

# Embedded Target for the TI TMS320C6000™ DSP Platform

**For Use with Simulink®**

- Modeling
- Simulation
- Implementation

## How to Contact The MathWorks:



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



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



508-647-7000	Phone
--------------	-------



508-647-7001	Fax
--------------	-----



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

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

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

© COPYRIGHT 2002 - 2003 by The MathWorks, Inc.

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

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

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

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

Printing History:	July 2002	Online only	New for Version 1.0 (Release 13)
	January 2003	Online only	New for Version 1.1
	September 2003	Online only	New for Version 2.0 (Release 13SP1+)

## Preface

---

<b>About Embedded Target for C6000 DSP</b> .....	<b>viii</b>
<b>Related Products</b> .....	<b>ix</b>
<b>Using This Guide</b> .....	<b>x</b>
Expected Background .....	<b>x</b>
Organization of the Document .....	<b>xi</b>
<b>Configuration Information</b> .....	<b>xiii</b>
<b>Typographical Conventions</b> .....	<b>xv</b>

## About the Embedded Target for TI C6000 DSP

---

**1**

<b>Introduction</b> .....	<b>1-2</b>
Suitable Applications .....	<b>1-3</b>
<b>Getting Started</b> .....	<b>1-4</b>
Platform Requirements—Hardware and Operating System ..	<b>1-4</b>
<b>Using Command Line Help</b> .....	<b>1-8</b>
Help for Embedded Target for TI C6000 DSP .....	<b>1-8</b>

<b>TI C6000 and Code Composer Studio IDE</b> .....	<b>2-4</b>
Supported Boards and Simulators .....	<b>2-4</b>
Typical Hardware Setup for Developing Models .....	<b>2-5</b>
<b>Using the C6000lib Blockset</b> .....	<b>2-8</b>
Configuring ADC Blocks .....	<b>2-13</b>
Configuring DAC Blocks .....	<b>2-18</b>
Configuring LED Blocks .....	<b>2-21</b>
Using the Overrun Indicator Feature .....	<b>2-22</b>
Configuring Reset Blocks .....	<b>2-23</b>
Creating DSP Application Models for Targeting .....	<b>2-24</b>
Using Logging in Your DSP Applications .....	<b>2-24</b>
Generating Code from Real-Time Models .....	<b>2-25</b>
<b>Setting Real-Time Workshop Build Options for C6000 Hardware</b> .....	<b>2-26</b>
Real-Time Workshop Options for C6000 Hardware .....	<b>2-26</b>
Target Configuration Options .....	<b>2-28</b>
Target Language Compiler Debugging Options .....	<b>2-30</b>
General Code Generation Category Options .....	<b>2-31</b>
TI C6000 Target Selection .....	<b>2-32</b>
TI C6000 Code Generation Options .....	<b>2-34</b>
TI C6000 Compiler Options .....	<b>2-37</b>
TI C6000 Linker Options .....	<b>2-38</b>
TI C6000 Run-Time Options .....	<b>2-42</b>
Embedded Target for TI C6000 DSP	
Default Project Configuration—custom_MW .....	<b>2-46</b>
<b>Targeting Your C6701 EVM</b> .....	<b>2-48</b>
Configuring Your C6701 EVM .....	<b>2-50</b>
Confirming Your C6701 EVM Installation .....	<b>2-51</b>
Testing Your C6701 EVM .....	<b>2-51</b>
Creating Your Simulink Model for Targeting .....	<b>2-55</b>
<b>C6701 EVM Tutorial 2-1—Single Rate Application</b> .....	<b>2-58</b>
Configuring Simulation Parameters for Your Model .....	<b>2-62</b>

<b>C6701 EVM Tutorial 2-2—A Multistage Application</b> . . . . .	<b>2-66</b>
<b>Targeting Your C6711 DSK</b> . . . . .	<b>2-80</b>
Configuring Your C6711 DSK . . . . .	<b>2-80</b>
Confirming Your C6711 DSK Installation . . . . .	<b>2-80</b>
Testing Your C6711 DSK . . . . .	<b>2-81</b>
<b>C6711 DSK Tutorial 2-3—Single Rate Application</b> . . . . .	<b>2-86</b>
Configuring Simulation Parameters for Your Model . . . . .	<b>2-91</b>
Running Models on Your C6711 DSK . . . . .	<b>2-94</b>
<b>C6711 DSK Tutorial 2-4—A More Complex Application</b> . . . . .	<b>2-97</b>
<b>Creating Code Composer Studio Projects Without Building</b> . . . . .	<b>2-109</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>
About Profiling Subsystems . . . . .	<b>3-10</b>
About the Profiling Report . . . . .	<b>3-11</b>
Interrupts and Profiling . . . . .	<b>3-12</b>
Reading Your Profile Report . . . . .	<b>3-13</b>
Definitions of Report Entries . . . . .	<b>3-14</b>
Profiling Your Generated Code . . . . .	<b>3-16</b>

To Enable Profiling for Your Generated Code . . . . .	3-17
To Create Atomic Subsystems for Profiling . . . . .	3-18
<b>Using DSP/BIOS with Your Target Application . . . . .</b>	<b>3-21</b>
To Enable DSP/BIOS When You Generate Code . . . . .	3-21

## 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 . . .	4-3
<b>Fixed-Point Numbers . . . . .</b>	<b>4-4</b>
Signed Fixed-Point Numbers . . . . .	4-4
Q Format Notation . . . . .	4-5
<b>Building Models . . . . .</b>	<b>4-8</b>
Converting Data Types . . . . .	4-8
Using Sources and Sinks . . . . .	4-9
Choosing Blocks to Optimize Code . . . . .	4-9

## Using FDATool with the Embedded Target for TI C6000 DSP

### 5

<b>Guidelines on Exporting Filters from FDATool to CCS IDE . . . . .</b>	<b>5-3</b>
Selecting the Export Mode . . . . .	5-4
Cautions Regarding Writing Directly to Memory . . . . .	5-4
Variables and Memory Necessary for Filter Export . . . . .	5-5
Selecting the Export Data Type . . . . .	5-7
<b>Tutorial—Exporting Filters from FDATool to CCS IDE . . . . .</b>	<b>5-9</b>
Task 1—Export Filter by Generating a C Header File . . . . .	5-9
Task 2—Export Filter by Writing Directly to Target Memory . . . . .	5-15

**6**

**Using the Embedded Target for C6000 DSP**

**Block Reference** . . . . . **6-2**

**Blocks—By Library** . . . . . **6-3**

    Embedded Target for C6000 DSP Blocks in  
    Library c6701evmlib . . . . . **6-3**

    Embedded Target for C6000 DSP Blocks in  
    Library c6711dsklib . . . . . **6-3**

    Embedded Target for C6000 DSP Blocks in Library rtdxblocks **6-4**

    Embedded Target for C6000 DSP in the C62x DSP Library . . . **6-4**

**Blocks—Alphabetical List** . . . . . **6-7**

**Hardware Supported by the  
Embedded Target for TI C6000 DSP**

**A**

**Supported Hardware For Targetting** . . . . . **A-2**





# Preface

---

About Embedded Target for C6000 DSP (p. viii)

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

Related Products (p. ix)

Lists a few products that you might find can expand the way you use the Embedded Target for TI C6000 DSP

Using This Guide (p. x)

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

Configuration Information (p. xiii)

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

Typographical Conventions (p. xv)

Lists the conventions we use in this User's Guide to distinguish between kinds of information, such as menu selections or MATLAB command line functions

## 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 DSP 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.

## Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Embedded Target for TI C6000 DSP.

For information about the products and hardware you need to run the Embedded Target for TI C6000 DSP, refer to “Getting Started” on page 1-4.

For more information about any of these products, refer to either

- The online documentation for that product, if it is installed or you are reading the documentation from the CD
- The MathWorks Web site, at <http://www.mathworks.com>. Navigate to the Products area

---

**Note** The toolboxes listed below include functions that extend MATLAB capabilities. The blocksets include blocks that extend Simulink capabilities.

---

Product	Description
Control System Toolbox	Design and analyze feedback control systems
Data Acquisition Toolbox	Capture and send data from plug-in data acquisition boards
DSP Blockset	Design and simulate DSP systems
Filter Design Toolbox	Design and analyze advanced floating-point and fixed-point filters
Image Processing Toolbox	Perform image processing, analysis, and algorithm development

## Using This Guide

### 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 TMS320C6701 Evaluation Module. To get the most out of this manual, you should be familiar with MATLAB and its associated programs, such as DSP Blockset and Simulink. We do not discuss details of digital signal processor operations and applications, except to introduce concepts related to using the C6701 EVM or C6711 DSK. 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, “About the 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 both your C6701 EVM or 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. Visit “Confirming Your C6701 EVM Installation” on page 2-51, to confirm that you installed and configured your C6701 EVM board to meet the needs of Embedded Target for C6000 DSPs. For C6711 DSK users, refer to “Configuring Your C6711 DSK” on page 2-80 for more information about installing and using your C6711 DSK.

## Organization of the Document

Chapter	Description
“About the Embedded Target for TI C6000 DSP”	Introduces the Embedded Target for C6000 DSPs and the related products that may be of interest.
“Targeting C6000 DSP Hardware”	Contains information about using the blocks and build software to create applications for the C6701 digital signal processor on the C6701 EVM and the C6711 DSK. Two example Simulink models introduce the targeting software for both targets, the target-specific blocks, the build files, and give you a glimpse of the range of applications supported by the Embedded Target for TI C6000 DSP. Most of the information here also applies to C6416 DSK and C6713 DSK targets.
“Targeting with DSP/BIOS™ Options”	Describes how to use the TI DSP/BIOS™ real-time operating system in your CCS projects and the generated code for your hardware targets.

<b>Chapter</b>	<b>Description</b>
“Using the C62x and C64x DSP Libraries”	Provides details about fixed-point arithmetic and the fixed-point blocks in the C62x and C64x DSP libraries. You use the fixed-point blocks to develop Simulink™ models and generate code targeted to the C62xx or C64xx family of digital signal processors.
“Using FDATool with the Embedded Target for TI C6000 DSP”	Describes how to use FDATool to export filters to your target
“Block Reference”	Provides reference information for the blocks in the Embedded Target for TI C6000 DSP. All the blocks are included in the blockset c6000lib.

## 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:

- C6701 EVM Board Support
- C6711 DSK Board Support
- C62x DSP Library
- RTDX Instrumentation

If you do not see the listed libraries, or MATLAB does not recognize the command, you need to 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
1	C6xxx Simulator (Texas Instrum ...	TMS320C6701	0	6701

0 C6x11 DSK (Texas Instruments) 0 CPU  
TMS320C6x1x

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.



# Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	Monospace font	To assign the value 5 to A, enter <code>A = 5</code>
Function names, syntax, filenames, directory/folder names, and user input	Monospace font	The <code>cos</code> function finds the cosine of each array element. Syntax line example is <code>MLGetVar ML_var_name</code>
Buttons and keys	<b>Boldface</b> with book title caps	Press the <b>Enter</b> key.
Literal strings (in syntax descriptions in reference chapters)	<b>Monospace bold</b> for literals	<code>f = freqspace(n, 'whole')</code>
Mathematical expressions	<i>Italics</i> for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$ .
MATLAB output	Monospace font	MATLAB responds with <code>A =</code> <code>5</code>
Menu and dialog titles	<b>Boldface</b> with book title caps	Choose the <b>File Options</b> menu.
New terms and for emphasis	<i>Italics</i>	An <i>array</i> is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	<code>[c, ia, ib] = union(...)</code>
String variables (from a finite list)	<i>Monospace italics</i>	<code>sysc = d2c(sysd, 'method')</code>



# About the Embedded Target for TI C6000 DSP

---

Introduction (p. 1-2)

Introduces the Embedded Target for TI C6000 DSP, some of the features and supported hardware

Getting Started (p. 1-4)

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

Using Command Line Help (p. 1-8)

Describes how you get help for functions in the Embedded Target for TI C6000 DSP

## Introduction

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.

Using the Embedded Target for TI C6000 DSP and Real-Time Workshop, you can create executable code for the following boards:

- C6416 DSP Starter Kit from Texas Instruments
- C6701 Evaluation Module from Texas Instruments, revision 1 or later
- C6711 DSP Starter Kit from Texas Instruments
- C6713 DSP Starter Kit from Texas Instruments
- TMDX326040A Daughter card for the C6711 DSK. Also known as the PCM3003 Audio Daughter Card.

Additionally, one of the Real-Time Workshop build options builds a Code Composer Studio® project from the C code generated by Real-Time Workshop.

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 1-4

## 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. Targetting does not work with continuous time solvers.

## 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 Microsoft Windows NT 4.0 Workstation and Server, 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
- One full-length peripheral component interface (PCI) slot available to use the C6701 EVM internally in your PC
- CD-ROM drive
- Microsoft Windows NT 4.0 Server or Workstation, Windows 2000, or Windows XP. Note that the C6416 DSK and C6713 DSK do not work on Microsoft Windows NT platforms.

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 Table 1-1. In all cases, Embedded Target for TI C6000 DSP requires that you install the latest versions of the required software.

**Table 1-1: 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
MATLAB 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
DSP Blockset	Block libraries used by Simulink

For information about the software required to use the MATLAB 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, the latest version, installs the software shown.

**Table 1-2: Required TI Software for Targeting Your TI C6000 Hardware**

<b>Installed Product</b>	<b>Additional Information</b>
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 Texas Instruments TMS320C6701 Evaluation Modules
- 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 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.

## Using Command Line Help

How you get command line help on a function depends on whether the function is overloaded. When a function name is used in multiple products, such as in the Embedded Target for TI C6000 DSP and MATLAB, it is said to be an overloaded function.

### Help for Embedded Target for TI C6000 DSP

To get general help for using the Embedded Target for TI C6000 DSP, use the help feature in MATLAB.

Type

```
help tic6000
```

at the command prompt to get a list of the functions and block libraries included in the Embedded Target for TI C6000 DSP. Or select **Help ->Full Product Family Help** from the menu bar in the MATLAB desktop. When you see the Table of Contents in Help, select Embedded Target for TI C6000 DSP.

# Targeting C6000 DSP Hardware

---

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.
Using the C6000lib Blockset (p. 2-8)	Describes the contents of the C6000lib blockset—what blocks are included and where
Setting Real-Time Workshop Build Options for C6000 Hardware (p. 2-26)	Provides the details on setting the Real-Time Workshop options when you generate code from your Simulink models to TI hardware
Targeting Your C6701 EVM (p. 2-48)	If you are targeting a C6701 EVM, this section details specific information about using the target
C6701 EVM Tutorial 2-1—Single Rate Application (p. 2-58)	Takes you through the process of creating models in Simulink and generating code for your C6701 EVM
C6701 EVM Tutorial 2-2—A Multistage Application (p. 2-66)	Using a more complex model than the previous tutorial, this exercise walks you through code generation for a multistage model
Targeting Your C6711 DSK (p. 2-80)	If you are targeting a C6711 DSK, this section details specific information about using the target
C6711 DSK Tutorial 2-3—Single Rate Application (p. 2-86)	Takes you through the process of creating models in Simulink and generating code for the C6711 DSK

C6711 DSK Tutorial 2-4—A More Complex Application (p. 2-97)

Using a more complex model than the previous C6711 DSK tutorial, this exercise walks you through code generation for a multistage model

Creating Code Composer Studio Projects Without Building (p. 2-109)

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.

---

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 Texas Instruments (TI) C6701 Evaluation Module (C6701 EVM) and C6711 DSP Starter Kit (DSK). In combination with the supported boards, your Embedded Target for TI C6000 DSP software is the ideal resource for rapid prototyping and developing embedded systems applications for the TI C6701 and 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.

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.

# 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 to invoke the code building process from within CCS. Once 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 “Hardware Supported by the Embedded Target for TI C6000 DSP” for the latest information about the hardware supported by the Embedded Target for TI C6000 DSP.

## About Simulators

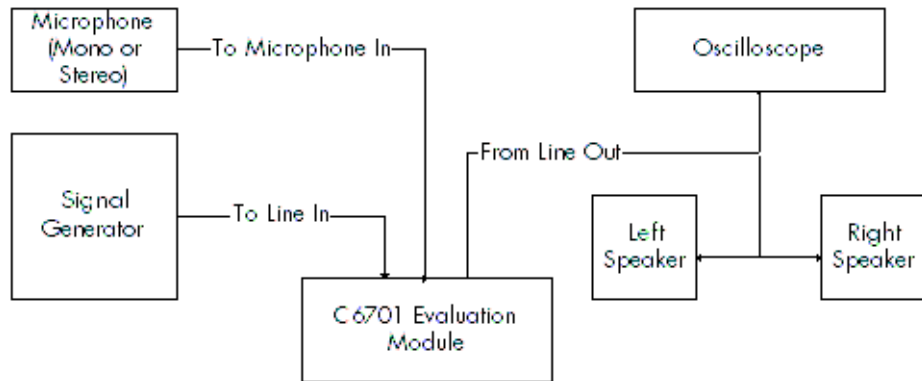
CCS offers many simulators for the C6701 and C6711 digital signal processors, in the CCS Setup utility. Much of your model and algorithm development efforts work with the simulators, such as code generation. And, since the Embedded Target for TI C6000 DSP provides a software-based scheduler, your models 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, you must match the processor on your target exactly to simulate your target hardware. If you plan to target C6701 EVM boards, your simulator must contain a C6701 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.

## Typical Hardware Setup for Developing Models

The next figure presents a block diagram of the typical setup for the inputs and output for the C6701 EVM. For the C6711 DSK, the typical layout is similar except the board accepts only monaural input from a microphone.



### Block Diagram of Typical Inputs and Outputs to the C6701 EVM

After you have installed one or more of the supported development boards shown in “Hardware Supported by the Embedded Target for TI C6000 DSP” 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:

- C6701 EVM Board Support blocks
  - C6701 EVM ADC—configures the analog to digital converter

- C6701 EVM DAC—configures the digital to analog converter
- C6701 EVM DIP Switch—simulates or reads the user-defined DIP switches on the C6701 EVM
- C6701 EVM LED—controls the user status light emitting diodes (LED) on the C6701 EVM
- C6701 EVM Reset—resets the current C6701 EVM
- C6711 DSK Board Support blocks
  - C6711 DSK ADC—configures the analog to digital converter
  - C6711 DSK DAC—configures the digital to analog converter
  - C6711 DSK DIP Switch—simulates or reads the user-defined DIP switches on the C6711 DSK
  - C6711 DSK LED—controls the three user status light emitting diodes (LED) on the C6711 DSK
  - C6711 DSK Reset—resets the current C6711 DSK
- RTDX blocks for C6000 hardware
  - From RTDX—adds an RTDX input channel to the code generated from your model. When you run the model on your target, the block code imports data from your host over an RTDX channel to your running process.
  - To RTDX—adds an RTDX output channel to the code generated from your model. When you run the model on your target, the block code exports data to your host over an RTDX channel from your running process.
- C62x DSPLIB blocks
  - Provides fixed-point blocks for models that use fixed-point mathematics and algorithms.

These I/O blocks are associated with your boards. As needed, add the devices to your model. If you choose not to include either an ADC or DAC block in your model, 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.

In addition to the blocks for specific boards, the C6000lib blockset includes the library rtdxBlocks that contains RTDX input and output blocks that apply to all C6000 development boards.



With your model open, select **Simulation Parameters** from the **Simulink** option to open the **Simulink Parameters** dialog box. From this dialog, click **Real-Time Workshop**. You must specify the appropriate versions of the system target file and template makefile. For the C6701 EVM or the C6711 DSK, in the **Real-Time Workshop** pane of the dialog, specify

```
ti_c6000.tlc
```

to select the correct target file. Or click **Browse** and select `ti_c6000.tlc` from the list of targets.

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 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 via the peripheral component interface (PCI) bus to your C6701 evaluation module, or 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.

### Using the C6000lib Blockset

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

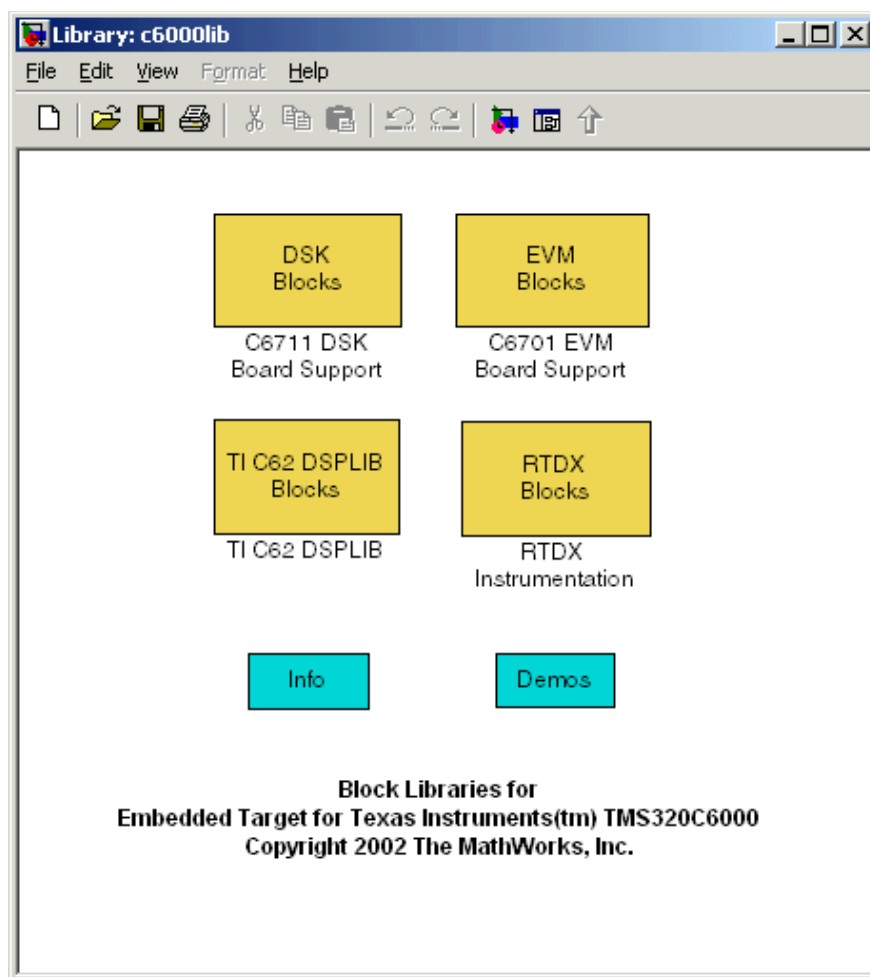
- C6701 EVM Board Support—blocks to configure the codec and LEDs on the C6701 EVM
- C6711 DSK Board Support—blocks to configure the codec and LEDs on the C6711 DSK
- RTDX Instrumentation—blocks for adding RTDX communications channels to Simulink models
- TI C62x DSPLIB—fixed-point blocks for developing models for fixed-point targets

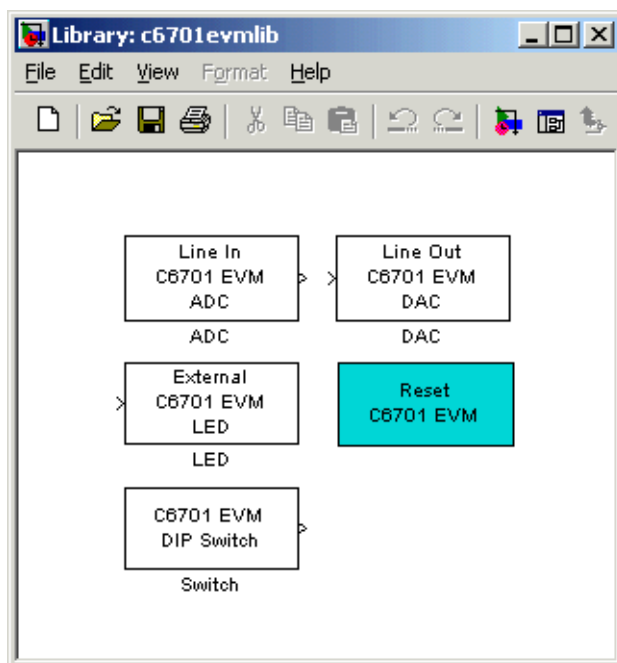
Each block library appears in one of the next figures. The sections after the figures review the configuration options for blocks in the EVM and DSK block libraries. For more information about the RTDX blocks, refer to “Constructing Link Object”s in your MATLAB Link for Code Composer Studio documentation.

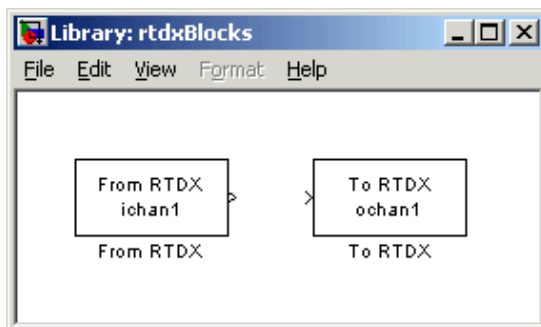
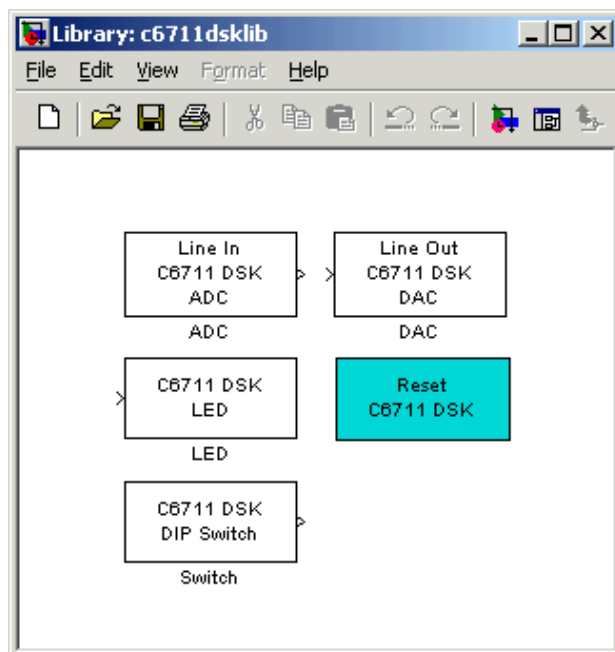
Each board-based block library contains a version of each of these blocks:

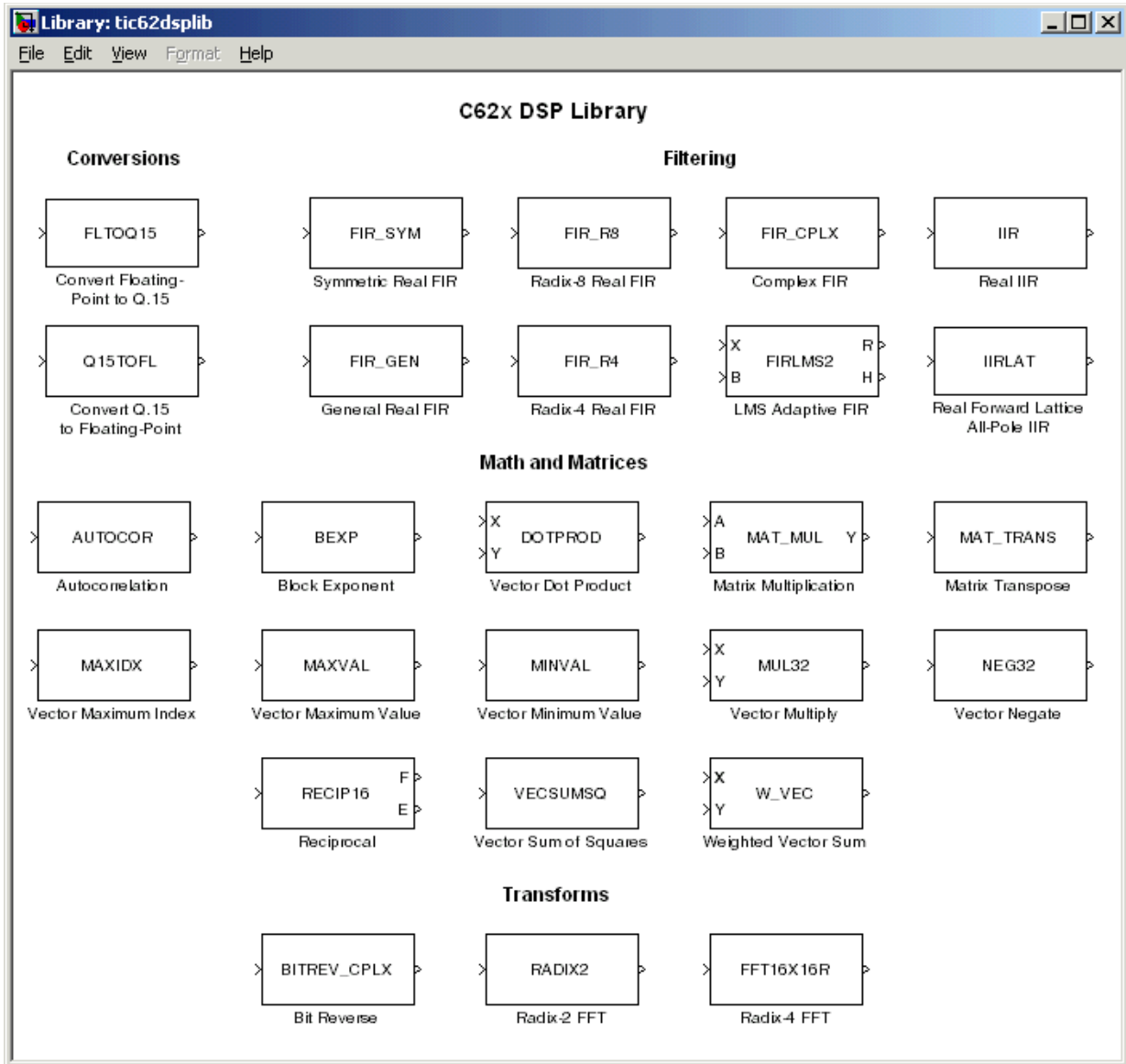
- ADC block
- DAC block
- DIP Switch block (refer to the reference page for the DIP Switch block for your target, either C6701 EVM or C6711 DSK)
- LED block
- Reset block

Similarities in the C6000 boards result in the ADC, DAC, DIP Switch, LED, and Reset blocks for the C6701 and C6711 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. For example, the **Codec data format** option for ADC blocks applies only to the C6701 EVM ADC.









## Configuring ADC Blocks

To drive and test your DSP application on a C6701 EVM or 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 DSP 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.

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 option listed with the target board hardware affected.

<b>Option</b>	<b>Affected Hardware</b>
<b>ADC Source</b>	Codec
<b>Codec data format</b> (C6701 EVM ADC only)	Codec
<b>Mic</b>	Codec
<b>Output data type</b>	TMS320C6xxx digital signal processor
<b>Sample rate (Hz)</b> (C6701 EVM ADC only)	Codec
<b>Samples per frame</b>	Direct memory access functions
<b>Scaling</b>	TMS320C6xxx digital signal processor
<b>Source gain (dB)</b>	Codec
<b>Stereo</b> (C6701 EVM ADC only)	Codec

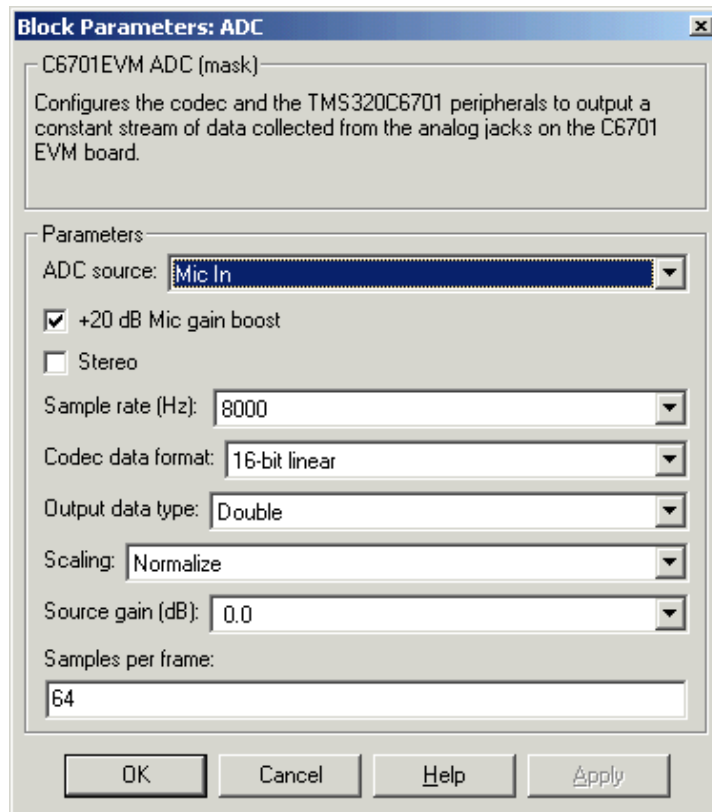
### Selecting the ADC Source

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 (C6701 EVM ADC Block Only)

To open the **Block Parameters** dialog, right-click the C6701 EVM ADC block in your Simulink model and select **Block Parameters** from the context menu. You see the dialog presented in Figure 2-1, Block Parameters for C6701 EVM ADC Dialog.





**Figure 2-1: Block Parameters for C6701 EVM ADC Dialog**

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. In addition, choose double-precision, fixed-point arithmetic format when you select the **Codec data format**.

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 one of the 8-bit data formats that use a compressed format such as A-law, best matches current standards.

### Choosing the Codec Data Format (C6701 EVM ADC Only)

When the codec performs A/D conversion, the output data format is partly determined by the setting for **Codec data format** in the **Block Parameters** dialog. **Codec data format** offers five choices:

- 16-bit linear—the standard method of representing 16-bit digital audio. Provides 96 dB theoretical dynamic range and best fits the standard for compact disk audio players. -32768 represents the maximum negative analog amplitude and 32767 represents the maximum positive analog amplitude.
- 8-bit linear—commonly used in the PC industry. Provides 48 dB theoretical dynamic range. 00 represents the maximum negative analog amplitude and 255 represents the maximum positive analog amplitude.
- 8-bit A law—used in the telephone industry most frequently. A-law is the European standard;  $\mu$ -law is the standard in Japan and the United States. Uses a nonlinear companding transfer function to digitize analog input to provide 64 dB or 72 dB maximum dynamic range.
- 8-bit  $\mu$  law—used in the telephone industry most frequently.  $\mu$ -law is the standard in Japan and the United States; A-law is the European standard. Uses a nonlinear companding transfer function to digitize analog input to provide 64 or 72 dB maximum dynamic range.
- 4-bit IMA ADPCM—voice digitization scheme that uses a lower bit rate than pulse code modulation (PCM). ADPCM records only the difference between samples, and adjusts the coding scale dynamically to accommodate large and small differences. This scheme is simple to implement, but can introduce significant noise. The G.721 method uses 32Kbps per voice channel, as compared to standard telephony's 64Kbps using PCM. Using ADPCM

instead of PCM is imperceptible to humans—but it can significantly reduce the throughput required by higher-speed modems and fax transmissions.

You will probably use 16-bit linear codec data format when you begin developing your model. If you do not care about, or do not expect, negative data values, as would be the case where you are measuring a voltage that varies from 0 volts to 5 volts, you could use 8-bit unsigned math. If appropriate for your application, choose one of the compression data formats for your model.

### Selecting the Data Type

You must select the data type when you include the ADC block in your DSP simulation. The data type you use in the simulation is likely not to be the one you use when you build and download the application to the target. Your choice of data type depends on a number of factors related to how the application runs on the target DSP and what limitations apply. Four factors can influence your choice:

- Processing time—how long does each iteration of your process take? Can the DSP process the data quickly enough to meet your needs?
- Power used—how much power does the application require? Doing lots of multiply and divide operations uses more power and generates more heat than add operations.
- Memory required—how does your application use memory and how much does it need?
- Does accuracy matter—do you need the accuracy provided by double-precision arithmetic or is fixed-point or integer acceptable?

When you have developed and tested your signal processing application in Simulink, you are ready to use Real-Time Workshop and your Embedded Target for TI C6000 DSP to build and download your model to the C6701 EVM or C6711 DSK. Your Simulink model should represent a general purpose implementation of your application, without specific features that depend on the target DSP. For instance, use floating-point arithmetic and single- or double-precision format to develop your simulation. Select DSP target-specific data and format requirements when you prepare your model for the board.

---

**Note** If you use fixed-point arithmetic in your processing algorithm or application on the C6701 EVM or C6711 DSK, verify that the blocks in your Simulink model will work with the processor in integer mode.

Blocks in the C62 DSPLIB library are designed specifically for fixed-point models.

---

To ease model development on the target, start by selecting floating-point data format when you build and download your application to the board. You could choose either single-precision or double-precision at this time to ensure that your model runs on the target processor. Often, normalized floating-point arithmetic is the best choice during development. As you tune your model for your target processor, consider whether such factors such as calculation time and memory use are important in your application environment. If time is a critical parameter, use either less precise math, such as single precision instead of double, or switch from floating point to fixed point or integer. Making these changes both speeds up your process and reduces the memory and power your processor consumes to complete its calculations.

For accuracy without regards to time, use floating-point arithmetic.

### Selecting the Scaling

Both the single-precision and double-precision data types are available as floating point or normalized values. During development, it is a good idea to start with normalized values to relieve you from worrying about overflows and underflows in the calculations performed in the algorithm. When you are happy that the process is under control, change the data type to the one you need for the deployed executable code.

Selecting a floating-point data type can reduce your algorithm processing overhead slightly.

### Configuring DAC Blocks

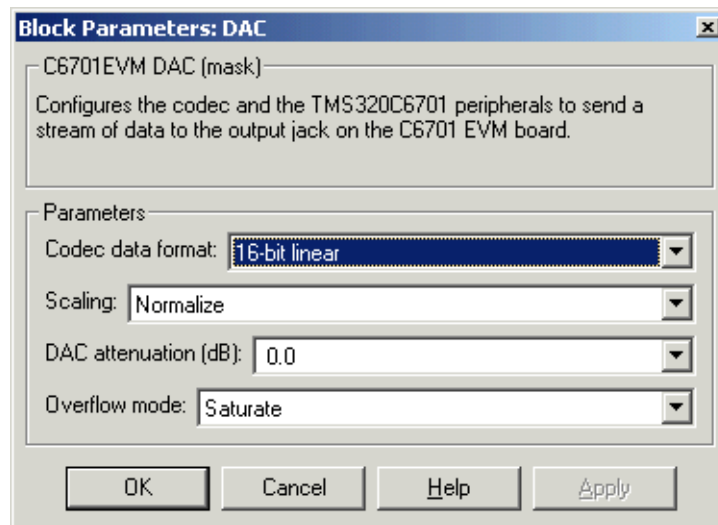
In most cases, DAC blocks inherit attributes from the ADC block in the model, or from the previous nonvirtual block. You must select **Codec data format**, **Scaling**, and **Overflow mode** when you use a C6701 EVM DAC block in your

model. 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.

Option	Affected Hardware
<b>Codec data format</b> (C6701 EVM DAC only)	Codec
<b>DAC attenuation</b>	Codec
<b>Overflow mode</b>	Digital Signal Processor
<b>Scaling</b>	Digital Signal Processor

When you double-click the C6701 EVM DAC block, you see the dialog shown here.



### Choosing the Codec Data Format (C6701 EVM DAC Block Only)

The **Codec data format** for the C6701 EVM DAC block must be the same as the **Codec data format** for the C6701 EVM ADC block, if you use one in your model. C6711 DSK codec blocks do not offer the **Codec Data Format** option.

### 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 ADC block. When you select **Integer** value for the scaling, the signal ranges between the minimum and maximum values representable by the number of bits specified by **Codec data format**.

### 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 **DAC** dialog 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

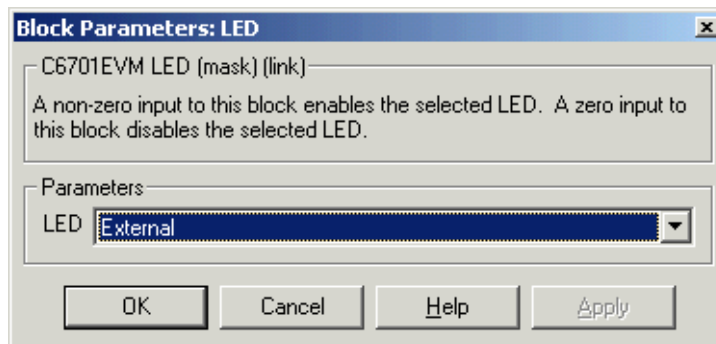
You use the LEDs on the evaluation module as indicators for your process. For example, you might use an LED to indicate that your algorithm has completed a specified calculation or reached a particular point in the processing.

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.

Since the C6701 EVM provides two LEDs, you can use two C6701 EVM LED blocks—one for each user status LED on the board. 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.

### Select the Target LED (C6701 EVM LED Block Only)

Double-clicking the C6701 EVM LED block opens the dialog shown here.



When you add an C6701 EVM LED block to your model, you use the **LED** list to select which LED the block controls—internal or external. Select **External** from the list to have the C6701 EVM LED block trigger the LED located on the C6701 EVM mounting bracket at the back of your PC. To trigger the internal LED located on the C6701 EVM board internally, select **Internal** for the **LED** setting.

### 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 C6701 EVM 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 on 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.

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



---

**Note** The **Overflow 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 overflow indicator, consider not using an LED block to trigger the external LED on the C6701 EVM, or the user LEDs on the C6711 DSK until you stop monitoring your process for overflow conditions.

---

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

- **None**—your model does not respond to overflow conditions during processing.
- **Notify\_and\_continue**—when your model runs out of clock cycles before completing enough of the processing algorithm, the overflow indicator executes the option you chose for **Overflow 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 overflow using the method you select for the **Overflow notification method**.

Overflow notification method provides three ways to tell you when an overflow has occurred:

- **Print\_message**—print a message to stdout, or the message log when your application uses DSP/BIOS
- **Turn\_on\_LEDs**—illuminate the user LEDs on the target, either the external LED on the C6701 EVM or all of 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, Resets 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, it 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.

### 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 DSP Blockset
- Use discrete time blocks from Simulink
- Use blocks from any other blockset that meet your needs and operate in the discrete time domain

### Using Logging in Your DSP Applications

Simulink offers various data logging capabilities in the **Simulation Parameters** dialog for your model. Found on the **Workspace I/O** pane of the **Simulation Parameters** dialog, 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 **Workspace I/O** pane in the **Simulation Parameters** dialog in your model.

#### To Turn Off Logging in Your Model

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

- 1 Select **Simulation->Simulation Parameters** from the menu bar in your model.
- 2 Click the **Workspace I/O** tab to open the **Workspace I/O** 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 **Workspace I/O** options in Real-Time Workshop 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 **Workspace I/O** options in the **Simulation Parameters** dialog 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 **Simulation Parameters** dialog. 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 **Simulation Parameters** dialog, `gmake` can download and execute the model on your TI target board.

### Setting Real-Time Workshop Build 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.

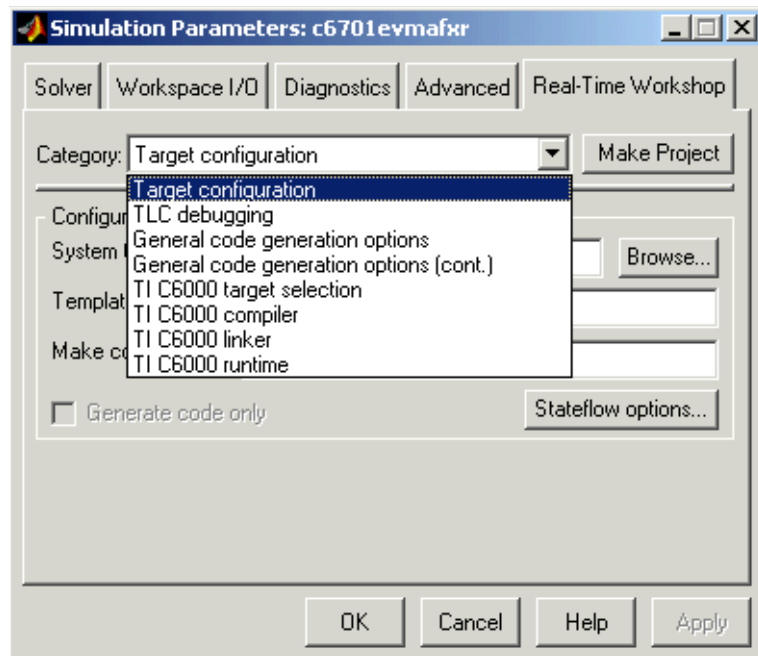
---

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

---

### Real-Time Workshop Options for C6000 Hardware

The **Real-Time Workshop** pane of the **Simulation Parameters** dialog lets you set several options for the real-time model. To open the **Simulation Parameters** dialog, select **Simulation -> Simulation 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.



On the **Real-Time Workshop** pane, the **Category** list provides access to the options you use to control how Real-Time Workshop builds and runs your model. Nine sets of options for configuring Real-Time Workshop target builds make up the categories—the first four apply to all Real-Time Workshop targets and always appear on the list.

The last four categories in the list shown are specific to the Embedded Target for TI C6000 DSP target `ti_C6000.tlc` and appear when you select the TI C6000 target:

- Target configuration—Real-Time Workshop general configuration options
- TLC debugging—Real-Time Workshop general debugging options
- General code generation options—Real-Time Workshop general code generation options
- General code generation options (cont.)—more Real-Time Workshop general code generation options
- TI C6000 target selection—target-specific options

- TI C6000 code generation—target-specific code generation options
- TI C6000 compiler—target-specific compiler options
- TI C6000 linker—target-specific linker options
- TI C6000 runtime—target-specific run-time options

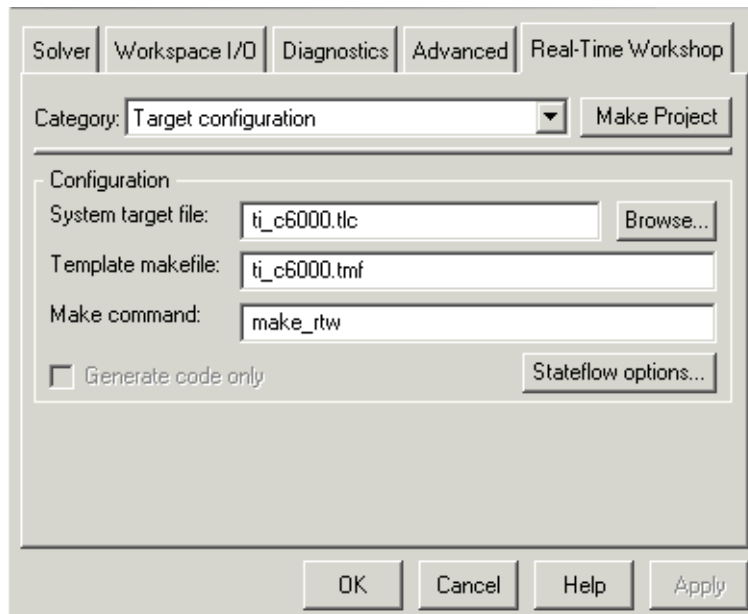
When you select your target in Target Configuration, the options change in the list. For the Embedded Target for TI C6000 DSP, the target to select is `ti_c6000.tlc`. When you make the change, the options that apply just to TI hardware get added to the list as shown.

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

### Target Configuration Options

Use the selections in this category to determine your target, either C6000 or some other target if you are not using the Embedded Target for TI C6000 DSP.

When you select the C6000 target (`ti_c6000.tlc`) in the category Target configuration, you enable the Automatic option for **Board and processor selection method** that appears as a control in the category TI C6000 target selection for your model as well. After that, opening the **Simulation Parameters** dialog 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.



**System target file.** Clicking **Browse** opens the Target File Browser where you select `ti_c6000.tlc` as your **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 will fill in the **Template makefile** and **Make command** selections.

**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**. On **Configuration** in the Target configuration category, enter `make_rtw` for the **Make command**. Parameters you set in this dialog belong to the model you are building. They are saved with the model and stored in the model file.

**Generate code only.** This option does not apply to targetting 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. Then, for **Build action**, select `Generate code only`.

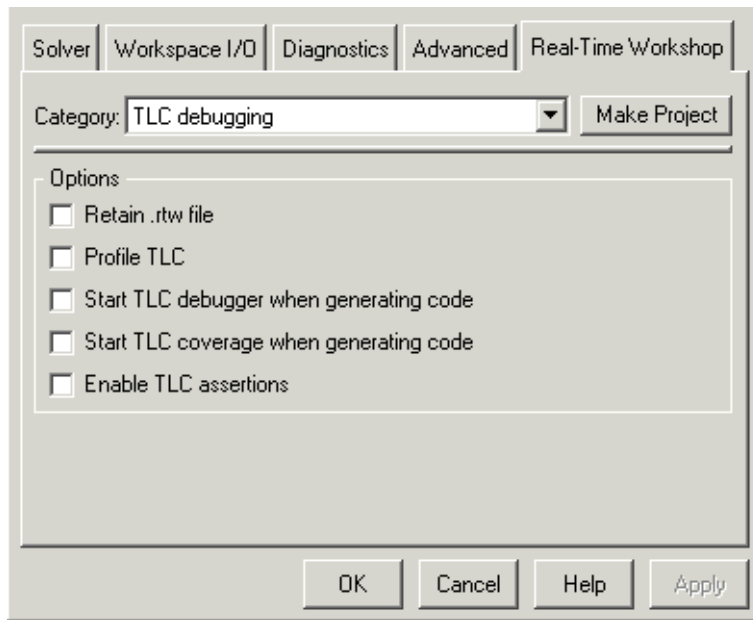
### Target Language Compiler Debugging Options

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 TLC debugging from the **Category** list, you see an **Options** pane as shown in the next figure. Within this, you set options that are specific to TLC debugging.

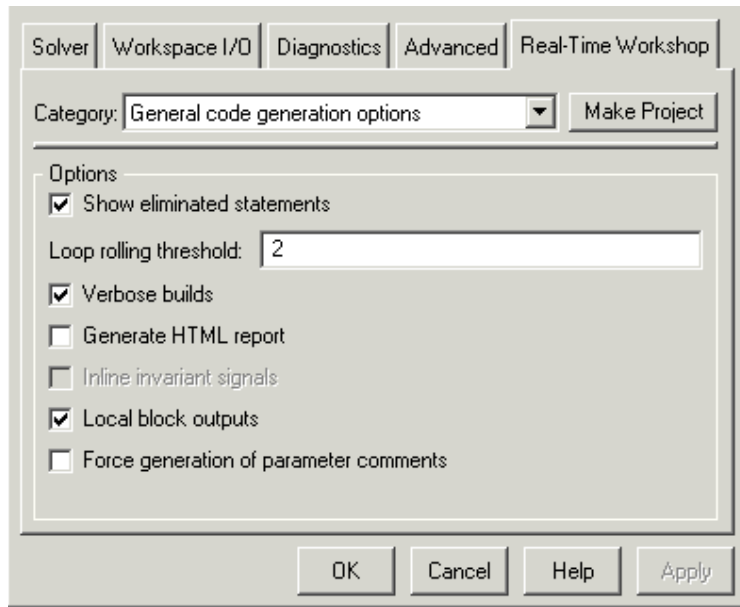




For details about using the options in TLC debugging, refer to the section “About TLC Debugger” in your Real-Time Workshop documentation.

## General Code Generation Category Options

From the **Real-Time Workshop** pane of the **Simulation Parameters** dialog, 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 General code generation options from the **Category** list on the **Real-Time Workshop** pane. The figure shows the General code generation options pane when you select the system target file `ti_C6000.tlc` under Target configuration.



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

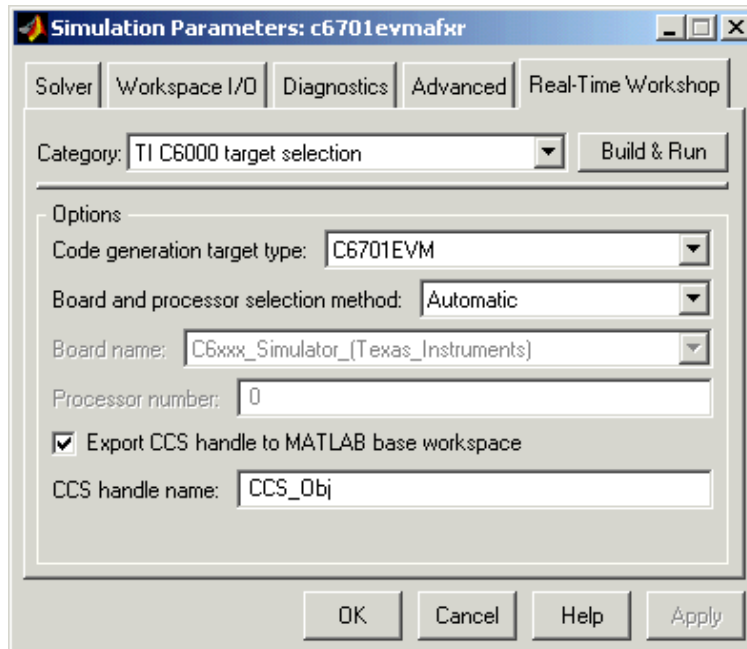
- **Show eliminated statements**
- **Loop rolling threshold**
- **Verbose builds**
- **Inline invariant signals**
- **Local block outputs**
- **Force generation of parameter comments**

For more information about using these options and the General code generation options options, refer to your Real-Time Workshop documentation.

### TI C6000 Target Selection

From this Real-Time Workshop category you select your specific C6000 family target using the **Code generation target type** option:

- C6701EVM for the C6701 Evaluation Module
- C6711DSK for the C6711 DSP Starter Kit



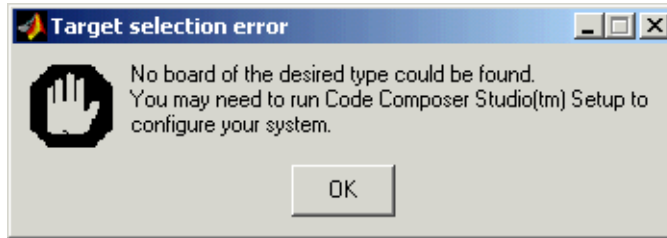
One feature of this pane lets you direct the Embedded Target for TI C6000 DSP to determine your target for you. When you choose **Automatic** for the **Board and processor selection method**, the Embedded Target for TI C6000 DSP runs `ccsboardinfo` when you open the **Simulation Parameters** dialog. `ccsboardinfo` determines what boards you have set up in CCS. If you have a board of the type you selected in **Code generation target type**, the Embedded Target for TI C6000 DSP identifies the board, reports the board name and processor number in this dialog, and uses that board as your target.

With the **Automatic** method selected for a Simulink model, each time you start the **Simulation Parameters** dialog from the model, the board selection process searches for a board of the type you selected in **Code generation target type**. To prevent this search, switch to **Manual** for the selection method.

Selecting **Manual** enables the **Board name** and **Processor number** options in the pane. Select your board and processor manually when these options are available. If you are not sure about which boards and processors you have, run `ccsboardinfo` at the MATLAB prompt to see which boards are installed and recognized in CCS. When you have more than one board of a type, such as two

C6701 EVM boards, you may need to use the manual selection option to target the correct board. In the Automatic selection mode, the Embedded Target for TI C6000 DSP targets the first instance of the selected target type.

Simulators do not work in the Automatic mode. If you use simulators, the target search does not work and returns an error like the following.



To avoid this error, set the **Board and processor selection method** for the model to Manual when you do not have actual boards as targets.

### Export CCS Handle to MATLAB Base Workspace

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 `ccdsp`, 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 work space, giving it the name you assign in **CCS handle name**.

After you build your model with the **Export handle to MATLAB base workspace** option selected, you see the CCS object in your MATLAB workspace browser with the name you provided and class type `ccdsp`.

### TI C6000 Code Generation Options

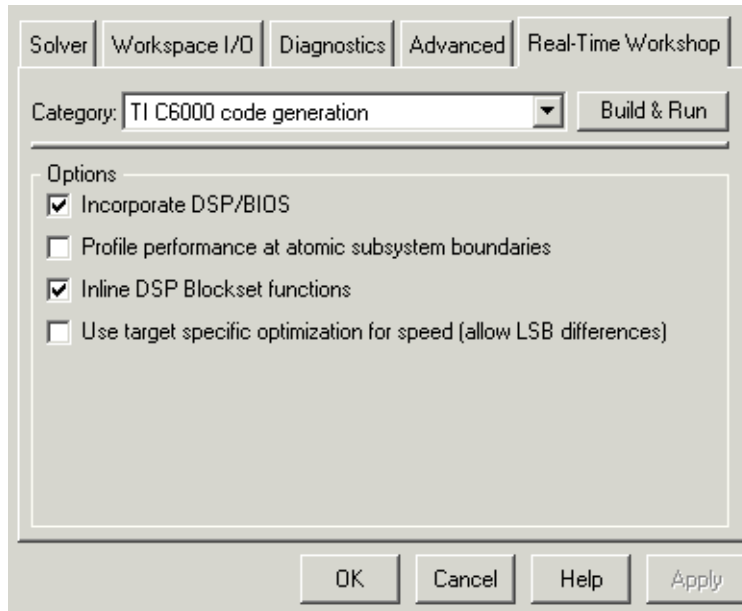
From this category, you choose from four options that define the way your code gets generated:

- **Incorporate DSP/BIOS**
- **Profile performance at atomic subsystem boundaries**
- **Inline DSP Blockset 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 include 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 opt to include DSP/BIOS.

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 DSP Blockset functions** tells the compiler to inline each DSP 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 may not make any difference in some models.

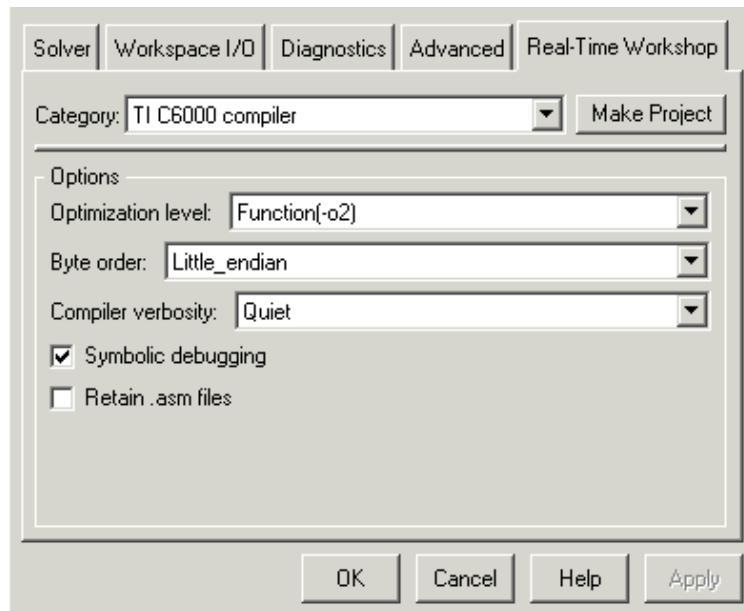
Notice that selecting target specific optimization allows your generated code to differ from your simulation results in the least significant bit (LSB) for the outputs of optimized blocks. You should review the results of the optimized and simulation-true code to see that they are sufficiently close 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 optimization improved the performance.

## TI C6000 Compiler Options

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

If you change the settings in this dialog, your changes become part of the build configuration options for your project in CCS. You can edit these settings in CCS to change them later. In the dialog, as presented in the figure, the options under **Options** let you configure compiler operations.



**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)`.

**Byte order.** The supported boards provide the ability of the digital signal processors to run in little-endian or big-endian mode. While your choice of Big-endian or Little-endian operation does not directly affect your algorithm processing, it does determine the ability of the processor to communicate with other processors. When you choose which byte ordering to use, select the option that matches other processors your board might need to communicate with. For example, most reduced instruction set compiler-based processors and Motorola processors—the PowerPC—use the big-endian configuration. Intel processors are little-endian machines. By default, the compiler generates code in little-endian format.

**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

**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.

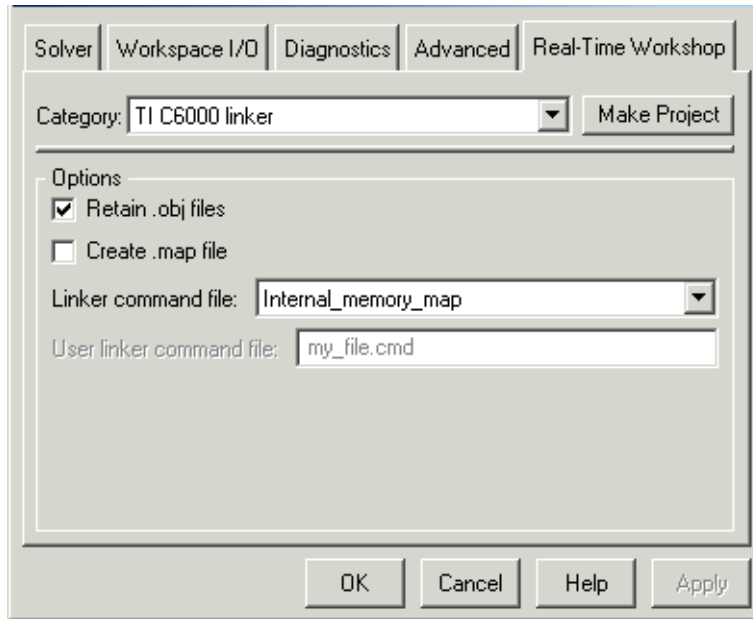
**Retain .asm files.** Select this option to direct Real-Time Workshop and the Embedded Target for TI C6000 DSP to save your assembly language (.asm) files after creation. The Embedded Target for TI C6000 DSP does not retain .asm files by default. If you choose to keep the .asm files, Real-Time Workshop 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.

## TI C6000 Linker Options

As shown in the figure, you can configure the TI C6000 linker to perform certain operations and use specified files. Note that the linker, not the compiler, defines the memory map used and allocated 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 and models on the target processor.



**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 C6701 EVM or 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 modelname.map. When you clear this check box, the linker does not produce the listing. New projects do not create the .map file.

**Linker command file.** Specifies the linker command file to use when the linker runs. Linker command files contain linker or hex conversion utility

options and the names of input files to the linker or hex conversion utility. You select one of the file options from the list:

- **User\_defined**—lets you use your own linker command file. Developing a custom `.cmd` file lets you set linker options that can be more appropriate for your needs. When you select this option, you enable the **User linker command file** option where you enter the name of your linker command file with the `.cmd` file extension.

If you enter your command filename with a full path description, the Embedded Target for TI C6000 DSP locates your file immediately when you build your model in Real-Time Workshop. If you supply a filename without the full path, the Embedded Target for TI C6000 DSP searches your MATLAB working directory for the file when you build your model. If it does not find your file in your working directory, the Embedded Target for TI C6000 DSP searches your MATLAB path for the file and takes the first instance it finds of the specified file. Refer to *TMS320C6000 Assembly Language Tools User's Guide* and *TMS320C6000 Optimizing C Compiler User's Guide*, both from Texas Instruments, for more information.

- **Full\_memory\_map**—invokes the linker command file that uses the large memory model on the target. The large memory model does not restrict the size of the uninitialized code sections. Your global and static data can use unlimited storage space up to the limits of the board.
- **Internal\_memory\_map**—invokes a linker command file that directs the linker to use the small memory model of the target. Using the small memory model requires that the uninitialized sections of the code fit in the 32KB memory space. The total space for the global and static data in the program must be less than 32KB. This setting offers the optimum use of memory on your target by using only near calls and data exclusively.

### Notes About Using The Linker Command File Options

Your selections for the linker command file options affect how the Embedded Target for TI C6000 DSP handles *near* and *far* data and *near* and *far* function calls. By default, the Embedded Target for TI C6000 DSP, and the TI compiler, generate small memory models that use near function calls and near data exclusively. Near accessing data require only one operation; far data require more operations. As a consequence, programs and code that use far data runs more slowly. You should refer to your CCS documentation for details about near and far data and the near and far function calls.

When you select the `Internal_memory_map` option, the Embedded Target for TI C6000 DSP specifies that only near calls get used to access static and global data. `Internal_memory_map` represents the most efficient memory use. In CCS, the equivalent setting is to choose `Near Calls & Data` for the **Memory Models** option in the build configuration. These are the default settings in CCS.

If you select `Internal_memory_map`, but your data or program requires far calls, 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 do not fit in internal memory. To eliminate these errors, select `Full_memory_map`.

Use the `Full_memory_map` 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, or
- You have calls in which the called function is more than  $\pm 1$  M word away from the call site

In the above instances, the TI linker issues the error message shown earlier if you select the `Internal_memory_map` setting when your program meets the conditions specified.

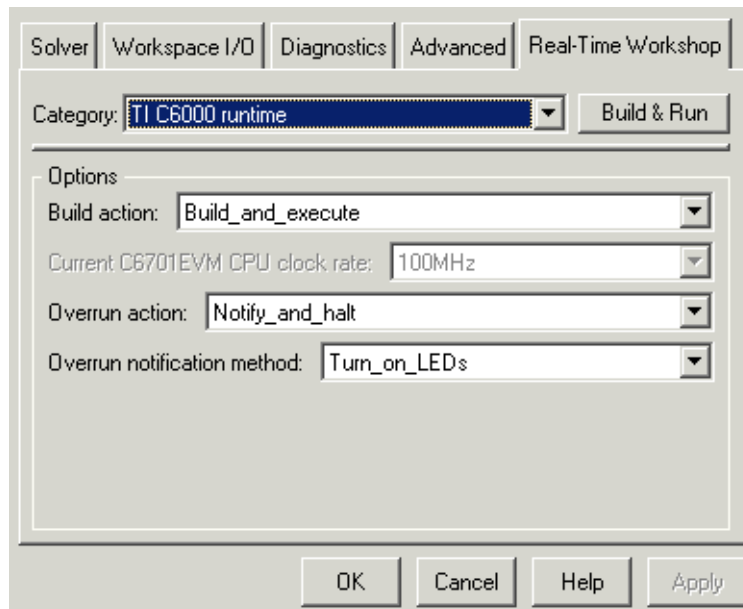
When you declare a function or data as far, its address is loaded into a register and the compiler does an indirect load of that register (the `-m1n` option in the **Memory Models** in the project build configuration). For more information on the `-m1n` option, refer to your CCS documentation.

You can avoid the error by selecting `Full_memory_map` for the **Linker Command File** option. This prevents the compiler from using near calls, offering you the ability to use all the available memory on your target. But note that your program may run more slowly than if you use the internal map option and your data and program fit into memory without needing far calls for access.

**User linker command file.** When you select the User defined option for **Linker command file**, enter the name of your command file in this box. If the file is not in your current directory, provide the full pathname for the file.

### TI C6000 Run-Time Options

Before you run your model as an executable on either the C6701 EVM or the C6711 DSK, you must configure the run-time options for the model on the board. When you select TI C6000 runtime from the **Category** list, you see the **Options** pane shown in this figure.



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.

This option 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. Selecting this setting enables the CCS board number option so you can select which installed board to target. This option offers a convenient way to build projects in CCS.
- `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 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.

---

**Current C6701 EVM CPU clock rate.** 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, either the C6701 EVM or the C6711 DSK. While the default clock rate is 100 MHz on the C6701 EVM, you can change the rate with the DIP switches on the board or from one of the software utilities provided by TI. C6711 DSK hardware uses a fixed clock rate of 150 MHz; you can not change the clock rate. While this option appears on the dialog, it is grayed out and does not apply to the C6711 DSK.

For the timer software to calculate the interrupts correctly, Embedded Target for TI C6000 DSP needs to know the actual clock rate of your C6701 EVM as you configured it. This option lets you choose the current clock rate—one of 25, 33.25, 100 or 133 MHz. Select the appropriate rate from the list. You are not changing the clock rate with this option. You are telling the software timer what rate to use to match the C6701 EVM clock rate.

The timer uses the CPU clock rate you select in **Current C6701 EVM CPU clock rate** 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.

**Overflow Action.** To enable the overflow indicator, choose one of three ways for the target processor to respond to an overflow condition in your model:

- None—ignore overflows encountered while running the model.
- `Notify_and_continue`—when the DSP encounters an overflow condition, it performs the operation you specify in **Overflow notification method** and continues running the executable. If you use an C6701 EVM LED block in your model, you cannot determine whether the C6701 EVM LED block enabled the external LED or if an overflow condition caused the LED to light.
- `Notify_and_halt`—respond to overflow conditions by stopping program execution and executing the **Overflow 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 overflow condition caused the LEDs to light.

**Overflow notification method.** In combination with the **Overflow action** option, you choose one of three ways for the Embedded Target for TI C6000 DSP to notify you when your application goes into an overflow state. From the **Overflow notification method** list, select one of the following notification functions:

- `Print_message`—when your application overflows and **Overflow action** is `Notify_and_continue` or `Notify_and_halt`, the software prints a message out to the standard output or the message log (for DSP/BIOS enabled projects).
- `Turn_on_LEDs`—when your application overflows and **Overflow action** is `Notify_and_continue` or `Notify_and_halt`, the software turns on the external LED on the C6701 EVM or the user LEDs on the C6711 DSK. Note that when you use an LED block in your model, you may not be able to determine whether the LED block enabled the external LED or user-defined LEDs, or an overflow condition caused the LEDs to light.
- `Print_message_and_turn_on_LEDs`—in an overflow situation where you have selected a notification action for **Overflow 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. When this is the case, 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, the timer provides the software interrupts that drive model processing.

### **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. In the tables and figures in the next section, you can find the build options that the Embedded Target for TI C6000 DSP presets in `custom_MW` and the default setting for each option. For more information about the build options, refer to your TI documentation.

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

### **Default 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 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 regime. For memory configuration, where Release



uses the default memory model that specifies near functions and data, `custom_MW` specifies far functions and data—the `-m/3` memory model—to accommodate large models. 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.

# Targeting Your C6701 EVM

Texas Instruments markets a complete set of tools for use with the C6701 EVM. 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 C6701 EVM 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 C6701 EVM), 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.

After you install the C6701 EVM development board and supporting TI products on your PC, start MATLAB. At the MATLAB command prompt, type `c6701evmlib`. This opens a Simulink block library, `c6701evmlib`, that includes a set of blocks for C6701 EVM I/O devices:

- C6701 EVM ADC (configure the analog to digital converter)
- C6701 EVM DAC (configure the digital to analog converter)
- C6701 EVM LED (control the user status light emitting diodes (LED) on the C6701 EVM)
- C6701 EVM Reset (reset the processor on the C6701 EVM)

These blocks are associated with your C6701 EVM board. As needed, add the devices to your model.

With your model open, select **Simulation -> Simulation Parameters**. From this dialog, click the **Real-Time Workshop** tab. You must specify the

appropriate versions of the system target file and template makefile. For the C6701 EVM, in the **Real-Time Workshop** pane, specify

- **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. Do this 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 inline 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 Writing S-Functions documentation.

During the same build operation, the template makefile and block parameter dialog entries are combined to form the target makefile for your TI evaluation module. This makefile invokes the TI compiler to build an executable file. When you select the `Build_and_execute` option, Real-Time Workshop automatically downloads the executable via the peripheral component interface (PCI) bus to the TI evaluation board. After downloading the executable file to the C6701 EVM, the build process runs the file on the processor.

### Starting and Stopping DSP Applications on the C6701 EVM

When you generate code, build the project, and download the code for your Simulink model to your C6701 EVM, 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 C6701 EVM, 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 running application encounters an error condition that stops the process.

If you included a Reset C6701 EVM 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 C6701 EVM, 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 C6701 EVM 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 C6701 EVM

When you install the C6701 EVM, set the dual inline pin (DIP) switches as shown below. 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)

DIP Switch	Name	Setting	Effect
SW2-10	USER2	On	user-defined option
SW2-11	USER1	On	user-defined option
SW2-12	USER0	On	user-defined option

## Confirming Your C6701 EVM 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 *TMS320C6201/6701 Evaluation Module User's Guide*.

To run the C6701 EVM 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 listed configuration, reconfigure your C6701 EVM. After you change your board configuration, rerun the verification utility to check your new settings.

## Testing Your C6701 EVM

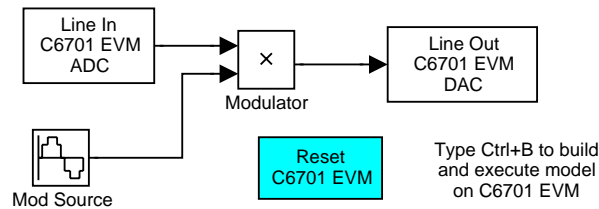
The Embedded Target for TI C6000 DSP includes a Simulink demonstration model called `c6701evmtest`. You can use this model to verify that you installed

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 C6701 EVM

As an initial test to determine that your Embedded Target for TI C6000 DSP software and C6701 EVM are installed and operating correctly, open and build the Simulink model `c6701evmtest`. See the model in the figure below.

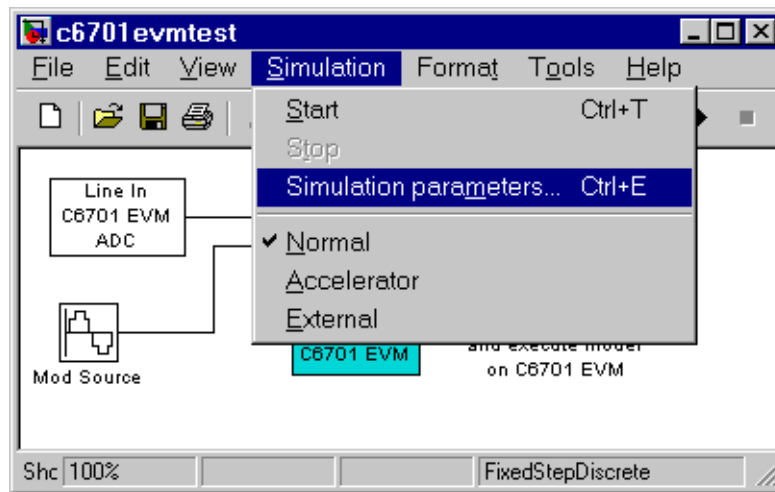


- 1 Enter `c6701evmtest` at the MATLAB command prompt.

The model opens in Simulink.

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

Figure 2-2, Using `c6701evmtest` to Test Your Embedded Target for TI C6000 DSP Installation, shows the model `c6701evmtest` with the **Simulation parameters** option selected.



**Figure 2-2: Using c6701evmtest to Test Your Embedded Target for TI C6000 DSP Installation**

- 3 On the **Simulation Parameters** dialog, click **Real-Time Workshop** 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 RTW build procedure for model: c6701evmtest.mdl
### Invoking Target Language Compiler on c6701evmtest.rtw
```

If c6701evmtest.mdl builds, compiles, and downloads to your C6701 EVM 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:EVM6x Revision:0
Using DSP memory map 1.
### Downloaded:c6701evmtest.out
```

```
### Successful completion of Real-Time Workshop build procedure  
for model:c6701evmtest
```

When you receive this message, your model is running on the C6701 EVM. 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 C6701 EVM and watch to see that the output amplitude modulation changes to match.

### **Error Message While Building C6701 EVMtest**

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 C6701 EVM” on page 2-50. 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 C6701 EVM immediately.

### **Verifying That C6701 EVMtest 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 your C6701 EVM. On the oscilloscope display, you should see the sine wave input from the signal generator, and the amplitude-modulated sine wave output from your C6701 EVM. 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 C6701 EVMtest.mdl on the C6701 EVM**

When you build and download the model `c6701evmtest.mdl` to your C6701 EVM, 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 C6701 EVM. The Embedded Target for TI C6000 DSP offers a function, `run`, that restarts your loaded program on your target.



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

- Using the **Debug -> Halt** function in CCS.
- Using `halt` from the MATLAB command prompt.
- Clicking the C6701 EVM Reset block in your model (if you added one) or in the C6701 EVM 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 DSP Blockset
- Use blocks from the fixed-point blocks library TI C62x 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 **Simulation Parameters** dialog. 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.

### Notes About Selecting Blocks for Your Models

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

they waste time waiting to send or receive data from your workspace, slowing your signal processing application without adding instrumentation value.

For this reason, we recommend 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. In the next table, we present the blocks you should not use in your target models.

<b>Block Name/Category</b>	<b>Library</b>	<b>Description</b>
Scope	Simulink, DSP 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 <b>'Scope' parameters</b> dialog.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	DSP 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	DSP Blockset	Send data to your MATLAB workspace.
Signal To Workspace	DSP Blockset	Send a signal to your MATLAB workspace.

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

## C6701 EVM Tutorial 2-1 – 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 DSP 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 `c6701evmafxr.mdl` as shown in the next figure. Open this model by typing `c6701evmafxr` at the MATLAB prompt.

To run this model you need a microphone connected to the **Mic In** connector on your C6701 EVM, and speakers and an oscilloscope connected to the **Line Out** connector on your C6701 EVM. 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 C6701 EVM, you complete the following tasks:

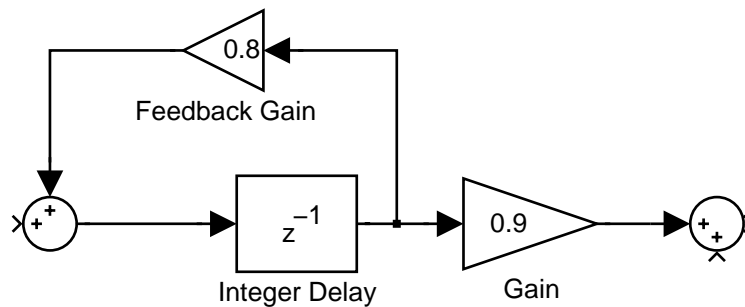
- 1** Use Simulink blocks and blocks from other blocksets to create your model application.
- 2** Add the Embedded Target for TI C6000 DSP blocks that let your signal sources and output devices communicate with your C6701 EVM—the C6701 EVM ADC and C6701 EVM DAC blocks that you find in the Embedded Target for TI C6000 DSP `c6000lib` blockset.
- 3** Configure the simulation parameters for your model, including
  - Simulation parameters such as simulation start and stop time and solver options
  - Real-Time Workshop options such as target configuration and target compiler selection
- 4** Build your model to the selected target.
- 5** 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 C6701 EVM installed on your PC. Be sure to configure and test your board as directed in “Configuring Your C6701 EVM” on page 2-50 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 to create the following model.



Look for the Integer Delay block in the Signal Operations library of the DSP Blockset. You do not need to add the input and output signal lines at this time. When you add the C6701 EVM 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.

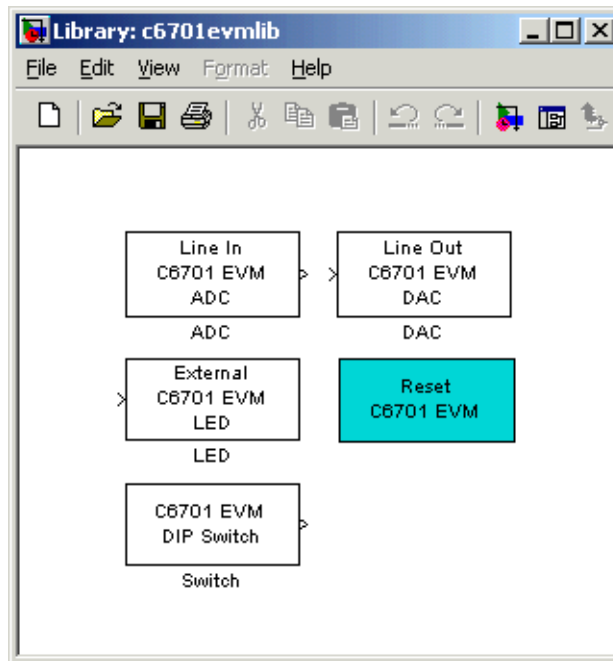
### Adding C6701 EVM Blocks to Your Model

So that you can send signals to your C6701 EVM 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 C6701 EVM:

- Input block (C6701 EVM ADC)
- Output block (C6701 EVM DAC)

- Light emitting diode block (C6701 EVM LED)
- Software reset block (Reset C6701 EVM)
- DIP switch block (C6701 EVM DIP Switch)

Typing `C6701EVMlib` at the MATLAB prompt brings up this window showing the library blocks.



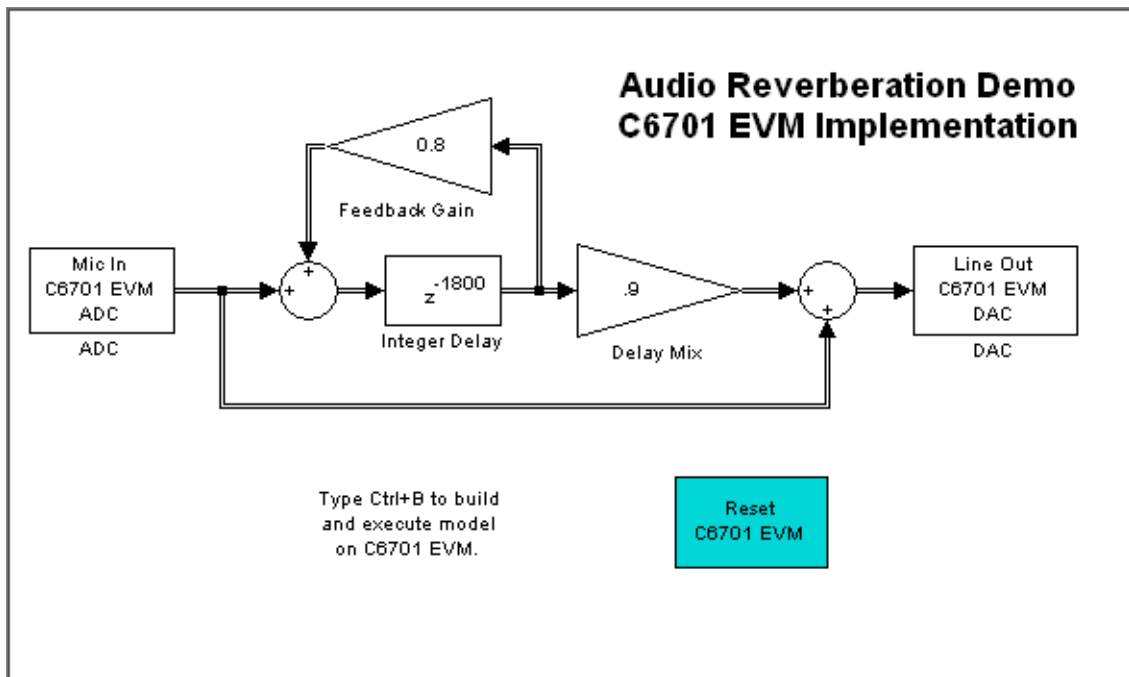
### Embedded Target for TI C6000 DSP Target for C6701 EVM Block Library C6701evmlib

These blocks are included in the Embedded Target for TI C6000 DSP `c6000lib` blockset in the Simulink Library browser.

The C6701 EVM ADC and C6701 EVM DAC blocks let you configure the codec on your C6701 EVM 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 C6701 EVM ADC and C6701 EVM DAC blocks add driver software that controls the behavior of the codec for your model.

To add C6701 EVM 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 C6701 EVM Board Support to see the blocks available for your C6701 EVM.
- 3 Drag and drop C6701 EVM ADC and C6701 EVM DAC blocks to your model as shown in the figure.



- 4 Connect new signal lines as shown in the figure.

### 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 C6701 EVM 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 **C6701 EVM ADC source**, select Mic In.
- 5 Click **OK** to close the **C6701 EVM ADC** dialog.
- 6 Now set the options for the C6701 EVM 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.

You have completed the model. Now you configure the Real-Time Workshop options to build and download your new model to your C6701 EVM.

## Configuring Simulation Parameters for Your Model

The following sections describe how to build and run real-time digital signal processing models on your C6701 EVM. Running a model on the target starts



with configuring and building your model from the **Simulation Parameters** dialog in Simulink.

### Setting Simulink Simulation Parameters

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

- 1 Open the **Simulation Parameters** dialog and set the appropriate options on the **Solver** pane 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)
  - Under **Solver options**, select the fixed-step and discrete settings from the lists
  - Set the **Fixed step size** to Auto and the **Mode** to Single Tasking

Ignore the **Workspace I/O**, **Diagnostics**, and **Advanced** panes in the **Simulation Parameters** dialog. 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 on the **Real-Time Workshop** pane of the **Simulation Parameters** dialog. Follow these steps to set the Real-Time Workshop options to target your C6701 EVM:

- 1 Click the **Real-Time Workshop** tab.
- 2 For **Category**, select Target configuration.
- 3 Under **Configuration**, click **Browse** to select the system target file for C6000 targets.
- 4 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.

- 5 To choose your C6000 target, change **Category** to TI C6000 target.

- 6 From the **Code generation target type** list, choose C6701\_EVM.
- 7 For **Category**, select TI C6000 code generation to specify code generation options that apply to the C6701 EVM target.
- 8 Select the **Inline DSP Blockset functions** option.
- 9 For **Category**, select TI C6000 compiler to set the Real-Time Workshop compile options.
- 10 Set the following options in the dialog:
  - **Byte order** should be Little endian
  - Set **Compiler verbosity** to Quiet
- 11 Change the **Category** to TI C6000 linker.
- 12 Set the linker operation options.
  - Select the **Retain .obj files** check box
  - For the **Linker command file**, select Full memory map
- 13 Change the **Category** to TI C6000 runtime.
- 14 Set the following Real-Time Workshop run-time options:
  - **Build action:** Build\_and\_execute
  - **Current C6701 EVM CPU clock rate:** 100 MHz
  - **Overrun action:** Notify\_and\_halt
  - **Overrun notification method:** Turn\_on\_LEDs

You have configured the Real-Time Workshop options that let you target your C6701 EVM. You may have noticed that you did not configure three Real-Time Workshop options on the **Category** list: **TLC debugging**, **General code generation options**, and **General code generation options (cont.)**.

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 C6701 EVM

After you set the simulation 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 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 C6701 DSP on your C6701 EVM, and then downloads the executable file to the target and runs the file.

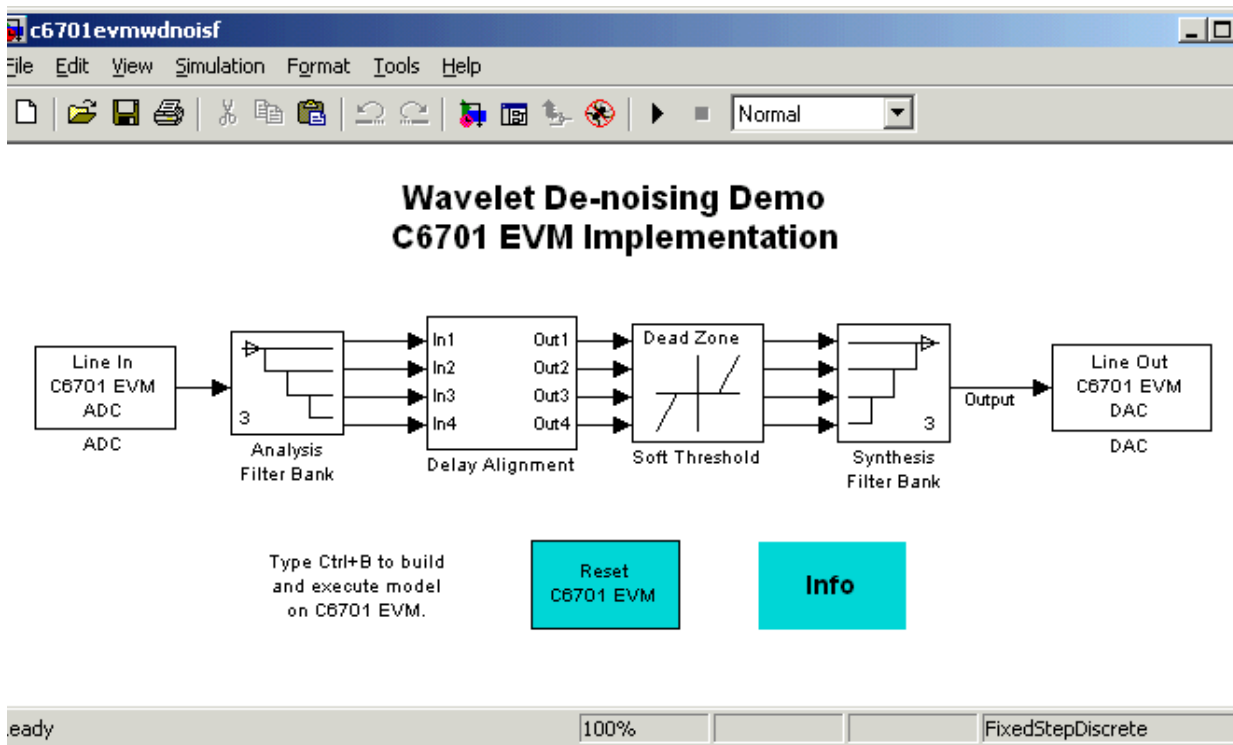
- 2 To stop model execution, click the **Reset C6701 EVM** block or use the **Halt** option in CCS. You could type `halt` from the MATLAB command prompt as well.

## Testing Your Audio Reverb Model

With your model running on your C6701 EVM, 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 `c6701evmafxr`, try running the demonstration model to compare the results.

## C6701 EVM Tutorial 2-2—A Multistage 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 `evm67xdnoisf`. As with any model file, you can run this denoising demonstration by typing `c6701evmdnoisf` at the MATLAB prompt. The model also appears in the MATLAB demos collection. 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 complicated 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 C6701 EVM as an executable COFF file.

### Working and Build Directories

It is convenient to work with a local copy of the `c6701evmwdnoisf` model, stored in its own directory, which you name (something like `c6701dnoisfex`). 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 `c6701evmwdnoisf` model.

```
c6701evmwdnoisf
```

The model appears in the Simulink window.

- 4 From the **File** menu, choose **Save As**. Save a copy of the `c6701evmwdnoisf` model as `d:/c6701dnoisfex/dnoisfrtw.mdl`.

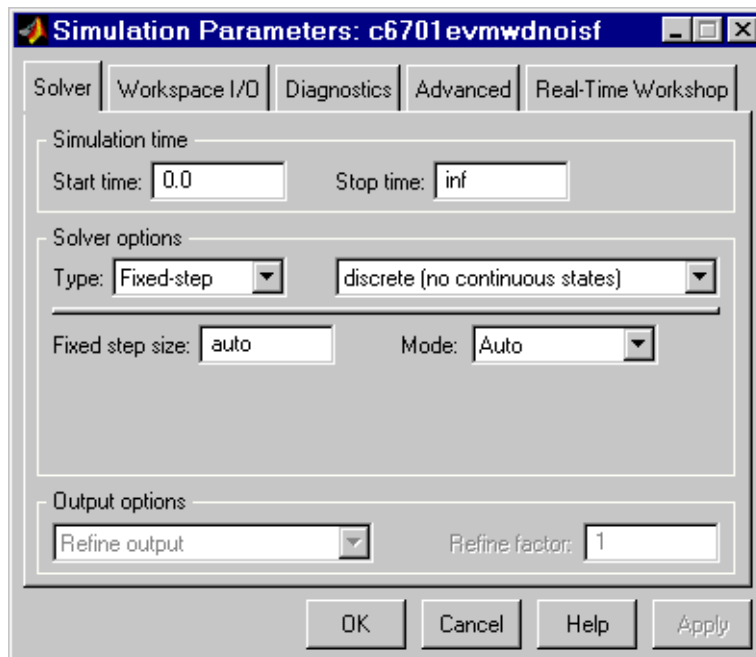
During code generation, Real-Time Workshop creates a build directory within your working directory. The build directory name is `model_target_rtw`, derived from the name of your source model and your chosen target. In the build directory, Real-Time Workshop stores generated source code and other files 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 simulation parameters. In particular, Real-Time Workshop uses a fixed-step solver. To set the parameters, use the **Simulation Parameters** dialog as follows:

- 1** From the **Simulation** menu, choose **Simulation parameters**. The **Simulation Parameters** dialog opens.
- 2** Click **Solver** and enter the following parameter values on the **Solver** pane.  
**Start Time:** 0.0  
**Stop Time:** inf  
**Solver options:** set **Type** to Fixed-step. Select the discrete solver algorithm.  
**Fixed step size:** auto  
**Mode:** Auto
- 3** Click **Apply**. Then click **OK** to close the dialog.
- 4** Save the model. Simulation parameters persist with the model, for you to use in future sessions.

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



### Solver Pane of Simulation Parameters Dialog

#### Selecting the Target Configuration

To specify the desired target configuration, you choose a system target file, a template makefile, and a 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.

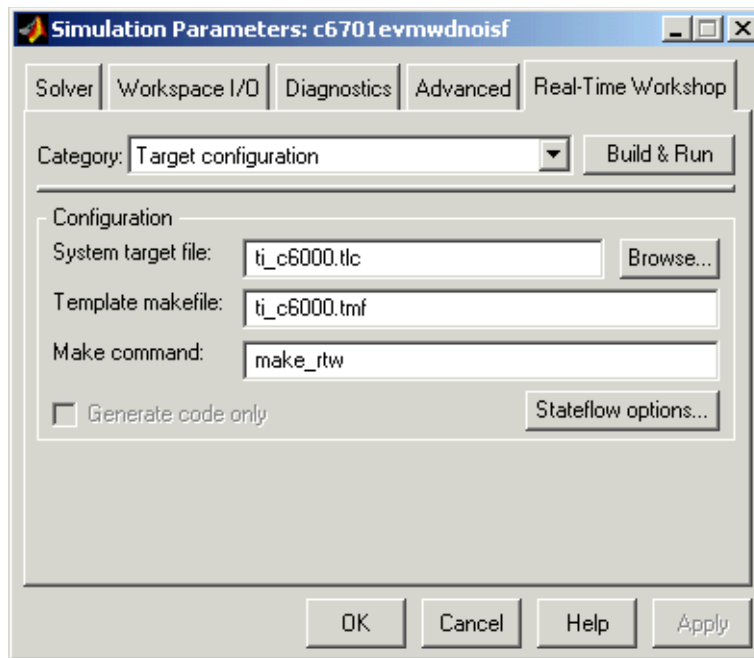
---

**Note** The **Real-Time Workshop** tab has several panes, which you select using the **Category** list. During this tutorial you change or review options on every pane in the list.

---

To target your C6701 EVM:

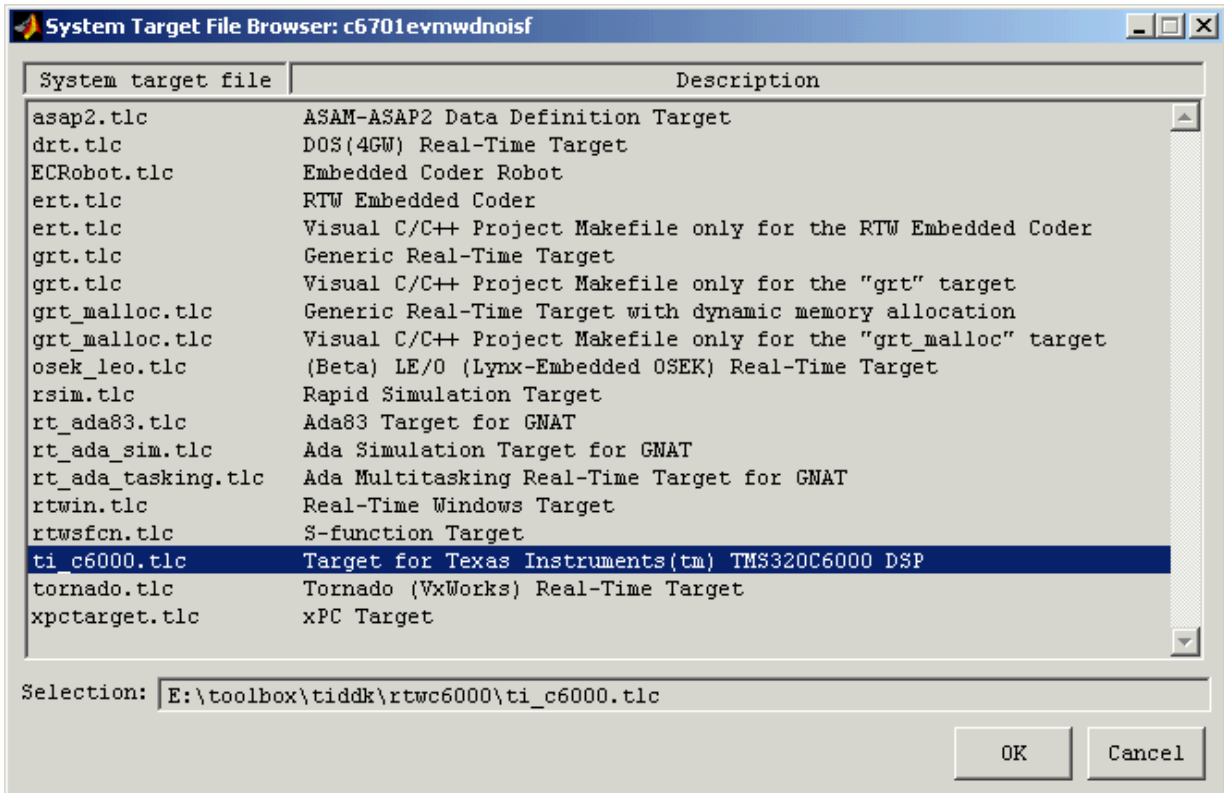
- 1 From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog opens.
- 2 Click **Real-Time Workshop** on the **Simulation Parameters** dialog. The **Real-Time Workshop** pane activates.
- 3 The **Real-Time Workshop** pane has several pages, which you select via the **Category** option. Select **Target configuration** from the **Category** list.



**Figure 2-3: Real-Time Workshop Pane (Target Configuration Category)**

- 4 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.



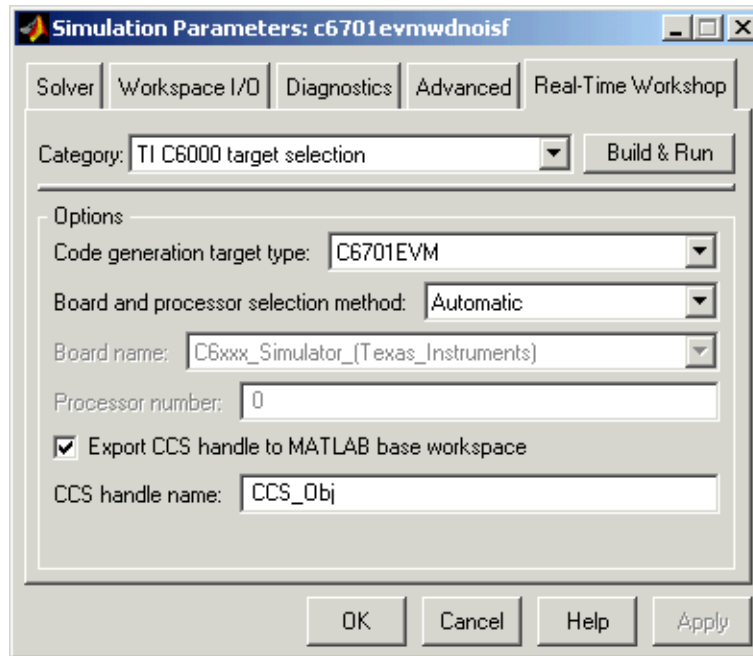


**Figure 2-4: The System Target File Browser**

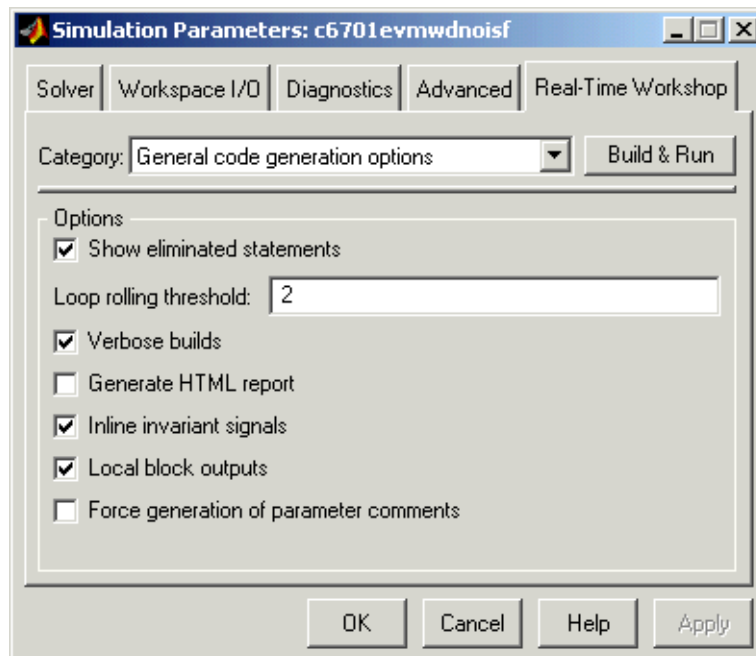
- From the list of available configurations, select `ti_c6000.tlc` (as in Figure 2-4) and click **OK**.

The **Real-Time Workshop** pane now displays the correct **System target file** (`ti_c6000.tlc`), **Template makefile** (`ti_c6000.tmf`), and **Make command** (`make_rtw`), as shown in Figure 2-3.

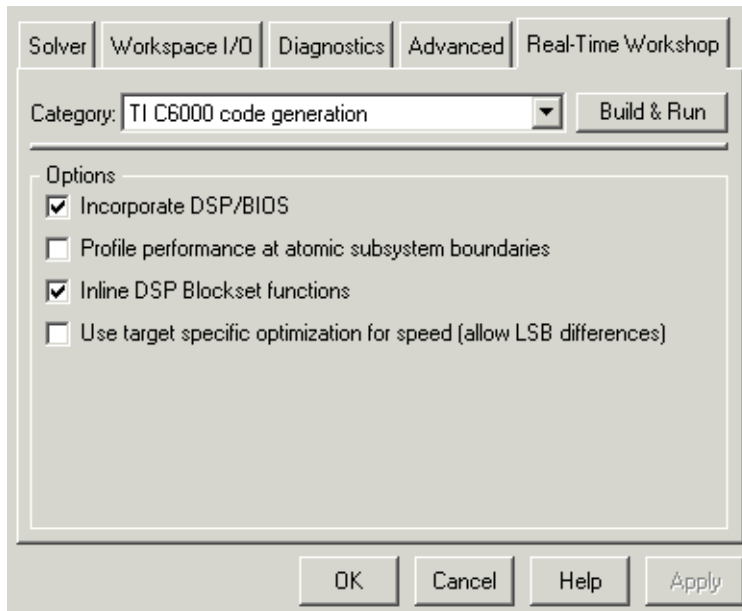
- To choose your specific C6000 hardware target, select **TI C6000** target selection from the **Category** list.



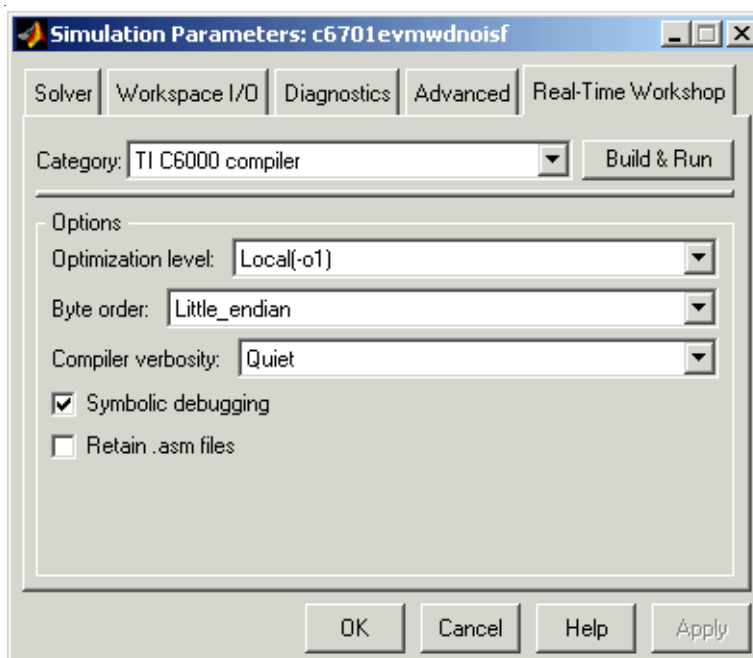
- 7 From the **Code generation target type** list, choose C6701EVM.
- 8 To export the handle that CCS creates when you generate code from your model, select **Export CCS handle to MATLAB workspace** and enter a name for the handle in **CCS handle name**.
- 9 Select General code generation options from **Category**. 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 below.



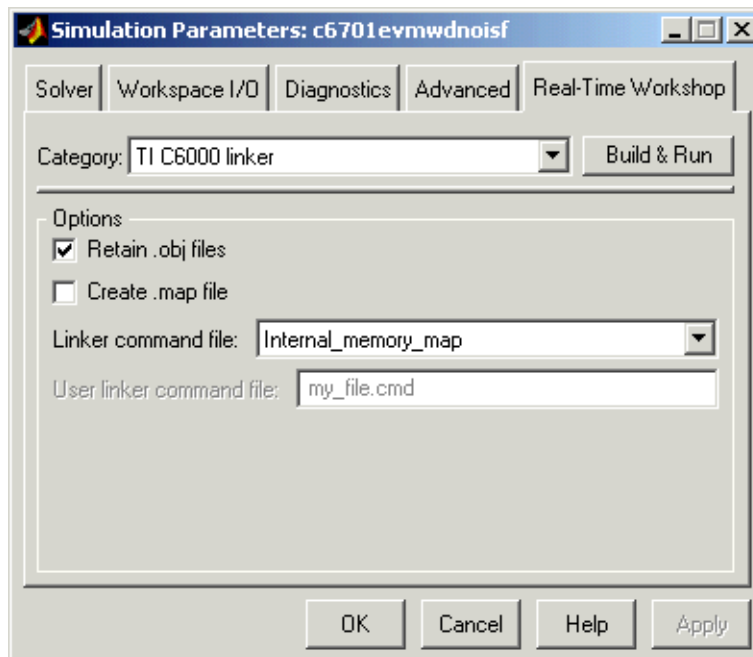
- 10** Select **TLC debugging** from the **Category** list to access the TLC debugging options. Clear all the check boxes on this pane.
- 11** To set the code generation options, change the **Category** again. Select **TI C6000 code generation** from the list to specify code generation options that apply to the C6701 EVM target.
- 12** Select the **Inline DSP Blockset functions** option, as shown.



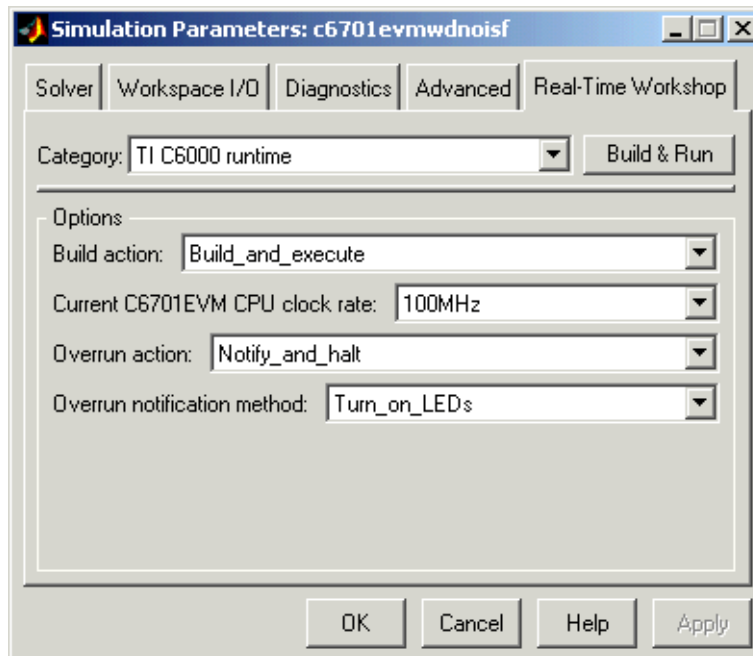
- 13 Select TI C6000 compiler from the **Category** list. The options displayed on the new pane are specific to the C6000 target. Check to make sure that the options are set as shown.



- 14** Select TI C6000 linker from **Category** list to access the options for the linker program. Set the linker options as shown.



- 15** Select TI C6000 runtime from the **Category** list to enable the C6000 run-time options pane. Set the run-time options as shown.



- 16** Click **OK** to close the **Simulation Parameters** dialog. 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 **Simulation Parameters** dialog for your model.
- 2** Click **Build** in the **Simulation Parameters** dialog 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.



- `dnoisf_rtw.h`—an include header file containing information about the state variables
- `dnoisf_rtw_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 (`dnoisf_rtw.mk`).

# Targeting Your C6711 DSK

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 “Real-Time Workshop Options for C6000 Hardware” on page 2-26
- Configuring your Texas Instruments TMS320C6711 DSP Starter Kit, in “Configuring Your C6711 DSK” on page 2-80
- Testing your hardware and software installation to be sure everything works, in “Confirming Your C6711 DSK Installation” on page 2-80 and “Testing Your C6711 DSK” on page 2-81

## 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\dsk6x11\confst`

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

- 3 Start the confidence test by typing `dsk6xtst` at the DOS prompt.

By default, the test utility creates a log file named `dsk6xtst.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.

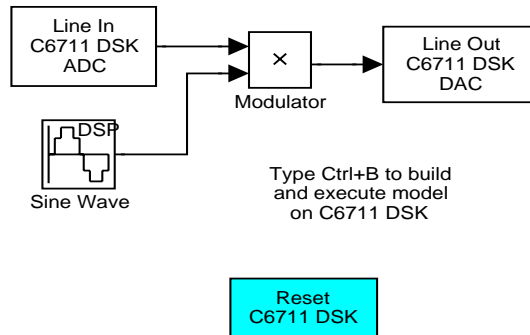
## Testing Your C6711 DSK

The Embedded Target for TI C6000 DSP includes a Simulink demonstration model called `c6711dsktest`. 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 figure below.

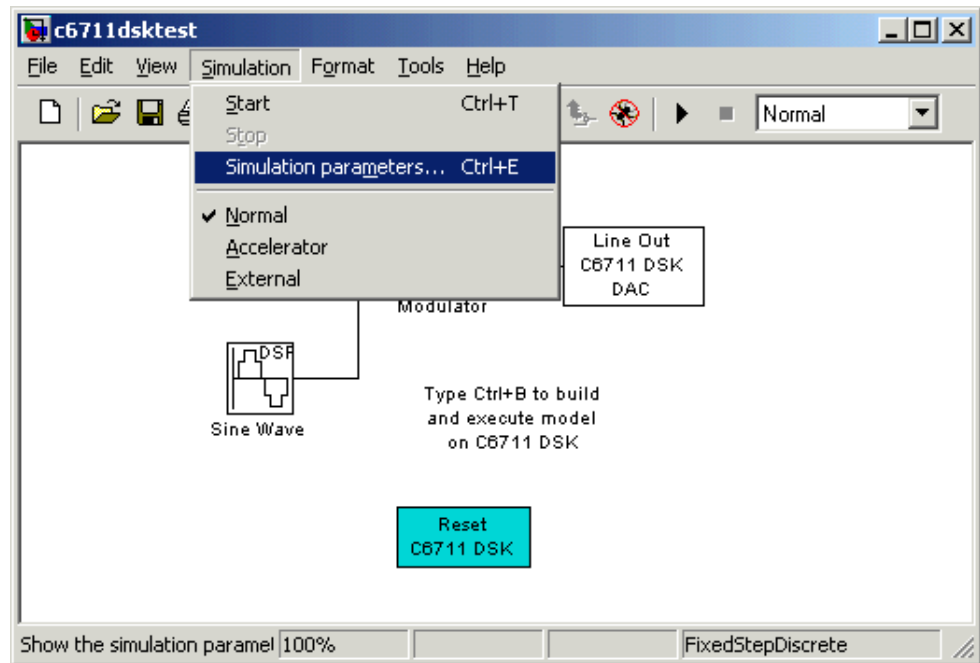


- 1 Enter c6711dsktest at the MATLAB command prompt.

Test model c6711dsktest opens in Simulink.

- 2 Select **Simulation -> Simulation parameters** from the menu bar.

Figure 2-5, Using c6711dsktest to Test Your Embedded Target for TI C6000 DSP Installation, shows the model c6711dsktest with **Simulation Parameters** selected.



**Figure 2-5: Using c6711dsktest to Test Your Embedded Target for TI C6000 DSP Installation**

- 3 Click **Real-Time Workshop** in the **Simulation Parameters** dialog to view the **Real-Time Workshop** dialog.
- 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.

MATLAB returns a lengthy series of messages in the Command Window, starting with

```
### Starting RTW 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 Message 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 2-80. 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.mdl 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

## C6711 DSK Tutorial 2-3—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 the DSP 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 `c6711dskafxr.mdl` as shown in the next figure. Open this model by typing `c6711dskafxr` 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 to your C6711 DSK, you complete the following tasks:

- 1** Use Simulink 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 C6711 DSK Board Support library.
- 3** Configure the simulation parameters for your model, including
  - Simulation parameters such as simulation start and stop time and solver options.
  - Real-Time Workshop options such as target configuration and target compiler selection.
- 4** Build your model to the selected target.
- 5** Test your model running on the target by changing the input to the target and observing the output from the target.

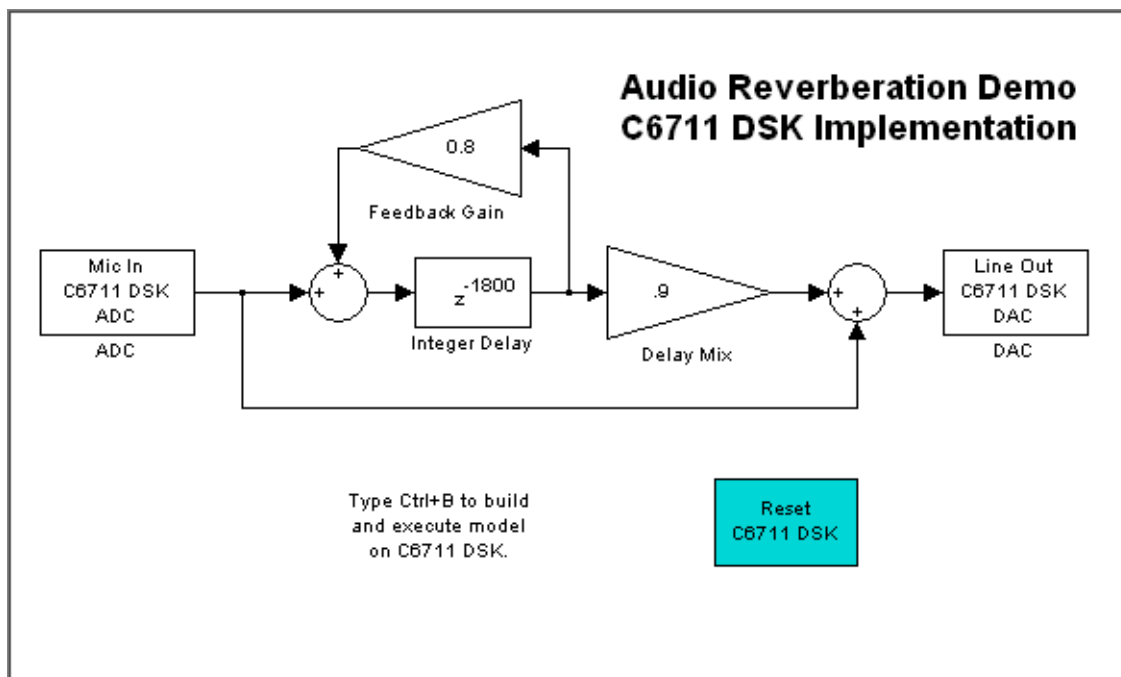


Your target for this tutorial is your C6711 DSK installed on your PC. Be sure to configure and test your board as directed in “Configuring Your C6711 DSK” on page 2-80.

### Building the Audio Reverberation Model

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

- 1 Open Simulink.
- 2 Create a new model by selecting **File -> New -> Model** from the **Simulink** menu bar.
- 3 Use Simulink blocks to create the following model.



Look for the Integer Delay block in the Signal Operations library of DSP 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.

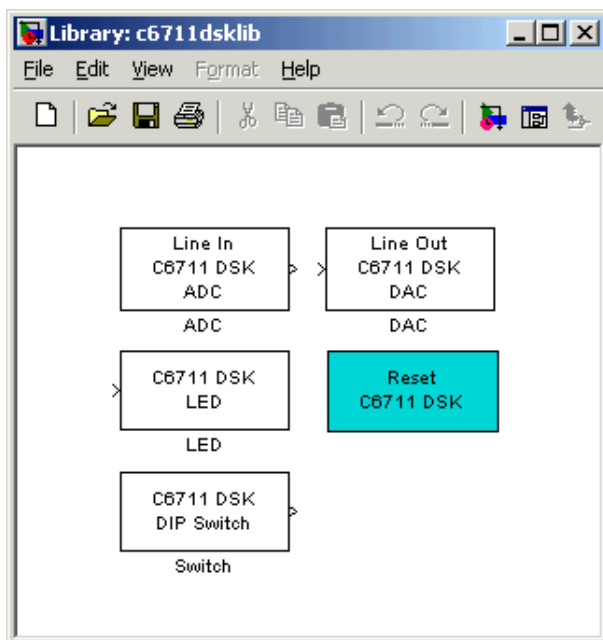
- 4 Save your model with a suitable name before continuing.

### **Adding C6711 DSK Blocks to Your Model**

So that you can send signals to your C6711 DSK and get signals back, TI C6000 includes a block library that contains five blocks designed to work with the codec and LEDs on your C6711 DSK:

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

Type `c6711dsklib` at the MATLAB prompt to bring up this window showing the library contents.



**Figure 2-6: Embedded Target for TI C6000 DSP Block Library c6711dsklib**

These blocks are included in the Embedded Target for TI C6000 DSP c6000lib blockset in the Simulink Library browser.

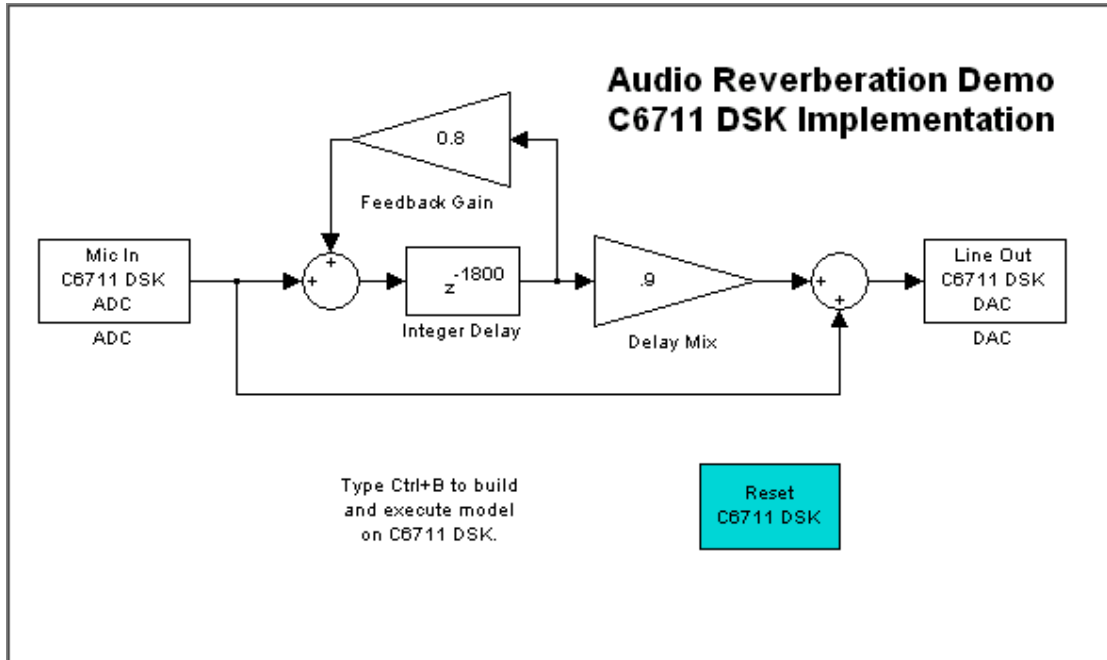
The C6711 DSK ADC and C6711 DSK DAC blocks let you configure the codec on the C6711 DSK to accept input signals from the input connectors on the board, and send the model output to the output connector on the board. Essentially, the C6711 DSK ADC and C6711 DSK DAC blocks add driver software that controls the behavior of the codec for your model.

To add an input to the your model without using a C6711 DSK ADC block, add a DSP source block, such as a signal generator, that creates the discrete time signal you need and use that signal as the input to 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 display the C6000lib blockset.

- 2 Double-click C6711 DSK Board Support to view the C6711 DSK blocks.
- 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.

### 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:
  - Select the **+20 dB mic gain boost** check box

- 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 running from the input to the output in your model so you can display both the input signal and the modified output signal on the oscilloscope for comparison.

- 4 For **ADC source**, select Mic In.
- 5 Click **OK** to close the **Block Parameters: ADC** dialog.
- 6 Now set the options for the C6711 DSK DAC block.
  - Set **Scaling** to Normalize
  - For **DAC attenuation**, enter 0.0
  - Set **Overflow mode** to Saturate.
- 7 Click **OK** to close the dialog.

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

## Configuring Simulation Parameters for Your Model

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

### Setting Simulink Simulation Parameters

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

- 1 Open the **Simulation Parameters** dialog and set the appropriate options on the **Solver** pane 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)
- Under **Solver options**, select the fixed-step and discrete settings from the lists
- Set the **Fixed step size** to auto and select Single Tasking for the **Mode**

Ignore the **Workspace I/O**, **Diagnostics**, and **Advanced** panes in the **Simulation Parameters** dialog. 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 the model executable, you set the options on the **Real-Time Workshop** pane of the **Simulation Parameters** dialog. Follow these steps to set the **Real-Time Workshop** options to target your C6711 DSK for running your model:

- 1 Click the **Real-Time Workshop** tab.
- 2 For **Category**, select Target configuration.
- 3 Under **Configuration**, click **Browse** to select the system target file for the C6000 hardware targets.
- 4 On the **System Target File Browser**, select the system target file `ti_c6000.tlc` and click **OK** to close the browser.

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

- 5 To choose your C6000 target, change **Category** to TI C6000 target selection.
- 6 From the **Code generation target type** list, choose C6711\_DSK.
- 7 For **Category**, select TI C6000 compiler to set the Real-Time Workshop compile options.
- 8 Set the following options in the dialog:
  - **Byte order** should be Little endian
  - Set **Compiler verbosity** to Quiet

- 9 Change the **Category** to TI C6000 linker.
- 10 Set the linker operation options.
  - Select the **Retain .obj files** check box
  - For the **Linker command file**, select Full memory map from the list
- 11 Change the **Category** to TI C6000 runtime.
- 12 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 cannot set the CPU clock rate on your C6711 DSK so that option is not available.

You have configured the Real-Time Workshop options that let you target your C6711 DSK. Notice that you did not configure three Real-Time Workshop options on the **Category** list: **TLC debugging**, **General code generation options**, and **General code generation options (cont.)**

For your model in this tutorial, the default values for 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 launch target language compiler (TLC) debugging when you generate code.

## Building and Executing Your Model on Your C6711 DSK

After you set the simulation parameters and configured 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 Click **Build** to generate and build an executable targeted to the C6711 DSK.

When you click **Build** with the Build\_and\_execute option selected, the automatic build process creates the executable file that can be run by the C6711 DSP on the C6711 DSK, and then downloads and runs the executable file on your target.
- 2 Stop model execution 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 DSK Block library

### 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.

### 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 DSK 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—lets you set 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 -> Simulation parameters** from the menu bar to open the **Simulation Parameters** dialog. From this dialog, click the **Real-Time Workshop** tab. 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, 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 inline 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 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 C6701 EVM, 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
- 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.

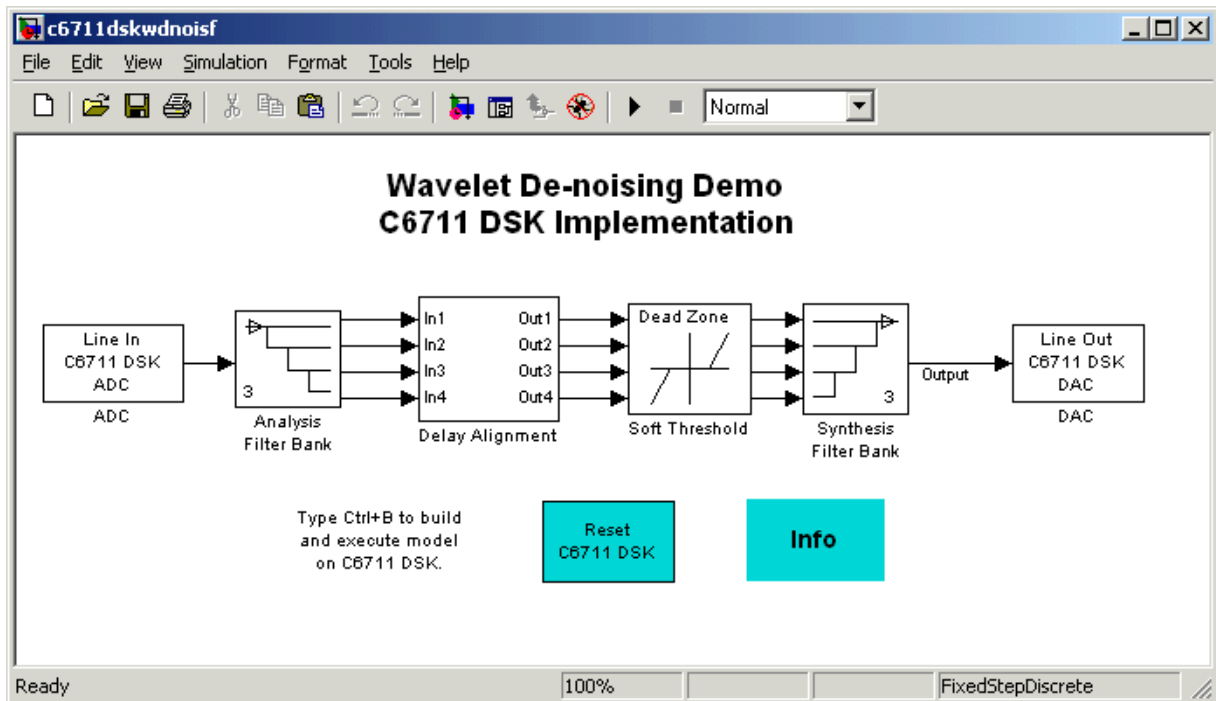
---

**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 Reset C6711 DSK block to stop processes that are running on your C6711 DSK, or to return your board to a known state for any reason.

---

## C6711 DSK Tutorial 2-4—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. Here is a picture of the model as it appears in the demonstration library.



Unlike the audio reverberation tutorial model used in tutorial 2-4, this model is difficult to build from blocks in Simulink. It uses complicated 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, that you name (something like `c6711dskwdnoisfex`). This discussion assumes that the `c6711dskwdnoisf` directory resides on drive `d:`. Use a different drive letter if necessary for your machine. Set up your working directory as follows:

- 1 Create your new model directory from the MATLAB command line by typing  
`mkdir d:\c6711dskwdnoisfex` (on PC)
- 2 Make `c6711dskwdnoisfex` your working directory in MATLAB.  
`cd d:/c6711dskwdnoisfex`
- 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:/c6711dskwdnoisfex/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 Simulink Simulation Parameters

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

- 1 From the **Simulation** menu, choose **Simulation Parameters**. The **Simulation Parameters** dialog opens.

**2** Click **Solver** and enter the following parameter values on the **Solver** pane.

**Start Time:** 0.0

**Stop Time:** inf

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

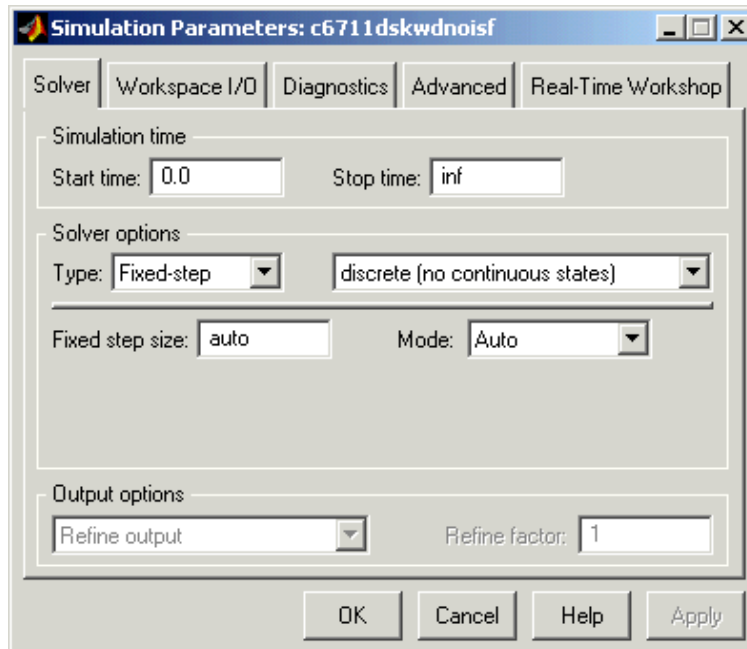
**Fixed step size:** auto

**Mode:** Auto

**3** Click **Apply**. Then click **OK** to close the dialog.

**4** Save the model. Simulation parameters persist with your model, for you to use in future sessions.

The next figure shows the **Solver** pane with the correct parameter settings.



### Selecting the Target Configuration

To specify the desired target configuration, you choose a system target file, which specifies the appropriate template makefile, and 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.

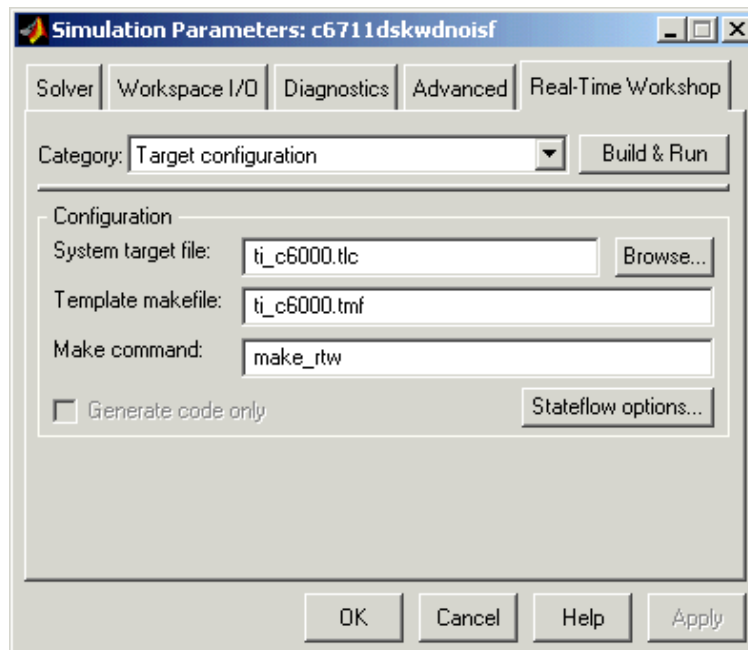
---

**Note** The **Real-Time Workshop** tab has several panes, which you select using the **Category** list. During this tutorial you change or review options on every pane in the list.

---

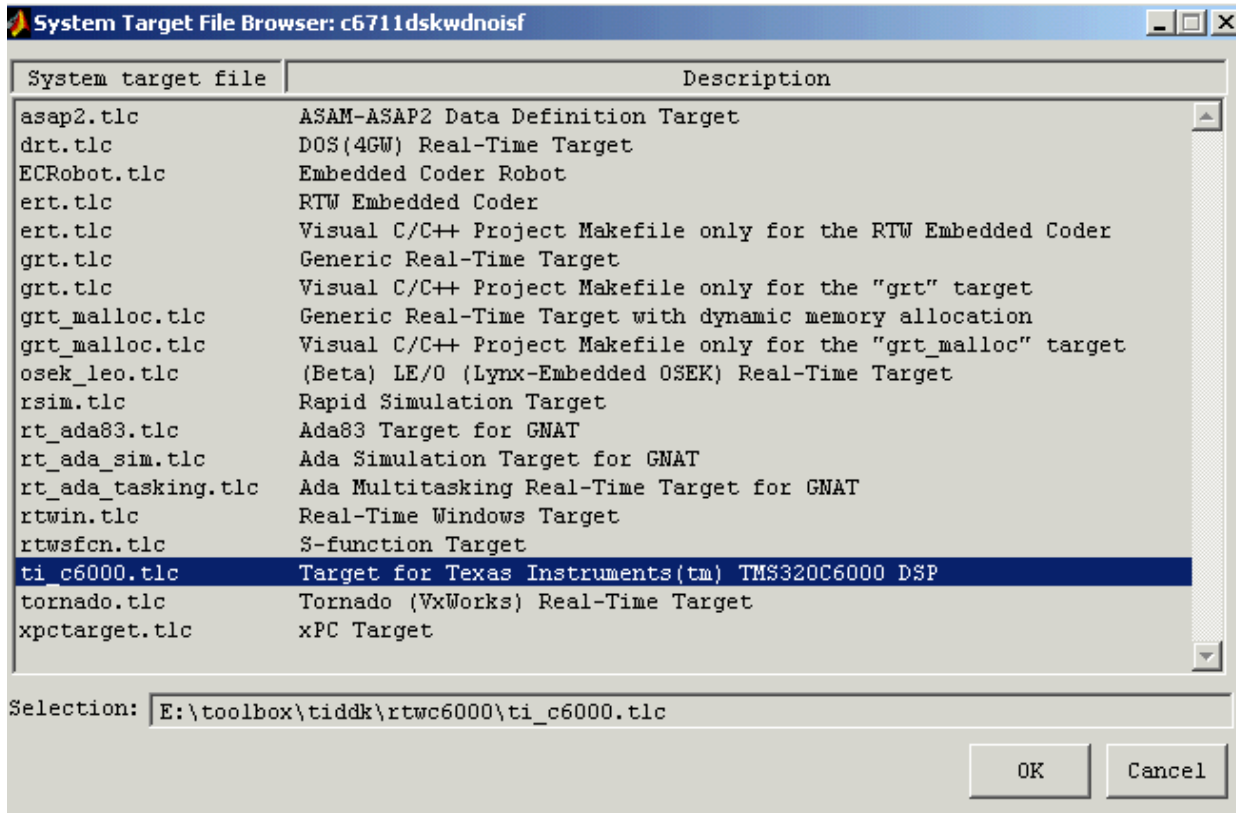
To select the C6000 target:

- 1** From the **Simulation** menu, choose **Simulation parameters**. The **Simulation Parameters** dialog opens.
- 2** Click **Real-Time Workshop** on the **Simulation Parameters** dialog. The **Real-Time Workshop** pane activates.
- 3** The **Real-Time Workshop** pane has several pages, which you select from **Category**. Select Target configuration from the **Category** list.



**Figure 2-7: Real-Time Workshop Pane (Target Configuration Category)**

- 4 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**.

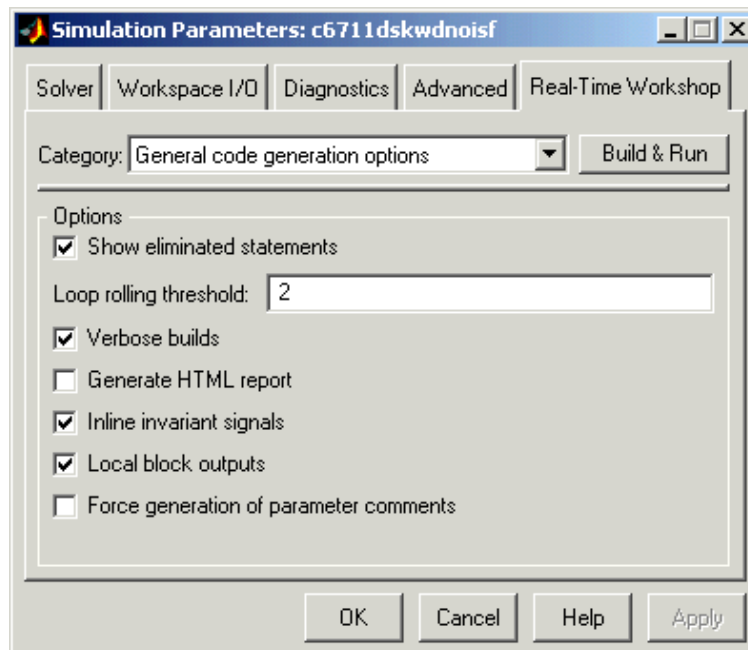


- 5 From the list of available configurations, select `ti_c6000.tlc` shown above and click **OK**.

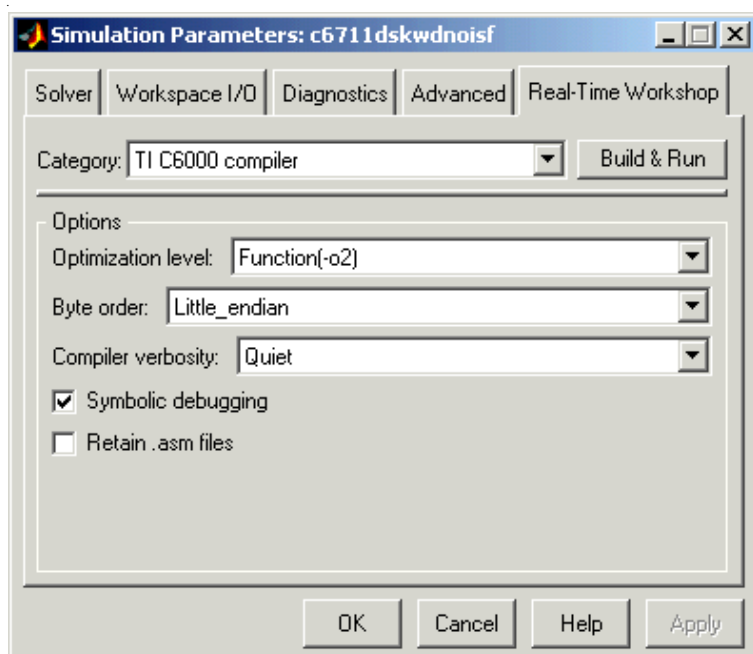
The **Real-time Workshop** pane now displays the correct **System target file** (`ti_c6000.tlc`), **Template makefile** (`ti_c6000.tmf`), and **Make command** (`make_rtw`), as shown in Figure 2-7.

- 6 Select General code generation options from the **Category** list. The options displayed here are common to all target configurations. Check to make sure that all options are set to their defaults, as below.

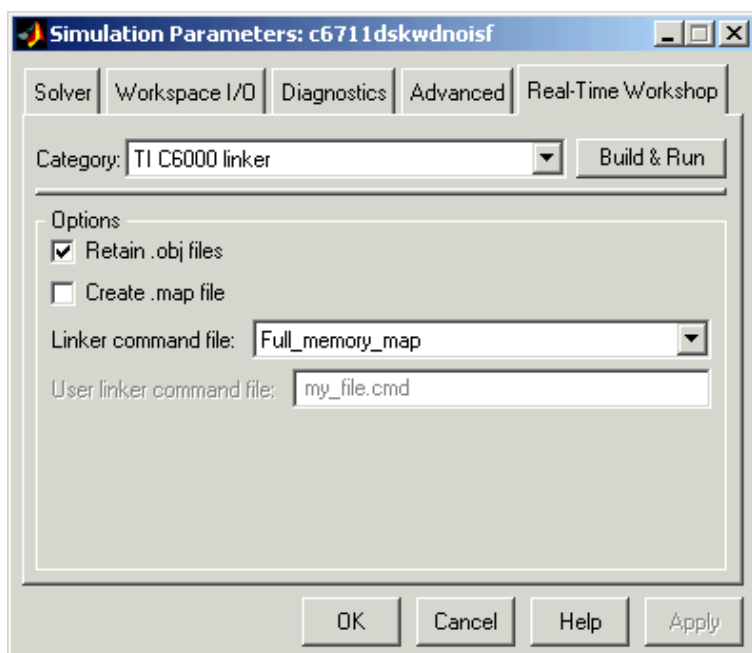




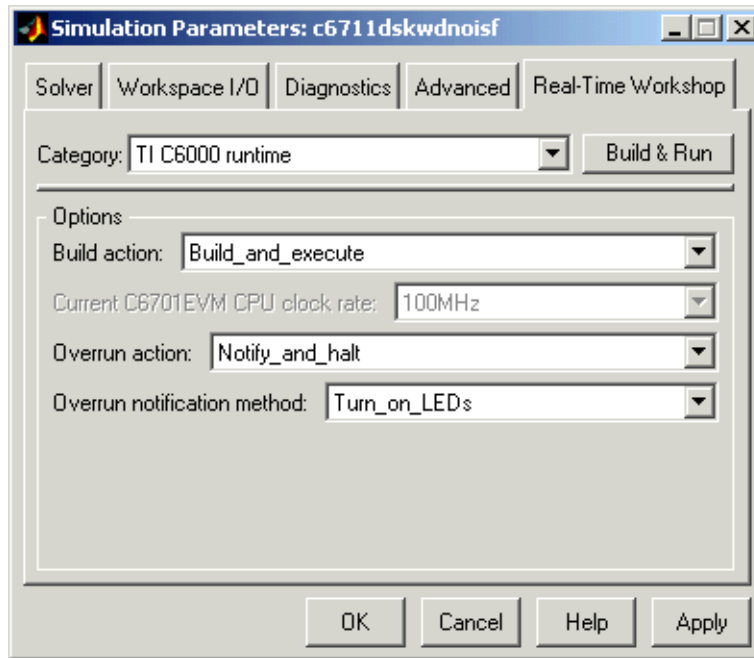
- 7** Select TLC debugging from **Category**. Clear all the check boxes in this category.
- 8** Select TI C6000 compiler from **Category**. The options displayed here are specific to the C6711 DSK target. Check to make sure that the options are set as shown below.



- 9 Select TI C6000 linker from **Category**. Set the linker options as shown.



**10** Select TI C6000 runtime from **Category**. Set the run-time options as shown.



- 11 Click **OK** to close the **Simulation Parameters** dialog. Save the model to retain your new build settings.

### Building and Running Your Model

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

To build and run the program:

- 1 Access the simulation parameters for your model.
- 2 Click **Build** in the **Simulation Parameters** dialog 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 `dir` at the MATLAB command prompt.

```
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_c6000_rtw
```

### Contents of the Build Directory

The build process creates a build directory and names it `modelName_target_rtw`, concatenating the name of your source model and your chosen target. In this example, your build directory is named `dnoisfrtw_c6000_rtw`.

`dnoisfrtw_c6000_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 or generated in the build process, such as the object (`.obj`) files, the command file (`.cmd`), and the generated makefile (`dnoisfrtw.mk`).

## Creating Code Composer Studio Projects Without Building

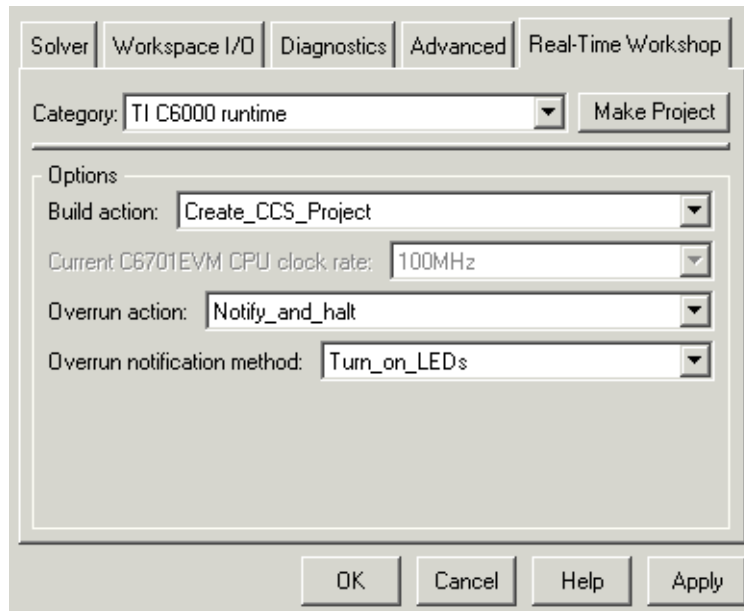
Rather than targeting your C6000 board when you build your signal processing application, you can create Texas Instruments Code Composer Studio (CCS) projects. Creating projects for CCS lets you use the tools provided by the CCS software suite to debug your real-time process.

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**.

### To Create Projects in CCS Without Loading Files to Your Target

From the **Real-Time Workshop** pane in the **Simulation Parameters** dialog, select TI C6000 runtime in **Category**. Select `Create_CCS_Project` for the **Build action**, as shown in the 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 CCS compiler and CCS linker categories on the **Real-Time Workshop** pane. Click **Make Project** 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 with DSP/BIOS™ Options

---

Introducing DSP/BIOS™ (p. 3-2)

Introduces DSP/BIOS from Texas Instruments.

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

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

“Using DSP/BIOS with Your Target Application”  
on page 3-21

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

Profiling Generated Code (p. 3-10)

Demonstrates how to set up and use profiling in  
your generated 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 3-21

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

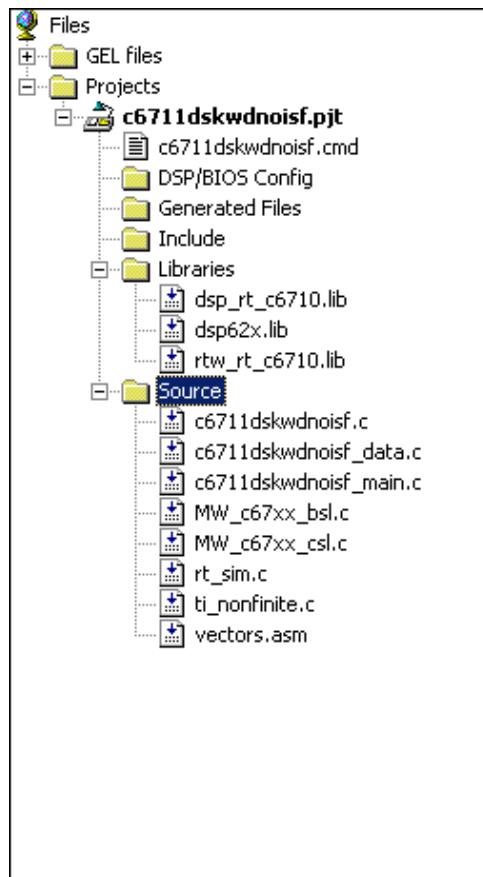
While generating code that includes the DSP/BIOS options is straightforward using the **Incorporate DSP/BIOS** option in the **TIC6000 code generation** options, changes occur between code that does not include DSP/BIOS and code that does. Two things change when you generate code with DSP/BIOS—files 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.

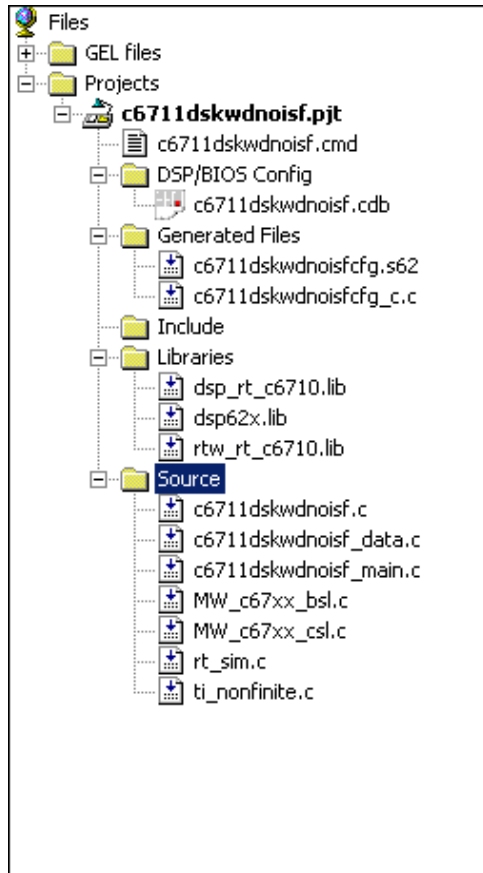
#### Example—c6711dskwdnoisf.pjt code generated without DSP/BIOS

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 the **Profile performance at atomic subsystem boundaries** option in the Real-Time Workshop options, you direct Real-Time Workshop to insert 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. For directions that explain how to use the profiling feature, refer to “Profiling Generated Code” on page 3-10.

Code profiling works only on atomic subsystems in your model. Before you build your model in Real-Time Workshop, to allow Embedded Target for TI C6000 DSP to profile it you must convert the segments of your model to profile into subsystems using **Create subsystem**. By designating one or more subsystems in your model as atomic, you force the subsystem to be executed all at once, that is only at a time step 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.

### About Profiling Subsystems

Nested subsystems are profiled inclusively—the execution time reported for the parent subsystem includes the time spent in any profiled child subsystems.

For models that include multiple sample times, there are circumstances when 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 you select the MultiTasking option for **Mode** (refer to the **Solver** pane in the **Simulation Parameters** dialog), or you set **Mode** to Auto, profiling applies only to single-rate subsystems that execute at the base rate of your model. This

limitation on profiling subsystems arises because all of the generated code segments must execute contiguously for the profiling timing measurements to be correct. The Auto setting does not guarantee contiguous execution for all code segments and subsystems and so does not work for all subsystems.

Notice the STS objects that are placed into the generated code and the generated DSP/BIOS configuration in the 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. Download the 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.

## About 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, use

```
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 MATLAB temporary directory, with the name `profileReport.html`. To locate your temporary directory, type

```
tempdir
```

at the MATLAB prompt. MATLAB returns the path to your temporary directory.

---

**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. 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—we call it critical headroom in the report), which does not include the small overhead required for each interrupt. 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
Overall CPU Statistics	Shows you how your code ran from when you started program execution to when you stopped execution to gather the statistics.
Summary of Subsystem Profiling	Lists the name of each subsystem and the performance of the code in the subsystem.
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. In particular, notice the <code>IDL_busyObj</code> entry. DSP/BIOS uses this to determine the CPU load statistics. For more information about this object, 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.

### **Critical Headroom**

The measured amount of time, in CPU cycles, the CPU spent in idle mode during the worst-case interrupt cycle, when all rates in the code coincide and the code exhibits the maximum measured number of clock cycles spent in idle mode (time the CPU spends waiting to process information). When critical headroom approaches zero, your code is at risk for overrunning. Interpret critical headroom as a portion of the time between interrupts.

### **Time Between Interrupts**

Time in microseconds between interrupts, where the interrupt is generated either by the ADC block or by a PRD timer object.

### **Number of Interrupts 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.

### **CPU Load**

The average CPU usage computed by CCS. Note that the actual load in any particular interrupt cycle may vary greatly from this average, especially in multirate applications.

### **Maximum Time Spent in This Subsystem per Interrupt (Max Time)**

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 Interrupt Interval (Max %)**

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.

## **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 profiling in the Real-Time Workshop.
- 2** Create atomic subsystems to profile in your model.
- 3** Build, download, and run your model.
- 4** 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.

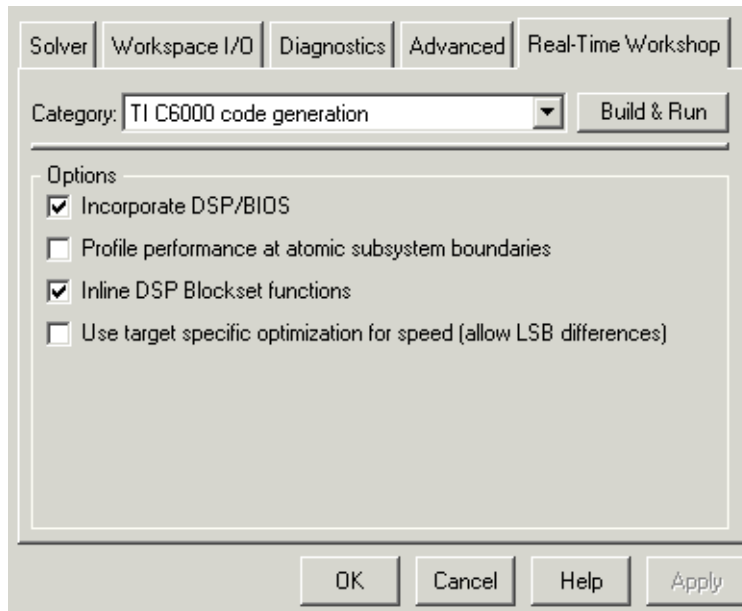
## To Enable Profiling for Your Generated Code

- 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 can 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.

- 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

```
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:** [c6711dskwdnoisf.mdl](#)  
**Target:** C6711DSK

MATLAB Link for Code Composer Studio (tm) Development Tools  
Report of profile data from Texas Instruments (tm) Code Composer Studio  
XX-May-XXXX 12:00:00

---

### Overall CPU Statistics

<b>Critical headroom</b>	7367 $\mu$ s <sup>3</sup>
<b>Time between interrupts</b>	8000 $\mu$ s
<b>Number of interrupts counted</b>	4804
<b>CPU Clock speed</b>	150 MHz <sup>1</sup>
<b>CPU Load</b>	unknown

---

### Summary of Subsystem Profiling

<b>System Name</b>	<b>Max time</b>	<b>Max %</b>
<a href="#">c6711dskwdnoisf/Subsystem</a>	302.8 $\mu$ s	3.8%
<a href="#">c6711dskwdnoisf/Subsystem1</a>	257 $\mu$ s	3.2%
<a href="#">c6711dskwdnoisf</a>	3.387 $\mu$ s	0.042%

## Using DSP/BIOS with Your Target Application

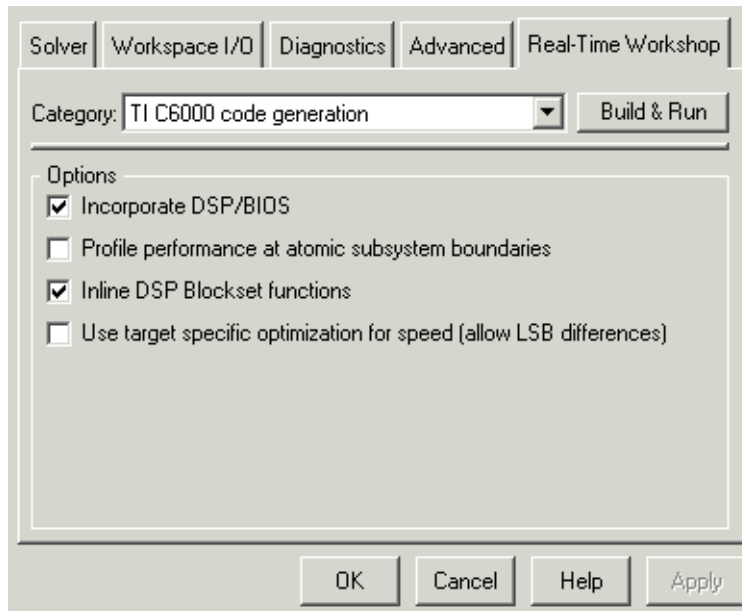
The Embedded Target for TI C6000 DSP lets you build projects and generate code with or without DSP/BIOS included.

### 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 need.
- 6 From the **Category** list, select TI C6000 runtime.
- 7 For the **Build action**, 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.

- 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 6, “Block Reference” 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 Blockset documentation.

You can use C62x DSP Library blocks with certain core Simulink blocks, as well as with certain blocks from the DSP Blockset and Fixed-Point Blockset. 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 6, “Block Reference” 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 Blockset documentation.

You can use C64x DSP Library blocks with certain core Simulink blocks, as well as with certain blocks from the DSP Blockset and Fixed-Point Blockset. 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.

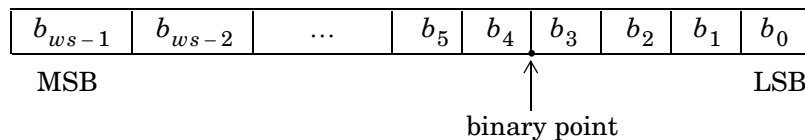
To learn more about characteristics particular to each block in the library, refer to Chapter 6, “Block Reference.”

## 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 Blockset, 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 Blockset, 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 Blockset, 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 Blockset, 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, DSP Blockset, and Fixed-Point Blockset blocks. This section discusses issues you should consider when you build models with blocks from these libraries.

## Converting Data Types

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

When you connect C62x or C64x DSP Library blocks and Fixed-Point Blockset blocks, you often need to set the data type and scaling in the block parameters of the Fixed-Point Blockset block to match the data type of the C62x DSP Library block. Many Fixed-Point Blockset 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 Fixed-Point Blockset block to match a connected C62x DSP Library block.

Some DSP Blockset blocks and core 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 DSP 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 Gateway In and Gateway Out blocks in the Data Type library of the Fixed-Point Blockset.
- To provide an interface to nonfixed-point blocks, use the Convert Floating-Point to Q.15 and 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 Conversion and Conversion Inherited blocks in the Data Type library of the Fixed-Point Blockset.

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 DSP 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 DSP 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.





# Using FDATool with the Embedded Target for TI C6000 DSP

---

Guidelines on Exporting Filters from FDATool to CCS IDE (p. 5-3)

Tutorial—Exporting Filters from FDATool to CCS IDE (p. 5-9)

Things to think about when you export filters to Code Composer Studio®

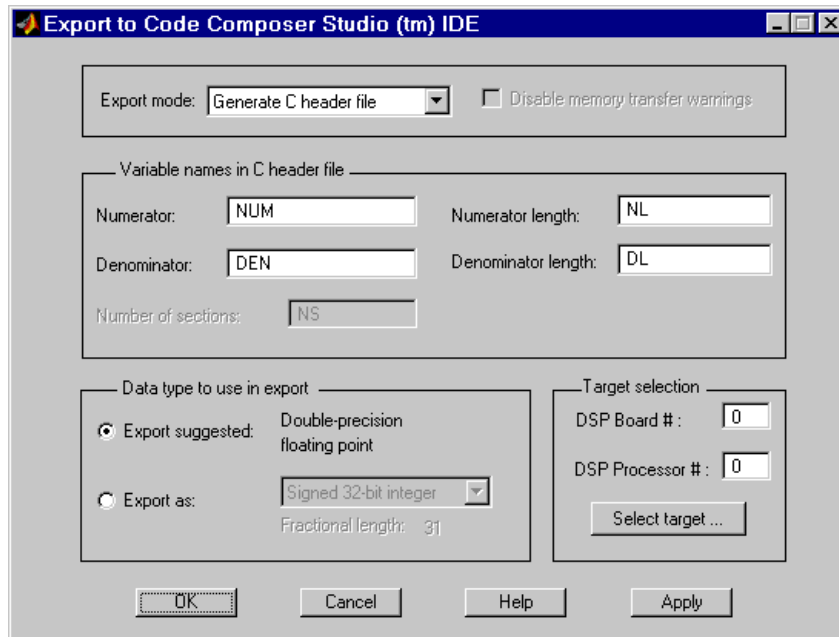
Takes you through the process of exporting a filter from FDATool to Code Composer Studio

The Filter Design and Analysis Tool (FDATool) in the Signal Processing Toolbox is a powerful user interface used for designing and analyzing filters. FDATool enables you to quickly design digital FIR or IIR filters by setting filter performance specifications, by importing filters from your MATLAB workspace, or by directly specifying filter coefficients. FDATool also provides tools for analyzing filters, such as magnitude and phase response plots and pole-zero plots.

Once you have designed a filter in FDATool, you can export it to Code Composer Studio<sup>®</sup> Integrated Development Environment (CCS IDE) to test the filter on an actual target digital signal processor (DSP). Using FDATool with CCS IDE takes filter design to the next level by enabling you to quickly test your filter on a target DSP, tune and optimize the filter in FDATool, and test your redesigned filter on the target.

## Guidelines on Exporting Filters from FDATool to CCS IDE

You can export filters from FDATool to CCS IDE by generating a C header file, or by writing the filter coefficients directly to target memory. To export a filter from FDATool to CCS IDE, use the **Export to Code Composer Studio (tm) IDE** dialog (shortened to **Export to CCS IDE** dialog in this section). Open the dialog from the FDATool **Targets** menu.



For guidelines on exporting filters using the **Export to CCS IDE** dialog, see the following sections:

- “Selecting the Export Mode” on page 5-4
- “Cautions Regarding Writing Directly to Memory” on page 5-4
- “Variables and Memory Necessary for Filter Export” on page 5-5
- “Selecting the Export Data Type” on page 5-7

For instructions on using the **Export to CCS IDE** dialog, see “Tutorial—Exporting Filters from FDATool to CCS IDE” on page 5-9.

## Selecting the Export Mode

You can export a filter by generating a C header file, or by writing the filter coefficients directly to target memory. The following table summarizes when and how to use the two export modes.

Export Mode	When to Use	Suggested Use
Generate C header file	You have not yet allocated memory in your target DSP for the filter coefficients to export.  (For a sample generated header file, see “Contents of the C Header File Generated in Task 1” on page 5-14.)	Create a program file from the generated C header file. Loading this program file into your target allocates static memory locations in the target, and exports your filter coefficients to these memory locations. You may want to edit the header file so that the program file allocates extra target memory, providing you more freedom to change your filter. See “Allocating Sufficient or Extra Memory for Filter Coefficients” in the next section.
Write directly to memory	You have already allocated memory in your target DSP for the filter coefficients to export.	Tune your filter coefficients in FDATool, and write the updated filter coefficients directly to the allocated target memory. See the next section, “Cautions Regarding Writing Directly to Memory”.

## Cautions Regarding Writing Directly to Memory

When you write filter coefficients directly to target memory, you need to allocate sufficient memory for the coefficients, and proceed with caution when you update your filter coefficients in target memory.

### Allocating Sufficient or Extra Memory for Filter Coefficients

When you export filter coefficients directly to target memory, the filter coefficients overwrite as many memory locations as they need. The export process does not check whether you allocated sufficient memory for your filter coefficients. You must allocate enough memory for your filter coefficients or you

may get unexpected results. To ensure you allocate enough target memory for your filter, export the filter by generating a C header file, as described in “Tutorial—Exporting Filters from FDATool to CCS IDE” on page 5-9.

You can allocate extra memory by editing the generated C header file, and then loading the associated program file into your target as described in the tutorial in “Step 8—Export the Filter by Generating a Program File” on page 5-13. Allocating extra memory provides more freedom for changing a filter and overwriting its previous version stored in target memory. Even after you allocate extra memory, you should still proceed with caution when overwriting old filter coefficients with updated coefficients as discussed in the next section.

### **Overwriting Old Filter Coefficients with Updated Coefficients**

When you tune a filter to overwrite its previous version in target memory, carefully consider changes that increase the memory required to store the filter coefficients, or that alter the export data type.

**Do Not Tune a Filter’s Export Data Type.** Never tune a filter by changing its data type, because the allocated memory expects the data type of the first version of the filter that you exported. Overwriting a filter with a filter that has a different data type usually yields unexpected results.

**Be Wary of Filter Changes that Increase Memory Required to Store Filter Coefficients.** If you do not allocate extra memory when exporting the first version of your filter, do not tune the filter in ways that increase the memory required to store its coefficients. For instance, you should not increase the order of the filter. When you overwrite your original filter with one of a higher order, the updated filter may overwrite data in memory locations that you did not intend for storing filter coefficients. Even if you do allocate extra memory for your filter coefficients, be cautious about making changes that increase the memory required to store the coefficients. Examples of such changes include

- Changing an FIR filter to an IIR filter
- Increasing the filter order
- Increasing the number of filter sections

### **Variables and Memory Necessary for Filter Export**

When you export a filter by generating a C header file, the header file stores the filter coefficients in filter coefficient variables. You must name these

variables in the **Export to CCS IDE** dialog. Variable names cannot be reserved words of the C programming language, such as `if`. By generating a program file from the C header file and loading the program file into your target, the filter coefficient variables in the header file appear in the target application symbol table.

When you export a filter by writing directly to target memory, the target stores the filter coefficients in memory locations. These memory locations correspond to filter coefficient variables in the target application symbol table. To export directly to target memory, you specify these variables in the **Export to CCS IDE** dialog.

The necessary filter coefficient variables depend on the structure of your filter. The **Export to CCS IDE** dialog provides you with the following parameters to specify or name the necessary filter coefficient variables. The dialog activates only the parameters you need to set; the others become invisible or inactive.

<b>Parameters for Specifying Filter Coefficient Variables</b>	<b>Description</b>
<b>Numerator</b>	Numerator filter coefficients
<b>Numerator length</b>	Number of numerator filter coefficients
<b>Denominator</b>	Denominator filter coefficients
<b>Denominator length</b>	Number of denominator filter coefficients
<b>Lattice coeffs</b>	Lattice coefficients
<b>Lattice coeffs length</b>	Number of lattice coefficients
<b>Ladder coeffs</b>	Ladder coefficients
<b>Ladder coeffs length</b>	Number of ladder coefficients
<b>Number of sections</b>	Number of filter sections (parameter is inactive if your filter has only one section)

In the following table, x marks indicate the parameters you need to set for each filter structure.

## Filter Coefficient Variables Necessary for Exporting Filters with Various Structures

### Parameters for Naming Filter Coefficient Variables

Filter Structures	Numerator	Numerator length	Denominator	Denominator length	Lattice coeffs	Lattice coeffs length	Ladder coeffs	Ladder coeffs length	Number of sections (Inactive for filters with one section.)
df1	X	X	X	X					X
df1t	X	X	X	X					X
df2	X	X	X	X					X
df2t	X	X	X	X					X
fir	X	X							X
firt	X	X							X
latticearma					X	X	X	X	X
latticeima					X	X			X

## Selecting the Export Data Type

When you export a filter, the export process suggests the export data type that best preserves the performance of your filter. Use the suggested export data type by selecting **Export suggested** in the **Export to CCS IDE** dialog. The supported export data types are

- Signed integer (8-, 16-, or 32-bit)
- Unsigned integer (8-, 16-, or 32-bit)

- Double-precision floating point
- Single-precision floating point

### Recommended Procedure for Selecting Export Data Type

By adhering to the following procedure when you set the export data type of your filter, the exported filter coefficients closely match the coefficients of the filter you designed in FDATool.

**Step 1 — Set the Numerical Precision of Your Filter in FDATool.** Set the numerical precision of your filter in FDATool by using the **Quantized Filter** pane, available when you install the Filter Design Toolbox. If you do not have the Filter Design Toolbox, your filters in FDATool have the default precision—double-precision floating point.

**Step 2 — Select an Export Data Type in the Export to CCS IDE Dialog.** Use the export data type indicated by the **Export suggested** parameter in the **Export to CCS IDE** dialog. See the following note.

Though Step 2 insists you use the **Export suggested** parameter, you may find it useful to select the **Export as** option and select an export data type other than the one suggested. However, if you deviate from the suggested data type, the exported filter coefficients can be very different from the coefficients of the filter you designed in FDATool.

---

**Note** When you design your filter using an unsupported data type, the **Export to CCS IDE** dialog rounds the word length up to the next supported data type, and maintains the specified difference between the word length and fraction length. For example, for a filter with 14-bit word length and an 11-bit fraction length, the **Export suggested** parameter sets the export data type to a signed 16-bit integer with a 13-bit fraction length.

---



## Tutorial—Exporting Filters from FDATool to CCS IDE

This tutorial shows you how to export filters from FDATool to CCS IDE with the **Export to CCS IDE** dialog. The tutorial covers exporting filters by generating C header files, and by writing filter coefficients directly to the target memory. Also see the previous section, “Guidelines on Exporting Filters from FDATool to CCS IDE” on page 5-3.

### Descriptions of the Two Tutorial Tasks

“Task 1—Export Filter by Generating a C Header File” — You should complete this task before starting Task 2. Exporting a filter by generating a C header file not only exports your filter; it also ensures that you allocate enough target memory for the exported filter coefficients.

“Task 2—Export Filter by Writing Directly to Target Memory” — You should complete Task 1 before starting this task to ensure you allocate enough target memory for the filter coefficients to export. Exporting directly to target memory is useful when you want to repeatedly tune your filter in FDATool, and then export the updated filter coefficients directly to the allocated target memory.

### Setting Up for the Tutorial

To complete this tutorial, you must install both the Signal Processing Toolbox and the Embedded Target for the TI TMS320C6000 DSP Platform (shortened to Embedded Target for TI C6000 DSP in this section). You do not need to open CCS IDE before starting the tutorial.

## Task 1—Export Filter by Generating a C Header File

In Task 1, you export a filter by generating a C header file. The generated C header file defines global arrays of filter coefficients that correspond to static memory locations in the final target program. By generating a program file from the C header file and loading the program file into your target, not only do you export your filter, but you ensure that you allocated enough memory for the exported filter coefficients. In Task 2, you write filter coefficients directly to the memory allocated in Task 1. You should complete Task 1 before starting Task 2.

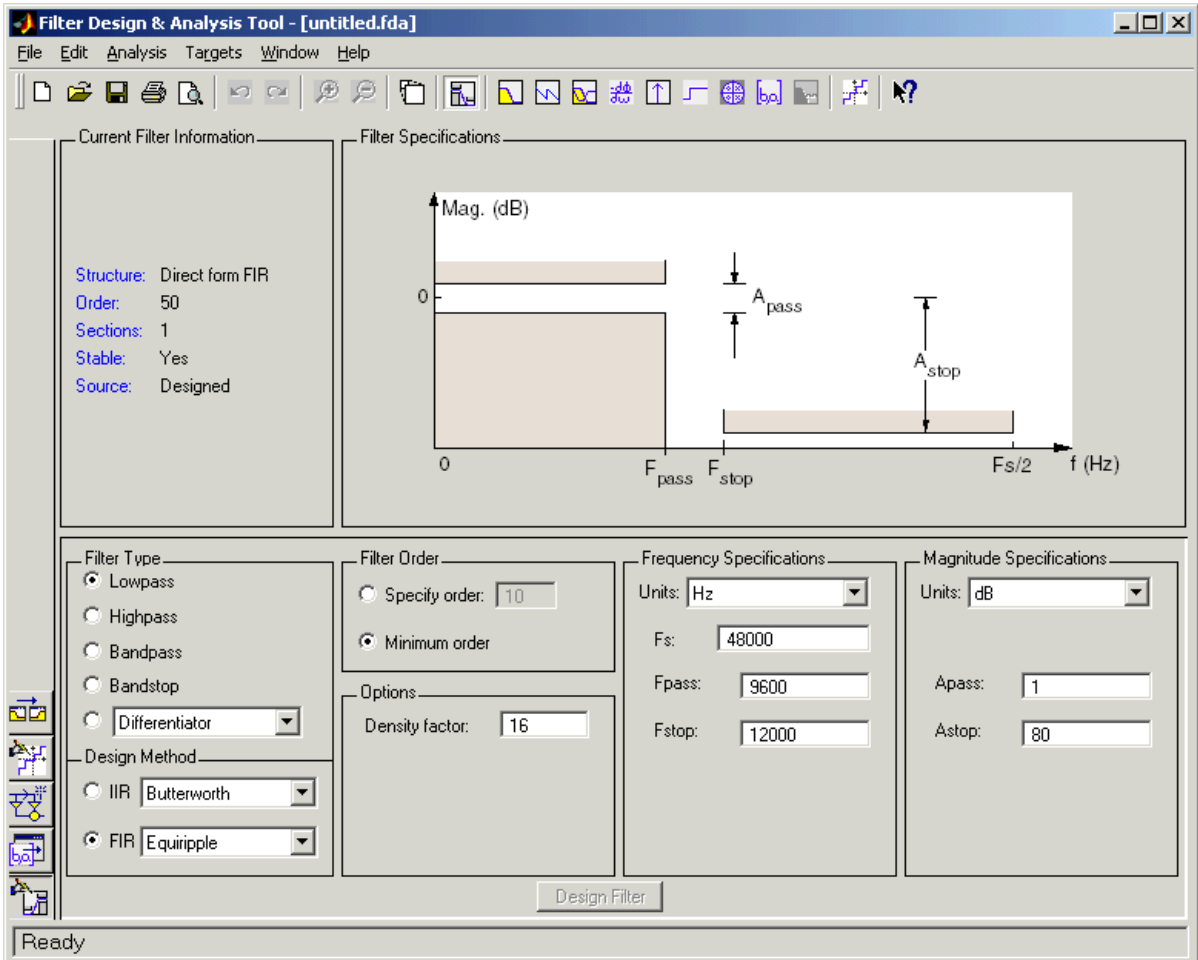
### Step 1—Open FDATool

Open FDATool by typing `fdatool` in the MATLAB command window.

## 5 Using FDATool with the Embedded Target for TI C6000 DSP

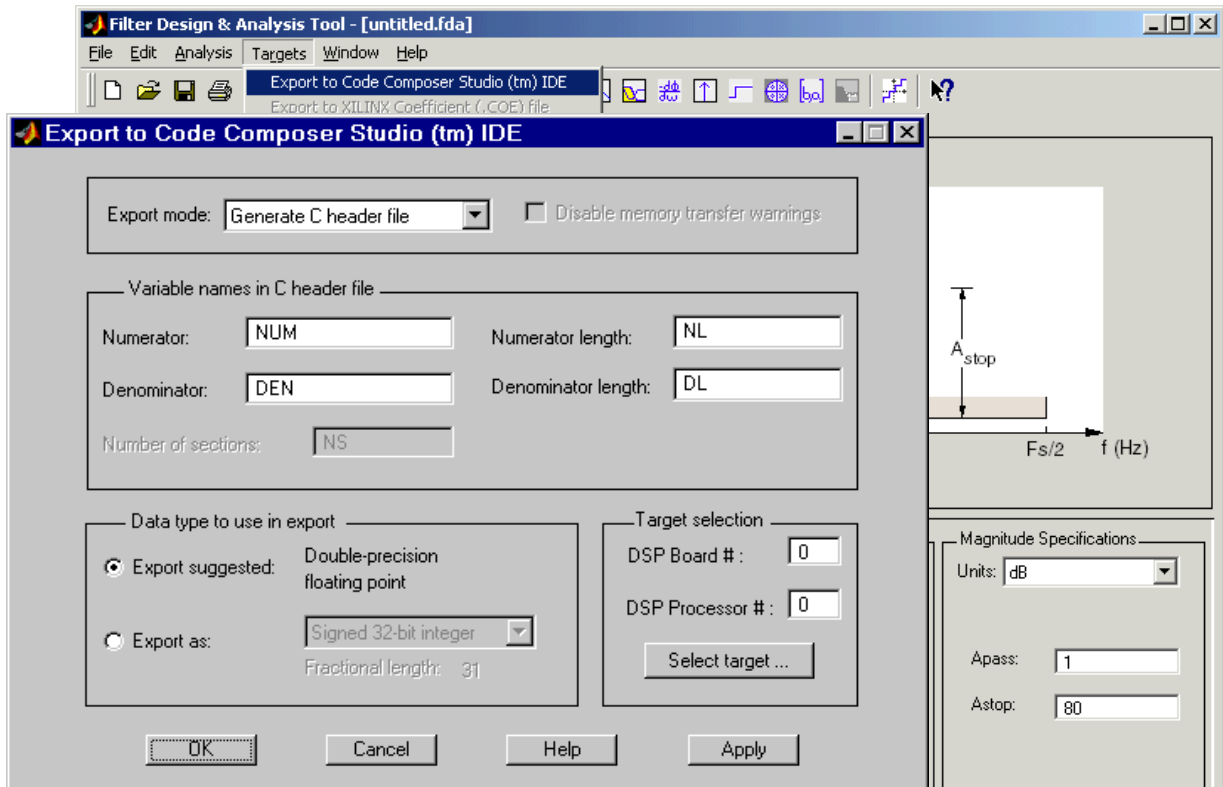
```
fdatool % Open FDATool
```

FDATool opens with a default lowpass equiripple FIR filter that you export in this tutorial (you do not need to design a filter for this tutorial).



## Step 2—Open the Export to Code Composer Studio (tm) IDE Dialog

Open the **Export to CCS IDE** dialog by selecting **Targets -> Export to Code Composer Studio (tm) IDE** from the FDATool menu bar.



## Step 3—Set the Export Mode

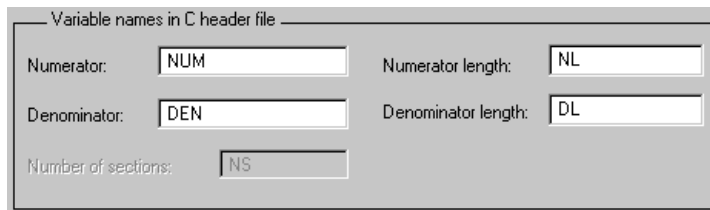
Set **Export mode** to **Generate C header file**.



## Step 4—Name the Filter Coefficient Variables

You must name the variables that store the filter coefficients in the generated C header file by setting the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters. (These correspond to the four variables for the numerator filter coefficients, denominator filter coefficients, number of numerator coefficients, and number of denominator coefficients.) For this tutorial, use the default variable names, NUM, DEN, NL, and DL.

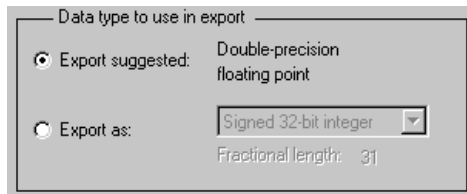
The generated C header file will define global arrays, NUM, DEN, NL, and DL, that correspond to static memory locations containing the filter coefficients in the final target program.



The dialog box titled "Variable names in C header file" contains four input fields. The "Numerator" field is set to "NUM", "Numerator length" is "NL", "Denominator" is "DEN", and "Denominator length" is "DL". There is also a "Number of sections" field set to "NS".

## Step 5—Select a Data Type

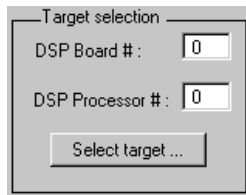
Use the suggested data type to export your filter coefficients by selecting the **Export suggested** parameter.



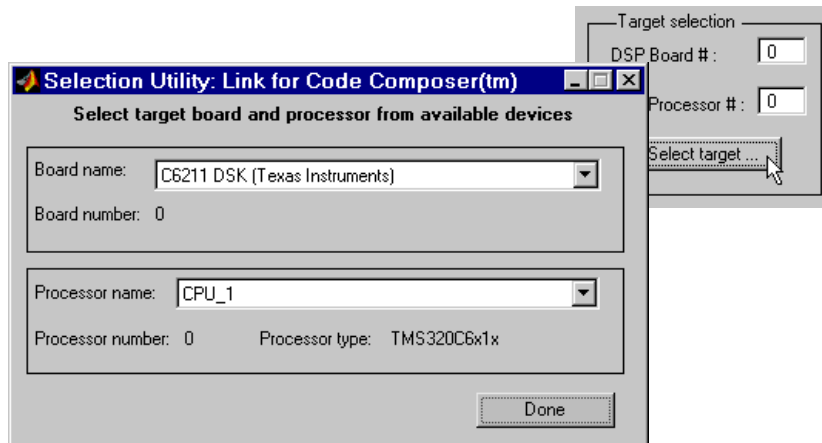
The dialog box titled "Data type to use in export" has two radio buttons. The "Export suggested" radio button is selected, with the text "Double-precision floating point" next to it. The "Export as:" radio button is unselected, with a dropdown menu set to "Signed 32-bit integer" and "Fractional length: 31" below it.

## Step 6—Select a Target

If you know the board number and processor number of your target DSP, select your target by setting the **DSP Board #** and **DSP Processor #** values.



Alternatively, click **Select target...**, which opens the **Selection Utility: Link for Code Composer(tm)** dialog. Select the board name and processor name of the DSP target and click **Done**. This automatically sets the **DSP Board #** and **DSP Processor #** values in the **Export to CCS IDE** dialog.



### Step 7—Generate the C Header File

Click **Apply** to generate the C header file. This opens the generated C header file in CCS IDE. (CCS IDE will be opened for you if you did not have it open.)

### Step 8—Export the Filter by Generating a Program File

Add the generated C header file to an appropriate project, generate a program file, and load the program file into your target DSP. The program file allocates static memory locations in the target, and writes the filter coefficients to these locations. See the following note.

By completing steps 1 through 8, you allocated target memory for the filter coefficients and exported the coefficients to these memory locations. Now you

can tune the filter in FDATool, then export the updated filter coefficients directly to the allocated memory locations as described in Task 2 of this tutorial.

---

**Note** You may want to edit the generated C header file so the associated program file allocates extra target memory. This allows you to change your filter and export the new filter coefficients directly to the allocated memory without having to worry about whether there is enough memory. For example, in the following header file, you could modify `const real64_T NUM[47] = {...}` to `const real64_T NUM[256] = {...}` to allow NUM to store up to 256 numerator filter coefficients rather than 47.

---

### Contents of the C Header File Generated in Task 1

```
/*
 * Filter Design and Analysis Tool - Generated Filter Coefficients
 * - C Source
 *   Generated by MATLAB - Signal Processing Toolbox
 */
/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * D:\v5\extern\include\tmwtypes.h
 */
const int NL = 47;
const real64_T NUM[47] = {
    -0.001384463093657, -0.004518981980449, -0.004897044657617,
    0.003407148842561,
    0.01572838996192, 0.01654194333933,
    0.00164298284835, -0.008806408558624,
    0.002021262464639, 0.01578576956127, 0.004464610692757,
    -0.01727452424144,
    -0.008835593007042, 0.02164594139449, 0.0179008188858,
    -0.02459850261385,
    -0.03066089435881, 0.02764920456168, 0.05260956118871,
    -0.02977511581716,
```

```

        -0.09918534387346, 0.03121885582524, 0.3159846926607,
0.4683345369683,
        0.3159846926607, 0.03121885582524, -0.09918534387346,
-0.02977511581716,
        0.05260956118871, 0.02764920456168, -0.03066089435881,
-0.02459850261385,
        0.0179008188858, 0.02164594139449, -0.008835593007042,
-0.01727452424144,
        0.004464610692757, 0.01578576956127,
0.002021262464639, -0.008806408558624,
        0.00164298284835, 0.01654194333933, 0.01572838996192,
0.003407148842561,
        -0.004897044657617, -0.004518981980449, -0.001384463093657
};
const int DL = 1;
const real64_T DEN[1] = {
        1
};

```

## Task 2—Export Filter by Writing Directly to Target Memory

In Task 2 you export a filter by writing the filter coefficients directly to target memory. Before starting this task, you must allocate enough target memory for the filter coefficients by completing “Task 1—Export Filter by Generating a C Header File”. Once you have allocated enough target memory, you can tune your filter in FDATool and export the updated filter coefficients directly to the allocated target memory by following the steps in this task. For important guidelines on writing directly to target memory, see “Cautions Regarding Writing Directly to Memory” on page 5-4.

### Step 9—Tune Your Filter in FDATool

Tune your filter coefficients in FDATool to improve its performance. Set the numerical precision of your filter by using the **Quantized Filter** pane in FDATool, available when you install the Filter Design Toolbox. If you do not have the Filter Design Toolbox, your filters in FDATool have the default precision, double-precision floating point.

If you have the **Export to CCS IDE** dialog open from Task 1, the dialog automatically updates itself as you tune the filter in FDATool. If you closed the

dialog, reopen it as described in “Step 2—Open the Export to Code Composer Studio (tm) IDE Dialog” on page 5-11.

---

**Note** If you allocated exactly enough memory for the filter coefficients in Task 1, you should not tune your filter such that it requires more memory than did the original filter (by increasing the filter order, for example). If you need more memory for your updated filter, allocate extra memory by editing the generated C header file from Task 1 (as described in the previous note), generating a program file, and loading the program file into your target.

---

## Step 10—Set the Export Mode

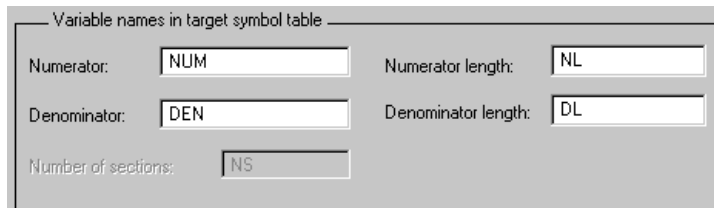
Set **Export mode** to Write directly to memory. Leave the parameter **Disable memory transfer warnings** unchecked so that you get a warning if your target does not support the export data type.



Export mode: Write directly to memory  Disable memory transfer warnings

## Step 11—Input Filter Variable Names

To write to the memory allocated in Task 1, enter the names of the variables in the target symbol table corresponding to the allocated memory. These names are the same as the names of the filter coefficient variables in the C header file from Task 1: NUM, DEN, NL, and DL. You do not need to type these names in, since they are the default setting of the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters. (These parameters correspond to the memory locations that store the numerator filter coefficients, denominator filter coefficients, number of numerator coefficients, and number of denominator coefficients.)



Variable names in target symbol table

Numerator: NUM Numerator length: NL

Denominator: DEN Denominator length: DL

Number of sections: NS



### Step 12—Set All Other Parameters for Export as in Task 1

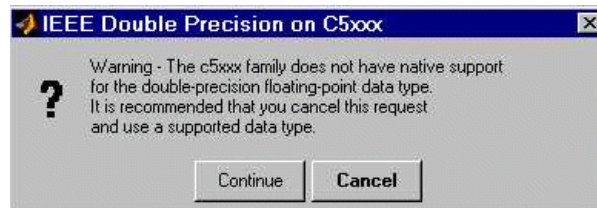
Select an export data type and indicate your target DSP as in Steps 5 and 6 of Task 1.

### Step 13—Load the Program File

Load the program file associated with your target into CCS IDE to activate the target symbol table. The program file must contain the global variables you entered in Step 11.

### Step 14—Export by Writing Directly to Target Memory

Click **Apply** to export your filter. Before the filter export begins, a warning dialog appears if your target does not support the export data type. You can choose to continue to export the filter, or to cancel the export. To prevent this warning dialog from appearing, you can select the parameter **Disable memory transfer warnings** in Step 10.



### Step 15 — Continue Optimizing Filter Performance

Continue to optimize filter performance by retuning your filter in FDATool and exporting the updated filter coefficients directly to target memory. Since you already set up the export process to write to specific memory locations, you can click **Apply** to export updated coefficients to these same memory locations.

When the **Export to CCS IDE** dialog is open, it automatically updates as you tune your filter in FDATool, and preserves the parameter settings from Steps 10 through 13. The dialog stays open as long as you do not click **Cancel** or **OK**. Keep the dialog open when exporting multiple times to the same memory locations so you do not have to repeat Steps 10 through 13, and can just click **Apply**.

### **Where to Find More Information**

For more information on exporting filters from FDATool to CCS IDE, see “Guidelines on Exporting Filters from FDATool to CCS IDE” on page 5-3, which contains the following sections:

- “Selecting the Export Mode”
- “Cautions Regarding Writing Directly to Memory”
- “Variables and Memory Necessary for Filter Export”
- “Selecting the Export Data Type”

To learn how to use FDATool, see the Filter Design and Analysis Tool section in the Signal Processing Toolbox documentation.

Also see the reference pages of the following Embedded Target for TI C6000 DSP functions:

- address
- ccscdsp
- write

# Block Reference

---

Using the Embedded Target for C6000 DSP Block Reference (p. 6-2)	Introduces the format for the block reference pages
Blocks—By Library (p. 6-3)	Provides tables that list each block in the Embedded Target for C6000 DSP by category, such as C6701 EVM or RTDX™
Blocks—Alphabetical List (p. 6-7)	Lists each block in the Embedded Target for C6000 DSP in alphabetical order

## Using the Embedded Target for C6000 DSP Block Reference

These sections provide complete information on each block in the Embedded Target for C6000 DSP c6000lib block library, in a structured form. Refer to these pages when you need details about a specific block. Click **Help** on the **Block Parameters** dialog for the block, or access the block reference page through Help.

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

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

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

## Blocks—By Library

The following reference pages list the blocks included in the Embedded Target for C6000 DSP. Each block page includes Purpose, Library, Description, and Dialog Box sections; Algorithm and Examples sections appear when needed. Where appropriate, a See Also section includes cross references to related blocks and functions.

### Embedded Target for C6000 DSP Blocks in Library **c6701evmlib**

Block	Description
C6701 EVM ADC	Configure digitized signal output from the codec to the processor
C6701 EVM DAC	Use and configure the codec to convert digital input to analog output
C6701 EVM DIP Switch	Simulate or read the three user-defined DIP switches on the C6701 EVM
C6701 EVM LED	Control the light emitting diodes on the C6701 EVM
C6701 EVM RESET	Reset the C6701 Evaluation Module to initial conditions

### Embedded Target for C6000 DSP Blocks in Library **c6711dsklib**

Block	Description
C6711 DSK ADC	Configure digitized signal output from the codec to the processor
C6711 DSK DAC	Use and configure the codec to convert digital input to analog output
C6711 DSK DIP Switch	Simulate or read the three user-defined DIP switches on the C6711 DSK

## Embedded Target for C6000 DSP Blocks in Library `c6711dsklib`

Block	Description
C6711 DSK LED	Control the user-configurable light emitting diodes on the C6711 DSK
C6711 DSK RESET	Reset the C6711 DSP Starter Kit to initial conditions

## Embedded Target for C6000 DSP Blocks in Library `rtdxblocks`

Block	Description
From Rtdx	Add an RTDX communication channel to your model to send data from MATLAB to the model running on your target
To Rtdx	Add an RTDX communication channel to your model to send data from the model running on your target to MATLAB

## Embedded Target for C6000 DSP in the C62x DSP Library

Block	Description
<b>Conversions</b>	
Convert Floating-Point to Q.15	Convert a floating-point signal to a Q.15 fixed-point signal
Convert Q.15 to Floating-Point	Convert a Q.15 fixed-point signal to a single-precision floating-point signal
<b>Filters</b>	
Complex FIR	Filter a complex input signal using a complex FIR filter

## Embedded Target for C6000 DSP in the C62x DSP Library

Block	Description
General Real FIR	Filter a real input signal using a real FIR filter
LMS Adaptive FIR	Perform least-mean-square adaptive FIR filtering
Radix-4 Real FIR	Filter a real input signal using a real FIR filter
Radix-8 Real FIR	Filter a real input signal using a real FIR filter
Real Forward Lattice All-Pole IIR	Filter a real input signal using an auto-regressive forward lattice filter
Real IIR	Filter a real input signal using a real auto-regressive moving-average IIR filter
Symmetric Real FIR	Filter a real input signal using a symmetric real FIR filter
<b>Math and Matrices</b>	
Autocorrelation	Compute the autocorrelation of an input vector or frame-based matrix
Block Exponent	Return the minimum exponent (number of extra sign bits) found in each channel of an input
Matrix Multiply	Perform matrix multiplication on two input signals
Matrix Transpose	Compute the matrix transpose of an input signal
Reciprocal	Compute the fractional and exponential portions of the reciprocal of a real input signal
Vector Dot Product	Compute the vector dot product of two real input signals

## Embedded Target for C6000 DSP in the C62x DSP Library

<b>Block</b>	<b>Description</b>
Vector Maximum Index	Compute the zero-based index of the maximum value element in each channel of an input signal
Vector Maximum Value	Compute the maximum value for each channel of an input signal
Vector Minimum Value	Compute the minimum value for each channel of an input signal
Vector Multiply	Perform element-wise multiplication on two inputs
Vector Negate	Negate each element of an input signal
Vector Sum of Squares	Compute the sum of squares over each channel of a real input
Weighted Vector Sum	Find the weighted sum of two input vectors
<b>Transforms</b>	
Bit Reverse	Bit-reverse the positions of the elements of each channel of a complex input signal
FFT	Compute the decimation-in-frequency forward FFT of a complex input vector
Radix-2 FFT	Compute the radix-2 decimation-in-frequency forward FFT of a complex input vector
Radix-2 IFFT	Compute the radix-2 inverse FFT of a complex input vector



## Blocks—Alphabetical List

C62x Autocorrelation . . . . .	6-10
C62x Bit Reverse . . . . .	6-12
C62x Block Exponent . . . . .	6-14
C62x Complex FIR . . . . .	6-15
C62x Convert Floating-Point to Q.15 . . . . .	6-17
C62x Convert Q.15 to Floating-Point . . . . .	6-18
C62x FFT . . . . .	6-19
C62x General Real FIR . . . . .	6-21
C62x LMS Adaptive FIR . . . . .	6-24
C62x Matrix Multiply . . . . .	6-27
C62x Matrix Transpose . . . . .	6-30
C62x Radix-2 FFT . . . . .	6-31
C62x Radix-2 IFFT . . . . .	6-33
C62x Radix-4 Real FIR . . . . .	6-35
C62x Radix-8 Real FIR . . . . .	6-37
C62x Real Forward Lattice All-Pole IIR . . . . .	6-39
C62x Real IIR . . . . .	6-41
C62x Reciprocal . . . . .	6-44
C62x Symmetric Real FIR . . . . .	6-45
C62x Vector Dot Product . . . . .	6-49
C62x Vector Maximum Index . . . . .	6-50
C62x Vector Maximum Value . . . . .	6-51
C62x Vector Minimum Value . . . . .	6-52
C62x Vector Multiply . . . . .	6-53
C62x Vector Negate . . . . .	6-54
C62x Vector Sum of Squares . . . . .	6-55
C62x Weighted Vector Sum . . . . .	6-56
C6416 DSK ADC . . . . .	6-58
C6416 DSK DAC . . . . .	6-62
C6416 DSK DIP Switch . . . . .	6-64
C6416 DSK LED . . . . .	6-69
C6416 DSK RESET . . . . .	6-70
C64x Autocorrelation . . . . .	6-71
C64x Bit Reverse . . . . .	6-73
C64x Block Exponent . . . . .	6-75

C64x Complex FIR	6-76
C64x Convert Floating-Point to Q.15	6-78
C64x Convert Q.15 to Floating-Point	6-79
C64x FFT	6-80
C64x General Real FIR	6-82
C64x LMS Adaptive FIR	6-84
C64x Matrix Multiply	6-87
C64x Matrix Transpose	6-90
C64x Radix-2 FFT	6-91
C64x Radix-2 IFFT	6-93
C64x Radix-4 Real FIR	6-95
C64x Radix-8 Real FIR	6-97
C64x Real Forward Lattice All-Pole IIR	6-99
C64x Real IIR	6-101
C64x Reciprocal	6-104
C64x Symmetric Real FIR	6-105
C64x Vector Dot Product	6-109
C64x Vector Maximum Index	6-110
C64x Vector Maximum Value	6-111
C64x Vector Minimum Value	6-112
C64x Vector Multiply	6-113
C64x Vector Negate	6-114
C64x Vector Sum of Squares	6-115
C64x Weighted Vector Sum	6-116
C6701 EVM ADC	6-118
C6701 EVM DAC	6-124
C6701 EVM DIP Switch	6-129
C6701 EVM LED	6-133
C6701 EVM RESET	6-135
C6711 DSK ADC	6-136
C6711 DSK DAC	6-140
C6711 DSK DIP Switch	6-142
C6711 DSK LED	6-146
C6711 DSK RESET	6-147
C6713 DSK ADC	6-148
C6713 DSK DAC	6-152
C6713 DSK DIP Switch	6-154

---

C6713 DSK LED .....	6-159
C6713 DSK RESET .....	6-160
From Memory .....	6-161
From Rtdx .....	6-164
TMDX326040 ADC .....	6-168
TMDX326040 DAC .....	6-171
To Memory .....	6-173
To Rtdx .....	6-177

# C62x Autocorrelation

**Purpose** Compute the autocorrelation of an input vector or frame-based matrix

**Library** C62x DSP Library—Math and Matrices

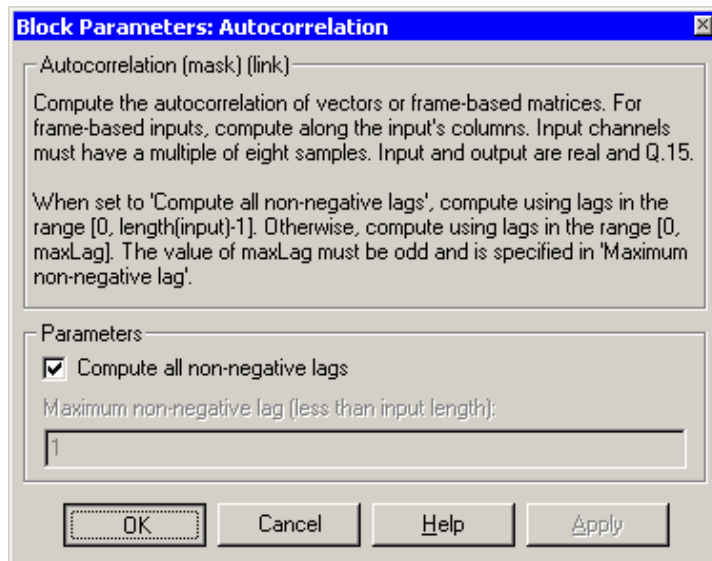
## Description



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, length(input)-1]. When this parameter is not selected, you specify the lags used in **Maximum non-negative lag (less than input length)**.

## **Maximum non-negative lag (less than input length)**

Specify the maximum lag (maxLag) the block should use in performing the autocorrelation. The lags used are in the range [0, maxLag]. The maximum lag must be odd. Enable this parameter by clearing the **Compute all non-negative lags** parameter.

## **Algorithm**

In simulation, the Autocorrelation block is equivalent to the TMS320C62x DSP Library assembly code function DSP\_autocor. During code generation, this block calls the DSP\_autocor routine to produce optimized code.

# C62x Bit Reverse

**Purpose** Bit-reverse the positions of the elements of each channel of a complex input signal

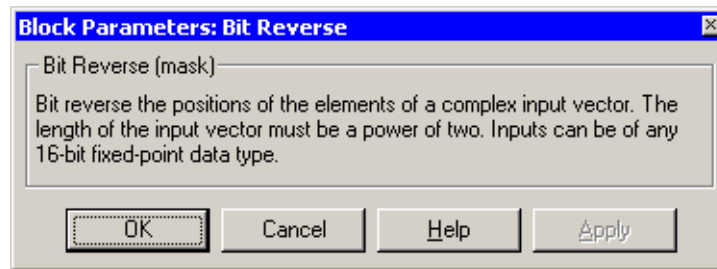
**Library** C62x DSP Library—Transforms

**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.



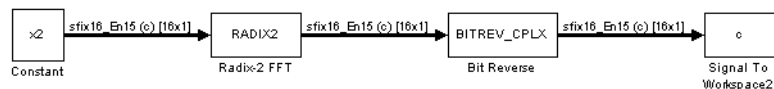
The Bit Reverse block supports discrete sample times and little-endian code generation only.

## 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 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

Radix-2 FFT, Radix-2 IFFT

# C62x Block Exponent

## Purpose

Return the minimum exponent (number of extra sign bits) found in each channel of an input

## Library

C62x DSP Library—Math and Matrices

## Description

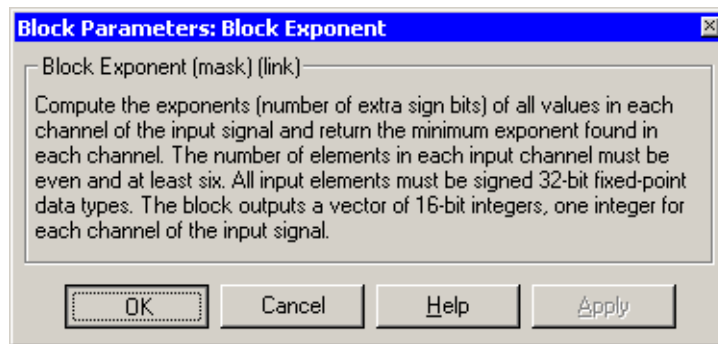


The Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be even and at least six. All input elements must be 32-bit 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 a complex input signal using a complex FIR filter

**Library** C62x DSP Library—Filtering

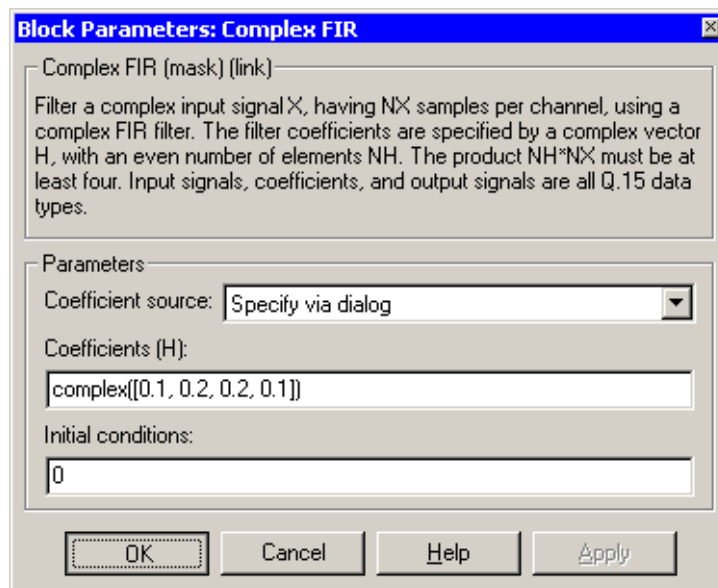
**Description** The Complex FIR block filters a complex input signal  $X$  using a complex FIR filter. This filter is implemented using a direct form structure.



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

# C62x Complex FIR

---

- **Input port**—Accept the coefficients from port H. This port must have the same rate as the input data port X.

## **Coefficients (H)**

Designate the filter coefficients in vector format. 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**

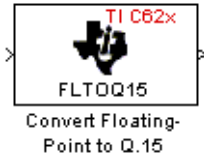
General Real FIR, Radix-4 Real FIR, Radix-8 Real FIR, Symmetric Real FIR

# C62x Convert Floating-Point to Q.15

**Purpose** Convert an input signal to a Q.15 fixed-point signal

**Library** C62x DSP Library—Conversions

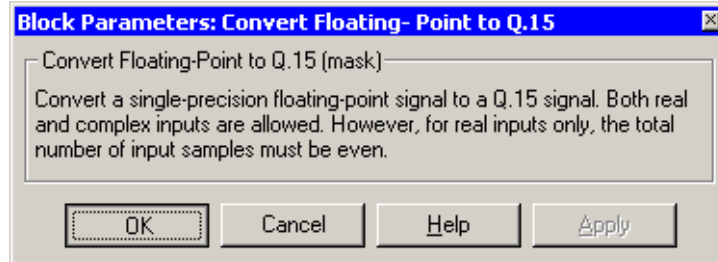
## Description



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** Convert 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 signal

**Library** C62x DSP Library—Conversions

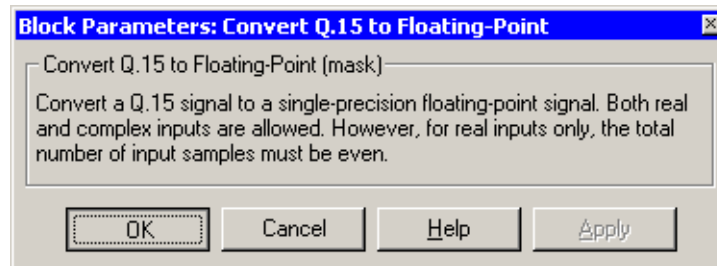
## Description



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** Convert Floating-Point to Q.15

**Purpose** Compute the decimation-in-frequency forward FFT of a 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  $(S-1)$ .

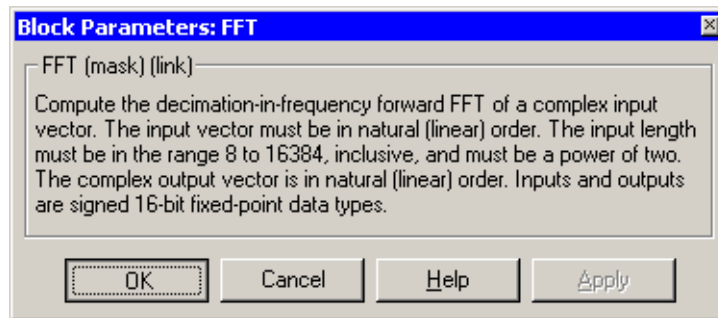
$$\text{OutputFractionalBits} = \text{InputFractionalBits} - (S - 1)$$

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

Radix-2 FFT, Radix-2 IFFT

**Purpose** Filter a real input signal using a real FIR filter

**Library** C62x DSP Library—Filtering

## Description

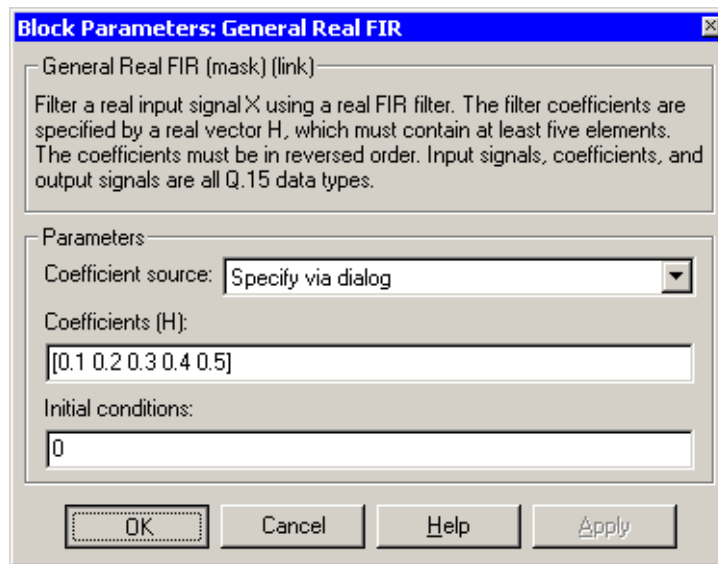


The General Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.

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

## C62x General Real FIR

---

- Input port—Accept the coefficients from port H. This port must have the same rate as the input data port X



## **Coefficients (H)**

Designate the filter coefficients in vector format. This parameter is only visible when Specify via dialog is selected for the **Coefficient source** parameter. 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**

Complex FIR, Radix-4 Real FIR, Radix-8 Real FIR, Symmetric Real FIR

# C62x LMS Adaptive FIR

**Purpose** Filter a scalar input using least-mean-square adaptive 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:

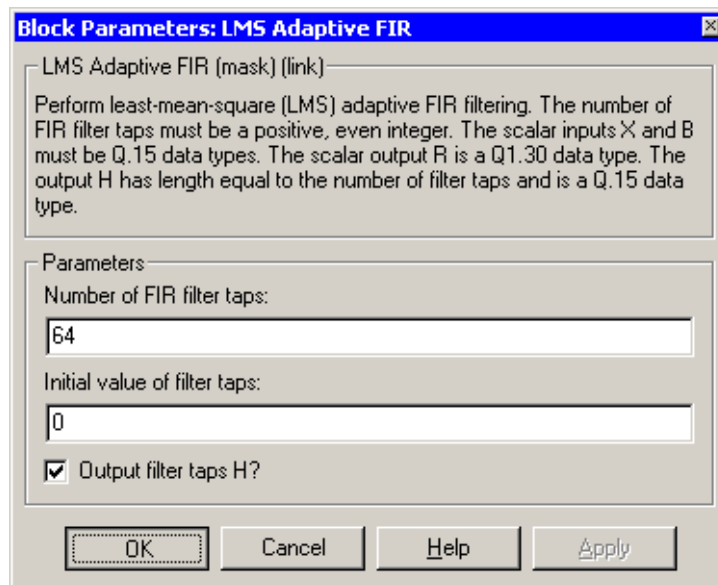
$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)]$$

$d$  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.

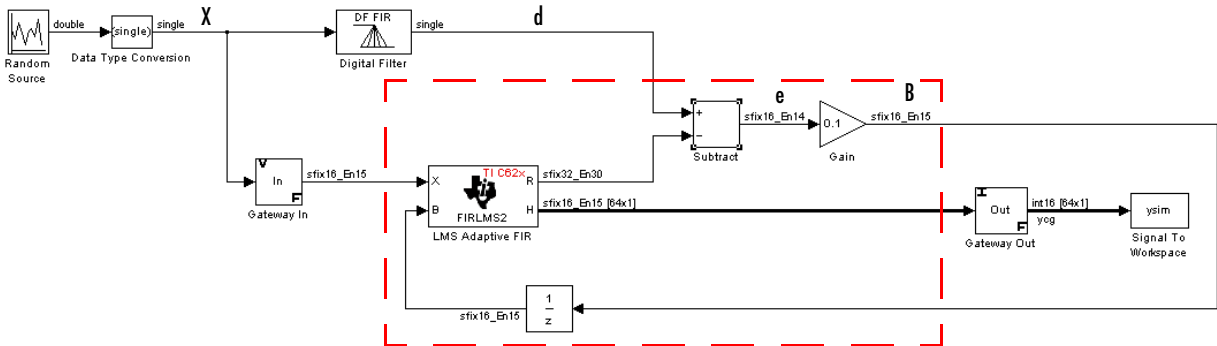
# C62x LMS Adaptive FIR

## 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 `adapt1ms` function in the Filter Design Toolbox. `adapt1ms` performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

To use the LMS Adaptive FIR block you must create the input  $B$  in some way similar to the one shown here. You must also provide the signals  $\bar{X}$  and  $d$ . This model simulates the desired signal  $d$  by feeding  $\bar{X}$  into a digital filter block. You can simulate your desired signal in a similar way, or you may bring  $d$  in from the workspace with a From Workspace or codec block.

**Purpose** Perform matrix multiplication on 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.

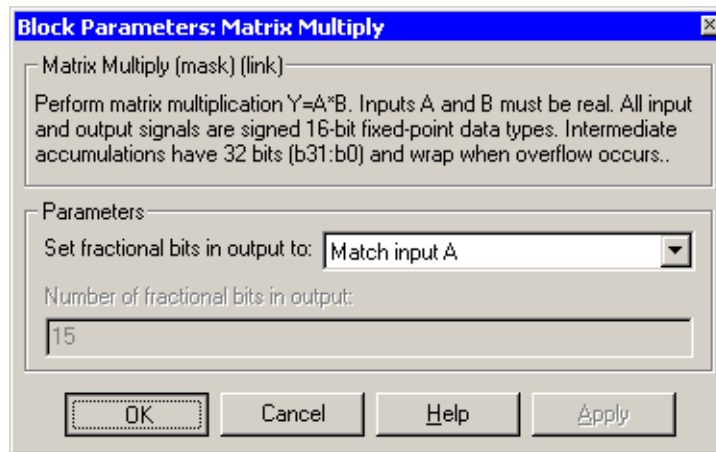
	<b>Input A</b>	<b>Input B</b>	<b>Accumulator Value</b>
<b>Total Bits</b>	16	16	32
<b>Fractional Bits</b>	$R$	$S$	$R + S$

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.

# C62x Matrix Multiply

## Dialog Box



### Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Choose which 16 bits to output from the list:

- **Match input A**—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or *R* in the discussion above).
- **Match input B**—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or *S* in the discussion above).
- **Match high bits of acc. (b31:b16)**—Output the highest 16 bits of the accumulator value.
- **Match high bits of prod. (b30:b15)**—Output the second-highest 16 bits of the accumulator value.
- **User-defined**—Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

### Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is enabled only when you select **User-defined** for **Set fractional bits in output to**.

## Algorithm

In simulation, the Matrix Multiply block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mat_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.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q.15	b30:b15
Match input B	Q.15	b30:b15
Match high bits of acc.	Q1.14	b31:b16
Match high bits of prod.	Q.15	b30:b15

**Example 2** Suppose A is Q12.3 and B is Q10.5. The data type of the resulting accumulator value is therefore Q23.8 ( $R + S = 8$ ). In the accumulator, bits 31:8 are the sign and integer bits, and bits 7:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q12.3	b20:b5
Match input B	Q10.5	b18:b3
Match high bits of acc.	Q23.-8	b31:b16
Match high bits of prod.	Q22.-7	b30:b15

## See Also

Vector Multiply

# C62x Matrix Transpose

**Purpose** Compute the matrix transpose of an input signal

**Library** C62x DSP Library—Math and Matrices

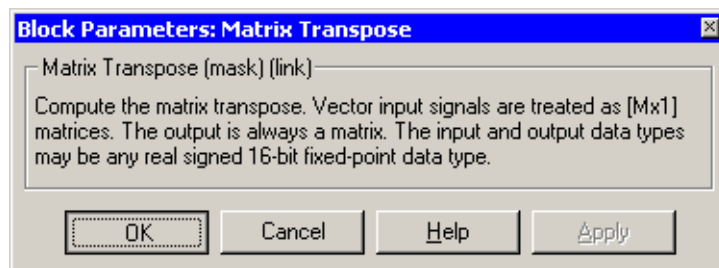
## Description



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.



**Purpose** Compute the radix-2 decimation-in-frequency forward FFT of a complex input vector

**Library** C62x DSP Library—Transforms

## Description

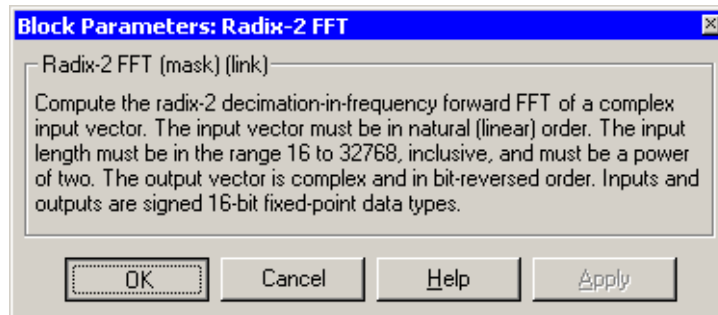


The Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is 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 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



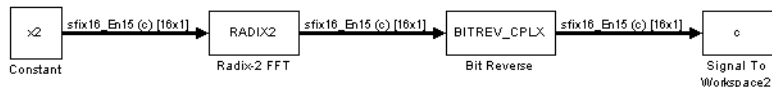
## Algorithm

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.

# C62x Radix-2 FFT

## Examples

The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the 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

Bit Reverse, FFT, Radix-2 IFFT

**Purpose** Compute the radix-2 inverse FFT of a 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.

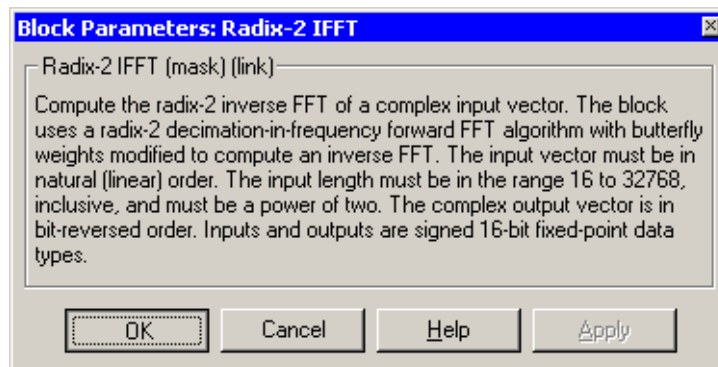
The radix2 routine used by this block employs a radix-2 FFT of length  $L=2^k$ . In order to ensure that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by  $k$  bits to the left relative to the location 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  $k$ .

$$OutputFractionalBits = InputFractionalBits + (k)$$

You can use the 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**



# C62x Radix-2 IFFT

---

## 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

Bit Reverse, FFT, Radix-2 FFT

**Purpose** Filter a real input signal using a real FIR filter

**Library** C62x DSP Library—Filtering

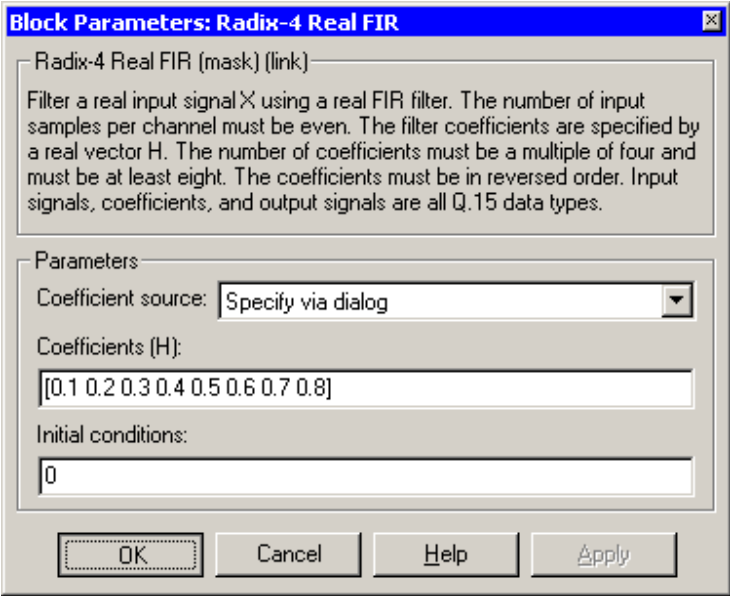
**Description** The Radix-4 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.



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

# C62x Radix-4 Real FIR

---

- Input port—Accept the coefficients from port H. This port must have the same rate as the input data port X

## **Coefficients (H)**

Designate the filter coefficients in vector format. This parameter is only visible when Specify via dialog is selected for the **Coefficient source** parameter. 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**

Complex FIR, General Real FIR, Radix-8 Real FIR, Symmetric Real FIR

**Purpose** Filter a real input signal using a real FIR filter

**Library** C62x DSP Library—Filtering

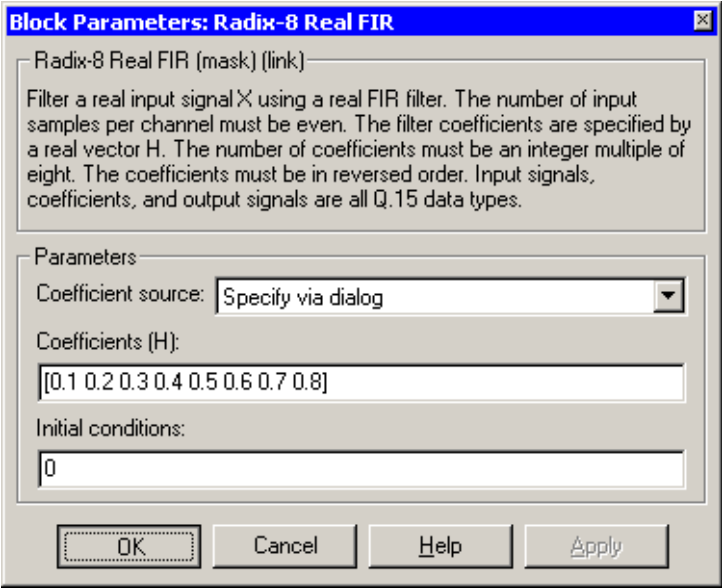
**Description** The Radix-8 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.



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

# C62x Radix-8 Real FIR

---

- Input port—Accept the coefficients from port H. This port must have the same rate as the input data port X

## **Coefficients (H)**

Designate the filter coefficients in vector format. 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**

Complex FIR, General Real FIR, Radix-4 Real FIR, Symmetric Real FIR



# C62x Real Forward Lattice All-Pole IIR

**Purpose** Filter a real input signal using an autoregressive forward lattice filter

**Library** C62x DSP Library—Filtering

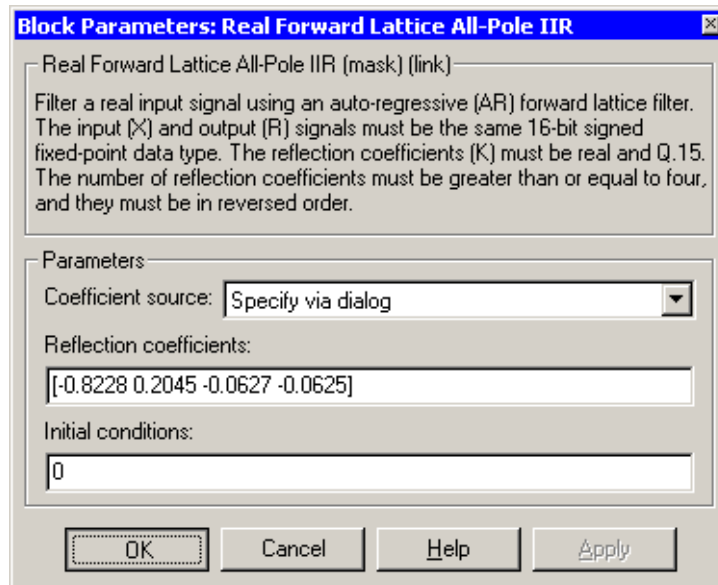
## Description



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

# C62x Real Forward Lattice All-Pole IIR

---

- `Input port`—Accept the coefficients from port K

## Reflection coefficients

Designate the reflection coefficients of the filter in vector format. The number of coefficients must be greater than or equal to four, and they must be in reverse order. Using an even number of reflection coefficients maximizes the speed of your generated code. This parameter is visible when you select `Specify via dialog` for the **Coefficient source** parameter. This parameter is tunable in simulation.

## Initial conditions

If your block initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length (number of elements) of this vector must be the same as the number of reflection coefficients in your filter.
- Different across channels, enter a matrix containing all initial conditions. The number of rows (initial conditions for one channel) of this matrix must be the same as the number of reflection coefficients, and the number of columns of this matrix must be equal to the number of channels.

## Algorithm

In simulation, the Real Forward Lattice All-Pole IIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_iirlat`. During code generation, this block calls the `DSP_iirlat` routine to produce optimized code.

## See Also

Real IIR

**Purpose** Filter a real input signal using a real autoregressive moving-average 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

Block Parameters: Real IIR ✖

Real IIR (mask) (link)

Filter a real input signal  $X$  using a real auto-regressive moving-average (ARMA) IIR filter. There must be five AR coefficients and five MA coefficients; however, the first AR coefficient is assumed to be equal to one. Inputs, coefficients, and output are all Q.15 data types.

Parameters

Coefficient sources:

MA (numerator) coefficients:

AR (denominator) coefficients:

Input state initial conditions:

Output state initial conditions:

## **Coefficient sources**

Specify the source of the filter coefficients:

- **Specify via dialog**—Enter the coefficients in the **MA (numerator) coefficients** and **AR (denominator) coefficients** parameters in the dialog
- **Input ports**—Accept the coefficients from ports MA and AR

## **MA (numerator) coefficients**

Designate the moving-average coefficients of the filter in vector format. There must be five MA coefficients. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

## **AR (denominator) coefficients**

Designate the autoregressive coefficients of the filter in vector format. There must be five AR coefficients, however the first AR coefficient is assumed to be equal to one. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

## **Input state initial conditions**

If the input state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the input state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all input state initial conditions. This matrix must have four rows.

## **Output state initial conditions**

If the output state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the output state initial conditions for one channel. The length of this vector must be four.

- Different across channels, enter a matrix containing all output state initial conditions. This matrix must have four rows.

**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**

Real Forward Lattice All-Pole IIR

# C62x Reciprocal

**Purpose** Compute the fractional and exponential portions of the reciprocal of a real input signal

**Library** C62x DSP Library—Math and Matrices

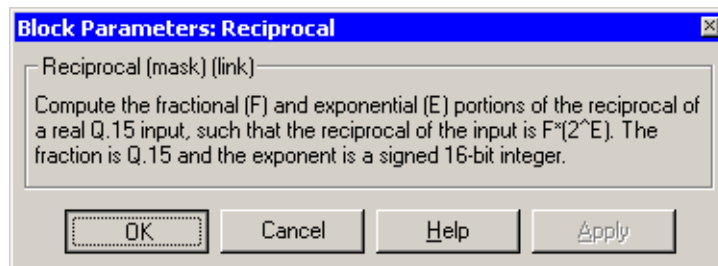
## Description



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.

**Purpose** Filter a real input signal using a symmetric real 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.

# C62x Symmetric Real FIR

## 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



## 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

# C62x Symmetric Real FIR

---

- **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 6-29 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**

Complex FIR, General Real FIR, Radix-4 Real FIR, Radix-8 Real FIR

**Purpose** Compute the vector dot product of two real input signals

**Library** C62x DSP Library—Math and Matrices

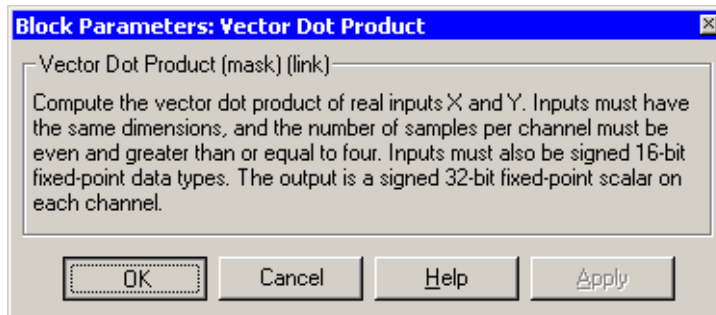
## Description



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.

# C62x Vector Maximum Index

**Purpose** Compute the index of the maximum value element in each channel of an input signal

**Library** C62x DSP Library—Math and Matrices

## Description

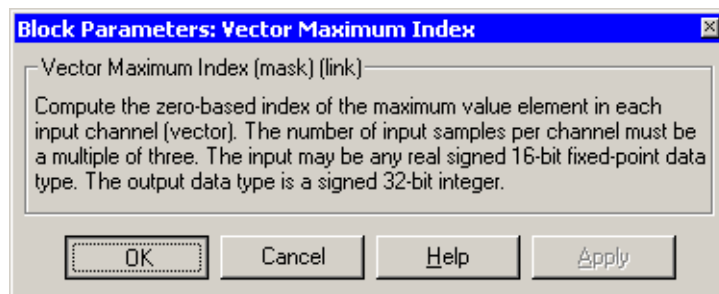


Vector Maximum Index

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.

**Purpose** Compute the maximum value for each channel of an input signal

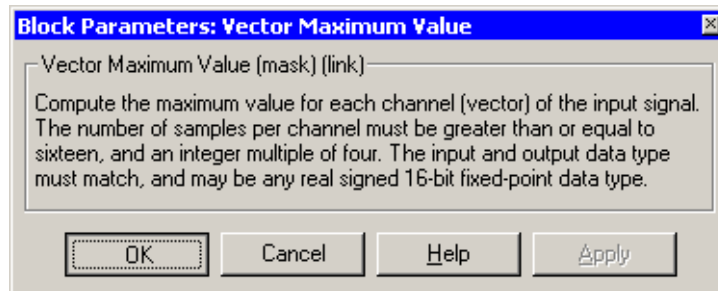
**Library** C62x DSP Library—Math and Matrices

**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** Vector Minimum Value

# C62x Vector Minimum Value

**Purpose** Compute the minimum value for each channel of an input signal

**Library** C62x DSP Library—Math and Matrices

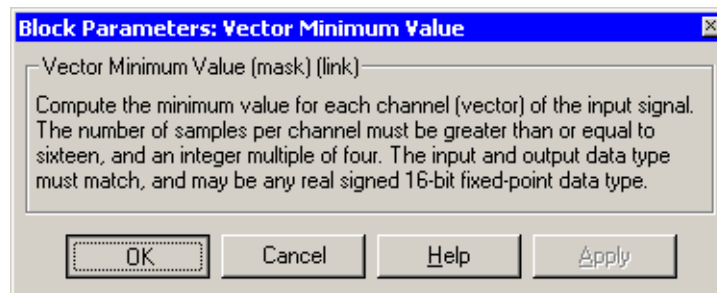
## Description



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

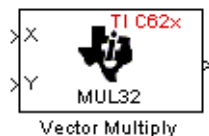
In simulation, the Vector Minimum Value block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_minval`. During code generation, this block calls the `DSP_minval` routine to produce optimized code.

**See Also** Vector Maximum Value

**Purpose** Perform element-wise multiplication on two inputs

**Library** C62x DSP Library—Math and Matrices

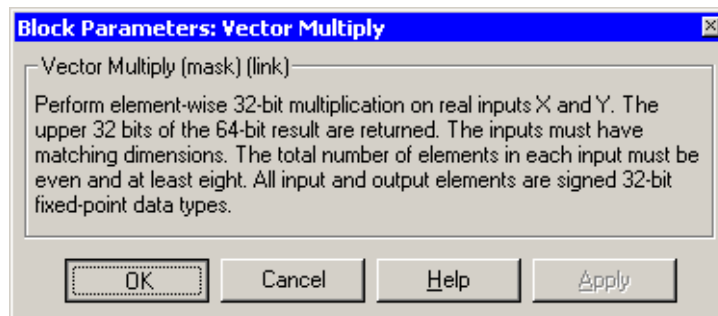
## Description



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

Matrix Multiply

# C62x Vector Negate

**Purpose** Negate each element of an input signal

**Library** C62x DSP Library—Math and Matrices

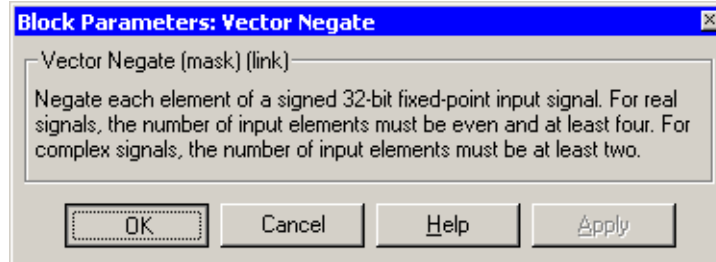
## Description



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** Compute the sum of squares over each channel of a real input

**Library** C62x DSP Library—Math and Matrices

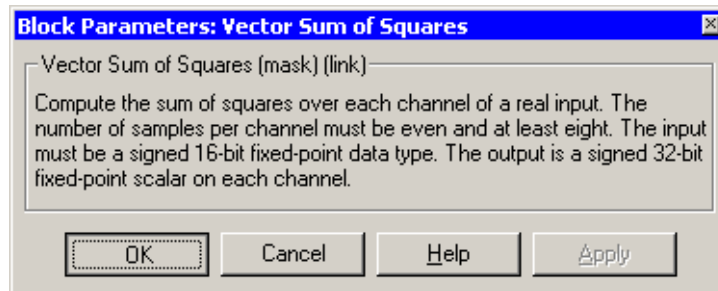
## Description



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.

# C62x Weighted Vector Sum

**Purpose** Find the weighted sum of two input vectors

**Library** C62x DSP Library—Math and Matrices

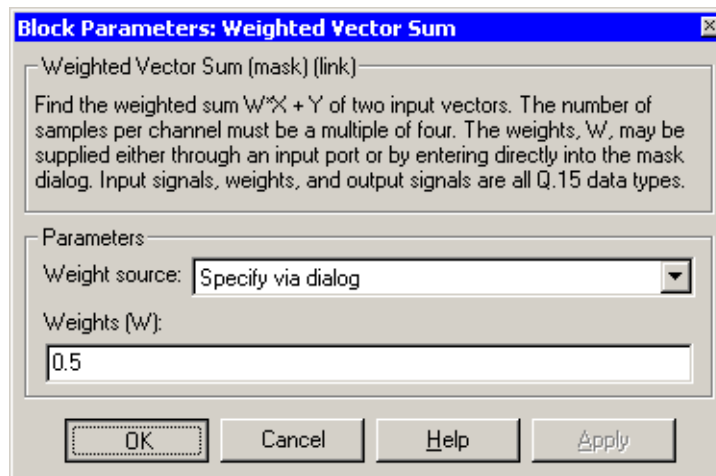
## Description



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

## Weights (W)

This parameter is visible only when Specify via dialog is specified for the **Weight source** parameter. This parameter is tunable in simulation. When the weights are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be a multiple of four.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be a multiple of four, and the number of columns of this matrix must be equal to the number of channels.

Weights must be in the range  $-1 < W < 1$ .

## Algorithm

In simulation, the Weighted Vector Sum block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

# C6416 DSK ADC

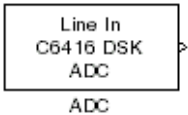
## Purpose

Configure the codec and peripherals to convert analog data from the input ports to digitized signal output for the 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
ADC Source	Codec
Mic	Codec
Output data type	TMS320C6416 digital signal processor
Samples per frame	Direct memory access module
Sample Rate	Codec
Scaling	TMS320C6416 digital signal processor
Word Length	Codec

You can select one of two input sources from the **ADC source** list:

- **Line In**—the codec accepts input from the line in connector (LINE IN) on the board's mounting bracket.
- **Mic**—the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.

Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels on input and output.

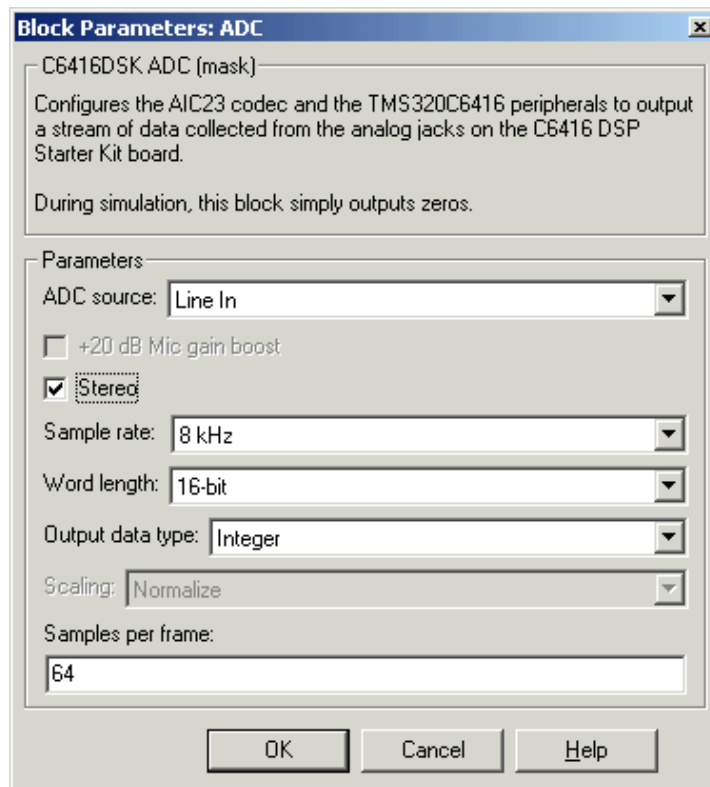
The block uses frame-based processing of inputs, buffering the input data into frames at the specified samples per frame rate. In Simulink, the block puts monaural data into an N-element column vector. Stereo data input forms an N-by-2 matrix with N data values and two stereo channels (left and right).

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.

# C6416 DSK ADC

## 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.

**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.

**See Also**

C6416 DSK DAC

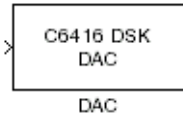
# C6416 DSK DAC

---

**Purpose** Configure the codec and peripherals to convert digital input to analog output at the analog output port of the board

**Library** C6416 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



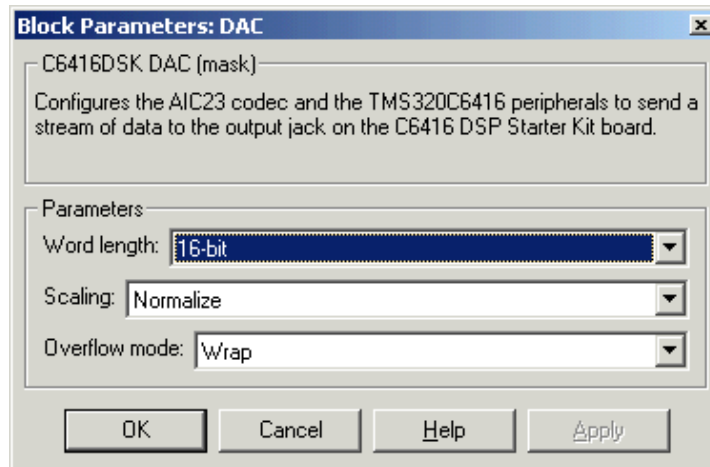
Adding the C6416 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the LINE OUT connection on the C6416 DSK board. When you add the C6416 DSK DAC block, the digital signal received by the codec is converted to an analog signal. After converting 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.

### 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

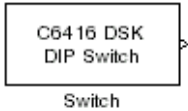
C6416 DSK ADC

# C6416 DSK DIP Switch

**Purpose** Simulate or read the user-defined DIP switches on the C6416 DSK

**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

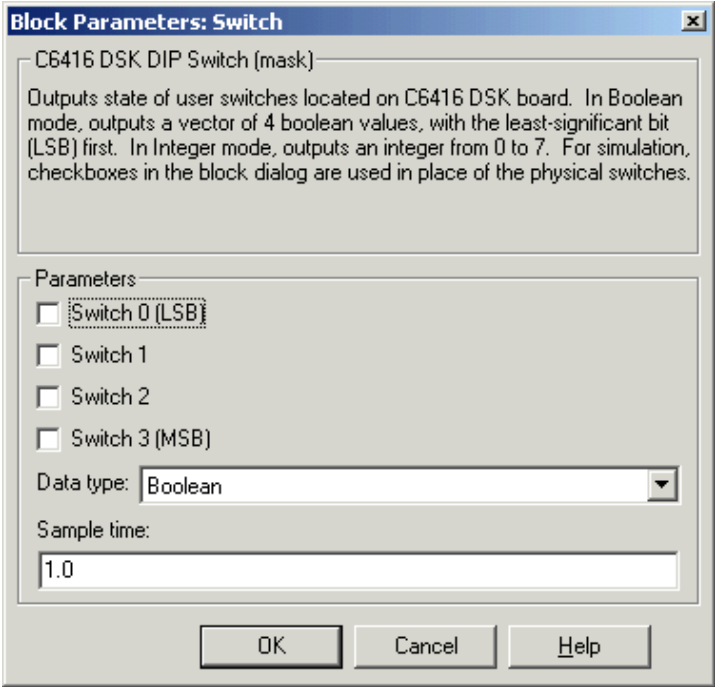
Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2

# 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



### 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.

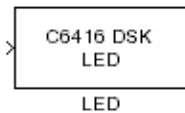
## **Sample time**

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 the user-defined light emitting diodes on the C6416 DSK

**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:

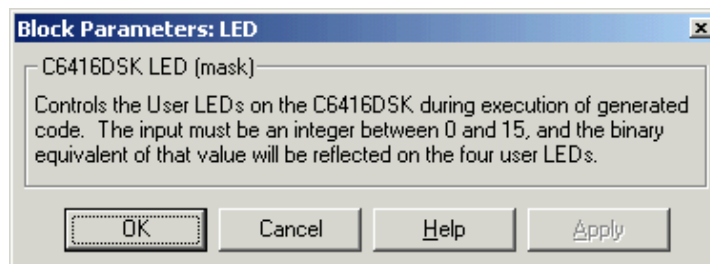
- When the block receives an input value equal to 0, the specified LEDs are turned off (disabled), 0000
- When the block receives a nonzero input value, the specified LEDs are turned on (enabled), 0001 to 1111

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.

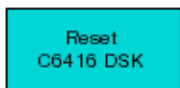
# C6416 DSK RESET

---

**Purpose** Reset the C6416 DSK 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** Compute the autocorrelation of an input vector or frame-based matrix

**Library** C64x DSP Library—Math and Matrices

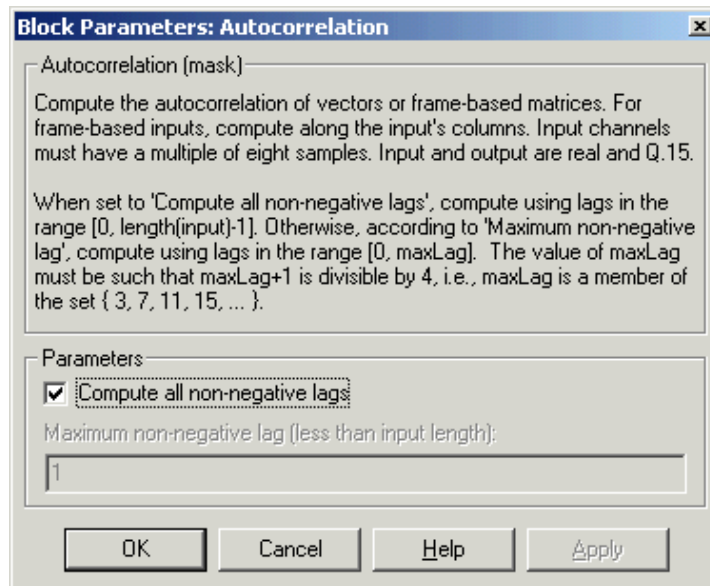
## Description



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)**.

# C64x Autocorrelation

---

## **Maximum non-negative lag (less than input length)**

Specify the maximum lag (`maxLag`) the block should use in performing the autocorrelation. The lags used are in the range  $[0, \text{maxLag}]$ . 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.

## Purpose

Bit-reverse the positions of the elements of each channel of a complex input signal

## Library

C64x DSP Library—Transforms

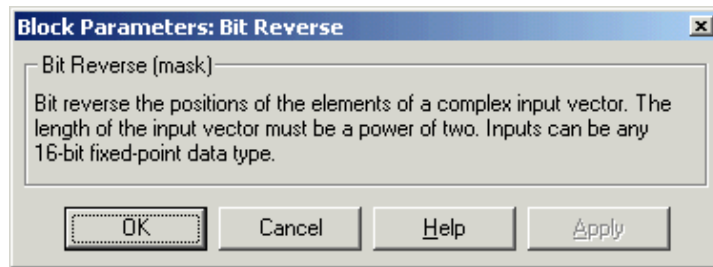
## Description



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

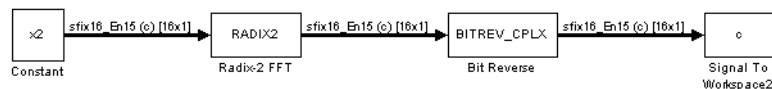


## Algorithm

In simulation, the Bit Reverse block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_bitrev_cp1x`. During code generation, this block calls the `DSP_bitrev_cp1x` routine to produce optimized code.

## Examples

The Bit Reverse block reorders the output of the Radix-2 FFT in the model below to natural order.



# C64x Bit Reverse

---

The following code calculates the same FFT in the workspace. The output from this calculation, `y2`, is displayed side-by-side with the output from the model, `c`. The outputs match, showing that the Bit Reverse block reorders the Radix-2 FFT output to natural order:

```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);

[y2, c]
0.5000          0.5000
0.4619 - 0.1913i 0.4619 - 0.1913i
0.3536 - 0.3536i 0.3535 - 0.3535i
0.1913 - 0.4619i 0.1913 - 0.4619i
0 - 0.5000i      0 - 0.5000i
-0.1913 - 0.4619i -0.1913 - 0.4619i
-0.3536 - 0.3536i -0.3535 - 0.3535i
-0.4619 - 0.1913i -0.4619 - 0.1913i
-0.5000          -0.5000
-0.4619 + 0.1913i -0.4619 + 0.1913i
-0.3536 + 0.3536i -0.3535 + 0.3535i
-0.1913 + 0.4619i -0.1913 + 0.4619i
0 + 0.5000i      0 + 0.5000i
0.1913 + 0.4619i 0.1913 + 0.4619i
0.3536 + 0.3536i 0.3535 + 0.3535i
0.4619 + 0.1913i 0.4619 + 0.1913i
```

## See Also

C64x Radix-2 FFT, C64x Radix-2 IFFT

**Purpose** Return the minimum exponent (number of extra sign bits) found in each channel of an input

**Library** C64x DSP Library—Math and Matrices

## Description

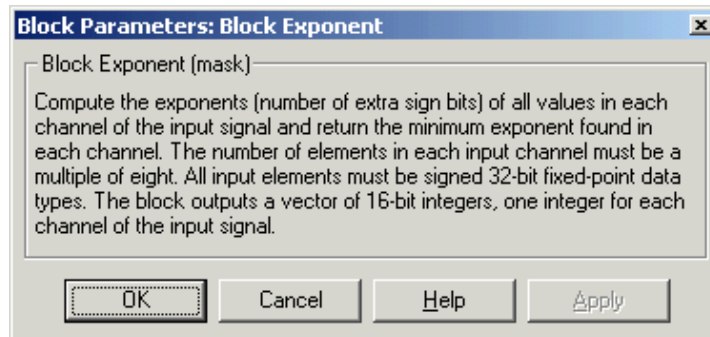


The C64x Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be a multiple of eight. Input elements must be 32-bit 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

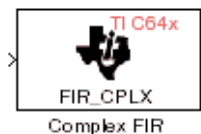
In simulation, the Block Exponent block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_bexp`. During code generation, this block calls the `DSP_bexp` routine given to produce optimized code.

# C64x Complex FIR

**Purpose** Filter a complex input signal using a complex FIR filter

**Library** C64x DSP Library—Filtering

## Description

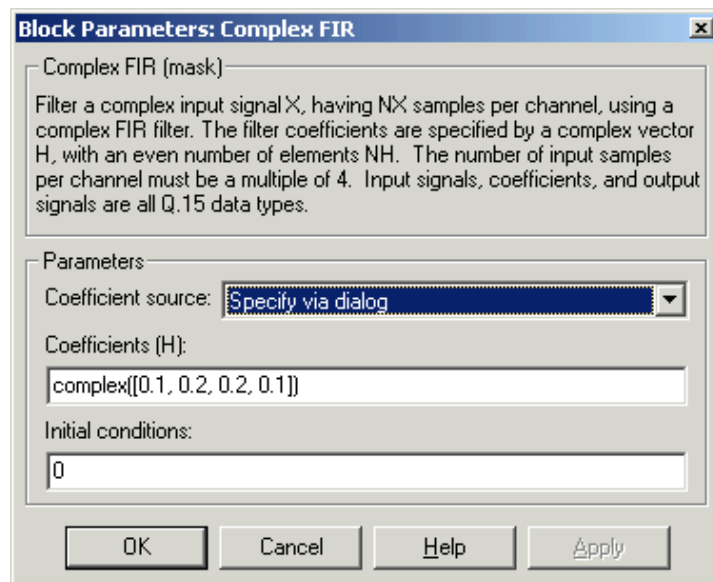


The C64x Complex FIR block filters a complex input signal  $X$  using a complex FIR filter. This filter is implemented using a direct form structure. Each input channel must contain an integer multiple of four samples, with four samples as the minimum required.

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

## See Also

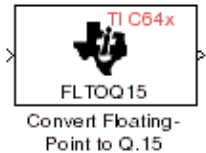
C64x General Real FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

# C64x Convert Floating-Point to Q.15

**Purpose** Convert an input signal to a Q.15 fixed-point signal

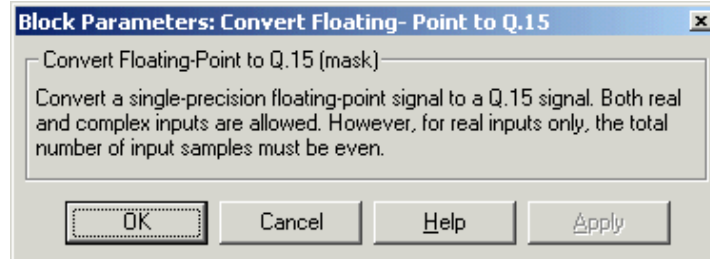
**Library** C64x DSP Library—Conversions

**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

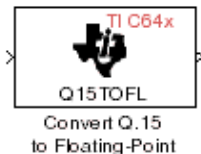


# C64x Convert Q.15 to Floating-Point

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

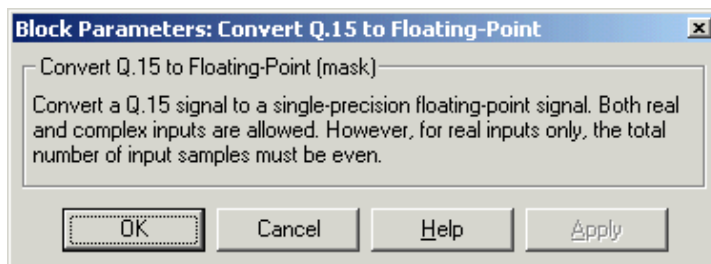
**Library** C64x DSP Library—Conversions

**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

# C64x FFT

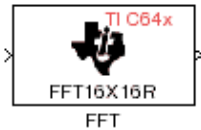
## Purpose

Compute the decimation-in-frequency forward FFT of a 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  $S = (k+1)/2$ .

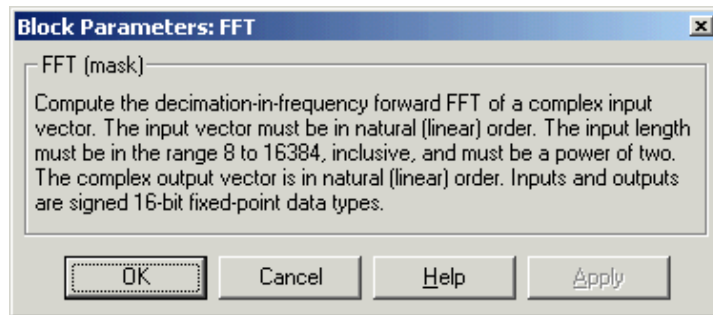
If  $k$  is even, then  $L$  is a power of two as well as a power of four, and this block performs all  $S$  stages with radix-4 butterflies to compute the output. If  $k$  is odd, then  $L$  is a power of two but not a power of four. In that case this block performs the first  $(S-1)$  stages with radix-4 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.

## Dialog Box



## Algorithm

In simulation, the FFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fft16x16r`. During code generation, this block calls the `DSP_fft16x16r` routine to produce optimized code.

## See Also

C64x Radix-2 FFT, C64x Radix-2 IFFT

# C64x General Real FIR

**Purpose** Filter a real input signal using a real FIR filter

**Library** C64x DSP Library—Filtering

## Description

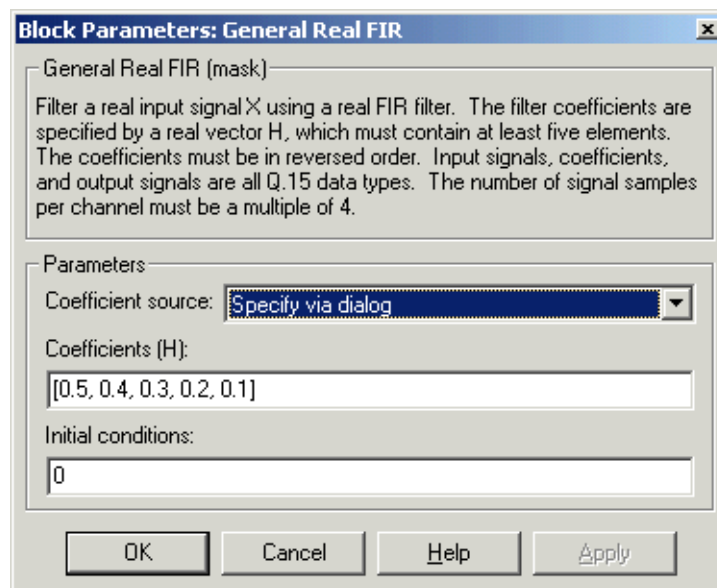


The C64x General Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure. Signal  $X$  must contain at least four samples per channel and the number of samples must be an integer multiple of four.

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
- **Input port**—Accept the coefficients from port H. This port must have the same rate as the input data port X

## **Coefficients (H)**

Designate the filter coefficients in vector format. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

## **Initial conditions**

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

The initial conditions must be real.

## **Algorithm**

In simulation, the General Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_gen`. During code generation, this block calls the `DSP_fir_gen` routine to produce optimized code.

## **See Also**

C64x Complex FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

# C64x LMS Adaptive FIR

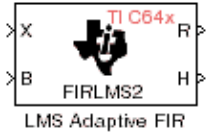
## Purpose

Filter a scalar input using least-mean-square adaptive 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

$$B = \mu e(n+1)$$

$$R = \bar{H}(n) \cdot \bar{X}(n+1)$$

which combined with the first two equations, result in the following equations that this block follows:

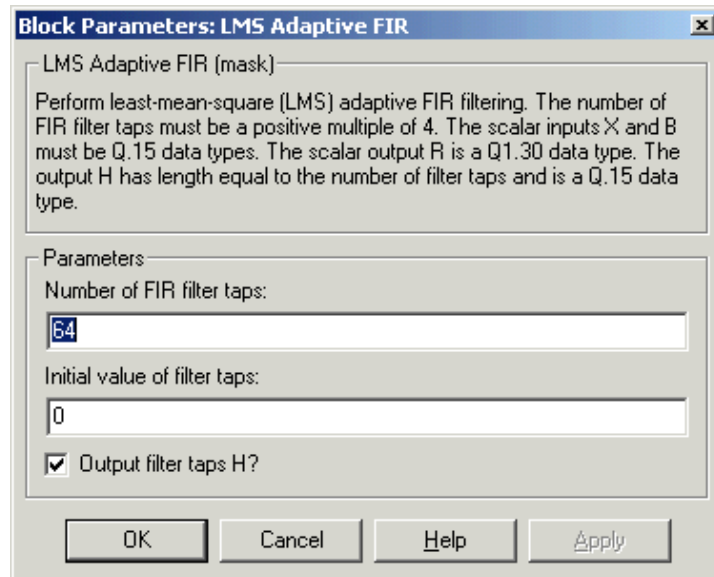
$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)]$$

$d$  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?

# C64x LMS Adaptive FIR

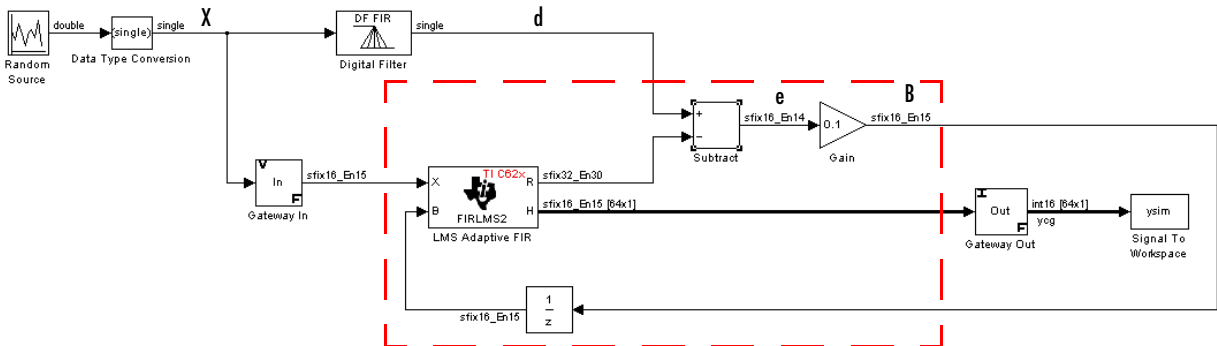
If selected, the filter taps are produced as output  $H$ . If not selected,  $H$  is suppressed.

## 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 `adapt1ms` function in the Filter Design Toolbox. `adapt1ms` performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

To use the LMS Adaptive FIR block you must create the input  $B$  in some way similar to the one shown here. You must also provide the signals  $\bar{X}$  and  $d$ . This model simulates the desired signal  $d$  by feeding  $\bar{X}$  into a digital filter block. You can simulate your desired signal in a similar way, or you may bring  $d$  in from the workspace with a From Workspace or codec block.



**Purpose** Perform matrix multiplication on 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.

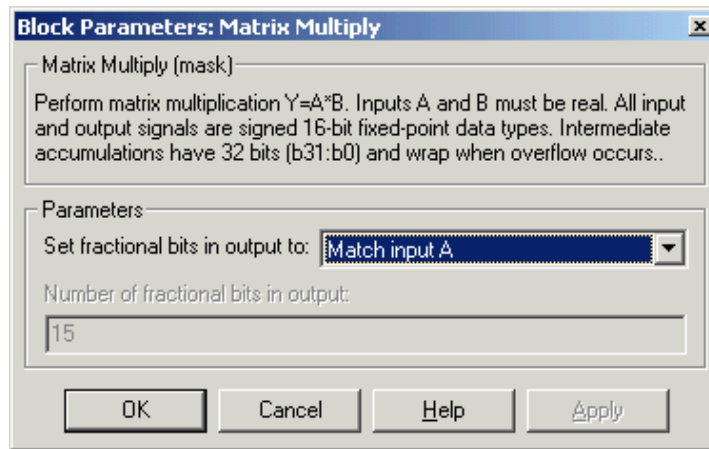
	<b>Input A</b>	<b>Input B</b>	<b>Accumulator Value</b>
<b>Total Bits</b>	16	16	32
<b>Fractional Bits</b>	<i>R</i>	<i>S</i>	<i>R + S</i>

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.

# C64x Matrix Multiply

## Dialog Box



### Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Choose which 16 bits to output from the list:

- Match input A—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or *R* in the discussion above).
- Match input B—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or *S* in the discussion above).
- Match high bits of acc. (b31:b16)—Output the highest 16 bits of the accumulator value.
- Match high bits of prod. (b30:b15)—Output the second-highest 16 bits of the accumulator value.
- User-defined—Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

### Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is enabled only when you select User-defined for **Set fractional bits in output to**.

## Algorithm

In simulation, the Matrix Multiply block is equivalent to the 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.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q.15	b30:b15
Match input B	Q.15	b30:b15
Match high bits of acc.	Q1.14	b31:b16
Match high bits of prod.	Q.15	b30:b15

**Example 2** Suppose A is Q12.3 and B is Q10.5. The data type of the resulting accumulator value is therefore Q23.8 ( $R + S = 8$ ). In the accumulator, bits 31:8 are the sign and integer bits, and bits 7:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q12.3	b20:b5
Match input B	Q10.5	b18:b3
Match high bits of acc.	Q23.-8	b31:b16
Match high bits of prod.	Q22.-7	b30:b15

## See Also

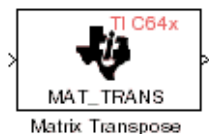
C64x Vector Multiply

# C64x Matrix Transpose

**Purpose** Compute the matrix transpose of an input signal

**Library** C64x DSP Library—Math and Matrices

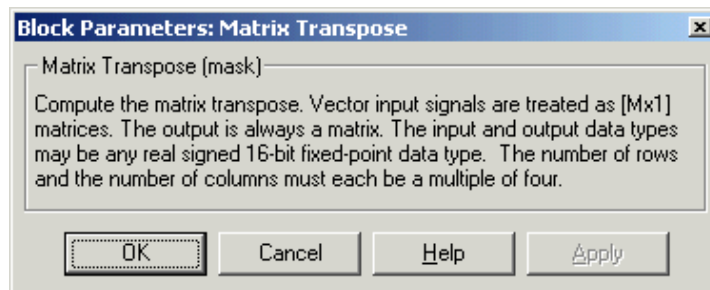
## Description



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.

**Purpose** Compute the radix-2 decimation-in-frequency forward FFT of a complex input vector

**Library** C64x DSP Library—Transforms

## Description

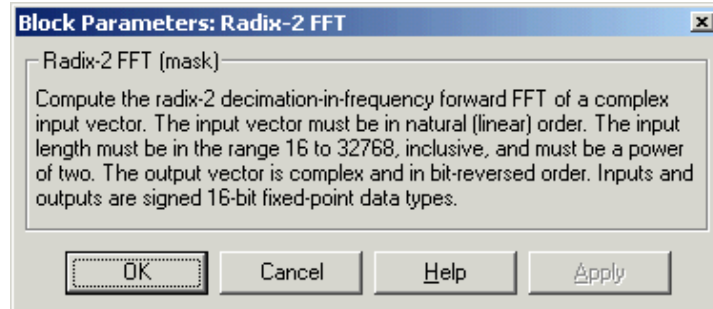


The C64x Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block 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 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



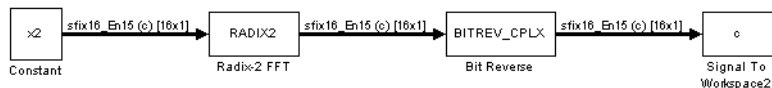
## Algorithm

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.

# C64x Radix-2 FFT

## Examples

The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the 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

**Purpose** Compute the radix-2 inverse FFT of a 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.

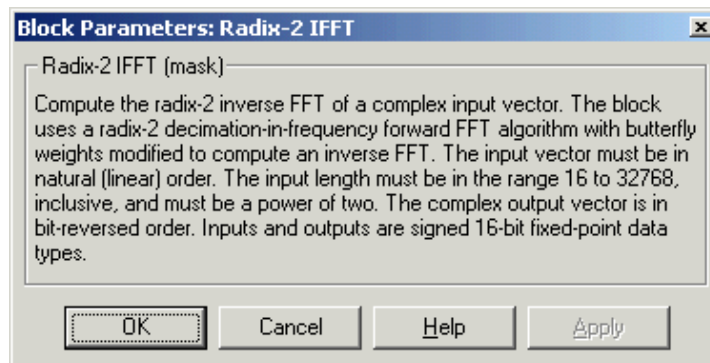
The radix2 routine used by this block employs a radix-2 FFT of length  $L=2^k$ . In order to ensure that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by  $k$  bits to the left relative to the location 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  $k$ .

$$OutputFractionalBits = InputFractionalBits + (k)$$

You can use the 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**



# C64x Radix-2 IFFT

---

## **Algorithm**

In simulation, the Radix-2 IFFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

## **See Also**

C64x Bit Reverse, C64x FFT, C64x Radix-2 FFT



**Purpose** Filter a real input signal using a real FIR filter

**Library** C64x DSP Library—Filtering

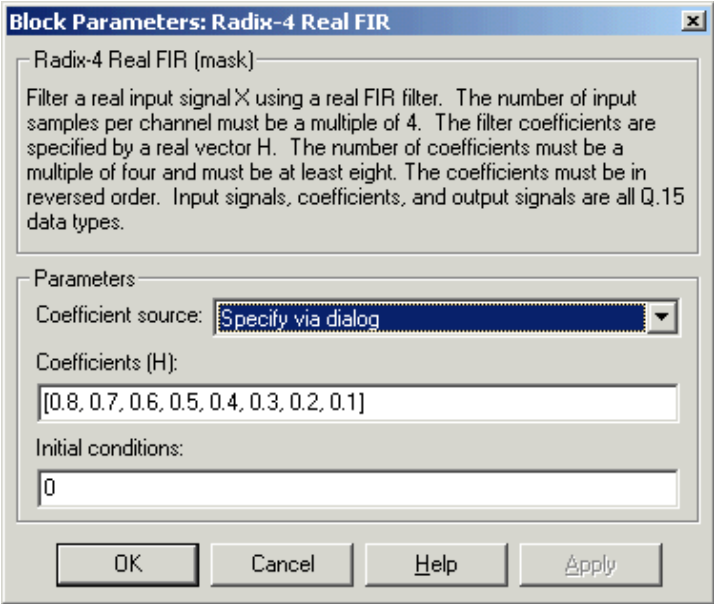
**Description** The C64x Radix-4 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.



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:

# C64x Radix-4 Real FIR

---

- 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. Enter the  $n$  coefficients in reversed order— $b(n), b(n-1), \dots, b(0)$ . This parameter is tunable in simulation.

## Initial conditions

If the initial conditions are

- All the same, enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

## Algorithm

In simulation, the Radix-4 Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_r4`. During code generation, this block calls the `DSP_fir_r4` routine to produce optimized code.

## See Also

C64x Complex FIR, C64x General Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

**Purpose** Filter a real input signal using a real FIR filter

**Library** C64x DSP Library—Filtering

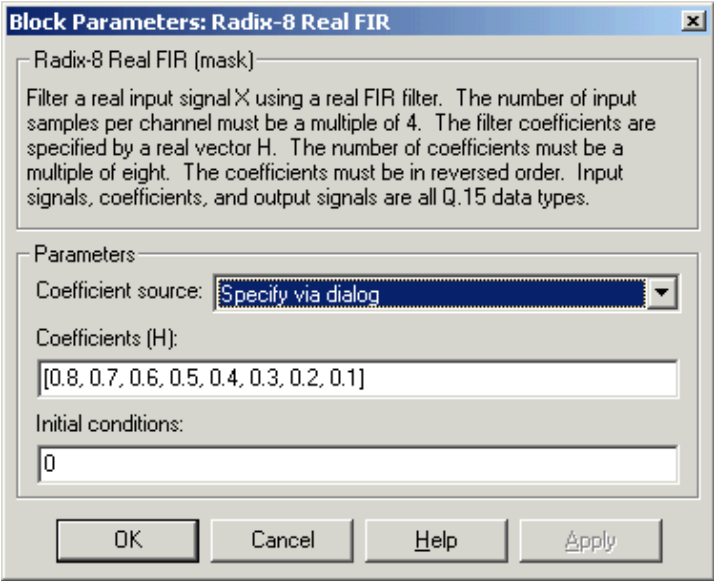
**Description** The C64x Radix-8 Real FIR block filters a real input signal  $X$  using a real FIR filter. This filter is implemented using a direct form structure.



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

# C64x Radix-8 Real FIR

---

- Input port—Accept the coefficients from port H. This port must have the same rate as the input data port X

## Coefficients (H)

Designate the filter coefficients in vector format, entering them in reversed order— $b(n)$ ,  $b(n-1)$ , ...,  $b(0)$ . 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 a real input signal using an autoregressive forward lattice filter

**Library** C64x DSP Library—Filtering

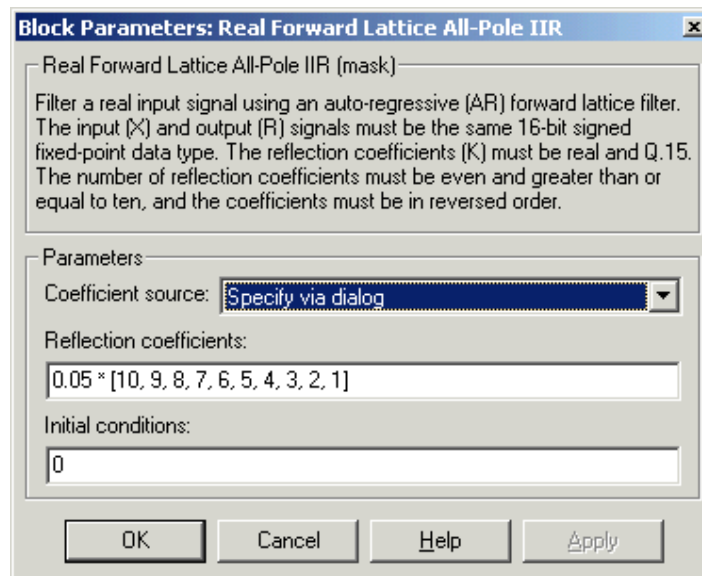
## Description



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:

# C64x Real Forward Lattice All-Pole IIR

---

- 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 ten and be even. Enter the coefficients in reverse order from  $k(n)$  to  $k(0)$ . Using an even number of reflection coefficients maximizes the speed of your generated code. This parameter is visible when you select Specify via dialog for the **Coefficient source** parameter. This parameter is tunable in simulation.

## Initial conditions

If your block initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length (number of elements) of this vector must be the same as the number of reflection coefficients in your filter.
- Different across channels, enter a matrix containing all initial conditions. The number of rows (initial conditions for one channel) of this matrix must be the same as the number of reflection coefficients, and the number of columns of this matrix must be equal to the number of channels.

## Algorithm

In simulation, the Real Forward Lattice All-Pole IIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_iirlat`. During code generation, this block calls the `DSP_iirlat` routine to produce optimized code.

## See Also

C64x Real IIR

**Purpose** Filter a real input signal using a real autoregressive moving-average IIR filter

**Library** C64x DSP Library—Filtering

## Description

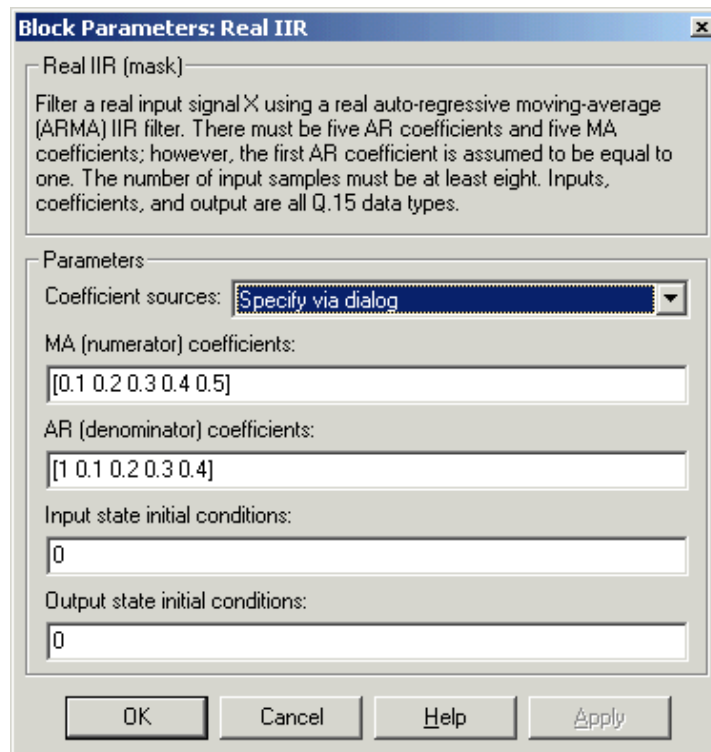


The C64x Real IIR block filters a real input signal  $X$  using a real autoregressive moving-average (ARMA) IIR Filter. This filter is implemented using a direct form I structure. You must use at least eight input samples.

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
- **Input ports**—Accept the coefficients from ports MA and AR

## **MA (numerator) coefficients**

Designate the moving-average coefficients of the filter in vector format. There must be five MA coefficients. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

## **AR (denominator) coefficients**

Designate the autoregressive coefficients of the filter in vector format. There must be five AR coefficients, however the first AR coefficient is assumed to be equal to one. This parameter is only visible when **Specify via dialog** is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

## **Input state initial conditions**

If the input state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the input state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all input state initial conditions. This matrix must have four rows.

## **Output state initial conditions**

If the output state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the output state initial conditions for one channel. The length of this vector must be four.



- Different across channels, enter a matrix containing all output state initial conditions. This matrix must have four rows.

**Algorithm**

In simulation, the Real IIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_iir`. During code generation, this block calls the `DSP_iir` routine to produce optimized code.

**See Also**

C64x Real Forward Lattice All-Pole IIR

# C64x Reciprocal

## Purpose

Compute the fractional and exponential portions of the reciprocal of a real input signal

## Library

C64x DSP Library—Math and Matrices

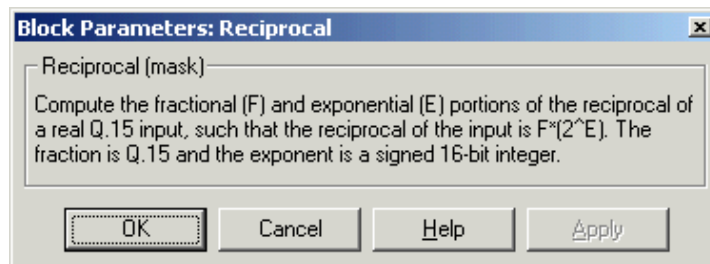
## Description



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.

**Purpose** Filter a real input signal using a symmetric real 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.

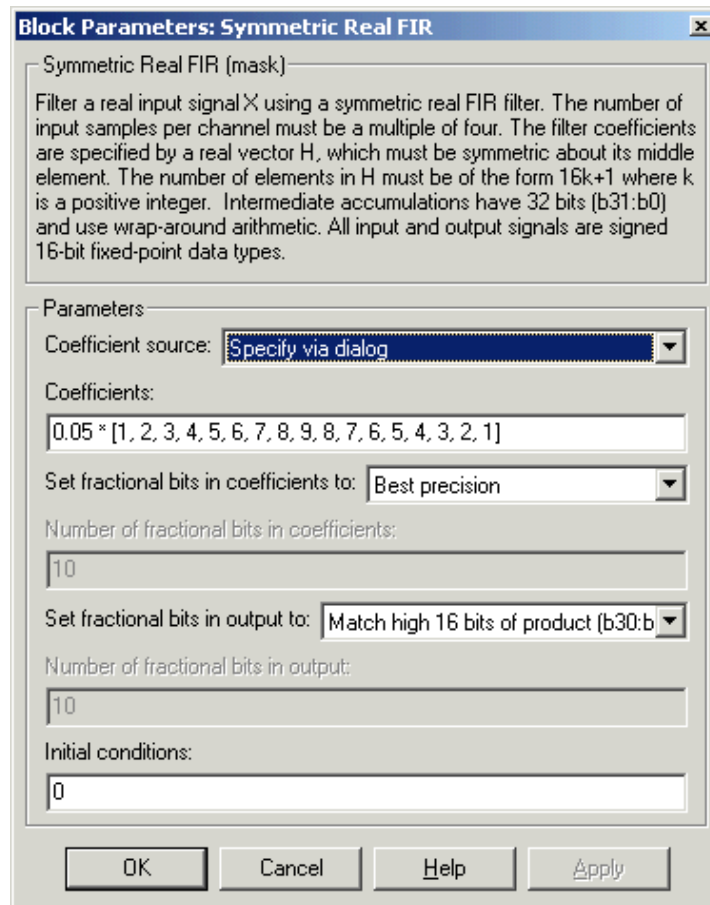
Intermediate multiplies and accumulates performed by this filter result in 32-bit accumulator values. However, the Symmetric Real FIR block only outputs 16 bits. You can choose to output 16 bits of the accumulator value in one of the following ways.

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.

# C64x Symmetric Real FIR

## 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

## 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

# C64x Symmetric Real FIR

---

- 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 6-89 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

**Purpose** Compute the vector dot product of two real input signals

**Library** C64x DSP Library—Math and Matrices

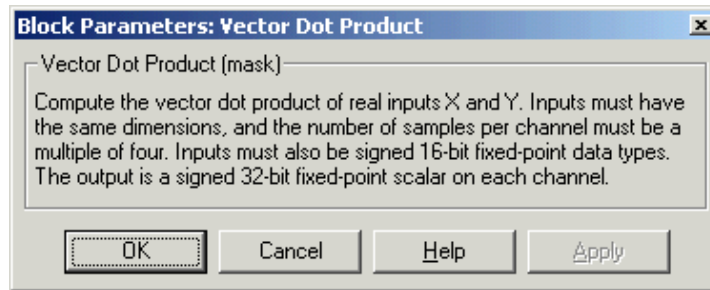
## Description



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.

# C64x Vector Maximum Index

**Purpose** Compute the index of the maximum value element in each channel of an input signal

**Library** C64x DSP Library—Math and Matrices

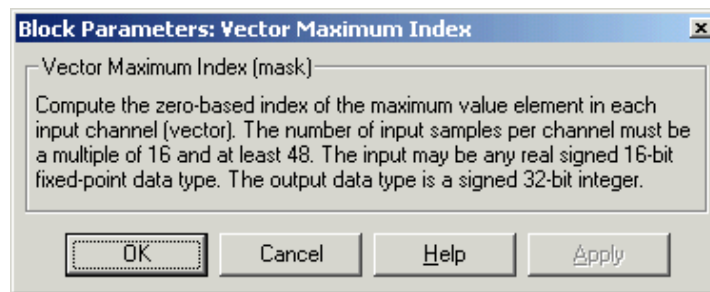
## Description



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** Compute the maximum value for each channel of an input signal

**Library** C64x DSP Library—Math and Matrices

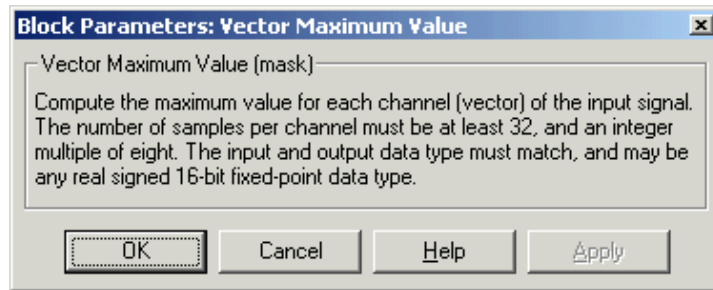
## Description



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** Compute the minimum value for each channel of an input signal

**Library** C64x DSP Library—Math and Matrices

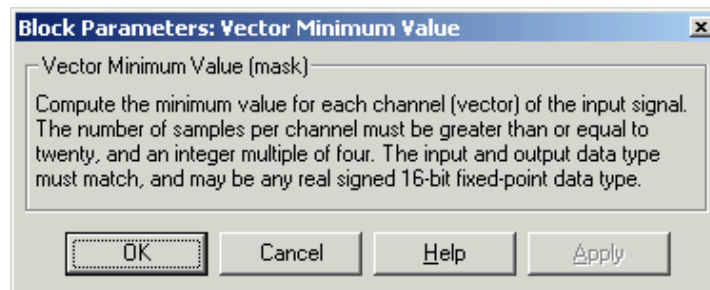
## Description



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.

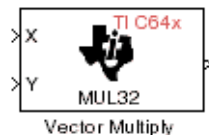
## See Also

C64x Vector Maximum Value

**Purpose** Perform element-wise multiplication on two inputs

**Library** C64x DSP Library—Math and Matrices

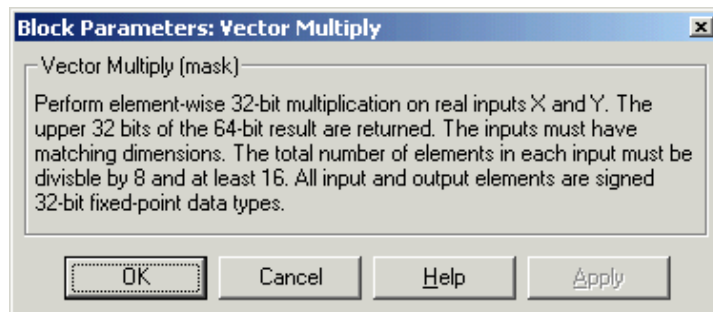
## Description



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

# C64x Vector Negate

**Purpose** Negate each element of an input signal

**Library** C64x DSP Library—Math and Matrices

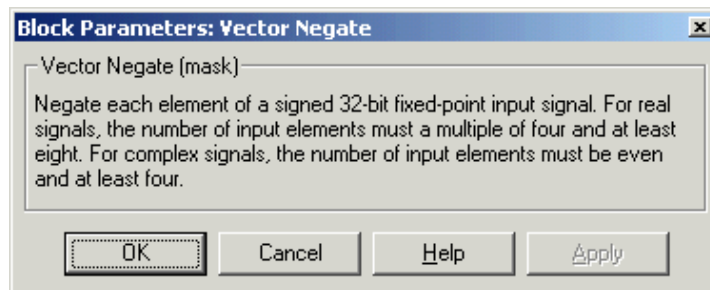
## Description



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



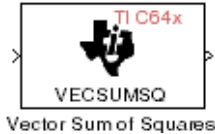
## Algorithm

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.

**Purpose** Compute the sum of squares over each channel of a real input

**Library** C64x DSP Library—Math and Matrices

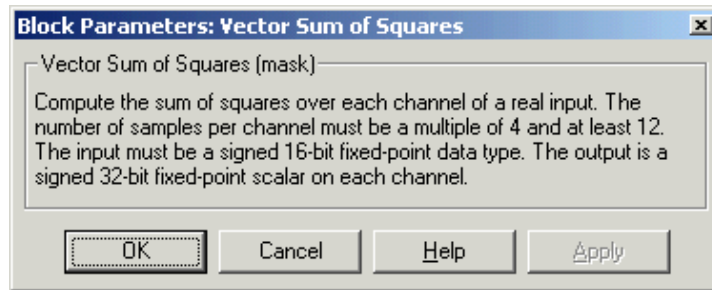
## Description



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.

# C64x Weighted Vector Sum

**Purpose** Find the weighted sum of two input vectors

**Library** C64x DSP Library—Math and Matrices

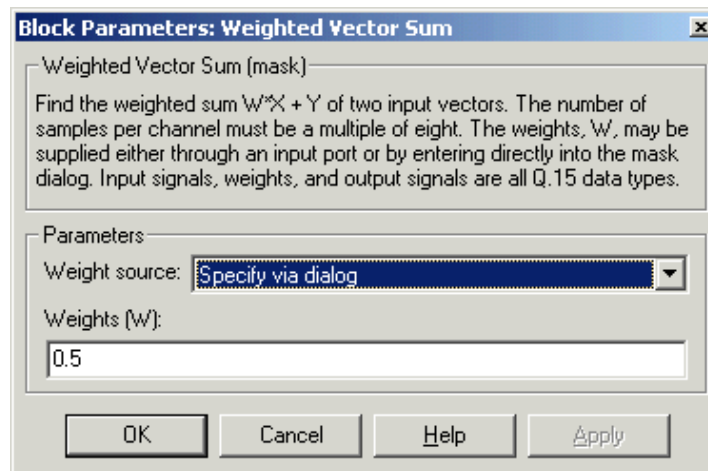
## Description



The C64x Weighted Vector Sum block computes the weighted sum of two inputs, X and Y, according to  $(W * X) + Y$ . 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
- Input port—Accept the weights from port W

## Weights (W)

This parameter is visible only when `Specify via dialog` is specified for the **Weight source** parameter. This parameter is tunable in simulation. When the weights are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be a multiple of four.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be a multiple of four, and the number of columns of this matrix must be equal to the number of channels.

Weights must be in the range  $-1 < W < 1$ .

## Algorithm

In simulation, the Weighted Vector Sum block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

# C6701 EVM ADC

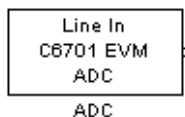
## Purpose

Configure digitized signal output from the codec to the processor

## Library

C6701 EVM Board Support in Embedded Target for TI C6000 DSP for TI DSP

## Description



Use the C6701 EVM 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 C6701 EVM C6701 EVM ADC block in your Simulink block diagram lets you use the multimedia audio coder-decoder module (codec) on the C6701 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 C6701 EVM hardware affected.

Option	Affected Hardware
ADC Source	Codec
Codec Data format	Codec
Mic	Codec
Output data type	TMS320C6701 digital signal processor
Sample rate (Hz)	Codec
Samples per frame	Direct memory access functions
Scaling	TMS320C6701 digital signal processor
Source gain (dB)	Codec
Stereo	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. Audio Word Byte Order for Mono and Stereo Inputs, shows how the codec stores monaural and stereo digitized signals in 32-bit words on the C6701 EVM. In the table, L means left channel, R means right channel, and O means that the 4-byte nibble does not contain data.

**Table 6-1: Audio Word Byte Order for Mono and Stereo Inputs**

<b>Format</b>	<b>Left and Mono Channel</b> (first 16 bits of data word)	<b>Right and Stereo Channel</b> (last 16 bits of data word)
16-bit mono	0xLLLL	0x0000
16-bit stereo	0xLLLL	0xRRRR
8-bit mono	0xLL00	0x0000
8-bit stereo	0xLL00	0xRR00
4-bit mono	0xL000	0x0000
4-bit stereo	0xL000	0xR000

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 C6701 EVM to capture the output from the codec as the input to the C6701 EVM ADC. When you select **Loopback**, your model must include both the C6701 EVM ADC and C6701 EVM DAC blocks.

# C6701 EVM ADC

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 5500 Hz to 48000 Hz, in preset rates. You must select from the list; you cannot enter a sample rate that is not on the list.

**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.

To enable the block and codec to generate data that your Simulink model can use, select the digitized data format. Three parameters—**Codec data format**, **Codec data type** and **Scaling**—control the format and range of the digital data generated by the block. Entries in Table 6-2 define the output ranges based on your selections for the **Data type**, **Scaling**, and **Codec data format** parameters in the **Block Parameters** dialog.

**Table 6-2: Data Type and Codec Data Format Parameters Choices Determine the Range**

		Data Type Parameter		
		Integer	Single- or Double Precision, Normalized	Single- or Double Precision, Floating-Point Integer
Codec Data Format	8-bit Unsigned	0-255	-1.0 to 1.0	0.0 to 255.0
	16-bit Linear	-32768 to 32767	-1.0 to 1.0	-32768.0 to 32767.0
	A-law	Unsigned 8-bit (0-255)	-1.0 to 1.0	0.0 to 255.0
	$\mu$ -law	Unsigned 8-bit (0-255)	-1.0 to 1.0	0.0 to 255.0
	ADPCM	Unsigned 8-bit (0-255)	N/A	0.0 to 255.0

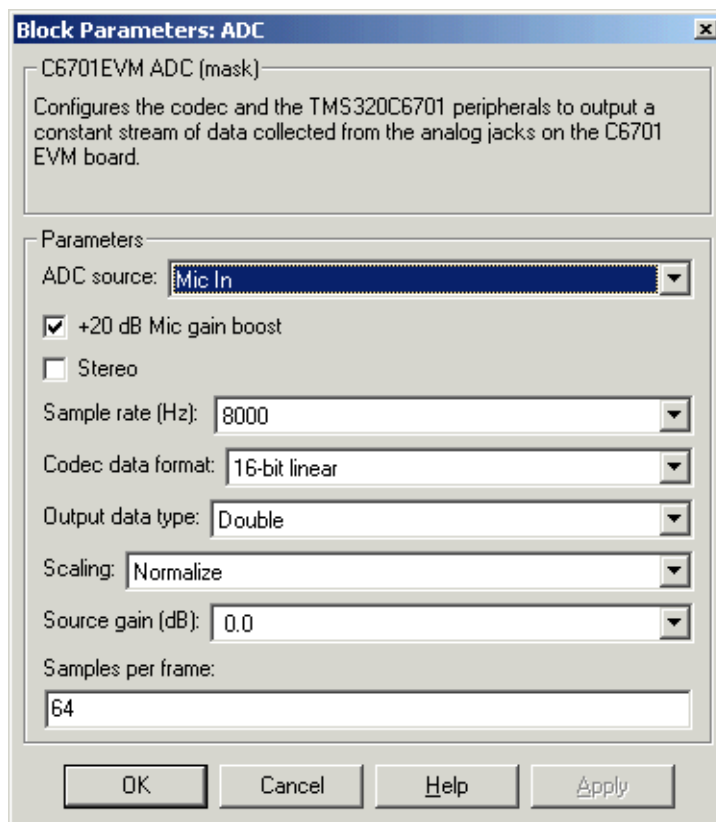
For example, when you select 16-bit linear data format, normalized scaling, and the Double data type, the C6701 EVM ADC block outputs a digitized signal composed of 16 bit samples ranging linearly from -1.0 to about 1.0.

Tables 4-2 and 4-3 list the five codec data formats you can select for the block. For reference purposes, the data types are described briefly in the following list:

- 8-bit unsigned—linear encoding that uses 8-bit words and constant steps between adjacent quantization levels. Compare to A-law or ADPCM encoding.
- 16-Bit Linear—linear encoding that uses 16-bit words and constant steps between adjacent quantization levels. Compare to A-law or ADPCM encoding.
- A-law—a variation on the basic pulse code modulation (PCM) encoding method. The quantization levels are distributed according to the logarithmic A-law. It has linear characteristics near zero and logarithmic character for higher amplitudes. Used in Europe as the telephony standard.
- $\mu$ -law—the American/Japanese equivalent of the European standard A-law encoding. For details, refer to the Consultative Committee for International Telegraphy and Telephony (CCITT) G.711 specification.
- IMA ADPCM—a modified differential pulse code modulation (PCM) encoding scheme. The step size for the difference quantization is adapted to the momentary rate of change of the input signal. For details, refer to the specification from the Interactive Multimedia Association (IMA) for their ADPCM implementation.

# C6701 EVM ADC

## Dialog Box



### ADC source

The input source to the codec. Line In is the default.

### Stereo

The number of channels input to the A/D converter. Clearing this option selects the left channel; selecting this option selects both left and right input channels. To configure the C6701 EVM board for monaural operation, clear the **Stereo** check box. When you first open the dialog, **Stereo** is cleared. The default is monaural operation.

### +20 dB Mic gain boost

Boosts the input signal by +20dB when **ADC source** is Mic. Gain is applied before analog-to-digital conversion.

**Sample rate (Hz)**

Sampling rate of the A/D converter. Available sample rates are set by the two clocks in the codec. Default rate is 8000 Hz.

**Codec data format**

Configures the format for output from the codec. Used in combination with the **Scaling** and **Data type** parameters to define the digital data leaving the block.

Your C6701 EVM ADC block format must match the codec data format for the C6701 EVM DAC block, if you use one in your model. The default setting is 16-bit linear.

**Output data type**

Selects the word length and shape of the data from the codec. By default, double is selected.

**Scaling**

Selects whether the codec data is unmodified, or normalized to the output range to  $\pm 1.0$ , based on the codec data format. Normalize is the default setting.

**Source gain (dB)**

Specifies the amount to boost the input before conversion. Applied to 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 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.

**See Also**

C6701 EVM DAC

# C6701 EVM DAC

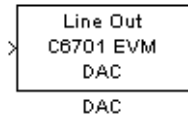
## Purpose

Use and configure the codec to convert digital input to analog output

## Library

C6701 EVM Board Support in Embedded Target for TI C6000 DSP

## Description



Adding the C6701 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 C6701 EVM mounting bracket. When you add the C6701 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 (D/A) conversion), the codec sends the signal to the output audio jack.

Two of the configuration options in 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 C6701 EVM hardware affected.

Option	Affected Hardware
Codec data format	Codec
DAC attenuation	Codec
Overflow mode	TMS320C6701 Digital Signal Processor
Scaling	TMS320C6701 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 94.5 dB in 1.5 dB increments. You must select from the list; you cannot enter a value for the attenuation.

For the block to accept data from your Simulink model, you must configure the data format. The parameters **Codec data format** and **Scaling** inform the block of the format of the digital data being received. Entries in Expected Data Range for Data Type and Codec Data Format Parameter Combinations, define the D/A input format based on the data type inherited from the preceding block and

your selection for the **Codec data format** parameter in the **Block Parameters** dialog box.

**Table 6-3: Expected Data Range for Data Type and Codec Data Format Parameter Combinations**

		Inherited Data Type Parameter		
		Integer	Single- or Double-Precision, Normalized	Single- or Double-Precision, Floating Point Integer
<b>Codec Data Format</b>	<b>8-bit Unsigned</b>	0-255	-1.0 to 1.0	0.0 to 255.0
	<b>16-bit Linear</b>	-32768 to 32767	-1.0 to 1.0	-32768.0 to 32767.0
	<b>A-law</b>	Unsigned 8-bit (0-255)	-1.0 to 1.0	0.0 to 255.0
	<b>μ-law</b>	Unsigned 8-bit (0-255)	-1.0 to 1.0	0.0 to 255.0
	<b>ADPCM</b>	Unsigned 8-bit (0-255)	N/A	0.0 to 255.0

For example, when you select 16-bit linear codec data format, with normalized scaling, and the block inherits the Double data type, the C6701 EVM DAC block expects to receive a digitized signal with each sample 16 bits long and ranging from -1.0 to 1.0. Signals that do not meet these criteria result in an error.

Tables 4-4 and 4-5 list the codec data formats you can select for the block. The following list provides brief descriptions of the available data formats:

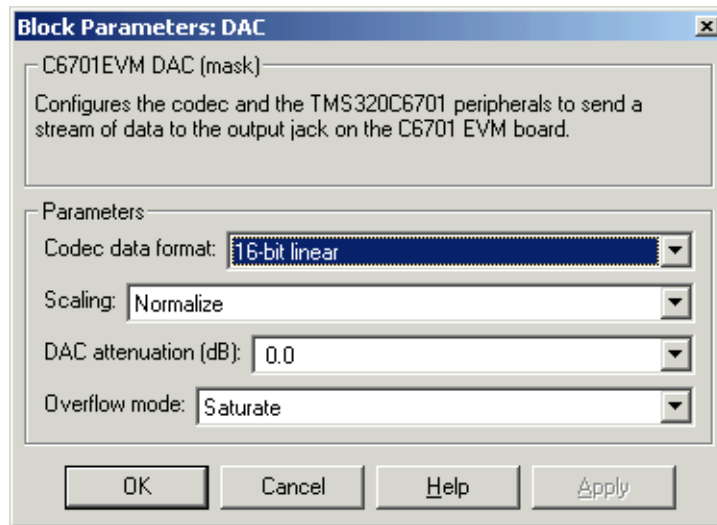
- 8-bit unsigned—linear encoding using 8-bit words and constant steps between adjacent quantization levels. Compare to A-law or ADPCM encoding.
- 16-Bit Linear—linear encoding using 16-bit words and constant steps between adjacent quantization levels. Compare to A-law or ADPCM encoding.
- A-law—a variation on the basic pulse code modulation (PCM) encoding method. The quantization levels are distributed according to the logarithmic A-law. It has linear characteristics near zero and logarithmic character for higher amplitudes. Used in Europe as the telephony standard.

- $\mu$ -law—the American/Japanese equivalent of the European standard A-law encoding. For details, refer to the Consultative Committee for International Telegraph and Telephony (CCITT) G.711 specification.
- IMA ADPCM—a modified differential pulse code modulation (PCM) encoding scheme. The step size for the difference quantization is adapted to the momentary rate of change of the input signal. For details, refer to the specification from the Interactive Multimedia Association (IMA) for their ADPCM implementation.

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. In addition, data that exceeds the range for a selected codec data format and data type is clipped or wrapped depending on the **Overflow mode** setting. Clipping is equivalent to saturating. To choose how the board handles data that falls outside the range that can be represented by the chosen data format, select an appropriate setting from **Overflow mode**. Saturate is the default setting. Selecting Saturate instructs the codec to clip output values to the maximum or minimum allowed value when output data exceeds the range of the data format. When you select Wrapping, the codec takes data that exceeds the acceptable output range and wraps the data back into the acceptable range using modular arithmetic relative to the smallest representable number. Selecting wrapping for the Overflow Mode can increase the performance of your application, but risks generating output values that exceed the codec data format limits and are wrapped back into the range of acceptable values.



## Dialog Box



### Codec data format

Tells the codec the format of data coming into it. Used in combination with the **Scaling** and **Data type** parameters to define the digital data entering the block. The block converts the input digital signal to an analog output signal based on how it interprets the input data stream. **Codec data format** tells the block how to interpret the input values.

The C6701 EVM DAC block **Codec data format** must match the **Codec data format** for the C6701 EVM ADC block in your model, if any.

### 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 C6701 EVM ADC block is usually appropriate here.

### Overflow mode

Determines how the codec responds to data that is outside the range specified by the **Codec data format** and **Scaling** parameters.

### DAC attenuation

Specifies the amount to attenuate the block output after D/A conversion.

# C6701 EVM DAC

---

## **See Also**

C6701 EVM ADC

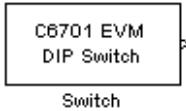
## Purpose

Simulate or read the user-defined DIP switches on the C6701 EVM

## Library

C6701 EVM 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 **USER0**, **USER1**, and **USER2** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6701 EVM. 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 simulated 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.

**Table 6-4: Option Settings to Simulate the User DIP Switches on the C6701 EVM**

<b>USER0 (LSB)</b>	<b>USER1</b>	<b>USER2 (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

# C6701 EVM DIP Switch

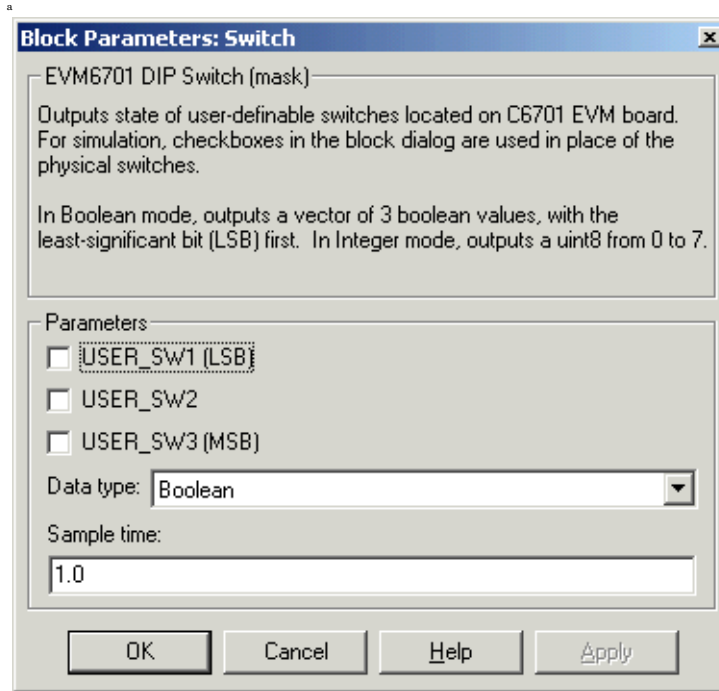
Selecting the **Integer** data type results in the switch settings generating an integer in the range from 0 to 7 (uint8), corresponding to converting the string of individual switch settings to a decimal value. In the **Boolean** data type, the output string presents the separate switch setting for each switch, with the status of **USER0** represented by the least significant bit (LSB) and the **USER2** status 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 Table 6-5. Your process uses the result in the same way whether simulating a process or generating code. In code generation and when running your application, the block code ignores the settings for **USER0**, **USER1**, and **USER2** in favor of the hardware switch settings. When the block reads the switch settings, it reports the status as shown in Output Values From the User DIP Switches on the C6701 EVM.

**Table 6-5: Output Values From the User DIP Switches on the C6701 EVM**

<b>USER0 (LSB)</b>	<b>USER1</b>	<b>USER2 (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



### **USER0**

Simulate the status of the user-defined DIP switch on the board.

### **USER1**

Simulate the status of the user-defined DIP switch on the board.

### **USER2**

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 **USER0** and the MSB is switch **USER2**. The other data type, **Integer**, converts the

# C6701 EVM DIP Switch

---

logical string to an equivalent unsigned 8-bit (uint8) 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**).

For further information about the user-defined DIP switches on the board, refer to your Texas Instruments *TMS320C6201/6701 Evaluation Module Technical Reference*.

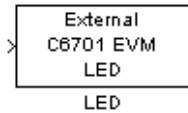
## Purpose

Control the light emitting diodes on the C6701 EVM

## Library

C6701 EVM Board Support in Embedded Target for TI C6000 DSP

## Description



Adding an C6701 EVM LED block to your Simulink block diagram lets you trigger one of the red light emitting diodes (LED) on the C6701 EVM. To use the block, select an LED from the LED list—internal (1) or external (0) and send a nonzero real scalar to the block. The C6701 EVM LED block triggers the external status LED (User Status LED0) located on the C6701 EVM mounting bracket when you select external. When you select internal, the C6701 EVM LED block triggers the internal status LED (User Status LED1) located at the top of the C6701 EVM board.

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 LED is turned off (disabled)
- When the block receives a nonzero input value, the specified LED is 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.

Both LEDs maintain their state until their controlling C6701 EVM LED blocks receive an input value that changes the state. An enabled LED stays on until its block receives an input value equal to zero and turns the LED off; a disabled LED stays off until turned on. Resetting the C6701 EVM turns both LEDs off.

---

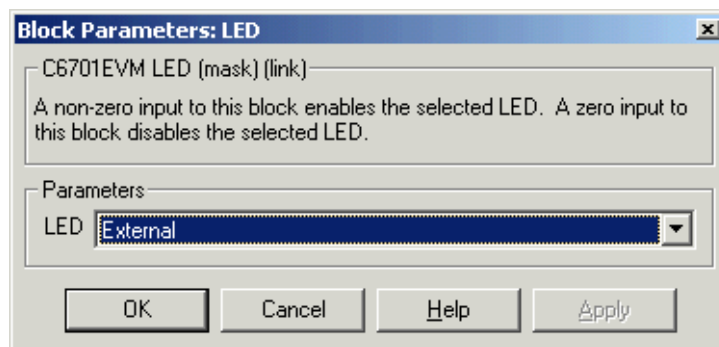
**Note** Target for C6701 EVM uses the external LED to signal overrun conditions during processing on the C6701 EVM, when you set the **Overrun** option on the Real-Time Workshop dialog to Halt or Continue. Using the external LED as a status indicator through a C6701 EVM LED block can conflict with overrun indications. When you are trying to determine why the external LED is on, recall this point.

---

# C6701 EVM LED

---

## Dialog Box



## LED

Selects which light emitting diode the block activates on the C6701 EVM. The default setting is external (0).

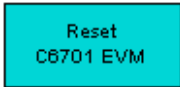


**Purpose**

Reset the C6701 Evaluation Module to initial conditions

**Library**

C6701 EVM Board Support in Embedded Target for TI C6000 DSP

**Description**

Double-clicking this block in a Simulink model window resets the C6701 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 C6701 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 C6701 EVM. In other words, anytime you double-click a C6701 EVM RESET block you reset your C6701 EVM.

**Dialog Box**

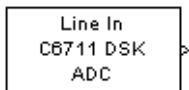
This block does not have settable options and does not provide a user interface dialog.

# C6711 DSK ADC

**Purpose** Configure digitized signal output from the codec to the 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.

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. Audio Word Byte Order for Mono and Stereo Inputs, shows how the codec stores monaural and stereo digitized signals in 32-bit words on the C6711 DSK. In the table, L means left channel, R means right channel, and O means that the 4-byte nibble does not contain data.

**Table 6-6: Audio Word Byte Order for Mono and Stereo Inputs**

<b>Format</b>	<b>Left and Mono Channel</b> (first 16 bits of data word)	<b>Right and Stereo Channel</b> (last 16 bits of data word)
16-bit mono	0xLLLL	0x0000
16-bit stereo	0xLLLL	0xRRRR
8-bit mono	0xLL00	0x0000
8-bit stereo	0xLL00	0xRR00
4-bit mono	0xL000	0x0000
4-bit stereo	0xL000	0xR000

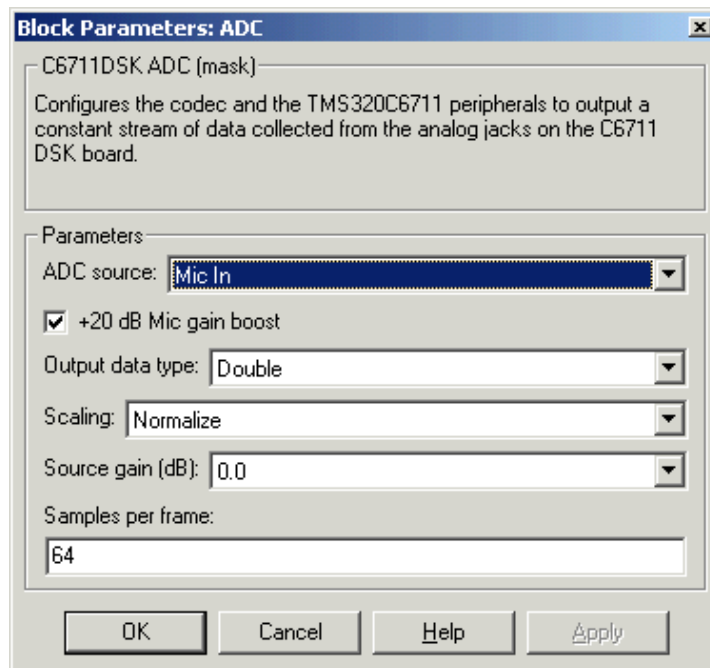
When you select **Mic** for **ADC source**, you can select the **+20 dB Mic gain boost** check box to add 20 dB to the microphone input signal before the codec digitizes the signal.

Selecting **Loopback** for **ADC source** configures the C6711 DSK to capture the output from the codec as the input to the C6711 EVM ADC. When you select **Loopback**, your model must include both the C6711 EVM ADC and C6711 EVM 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.

# C6711 DSK ADC

## 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.

### 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**

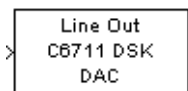
C6711 DSK DAC

# C6711 DSK DAC

**Purpose** Use and configure the codec to convert digital input to analog output

**Library** C6711 DSK Board Support in Embedded Target for TI C6000 DSP

## Description



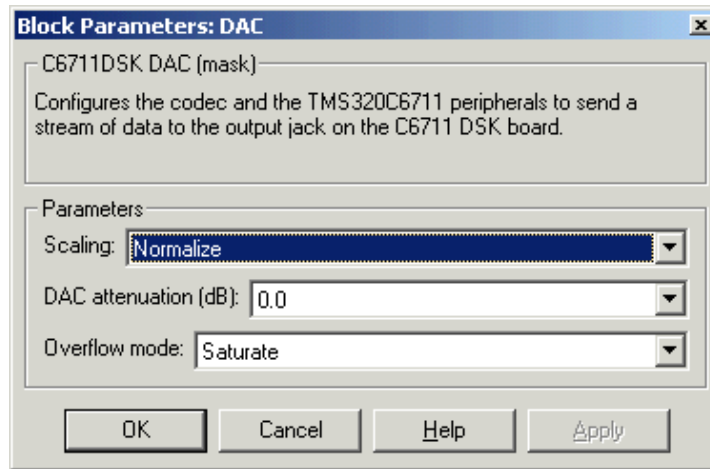
Adding the C6711 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the LINE OUT connection on the C6711 DSK mounting bracket. When you add the C6711 DSK DAC block, the digital signal received by the codec is converted to an analog signal. 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.

## 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

# C6711 DSK DIP Switch

## Purpose

Simulate or read the user-defined DIP switches on the C6711 DSK

## 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.

**Table 6-7: 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 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).

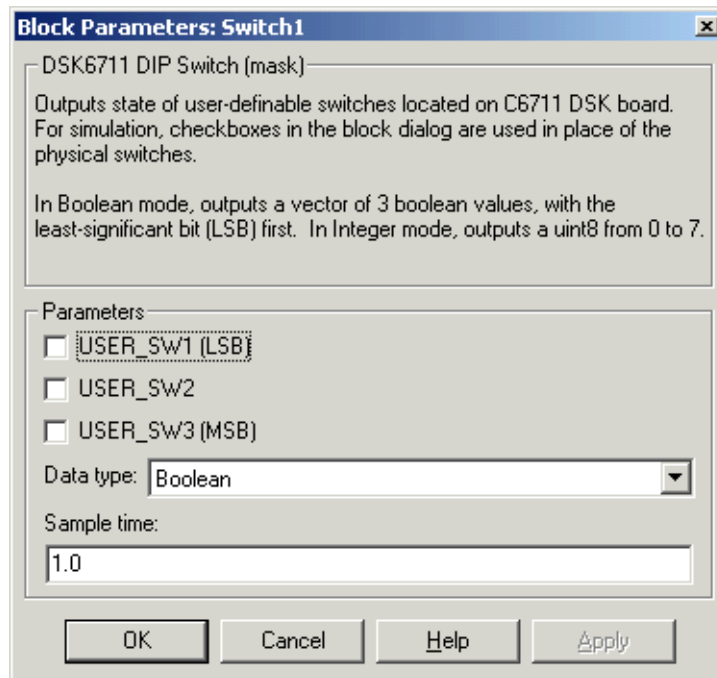
**Code generation and targeting**—the code generated by the block reads the physical switch settings of the user switches on the board and reports them as shown in Table 6-8. Your process uses the result in the same way whether in simulation or in code generation. In code generation and when running 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

**Table 6-8: 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

# C6711 DSK DIP Switch

## Dialog Box



### **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)

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**).

# C6711 DSK LED

---

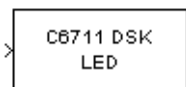
## Purpose

Control the user-defined light emitting diodes on the C6711 DSK

## 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.

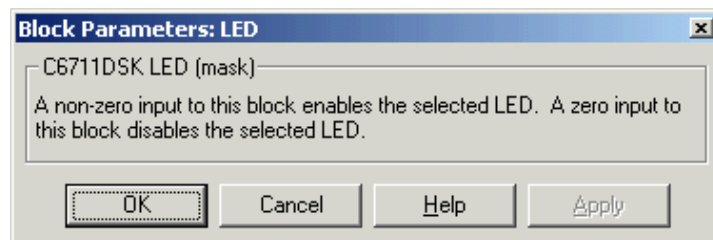
When you add this block to a model, and send a real scalar to the block input, the block sets the LED state based on the input value it receives:

- 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



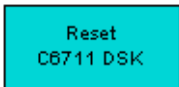
This dialog does not have any user-selectable options.

**Purpose**

Reset the C6711 DSK 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.

# C6713 DSK ADC

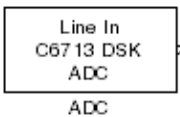
## Purpose

Configure digitized signal output from the codec to the 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).

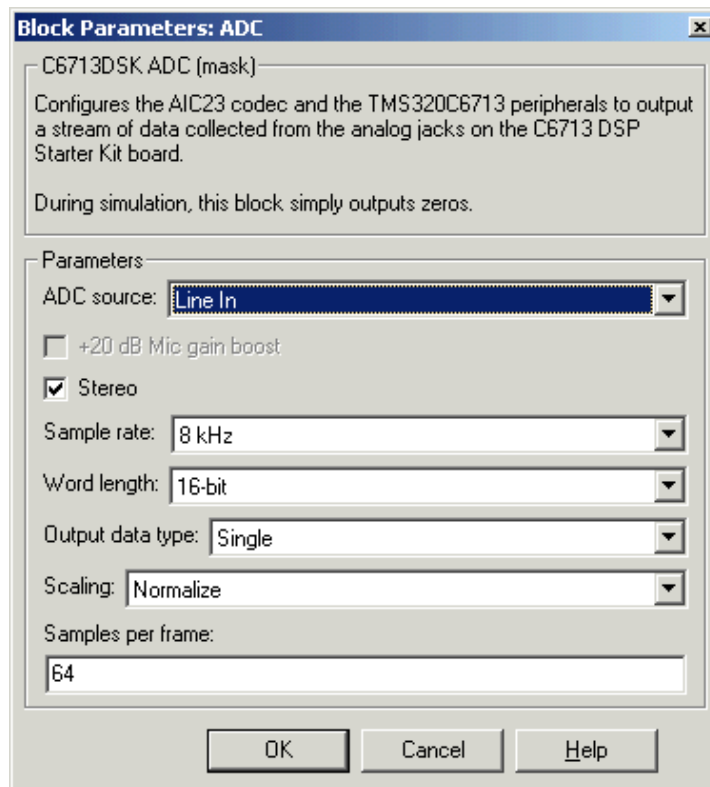
When the samples per frame setting is more than one, each frame of data is either the N-element vector (monaural input) or N-by-2 matrix (stereo input). For monaural input, the elements in each frame form the column vector of input audio data. In the stereo format, the frame is the matrix of audio data 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

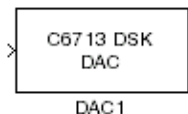
# C6713 DSK DAC

---

**Purpose** Configure the codec and peripherals to convert digital input to analog output at the analog output port of the board

**Library** C6713 DSK Board Support in Embedded Target for TI C6000 DSP

## Description

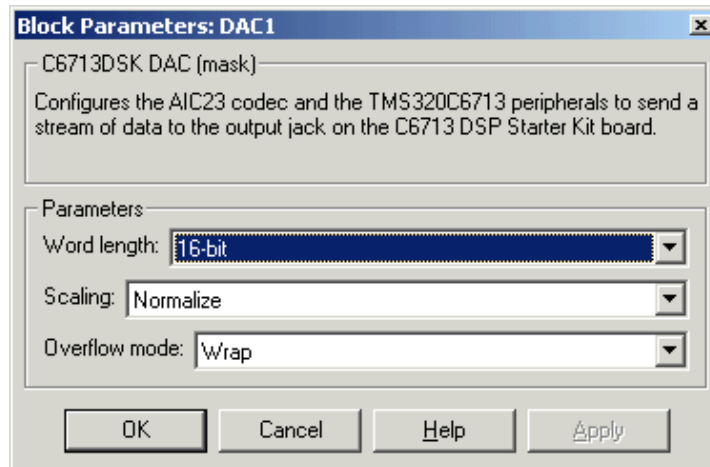


Adding the C6713 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the analog output jack on the C6713 DSK. When you add the C6713 DSK DAC block, the digital signal received by the codec is converted to an analog signal. After converting 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 C6416 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

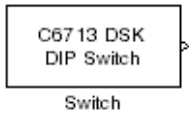
C6713 DSK ADC

# C6713 DSK DIP Switch

**Purpose** Simulate or read the user-defined DIP switches on the C6713 DSK

**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

## 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

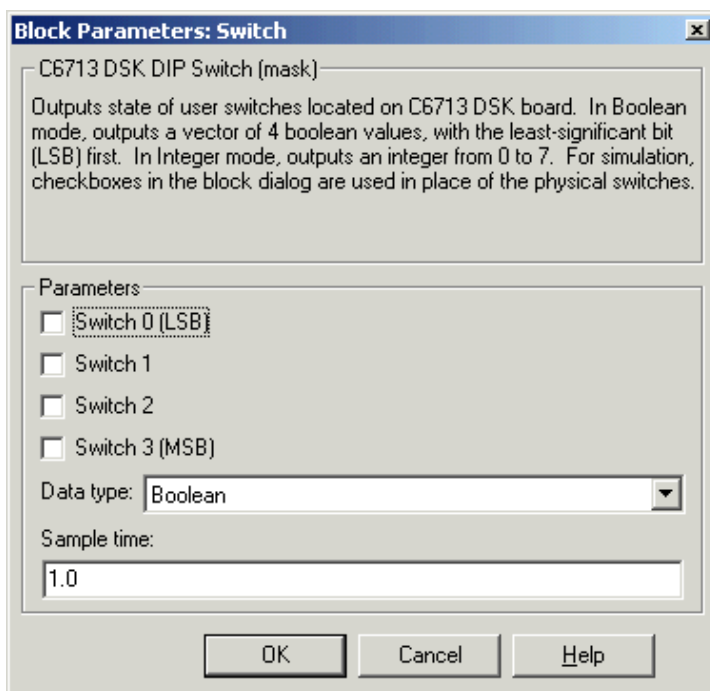
Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2

# 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



### 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**).



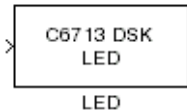
## Purpose

Control the user-defined light emitting diodes on the C6713 DSK

## 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.

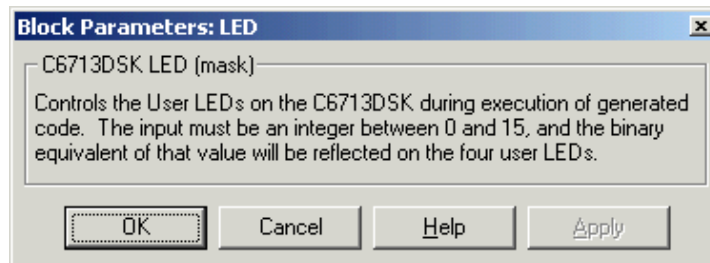
When you add this block to a model, and send a real scalar to the block input, the block sets the LED state based on the input value it receives:

- 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 does not have any user-selectable options.

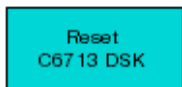
# C6713 DSK RESET

---

**Purpose** Reset the C6713 DSK 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.

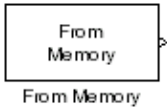
## Purpose

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



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.

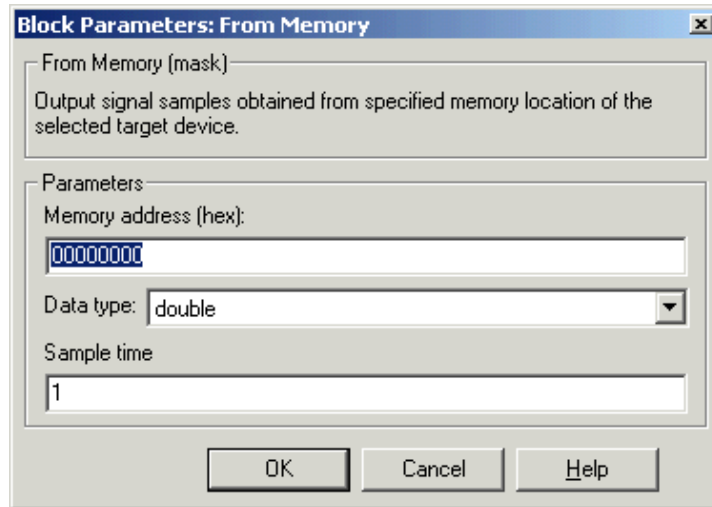
When you look at your generated code, you find lines of code like the following that represent the From Memory block operation:

```
/* 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.

# From Memory

## 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/Sample time**).

## See Also

To Memory

# From Rtdx

---

**Purpose** Add a named RTDX input channel to Simulink models

**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.

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

- 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 3-2—Using Links for RTDX” on page 3-39.

## 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

Enable blocking mode

Initial conditions:

Sample Time

Output dimensions

Frame-based

Data type:

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.

## **Blocking**

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 nonblocking mode, the processor uses old data from the block when new data is not available. Nonblocking 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 MATLAB 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`, `readmsg`, `To RTDX`, `writemsg`

# TMDX326040 ADC

---

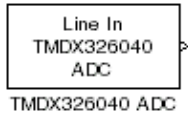
## Purpose

Configure the codec on the TMDX326040A daughter card to generate a stream of digital data to the processor on the 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:

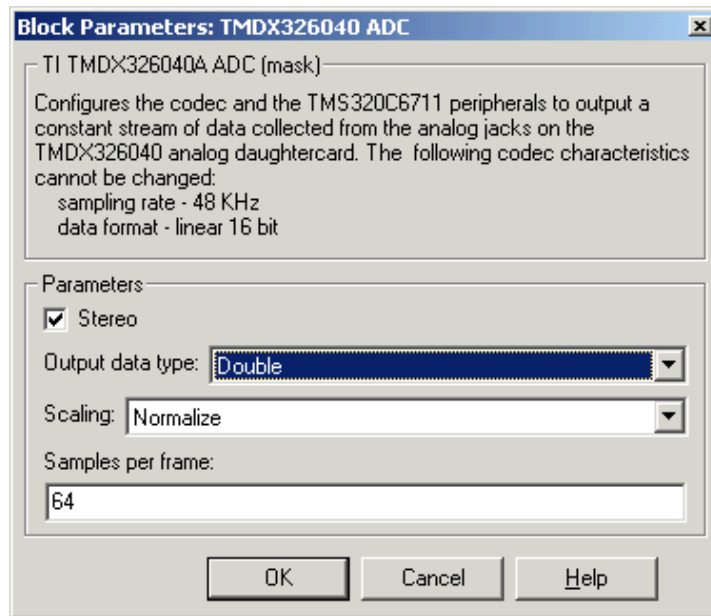
- Sampling rate is 48 kHz.
- Data format is linear 16-bit words.

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.

## 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.

# TMDX326040 ADC

---

## **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

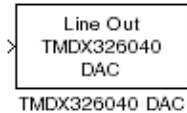
## Purpose

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

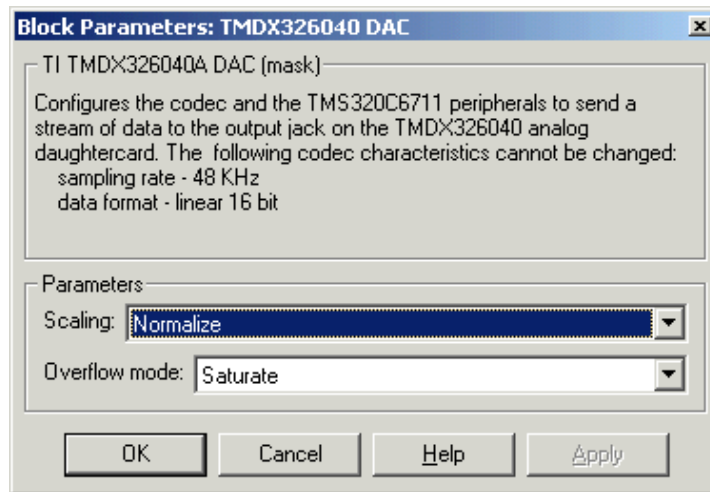
Analog data leaving the codec on the daughter card outputs on the card and then to the C6711 DSK output connectors. Whether the output is in monaural or stereo depends on the setting of the TMDX326040 ADC block in your model.

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.

# TMDX326040 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 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

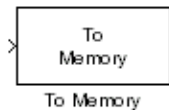
## Purpose

Send data from your model to memory on your C6000 target

## Library

C6000 DSP Core Support 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 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:

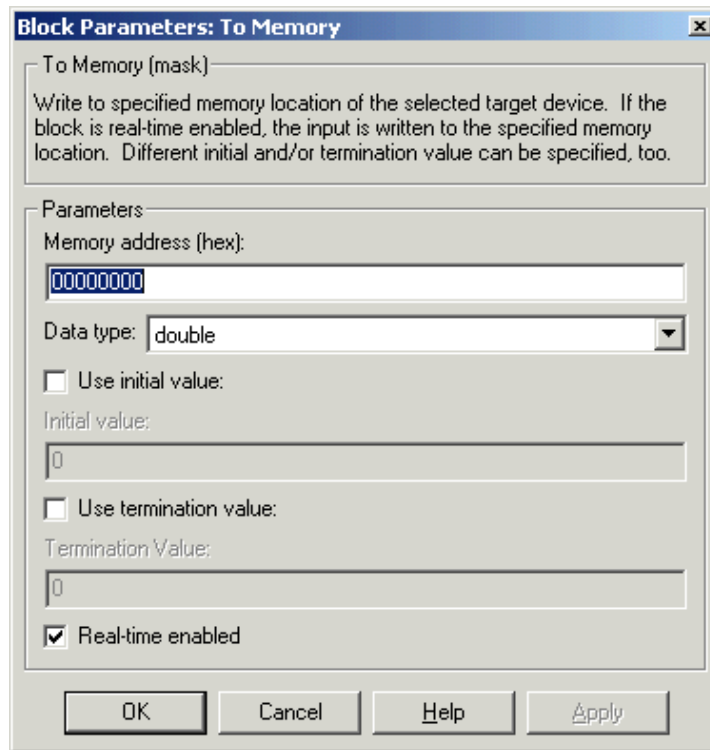
```
/* S-Function Block: <Root>/To Memory (c6000mem_snk) */
{
    /* Memory Mapped Output */
    *((volatile int *) (4096U)) = (real32_T) 8;
}
```

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.

# To Memory

## 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 gets written to memory)
- 3 Output (execute loop for each time step)
- 4 Terminate execution (your termination value gets written to memory)

**Real-time enabled** determines whether the code does anything during the output stage of executing generated code. 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**

## To Memory

---

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 checkbox.

Selecting **Real-time enabled** causes the code to write data to memory during the output phase of code execution.

### See Also

From Memory

**Purpose**

Add a named RTDX output channel to Simulink models

**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:

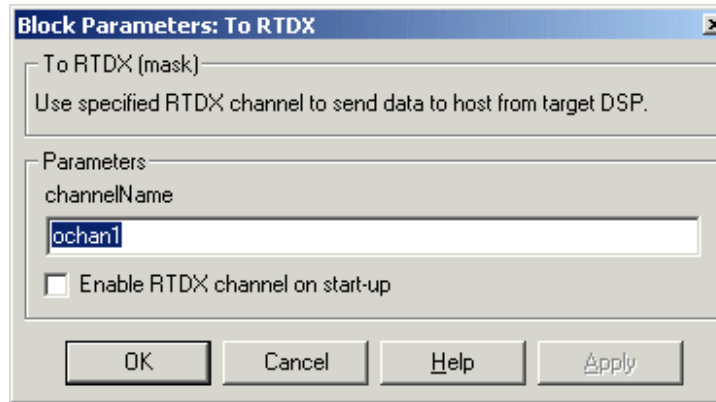
- Adding one or more RTDX blocks to your model to prepare your target
- Downloading and running your model on your target
- Enabling the RTDX channels from MATLAB
- Using the `readmsg` and `writemsg` functions in MATLAB to send and retrieve data from the target over RTDX

To see more details about using RTDX in your model, refer to “Tutorial 3-2—Using Links for RTDX” on page 3-39.

# To Rtdx

---

## 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 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 MATLAB 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

# Hardware Supported by the Embedded Target for TI C6000 DSP

---

Supported Hardware For Targetting  
(p. A-2)

Lists the hardware that the Embedded Target for TI C6000 DSP supports. Includes comments about supported operating systems where needed.

## Supported Hardware For Targetting

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
TMS320C6416 DSK	C6416 DSP Starter Kit. Does not work on Microsoft Windows™ NT platforms.
TMS320C6701 EVM	C6701 Evaluation Module.
TMS320C6711 DSK	C6711 DSP Starter Kit.
TMS320C6713 DSK	C6713 DSP Starter Kit. Does not work on Microsoft Windows™ NT platforms.
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.
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.

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.





**Numerics**

4-bit IMA ADPCM 2-16

**A**

adding DSP/BIOS to generated code 2-34

applications

    logging 2-24

Autocorrelation block 6-10, 6-71

automatic board selection 2-33

**B**

Bit Reverse block 6-14, 6-75

Block Exponent block 6-14, 6-75

block recommendations 2-55

blocks

    use in target models 2-55

Board and processor selection 2-33

build configuration

    default 2-46

    MW\_custom 2-46

build directory

    contents of 2-78, 2-107

    naming convention 2-67

building models

    use C62x DSP Library blocks 4-8

**C**

C6000 Target

    code generation options 2-34

    compiler options 2-37

    MATLAB to CCS link 2-34

    runtime options 2-42

    target selection 2-32

    targeting Code Composer Studio 2-109

    TI C6000 linker options 2-38

C62x DSP Library blocks

    building models 4-8

    choose blocks to optimize code 4-9

    common characteristics 4-3

    Q format notation 4-5

    use source and sink blocks 4-9

C6701 EVM

    confirming proper configuration 2-51

    general code generation options 2-31

    start/stop models 2-49

    target options 2-28

    TLC debugging options 2-30

    tutorial about multirate applications 2-66

C6701 EVM ADC block 6-118

C6701 EVM blocks

    tutorial 2-66

C6701 EVM directories

    build 2-67

    working 2-98

C6701EVM ADC

    choose the codec data format 2-16

    choose the sample rate 2-14

    select the data type 2-17

    select the scaling 2-18

    use fixed-point arithmetic 2-18

C6701EVM ADC block 2-13

C6701EVM ADC dialog 2-15

C6701EVM DAC block 2-19, 6-124

    choose the codec data format 2-20

    choose the scaling 2-20

    overflow mode 2-20

C6701EVM DAC dialog 2-19

C6701EVM DIP Switch block 6-129

C6701EVM LED block 6-133

    configure the block 2-21

- select the target LED 2-21
- the overrun indicator 2-22
- C6701EVM RESET block 6-135
- c6701evmtest.mdl 2-52
  - errors while running 2-84
  - use 2-52
  - verifying that the model is running 2-54
- C6711 DSK
  - configure 2-80
  - fixed sample rate 2-15
  - start/stop models 2-96
  - tutorial about multirate applications 2-97
- C6711 DSK blocks
  - tutorial 2-97
- C6711 DSK directories
  - build 2-98
  - working 2-67
- C6711DSK ADC block 6-58, 6-136, 6-148
- C6711DSK DAC block 6-62, 6-140, 6-152
- C6711DSK DIP Switch block 6-64, 6-142, 6-154
- C6711DSK LED block 6-69, 6-146, 6-159
- C6711DSK RESET block 6-70, 6-147, 6-160
- c6711dsktest.mdl 2-82
  - errors while running 2-54
  - verifying that the model is running 2-84
- calls
  - near and far 2-40
- CCS IDE
  - create projects for the IDE 2-109
- Code Composer Studio 2-109
- codec data format 2-16
- command line help 1-8
- Complex FIR block 6-15, 6-76
- configure the software timer 2-44
- configure your C6711 DSK for Embedded Target
  - for TI C6000 DSP 2-80
- confirm your C6701 EVM configuration 2-51

- convert data types 4-8
- Convert Floating-Point to Q.15 block 6-17, 6-78
- Convert Q.15 to Floating-Point block 6-18, 6-79
- current C6701 EVM CPU clock rate 2-44

## D

- data format, 16-bit linear 2-16
- data format, 8-bit A law 2-16
- data format, 8-bit mu law 2-16
- data format, codec 2-16
- data type, select 2-17
- default build configuration 2-46
- disabling logging 2-24
- discrete solver 2-26
- DSP/BIOS
  - added files 3-8
  - adding to generated code 2-34
  - files removed from project 3-9

## E

- Embedded Target for TI C6000 DSP viii
  - about viii, 1-2
  - configure C6701EVM ADC blocks 2-13
  - configure C6701EVM LED blocks 2-21
  - configure C6711DSK LED blocks 2-21
  - configure the C6701EVM DAC block 2-18
  - create Simulink model for targeting 2-55
  - errors while running test model c6701evmtest 2-54
  - errors while running test model c6711dsktest 2-84
  - expected background for use x
  - general help 1-8
  - hardware and OS requirements 1-4
  - information for new users x

- peripheral hardware for testing C6701 EVM operation 2-51
- peripheral hardware for testing C6711 DSK operation 2-81
- procedure for testing the operation 2-52
- requirements for TI software 1-5
- requirements for use ix
- select the target LED 2-21
- starting/stopping test model c6701evmtest 2-54
- starting/stopping test model c6711dsktest 2-85
- suitable applications 1-3
- test installation and operation of the C6701 EVM 2-51
- test installation and operation of the C6711 DSK 2-81
- use C6701 EVM blocks 2-8

error select board automatically 2-33

errors running c6701evmtest.md1 2-54

errors running c6711dsktest.md1 2-84

export filters to CCS IDE from FDATool 5-2

- select the export data type 5-7
- set the Export mode option 5-4
- set the Target selection options 5-12
- set Variable names in C header file 5-5
- set Variable names in target symbol table 5-5

exporting filters to CCS IDE from FDATool tutorial 5-9

external LED 2-21

**F**

far calls 2-40

FDATool

- See export filters to CCS IDE from FDATool

FFT block 6-19, 6-80

- 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 2-26
- From Rtdx block 6-164

## G

- General Real FIR block 6-21, 6-82
- generate optimized code 2-34
- generate\_code\_only option 2-42

## H

- halting a running process 2-54
- hardware requirements for Embedded Target for TI C6000 DSP 1-4
- headroom meter 2-22
- help
  - command line 1-8

## I

- Incorporate DSP/BIOS option 2-34
- indicator, overrun 2-22
- inline DSP Blockset functions option 2-34
- internal LED 2-21

## L

- LED block 2-21
- LED target 2-21
- linker error 2-40
- LMS Adaptive Filter block 6-24, 6-84
- logging
  - about 2-24

logging in models 2-24  
logging options in Simulink Parameters dialog  
2-24  
logging, disabling 2-24

## M

manual board selection 2-33  
Matrix Multiplication block 6-27, 6-87  
Matrix Transpose block 6-30, 6-90  
models  
    logging 2-24  
multiple boards, select a target 2-33  
MW\_custom build configuration 2-46

## N

near calls 2-40

## O

optimization, target specific 2-34  
optimize code 4-9  
OS requirements for Embedded Target for TI  
    C6000 DSP 1-4  
overflow mode 2-20  
overflow mode, about 2-20  
overrun indicator 2-22  
overrun notification method 2-45

## P

prerequisites for using Embedded Target for TI  
    C6000 DSP ix  
procedure for testing Embedded Target for TI  
    C6000 DSP 2-52  
profile generated code 3-10

profile report  
    about 3-10  
    reading 3-13  
    sample 3-13  
projects, create for CCS 2-109

## Q

Q format notation 4-5

## R

Radix-2 FFT block 6-31, 6-91  
Radix-2 IFFT block 6-33, 6-93  
Radix-4 Real FIR block 6-35, 6-95  
Radix-8 Real FIR block 6-37, 6-97  
Real IIR block 6-41, 6-101  
Real-Time Workshop solver options 2-26  
Reciprocal block 6-44, 6-104  
RTDX links. *See* links.  
RTW build options  
    generate\_code\_only 2-42  
run the EVM confidence test 2-51

## S

sample rate for C6711 DSK 2-15  
saturate 2-20  
select blocks for models 2-55  
select data type 2-17  
signed fixed-point numbers 4-4  
simulators and board selection, board selection,  
    simulators 2-34  
simulators, about 2-4  
solver option settings 2-26  
source and sink blocks 4-9  
stopping running models 2-54

suitable applications for Embedded Target for TI  
C6000 DSP 1-3  
Symmetric Real FIR block 6-45, 6-105

## T

table of blocks to avoid in models 2-55  
target Code Composer Studio 2-109  
target configuration options  
    build action 2-42  
    byte order 2-38  
    compiler verbosity 2-38  
    create .map files 2-39  
    generate code only 2-30  
    linker command file 2-39  
    make command 2-30  
    overrun action 2-45  
    overrun notification method 2-45  
    retain .asm files 2-38  
    retain .obj files 2-39  
    symbolic debugging 2-38  
    system target file 2-29  
    template makefile 2-29  
    user linker command file 2-42  
target LED 2-21  
target specific optimization 2-34  
test your Embedded Target for TI C6000 DSP  
    installation  
        C6701 EVM 2-51  
        C6711 DSK 2-81  
timer, configure 2-44  
To Rtdx block 6-177  
tutorial for C6701 EVM blocks 2-66  
tutorial for C6711 DSK blocks 2-97  
typographical conventions (table) xv

## U

use blocks for the C6701 EVM 2-66  
use blocks for the C6711 DSK 2-97  
use C62x DSP Library blocks 4-1  
use C6701 EVM blocks 2-66  
use c6701evmtest.md1 2-52  
use C6711 DSK blocks 2-97  
use logging in models 2-24  
use simulators for development 2-4  
user LED 2-21

## V

Vector Dot Product block 6-49, 6-109  
Vector Maximum Index block 6-50, 6-110  
Vector Maximum Value block 6-51, 6-111  
Vector Minimum Value block 6-52, 6-112  
Vector Multiply block 6-53, 6-113  
Vector Negate block 6-54, 6-114  
Vector Sum of Squares block 6-55, 6-115  
verify that c6701evmtest.md1 is running 2-54,  
    2-84

## W

Weighted Vector Sum block 6-56, 6-116  
working directory 2-67  
wrap 2-20