

# Embedded Target for the TI TMS320C6000™ DSP Platform For Use with Real-Time Workshop®

- Modeling
- Simulation
- Implementation

## How to Contact The MathWorks:



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information  
508-647-7000 Phone



508-647-7001 Fax



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

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

### *Embedded Target for TI TMS320C6000 DSP Platform User's Guide*

© COPYRIGHT 2002–2006 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc. Other product or brand names are 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](http://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)



## What Is Embedded Target for TI C6000 DSP?

### 1

<b>Introducing Embedded Target for the TI TMS320C6000 DSP Platform</b> .....	<b>1-2</b>
Suitable Applications .....	1-3
<b>About Embedded Target for C6000 DSP</b> .....	<b>1-4</b>
<b>Using This Guide</b> .....	<b>1-5</b>
Expected Background .....	1-5
<b>Configuration Information</b> .....	<b>1-7</b>
<b>Getting Started</b> .....	<b>1-9</b>
Platform Requirements—Hardware and Operating System ..	1-9

## Targeting C6000 DSP Hardware

### 2

<b>Overview</b> .....	<b>2-3</b>
About the Tutorials .....	2-3
<b>TI C6000 and Code Composer Studio IDE</b> .....	<b>2-4</b>
Supported Boards and Simulators .....	2-4
Typical Hardware Setup for C6711 DSK in Models .....	2-6
Typical Hardware Setup for RTDX in Models .....	2-8
<b>Targeting Tutorial — Single Rate Application</b> .....	<b>2-10</b>
Specifying Configuration Parameters for Your Model .....	2-17
<b>Using the C6000lib Blockset</b> .....	<b>2-21</b>
Configuring ADC Blocks .....	2-32
Configuring DAC Blocks .....	2-36

Configuring LED Blocks .....	2-38
Using the Overrun Indicator Feature .....	2-38
Configuring Reset Blocks .....	2-40
Configuring Target Preferences Blocks .....	2-40
Board Info Pane .....	2-45
Memory Pane .....	2-47
Sections Pane .....	2-52
DSP/BIOS Pane .....	2-58
Configuring DM642 EVM Video ADC and Video DAC Blocks	2-61
Creating DSP Application Models for Targeting .....	2-68
Using Logging in Your DSP Applications .....	2-68
Generating Code from Real-Time Models .....	2-69
<b>Schedulers and Timing .....</b>	<b>2-71</b>
Timer-Based Versus Asynchronous Interrupt Processing . . . .	2-71
Synchronous Scheduling .....	2-72
Asynchronous Scheduling .....	2-73
Asynchronous Scheduler Examples .....	2-74
Uses for Asynchronous Scheduling .....	2-76
Scheduling Considerations .....	2-81
<b>Setting Real-Time Workshop Options for</b>	
<b>C6000 Hardware .....</b>	<b>2-82</b>
<b>Setting Real-Time Workshop Pane Options .....</b>	<b>2-85</b>
Target Selection .....	2-86
Documentation .....	2-87
Build Process .....	2-87
Custom Storage Class .....	2-88
Debug Pane Options .....	2-89
Optimization Pane Options .....	2-90
TI C6000 Code Generation Pane Options .....	2-92
Overrun Indicator and Software-Based Timer .....	2-98
TI C6000 Compiler/Linker Options .....	2-98
Linker Options .....	2-105
Embedded Target for TI C6000 DSP	
Default Project Configuration — custom_MW .....	2-106
<b>Model Reference and Embedded Target for</b>	
<b>TI C6000 DSP .....</b>	<b>2-107</b>

How Model Reference Works . . . . .	2-107
Using Model Reference with Embedded Target for TI C6000 DSP . . . . .	2-108
Configuring Targets to Use Model Reference . . . . .	2-110
<b>Targeting Supported Boards . . . . .</b>	<b>2-111</b>
Typical Targeting Process . . . . .	2-111
Targeting the C6711 DSP Starter Kit . . . . .	2-112
Configuring Your C6711 DSK . . . . .	2-114
Confirming Your C6711 DSK Installation . . . . .	2-115
Testing Your C6711 DSK . . . . .	2-115
Creating Your Simulink Model for Targeting . . . . .	2-119
<b>Targeting Tutorial II — A More Complex Application . . . . .</b>	<b>2-123</b>
<b>Targeting Your C6711 DSK and Other Hardware . . . . .</b>	<b>2-136</b>
Configuring Your C6711 DSK . . . . .	2-136
Confirming Your C6711 DSK Installation . . . . .	2-136
Testing Your C6711 DSK . . . . .	2-137
Running Models on Your C6711 DSK . . . . .	2-141
<b>Creating Code Composer Studio Projects Without Building . . . . .</b>	<b>2-144</b>
<b>Targeting Custom Hardware . . . . .</b>	<b>2-146</b>
Typical Targeting Process . . . . .	2-148
Targeting a Custom Target . . . . .	2-150
Sections Pane . . . . .	2-158
To Create Memory Maps for Targets . . . . .	2-164
<b>Using Embedded Target for TI C6000 DSP with Real-Time Workshop Embedded Coder . . . . .</b>	<b>2-165</b>

## Targeting with DSP/BIOS Options

# 3

<b>Introducing DSP/BIOS . . . . .</b>	<b>3-2</b>
---------------------------------------	------------

<b>DSP/BIOS and Targeting Your TI C6000 DSP</b> .....	<b>3-3</b>
DSP/BIOS Configuration File .....	<b>3-3</b>
Memory Mapping .....	<b>3-4</b>
Hardware Interrupt Vector Table .....	<b>3-4</b>
Linker Command File .....	<b>3-5</b>
<b>Code Generation with DSP/BIOS</b> .....	<b>3-6</b>
Generated Code Without and With DSP/BIOS .....	<b>3-6</b>
<b>Profiling Generated Code</b> .....	<b>3-10</b>
Profiling Subsystems .....	<b>3-10</b>
Details About Timing and Profiling .....	<b>3-11</b>
Profiling Multitasking Systems .....	<b>3-13</b>
The Profiling Report .....	<b>3-14</b>
Interrupts and Profiling .....	<b>3-15</b>
Reading Your Profile Report .....	<b>3-16</b>
Definitions of Report Entries .....	<b>3-17</b>
Profiling Your Generated Code .....	<b>3-19</b>
To Enable Profiling for Your Generated Code .....	<b>3-20</b>
To Create Atomic Subsystems for Profiling .....	<b>3-21</b>
<b>Using DSP/BIOS with Your Target Application</b> .....	<b>3-25</b>
To Enable DSP/BIOS When You Generate Code .....	<b>3-25</b>

## Using the C62x and C64x DSP Libraries

# 4

<b>About the C62x and C64x DSP Libraries</b> .....	<b>4-2</b>
Characteristics Common to C62x and C64x Library Blocks ...	<b>4-3</b>
<b>Fixed-Point Numbers</b> .....	<b>4-4</b>
Signed Fixed-Point Numbers .....	<b>4-4</b>
Q Format Notation .....	<b>4-5</b>
<b>Building Models</b> .....	<b>4-8</b>
Converting Data Types .....	<b>4-8</b>
Using Sources and Sinks .....	<b>4-9</b>



Choosing Blocks to Optimize Code .....	4-9
--	-----

## Blocks — By Category

---

# 5

<b>Target Preferences (c6000tgtprefs)</b> .....	<b>5-2</b>
<b>RTDX Instrumentation (rtdxblocks)</b> .....	<b>5-3</b>
<b>C62x DSP (tic62dsplib)</b> .....	<b>5-4</b>
Conversions .....	<b>5-4</b>
Filters .....	<b>5-4</b>
Math and Matrices .....	<b>5-5</b>
Transforms .....	<b>5-5</b>
<b>C64x DSP (tic64dsplib)</b> .....	<b>5-7</b>
Conversions .....	<b>5-7</b>
Filters .....	<b>5-7</b>
Math and Matrices .....	<b>5-8</b>
Transforms .....	<b>5-8</b>

<b>C6416 DSK (c6416dsklib)</b> .....	<b>5-10</b>
<b>C6711 DSK (c6711dsklib)</b> .....	<b>5-11</b>
<b>C6713 DSK (c6713dsklib)</b> .....	<b>5-12</b>
<b>DM642 EVM (dm642evmlib)</b> .....	<b>5-13</b>
<b>C6000 DSP Core Support (c6000dspcorelib)</b> .....	<b>5-14</b>
<b>TMDX3206040A DSP Support (tmdx326040lib)</b> .....	<b>5-15</b>
<b>Host Communication (hostcommlib)</b> .....	<b>5-16</b>
<b>C6000 DSP Communication (targetcommlib)</b> .....	<b>5-17</b>
<b>DSP/BIOS (dspbioslib)</b> .....	<b>5-18</b>
<b>Blocks — Alphabetical List</b> .....	<b>5-19</b>

## Supported Hardware and Issues

### A

<b>Supported Hardware for Targeting</b> .....	<b>A-2</b>
<b>Requirements for the DM642 EVM</b> .....	<b>A-6</b>
About DM642 EVM Board Revisions .....	<b>A-6</b>
Setting Up Code Composer Studio for the DM642 EVM .....	<b>A-7</b>
About the XDS560 PCI-Bus JTAG Scan-Based Emulator .....	<b>A-8</b>
Configuring the Target Preferences Block for Your DM642 EVM .....	<b>A-8</b>
Configuring the DM642 EVM Video ADC Block .....	<b>A-9</b>
<b>Continuing Issues with Embedded Target for TI C6000 DSP</b> .....	<b>A-11</b>
Setting the Clock Speed on the C6713 DSK .....	<b>A-11</b>

On the DM642 EVM, ADC-DAC Loopback Does Not  
Display An RGB Image Correctly After Power-Up ..... **A-12**  
Simulink Stop Block Works Differently When Not  
Using DSP/BIOS Features ..... **A-12**

## **Index**

---



# What Is Embedded Target for TI C6000 DSP?

---

Introducing Embedded Target for the TI TMS320C6000 DSP Platform (p. 1-2)

About Embedded Target for C6000 DSP (p. 1-4)

Using This Guide (p. 1-5)

Configuration Information (p. 1-7)

Getting Started (p. 1-9)

Introduces the Embedded Target for TI C6000 DSP and some of the features it provides. Also links to the supported hardware section in the Appendix.

Presents an overview of the capabilities of the Embedded Target for TI C6000 DSP

Introduces the organization of the User's Guide and provides summaries of each section

Describes how to determine if you have installed Embedded Target for TI C6000 DSP

Talks about the software and hardware required to use the Embedded Target for TI C6000 DSP, from both The MathWorks and from Texas Instruments

## Introducing Embedded Target for the TI TMS320C6000 DSP Platform

Embedded Target for the TI TMS320C6000 DSP Platform integrates Simulink® and MATLAB® with Texas Instruments eXpressDSP™ tools. The software collection lets you develop and validate digital signal processing designs from concept through code. The Embedded Target for TI C6000 DSP consists of the TI C6000 target that automates rapid prototyping on your C6000 hardware targets. The target uses C code generated by Real-Time Workshop® and your TI development tools to build an executable file for your targeted processor. The Real-Time Workshop build process loads the targeted machine code to your board and runs the executable file on the digital signal processor.

Refer to “Supported Hardware for Targeting” on page A-2 for a list of the targets that Embedded Target for TI C6000 DSP supports. You can generate executable code for any of the supported targets.

All the features provided by Code Composer Studio (CCS), such as tools for editing, building, debugging, code profiling, and project management, work to help you develop applications using MATLAB, Simulink, Real-Time Workshop, and your supported hardware. When you use this target, the build process creates a new project in Code Composer Studio and populates the project with the files the project requires.

As long as your TI hardware, whether built by TI or custom, supports communications over JTAG and RTDX, you can use the Embedded Target for TI C6000 DSP with your hardware, enabling you to maximize the results of your development time and effort.

This chapter provides sections that describe the following:

- Some of the digital signal processing applications you can develop with Embedded Target for TI C6000 DSP, in the section “Suitable Applications” on page 1-3
- Prerequisites for using Embedded Target for TI C6000 DSP, in the section “Platform Requirements—Hardware and Operating System” on page 1-9

## Suitable Applications

The Embedded Target for TI C6000 DSP enables you to develop digital signal processing applications that have any of the following characteristics:

- Single rate
- Multirate
- Multistage
- Adaptive
- Frame based
- Fixed point when you use the C62x or C64x blocks with C64xx and C67xx targets.

Your supported boards, and the Embedded Target for TI C6000 DSP, cover a range of standard input sampling frequencies from 5.5 KHz to 48 KHz or more. The specific supported input range depends on the board you own.

For any model to work in the targetting environment, you must select the discrete-time solver in the Simulink **Solver** options. Targeting does not work with continuous time solvers.

## About Embedded Target for C6000 DSP

Embedded Target for TI C6000 DSP lets you use Simulink to model digital signal processing algorithms from blocks in the Signal Processing Blockset, and then use Real-Time Workshop to generate (or build) ANSI C code targeted to the Texas Instruments DSP development boards or Texas Instruments Code Composer Studio™ Integrated Development Environment (CCS IDE). The Embedded Target for TI C6000 DSP takes the generated C code and uses Texas Instruments (TI) tools to build specific machine code depending on the TI board you use. The build process downloads the targeted machine code to the selected hardware and runs the executable on the digital signal processor. After downloading the code to the board, your digital signal processing (DSP) application runs automatically on your target.



## Using This Guide

This section provides some guidance for using this book to learn more about the Embedded Target for TI C6000 DSP.

### Expected Background

This document introduces you to using Embedded Target for C6000 DSPs with Real-Time Workshop to develop digital signal processing applications for the Texas Instruments CC6000 family of DSP development hardware, such as the TI TMS320C6711 DSP Starter Kit. To get the most out of this manual, you should be familiar with MATLAB and its associated programs, such as Signal Processing Blockset and Simulink. We do not discuss details of digital signal processor operations and applications, except to introduce concepts related to using the C6711 DSP or other 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.

For information about Code Composer Studio and Real-Time Data Exchange™ (RTDX™), refer to your Texas Instruments documentation for each product. 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, “What Is Embedded Target for TI C6000 DSP?” This introduces the Embedded Target for TI C6000 DSP 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 that Embedded Target for C6000 DSPs supports.

## **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 C6711 DSK. Two example models introduce the targeting software and build files, and give you an idea of the range of applications supported by Embedded Target for C6000 DSPs. For C6711 DSK users, refer to “Configuring Your C6711 DSK” on page 2-136 for more information about installing and using your C6711 DSK.

## Configuration Information

To determine whether the Embedded Target for TI C6000 DSP is installed on your system, type this command at the MATLAB prompt.

```
c6000lib
```

When you enter this command, MATLAB displays the C6000 block library containing the following libraries that comprise the C6000 library:

- C6000 DSP Core Support
- C62x DSP Library
- C64x DSP Library
- C6416 DSK Board Support
- C6711 DSK Board Support
- C6713 DSK Board Support
- DM642 EVM Board Support
- DSP/BIOS Library
- Host Communication Library
- RTDX Instrumentation
- Target Preferences
- TMDX326040 Daughtercard Support

If you do not see the listed libraries, or MATLAB does not recognize the command, install the Embedded Target for TI C6000 DSP. Without the software, you cannot use Simulink and Real-Time Workshop to develop applications targeted to the TI boards.

---

**Note** For up-to-date information about system requirements, refer to the system requirements page, available in the products area at the MathWorks Web site (<http://www.mathworks.com>).

---

To verify that CCS is installed on your machine, enter

```
ccsboardinfo
```

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

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

If MATLAB does not return information about any boards, revisit your CCS installation and setup in your CCS documentation.

As a final test, launch CCS to ensure that it starts up successfully. For the Embedded Target for TI C6000 DSP to operate with CCS, the CCS IDE must be able to run on its own.

## Getting Started

This section describes the hardware and software you need to run the Embedded Target for TI C6000 DSP on your Microsoft Windows PC.

Embedded Target for TI C6000 DSP runs on Windows 2000, and Windows XP platforms.

### **Platform Requirements—Hardware and Operating System**

To run the Embedded Target for TI C6000 DSP, your host PC must meet the following hardware configuration:

- Intel Pentium or Intel Pentium processor compatible PC
- 64 MB RAM (128 MB recommended)
- 20 MB hard disk space available after installing MATLAB
- Color monitor
- CD-ROM drive
- Windows 2000 or Windows XP.

You may need additional hardware, such as signal sources and generators, microphones, oscilloscopes or signal display systems, and assorted audio cables to test and evaluate your digital signal processing application on your hardware.

Refer to your documentation from The MathWorks for more information on installing the software required to support Embedded Target for TI C6000 DSP, as shown in the next table. In all cases, Embedded Target for TI C6000

DSP requires that you install one of the two most recent versions of the required software.

## **Prerequisites for Using Embedded Target for TI C6000 DSP Software for Targeting**

<b>Installed Product</b>	<b>Additional Information</b>
MATLAB	Core software from The MathWorks
Link for Code Composer Studio™ Development Tools	Software to enable communications between MATLAB and the Code Composer Studio development environment. Required for the Embedded Target for TI C6000 DSP to work in code generation and targeting.
Real-Time Workshop	Software used to generate C code from Simulink models
Simulink	Software package for modeling, simulating, and analyzing dynamic systems
Signal Processing Toolbox	Software package for analyzing signals, processing signals, and developing algorithms
Signal Processing Blockset	Block libraries used by Simulink

For information about the software required to use the Link for Code Composer Studio Development Tools, refer to the Products area of the MathWorks Web site—<http://www.mathworks.com>.

## **Texas Instruments Software**

In addition to the required software from The MathWorks, Embedded Target for TI C6000 DSP requires that you install the Texas Instruments development

tools and software listed in the following table. Installing Code Composer Studio IDE for the C6000 series installs the software shown.

#### Required TI Software for Targeting Your TI C6000 Hardware

Installed Product	Additional Information
Assembler	Creates object code (.obj) for C6000 boards from assembly code.
Compiler	Compiles C code from the blocks in Simulink models into object code (.obj). As a byproduct of the compilation process, you get assembly code (.asm) as well.
Linker	Combines various input files, such as object files and libraries.
Code Composer Studio	Texas Instruments integrated development environment (IDE) that provides code debugging and development tools.
TI C6000 miscellaneous utilities	Various tools for developing applications for the C6000 digital signal processor family.
Code Composer Setup Utility	Program you use to configure your CCS installation by selecting your target boards or simulator.

In addition to the TI software, you need one or more of the following in any combination:

- One or more Texas Instruments TMS320C6416 DSP Starter Kits
- One or more TMS320C6711 DSP Starter Kits
- One or more TMS320C6713 DSP Starter Kits
- One or more TMDX326040A Daughter Cards for the C6711 DSK, used with the DSK. This daughter card is also known as the PCM3003 Audio Daughter Card
- One or more DM642 Evaluation Modules
- One or more boards from the supported hardware lists

- One or more configured simulators for any supported digital signal processors

For up-to-date information about the software from The MathWorks you need to use the Embedded Target for TI C6000 DSP, refer to the MathWorks Web site—<http://www.mathworks.com>. Check the Product area for the Embedded Target for the TI TMS320C6000 DSP Platform.



# Targeting C6000 DSP Hardware

---

Overview (p. 2-3)	Introduces Embedded Target for TI C6000 DSP and the tutorials in this chapter.
TI C6000 and Code Composer Studio IDE (p. 2-4)	Discusses the blocks provided by the Embedded Target for TI C6000 DSP for developing models for TI C6000™ DSP platforms. Also lists the supported hardware.
Targeting Tutorial — Single Rate Application (p. 2-10)	Takes you through the process of creating models in Simulink and generating code for your targets. Uses the 6701 EVM as the example board.
Using the C6000lib Blockset (p. 2-21)	Describes the contents of the C6000lib blockset—what blocks are included and where, and briefly describes how to configure the blocks.
Schedulers and Timing (p. 2-71)	Describes the timer-based and asynchronous schedulers
Setting Real-Time Workshop Options for C6000 Hardware (p. 2-82)	Provides the details on setting the Real-Time Workshop options when you generate code from your Simulink models to TI hardware.
Targeting Supported Boards (p. 2-111)	If you are targeting a C6711 DSK, this section details specific information about using the target.
Model Reference and Embedded Target for TI C6000 DSP (p. 2-107)	Introduces model reference and how you use model reference with Embedded Target for TI C6000 DSP
Targeting Tutorial II — A More Complex Application (p. 2-123)	Using a more complex model than the previous tutorial, this exercise walks you through code generation for a multistage model.
Targeting Your C6711 DSK and Other Hardware (p. 2-136)	If you are targeting a C6711 DSK, this section details specific information about using the target, although the process shown applies to other targets equally.

Creating Code Composer Studio  
Projects Without Building (p. 2-144)

You have the option of generating code into a Code Composer Studio project, rather than to hardware. This section introduces the `Generate_CCS_project` selection in the Real-Time Workshop build options.

Targeting Custom Hardware (p. 2-146)

Discusses how you target processors on boards that are not supported boards. We call these boards custom targets.

Using Embedded Target for TI C6000  
DSP with Real-Time Workshop  
Embedded Coder (p. 2-165)

Provides details about using Embedded Target for TI C6000 DSP with your Real-Time Workshop Embedded Coder software and embedded real-time target.

## Overview

The Embedded Target for the TI TMS320C6000 DSP Platform lets you use Real-Time Workshop to generate a C language real-time implementation of your Simulink model. You can compile, link, download, and execute the generated code on the C6711 DSP Starter Kit (DSK). In combination with the supported boards (refer to “Supported Hardware for Targeting” on page A-2), your Embedded Target for TI C6000 DSP software is the ideal resource for rapid prototyping and developing embedded systems applications for C6711 digital signal processors. The Embedded Target for TI C6000 DSP software focuses on developing real-time digital signal processing (DSP) applications for C6000 hardware. Additional hardware that we support is listed in “Supported Hardware and Issues” on page A-1.

Although the tutorials in this chapter focus on the C6711 DSK, the techniques and processes apply to any supported hardware, with minor adjustments for the processor involved.

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

### About the Tutorials

In most cases, this chapter deals with the C6711 DSK targets. Fortunately, all members of the C6000 family of processors that we support work in a manner similar to the C6711 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 and targeting custom hardware.

# TI C6000 and Code Composer Studio IDE

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 the Embedded Target for TI C6000 DSP uses Code Composer Studio™ (CCS) Integrated Development Environment (IDE) with the Real-Time Workshop and the C6000lib blockset.

Executing code generated from Real-Time Workshop on a particular target in real time requires that Real-Time Workshop 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 Link for Code Composer Studio Development Tools to invoke the code building process within CCS. 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 debugging tools or across a link for CCS or Real-Time Data Exchange (RTDX). Otherwise the running process is not accessible.

Used in combination with your Embedded Target for TI C6000 DSP and Real-Time Workshop, TI products provide an integrated development environment that, once installed, needs no additional coding.

## Supported Boards and Simulators

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

Refer to “Supported Hardware and Issues” for the latest information about the hardware supported by the Embedded Target for TI C6000 DSP.

## About Simulators

CCS offers many simulators for the C6701 and C6711 digital signal processors, and other C6000 processors in the CCS Setup utility. Much of your model and algorithm development efforts work with the simulators, such as code generation. And, since the Embedded Target for TI C6000 DSP provides a software-based scheduler, your models and generated code run on the

simulators just as they do on your hardware. You can use the RTDX links with the simulators as well. For more information about the simulators in CCS, 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. To target C6711DSK boards, your simulator must contain a C6711 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.

### Using a Simulator

You can use the simulator alone to develop projects with Embedded Target for TI C6000 DSP. The simulator can generate and handle timer interrupts properly to enable your generated code to run.

To use the simulator, you configure the target preferences block in your model to use the simulator target.

- 1 Click the target preferences block in your model and select **Edit > Open Block** from the menu bar for your model. This step opens the C6000 Target Preferences dialog box for your target.
- 2 On the **Board info** pane in the C6000 Target Preferences dialog box, select **Simulator**.
- 3 Click **Apply** to apply the change, or click **OK** to apply the new setting and close the dialog box.

There is one manual step to do to use the simulator. After you generate code from a model to a CCS project, you must modify the project by setting the **RTDX Mode** in CCS to Simulator.

In addition, you must substitute the file `rtdxsim.lib` instead of the default `rtdx.lib` library file in the project. Accomplish this project file modification by navigating to the **Include Libraries** option in CCS:

**Build Options > Linker > Basic**

and replacing the file as needed in the **Include Libraries** option.

After you make this file substitution, you cannot use the Line In and Line Out ADC block options or any other target-specific board-level blocks. You can substitute any discrete-time sources and sinks from Simulink, Signal Processing Blockset, or other blockset. When there are no codec blocks (ADC or DAC blocks) in your model, the Embedded Target for TI C6000 DSP configures an on-chip timer to trigger the system at the appropriate sample time. As a result, whatever happens in the model is completely up to you, the user, as long as you provide the discrete sample time.

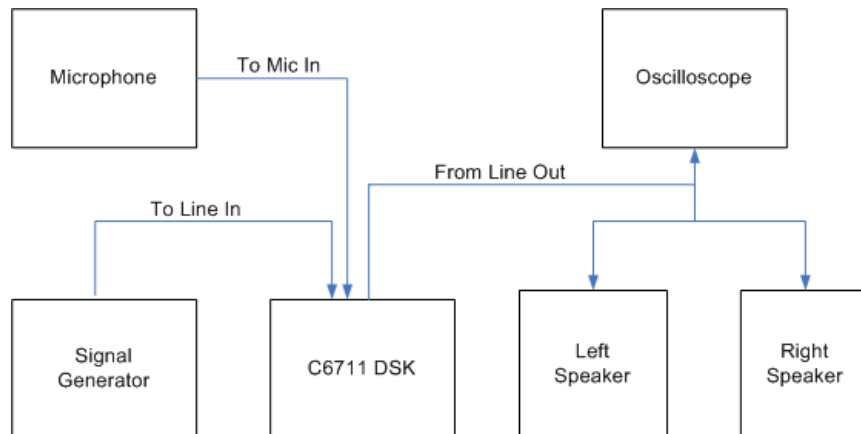
### Using RTDX with a Simulator

If you are using DSP/BIOS in your project, you configure RTDX by opening the DSP/BIOS Config properties in the project tree in CCS, opening the project .cdb file, and navigating to Input/Output. In the Input/Output properties you set the **RTDX mode** to Simulator.

If your project is not using DSP/BIOS, you only have to change the RTDX mode when you are using RTDX blocks in your model. Otherwise, RTDX is not needed.

### Typical Hardware Setup for C6711 DSK in Models

The next figure presents a block diagram of the typical setup for the inputs and output for the C6711 DSK.



After you have installed one or more of the supported development boards shown in “Supported Hardware and Issues” on page A-1, start MATLAB. At the MATLAB command prompt, type `c6000lib`. This opens a Simulink blockset named C6000lib that includes libraries that contain blocks predefined for C6000 input and output devices:

<b>Library</b>	<b>Description</b>
“C6000 DSP Communication (targetcommlib)” on page 5-17	Blocks that provide UDP and TCP/IP communications capability on the target. Includes byte manipulation blocks.
“C6000 DSP Core Support (c6000dspcorelib)” on page 5-14	Blocks for managing memory and task scheduling on C6000-based targets.
“C62x DSP (tic62dsplib)” on page 5-4	Blocks that provide C62x-optimized algorithms such as filtering and matrix manipulation.
“C64x DSP (tic64dsplib)” on page 5-7	Blocks that provide C64x-optimized algorithms such as filtering and matrix manipulation.
“C6416 DSK (c6416dsklib)” on page 5-10	Blocks to configure the peripherals on the C6416 DSK.
“C6711 DSK (c6711dsklib)” on page 5-11	Blocks to configure the peripherals on the C6711 DSK.
“C6713 DSK (c6713dsklib)” on page 5-12	Blocks to configure the peripherals on the C6713 DSK.
“DM642 EVM (dm642evmlib)” on page 5-13	Blocks to configure the peripherals on the DM642 EVM and configure video capture.
“DSP/BIOS (dspbioslib)” on page 5-18	Blocks that provide scheduling management using DSP/BIOS.

<b>Library</b>	<b>Description</b>
“Host Communication (hostcommlib)” on page 5-16	Blocks that configure the target for UDP communications. Includes byte manipulation blocks.
“RTDX Instrumentation (rtdxblocks)” on page 5-3	Blocks that provide RTDX instrumentation for communicating between your target and host.
“Target Preferences (c6000tgtprefs)” on page 5-2	Blocks that configure models for specific targets or custom C6000 hardware.
“TMDX3206040A DSP Support (tmdx326040lib)” on page 5-15	Blocks that configure the input and output for the daughtercard

Each board-based block library, such as C6713 DSK contains a version of each of 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), Embedded Target for TI C6000 DSP provides a timer that produces the interrupts required for timing and running your model, either on your hardware target or on a simulator.

### **Typical Hardware Setup for RTDX in Models**

In addition to the blocks for specific boards, the C6000lib blockset includes the library RTDX Instrumentation that contains RTDX input and output blocks that apply to all C6000 development boards and the C6000 DSP Core support library that contain blocks that let you transfer data to and from memory on



any C6000-based target. Like the RTDX blocks, the core support blocks are not hardware dependent.

With your model open, select **Configuration Parameters** from the **Simulink** option to open the Configuration Parameters dialog box. In the **Select** tree, click **Real-Time Workshop**. You must specify the appropriate versions of the system target file and template makefile. For the C6711 DSK, in the **Real-Time Workshop** pane of the dialog box, specify

```
ti_c6000.tlc
```

to select the correct target file in **Real-Time Workshop** system target file. Or click **Browse** and select `ti_c6000.tlc` from the list of targets, or whichever target best matches your hardware.

With this configuration, you can generate a real-time executable and download it to the TI development boards. You do this by clicking **Build** on the **Real-Time Workshop** pane. **Real-Time Workshop** automatically generates C code and inserts the I/O device drivers as specified by the ADC and DAC blocks in your block diagram, if any.

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 template makefile and block parameter dialog box entries get combined to form the target makefile for your TI C6000 board. Your makefile invokes the TI cross-compiler to build an executable file.

If you selected the **Build** and **execute build** action, the executable file is automatically downloaded over the parallel port to your C6711 DSK. After downloading the executable file to the target, the build process runs the file on the board's DSP.

# Targeting Tutorial – Single Rate Application

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 and Simulink block libraries, or you can find the model in the Embedded Target for TI C6000 DSP demos. For this example, we show the model as it appears in the demonstration program. The demonstration model name is `c6701evmaf_xr.mdl` as shown in the next figure. Open this model by entering `c6701evmaf_xr` at the MATLAB prompt.

To run this model you need a microphone connected to the **Mic In** connector on your C6711 DSK, and speakers and an oscilloscope connected to the **Line Out** connector on your C6711 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 C6711 DSK, complete the following tasks:

- 1 Use Simulink blocks, Signal Processing Blockset blocks, and blocks from other blocksets to create your model application.
- 2 Add the Embedded Target for TI C6000 DSP blocks that let your signal sources and output devices communicate with your C6711 DSK—the C6711 DSK ADC and C6711 DSK DAC blocks that you find in the Embedded Target for TI C6000 DSP `c6000lib` blockset.
- 3 Add the C6711DSK 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 C6711 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 stop time and solver options

- Real-Time Workshop options such as target configuration and target compiler selection

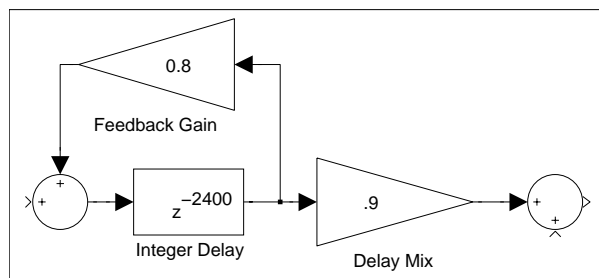
- 5 Build your model to the selected target.
- 6 Test your model running on the target by changing the input to the target and observing the output from the target.

Your target for this tutorial is your C6711 DSK installed on your PC. Be sure to configure and test your board as directed in “Configuring Your C6711 DSK” on page 2-114 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 blocks to create the following model.



Look for the Integer Delay block in the Signal Operations library of the Signal Processing Blockset. You do not need to add the input and output signal lines at this time. When you add the C6711 DSK blocks in the next section, you add the input and output to the sum blocks.

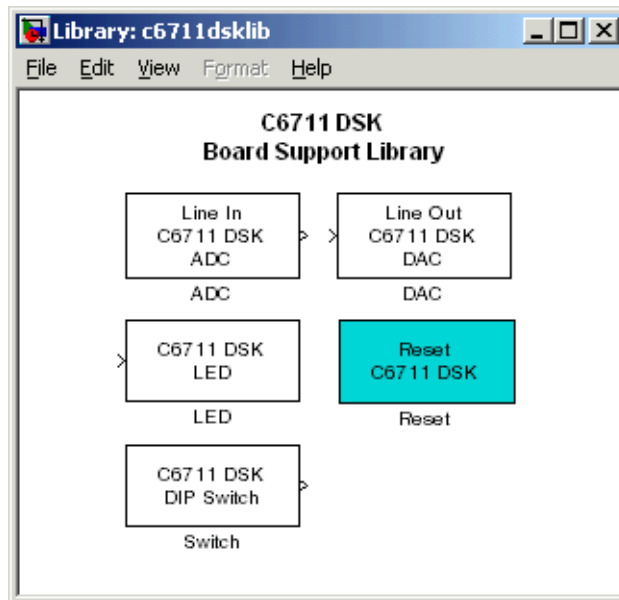
- 4 Save your model with a suitable name before continuing.

### Adding C6711 DSK Blocks to Your Model

So that you can send signals to your C6711 DSK and get signals back from the board, the Embedded Target for TI C6000 DSP includes a block library containing five blocks designed to work with the codec on your C6711 DSK:

- Input block (C6711 DSK ADC)
- Output block (C6711 DSK DAC)
- Light emitting diode block (C6711 DSK LED)
- Software reset block (Reset C6711 DSK)
- DIP switch block (C6711 DSK DIP Switch)

Entering `c6711dsklib` at the MATLAB prompt opens this window showing the library blocks. This block library is included in the Embedded Target for TI C6000 DSP `c6000lib` blockset in the Simulink Library browser.

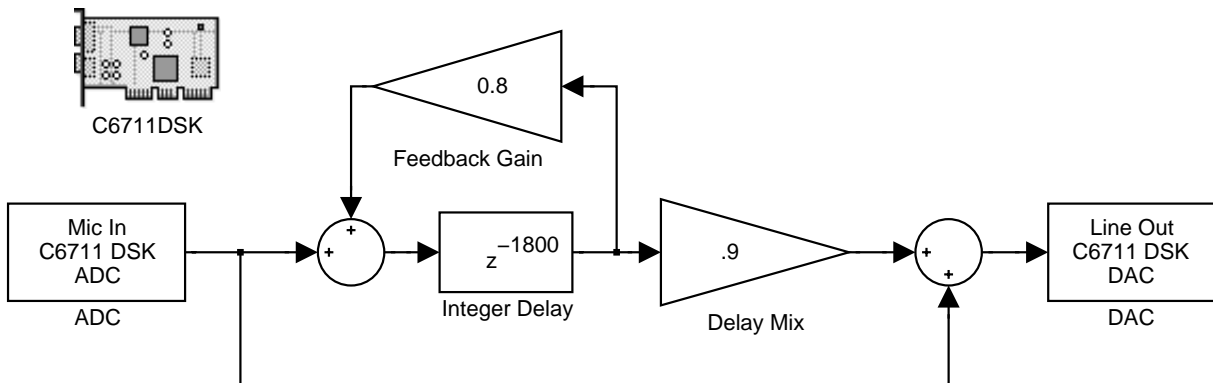


The C6711 DSK ADC and C6711 DSK DAC blocks generate code that configures the codec on your C6711 DSK to accept input signals from the input connectors on the board, and send the model output to the output connector on

the board. Essentially, the C6711 DSK ADC and C6711 DSK DAC blocks add driver software that controls the behavior of the codec for your model.

To add C6711 DSK target blocks to your model, follow these steps:

- 1 Double-click Embedded Target for TI C6000 DSP in the Simulink Library browser to open the c6000lib blockset.
- 2 Click the library C6711 DSK Board Support to see the blocks available for your C6701 EVM.
- 3 Drag and drop C6711 DSK ADC and C6711 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 C6711DSK Target Preferences block to the model. Notice that it is not connected to any other block in the model.

### Configuring the Embedded Target for TI C6000 DSP Blocks

To configure the Embedded Target for TI C6000 DSP blocks in your model, follow these steps:

- 1 Click the C6711 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 **C6711 DSK ADC source**, select Mic In.

**5** Click **OK** to close the C6711 DSK ADC dialog box.

**6** Now set the options for the C6711 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.

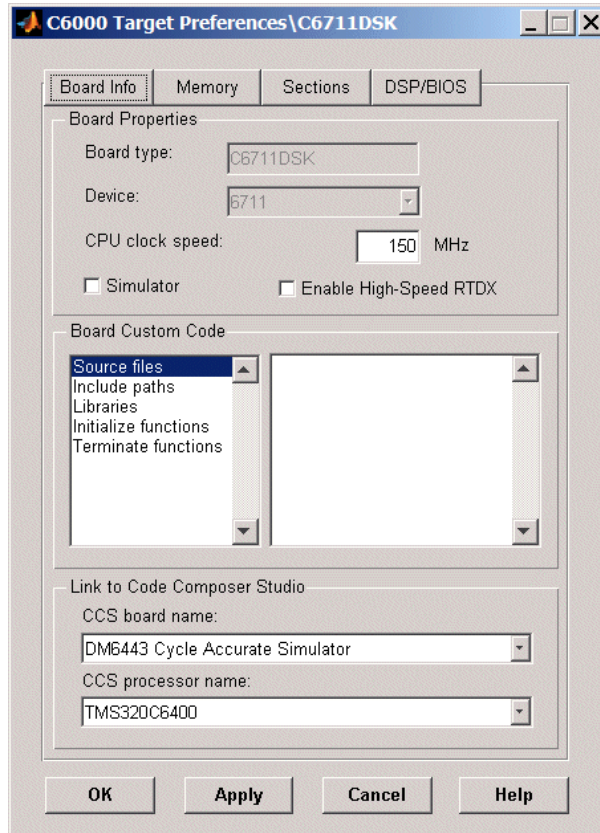
**7** Click **OK** to close the dialog box.

**8** Click the C6711DSK Target Preferences block.

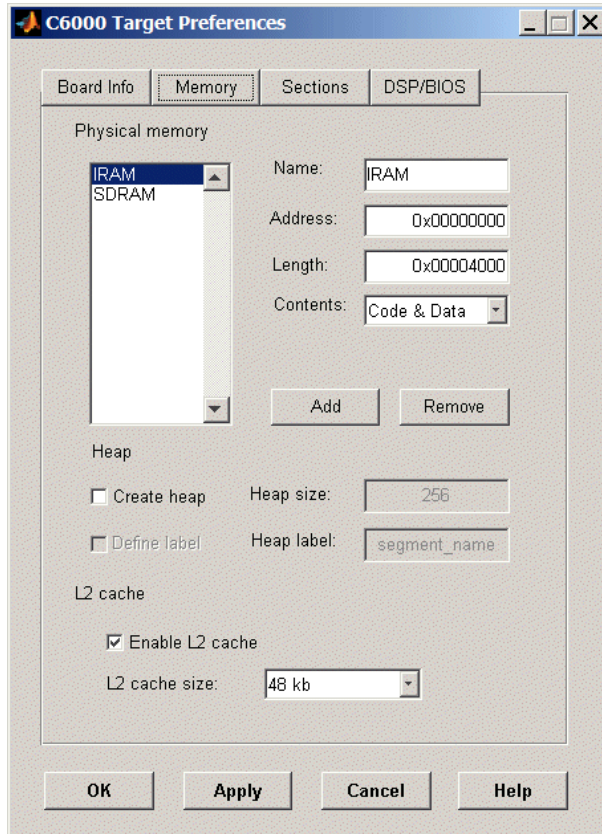
**9** Select **Block Parameters** from the Simulink **Edit** menu.

- 10** Verify the parameter settings for the C6711 DSK target. The figures below show the proper values.

### Board info Settings

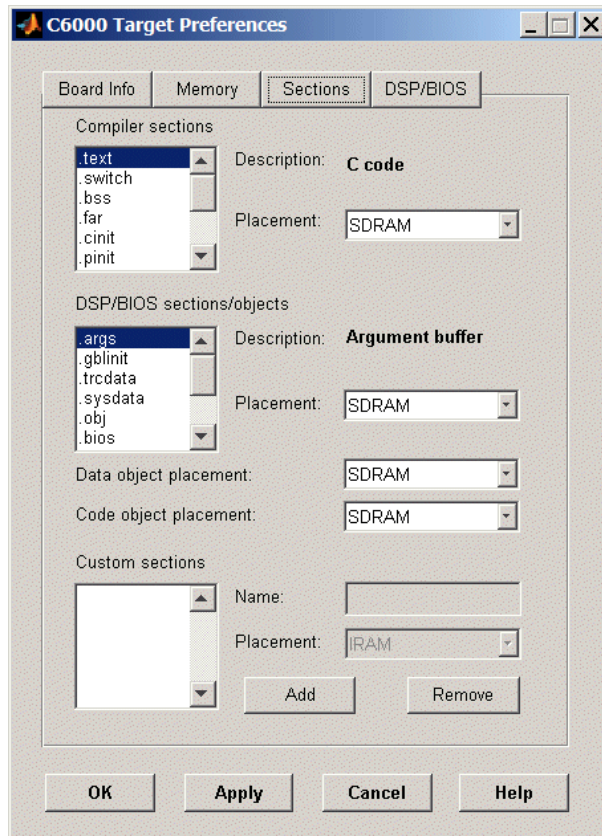


## Memory Settings





## Section Settings



You have completed the model. Now configure the Real-Time Workshop options to build and download your new model to your C6711 DSK.

## Specifying Configuration Parameters for Your Model

The following sections describe how to build and run real-time digital signal processing models on your C6711 DSK. Running a model on the target starts with configuring and building your model from the Configuration Parameters dialog box in Simulink.

### Setting Simulink Configuration Parameters

After you have designed and implemented your digital signal processing model in Simulink, 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 the Embedded Target for TI C6000 DSP.
  - 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

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

To configure Real-Time Workshop to use the correct target files and to compile and run your model executable file, you set the options in the Real-Time Workshop category of the Configuration Parameters dialog box. Follow these steps to set the Real-Time Workshop options to target your C6711 DSK:

- 1 Select Real-Time Workshop on the **Select** tree.
- 2 In Target selection, click **Browse** to select the system target file for C6000 targets—`ti_c6000.tlc`. It may already be the selected target.

Clicking **Browse** opens the **System Target File Browser**.

- 3 On the **System Target File Browser**, select the system target file `ti_c6000.tlc` and click **OK** to close the browser.

Real-Time Workshop updates the **Template makefile** and **Make command** options with the appropriate files based on your system target file selection.

- 4 From the **Select** tree, choose TI C6000 code generation to specify code generation options that apply to the C6711 DSK target.

- 5 Under **Code Generation**, select the **Inline run-time library functions** option. Clear the other options.
- 6 Under **Target Selection**, verify that **Export CCS handle to MATLAB base workspace** is selected and provide a name for the handle (optional).
- 7 Select TI C6000 Compiler/Linker on the **Select** tree to set the compiler options.
- 8 Set the following options in the dialog box under **Compiler**:
  - **Optimization level** should be Function (-o2).
  - Set **Compiler verbosity** to Quiet.Clear the other options under **Compiler**.
- 9 Set the linker operation options by selecting the **Retain .obj files** check box.
- 10 Change the category on the **Select** tree to Hardware Implementation.
- 11 Set **Byte ordering** to Little endian.
- 12 Change the category again to TI C6000 Code Generation.
- 13 Set the following Real-Time Workshop run-time options:
  - **Build action**: Build\_and\_execute.
  - **Overrun action**: Notify\_and\_halt.
  - **Overrun notification method**: Turn\_on\_LEDs.

You have configured the Real-Time Workshop options that let you target your C6711 DSK. You may have noticed that you did not configure a few Real-Time Workshop categories on the **Select** tree, such as Comments, Symbols, and Optimization.

For your new model, the default values for the options in these categories are correct. For other models you develop, 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 C6711 DSK

After you set the configuration parameters and configure Real-Time Workshop to create the files you need, you direct Real-Time Workshop 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 C6701 EVM.

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 C6711 DSP on your C6711 DSK, and then downloads the executable file to the target and runs the file.

- 3 To stop model execution, click the **Reset C6711 DSK** block or use the **Halt** option in CCS. You could type `halt` from the MATLAB command prompt as well.

### Testing Your Audio Reverb Model

With your model running on your C6711 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 `c6711dskafxr`, try running the demonstration model to compare the results.

## Using the C6000lib Blockset

The Embedded Target for TI C6000 DSP block library C6000lib comprises block libraries that contain blocks designed for targeting specific boards or using RTDX. The libraries are

<b>Library</b>	<b>Description</b>
“C6000 DSP Communication (targetcommplib)” on page 5-17	Blocks that provide UDP and TCP/IP communications capability on the target. Includes byte manipulation blocks.
“C6000 DSP Core Support (c6000dspcorelib)” on page 5-14	Blocks for managing memory and task scheduling on C6000-based targets.
“C62x DSP (tic62dsplib)” on page 5-4	Blocks that provide C62x-optimized algorithms such as filtering and matrix manipulation.
“C64x DSP (tic64dsplib)” on page 5-7	Blocks that provide C64x-optimized algorithms such as filtering and matrix manipulation.
“C6416 DSK (c6416dsklib)” on page 5-10	Blocks to configure the peripherals on the C6416 DSK.
“C6711 DSK (c6711dsklib)” on page 5-11	Blocks to configure the peripherals on the C6711 DSK.
“C6713 DSK (c6713dsklib)” on page 5-12	Blocks to configure the peripherals on the C6713 DSK.
“DM642 EVM (dm642evmlib)” on page 5-13	Blocks to configure the peripherals on the DM642 EVM and configure video capture.
“DSP/BIOS (dspbioslib)” on page 5-18	Blocks that provide scheduling management using DSP/BIOS.

<b>Library</b>	<b>Description</b>
“Host Communication (hostcommlib)” on page 5-16	Blocks that configure the target for UDP communications. Includes byte manipulation blocks.
“RTDX Instrumentation (rtidxblocks)” on page 5-3	Blocks that provide RTDX instrumentation for communicating between your target and host.
“Target Preferences (c6000tgtprefs)” on page 5-2	Blocks that configure models for specific targets or custom C6000 hardware.
“TMDX3206040A DSP Support (tmdx326040lib)” on page 5-15	Blocks that configure the input and output for the daughtercard

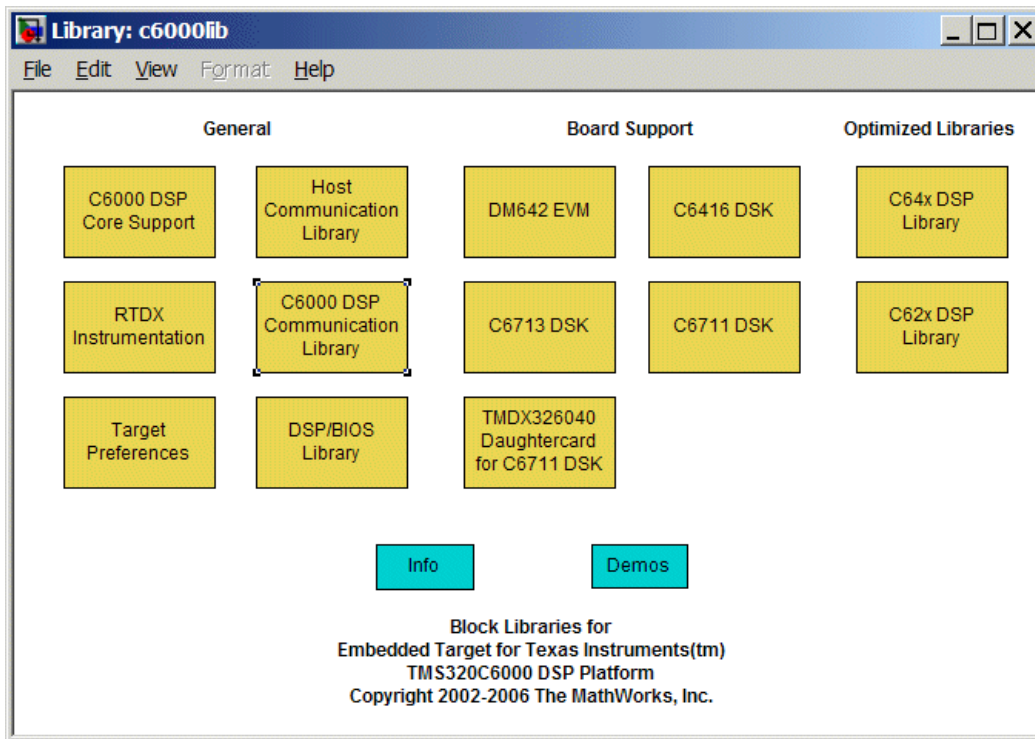
Each block library appears in one of the next figures. The sections after the figures review the configuration options for blocks in the EVM and DSK block libraries. For more information about the RTDX blocks, refer to “Constructing Link Objects” in your Link for Code Composer Studio documentation.

Each board-based block library contains a version of each of these blocks:

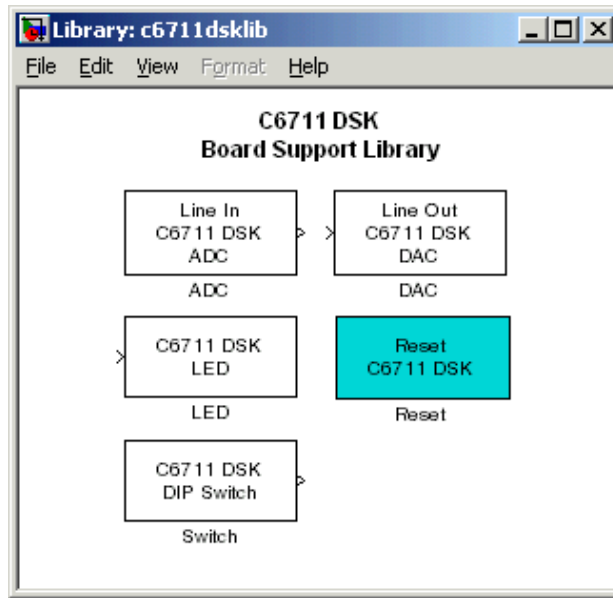
- ADC block
- DAC block
- DIP Switch block (optional, refer to the reference page for the DIP Switch block for your target)
- LED block
- Reset block

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.

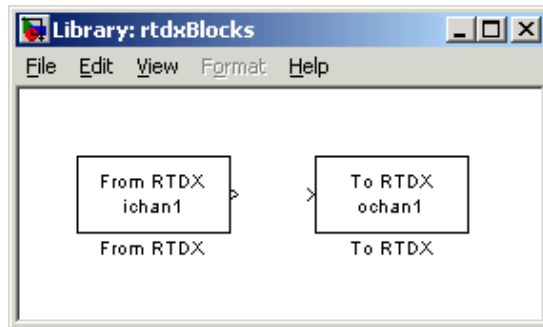
Here is the main library of blocks for Embedded Target for TI C6000 DSP.



The next figure shows the C6711 DSK block library.

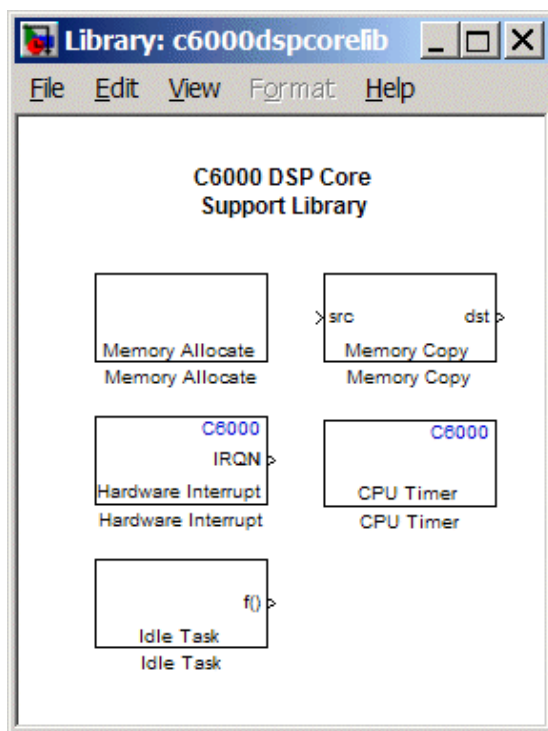


The RTDX Library contains the blocks shown in the next figure.

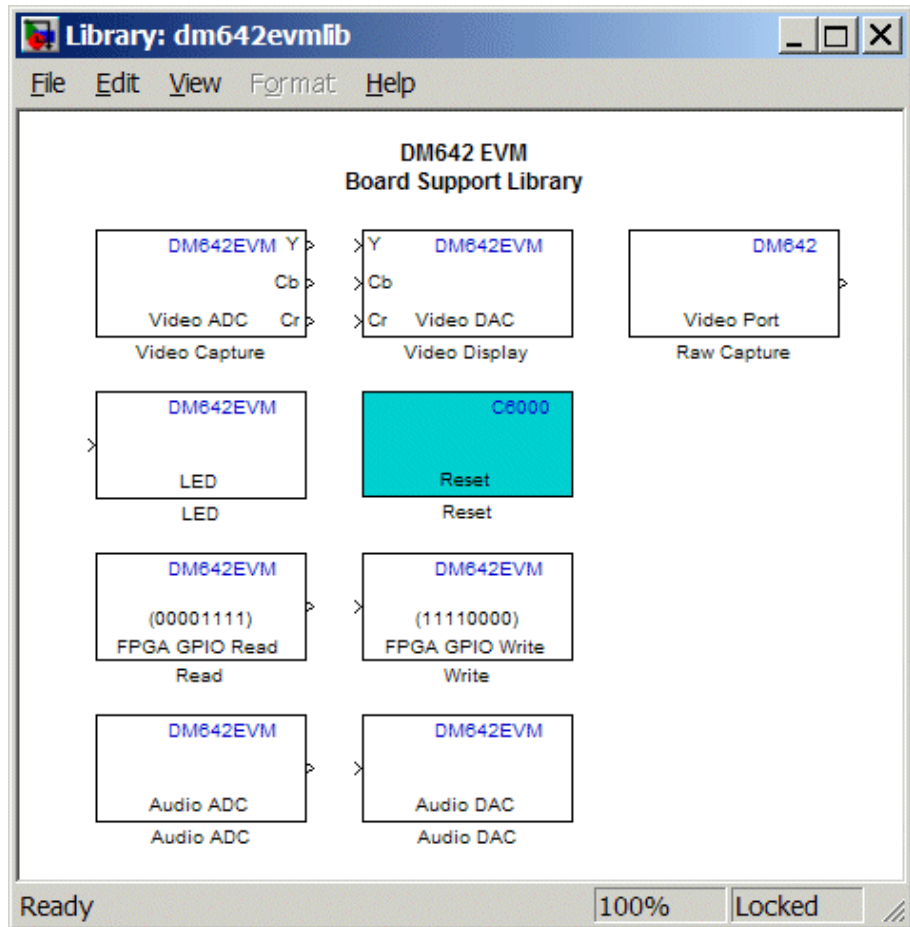


The core support blocks, stored in the DSP Core Support library, appear in the next figure.

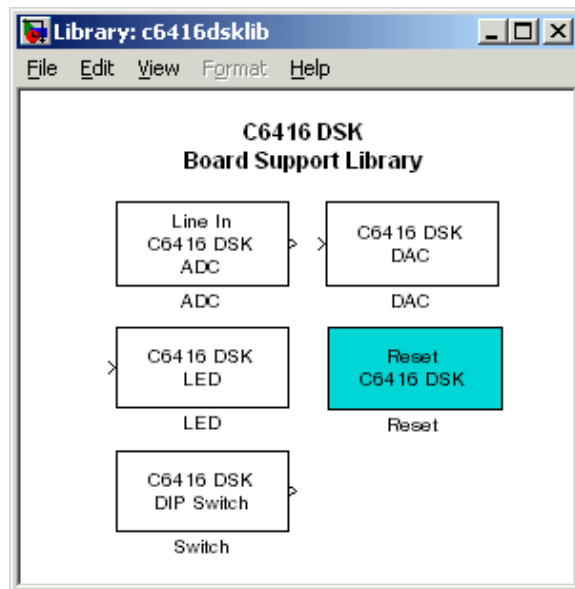




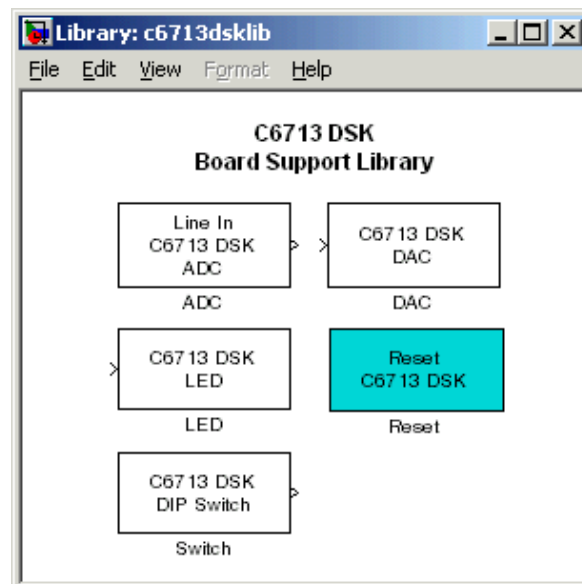
The following figure shows the DM642 EVM library contents.



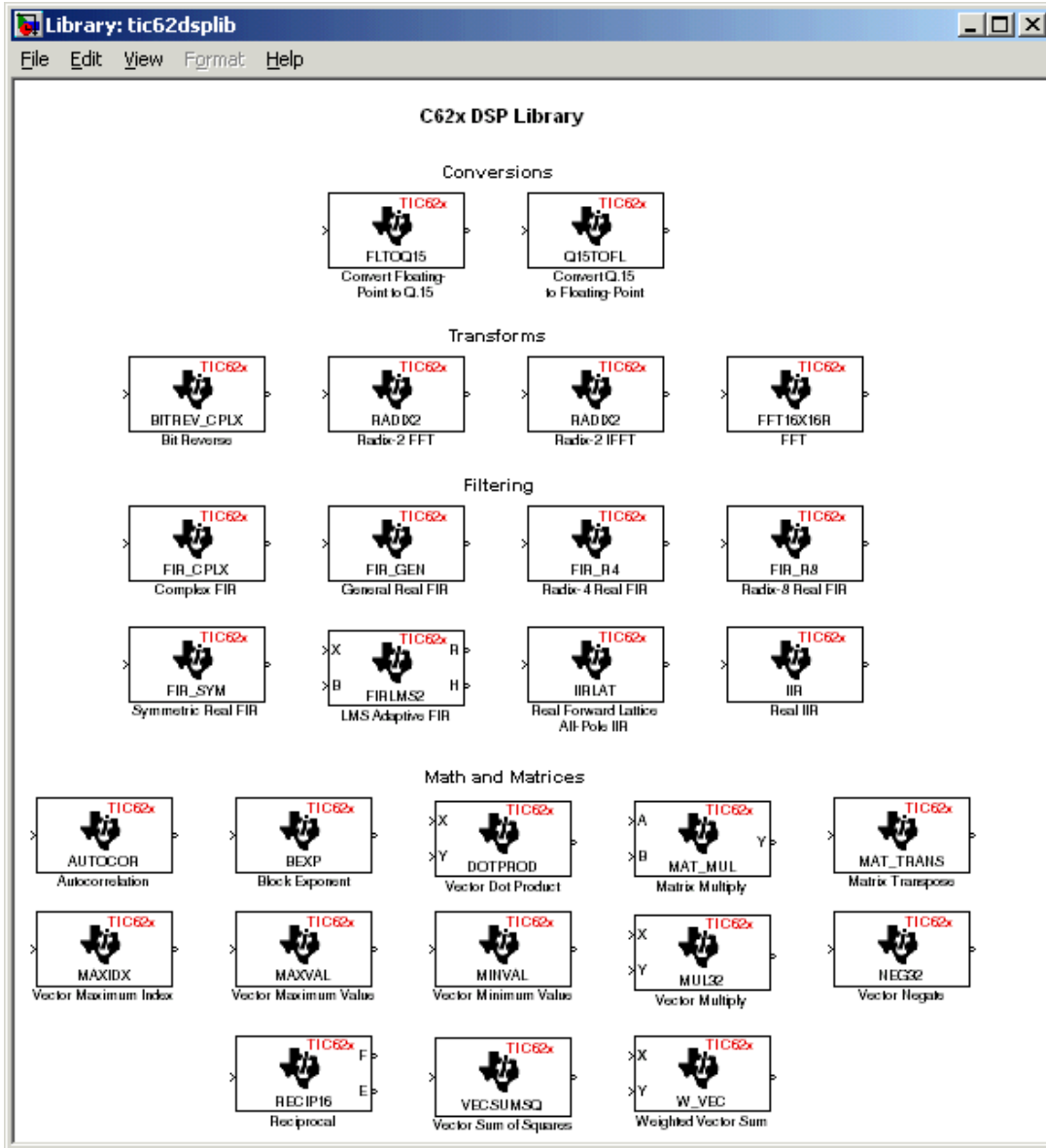
In the next figure you see the blocks in the C6416 DSK library.



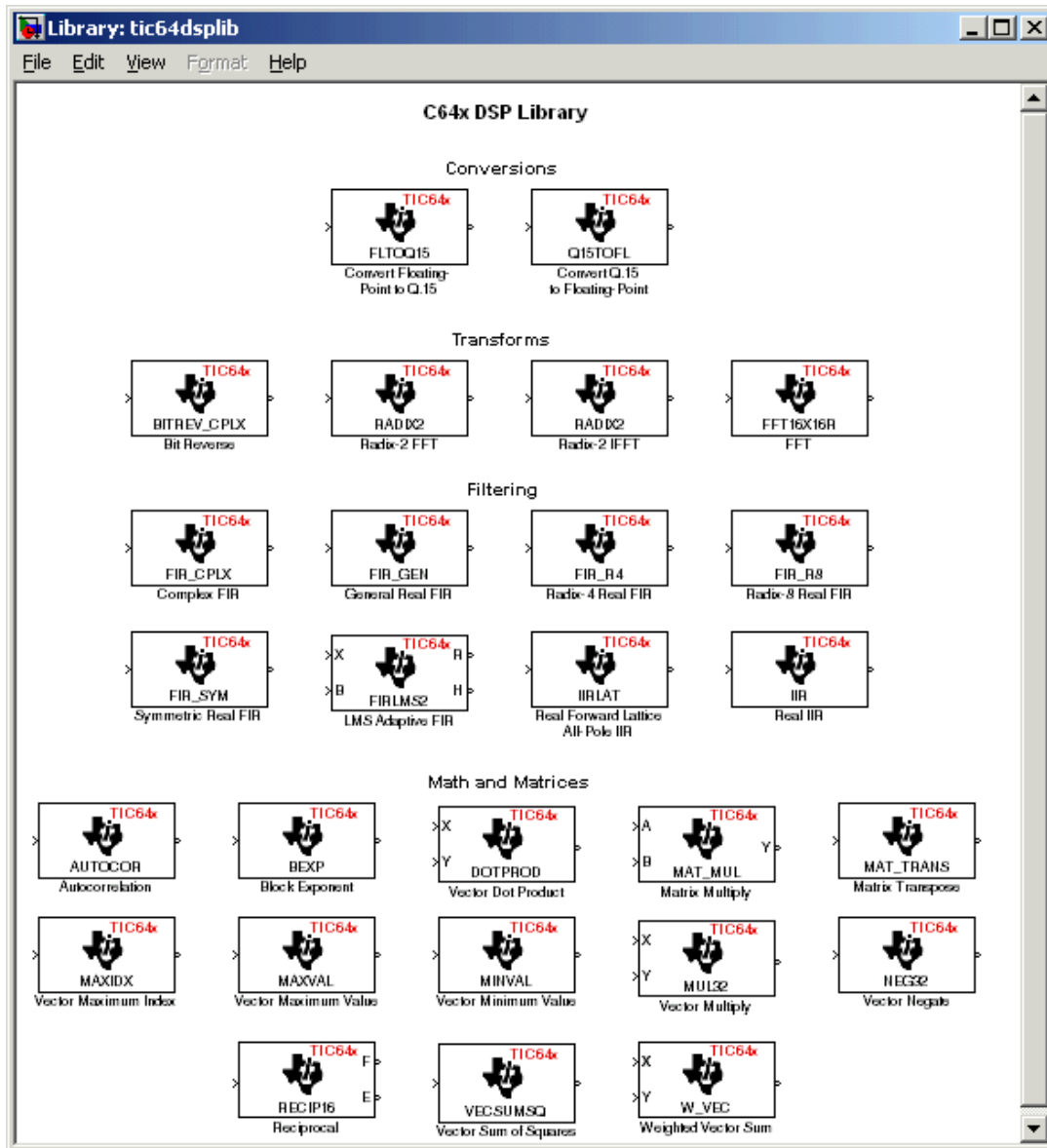
Similarly, the figure below shows the C6713 DSK blocks.



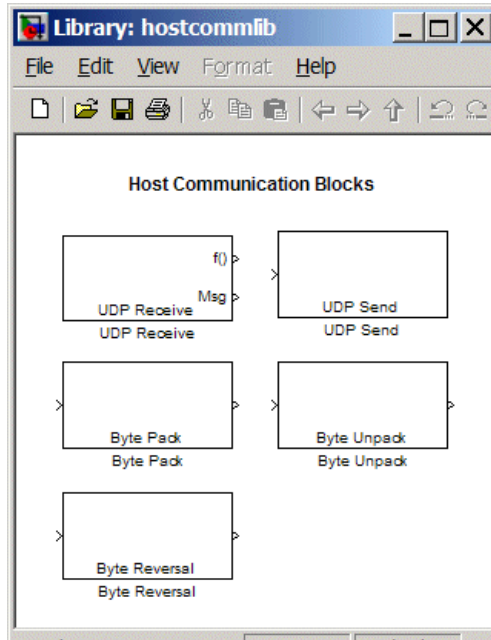
All C62x blocks appear in this figure that shows the C62x DSP library.



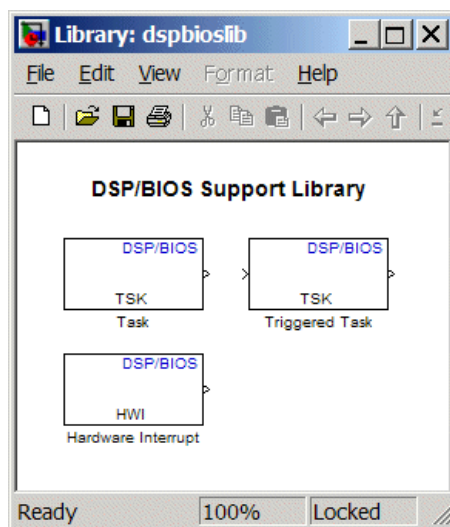
The C64x DSP library contains all of the blocks shown in the following figure.



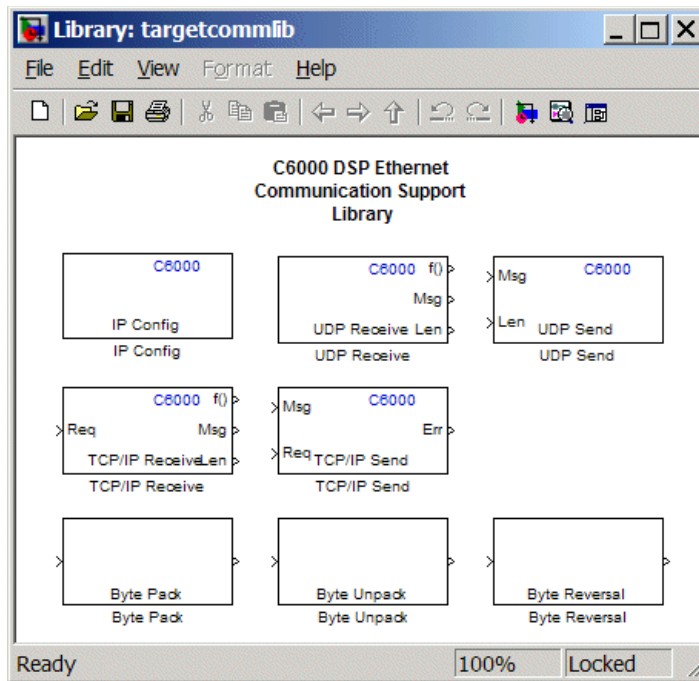
Blocks for communicating from the host to the target appear in the Host Communication library, shown here.



As shown in the next figure, library DSP/BIOS Support holds the blocks for adding DSP/BIOS to generated code.



The final library holds blocks for communicating from the target to the host—C6000 DSP Communication.



### Configuring ADC Blocks

To drive and test your DSP application on a C6711 DSK, you use signals from external sources, such as signal generators, audio equipment, or microphones. In some cases, you may generate your input data in code using Simulink blocks in your model or from a source block, such as a signal generator; configuring the ADC block remains the same.

The ADC and DAC blocks provide physical pathways from and to external sources and displays. They behave like source and sink blocks. They differ from sources and sinks in that they exchange data with external devices through analog input and output connectors, not the MATLAB workspace, and they work only for the C6000 boards.

You add ADC blocks to a model in the same way that you add other Signal Processing Blockset blocks, or Simulink blocks. You can add at most one ADC block to a model. When you add C6000 blocks to your Simulink model, you set parameters that determine how each block handles data.



Adding the ADC block to your Simulink model enables the codec on the target to accept input from your external source. By connecting your source to the LINE IN connector on the board mounting bracket, you introduce signals to the board. Your ADC block defines the signal format the codec uses to sample, digitize, and send signals to the digital signal processor.

When you build your Simulink model, the build process includes the software to implement the ADC-defined codec operation into the code downloaded to the board.

Configuring an ADC block includes setting as many as nine parameters on the Block Parameters dialog box.

Choosing and setting these parameters are covered in the following sections. To help you select the settings, this section provides some guidelines for common DSP uses and applications for each parameter. While the examples are not exhaustive, the suggestions may help you select settings that work well for your application.

Most of the configuration options for 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, the signal processor on the board, or direct memory access (DMA) on the board. In the following table, you find each ADC block option listed with the target board hardware affected (all parameters may not appear in every ADC blocks).

<b>Option</b>	<b>Affected Hardware</b>
<b>ADC Source</b>	Codec
<b>Inherit sample time</b>	TMS320C6xxx digital signal processor
<b>Mic</b>	Codec
<b>Output data type</b>	TMS320C6xxx digital signal processor
<b>Sample rate (Hz)</b>	Codec
<b>Samples per frame</b>	Direct memory access functions
<b>Scaling</b>	TMS320C6xxx digital signal processor
<b>Source gain (dB)</b>	Codec

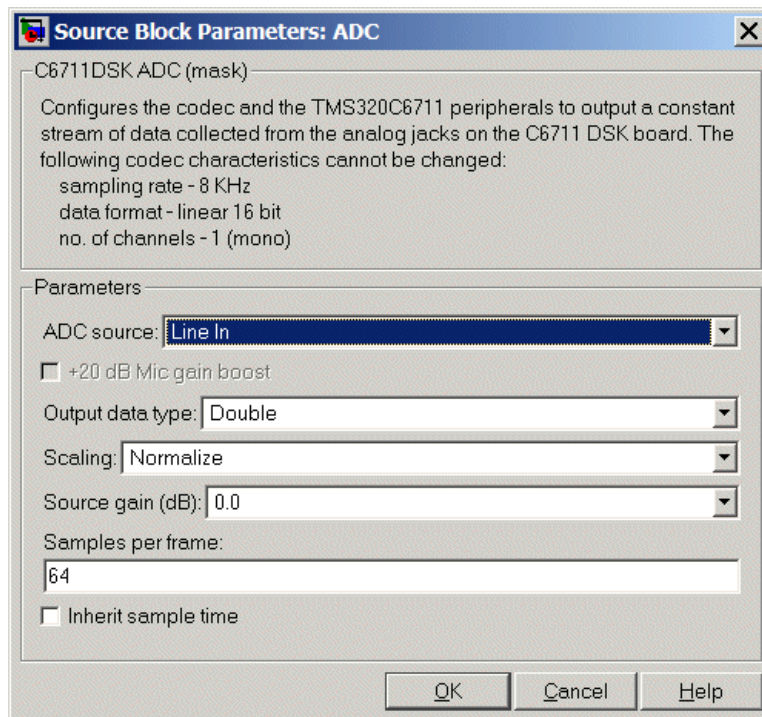
<b>Option (Continued)</b>	<b>Affected Hardware</b>
<b>Stereo</b> (C6701 EVM ADC only)	Codec
<b>Word length</b>	Codec

### Selecting the ADC Source

When you set up your target to accept input for your model, you tell the hardware where the input to the codec comes from. Selecting **Line in** and **Mic in** on the C6701 EVM corresponds to the two different input connectors on the board, with different input signal levels expected. On the DSK, the **Line in** and **Mic in** options use the same connector, but generate different signal levels to the codec. Both boards include the **Loopback** option that feeds the output from the DAC back to the ADC input.

### Choosing the Sample Rate

To open the Block Parameters dialog box, right-click the ADC block in your Simulink model and select **Block Parameters** from the context menu. You see the C6711 ADC block dialog box presented below.



Select your sample rate from the list. 5521 Hz is the lowest rate and 48000 Hz is the highest. You cannot set a sample rate that is not on the list. The available rates are derived from the clocks on the codec and cannot be changed.

The C6711 DSK uses a fixed sample rate of 8 KHz.

For many applications, your sample rate should reflect the standards for the industry. For example, if you are developing a professional audio application, working with digital audio tape (DAT) processes, or developing applications for high fidelity audio use, consider using 48000 Hz sampling rate in your model.

For applications used by CD players and sound cards in personal computers, the standard sampling rate is 44.1 KHz, and some of the lower rates such as 22.05 or 16 KHz. Moving Picture Expert Group (MPEG) audio applications often select a 32 KHz sampling rate.

When you are developing an application for speech, telephony, or “toll quality” speech processing, the 8 KHz sampling rate, paired with an 8-bit data format that uses a compressed format such as A-law, best matches current standards.

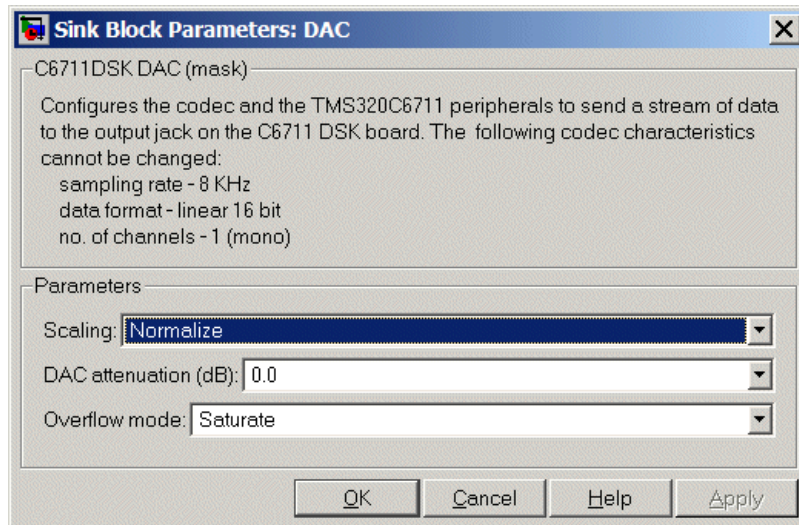
### Configuring DAC Blocks

In most cases, DAC blocks inherit attributes from the ADC block in the model, or from the previous nonvirtual block. In addition, you can choose to use the overrun indicator function provided in the Embedded Target for TI C6000 DSP.

Two of the configuration options for the block affect the codec. The remaining options relate to the model you are using in Simulink and the signal processor on the board. In the following table, you find each option listed with the hardware affected.

<b>Option</b>	<b>Affected Hardware</b>
<b>DAC attenuation</b>	Codec
<b>Overflow mode</b>	Digital Signal Processor
<b>Scaling</b>	Digital Signal Processor

When you double-click the C6711 DSK DAC block, you see the dialog box shown here.



### Selecting the Scaling

Select the scaling that best suits your model and your output device. For most applications, choose the scaling to match the setting of the ADC block if your model uses it.

Scaling defines the range of the input values from the codec. Independent of your setting for **Scaling**, signal values are stored as floating-point data. In Normalize mode, the signal ranges from -1 to 1 at the output of the DAC block.

### Selecting the Overflow Mode

Models running on the target can encounter situations where calculations exceed the range represented by the data type. The **Overflow mode** option on the Sink Block Parameters: DAC dialog box lets you select how the block responds to overflow conditions. Select one of the following settings:

- Saturate—Arithmetic results that fall outside the representable range of the selected data type are limited to the largest or smallest values. Saturated values are set to the nearest value that the data type can represent, either

the largest representable value in the case of arithmetic overflow or the smallest representable value in the case of arithmetic underflow.

Before input data reaches the codec, the Embedded Target for TI C6000 DSP uses an efficient linear assembly algorithm to determine whether the input values exceed the representable range of your selected data type. When input values exceed the range of the data type, the saturation algorithm clips the input to the nearest representable value and passes the clipped, or *saturated*, value to the codec.

- **Wrap**—Arithmetic results that fall outside the numeric range of the selected data type are wrapped into the range of the data type. The wrapping algorithm uses modular arithmetic relative to the largest or smallest representable number to determine the value of the result after wrapping.

### Configuring LED Blocks

You use the LEDs on the evaluation module as indicators for your process. For example, you might use an LED to indicate that your algorithm has completed a specified calculation or reached a particular point in the processing.

To use an LED as an indicator, add an LED block to your model, and send a nonzero signal to the block to light the specified LED—either internal or external. Any nonzero scalar sent to the LED block lights the LED and keeps it lit until the block receives a scalar with zero value. The zero value scalar turns off the selected LED.

Although the C6711 DSK offers three user-defined LEDs, the C6711 DSK LED block treats all three as one LED, enabling them as a group. For this reason you can include only one C6711 DSK LED block in a model.

### Using the Overrun Indicator Feature

When your digital signal process application cannot complete the calculations and data manipulations required to yield a result before the available clock cycles expire, your model can generate unreliable data. Failing to complete an algorithm is called **overrunning**, and is one of the most important errors to identify and eliminate in digital signal processing design and implementation.

The Embedded Target for TI C6000 DSP provides a pair of overrun indicator options—**Overrun action** and **Overrun notification method**—that you use to determine what happens when your application overruns and how or if to notify you when your process runs out of processing time before it completes its

tasks. To signal that your algorithm has overrun its limits, the Embedded Target for TI C6000 DSP can turn on the external LED on your C6711 DSK and leave it on until you reset the evaluation module.

The overrun feature can also print a message that the overrun occurred to the standard output device—`stdout` (or the message log if your application uses DSP/BIOS). One more option lets you both light the LED and print a message.

### Limitations of the Overrun Indicator

In two cases, the overrun indicator does not work:

- In multirate systems where the rate in the process is not the same as the base clock rate for your model. In this case, the timer/scheduler in the Embedded Target for TI C6000 DSP provides the interrupts for setting the model rate and you cannot use the overrun indicator.
- In models that do not include ADC or DAC blocks. In this case, the timer/scheduler provides the software interrupts that drive model processing.

To detect overrun conditions, the generated C code sets and checks a persistent flag during each iteration of the direct memory access (DMA) interrupt service routine.

On the C6711 DSK, the software turns on all three user-defined LEDs on the board to indicate an overrun condition.

---

**Note** The **Overrun notification method** selections that turn on the LEDs use the external LED or user-defined LEDs to signal model conditions. If you are using the overrun indicator, consider not using an LED block to trigger the user LEDs on the C6711 DSK until you stop monitoring your process for overrun conditions.

---

To enable the overrun indicator, choose one of three options for **Overrun action** to determine how to respond to an overrun condition in your model:

- None—Your model does not respond to overrun conditions during processing.
- `Notify_and_continue`—When your model runs out of clock cycles before completing enough of the processing algorithm, the overrun indicator

executes the option you chose for **Overrun notification method**. The model continues to run without pause.

- **Notify\_and\_halt**—When your algorithm runs out of clock cycles before completing the required calculations and manipulations, the model stops executing and notifies you about the overrun using the method you select for the **Overrun notification method**.

Overrun notification method provides three ways to tell you when an overrun has occurred:

- **Print\_message**—Print a message to stdout, or the message log when your application uses DSP/BIOS
- **Turn\_on\_LEDs**—Illuminate all the user LEDs on the C6711 DSK
- **Print\_message\_and\_turn\_on\_LEDs**—Light the LEDs and print a message

### Configuring Reset Blocks

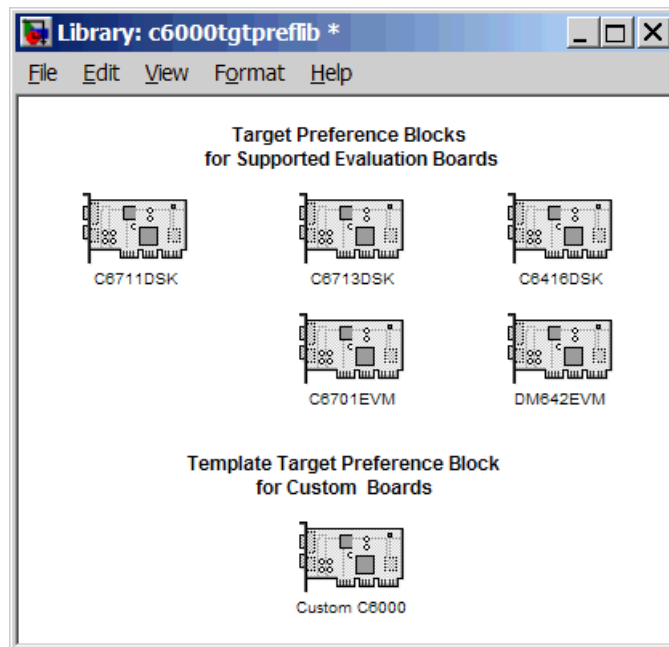
Each target library offers a block that performs a software reset of the appropriate board—a Reset block. While they are blocks, Reset blocks do not require input; they do not provide output; and they do not need to be connected to any other block.

When you add a Reset block to a model window, the block provides single-click access to resetting your board. Click on the block in your model and your target processor returns to its original state, with the memory locations, registers, and other peripherals reset to their default values before you loaded or ran a program.

### Configuring Target Preferences Blocks

One block is required in all C6000 targets models—a block from the Target Preferences library in the C6000 Block (c6000lib) library. Adding one of these blocks to your model provides direct access from the model to the C6000 Target Preferences dialog box, as shown in the following figure. From this dialog box, you choose your target board and processor, define your target board memory map, and set data and code sections, such as compiler, DSP/BIOS, and user sections.





To target a model to a C6000 processor-based target, add only one of the target preferences blocks to the model. The following table describes these blocks. The blocks with specific board names represent blocks that are preconfigured with

default values to match the attributes of the named Texas Instruments DSP starter kits and evaluation modules:

<b>Target Preferences Block</b>	<b>Description</b>
Custom 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, section layout, and DSP/BIOS.
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.
C6711DSK	Sets default values for targeting the C6711 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.
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.

---

**Note** Every model that you target to a C6000 processor-based board must include one target preferences block. An error is reported if you do not include the target preferences block or include more than one.

---

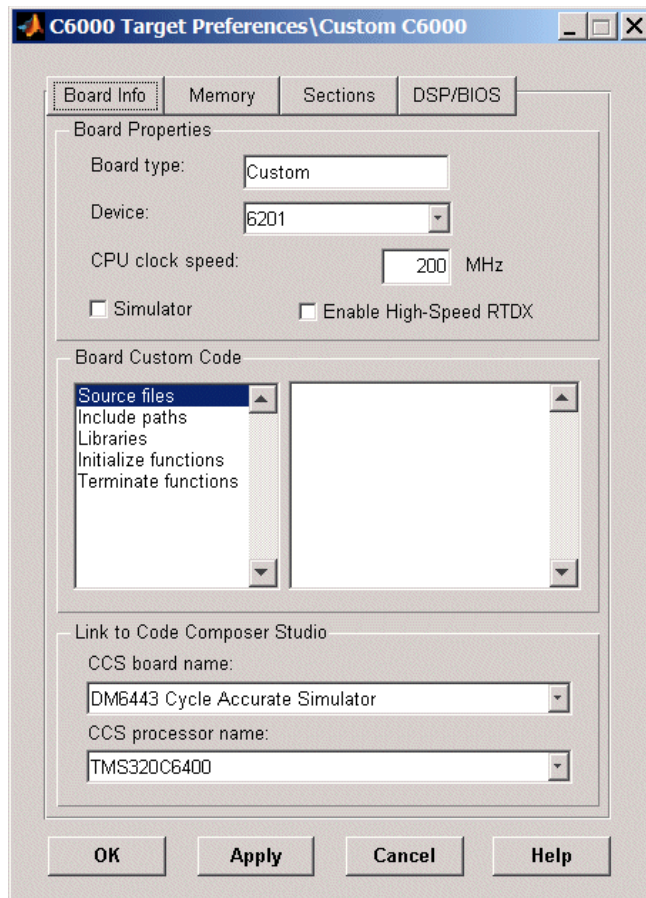
If the target preferences block in your model indicates a target name which does not exist in CCS Setup, Embedded Target for TI C6000 DSP places the new project into the CCS IDE window for board 0 as defined in the CCS Setup Utility when you generate a project or code from the model.

When you have more than one target configured for CCS, such as boards 0 and 1, this default behavior may not put the project in the target you intend. Warning messages such as

```
File does not match the target type.
```

may indicate that the project is in the wrong, or an unintended, target location.

Double-clicking one of the above blocks in a model opens the C6000 Target Preferences dialog box, shown in the following figure in the custom board configuration.



All target preferences block dialog boxes provide tabbed access to panes that include options you set for the target processor and target board:

- **Board info**—Select the target processor and board, set the clock speed, and identify the target.
- **Memory**—Set the physical memory layout on the target processor and board.
- **Sections**—Determine the allocation of the code and data sections in the target memory such as where to put the DSP/BIOS and compiler sections.

- DSP/BIOS—Set the DSP/BIOS options for the task manager.

## Board Info Pane

The following options appear on the **Board Info** pane for the C6000 Target Preferences dialog box.

### Board Type

Lets you enter the type of board you are targeting with the model. You can enter **Custom** to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as **C6711DSK**. If you are using one of the explicitly supported boards, choose the target preferences block for that board and this field shows the proper board type.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows 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 may 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 speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. If your model does not include ADC or DAC blocks, or when the processing rates in your model change (the model is multirate), you automatically invoke the timer to handle and create interrupts to drive your model.

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. C6711 DSK hardware uses a fixed clock rate of 150 MHz;

you can not change the clock rate. Other C6000 processors allow different clock speeds.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs. You are telling the software timer what rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate.

Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

- $100000000/1000 = 1$  Sine block interrupt per 1000000 clock ticks

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

### **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

### **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for this model.

### **CCS Board Name**

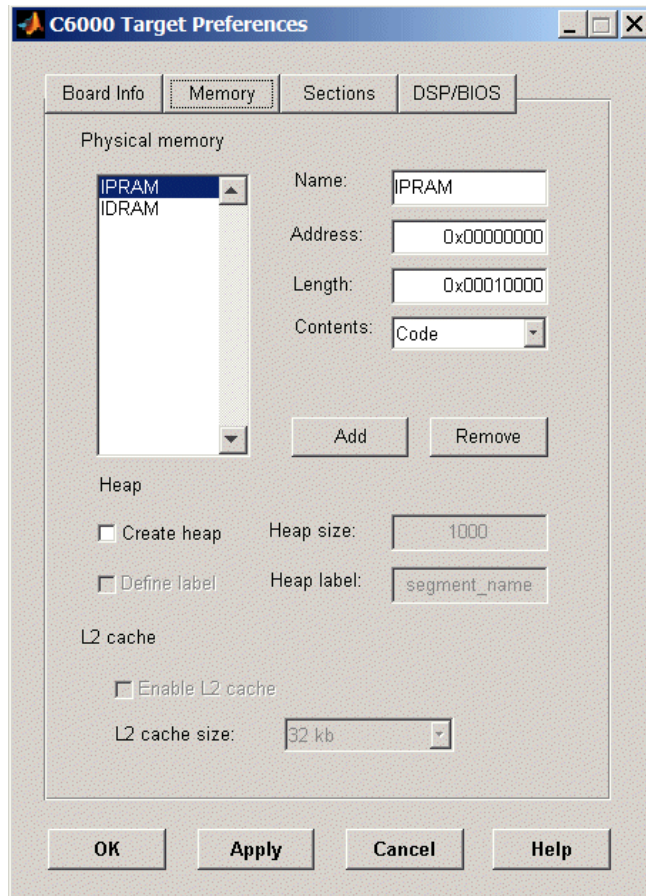
Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one that you are targeting your code for.

### **CCS Processor Name**

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.

### **Memory Pane**

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



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

- **Physical Memory**—specifies the processor and board memory map
- **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

These options may affect the options on the **Sections** pane. Selections you make here can change how you configure options on the **Sections** pane.

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



## 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.
- C6711DSK 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** details 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 may 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 may 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 on the **Physical memory** list and click **Remove** to delete the segment.

## Create Heap

If your processor supports using a heap, as do the C6711 or C6701, 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

Enabled by selecting **Define label**, you use this option 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.

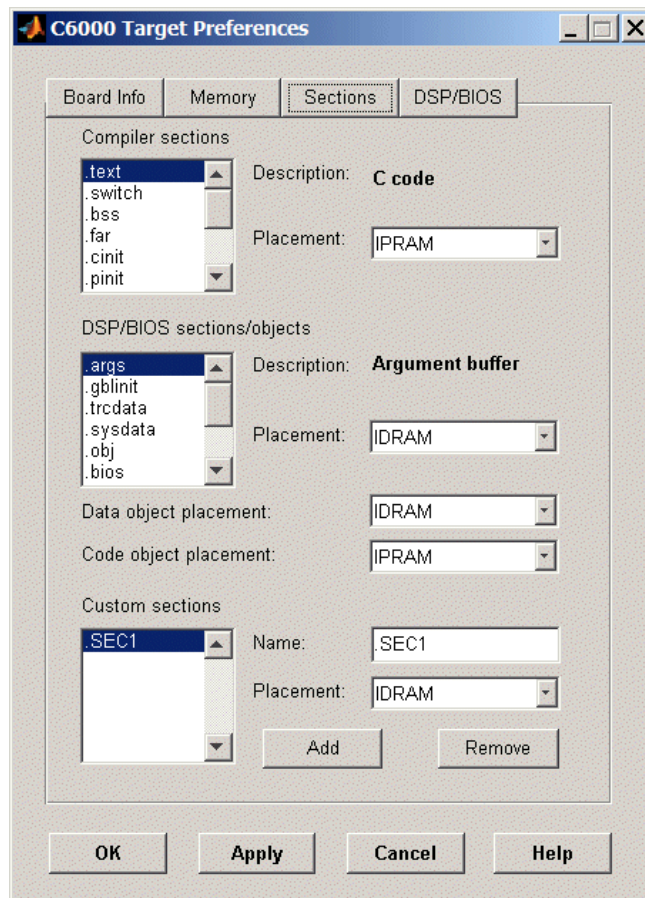
### **L2 Cache size**

When you enable the L2 cache, use this list to determine the size of the cache allotted. 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 online help. Most of the definitions and descriptions in this section come from CCS.



Within this pane, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

The following table provides brief definitions of the kinds of sections in the lists.

**Note** All sections do not appear on both lists. The list on which the string appears is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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	The 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.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules.
.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.
.system	Compiler	Dynamically allocated object in the code. Contains 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. Most of the definitions and descriptions in this section come from the online help for CCS.

### Compiler Sections

During program compilation, the C6000 compiler 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 **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 presently allocated in memory.

### **Description**

Provides a brief explanation of the contents of the selected entry on 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, get 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 location 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. Instead, you can 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.

## DSP/BIOS Pane

Options on this pane let you specify how to configure tasking features of DSP/BIOS.

This pane provides options the asynchronous task scheduler uses when you select the **Incorporate DSP/BIOS** option in the configuration set for your model. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their block dialog boxes so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

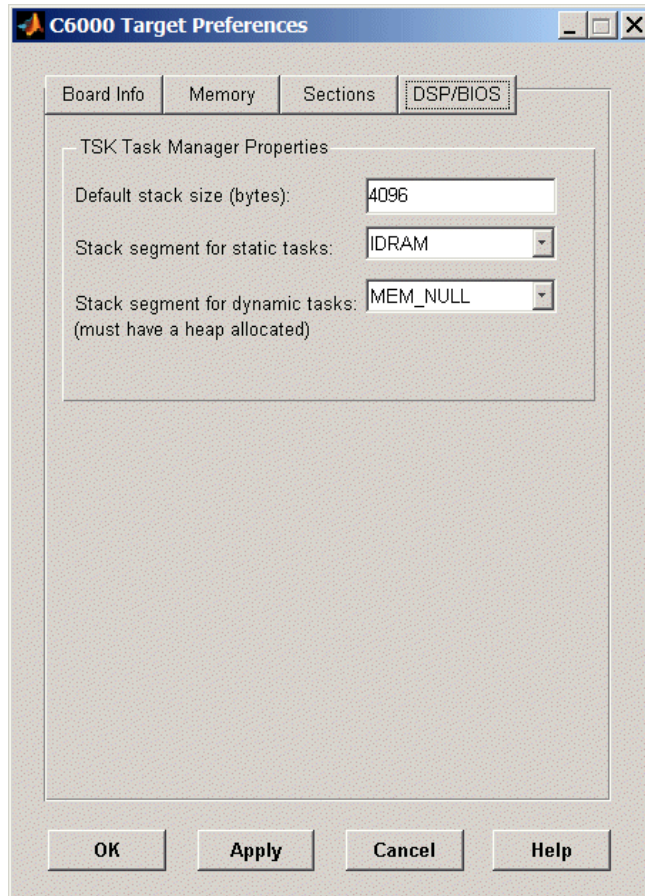
The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you choose not to use DSP/BIOS in your project, by clearing the **Incorporate DSP/BIOS** the configuration set for your model, you disable the options in this pane and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler. It does not create or use DSP/BIOS tasks.

For more information about tasks, refer to the Code Composer Studio online help. Most of the definitions and descriptions in this section come from CCS.

With controls on this pane, shown in the next figure, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. Note that the Sections pane includes DSP/BIOS configuration options as well.

The options specify the stack use and locations on the stack for static and dynamic tasks.



### Default stack size (bytes)

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the DSP/BIOS stack in bytes allocated for each task. 4096 bytes is the default value. You can set any size up to the limits for the processor. Set the stack size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, this might cause the

task to write into other memory or data areas, possibly causing unpredictable behavior.

### **Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers options SDRAM and ISRAM for locating the stack in memory, with SDRAM as the default section. The Memory pane provide more options for the physical memory on the processor.

### **Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, SDRAM is the only valid stack location in memory.

### **Code Generation for Subsystems**

Generating code for most models is a matter of adding the target preference block to your model, configuring the target options, and then generating code with Real-Time Workshop.

When you generate code for a subsystem in a model, as opposed to generating code for the entire model, you must treat the target preferences block differently. Real-Time Workshop lets you choose to generate code or build executables for entire model or for subsystems of the models. For more about this feature, refer to “Generating Code and Executables from Subsystems” in your Real-Time Workshop documentation in the online Help system.

With the Embedded Target for TI C6000 DSP, you must do one thing before you generate code or build an executable from a subsystem—you must add the target preferences block to the subsystem.

To generate code for a subsystem:

- 1 Set the top model simulation parameters for the model, such as selecting the target, the make file, the solver, and more.

- 2 Put a target preferences block for your target in the subsystem from which you want to generate code.
- 3 Configure the target preferences block options as required.
- 4 Right-click on the subsystem and select **Real-Time Workshop > Build Subsystem** from the context menu.

The Build Subsystem window opens, displaying a list of the subsystem parameters. The upper pane displays the name, class, and storage class of each variable or data object that is referenced as a block parameter in the subsystem. When you select a parameter in the upper pane, the lower pane shows the blocks that reference the parameter and the parent system of each block.

- 5 After you select the parameters and make adjustments as needed, click **Build** to start the code generation and build process.

---

**Note** Putting the target preferences block in the subsystem disables automatic model parameter setting. You do not see a confirmation dialog about the build process when you generate code for the subsystem.

---

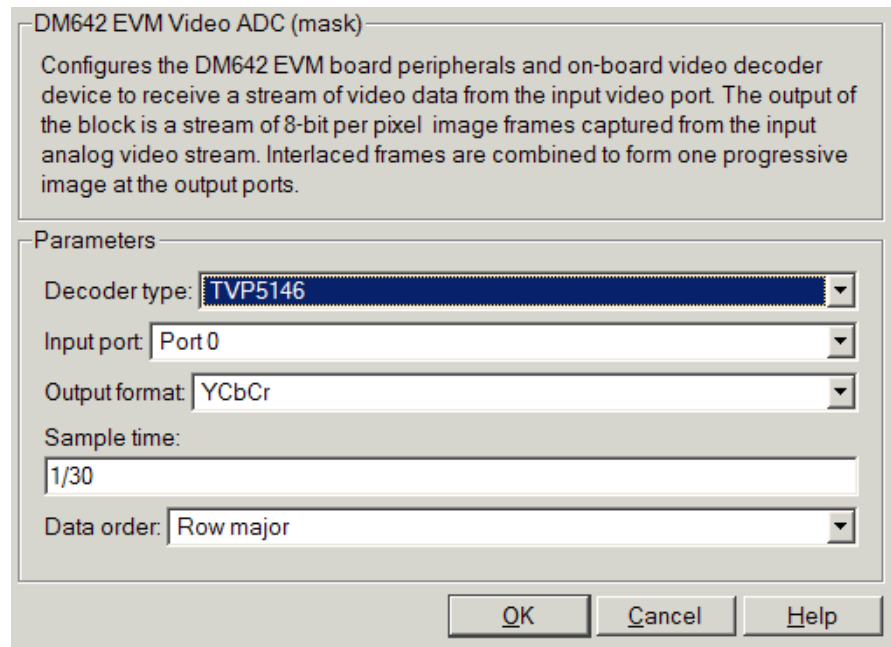
## **Configuring DM642 EVM Video ADC and Video DAC Blocks**

Preparing the Video ADC and Video DAC blocks for the DM642 EVM is quite different from configuring the ADC and DAC blocks for the other supported hardware. Rather than being just analog-to-digital converters, or digital-to-analog, the DM642 EVM blocks allow you to capture and display video data.

While conversion is a part of what they do, the configuration process sets up the video formats that the video ADC (capture) block accepts and the video DAC (output) block provides for display.

### Configuring the Video ADC or Video Capture Block

To capture video from the video inputs on your DM642 EVM, add the DM642 EVM Video ADC block to capture the video input. Options in the block let you set the output format and output mode of video that leaves the block for processing in your model. Setting the options configures the peripherals on your board, and the SAA7115 or TVP5146/5150 video decoders to capture and reformat video for output from the block for further processing.



From the available options in the dialog box, you see that you do not need to configure the block for the type of video input. The block accepts whatever supported video you provide at the video input ports. Options in the dialog box let you specify how to output the video from the block to downstream model

elements and which video decoder you DM642 EVM uses. The options are described in the following table.

<b>Option</b>	<b>Affected Hardware</b>
<b>Decoder type</b>	Video decoder, either SAA7115 or TVP5146/5150
<b>Input port</b>	Video decoder
<b>Output mode</b>	SAA7115H video decoders (available when you set <b>Decoder type</b> to SAA7115)
<b>Output format</b>	SAA7115H video decoders
<b>Sample time</b>	Clock rate and decoders
<b>Data order</b>	Buffers and decoder

### Setting the Decoder Type

Your selection from the list 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 an option—**Output Mode**. Generally, older DM642 EVM boards use the SAA7115 decoder option. Newer boards use the default setting TVP5146 decoder. Refer to “Identifying Your DM642 EVM Board Revision” on page A-6 for information about identifying the revision of your DM642 EVM.

### Choosing the Input Port

Directs the block to capture video from either the 0 or 1 video input port on the DM642 EVM based on whether you select 0 or 1. The block does not support port 2 for video input.

### Selecting the Output Mode (SAA7115 Decoder Only)

In the DM642 Video ADC dialog box, the option **Output mode**, available when **Decoder type** is SAA7115, sets the size of the frames the block outputs in pixels and lines, and the frame rate. It also determines how the data frames get assembled into images. Choosing a mode from the list tells the video decoder to take the input video stream and convert it to the mode/size you select. Mode

selection supports the following video standards, shown in the **Output mode** list.

<b>Output Mode</b>	<b>Description</b>
NTSC 720x480	525 horizontal lines stacked on top of each other, with varying numbers of lines making up the horizontal resolution. There are 59.94 fields displayed per second. Each field is a set of even lines, or odd lines. Displaying the even and odd fields sequentially by interlacing them creates each full 60 field frame. One full frame is displayed about every 1/30 of a second (30 Hz refresh rate).
NTSC 640x480	Scales the output to standard (SDTV) mode.

### Setting the Output Format

Choose one of the following color representations according to what your model and algorithm require.

<b>Digital Output Format</b>	<b>Description</b>
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	Output from the block includes one luminance channel Y (essentially the black/white signal) and two chrominance (color) channels Cb and Cr to represent the color image data per pixel. This is the digital standard color space DVDs use.
Y	Black and white video. No color or chromaticity values.



Your selection determines how the block represents color data in the output.

### Sample Time

Sample time tells the block how often to take frames from the video decoder and buffers. While NTSC video runs at 30 frames/s (1/30 s sample time), you can sample at any rate below or above the TV rate. Remember that sampling times that are not 1/30th of a second may either capture incomplete frames, when **Sample time** < frame time, or miss frames/fields when **Sample time** > frame time.

Note that

- The sample time you specify becomes the DM642 timer period that drives the execution of your model.
- Your generated application is not synchronized with the input video signal—the application always runs on the processor timer.

### 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.

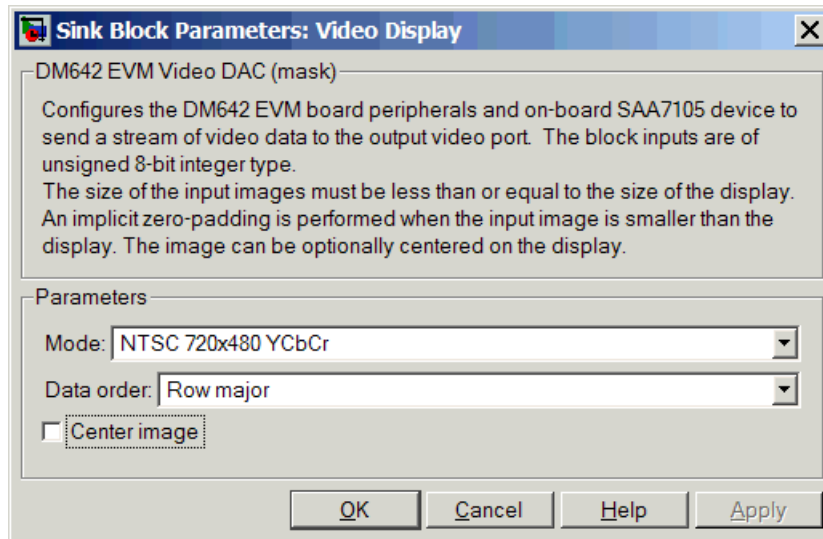
- Row major—store video data in row major order. This is the default setting and matches most video data.
- Column major—store video data in column major order. Simulink® and MATLAB both use this format to store images and matrices.

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 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.

### Configuring the Video DAC or Video Output Block

To provide video output from your running process on your target, add the Video DAC block to your model. Options for the block let you determine the video format for the output and center the image in the display, as shown here.



You see from the options that the block provides parameters that control the video encoder on the DM642 EVM. The following table describes the hardware affected by these options.

Option	Affected Hardware
<b>Mode</b>	SAA7105 video encoder
<b>Data order</b>	SAA7105 video encoder
<b>Center image</b>	SAA7105 video encoder

### Selecting the Mode

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 your 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.

<b>Analog Output Mode</b>	<b>Description</b>
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.

### 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 both 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 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.

Selecting the **Center image** option instructs the block to center the output image on the display. Note that 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 can not see) is centered in the display area.

### **Creating DSP Application Models for Targeting**

Create your real-time model for your application the way you create any other Simulink model—by using standard blocks and C-MEX S-functions. Select blocks to build your model from any of the following sources:

- Use the ADC, DAC, and LED blocks from libraries in the C6000lib block library to handle input and output functions for your target hardware
- Use blocks from the TI C62x DSP library in the C6000lib block library to build fixed-point models
- Use blocks provided with the Real-Time Workshop
- Use blocks from the Signal Processing Blockset
- Use discrete time blocks from Simulink
- Use blocks from any other blockset that meet your needs and operate in the discrete time domain
- Add a target preferences block from the Target Preferences library (c6000tgtpreflib) to configure your generated code for your target processor

### **Using Logging in Your DSP Applications**

Simulink offers various data logging capabilities in the Configuration Parameters dialog box for your model. Found on the Data Import/Export pane of the Configuration Parameters dialog box, the implicit logging options let you specify how and when Simulink logs model operations and gets data from your workspace.

When your model is running on the target, it cannot communicate directly with MATLAB. Configuration options that tell your model to send or retrieve data from your MATLAB workspace do not work and use processing time to no benefit.

To avoid these effects, do not enable options on the Data Import/Export pane in the Configuration Parameters dialog box in your model.

### Turning Off Logging in Your Model

Follow this procedure to disable the logging options in your existing Simulink model:

- 1 Select **Simulation > Configuration Parameters** from the menu bar in your model.
- 2 Click Data Import/Export in the **Select** tree to access the Data Import/Export pane.
- 3 Clear the options in the **Load from workspace** and **Save to workspace** fields.
  - **Input**
  - **Initial state**
  - **Time**
  - **States**
  - **Output**
  - **Final states**

Instead of using the Data Import/Export options in **Configuration Parameters** to eliminate logging during code generation and operation, run `dspstartup`

from your MATLAB command prompt before you create new Simulink models. Running `dspstartup` disables the Data Import/Export options in the Configuration Parameters dialog box for your new models.

### Generating Code from Real-Time Models

This section summarizes how to generate code from your real-time model. For details about generating code from models in Real-Time Workshop, refer to your Real-Time Workshop documentation.

You start the automatic code generation process from the Simulink model window by clicking **Build** in the Real-Time Workshop pane of the Configuration Parameters dialog box. The code building process consists of these tasks:

- 1** Real-Time Workshop invokes the function `make_rtw` to start the Real-Time Workshop build procedure for a block diagram. `make_rtw` invokes the Target Language Compiler to generate the code and then invokes the language specific make procedure.
- 2** `gmake` builds file `modelName.out`. Depending on the build options you select in the Configuration Parameters dialog box, `gmake` can download and execute the model on your TI target board.

## Schedulers and Timing

The next sections describe how the Embedded Target for TI C6000 DSP provides timing and scheduling for generated code running on your target.

### 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.

---

**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 many 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 Embedded Target for TI C6000 DSP library to your model, listed here.

#### **Blocks in the DSP/BIOS Library**

- HWI—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 C6000 DSP Core Support Library**

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

- 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.

---

**Note** The preceding 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.

---

### 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 Embedded Target for TI C6000 DSP 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 Embedded Target for TI C6000 DSP 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

Embedded Target for TI C6000 DSP facilitates modeling and automatically generating code for asynchronous systems by using the following scheduling blocks:

- 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

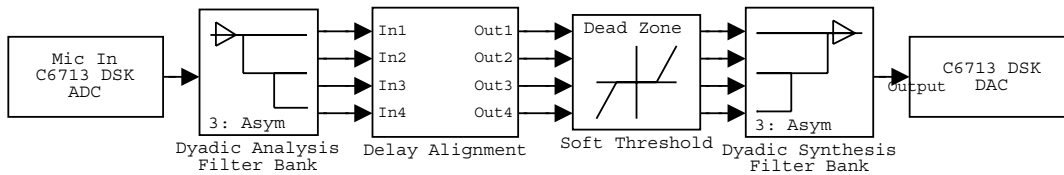
Now you can use an asynchronous (real-time) scheduler for your target application. Earlier versions of the Embedded Target for TI C6000 DSP used a synchronous CPU timer interrupt-driven scheduler. With the asynchronous scheduler you can define interrupts and tasks to occur when you want them to by using blocks in the following libraries:

- C6000 DSP Core Support
- DSP/BIOS Library

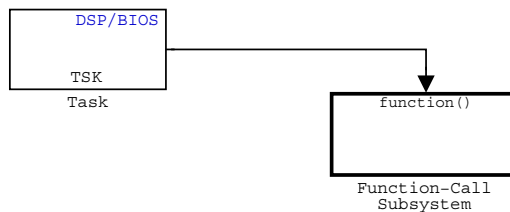
Also, you can schedule multiple tasks for asynchronous execution using the blocks in the C6000 DSP Core Support and DSP/BIOS Library block libraries.

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

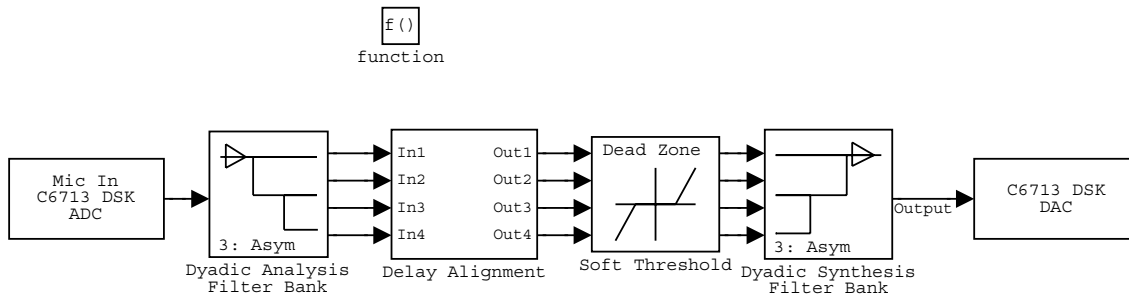
### Before



### After



## Model Inside the Function Call Subsystem Block

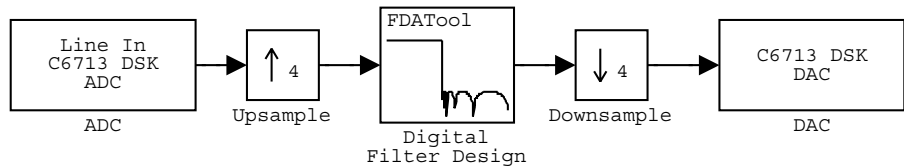


**Compatibility Consideration.** The V3.0 changes in the real-time scheduler can break some existing multirate models that contain codec blocks such as the ADC and DAC. The models affected contain at least one sample rate that is faster than the codec block rate. You do not run into this problem if all 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 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 TIC6000 Code Generation 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.

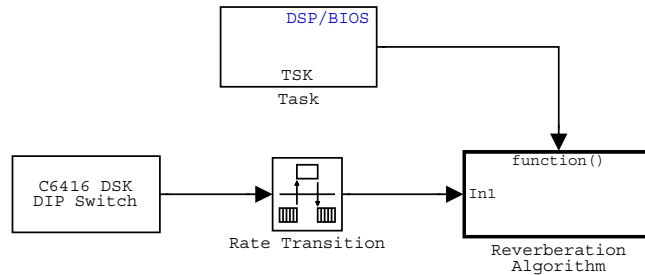
To avoid the audio problems in an existing model that you cannot update to the new scheduler, set the run-time overrun action for the model to either None or Notify\_and\_continue to prevent the program from overrunning.

### Uses for Asynchronous Scheduling

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

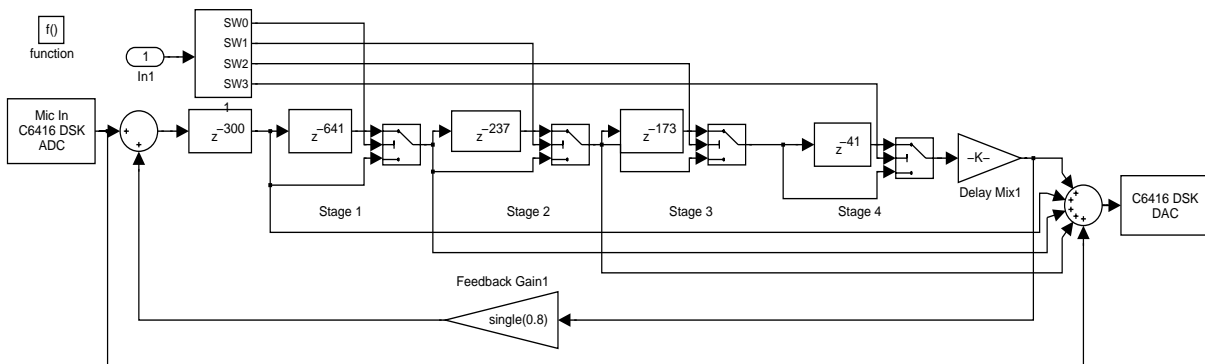
## Free-Running DSP/BIOS Task

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.



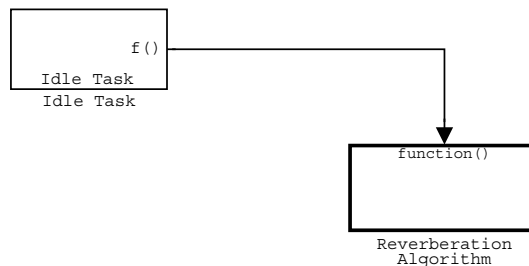
This model also illustrates how synchronous and asynchronous tasks can work together. The code generated for C6416 DSK DIP Switch block runs as a periodic task at the rate of 0.01 s. This is the only periodic task in the model. It runs out of the context of a DSP/BIOS task scheduled via a timer interrupt 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 TIC6000 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).

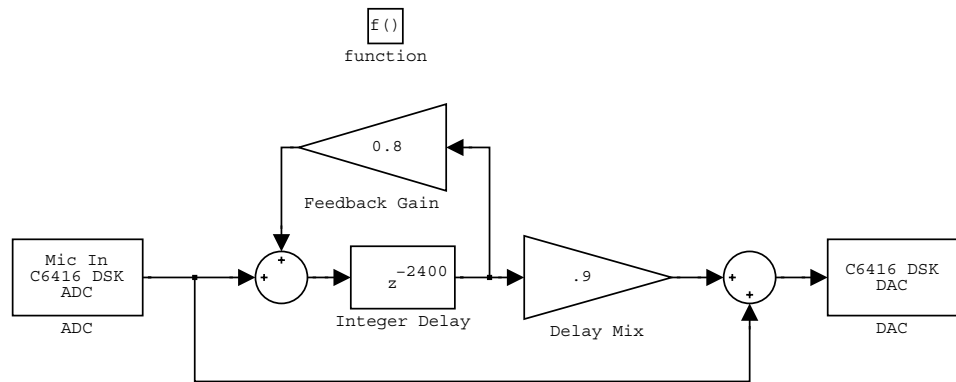
To ensure data integrity, Simulink Rate Transition blocks connect the C6416 DSK DIP Switch block with the reverberation algorithm. This transition is required because the blocks belong to different rate groups. If the synchronous and asynchronous parts of the model do not interact, the Rate Transition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

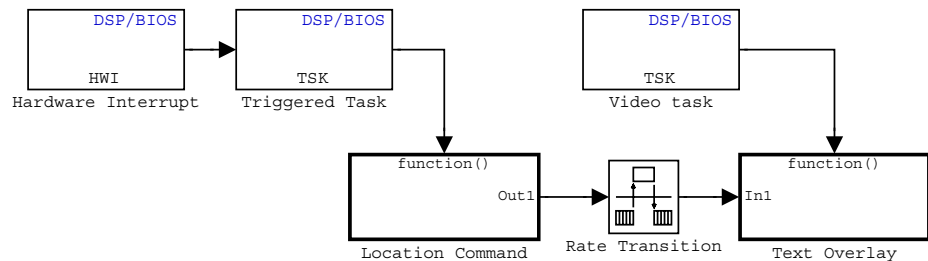


The function generated for this task normally runs in free-running mode—repetitively and indefinitely. However, the ADC and DAC blocks in this subsystem run in blocking mode. As a result, subsystem execution of the reverberation function is the same as the subsystem described for the Free-Running DSP/BIOS 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.



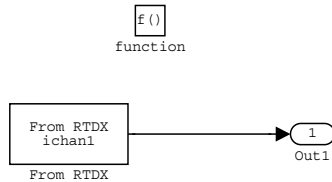
The DSP/BIOS Hardware Interrupt block installs an ISR function that signals a DSP/BIOS task to run when the ISR detects an RTDX interrupt. Signaling between the ISR and DSP/BIOS triggered task occurs via semaphores. This task receives an RTDX message carrying the location command for the downstream Text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

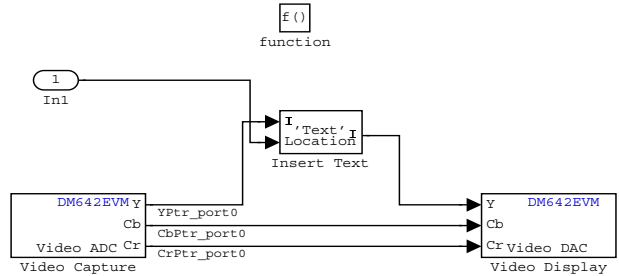
The text overlay subsystem is executed as for the Free-Running DSP/BIOS Task. A Rate Transition block connects the two subsystems that run at two different asynchronous rates to ensure data integrity. The execution of two

asynchronous rates is ordered based on the priority settings for the DSP/BIOS Task blocks.

### Location Command Subsystem

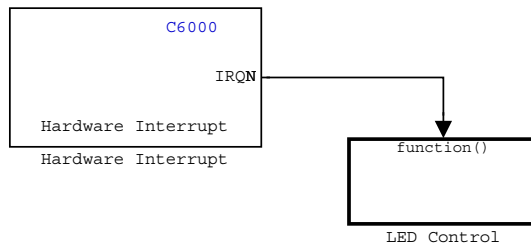


### Text Overlay Subsystem



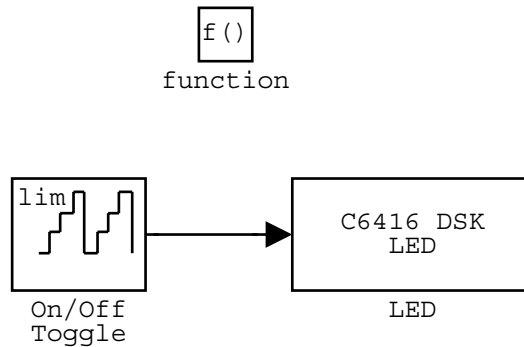
### Hardware Interrupt Triggered Task

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 section on Asynchronous Support in your Real-Time Workshop documentation in the online Help system.

# Setting Real-Time Workshop Options for C6000 Hardware

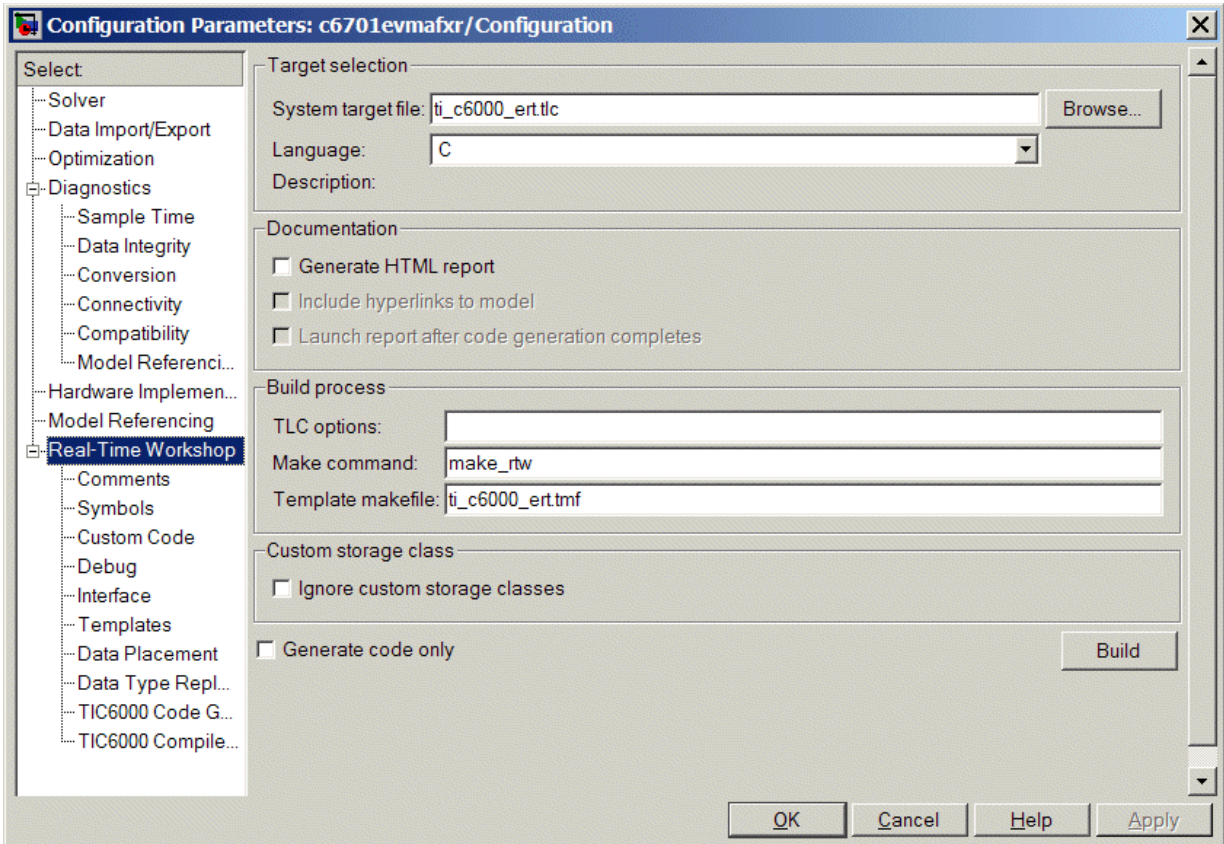
Before you generate code with the Real-Time Workshop, set the fixed-step solver step size and specify an appropriate fixed-step solver if the model contains any continuous-time states. At this time, you should also select an appropriate sample rate for your system. Refer to your Real-Time Workshop documentation for additional information.

---

**Note** Embedded Target for TI C6000 does not support continuous states in Simulink models for code generation. In the Solver options in the Configuration Parameters dialog box, you must select discrete (no continuous states) as the **Type**, along with Fixed step.

---

The Real-Time Workshop pane of the Configuration Parameters dialog box lets you set numerous options for the real-time model. To open the Configuration Parameters dialog box, select **Simulation > Configuration Parameters** from the menu bar in your model. The following figure shows the Real-Time Workshop categories when you are using the Embedded Target for TI C6000 DSP.



In the **Select** tree, the categories provide access to the options you use to control how Real-Time Workshop builds and runs your model. The first categories under Real-Time Workshop in the tree apply to all Real-Time Workshop targets including the target and always appear on the list.

The last categories under Real-Time Workshop are specific to the Embedded Target for TI C6000 DSP target `ti_c6000.tlc` and appear when you select any TI C6000 target.

- TI C6000 code generation — target-specific code generation options.
- TI C6000 compiler/linker — target-specific compiler and linker options. Also includes the target-specific run-time options.

When you select your target in Target Selection on the **Real-Time Workshop** pane, the options change in the tree. For the Embedded Target for TI C6000 DSP, the target to select is `ti_c6000.tlc`. Selecting either the `ti_c6000.tlc` or `ti_c6000_ert.tlc` adds the TI C6000-specific options to the **Select** tree.

The following sections present each Real-Time Workshop category and the options available in each.

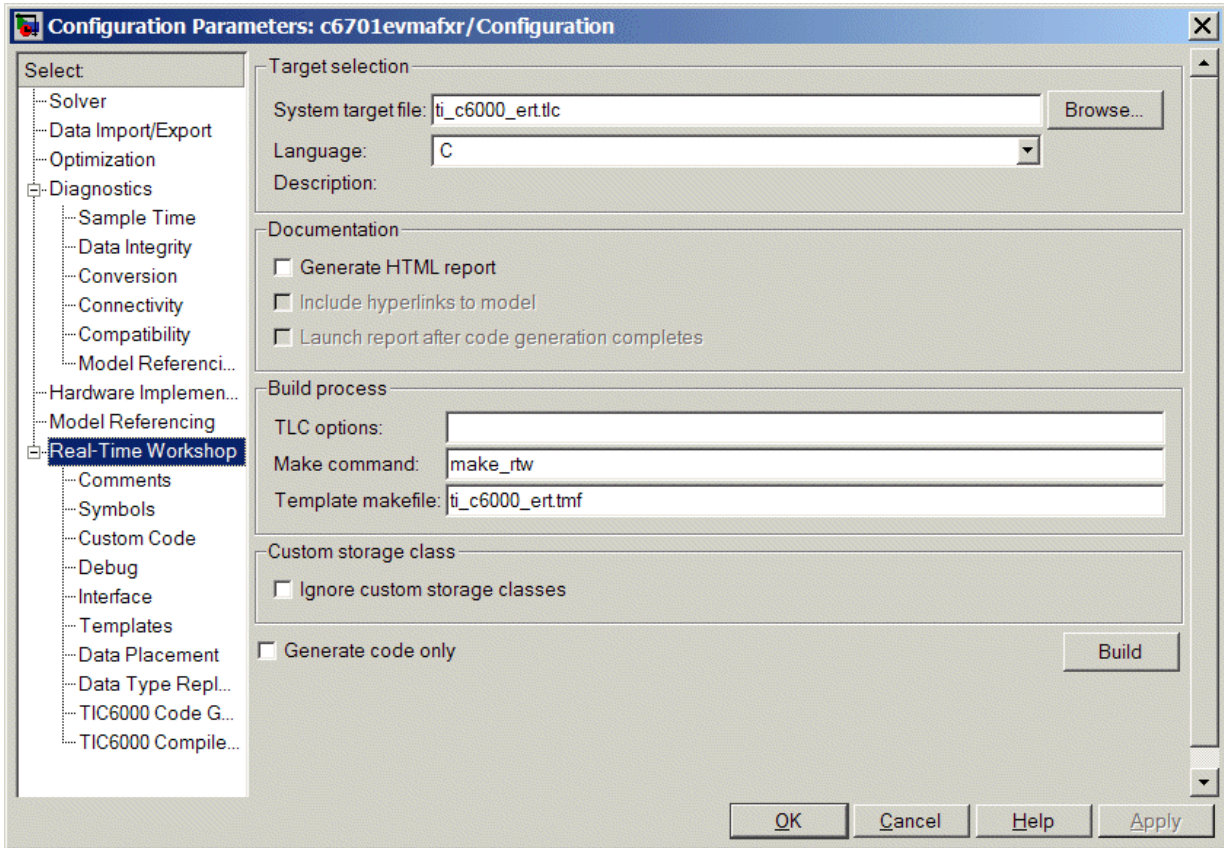
## Setting Real-Time Workshop Pane Options

Use the options in the **Select** tree under **Real-Time Workshop** to perform the following configuration tasks.

- Determine your target, either C6000 or some other target if you are not using the Embedded Target for TI C6000 DSP.
- Select your documentation needs.
- Configure your build process.
- Specify whether to use custom storage classes.

When you select the appropriate C6000 target (`ti_c6000.tlc`) in **System target file**, you enable the automatic board selection for your model. After that, opening the Configuration Parameters dialog box for your model triggers the automatic board and processor selection tool, which searches for your

C6701 EVM. If MATLAB and CCS cannot find a board that matches the C6701 EVM designation, you see an error message dialog box.



## Target Selection

### System target file

Clicking **Browse** opens the Target File Browser where you select `ti_c6000.tlc` as your Real-Time Workshop **System target file** for the Embedded Target for TI C6000 DSP. When you select your target configuration, Real-Time Workshop chooses the appropriate system target file, template makefile, and make command. You can also enter the target

configuration filename, and Real-Time Workshop fills in the **Template makefile** and **Make command** selections.

If you are using the Real-Time Workshop Embedded Coder software, select the `ti_c6000_ert.tlc` target in **System target file**.

## Documentation

### Generate HTML report

After you generate code, this option tells the software whether to generate an HTML report that documents the C code generated from your model. When you select this option, Real-Time Workshop writes the code generation report files in the `html` subdirectory of the build directory. The top-level HTML report file is named `modelName_codegen_rpt.html` or `subsystemname_codegen_rpt.html`. For more information about the report, refer to the online help for Real-Time Workshop. You can also use

```
docsearch 'Generate HTML report'
```

at the MATLAB prompt to get more information.

When you select **Include hyperlinks to model**, your HTML report adds hyperlinks to various features in your Simulink model. Hyperlinks within the displayed report let you view the blocks or subsystems that generated the report. Click the hyperlinks to view the relevant blocks or subsystems in your Simulink model.

### Launch report after code generation completes

Automatically opens a MATLAB Web browser window and displays the code generation report. When you clear this option, you can open the code generation report (`modelName_codegen_rpt.html` or `subsystemname_codegen_rpt.html`) manually in a MATLAB Web browser window or in another Web browser manually.

## Build Process

### Template makefile

Real-Time Workshop uses template makefiles to generate the makefile for building the executable file. During the automatic build process, MATLAB

issues the `make_rtw` command. `make_rtw` extracts information from the template makefile `ti_c6000.tmf` and creates the actual makefile `c6000.mk`. When Real-Time Workshop compiles the model, it uses the actual makefile to generate the compiled code for the target.

Set the **Template makefile** option to `ti_c6000.tmf` when you build your application for the C6000 target. If the template makefile shown in the option is not `ti_c6000.tmf`, click **Browse** to open the list of available system target files and select the correct file from the list. Real-Time Workshop then selects the appropriate template makefile.

### Make command

When you generate code from your digital signal processing application, use the standard command `make_rtw` as the **Make command**. In the **Build process** area in the Target configuration category, enter `make_rtw` for the **Make command**. Parameters you set in this dialog box belong to the model you are building. They are saved with the model and stored in the model file.

### Custom Storage Class

When you generate code from a model employing custom storage classes (CSC), make sure to clear **Ignore custom storage classes**. This setting is the default value for the Embedded Target for TI C6000 DSP and for Real-Time Workshop Embedded Coder.

When you select **Ignore custom storage classes**,

- Objects with CSCs are treated as if you set their storage class attribute to Auto.
- The storage class of signals that have CSCs does not appear on the signal line, even when you select Storage class from **Format > Port/Signals Display** in your Simulink menus.

**Ignore custom storage classes** lets you switch to a target that does not support CSCs, such as the generic real-time target (GRT), without having to reconfigure your parameter and signal objects.

### Generate code only

The **Generate code only** option does not apply to targeting with the Embedded Target for TI C6000 DSP. To generate source code without building and



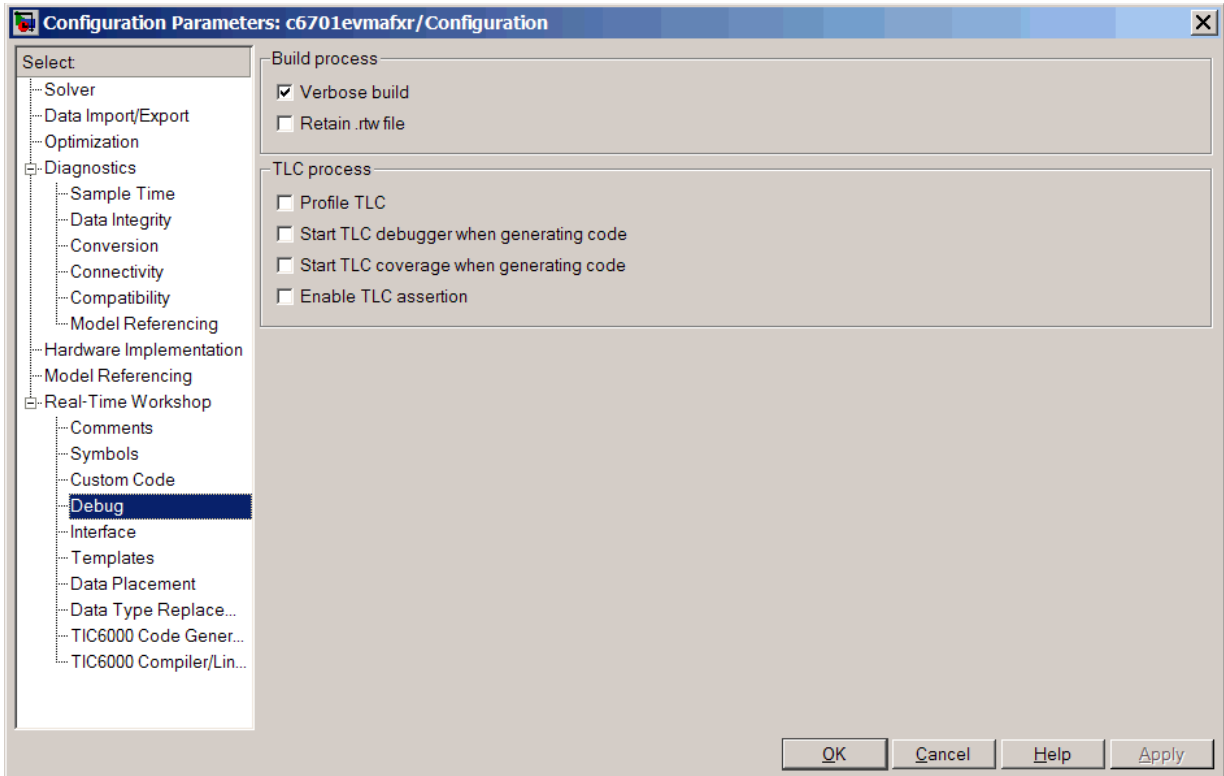
executing the code on your target, select TI C6000 runtime from the **Category** list in the Select tree. Then, under **Runtime**, select Generate code only for **Build action**. You cannot use DSP/BIOS features when you use the Generate code only option for the **Build action**.

## Debug Pane Options

Real-Time Workshop uses the Target Language Compiler (TLC) to generate C code from the *model.rtw* file. The TLC debugger helps you identify programming errors in your TLC code. Using the debugger, you can

- View the TLC call stack.
- Execute TLC code line-by-line and analyze and/or change variables in a specified block scope.

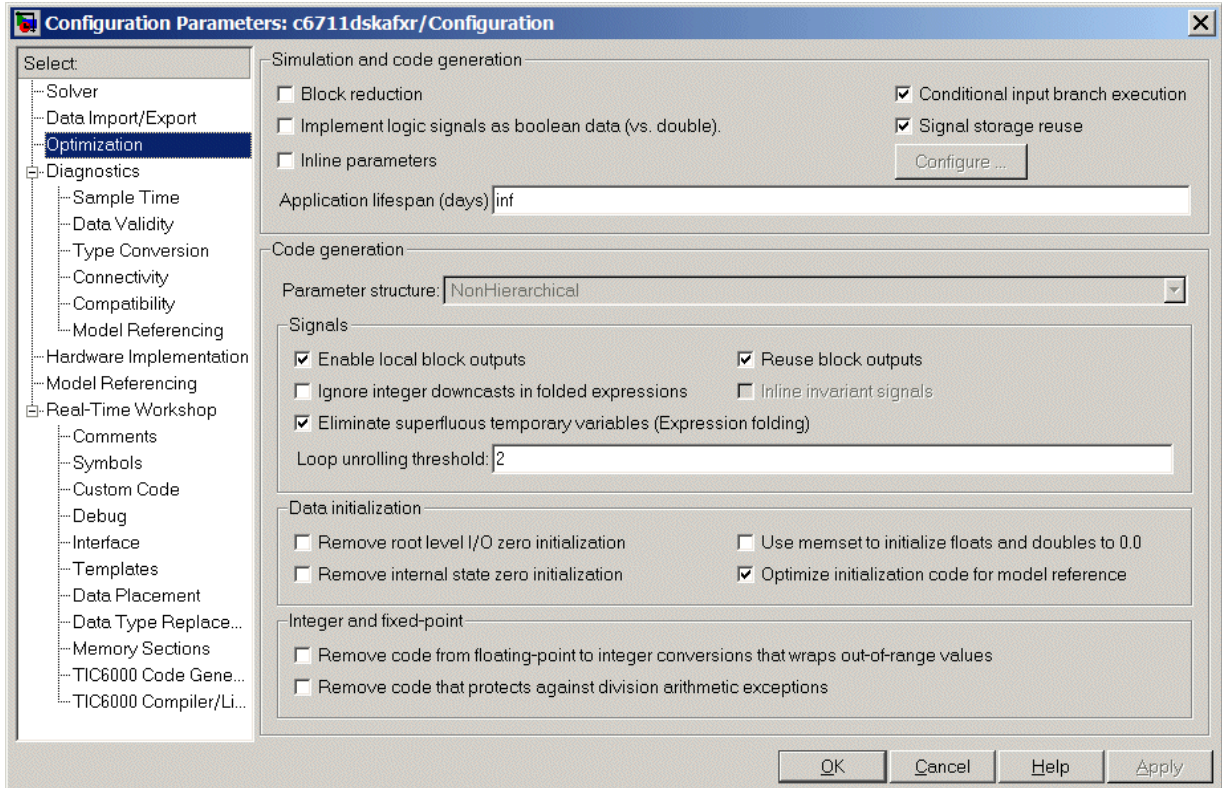
When you select Debug from the **Select** tree, you see the Debug options as shown in the next figure. In this dialog box, you set options that are specific to Real-Time Workshop process and TLC debugging.



For details about using the options in Debug, refer to the section “About TLC Debugger” in your Real-Time Workshop documentation.

### Optimization Pane Options

On the Optimization pane in the Configuration Parameters dialog box, you set options for the code that Real-Time Workshop generates during the build process. You use these options to tailor the generated code to your needs. Select Optimization from the **Select** tree on the Configuration Parameters dialog box. The figure shows the Optimization pane when you select the system target file `ti_C6000.tlc` under **Real-Time Workshop system target file**.



These are the options typically selected for Real-Time Workshop:

- **Conditional input branch execution**
- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse block outputs**
- **Eliminate superfluous temporary variables (Expression folding)**
- **Loop unrolling threshold**
- **Optimize initialization code for model reference**

For more information about using these and the other Optimization options, refer to your Real-Time Workshop documentation.

### TI C6000 Code Generation Pane Options

On the select tree, the TIC6000 Code Generation entry provides options in these areas:

- **Target Selection** — Export a handle to your MATLAB workspace
- **Code Generation** — Configure your code generation requirements, such as enabling DSP/BIOS
- **Runtime** — Set options for run-time operations, like the build action

#### Target Selection

When you use Real-Time Workshop to build a model to a C6000 target, Embedded Target for TI C6000 DSP makes a link between MATLAB and CCS. If you have used the link portion of the Embedded Target for TI C6000 DSP, you are familiar with function `ccsdsp`, which creates links between the IDE and MATLAB. This option refers to the same link, called `cc` in the function reference pages. Although MATLAB to CCS is a link, what it really is a handle to an object that contains information about the object, such as the target board and processor it accesses. In this pane, the **Export handle to MATLAB base workspace** option lets you instruct the Embedded Target for TI C6000 DSP to export the link to your MATLAB workspace, giving it the name you assign in **CCS handle name**.

#### Code Generation

From this category, you choose from options that define the way your code is generated:

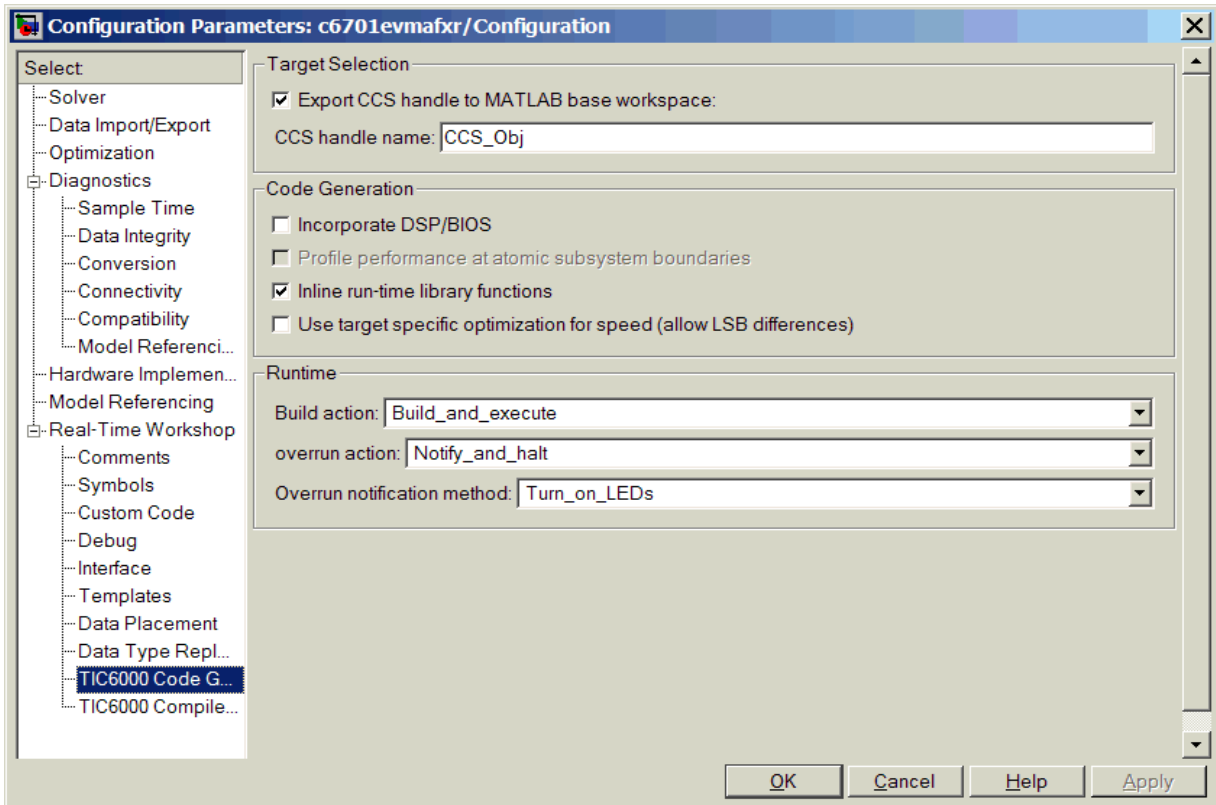
- **Incorporate DSP/BIOS**
- **Profile performance at atomic subsystem boundaries**
- **Inline run-time library functions**
- **Use target-specific optimization for speed (allow LSB differences)**

**Incorporate DSP/BIOS** determines whether the build process incorporates DSP/BIOS features in your generated code. When you select **Incorporate DSP/BIOS**, the build process inserts the DSP/BIOS options and files (the `.cmd` file that contains DSP/BIOS configuration information) in the generated code. The resulting code includes instrumentation based on DSP/BIOS objects. “Introducing DSP/BIOS” on page 3-2 provides details about the changes that occur in your generated code when you include DSP/BIOS.

If you are using the **Generate code only build** action option, you cannot use DSP/BIOS features in your generated code—do not select the **Incorporate DSP/BIOS** option here.

For profiling your generated code, the code generation options include the **Profile performance at atomic subsystem boundaries** option. When your model includes atomic subsystems, you can select this option to have Embedded Target for TI C6000 DSP generate a run-time report about the way your generated code performs when you run the code on your target. For more information about using code profiling, refer to “Profiling Generated Code” on page 3-10.

To allow you to specify whether the functions generated from blocks in your model are used inline or by pointers, **Inline run-time library functions** tells the compiler to inline each Signal Processing blockset and Video and Imaging blockset function. Inlining functions can make your code run more efficiently (better optimized) at the expense of using more memory. As shown in the figure, the default setting uses inlining to optimize your generated code.



When you inline a block function, the compiler replaces each call to a block function with the equivalent function code from the static run-time library. If your model use the same block four times, your generated code contains four copies of the function.

While this redundancy uses more memory, inline functions run more quickly than calls to the functions outside the generated code.

The final option in this category is **Use target-specific optimization for speed (allow LSB differences)**, which determines whether Embedded Target for TI C6000 DSP attempts to optimize the code generated from your model to make it run more quickly on your selected target. This option might not make any difference in some models.

Notice that selecting target-specific optimization allows your generated code to differ from your simulation results in the least significant bit (LSB) for the outputs of optimized blocks. You should review the results of the optimized and simulation-true code to see that they are sufficiently close for your needs. For many models, the LSB differences do not matter. Clearing this option results in generated code whose results match your model simulation results.

The preferred way to use **Use target-specific optimization for speed (allow LSB differences)** is to create your model, generate code from the model, and run the code on your target with profiling enabled. After you have your model and code running the way it should (generating the correct answers), try selecting this option and regenerating your code. Run your new code with profiling and compare the profile reports to see whether target-specific optimization improved the performance.

### Run-Time Options

Before you run your model as an executable on any C6000 target, you must configure the run-time options for the model on the board.

By selecting values for the options available, you configure the operation of your target.

### Build action

To specify to Real-Time Workshop what to do when you click **Build**, select one of the following options. The actions are cumulative—each listed action adds features to the previous action on the list and includes all the previous features:

- `Generate_code_only` — Directs Real-Time Workshop to generate C code only from the model. It does not use the TI software tools, such as the compiler and linker, and you do not need to have CCS installed. Also, MATLAB does not create the handle to CCS that results from the other options.

---

**Note** You cannot use `Generate_code_only` with DSP/BIOS enabled in your project. To use the option of generating code without creating a project, or using TI tools, you must clear **Incorporate DSP/BIOS** in the TIC6000 code generation options.

---

`Generate_code_only` creates a file named `model.bat`—an MS-DOS batch file that contains the TI C6000 compiler command line (c16x) you use to compile and link your generated code.

In this file you find information you need, such as the include paths, library locations, and default compiler options to compile the code, and that are not stored in any other file generated by the `Generate_code_only` build action. Learn more about the batch file by reading the comments included in the file.

The build process for a model also generates the files `modelName.c`, `modelName.cmd`, `modelName.bld`, and many others. It puts the files in a build directory named `modelName_c6000_rtw` in your MATLAB working directory. This file set contains many of the same files that Real-Time Workshop generates to populate a CCS project when you choose `Create_CCS_Project` for the build action.

- `Create_CCS_Project`—Directs Real-Time Workshop to start CCS and populate a new project with the files from the build process. This option offers a convenient way to build projects in CCS.
- `Archive_CCS_Library`—Directs Real-Time Workshop to archive the project for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your CCS projects for models that you use in model referencing.
- `Build`—Builds the executable COFF file, but does not download the file to the target.
- `Build_and_execute`—Directs Real-Time Workshop to download and run your generated code as an executable on your target.

Your selection for **Build action** determines what happens when you click **Build** or press **ctrl+B**. Your selection tells Real-Time Workshop when to stop the code generation and build process.



To run your model on the target, select `Build_and_execute`. This selection is the default build action; Real-Time Workshop automatically downloads and runs the model on your target board.

---

**Note** When you build and execute a model on your target, the Real-Time Workshop build process resets the target automatically. You do not need to reset the board before building models.

---

### Overrun action

To enable the overrun indicator, choose one of three ways for the target processor to respond to an overrun condition in your model:

- `None`—Ignore overruns encountered while running the model.
- `Notify_and_continue`—When the DSP encounters an overrun condition, it performs the operation you specify in **Overrun notification method** and continues running the executable. If you use a C6711 DSK LED block in your model, you cannot determine whether the C6711 DSK LED block enabled the external LED or if an overrun condition caused the LED to light.
- `Notify_and_halt`—Respond to overrun conditions by stopping program execution and executing the **Overrun notification method** option you select. If you use an LED block in your model, you cannot determine whether the LED block enabled the external LED or user-defined LEDs, or if an overrun condition caused the LEDs to light.

### Overrun notification method

In combination with the **Overrun action** option, you choose how the Embedded Target for TI C6000 DSP notifies you when your application goes into an overrun state. From the **Overrun notification method** list, select one of the following notification functions:

- `Print_message`—When your application overruns, and **Overrun action** is `Notify_and_continue` or `Notify_and_halt`, the software prints a message to the standard output or the message log (for DSP/BIOS enabled projects).
- `Turn_on_LEDs`—When your application overruns, and **Overrun action** is `Notify_and_continue` or `Notify_and_halt`, the software turns on the user LEDs on the C6711 DSK. Note that when you use an LED block in your

model, you might not be able to determine whether the LED block enabled the external LED or user-defined LEDs, or an overrun condition caused the LEDs to light. Other target boards might not provide LEDs and the LED option does not apply.

- `Print_message_and_turn_on_LEDs`—In an overrun situation where you have selected a notification action for `Overrun action`, the software prints a message and turns on the LEDs. The same rules apply as for the individual notification actions.

### **Overrun Indicator and Software-Based Timer**

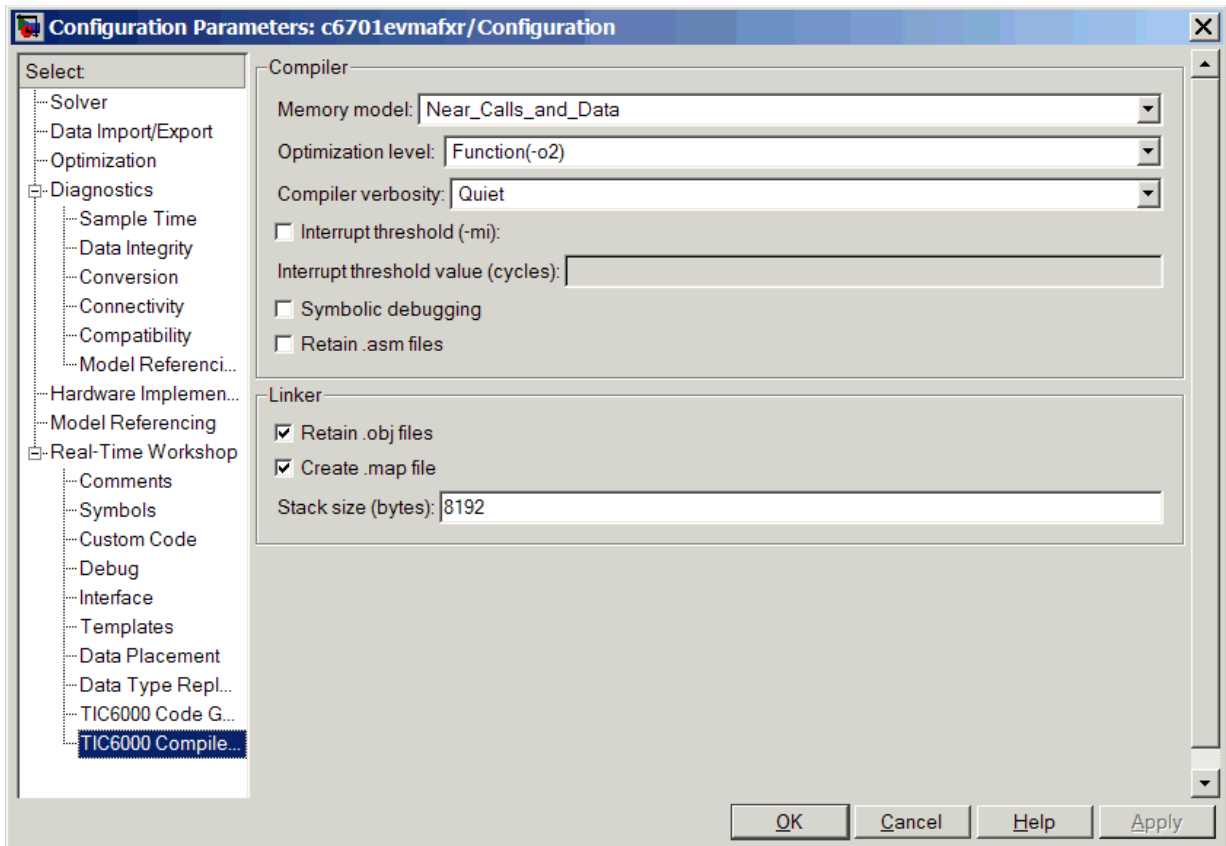
Embedded Target for TI C6000 DSP includes software that generates interrupts in models that do not have ADC or DAC blocks, or that use multiple clock rates. In the following cases, the overrun indicator does not work:

- In multirate systems where the rate in the model is not the same as the base clock rate for your model. In such cases, the timer in the Embedded Target for TI C6000 DSP provides the interrupts for setting the model rate.
- In models that do not include ADC or DAC blocks. In such cases, the timer provides the software interrupts that drive model processing.

### **TI C6000 Compiler/Linker Options**

Options in this category determine how the TI C6000 compiler generates compiled code for the assembler and linker to use.

If you change the settings in this dialog box, your changes become part of the build configuration options for your project in CCS. You can change these settings in CCS later. In the dialog box, as presented in the figure, the controls under `TI C6000 compiler` let you configure compiler operations.



## Compiler Options

**Memory model.** You must specify how to map the memory for your processor in **Memory model**. The list provides options that affect how the compiler handles near and far calls and data, and aggregate data.

Your selections for the memory model options affect how the Embedded Target for TI C6000 DSP handles *near* and *far* data. By default, the Embedded Target for TI C6000 DSP, and the TI compiler, generate memory models that use both near data calls and far data calls.

Accessing near data requires only one operation; far data access requires more operations. As a consequence, programs and code that use far data sometimes run more slowly. Refer to your CCS documentation for details about near data, far data, aggregate data, and near and far accesses.

- **Near\_Data**—Tells the compiler to allocate all data as near accesses.
- **Far\_Aggregate**—Tells the compiler to allocate aggregate data, like arrays and structures, as far accesses. Non-aggregate data (all data that is not aggregate) default to near accesses. Structures, unions, C++ classes, and arrays, which are aggregate data, are not accessed using the more efficient data-page (DP) pointer. This is the default behavior and setting.
- **Far\_Data**—Tells the compiler to allocate all data as far accesses.

When you select the **Near\_Data** option, the Embedded Target for TI C6000 DSP specifies that only near calls are used to access static and global or extern data. **Near\_Data** represents the most efficient memory use. In CCS, the equivalent setting is to choose `Near(--mem_model:data=near)` for the **Memory model** option in the build configuration.

If you select **Near\_Data**, but your data or program requires far accesses, the TI compiler returns an error message like the following in the CCS IDE

```
error: can't allocate '.far'
```

or

```
error: can't allocate '.text'
```

indicating that your data does not fit in internal memory or your code or program does not fit in internal memory. The DP could not reach the specified data using relative offset addressing.

To eliminate these errors, select **Far\_Data** from the **Memory model** list.

Use the **Far\_Data** selection when either or both of the following conditions are true:

- Your static and external data do not fit within a 15-bit scaled offset from the beginning of the `.bss` section of memory.
- You have calls in which the called function is more than  $\pm 1$  Mword away from the call site.

If your program meets the conditions noted and you select `Near_Data` for **Memory model**, the TI linker issues the error message shown earlier.

When you declare a function or data as far, the compiler loads its address into a register and then does an indirect load of that register (the `-mln` option in **Memory models** in the project build configuration in CCS). For more information on the `-mln` option, refer to your CCS documentation.

You can avoid the allocation error by selecting `Far_Data` for the **Memory model** option. This option prevents the compiler from using near calls, offering you the ability to use all the available memory on your target. However, your program might run more slowly than if you use the internal map option `Near_Data`, and your data and program fit into memory without needing far calls for access.

The `Far_Aggregate` option allocates aggregate data as far calls and non-aggregate data as near calls. This setting is the default for your projects in Embedded Target for TI C6000 DSP. In CCS, the equivalent setting is `Far_aggregate` in the **Memory models** option and represents the CCS default setting as well.

To use near accesses for aggregate data, you must select the `Near_Data` setting for **Memory model**.

## Optimization level

To let you determine the degree of optimization provided by the TI optimizing compiler, you select the optimization level to apply to files in your project. For details about the compiler options, refer to your CCS documentation. When you create new projects, the Embedded Target for TI C6000 DSP sets the optimization to `Function(-o2)`.

## Compiler verbosity

You can choose how much information the compiler returns while it runs. Select from

- **Verbose**—Returns all compiler messages
- **Quiet**—Suppresses compiler progress messages
- **Super Quiet**—Suppresses all compiler messages

### Interrupt threshold (-mi)

**Interrupt threshold (-mi)** enables an interrupt threshold that defines the maximum number of cycles over which the compiler can disable interrupts. You can use this option in various ways by setting combinations for **Interrupt threshold** and **Interrupt threshold value**. The following table describes the effects of the possible combinations

Interrupt Threshold Option	Interrupt Threshold Value (Cycles)	Effect of the Option Settings
Cleared	Not Applicable	(Default) Interrupts are explicitly disabled around software pipelined loops. The remainder of the code is interruptible.
Selected	Empty	Compiler assumes the code is never interrupted during execution.
Selected	1	Compiler generates fully interruptible code.
Selected	Integer = 2 or greater	<p>The compiler analyzes the code for each loop and determines the maximum number of cycles required to execute the loop.</p> <ul style="list-style-type: none"> <li>• If the maximum number of cycles is less than the threshold value, the compiler generates the optimal or fastest version of the loop.</li> <li>• If the maximum is larger than the threshold, the compiler generates an interruptible loop that still generates correct output. In most cases, an interruptible loop is less efficient than an optimized loop.</li> </ul>

### Using the Interrupt Threshold Option and Setting a Threshold Value

By default, **Interrupt threshold** is cleared and interrupts are explicitly disabled around software pipelined loops. You can change this setting and behavior to suit your application. Use these tips for deciding when and how to use the threshold value options:

## Profiling and the Threshold Option

If your program uses profiling, the threshold option can help you prevent profiling errors. The profiling feature in Embedded Target for TI C6000 DSP uses DSP/BIOS STS objects in CCS that in turn use the DSP/BIOS clock manager to keep track of elapsed time. The clock manager interrupt, which defaults to a 1 KHz rate, or 1 cycle/ms, is used to increment a counter. Whenever the code disables interrupts, your program risks delaying the clock manager interrupt operation and the STS timing operations. If interrupts are disabled for 1 ms or longer, your profiling measurements will contain timing errors.

## Timing Errors

This profiling error condition can arise if you use the compiler optimization level -o2 or higher, because software pipelining causes interrupts to be disabled during pipelined loops. Also, some C62x/C64x DSP library assembly routines disable interrupts during execution.

If your profiling measurements are affected by this interaction, measured and reported execution times might be shorter than the real execution times. To resolve this issue, consider one of the following recommendations:

- Generate fully interruptible code by selecting the **Interrupt threshold** option and entering 1 for the **Interrupt threshold value**. This selection disables software pipelining and lowers the execution speed of your model. If your model still runs without changing the desired sampling rate, this method might be a good way to maintain your profiling.
- Avoid disabling interrupts for more than 1 ms. This precaution ensures that the clock manager can always interrupt the code at 1 KHz (1 ms intervals). You need to translate 1 ms to cycles to enter a threshold value. The value in cycles varies depending on your CPU clock speed (value in cycles = 1/1000 x CPU speed (in cycles/second)). Use the calculated cycles value to set the **Interrupt threshold value**.

If your program uses -o2 level compiler optimization (which is the default optimization level), setting the threshold interrupt value can ensure that critical interrupts happen as you expect. Using -o2 optimization adds the benefit of software pipelined loops around which interrupts are disabled.

Disabling interrupts could cause problems in the following cases:

- When your application has time-critical interrupts that must be serviced,
- In situations where preemption occurs, such as in multitasking, multirate models

Consider the following techniques to resolve this problem.

- Generate fully interruptible code by selecting the **Interrupt threshold** option and entering 1 for the **Interrupt threshold value**. Setting 1 as the interrupt value prevents the compiler from disabling interrupts for more than one cycle—essentially allowing any interrupt to occur.
- Select **Interrupt threshold** and set **Interrupt threshold value** to something close to the base sampling time of your model. To determine the recommended threshold value, use the base sampling time of your model and the CPU clock speed as shown here because this value usually produces good results with -o2 optimization:

Interrupt threshold value < (0.8 x base sampling time x CPU clock speed).

For more information on using the **Interrupt threshold (-mi)** option, refer to the documentation for the -mi compiler option provided in the online help for CCS.

### Interrupt threshold value (cycles)

Setting this option to a positive integer tells the compiler that uninterruptible code sections cannot exceed the specified number of clock cycles in length. When you enable the **Interrupt threshold** option, this option becomes active. Although any integer is acceptable as a value, two have special significance:

- Value = 0. A 0 value (or empty) for **Interrupt threshold value** disables the threshold and the compiler assumes the code is never interrupted.
- Value = 1. A 1 for the threshold value means the code is fully interruptible. The clock/interrupt service can interrupt any code segment at any time.

### Symbolic debugging

Selecting this option generates symbolic debugging directives that the C source-level debugger uses, and enables assembly source debugging. By default this option is selected—symbolic debugging is provided.



### Retain .asm files

Select this option to direct Real-Time Workshop and the Embedded Target for TI C6000 DSP to save your assembly language (.asm) files after creation. The Embedded Target for TI C6000 DSP does not retain .asm files by default. If you choose to keep the .asm files, Real-Time Workshop saves the files to your current directory. When you create new projects, the Embedded Target for TI C6000 DSP does not save your .asm files unless you select this option.

## Linker Options

You can configure the TI C6000 linker to perform certain operations and use specified files. The linker, not the compiler, defines the memory map used and allocates code and data into memory on the target. Refer to Texas Instruments *TMS320C000 Optimizing C Compiler User's Guide* and to the online help in CCS for more details about using memory maps on TI processors.

### Retain .obj files

The linker uses object (.obj extension) files to generate a single executable common object file format (COFF) file that you run on the C6711 DSK. Select this option to direct Real-Time Workshop and the Embedded Target for TI C6000 DSP to save your object (.obj) files after creation.

Real-Time Workshop saves the files to your current directory. Saving your .obj files can speed up the compile process by not having to compile files that you have not changed since you most recently compiled your project. Retaining the .obj files is the default setting for new projects.

### Create .map file

You can direct the linker to produce a map of the input and output sections, including null areas, and place the listing in a file in your current directory with the name *mode1name.map*. When you clear this check box, the linker does not produce the listing. New projects do not create the .map file.

### Stack Size (Bytes)

You can enter any stack size in bytes, using decimal format. For more information about the stack and block output, refer to **Enable local block outputs** in the **Code Generation** options in the Optimization category in the **Select** tree. Also refer to the online Help system for more information about

Real-Time Workshop options for configuring and building models and generating code.

### **Embedded Target for TI C6000 DSP Default Project Configuration – custom\_MW**

Although CCS offers two standard project configurations, Release and Debug, models you build with the Embedded Target for TI C6000 DSP use a custom configuration that provides a third combination of build and optimization settings—`custom_MW`.

Project configurations define sets of project build options. When you specify the build options at the project level, the options apply to all files in your project. For more information about the build options, refer to your TI CCS documentation.

The default settings for `custom_MW` are the same as the Release project configuration in CCS, except for the compiler options discussed in the next section. `custom_MW` uses different compiler optimization levels to preserve important features of the generated code.

#### **Default Compiler Build Options in custom\_MW**

When you create a new project or build a model to your TI C6000 hardware, your project and model inherit the build configuration settings from the configuration `custom_MW`. The settings in `custom_MW` differ from the settings in the default Release configuration in CCS in the compiler settings.

For the compiler options, `custom_MW` uses the `Function(-o2)` compiler setting. The CCS default Release configuration uses `File(-o3)`, a slightly more aggressive optimization model.

For memory configuration, where Release uses the default memory model that specifies near functions and data, `custom_MW` specifies near functions and data—the `-m11` memory model—because some custom hardware might not support far data or aggregate data. Your CCS documentation provides complete details on the compiler build options.

You can change the individual settings or the build configuration within CCS. Build configuration options that do not appear on these panes default to match the settings for the Release build configuration in CCS.

## Model Reference and Embedded Target for TI C6000 DSP

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 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 dynamic linked library, .dll) 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 requires executables to generate code from models. If you have not simulated your model at least once, Real-Time Workshop creates a .dll 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 calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

## Using Model Reference with Embedded Target for TI C6000 DSP

With few limitations or restrictions, Embedded Target for TI C6000 DSP 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 > TIC6000 Code Generation**) for all models referred to in the simulation to `Archive_CCS_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 **TIC6000 Code Generation**.

- 4 In the right pane, under **Runtime**, set **Build action** to `Archive_CCS_Library`.

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

As a result of selecting the `Archive_CCS_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 Embedded Target for TI C6000 DSP does not allow you to use certain blocks or S-functions in reference models:

- No blocks from the C62x DSP Library (in `c6000lib`) (because these are noninlined S-functions)
- No blocks from the C64x DSP Library (in `c6000lib`) (because these are noninlined S-functions)

- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Embedded Target for TI C6000 DSP 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 and Embedded Target for TI C6000 DSP does not support External mode. If you select this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities directory, as described in Supporting Shared Utility Directories 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 Embedded Target for TI C6000 DSP versions 2.4 and later automatically include the model reference capability. You do not need to set the flag.

## Targeting Supported Boards

Texas Instruments markets a complete set of tools for you to use with the a range of development boards, such as the C6711 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 and the C6711 DSK blocks.

Executing code generated from Real-Time Workshop 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 a communication link with Simulink, 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 C6711 DSK), you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, the Embedded Target for TI C6000 DSP provides a target makefile specific to the evaluation module. This target makefile invokes the optimizing compiler, provided as part of TI Code Composer Studio.

Used in combination with Real-Time Workshop, 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 C6711 DSK, or

the Custom C6000 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 C6711 DSP Starter Kit

After you install the C6711 DSK development board and supporting TI products on your PC, start MATLAB. At the MATLAB command prompt, enter `c6711dsklib`. This opens a Simulink block library, `c6711dsklib`, that includes a set of blocks for C6711 DSK I/O devices, as described in the following table.

Block	Description
C6711 DSK ADC	Configure the analog to digital converter
C6711 DSK DAC	Configure the digital to analog converter
C6711 DSK LED	Control the user status LEDs on the C6711 DSK
C6711 DSK Reset	Reset the processor on the C6711 DSK

These blocks are associated with your C6711 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 and template makefile. For the C6711 DSK, in the Real-Time Workshop pane, specify

- **Real-Time Workshop system target file**—`ti_c6000.tlc`
- **Template makefile**—`ti_c6000.tmf`



With this configuration, you can generate a real-time executable and download it to the TI C6701 evaluation board. You generate the executable by clicking **Build** on the Real-Time Workshop pane. The Real-Time Workshop 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 template makefile and block parameter dialog box entries are combined to form the target makefile for your TI evaluation module. This makefile invokes the TI compiler to build an executable file. If you select the `Build_and_execute` option, Real-Time Workshop automatically downloads the executable to the TI evaluation board via the peripheral component interface (PCI) bus. After downloading the executable file to the C6711 DSK, the build process runs the file on the processor.

### Starting and Stopping DSP Applications on the C6711 DSK

When you generate code, build the project, and download the code for your Simulink model to your C6711 DSK, you are running actual machine code corresponding to the block diagram you built in Simulink. 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 C6711 DSK, you use Real-Time Workshop 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.
- 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 C6711 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 C6711 DSK, the Real-Time Workshop build process resets the evaluation module automatically. You do not need to reset the board before building models. Use the Reset C6711 DSK block to stop processes that are running on the evaluation module, or to return the board to a known state for any reason.

---

### Configuring Your C6711 DSK

When you install the C6711 DSK, 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 *TMS320C6201/6701 Evaluation 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
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 C6711 DSK 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 *TMS320C6201/6711 DSP Starter Kit User's Guide*.

To run the C6711 DSK verification test, complete the following steps after you install your board:

- 1 Start CCS.
- 2 Select **Start > Programs > Code Composer Studio > EVM 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 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 *TMS320C6201/6701 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 C6701 EVM. After you change your board configuration, rerun the verification utility to check your new settings.

## Testing Your C6711 DSK

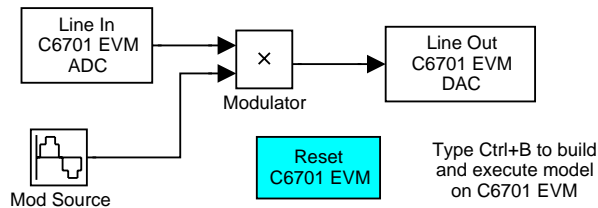
The Embedded Target for TI C6000 DSP includes a Simulink demonstration model called `c6701evmtest`. You can use this model to verify that you installed your C6701 EVM hardware and your Embedded Target for TI C6000 DSP software correctly and the board settings are suitable for targeting. The demonstration model presets the Real-Time Workshop settings to build and run the model on your board.

To run the model you need a signal generator, an oscilloscope, and audio cables to connect the signal generator and scope to your C6701 EVM. Refer to the *Texas Instruments TMS320C6201/6701 Evaluation Module User's Guide* for more information on connecting sources and scopes to your C6701 EVM. In

addition, connect your signal generator to the oscilloscope input so you can display the source and output signals together.

### To Confirm the Operation of Your C6711 DSK

As an initial test to determine that your Embedded Target for TI C6000 DSP software and C6711 DSK are installed and operating correctly, open and build the Simulink model `c6711dsktest`. See the model in the following figure.

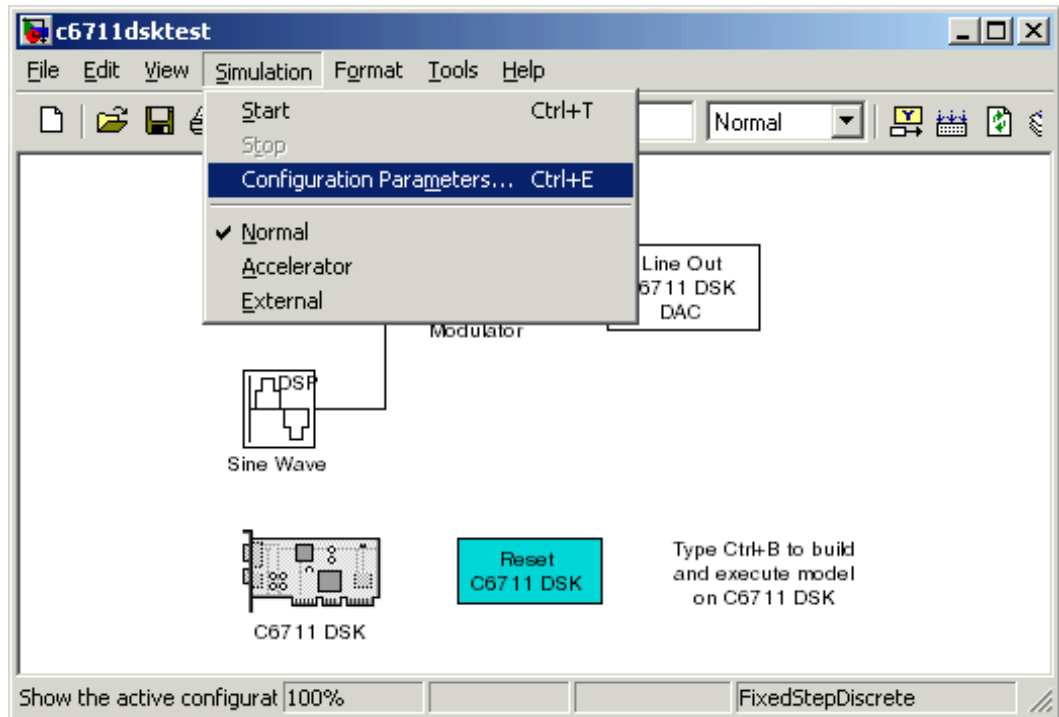


- 1 Enter `c6711dsktest` at the MATLAB command prompt.

The model opens in Simulink.

- 2 Select **Configuration Parameters** from the **Simulation** menu.

The next figure shows the model `c6711dsktest` with the **Configuration Parameters** option selected.



- 3 On the Configuration Parameters dialog box, click Real-Time Workshop in the **Select** tree to view the Real-Time Workshop pane.
- 4 Click **Build** to run the model. Building the model provides a comprehensive test of the build, download, and run processes in the Embedded Target for TI C6000 DSP.

A lengthy series of messages appears in the MATLAB Command Window, starting with

```
### Starting Real-Time Workshop build procedure for model:
c6711dsktest.mdl
```

```
### Invoking Target Language Compiler on c6711dsktest.rtw
```

If `c6711dsktest.mdl` builds, compiles, and downloads to your C6711 DSK successfully, the following message strings appear at the end of the build process messages.

```
C6x EVM Command Line COFF Loader Utility, Version 1.20a
Copyright (c) 1998 by DNA Enterprises, Inc.
Found board type:DSK6x Revision:0
Using DSP memory map 1.
### Downloaded:c6711dsktest.out
### Successful completion of Real-Time Workshop build procedure
for model:c6711dsktest
```

When you receive this message, your model is running on the C6711 DSK. You should be able to see the input and output on your oscilloscope. When you change the input, the output should change as well.

Try increasing the frequency you send to your C6711 DSK and watch to see that the output amplitude modulation changes to match.

### **Error Messages While Building C6711dsktest**

If you receive an error message from the build and compile process, your board or the software may not be configured correctly. Reinstall the board and review the configurations listed in “Configuring Your C6711 DSK” on page 2-114. You need to resolve errors that appear in this build before you start to develop and build your own models.

Note that after you build and download the model to the board, the build process runs the downloaded code on your C6711 DSK immediately.

### **Verifying That C6711dsktest Is Running**

To see that the model is running, turn on your signal generator and set the output to produce a sine wave at 8000 Hz. Connect your oscilloscope to display both the input signal from the signal generator and the output from your C6711 DSK.

On the oscilloscope display, you should see the sine wave input from the signal generator, and the amplitude-modulated sine wave output from your C6711 DSK. If you change the frequency of the sine wave input, you should see the change in the input and output traces on the oscilloscope.

## Starting and Stopping C6711dsktest on the C6711 DSK

When you build and download the model `c6711dsktest.mdl` to your C6711 DSK, you are not running a simulation of the model. You are running the actual machine code, in real time, corresponding to the block diagram in `c6701evmtest.mdl`. To run `c6701evmtest.mdl` on the evaluation module, open the Simulink model and click **Build** on the **Real-Time Workshop** pane.

Clicking **Build** rebuilds the machine executable and downloads the new executable to your board. Building and downloading the new executable starts the process running on your C6711 DSK. The Embedded Target for TI C6000 DSP offers a function, `run`, that restarts your loaded program on your target.

After your application is running on your target, use one of the following methods to stop the process.

- Use the **Debug > Halt** function in CCS.
- Use `halt` entered at the MATLAB command prompt.
- Click the C6711 DSK Reset block in your model (if you added one) or in the C6711 DSK board support library.

## 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
- 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 Embedded Target for TI C6000 DSP 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.

<b>Block Name/Category</b>	<b>Library</b>	<b>Description</b>
Scope	Simulink, Signal Processing Blockset	Provides oscilloscope view of your output. Do not use the <b>Save data to workspace</b> option on the <b>Data history</b> 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.



<b>Block Name/Category</b>	<b>Library</b>	<b>Description</b>
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.
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.
To Wave file	Signal Processing Blockset	Send data to a .wav file.
From Wave file	Signal Processing Blockset	Get data from a .wav file.

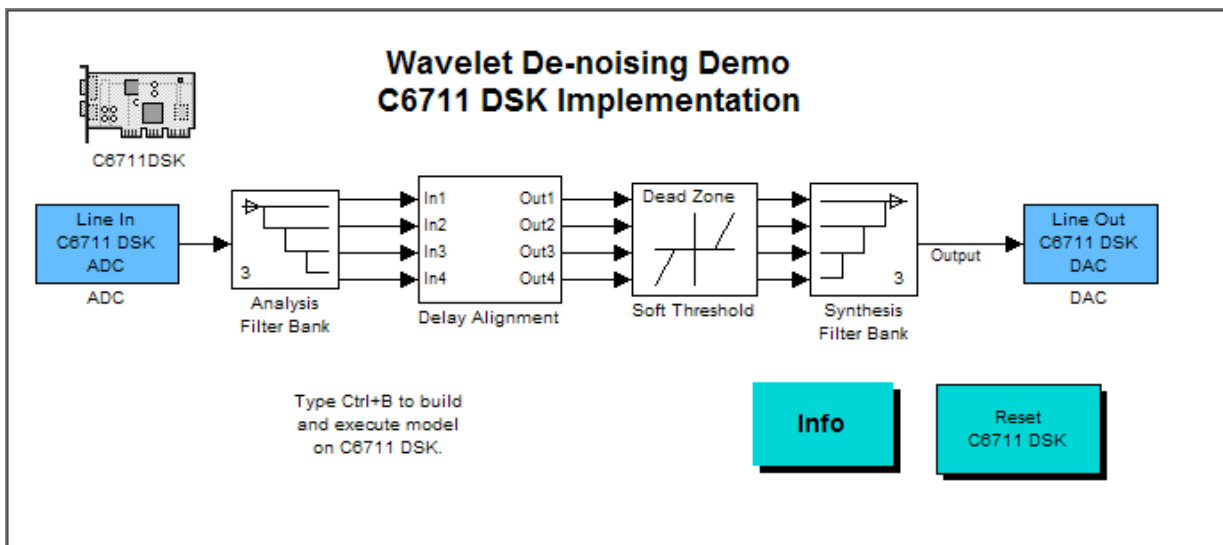
In general, using blocks to add instrumentation to your application is a valuable tool. In most cases, blocks you add to your model to display results

or create plots, such as Histogram blocks, add to your generated code without affecting your running application.

When you need to send data to or receive data from your target, use the To Rtdx and From Rtdx blocks to accomplish the data transfer.

## Targeting Tutorial II — A More Complex Application

For this tutorial, we demonstrate an application that uses multiple stages—using wavelets to remove noise from a noisy signal. The model name is `c6711dskwdnoisf`. As with any model file, you can run this denoising demonstration by typing `c6711dskwdnoisf` at the MATLAB prompt. The model also appears in the MATLAB demos collection in the Help browser—under Simulink demos, in the Embedded Target for TI C6000 DSP category. Here is a picture of the model as it appears in the demonstration library.



Unlike the audio reverberation demo, this model is difficult to build from blocks in Simulink. 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 C6711 DSK as an executable COFF file.

### Working and Build Directories

It is convenient to work with a local copy of the `c6711dskwdnoisf` model, stored in its own directory, which you named (something like `c6711dnoisfex`). This discussion assumes that the `c6701dnoisfex` directory resides on drive `d:`. Use a different drive letter if necessary for your machine. Set up your working directory as follows:

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

```
!mkdir d:\c6701dnoisfex (on PC)
```

- 2 Make `c6701dnoisfex` your working directory in MATLAB.

```
cd d:/c6701dnoisfex
```

- 3 Open the `c6711dskwdnoisf` model.

```
c6711dskwdnoisf
```

The model appears in the Simulink window.

- 4 From the **File** menu, choose **Save As**. Save a copy of the `c6711dskwdnoisf` model as `d:/c6701dnoisfex/dnoisfrtw.mdl`.

During code generation, Real-Time Workshop creates a build directory within your working directory. The build directory name is `model_target_rtw`, derived from the name of your source model and your chosen target. In the build directory, Real-Time Workshop stores generated source code and other files created during the build process. You examine the contents of the build directory 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 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 Embedded Target for TI C6000 DSP does not honor a stop time if you set one here.

**Start Time:** 0.0

**Stop Time:** inf

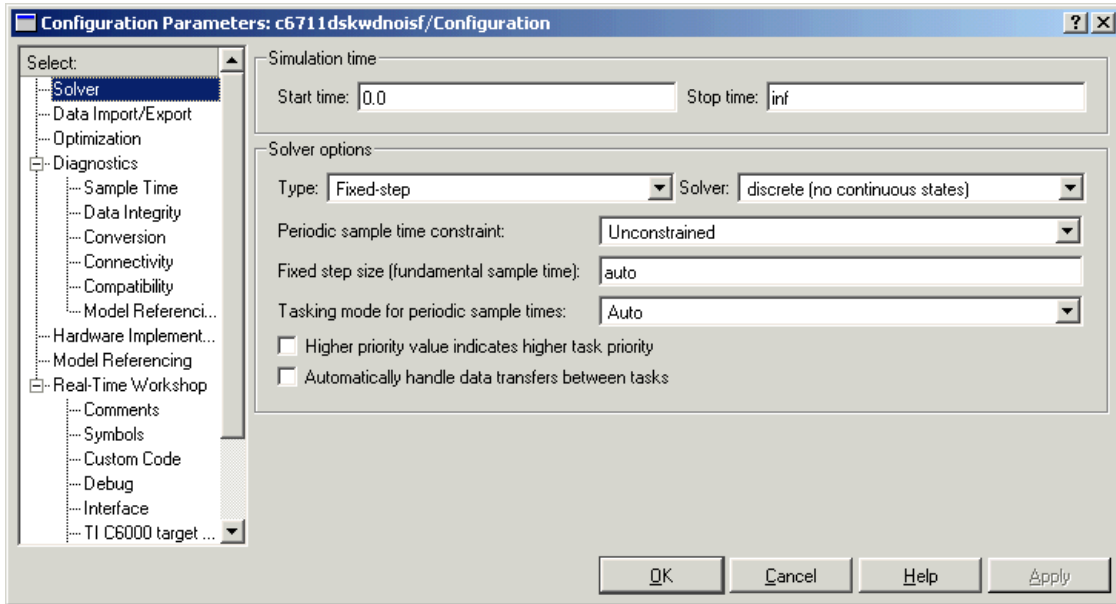
**Solver options:** set **Type** to Fixed-step. Select the discrete solver algorithm.

**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, you choose the

- System target file
- Template makefile
- make command

In these tutorials, you do not need to specify these parameters individually. Instead, you use the ready-to-run `ti_c6000.tlc` target configuration.

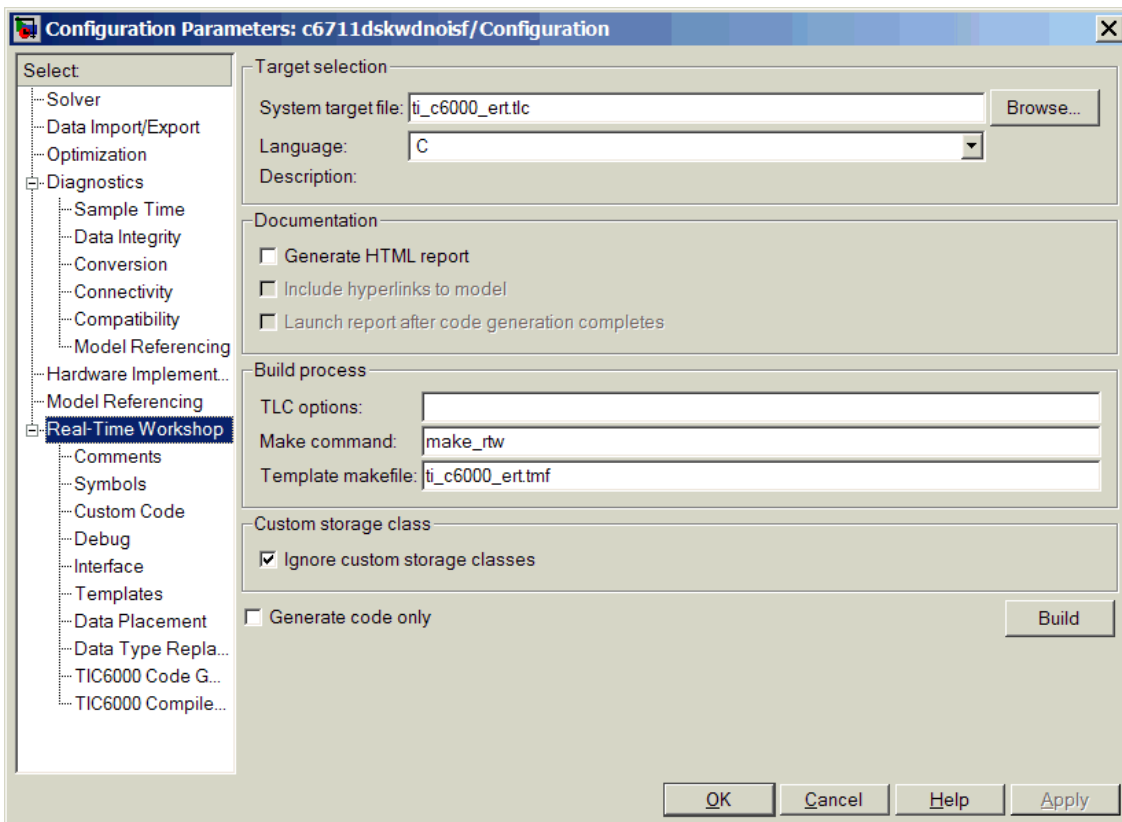
---

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

---

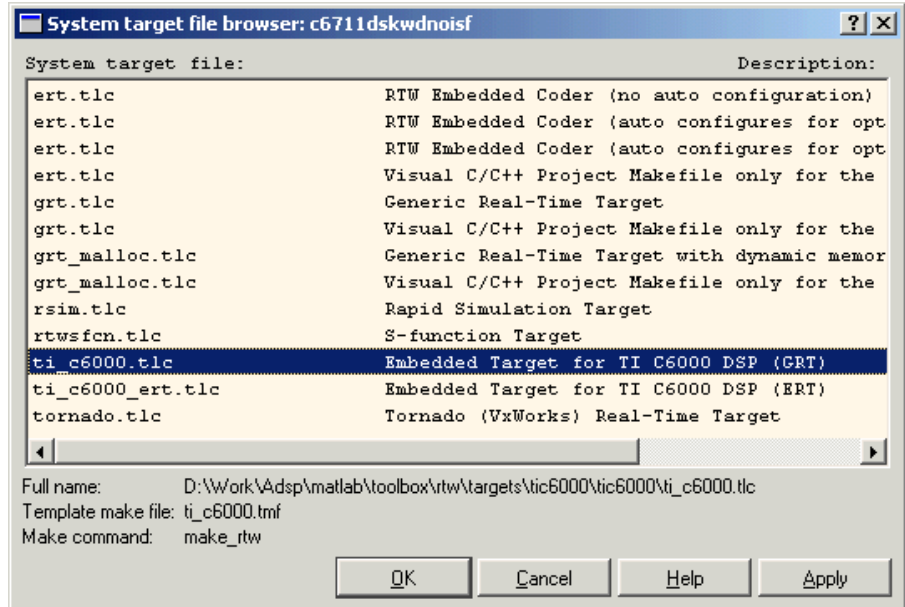
To target your C6711 DSK:

- 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 automatically chooses the appropriate system target file, template makefile, and make command.

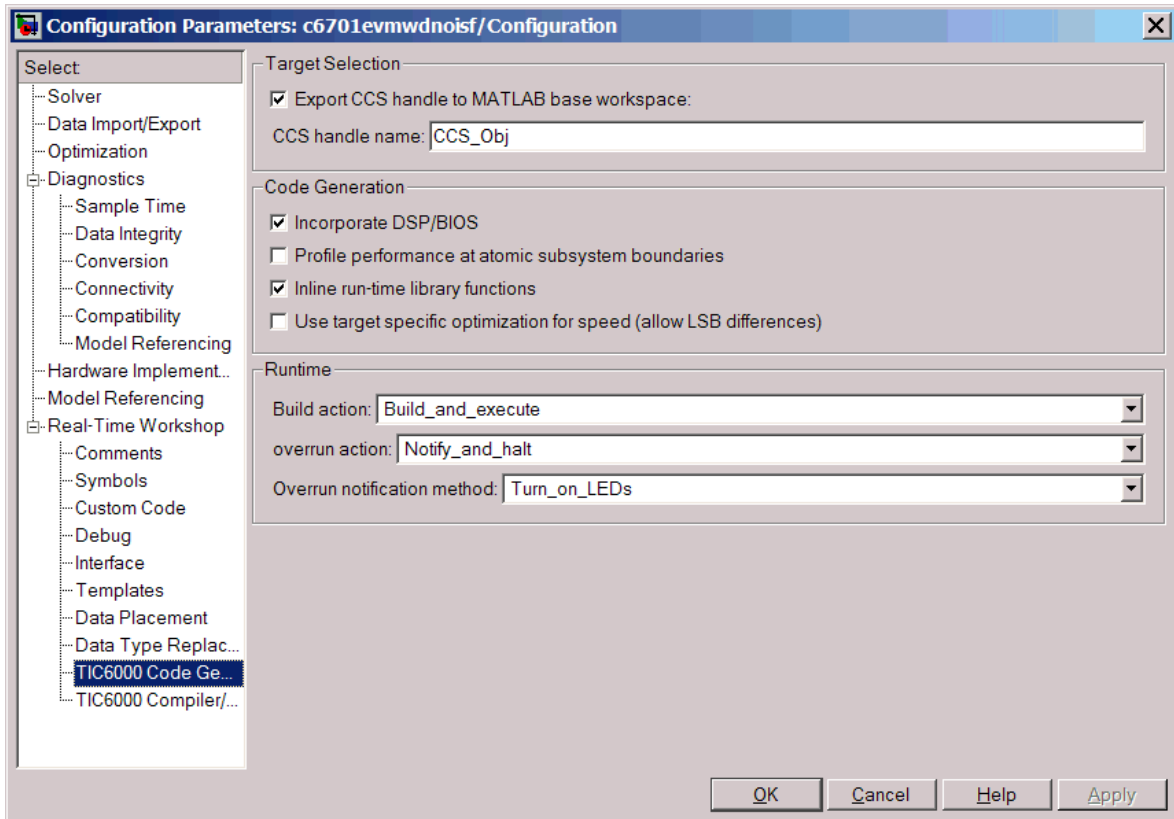


- 4 From the list of available configurations, select `ti_c6000.tlc`, and click **OK**.

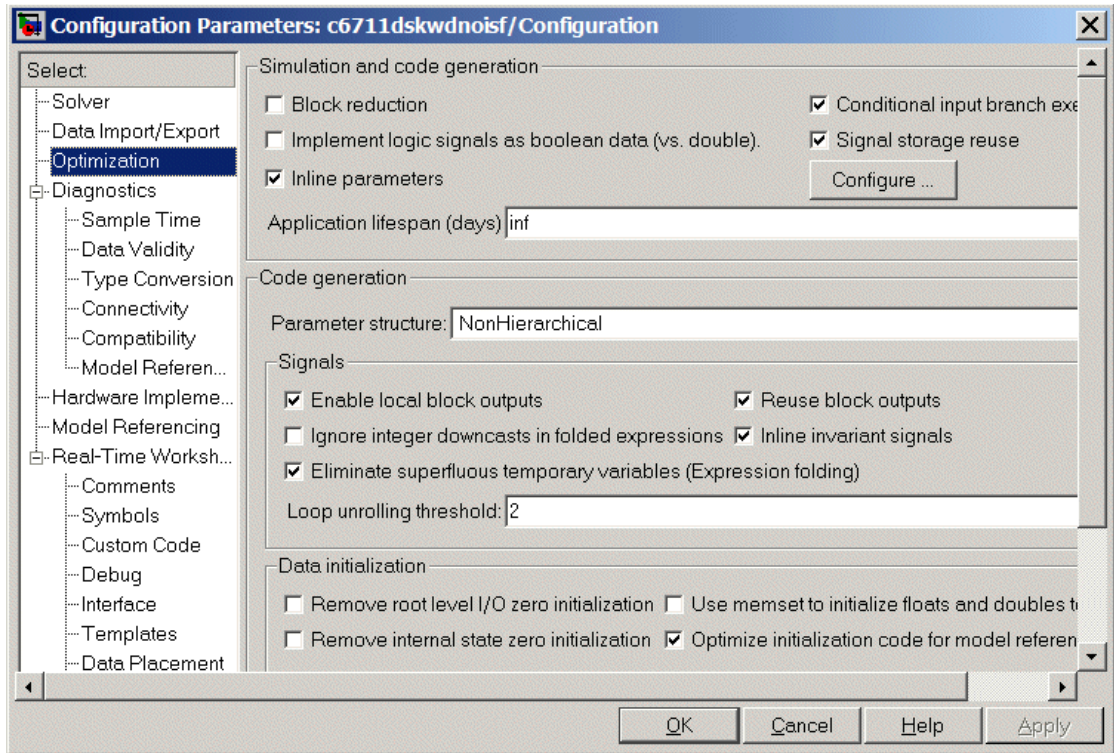
The Real-Time Workshop pane now displays the correct **Real-Time Workshop system target file** (`ti_c6000.tlc`), **Template makefile** (`ti_c6000.tmf`), and **Make command** (`make_rtw`).

- 5 To decide whether to export a CCS handle to your MATLAB work space when you generate code, or run your model, select **TI C6000 Code Generation** from the **Select** tree.

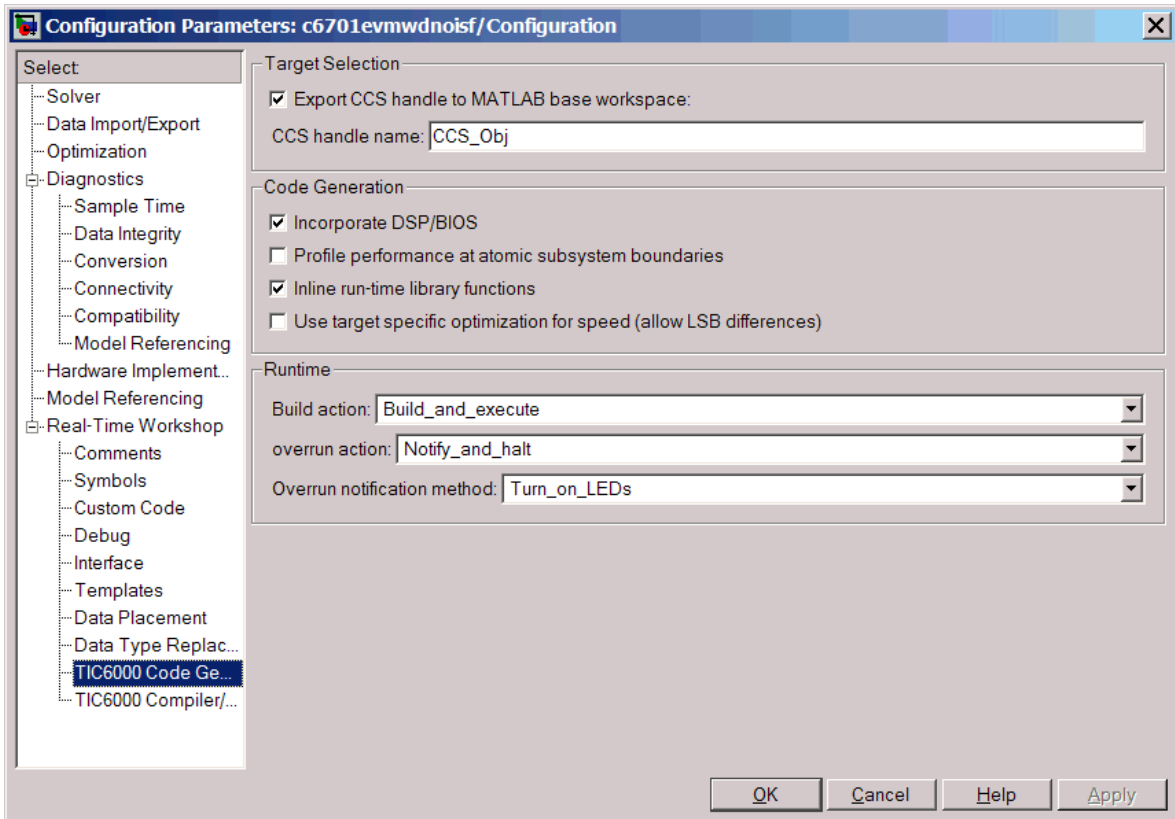




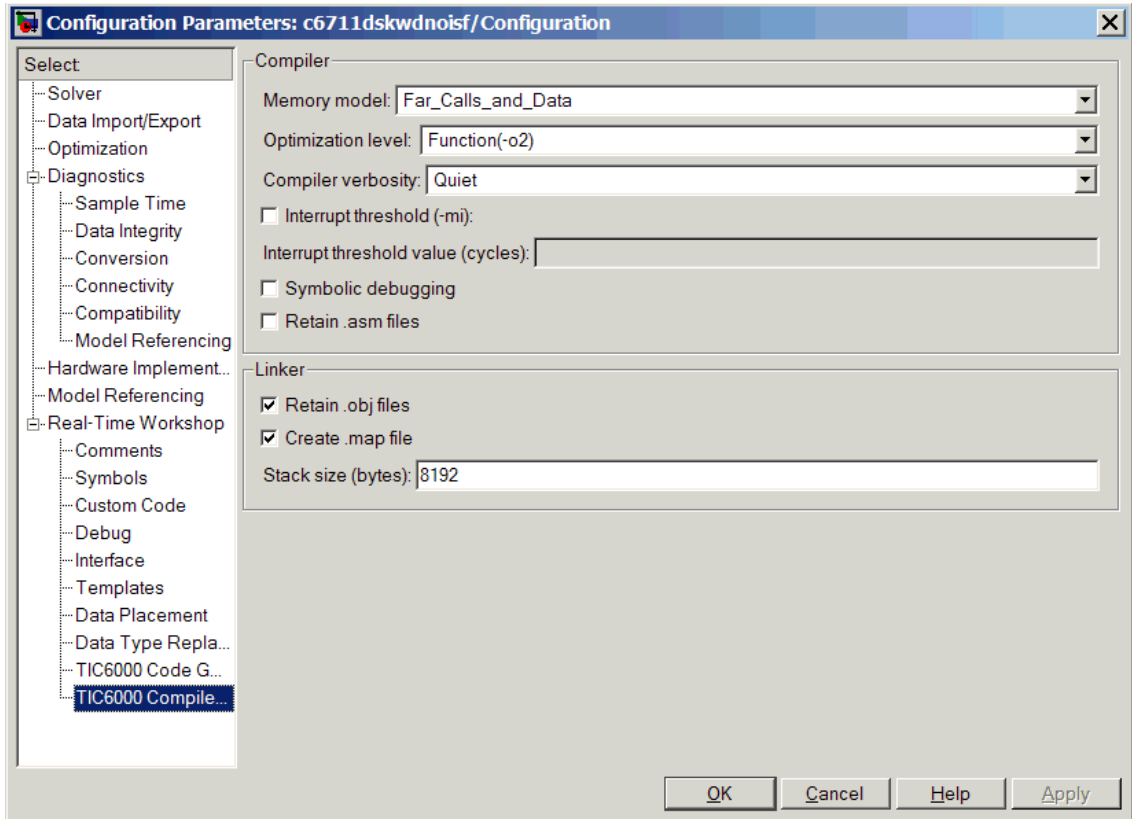
- 6 To export the handle (a variable) that CCS creates when you generate code from your model, select **Export CCS handle to MATLAB workspace**, and enter a name for the handle in **CCS handle name**.
- 7 Select the **Inline run-time library functions** and the **Incorporate DSP/BIOS** options, as shown in the previous figure.
- 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.



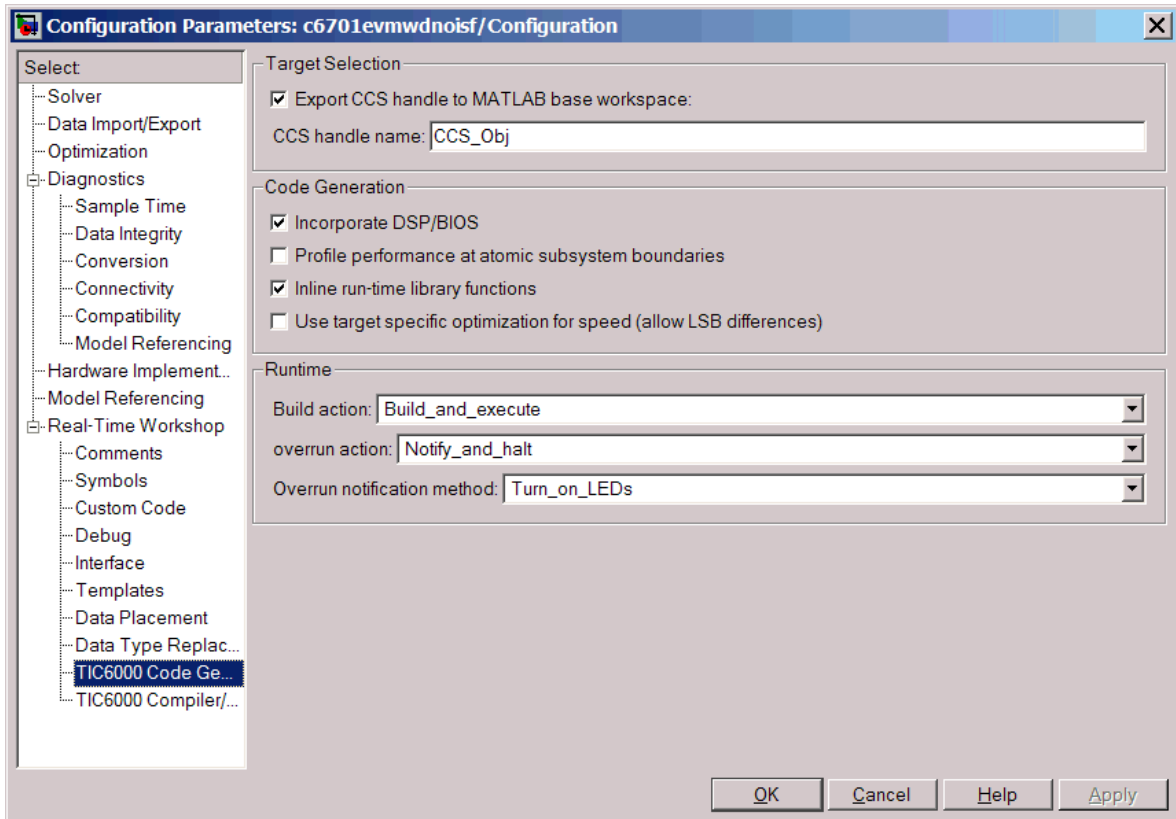
- 9 Select **Debug** from the **Select** tree to access the Real-Time Workshop Process and TLC process debugging options. Clear the check boxes on this pane. Check **Verbose build** to see all the messages that Real-Time Workshop issues while it generates your code or project. Selecting **Verbose build** is optional, but can be useful when you are new to the code generation process



- 10 Select TI C6000 Compiler/Linker from the **Select** tree. The options displayed on the new pane are specific to the C6000 target and TI compiler. Check to make sure that the options are set as shown in the following figure.



- 11 Select TI C6000 Code Generation on the **Select** tree to access the C6000 run-time options. Set the run-time options as shown in the following figure.



**12** 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** Clear **Generate code only** and 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 Command Window. The initial messages are

```
### Starting Real-Time Workshop build procedure for model:
dnoisfrtw
### Generating code into build directory: .\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 directory now contains an executable, `dnoisfrtw.exe`. In addition, Real-Time Workshop created a build directory, `dnoisfrtw_c6000_rtw`.

To review the contents of the working directory after the build, type the `dir` command from the MATLAB Command Window.

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

**5** To run the executable from the MATLAB Command Window, 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 directory, type

```
dir dnoisfrtw_c6701_rtw
```

## Contents of the Build Directory

The build process creates a build directory and names it `model_target_rtw`, concatenating the name of your source model and your chosen target. In this example, your build directory is named `dnoisfrtw_c6701_rtw`.

`dnoisfrtw_c6701_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 directory also contains other files used in the build process, such as the object (`.obj`) files and the generated makefile (`dnoisfrtw.mk`).

# Targeting Your C6711 DSK and Other Hardware

The Embedded Target for TI C6000 DSP for Texas Instruments DSP lets you use Real-Time Workshop to generate, target, and execute Simulink models on the Texas Instruments (TI) C6711 DSP Starter Kit (C6711 DSK). In combination with the C6711 DSK, your Embedded Target for TI C6000 DSP software is the ideal resource for rapidly prototyping and developing embedded systems applications for the TI C6711 Digital Signal Processor. The Embedded Target for TI C6000 DSP software focuses on developing real-time digital signal processing (DSP) applications for the C6711 DSK.

This chapter describes how to use the Embedded Target for TI C6000 DSP to create and execute applications on the C6711 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 automatic code generation. To read more about Real-Time Workshop, refer to your Real-Time Workshop documentation.

In this chapter, you will find sections that detail how to use your Embedded Target for TI C6000 DSP to build and download DSP applications in Simulink to your C6711 DSK and to Texas Instruments Code Composer Studio (CCS):

- Configuring your Embedded Target for TI C6000 DSP software, in “Setting Real-Time Workshop Options for C6000 Hardware” on page 2-82
- Configuring your Texas Instruments TMS320C6711 DSP Starter Kit, in “Configuring Your C6711 DSK” on page 2-136
- Testing your hardware and software installation to be sure everything works, in “Confirming Your C6711 DSK Installation” on page 2-136 and “Testing Your C6711 DSK” on page 2-137

## Configuring Your C6711 DSK

After you install and configure your C6711 DSK according to the instructions in the online help for CCS, you do not need to configure further your C6711 DSK.

## Confirming Your C6711 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 “TMS320CDSK Help” under TMS320C6000 Code Composer Studio Help in the CCS online help system.

To run the C6711 DSK confidence test, complete the following steps after you install and configure your board.

**1** Open a DOS command window.

**2** Access the directory `..\ti\c6000\dk6x11\confst`

CCS creates this directory when you install your CCS software. It contains the files to run the C6711 confidence test.

**3** Start the confidence test by typing `dk6xtst` at the DOS prompt.

By default, the test utility creates a log file named `dk6xtst.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 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 online help for CCS.

**4** Review the test results to verify that everything works.

If your confidence test fails, reconfigure your C6711 DSK. After you change your board configuration, rerun the confidence utility to check your new settings.

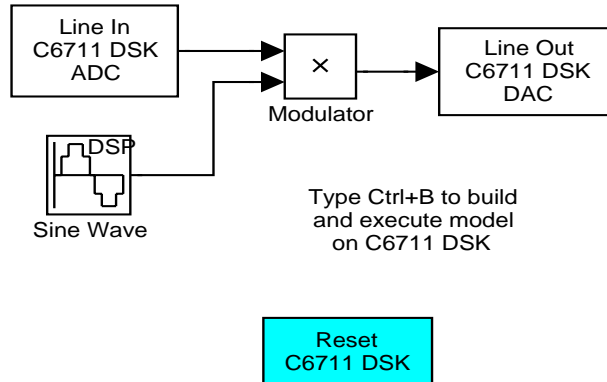
## Testing Your C6711 DSK

The Embedded Target for TI C6000 DSP includes a Simulink demonstration model called `c6711dkstest`. You can use this model to verify that you installed your C6711 DSK hardware and your Embedded Target for TI C6000 DSP software correctly and the board settings are suitable for targeting. The demonstration model presets the Real-Time Workshop settings to build and run the model on your board.

To run the model you need a signal generator, an oscilloscope, and audio cables to connect the signal generator and scope to your C6711 DSK. Refer to your CCS documentation for more information on connecting sources and scopes to your C6711 DSK. In addition, you should connect your signal generator to the oscilloscope input so you can display the source and output signals together.

### To Test the Operation of Your C6711 DSK

As a test to verify that your Embedded Target for TI C6000 DSP software and C6711 DSK are installed and operating correctly, open and build the Simulink model `c6711dsktest`. Test model `c6711dsktest` appears in the following figure.

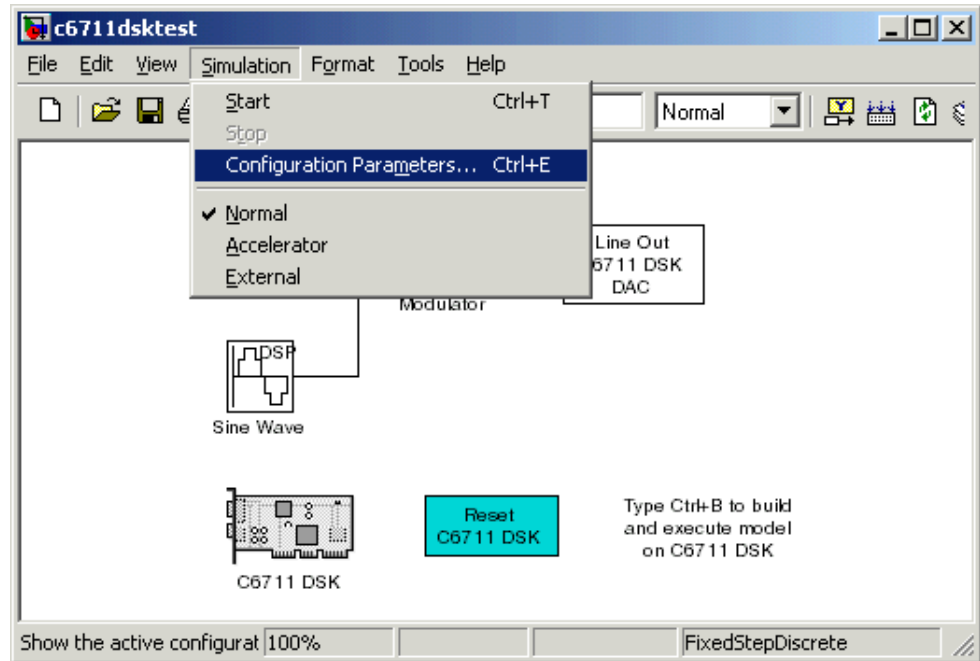


- 1 Enter `c6711dsktest` at the MATLAB command prompt.

Test model `c6711dsktest` opens in Simulink.

- 2 Select **Simulation** > **Configuration Parameters** from the menu bar.

The next figure shows the model `c6711dsktest` with **Configuration Parameters** selected.



- 3 Click Real-Time Workshop in the **Select** tree on the Configuration Parameters dialog box to view the Real-Time Workshop options.
- 4 Click **Build** to run the model. Building the model provides a comprehensive test of the build, download, and run processes in the Embedded Target for TI C6000 DSP.

Real-Time Workshop returns a lengthy series of messages in the Command Window, starting with

```
### Starting Real-Time Workshop build procedure for model:
c6711dsktest.mdl
### Invoking Target Language Compiler on c6711dsktest.rtw
```

If c6711dsktest.mdl builds, compiles, and downloads to the C6711 DSK successfully, the following message strings appear at the end of the build process messages.

```
C6x DSK Command Line COFF Loader Utility, Version 1.20a
Copyright (c) 1998 by DNA Enterprises, Inc.
Found board type:DSK6x Revision:0
Using DSP memory map 1.
### Downloaded:c6711dsktest.out
### Successful completion of Real-Time Workshop build procedure
for model:c6711dsktest
```

When you receive this message, your model is running on the C6711 DSK. You should be able to see the input and output on your oscilloscope. When you change the input, the output should change as well.

Try increasing the frequency you send to the C6711 DSK and watching to see that the output changes to match by changing the amplitude modulation.

### **Error Messages While Building c6711dsktest**

If you receive an error message from the build and compile process, your board or the software may not be configured correctly. Reinstall the board and review the configurations listed in section “Configuring Your C6711 DSK” on page 2-136. You need to resolve errors that appear in this build before you start to develop and build your own models.

Note that after you build and download the model to the board, the build process runs the downloaded code on the C6711 DSK immediately.

### **Verifying That c6711dsktest Is Running**

To see that the model is running, turn on your signal generator and set the output to produce a sine wave at 8000 Hz. Set your oscilloscope to display both the input signal from the signal generator and the output from the C6711 DSK. On the oscilloscope display, you should see the sine wave input from the signal generator, and the amplitude-modulated sine wave output from the C6711 DSK. If you change the frequency of the sine wave input, you should see the change on the oscilloscope in the input and output traces.

### **Starting and Stopping c6711dsktest on the C6711 DSK**

When you build and download the model `c6711dsktest.mdl` to your C6711 DSK, you are not running a simulation of the model. You are running the actual machine code, in real time, corresponding to the block diagram in `c6711dsktest.mdl`. To run `c6711dsktest.mdl` on your C6711 DSK, open the Simulink model and click **Build** on the **Real-Time Workshop** pane to rebuild

the machine executable and download the new executable to the board. Building and downloading the new executable starts the process running on the C6711 DSK.

Once your application is running on your target, stop the process by one of the following methods:

- Using the **Debug > Halt** option in CCS.
- Using `halt` from the MATLAB command prompt.
- Clicking the C6711 DSK Reset block in your model (if you added one) or in the C6711 DSK Board Support library.

## Running Models on Your C6711 DSK

Texas Instruments markets a complete set of tools for use with the C6711 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, the Embedded Target for TI C6000 DSP, and the C6711 DSK Board Support block library.

Executing code generated from Real-Time Workshop 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 a communication link with Simulink, 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 C6711 DSK), you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, the Embedded Target for TI C6000 DSP provides a target makefile specific to C6000 hardware targets. This target makefile invokes the optimizing compiler provided as part of CCS.

Used in combination with the Embedded Target for TI C6000 DSP and Real-Time Workshop, TI products provide an integrated development environment that, once installed, needs no additional coding.

After you have installed the C6711 DSK development board and supporting TI products on your PC, start MATLAB. At the MATLAB command prompt, type `c6711dsklib`. This opens a Simulink block library, `c6711dsklib`, that includes a set of blocks for C6711 DSK I/O devices:

- C6711 DSK ADC—Configures the analog to digital converter
- C6711 DSK DAC—Configures the digital to analog converter
- C6711 DSK LED—Controls the user-defined light emitting diodes (LED) on the C6711 DSK
- C6711 DSK DIP Switch—Sets the dual inline pin switches on the C6711 DSK
- C6711 EVM Reset—Resets the processor on the C6711 DSK

These devices are associated with your C6711 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 and template makefile. For the C6711 DSK, in the **Real-Time Workshop** pane of the dialog box, specify

- **System target file**—`ti_c6000.tlc`
- **Template makefile**—`ti_c6000.tmf`

With this configuration, you can generate and download a real-time executable to your TI C6711 DSK. Start the Real-Time Workshop build process by clicking **Build** on the **Real-Time Workshop** pane. Real-Time Workshop 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 template makefile and block parameter dialog box entries are combined to form the target makefile for your TI

evaluation module. This makefile 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 C6711 DSK, the build process runs the file on the digital signal processor.

### Starting and Stopping DSP Applications on the C6711 DSK

When you create, build, and download a Simulink model to the C6711 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. 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 C6711 DSK, you use Real-Time Workshop 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
- Using `halt` from the MATLAB command prompt
- Encountering a Stop block in the model.
- Clicking the C6711 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.

---

**Note** When you build and execute a model on your C6711 DSK, the Real-Time Workshop build process resets the DSK automatically. You do not need to reset the board before building models. Use the C6711 DSK Reset block to stop processes that are running on your C6711 DSK, or to return your board to a known state for any reason.

---

# Creating Code Composer Studio Projects Without Building

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

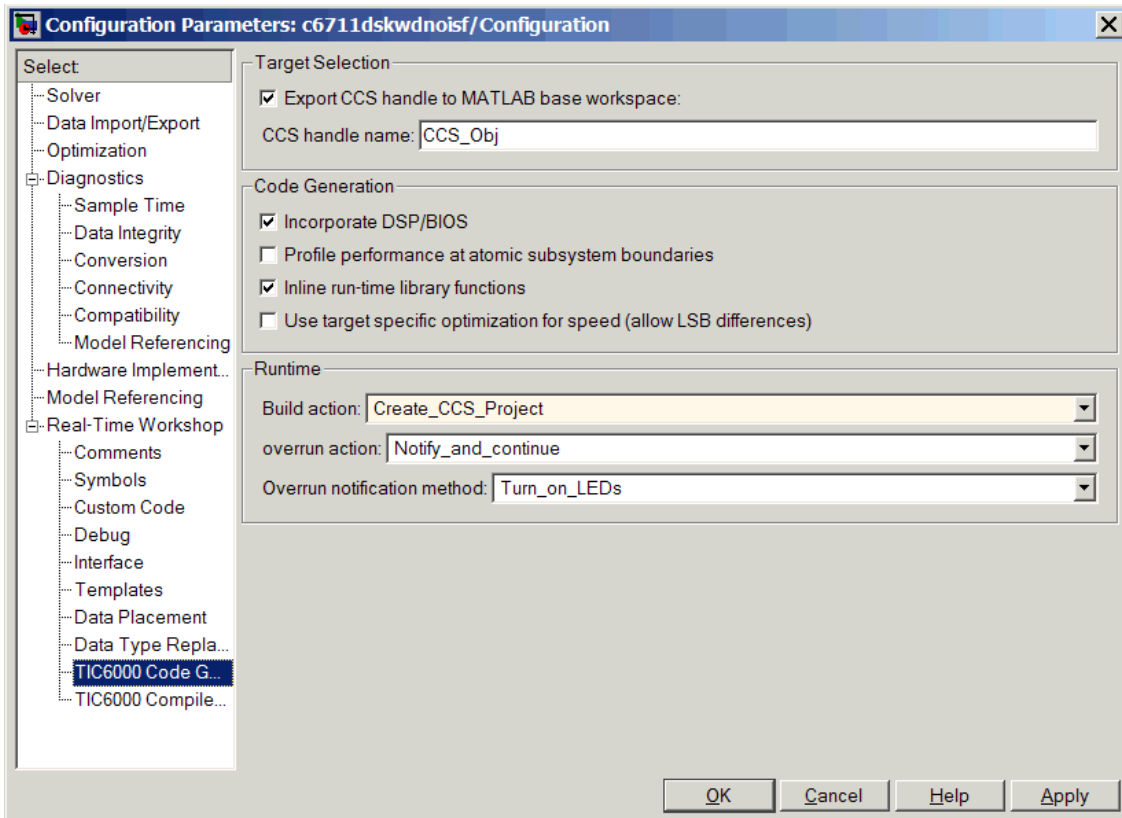
If you build and download your Simulink model to CCS, the Embedded Target for TI C6000 DSP opens Code Composer Studio, creates a new CCS project named for your model, and populates the new project with all the files it creates during the build process—the object code files, the assembly language files, the map files, and any other necessary files. As a result, you can immediately use CCS to debug your model using the features provided by CCS.

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

## Creating Projects in CCS Without Loading Files to Your Target

From the **Select** tree in the Configuration Parameters dialog box, select TI C6000 Code Generation. Select `Create_CCS_Project` for the **Build action**, as shown in the next figure. Note that the `Build` and `Build_and_execute` options create CCS projects as well. The `Generate_code_only` option does not create a CCS project. None of the other options has an effect here. Ignore them when you are creating a project in CCS rather than generating code.





After you select `Create_CCS_Project`, set the options for the **Compiler** and **Linker** categories on the TI C6000 Compiler/Linker category on the **Select** tree.

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

Real-Time Workshop and the Embedded Target for TI C6000 DSP generate all the files for your project in CCS and create a new project in the IDE. Your new project is named for the model you built, with a custom project build configuration `custom_MW`, not Release or Debug.

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

## Targeting Custom Hardware

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

Note that the blocks for the peripherals in the C6000 DSP Library, such as the C6416 DSK ADC or C6711 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. However, the RTDX and core support blocks should work for standard processors.

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 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.
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.

Target Preferences Block	Description
C6711DSK	Sets default values for targeting the C6711 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 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 the block that best matches your target: one of the device-specific blocks, like C6711DSK or the

Custom C6000 block when none of the specific blocks is appropriate. All models that you target to C6000-processor-based 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 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 online help for CCS, using keywords like memory map, memory segment, and section.

Note that 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 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 Configuration Tool in CCS.

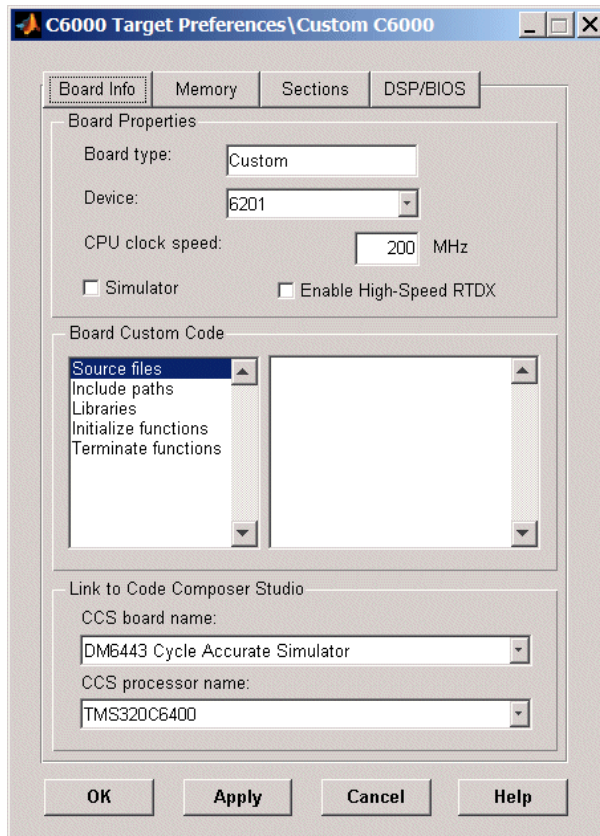
### Targeting a Custom Target

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

Configuring the block parameters tell Simulink, Embedded Target for TI C6000 DSP, and Real-Time Workshop about your target processor and how to generate code that will run on the target.

- 1 After you add the Custom 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 Embedded Target for TI C6000 DSP does not explicitly support.
- 4 Select your target processor from the **Device** 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 speed** 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 high-speed RTDX, meaning that you are using a high-speed RTDX emulator or your hardware configuration supports high-speed RTDX, select **Enable High Speed RTDX**.
- 8 To enable Embedded Target for TI C6000 DSP to connect to CCS, 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.
- 9 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 Embedded Target for TI C6000 DSP and Real-Time Workshop. While this process is necessary, it represents only one small part of enabling you to generate code to run on your custom board.

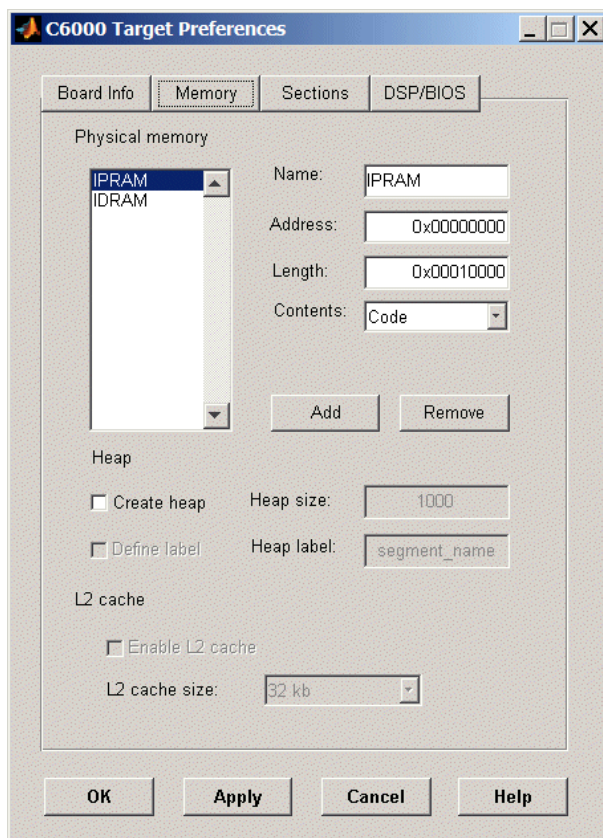
One very important part of targeting custom hardware is to provide the target memory map configuration to the linker and assembler.

**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.

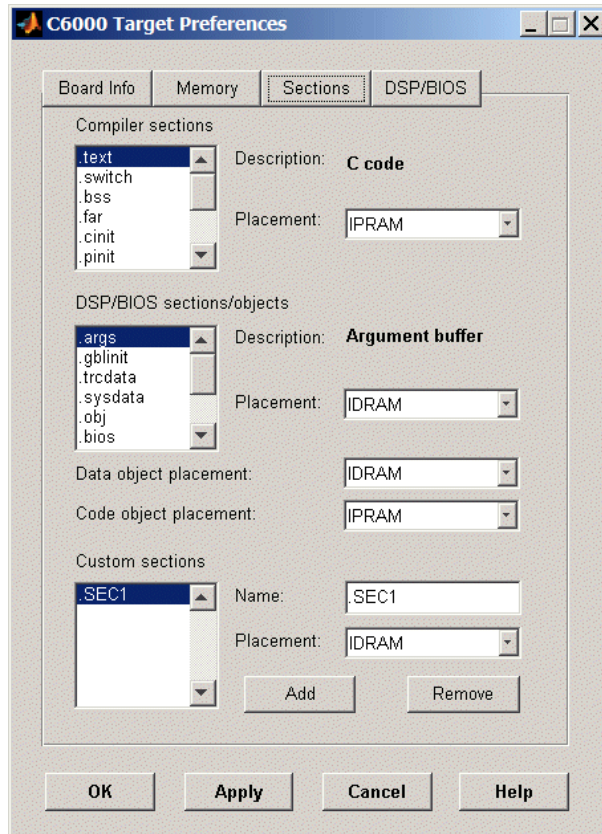
The following figures show the **Memory** and **Sections** panes with the default values for all options.



## 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 online help system for Code Composer Studio.

### 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.
- C6711 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 do the C6711 or C6701, 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.

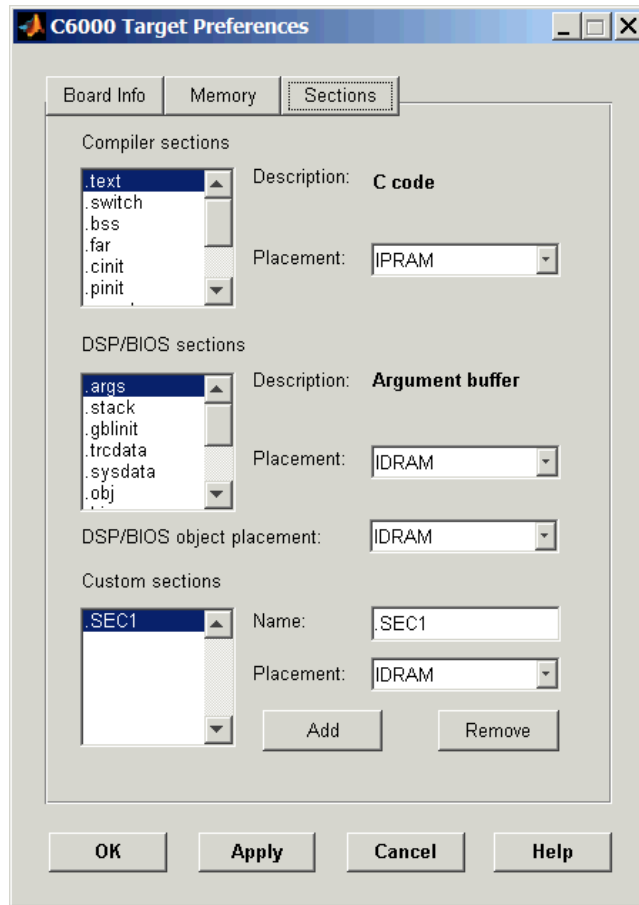
### **L2 Cache Size**

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 online help. Most of the definitions and descriptions in this section come from CCS.



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.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack



<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.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 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 Embedded Target for TI C6000 DSP with Real-Time Workshop Embedded Coder

To take advantage of Embedded Coder features, you must migrate your models to a system target file called `ti_c6000_ert.tlc`. This target is based on the embedded real-time target (ERT) used by Embedded Coder. Other TI C6000 target files are based on the generic real-time target (GRT).

To use Embedded Coder with the Embedded Target for TI C6000 DSP Platform, you must choose the system target file `ti_c6000_ert.tlc`, available in the **System Target File Browser**. If you already have a model with code generation options configured for the target `ti_c6000.tlc`, Embedded Target for TI C6000 DSP provide a special utility function `switchc6000target` to migrate the model instead.

If you simply choose the system target file `ti_c6000_ert.tlc` in the **System Target File Browser** directly to change the target for the model, all the TI C6000 code generation 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 preserve the option values in the configuration set when you migrate your model to the ERT-based target (or back to the GRT-based target), use the function `switchc6000target.m`.

For example, the command

```
switchc6000target(bdroot, 'ti_c6000_ert.tlc')
```

entered at the MATLAB prompt sets your current Simulink model to use the desired system target file—`ti_c6000_ert.tlc`—while preserving the TI C6000 Real-Time Workshop options.

Conversely,

```
switchc6000target(bdroot, 'ti_c6000.tlc')
```

sets your model to use the generic real-time (GRT)-based target.

### To Use the 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 options.
- 3 Click **Browse** to open the **System Target File Browser**.
- 4 On the **System Target File Browser**, find and select the file `ti_c6000_ert.tlc`.
- 5 Click **OK**.

For changing a model that uses the GRT-based target `ti_c6000.tlc` to use the ERT-based target.

- 1 Open your Simulink model to change.
- 2 At the MATLAB prompt, enter

```
switchc6000target(gcs, 'ti_c6000_ert.tlc')
```

Now the current model uses the ERT-based target and the configuration set that you developed for the GRT-based target.

When you return to the Configuration Parameters dialog box and check the **Real-Time Workshop system target file** entry in the Real-Time Workshop pane, you see `ti_c6000_ert.tlc`. The rest of the configuration options are unchanged.

# Targeting with DSP/BIOS Options

---

Introducing DSP/BIOS (p. 3-2)

DSP/BIOS and Targeting Your TI C6000 DSP  
(p. 3-3)

Profiling Generated Code (p. 3-10)

Using DSP/BIOS with Your Target Application  
(p. 3-25)

Introduces DSP/BIOS from Texas Instruments.

Discusses the concepts and files used by Embedded Target for TI C6000 DSP in DSP/BIOS projects.

Demonstrates how to set up and use profiling in your generated code.

Shows you how to add DSP/ BIOS features to your projects when you generate code.

# Introducing DSP/BIOS

The Embedded Target for TI C6000 DSP supports DSP/BIOS™ features as options when you generate code for your target. In the sections that follow, you can read more about what DSP/BIOS is, how the Embedded Target for TI C6000 DSP incorporates the DSP/BIOS features into your generated code, and some ways you might use the real-time operating system (RTOS) features of DSP/BIOS in your application. Follow these links for more information on specific areas that interest you, or read on for more details.

- “DSP/BIOS and Targeting Your TI C6000 DSP” on page 3-3
- “Code Generation with DSP/BIOS” on page 3-6
- “Profiling Generated Code” on page 3-10
- “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 and windows within Code Composer Studio™ to view your program as it executes on the target in real-time.
- DSP/BIOS Configuration Tool—enables you to add and configure any and all 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)—lets you use C or assembly language functions to access and configure DSP/BIOS functions by calling any of over 150 API functions. The Embedded Target for TI C6000 DSP uses the API to let you access DSP/BIOS from MATLAB.

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. Others are not included to avoid adding unused code to your project. In addition, after you add one or more functions from DSP/BIOS, the configuration tool help you disable feature you do not need later, letting you optimize your program for speed and size.

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



## DSP/BIOS and Targeting Your TI C6000 DSP

When you use Real-Time Workshop to generate code from the Simulink model of your digital signal processing application, you can choose to include the DSP/BIOS features provided by the Embedded Target for TI C6000 DSP in your generated code.

By electing to include DSP/BIOS in your generated project, the Embedded Target for TI C6000 DSP adds a DSP/BIOS configuration file (with the filename `modelName.cdb`) to your project, and adds the following files as well:

- `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.

Rather than having you incorporate the DSP/BIOS files manually when you create your application, as you would if you used CCS alone, or another text editor, the Embedded Target for TI C6000 DSP starts from your Simulink model and adds the DSP/BIOS files automatically. As it adds the files it

- Configures the DSP/BIOS configuration file for your model needs
- Sets up the objects you need 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 `.cdb`. 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. Included in any DSP/BIOS project might be

- 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 the Embedded Target for TI C6000 DSP. In the next sections, you learn about which objects the Embedded Target uses and how. Of course, you can still add more objects to your code through CCS. Note, however, that if you add additional DSP/BIOS objects beyond those provided by the Embedded Target for TI C6000 DSP, you lose your additions when you regenerate your 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 choose to use DSP/BIOS capabilities, the interrupts defined in the vector table move to the Hardware Interrupt Service Routine Manager in the CCS Configuration Tool. With all of your interrupts now defined as Hardware

Interrupts (HWI) 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 and segment, and your hardware interrupt vector table to the configuration file, building with the DSP/BIOS option creates a compound linker command file. Since 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, Embedded Target for TI C6000 DSP 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

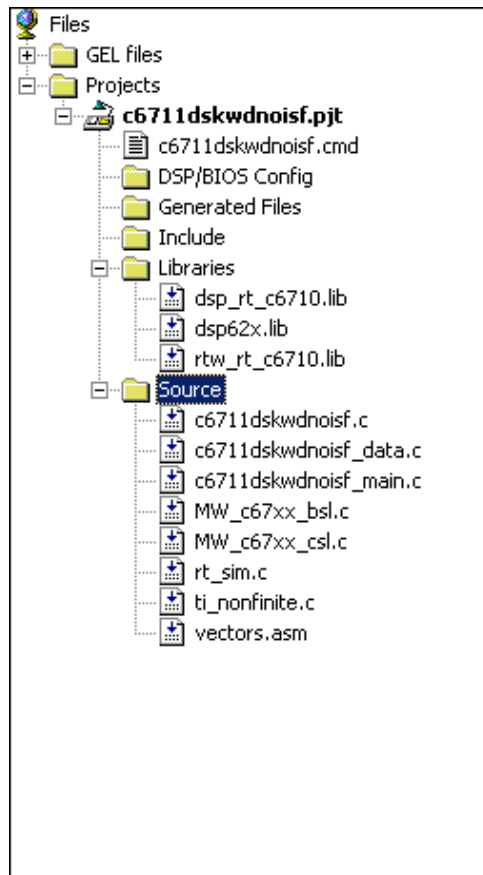
While generating code that includes the DSP/BIOS options is straightforward using the **Incorporate DSP/BIOS** option in the **TIC6000 code generation** options, changes occur between code that does not include DSP/BIOS and code that does. Two things change when you generate code with DSP/BIOS—files are added and removed from the project in CCS, and DSP/BIOS objects become part of your generated code. With these in place, you can use the DSP/BIOS features in CCS to debug your project, as well as use the profiling option in Embedded Target for TI C6000 DSP to check the performance of your application running on your target.

#### 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.

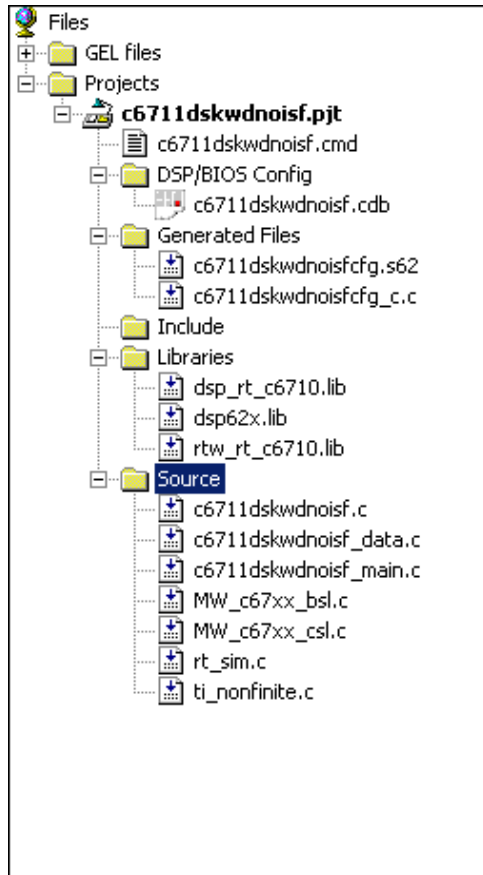
#### Example—c6711dskwdnoisf.pjt code Generated Without DSP/BIOS

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



### Example—c6711dskwdnoisf.pjt Code Including DSP/BIOS

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



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

Added File	Description
modelName.cdb	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 .cdb file

Added File	Description
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.

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 <b>HWI</b> section of the Configuration Tool.
Original linker command file— modelname.cmd	Assigns memory sections on the processor. This file is removed if the <b>SECTION</b> 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

When you use the Embedded Target for TI C6000 DSP 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 performance at atomic subsystem boundaries** in the Real-Time Workshop options, Real-Time Workshop 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 back to MATLAB and display the information in a custom HTML report.

Code profiling works only on atomic subsystems in your model. To allow Embedded Target for TI C6000 DSP to profile your model when you build it in Real-Time Workshop, 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, choose **Tools > Real-Time Workshop > Options** from the model menu bar. Navigate to the **TI C6000 code generation** category, and select the **Profile performance at atomic subsystem boundaries** check box.

## 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

The Embedded Target for TI C6000 DSP 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, 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 Embedded Target for TI C6000 DSP 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_getltime()` and `CLK_gethtime()` can be inaccurate.

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

```
A = CLK_gethtime();
IRQ_globalDisable();
{
    Code segment;
}
IRQ_globalEnable();
B = CLK_gethtime();
```

In this situation, if interrupts are disabled longer than 1ms around the code segment to be profiled, B might be smaller than A since CTNO 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 `.cdb` 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, Embedded Target for TI C6000 DSP 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 **Tools > Real-Time-Workshop > Options** 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 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.

---

**Note** 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 to measure subsystem performance, Embedded Target for TI C6000 DSP provides a custom HTML report that analyzes and displays the profile statistics. The HTML page shows you the amount of time spent computing each subsystem, including both `Outputs` and `Update` code segments, and provides links to 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` and `report` is one of the input arguments for `profile`.

When you generate the report, Embedded Target for TI C6000 DSP stores the report in your code generation working directory, something like `modelName.c6000.rtw`, with the name `profileReport.html`.

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

```
tempdir
```

at the MATLAB command prompt.

---

**Note** Each time you run the profiling process, Embedded Target for TI C6000 DSP 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 directory in MATLAB.

---

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 project that was previously created with Real-Time Workshop, 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 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_gettime());
```

or

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

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

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

In some cases, Real-Time Workshop 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 `cdb` file.

The interrupt rate for a DSP/BIOS application created by the Embedded Target for TI C6000 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.

## 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 `c6711dskwdnoisf` 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  $4*x$ , 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 C6701 EVM, 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 options. Use the **Current C6701EVM CPU clock rate** option on the TIC6000 runtime category on the RTW tab.

Set at a fixed 150 MHz, you cannot change the CPU clock rate on the C6711 DSK. You do not need to report the setting in the Real-Time Workshop 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 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 to support the profiling features in Embedded Target for TI C6000 DSP. 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.
- 3 Create atomic subsystems to profile in your model.
- 4 Build, download, and run your model.
- 5 In MATLAB, use `profile` to view the profile report.

To demonstrate profiling generated code, this procedure uses the wavelet denoising model `c6711dskwdnoisf.mdl` that is included with the Embedded Target for TI C6000 DSP demo programs. If you are using the C6701 EVM as your target, use the model `C6710evmwdnoisf` instead throughout this procedure. Simulators work as well, just choose the appropriate model for your simulator.

Begin by loading the model, entering

```
c6711dskwdnoisf
```

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

### To Enable Profiling for Your Generated Code

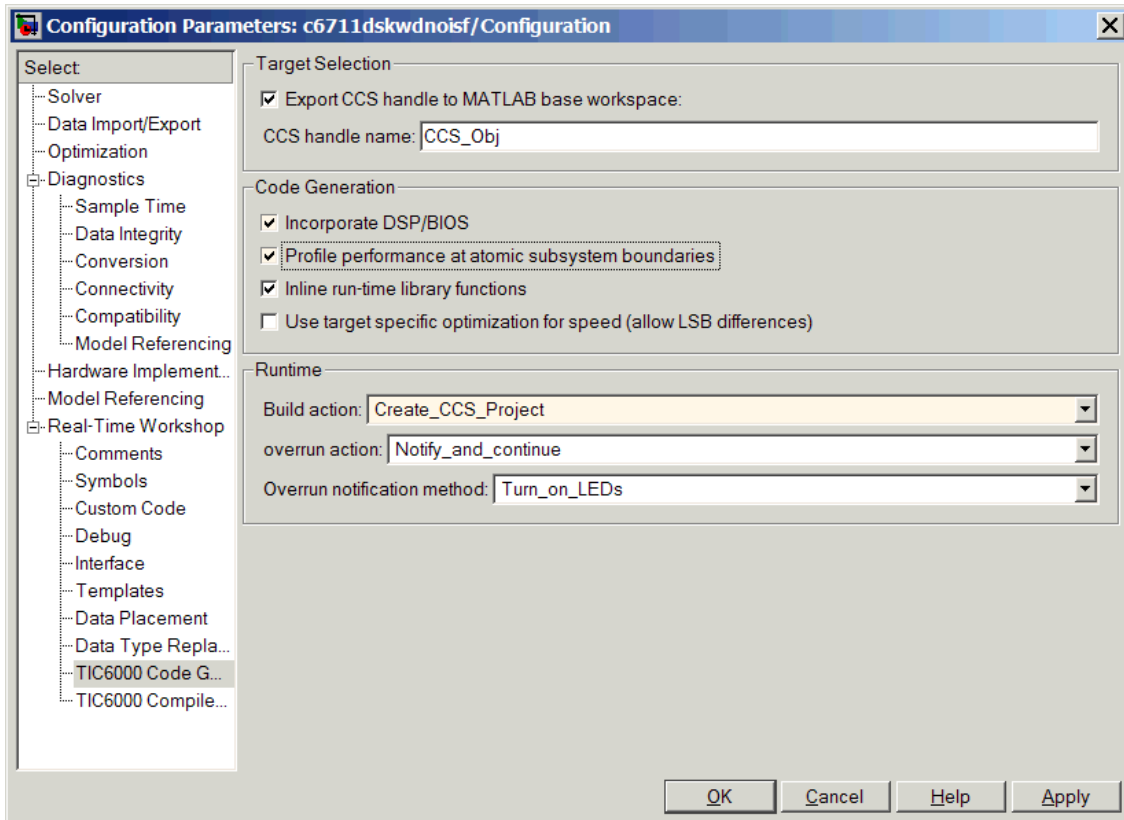
Recall that you must use DSP/BIOS in your code to use profiling.

- 1 To enable the profile feature for your Simulink model, select **Tools > Real-Time Workshop > Options...** from the model menu bar.

The **Simulation Parameters** dialog opens for you to set the code generation options for your model.

- 2 Click **Real-Time Workshop** to display the configuration panes for setting your code generation options.
- 3 From the **Category** list, select **TI C6000 Code Generation**.

Your display changes to show the options you set to control code generation for TI C6000 targets, as shown here.



- 4 Select the **Profile performance at atomic subsystem boundaries** option. Selecting this option enables profiling in your generated code. However, you still need to configure your model to support the profiling process.

## 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. Note that 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 Embedded Target for TI C6000 DSP to see how your code runs and check the performance for bottlenecks and slowdowns as the code runs on your target.

---

**Note** 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 RTW 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 by entering the following command

```
cc = ccstdsp;
```

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 portion of a sample report provided here; your results may differ based on your target and your 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 <sup>1</sup>	720 MHz

### Profiled Simulink Subsystems

System name	<a href="#">c6416dskprofile</a>
STS object	stsSys2_OutputUpdate
Max time spent in this subsystem per interrupt	14.93 $\mu$ s
Max percent of base interval	0.00597%
Number of iterations counted	144

System name	<a href="#">c6416dskprofile.Subsystem1</a>
STS object	stsSys1_OutputUpdate
Max time spent in this subsystem per interrupt	12.8 $\mu$ s
Max percent of base interval	0.00512%
Number of iterations counted	144

## Using DSP/BIOS with Your Target Application

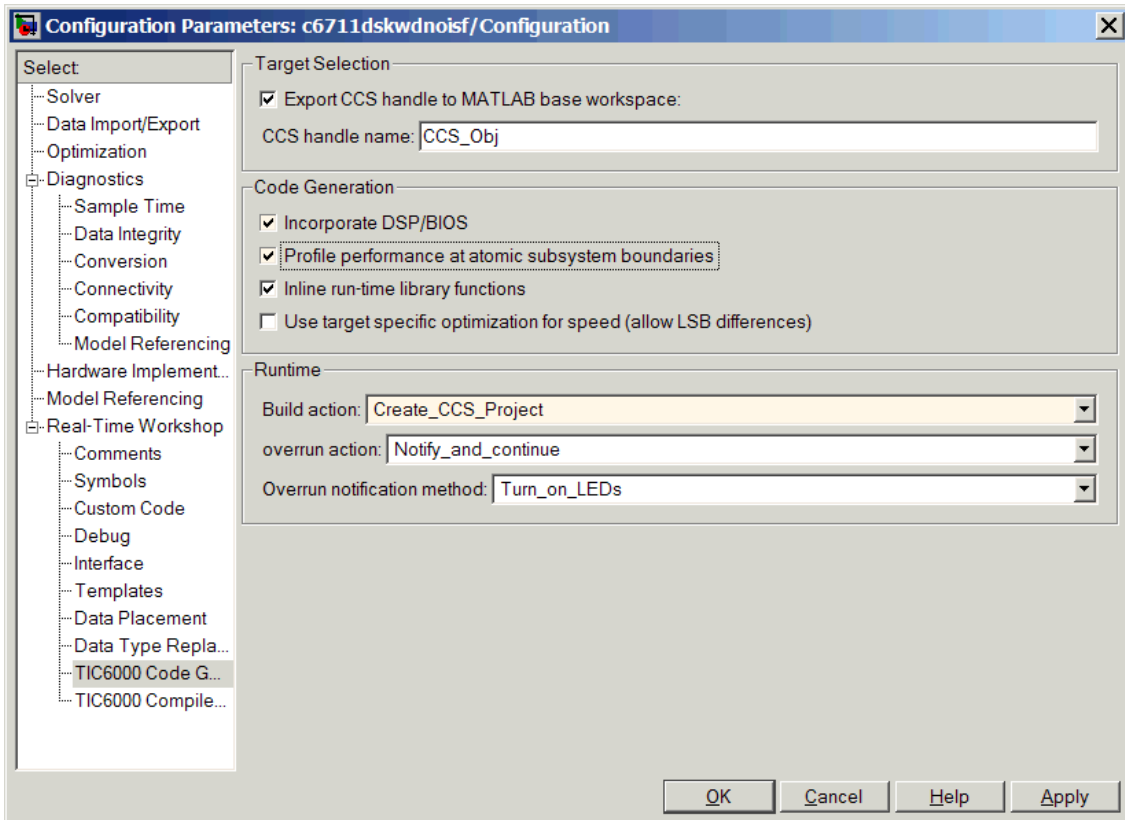
The Embedded Target for TI C6000 DSP lets you build projects and generate code with or without DSP/BIOS included.

### To Enable DSP/BIOS When You Generate Code

For any code you generate using Real-Time Workshop and the Embedded Target for TI C6000 DSP, 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 TI C6000 Code Generation settings—**Incorporate DSP/BIOS**.

- 1 Open the model to use to generate code.
- 2 From your model menu bar, select **Simulation > Simulation parameters...** to start the **Simulation Parameters** dialog.
- 3 From the **Category** list, select TI C6000 Code Generation.

To provide access to the options, the display changes to show the following options.



- 4 As shown in the figure, select **Incorporate DSP/BIOS**.
- 5 Using the other entries on the **Category** list, set other options as you require for your project.
- 6 For the **Build action** (under **Runtime**), select one of the following choices. Each option generates code that includes the DSP/BIOS instrumentation:
  - Create\_CCS\_project
  - Build



- `Build_and_execute`

Notice that the `Generate_code_only` option is not on the preceding list. Using the `Generate_code_only` option does not generate DSP/BIOS enabled code.

**7** Return to Real-Time Workshop on the **Select** tree.

**8** Click **Make Project**, **Build**, or **Build & Run** to generate code.



# Using the C62x and C64x DSP Libraries

---

About the C62x and C64x  
DSP Libraries (p. 4-2)

Fixed-Point Numbers (p. 4-4)

Building Models (p. 4-8)

Introduces the C62x and C64x DSP libraries

Discusses the representation of fixed-point numbers in  
the C62x and C64x DSP libraries

Reviews some issues to consider when you build models  
with blocks from the C62x or C64x DSP libraries

# About the C62x and C64x DSP Libraries

## 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 before generating code. Once you develop your model, you can invoke Real-Time Workshop to generate code that is optimized to run on the C6711 DSK or C6701 EVM 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 5, “Blocks — By Category” discusses the data types accepted and produced by each block in the library. “Fixed-Point Numbers” on page 4-4 gives a brief overview of using fixed-point data types in Simulink. 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 and Simulink. To learn more about creating models that include both C62x DSP Library blocks and blocks from other blocksets, refer to “Building Models” on page 4-8.

## 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 before generating code. Once you develop your model, you can invoke Real-Time Workshop 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 5, “Blocks — By Category” discusses the data types accepted and produced by each block in the library. “Fixed-Point Numbers” on page 4-4 gives a brief overview of using fixed-point data types in Simulink. 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 certain blocks from the Signal Processing Blockset and Simulink. To learn more about creating models that include both C64x DSP Library blocks and blocks from other blocksets, refer to “Building Models” on page 4-8.

---

**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. (Embedded Target for TIC6000 generates a warning message when you try to do this but allows you to use the block.)

You cannot use the C64x blocks on your C62x target.

---

## 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.

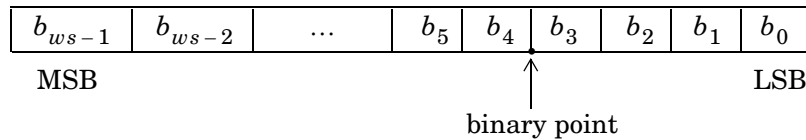
To learn more about characteristics particular to each block in the library, refer to Chapter 5, “Blocks — By Category.”

## Fixed-Point Numbers

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.

## 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 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

$$Qm.n$$

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

Q0.15

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 the Fixed-Point Toolbox, this data type is expressed as

`sfrac16`

or

`sfix16_En15`

The Filter Design Toolbox 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 the Fixed-Point Toolbox, this data type is expressed as

`sfix32_En30`

In the Filter Design Toolbox, 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 the Fixed-Point Toolbox, this data type is expressed as

`sfix16_En17`

To express this data type in the Filter Design Toolbox, 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 the Fixed-Point Toolbox, this data type is expressed as

`sfix16_E2`

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

`[16 -2]`

# Building Models

You can use C62x or C64x DSP Library blocks in models along with certain core Simulink and Signal Processing Blockset. 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 5, “Blocks — By Category” 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 blocks and Simulink blocks also accept fixed-point data types. Make the appropriate settings in these blocks’ parameters when you connect them to a C62x DSP Library block.

However, to use Signal Processing Blockset 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 Simulink Data Type Conversion block in the Data Type library of Simulink.
- 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 in the C62x DSP Library.
- To connect blocks of varying nonfixed-point data types in your model, use the Data Type Conversion block in the Signals and Systems Simulink library
- To connect blocks of varying fixed-point data types in your model, use the Simulink Data Type Conversion Inherited block in the Data Type library of Simulink.

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 in your models with C62x DSP Library blocks. See “Converting Data Types” on page 4-8 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 all have Autocorrelation blocks. How do you choose which to include in your model? If you are building a model to run on the C6711 DSK or C6701 EVM, 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.



# Blocks — By Category

---

Target Preferences (c6000tgtprefs) (p. 5-2)	Configure targets for code generation and targeting
RTDX Instrumentation (rtDXblocks) (p. 5-3)	Add RTDX instrumentation to generated code
C62x DSP (tic62dsplib) (p. 5-4)	Work with C62x processor targets
C64x DSP (tic64dsplib) (p. 5-7)	Work with C64x processor targets
C6416 DSK (c6416dsklib) (p. 5-10)	Work with C6416 DSK targets
C6711 DSK (c6711dsklib) (p. 5-11)	Work with C6711 DSK targets
C6713 DSK (c6713dsklib) (p. 5-12)	Blocks provided for C6713 DSK targets
DM642 EVM (dm642evmlib) (p. 5-13)	Work with DM642 EVM targets
C6000 DSP Core Support (c6000dspcorelib) (p. 5-14)	Work with all C6000 targets
TMDX3206040A DSP Support (tmdx326040lib) (p. 5-15)	Work with target that uses TMDX3206040A daughter cards
Host Communication (hostcommlib) (p. 5-16)	Work with host-side models that communicate with C6000 targets
C6000 DSP Communication (targetcommlib) (p. 5-17)	Work with target-side models that communicate with hosts such as xPC or host-side models
DSP/BIOS (dspbioslib) (p. 5-18)	Work with C6000 models to provide DSP/BIOS tasks and interrupts

## **Target Preferences (c6000tgtprefs)**

C6416DSK	Configure model for C6416 DSP Starter Kit
C6455DSK	Configure model for C6455 DSP Starter Kit
C6711DSK	Configure model for C6711 DSP Starter Kit
C6713DSK	Configure model for C6713 DSP Starter Kit
C6727PADK	Configure model for C6727 Professional Audio Development Kit
Custom C6000	Configure model for C6000-processor-based custom hardware targets
DM642EVM	Configure model for DM642 Evaluation Module

## RTDX Instrumentation (rtdxblocks)

From Rtdx	Add RTDX communication channel to send data from MATLAB to target
To Rtdx	Add RTDX communication channel to send data from target to MATLAB

## **C62x DSP (tic62dsplib)**

Conversions (p. 5-4)	Convert data types
Filters (p. 5-4)	Filter input signals
Math and Matrices (p. 5-5)	Perform mathematical operations
Transforms (p. 5-5)	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

C62xRadix-2 FFT

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

C62xRadix-2 IFFT

Radix-2 inverse FFT of complex input vector

## C64x DSP (tic64dsplib)

Conversions (p. 5-7)	Data conversion
Filters (p. 5-7)	Filter input signals
Math and Matrices (p. 5-8)	Mathematical operations
Transforms (p. 5-8)	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

## **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

## **C6711 DSK (c6711dsklib)**

C6711 DSK ADC	Digitized output from codec to processor
C6711 DSK DAC	Use codec to convert digital input to analog output
C6711 DSK DIP Switch	Simulate or read DIP switches
C6711 DSK LED	Control LEDs
C6711 DSK RESET	Reset to initial conditions

## **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



## 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

## **C6000 DSP Core Support (c6000dspcorelib)**

CPU Timer	Select timer and configure periodic interrupt
Hardware Interrupt	Generate Interrupt Service Routine
Idle Task	Create free-running task
Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

## **TMDX3206040A DSP Support (tmdx326040lib)**

TMDX326040 ADC

Configure codec on daughter card for output to C6711 DSK

TMDX326040 DAC

Configure codec on daughter card to send data to analog output on card

## Host Communication (hostcommlib)

Byte Pack	Convert input signals into uint8 vector
Byte Reversal	Reverse order of bytes in input word
Byte Unpack	Unpack UDP uint8 input vector into Simulink data type values
UDP Receive	Receive uint8 vector as UDP message
UDP Send	Send UDP message to target

## **C6000 DSP Communication (targetcommlib)**

Byte Pack	Convert input signals into uint8 vector
Byte Reversal	Reverse order of bytes in input word
Byte Unpack	Unpack UDP uint8 input vector into Simulink data type values
C6000 IP Config	Internet protocol configuration on C6000 target
C6000 TCP/IP Receive	Receive message from remote IP address
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

## **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

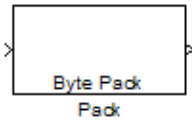
## **Blocks — Alphabetical List**

# Byte Pack

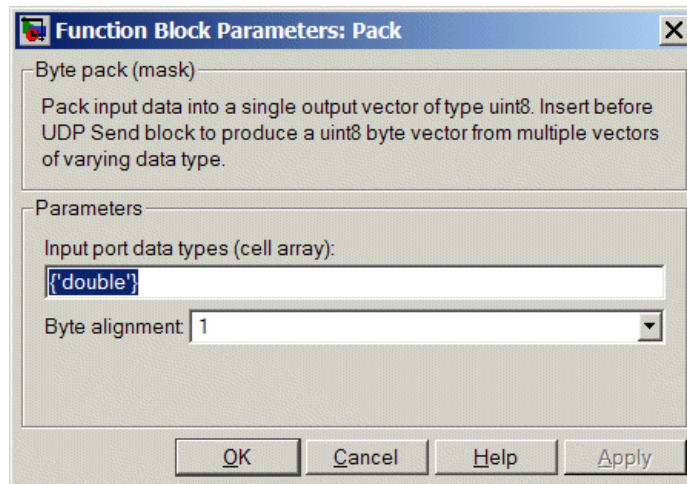
**Purpose** Convert input signals into uint8 vector

**Library** Host Communication Library in Embedded Target for TI C6000 DSP

**Description** Using the input port, the block converts data of one or more data types into a single uint8 vector for output. With the options available, you specify the input data types and the alignment of the data in the output vector. Since UDP messages are in uint8 data format, use this block before a UDP Send block to format the data for transmission using the UDP protocol.



## Dialog Box



### Input port data types (cell array)

Specify the data types for the different signals as part of the parameters. The block supports all Simulink data types except characters. Enter the data types as Simulink types in the cell array, such as 'double' or 'int32'. The order of the data type entries in the cell array must match the order in which the data arrives at the block input. Signal sizes are determined automatically by the block. The block always has at least one input port and only one output port.

### Byte alignment

This option specifies how the data types are aligned to form the uint8 output vector. Select one of the values in bytes from the list.



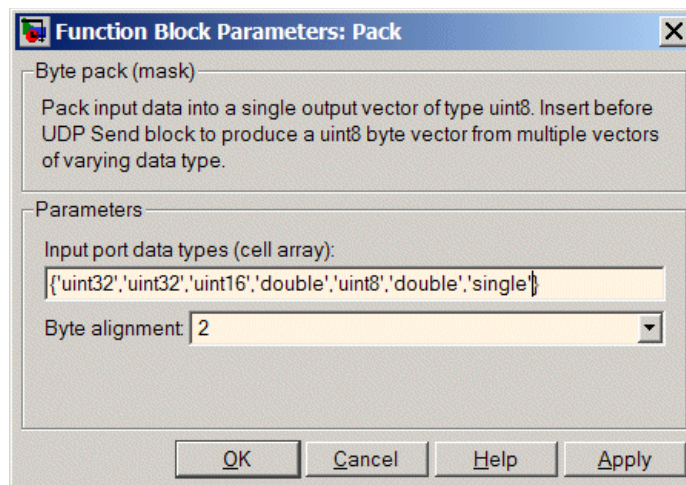
Alignment can occur on 1, 2, 4, or 8 byte boundaries depending on the value you choose. The default is 1. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithm ensures that each element in the output vector begins on a byte boundary specified by the alignment value and relative to the starting point of the vector.

Selecting 1 for **Byte alignment** provides the tightest packing, with no holes between any data types for any combination of data types and signals.

In general, when you have multiple data types of varying lengths, specifying two-byte alignment means there might be gaps of 1 byte between a uint8 or int8 value and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time. You can specify any of the data alignment options with any of the data types.

## Example

As you see in the following figure, enter input data types in a cell array in **Input port data types**. The order of the data types you enter must match the order of the data types at the block input.



In the cell array, you provide the order in which the block expects to receive data — uint32, uint32, uint16, double, uint8, double, and single. With this information, the block automatically provides the proper number of input ports.

# Byte Pack

---

Byte alignment equal to 2 specifies that each new value begins 2 bytes from the previous data boundary.

In the example shown, the data types are

```
{'uint32', 'uint32', 'uint16', 'double', 'uint8', 'double', 'single'}
```

Assuming that all of the signals are scalars (no matrices or vectors in this example), the first signal value in the vector starts at 0 bytes, the second at 2 bytes, the third at 4 bytes, the fourth at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. Notice that the packing algorithm leaves a one byte gap between the `uint8` data value and the `double` value.

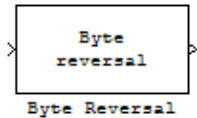
## See Also

Byte Reversal, Byte Unpack

**Purpose** Reverse order of bytes in input word

**Library** Host Communication Library in Embedded Target for TI C6000 DSP

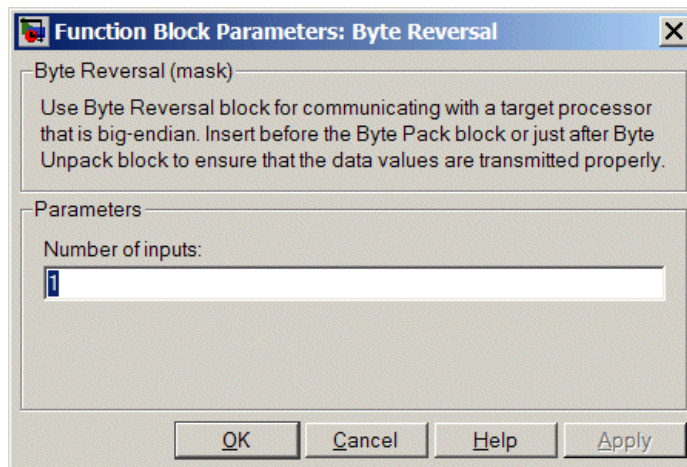
## Description



Byte reversal changes the order of the bytes in data you input to the block. Use this when your process communicates between targets that use different endianness, such as between Intel processors that are little-endian and others that are big-endian. Texas Instruments processors are generally little-endian by default.

When you transmit data to a processor with different endianness, place a byte reversal block just before the send block in a model and immediately after the receive block to ensure that transmitted data has the correct endianness.

## Dialog Box



### Number of inputs

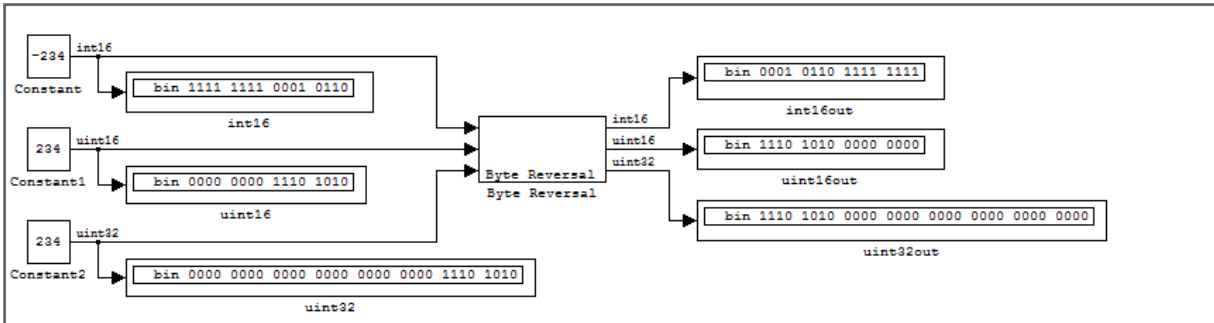
Specify the number of input ports for the block. The number of input ports adjusts automatically to match value so the number of outputs is equal to the number of inputs.

When you use more than one input port, each input port maps to the matching output port. Data entering input port 1 leaves through output port 1 and so on.

# Byte Reversal

Reversing the bytes does not change the data type. Input and output retain matching data type.

The following model shows byte reversal in use. Notice that the input and output ports match for each path.



## See Also

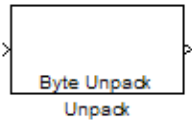
Byte Pack, Byte Unpack

**Purpose**

Unpack UDP uint8 input vector into Simulink data type values

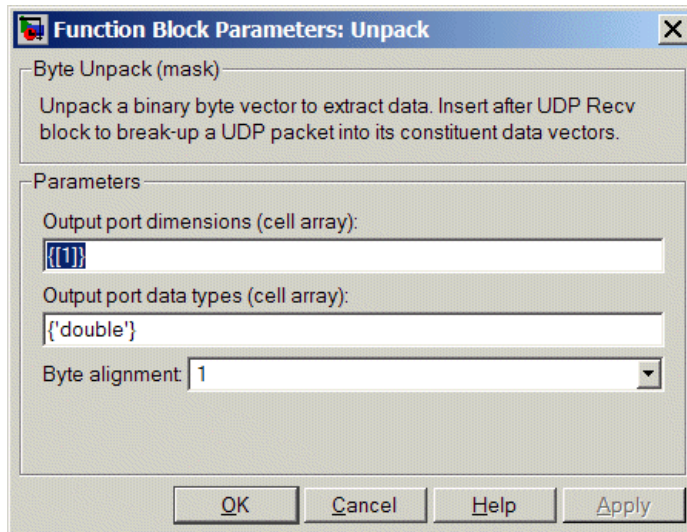
**Library**

Host Communication Library in Embedded Target for TI C6000 DSP

**Description**

Byte Unpack is the inverse of the Byte Pack block. It pairs with the UDP Receive block in models, receiving a vector of uint8 from a UDP message and outputting Simulink data types in different sizes depending on the input vector.

The block supports all Simulink data types.

**Dialog Box****Output port dimensions (cell array)**

Containing a cell array, each element in the array specifies the dimension that the size function in MATLAB returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding Byte Pack block in the model. Entering one value means the block applies that dimension to all data types.

**Output port data types (cell array)**

Specify the data types for the different input signals to the Pack block. The block supports all Simulink data types — single, double, int8, uint8,

# Byte Unpack

int16, uint16, int32, and uint32, and boolean. The entry here is the same as the Input port data types parameter in the Byte Pack block in the model. You can enter one data type and the block applies that type to all output ports.

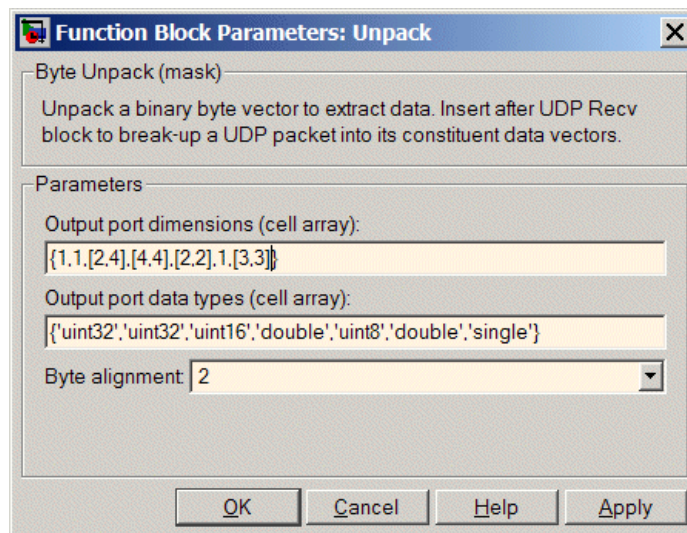
## Byte Alignment

Specifies how the data types are aligned in the input uint8 vector. This should match the corresponding Byte Pack block alignment value, and supports the same settings of 1, 2, 4, and 8 bytes.

## Example

Here is an example of the Byte Unpack block that corresponds to the example in the Byte Pack example. The **Output port data types (cell array)** entry here is the same as the **Input port data types (cell array)** entry in the Byte Pack block

```
{'uint32','uint32','uint16','double','uint8','double','single'}.
```



In addition, the **Byte alignment** setting matches as well. **Output port dimensions (cell array)** now includes scalars and matrices to demonstrate entering nonscalar values. The example for the Byte Pack block assumed only scalar inputs.

**See Also**

Byte Pack, Byte Reversal

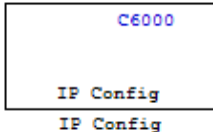
# C6000 IP Config

---

**Purpose** Internet protocol configuration on C6000 target

**Library** C6000 DSP Communication Library in Embedded Target for TI C6000 DSP

## 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, C6711, 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-3 for more information about configuring the card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

By default, the block uses dynamic addressing, getting the address from the local server. 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. 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 **Use the following CPU interrupt for Ethernet driver (4-13)** options.

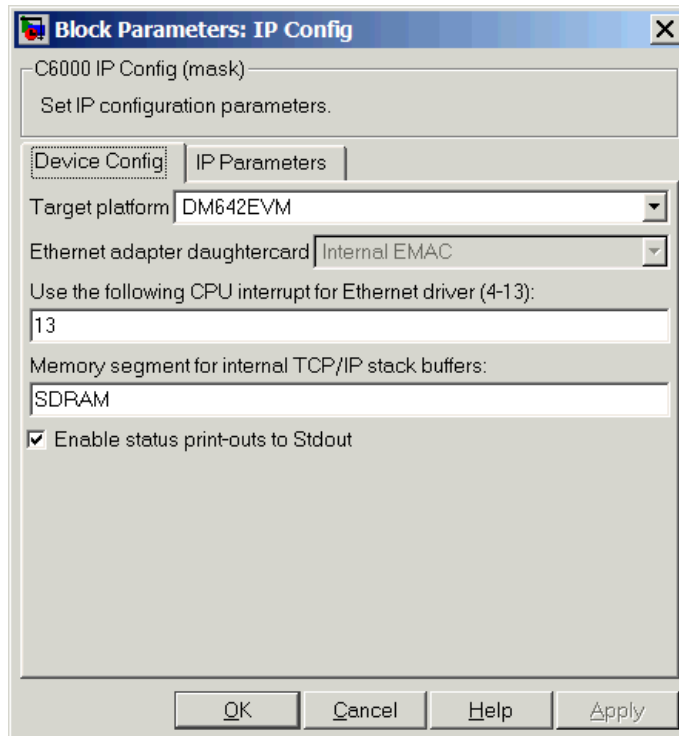
When you build and run your model, this block has no effect. It outputs zeros. When you generate code from your model, this block adds the code that configures IP on your board.



## 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.

## **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 for DM642 EVM (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 for DM642 EVM**

Specify an IP address for the DM642 EVM. This value is the address that others use to communicate with the evaluation module over IP. Use the full xxx.xxx.xxx.xxx format.

# C6000 IP Config

---

## **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.

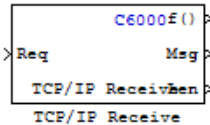
## **See Also**

C6000 TCP/IP Receive, C6000 TCP/IP Send

**Purpose** Receive message from remote IP address

**Library** C6000 DSP Communication Library in Embedded Target for TI C6000 DSP

**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, C6711, 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-3 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. Msg port width is defined by **Maximum read size** in the parameters, which also defines the largest message the block can pass to downstream blocks. Use `uint8_T` for message read size parameter.

A second block output is a function call port that issues a function call whenever a new message is available on the receive buffer. The third output, Len, data type `uint16_T`, specifies the length of the message to pass to downstream blocks for variable length messages.

To specify the length of the TCP/IP message read request, provide the message sizes as the input to the Req port. The block returns a message that contains the number of bytes specified to be read by the value at the Req port.

If the number of bytes to read is greater than the value of **Maximum read size**, the block ignores the maximum read size and reads as many bytes of data as the output port Msg can accommodate. The output is in `uint16_T` format.

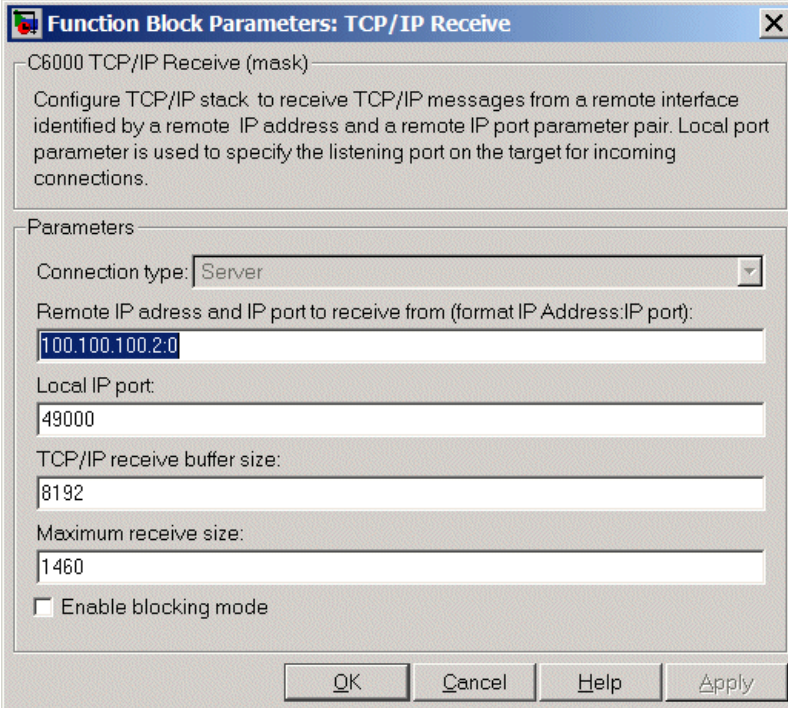
In simulations, this block outputs a stream of `uint8_T` data from the Msg port with the first (Req) bytes set to `0xFF` and the rest set to `0x00`. When the function call port exists, it generates a function call for every sample time hit.

The Len port outputs a scalar that matches the number of requested bytes coming from the Req port. Models that contain this block generate code for the

# C6000 TCP/IP Receive

parameters that configure TCP/IP on the target, including the ports, buffers, and message sizes.

## Dialog Box



The dialog box is titled "Function Block Parameters: TCP/IP Receive". It contains a description of the block's function and a "Parameters" section with several input fields and a checkbox.

C6000 TCP/IP Receive (mask)  
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.

Parameters

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

Maximum receive size:  
1460

Enable blocking mode

Buttons: OK, Cancel, Help, Apply

### Connection type

**Connection type** specifies the connection initiation method used for the block. Choose either **Server** or **Client** from the list. This is a read-only parameter. The setting is for your information — you cannot change it.

When you set this to **Server**, you create 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.

## **Maximum receive size**

Specifies the width in bytes of the output port. Set this value to the maximum number of bytes you expect to read out of the TCP/IP receive buffer during program execution. The block provides this parameter because every Simulink block needs to specify its output port dimension statically. Port dimension cannot change at run time.

# C6000 TCP/IP Receive

---

## **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**

When the block is in non blocking mode, this value, in seconds, specifies how frequently the block checks the receive buffer for new data.

## **See Also**

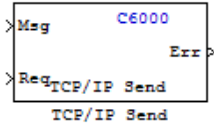
C6000 TCP/IP Send, C6000 UDP Receive, C6000 UDP Send



**Purpose** Send message to remote IP interface

**Library** C6000 DSP Communication Library in Embedded Target for TI C6000 DSP

**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, C6711, 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-3 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. One block input (Msg) is the data vector with the message in data type `uint8_T`.

A second input is the length of the message to send. Provide this length as a `uint16_T` data type. If the length value is smaller than the Msg port width, the block transmits only the portion of the message that fits the specified length. The rest of the message is not sent.

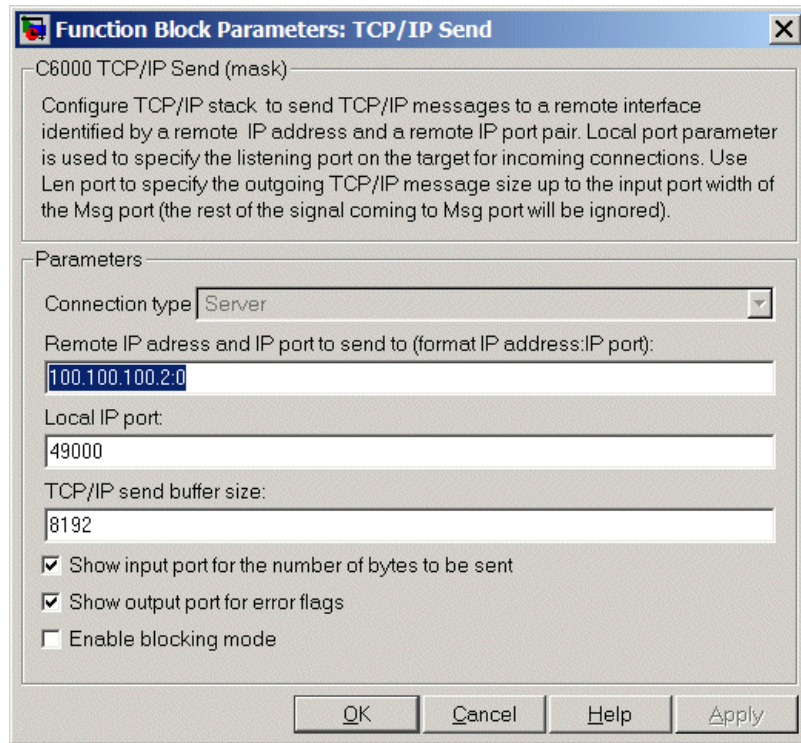
The optional output port Err specifies error conditions when a send operation fails. Completed send operations return 0 at the Err output port.

In simulations, this block acts as a sink. It discards data passed to it via the Msg port. If you enable the Err output port, it is set to 0 for simulation.

Models that contain this block generate code for the parameters that configure TCP/IP on the target, including the ports, buffers, and message sizes.

# C6000 TCP/IP Send

## Dialog Box



### Connection type

**Connection type** specifies the connection initiation method used for the block. Choose either **Server** or **Client** from the list. This is a read-only value. You cannot change the setting from **Server** to **Client**.

When you set this parameter to **Server**, you create a listening socket at the IP address and port you enter 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.

## **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.

## **Show input port for the number of bytes to be sent**

Selecting this option adds an additional input port, **Len**, to the block. **Len** specifies the number of bytes to send in the message. Use this option to dynamically change the size of outgoing TCP/IP messages.

## **Show output port for error flags**

Selecting this option adds a block output port for error messages. If the send operation is successful, this port carries a value of 0 to indicate

# C6000 TCP/IP Send

a successful send. The port carries a nonzero value to indicate an error condition when the send operation fails. Four error codes exist.

<b>Output Error Flag</b>	<b>Meaning</b>	<b>Generated Code Entry</b>
(-1)	Socket not connected	<code>#define MW_ENOTCONNECTED (-1)</code>
(-2)	Reserved for future use. Block does not output this error condition	<code>#define MW_ESENDABORTED (-2) // Reserved</code>
(-3)	Socket operation failed with an error from the TCP/IP stack	<code>#define MW_ESOCKETERROR (-3)</code>
(-4)	Reserved for future use. Block does not output this error condition	<code>#define MW_ETIMEDOUT (-4) // Reserved</code>

## Enable blocking mode

Select this option to put the TCP/IP send process in blocking mode so that the block sends messages before accepting new input messages. In blocking mode, if a connection cannot be established, the send operation blocks further execution until the connection is established and the data in the message buffer is sent. Selecting **Enable blocking mode** adds the **Timeout** option to the block parameters.

In non blocking mode, the block sends data only when it can establish a connection and there is sufficient space in the TCP/IP send buffer. If either of these conditions is not met, model execution skips the send operation and continues with the next downstream block.

**See Also**

C6000 TCP/IP Receive, C6000 UDP Receive, UDP Receive

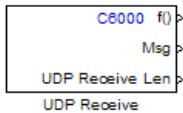
# C6000 UDP Receive

---

**Purpose** Receive uint8 vector as UDP message

**Library** C6000 DSP Communication Library in Embedded Target for TI C6000 DSP

## 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. The third output specifies the length of the message for variable length messages.

To use this block with the C6416, C6711, 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-3 for more information about configuring the card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

This block read a single UDP packet every sample hit. The length of the packet read from the receive buffer is returned via the Len port. 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 truncated and lost as a result. The missing message content cannot be retrieved.

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.

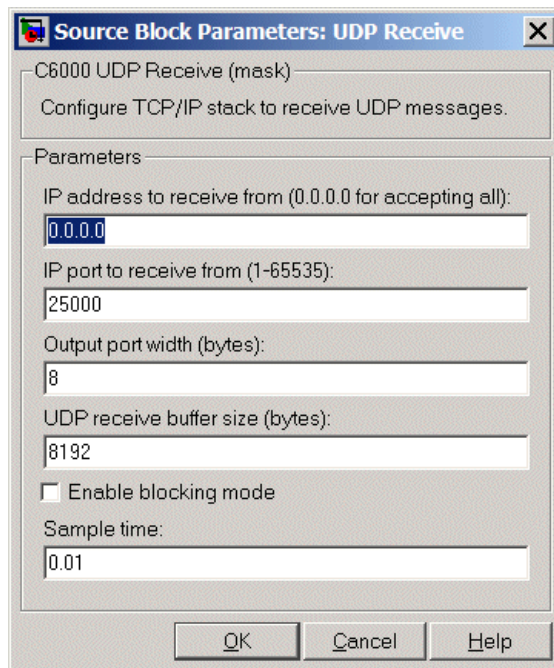
---

**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 any 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

# C6000 UDP Receive

---

from to any value other than 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 default value is 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. Entering 1 lets the block inherit the sample time from an upstream block. Setting this to a specific value, often large, can reduce the chances of UDP messages getting dropped. The default sample time is 0.01 seconds.

## **See Also**

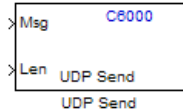
C6000 TCP/IP Receive, C6000 TCP/IP Send, C6000 UDP Send



**Purpose** Send UDP message to host

**Library** C6000 DSP Communication Library in Embedded Target for TI C6000 DSP

## 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, C6711, 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-3 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.

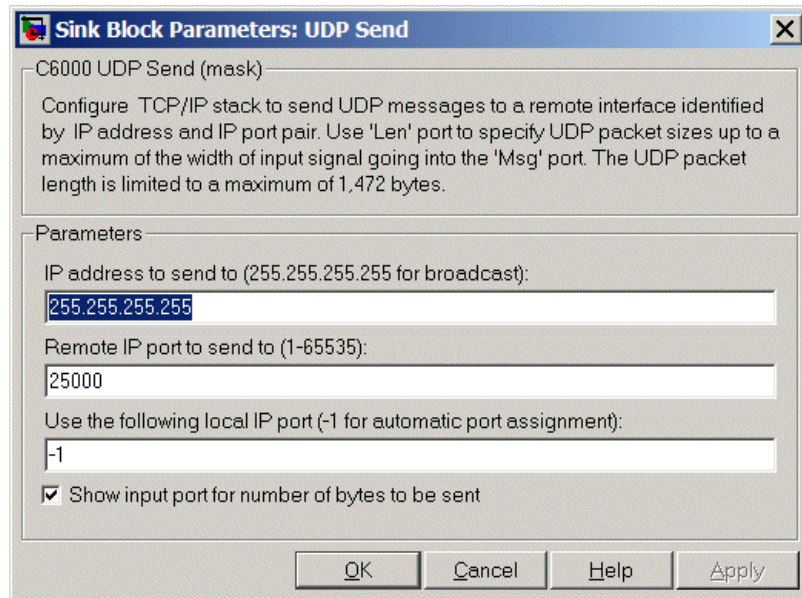
---

**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 the 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 — Math and Matrices

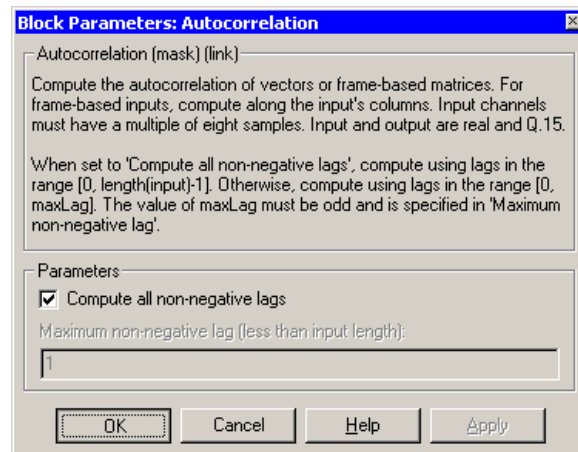
## Description



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.

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)**.

### 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  $[0, \text{maxLag}]$ . 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 — Transforms

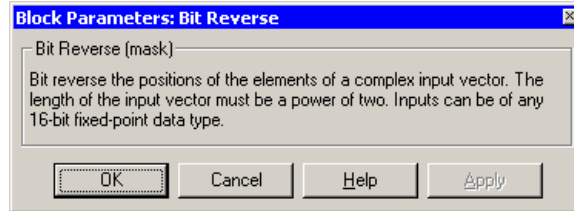
## Description



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.

The Bit Reverse block supports discrete sample times and little-endian code generation only.

## Dialog Box

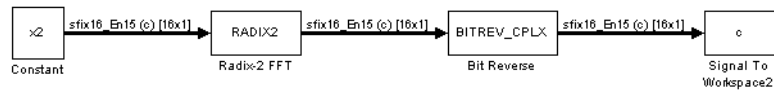


## Algorithm

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 — Math and Matrices

## Description

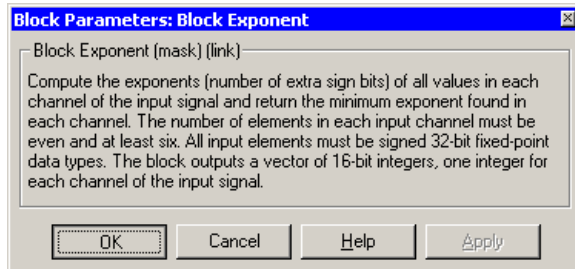


The Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be even and at least six. All input elements must be 32-bit signed fixed-point data types. The output is a vector of 16-bit integers — one integer for each channel of the input signal.

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.

The Block Exponent block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## Dialog Box



## 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 — Filtering

## Description

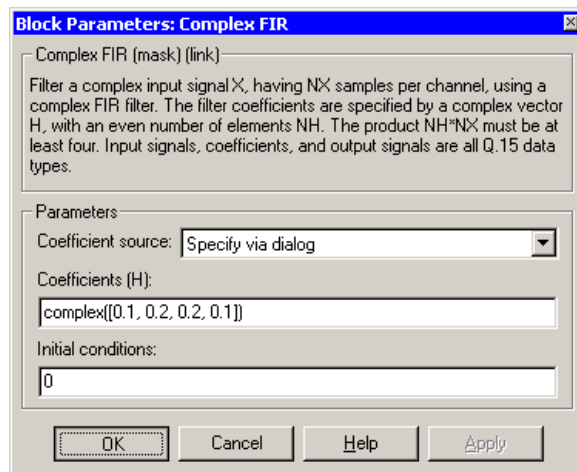


The Complex FIR block filters a complex input signal  $X$  using a complex FIR filter. This filter is implemented using a direct form structure.

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.

# C62x Complex FIR

---

## **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_cp1x`. During code generation, this block calls the `DSP_fir_cp1x` routine to produce optimized code.

## **See Also**

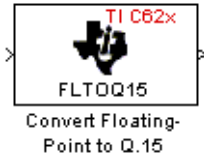
C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

# C62x Convert Floating-Point to Q.15

**Purpose** Convert single-precision floating-point input signal to Q.15 fixed-point

**Library** C62x DSP Library — Conversions

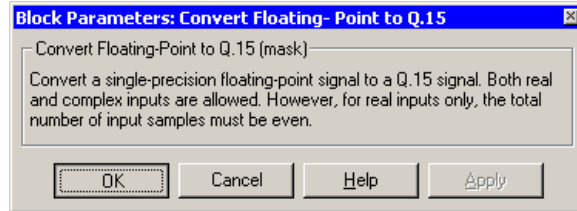
## Description



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 also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

In simulation, the Convert Floating-Point to Q.15 block is equivalent to the TMS320C62x DSP Library assembly code function DSP\_f1toq15. During code generation, this block calls the DSP\_f1toq15 routine to produce optimized code.

## See Also

C62xConvert Q.15 to Floating Point

# C62x Convert Q.15 to Floating-Point

**Purpose** Convert a Q.15 fixed-point signal to a single-precision floating-point

**Library** C62x DSP Library — Conversions

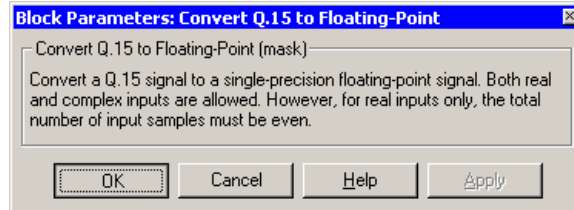
## Description



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.

The Convert Q.15 to Floating-Point block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## 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 — Transforms

### Description



The FFT block computes the decimation-in-frequency forward FFT, with interstage scaling, 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, in order 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  $(S-1)$ .

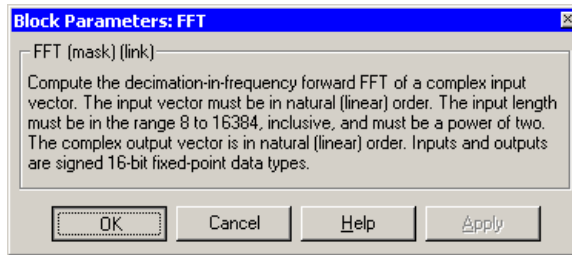
$$\text{OutputFractionalBits} = \text{InputFractionalBits} - (S - 1)$$

The FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

# C62x FFT

---

## 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 — Filtering

## Description

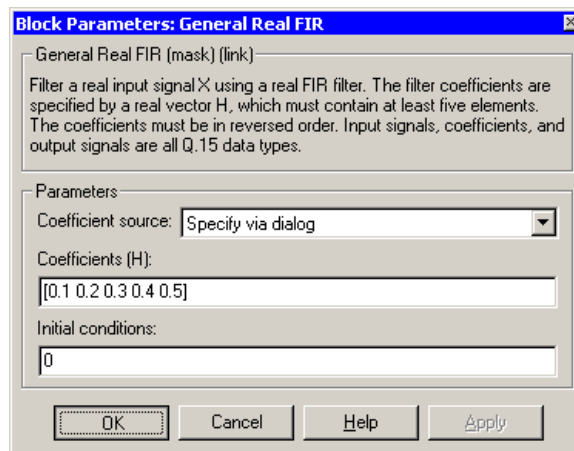


The General Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.

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 both little-endian and big-endian code generation.

## 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 — Filtering

## 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  $d - [\bar{H}(n) \cdot \bar{X}(n+1)]$ .
- $\mu$  is the step size.

For this block, the input  $B$  and the output  $R$  are defined by

$$B = \mu e(n+1)$$

$$R = \bar{H}(n) \cdot \bar{X}(n+1)$$

which combined with the first two equations, result in the following equations that this block follows:

# C62x LMS Adaptive FIR

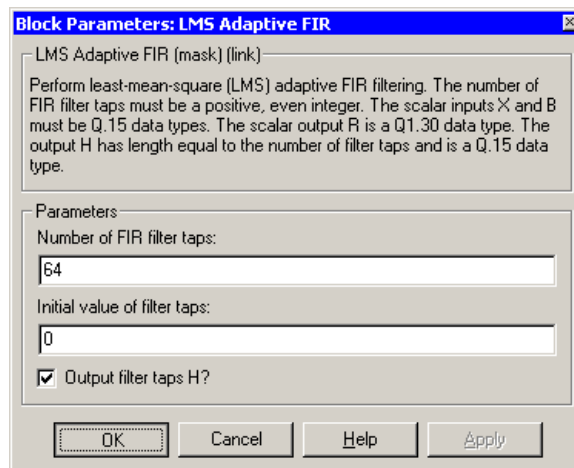
$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)]$$

$d$  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 both little-endian and big-endian code generation.

## 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?

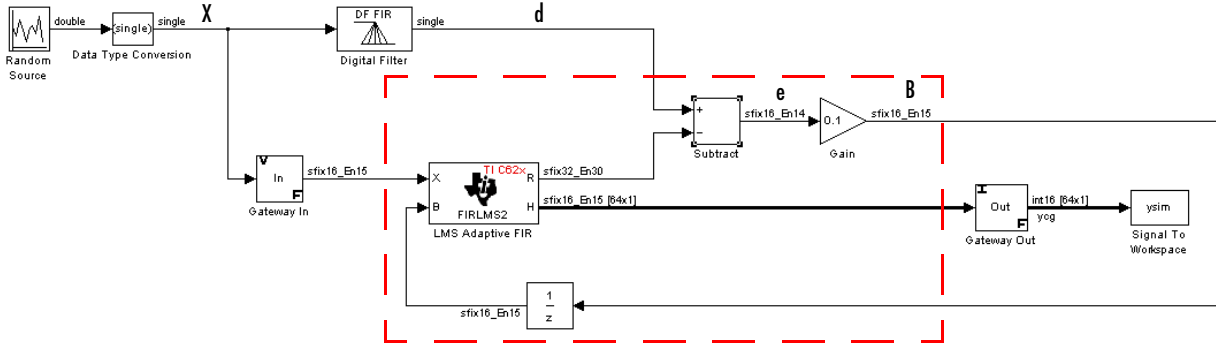
If selected, the filter taps are produced as output H. If not selected, H is suppressed.

## Algorithm

In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C62x DSP Library assembly code function DSP\_fir1ms2. During code generation, this block calls the DSP\_fir1ms2 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 the Filter Design Toolbox. `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.

# C62x Matrix Multiply

**Purpose** Matrix multiply two input signals

**Library** C62x DSP Library — Math and Matrices

## 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.

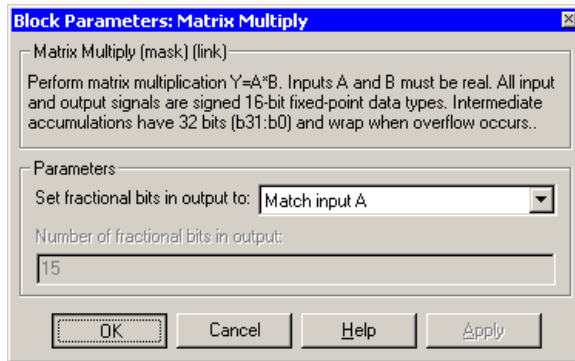
Alternatively, you can choose to output 16 bits according to how many fractional bits you want in the output. The number of fractional bits in the accumulator value is the sum of the fractional bits of the two inputs.

	<b>Input A</b>	<b>Input B</b>	<b>Accumulator Value</b>
<b>Total Bits</b>	16	16	32
<b>Fractional Bits</b>	$R$	$S$	$R + S$

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 also supports both little-endian and big-endian code generation.

## 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**.

# C62x Matrix Multiply

---

## Algorithm

In simulation, the Matrix Multiply block is equivalent to the TMS320C62x DSP Library assembly code function DSP\_mat\_mu1. During code generation, this block calls the DSP\_mat\_mu1 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.

<b>Set fractional bits in output to</b>	<b>Data Type</b>	<b>Accumulator Bits</b>
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.

<b>Set fractional bits in output to</b>	<b>Data Type</b>	<b>Accumulator Bits</b>
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 — Math and Matrices

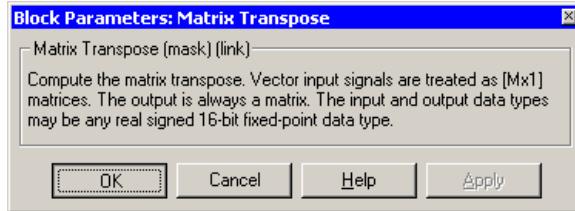
## Description



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 also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

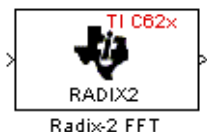
In simulation, the Matrix Transpose block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mat_trans`. During code generation, this block calls the `DSP_mat_trans` routine to produce optimized code.

# C62x Radix-2 FFT

**Purpose** Radix-2 decimation-in-frequency forward FFT of complex input vector

**Library** C62x DSP Library — Transforms

## Description

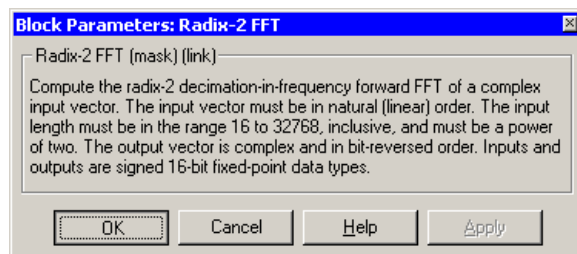


The Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is 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

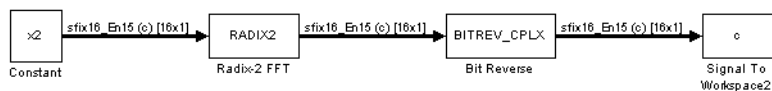


## Algorithm

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 — Transforms

## 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.

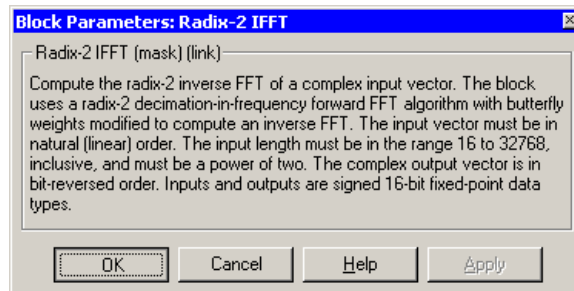
The radix2 routine used by this block employs a radix-2 FFT of length  $L=2^k$ . In order to ensure that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by  $k$  bits to the left relative to the location 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 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 — Filtering

## Description

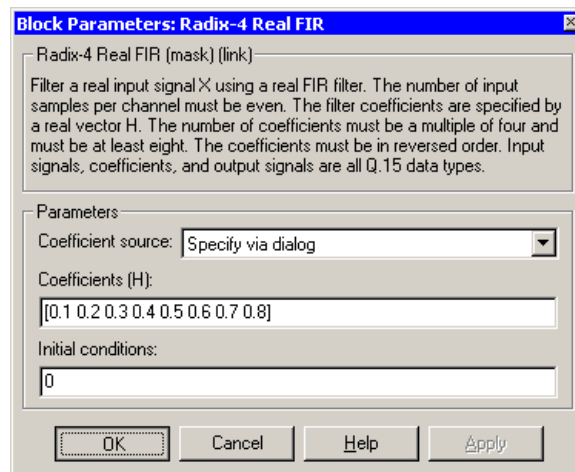


The Radix-4 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.

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.

The Radix-4 Real FIR block supports discrete sample times and both little-endian and big-endian code generation.

## 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 — Filtering

## Description

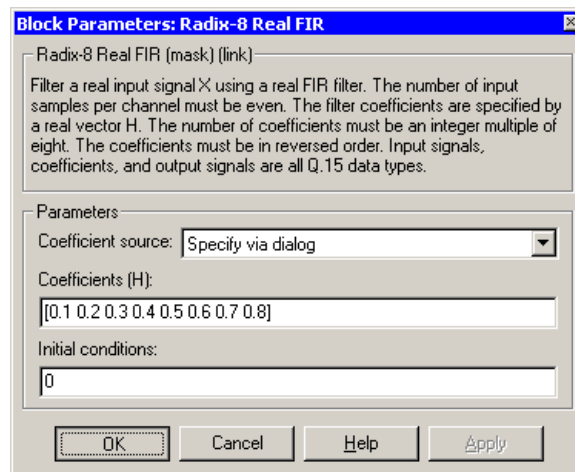


The Radix-8 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.

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 — Filtering

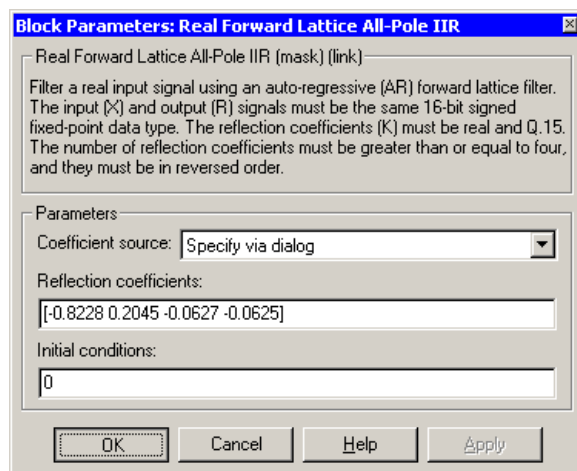
## Description



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 both little-endian and big-endian code generation.

## 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_iir1at`. During code generation, this block calls the `DSP_iir1at` routine to produce optimized code.

## See Also

C62xReal IIR

# C62x Real IIR

**Purpose** Filter real input signal using IIR filter

**Library** C62x DSP Library — Filtering

## Description

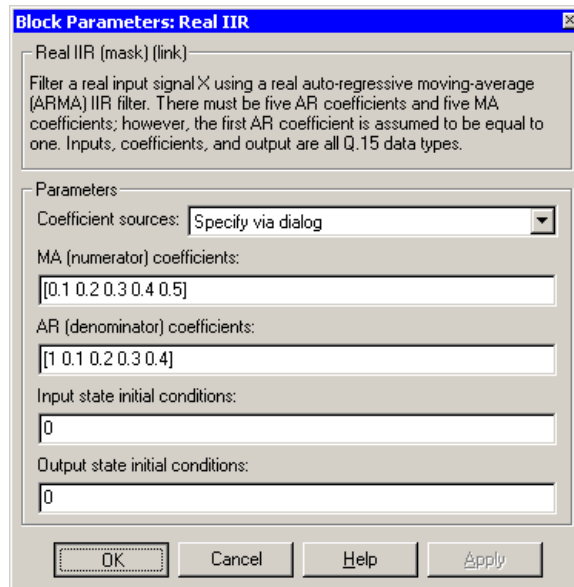


The Real IIR block filters a real input signal  $X$  using a real autoregressive moving-average (ARMA) IIR Filter. This filter is implemented using a direct form I structure.

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 both little-endian and big-endian code generation.

## 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
- 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.

# C62x Real IIR

---

- 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 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 — Math and Matrices

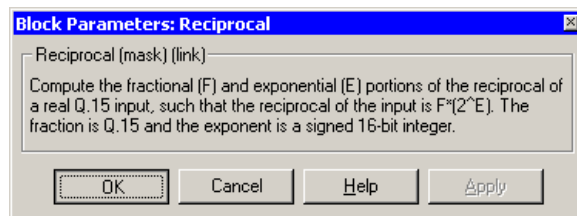
## Description



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 both little-endian and big-endian code generation.

## 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 — Filtering

## 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 <b>Number of fractional bits in output</b> 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

### 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

# C62x Symmetric Real FIR

---

## 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 5-66 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 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

# C62x Vector Dot Product

**Purpose** Vector dot product of real input signals

**Library** C62x DSP Library — Math and Matrices

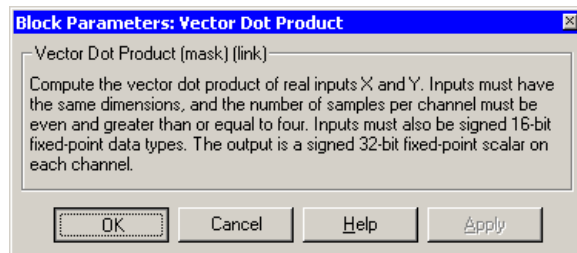
## Description



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 also supports both little-endian and big-endian code generation.

## 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.

**Purpose** Zero-based index of maximum value element in each input signal channel

**Library** C62x DSP Library — Math and Matrices

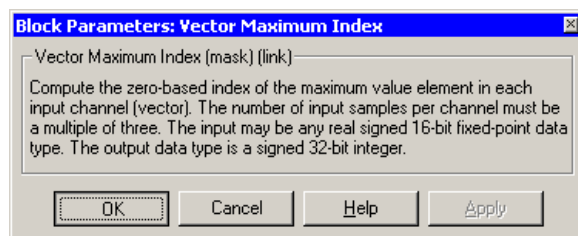
## Description



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.

The Vector Maximum Index block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

In simulation, the Vector Maximum Index block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_maxidx`. During code generation, this block calls the `DSP_maxidx` routine to produce optimized code.

# C62x Vector Maximum Value

**Purpose** Maximum value for each input signal channel

**Library** C62x DSP Library — Math and Matrices

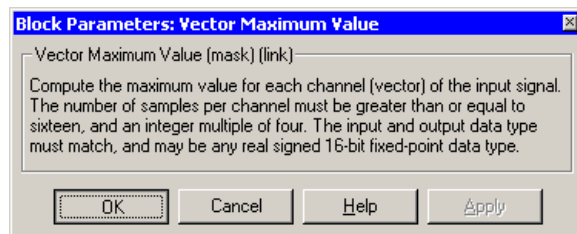
## Description



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 also supports both little-endian and big-endian code generation.

## 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 — Math and Matrices

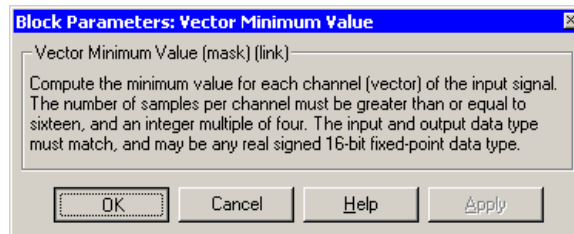
## Description



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 also supports both little-endian and big-endian code generation.

## 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

# C62x Vector Multiply

**Purpose** Element-wise multiplication on inputs

**Library** C62x DSP Library — Math and Matrices

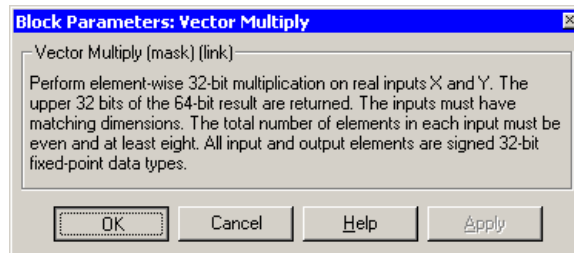
## Description



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 also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

In simulation, the Vector Multiply block is equivalent to the TMS320C62x DSP Library assembly code function DSP\_mu132. During code generation, this block calls the DSP\_mu132 routine to produce optimized code.

**See Also** C62xMatrix Multiply

**Purpose** Negate each input signal element

**Library** C62x DSP Library — Math and Matrices

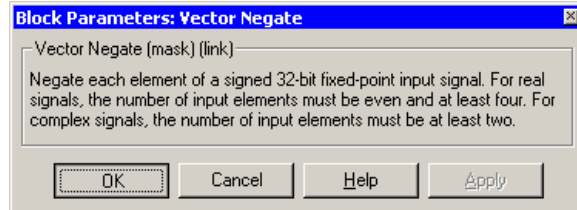
## Description



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 also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

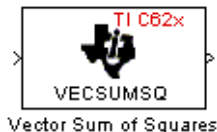
In simulation, the Vector Negate block is equivalent to the TMS320C62x DSP Library assembly code function DSP\_neg32. During code generation, this block calls the DSP\_neg32 routine to produce optimized code.

# C62x Vector Sum of Squares

**Purpose** Sum of squares over each real input channel

**Library** C62x DSP Library — Math and Matrices

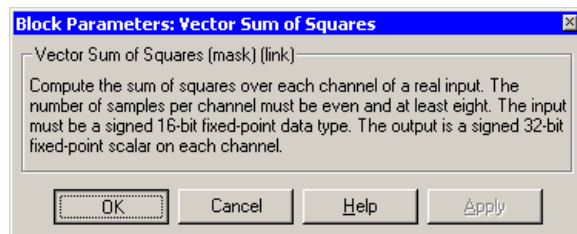
## Description



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 also supports both little-endian and big-endian code generation.

## 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.



**Purpose** Weighted sum of input vectors

**Library** C62x DSP Library — Math and Matrices

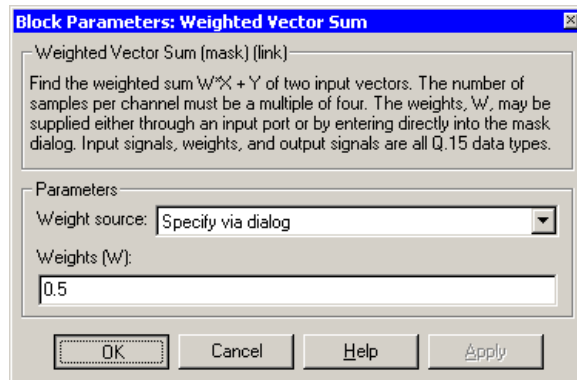
## Description



The Weighted Vector Sum block computes the weighted sum of two inputs, X and Y, according to  $(W * X) + Y$ . 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  $-1 < W < 1$ .

The Weighted Vector Sum block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## 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

# C62x 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

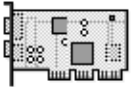
In simulation, the Weighted Vector Sum block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

**Purpose**

Configure model for C6416 DSP Starter Kit

**Library**

Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

**Description**

C6416DSK

Options on the block mask let you set features of code generation for your C6416 DSP Starter Kit target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6416 DSK must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

---

**Note** This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

---

The processor and target options you specify on this block are:

- Target board information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, DSP/BIOS, and custom sections.

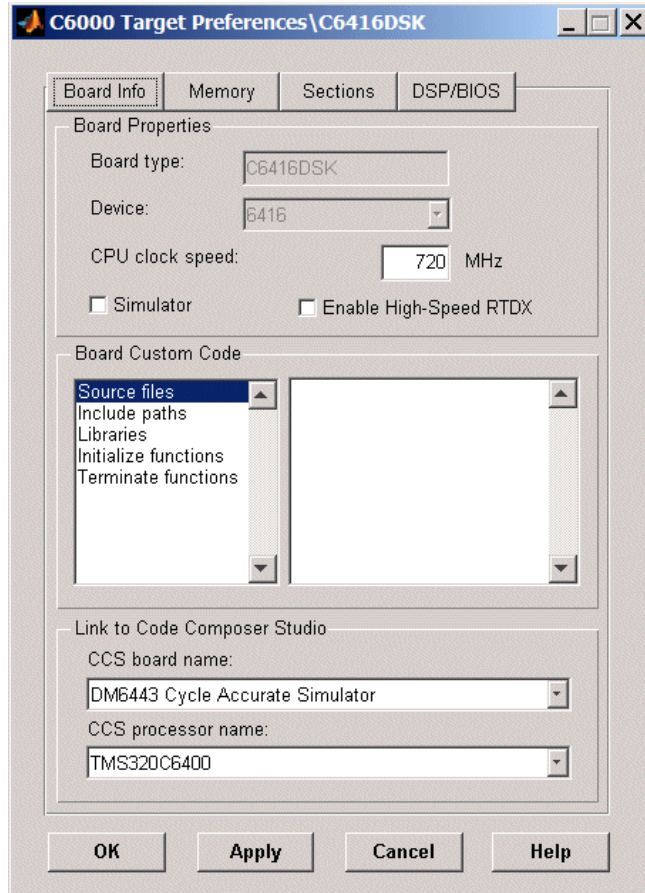
Setting the options included in this dialogbox results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both steps are essential for targeting any board that is custom or explicitly supported, such as the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

## Generating Code from Model Subsystems

Real-Time Workshop provides the ability to generate code from a selected subsystem in a model. To generate code for the C6416 DSK from a subsystem, the subsystem model must include a C6416DSK target preferences block.

### Dialog Box



All target preferences block dialog boxes provide tabbed access to panes the following panes with options you set for the target processor and target board:

- **Board info** — Select the target board and processor, set the clock speed, and identify the target.

- **Memory** — Set the memory allocation and layout on the target processor (memory mapping).
- **Sections** — Determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS and compiler information.
- **DSP/BIOS** — Specify how to configure tasking features of DSP/BIOS.

## Board Info Pane

The following options appear on the **Board Info** pane for the **C6000 Target Preferences** dialog.

### Board Type

Lets you enter the type of board you are targeting with the model. You can enter **Custom** to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as **C6711DSK**. If you are using one of the explicitly supported boards, choose the target preferences block for that board and this field shows the proper board type.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model's real-time results may be wrong, and code profiling results are not correct.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if either of the following conditions occur:

- If your model does not include ADC or DAC blocks

- When the processing rates in your model change (the model is multirate)

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. You can change the rate with the DIP switches on the board or from one of the software utilities provided by Texas Instruments.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 1,000,000 \text{ clock ticks}$$

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

## **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

## **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for this model.

## **Board Custom Code**

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths.

When you enter a path to a file, library, or other custom code, use the string

```
$(install_dir)
```

to refer to the CCS installation directory.

Enter new paths or files (custom code items) one to a line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

- **Source files** — you enter the full paths to source code files to use with this target. By default there are no entries in this parameter.
- **Include paths** — If you require additional files on your path, you add them by typing the path into the text area. The default setting does not include additional paths.
- **Libraries** — these entries identify specific libraries that the target requires. They appear on the list by default if required. Add more as you require by entering the full path to the library with the library file in the text area. No additional libraries appear here in the default configuration.
- **Initialize functions** — If your project requires an initialize function, enter it here. By default, this is empty.
- **Terminate functions** — enter a function to run when a program terminates. The default setting is not to include a specific termination function.

### **CCS Board Name**

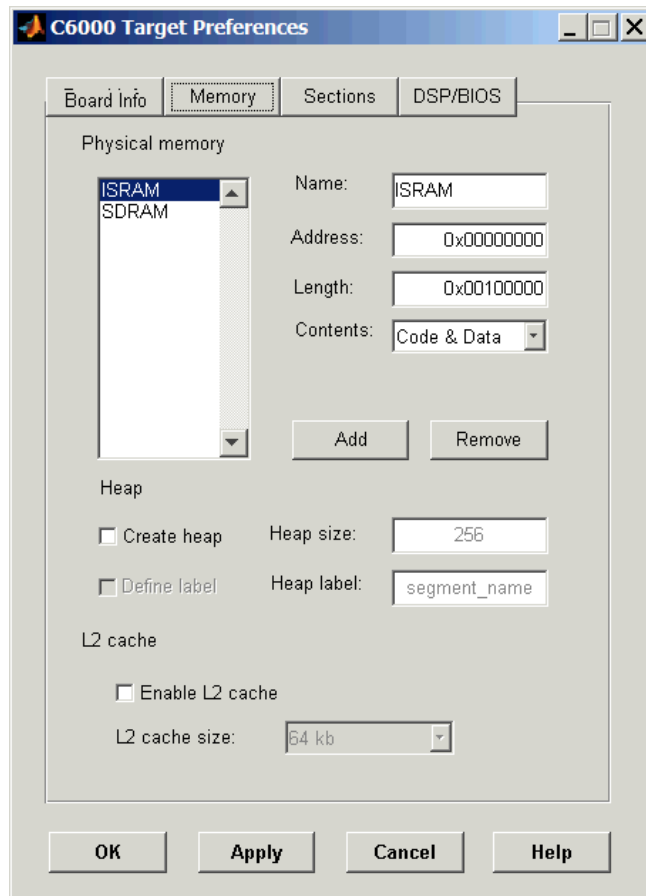
Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one that you are targeting your code for.

### **CCS Processor Name**

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.

### **Memory Pane**

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



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

- **Physical Memory** — specifies the processor and board memory map
- **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 may 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 online help system for Code Composer Studio.

## 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.
- C6711DSK boards provide SDRAM memory segments 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** details 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 may 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 may 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 on the **Physical memory** list and click **Remove** to delete the segment.

## **Create Heap**

If your processor supports using a heap, as does the C6711, 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**

Enabled by selecting **Define label**, you use this option 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.

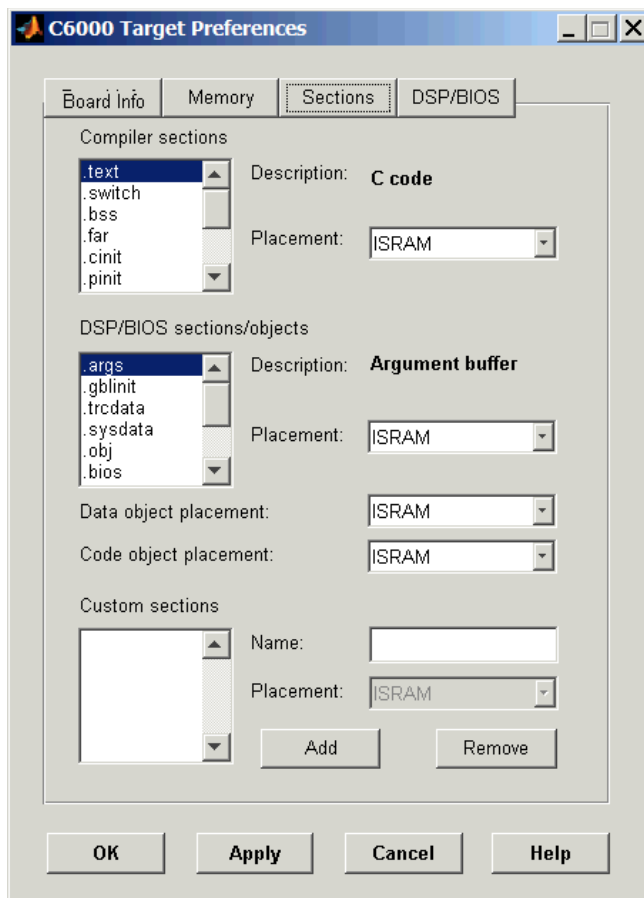
### **L2 Cache size**

When you enable the L2 cache, use this list to determine the size of the cache allotted. 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. Commonly used sections include `.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 online help.



Within this pane, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections**, **DSP/BIOS sections/objects**, and **Custom sections** lists in the pane.

All sections do not appear on all lists. The list the string appears on is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.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 get 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 the .data 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 presently allocated in memory.

### **Description**

Provides a brief explanation of the contents of the selected entry on 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, get 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 location 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. You can also click **OK** to add the section to the list and close the dialog.

### Remove

To remove a section from the **Custom sections** list, select the section and click **Remove**.

## DSP/BIOS Pane

Options on this pane let you specify how to configure tasking features of DSP/BIOS.

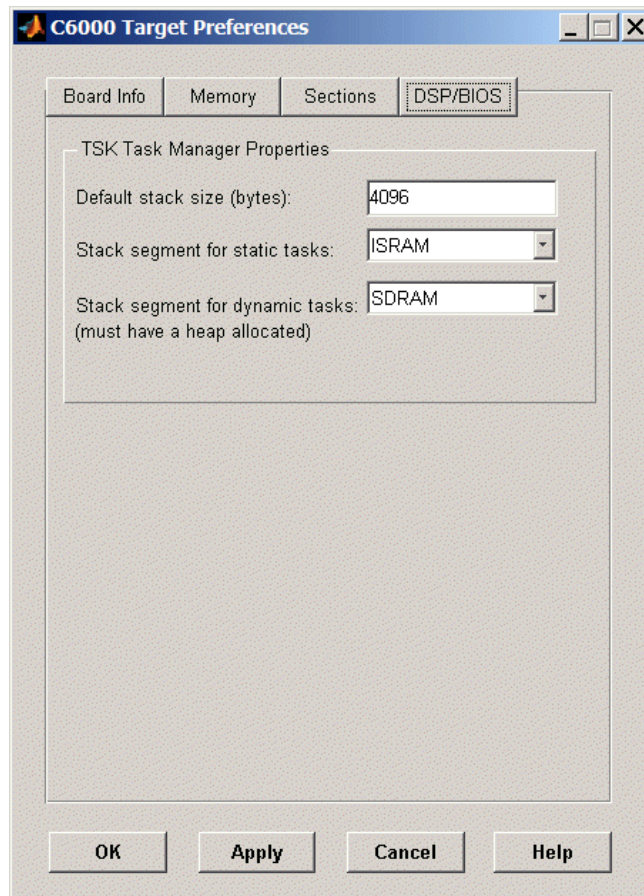
The asynchronous task scheduler uses these options when you select the **Incorporate DSP/BIOS** option in the model configuration set. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their block dialogs so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you clear the **Incorporate DSP/BIOS** option, you disable the options in this pane. Your project does not include DSP/BIOS tasks, and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler.

For more information about tasks, refer to the Code Composer Studio online help.



Within this pane, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. Note that the Sections pane includes DSP/BIOS configuration options as well. The options specify the stack use and locations on the stack for static and dynamic tasks.

#### **Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the DSP/BIOS stack in bytes allocated for each task. 4096 bytes is the default value. You can set any size up to the limits for the processor. Set the stack

size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, this might cause the task to write into other memory or data areas, possibly causing unpredictable behavior.

### **Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers options SDRAM and ISRAM for locating the stack in memory, with SDRAM as the default section. The Memory pane provide more options for the physical memory on the processor.

### **Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, SDRAM is the only valid stack location in memory.

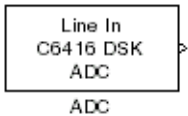
### **See Also**

Custom C6000

**Purpose** Digitized output from codec to processor

**Library** C6416 DSK Board Support in Embedded Target for TI C6000 DSP

## 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, 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
<b>ADC Source</b>	Codec
<b>Mic</b>	Codec
<b>Output data type</b>	TMS320C6416 digital signal processor
<b>Samples per frame</b>	Direct memory access module
<b>Sample Rate</b>	Codec
<b>Scaling</b>	TMS320C6416 digital signal processor
<b>Word Length</b>	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

## C6416 DSK ADC

---

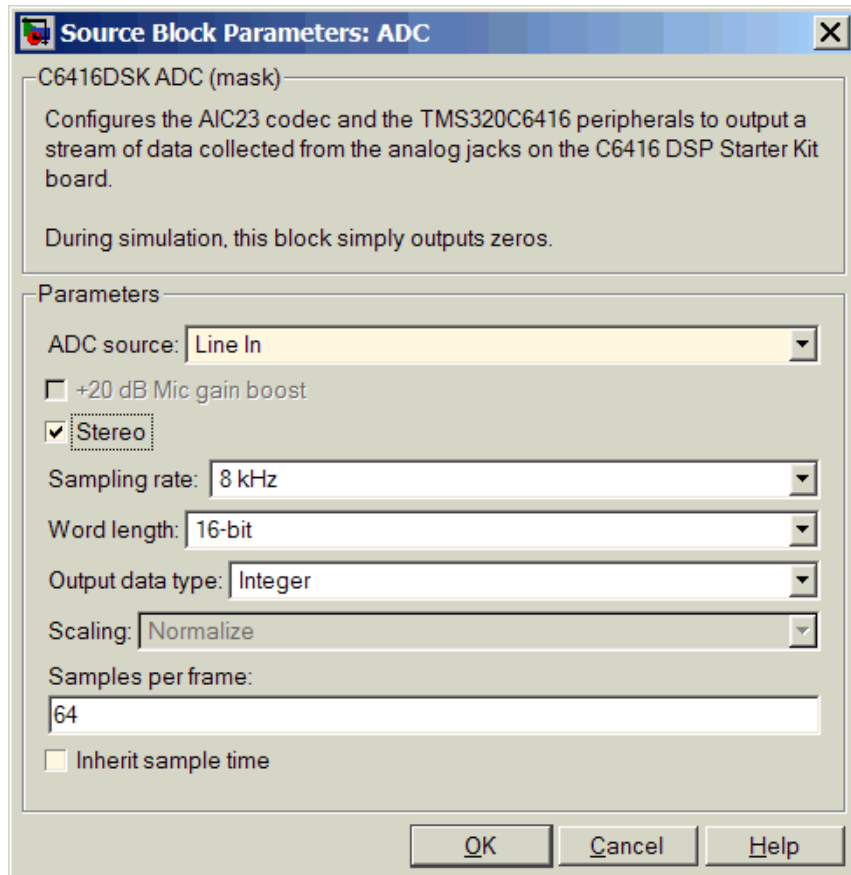
box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels on input and output.

The block uses frame-based processing of inputs, buffering the input data into frames at the specified samples per frame rate. In Simulink, 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 length of each data word output from the codec, since the input is analog. You choose from 16-, 20-, 24-, and 32-bit options.

## **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  $\pm 1.0$ , 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. 64 samples per frame is the default setting. 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/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

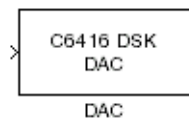
# C6416 DSK DAC

---

**Purpose** Use codec to convert digital input to analog output

**Library** C6416 DSK Board Support in Embedded Target for TI C6000 DSP

## Description

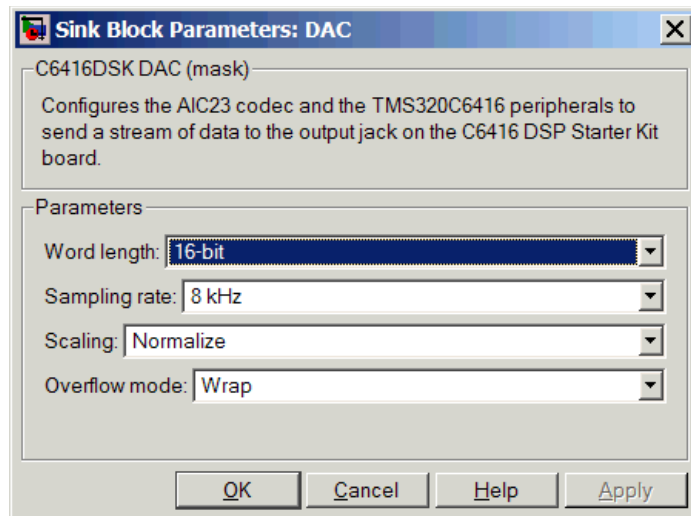


Adding the C6416 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the LINE OUT connection on the C6416 DSK board. When you add the C6416 DSK DAC block, the digital signal received by the codec is converted to an analog signal. After converting the digital signal to analog form (digital-to-analog (D/A) conversion), the codec sends the signal to the output jack.

One of the configuration options in the block affects the codec. The remaining options relate to the model you are using in Simulink and the signal processor on the board. In the following table, you find each option listed with the C6416 DSK hardware affected by your selection.

<b>Option</b>	<b>Affected Hardware</b>
<b>Overflow mode</b>	TMS320C6416 Digital Signal Processor
<b>Scaling</b>	TMS320C6416 Digital Signal Processor
<b>Word Length</b>	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 default value is 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  $\pm 1.0$ . 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.

## See Also

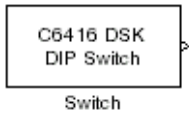
C6416 DSK ADC

# C6416 DSK DIP Switch

**Purpose** Simulate or read DIP switches

**Library** C6416 DSK Board Support in Embedded Target for TI C6000 DSP

**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
Selected	Selected	Selected	Cleared	0111	7
Cleared	Cleared	Cleared	Selected	1000	8
Selected	Cleared	Cleared	Selected	1001	9

# 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
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 table below shows.

## Output Values From The User DIP Switches on the C6416 DSK

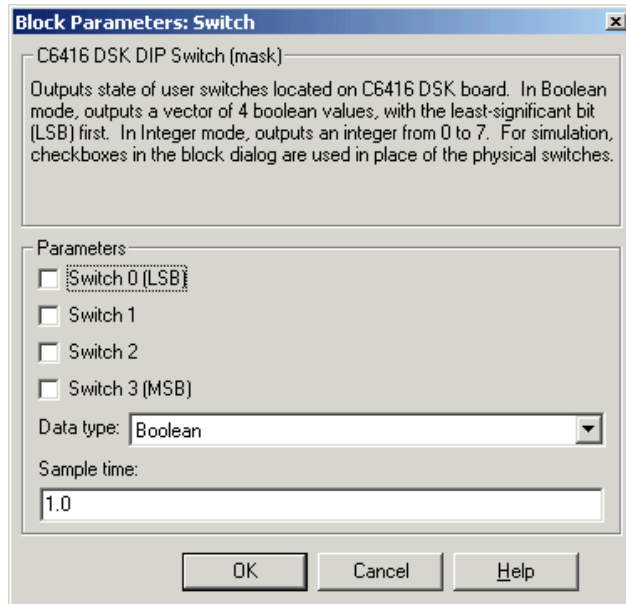
Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2

# C6416 DSK DIP Switch

**Output Values From The User DIP Switches on the C6416 DSK (Continued)**

<b>Switch 0 (LSB)</b>	<b>Switch 1</b>	<b>Switch 2</b>	<b>Switch 3 (MSB)</b>	<b>Boolean Output</b>	<b>Integer Output</b>
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 causes a running simulation to pause. Refer to “Changing Source Block Parameters” 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.

# C6416 DSK DIP Switch

---

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**

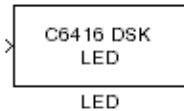
Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).



**Purpose** Control LEDs

**Library** C6416 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



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.

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:

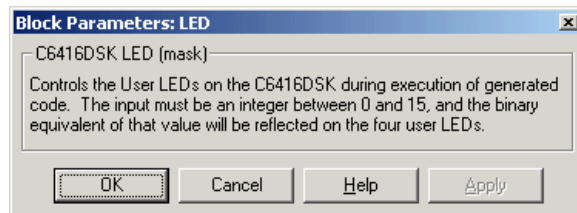
- When the block receives an input value equal to 0, the specified LEDs are turned off (disabled), 0000
- When the block receives a nonzero input value, the specified LEDs are turned on (enabled), 0001 to 1111

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.

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.

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.

## Dialog Box



This dialog does not have any user-selectable options.

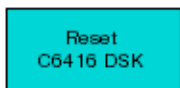
# C6416 DSK RESET

---

**Purpose** Reset to initial conditions

**Library** C6416 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



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 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.

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.

**Dialog Box** This block does not have settable options and does not provide a user interface dialog.

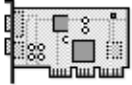
## Purpose

Configure model for C6455 DSP Starter Kit

## Library

Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

## Description



C6455DSK

Options on the block mask let you set features of code generation for your C6455 DSP Starter Kit target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6455 DSK must include this block or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

---

**Note** This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

---

The processor and target options you specify on this block are:

- Target board information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, DSP/BIOS, and custom sections.

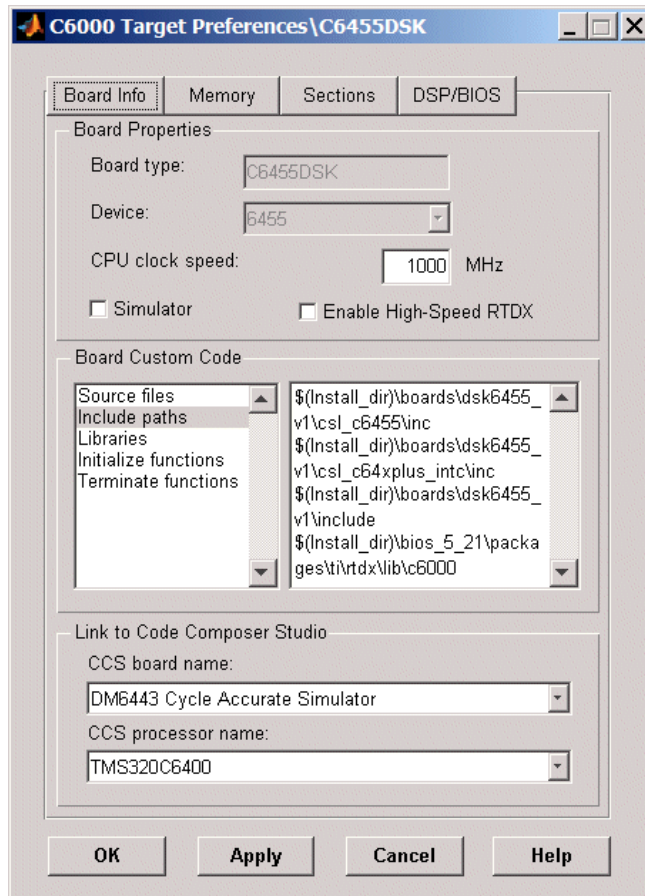
Setting the options included in this dialogbox results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both steps are essential for targeting any board that is custom or explicitly supported, such as the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

## Generating Code from Model Subsystems

Real-Time Workshop provides the ability to generate code from a selected subsystem in a model. To generate code for the C6455 DSK from a subsystem, the subsystem model must include a C6455DSK target preferences block.

### Dialog Box



All target preferences block dialog boxes provide tabbed access to panes the following panes with options you set for the target processor and target board:

- **Board info** — Select the target board and processor, set the clock speed, and identify the target.

- **Memory** — Set the memory allocation and layout on the target processor (memory mapping).
- **Sections** — Determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS and compiler information.
- **DSP/BIOS** — Specify how to configure tasking features of DSP/BIOS.

## Board Info Pane

The following options appear on the **Board Info** pane for the **C6000 Target Preferences** dialog box.

### Board Type

Lets you enter the type of board you are targeting with the model. You can enter **Custom** to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as **C6711DSK**. If you are using one of the explicitly supported boards, choose the target preferences block for that board and this field shows the proper board type.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model's real-time results may be wrong, and code profiling results are not correct.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if either of the following conditions occur:

- If your model does not include ADC or DAC blocks
- When the processing rates in your model change (the model is multirate)

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. You can change the rate with the DIP switches on the board or from one of the software utilities provided by Texas Instruments.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 1,000,000 \text{ clock ticks}$$

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

## **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

## **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for this model.

## **Board Custom Code**

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths.

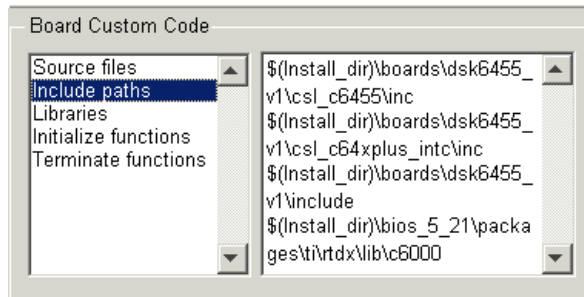
When you enter a path to a file, library, or other custom code, use the string

`$(install_dir)`

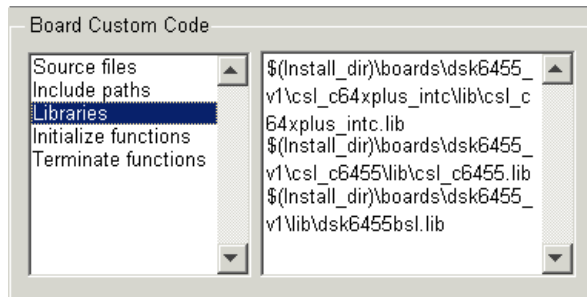
to refer to the CCS installation directory. The examples in the following figure use the string.

Enter new paths or files (custom code items) one to a line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

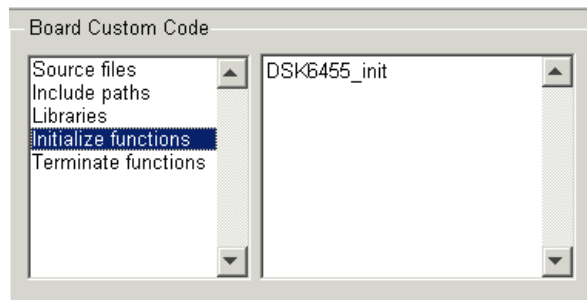
- **Source files** — Enter the full paths to source code files to use with this target. The default is blank.
- **Include paths** — C6455 DSK requires some additional files to work correctly. When you add this block to your model, the default include paths appear as shown in the following figure. These entries include chip support libraries, a BIOS addition, and an RTDX library. All are necessary for use. You can add further paths by typing the path into the text area.



- **Libraries** — These entries identify specific libraries that the target requires. They appear on the list by default, as shown on the following figure.



- **Initialize functions** — C6455 DSK targets require a specific initialization function, listed here as `DSK6455_init`. Enter others if needed.



- **Terminate functions** — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

## CCS Board Name

Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one for which you are targeting your code.

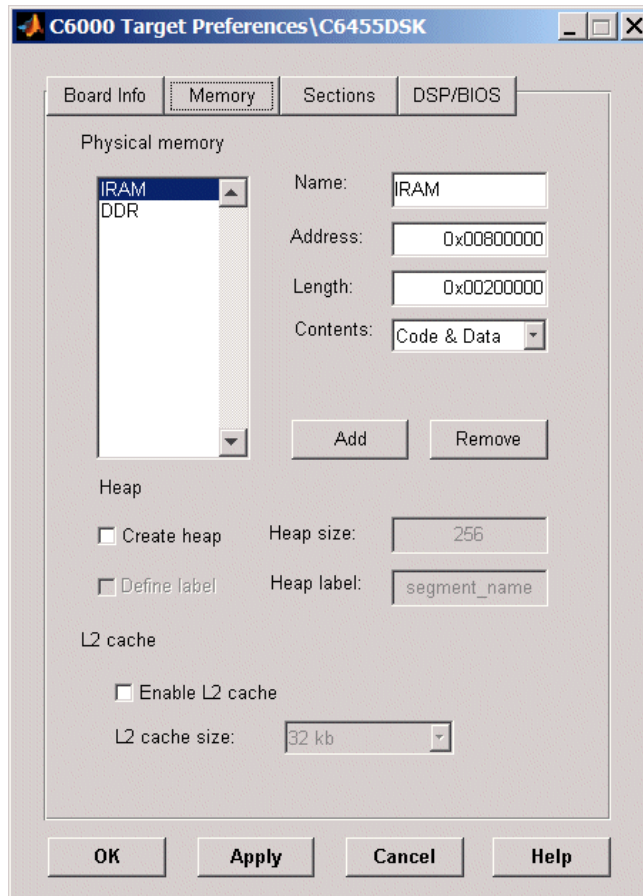
## CCS Processor Name

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.



## Memory Pane

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



The **Memory** pane contains memory options in three areas as shown in the preceding figure:

- **Physical Memory** — Specifies the processor and board memory map

- **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 may 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 online help system for Code Composer Studio.

## 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.
- C6711DSK boards provide SDRAM memory segments by default.

### Name

When you highlight an entry on the **Physical memory** list, the name of the entry appears in this field. 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).

---

**Note** 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** details 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 may 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 may 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. Enter the new name or click **Apply** to update the temporary name on the list to the name you want.

## **Remove**

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

## **Create Heap**

If your processor supports using a heap, as does the C6711, for example, selecting this option allows you to create 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.

---

**Note** 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** allows you to name the heap. Enter your label for the heap in **Heap Label**.

## **Heap Label**

You enable this option by selecting **Define label**. Use this option 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.

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

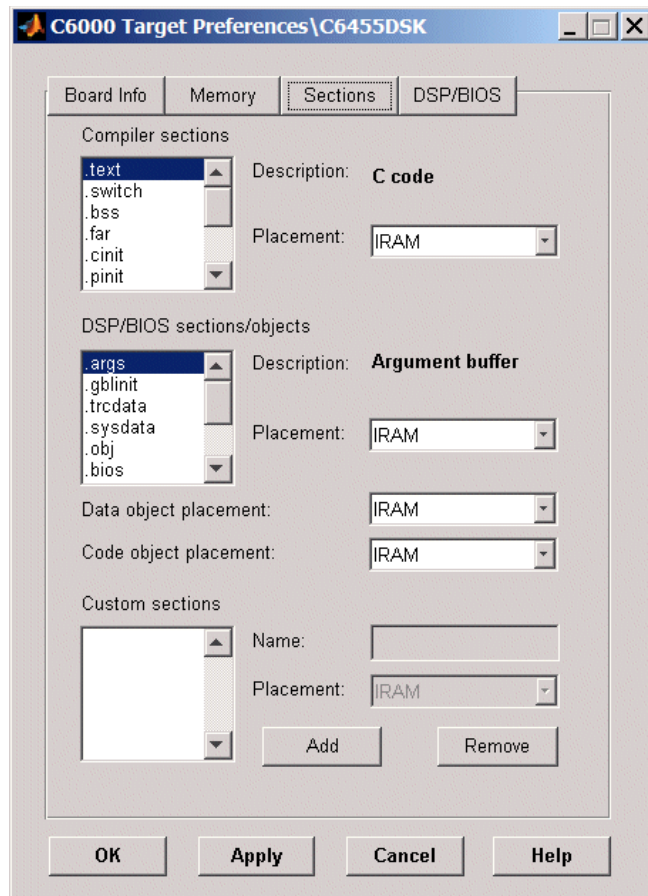
## **L2 Cache size**

When you enable the L2 cache, use this list to determine the size of the cache allotted. 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. Commonly used sections include `.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 online help.



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

This table provides brief definitions of the kinds of sections in the **Compiler sections**, **DSP/BIOS sections/objects**, and **Custom sections** lists in the pane.

All sections do not appear on all lists. The list the string appears on is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.system	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 get allocated into memory as required by the configuration of your system. On the **Compiler sections** list, you find both initialized sections (sections that contain data or executable code) and uninitialized sections (sections that reserve space in memory). The initialized sections are:

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

These sections are uninitialized:

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



Other sections appear on the list as well:

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

---

**Note** The C/C++ compiler does not use the .data 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 presently allocated in memory.

### **Description**

Provides a brief explanation of the contents of the selected entry on 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

These objects are distinct from the entries on the **DSP/BIOS sections** list. DSP/BIOS objects such as STS or LOG 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 location 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, but instead, a placeholder for a section for you to define.

### Name

You enter the name for your new section in this field. To add a new section, click **Add**. Then, replace the temporary name with the name you want 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 a 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. You can also 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 and click **Remove**.

## DSP/BIOS Pane

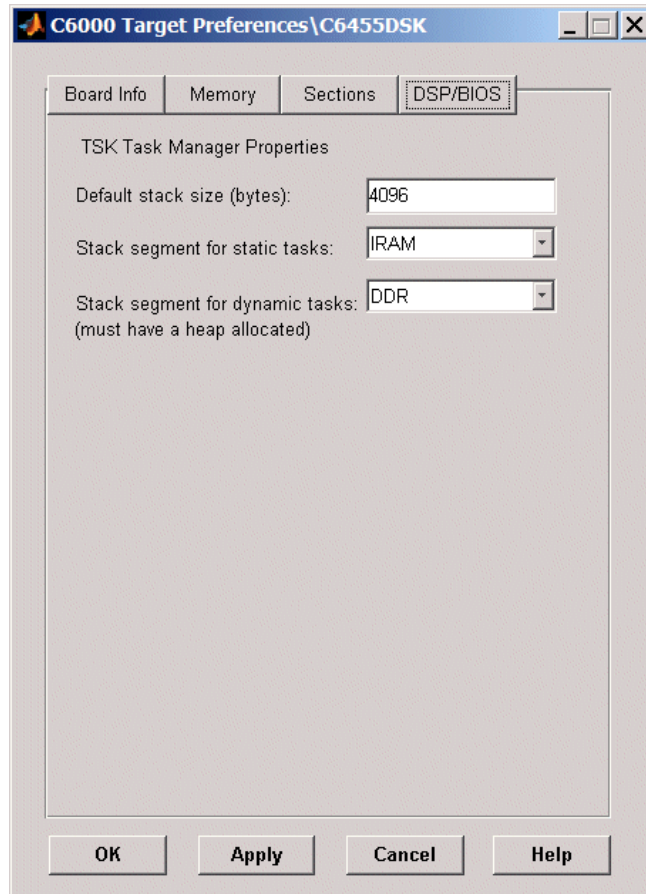
Options on this pane let you specify how to configure tasking features of DSP/BIOS.

The asynchronous task scheduler uses these options when you select the **Incorporate DSP/BIOS** option in the model configuration set. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their dialog boxes so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you clear the **Incorporate DSP/BIOS** option, you disable the options in this pane. Your project does not include DSP/BIOS tasks, and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler.



In the pane shown in this figure, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. The Sections pane includes DSP/BIOS configuration options as well. The options specify the stack use and locations on the stack for static and dynamic tasks.

### **Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the DSP/BIOS stack in bytes allocated for each task. A value of 4096 bytes is the default. You can set any size up to the limits for the processor. Set the

stack size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, failure to set the stack size might cause the task to write into other memory or data areas, possibly causing unpredictable behavior.

**Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. However, infrequently used tasks usually work best as dynamic tasks.

The list offers options SDRAM and ISRAM for locating the stack in memory, with SDRAM as the default section. The Memory pane provide more options for the physical memory on the processor.

**Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, SDRAM is the only valid stack location in memory.

**See Also**

Custom C6000

# C64x Autocorrelation

**Purpose** Autocorrelate input vector or frame-based matrix

**Library** C64x DSP Library — Math and Matrices

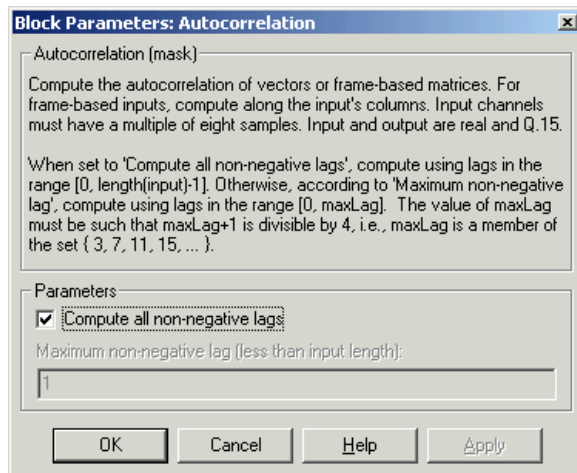
## Description



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)**.

### 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  $[0, \text{maxLag}]$ . The maximum

lag must be odd, and  $(\text{maxLag}+1)$  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.

# C64x Bit Reverse

**Purpose** Bit-reverse elements of each complex input signal channel

**Library** C64x DSP Library — Transforms

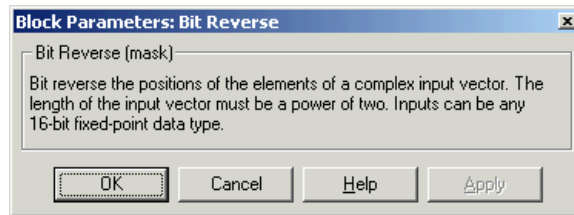
## Description



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.

The Bit Reverse block supports discrete sample times and little-endian code generation only.

## Dialog Box

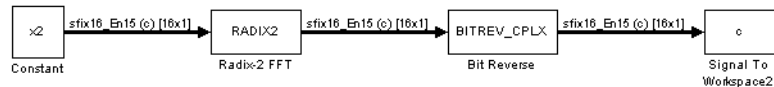


## Algorithm

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

The Bit Reverse block reorders the output of the C64x Radix-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

C64x Radix-2 FFT, C64x Radix-2 IFFT

# C64x Block Exponent

**Purpose** Minimum number of extra sign bits) in each input channel

**Library** C64x DSP Library — Math and Matrices

## Description

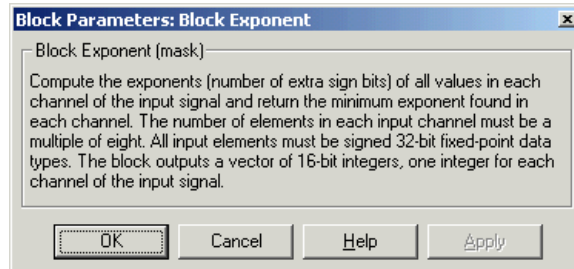


The C64x Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be a multiple of eight. Input elements must be 32-bit 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 also supports both little-endian and big-endian code generation.

## 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.

**Purpose** Filter complex input signal using complex FIR filter

**Library** C64x DSP Library — Filtering

## Description

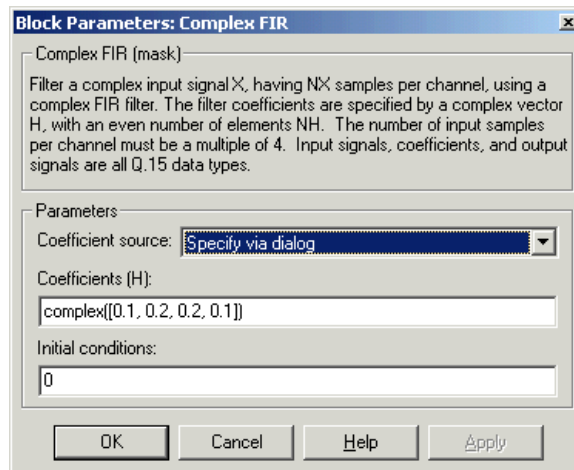


The C64x Complex FIR block filters a complex input signal  $X$  using a complex FIR filter. This filter is implemented using a direct form structure. Each input channel must contain an integer multiple of four samples, with four samples as the minimum required.

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

# C64x Complex FIR

---

- **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**

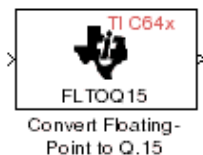
C64x General Real FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

# C64x Convert Floating-Point to Q.15

**Purpose** Convert floating-point signal to Q.15 fixed-point

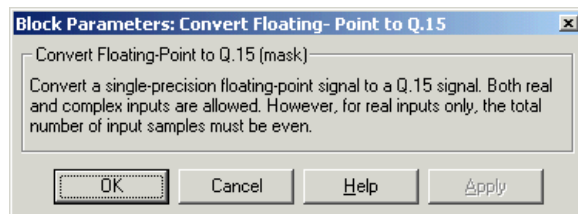
**Library** C64x DSP Library — Conversions

**Description** 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.



The Convert Floating-Point to Q.15 block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## Dialog Box



**Algorithm** In simulation, the Convert Floating-Point to Q.15 block is equivalent to the TMS320C64x DSP Library assembly code function DSP\_f1toq15. During code generation, this block calls the DSP\_f1toq15 routine to produce optimized code.

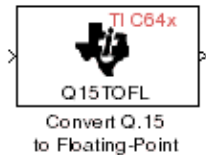
**See Also** C64x Convert Q.15 to Floating Point

# C64x Convert Q.15 to Floating-Point

**Purpose** Convert Q.15 fixed-point signal to single-precision floating-point

**Library** C64x DSP Library — Conversions

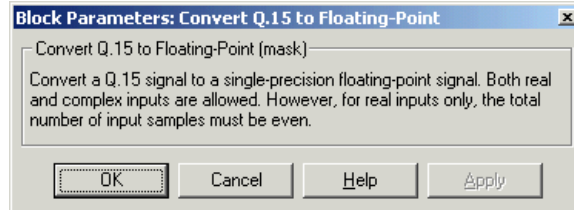
## Description



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 also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

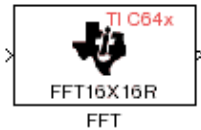
In simulation, the Convert Q.15 to Floating-Point block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_q15tof1`. During code generation, this block calls the `DSP_q15tof1` routine to produce optimized code.

**See Also** C64x Convert Floating-Point to Q.15

**Purpose** Decimation-in-frequency forward FFT of complex input vector

**Library** C64x DSP Library — Transforms

**Description**



The C64x FFT block computes the decimation-in-frequency forward FFT, with interstage scaling, 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, in order 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  $(S-1)$ .

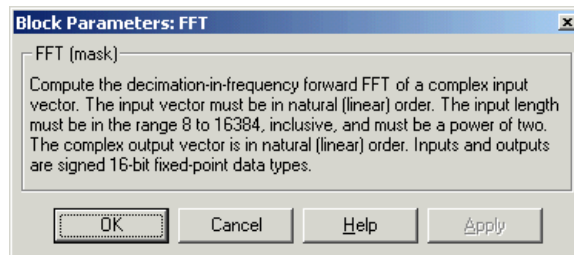
$$\text{OutputFractionalBits} = \text{InputFractionalBits} - (S - 1)$$

The FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

# C64x FFT

---

## 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



**Purpose** Filter real input signal using real FIR filter

**Library** C64x DSP Library — Filtering

## Description

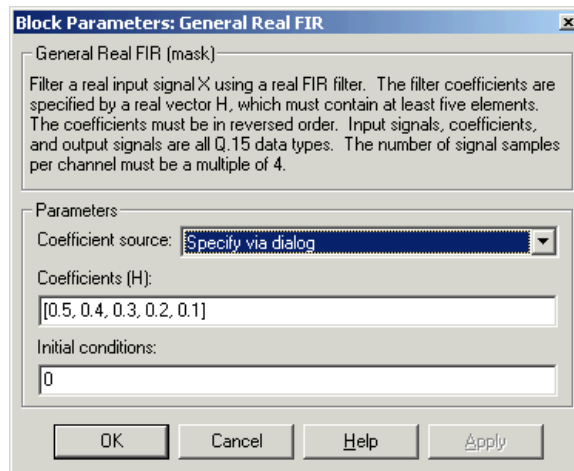


The C64x General Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure. Signal  $X$  must contain at least four samples per channel and the number of samples must be an integer multiple of four.

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 both little-endian and big-endian code generation.

## 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

# C64x 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 TMS320C64x DSP Library assembly code function `DSP_fir_gen`. During code generation, this block calls the `DSP_fir_gen` routine to produce optimized code.

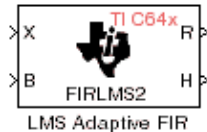
## **See Also**

C64x Complex FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

**Purpose** LMS adaptive FIR filtering

**Library** C64x DSP Library — Filtering

## 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  $d - [\bar{H}(n) \cdot \bar{X}(n+1)]$ .
- $\mu$  is the step size.

For this block, the input  $B$  and the output  $R$  are defined by

$$B = \mu e(n+1)$$

$$R = \bar{H}(n) \cdot \bar{X}(n+1)$$

# C64x LMS Adaptive FIR

which combined with the first two equations, result in the following equations that this block follows:

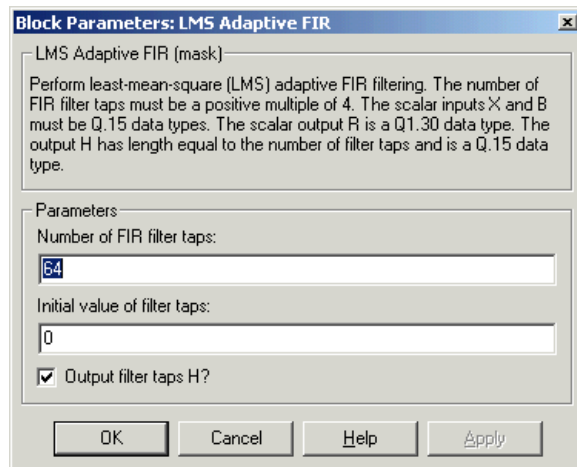
$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)]$$

$d$  and  $B$  must be produced externally to the LMS Adaptive FIR block. See “Examples” below for a sample model where this is done.

The LMS Adaptive FIR block supports discrete sample times and both little-endian and big-endian code generation.

## 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?

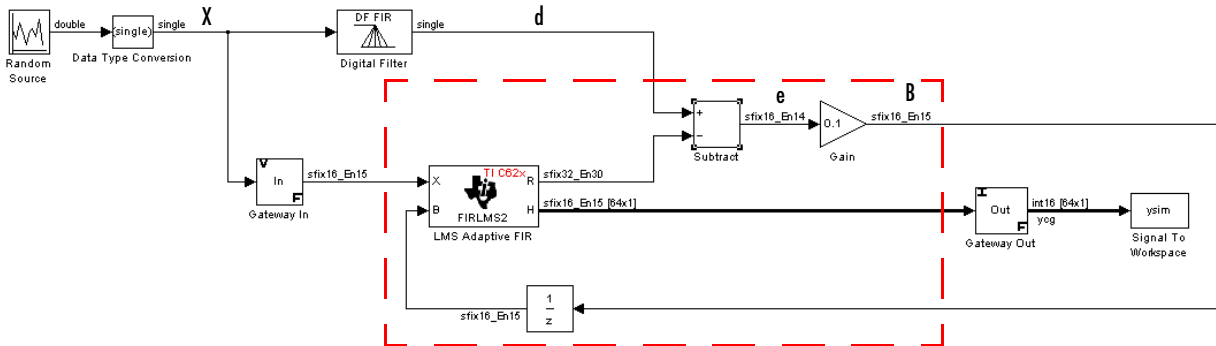
If selected, the filter taps are produced as output H. If not selected, H is suppressed.

## Algorithm

In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir1ms2`. During code generation, this block calls the `DSP_fir1ms2` 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 the Filter Design Toolbox. `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 — Math and Matrices

## 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.

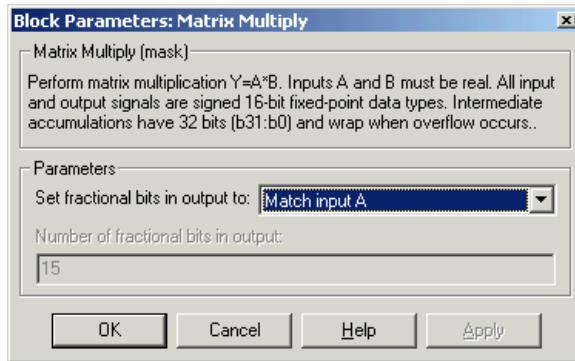
Alternatively, you can choose to output 16 bits according to how many fractional bits you want in the output. The number of fractional bits in the accumulator value is the sum of the fractional bits of the two inputs.

	<b>Input A</b>	<b>Input B</b>	<b>Accumulator Value</b>
<b>Total Bits</b>	16	16	32
<b>Fractional Bits</b>	$R$	$S$	$R + S$

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 also supports both little-endian and big-endian code generation.

## 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**.

# C64x Matrix Multiply

---

## Algorithm

In simulation, the Matrix Multiply block is equivalent to the TMS320C64x DSP Library assembly code function DSP\_mat\_mu1. During code generation, this block calls the DSP\_mat\_mu1 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.

<b>Set fractional bits in output to</b>	<b>Data Type</b>	<b>Accumulator Bits</b>
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.

<b>Set fractional bits in output to</b>	<b>Data Type</b>	<b>Accumulator Bits</b>
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

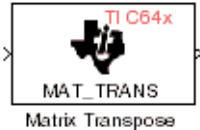
C64x Vector Multiply



**Purpose** Matrix transpose input signal

**Library** C64x DSP Library — Math and Matrices

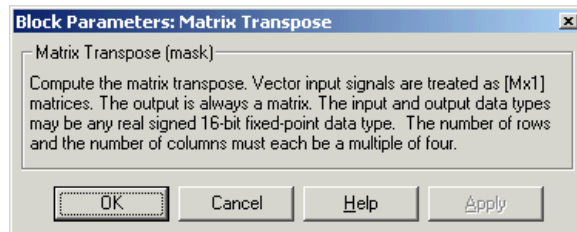
## Description



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 also supports both little-endian and big-endian code generation.

## 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.

# C64x Radix-2 FFT

**Purpose** Radix-2 decimation-in-frequency forward FFT of complex input vector

**Library** C64x DSP Library — Transforms

## Description

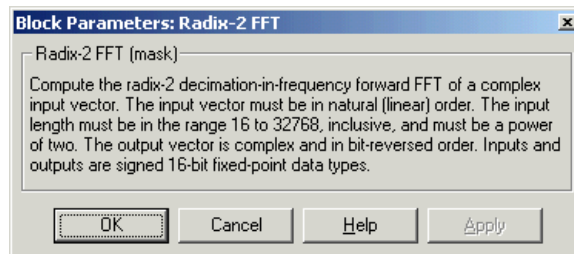


The C64x Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block 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

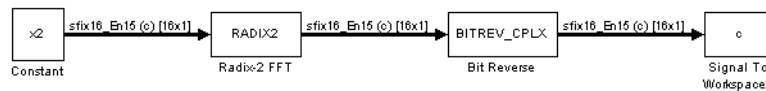


## Algorithm

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.



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

# C64x Radix-2 IFFT

**Purpose** Radix-2 inverse FFT of complex input vector

**Library** C64x DSP Library — Transforms

## 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.

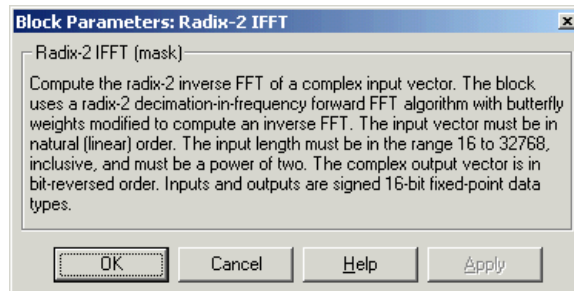
The radix2 routine used by this block employs a radix-2 FFT of length  $L=2^k$ . In order to ensure that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by  $k$  bits to the left relative to the location 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.

## 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

# C64x Radix-4 Real FIR

**Purpose** Filter real input signal using real FIR filter

**Library** C64x DSP Library — Filtering

## Description

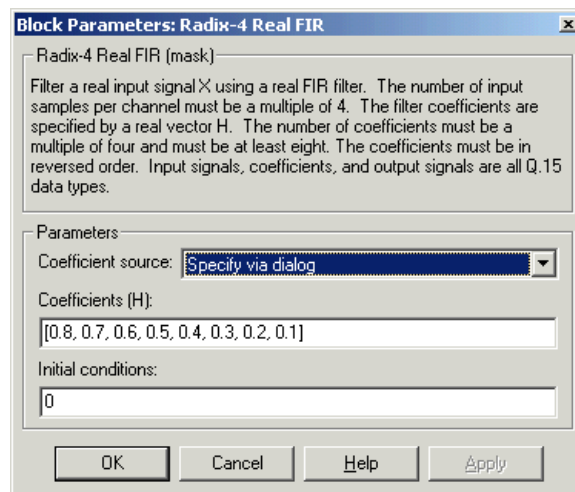


The C64x Radix-4 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.

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 both little-endian and big-endian code generation.

## 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
- **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

# C64x Radix-8 Real FIR

**Purpose** Filter real input signal using real FIR filter

**Library** C64x DSP Library — Filtering

## Description

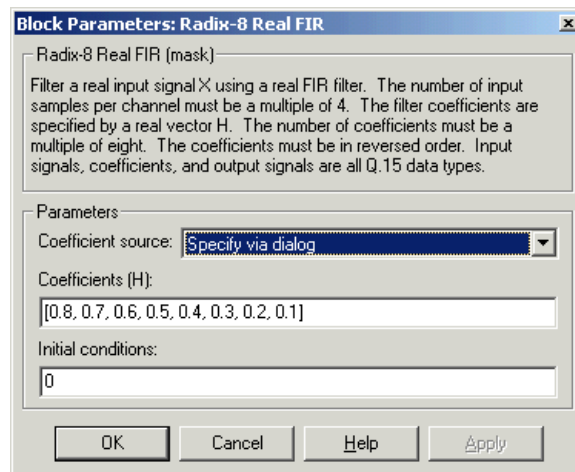


The C64x Radix-8 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.

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
- 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 —  $b(n)$ ,  $b(n-1)$ , ...,  $b(0)$ . 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

# C64x Real Forward Lattice All-Pole IIR

**Purpose** Filter real input signal using lattice IIR filter

**Library** C64x DSP Library — Filtering

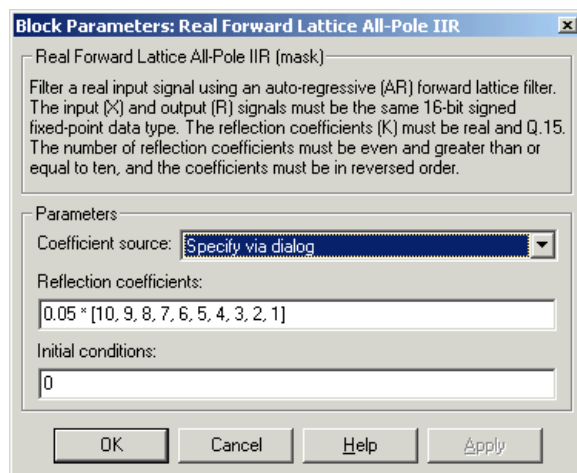
## Description



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 —  $k(n), k(n-1), \dots, k(0)$ . 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 both little-endian and big-endian code generation.

## 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
- **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

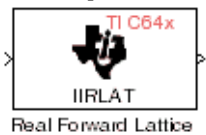
C64x Real IIR

# C64x Real IIR

**Purpose** Filter real input signal using IIR filter

**Library** C64x DSP Library — Filtering

## Description

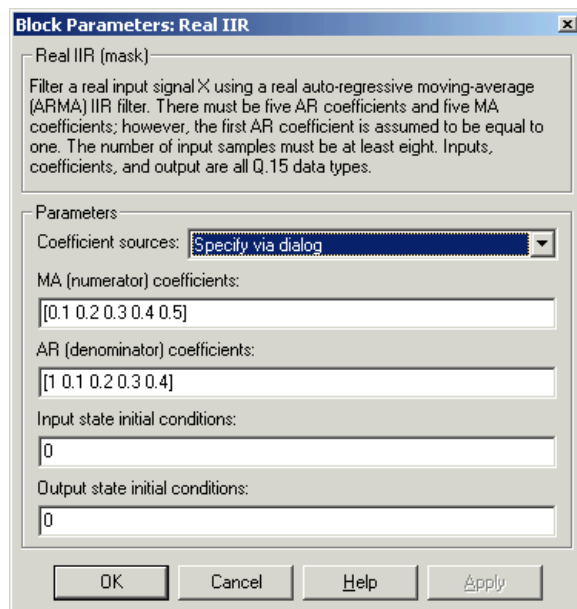


The C64x Real IIR block filters a real input signal  $X$  using a real autoregressive moving-average (ARMA) IIR Filter. This filter is implemented using a direct form I structure. You must use at least eight input samples.

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 both little-endian and big-endian code generation.

## 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.

# C64x Real IIR

---

- 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

**Purpose** Fraction and exponent of reciprocal of real input signal

**Library** C64x DSP Library — Math and Matrices

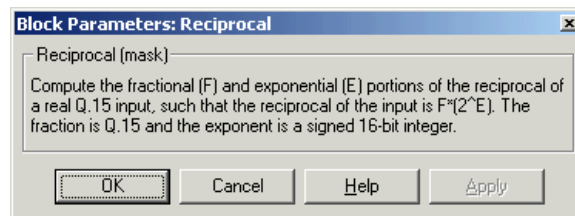
## Description



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 also supports both little-endian and big-endian code generation.

## 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.

# C64x Symmetric Real FIR

**Purpose** Filter real input signal using FIR filter

**Library** C64x DSP Library — Filtering

## 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.

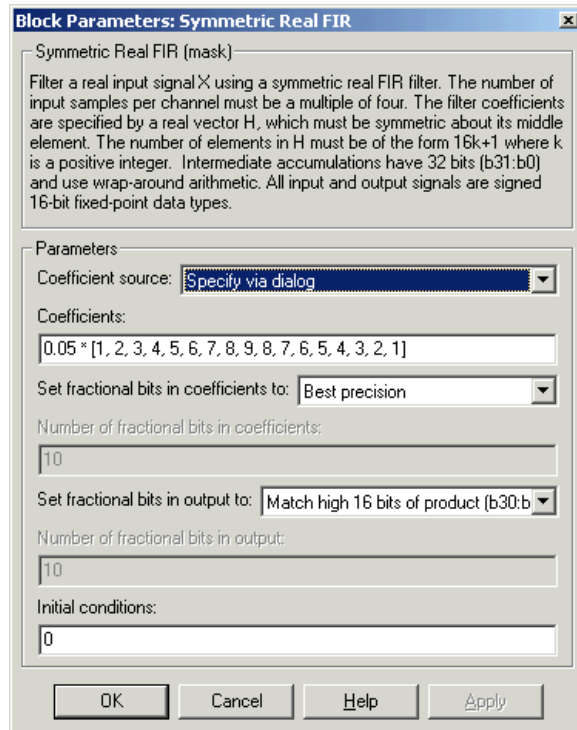
Intermediate multiplies and accumulates performed by this filter result in 32-bit accumulator values. However, the Symmetric Real FIR block only outputs 16 bits. You can choose to output 16 bits of the accumulator value in one of the following ways.

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 <b>Number of fractional bits in output</b> parameter

The Symmetric Real FIR block supports discrete sample times and only little-endian code generation.



## 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
- **Input port** — Accept the coefficients from port H

# C64x Symmetric Real FIR

---

## 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
- **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 5-164 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 — Math and Matrices

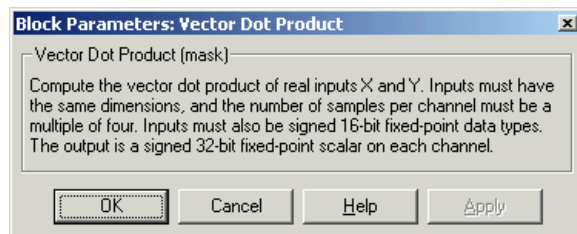
## Description



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.

The Vector Dot Product block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## Dialog Box



## 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 — Math and Matrices

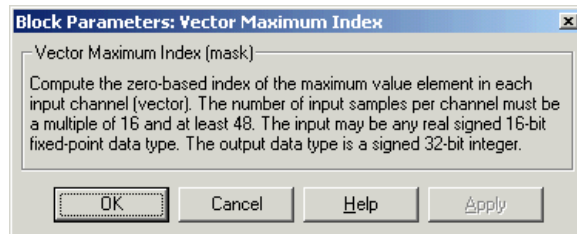
## Description



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 also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

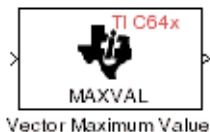
In simulation, the Vector Maximum Index block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_maxidx`. During code generation, this block calls the `DSP_maxidx` routine to produce optimized code.

# C64x Vector Maximum Value

**Purpose** Maximum value for each input signal channel

**Library** C64x DSP Library — Math and Matrices

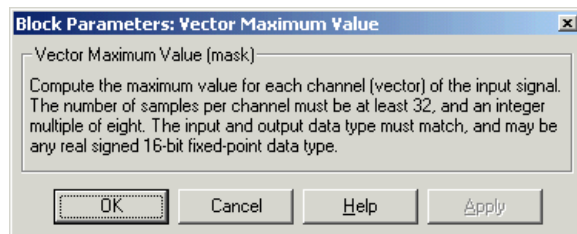
## Description



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.

The Vector Maximum Value block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## 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

# C64x Vector Minimum Value

**Purpose** Minimum value for each input signal channel

**Library** C64x DSP Library — Math and Matrices

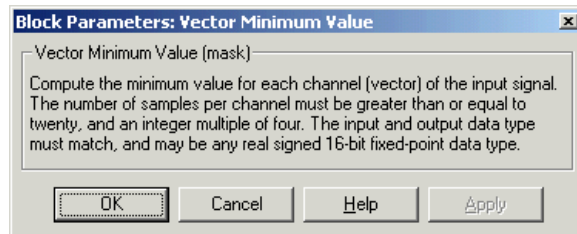
## Description



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 also supports both little-endian and big-endian code generation.

## 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 — Math and Matrices

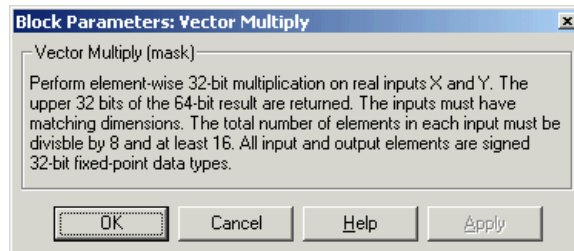
## Description



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.

The Vector Multiply block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

In simulation, the Vector Multiply block is equivalent to the TMS320C64x DSP Library assembly code function DSP\_mu132. During code generation, this block calls the DSP\_mu132 routine to produce optimized code.

## See Also

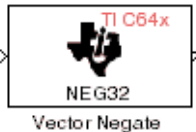
C64x Matrix Multiply



**Purpose** Negate each input signal element

**Library** C64x DSP Library — Math and Matrices

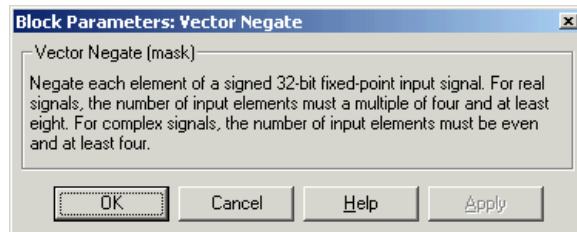
## Description



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 also supports both little-endian and big-endian code generation.

## Dialog Box



## Algorithm

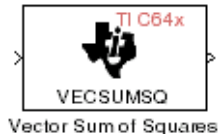
In simulation, the Vector Negate block is equivalent to the TMS320C64x DSP Library assembly code function DSP\_neg32. During code generation, this block calls the DSP\_neg32 routine to produce optimized code.

# C64x Vector Sum of Squares

**Purpose** Sum of squares over each real input channel

**Library** C64x DSP Library — Math and Matrices

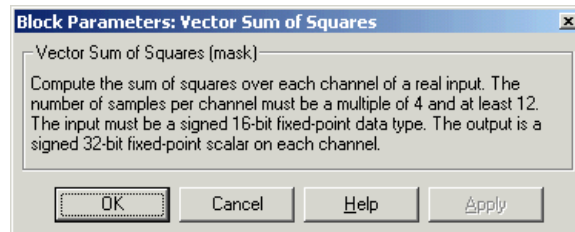
## Description



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 also supports both little-endian and big-endian code generation.

## 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 — Math and Matrices

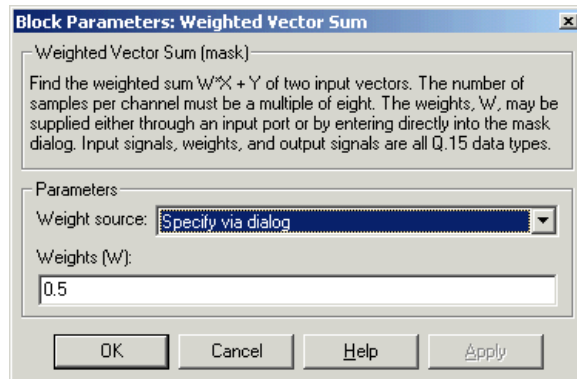
## Description



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 also supports both little-endian and big-endian code generation.

## 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

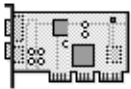
In simulation, the Weighted Vector Sum block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

**Purpose**

Configure model for C6711 DSP Starter Kit

**Library**

Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

**Description**

C6711DSK

Options on the block mask let you set features of code generation for your C6711 DSP Starter Kit target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6711 DSK must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

---

**Note** This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

---

The processor and target options you specify on this block are:

- Target board information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, DSP/BIOS, and custom sections

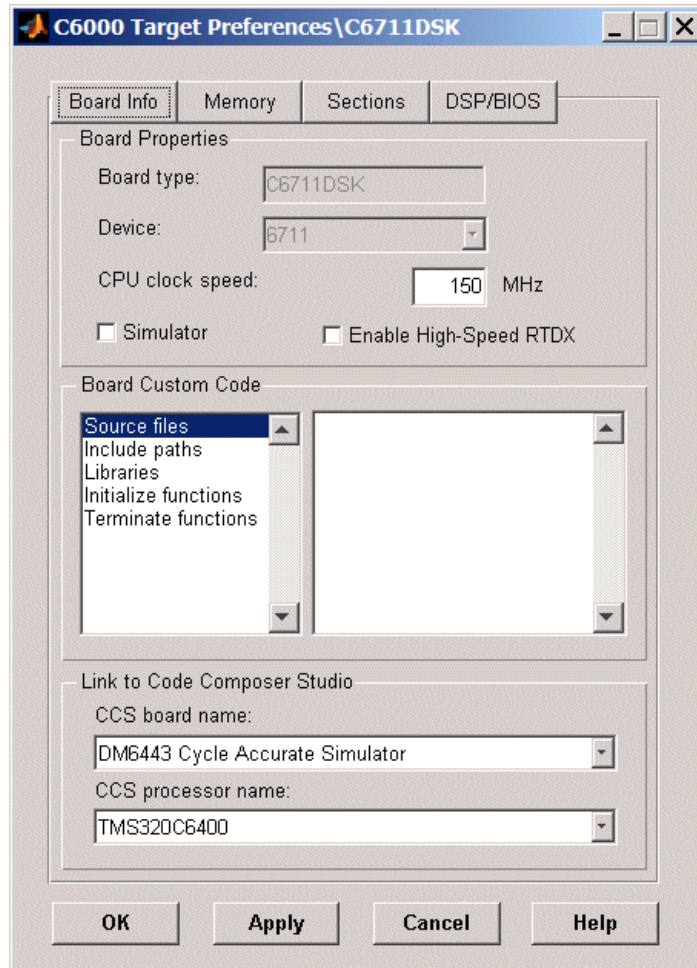
Setting the options included in this dialogbox results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both steps are essential for targeting any board that is custom or explicitly supported, such as the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

## Generating Code from Model Subsystems

Real-Time Workshop provides the ability to generate code from a selected subsystem in a model. To generate code for the C6711 DSK from a subsystem, the subsystem model must include a C6711DSK target preferences block.

### Dialog Box



All target preferences block dialog boxes provide tabbed access to panes the following panes with options you set for the target processor and target board:

- **Board info** — Select the target board and processor, set the clock speed, and identify the target.
- **Memory** — Set the memory allocation and layout on the target processor (memory mapping).
- **Sections** — Determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS and compiler information.
- **DSP/BIOS** — Specify how to configure tasking features of DSP/BIOS.

## Board Info Pane

The following options appear on the **Board Info** pane for the **C6000 Target Preferences** dialog box.

### Board Type

Lets you enter the type of board you are targeting with the model. The C6711DSK block comes with C6711DSK defined as the default board type.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model's real-time results may be wrong, and code profiling results are not correct.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if either of the following conditions occur:

- If your model does not include ADC or DAC blocks
- When the processing rates in your model change (the model is multirate)

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. You can change the rate with the DIP switches on the board or from one of the software utilities provided by Texas Instruments.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 1,000,000 \text{ clock ticks}$$

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

## **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

## **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for this model.

## **Board Custom Code**

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths.

When you enter a path to a file, library, or other custom code, use the string



`$(install_dir)`

to refer to the CCS installation directory.

Enter new paths or files (custom code items) one to a line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

- **Source files** — you enter the full paths to source code files to use with this target. By default there are no entries in this parameter.
- **Include paths** — If you require additional files on your path, you add them by typing the path into the text area. The default setting does not include additional paths.
- **Libraries** — these entries identify specific libraries that the target requires. They appear on the list by default if required. Add more as you require by entering the full path to the library with the library file in the text area. No additional libraries appear here in the default configuration.
- **Initialize functions** — If your project requires an initialize function, enter it here. By default, this is empty.
- **Terminate functions** — enter a function to run when a program terminates. The default setting is not to include a specific termination function.

## CCS Board Name

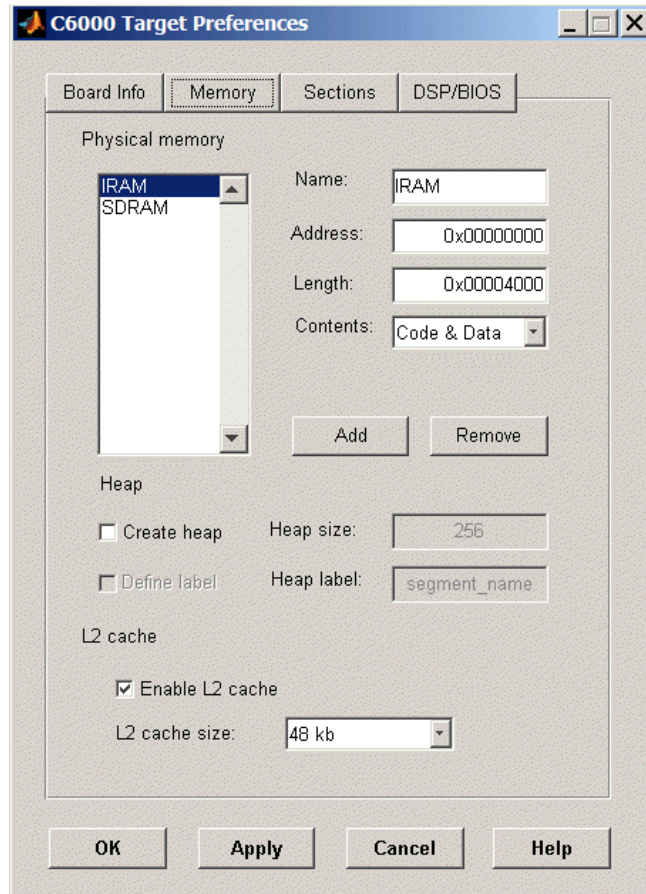
Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one that you are targeting your code for.

## CCS Processor Name

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.

## Memory Pane

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



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

- **Physical Memory** — specifies the processor and board memory map
- **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 may 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 online help system for Code Composer Studio.

## 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. C6711DSK boards provide SDRAM memory segments 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** details 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 may 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 may 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 on the **Physical memory** list and click **Remove** to delete the segment.

## Create Heap

Selecting this option enables creating the heap, and enables the **Heap size** option.

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**

Enabled by selecting **Define label**, you use this option 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**

C6711 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.

### **L2 Cache size**

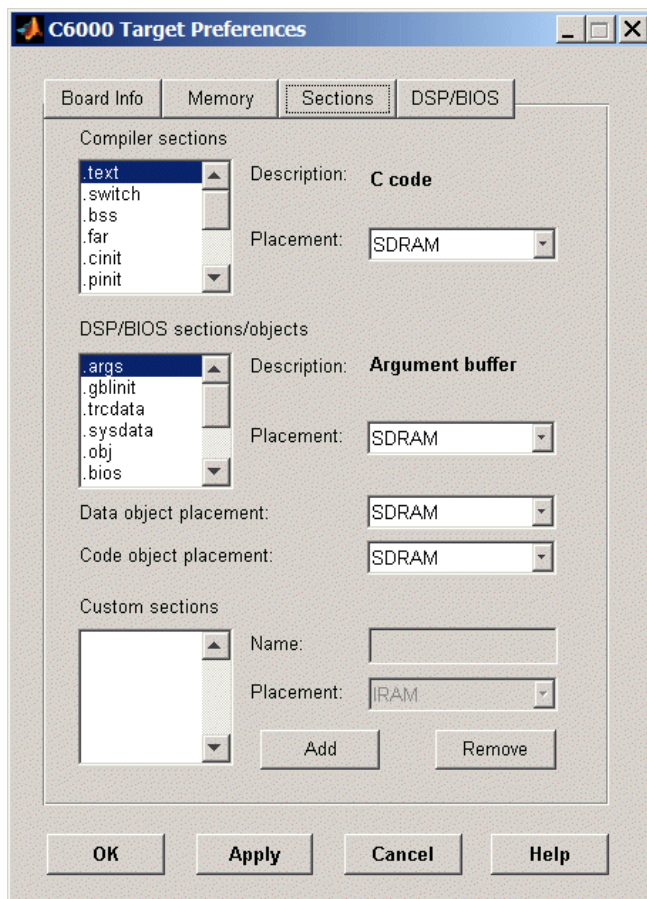
When you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

### **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. Commonly used sections include `.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 online help.



Within this pane, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections**, **DSP/BIOS sections/objects**, and **Custom sections** lists in the pane.

All sections do not appear on all lists. The list the string appears on is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.system	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 get 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
- .system



Other sections appear on the list as well:

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

---

**Note** The C/C++ compiler does not use the .data 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 presently allocated in memory.

### **Description**

Provides a brief explanation of the contents of the selected entry on 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. C6711 DSK boards provide IRAM and SDRAM.

### **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, get 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 location 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. You can also 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 and click **Remove**.

## DSP/BIOS Pane

Options on this pane let you specify how to configure tasking features of DSP/BIOS.

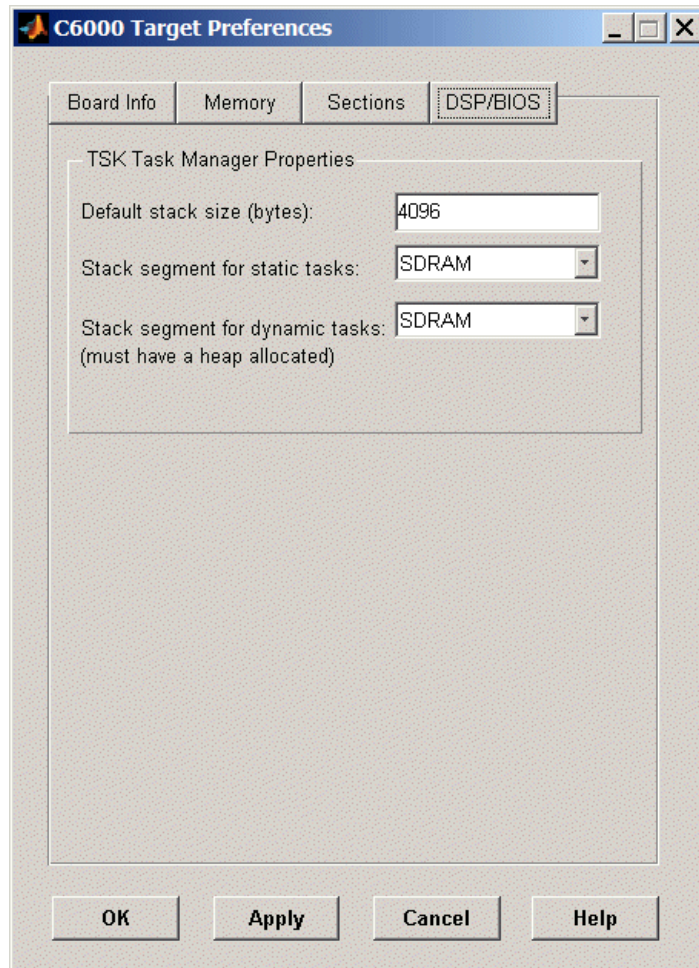
The asynchronous task scheduler uses these options when you select the **Incorporate DSP/BIOS** option in the model configuration set. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their block dialog boxes so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you clear the **Incorporate DSP/BIOS** option, you disable the options in this pane. Your project does not include DSP/BIOS tasks, and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler.

For more information about tasks, refer to the Code Composer Studio online help.



Within this pane, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. Note that the Sections pane includes DSP/BIOS configuration options as well. The options specify the stack use and locations on the stack for static and dynamic tasks.

### **Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the

DSP/BIOS stack in bytes allocated for each task. 4096 bytes is the default value. You can set any size up to the limits for the processor. Set the stack size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, this might cause the task to write into other memory or data areas, possibly causing unpredictable behavior.

**Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers options SDRAM and ISRAM for locating the stack in memory, with SDRAM as the default section. The Memory pane provide more options for the physical memory on the processor.

**Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, SDRAM is the only valid stack location in memory.

**See Also**

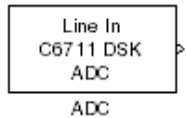
Custom C6000

# C6711 DSK ADC

**Purpose** Digitized output from codec to processor

**Library** C6711 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



Use the C6711 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 C6711 DSK ADC block in your Simulink block diagram lets you use the audio coder-decoder module (codec) on the C6711 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 are related to the model you are using in Simulink, 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 C6711 DSK hardware affected.

Option	Affected Hardware
ADC Source	Codec
Mic	Codec
Output data type	TMS320C6711 digital signal processor
Samples per frame	Direct memory access functions
Scaling	TMS320C6711 digital signal processor
Source gain (dB)	Codec

You can select one of three input sources from the **ADC source** list:

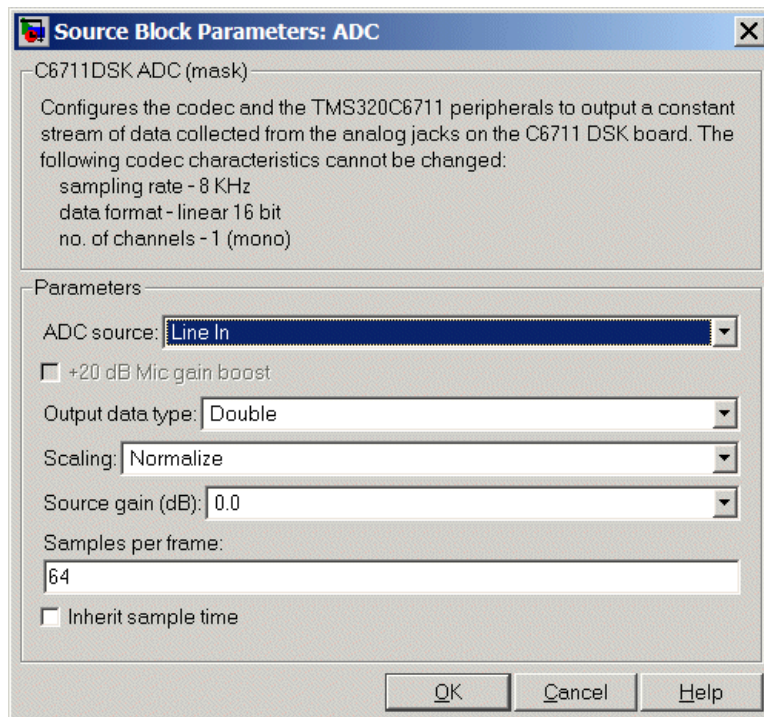
- **Line In** — the codec accepts input from the line in connector (LINE IN) on the board's mounting bracket.
- **Mic** — the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.
- **Loopback** — routes the analog signal from the codec output back to the codec input. Can be useful in some feedback applications.

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.

Selecting **Loopback** for **ADC source** configures the C6711 DSK to capture the output from the codec as the input to the C6711 DSK ADC. When you select **Loopback**, your model must include both the C6711 DSK ADC and C6711 DSK DAC blocks.

**Source gain (dB)** lets you add gain to the input signal before the A/D conversion. When you select **Loopback** as the **ADC source**, your specified source gain is not added to the input signal. Select the appropriate gain from the list.

## Dialog Box



### ADC source

The input source to the codec. **Line In** is the default.

# C6711 DSK ADC

---

## **+20 dB Mic gain boost**

Boosts the input signal by +20dB when **ADC source** is Mic. Gain is applied before analog-to-digital conversion.

## **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  $\pm 1.0$ , based on the codec data format. Select either Normalize or Integer Value. Normalize is the default setting.

## **Source gain (dB)**

Specifies the amount to boost the input before conversion. Select from the range 0.0 to 12.0 dB in 1.5 dB increments. Applies to the input signal when **ADC source** is Line In or Mic In.

## **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. 64 samples per frame is the default setting. 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.

## **See Also**

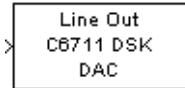
C6711 DSK DAC



**Purpose** Use codec to convert digital input to analog output

**Library** C6711 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



Adding the C6711 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the LINE OUT connection on the C6711 DSK mounting bracket. When you add the C6711 DSK DAC block, the digital signal received by the codec is converted to an analog signal. After converting the digital signal to analog form (digital-to-analog (D/A) conversion), the codec sends the signal to the output audio jack.

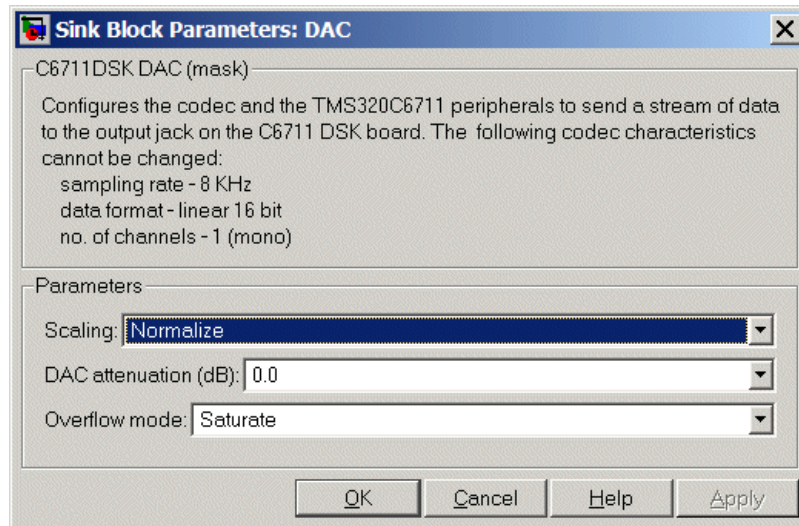
One of the configuration options in the block affects the codec. The remaining options relate to the model you are using in Simulink and the signal processor on the board. In the following table, you find each option listed with the C6711 DSK hardware affected by your selection.

Option	Affected Hardware
DAC attenuation	Codec
Overflow mode	TMS320C6711 Digital Signal Processor
Scaling	TMS320C6711 Digital Signal Processor

To attenuate the output signal after the D/A conversion, select an attenuation from the **DAC attenuation** list. Available attenuation values range from 0.0 to 36.0 dB in 1.5 dB increments. You must select from the list; you cannot enter a value for the attenuation.

# C6711 DSK DAC

## Dialog Box



### Scaling

Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range  $\pm 1.0$ . Matching the setting for the C6711 DSK ADC block is usually appropriate here.

### DAC attenuation

Specifies the amount to attenuate the block output after D/A conversion.

### Overflow mode

Determines how the codec responds to data that is outside the range specified by the **Scaling** parameter.

## See Also

C6711 DSK ADC

**Purpose** Simulate or read DIP switches

**Library** C6711 DSK Board Support in Embedded Target for TI C6000 DSP

**Description** Added to your model, this block behaves differently in simulation than in code generation and targeting.



**Simulation** — the options **USER\_SW1**, **USER\_SW2**, and **USER\_SW3** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6711 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 C6711 DSK

<b>USER_SW1 (LSB)</b>	<b>USER_SW2</b>	<b>USER_SW3 (MSB)</b>	<b>Boolean Output</b>	<b>Integer Output</b>
Cleared	Cleared	Cleared	000	0
Selected	Cleared	Cleared	001	1
Cleared	Selected	Cleared	010	2
Selected	Selected	Cleared	011	3
Cleared	Cleared	Selected	100	4
Selected	Cleared	Selected	101	5
Cleared	Selected	Selected	110	6
Selected	Selected	Selected	111	7

Selecting the **Integer** data type results in the switch settings generating integers in the range from 0 to 7 (uint8), corresponding to converting the string

# C6711 DSK DIP Switch

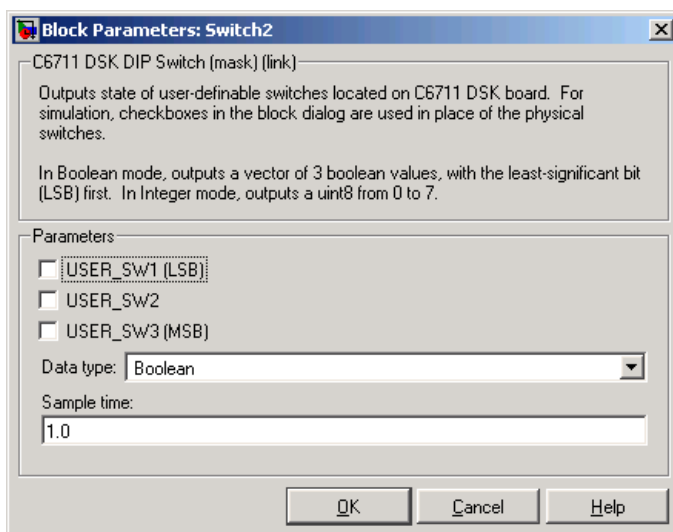
of individual switch settings to a decimal value. In the **Boolean** data type, the output string presents the separate switch setting for each switch, with the **USER\_SW1** status represented by the least significant bit (LSB) and the status of **USER\_SW3** represented by the most significant bit (MSB).

**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 Table . 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 **USER\_SW1**, **USER\_SW2**, and **USER\_SW3** 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 Output Values From The User DIP Switches on the C6711 DSK shows

**Output Values From The User DIP Switches on the C6711 DSK**

<b>USER_SW1 (LSB)</b>	<b>USER_SW2</b>	<b>USER_SW3 (MSB)</b>	<b>Boolean Output</b>	<b>Integer Output</b>
Off	Off	Off	000	0
On	Off	Off	001	1
Off	On	Off	010	2
On	On	Off	011	3
Off	Off	On	100	4
On	Off	On	101	5
Off	On	On	110	6
On	On	On	111	7

## Dialog Box



Opening this dialog box causes a running simulation to pause. Refer to “Changing Source Block Parameters” in your online Simulink documentation for details.

### **USER\_SW1**

Simulate the status of the user-defined DIP switch on the board.

### **USER\_SW2**

Simulate the status of the user-defined DIP switch on the board.

### **USER\_SW3**

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 logical string of three bits.

Each bit represents the status of one DIP switch; the LSB is switch **USER\_SW1** and the MSB is switch **USER\_SW3**. The other data type, **Integer**, converts the logical string to an equivalent unsigned 8-bit (uint8)

# C6711 DSK DIP Switch

---

decimal value. For example, if the logical string is 101, the decimal conversion yields 5.

## **Sample time**

Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second (**1/Sample time**).

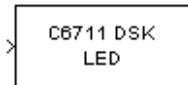
## Purpose

Control LEDs

## Library

C6711 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



Adding the C6711 DSK LED block to your Simulink block diagram lets you trigger all three of the user red light emitting diodes (LED) on the C6711 DSK. To use the block, send a nonzero real scalar to the block. The C6711 DSK LED block triggers all three user LEDs located on the C6711 DSK.

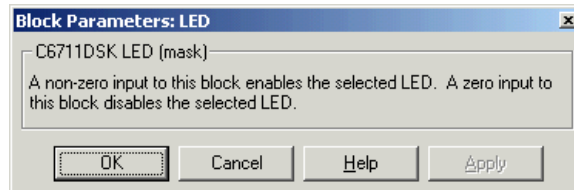
When you add this block to a model, and send a real scalar to the block input, the block sets the LED state based on the input value it receives:

- When the block receives an input value equal to 0, the specified LEDs are turned off (disabled)
- When the block receives a nonzero input value, the specified LEDs are turned on (enabled)

To activate the block, send it a scalar of any real data type. Vectors do not work to activate LEDs; nor do complex numbers as scalars or vectors.

All LEDs maintain their state until their controlling C6711 DSK LED block receives an input value that changes the state. Enabled LEDs stay on until the block receives an input value equal to zero and turns the LEDs off; disabled LEDs stays off until turned on. Resetting the C6711 DSK turns off all user LEDs.

## Dialog Box



This dialog box does not have any user-selectable options.

# C6711 DSK RESET

---

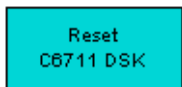
## Purpose

Reset to initial conditions

## Library

C6711 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



Double-clicking this block in a Simulink model window resets the C6711 DSK that is running the executable code built from the model. When you double-click the C6713 DSK RESET block, the block runs the software reset function provided by CCS that resets the processor on your C6711 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 C6711 DSK. In other words, anytime you double-click a C6711 DSK RESET block you reset your C6711 DSK.

## Dialog Box

This block does not have settable options and does not provide a user interface dialog box.



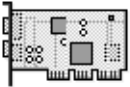
## Purpose

Configure model for C6713 DSP Starter Kit

## Library

Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

## Description



C6713DSK

Options on the block mask let you set features of code generation for your C6713 DSP Starter Kit target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6713 DSK must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

---

**Note** This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

---

The processor and target options you specify on this block are:

- Target board information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, DSP/BIOS, and custom sections

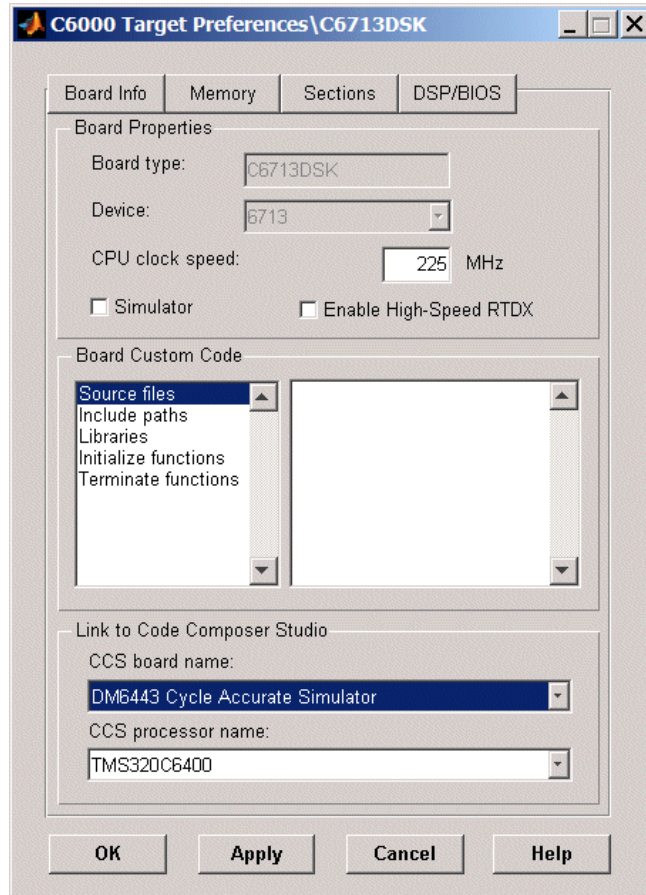
Setting the options included in this dialogbox results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both steps are essential for targeting any board that is custom or explicitly supported, such as the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

## Generating Code from Model Subsystems

Real-Time Workshop provides the ability to generate code from a selected subsystem in a model. To generate code for the C6713 DSK from a subsystem, the subsystem model must include a C6713DSK target preferences block.

### Dialog Box



All target preferences block dialog boxes provide tabbed access to panes the following panes with options you set for the target processor and target board:

- **Board info** — Select the target board and processor, set the clock speed, and identify the target.

- **Memory** — Set the memory allocation and layout on the target processor (memory mapping).
- **Sections** — Determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS and compiler information.
- **DSP/BIOS** — Specify how to configure tasking features of DSP/BIOS.

## Board Info Pane

The following options appear on the **Board Info** pane for the **C6000 Target Preferences** dialog box.

### Board Type

Lets you enter the type of board you are targeting with the model. You can enter **Custom** to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as **C6711DSK**. **Board type** for this block is set to **C6713 DSK** by default.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model's real-time results may be wrong, and code profiling results are not correct.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if either of the following conditions occur:

- If your model does not include ADC or DAC blocks

- When the processing rates in your model change (the model is multirate)

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. You can change the rate with the DIP switches on the board or from one of the software utilities provided by Texas Instruments.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 1,000,000 \text{ clock ticks}$$

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

## **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

## **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for code generated from this model.

## **Board Custom Code**

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths.

When you enter a path to a file, library, or other custom code, use the string

```
$(install_dir)
```

to refer to the CCS installation directory.

Enter new paths or files (custom code items) one to a line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

- **Source files** — you enter the full paths to source code files to use with this target. By default there are no entries in this parameter.
- **Include paths** — If you require additional files on your path, you add them by typing the path into the text area. The default setting does not include additional paths.
- **Libraries** — these entries identify specific libraries that the target requires. They appear on the list by default if required. Add more as you require by entering the full path to the library with the library file in the text area. No additional libraries appear here in the default configuration.
- **Initialize functions** — If your project requires an initialize function, enter it here. By default, this is empty.
- **Terminate functions** — enter a function to run when a program terminates. The default setting is not to include a specific termination function.

### **CCS Board Name**

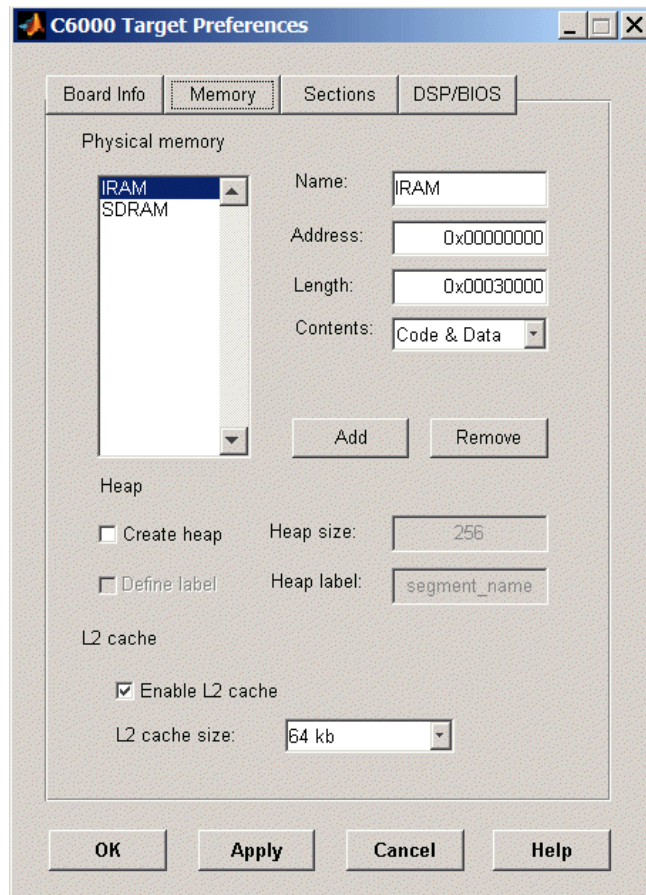
Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one that you are targeting your code for.

### **CCS Processor Name**

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.

### **Memory Pane**

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



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

- **Physical Memory** — specifies the processor and board memory map
- **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 may 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 online help system for Code Composer Studio.

## 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. C6713 DSK boards provide IRAM and SDRAM memory segments 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** details 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 may 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 may 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 on the **Physical memory** list and click **Remove** to delete the segment.

## Create Heap

Selecting this option enables creating the heap, and enables the **Heap size** option.



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**

Enabled by selecting **Define label**, you use this option 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**

C6713 processors support an L2 cache memory structure that you can configure as SRAM and partial cache.

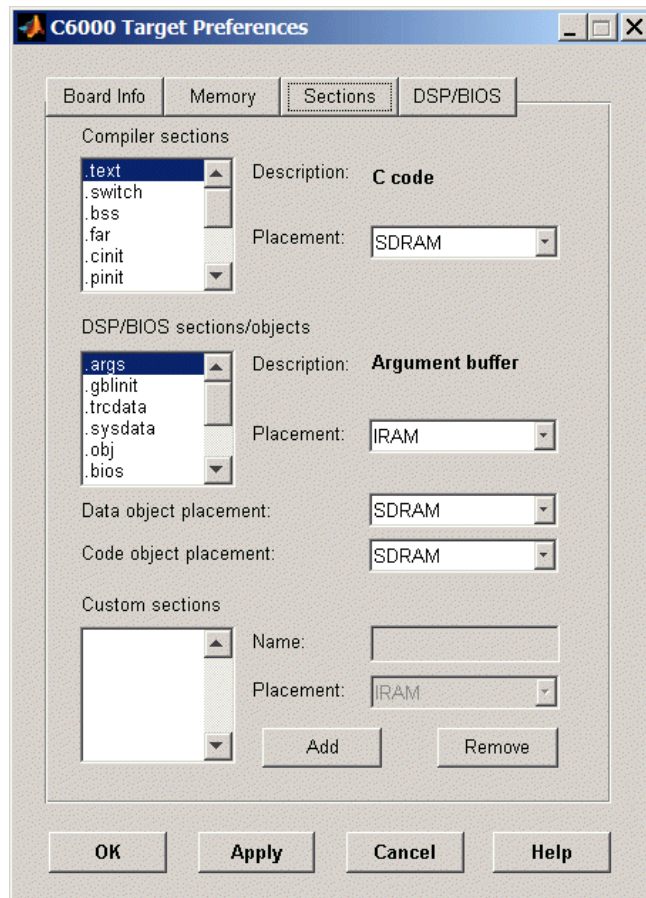
### **L2 Cache size**

When you enable the L2 cache, use this list to determine the size of the cache allotted. 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. Commonly used sections include `.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 online help.



Within this pane, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections**, **DSP/BIOS sections/objects**, and **Custom sections** lists in the pane.

All sections do not appear on all lists. The list the string appears on is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.system	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 get 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
- .system

Other sections appear on the list as well:

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

---

**Note** The C/C++ compiler does not use the .data 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 presently allocated in memory.

### **Description**

Provides a brief explanation of the contents of the selected entry on 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. For C6713 DSK targets, the list include IRAM and SDRAM segments.

### **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, get 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 location 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. You can also 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 and click **Remove**.

## DSP/BIOS Pane

Options on this pane let you specify how to configure tasking features of DSP/BIOS.

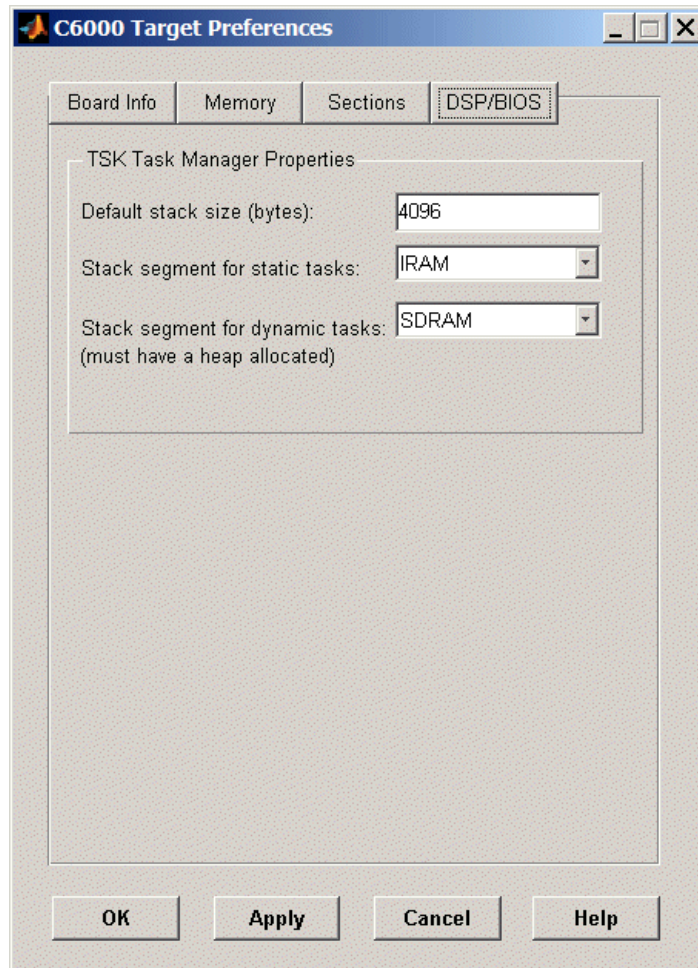
The asynchronous task scheduler uses these options when you select the **Incorporate DSP/BIOS** option in the model configuration set. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their block dialog boxes so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you clear the **Incorporate DSP/BIOS** option, you disable the options in this pane. Your project does not include DSP/BIOS tasks, and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler.

For more information about tasks, refer to the Code Composer Studio online help.



Within this pane, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. Note that the Sections pane includes DSP/BIOS configuration options as well. The options specify the stack use and locations on the stack for static and dynamic tasks.

### **Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the



DSP/BIOS stack in bytes allocated for each task. 4096 bytes is the default value. You can set any size up to the limits for the processor. Set the stack size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, this might cause the task to write into other memory or data areas, possibly causing unpredictable behavior.

**Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers options SDRAM and IRAM for locating the stack in memory, with SDRAM as the default section. The Memory pane provide more options for the physical memory on the processor.

**Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, SDRAM is the only valid stack location in memory.

**See Also**

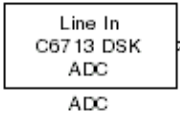
Custom C6000

# C6713 DSK ADC

**Purpose** Digitized signal output from codec to processor

**Library** C6713 DSK Board Support in Embedded Target for TI C6000 DSP

## 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.

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, 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.
- **Loopback** — routes the analog signal from the codec output back to the codec input. Can be useful in some feedback applications.

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, 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).

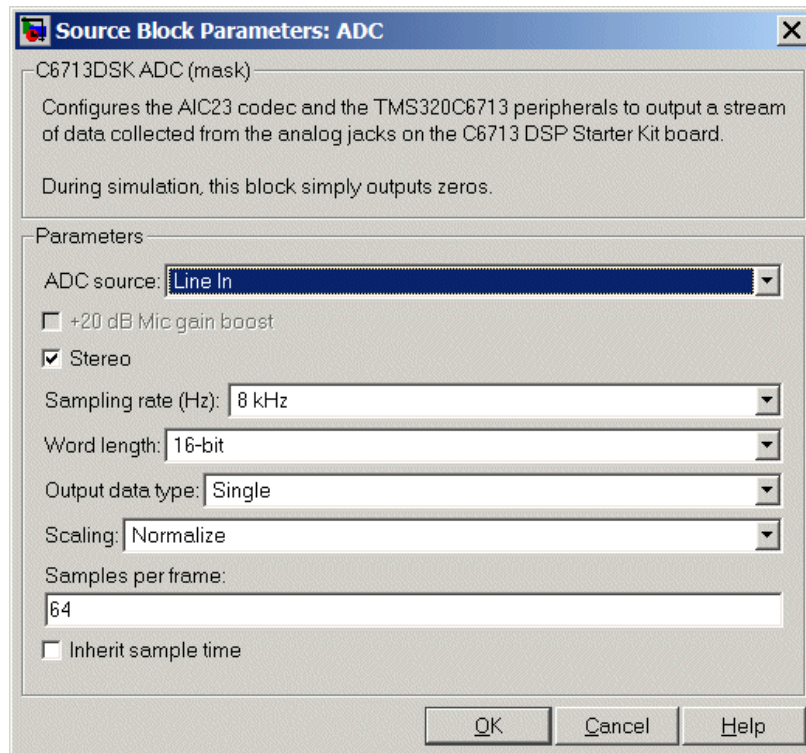
When the samples per frame setting is more than one, each frame of data is either the N-element vector (monaural input) or N-by-2 matrix (stereo input). For monaural input, the elements in each frame form the column vector of input audio data. In the stereo format, the frame is the matrix of audio data 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. When you select Loopback as the **ADC source**, your specified source gain is not added to the input signal. Select the appropriate gain from the list.

# C6713 DSK ADC

## Dialog Box



### ADC source

The input source to the codec. Line In is the default. Selecting the Mic option 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.

## **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  $\pm 1.0$ , 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. 64 samples per frame is the default setting. 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.

## **See Also**

C6713 DSK DAC

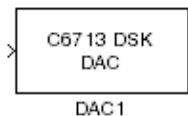
# C6713 DSK DAC

---

**Purpose** Configure codec to convert digital input to analog output

**Library** C6713 DSK Board Support in Embedded Target for TI C6000 DSP

## Description

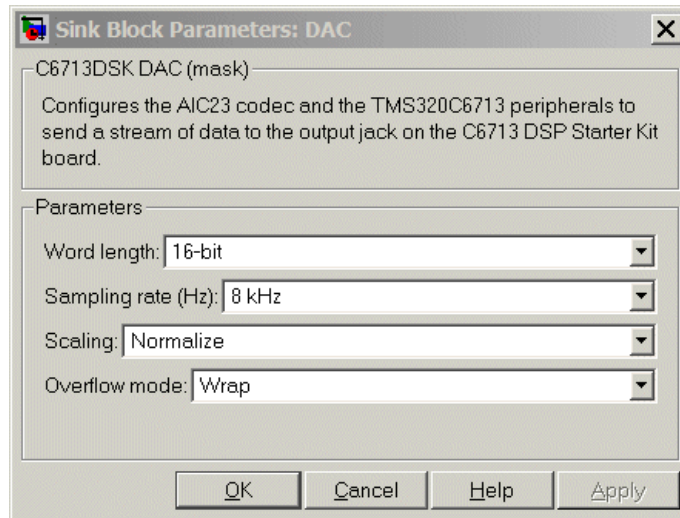


Adding the C6713 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the analog output jack on the C6713 DSK. When you add the C6713 DSK DAC block, the digital signal received by the codec is converted to an analog signal. After converting the digital signal to analog form (digital-to-analog (D/A) conversion), the codec sends the signal to the output jack.

One of the configuration options in the block affects the codec. The remaining options relate to the model you are using in Simulink and the signal processor on the board. In the following table, you find each option listed with the C6713 DSK hardware affected by your selection.

<b>Option</b>	<b>Affected Hardware</b>
<b>Overflow mode</b>	TMS320C6713 Digital Signal Processor
<b>Scaling</b>	TMS320C6713 Digital Signal Processor
<b>Word length</b>	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 default value is 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  $\pm 1.0$ . 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

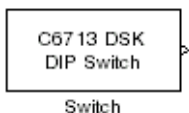
C6713 DSK ADC

# C6713 DSK DIP Switch

**Purpose** Simulate or read DIP switches

**Library** C6713 DSK Board Support in Embedded Target for TI C6000 DSP

**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
Selected	Selected	Selected	Cleared	0111	7
Cleared	Cleared	Cleared	Selected	1000	8
Selected	Cleared	Cleared	Selected	1001	9



# C6713 DSK DIP Switch

## 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
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.

## Output Values From The User DIP Switches on the C6713 DSK

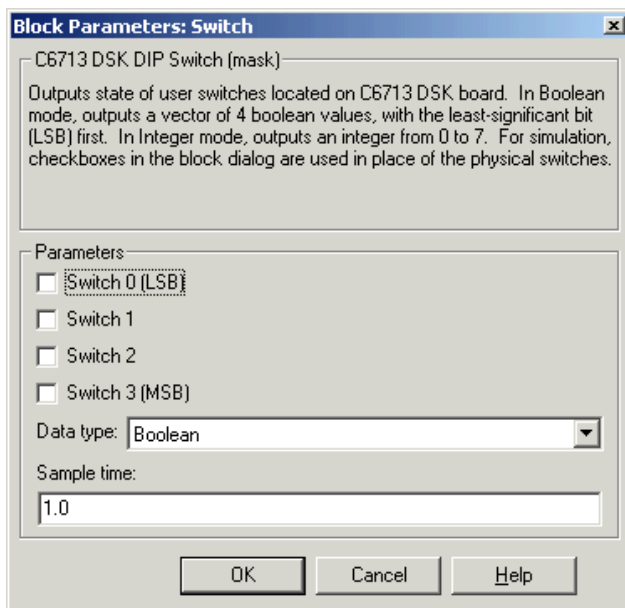
Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2

# C6713 DSK DIP Switch

**Output Values From The User DIP Switches on the C6713 DSK**

<b>Switch 0 (LSB)</b>	<b>Switch 1</b>	<b>Switch 2</b>	<b>Switch 3 (MSB)</b>	<b>Boolean Output</b>	<b>Integer Output</b>
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” 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, either 0 or 1.

## C6713 DSK DIP Switch

---

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. The default is 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

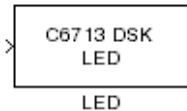
## Purpose

Control LEDs

## Library

C6713 DSK Board Support in Embedded Target for TI C6000 DSP

## 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.

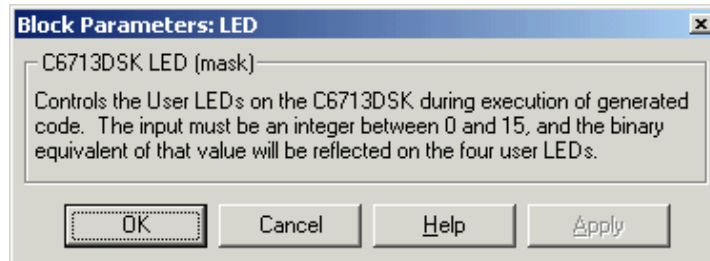
When you add this block to a model, and send a real scalar to the block input, the block sets the LED state based on the input value it receives:

- 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



This dialog box does not have any user-selectable options.

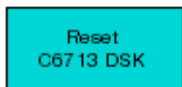
# C6713 DSK RESET

---

**Purpose** Reset to initial conditions

**Library** C6713 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



Reset

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 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.

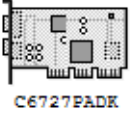
## Purpose

Configure model for C6727 Professional Audio Development Kit

## Library

Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

## Description



Options on the block mask let you set features of code generation for your C6727 Professional Audio Development Kit (PADK) target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6727 PADK must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

---

**Note** This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

---

The processor and target options you specify on this block are:

- Target board information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, DSP/BIOS, and custom sections

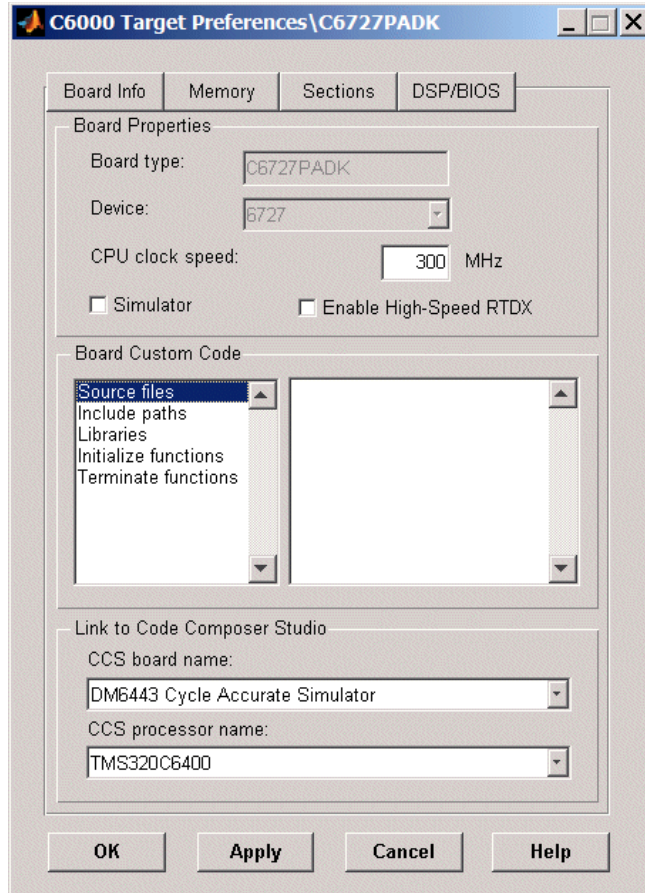
Setting the options included in this dialogbox results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both steps are essential for targeting any board that is custom or explicitly supported, such as the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

## Generating Code from Model Subsystems

Real-Time Workshop provides the ability to generate code from a selected subsystem in a model. To generate code for the C6727 PADK from a subsystem, the subsystem model must include a C6727PADK target preferences block.

### Dialog Box



All target preferences block dialog boxes provide tabbed access to panes the following panes with options you set for the target processor and target board:

- **Board info** — Select the target board and processor, set the clock speed, and identify the target.



- **Memory** — Set the memory allocation and layout on the target processor (memory mapping).
- **Sections** — Determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS and compiler information.
- **DSP/BIOS** — Specify how to configure tasking features of DSP/BIOS.

## Board Info Pane

The following options appear on the **Board Info** pane for the **C6000 Target Preferences** dialog box:

### Board Type

Lets you enter the type of board you are targeting with the model. You can enter **Custom** to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as **C6711DSK**. If you are using one of the explicitly supported boards, choose the target preferences block for that board and this field shows the proper board type.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model's real-time results may be wrong, and code profiling results are not correct.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if either of the following conditions occur:

- If your model does not include ADC or DAC blocks
- When the processing rates in your model change (the model is multirate)

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. You can change the rate with the DIP switches on the board or from one of the software utilities provided by Texas Instruments.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 1,000,000 \text{ clock ticks}$$

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

## **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

## **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for this model.

## **Board Custom Code**

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths.

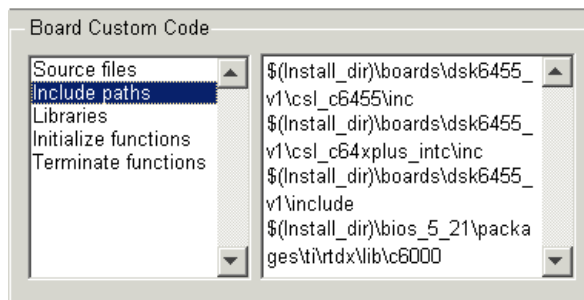
When you enter a path to a file, library, or other custom code, use the string

```
$(install_dir)
```

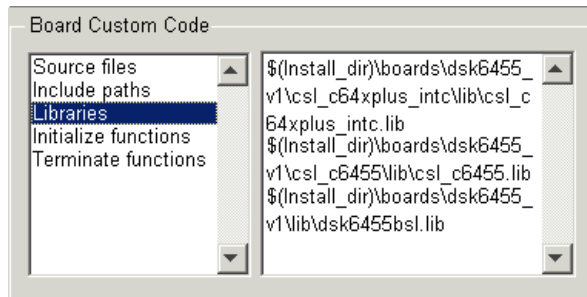
to refer to the CCS installation directory. The examples in the following figure use the string.

Enter new paths or files (custom code items) one to a line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

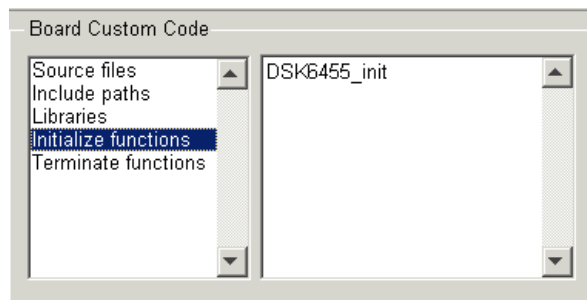
- **Source files** — Enter the full paths to source code files to use with this target. The default is blank.
- **Include paths** — C6727 PADK requires some additional files to work correctly. When you add this block to your model, the default `include` paths appear as shown in the following figure. These entries include chip support libraries, a BIOS addition, and an RTDX library. All are necessary for use. You can add further paths by typing the path into the text area.



- **Libraries** — These entries identify specific libraries that the target requires. They appear on the list by default, as shown in the following figure.



- **Initialize functions** — C6727 PADK targets require a specific initialization function, listed listed on the following figure as PADK6727\_init. Enter others if needed.



- **Terminate functions** — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

## CCS Board Name

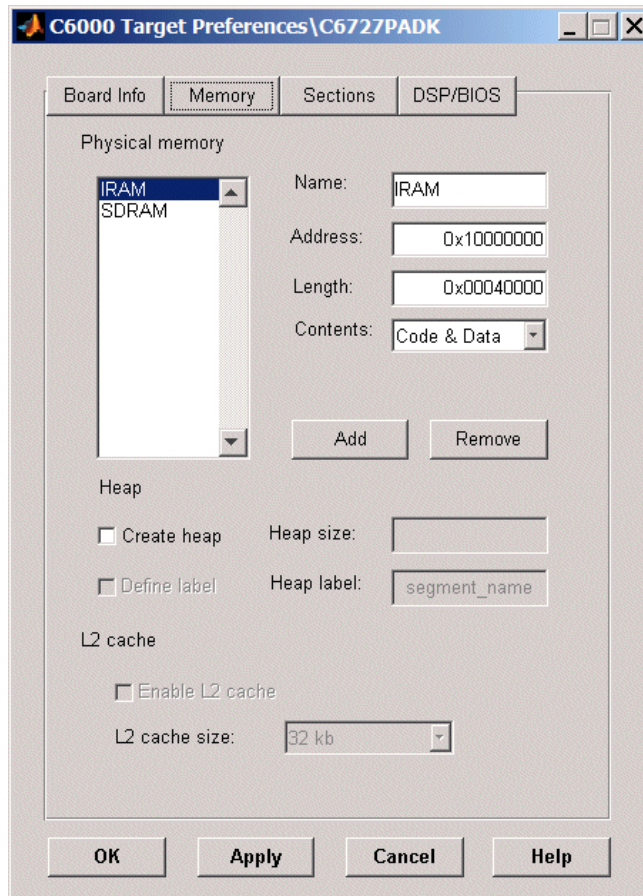
Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one to which you are targeting your code.

## CCS Processor Name

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.

## Memory Pane

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



The **Memory** pane contains memory options in three areas as shown in the preceding figure:

- **Physical Memory** — Specifies the processor and board memory map

- **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 may 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 online help system for Code Composer Studio.

## 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.
- C6711DSK boards provide SDRAM memory segments by default.

### Name

When you highlight an entry on the **Physical memory** list, the name of the entry appears in this field. 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).

---

**Note** 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** details 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 may 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 may 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. Enter the new name or click **Apply** to update the temporary name on the list to the name you want.

## **Remove**

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

## **Create Heap**

If your processor supports using a heap, as do the C6711, for example, selecting this option allows you to create 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.

---

**Note** 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** allows you to name the heap. Enter your label for the heap in the **Heap label** option.

## **Heap Label**

You enable this by selecting **Define label**. Use this option 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.

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

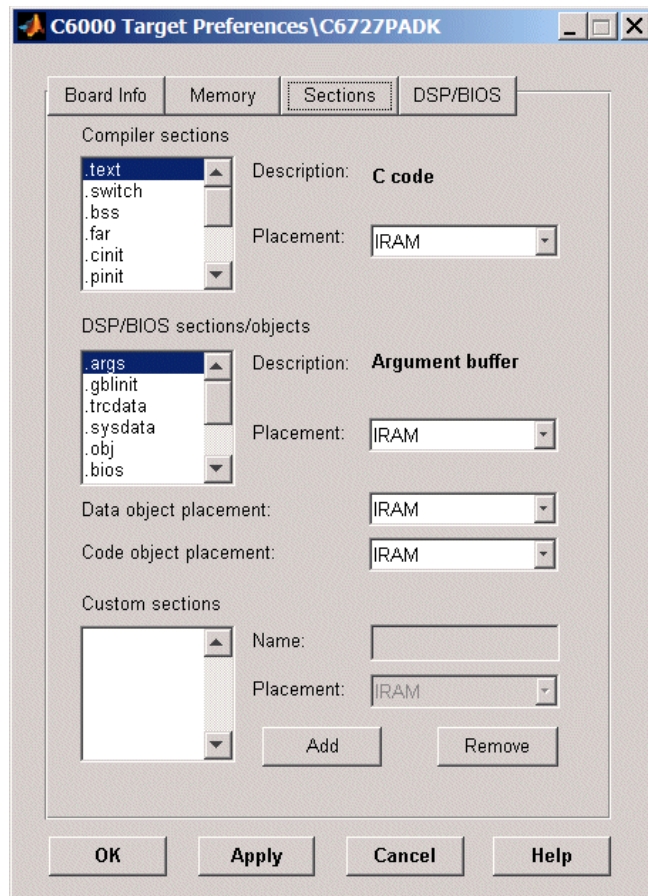
## **L2 Cache size**

When you enable the L2 cache, use this list to determine the size of the cache allotted. 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. Commonly used sections include `.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 online help.



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

This table provides brief definitions of the kinds of sections in the **Compiler sections**, **DSP/BIOS sections/objects**, and **Custom sections** lists in the pane.

All sections do not appear on all lists. The list the string appears on is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rt dx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.system	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 get allocated into memory as required by the configuration of your system. On the **Compiler sections** list, you find both initialized sections (sections that contain data or executable code) and uninitialized sections (sections that reserve space in memory). The initialized sections are:

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

These sections are uninitialized:

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

Other sections appear on the list as well:

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

---

**Note** The C/C++ compiler does not use the .data 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 presently allocated in memory.

### **Description**

Provides a brief explanation of the contents of the selected entry on 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

These objects are distinct from the entries on the **DSP/BIOS sections** list. DSP/BIOS objects, such as STS or LOG, 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 location 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, but instead a placeholder for a section for you to define.

### Name

You enter the name for your new section in this field. To add a new section, click **Add**. Then replace the temporary name with the name you want 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 a 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. You can also 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 and click **Remove**.

## DSP/BIOS Pane

Options on this pane let you specify how to configure tasking features of DSP/BIOS.

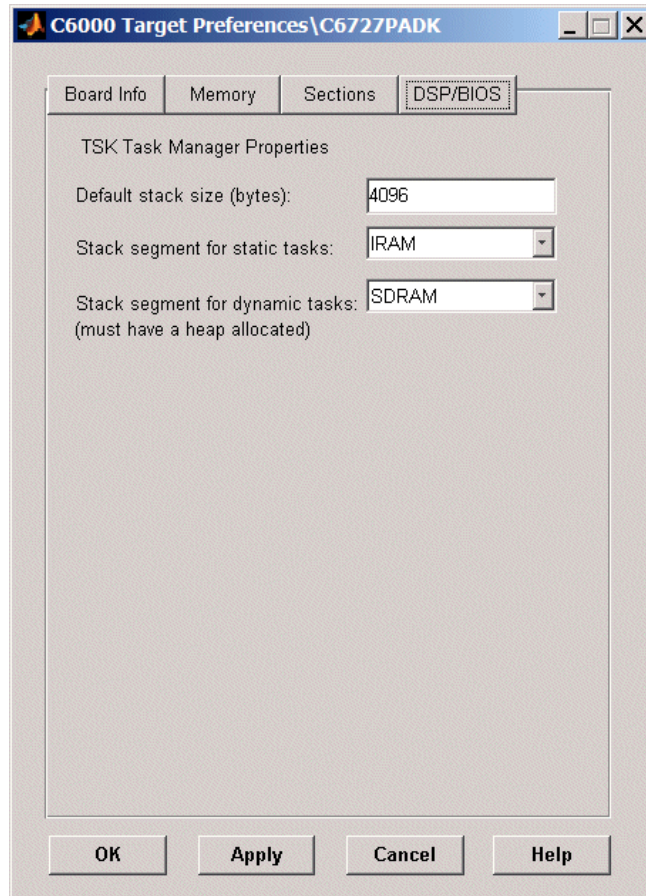
The asynchronous task scheduler uses these options when you select the **Incorporate DSP/BIOS** option in the model configuration set. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their block dialog boxes so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you clear the **Incorporate DSP/BIOS** option, you disable the options in this pane. Your project does not include DSP/BIOS tasks, and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler.

For more information about tasks, refer to the Code Composer Studio online help.



In the pane shown in this figure, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. The Sections pane includes DSP/BIOS configuration options as well. The options specify the stack use and locations on the stack for static and dynamic tasks.

### **Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the DSP/BIOS stack in bytes allocated for each task. A value 4096 bytes is the default. You can set any size up to the limits for the processor. Set the stack



size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, failure to set the stack size might cause the task to write into other memory or data areas, possibly causing unpredictable behavior.

**Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. However, infrequently used tasks usually work best as dynamic tasks.

The list offers options SDRAM and ISRAM for locating the stack in memory, with SDRAM as the default section. The Memory pane provide more options for the physical memory on the processor.

**Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, SDRAM is the only valid stack location in memory.

**See Also**

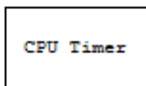
Custom C6000

# CPU Timer

**Purpose** Select timer and configure periodic interrupt

**Library** C6000 DSP Core Support Library in Embedded Target for TI C6000 DSP

## Description

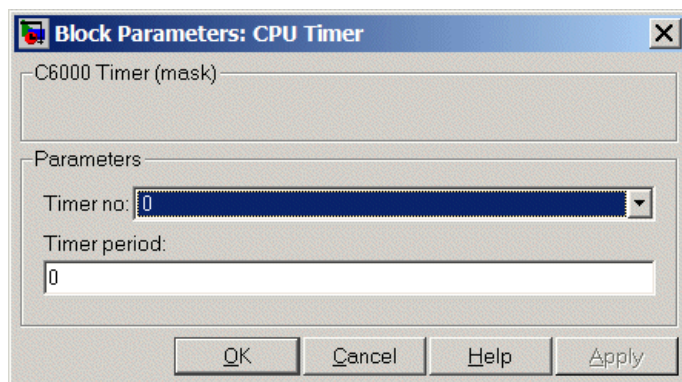


C6000 Timer

Use this block in a model to select the CPU timer on your board and specify a periodic interrupt. While the list provides two timers, 0 and 1, some boards offer either fewer or more timers. For example, the DM642 provides three timers.

CPU timer does not have input or output ports. Adding the block to your model serves to configure periodic interrupts in the generated code.

## Dialog Box



### Timer no.

Select the timer to use from the list. Be sure your 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. Use this block to configure the selected CPU timer to generate a periodic interrupt.

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**      Hardware Interrupt, Idle Task

# Custom C6000

---

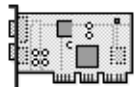
## Purpose

Configure model for C6000-processor-based custom hardware targets

## Library

Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

## Description



Custom C6000

Options on the block mask let you set features of code generation for your custom C6000 processor-based target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to custom hardware must include this block or the target preferences block that best matches your processor, such as the C6416DSK target preferences block to target custom hardware based on the C6416 processor. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

---

**Note** This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model. Simulink returns an error when your model does not include a target preferences block or has more than one.

---

The processor and target options you specify on this block are:

- Target board information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, DSP/BIOS, and custom sections

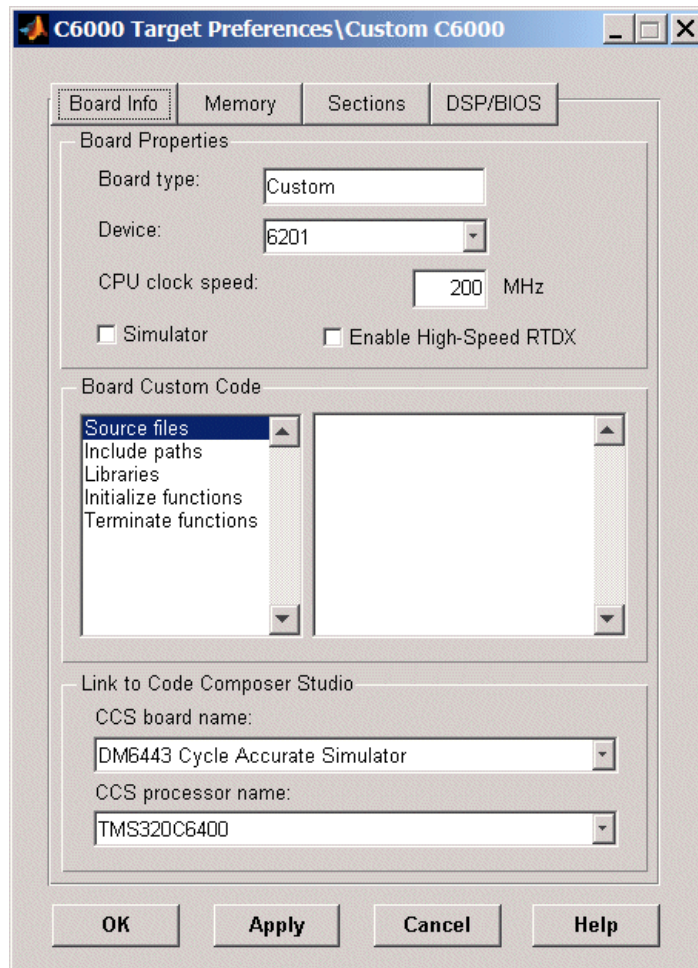
Setting the options included in this dialogbox results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both steps are essential for targeting any board that is custom or explicitly supported, such as the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

## Generating Code from Model Subsystems

Real-Time Workshop provides the ability to generate code from a selected subsystem in a model. To generate code for a custom C6000-based target from a subsystem, the subsystem model must include a Custom C6000 target preferences block.

### Dialog Box



All target preferences block dialog boxes provide tabbed access to panes the following panes with options you set for the target processor and target board:

- **Board info** — Select the target board and processor, set the clock speed, and identify the target.
- **Memory** — Set the memory allocation and layout on the target processor (memory mapping).
- **Sections** — Determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS and compiler information.
- **DSP/BIOS** — Specify how to configure tasking features of DSP/BIOS.

## Board Info Pane

The following options appear on the **Board Info** pane for the **C6000 Target Preferences** dialog box.

### Board Type

Lets you enter the type of board you are targeting with the model. You can enter **Custom** to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as **C6711DSK**. If you are using one of the explicitly supported boards, choose the target preferences block for that board and this field shows the proper board type.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model's real-time results may be wrong, and code profiling results are not correct.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if either of the following conditions occur:

- If your model does not include ADC or DAC blocks
- When the processing rates in your model change (the model is multirate)

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. You can change the rate with the DIP switches on the board or from one of the software utilities provided by Texas Instruments.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.00000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 1,000,000 \text{ clock ticks}$$

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

## **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

## **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for this model.

## Board Custom Code

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths.

enter files and file paths.

When you enter a path to a file, library, or other custom code, use the string

```
$(install_dir)
```

to refer to the CCS installation directory.

Enter new paths or files (custom code items) one to a line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

- **Source files** — you enter the full paths to source code files to use with this target. By default there are no entries in this parameter.
- **Include paths** — If you require additional files on your path, you add them by typing the path into the text area. The default setting does not include additional paths.
- **Libraries** — these entries identify specific libraries that the target requires. They appear on the list by default if required. Add more as you require by entering the full path to the library with the library file in the text area. No additional libraries appear here in the default configuration.
- **Initialize functions** — If your project requires an initialize function, enter it here. By default, this is empty.
- **Terminate functions** — enter a function to run when a program terminates. The default setting is not to include a specific termination function.

## CCS Board Name

Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one that you are targeting your code for.

## CCS Processor Name

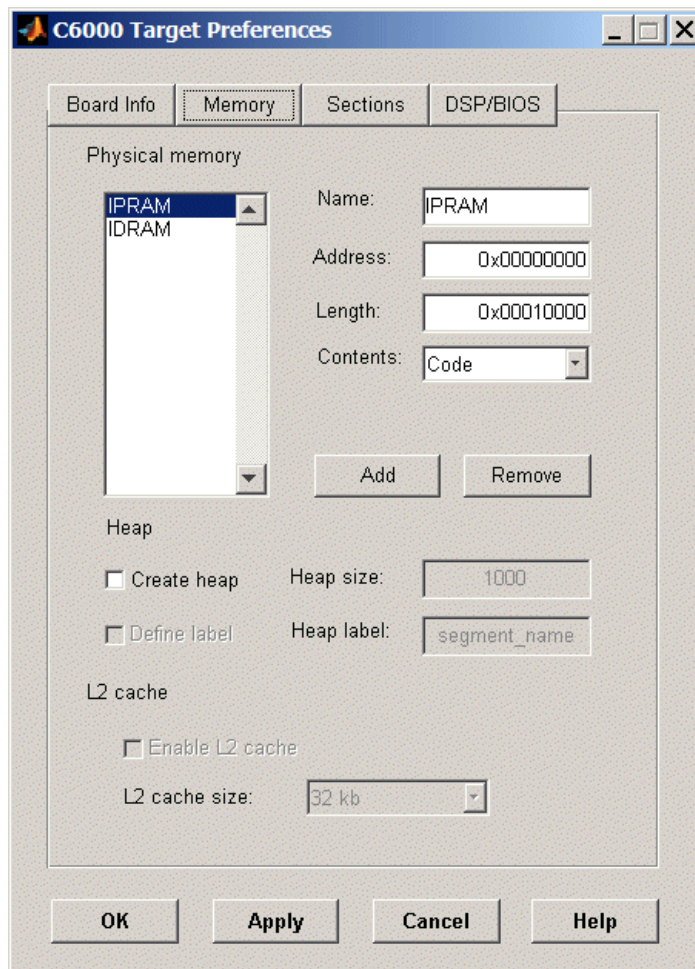
Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one



processor. In the multiprocessor case, you select the processor by name from the list.

## Memory Pane

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



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

- **Physical Memory** — specifies the processor and board memory map
- **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 may 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 online help system for Code Composer Studio.

## 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.
- C6711DSK boards provide SDRAM memory segments 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** details 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 may 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 may 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 on the **Physical memory** list and click **Remove** to delete the segment.

## Create Heap

If your processor supports using a heap, as does the C6711, 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

Enabled by selecting **Define label**, you use this option 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

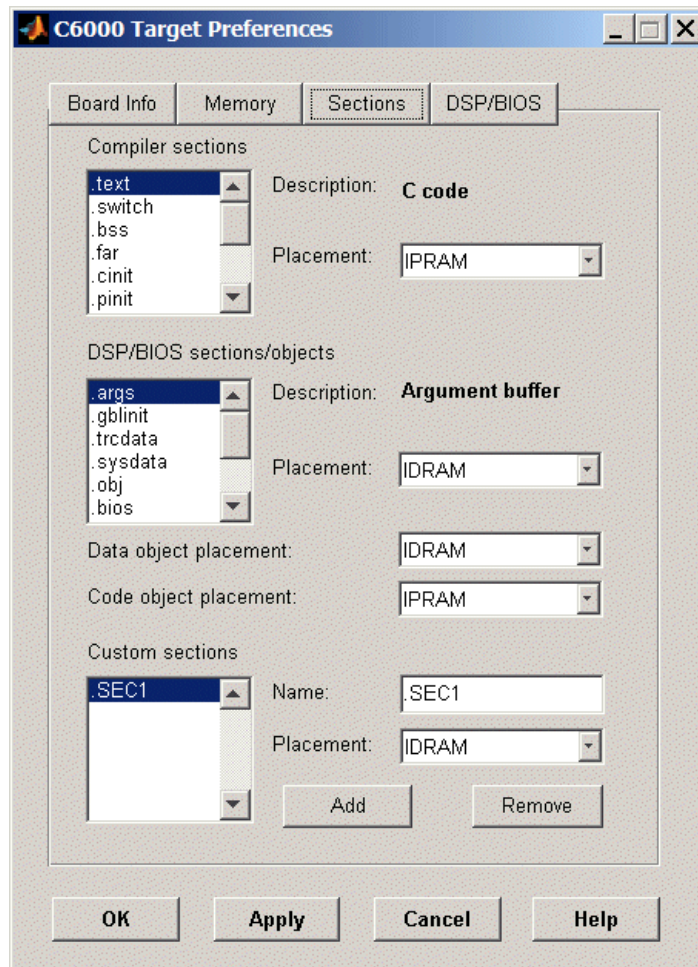
When you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

## 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. Commonly used sections include `.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 online help.

# Custom C6000



Within this pane, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections**, **DSP/BIOS sections/objects**, and **Custom sections** lists in the pane.

All sections do not appear on all lists. The list the string appears on is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rt dx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.system	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 get 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
- .system



Other sections appear on the list as well:

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

---

**Note** The C/C++ compiler does not use the .data 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 presently allocated in memory.

### **Description**

Provides a brief explanation of the contents of the selected entry on 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, get 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 location 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. You can also 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 and click **Remove**.

## DSP/BIOS Pane

Options on this pane let you specify how to configure tasking features of DSP/BIOS.

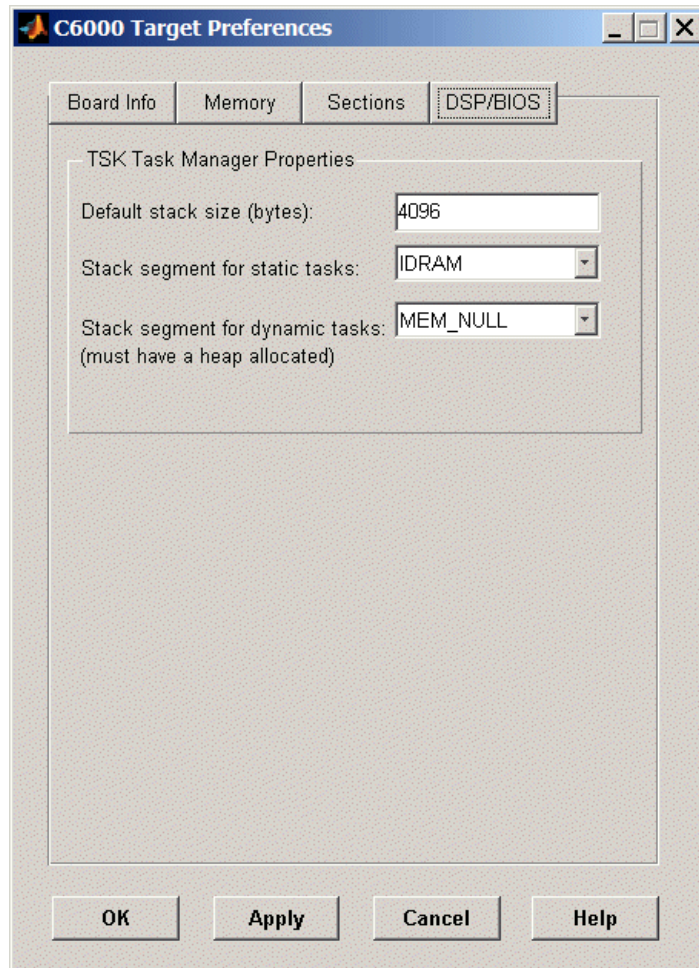
The asynchronous task scheduler uses these options when you select the **Incorporate DSP/BIOS** option in the model configuration set. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their block dialog boxes so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you clear the **Incorporate DSP/BIOS** option, you disable the options in this pane. Your project does not include DSP/BIOS tasks, and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler.

For more information about tasks, refer to the Code Composer Studio online help.



Within this pane, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. Note that the Sections pane includes DSP/BIOS configuration options as well. The options specify the stack use and locations on the stack for static and dynamic tasks.

### **Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the

DSP/BIOS stack in bytes allocated for each task. 4096 bytes is the default value. You can set any size up to the limits for the processor. Set the stack size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, this might cause the task to write into other memory or data areas, possibly causing unpredictable behavior.

### **Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers IDRAM for locating the stack in memory. The Memory pane provide more options for the physical memory on the processor.

### **Stack segment for dynamic tasks**

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, MEM\_NULL is the only valid stack location in memory.

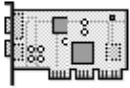
# DM642EVM

---

**Purpose** Configure model for DM642 Evaluation Module

**Library** Target Preferences in Embedded Target for TI C6000 DSP

## Description



DM642EVM

Options on the block mask let you set features of code generation for your DM642 Evaluation Module target. Adding this block to your Simulink model provides access to the processor hardware settings to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the DM642 evaluation module must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

---

**Note** This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

---

The processor and target options you specify on this block are:

- Target board information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, DSP/BIOS, and custom sections

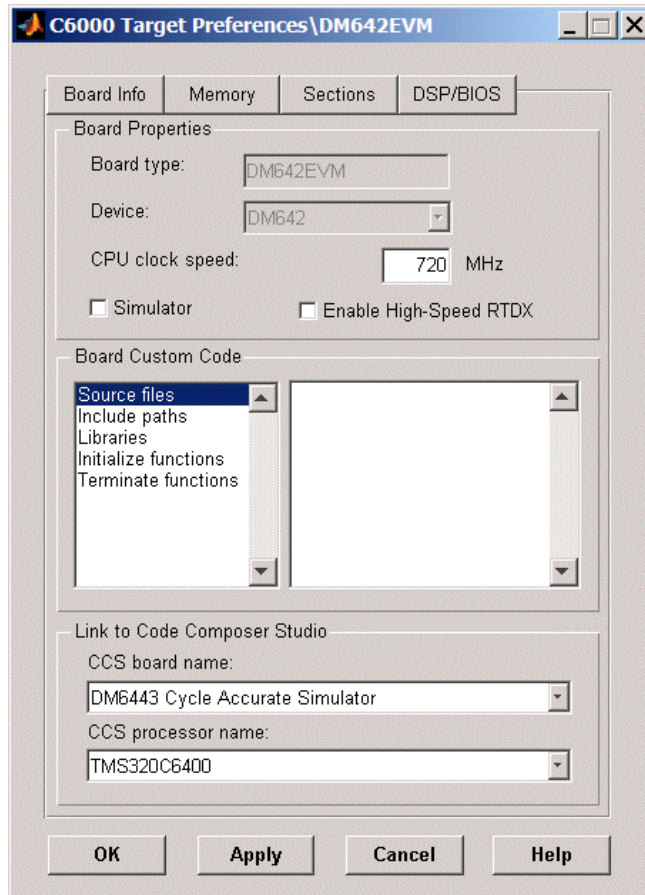
Setting the options included in this dialogbox results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both steps are essential for targeting any board that is custom or explicitly supported, such as the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

## Generating Code from Model Subsystems

Real-Time Workshop provides the ability to generate code from a selected subsystem in a model. To generate code for the DM642 EVM from a subsystem, the subsystem model must include a DM642EVM target preferences block.

### dialog box Box



All target preferences block dialog boxes provide tabbed access to panes the following panes with options you set for the target processor and target board:

- **Board info** — Select the target board and processor, set the clock speed, and identify the target.

- **Memory** — Set the memory allocation and layout on the target processor (memory mapping).
- **Sections** — Determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS and compiler information.
- **DSP/BIOS** — Specify how to configure tasking features of DSP/BIOS.

## Board Info Pane

The following options appear on the **Board Info** pane for the **C6000 Target Preferences** dialog box.

### Board Type

Lets you enter the type of board you are targeting with the model. You can enter **Custom** to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as **C6711DSK**. By default, the **DM642EVM** block specifies the **DM642EVM** for the board type.

### Device

Lets you select the type of processor on the board you select in **CCS board name**. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you are targeting one of the supported boards, **Device** is disabled and the selected device is fixed.

### CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model's real-time results may be wrong, and code profiling results are not correct.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for C6000 targets from Simulink models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if either of the following conditions occur:

- If your model does not include ADC or DAC blocks



- When the processing rates in your model change (the model is multirate)

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. You can change the rate with the DIP switches on the board or from one of the software utilities provided by Texas Instruments.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your target processor as you configured it. CPU clock speed lets you tell the timer the rate at which your target CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 KHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.00000001 s/sample

To create sine block interrupts at 0.001 s/sample requires

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 1,000,000 \text{ clock ticks}$$

So you must report the correct clock rate or the interrupts come at the wrong times and the results are incorrect.

### **Simulator**

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

### **Enable High-Speed RTDX**

Select this option to tell the code generation process to enable high-speed RTDX for this model.

### **Board Custom Code**

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths.

enter files and file paths.

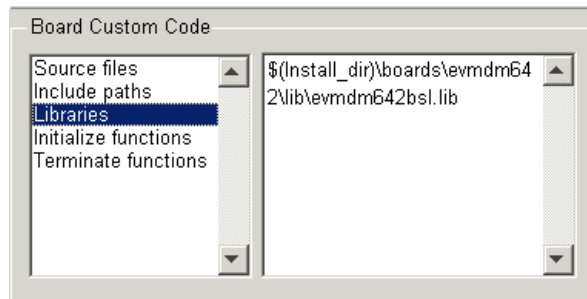
When you enter a path to a file, library, or other custom code, use the string

`$(install_dir)`

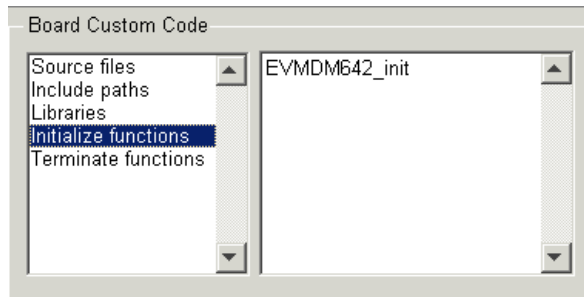
to refer to the CCS installation directory. The examples in the following figure use the string.

Enter new paths or files (custom code items) one to a line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

- **Source files** — you enter the full paths to source code files to use with this target. By default there are no entries in this parameter.
- **Include paths** — If you require additional files on your path, you add them by typing the path into the text area. The default setting does not include additional paths.
- **Libraries** — these entries identify specific libraries that the target requires. They appear on the list by default.



- **Initialize functions** — C6455 DSK targets require a specific initialization function, listed here as `DSK6455_init`. Enter others if needed.



- **Terminate functions** — enter a function to run when a program terminates. The default setting is not to include a specific termination function.

### CCS Board Name

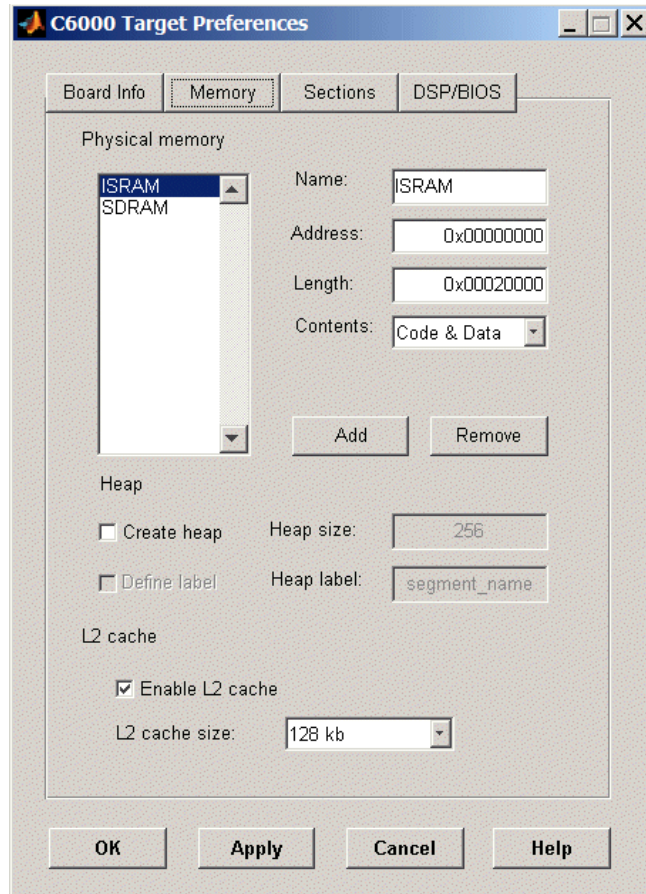
Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one that you are targeting your code for.

### CCS Processor Name

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.

### Memory Pane

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.



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

- **Physical Memory** — specifies the processor and board memory map
- **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 may 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 online help system for Code Composer Studio.

## 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. DM642EVM boards provide ISRAM and SDRAM memory segments 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** details 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 may 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 may 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 on the **Physical memory** list and click **Remove** to delete the segment.

## Create Heap

Selecting this option enables creating the heap, and enables the **Heap size** option.

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**

Enabled by selecting **Define label**, you use this option 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**

DM642 processors support an L2 cache memory structure that you can configure as ISRAM and partial cache.

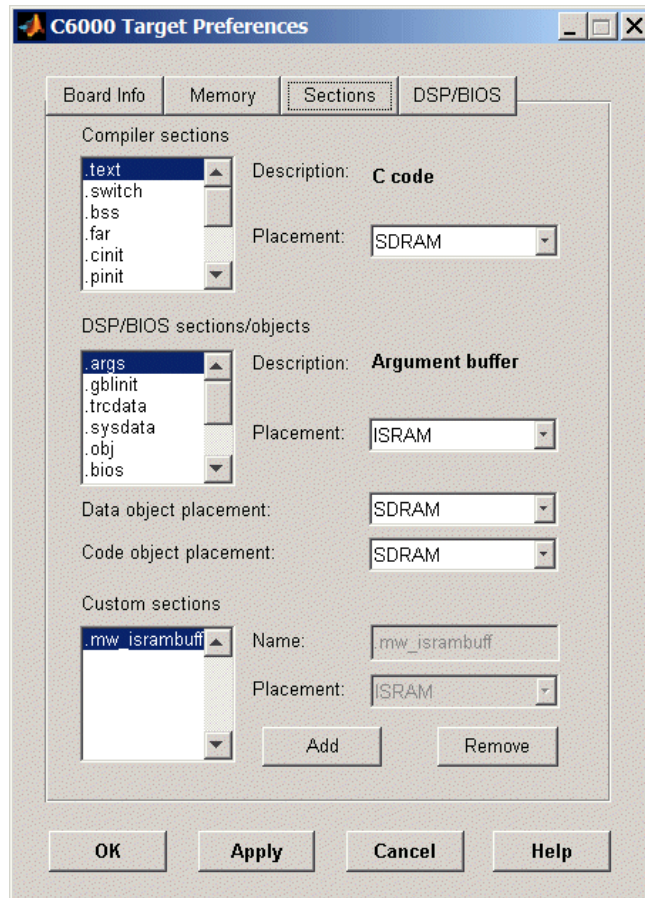
### **L2 Cache size**

When you enable the L2 cache, use this list to determine the size of the cache allotted. 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. Commonly used sections include `.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 online help.



Within this pane, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections**, **DSP/BIOS sections/objects**, and **Custom sections** lists in the pane.



All sections do not appear on all lists. The list the string appears on is shown in the table.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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.
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
.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
.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 get 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 the .data 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 presently allocated in memory.

#### **Description**

Provides a brief explanation of the contents of the selected entry on 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 ISRAM and SDRAM when you use this block.

#### **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, get 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 location 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. You can also 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 and click **Remove**.

## DSP/BIOS Pane

Options on this pane let you specify how to configure tasking features of DSP/BIOS.

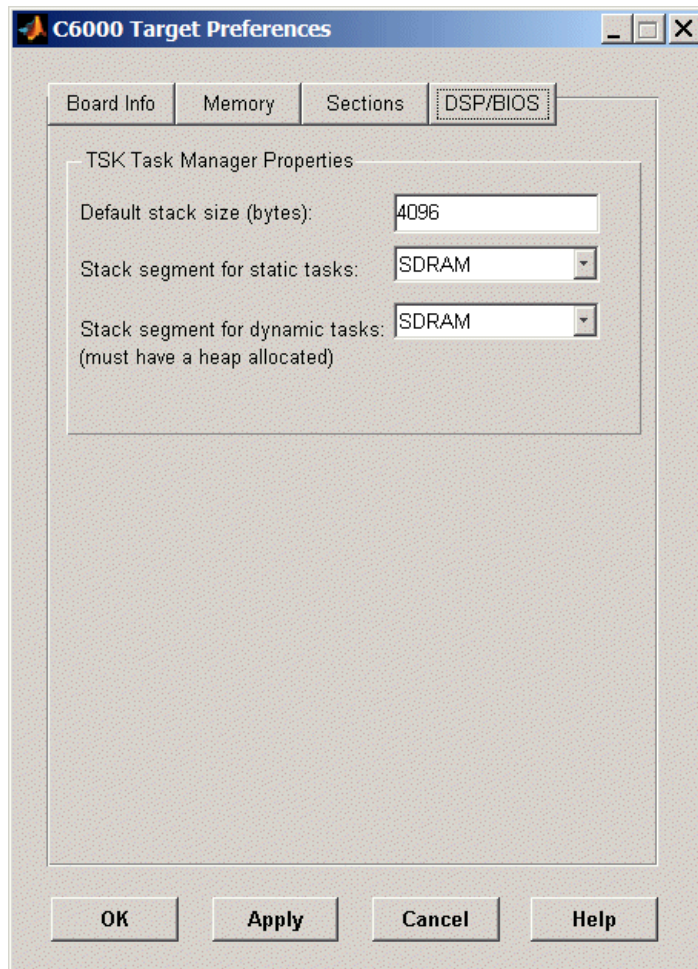
The asynchronous task scheduler uses these options when you select the **Incorporate DSP/BIOS** option in the model configuration set. By default, **Incorporate DSP/BIOS** is selected and the Embedded Target for TI C6000 DSP creates separate DSP/BIOS tasks for each sample time in your Simulink model.

DSP/BIOS tasking blocks provide parameters on their block dialog boxes so you can specify the DSP/BIOS stack size and stack segment (where the stack is in memory) for asynchronous tasks created by the DSP/BIOS Task and DSP/BIOS Triggered Task blocks.

The code generation process uses the options on this pane to configure TSK entries in the TSK Task Manager in CCS when it creates DSP/BIOS tasks.

When you clear the **Incorporate DSP/BIOS** option, you disable the options in this pane. Your project does not include DSP/BIOS tasks, and Embedded Target for TI C6000 DSP uses an interrupt-based scheduler.

For more information about tasks, refer to the Code Composer Studio online help.



Within this pane, you configure the options for DSP/BIOS tasks, such as the task manager and scheduler configuration. Note that the Sections pane includes DSP/BIOS configuration options as well. The options specify the stack use and locations on the stack for static and dynamic tasks.

### **Default stack size (bytes)**

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the

DSP/BIOS stack in bytes allocated for each task. 4096 bytes is the default value. You can set any size up to the limits for the processor. Set the stack size so that tasks do not use more memory than you allocate. While any task can use more memory than the stack includes, this might cause the task to write into other memory or data areas, possibly causing unpredictable behavior.

### **Stack segment for static tasks**

Use this option to specify where to allocate the stack for static tasks. Static tasks are created whether or not they are needed for operation, compared to dynamic tasks that the system creates as needed. Tasks that your program uses often might be good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers options SDRAM and ISRAM for locating the stack in memory, with SDRAM as the default section. The Memory pane provide more options for the physical memory on the processor.

### **Stack segment for dynamic tasks**

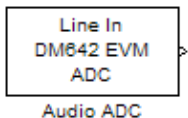
Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, SDRAM is the only valid stack location in memory.

# DM642 EVM Audio ADC

**Purpose** Audio codec and peripherals

**Library** DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

## 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.

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, 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
Output data type	TMS320DM642 digital signal processor
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.

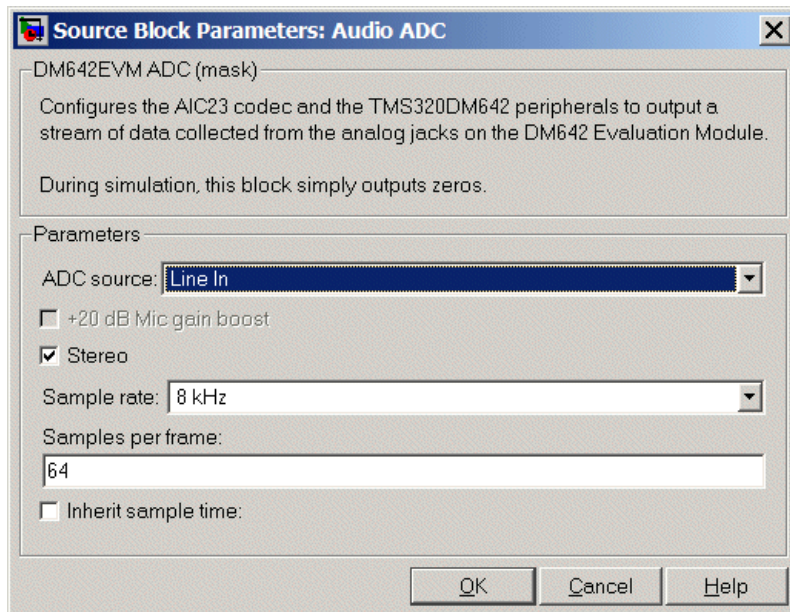
Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only,



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

# DM642 EVM Audio ADC

---

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. The default is 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. 64 samples per frame is the default setting. 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/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.

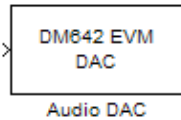
## **See Also**

DM642 EVM Audio DAC

**Purpose** Configure codec to convert digital audio input to analog audio output

**Library** DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

## Description

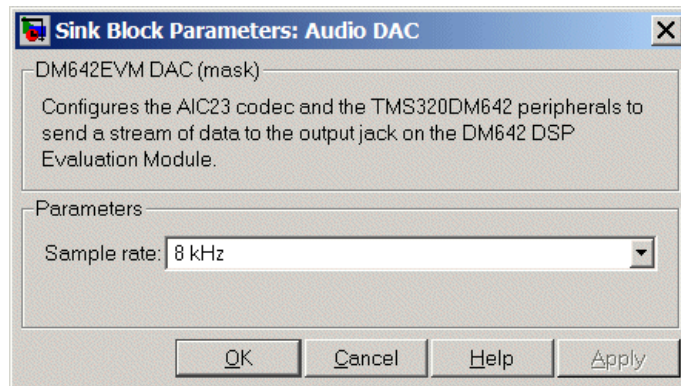


Adding the DM642 EVM DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the LINE OUT connection on the DM642 EVM mounting bracket. When you add the DM642 EVM DAC block, the digital signal received by the codec is converted to an analog signal. After converting the digital signal to analog form (digital-to-analog conversion), the codec sends the signal to the output audio jack.

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.

## 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

# DM642 EVM FPGA GPIO Read

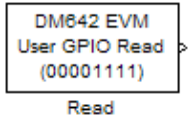
## Purpose

User GPIO registers to read from selected pins

## Library

DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

## Description

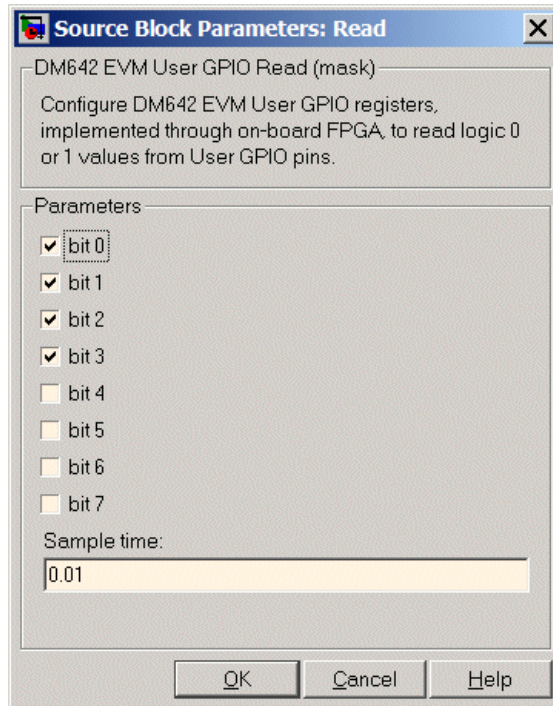


Added to your model, this block reads logical values from the GPIO registers you select in the dialog box and sends the data out to downstream blocks as an unsigned 8-bit word.

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.

## Dialog Box



## **bit 0 to bit 7**

Each bit represents the logical value at one GPIO register. **Bit 0** is register 0, **bit7** is register 7. Select the bits that represent the registers to read. Note that 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

# DM642 EVM FPGA GPIO Write

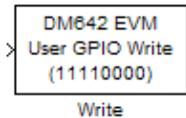
## Purpose

Write to GPIO registers

## Library

DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

## Description

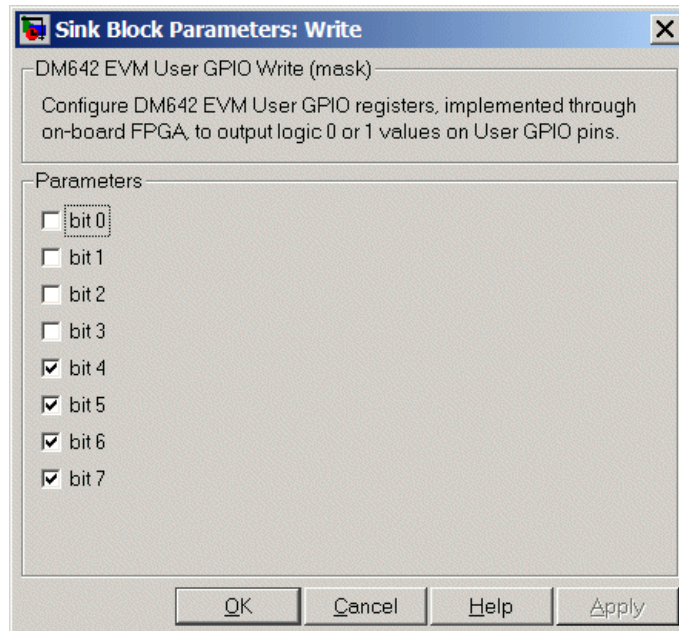


Added to your model, this block writes logical values to the GPIO registers you select in the dialog box, reading the data from an upstream block as an unsigned 8-bit word.

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



## **bit 0 to bit 7**

Each bit represents the logical value at one GPIO register. **Bit 0** is register 0, **bit7** is register 7. Select the bits that represent the registers to write. Note that the read and write functions cannot share the same registers. When you select a register to write to, you cannot read 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 Read

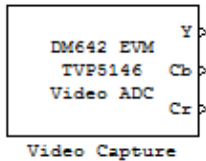
# DM642 EVM Video ADC

---

**Purpose** Video decoders to capture analog video

**Library** DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

**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) 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.

---

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.

## 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 your YCbCr 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.

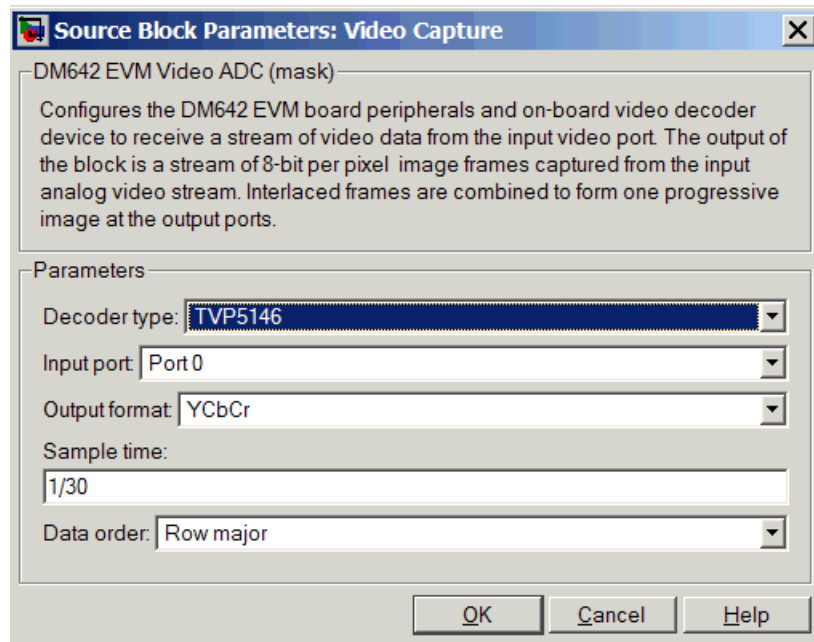
# DM642 EVM Video ADC

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 hand-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 (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 Revision” on page A-6 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.

## Output mode (available only when Decoder type is SAA7115)

Because all input video to the board is in analog NTSC 720-by-480 mode, this option scales the output from the block to different dimensions. Output modes for the block include various modes, as shown in this table. The important information in the table is the size of the images. Though the input to the block is always analog NTSC video, the output from the block uses the scaling capability of the video decoder to scale the digitized output image to one of the available sizes listed in the table

Digital Output Mode	Description
NTSC 720x480	Scales the output to higher definition TV mode.
NTSC 640x480	Scales the output to standard (SDTV) mode.
NTSC SIF 320x240	Scales the output to standard interchange format NTSC. Derived from CCIR 601 video (most often).
NTSC QCIF 176x144	Scales the output to 1/4 the resolution of CIF video.
NTSC SQCIF 128x96	Scales the output to 1/4 the resolution of QCIF video.

This option appears in the dialog box when you select SAA7115 for the **Decoder type**.

# DM642 EVM Video ADC

Web cameras, PAL format video, and S-Video inputs are not supported.

## 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	Output from the block includes one luminance channel Y (essentially the black/white signal) and two chrominance (color) channels Cb and Cr to represent the color image data per pixel. This is the digital standard color space DVDs use.
Y	Black/White video. No color/chromaticity values.

## Sample time

Tells the block how often to take frames from the video decoder and buffers. While NTSC video runs at 30 frames/s (1/30 s sample time), you can sample at any rate below or at the NTSC rate. When you enter a sample time shorter than 1/30 s (more than 30 frames per second), the block returns an error.

Remember that sampling times that are not 1/30th of a second may capture incomplete frames.

Also note that

- The sample time you specify becomes the DM642 timer period that drives the execution of your model.
- Your generated application is not synchronized with the input video signal — the application always runs on the processor timer.

## 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<sup>®</sup> and MATLAB both 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 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, 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.

## See Also

DM642 EVM Video DAC

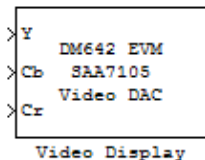
# DM642 EVM Video DAC

---

**Purpose** Video encoder to display video

**Library** DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

## 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.

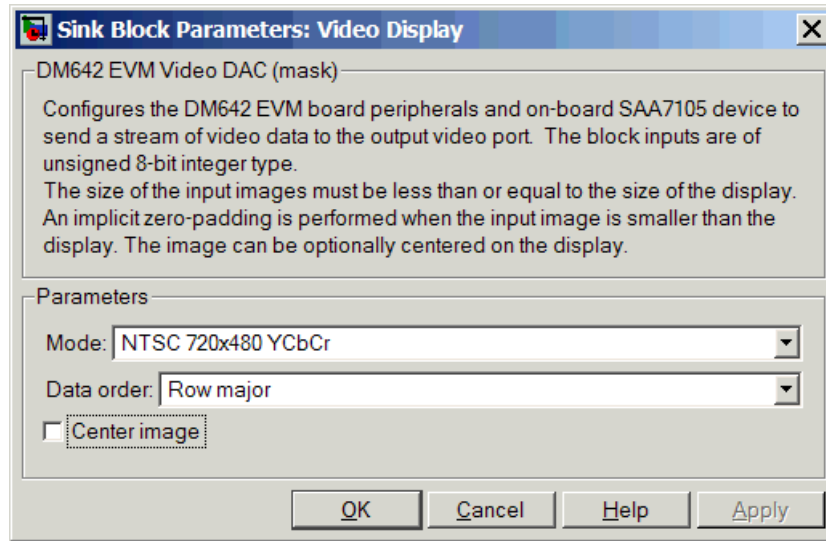
Unlike the DM642 EVM Video ADC block, this DAC block does not convert the video between formats. Nor does this block inherit any settings from the DM642 EVM Video ADC block, as some of the other C6000 DAC blocks do.

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.

## 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.

## 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 both 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 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. Note that 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

# DM642 EVM LED

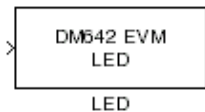
## Purpose

Control LEDs

## Library

DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

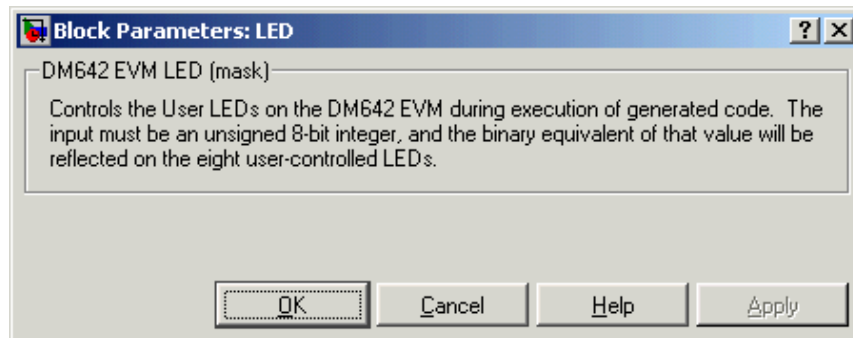
## Description



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



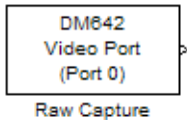
You see the block does not provide user options. Adding the block to your model adds the ability to control the LEDs.

**Purpose**

Video port to receive video data from video input port

**Library**

DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

**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.

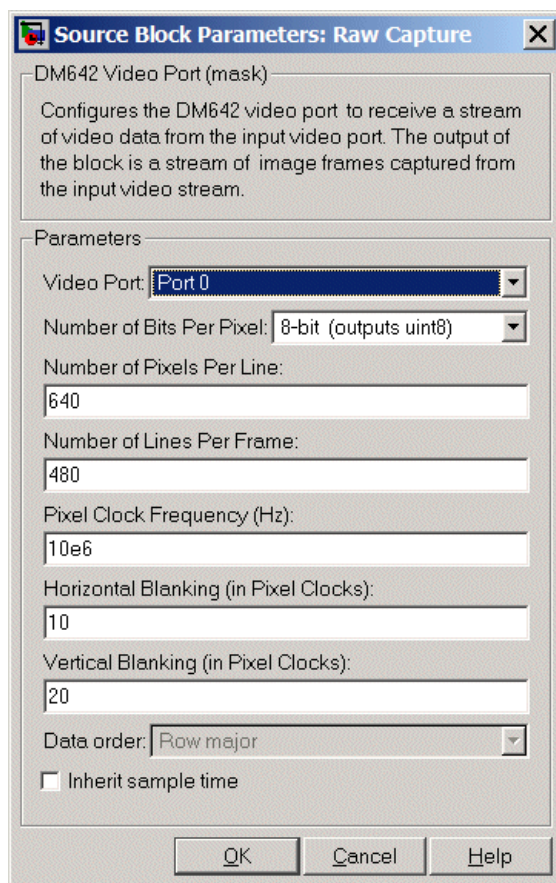
The block does not check your inputs to determine whether they form valid frames. You must be sure the values you assign work for you application.

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).

# DM642 EVM Video Port

## Dialog Box



### 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 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/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.

# DM642 EVM Video Port

---

## **See Also**

DM642 EVM Video ADC, DM642 EVM Video DAC

**Purpose**

Reset to initial conditions

**Library**

DM642 EVM Board Support in Embedded Target for TI C6000 DSP

**Description**

Reset

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 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.

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.

**Dialog Box**

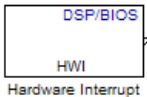
This block does not have settable options and does not provide a user interface dialog box.

# DSP/BIOS Hardware Interrupt

**Purpose** Generate Interrupt Service Routine

**Library** DSP/BIOS Library in Embedded Target for TI C6000 DSP

## 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.

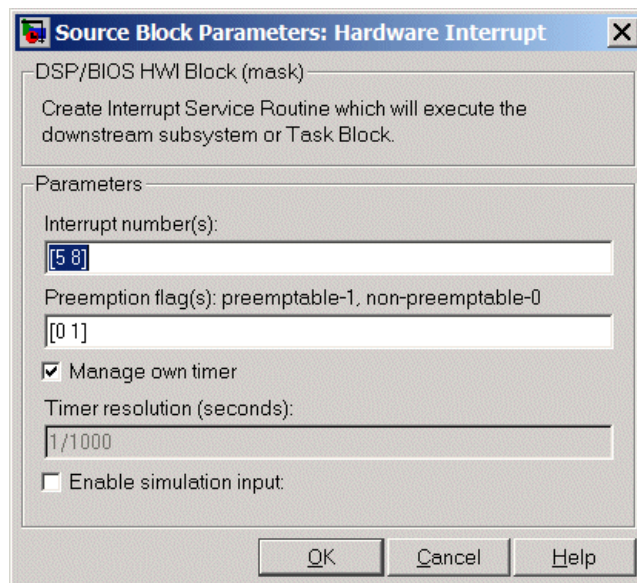
Interrupt Number	Default Event	Module
0	RESET	
1	NMI	
2	Reserved	
3	Reserved	
4	GPINT4	GPIO
5	GPINT5	GPIO
6	GPINT6	GPIO
7	GPINT7	GPIO
8	EDMAINT	EDMA
9	EMUDTDMA	Emulation
10	SDINT	EMIF
11	EMURTDXRX	Emulation
12	EMURTDXTX	Emulation
13	DSPINT	HPI



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.

# DSP/BIOS Hardware Interrupt

---

## 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. Connect interrupt simulation sources to the input. This option affects simulation only. It does not affect generated code.

## See Also

DSP/BIOS Task, DSP/BIOS Triggered Task

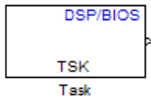
## Purpose

Create task that runs as separate DSP/BIOS thread

## Library

DSP/BIOS Library in Embedded Target for TI C6000 DSP

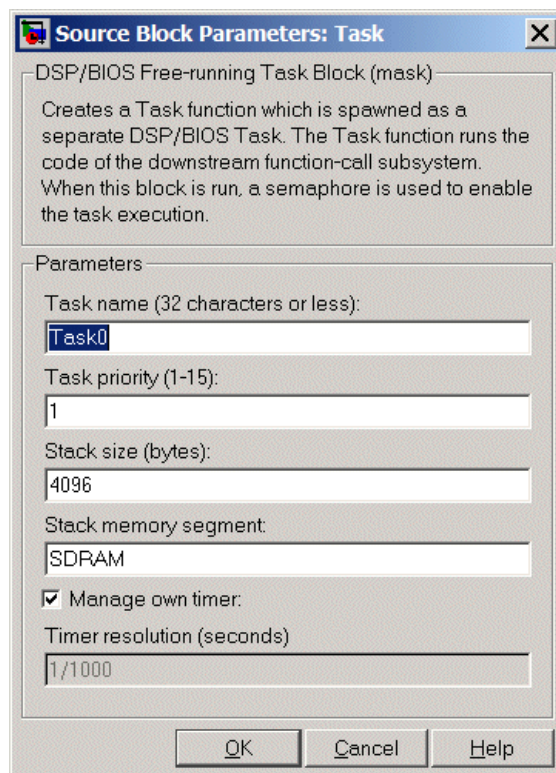
## Description



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



# 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 / or : 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 default value is 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.

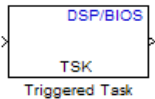
## **See Also**

DSP/BIOS Hardware Interrupt, DSP/BIOS Triggered Task

**Purpose** Create asynchronously triggered task

**Library** DSP/BIOS Library in Embedded Target for TI C6000 DSP

## 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):  
Task0

Task priority (1-15):  
8

Stack size (bytes):  
4096

Stack memory segment:  
SDRAM

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 Hardware Interrupt block) prevents preempting the task.

## **Stack size (bytes)**

Specify the size of the stack the task uses. The default value is 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

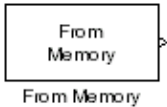
## Purpose

Get data from a specific memory location into your code running on the C6000 target

## Library

C6000 DSP Core Support in Embedded Target for TI C6000 DSP for TI DSP

## Description



**Note** This block will be removed in the future. Please use the Memory Allocate and Memory Copy blocks instead.

When you generate code from your Simulink model in Real-Time Workshop with this block in place, code generation inserts the C commands to create a read process that gets data from memory on the target. The inserted code reads the specified memory location in **Memory address** and returns the data stored there. Any valid memory location on the target works with the block.

When you look at your generated code, you find lines of code like the following that represent the From Memory block operation:

```
/* S-Function Block: <Root>/From Memory (c6000mem_src) */
{
    /* Memory Mapped Input */
    rtB.From_Memory = (real_T)((volatile int *) (4096U));
}
```

In simulations this block does not perform any operations, with the exception that the block does output port checking. From Memory blocks work only in code generation and when your model runs on your target.

## Using From Memory Blocks

Be careful when you use From Memory blocks in your models in combination with To Memory blocks. Because the To Memory blocks give you control over where the target stores information in memory, pay attention to how you use the From Memory block to retrieve data from memory. You can return data that is not what you expect.

Using the From Memory block itself does not cause problems in generated code on your target.

When you use the options in the To Memory block to specify where the project writes data in memory, you might be writing to memory locations that are

# From Memory

---

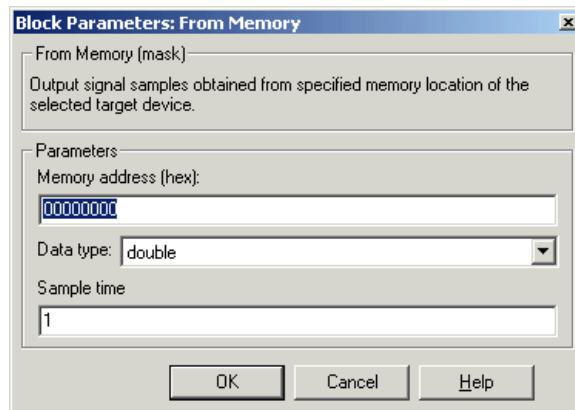
reserved for the compiler or for other uses. Reading from those locations could return the wrong answer.

To prevent your model from encountering memory errors like these, generate your code once without loading the COFF file to the target. Look at the generated file *projectname.map*, where *projectname* is the name of your project, to see the memory range that the compiler uses.

From this list of allocated memory, determine the memory ranges that the compiler uses and the locations of free memory or the memory to read with your From Memory block. Determine the memory locations from which to read your data from the .map file listings.

You should examine the .map file for your project each time you change the Simulink model associated with your project.

## Dialog Box



### Memory address (hex)

Enter the address of the memory location that contains the data to return. Note that you do not need to start the address with 0x to indicate that it is hexadecimal.

### Data type

Sets the type for the data coming from the block. Select one of the following types:



- `double` — double-precision floating-point values. This is the default setting.
- `single` — single-precision floating-point values.
- `uint8` — 8-bit unsigned integers. Output values range from 0 to 255.
- `int16` — 16-bit signed integers. With the sign, the values range from -32768 to 32767.
- `int32` — 32-bit signed integers. Values range from  $-2^{31}$  to  $(2^{31}-1)$ .

### **Sample time**

Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second ( $1/\mathbf{Sample\ time}$ ).

### **See Also**

To Memory

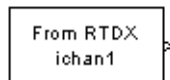
# From Rtdx

---

**Purpose** Add RTDX communication channel to send data from MATLAB to target

**Library** RTDX Instrumentation in Embedded Target for TI C6000 DSP for TI DSP

## Description



When you generate code from your Simulink model in Real-Time Workshop with this block in place, code generation inserts the C commands to create an RTDX input channel on the target. The inserted code opens and enables the channel with the name you specify in **Channel name** in the block parameters. You can open, close, disable, and enable the channel from the host side afterwards, overriding the target side status.

In the generated code, you see a command like the following

```
RTDX_enableInput(&channelname)
```

where `channelname` is the name you enter in **Channel name**.

In simulations this block does not perform any operations with the exception that the block will generate an output matching your specified initial conditions. From Rtdx blocks work only in code generation and when your model runs on your target.

If you are using the Link for Code Composer Studio Development Tools, you need to configure and cleanup RTDX properly before and after executing your model or code. Refer to the RTDX tutorials in the Link for Code Composer Studio documentation in the online Help system to see an example of how to do this housekeeping task.

The initial conditions you set in the block parameters determine the output from the block to the target for the first read attempt. Specify the initial conditions in one of the following ways:

- Scalar value — the block generates one output sample with the value of the scalar. For a value of 0, the block outputs a zero to the processor. When **Output dimension** specifies an array, every element in the array has the same scalar value.
- Null array ([ ]) — same output as a scalar with the value zero for every sample.

Using RTDX in your model involves:

- Adding one or more To Rtdx or From Rtdx blocks to your model to prepare your target
- Downloading and running your model on your target
- Enabling the RTDX channels from MATLAB or using **Enable RTDX channel on start-up** on the block dialog box
- Using the `readmsg` and `writemsg` functions in MATLAB to send and retrieve data from the target over RTDX

To see more details about using RTDX in your model, refer to [Using Links in your Link for Code Composer Studio Development Tools](#) documentation in the online Help system.

## Dialog Box

**Block Parameters: From RTDX**

From RTDX (mask)  
Use specified RTDX channel to send data from host to target DSP. In blocking mode, the DSP waits for new data from the block. In non-blocking mode, the DSP uses previous data when new data is not available from the block.

Parameters

Channel name  
ichan1

Enable blocking mode

Initial conditions:  
0

Sample Time  
1

Output dimensions  
[1 64]

Frame-based

Data type: double

Enable RTDX channel on start-up

OK Cancel Help Apply

### Channel name

Defines the name of the input channel to be created by the generated code. Recall that input channels refer to transferring data from the host to the

target (input to the target). To use this RTDX channel, you enable and open the channel with the name, and send data from the host to the target across this channel. Specify any name as long as it meets C syntax requirements for length and character content.

## **Enable blocking mode**

Puts RTDX communications into blocking mode where the target processor waits to continue processing until new data is available from the From Rtdx block. Selecting blocking mode slows your processing while the processor waits — if your new data is not available when the processor needs it, your process stops. In non blocking mode, the processor uses old data from the block when new data is not available. Non blocking operation is the default and recommended for most operations.

Selecting the **Blocking** option disables the **Initial conditions** option.

## **Initial conditions**

Specifies what data the processor reads from RTDX for the first read. This can be 0, null ([ ]), or a scalar. You must have an entry for this option. Leaving the option blank causes an error in Real-Time Workshop.

## **Sample time**

Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

## **Output dimensions**

Defines a matrix for the output signal from the block, where the first value is the number of rows and the second is the number of columns in the output matrix. For example, the default setting [1 64] represents a 1-by-64 matrix of output values. Enter a 1-by-2 vector of doubles for the dimensions.

## **Frame-based**

Sets a flag at the block output that directs downstream blocks to use frame-based processing on the data from this block. In frame-based processing, the samples in a frame are processed simultaneously. In sample-based processing, samples are processed one at a time. Frame-based processing can greatly increase the speed of your application running on your target. Note that throughput remains the same in samples per second processed. Frame-based operation is the default.

## Data type

Sets the type for the data coming from the block. Select one of the following types:

- **Double** — double-precision floating-point values. This is the default setting. Values range from -1 to 1.
- **Single** — single-precision floating-point values ranging from -1 to 1.
- **Uint8** — 8-bit unsigned integers. Output values range from 0 to 255.
- **Int16** — 16-bit signed integers. With the sign, the values range from -32768 to 32767.
- **Int32** — 32-bit signed integers. Values range from  $-2^{31}$  to  $(2^{31}-1)$ .

## Enable RTDX channel on start-up

When your application code includes RTDX channel definitions, selecting this option enables the channels when you start the channel from MATLAB. With this selected, you do not need to use the Link for Code Composer Studio Development Tools `enable` function to prepare your RTDX channels. Note that the option applies only to the channel you specify in **Channel name**. You do have to open the channel.

## See Also

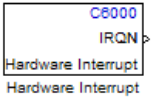
`cccdsp`, `readmsg`, `To Rtdx`, `writemsg`

# Hardware Interrupt

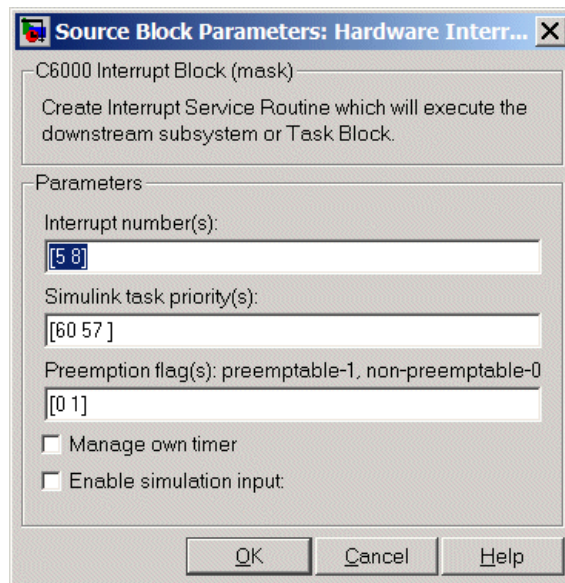
**Purpose** Generate Interrupt Service Routine

**Library** C6000 DSP Core Support in Embedded Target for TI C6000 DSP

**Description** Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the target that run the processes that are downstream from the this block or a Task block connected to this block.



## Dialog Box



### Interrupt Number(s)

Specify an array of interrupt numbers for the interrupts to install. The valid range is 1 to 15.

The width of the block output signal corresponds to the number of interrupt numbers specified here. Combined with the **Simulink task priority(s)** you enter and the **preemption flag** you enter for each interrupt, these three values define how the code and target process interrupts during asynchronous scheduler operations.

## Simulink task priority(s)

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink task priority specifies the Simulink priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt number(s)**.

Simulink task priority values are required to generate the proper rate transition code (see Rate Transitions and Asynchronous Blocks). The task priority values are also required to ensure absolute time integrity when the asynchronous task needs to obtain real time from its base rate or its caller. Typically, you assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

## 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 lets you set the timer resolution the ISR uses.

## Enable simulation input

When you select this option, Simulink adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

# Idle Task

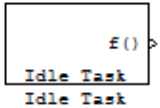
## Purpose

Create free-running task

## Library

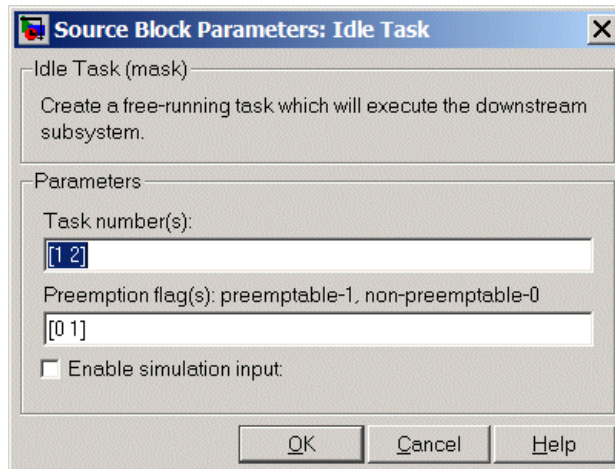
C6000 DSP Core Support in Embedded Target for TI C6000 DSP

## Description



Create a task that runs during DSP/BIOS idle loop processing. Tasks assigned in this block run the downstream subsystems while the processor is in the idle loop.

## Dialog Box



### Task number(s)

Identifies the created tasks by number. Enter as many tasks as you need by entering a vector of integers.

### 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 **Task number(s)** contains more than one task, you can assign different preemption flags to each task by entering a vector of flag values, corresponding to the order of



the tasks in **Task number(s)**. If **Task number(s)** contains more than one task, and you enter only one flag value here, that status applies to all tasks.

In the default settings [0 1], the task with priority 5 in **Task number(s)** is not preemptible and the priority 8 task can be preempted.

### **Enable simulation input**

When you select this option, Simulink adds an input port to the Idle Task block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

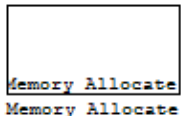
# Memory Allocate

---

**Purpose** Allocate memory section on C6000 target

**Library** C6000 DSP Core Support in Embedded Target for TI C6000 DSP

## Description



On your C6000 target, this block directs the TI compiler to allocate memory for a new variable you specify. Parameters in the block dialog box let you specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.

The block does not verify whether the entries for your variable are valid, such as checking the variable name, data type, or section. You must ensure that all variable names are valid, that they use valid data types, and that all section names you specify are valid as well.

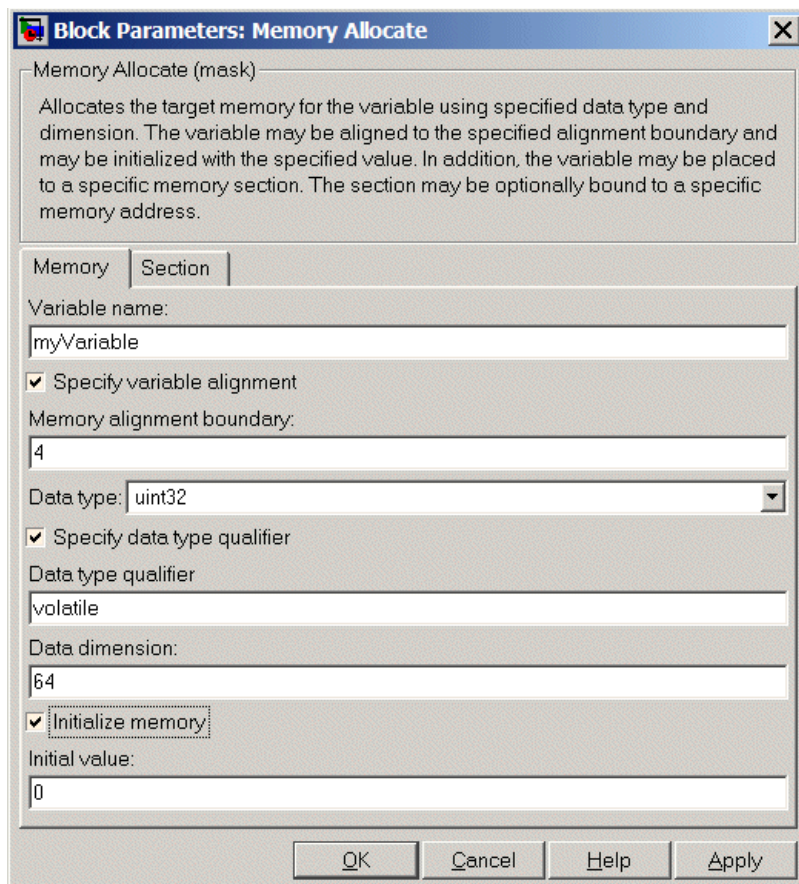
Notice that the block does not have input or output ports. It serves only to allocate a memory location. You do not connect it to other blocks in your model.

## Dialog Box

The block dialog box comprises multiple tabs:

- **Memory** — Allocate the memory for storing variables. Specify the data type and size.
- **Section** — Specify the memory section in which to allocate the variable.

Note that 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.



Sections below describe the contents of each tab in the dialog box.

# Memory Allocate

## Memory Parameters

**Block Parameters: Memory Allocate**

Memory Allocate (mask)

Allocates the target memory for the variable using specified data type and dimension. The variable may be aligned to the specified alignment boundary and may be initialized with the specified value. In addition, the variable may be placed to a specific memory section. The section may be optionally bound to a specific memory address.

Memory | Section

Variable name:  
myVariable

Specify variable alignment

Memory alignment boundary:  
4

Data type: uint32

Specify data type qualifier

Data type qualifier  
volatile

Data dimension:  
64

Initialize memory

Initial value:  
0

OK Cancel Help Apply

You find the following memory parameters on this tab.

### Variable name

Specify the name of the variable to allocate. The variable will be allocated in the generated code.

### Specify variable alignment

Select this option to direct the compiler to align the variable in **Variable name** to an alignment boundary. When you select this option, the **Memory**

**alignment boundary** parameter appears so you can specify the alignment. Use this parameter and **Memory alignment boundary** when your process requires this feature.

## **Memory alignment boundary**

After you select **Specify variable alignment**, this option enables you to specify the alignment boundary in bytes. If your variable contains more than one value, such as a vector or an array, the elements are aligned according to rules applied by the compiler.

## **Data type**

Defines the data type for the variable. Select from the list of types available.

## **Specify data type qualifier**

Selecting this enables **Data type qualifier** so you can specify the qualifier to apply to your variable.

## **Data type qualifier**

After you select **Specify data type qualifier**, you enter the desired qualifier here. `volatile` is the default qualifier. Enter the qualifier you need as text. Common qualifiers are `static` and `register`. The block does not check for valid qualifiers.

## **Data dimension**

Specifies the number of elements of the type you specify in **Data type**. Enter an integer here for the number of elements.

## **Initialize memory**

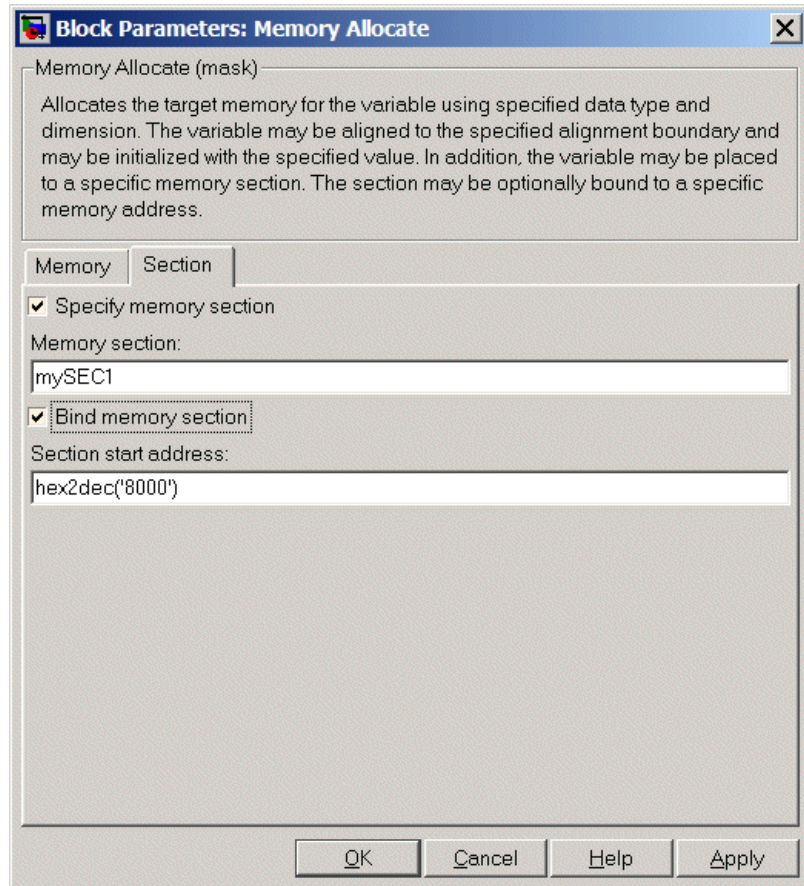
Directs the block to initialize the memory location to a fixed value before processing.

## **Initial value**

Specifies the initialization value for the variable. At run time, the block sets the memory location to this value.

# Memory Allocate

## Section Parameters



Parameters on this tab relate to specifying the section in memory to store the variable.

### Specify memory section

Selecting this parameter enables you to specify the memory section to allocate space for the variable. Enter either one of the standard memory sections or a custom section that you declare elsewhere in your code.

## Memory section

Identify a specific memory section to allocate the variable in **Variable name**. You must be sure the section has sufficient space to store your variable.

## Bind memory section

After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use this parameter to bind the memory section to the location in memory specified in **Section start address**. When you select this, you enable the **Section start address** parameter.

Note that the new memory section (specified in **Memory section**) is defined when you check this parameter. Do not use **Bind memory section** for existing memory sections.

## Section start address

Specify the address to which to bind the memory section. Enter the address in decimal form or in hexadecimal with a conversion to decimal as shown by the default value `hex2dec('8000')`. The block does not verify the address — you must be sure the address exists and can contain the memory section you entered in **Memory section**.

## See Also

Memory Copy

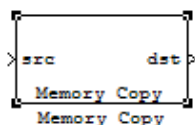
# Memory Copy

---

**Purpose** Copy to and from memory section

**Library** C6000 DSP Core Support in Embedded Target for TI C6000 DSP

## Description



In generated code, this block copies variables or data from and to target memory as configured by the block parameters. Your model can contain as many of these blocks as you require to manipulate memory on your target.

Each block works with one variable, address, or set of addresses provided to the block. Parameters for the block let you specify both the source and destination for the memory copy, as well as options for initializing the memory locations.

Using parameters provided by the block, you can change options like the memory stride and offset at run time. In addition, by selecting various parameters in the block, you can write to memory at program initialization, at program termination, and at every sample time. The initialization process occurs once, not for every read and write operation.

With the custom source code options, the block enables you to add custom C source code before and after each memory read and write (copy) operation. One use for the custom code capability would be to lock and unlock registers before and after accessing them. For example, some processors have registers that you may need to unlock and lock with `EALLOW` and `EDIS` macros before and after your program accesses them.

If your processor or target supports quick direct memory access (QDMA) the block provides a parameter to check that implements the QDMA copy operation, and provides you the ability to specify a function call that can indicate that the QDMA copy is finished. Only the C621x, C64xx, and C671x processor families support QDMA copy.

### Block Operations

This block performs operations at three periods during program execution — initialization, real-time operations, and termination. With the options for setting memory initialization and termination, you control when and how the block initializes memory, copies to and from memory, and terminates memory operations. The parameters enable you to turn on and off memory operations in all three periods independently.



Used in combination with the Memory Allocate block, this block supports building custom device drivers, such as PCI bus drivers or codec-style drivers, by letting you manipulate and allocate memory. Note that this block does not require the Memory Allocate block to be in the model.

In a simulation, this block does not perform any operation. The block output is not defined.

## Copying Memory

When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

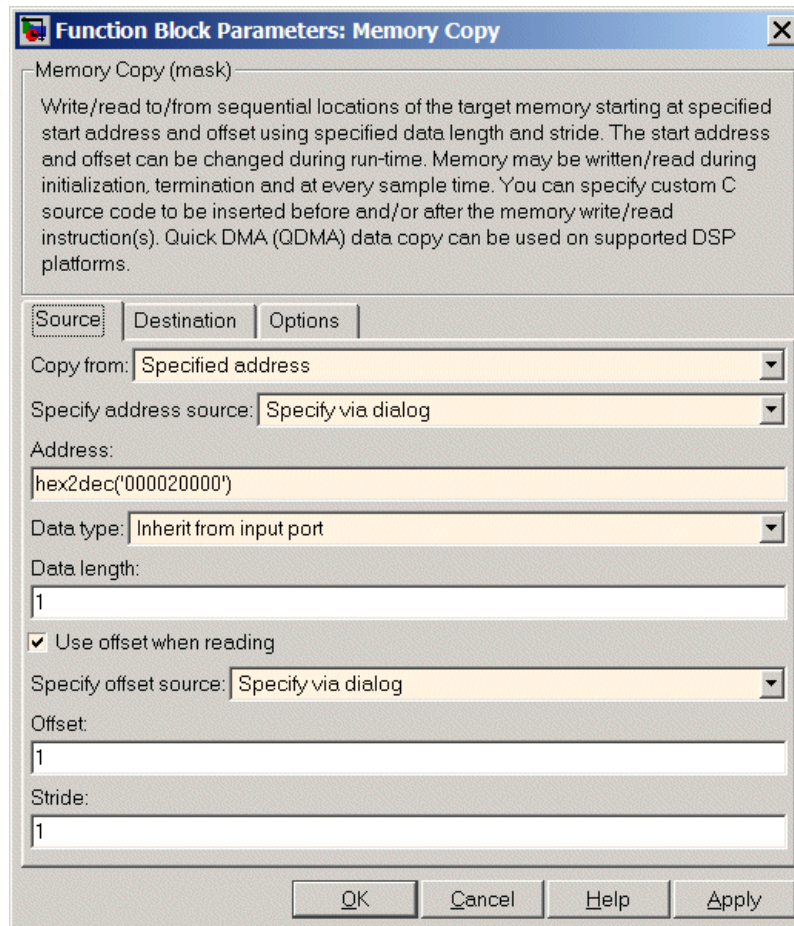
### Dialog Box

The block dialog box contains multiple tabs:

- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various parameters to control the copy process.

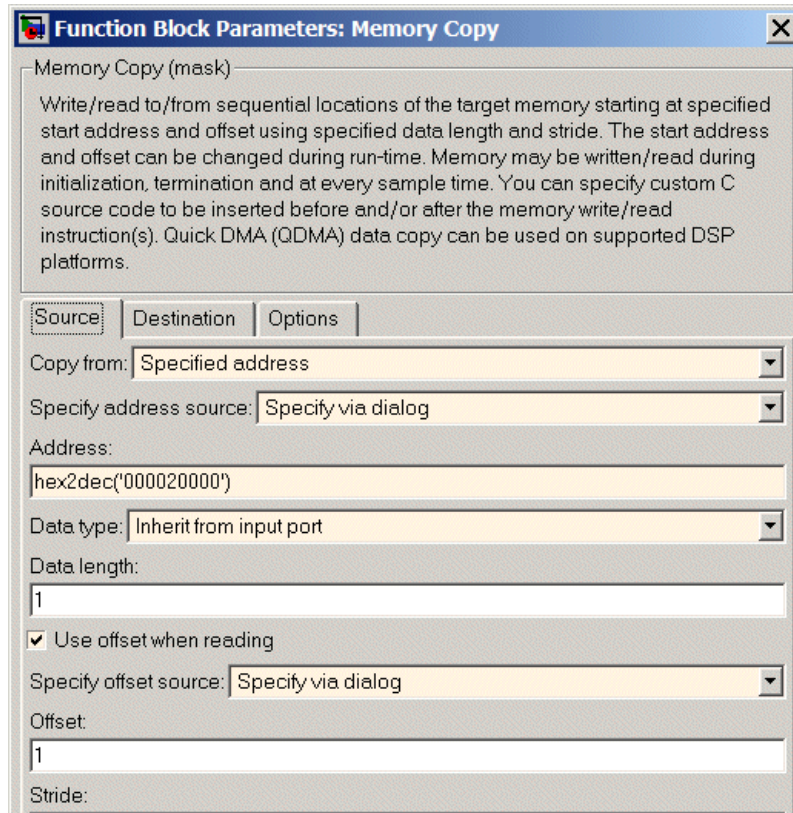
Note that the dialog box images show many of the available parameters enabled. Some parameters shown do not appear until you select one or more other parameters. Some parameters are not shown in the figures, but the text describes them and how to make them available.

# Memory Copy



Sections that follow describe the parameters on each tab in the dialog box.

## Source Parameters



### Copy from

Select the source of the data to copy. Choose one of the entries on the list:

- Input port — this reads the data from the block input port.
- Specified address — this reads the data at the specified location in **Specify address source** and **Address**.
- Specified source code symbol — tells the block to read the symbol (variable) you enter in **Source code symbol**. When you select this copy from option, you enable the **Source code symbol** parameter.

# Memory Copy

---

---

**Note** If you do not select the Input port option for **Copy from**, you must change the **Data type** parameter setting from the default **Inherit from input port** to one of the data types on the **Data type** list. If you do not make the change, you receive an error message that the data type cannot be inherited because the input port does not exist.

---

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

## Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either **Specify via dialog** or **Input port** from the list. Selecting **Specify via dialog** activates the **Address** parameter for you to enter the address for the variable.

When you select **Input port**, the port label on the block changes to **&src**, indicating that the block expects the address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

## Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax. Enter a string to specify the symbol exactly as you use it in your code.

## Address

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use **hex2dec** to convert the address to the proper format. Here is one example that converts **0x1000** to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you could enter either 4096 or `hex2dec('1000')` as the address.

## Data type

Use this parameter to specify the type of data that your source uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `Inherit` from `input port` for inheriting the data type for the variable from the block `input port`.

## Data length

Specifies the number of elements to copy from the source location. Each element has the data type specified in **Data type**.

## Use offset when reading

When you are reading the input, use this parameter to specify an offset for the input read. The offset value is in elements with the assigned data type. The **Specify offset source** parameter becomes available when you check this option.

## Specify offset source

The block provides two sources for the offset — `Input port` and `Specify via dialog`. Selecting `Input port` configures the block `input` to read the offset value by adding an `input port` labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when reading the input data.

## Offset

**Offset** tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the default value and **Offset** accepts only positive integers of one or greater.

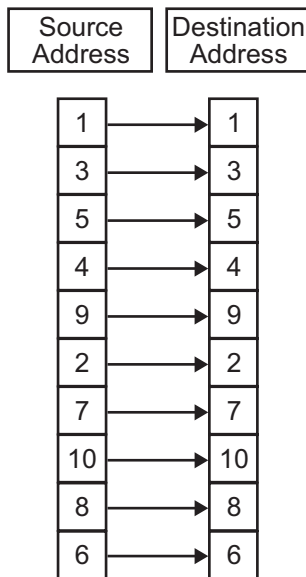
## Stride

Stride lets you specify the spacing for reading the input. By default, the stride value is one, meaning the generated code reads the input data sequentially. When you add a stride value that is not equal to one, the block reads the input data elements not sequentially, but by skipping spaces in

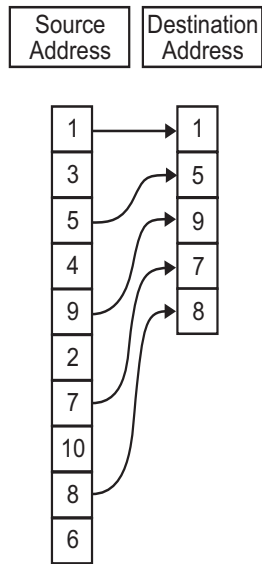
# Memory Copy

the source address equal to the stride. **Stride** must be a scalar with positive integer value of one or greater.

The next two figures help explain the stride concept. In the first figure you see data copied without any stride. Following that figure, the second figure shows a stride value of two applied to reading the input when the block is copying the input to an output location. You can specify a stride value for the output with parameter **Stride** on the **Destination** pane. Compare stride with offset to see the differences.



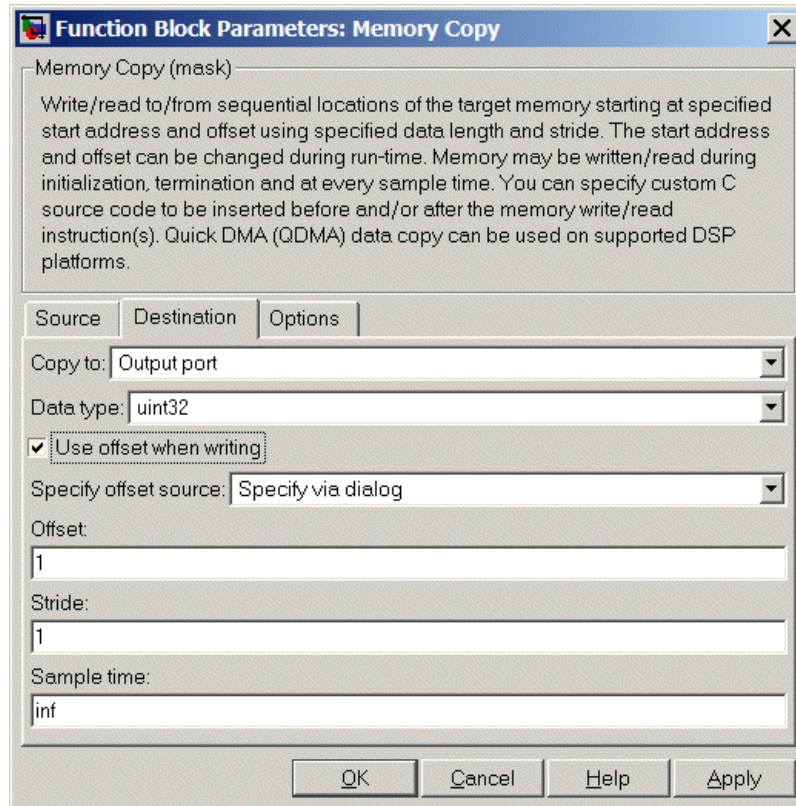
Input Stride = 1  
Output Stride = 1  
Number of Elements Copied = 10



Input Stride = 2  
Output Stride = 1  
Number of Elements Copied = 5

# Memory Copy

## Destination Parameters



### Copy to

Select the destination for the data. Choose one of the entries on the list:

- **Output port** — Copies the data to the block output port. From the output port the block passes data to downstream blocks in the code.
- **Specified address** — Copies the data to the specified location in **Specify address source** and **Address**.
- **Specified source code symbol** — tells the block to copy the variable or symbol (variable) to the symbol you enter in **Source code symbol**. When you select this copy to option, you enable the **Source code symbol** parameter.



---

**Note** If you do not select the Output port option for **Copy to**, you must change the **Data type** parameter setting from the default **Inherit from source** to one of the data types on the **Data type** list. If you do not make the change, you receive an error message that the data type cannot be inherited because the input port does not exist.

---

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

### Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either **Specify via dialog** or **Input port** from the list. Selecting **Specify via dialog** activates the **Address** parameter for you to enter the address for the variable.

When you select **Input port**, the port label on the block changes to **&dst**, indicating that the block expects the destination address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

### Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax.

### Address

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use `hex2dec` to convert the address to the proper format. Here is one example that converts `0x2000` to decimal form.

```
8192 = hex2dec('2000');
```

For this example, you could enter either `8192` or `hex2dec('2000')` as the address.

# Memory Copy

---

## Data type

Use this parameter to specify the type of data that your variable uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `Inherit` from `input port` for inheriting the data type for the variable from the block input port.

## Specify offset source

The block provides two sources for the offset — `Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when writing the output data.

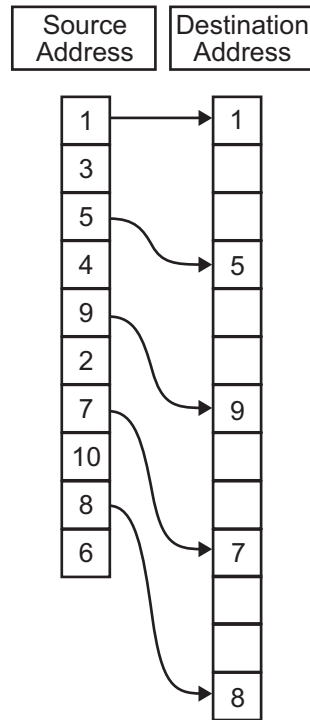
## Offset

**Offset** tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before writing the first value to the destination. One is the default offset value and **Offset** accepts only positive integers of one or greater.

## Stride

Stride lets you specify the spacing for copying the input to the destination. By default, the stride value is one, meaning the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value not equal to one, the output data is stored not sequentially, but by skipping addresses equal to the stride. **Stride** must be a scalar with positive integer value of one or greater.

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.



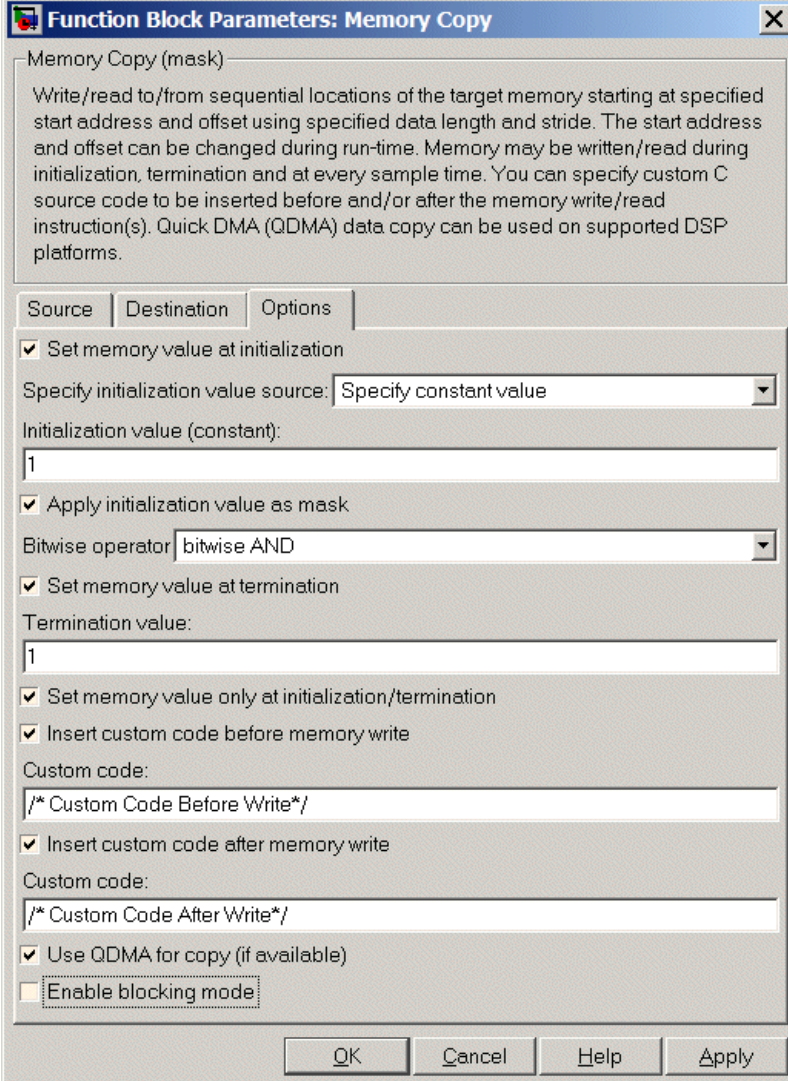
Input Stride = 2  
Output Stride = 3  
Number of Elements Copied = 5

### Sample time

Sample time sets the rate at which the memory copy operation occurs, in seconds. The default value `Inf` tells the block to inherit the sample time from the input if there is one or the Simulink model (when there are no input ports on the block). Enter the sample time in seconds as you need.

# Memory Copy

## Options Parameters



**Function Block Parameters: Memory Copy**

Memory Copy (mask)

Write/read to/from sequential locations of the target memory starting at specified start address and offset using specified data length and stride. The start address and offset can be changed during run-time. Memory may be written/read during initialization, termination and at every sample time. You can specify custom C source code to be inserted before and/or after the memory write/read instruction(s). Quick DMA (QDMA) data copy can be used on supported DSP platforms.

Source | Destination | Options

- Set memory value at initialization
- Specify initialization value source: Specify constant value
- Initialization value (constant): 1
- Apply initialization value as mask
- Bitwise operator: bitwise AND
- Set memory value at termination
- Termination value: 1
- Set memory value only at initialization/termination
- Insert custom code before memory write
- Custom code: /\* Custom Code Before Write\*/
- Insert custom code after memory write
- Custom code: /\* Custom Code After Write\*/
- Use QDMA for copy (if available)
- Enable blocking mode

OK Cancel Help Apply

## Set memory value at initialization

When you check this option, you direct the block to initialize the memory location to a specific value when you initialize your program at run time. After you check this option, use the **Set memory value at termination** and **Specify initialization value source** parameters to set your desired value, or tell the block to get the initial value from the input

## Specify initialization value source

After you check Set memory value at initialization, use this parameter to select the source of the initial value. Choose either

- **Specify constant value** — Sets a single value to use when your program initializes memory. Enter any value that meets your needs.
- **Specify source code symbol** — Specifies a variable (a symbol) to use for the initial value. Enter the symbol as a string.

## Initialization value (constant)

If you check **Set memory value at initialization** and choose **Specify constant value** for **Specify initialization value source**, enter the constant value to use here. Any real value that meets your needs is acceptable.

## Initialization value (source code symbol)

If you check **Set memory value at initialization** and choose **Specify source code symbol** for **Specify initialization value source**, enter the symbol to use here. Any symbol that meets your needs and is in the symbol table for the program is acceptable. When you enter the symbol, the block does not verify whether the symbol is a valid one. If it is not valid you get an error when you try to compile, link, and run your generated code.

## Apply initialization value as mask

You can use the initialization value as a mask to manipulate register contents at the bit level. Your initialization value is treated as a string of bits for the mask.

Checking this parameter enables the **Bitwise operator** parameter for you to define how to apply the mask value.

To use your initialization value as a mask, the output from the copy has to be a specific address. It cannot be an output port, but it can be a symbol as well.

# Memory Copy

---

## Bitwise operator

To use the initialization value as a mask, select one of the following from the **Bitwise operator** list to describe how to apply the value as a mask to the memory value.

Bitwise Operator List Entry	Description
bitwise AND	Apply the mask value as a bitwise AND to the value in the register.
bitwise OR	Apply the mask value as a bitwise OR to the value in the register.
bitwise exclusive OR	Apply the mask value as a bitwise exclusive OR to the value in the register.
left shift	Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer.
right shift	Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer.

Applying a mask to the copy process lets you select individual bits in the result, for example, to read the value of the fifth bit by applying the mask.

## Set memory value at termination

Along with initializing memory when the program starts to access this memory location, this parameter directs the program to set memory to a specific value when the program terminates.

## **Set memory value only at initialization/termination**

This block performs operations at three periods during program execution — initialization, real-time operations, and termination. When you check this option, the block only does the memory initialization and termination processes. It does not perform any copies during real-time operations.

## **Insert custom code before memory write**

Select this parameter to add custom C code before the program writes to the specified memory location. When you check this, you enable the Custom code parameter where you enter your C code.

## **Custom code**

Enter the custom C code to insert into the generated code just before the memory write operation. Code you enter here appears in the generated code exactly as you enter it.

## **Insert custom code after memory write**

Select this parameter to add custom C code immediately after the program writes to the specified memory location. When you check this, you enable the Custom code parameter where you enter your C code.

## **Custom code**

Enter the custom C code to insert into the generated code just after the memory write operation. Code you enter here appears in the generated code exactly as you enter it.

## **Use QDMA for copy (if available)**

For processors that support quick direct memory access (QDMA), check this parameter to enable the QDMA operation and to access the blocking mode parameter.

If you check this parameter, your source and destination data types must be the same or the copy operation returns an error. Also, the input and output stride values must be one.

## **Enable blocking mode**

If you check the **Use QDMA for copy** parameter, check this option to direct the memory copy operations to be blocking processes. With blocking enabled, other processing in the program waits while the memory copy operation finishes.

# Memory Copy

---

## See Also

Memory Allocate



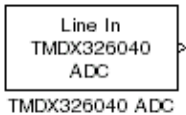
## Purpose

Configure codec on daughter card for output to C6711 DSK

## Library

TMDX326040 Support in Embedded Target for TI C6000 DSP for TI DSP

## Description



With the TMDX326040A daughter card installed on your C6711 DSK, you use this block to configure the codec on the card. The daughter card codec replaces the codec on the C6711 DSK, taking the analog input from the analog ports on the DSK and converting them to digital data. This block configures the analog-to-digital conversion performed by the daughter card.

---

**Note** This card is also known as the PCM3003 Audio Daughter Card.

---

Both the sampling rate and data format for the daughter card codec are fixed:

- Sampling rate is 48 kHz.
- Data format is linear 16-bit words.

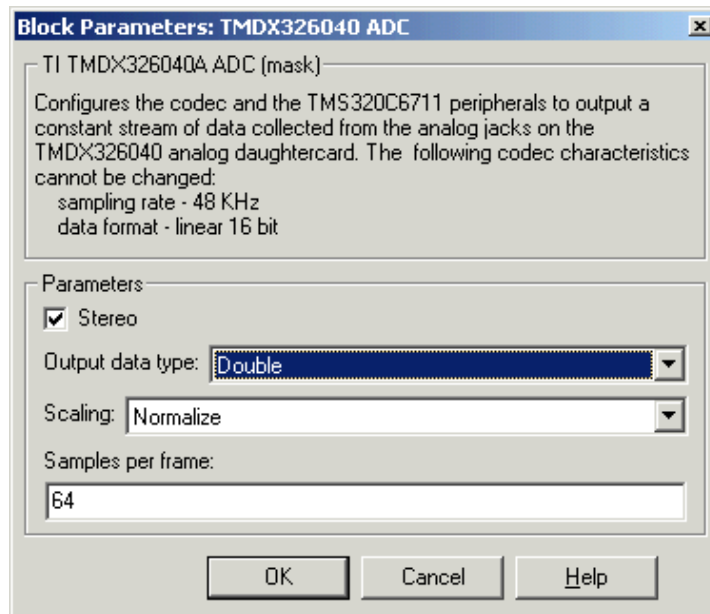
Data leaving the codec on the daughter card goes to the COM port on the C6711 DSK and then to the C6711 digital signal processor.

You have the choice of using either stereo or monaural input to the card. The **Stereo** option tells the codec whether the input is in stereo or mono format. When you use the block, the **Stereo** option is selected by default.

Other codec options help you configure the digital data from the daughter card, such as setting the data type (double, single, or integer) and selecting whether the output data should be unmodified or scaled to the range between -1 and 1.

# TMDX326040 ADC

## Dialog Box



### Stereo

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.

### 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  $\pm 1.0$ , based on the codec data format. Select either Normalize or Integer for the scaling. Normalize is the default setting. Scaling applies only to the floating-point data types double and single. When you use integer data, data values get scaled to be between -32768 to 32767.

## **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. 64 samples per frame is the default setting. 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.

## **See Also**

TMDX326040 DAC

# TMDX326040 DAC

---

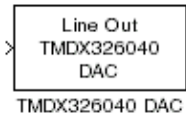
## Purpose

Configure codec on daughter card to send data to analog output on card

## Library

TMDX326040 Support in Embedded Target for TI C6000 DSP for TI DSP

## Description



With the TMDX326040A daughter card installed on your C6711 DSK, you use this block to configure the codec on the card. The daughter card codec replaces the codec on the C6711 DSK, sending its output to the output connectors on the card. This block configures the digital-to-analog conversion performed by the daughter card.

---

**Note** This card is also known as the PCM3003 Audio Daughter Card.

---

Both the sampling rate and data format for the daughter card codec are fixed:

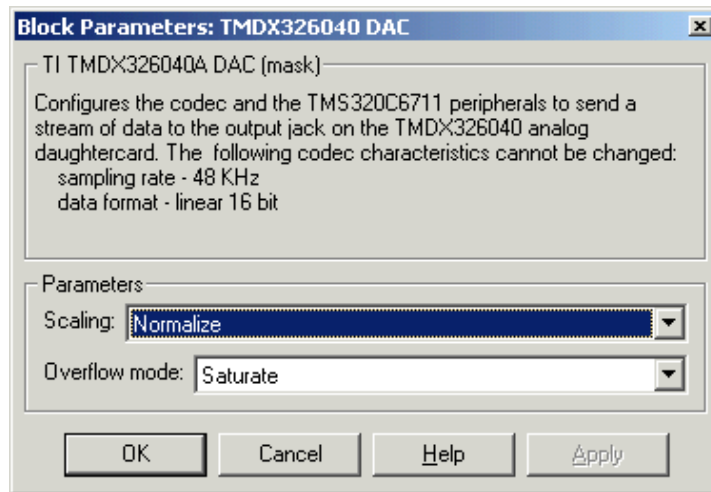
- Sampling rate is 48 kHz.
- Data format is linear 16-bit words.

Analog data leaving the codec on the daughter card outputs on the card and then to the C6711 DSK output connectors. Whether the output is in monaural or stereo depends on the setting of the TMDX326040 ADC block in your model.

To work properly, you must be sure the input signal is a column vector when you use the monaural mode, or an N-by-2 matrix in stereo mode. This input format must match the ADC block mode — when you select the **Stereo** option for the C6711 ADC block operating mode, you must format your input data as an N-by-2 matrix here.

Other codec options help you configure the digital data from the daughter card — scaling and overflow mode. The scaling option determines whether the input data remains unmodified or is scaled to the range between -1 and 1.

## Dialog Box



### Scaling

Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range  $\pm 1.0$ . Matching the setting for the TMDX326040 ADC block is appropriate here. Normalized scaling is the default setting.

When the data type is integer, this scaling option does not apply. Scaling applies only to floating-point data types single and double. You can select different data types and scaling for the ADC and DAC blocks within your model.

### Overflow mode

Determines how the codec responds to data that is outside the range specified by the **Scaling** parameter. You choose either Wrap or Saturate from the list to specify how to handle the result of an overflow in an operation. Saturate mode is slightly less efficient because of the logic executed for each sample to determine whether to saturate the value.

## See Also

TMDX326040 ADC

# To Memory

---

## Purpose

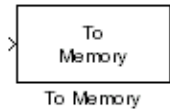
Send data from processor to memory on C6000 target

## Library

C6000 DSP Core Support in Embedded Target for TI C6000 DSP for TI DSP

---

## Description



**Note** This block will be removed in the future. Please use the Memory Allocate and Memory Copy blocks instead.

---

When your Simulink model has this block in place, Real-Time Workshop code generation inserts the C commands to write data to the specified memory location on the target. The inserted code takes the value you send to the block input port and writes it to the location in **Memory address**.

In the generated code, you see something like these lines representing the To Memory block operation:

```
/* S-Function Block: <Root>/To Memory (c6000mem_snk) */
{
    /* Memory Mapped Output */
    *((volatile int *) (4096U)) = (real32_T) 8;
}
```

In simulations this block does not perform any operations. To Memory blocks work only in code generation and when your model runs on your target.

Options for the block let you send different starting and ending values to memory when the program runs on the digital signal processor.

## Using To Memory Blocks

You must take care when you use To Memory blocks in your models. Because the To Memory blocks give you control over where the target stores information in memory, and provide full flexibility to copy any section of target data and write to any memory sections, pay attention to how you use the block memory options. This flexibility can lead to unexpected behavior caused by memory operations in your model.

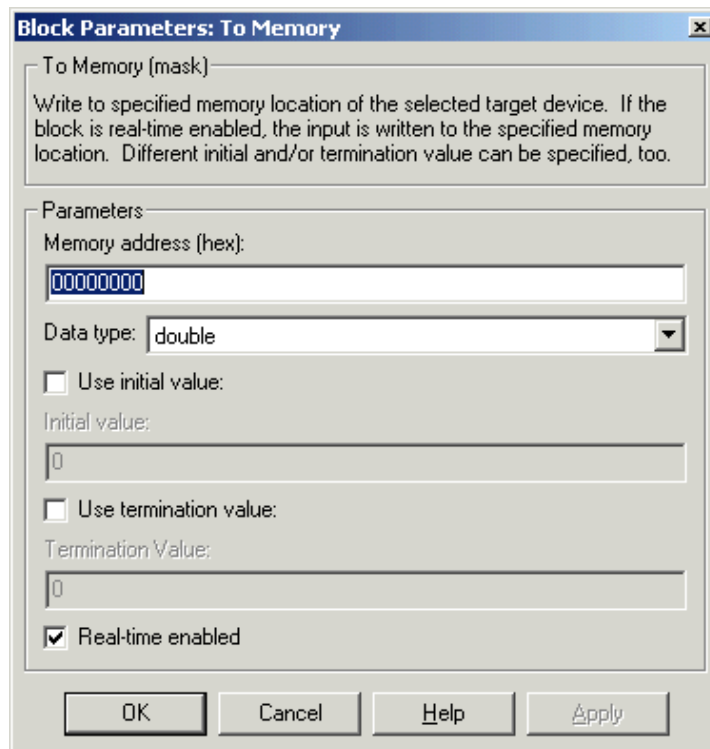
When you use the optional features in the To Memory block to write data to specified locations in memory, you might be writing to memory locations that are reserved for the compiler to use. Writing to those locations can cause CCS to crash.

To prevent your model from encountering memory errors like these, generate your code once without loading the COFF file to the target. Look at the generated file *projectname.map*, where *projectname* is the name of your project, to see the memory range that the compiler uses.

From this list of allocated memory, you can determine the memory ranges that you can use safely without overwriting the reserved compiler sections.

You should examine the .map file for your project each time you change the Simulink model associated with your project.

## Dialog Box



### Memory address (hex)

Specifies the address to which you are sending data from the code. Enter the address as a hexadecimal value, without the leading 0x indicator. Any valid memory address works, as long as the processor can write to it.

## Data type

Sets the type for the data going to memory. Select one of the following types:

- **double** — double-precision floating-point values. This is the default setting and allows the full range of values representable in double-precision arithmetic as defined by the IEEE specification.
- **single** — single-precision floating-point values whose range is defined by the IEEE specification on single-precision values.
- **uint8** — 8-bit unsigned integers. Input values range from 0 to 255.
- **int16** — 16-bit signed integers. With the sign, the values range from -32768 to 32767.
- **int32** — 32-bit signed integers. Values range from  $-2^{31}$  to  $(2^{31}-1)$ .

## Use initial value

Select this option when you want to send a specific value to memory during the first execution on your model. Enter your desired value in **Initial value**.

## Initial value

Enter the value to send to memory on the first execution of this code. Enter a floating-point integer here. The block interprets the value you enter as an integer. For example, to place the integer value 100 in memory, enter 100 here. Note that the block does not support MATLAB integer data types.

## Use termination value

Select this option when you want to send a specific value to memory during the last or final execution of your model. Enter your desired value in **Termination value**.

## Termination value

Enter the value to send to memory on the last execution of this code. Enter a floating-point integer here. The block interprets the value you enter as an integer. For example, to place the integer value 100 in memory on the final execution pass, enter 100 here.

## Real-time enabled

In basic terms, generated code executes as follows:



- 1 Initialize
- 2 Start execution (your initial value is written to memory)
- 3 Output (execute loop for each time step)
- 4 Terminate execution (your termination value is written to memory)

Selecting **Real-time enabled** causes the code to write data to memory during the output phase of code execution. When you clear this option, the code does nothing during the output phase — it does not write data to the memory address you specify in **Memory address**. You might clear **Real-time enabled** when you want to write a value to memory only during the start phase (or only during the terminate phase, or during both phases) but not during output execution). The block input port disappears when you clear this check box.

## See Also

From Memory

# To Rtdx

---

## Purpose

Add RTDX communication channel to send data from target to MATLAB

## Library

RTDX Instrumentation in Embedded Target for TI C6000 DSP for TI DSP

## Description



When your Simulink model has this block in place, Real-Time Workshop code generation inserts the C commands to create an RTDX output channel on the target. The inserted code opens and enables the channel with the name you specify in **Channel name**. You can open, close, disable, and enable the channel from the host side afterwards, overriding the target side status.

In the generated code from models with this block, you see a command like

```
RTDX_enableOutput(&channelname)
```

where `channelname` is the name you enter in **Channel name**.

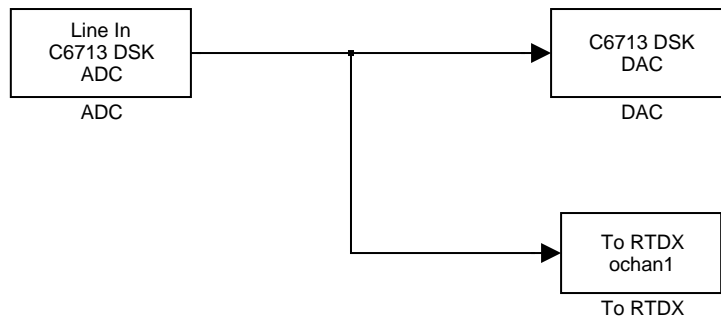
In simulations this block does not perform any operations. To Rtdx blocks work only in code generation and when your model runs on your target.

Using RTDX in your model involves:

- Adding one or more RTDX blocks to your model to prepare your target
- Downloading and running your model on your target
- Enabling the RTDX channels from MATLAB
- Using the `readmsg` and `writemsg` functions in MATLAB to send and retrieve data from the target over RTDX

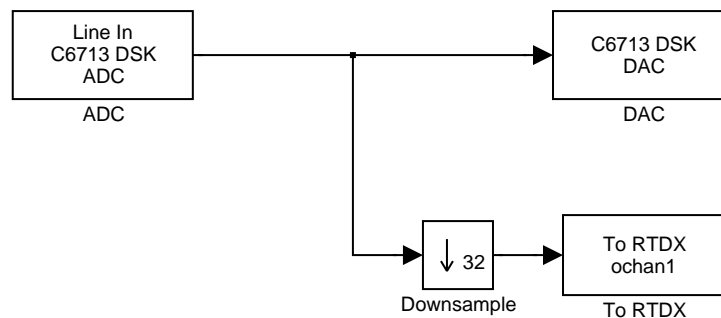
To see more details about using RTDX in your model, refer to Tutorial 1-2 — Using Links for RTDX in your Link for Code Composer Studio documentation.

One mistake is to connect a To Rtdx block directly to an ADC block, or another source block. Due to current RTDX timing constraints, the generated code from this arrangement does not work as you expect. Look at the following model for an example that does not properly transfer data.



Applications that you generate from models that contain the directly connected blocks are likely to overrun because the sampling time of the codec or source is much faster than RTDX processing time. RTDX will not be able to keep up. This is true even if you do not generate your application from Simulink.

Adding additional blocks can fix the problem. In the next model, adding the Downsample block with 32 for **K**, the **Downsample factor**, allows RTDX to return messages as expected.



When you are using the Link for CCS to transfer data from the target to the host (MATLAB), this overrun is manifested by `cc.rtdx.msgcount` eventually decreasing to zero.

You need to modify your application such that:

- Data going to the RTDX block channel is slowed down. For example, use the Downsample block. Downsampling to 32 usually works fine.

# To Rtdx

- Give the RTDX more time to process a message transfer by decreasing the size of the data (such as using short instead of int data types).
- Depending on the application type you are developing, you can use the standard RTDX or the high-speed RTDX. If you are using video processing using standard RTDX, you will not get the desired output. Please refer to the TI documentation for more information.

To enable high-speed RTDX in Simulink:

- In your model, go to **Simulation > Configuration Parameters**.
- Select **Real-Time Workshop** on the left pane. Highlight the **RTW system target file** options of the TI 6000 target.
- Select **TIC6000 target selection** on the left pane and on the right pane, select **Enable High-Speed RTDX**.

If you are using the Link for Code Composer Studio Development Tools, you need to configure and cleanup RTDX properly before and after executing your model or code. Refer to the RTDX tutorials in the Link for Code Composer Studio documentation in the online Help system to see an example of how to do this housekeeping task.

## Dialog Box



### Channel name

Defines the name of the output channel on the target DSP. Recall that output channels refer to transferring data from the target to the host (output from the target). To use this RTDX channel, you enable and open the channel with the name, and send data from the target to the host across

this channel. Specify any name as long as it meets C syntax requirements for length and character content.

**Enable blocking mode**

Puts RTDX communications into blocking mode where the target processor waits to continue processing until new data is available from the To Rtdx block.

In blocking mode, writing a message is suspended while the RTDX channel is busy, for example if a message is being written in or out of the channel. While suspended, the code waits at the RTDX\_write call site until the channel is no longer busy. Note that higher priority interrupts temporarily divert the program execution from this call site. Eventually, program execution comes back and waits until the channel stops writing.

In non blocking mode, writing a message is abandoned if the RTDX channel is busy (when it is writing — the data is being written in or out of the channel). The code continues with the current iteration.

Selecting blocking mode slows your processing while the processor waits — if the previous message is not written before the next write, your process stops. **Enable blocking mode** is selected by default and is recommended for most operations.

**Enable RTDX channel on start-up**

When your application code includes RTDX channel definitions, selecting this option enables the channels when you start the channel from MATLAB. With this selected, you do not need to use the Link for Code Composer Studio Development Tools enable function to prepare your RTDX channels. Note that the option applies only to the channel you specify in **Channel name**. You do have to open the channel.

**See Also**

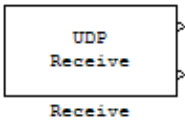
ccsdsp, From Rtdx, readmsg, writemsg

# UDP Receive

**Purpose** Receive uint8 vector as UDP message

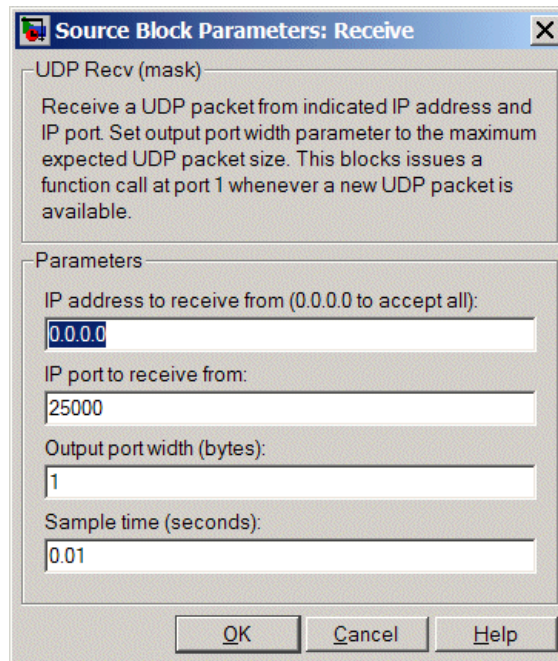
**Library** Host Communication Library in Embedded Target for TI C6000 DSP

**Description** A UDP message comes into this block from the transport layer. The block passes the message to the next downstream block. One block output is the data vector from the message. The second output is a flag that indicates when a new UDP message is available.



Models can contain only one UDP Receive block.

## 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.

**IP port to receive from**

Specify the port the block accepts messages from on this machine. The other end of the communication, usually a UDP Send block, sends messages to this port. The default value is 25000, but the values 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 to a value as large or larger than any message you expect to receive.

**Sample time (seconds)**

Use this option to specify when the block polls for new messages. Entering -1 lets the block inherit the sample time from an upstream block. Setting this to a specific value, often large, can reduce the chances that UDP messages get dropped. By default, the sample time is 0.01 seconds.

**See Also**

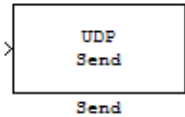
Byte Pack, Byte Reversal, Byte Unpack, UDP Send

# UDP Send

**Purpose** Send UDP message to target

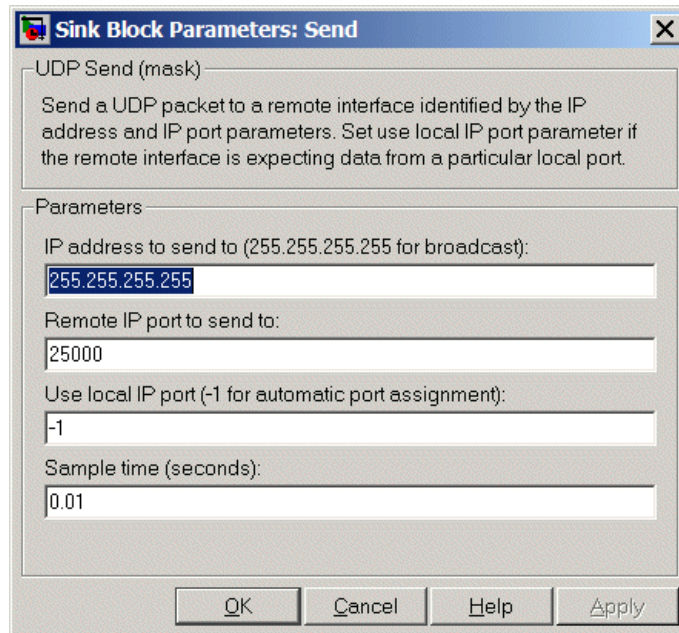
**Library** Host Communication Library in Embedded Target for TI C6000 DSP

**Description** The UDP send block receives a uint8 vector that it sends as a UDP message to the host. Input must be in the form of a uint8 vector with UDP format.



Models can contain only one UDP Send block.

## Dialog Box



### **IP address to send to (255.255.255.255 for broadcast)**

Specify the IP address to which the block sends the message. Entering the address 255.255.255.255 tells the block to broadcast the message to any listening IP address. Entering a specific IP address limits the block to sending the message to the specified address.



**IP port to send to**

Specify the port on the host to which the block sends the message. Port numbers range from 1 to 65535. This port designation must match the port number where you configure the C6000 target to receive UDP messages.

**Use the following local IP port**

Specify the local IP port the block sends the message from. Entering - 1 (the default value) for this option allows the network to select automatically the local IP port to use to send the message.

If the address you are sending to expects the message to come from a specific port, enter that port address here. If you enter a port number in the UDP Receive block option **IP port to receive from**, enter that port identifier here.

**Sample time**

Sample time tells the block how long to wait before polling for new messages.

**See Also**

Byte Pack, Byte Reversal, Byte Unpack, UDP Receive

# UDP Send

---

# Supported Hardware and Issues

---

Supported Hardware for Targeting  
(p. A-2)

Lists the hardware that the Embedded Target for TI C6000 DSP supports. Includes comments about supported operating systems where needed.

Requirements for the DM642 EVM  
(p. A-6)

Points out some details about using the DM642 target.

Continuing Issues with Embedded  
Target for TI C6000 DSP (p. A-11)

Describes some important features about targeting particular hardware.

## Supported Hardware for Targeting

Using the C6000 target in Real-Time Workshop, the Embedded Target for TI C6000 DSP supports the following boards produced by TI and other manufacturers.

Supported Board Designation	Board Description
DM642 EVM	DM642 Evaluation module for developing video processing algorithms and applications
D.signT DSK-91c111	Ethernet adapter daughter card to use with the C6416, C6711, and C6713 DSK targets. This card provides support for TCP/IP and UDP communications. Refer to “Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP” on page A-3 for more information.
TMS320C6416 DSK	C6416 DSP Starter Kit. Does not work on Microsoft Windows™ NT platforms. To use the UDP and TCP/IP blocks with this board, you must have a supported daughtercard—the TMDX326040A Daughter Card.
TMS320C6455 DSK	C6455 DSP Starter Kit.
TMS320C6711 DSK	C6711 DSP Starter Kit. To use the UDP and TCP/IP blocks with this board, you must have a supported daughtercard—the TMDX326040A Daughter Card.
TMS320C6713 DSK	C6713 DSP Starter Kit. Does not work on Microsoft Windows™ NT platforms. To use the UDP and TCP/IP blocks with this board, you must have a supported daughtercard—the TMDX326040A Daughter Card.

<b>Supported Board Designation</b>	<b>Board Description</b>
TMDXPDK6727	C6727 Professional Audio Development Kit.
TMDX326040A Daughter Card. Also called the PCM3003 Audio Daughter Card	Supplemental card to use with the C6711 DSK. Provides a high quality (48 KHz, 16-bit) codec to act in place of the one on the C6711 DSK. Use this card to provide support for TCP/IP and UDP on the C6414 DSK, C6711 DSK, and C6713 DSK targets.
C6xxx simulators in CCS	Digital signal processor simulators in CCS. You cannot run models on your simulator because simulators do not simulate the codec on the board. You can generate code to the simulators and use CCS and RTDX links with them.
Custom boards based on supported C6000 processors	Code generation for targets that are not explicitly supported, but that use supported DSPs, such as C6711 or C6416. Refer to “Supported Processors” on page A-4 for the list of processors.

### **Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP**

To use the D.signT DSK-91C111 with the required Texas Instruments TMS320C6000 TCP/IP Stack, 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.

To support code generation for your targets, the Embedded Target for TI C6000 DSP offers an option for the C6000 target that provides a Real-Time Workshop (RTW) target you use to generate executable code that runs on the supported boards, or to build a project in CCS IDE. You select this option when you set the simulation parameters in Real-Time Workshop for your model.

Within the same C6000 target in Real-Time Workshop, the options let you generate code specifically for any of the supported targets, or to build a project in CCS.

When you set the simulation parameters for your model in Real-Time Workshop, you can choose to generate target-specific executable code when you use target-specific blocks in your Simulink model. Target specific blocks, like the blocks in the C64x DSP library, use code optimized for your specified target.

Texas Instruments produces the evaluation modules and DSP starter kits to help developers create digital signal processing applications for the Texas Instruments digital signal processors.

You can create, test, and deploy your processing software and algorithms or filters on the target processors without the difficulties inherent in starting with the digital signal processor itself and building the support hardware to test the application on the processor.

Instead, the development boards provide the input hardware, output hardware, timing circuitry, memory, and power for the digital signal processors. TI provides the software tools, such as the C compiler, linker, assembler, and integrated development environment, for PC users to develop, download, and test their algorithms and applications on the processors.

## Supported Processors

Embedded Target for TI C6000 DSP provides code generation for boards that use the following Texas Instruments processors:

- C62x
  - 6201
  - 6202
  - 6203
  - 6204
  - 6205
- C64x
  - 6410
  - 6411
  - 6412
  - 6413

- 6414
- 6415
- 6416
- 6418
- 6455
- 6482
- C67x
  - 6701
  - 6711
  - 6712
  - 6713
  - 6722
  - 6726
  - 6727
- DM64x
  - DM640
  - DM641
  - DM642

## Requirements for the DM642 EVM

Certain requirements for the DM642 EVM differ from the other supported targets. This section provides details about using both the DM642 EVM hardware target and the simulator. Using the DM642 requires the following:

- DM642 EVM version identification
- CCS installation version 3.1 with DSP/BIOS 4.90
- XDS560 (high speed RTDX emulator) or XDS510 (Regular RTDX emulator, if your model does not require high-speed RTDX capability)
- Device Driver Development Kit (DDK) patch as required by your DM642 version (refer to “Required DDK Versions for DM642 EVM Revisions” on page A-8).
- TMS320DM642 Digital Media Development Kit (DMDK)
- To use the UDP and TCP blocks for the board, you must install the TMS320C6000 TCP/IP Stack from Texas Instruments
- Projects must enable DSP/BIOS. Embedded Target for TI C6000 DSP does not support operations on the DM642 EVM without DSP/BIOS.

### About DM642 EVM Board Revisions

Working with DM642 EVM boards requires that you identify the board revision that you own.

#### Identifying Your DM642 EVM Board Revision

Spectrum Digital has released three different versions of the DM642 EVM board. DM642 EVM board Versions 1 and 2 are the same except for the CPU clock speed. Version 2 uses a 720 MHz clock, rather than the 600 MHz clock on Version 1. Both versions use Phillips SAA7115 video decoders.

Version 3 is a redesign of the board that uses TI TVP5146/5150 video decoders and a CPU clock speed of 720 MHz. To use Version 3 boards, you must install updated TI video drivers (included with TI Device Driver Developer’s Kit (DDK)) to match the new decoders.

Here is how you identify the correct version number of your 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.

## Setting Up Code Composer Studio for the DM642 EVM

Your DM642 EVM requires a separate Code Composer Studio installation. To use the EVM when you have more than one CCS installation, you need to install the CCS for the DM642 EVM in a separate location. You cannot merge your DM642 CCS installation with existing or other CCS installations. Follow the installation guidelines provided by Texas Instruments when you install CCS, to use your DM642 EVM.

Install the patch C6000-2.20.00-FULL-to-C6000-2.20.18-FULL.EXE as directed by Texas Instruments.

Finally, install the Device Driver Development Kit patch `ddk-v1-10-00-23.exe`.

### About the Device Driver Development Kit

To use the Embedded Target for TI C6000 DSP software with your DM642, you need to install the Device Driver Developer Kit (DDK) patch that you get from Texas Instruments.

While the DDK is optional for some DM642 operations, the Embedded Target for TI C6000 DSP requires the DDK for code generation. According to TI, the DDK is the TI Device Driver Development Kit. Version 1.1 of the DDK includes device drivers for the DM642 EVM peripherals that are used by many of the examples and demos.

While the DDK is not required to run precompiled code, it is needed to rebuild or develop code. You should install the DDK in the folder `TI_DIR`.

The DDK patch for CCS is an optional patch that the Embedded Target for TI C6000 DSP requires.

### **Required DDK Versions for DM642 EVM Revisions**

There are three different versions of DM642 EVM board. They differ in a number of areas including CPU speed and the video decoders used, as described in “About DM642 EVM Board Revisions” on page A-6. To use the Embedded Target for TI C6000 DSP with your DM642 EVM, you must install the correct DDK version, shown here.

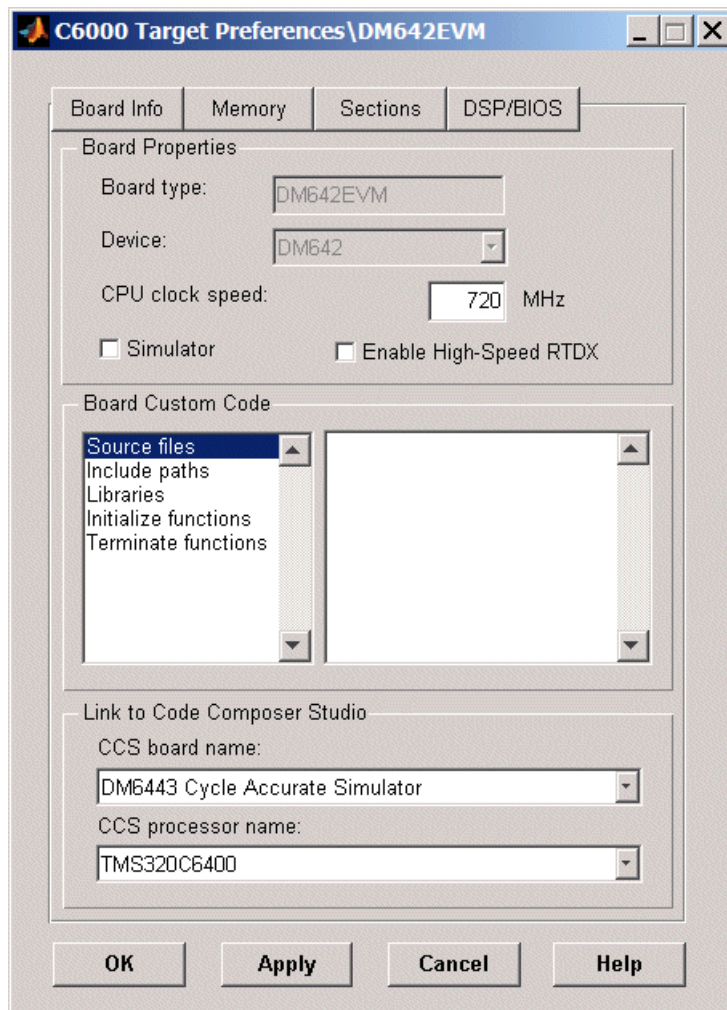
<b>DM642 EVM Board Revision</b>	<b>Required DDK Version</b>
Version 1	DDK 1.10
Version 2	DDK 1.11
Version 3	DDK 1.11

### **About the XDS560 PCI-Bus JTAG Scan-Based Emulator**

You need the XDS560 Emulator to use the DM642 with the Embedded Target for TI C6000 DSP. While the XDS510 Emulator might work, the target software has not been tested with it.

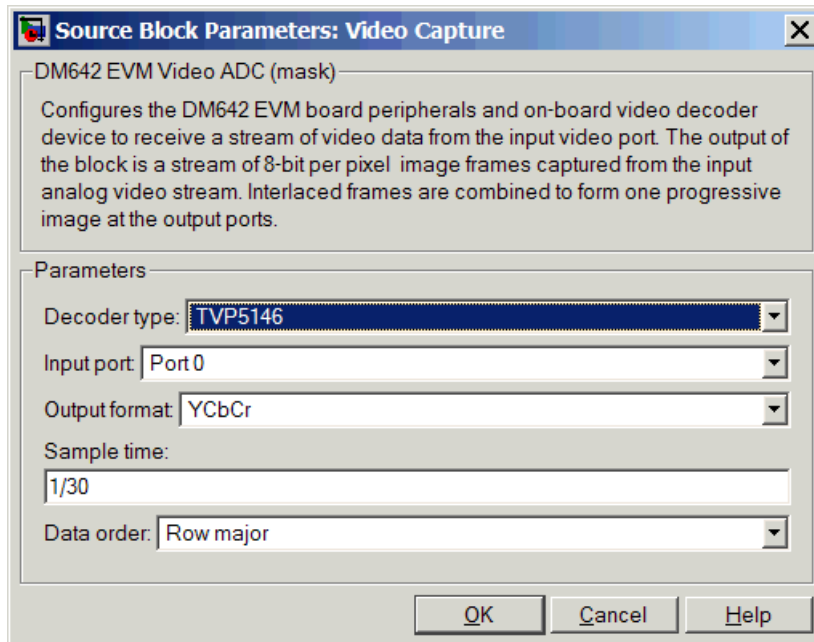
### **Configuring the Target Preferences Block for Your DM642 EVM**

When you use the DM642EVM 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 installation directory and that you select the correct decoder type TVP5146 when you use DM642 EVM Video ADC blocks as shown in the following figure.



## Continuing Issues with Embedded Target for TI C6000 DSP

This section details some target operations that you should know about as you use the Embedded Target for TI C6000 DSP.

- “Setting the Clock Speed on the C6713 DSK” on page A-11
- “Simulink Stop Block Works Differently When Not Using DSP/BIOS Features” on page A-12

### 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.

- 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.

To make this happen whenever you open Code Composer Studio to use your C6713 DSK, edit the file `\ti\cc\gel\dsk6713.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.

## A

- adding DSP/BIOS to generated code 2-92
- applications
  - logging 2-68
- Archive\_CCS\_Library 2-108
- asynchronous scheduling 2-71

## B

- block limitations using model reference 2-109
- block recommendations 2-120
- blocks
  - C62x 5-4
  - DM642 5-13
  - RTDX 5-3
  - use in target models 2-120
- blocks to avoid in models 2-120
- build configuration
  - compiler options, default 2-106
  - default 2-106
  - MW\_custom 2-106
- build directory
  - contents of 2-135
  - naming convention 2-124
- building models
  - use C62x DSP Library blocks 4-8

## C

- C6000 model reference 2-107
- C6000 Target
  - code generation options 2-92
  - compiler options 2-98
  - run-time options 2-95
  - targeting Code Composer Studio 2-144
  - TI C6000 linker options 2-105
- c6000 target preferences block 2-40

- C62x Autocorrelation block 5-48
- C62x Bit Reverse block 5-52
- C62x Block Exponent block 5-52
- C62x blocks 5-4
- C62x Complex FIR block 5-53
- C62x Convert Floating-Point to Q.15 block 5-55
- C62x Convert Q.15 to Floating-Point block 5-56
- C62x DSP Library blocks
  - building models 4-8
  - choosing blocks to optimize code 4-9
  - common characteristics 4-3
  - Q format notation 4-5
  - using source and sink blocks 4-9
- C62x FFT block 5-57
- C62x General Real FIR block 5-59
- C62x LMS Adaptive Filter block 5-61
- C62x Matrix Multiplication block 5-64
- C62x Matrix Transpose block 5-67
- C62x Radix-2 FFT block 5-68
- C62x Radix-2 IFFT block 5-70
- C62x Radix-4 Real FIR block 5-72
- C62x Radix-8 Real FIR block 5-74
- C62x Real IIR block 5-78
- C62x Reciprocal block 5-81
- C62x Symmetric Real FIR block 5-82
- C62x Vector Dot Product block 5-86
- C62x Vector Maximum Index block 5-87
- C62x Vector Maximum Value block 5-88
- C62x Vector Minimum Value block 5-89
- C62x Vector Multiply block 5-90
- C62x Vector Negate block 5-91
- C62x Vector Sum of Squares block 5-92
- C62x Weighted Vector Sum block 5-93
- C6416 DSK ADC block 5-113
- C6416 DSK DAC block 5-118
- C6416 DSK DIP Switch block 5-120

- C6416 DSK LED block 5-125
- C6416 DSK RESET block 5-126
- C64x Autocorrelation block 5-146
- C64x Bit Reverse block 5-150
- C64x Block Exponent block 5-150
- C64x Complex FIR block 5-151
- C64x Convert Floating-Point to Q.15 block 5-153
- C64x Convert Q.15 to Floating-Point block 5-154
- C64x FFT block 5-155
- C64x General Real FIR block 5-157
- C64x LMS Adaptive Filter block 5-159
- C64x Matrix Multiplication block 5-162
- C64x Matrix Transpose block 5-165
- C64x Radix-2 FFT block 5-166
- C64x Radix-2 IFFT block 5-168
- C64x Radix-4 Real FIR block 5-170
- C64x Radix-8 Real FIR block 5-172
- C64x Real IIR block 5-176
- C64x Reciprocal block 5-179
- C64x Symmetric Real FIR block 5-180
- C64x Vector Dot Product block 5-184
- C64x Vector Maximum Index block 5-185
- C64x Vector Maximum Value block 5-186
- C64x Vector Minimum Value block 5-187
- C64x Vector Multiply block 5-188
- C64x Vector Negate block 5-189
- C64x Vector Sum of Squares block 5-190
- C64x Weighted Vector Sum block 5-191
- C6711 DSK
  - configure 2-136
  - confirming proper configuration 2-115
  - fixed sample rate 2-35
  - general code generation options 2-90
  - start/stop models 2-113, 2-143
  - target options 2-85
  - TLC debugging options 2-89
  - tutorial about multirate applications 2-123
- C6711 DSK ADC
  - choose the sample rate 2-34
- C6711 DSK ADC block 2-32, 5-210
- C6711 DSK ADC dialog 2-35
- C6711 DSK block
  - the overrun indicator 2-38
- C6711 DSK blocks
  - tutorial 2-123
- C6711 DSK DAC block 2-36, 5-213
  - choose the scaling 2-37
- C6711 DSK DAC dialog 2-36
- C6711 DSK DIP Switch block 5-215
- C6711 DSK directories
  - build 2-124
  - working 2-124
- C6711 DSK LED block 5-219
  - configure the block 2-38
- C6711 DSK RESET block 5-220
- c6711dsktest.mdl 2-116, 2-138
  - errors while running 2-118, 2-140
  - use 2-116
  - verifying that the model is running 2-118, 2-140
- C6713 DSK ADC block 5-238
- C6713 DSK DAC block 5-242
- C6713 DSK DIP Switch block 5-244
- C6713 DSK LED block 5-249
- C6713 DSK RESET block 5-250
- CCS IDE
  - create projects for the IDE 2-144
- Code Composer Studio 2-144
- code generation for subsystem 2-60
- configure DM642 blocks 2-61
- configure the software timer 2-45, 5-97, 5-129, 5-195, 5-223, 5-253, 5-274, 5-292



- configure your C6711 DSK for Embedded Target for TI C6000 DSP 2-136
- confirm your C6711 DSK configuration 2-115
- convert data types 4-8
- CPU clock speed 2-45, 5-97, 5-129, 5-195, 5-223, 5-253, 5-274, 5-292
- current CPU clock speed 2-45, 5-97, 5-129, 5-195, 5-223, 5-253, 5-274, 5-292
- custom C6000 target
  - about 2-150
  - preferences block 2-150
  - setup 2-150
- custom hardware guidelines 2-146
- custom hardware, target 2-146
- custom\_MW compiler options 2-106

## D

- default build configuration 2-106
- default compiler options 2-106
- disabling logging 2-69
- discrete solver 2-82
- DM642 blocks 5-13
- DM642 blocks, configure 2-61
- DSP/BIOS
  - added files 3-8
  - adding to generated code 2-92
  - files removed from project 3-9
- DSP/BIOS, enabling 3-25

## E

- Embedded Target for TI C6000 DSP 1-4
  - about 1-2, 1-4
  - configure C6711 DSK ADC blocks 2-32
  - configure C6711 DSK LED blocks 2-38
  - configure C6711DSK LED blocks 2-38

- configure the C6711 DSK DAC block 2-36
- create Simulink model for targeting 2-119
- errors while running test model c6711dsktest 2-118, 2-140
- expected background for use 1-5
- hardware and OS requirements 1-9
- information for new users 1-5
- peripheral hardware for testing C6711 DSK operation 2-115, 2-137
- procedure for testing the operation 2-116
- requirements for TI software 1-10
- starting/stopping test model c6711dsktest 2-119, 2-140
- suitable applications 1-3
- test installation and operation of the C6711 DSK 2-115, 2-137
- use C6711 DSK blocks 2-21
- enabling DSP/BIOS 3-25
- errors running c6711dsktest.mdl 2-118, 2-140
- execution in timer-based models 2-72

## F

- files added to DSP/BIOS project 3-8
- files removed from DSP/BIOS projects 3-9
- fixed-point numbers 4-4
  - signed 4-4
- fixed-step solver 2-82
- From Rtdx block 5-342

## G

- generate code for subsystem 2-60
- generate optimized code 2-92
- generate\_code\_only option 2-95

**H**

- halting a running process 2-119
- hardware requirements for Embedded Target for TI C6000 DSP 1-9
- hardware, custom 2-146
- hardware, guidelines for using custom boards 2-146
- headroom meter 2-39

**I**

- inaccurate profile information 3-12
- Incorporate DSP/BIOS option 2-92
- indicator, overrun 2-39
- initialized memory 2-149
- inline Signal Processing Blockset functions option 2-92
- interrupt threshold 2-102
  - set value 2-104

**L**

- LED block 2-38
- logging
  - about 2-69
- logging in models 2-68
- logging options in Simulink Parameters dialog 2-69
- logging, disabling 2-69

**M**

- management, memory 2-150
- map memory 2-149
- map, memory 2-149
- memory
  - initialized 2-149

- management 2-150
- map 2-149
  - section 2-149
  - segment 2-149
  - uninitialized 2-149
- memory maps 2-149
- mi option 2-102
- model execution 2-71
- model reference 2-107
  - about 2-107
  - Archive\_CCS\_Library 2-108
  - block limitations 2-109
  - modelreferencecompliant flag 2-110
  - setting build action 2-108
  - target preferences blocks 2-109
  - using 2-108
- model schedulers 2-71
- modelreferencecompliant flag 2-110
- models
  - logging 2-68
- MW\_custom build configuration 2-106

**O**

- optimization,target specific 2-92
- optimize code 4-9
- OS requirements for Embedded Target for TI C6000 DSP 1-9
- overflow mode 2-37
- overrun indicator 2-39
- overrun notification method 2-97

**P**

- procedure for testing Embedded Target for TI C6000 DSP 2-116
- profile generated code 3-10

- profile report
  - about 3-10
  - correcting inaccurate profile information 3-12
  - profiling subsystems 3-10
  - reading 3-16
  - sample 3-16
  - timing details 3-11
- projects, create for CCS 2-144

## Q

- Q format notation 4-5

## R

- Real-Time Workshop solver options 2-82
- RTDX blocks 5-3
- RTDX links. *See* links.
- RTW build options
  - generate\_code\_only 2-95
- run the EVM confidence test 2-115

## S

- sample rate for C6711 DSK 2-35
- saturate 2-38
- section, memory 2-149
- segment, memory 2-149
- select blocks for models 2-120
- set stack size 2-105
- set target preferences 2-40
- signed fixed-point numbers 4-4
- simulator
  - device cycle accurate 2-4
  - general use 2-5
  - use simulators for development 2-4
  - use with DSP/BIOS 2-4

- use with RTDX 2-4
- simulators, about 2-4
- solver option settings 2-82
- source and sink blocks 4-9
- stack size, set stack size 2-105
- stopping running models 2-119
- subsystem model code generation 2-60
- suitable applications for Embedded Target for TI C6000 DSP 1-3
- synchronous scheduling 2-72

## T

- table of blocks to avoid in models 2-120
- target Code Composer Studio 2-144
- target configuration options
  - build action 2-95
  - compiler verbosity 2-101
  - create .map files 2-105
  - generate code only 2-88
  - make command 2-88
  - memory model 2-99
  - overrun action 2-97
  - overrun notification method 2-97
  - retain .asm files 2-105
  - retain .obj files 2-105
  - symbolic debugging 2-104
  - system target file 2-86
  - template makefile 2-87
- target custom hardware 2-146
- target preferences
  - set 2-40
- target preferences block
  - about 2-40
- target preferences blocks in referenced models 2-109
- target specific optimization 2-92

- test your Embedded Target for TI C6000 DSP
  - installation
    - C6711 DSK 2-115, 2-137
- threshold interrupt
  - about 2-102
  - enable 2-102
- threshold, interrupt
  - set value 2-104
- timer, configure 2-45, 5-97, 5-129, 5-195, 5-223,  
5-253, 5-274, 5-292
- timer-based models, execution 2-72
- timer-based scheduler 2-72
- timing 2-71
- To Rtdx block 5-382
- tutorial for C6711 DSK blocks 2-123

## U

- uninitialized memory 2-149
- use blocks for the C6711 DSK 2-123
- use C62x DSP Library blocks 4-1
- use C6711 DSK blocks 2-123
- use c6711dsktest.mdl 2-116
- use logging in models 2-68

## V

- verify that c6711dsktest.mdl is running 2-118,  
2-140
- video capture setup 2-61
- video display setup 2-61

## W

- working directory 2-124
- wrap 2-38