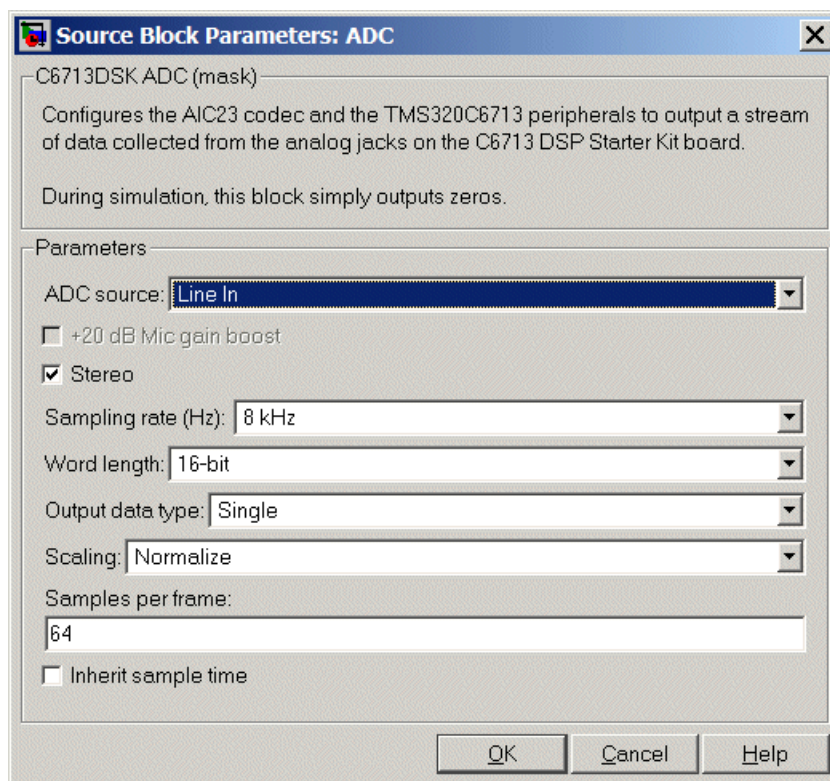
The block uses frame-based processing of inputs, buffering the input data into frames at the specified samples per frame rate. In Simulink software, the block puts monaural data into an N-element column vector. Stereo data input forms an N-by-2 matrix with N data values and two stereo channels (left and right).

When the samples per frame setting is more than one, each frame of data is either the N-element vector (monaural input) or N-by-2 matrix (stereo input). For monaural input, the elements in each frame form the column vector of input audio data. In the stereo format, the frame is the matrix of audio data represented by the matrix rows and columns — the rows are the audio data samples and the columns are the left and right audio channels.

When you select **Mic** for **ADC source**, you can select the **+20 dB Mic gain boost** check box to add 20 dB to the microphone input signal before the codec digitizes the signal.

Source gain (dB) lets you add gain to the input signal before the A/D conversion. Select the appropriate gain from the list.

Dialog Box



ADC source

The input source to the codec. **Line In** is the default setting. Selecting **Mic** enables the **+20 dB Mic gain boost** option.

+20 dB Mic gain boost

Boosts the input signal by +20dB when **ADC source** is **Mic**. Gain is applied before analog-to-digital conversion.

Stereo

Indicates whether the input audio data is in monaural or stereo format. Select the check box to enable stereo input. Clear the

check box when you input monaural data. By default, stereo operation is enabled.

Sampling Rate

Set the sampling rate of the analog-to-digital converter. Increasing the frequency increases the accuracy of the sampling data over time.

Word length

Sets the resolution with which the ADC samples the analog input. Increasing the word length increases the accuracy of the data in each sample. If your model also contains a DAC block, set its word length match that of the ADC block.

Output data type

Selects the word length and shape of the data from the codec. By default, double is selected. Options are Double, Single, and Integer.

Scaling

Selects whether the codec data is unmodified, or normalized to the output range to ± 1.0 , based on the codec data format. Select either Normalize or Integer Value. Normalize is the default setting.

Samples per frame

Creates frame-based outputs from sample-based inputs. This parameter specifies the number of samples of the signal the block buffers internally before it sends the digitized signals, as a frame vector, to the next block in the model. This value defaults to 64 samples per frame. Notice that the frame rate depends on the sample rate and frame size. For example, if your input is 8kHz samples per second, and you select 64 samples per frame, the frame rate is 125 frames every second. The throughput remains the same at 64 samples per second.

Inherit sample time

Select whether the block inherits the sample time from the model base rate or from the Simulink base rate. You can locate the Simulink base rate in the Solver options in Configuration Parameters. Selecting Inherit sample time directs the block to use

the specified rate in model configuration. Entering -1 configures the block to accept the sample rate from the upstream Interrupt, Task, or Triggered Task blocks.

See Also

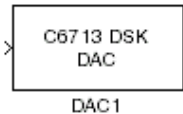
C6713 DSK DAC

C6713 DSK DAC

Purpose Configure codec to convert digital input to analog output

Library “C6713 DSK (c6713dsklib)” on page 6-4

Description Adding the C6713 DSK DAC (digital-to-analog converter) block to your Simulink model lets you connect an analog signal to the analog output jack on the C6713 DSK. When you add the C6713 DSK DAC block, the digital signal received by the codec is converted to an analog signal and sent to the output jack.

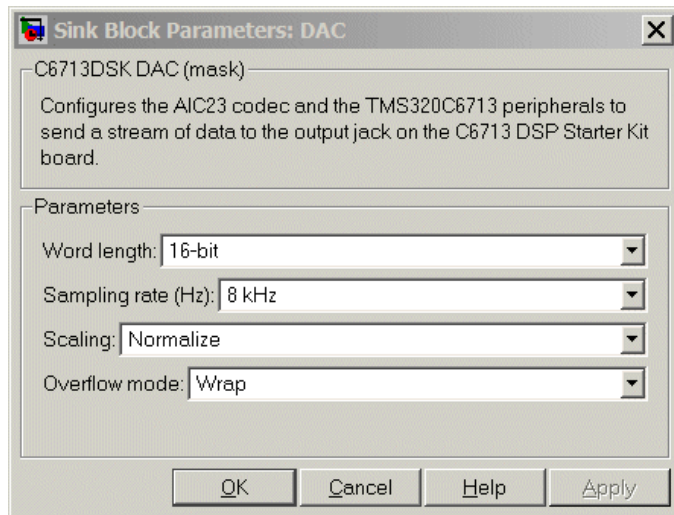


The input on the C6713DSK DAC block takes [Nx1] and [Nx2] signals. The AIC23 audio codec on the C6713DSK board always outputs stereo samples, even though it accepts both mono [Nx1] and stereo [Nx2] signals. If the input is a mono signal with dimension [Nx1], the block outputs the same signal on both the left and right channels. If the input is a stereo signal with dimension [Nx2], each of the N samples are output separately through the left and right channels.

Only the **Word length** option in the block affects the codec. The other options relate to the model you are using in Simulink software and the signal processor on the board. Refer to the following table for information.

Option	Affected Hardware
Overflow mode	TMS320C6713 Digital Signal Processor
Scaling	TMS320C6713 Digital Signal Processor
Word length	Codec

Dialog Box



Word length

Sets the DAC to interpret the input data word length. Without this setting, the DAC cannot convert the digital data to analog correctly. The value defaults to 16 bits, with options of 20, 24, and 32 bits. Select the word length to match the ADC setting.

Scaling

Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range ± 1.0 . Matching the setting for the C6713 DSK ADC block is appropriate here.

Overflow mode

Determines how the codec responds to data that is outside the range specified by the **Scaling** parameter. You can choose **Wrap** or **Saturate** options to apply to the result of an overflow in an operation. **Saturation** is the less efficient operating mode if efficiency is important to your development.

See Also

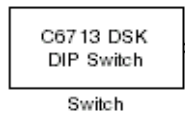
C6713 DSK ADC

C6713 DSK DIP Switch

Purpose Simulate or read DIP switches

Library “C6713 DSK (c6713dsklib)” on page 6-4

Description Added to your model, this block behaves differently in simulation than in code generation and targeting.



In Simulation — the options **Switch 0**, **Switch 1**, **Switch 2**, and **Switch 3** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6713 DSK. Each option turns the associated DIP switch on when you select it. The switches are independent of one another.

By defining the switches to represent actions on your target, DIP switches let you modify the operation of your process by reconfiguring the switch settings.

Use the **Data type** to specify whether the DIP switch options output an integer or a logical string of bits to represent the status of the switches. The table that follows presents all the option setting combinations with the result of your **Data type** selection.

Option Settings to Simulate the User DIP Switches on the C6713 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Cleared	Cleared	Cleared	0000	0
Selected	Cleared	Cleared	Cleared	0001	1
Cleared	Selected	Cleared	Cleared	0010	2
Selected	Selected	Cleared	Cleared	0011	3
Cleared	Cleared	Selected	Cleared	0100	4
Selected	Cleared	Selected	Cleared	0101	5
Cleared	Selected	Selected	Cleared	0110	6

Option Settings to Simulate the User DIP Switches on the C6713 DSK (Continued)

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Selected	Selected	Selected	Cleared	0111	7
Cleared	Cleared	Cleared	Selected	1000	8
Selected	Cleared	Cleared	Selected	1001	9
Cleared	Selected	Cleared	Selected	1010	10
Selected	Selected	Cleared	Selected	1011	11
Cleared	Cleared	Selected	Selected	1100	12
Selected	Cleared	Selected	Selected	1101	13
Cleared	Selected	Selected	Selected	1110	14
Selected	Selected	Selected	Selected	1111	15

Selecting the `Integer` data type results in the switch settings generating integers in the range from 0 to 15 (`uint8`), corresponding to converting the string of individual switch settings to a decimal value. In the `Boolean` data type, the output string presents the separate switch setting for each switch, with the **Switch 0** status represented by the least significant bit (LSB) and the status of **Switch 3** represented by the most significant bit (MSB).

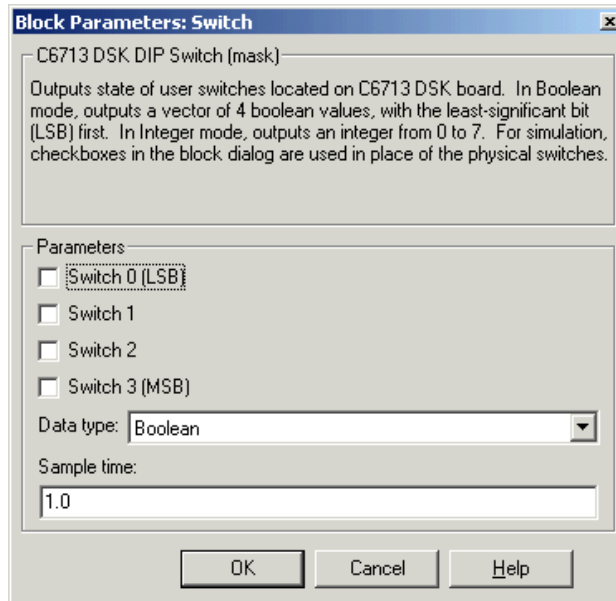
In Code generation and targeting — the code generated by the block reads the physical switch settings of the user switches on the board and reports them as shown above. Your process uses the result in the same way whether in simulation or in code generation. In code generation and when running your application, the block code ignores the settings for **Switch 0**, **Switch 1**, **Switch 2** and **Switch 3** in favor of reading the hardware switch settings. When the block reads the DIP switches, it reports the results as either a Boolean string or an integer value as the table below shows.

C6713 DSK DIP Switch

Output Values From The User DIP Switches on the C6713 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2
On	On	Off	Off	0011	3
Off	Off	On	Off	0100	4
On	Off	On	Off	0101	5
Off	On	On	Off	0110	6
On	On	On	Off	0111	7
Off	Off	Off	On	1000	8
On	Off	Off	On	1001	9
Off	On	Off	On	1010	10
On	On	Off	On	1011	11
Off	Off	On	On	1100	12
On	Off	On	On	1101	13
Off	On	On	On	1110	14
On	On	On	On	1111	15

Dialog Box



Opening this dialog box causes a running simulation to pause. Refer to “Changing Source Block Parameters During Simulation” in your online Simulink documentation for details.

Switch 0

Simulate the status of the user-defined DIP switch on the board.

Switch 1

Simulate the status of the user-defined DIP switch on the board.

Switch 2

Simulate the status of the user-defined DIP switch on the board.

Switch 3

Simulate the status of the user-defined DIP switch on the board.

C6713 DSK DIP Switch

Data type

Determines how the block reports the status of the user-defined DIP switches. **Boolean** is the default, indicating that the output is a vector of four logical values, either 0 or 1.

Each vector element represents the status of one DIP switch; the first switch is switch **Switch 0** and the fourth is switch **Switch 3**. The data type **Integer** converts the logical string to an equivalent unsigned 8-bit (**uint8**) value. For example, when the logical string generated by the switches is 0101, the conversion yields 5 — the LSB is 1 and the MSB is 0.

Sample time

Specifies the time between samples of the signal. This value defaults to 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

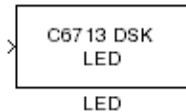
Purpose

Control LEDs

Library

“C6713 DSK (c6713dsklib)” on page 6-4

Description



Adding the C6713 DSK LED block to your Simulink block diagram lets you trigger all four of the user light emitting diodes (LED) on the C6713 DSK. To use the block, send a nonzero real scalar to the block. The C6713 DSK LED block controls all four User LEDs located on the C6713 DSK.

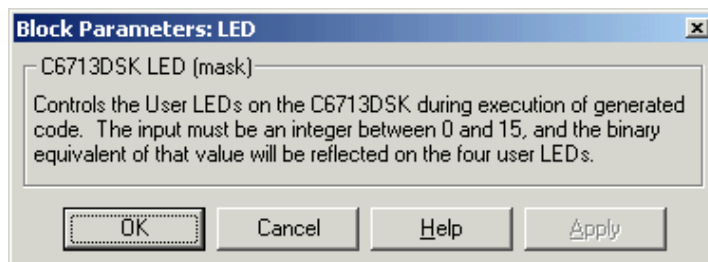
When you add this block to a model, and send a real scalar to the block input, the block sets the LED state based on the input value it receives:

- When the block receives an input value equal to 0, the specified LEDs are turned off (disabled), 0000
- When the block receives a nonzero input value, the specified LEDs are turned on (enabled), 0001 to 1111

To activate the block, send it an integer in the range 0 to 15. Vectors do not work to activate LEDs; nor do complex numbers as scalars or vectors.

All LEDs maintain their state until they receive an input value that changes the state. Enabled LEDs stay on until the block receives an input value that turns the LEDs off; disabled LEDs stays off until turned on. Resetting the C6713 DSK turns off all User LEDs. By default, the LEDs are turned off when you start an application.

Dialog Box



C6713 DSK LED

This dialog box does not have any user-selectable options.

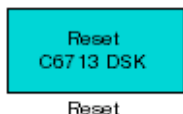
Purpose

Reset to initial conditions

Library

“C6713 DSK (c6713dsklib)” on page 6-4

Description



Double-clicking this block in a Simulink model window resets the C6713 DSK that is running the executable code built from the model. When you double-click the Reset block, the block runs the software reset function provided by CCS IDE that resets the processor on your C6713 DSK. Applications running on the board stop and the signal processor returns to the initial conditions you defined.

Before you build and download your model, add the block to the model as a stand-alone block. You do not need to connect the block to any block in the model. When you double-click this block in the block library it resets your C6713 DSK. In other words, anytime you double-click a C6713 DSK Reset block you reset your C6713 DSK.

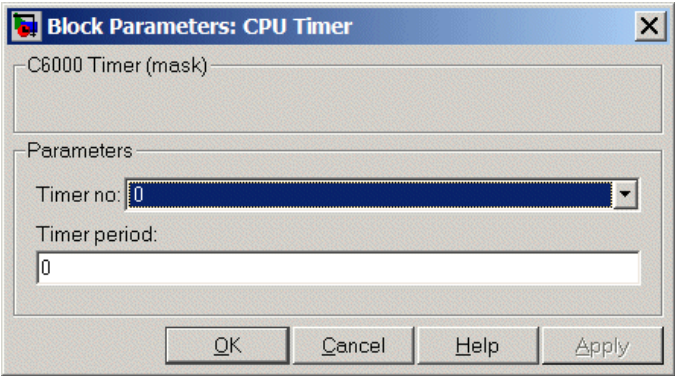
Dialog Box

This block does not have settable options and does not provide a user interface dialog box.

C6000 CPU Timer

Purpose	Select timer and configure periodic interrupt
Library	“Scheduling (c6000dspcorelib)” on page 6-14
Description	<p>Configures the CPU timer period on your board. The timer raises periodic interrupts when the timer counter reaches the timer period. While the block provides two timers, 0 and 1, some CPU’s have more or fewer than two timers. For example, the DM642 provides three timers. If you set Timer no to 1, verify that your CPU has two or more timers.</p> <p>The C6000 CPU Timer block does not support C64x processors.</p>

Dialog Box



- Timer no.**
Select the timer to use from the list. Verify that the target offers a timer with the timer number you choose. Timer 0 is selected by default.
- Timer period**
Set the timer interrupt period in terms of CPU clock cycles.
- Enter the timer period in clock cycles, either as an integer, fraction, decimal, or a variable in your workspace. 0 is the default value.

For example, to generate a periodic timer interrupt every second when the CPU clock operates at 720MHz, set **Timer period** to 720e6 clock cycles.

See Also

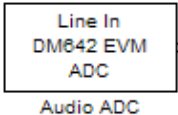
C5000/C6000 Hardware Interrupt, Idle Task

DM642 EVM Audio ADC

Purpose Audio codec and peripherals

Library “DM642 EVM (dm642evmlib)” on page 6-7

Description Use the DM642 EVM ADC (analog-to-digital converter) block to capture and digitize analog audio signals from external sources, such as signal generators, frequency generators, or audio devices. Placing a DM642 EVM ADC block in your Simulink block diagram lets you use the audio coder-decoder module (codec) on the DM642 EVM to convert an analog input signal to a digital signal for the digital signal processor.



ADC blocks output int16 data independent of the data type you provide as input to the block.

Most of the configuration options in the block affect the codec. However, the **Samples per frame** and **Scaling** options are related to the model you are using in Simulink software, the signal processor on the board, or direct memory access (DMA) on the board. In the following table, you find each option listed with the DM642 EVM hardware affected.

Option	Affected Hardware
ADC Source	Codec
Mic	Codec
Sample rate (Hz)	Codec
Samples per frame	Direct memory access functions
Stereo	Codec

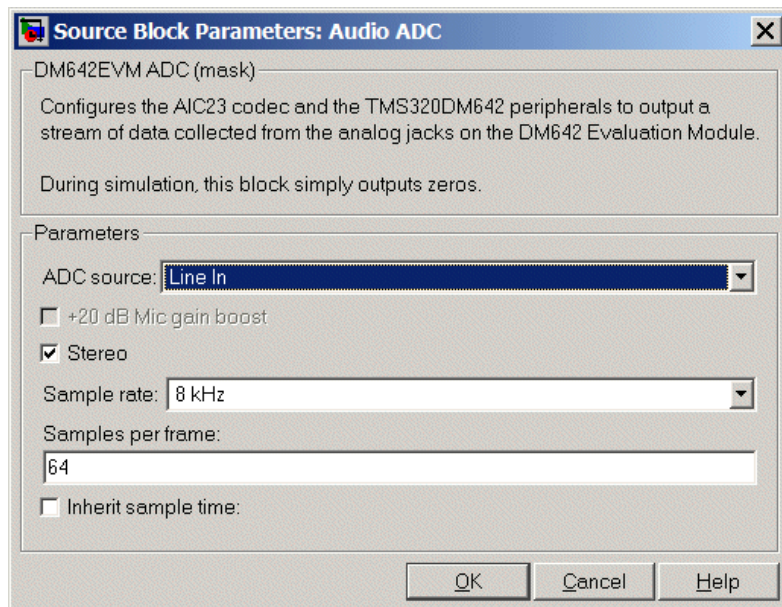
You can select one of two input sources from the **ADC source** list:

- **Line In** — the codec accepts input from the line in connector (LINE IN) on the board’s mounting bracket.
- **Mic in** — the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.

Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels.

You must set the sample rate for the block. From **Sample rate (Hz)**, select the sample rate for your model. **Sample rate (Hz)** specifies the number of times each second that the codec samples the input signal. Sample rates range from 8 kHz to 96 kHz, in preset rates. You must select from the list; you cannot enter a sample rate that is not on the list.

Dialog Box



ADC source

The input source to the codec. Line In is the default.

+20 dB Mic gain boost

Boosts the input signal by +20dB when **ADC source** is Mic. Gain is applied before analog-to-digital conversion.

Stereo

The number of channels input to the A/D converter. Clearing this option selects the left channel; selecting this option selects both left and right input channels. To configure the DM642 EVM board for monaural operation, clear the **Stereo** check box. When you first open the dialog box, **Stereo** is selected. This value defaults to stereo operation.

Sample rate (Hz)

Sampling rate of the A/D converter. Available sample rates are set by the codec. Default rate is 8 kHz. Options range up to 96 kHz. Select the sample rate from the list.

Samples per frame

Creates frame-based outputs from sample-based inputs. This parameter specifies the number of samples of the signal buffered internally by the block before it sends the digitized signals, as a frame vector, to the next block in the model. This value defaults to 64 samples per frame. Notice that the frame rate depends on the sample rate and frame size. For example, if your input is 32 samples per second, and you select 64 samples per frame, the frame rate is one frame every two seconds. The throughput remains the same at 32 samples per second.

Inherit sample time

Selects whether the block inherits the sample time from the model base rate or Simulink base rate as determined in the Solver options in Configuration Parameters. Selecting **Inherit sample time** directs the block to use the specified rate in model configuration. You must select this option to use the block in a function subsystem with the asynchronous scheduler.

See Also

DM642 EVM Audio DAC

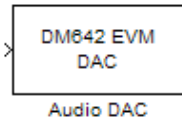
Purpose

Configure codec to convert digital audio input to analog audio output

Library

“DM642 EVM (dm642evmlib)” on page 6-7

Description



Adding the DM642 EVM DAC (digital-to-analog converter) block to your Simulink model lets you output an analog signal to the LINE OUT connection on the DM642 EVM mounting bracket. When you add the DM642 EVM DAC block, the digital signal received by the codec is converted to an analog signal (digital-to-analog conversion) and sent to the output audio jack.

The DAC data word length is 16 bits. The block converts all input data to `int16` before it writes the data out to the DAC output buffer.

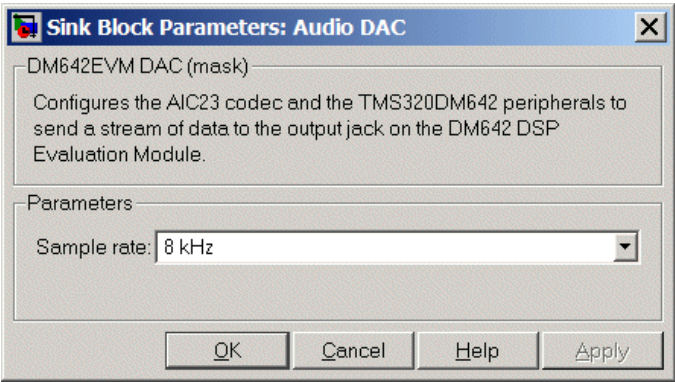
With an integer data word length of 16 bits, any data value above $2^{15}-1$ or below -2^{15} wraps back into the representable range of values between -2^{15} to $2^{15}-1$. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. For more information about wrapping, refer to “Modulo Arithmetic”. Saturate arithmetic is not available. For example,

While converting the digital signal to an analog signal, the codec rounds floating point data to the nearest integer, thus rounding 0.51 up to 1.0 or 4.49 down to 4.0.

Setting the sample rate configures the codec sampling rate for the analog output data stream. The rates range from 8000 Hz, similar to plain old telephone service quality, to 48 kHz (CD quality audio) to 96 kHz.

DM642 EVM Audio DAC

Dialog Box



Sample rate (Hz)

Sampling rate of the D/A converter. Available output sample rates are set by the codec. Default rate is 8000 Hz (8 kHz) and the maximum rate is 96000 Hz (96 kHz). Choose the appropriate rate from the list.

See Also

DM642 EVM Audio ADC

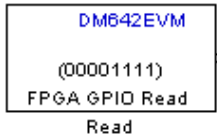
Purpose

User GPIO registers to read from selected pins

Library

“DM642 EVM (dm642evmlib)” on page 6-7

Description



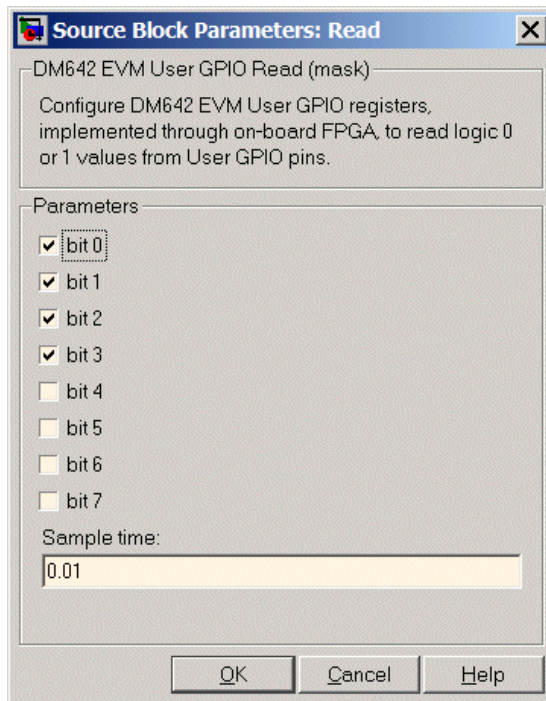
Added to your model, this block reads logical values from the GPIO registers you select in the dialog box and sends the data out to downstream blocks as an unsigned 8-bit word.

The DM642 EVM offers eight general purpose I/O registers that you can read from and write to for your needs. Each I/O pin represents either a logical 0 or 1 depending on the signal at the pin.

An important note — you cannot read and write to the same I/O registers with the FPGA GPIO Read and FPGA GPIO Write blocks. If you read register 1 with the read block you cannot write to register 1 with the write block. This applies to all eight registers.

DM642 EVM FPGA GPIO Read

Dialog Box



bit 0 to bit 7

Each bit represents the logical value at one GPIO register. **Bit 0** is register 0, **bit 7** is register 7. Select the bits that represent the registers to read. The read and write functions cannot share the same registers. If you select a register to read, you cannot write to that register.

Sample time

Time in seconds between consecutive inputs to the registers. Enter any real positive value or a variable name from your workspace.

See Also

DM642 EVM FPGA GPIO Write

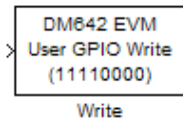
Purpose

Write to GPIO registers

Library

“DM642 EVM (dm642evmlib)” on page 6-7

Description

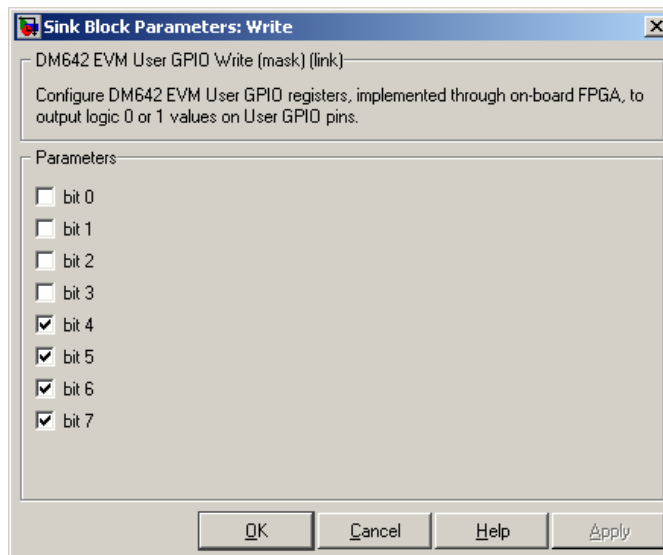


Added to your model, this block writes logical values to the GPIO registers you select in the dialog box, reading the data from an upstream block as an unsigned 8-bit word.

The DM642 EVM offers eight general purpose I/O registers that you can read from and write to for your needs. Each I/O pin represents either a logical 0 or 1 depending on the signal at the pin.

An important note — you cannot read and write to the same I/O registers with the FPGA GPIO Read and FPGA GPIO Write blocks. If you write register 1 with the write block you cannot read from register 1 with the read block. This applies to all eight registers.

Dialog Box



DM642 EVM FPGA GPIO Write

bit 0 to bit 7

Each bit represents the logical value at one GPIO register. **Bit 0** is register 0, **bit 7** is register 7. Select the bits that represent the registers to write. The read and write functions cannot share the same registers. When you select a register to write to, you cannot read that register.

See Also

DM642 EVM FPGA GPIO Read

Purpose Video decoders to capture analog video

Library “DM642 EVM (dm642evmlib)” on page 6-7

Description



Adding this block to a model enables code generated from your model to perform the following tasks:

- 1** Capture analog video data from the video input ports on the DM642 EVM.
- 2** Convert the input to a format and mode you define in the block.
- 3** Output the converted digital video for further downstream processing.

Adding two of these blocks to a model lets you capture two separate video data streams and prepare them for display simultaneously, such as in picture-in-picture mode.

The block captures and buffers one frame (two fields for NTSC standard) of analog input video from the input ports, converts the buffered video to the specified format, and then outputs the converted video frame as 8-bit unsigned integer data for further processing.

Input to the DM642 EVM must be analog National Television Standards Committee (NTSC) or Phase Alternating Line (PAL) video format. The block captures and processes data in frames, not fields.

To configure the format for the output video, the block offers output format options that control how the block handles color data. The block also offers a sample time option to let you set the frame rate for video output from the block.

Note This block does not provide output video for display. Use the DM642 EVM Video DAC to generate video data to output to the board video output connectors. The DM642EVM board provides both composite and S-video connectors for output. However, these are driven simultaneously, so you do not need to specify which one is to be used.

When you add this block to a Simulink model, it has no affect in your simulation — it outputs a string of zeros. Generating code from a model that includes this block produces the code needed for capturing data on your evaluation module by adding

- Video device configuration code for the chosen mode
- Code used to copy the run time buffer

To use video in a Simulink model, use one of the available video source blocks to introduce video data to your model.

Options for the block let you configure the digital video format and video mode for the data output by the block.

NTSC TV systems use interlaced scanning to create TV frames from fields. The even and odd TV lines are separated into even and odd fields that combine to make a complete TV frame image. For output, the block always provides complete frames, consisting of two fields, which are available at any instant. When the sample time you specify for the block is different from the NTSC frame rate of 30Hz, you may encounter visible anomalies in the video stream from the block.

Memory Use

This block allocates video capture buffers on the system heap, using a TI driver that allocates three frame buffers on the heap for continuous video capture. To use the block you must create a heap in external memory on the target with the label EXTERNALHEAP. If you do not create the heap, either using the default values in the DM642 Target

Preferences block or setting your own values. Target Support Package software returns an error.

Use **Create heap** and **Heap size** and set the heap size in the DM642EVM Target Preferences block to configure the heap. Select **Define label** and name the heap EXTERNALHEAP in **Heap label**.

The default settings for the target preferences create a heap with sufficient memory to handle the worst case memory allocation needs automatically. If you configure the heap without sufficient memory, you get a run-time error because the system cannot initialize the video driver.

Notes About Converting NTSC Video Input From YCbCr to RGB24

When you choose to convert your NTSC YCbCr-defined video input to RGB24 (8:8:8 RGB) for output from the block, the block performs an intermediate conversion step that follows a standard process for conversion (as described by Graphical Device Interface (GDI) color space conversions documentation from the International Color Consortium (ICC)).

First, the block converts the luma component (Y), blue-difference chroma component (Cb), and red-difference chroma component (Cr) of the input signal to 5:6:5 RGB format where the red and blue channels of the source use a 5-bit representation and the green channel uses 6 bits.

Now the block converts your 5:6:5 RGB to 8:8:8 RGB using the following conventions:

- 1** For the red and blue 5-bit channels, it copies the three most significant bits (MSB) from the 5-bit source word and append them to the lower order end of the target word.
- 2** For the green 6-bit channel, it copies the two MSBs from the green source word and append them to the lower order end of the target green word.

The results is to output three RGB channels — red, green, and blue — each with 8-bit words.

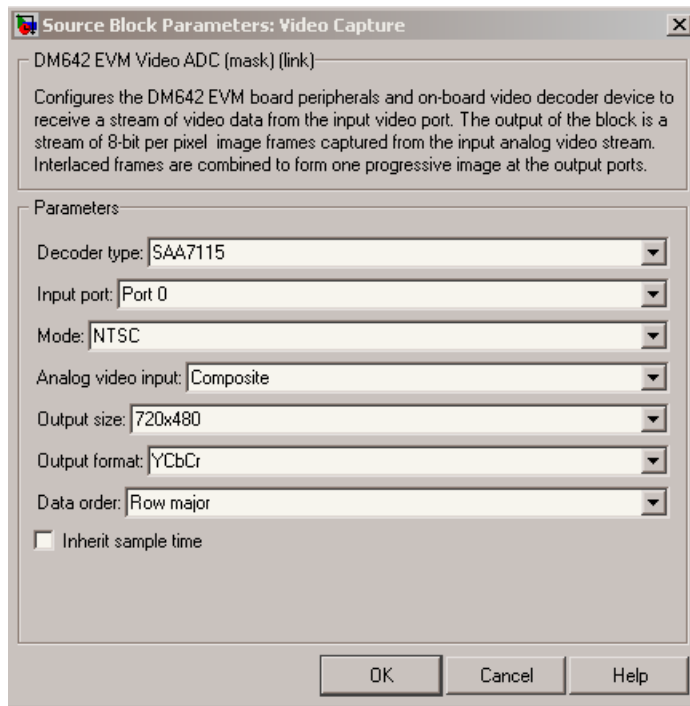
For example, to convert hexadecimal values by this algorithm, 5:5:5 RGB data of (0x19, 0x33, 0x1A) becomes (0xCE, 0xCF, 0xD6) of 8:8:8 RGB output.

To do the conversion in the binary case for 5:5:5 RGB data:

- 1** blue data 1 1101 converts to 11101111
- 2** for the green channel, conversion takes 11 0011 to 1100 1111
- 3** red data 1 0101 becomes 1010 1101 (same algorithm as blue data)

To maximize the speed of the RGB conversion, the Video ADC block provides color space conversion using a routine written in assembly language and optimized for the DM64x processor core. Using the optimized color space conversion code replaces the Color Space Conversion block available from the Video and Image Processing Blockset™ (VIP blockset). While you can use any compatible VIP blockset block with the DM642, this particular color space conversion operation is handled better by the conversion code included in the ADC block.

Dialog Box



Decoder type

Configures the block options to support either the TVP5146 Decoder on the DM642 EVM or the SAA7115 Decoder, depending on the model of your board. Choose one option from the list — TVP5146 or SAA7115. When you select SAA7115 for the type of decoder, the dialog box adds a new option — **Output Mode**. Generally, older DM642 EVM boards use the SAA7115 decoder. Newer boards use the default setting TVP5146 decoder. Refer to “Identifying Your DM642 EVM Board Version” on page A-3 for information about identifying the revision of your DM642 EVM.

Input port

Directs the block to capture video from either the 0 or 1 video input port on the DM642 EVM. The block does not support port 2

for video input. Input port 0 provides both composite video (via connector J15) and S-video (connector J16) inputs.

Mode

Select the video format to capture from the list. The block supports NTSC and PAL video formats.

Analog Video Input

Select composite video or S-video. The video decoder connected to port 0 has both composite and S-video inputs. These are available via connector J15 and J16, respectively. Port 1 has two composite video connectors and no S-video availability.

Output size

Reports the size of the video images to output. **Output size** is a read-only parameter set to 720 x 576 resolution elements when you select PAL mode and the TVP5146 decoder in **Decoder type**. When you select NTSC mode with the TVP5146 decoder, **Output size** reports the read-only value 720 x 480.

If you select the SAA7115 decoder, **Output size** lists the available video sizes to output for further processing, depending on the **Mode** setting. The following tables show the sizes to pick from depending on whether you pick NTSC or PAL for **Mode**. The block scales the input video to the selected size for output.

Video Output Size Options For NTSC Mode	Description
128 x 96	Output NTSC video with dimensions 128 pixels by 96 pixels. Scales the output to 1/4 the resolution of QCIF video.
176 x 144	Output NTSC video with dimensions 176 pixels by 144 pixels. Scales the output to 1/4 the resolution of CIF video.

Video Output Size Options For NTSC Mode	Description
320 x 240	Output NTSC video with dimensions 320 pixels by 240 pixels. Scales the output to standard interchange format NTSC. Derived from CCIR 601 video (most often).
720 x 480	Output NTSC video with dimensions 720 pixels by 480 pixels. Scales the output to higher definition TV mode.

Video Output Size Options For PAL Mode	Description
128 x 96	Output video with dimensions 128 pixels by 96 pixels
176 x 144	Output video with dimensions 176 pixels by 144 pixels.
320 x 240	Output video with dimensions 320 pixels by 240 pixels
720 x 576	Output video with dimensions 720 pixels by 576 pixels

Output format

Determines how the block represents color data in the output. Choose one of the following color representations according to what your model and algorithm require.

Digital Output Format	Description
RGB24	Output uses 8 bits each of red, green, and blue colors to represent the color of each pixel in the image. RGB color space is device-dependent.
YCbCr	<p>Output from the block includes three channels to represent the color image data per pixel:</p> <ul style="list-style-type: none">• Y — the luma component (essentially a black/white signal)• Cb — the blue-difference chroma component• Cr — the red-difference chroma component <p>This is the digital standard color space DVDs use.</p>
Y	Black/White video. No color/chromaticity values.

Data order

With data order, you control the way the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

- Row major — store video data in row major order. This is the default setting and matches most video data.
- Column major — store video data in column major order. The Simulink and MATLAB software use this format to store images and matrices.

DM642 EVM Video ADC blocks store the image data in row major format because most video capture devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

MATLAB and Simulink software use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video ADC blocks.

To address this problem, the Video ADC blocks include an option **Data order** to let you select either row major or the column major storage formats. By default, this block uses row major data format.

When you select **Column major**, the block performs an explicit transposition on the image data to map the data format from row major to column major order. To minimize the processor time spent on the transposition, the block uses optimized assembly routines to transpose the image data.

Inherit sample time

Selecting **Inherit sample time** sets the sample time to -1 . To use this block in a function call subsystem, you must select this option. **Inherit sample time** is cleared by default and the block uses the model sample time.

Specifying sample-time inheritance for a this block, a source block, can cause Simulink software to assign an inappropriate sample time to the block. You should avoid selecting **Inherit sample time** unless you are required to do so because you placed the block in a function call subsystem. When you select **Inherit sample time**, Simulink software displays a warning message when you update or simulate the model.

See Also

DM642 EVM Video DAC

DM642 EVM Video DAC

Purpose Video encoder to display video

Library “DM642 EVM (dm642evmlib)” on page 6-7

Description



In the project generated from a model, this block provides the code to gather video from another block in the model, and direct the video stream to the video output port on the board.

You should input unsigned 8-bit integers to the block in the specified mode.

Adding this block to a model enables code generated from your model to perform the following tasks:

- 1 Capture digital video data from the application on your DM642 EVM.
- 2 Buffer the captured video into frames for NTSC display — two fields per frame and 30 frames per second, or SVGA display — RGB24 color with noninterlaced frames.
- 3 Convert to analog video.
- 4 Output the converted analog video to the EVM Video Out ports.

Unlike the DM642 EVM Video ADC block, this DAC block does not convert the video between formats. Nor does this block inherit any settings from the DM642 EVM Video ADC block, as some of the other C6000 DAC blocks do.

The **Mode** option specifies both the video format the block accepts and the format the block outputs to the video output ports on the EVM.

To be able to be displayed, images that you send to the block should be equal to or smaller than the target display size. If the input images are smaller than the target display size, the block pads the image by adding zeros to the image.

When you add this block to your Simulink model, it has no affect on your simulation — it outputs a string of zeros. In code generation, the

block creates the device code needed to buffer, convert, and send video to the output port on the EVM.

Note The DM642EVM board provides both composite and S-video connectors for output. However, these are driven simultaneously, so you do not need to specify which one is to be used.

Memory Use

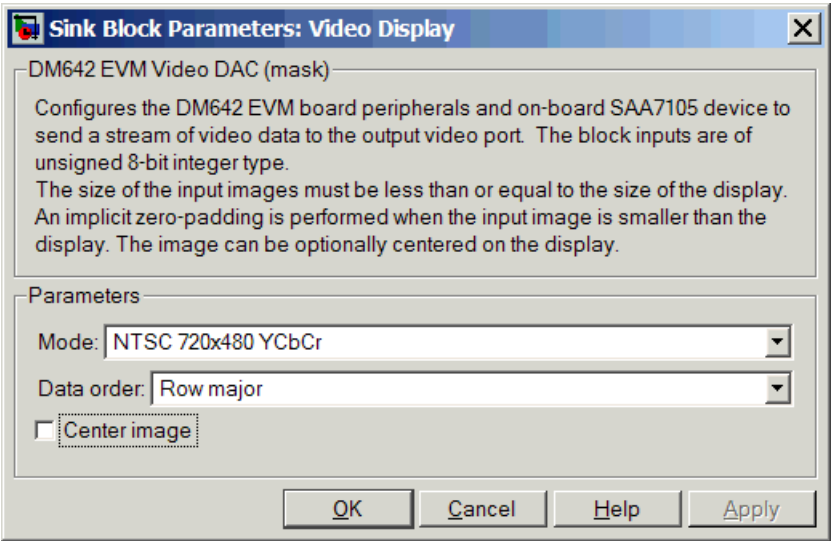
This block allocates video capture buffers on the system heap, using a TI driver that allocates three frame buffers on the heap for continuous video capture. To use the block you must create a heap in external memory on the target with the label EXTERNALHEAP. If you do not create the heap, either using the default values in the DM642 Target Preferences block or setting your own values. Target Support Package software returns an error.

Use **Create heap** and **Heap size** and set the heap size in the DM642EVM Target Preferences block to configure the heap. Select **Define label** and name the heap EXTERNALHEAP in **Heap label**.

The default settings for the target preferences create a heap with sufficient memory to handle the worst case memory allocation needs automatically. If you configure the heap without sufficient memory, you get a run-time error because the system cannot initialize the video driver.

DM642 EVM Video DAC

Dialog Box



Mode

Specifies the video format for the block. The block then sends video in this format to the video output port on the EVM. The **Mode** parameter offers the following options:

Analog Output Mode	Description
NTSC 720x480 YCbCr	Analog output of video data in 720-by-480 pixels format with full color.
NTSC 640x480 Y	Analog video output in 640-by-480 pixels format with black and white only (luminance). No color data.
SVGA 800x600 RGB24	Full super VGA format 800-by-600 pixels with three color channels: 8-bit red, 8-bit green, and 8-bit blue data.

Analog Output Mode	Description
PAL 720x570 YCbCr	Analog output of video data in 720-by-570 pixels PAL format with full color.
PAL 720 x 570 Y	Analog output of video data in 720-by-570 pixels PAL format with black and white only (luminance). No color data.

Data order

With data order, you control the way the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

- **Row major** — store video data in row major order. This is the default setting and matches most video data.
- **Column major** — store video data in column major order. Simulink and MATLAB software use this format to store images and matrices.

DM642 EVM Video DAC blocks store the image data in row major format because most video display devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

MATLAB and Simulink software use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video DAC blocks.

To address this problem, the Video DAC blocks include an option **Data order** to let you select either row major or the column major storage formats. By default, these blocks use row major data format.

When the column major data ordering option is selected, the block performs an explicit transposition on the image data to map the data format from row major to column major order.

To minimize the processor time spent on the transposition, the block uses optimized assembly routines to accomplish the image transposition.

Center Image

Directs the block to center the output image on the display. Centering the image requires some computation by the processor so there are small time and CPU cycles penalties for choosing this option. For that reason, **Center image** is cleared by default.

Another note of interest — some cameras pad their video output with zeros to ensure that the display does not cut off the image on one side, usually the left. Images that include such padding may appear to be off-center on the display. In fact, while the displayed image may not appear centered, the electronic image (the data that compose the displayed image plus the padding which you cannot see) is centered in the display area.

See Also

DM642 EVM Video ADC

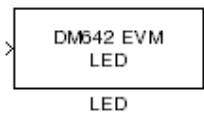
Purpose

Control LEDs

Library

“DM642 EVM (dm642evmlib)” on page 6-7

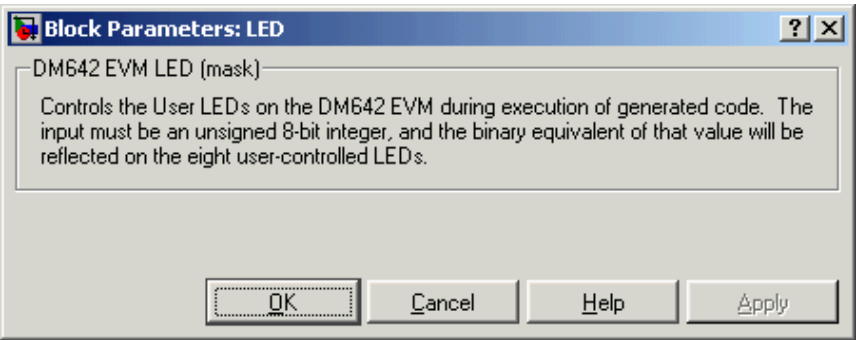
Description



Controls the User LEDs on the DM642 EVM while the processor executes your generated code. To trigger the LEDs, input an unsigned 8-bit integer to the block. In response, the eight user-controlled LEDs reflect the binary equivalent of that input value — turning off an LED is 0 and turning on an LED is 1.

During operation, the LED block inherits the sample time from the upstream block in the model. Therefore, each time the model operation encounters the LED block, the block writes the desired output value to the LEDs.

Dialog Box



You see the block does not provide user options. Adding the block to your model adds the ability to control the LEDs.

DM642 EVM Video Port

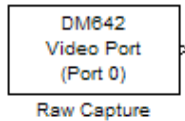
Purpose

Video port to receive video data from video input port

Library

“DM642 EVM (dm642evmlib)” on page 6-7

Description



Adding this block to your model lets you define the format of raw video captured by the video port on the DM642 EVM. The block outputs video as a stream of image frames built from the defined input.

You can select the video port the block reads from, set the size of the input data in bits per pixel, and define the frame sizes in pixels and lines.

When your process captures standard video input, like NTSC format video, another block for the DM642 EVM may be appropriate — the DM642 EVM Video ADC block.

By default, the block settings define NTSC format input video to capture — 640 pixels wide by 480 lines tall using 8 bits per pixel.

The block does not check your inputs to determine whether they form valid frames. You must be sure the values you assign work for your application.

The block does not support video capture from port 2 on the EVM.

Blanking intervals, both horizontal and vertical, represent the time needed for the scan to return to the starting point of the next line (the horizontal blanking period) or field or frame (the vertical blanking period).

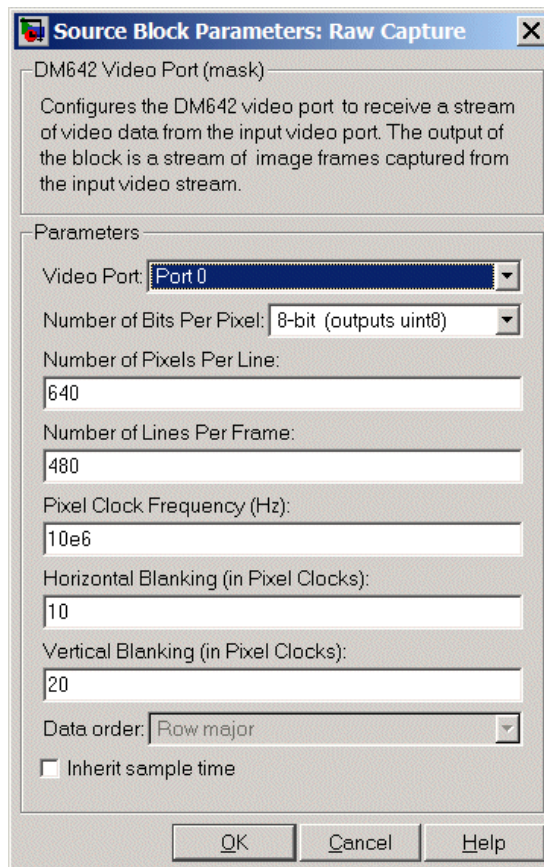
Memory Use

This block allocates video capture buffers on the system heap, using a TI driver that allocates three frame buffers on the heap for continuous video capture. To use the block you must create a heap in external memory on the target with the label EXTERNALHEAP. If you do not create the heap, either using the default values in the DM642 Target Preferences block or setting your own values. Target Support Package software returns an error.

Use **Create heap** and **Heap size** and set the heap size in the DM642EVM Target Preferences block to configure the heap. Select **Define label** and name the heap EXTERNALHEAP in **Heap label**.

The default settings for the target preferences create a heap with sufficient memory to handle the worst case memory allocation needs automatically. If you configure the heap without sufficient memory, you get a run-time error because the system cannot initialize the video driver.

Dialog Box



Source Block Parameters: Raw Capture [X]

DM642 Video Port (mask)

Configures the DM642 video port to receive a stream of video data from the input video port. The output of the block is a stream of image frames captured from the input video stream.

Parameters

Video Port: Port 0

Number of Bits Per Pixel: 8-bit (outputs uint8)

Number of Pixels Per Line: 640

Number of Lines Per Frame: 480

Pixel Clock Frequency (Hz): 10e6

Horizontal Blanking (in Pixel Clocks): 10

Vertical Blanking (in Pixel Clocks): 20

Data order: Row major

☐ Inherit sample time

OK Cancel Help

Video Port

Select the video port to be the source of the raw video data stream. Either 0 or 1 appear on the list and 0 is the default port.

Number of bits per pixel

Select the number of bits used to represent a pixel in the input video stream. List entries tell you the input pixel representation and the data type of the output pixels for each input size. You cannot enter values here. Select from the list.

Number of pixels per line

Configure the width of each video frame in pixels. Enter the pixel count as an integer greater than zero.

Number of lines per frame

Configure the height of a single frame of video in lines. Enter the number of lines as an integer greater than zero. Combined with the **Number of bits per pixel**, this specifies the video frame format.

Pixel clock frequency

Specify the rate at which picture elements (pixels) arrive at the block input. Usually you enter this in Hz using scientific notation as shown by the default value. You can enter the value in decimal notation as well.

Horizontal blanking (in pixel clocks)

The blanking signal that occurs at the end of each video scanning line. Enter the value as an integer number of pixels. One video line comprises the number of pixels in the line plus the horizontal blanking pixels.

Vertical blanking (in pixel clocks)

The blanking signal that occurs at the end of each video field or frame. Enter this value as an integer number of lines (pixels). One frame includes the number of lines in the height of the frame plus the additional blanking lines.

Data order

With this option you tell the encoder whether to output video in row major or column major order. Most video capture and display systems use row major ordering. MATLAB and Simulink software use column major order. As a result, some Simulink blocks and MATLAB operations may not produce the output you expect unless you change the ordering for video from the default row major setting to column major.

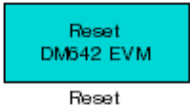
Inherit sample time

Selects whether the block inherits the sample time from the model base rate or Simulink base rate as determined in the Solver options in Configuration Parameters. Selecting **Inherit sample time** directs the block to use the specified rate in model configuration. Entering -1 configures the block to accept the sample rate from the upstream HWI, Task, or Triggered Task blocks.

See Also

DM642 EVM Video ADC, DM642 EVM Video DAC

DM642 EVM Reset

Purpose	Reset to initial conditions
Library	“DM642 EVM (dm642evmlib)” on page 6-7
Description	<div></div> <p>Double-clicking this block in a Simulink model window resets the DM642 EVM that is running the executable code built from the model. When you double-click the Reset block, the block runs the software reset function provided by CCS IDE that resets the processor on your DM642 EVM. Applications running on the board stop and the signal processor returns to the initial conditions you defined.</p> <p>Before you build and download your model, add the block to the model as a stand-alone block. You do not need to connect the block to any block in the model. When you double-click this block in the block library it resets your DM642 EVM. In other words, anytime you double-click a DM642 EVM Reset block you reset your DM642 EVM.</p>
Dialog Box	This block does not have settable options and does not provide a user interface dialog box.

Purpose

Configure AIC33 audio codec to capture audio stream from LINE-IN or MIC

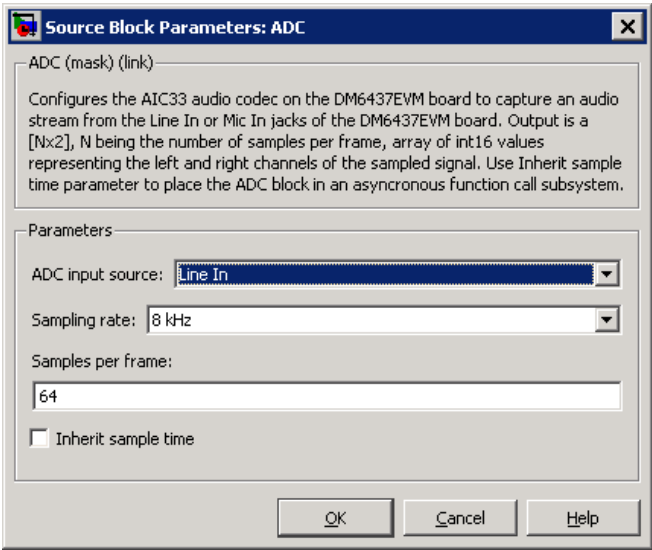
Library

“DM6437 EVM (dm6437evmlib)” on page 6-7

Description

This block uses the AIC33 audio codec on the DM6437 EVM board to capture an analog audio stream from the **Line In** or **Mic** jacks and generate a digital frame-based output. Output is a [Nx2] array of int16 values representing the left and right channels of the sampled signal, where N is the number of samples per frame. Use the **Inherit sample time** parameter to place the ADC block in an asynchronous function call subsystem.

Dialog Box



ADC input source

Select **Line In** or **Mic In** as the input source.

Sampling Rate

Set the sampling rate of the analog-to-digital converter, from 8 kHz (the default) to 96 kHz.

Samples per frame

Set the number of samples the block buffers internally before it sends the digitized signals, as a frame vector, to the next block in the model. This value defaults to **64** samples per frame. The frame rate depends on the sample rate and frame size. For example, if **Sampling Rate** is 8 kHz, and **Samples per frame** is 32, the frame rate is 250 frames per second ($8000/32 = 250$).

Inherit sample time

Select whether the block inherits the sample time from the model base rate or Simulink base rate as determined in the Solver options in Configuration Parameters. Selecting Inherit sample time directs the block to use the specified rate in model configuration. Entering -1 configures the block to accept the sample rate from the upstream HWI, Task, or Triggered Task blocks.

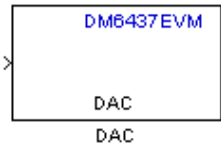
See Also

DM6437 EVM DAC

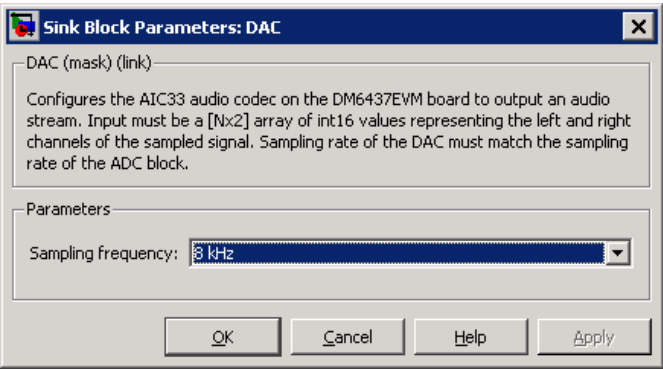
Purpose Configure AIC33 codec to convert digital signal to audio output on LINE OUT and HP OUT

Library “DM6437 EVM (dm6437evmlib)” on page 6-7

Description Configure the AIC33 stereo codec on the DM6437 EVM board to convert a digital signal to an analog audio stream on the LINE OUT and HP OUT output jacks. The digital signal input must be an [Nx2] array of int16 values. Column 1 of the array is the left channel and column 2 is the right channel of the sampled signal. The sampling rate of the DAC output must match the sampling rate of the digital signal from the ADC.



Dialog Box



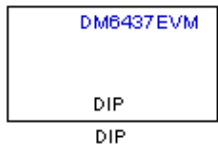
Sampling frequency Select the sampling rate of the digital signal input. This value must match the **Sampling rate** of the ADC block in your model.

See Also DM6437 EVM ADC

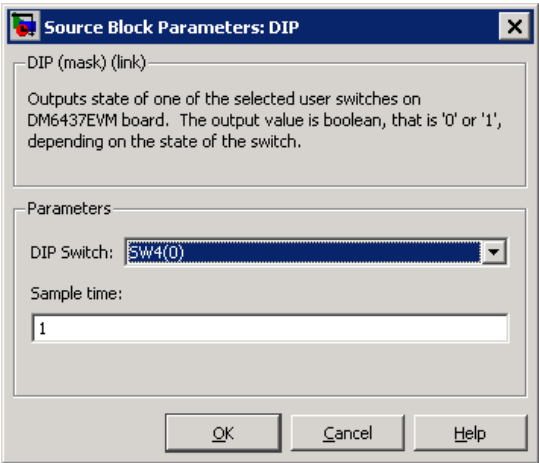
Purpose Output state of user-selected DIP switch as Boolean

Library “DM6437 EVM (dm6437evmlib)” on page 6-7

Description Outputs the state of a user-selected DIP switch or jumper on the DM6437 EVM board. The output is a Boolean value, 0 (open) or 1 (closed). Use multiple blocks to output the state of multiple DIP switches.



Dialog Box



DIP Switch Select the switch or jumper to sample: SW4(0),SW4(1), SW4(2), SW4(3), JP1, SW7.

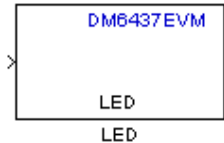
SW4 is a read-only user switch. JP1 is for NTSC/PAL selection. SW7 is a slide switch.

Sample time The interval between samples, in seconds. This value defaults to 1 second between samples.

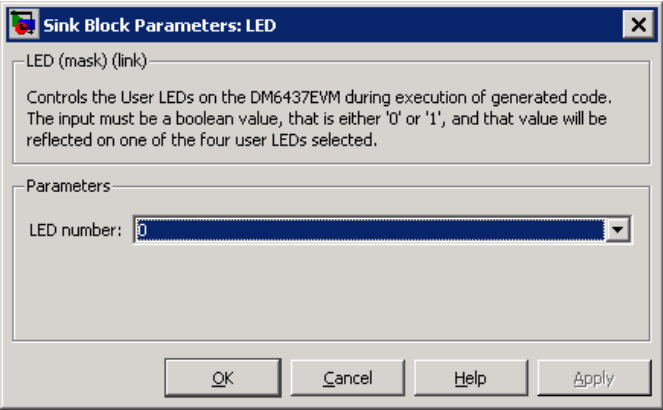
Purpose Apply Boolean input to user-selected LED

Library “DM6437 EVM (dm6437evmlib)” on page 6-7

Description This block controls an individual LED among the User LEDs on the DM6437 EVM during execution of generated code. The block input accepts Boolean values, 0 (off) or 1 (on). Use multiple blocks to control multiple LEDs.



Dialog Box



LED number Specify the number of the User LED that the Boolean input controls.

DM6437 EVM Video Capture

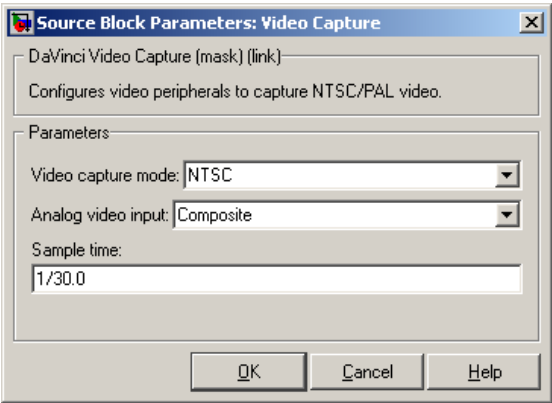
Purpose Configure video peripherals to capture NTSC/PAL video

Library “DM6437 EVM (dm6437evmlib)” on page 6-7

Description Configure the video peripherals to capture an NTSC/PAL video input and make it available as a stream of YCbCr 4:2:2 interleaved data.



Dialog Box



Video capture mode
Set the video format to match that of the input, **NTSC** or **PAL**.

Analog video input
Set the input type to match that of the input, **Composite** or **S-video**.

Sample time
Set a sample time rate that matches the frame rate of the input signal, typically 1/30 for NTSC and 1/25 for PAL. A mismatch between these two rates may cause discontinuities in the video output signal.

See Also

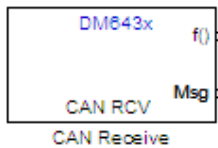
DM643x Draw Rectangles, DM643x OSD, DM643x Video Display

DM643x CAN Receive

Purpose Receive messages from CAN serial communications bus on DM643x

Library “DM6437 EVM (dm6437evmlib)” on page 6-7
“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

Description The CAN Receive block listens to broadcast messages on the DM643x CAN protocol bus. It saves messages with the user-specified **Message Identifier** to its message buffer. The CAN Receive block polls the message buffer at a rate determined by **Sample time**. When it detects a message in the message buffer, the block triggers the function-call output (f0) and makes the CAN message data available at the message output (Msg).



Dialog Box

Source Block Parameters: CAN Receive

DM6437EVM CAN Receive (mask) (link)

Configures a CAN mailbox to receive messages from the CAN bus on the DM6437EVM. When the message is received, emits the function call to the connected function-call subsystem as well as outputs the message data in selected format and the message data length in bytes.

Parameters

Mailbox number:
0

Message identifier:
bin2dec('111000111')

Message type: Standard (11-bit identifier)

Sample time:
1

Data type: uint16

☐ Output message length

OK Cancel Help

Mailbox number

Enter a unique number from 0 to 15 for standard or from 0 to 31 for enhanced CAN mode. This field refers to a mailbox area in RAM. In standard mode, the mailbox number determines priority.

Message identifier

Identifies the length of the message—11 bits for standard frame size or 29 bits for extended frame size in decimal, binary, or hex formats. If the format is binary or hex, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry. The message identifier is associated with a receive mailbox. This mailbox only accepts messages that match the mailbox message identifier.

Message type

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

Sample time

Frequency with which the mailbox is polled to determine if a new message has been received. A new message causes a function call to be emitted from the mailbox. To update the message output only when a new message arrives, the block must be executed asynchronously. To execute this block asynchronously, set **Sample Time** to -1. Refer to “Schedulers and Timing” on page 2-23 for a discussion of block placement and other necessary settings.

For information about setting the timing parameters of the CAN module see Chapter 5, “Configuring Timing Parameters for CAN Blocks”.

Data type

Type of data in the data vector. The length of the vector for the received message is, at most, 8 bytes. If the message is less than 8 bytes, the data buffer bytes are right-aligned in the output. Only `uint16` (vector length = 4 elements) or `uint32` (vector length = 8 elements) data are allowed. This block uses an 8-byte data buffer to unpack the data, as follows:

DM643x CAN Receive

For uint16 data,

```
Output[0] = data_buffer[1..0];
Output[1] = data_buffer[3..2];
Output[2] = data_buffer[5..4];
Output[3] = data_buffer[7..6];
```

For uint32 data,

```
Output[0] = data_buffer[3..0];
Output[1] = data_buffer[7..4];
```

For example, if the received message has two bytes,

```
data_buffer[0] = 0x21
data_buffer[1] = 0x43
```

the uint16 output would be:

```
Output[0] = 0x4321
Output[1] = 0x0000
Output[2] = 0x0000
Output[3] = 0x0000
```

Output message length

Select this option to output the message length, in bytes, to the third output port. If you do not select this option, the block has only two output ports.

References

For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981, available at the Texas Instruments Web site.

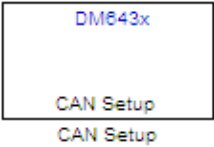
See Also

Chapter 5, “Configuring Timing Parameters for CAN Blocks”, DM643x CAN Setup, DM643x CAN Transmit

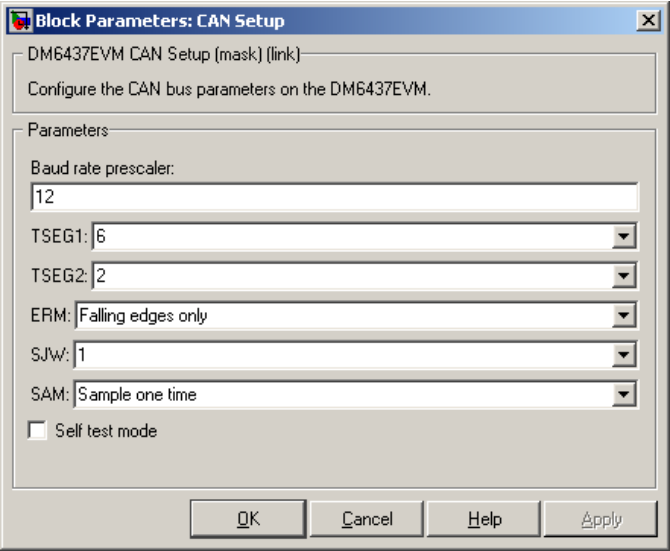
Purpose Configure CAN serial communications bus parameters on DM643x

Library “DM6437 EVM (dm6437evmlib)” on page 6-7
“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

Description This block configures the CAN serial communications bus parameters on the DM6437EVM. The Chapter 5, “Configuring Timing Parameters for CAN Blocks” topic provides instructions and examples for configuring this block.



Dialog Box



Baud rate prescaler
Value by which to scale the bit rate. Valid values are 0 to 255.

TSEG1

(Time SEGment 1) Sets the value of time segment 1, which, with **TSEG2** and **Baud rate prescaler**, determines the length of a bit on the CAN bus. Valid values for **TSEG1** are 2 through 16.

TSEG2

(Time SEGment 2) Sets the value of time segment 2, which, with **TSEG1** and **Baud rate prescaler**, determines the length of a bit on the CAN bus. Valid values for **TSEG2** are 2 through 8.

ERM

(Edge Resynchronization Mode) Sets the message resynchronization triggering. Options are **Falling edges only** and **Both falling and rising edges**.

SJW

(Synchronization Jump Width) For CAN to work successfully, all nodes on the network must be synchronized. However, as time passes, clocks on different nodes drift out of sync, and must resynchronize. **SJW** specifies the maximum width (in time quanta) that can be added to **TSEG1** (in the case of a slower transmitter), or subtracted from **TSEG2** (in the case of a faster transmitter) to regain synchronization during the receipt of a CAN message. Valid values for **SJW** are 1 to 4.

SAM

(SAMple point setting) Number of samples used by the CAN module to determine the CAN bus level. Selecting **Sample one** time samples once at the sampling point. Selecting **Sample three** times samples once at the sampling point and twice before at a distance of $TQ/2$ (Time Quanta/2). A majority decision is derived from the three points.

Self test mode

Puts the CAN module into loopback mode, that sends a dummy acknowledge message without requiring an acknowledge bit.

References

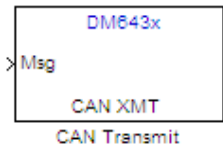
For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981, available at the Texas Instruments Web site.

See Also

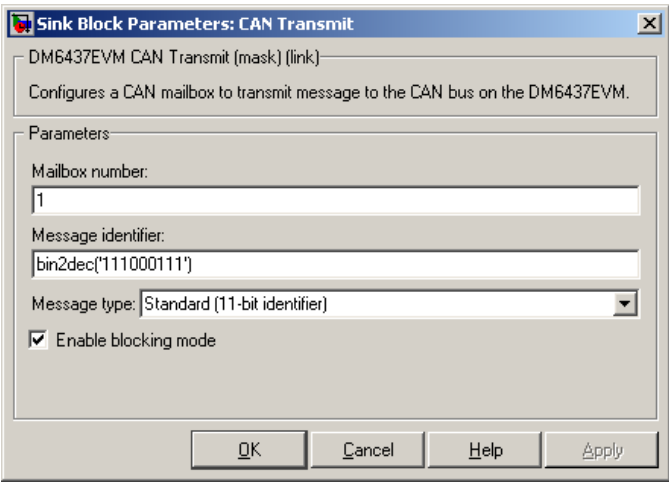
Chapter 5, “Configuring Timing Parameters for CAN Blocks”, DM643x CAN Transmit, DM643x CAN Receive

DM643x CAN Transmit

Purpose	Configure CAN mailbox to transmit messages on CAN serial communications bus on DM643x
Library	“DM6437 EVM (dm6437evmlib)” on page 6-7 “AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2
Description	The CAN Transmit block receives messages through the message input (Msg) and broadcasts them to the CAN serial communication bus on the DM643x.



Dialog Box



Mailbox number
Sets the value of the mailbox number register (MBNR). For standard CAN controller (SCC) mode, enter a unique number from 0 to 15. For high-end CAN controller (HECC) mode enter a unique number from 0 to 31 . In SCC mode, transmissions from the mailbox with the highest number have the highest priority. In

HECC mode, the mailbox number only determines priority if the Transmit priority level (TPL) of two mailboxes is equal.

Message identifier

Sets the value of the message identifier register (MID). The message identifier is 11 bits long for standard frame size or 29 bits long for extended frame size in decimal, binary, or hex format. For the binary and hex formats, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry.

Message type

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

Enable blocking mode

If you enable blocking mode, the CAN block code blocks further transmissions indefinitely until it receives a successful transmit acknowledge (TA bit in the CANTA register = 1). If you disable blocking mode, the CAN block code continues transmitting without receiving successful transmit acknowledgements. This is useful when the hardware might fail to acknowledge transmissions.

References

For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981, available at the Texas Instruments Web site.

See Also

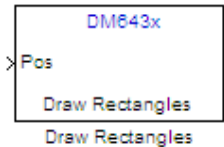
Chapter 5, “Configuring Timing Parameters for CAN Blocks”, DM643x CAN Setup, DM643x CAN Receive

DM643x Draw Rectangles

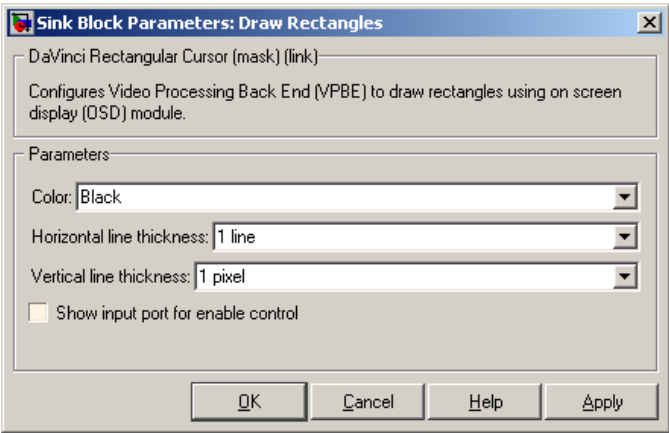
Purpose Configure Video Processing Back End to draw rectangles using On Screen Display (OSD) module

Library “DM6437 EVM (dm6437evmlib)” on page 6-7
“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

Description This block configures the Video Processing Back End (VPBE) to draw and position rectangles using the On Screen Display (OSD) module. The position input (**Pos**) is a 1x4 vector, designates the location of the upper-left corner of the rectangle. The position coordinates (0,0) originate in the upper-left corner of the video display.



Dialog Box



Color Select the rectangle color. For **Specify via dialog**, enter an integer between 0–255. This integer specifies a corresponding RGB color in the DM643x ROM0 color lookup table (DM643x ROM0 CLUT). If you select **Specify via input port**, the block displays an additional input port, Color. Like **Specify via dialog**, the Color input takes an integer between 0–255 that fetches a color from the DM643x ROM0 CLUT. Changing the input value to

the Color input port can change the color of the rectangle while the model is running.

DM643x ROM0 color lookup table																									
	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240										
1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241										
2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242										
3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243										
4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244										
5	21	37	53	69	85	101	117	133	149	165	181	197	213	229											
6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246										
7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247										
8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248										
9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249										
10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250										
	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251										
	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252										
	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253										
	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254										
	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255										

For more information about the DM643x ROM0 CLUT, enter the following text at the MATLAB command prompt:

```
help 'dm643x_clut'
```

Horizontal line thickness

Select the cursor height in lines.

Vertical line thickness

Select the cursor width in pixels.

Show input port for enable control

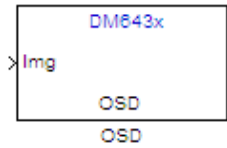
Create an input port (**En**) that can be used to enable or disable the position input.

See Also

DM643x OSD, DM643x Video Capture, DM643x Video Display

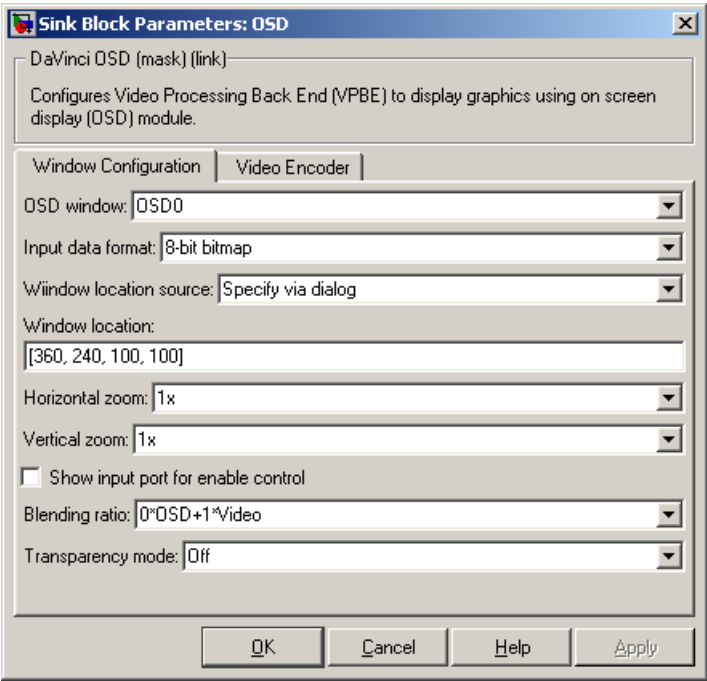
DM643x OSD

Purpose	Overlay graphics and text on video
Library	“DM6437 EVM (dm6437evmlib)” on page 6-7 “AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2
Description	Use the On Screen Display (OSD) capabilities of the Video Processing Back End (VPBE) to overlay graphics and text on video.



Dialog Box

Window Configuration Pane



OSD window

Display graphics using OSD window 0 or 1.

Window Mode

If you set **OSD Window** to **OSD1**, the **Window Mode** parameter appears. Selecting **Display** configures OSD1 to display graphics. Selecting **Attribute** configures OSD1 to serve as an “alpha” input for controlling the transparency of OSD0. The positions of the two OSD windows must match for this to work.

Input data format

Set the format of the input data to 1-, 2-, 4-, 8-bit bitmap, or RGB565 which provides 16-bit color depth (64k colors).

Due to bandwidth constraints, RGB565 can only be used with one OSD window at a time. If you are using OSD1 to control transparency (i.e., OSD1 **Window Mode** is **Attribute**), get the best color depth by setting OSD1 **Input data format** to one of the bitmap settings and OSD0 **Input data format** to **RGB565**.

Window location source

Select the method for setting the location of the graphics display window. **Specify via dialog** creates the **Window location** field. **Specify via input port** creates an position input (Pos) on the OSD block which accepts the location of the window as data.

Window location

This parameter appears when you set **Window location source** to **Specify via dialog**. Set the pixel width, height, and base coordinates. For example, the default values, [360, 240, 100, 100] set the width to 360 pixels, the height to 240 pixels, the base coordinates for x to 100 pixels, and the base coordinates for y to 100 pixels.

Note [0, 0], the origin of the coordinate system, is the located in the upper-left corner of the Video0 window.

Horizontal zoom

Set the horizontal magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the OSD block.

Vertical zoom

Set the vertical magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the OSD block.

Show input port for enable control

Create an input port (**En**) to enable or disable the OSD graphics display window. This parameter is not available when **Window Mode** is **Attribute**.

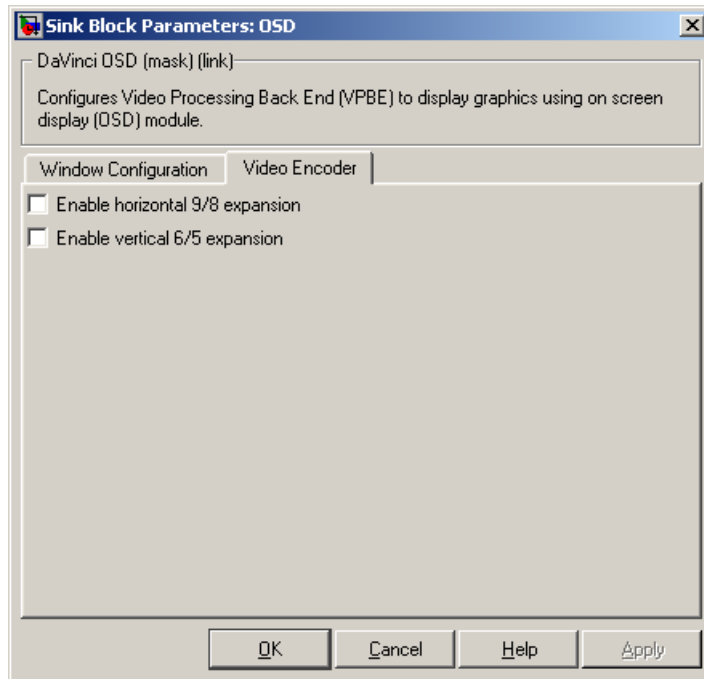
Blending ratio

Control the degree of blending between the OSD graphics display window and the Video display window in the background. This can be used to superimpose a semitransparent OSD graphic on a video background or to create fade-in and fade-out effects. The settings range from full OSD to full video in steps of 1/8. An additional setting, **Specify via input port**, creates an input port (**Blend**) for changing the ratio dynamically.

Transparency mode

Turn the transparency mode of the graphics display window **On** or **Off**, or select **Specify via input port** to create an input (**Trans**) on the OSD block. With transparency enabled, OSD pixels that match the color of the Video background color are rendered transparent. This is used for typical “bluescreen” type effects.

Video Encoder Pane



Enable horizontal 9/8 expansion

Expands the image horizontally and is typically used to compensate for spatially compressed NTSC and PAL video signals. For example, you can use this setting to correct a 720 x 480 pixel NTSC analog video input that is displayed as a 640 x 480 pixel image.

Enable vertical 6/5 expansion

Expands the image vertically and is typically used to compensate for spatially compressed PAL video signals. For example, you can use this setting in combination with the **Enable horizontal 9/8 expansion** setting to correct a 720 x 576 pixel PAL analog video input that is displayed as a 640 x 480 pixel image.

DM643x OSD

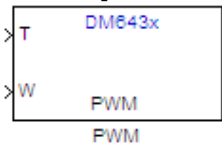
See Also

DM643x Draw Rectangles, DM643x Video Capture, DM6437 EVM
Video Capture, DM643x Video Display

Purpose Configure DM643x DSP Event Manager to generate PWM waveforms

Library “DM6437 EVM (dm6437evmlib)” on page 6-7
“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

Description This block configures any one of the three PWM modules on the DM6437; each module has one output. The PWM module’s clock cycles depend on the DM6437’s 27 MHz input clock, and are not affected by the DM6437’s PLL module. Upon startup, the PWM module uses the **Initial waveform period** and **Initial duty-cycle** values. Inputs to the waveform period port, **T**, and the duty-cycle port, **W**, can change those values while the application is running.

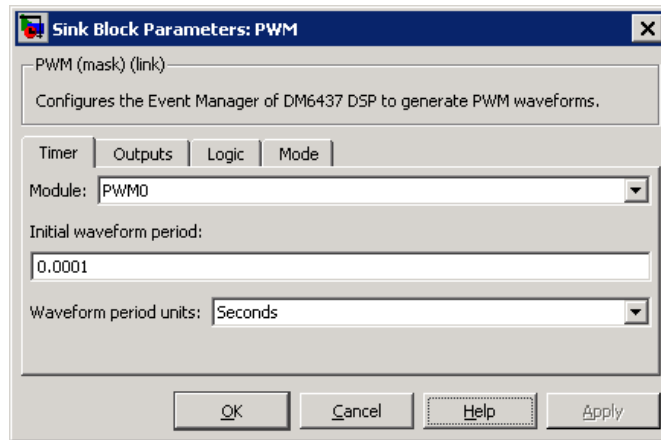


Dialog Box The PWM block dialog box comprises four tabs:

- **Timer** — Select the PWM module, and configure the initial waveform.
- **Outputs** — Configure the initial duty cycle.
- **Logic** — Configure the control logic.
- **Mode** — Configure one-shot or continuous operation.

The following sections describe the contents of each tab in the dialog box.

Timer



Module

Select the PWM module for this block. All the parameter settings in this block configure the registers of the PWM module selected.

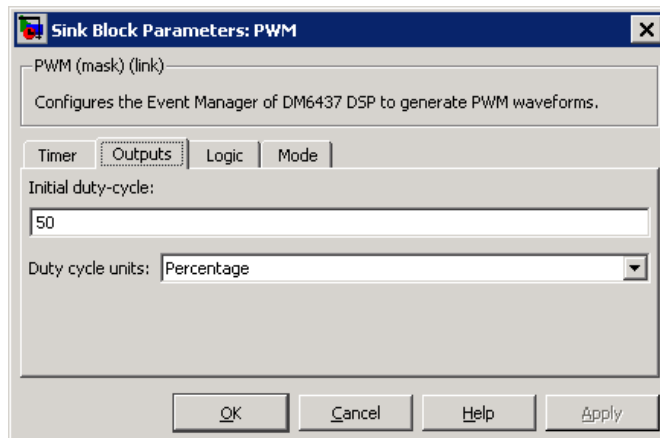
Initial waveform period

Set the initial period of the PWM waveform. The waveform period applied at the input port, **T**, changes this value. The range of acceptable values is 0.000000296 to 79.536431370 seconds or 8 to $2^{31}-1$ clock cycles. These ranges depend on the 27 MHz clock frequency and the width of the 32-bit register.

Waveform period units

Set the unit of measure of the waveform period to **Seconds** or **Clock cycles**. This setting applies to both the **Initial waveform period** and the waveform period input, **T**. Clock cycles depend on the DM6437's 27 MHz input clock.

Outputs



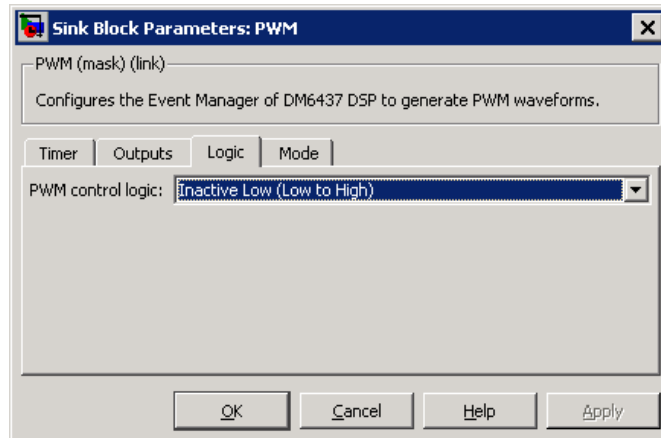
Initial duty-cycle

Set the initial duty-cycle of the PWM. The duty-cycle applied at the input port, **W**, changes this value. The range of acceptable values is 0 to 100 percent or 8 to $2^{31}-1$ clock cycles. These ranges depend on the 27 MHz clock frequency and the width of the 32-bit register.

Duty-cycle units

Set the unit of measure of the duty-cycle to percentage or clock cycles. This setting applies to both the **Initial duty-cycle** and the duty-cycle input, **W**. Clock cycles depend on the DM6437's 27 MHz input clock.

Logic

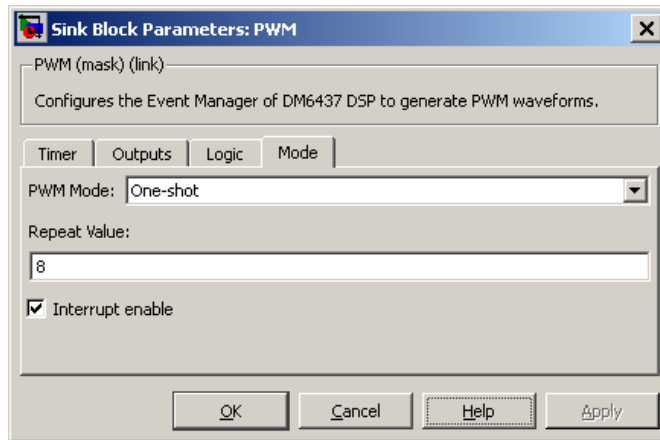


PWM control logic

Control the state of the PWM output while it is inactive and the polarity of the PWM waveform when it is active:

- **Inactive Low (Low to High):** When the PWM output is inactive, the output remains low. When it is active, the first phase is low, and the second phase is high.
- **Inactive Low (High to Low):** When the PWM output is inactive, the output remains low. When it is active, the first phase is high, and the second phase is low.
- **Inactive High (Low to High):** When the PWM output is inactive, the output remains high. When it is active, the first phase is low, and the second phase is high.
- **Inactive High (High to Low):** When the PWM output is inactive, the output remains high. When it is active, the first phase is high, and the second phase is low.

Mode



PWM Mode

Set the mode to one-shot or continuous. One-shot repeats the waveform for the number of periods given by repeat value and then, if interrupts are enabled, generates an interrupt at the end of operation. Continuous repeats the waveform infinitely and generates an interrupt, if enabled, every period.

Repeat Value

Set the repeat value if **PWM Mode** is set to **One-shot**. The PWM module outputs the waveform the specified number of times +1.

Interrupt enable

Enable the PWM module to generate an interrupt.

In one-shot mode, the PWM module generates an interrupt when number of periods given by **Repeat value** have been completed.

In continuous mode, the PWM module generates an interrupt during each period signaling that it is safe to set values for the subsequent waveform period and duty cycle.

References

For detailed information on the PWM module, see *TMS320DM643x DMP Pulse-Width Modulator (PWM) Peripheral User's Guide*, Literature Number SPRU995, available at the Texas Instruments Web site.

Purpose

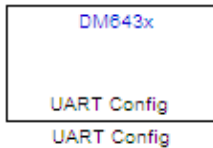
Configure DM643x UART for serial communication

Library

“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

“DM6437 EVM (dm6437evmlib)” on page 6-7

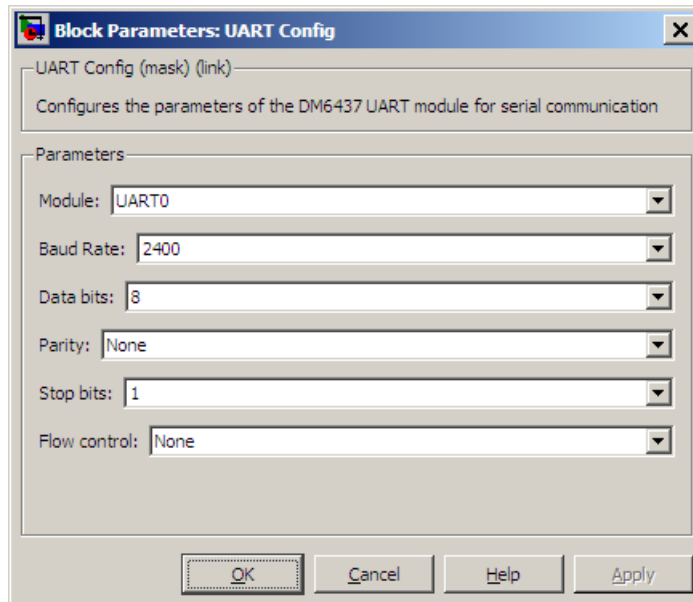
Description



Configure the serial communication parameters that are common to the transmit and receive elements of the DM643x UART module. If your model contains a DM643x UART Transmit block or a DM643x UART Receive block, it must also contain a DM643x UART Config block.

The UART module converts data between parallel and serial formats depending on whether it is transmitting or receiving data from external peripheral devices. Except for the **Module** parameter, configure all of the parameters in this block so they match the serial communication settings of the external peripheral devices.

Dialog Box



DM643x UART Config

Module

Select the UART module this block configures, UART0 or UART1. Your model can only contain one DM643x UART Config block per module.

Baud rate

Set the rate of signal modulations per second. Choose from 2400, 4800, 9600, 19200, 38400, 57600, or 115200.

Data bits

Set the number of data bits in the character frame, from 5, 6, 7, or 8.

Parity

Enable and configure parity error detection.

In parity error detection, the transmitter reserves a parity bit at the end of the character frame, adds the number of 1's in the data bits, and assigns a value to the parity bit. The receiver compares the number of 1's in the data bits with the value of the parity bit. If the two values don't match, the receiver signals the transmitter that an error has occurred.

- **None** disables parity error detection. The character frame does not include a parity bit.
- **Odd** enables parity error detection and reserves a parity bit at the end of the character frame. If the data bits contain an odd number of 1's, the method assigns a value of 0 to the parity bit.
- **Even** enables parity error detection and reserves a parity bit to the end of the character frame. If the data bits contain an even number of 1's, the method assigns a value of 0 to the parity bit.

Stop bits

Select 1 or 2.

Flow control

Select None or Hardware.

References

For detailed information on the UART module, see *TMS320DM643x DMP Universal Asynchronous Receiver/Transmitter (UART) User's Guide*, Literature Number: SPRU997, available at the Texas Instruments Web site.

See Also

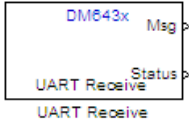
DM643x UART Receive, DM643x UART Transmit

DM643x UART Receive

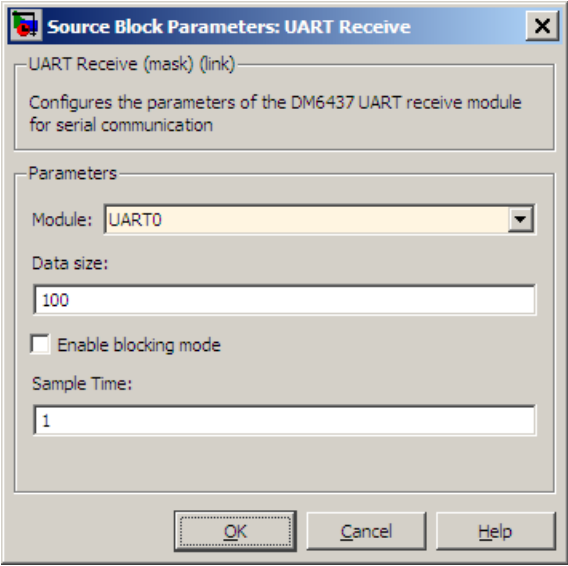
Purpose Configure receiver element of DM643x UART module for serial communication

Library “AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2
“DM6437 EVM (dm6437evmlib)” on page 6-7

Description Configure the serial communication parameters of the receiver element of the DM643x UART module. The receiver element converts data from external peripheral devices from serial to parallel format and passes it to the CPU. If your model contains a DM643x UART Receive block, it must also contain a DM643x UART Config block.



Dialog Box



Module Select the UART module this block configures, UART0 or UART1. Your model can only contain one DM643x UART Receive block per module. This parameter must also match the **Module** parameter in the DM643x UART Config block.

Data size

Set the data size, in bytes, of each transmission. Blocking mode uses this parameter to determine whether to generate an error.

Enable blocking mode

Enable this parameter to generate an error if the size of the last data transmission does not match the value of the **Data size** parameter. The DM643x UART Receive block sends the error message as a negative value on its **Status** output. If you disable **Enable blocking mode**, the block sends the number of bytes it received as a positive value on its **Status** output.

Sample time

Set the sample time for the block's input sampling. To execute this block asynchronously, set **Sample Time** to -1, and refer to "Asynchronous Scheduling" on page 2-25 for a discussion of block placement and other necessary settings.

References

For detailed information on the UART module, see *TMS320DM643x DMP Universal Asynchronous Receiver/Transmitter (UART) User's Guide*, Literature Number: SPRU997, available at the Texas Instruments Web site.

See Also

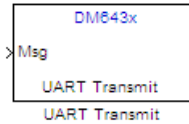
DM643x UART Config, DM643x UART Transmit

DM643x UART Transmit

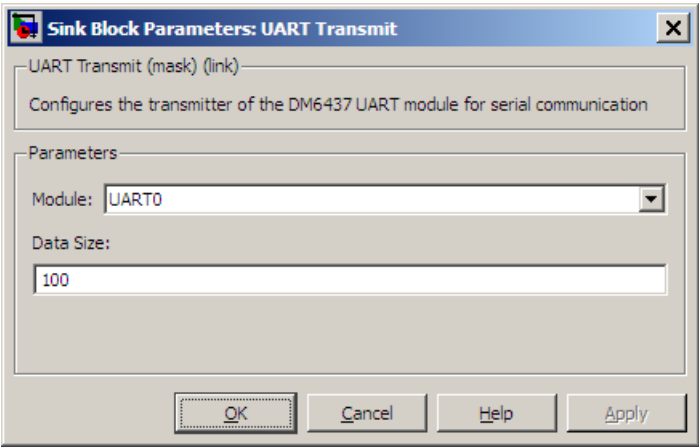
Purpose Configure transmitter element of DM643x UART module for serial communication

Library “AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2
“DM6437 EVM (dm6437evmlib)” on page 6-7

Description Configure the serial communication parameters of the transmitter element of the DM643x UART module. If your model contains a DM643x UART Receive block, it must also contain a DM643x UART Config block. The transmitter element converts parallel data from the CPU to a serial data format for output to external peripheral devices.



Dialog Box



Module Select the UART module this block configures, UART0 or UART1. Your model can only contain one DM643x UART Transmit block per module. This parameter must also match the Module parameter in the DM643x UART Config block.

Data size Set the number of bytes to send per transmission.

References

For detailed information on the UART module, see *TMS320DM643x DMP Universal Asynchronous Receiver/Transmitter (UART) User's Guide*, Literature Number: SPRU997, available at the Texas Instruments Web site.

See Also

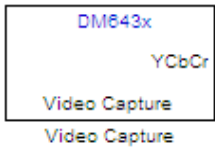
DM643x UART Config, DM643x UART Receive

DM643x Video Capture

Purpose Configure Video Processing Front End (VPFE) to capture REC656 or generic YCbCr 4:2:2 video

Library “AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

Description Configure the video processing front end (VPFE) to capture NTSC or PAL video.



Dialog Box **VPFE**

Video capture interface

Configure this parameter to match the format of the input signal using either the **REC656** or **Generic YCbCr-4:2:2** option. The **REC656** format is also known to as ITU-R BT.656 or CCIR-656 and comprises an 8-bit YCbCr 422 input signal. **Generic YCbCr-4:2:2** comprises an 8-bit signal with discrete horizontal (H) and vertical (VSYNC) signals, such as a computer monitor signal.

Data input mode

When **Video capture interface** is set to **Generic YCbCr-4:2:2**, set this parameter depending on the number of pins used by the physical interface. If the physical interfaces uses pins 0–8, select **8-bit**. If the physical interface uses pins 0–15, use **16-bit**. When you select **16-bit**, the lower 8 pins capture Y and the upper 8 pins capture the C (chroma) components.

For more information, refer to *Table 1. Interface Signals for Video Processing Front End* in the *TMS320DM643x DMP Video Processing Front End (VPFE) User's Guide*, Literature Number: SPRU977, available on the Texas Instruments Web site.

Scan mode

If you set **Video capture interface** to **Generic YCbCr-4:2:2**, set **Scan mode** to match the scan mode of the input signal, **Interlaced** or **Progressive**. Regardless of the setting, the block outputs an interleaved YCbCr 422 signal, which you can deinterleave using the C6000 Deinterleave block.

Note If you set **Scan mode** to **Interlaced**, verify that the Field ID signal is connected to the correct input pin for this video capture driver to work correctly.

Frame size

Define the size of the capture frame. You can use this parameter to capture the entire input frame or to capture just a portion of it.

DM643x Video Capture

The **Frame size** parameter values must be greater than zero and no greater than the size of the input frame. Enter the row and column dimensions of the capture frame in pixels. For example, entering [740, 480] sets the row width to 740 pixels, and the column height to 480 pixels.

Capture start pixel

Set the location of the capture frame relative to the display frame, using the upper-left corners of both frames (e.g., [0, 0]) as the point of reference. You can position the start pixel anywhere in the input frame. Enter the row and column dimensions of the Capture start pixel in pixels. For example, entering [10, 20] positions the upper-left corner of the capture frame at row 10, column 20 from the upper-left corner of the display frame.

The combination of the **Frame size** and **Capture start pixel** parameters may place the capture frame outside the display frame. If so, the portions of the capture frame that lie outside the display frame capture null video data (black screen) without generating an error.

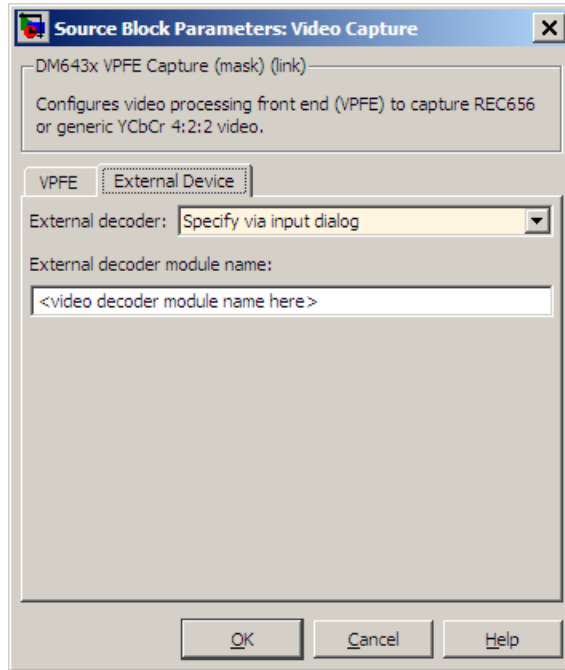
Sample time

Set the sampling rate of the video capture frame. Enter **Sample time** as a fraction of 1 over the sample rate per second. For example, to obtain a sample rate of 30 frames per second, enter 1/30.0. NTSC has a typical frame rate of 1/30, while PAL usually requires 1/25.

You can set this parameter to match the frame rate of the input signal, or you can use it to downsample the input signal. For example, sampling a 1/30 input at 1/15 halves the data throughput of the signal.

Setting the sample time to a different value from the input signal refresh rate may cause discontinuities in the video image. Avoid exceeding the sample rate of the input signal.

External Device



The **External Device** tab enables you to connect a video device with an external video decoder to the VPFE. When you specify the external coder, you create hookpoints in the VPFE driver initialization code for opening the external video decoder, starting the data output, and closing the external video decoder. The external decoder plugs into the following function pointers:

- EVD_Handle (*Open)()
- Int (*Close)(Ptr handle)
- Int (*Control)(Ptr handle, Uint32 Cmd, Ptr CmdArg)

DM643x Video Capture

For example, if you were to enter “PSP_VPFE_TVP5146” for **External decoder module name**, you would declare the following functions as shown:

```
// External device open function
EVD_Handle PSP_VPFE_TVP5146_Open(void);
// External device close function
Int PSP_VPFE_TVP5146_Close(EVD_Handle handle);
// External device control function
Int PSP_VPFE_TVP5146_Control(EVD_Handle handle, Uint32 Cmd, Ptr CmdArg);
```

The VPFE driver also assumes that a user structure named TVP5146_ConfigParams and a variable called PSP_VPFE_TVP5146_params exists to pass to the PSP_VPFE_TVP5146_Control function. In other words, there must be a declaration like the following:

```
typedef struct _PSP_VPFE_TVP5146_ConfigParams
{
    int dummy; // User defined fields
} PSP_VPFE_TVP5146_ConfigParams;
TVP5146_ConfigParams PSP_VPFE_TVP5146_params;
```

You must use the custom code interface to add the header file that declares function prototypes and the source files that contain the implementation of the _Open, *_Close and *_Control functions to the generated project. To see an example, download the Avnet S3ADSP DaVinci Evaluation Platform Support Package from <http://www.mathworks.com/matlabcentral/fileexchange/22191>, and open the model, avnet_test_dm6437evm.mdl. (Do not install the Avnet S3ADSP DaVinci Evaluation Platform Support Package. It is for R2008a only.)

External decoder

If your target is connected to a video device that outputs a RAW video signal and relies on the DM643x VPFE's built-in decoder, select **None**. If your target is connected to a video device with

a decoder that outputs REC656 or generic YCbCr-4:2:2, select **Specify via input dialog**.

External decoder module name

If you set the **External decoder** to Specify via input dialog, then enter a name for the external video decoder module name in this field.

See Also

DM643x Draw Rectangles, DM643x OSD, DM643x Video Display

References

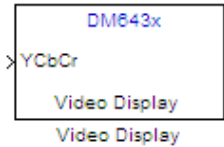
TMS320DM643x DMP Video Processing Front End (VPFE) User's Guide, Literature Number: SPRU977, available from the Texas Instruments Web site.

DM643x Video Display

Purpose Configure Video Processing Back End to display NTSC/PAL video

Library “DM6437 EVM (dm6437evmlib)” on page 6-7
“AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)” on page 6-2

Description This block configures the Video Processing Back End (VPBE) to display NTSC/PAL video.

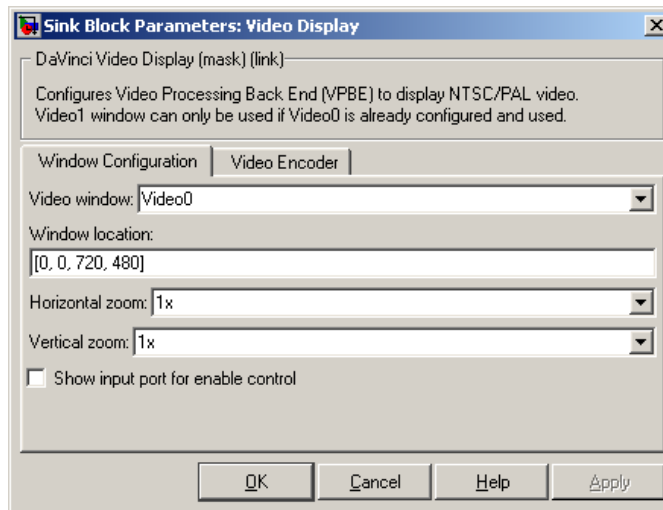


Dialog Box The block dialog box comprises multiple tabs:

- **Window Configuration** — Configure the video window, position, zoom, and whether to display the input port.
- **Video Encoder** — Configure the video display mode, analog video output, and horizontal or vertical expansion.

The dialog box images show all of the available parameters enabled. Some of the parameters shown do not appear until you select one or more other parameters.

Window Configuration



Video window

Create a video display window, **Video0** or **Video1**.

You must create a **Video0** display window before you can use the following video elements:

- a Video1 video display window from the DM643x Video Display block
- an on-screen display from the DM643x OSD block
- a video rectangle from the DM643x Draw Rectangles block

Window location source

Select the method for setting the location of the graphics display window. **Specify via dialog** creates the **Window location** field. **Specify via input port** creates an position input (Pos) on the OSD block which accepts the location of the window as data.

DM643x Video Display

Window location

This parameter appears when you set **Window location source** to **Specify via dialog**. Set the pixel width, height, and base coordinates. For example, the default values, [360, 240, 100, 100] set the width to 360 pixels, the height to 240 pixels, the base coordinates for x to 100 pixels, and the base coordinates for y to 100 pixels.

Note [0, 0], the origin of the coordinate system, is the located in the upper-left corner of the Video0 window.

Horizontal zoom

Set the horizontal magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the video display block.

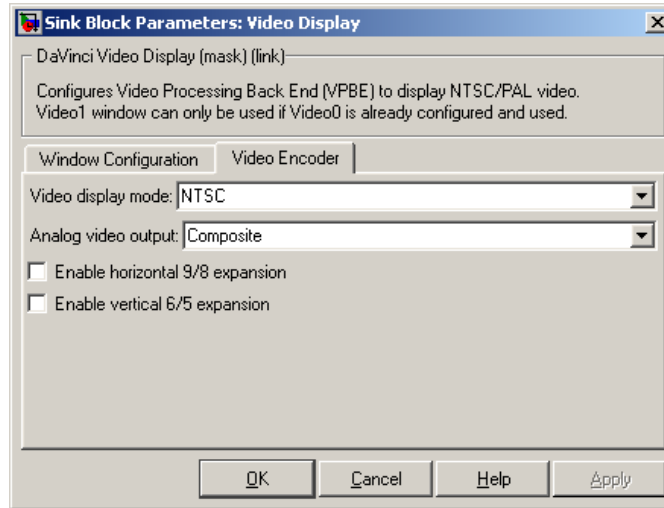
Vertical zoom

Set the vertical magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the video display block.

Show input port for enable control

Create an input port (**En**) to enable or disable the video display window.

Video Encoder Pane



Video display mode

Set the video format to **NTSC** , **PAL**, **HD 480p60**, or **HD 576p50**.

Analog video output

Set the output type to **Composite**, **S-video**, or **Component**.

Enable horizontal 9/8 expansion

Expands the image horizontally. Typically used to compensate for spatially compressed NTSC and PAL video signals. For example, use this setting to correct a 720 x 480 pixel NTSC analog video input that is displayed as a 640 x 480 pixel image.

Enable vertical 6/5 expansion

Expands the image vertically. Typically used to compensate for spatially compressed PAL video signals. For example, use this setting in combination with the **Enable horizontal 9/8 expansion** setting to correct a 720 x 576 pixel PAL analog video input that is displayed as a 640 x 480 pixel image.

DM643x Video Display

See Also

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture,
DM643x Video Capture

Purpose

Configure DSP peripherals to capture NTSC/PAL or HD video

Library

“DM648 EVM (dm648evmlib)” on page 6-9

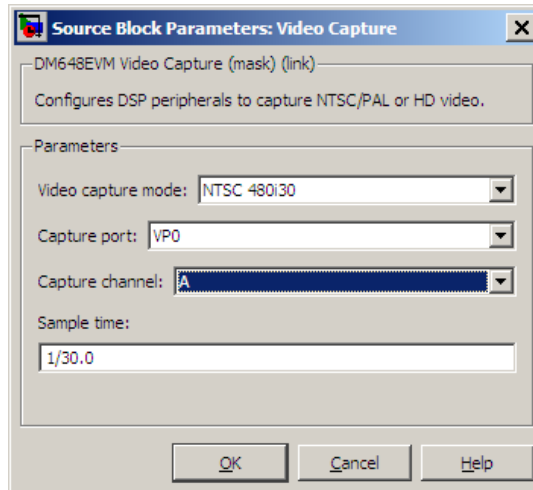
Description



This block configures the Video Processing Back End (VPBE) to capture NTSC, PAL, or HD video.

To capture multiple video data streams for applications such as multipicture displays, use multiple **Video capture** blocks. For NTSC and PAL, you can capture eight video streams by combining four **Capture ports** with two **Capture channels**. For HD, you can capture two video streams using two **Capture ports**.

Dialog



Video capture mode

Set the video format to **NTSC**, **PAL**, or **HD**. Each menu item gives the encoding type, the vertical lines of resolution, whether the scanning type is interlaced (i) or progressive (p), and the frame rate of the input. For example, the “NTSC 480i30” indicates NTSC encoding, 480 lines of vertical resolution, interlaced, and 30 frames per second.

DM648 EVM Video Capture

Capture port

Select the video input port. When you configure Video capture mode for an NTSC or PAL input, four capture ports become available. When you configure Video capture mode for an HD input, two capture ports become available. VP1 is not available in the list of capture ports because it is reserved for video display.

Capture channel

Two capture channels, A and B, are available for NTSC or PAL. **Capture channel** is not available when **Video capture mode** is configured for an HD input.

Sample time

Set the interval between samples in fractions of a second. This value defaults to 1/30.0, or one-thirtieth of a second. If the sample time does not match the frame rate of the video input, some irregularities may occur.

See Also

DM648 EVM Video Display

Purpose

Configure DSP peripherals to display NTSC, PAL, HD, or VESA video

Library

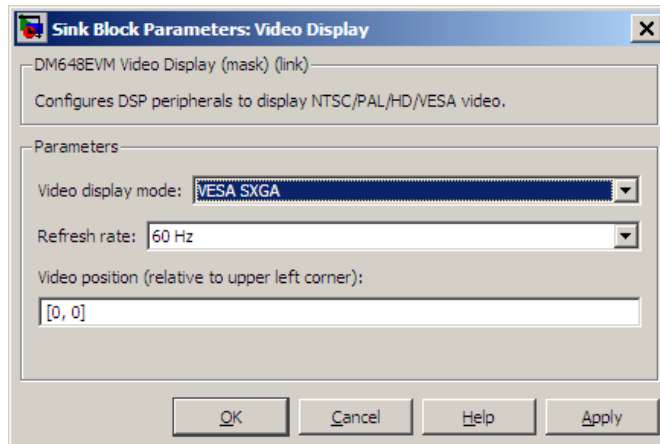
“DM648 EVM (dm648evmlib)” on page 6-9

Description



This block configures the Video Processing Back End (VPBE) to display NTSC/PAL/HD/VESA video. When sending the video output to a computer display, verify that the combination of the resolution of the **VESA** in **Video display mode** and the frequency in **Refresh rate** are valid settings for the monitor. Using unsupported combinations may permanently damage the computer display connected to a video output.

Dialog Box



Video display mode

Set the video display mode to **NTSC**, **PAL**, **HD**, or **VESA**. The **NTSC**, **PAL**, and **HD** menu items give the encoding type, the vertical lines of resolution, whether the scanning type is interlaced (i) or progressive (p), and the frame rate of the input. For example, the “NTSC 480i30” indicates NTSC encoding, 480 lines of vertical resolution, interlaced, and 30 frames per second. The **VESA** modes correspond to a range of standard computer display modes.

DM648 EVM Video Display

Refresh rate

When **Video display mode** is one of the VESA modes, set the refresh rate of the video output.

Video position

Position the upper-left corner of the video output in the video display by entering coordinates. The default coordinates, [0,0], correspond to the upper-left corner of the video display. Increasing the horizontal and vertical coordinates moves the video output to the right and down.

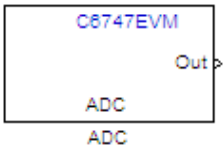
See Also

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

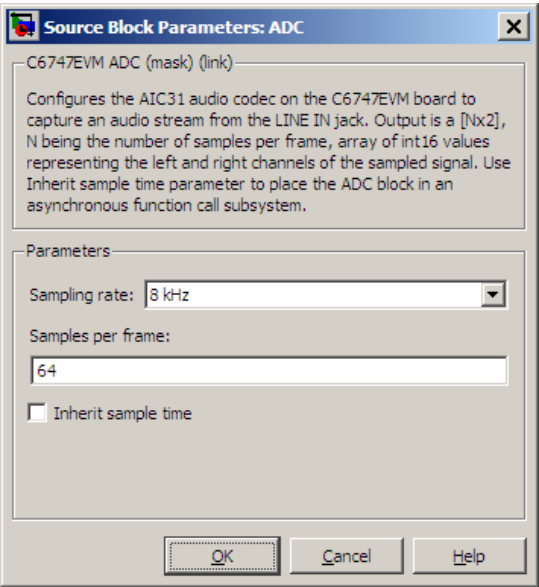
Purpose Capture audio stream from LINE IN jack

Library “C6747 EVM (c6747evmlib)” on page 6-5

Description Configures the AIC31 audio codec on the C6747EVM board to capture an audio stream from the LINE IN jack. Output is a [Nx2], N being the number of samples per frame, array of int16 values representing the left and right channels of the sampled signal. Use Inherit sample time parameter to place the ADC block in an asynchronous function call subsystem.



Dialog Box



Sampling rate Set the rate at which the analog-to-digital converter samples the analog input. A higher rate increases the resolution of the data the ADC outputs.

Samples per frame

Set the number of samples the ADC buffers internally before it sends the digitized signals, as a frame vector, to the next block in the model. This value defaults to 64 samples per frame. The frame rate depends on the sample rate and frame size. Thus, if you set Sampling Rate to 8 kHz, and Samples per frame to 64, the resulting frame rate is 125 frames per second ($8000/64 = 125$).

Inherit sample time

Select whether the block inherits the sample time from the model base rate or from the Simulink base rate. You can locate the Simulink base rate in the Solver options in Configuration Parameters. Selecting Inherit sample time directs the block to use the specified rate in model configuration. Entering -1 configures the block to accept the sample rate from the upstream Interrupt, Task, or Triggered Task blocks.

See Also

C6747EVM DAC

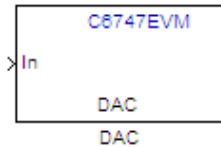
Purpose

Output audio on LINE OUT / HP OUT jacks

Library

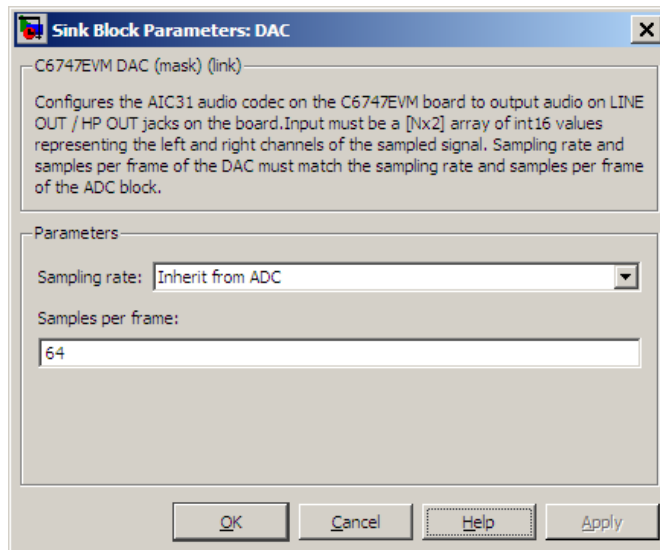
“C6747 EVM (c6747evmlib)” on page 6-5

Description



Configures the AIC31 audio codec on the C6747EVM board to output audio on LINE OUT / HP OUT jacks on the board. Input must be a [Nx2] array of int16 values representing the left and right channels of the sampled signal. Sampling rate and samples per frame of the DAC must match the sampling rate and samples per frame of the ADC block.

Dialog Box



Sampling rate

Set the rate at which the digital-to-analog converter receives each data sample. If your model contains an ADC block, select **Inherit from ADC**.

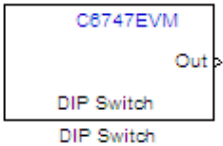
Samples per frame

Set the number of samples per data input frame. Match this value with the value of the block creating the data frames. This value defaults to 64 samples per frame.

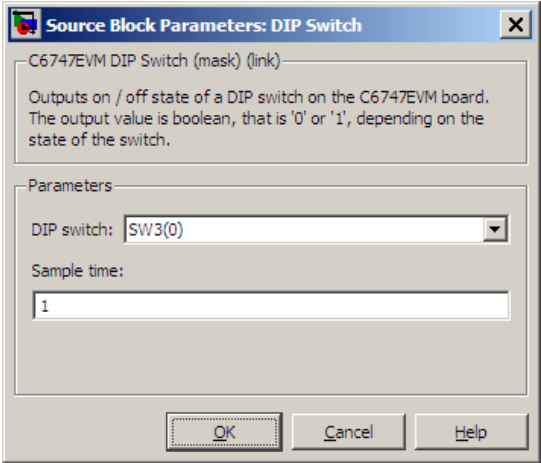
See Also

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

Purpose	Output DIP switch status
Library	“C6747 EVM (c6747evmlib)” on page 6-5
Description	Outputs on / off state of a DIP switch on the C6747EVM board. The output value is boolean, that is '0' or '1', depending on the state of the switch.



Dialog Box



- DIP Switch**
Select the switch, 0 through 3, from the SW3 bank of switches.
- Sample time**
Specify the time between samples of the signal in seconds. This value defaults to 1 second between samples.

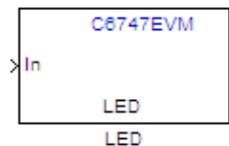
See Also DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

C6747EVM LED

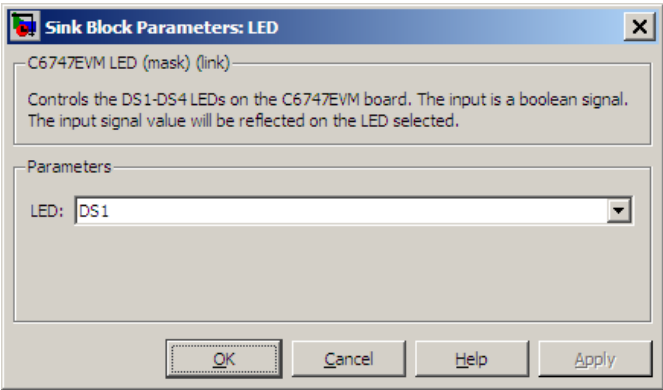
Purpose Control four on-board LEDs

Library “C6747 EVM (c6747evmlib)” on page 6-5

Description Controls the DS1-DS4 LEDs on the C6747EVM board. The input is a boolean signal. The input signal value will be reflected on the LED selected.



Dialog Box



LED Specify the number of the User LED that the Boolean input controls.

See Also DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

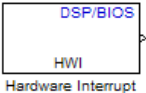
Purpose

Generate Interrupt Service Routine

Library

“DSP/BIOS (dsplib)” on page 6-9

Description



Creates an Interrupt Service Routine (ISR) that executes the task block or subsystem that is downstream from the block. ISRs are functions that the CPU executes in response to an external event.

Interrupt numbers for C6000 family processors range from 0 to 15, with 0 reserved for the reset ISR. The following table presents the set of interrupt numbers for the C6713 processor. For more detailed and specific information about interrupts, refer to Texas Instruments technical documentation for your target processor.

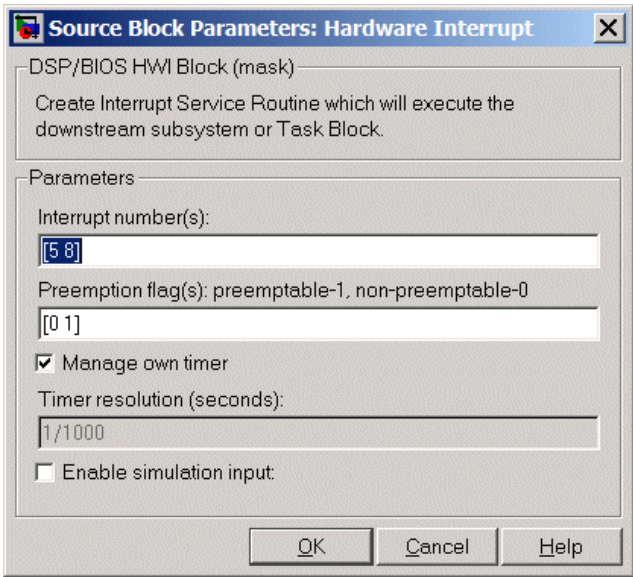
Interrupt Number	Default Event	Module
0	Reset	
1	NMI	
2	Reserved	
3	Reserved	
4	GPINT4	GPIO
5	GPINT5	GPIO
6	GPINT6	GPIO
7	GPINT7	GPIO
8	EDMAINT	EDMA
9	EMUDTDMA	Emulation
10	SDINT	EMIF
11	EMURTDXR	Emulation
12	EMURTDXTX	Emulation
13	DSPINT	HPI

DSP/BIOS Hardware Interrupt

Interrupt Number	Default Event	Module
14	TINT0	Timer 0
15	TINT1	Timer 1

In models, you usually follow this block with either a DSP/BIOS Task or DSP/BIOS Triggered Task block, or a subsystem function call block.

Dialog Box



Interrupt number(s)

Enter one or more integer values as a vector that represent interrupts. Interrupts have any value from 0, the highest priority to 15, lowest priority. As shown, enter the values enclosed in square brackets. For example, entering

[3 5 15]

results in three interrupt routines. [5 8] is the default entry, specifying two interrupts.

Preemption flag(s)

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value here, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 5 in **Interrupt numbers** is not preemptible and the priority 8 interrupt can be preempted.

Manage own timer

The ISR generated by the this block can manage its own time by reading time from the clock on the board. Selecting this option directs the ISR to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the ISR uses.

Timer resolution (seconds)

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

Enable simulation input

Selecting this option adds an input port to the block for simulating inputs in Simulink software. Connect interrupt simulation sources to the input. This option affects simulation only. It does not affect generated code.

DSP/BIOS Hardware Interrupt

See Also

DSP/BIOS Task, DSP/BIOS Triggered Task

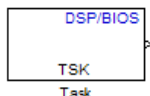
Purpose

Create task that runs as separate DSP/BIOS thread

Library

“DSP/BIOS (dspbioslib)” on page 6-9

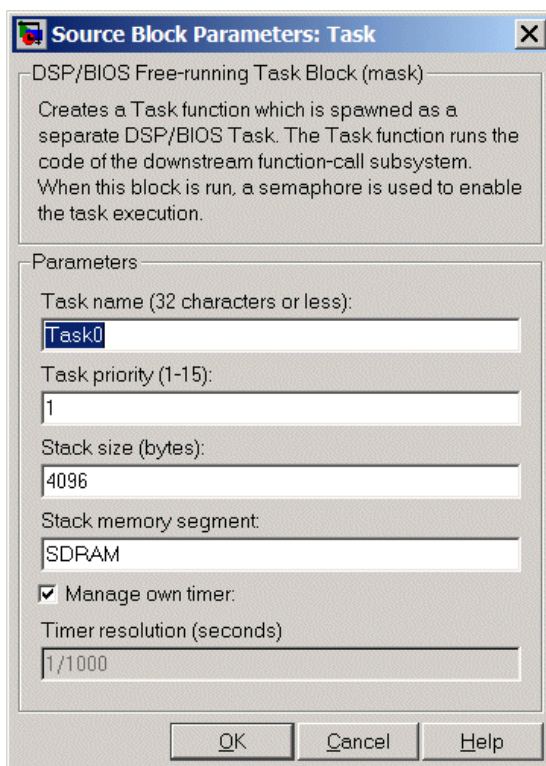
Description



Creates a free-running task that runs in response to an ISR and as a separate DSP/BIOS thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

Dialog Box



Source Block Parameters: Task

DSP/BIOS Free-running Task Block (mask)

Creates a Task function which is spawned as a separate DSP/BIOS Task. The Task function runs the code of the downstream function-call subsystem. When this block is run, a semaphore is used to enable the task execution.

Parameters

Task name (32 characters or less):

Task priority (1-15):

Stack size (bytes):

Stack memory segment:

☒ Manage own timer.

Timer resolution (seconds)

OK Cancel Help

DSP/BIOS Task

Task name (32 characters or less)

Creates a name for the task. Enter a string of up to 32 characters, including numbers and letters as needed. You cannot use the standard C reserved characters, such as / and : in the name.

Task priority (1-15)

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority.

Stack size (bytes)

Specify the size of the stack the task uses. The value defaults to 4096 bytes.

Stack memory segment

Specify where the stack resides in memory.

Manage own timer

This block can manage its own time by reading time from the clock on the board. Selecting this option directs the task/block to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the task uses.

Timer resolution (seconds)

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

See Also

DSP/BIOS Hardware Interrupt, DSP/BIOS Triggered Task

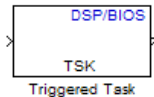
Purpose

Create asynchronously triggered task

Library

“DSP/BIOS (dspbioslib)” on page 6-9

Description



Creates a task that runs asynchronously in response to an ISR and as a separate DSP/BIOS thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

Dialog Box

Function Block Parameters: Triggered Task

DSP/BIOS Triggered Task Block (mask)

Creates a Task function which is spawned as a separate DSP/BIOS Task. The Task function runs the code of the downstream function-call subsystem. When this block is run, a semaphore is used to enable the task execution.

Parameters

Task name (32 characters or less):

Task priority (1-15):

Stack size (bytes):

Stack memory segment:

☒ Synchronize the data transfer of this task with the caller task

OK Cancel Help Apply

Task name (32 characters or less)

Creates a name for the task. Enter a string of up to 32 characters, including numbers and letters as needed. You cannot use the standard C reserved characters, such as / or : in the name.

DSP/BIOS Triggered Task

Task priority (1-15)

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority, unless the preemptible flag (**Preemption flag** option on the C5000/C6000 Hardware Interrupt block) prevents preempting the task.

Stack size (bytes)

Specify the size of the stack the task uses. The value defaults to 4096 bytes. Take care to set the stack size as large as necessary. If the task uses more than the allotted space it can write into other memory areas with unintended results.

Stack memory segment

Specify where the stack resides in memory by specifying the memory segment. Additional information about DSP/BIOS memory segments also appears in the Target Preferences block in the model.

Synchronize data transfer of this task with caller task

Specify whether this task should synchronize data transfer with the calling task. Select this option to enable synchronization. Clearing this option enables the **Timer resolution** option.

Timer resolution

When you direct the block not to synchronize data with the calling task (by clearing **Synchronize data transfer of this task with caller task**), **Timer resolution** reports the resolution of the timer. **Timer resolution** is a read-only parameter. You cannot change the value.

See Also

DSP/BIOS Hardware Interrupt, DSP/BIOS Task

Hardware Issues

- “Configuring the DesignT DSK-91C111 to Use TCP/IP and UDP” on page A-2
- “Requirements for the DM642 EVM” on page A-3
- “Installing and Configuring the Avnet Board Support Library” on page A-6
- “Continuing Issues with Target Support Package Software” on page A-9

Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP

Specific evaluation boards that don't have a build-in Ethernet ports accept the D.signT DSK-91C111 daughter card with the required Texas Instruments TMS320C6000 TCP/IP Stack. To use the D.signT DSK-91C111, change the position of solder point jumper JPINTPOL. Set the jumper to the "b" position from the default "a" position. Refer to your TI TCP/IP Stack User's Guide documentation for additional information about configuring the daughter card.

Requirements for the DM642 EVM

In this section...
“Identifying Your DM642 EVM Board Version” on page A-3
“Installing Third-party Software” on page A-3
“Configuring the Target Preferences Block for Your DM642 EVM” on page A-4
“Configuring the DM642 EVM Video ADC Block” on page A-5

This section provides details about using both the DM642 EVM hardware target and the simulator.

Identifying Your DM642 EVM Board Version

Spectrum Digital has released three versions of the DM642 EVM board:

- **Version 1** — Original board with 600 MHz DM642, Philips SAA7115 video decoders. ASSY 506840 Rev. D on back of board, 50 MHz oscillator.
- **Version 2** — Original board revised to use 720 MHz DM642, Philips SAA7115 video decoders. ASSY 506840 Rev. D on back of board, 60 MHz oscillator.
- **Version 3** — Revised board with 720 MHz DM642, TI TVP5146/5150 video decoders and HD filters. ASSY 507340 Rev. B on back of board, 60 MHz oscillator.

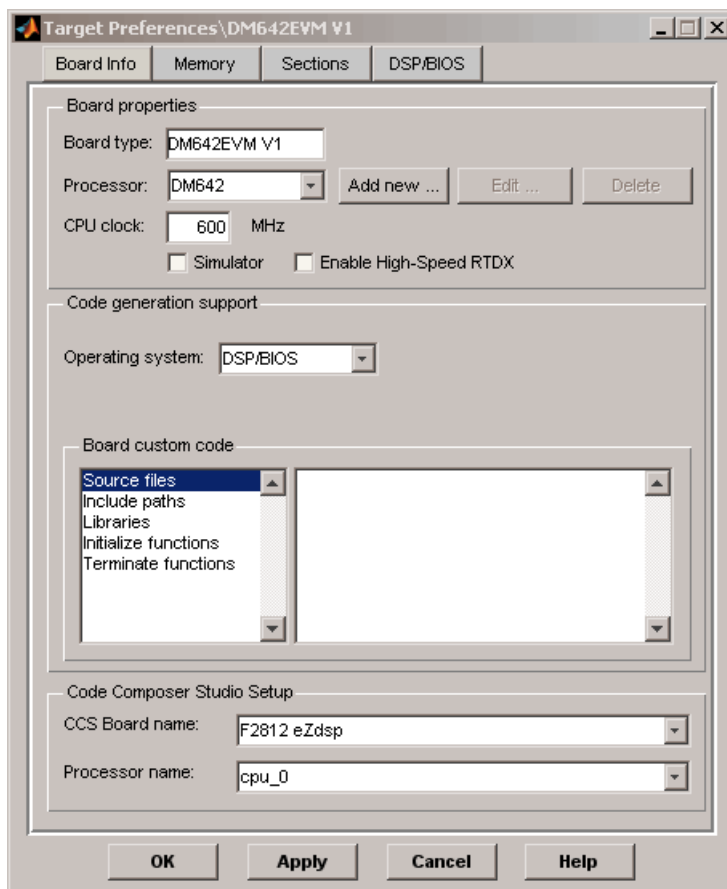
To determine the board version, consult the documentation provided with your board, or refer to the ASSY number located on the bottom surface of the board.

Installing Third-party Software

After determining the board version, install the *supported* versions of the third-party software for that board version. See the “System Requirements” on page 1-7 for the Target Support Package software.

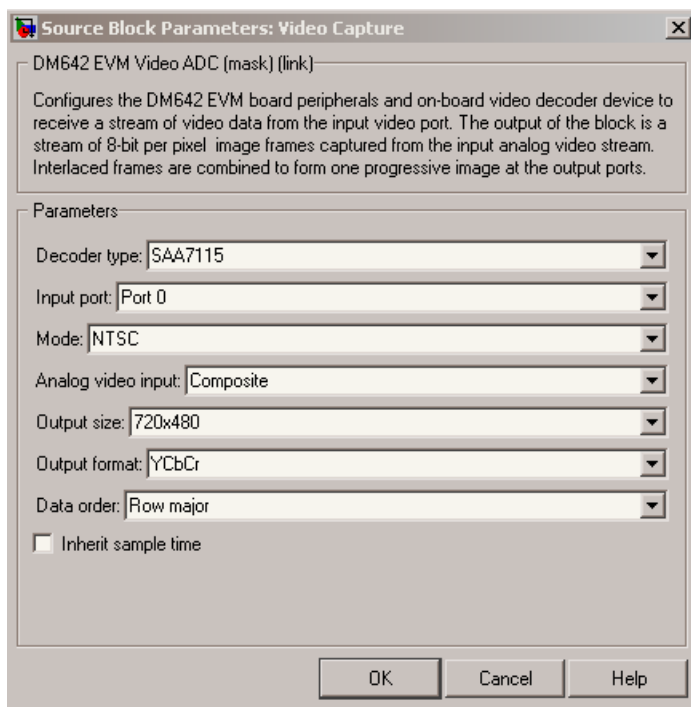
Configuring the Target Preferences Block for Your DM642 EVM

When you use the DM642EVM V1, V2, and V3 Target Preferences block, make sure that you enter the CPU clock speed that matches the CPU clock on your board. The figure below shows the correct setting of **600** for Version 1 boards in **CPU clock speed (MHz)**. For Version 2 and 3 boards, change the clock speed to 720.



Configuring the DM642 EVM Video ADC Block

If you have a DM642 EVM Version 2 or 3 board, make sure that you have the updated video drivers in your CCS IDE installation folder and that you select the correct decoder type TVP5146 when you use DM642 EVM Video ADC blocks as shown in the following figure.



Installing and Configuring the Avnet Board Support Library

In this section...
“Preface” on page A-6
“Installing the Avnet Board Support Library” on page A-6
“Setting the MATLAB Environment” on page A-6
“For Spectrum Digital DM6437EVM Users” on page A-7
“Verifying Your Installation ” on page A-8

Preface

The Avnet S3ADSP DaVinci evaluation platform is designed for joint software and hardware design. It brings the Texas Instruments TMS320DM6437 DSP and Xilinx Sparta-3A FPGA together. This chapter provides an overview of the board, and instructions for installing, configuring, and using the Avnet S3ADSP DM6437.

Installing the Avnet Board Support Library

Download and install the current Avnet Board Support Package for Simulink (Avnet BSL), available from the Avnet Web site, www.avnet.com. Doing so creates environment variables that the Target Support Package software uses to locate files in the Avnet BSP.

Make a note of the installation folder for the Avnet BSL.

Setting the MATLAB Environment

The Target Support Package software uses environment variables to locate files in the Avnet BSP.

The MathWorks utility, `setTgtEnv.m`, automatically maps the following environment variables (where <Avnet BSL> is the Avnet BSL installation folder):

- PSP_EVMDM6437_INSTALLDIR: must be mapped to “<Avnet BSL>\psp”
- CSLR_DM6437_INSTALLDIR: must be mapped to “<Avnet BSL>\psp\pspdriers\soc\dm6437\dsp\inc”
- NDK_INSTALL_DIR: must be mapped to “<Avnet BSL>\ndk”

Run setTgtEnv by entering the following command at the MATLAB command prompt: `setTgtEnv('avnet_s3adsp_dm6437')`

If you installed the Avnet BSL prior to installing the MathWorks BSL, the utility detects the AVNET_S3ADSP_DM6437_INSTALLDIR environment variable created by the Avnet BSL installer. It will automatically set the environment variables above based on the path stored in the AVNET_S3ADSP_DM6437_INSTALLDIR environment variable. On a successful run, you should see the following messages printed on the MATLAB command window:

```
Setting environment variable "PSP_EVMDM6437_INSTALLDIR" to  
C:\avnet_s3adsp_dm6437_1_06\psp"
```

```
Setting environment variable "CSLR_DM6437_INSTALLDIR" to  
C:\avnet_s3adsp_dm6437_1_06\psp\pspdriers\soc\dm6437\dsp\inc"
```

```
Setting environment variable "NDK_INSTALL_DIR" to  
C:\avnet_s3adsp_dm6437_1_06\ndk"
```

If automatic mapping fails for any reason, the script will prompt you to browse for the “avnet_s3adsp_dm6437_version.txt” file stored in the top-level Avnet BSL installation folder. If so, browse for the file and click the **Open** button. This will set the required environment variables.

For Spectrum Digital DM6437EVM Users

If you have a Spectrum Digital DM6437EVM board together with an Avnet S3ADSP DM6437 board, setting environment for the Avnet board as explained in section 2.3 will override DM6437EVM environment setup. To revert back to DM6437EVM environment after using Avnet board, execute the following at the MATLAB command prompt: `setTgtEnv('dm6437evm')`

Follow the instructions printed on the MATLAB command window to complete environment configuration. To go back and forth between DM6437EVM environment and Avnet S3ADSP DM6437 environment, use the `setTgtEnv` script with the appropriate platform name specified as the argument.

Verifying Your Installation

Open the Avnet S3ADSP Board Support Library by entering the following command at the MATLAB command prompt: `avnet_s3adsp_dm6437` This opens the **Avnet Spartan-3A DSP DaVinci Evaluation Platform Board Support Library**. You have completed installing and configuring the MathWorks and Avnet Board Support Libraries. You are ready to start using the Avnet S3ADSP DaVinci evaluation platform.

Continuing Issues with Target Support Package Software

This section details some target operations that you should know about as you use Target Support Package software.

In this section...
“Setting the Clock Speed on the C6713 DSK” on page A-9
“Simulink Stop Block Works Differently When Not Using DSP/BIOS Features” on page A-10
“Installing Third-Party Target Support Packages” on page A-10

Setting the Clock Speed on the C6713 DSK

The C6713DSK PLL is not automatically set to the correct CPU Clock frequency when you try to target the board. When you power-up your DSK, it runs at a clock speed of 50 MHz. However, the C6713 is capable of running at 225 MHz.

If you generate code incorporating the DSP/BIOS real-time operating system, the PLL is automatically configured for you at run-time to use the correct clock speed. If you are not using DSP/BIOS in your project, you must manually configure the PLL to the correct clock rate before running your code.

Setting the PLL to Drive the CPU at 225 MHz

To set the C6713 DSK PLL to drive the CPU at 225 MHz, perform the following steps. Be sure you have defined your GEL file for your DSK in the Setup Utility for CCS IDE.

- 1 Launch Code Composer Studio.
- 2 Open your C6713 DSK project with the GEL file.
- 3 Select **GEL > Resets > InitPLL** from the menu bar in CCS IDE.

To make this happen whenever you open Code Composer Studio to use your C6713 DSK, edit the file `\ti\cc\gel\ds6713.gel`. Add the following command to the `StartUp()` function:

```
init_pll();
```

This tells the GEL file to initialize the PLL to operate at 225 MHz.

On the DM642 EVM, ADC-DAC Loopback Does Not Display An RGB Image Correctly After Power-Up

When you set up the DM642 EVM to use loopback from the ADC to the DAC, the DAC block does not reproduce the captured image correctly immediately after you power up the board. Colors in the image are not shown correctly.

To get a clean image, reload the program to the target and run the program again. This also happens with the examples Texas Instruments ships with the DM642 EVM product.

Simulink Stop Block Works Differently When Not Using DSP/BIOS Features

If you are using the Simulink Stop block in your model, but you are not using DSP/BIOS features, your model might take longer to stop when it is running on the target than if you are using DSP/BIOS.

The condition the model uses to detect the stop processing flag is different when you do not use DSP/BIOS. The result is that the model may not detect and respond to the flag as promptly, taking longer to stop the running model on the target.

Installing Third-Party Target Support Packages

For a list of required third-party target support packages, with version numbers, see the Target Support Package System Requirements page at <http://www.mathworks.com/products/target-package/requirements.html>.

When you install any of the third-party target support packages listed below, perform a default installation using the installation path provided for that package and perform any additional steps given.

This documentation uses placeholders for portions of the installation path that may vary by software version or environment. Please replace the placeholders with the correct path information for your

software environment. For example, if the CCS IDE installation path is `C:\CCStudio_v3.3`, then enter `C:\CCStudio_v3.3\boards\evmdm642` instead of `<CCStudio_vn.n>\boards\evmdm642`.

Placeholders:

- `<CCStudio_vn.n>` — The installation path for Code Composer Studio
- `<n.n>` — Version-specific path information

Note If you do not use the installation paths provided, update the **Libraries** and **Include paths** parameters in the Target Preferences and C6000 IP Config blocks of the Target Support Package™ software with the correct paths. Otherwise, the software produces error messages when you attempt to generate code.

DM642EVM Version 3 Board

- Spectrum Digital EVMDM642 Board Support Package — `<CCSvn.n>\boards\evmdm642`
- TI's Network Developer's Kit (NDK) — `<CCSvn.n>\C6000\NDK`

DM642EVM Version 1 & 2 Boards

- Spectrum Digital EVMDM642 Board Support Package — `<CCSvn.n>\boards\evmdm642`
- Device Driver Developer's Kit (DDK) — `<CCSvn.n>\ddk`
- TI's Network Developer's Kit (NDK) — `<CCSvn.n>\C6000\NDK`

DM6437EVM

- Spectrum Digital DM6437EVM DVSDK RTM — Install anywhere. TI recommends using the root path of your main drive. For example, `C:\dvsdk_<n.n>`

Also, set the following environment variables, replacing DVSK with the DVSDK installation path (e.g., `C:\dvsdk_<n.n>`).

The first time you generate code, the Target Support Package™ software prompts you to locate specific files in the DVSDK folders and creates environment variables mapped to the location of required folders. For example, the application creates an environment variable called `CSLR_DM6437_INSTALLDIR` for the path of the Register Layer Chip Support Library.

C6455DSK

Spectrum Digital DSK6455/EVM6455 Target Content Package — `<CCSvn.n>\boards\dsk6455_v<n.n>`

Network Developer's Kit NDK — `<CCSvn.n>\C6000\NDK`

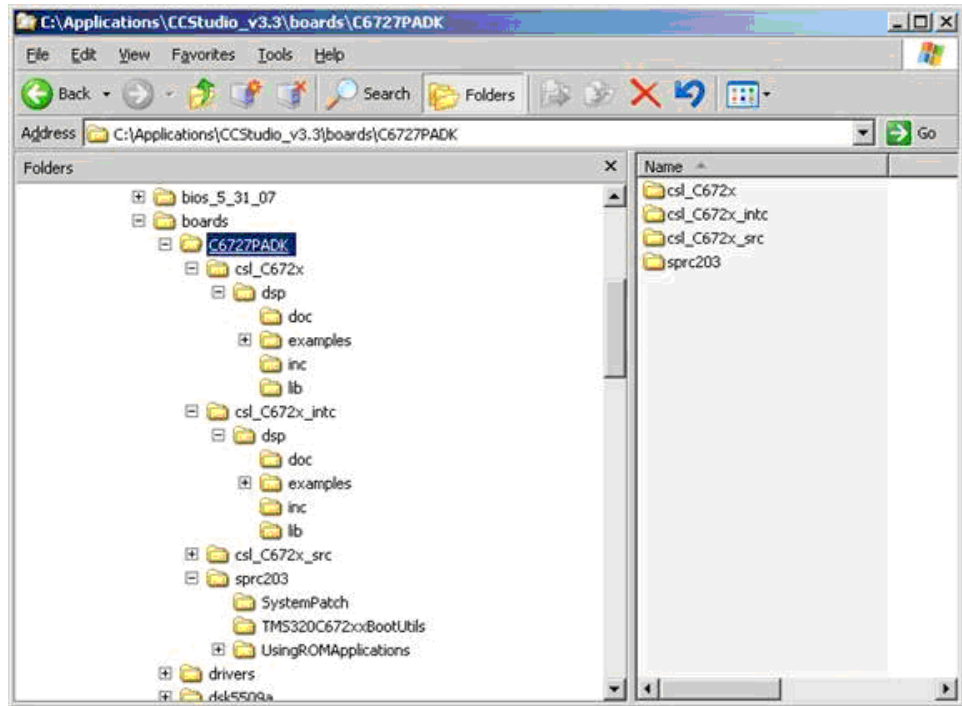
C6727PADK

Lyrtech's PADK Software — Install anywhere.

TI's C672x Chip Support Libraries (CSL) — Extract all three C672x CSL components from **sprc223.zip** to `<CCSvn.n>\boards\C6727PADK`.

TI's System Patch Code, FastRts(V<n.n>)/DSPLIB (V<n.n>) — `<CCSvn.n>\boards\C6727PADK\sprc203`

After installation, the path structure for the C672x CSL libraries should resemble the following figure:



The PADK Software installer automatically sets the PADK_DIR environment variable with the correct installation path.

The first time you generate code, the Target Support Package™ software prompts you to locate the following files under <CCSvn.n>\boards\C6727PADK\ and sets the environment variables accordingly:

- \$(CSL_C672x_INSTALLDIR)\lib\csl_C6727.lib
- \$(CSL_C672x_INTC_INSTALLDIR)\lib\csl_C672x_intc.lib
- \$(SYSPATCH_C672x_INSTALLDIR)\applySystemPatch.obj

You have completed installation of the third-party target support packages.

A

- Archive_library 2-37
- asynchronous scheduling 2-23
- Avnet Spartan 3-A Video Capture 7-262
- Avnet Spartan-3A blocks 6-2

B

- block limitations using model reference 2-38
- Block Processing block 7-2
- block recommendations 2-46
- blocks
 - Avnet Spartan-3A 6-2
 - C62x 6-10
 - C6747 6-5
 - DM642 6-7
 - DM6437 6-7
 - DM643x 6-2
 - DM648 6-9
 - use in target models 2-46
- blocks to avoid in models 2-46
- build folder
 - contents of 2-56
 - naming convention 2-50
- building models
 - use C62x DSP Library blocks 4-10

C

- C6000 Deinterleave 7-12
- C6000 EDMA block 7-13
- C6000 Interleave 7-22
- C6000 IP Config block 7-24
- C6000 Library
 - DM643x UART Config
 - Host side 7-255
- C6000 model reference 2-35
- C6000 Target
 - targeting Code Composer Studio 2-63
- C6000 TCP/IP Receive block 7-30

- C6000 TCP/IP Send block 7-36
- C6000 UDP Receive block 7-39
- C6000 UDP Send block 7-43
- C62x Autocorrelation block 7-46
- C62x Bit Reverse block 7-48
- C62x Block Exponent block 7-50
- C62x blocks 6-10
- C62x Complex FIR block 7-51
- C62x Convert Floating-Point to Q.15 block 7-53
- C62x Convert Q.15 to Floating-Point block 7-54
- C62x DSP Library blocks
 - building models 4-10
 - choosing blocks to optimize code 4-11
 - common characteristics 4-4
 - Q format notation 4-6
 - using source and sink blocks 4-11
- C62x FFT block 7-55
- C62x General Real FIR block 7-57
- C62x LMS Adaptive Filter block 7-59
- C62x Matrix Multiplication block 7-63
- C62x Matrix Transpose block 7-67
- C62x Radix-2 FFT block 7-68
- C62x Radix-2 IFFT block 7-70
- C62x Radix-4 Real FIR block 7-72
- C62x Radix-8 Real FIR block 7-74
- C62x Real Forward Lattice All-Pole IIR block 7-76
- C62x Real IIR block 7-78
- C62x Reciprocal block 7-81
- C62x Symmetric Real FIR block 7-82
- C62x Vector Dot Product block 7-87
- C62x Vector Maximum Index block 7-88
- C62x Vector Maximum Value block 7-89
- C62x Vector Minimum Value block 7-90
- C62x Vector Multiply block 7-91
- C62x Vector Negate block 7-92
- C62x Vector Sum of Squares block 7-93
- C62x Weighted Vector Sum block 7-94
- C6416 DSK ADC block 7-96
- C6416 DSK DAC block 7-100

- C6416 DSK DIP Switch block 7-103
 - C6416 DSK LED block 7-108
 - C6416 DSK Reset block 7-110
 - C6455 DSK ADC block 7-111
 - C6455 DSK DAC block 7-113
 - C6455 DSK DIP block 7-114
 - C6455 DSK LED block 7-116
 - C6455 DSK SRIO Config block 7-117
 - C6455 DSK SRIO Receive block 7-120
 - C6455 DSK SRIO Transmit block 7-127
 - C64x Autocorrelation block 7-131
 - C64x Bit Reverse block 7-133
 - C64x Block Exponent block 7-135
 - C64x Complex FIR block 7-136
 - C64x Convert Floating-Point to Q.15 block 7-138
 - C64x Convert Q.15 to Floating-Point block 7-139
 - C64x FFT block 7-140
 - C64x General Real FIR block 7-142
 - C64x LMS Adaptive Filter block 7-144
 - C64x Matrix Multiplication block 7-148
 - C64x Matrix Transpose block 7-152
 - C64x Radix-2 FFT block 7-153
 - C64x Radix-2 IFFT block 7-155
 - C64x Radix-4 Real FIR block 7-157
 - C64x Radix-8 Real FIR block 7-159
 - C64x Real Forward Lattice All-Pole IIR block 7-161
 - C64x Real IIR block 7-163
 - C64x Reciprocal block 7-166
 - C64x Symmetric Real FIR block 7-167
 - C64x Vector Dot Product block 7-172
 - C64x Vector Maximum Index block 7-173
 - C64x Vector Maximum Value block 7-174
 - C64x Vector Minimum Value block 7-175
 - C64x Vector Multiply block 7-176
 - C64x Vector Negate block 7-177
 - C64x Vector Sum of Squares block 7-178
 - C64x Weighted Vector Sum block 7-179
 - C6713 DSK
 - confirming proper configuration 2-44
 - start/stop models 2-42 2-61
 - tutorial about multirate applications 2-48
 - C6713 DSK ADC block 7-181
 - C6713 DSK blocks
 - tutorial 2-48
 - C6713 DSK DAC block 7-186
 - C6713 DSK DIP Switch block 7-188
 - C6713 DSK folders
 - build 2-49
 - working 2-49
 - C6713 DSK LED block 7-193
 - C6713 DSK Reset block 7-195
 - C6747 blocks 6-5
 - C6747EVM ADC 7-277
 - C6747EVM DAC 7-279
 - C6747EVM DIP Switch 7-281
 - C6747EVM LED 7-282
 - CAN
 - timing parameters
 - Bitrate 5-2
 - CCS IDE
 - create projects for the IDE 2-63
 - Code Composer Studio 2-63
 - configure your C6713 DSK for Target Support Package™ 2-59
 - confirm your C6713 DSK configuration 2-44
 - convert data types 4-10
 - CPU Timer block 7-196
 - custom C6000 target
 - about 2-69
 - preferences block 2-69
 - setup 2-69
 - custom hardware guidelines 2-65
 - custom hardware, target 2-65
- D**
- DM642 blocks 6-7
 - DM642 EVM Audio ADC block 7-198

- DM642 EVM Audio DAC block 7-201
- DM642 EVM FPGA GPIO Read block 7-203
- DM642 EVM FPGA GPIO Write block 7-205
- DM642 EVM LED block 7-221
- DM642 EVM Reset block 7-226
- DM642 EVM Video ADC block 7-207
- DM642 EVM Video DAC block 7-216
- DM642 EVM Video Port block 7-222
- DM6437 blocks 6-7
- DM6437 EVM ADC 7-227
- DM6437 EVM DAC 7-229
- DM6437 EVM DIP 7-230
- DM6437 EVM LED 7-231
- DM6437 EVM Video Capture 7-232
- DM643x blocks 6-2
- DM643x CAN Receive 7-234
- DM643x CAN Setup 7-237
- DM643x CAN Transmit 7-240
- DM643x Draw Rectangles 7-242
- DM643x OSD 7-244
- DM643x PWM 7-249
- DM643x UART Config
 - Host side 7-255
- DM643x UART Receive block 7-258
- DM643x UART Transmit block 7-260
- DM643x Video Display 7-268
- DM648 blocks 6-9
- DM648 EVM Video Capture 7-273
- DM648 EVM Video Display 7-275
- DSP/BIOS
 - added files 3-10
 - files removed from project 3-10
 - to enable 3-25
- DSP/BIOS Hardware Interrupt block 7-283
- DSP/BIOS Task block 7-287
- DSP/BIOS Triggered Task block 7-289
- DSP/BIOS, enabling 3-25

E

- enabling DSP/BIOS 3-25
- execution in timer-based models 2-24

F

- files added to DSP/BIOS project 3-10
- files removed from DSP/BIOS projects 3-10
- fixed-point numbers 4-5
 - signed 4-6

H

- hardware 1-7
- hardware, custom 2-65
- hardware, guidelines for using custom boards 2-65

I

- inaccurate profile information 3-14
- initialized memory 2-69
- installing software 1-7

M

- management, memory 2-69
- map memory 2-68
- map, memory 2-68
- memory
 - initialized 2-69
 - management 2-69
 - map 2-68
 - section 2-68
 - segment 2-68
 - uninitialized 2-69
- memory maps 2-68
- messages
 - DM643x 7-235
- model execution 2-23
- model reference 2-35

- about 2-35
- Archive__library 2-37
- block limitations 2-38
- modelreferencecompliant flag 2-38
- setting build action 2-37
- target preferences blocks 2-38
- using 2-37
- model schedulers 2-23
- modelreferencecompliant flag 2-38

O

- optimize code 4-11

P

- profile generated code 3-11
- profile report
 - about 3-11
 - correcting inaccurate profile information 3-14
 - CPU clock speed 3-20
 - maximum percent of interrupt interval (Max %) 3-20 to 3-21
 - maximum time spent in this subsystem per interrupt (Max time) 3-20
 - number of interrupts counted 3-20
 - profiling subsystems 3-12
 - reading 3-18
 - sample 3-18
 - STS objects 3-21
 - timing details 3-13
 - to generate 3-21
- projects, create for CCS IDE 2-63

Q

- Q format notation 4-6

R

- run the DSK confidence test 2-44

S

- sample time
 - DM643x 7-235
- section, memory 2-68
- segment, memory 2-68
- select blocks for models 2-46
- signed fixed-point numbers 4-6
- simulator
 - device cycle accurate 2-5
 - use simulators for development 2-5
 - use with DSP/BIOS 2-5
- simulators, about 2-5
- source and sink blocks 4-11
- supported hardware 1-7
- synchronous scheduling 2-24
- system requirements 1-7

T

- table of blocks to avoid in models 2-46
- target Code Composer Studio 2-63
- target custom hardware 2-65
- target preferences blocks in referenced models 2-38
- Target Support Package™
 - about 1-2
 - create Simulink® model for targeting 2-45
 - expected background for use 1-3
 - information for new users 1-3
 - use C6713 DSK blocks 2-19
- timer-based models, execution 2-24
- timer-based scheduler 2-24
- timing 2-23
- tutorial for C6713 DSK blocks 2-48

U

uninitialized memory 2-69
use blocks for the C6713 DSK 2-48
use C62x and C64x DSP Library blocks 4-1
use C6713 DSK blocks 2-48

W

working folder 2-49