Embedded Target for the TI TMS320C6000™ DSP Platform

For Use with Real-Time Workshop®

■ Modeling

■ Simulation

■ Implementation

User's Guide

Version 2

The MathWorks

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.

Revision History

Contents

Targeting with DSP/BIOS Options

3

Using the C62x and C64x DSP Libraries

4

Block Reference

5

Supported Hardware and Issues

A

Index

What Is the Embedded Target for TI C6000 DSP

Introducing Embedded Target for the TI TMS320C6000 DSP Platform

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

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 1-9

Suitable Applications

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

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

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

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

About Embedded Target for C6000 DSP

Embedded Target for TI C6000 DSP lets you use Simulink to model digital signal processing algorithms from blocks in the 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.

Using This Guide

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

Expected Background

This document introduces you to using Embedded Target for C6000 DSPs with Real-Time Workshop to develop digital signal processing applications for the Texas Instruments CC6000 family of DSP development hardware, such as the TI 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, "What Is 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 2-88, 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 2-121 for more information about installing and using your C6711 DSK.

Configuration Information

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

```
c6000lib
```

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

- C6000 DSP Core Support
- C62x DSP Library
- C64x DSP Library
- C6416 DSK Board Support
- C6701 EVM Board Support
- C6711 DSK Board Support
- C6713 DSK Board Support
- DM642 EVM Board Support
- RTDX Instrumentation
- Target Preferences
- TMDX326040 Daughtercard Support

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

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

To verify that CCS is installed on your machine, enter

```
ccsboardinfo
```

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

```
Board Board                                  Proc Processor
            Processor
 Num  Name                                    Num  Name
                  Type
 ---  --------------------------------  ---
--------------------------------
  1   C6xxx Simulator (Texas Instrum ...  O    6701
                  TMS320C6701
  O   C6x11 DSK (Texas Instruments)        O    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.

Getting Started

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

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

Platform Requirements—Hardware and Operating System

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

- Intel Pentium or Intel Pentium processor compatible PC
- 64 MB RAM (128 MB recommended)
- 20 MB hard disk space available after installing MATLAB
- Color monitor
- One full-length peripheral component interface (PCI) slot available to use the C6701 EVM internally in your PC
- CD-ROM drive
- Windows 2000 or Windows XP.

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

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

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

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

Texas Instruments Software

In addition to the required software from The MathWorks, Embedded Target for TI C6000 DSP requires that you install the Texas Instruments development tools and software listed in the following table. Installing Code Composer Studio IDE for the C6000 series, the latest version, installs the software shown.

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

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

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

- One or more Texas Instruments TMS320C6416 DSP Starter Kits
- One or more 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 DM642 Evaluation Modules
- One or more boards from the supported hardware lists

1-11

• 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.

2

Targeting C6000 DSP Hardware

Overview

The Embedded Target for the TI TMS320C6000 DSP Platform lets you use Real-Time Workshop to generate a C language real-time implementation of your Simulink model. You can compile, link, download, and execute the generated code on the Texas Instruments (TI) C6701 Evaluation Module (C6701 EVM) and C6711 DSP Starter Kit (DSK). In combination with the supported boards (refer to"Supported Hardware for Targeting" on page A-2), your Embedded Target for TI C6000 DSP software is the ideal resource for rapid prototyping and developing embedded systems applications for 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. Additional hardware that we support is listed in "Supported Hardware and Issues" on page A-1.

Although the tutorials in this chapter focus on the C6701 EVM and the C6711 DSK, the techniques and processes apply to any supported hardware, with minor adjustments for the processor involved.

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

About the Tutorials

In most cases, this chapter deals with either the C6701 EVM or the C6711 DSK targets. Fortunately, all members of the C6000 family of processors that we support work in a manner similar to the C6701 EVM and C6711 DSK. While you review the contents of this chapter, and follow the tutorials, recall that the concepts and techniques or development processes apply, with a few adjustments, to all supported C6000 processors and boards.

Later sections discuss the Real-Time Workshop embedded coder and targeting custom hardware.

TI C6000 and Code Composer Studio IDE

Texas Instruments (TI) markets a complete set of software tools to use when you develop applications for your C6000 hardware boards. This section provides a brief example of how the Embedded Target for TI C6000 DSP uses Code Composer Studio™ (CCS) Integrated Development Environment (IDE) with the Real-Time Workshop and the C6000lib blockset.

Executing code generated from Real-Time Workshop on a particular target in real time requires that Real-Time Workshop generate target code that is tailored to the specific hardware target. Target-specific code includes I/O device drivers and an interrupt service routine (ISR). Since these device drivers and ISRs are specific to particular hardware targets, you must ensure that the target-specific components are compatible with the target hardware. To allow you to build an executable, TI C6000 uses the MATLAB links 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 "Supported Hardware and Issues" for the latest information about the hardware supported by the Embedded Target for TI C6000 DSP.

About Simulators

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

the simulators as well. For more information about the simulators in CCS, refer to your CCS online help system.

When you set up a simulator, match the processor on your target exactly to simulate your target hardware. To target 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.

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

Using a Simulator

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

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

1 Click the target preferences block in your model and select **Edit—>Open Block** from the menu bar for your model. This step opens the **C6000 Target Preferences** dialog for your target.

2 On the **Board info** pane in the C6000 Target Preferences dialog, select **Simulator**.

3 Click **Apply** to apply the change, or click **OK** to apply the new setting and close the dialog.

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

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

Build Options -> Linker -> Basic

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

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

Using RTDX with a Simulator

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

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

Typical Hardware Setup for 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.
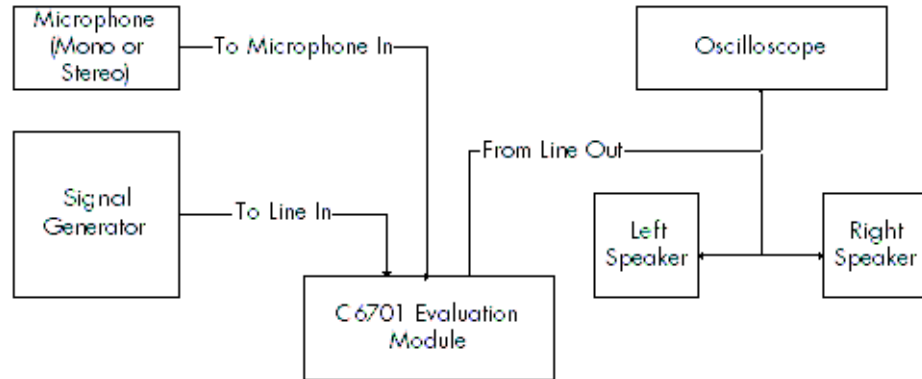
Block Diagram of Typical Inputs and Outputs to the C6701 EVM

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

- 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 Instrumentation
  - 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 DSP Library
  - Provides fixed-point blocks for models that use fixed-point mathematics and algorithms.
- C64x DSP Library
  - Provides fixed-point blocks for models that run on C64x processors.

- C6416 DSK Board Support blocks
  - C6416 DSK ADC—configures the analog to digital converter
  - C6416 DSK DAC—configures the digital to analog converter
  - C6416 DSK DIP Switch—simulates or reads the user-defined DIP switches on the C6701 EVM
  - C6416 DSK LED—controls the user status light emitting diodes (LED) on the C6416 DSK
  - C6416 DSK Reset—resets the current C6701 EVM
- C6713 DSK Board Support blocks
  - C6713 DSK ADC—configures the analog to digital converter
  - C6713 DSK DAC—configures the digital to analog converter
  - C6713 DSK DIP Switch—simulates or reads the user-defined DIP switches on the C6713 EVM
  - C6713 DSK LED—controls the user status light emitting diodes (LED) on the C6713 EVM
  - C6713 DSK Reset—resets the current C6713 EVM
- C6000 DSP Core Support blocks
  - Blocks that provide data transfer to and from memory on all C6000 hardware—To Memory and From Memory.
- DM642 EVM Board Support blocks
  - DM642 EVM Video ADC—configures the analog to digital converter
  - DM642 EVM Video DAC—configures the digital to analog converter
  - DM642 EVM LED—controls the user status light emitting diodes (LED) on the DM642 EVM
  - DM642 EVM Reset—resets the current DM642 EVM
- TMDX326040 Daughtercard Support blocks
  - Blocks that configure the peripherals associated with the daughtercard.
- Target Preferences blocks

  Blocks that allow you to set the target processor for a model and configure processor specific features, such as memory allocation, target board selection, and compiler section layout. Every model that you target to a C6000 processor or board requires one of these blocks. You cannot add more than one target preferences block to a model. MATLAB returns an

error if you do not include a target preferences block in your C6000 target model, or you include more than one preferences block.

- C6416DSK—set the model to target a board that uses the C6416 processor
- C6701EVM—set the model to target a board that uses the C6701 processor
- C6711DSK—set the model to target a board that uses the C6711 processor
- C6713DSK—set the model to target a board that uses the C6713 processor
- DM642EVM—set the model to target a board that uses the DM641 processor
- Custom C6000—set the model to target a board that uses any C6000 family processor

These blocks are associated with your boards and hardware. As needed, add the devices to your model. If you choose not to include either an ADC or DAC block in your model, 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 RTDX Instrumentation that contains RTDX input and output blocks that apply to all C6000 development boards and the C6000 DSP Core support library that contain blocks that let you transfer data to and from memory on any C6000-based target. Like the RTDX blocks, the core support blocks are not hardware dependent.

With your model open, select **Configuration Parameters** from the **Simulink** option to open the **Configuration Parameters** dialog box. In the **Select** tree, click `Real-Time Workshop`. You must specify the appropriate versions of the system target file and template makefile. For the 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 in `Real-Time Workshop system target file`. Or click **Browse** and select `ti_c6000.tlc` from the list of targets, or whichever target best matches your hardware.

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

generated C code as inlined S-functions. Inlined S-functions offer speed advantages and simplify the generated code. For more information about inlining S-functions, refer to your target language compiler documentation. For a complete discussion of S-functions, refer to your documentation about writing S-functions.

During the same build operation, the template makefile and block parameter dialog 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 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
- C64x DSP Library—provides fixed-point blocks for models that run on C64x processors.
- C6416 DSK Board Support—blocks to configure the peripherals on the C6416 DSK.
- C6713 DSK Board Support—blocks to configure the peripherals on the C6713 DSK.
- DM642 EVM Board Support—blocks to configure the video peripheral devices and LEDs on the DM642 EVM
- C6000 DSP Core Support—blocks that provide data transfer to and from memory on all C6000 hardware.
- TMDX326040 Daughtercard Support—blocks that configure the peripherals associated with the daughtercard.
- Target Preferences—icon blocks that allow you to set the target type for a model. You must include one of these blocks in any model you plan to target to a C6000 processor or board

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

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

- ADC block

- DAC block

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

- LED block

- Reset block

Similarities in the C6000 boards result in the ADC, DAC, DIP Switch, LED, and Reset blocks for the C6000-based boards being almost identical. Each section about a block, such as the ADC block, presents all possible options for the block, noting when an option applies only to a board-specific version of the ADC block. For example, the **Codec data format** option for ADC blocks applies only to the C6701 EVM ADC.
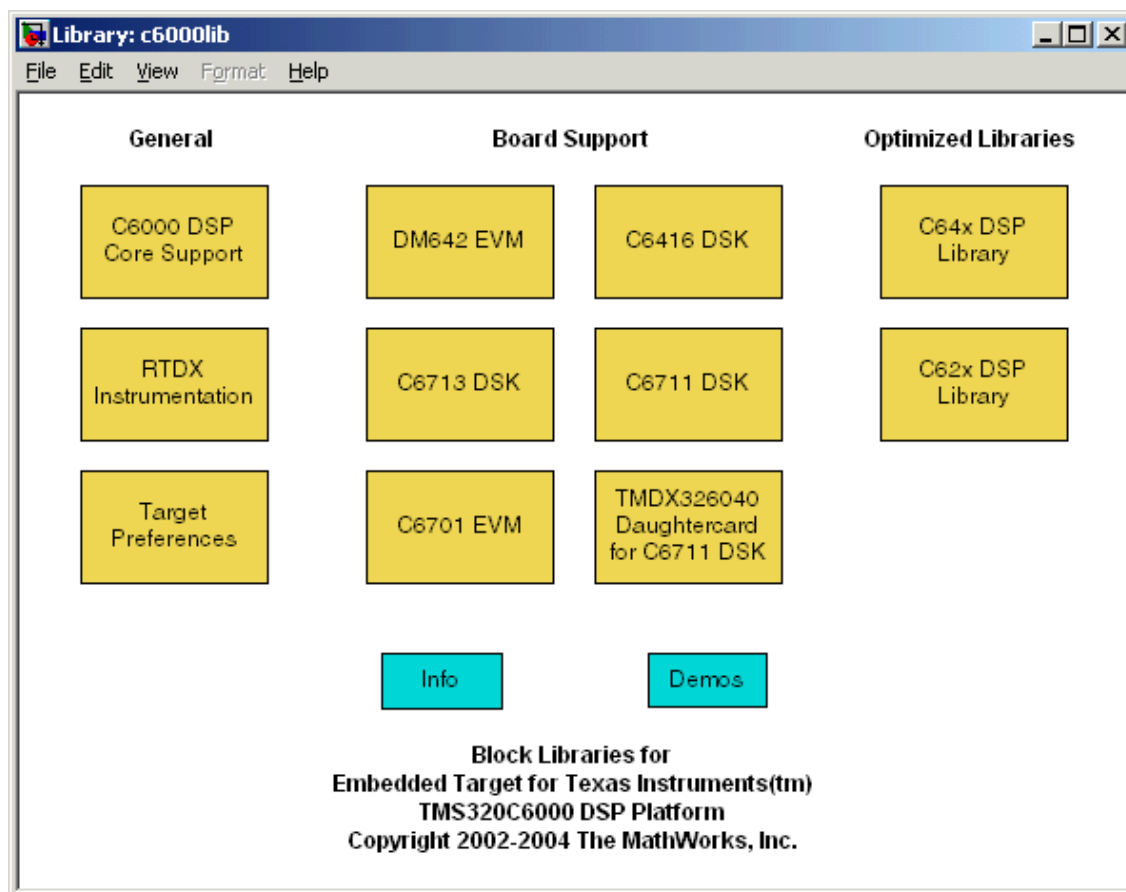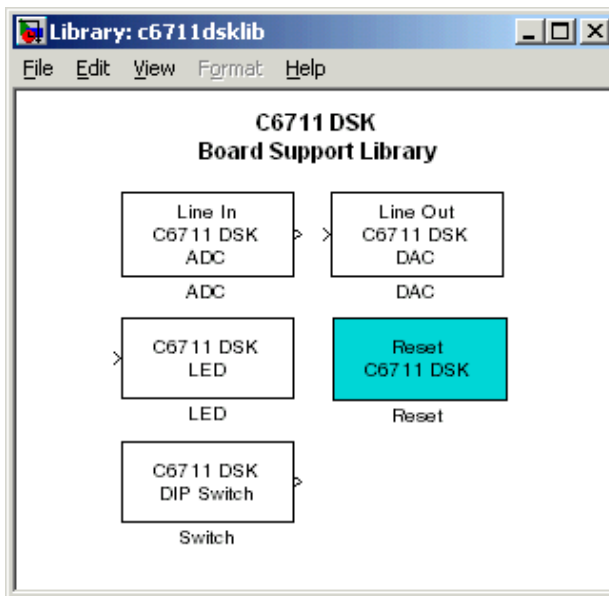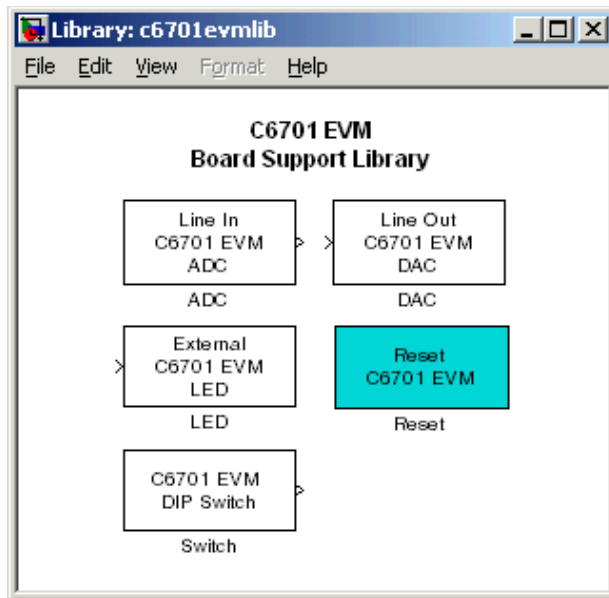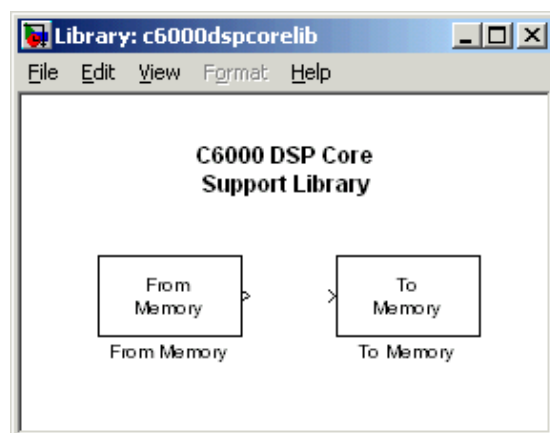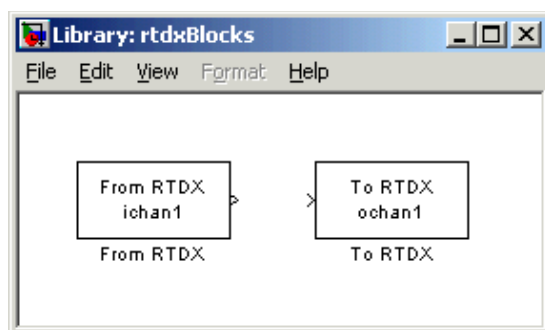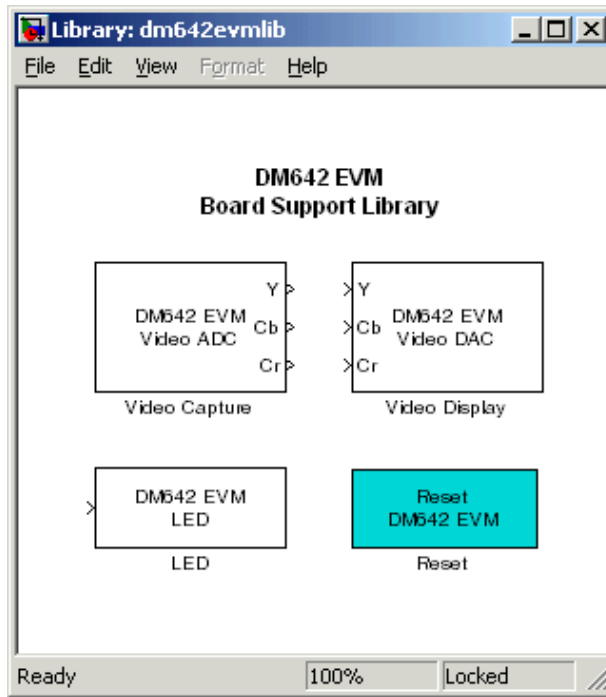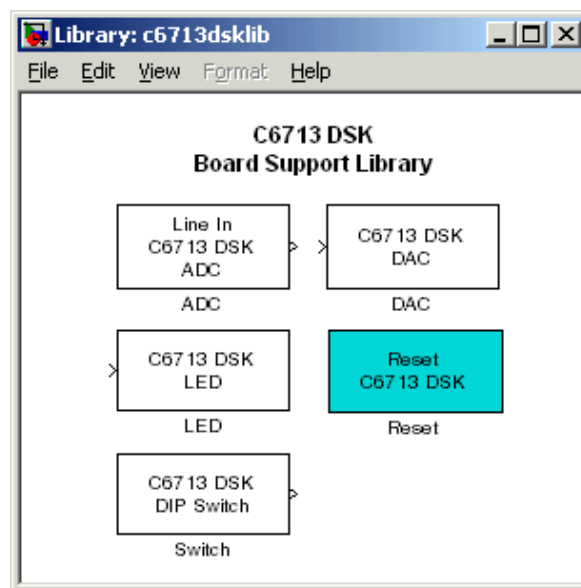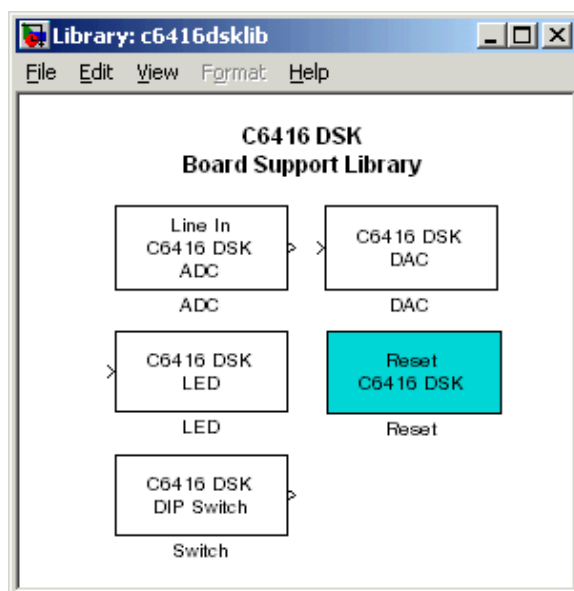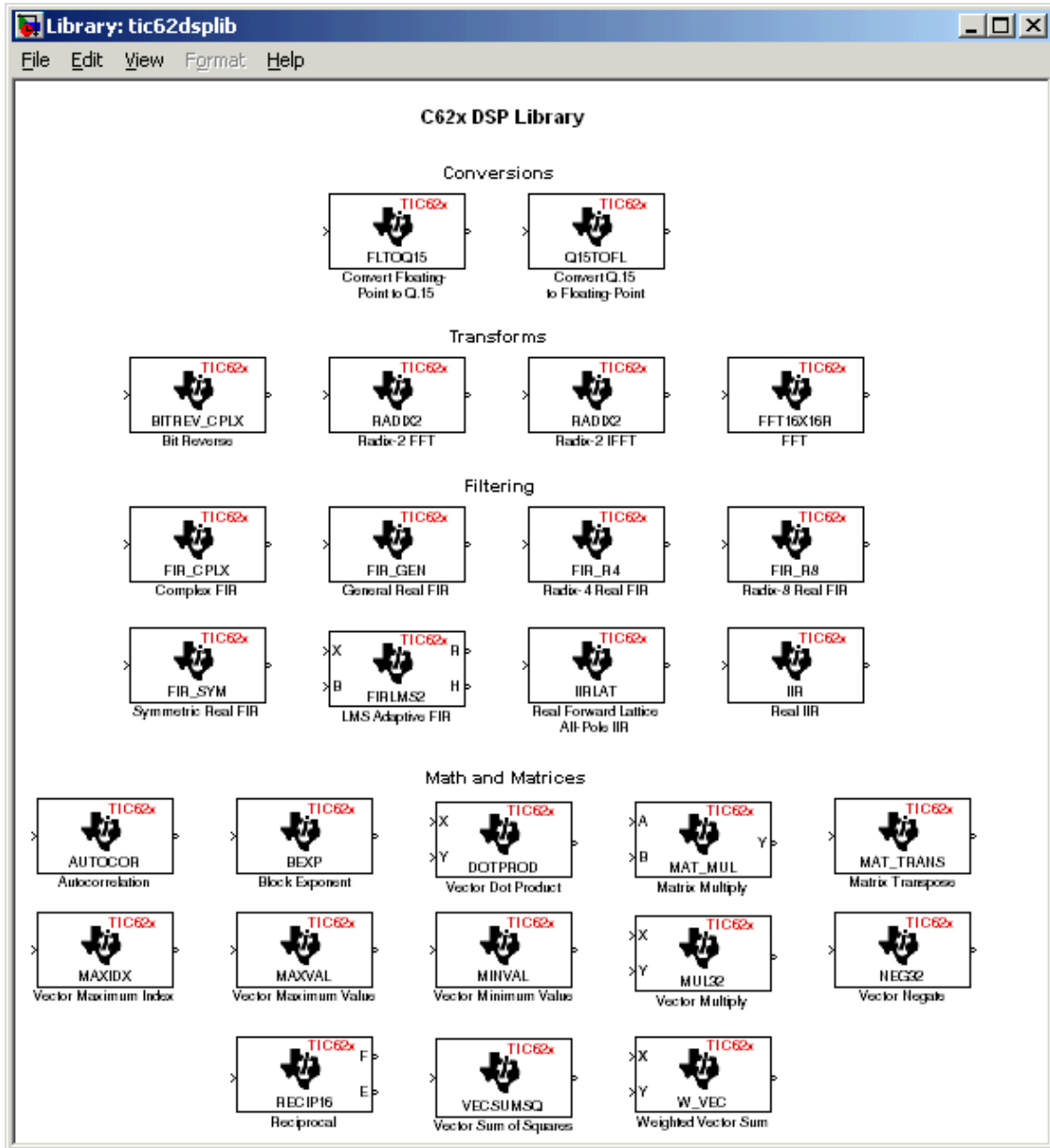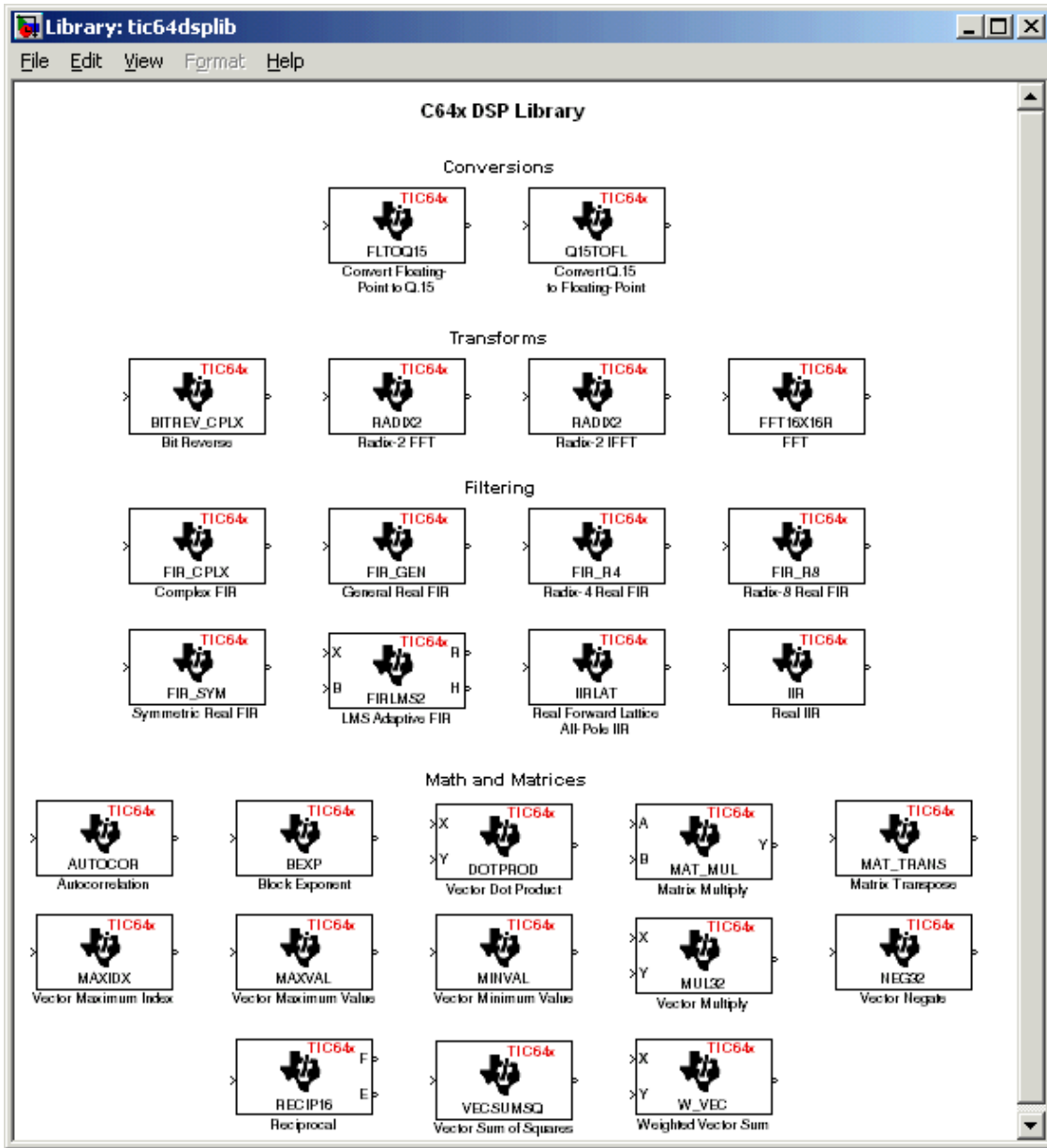
Configuring ADC Blocks

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

Adding the ADC block to your Simulink model enables the codec on the target to accept input from your external source. By connecting your source to the LINE IN connector on the board mounting bracket, you introduce signals to the board. Your ADC block defines the signal format the codec uses to sample, digitize, and send signals to the digital signal processor. When you build your Simulink model, the build process includes the software to implement the ADC-defined codec operation into the code downloaded to the board.

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

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.

Option	Affected Hardware
ADC Source	Codec
Codec data format (C6701 EVM ADC only)	Codec
Mic	Codec
Output data type	TMS320C6xxx digital signal processor
Sample rate (Hz) (C6701 EVM ADC only)	Codec
Samples per frame	Direct memory access functions
Scaling	TMS320C6xxx digital signal processor
Source gain (dB)	Codec
Stereo (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.
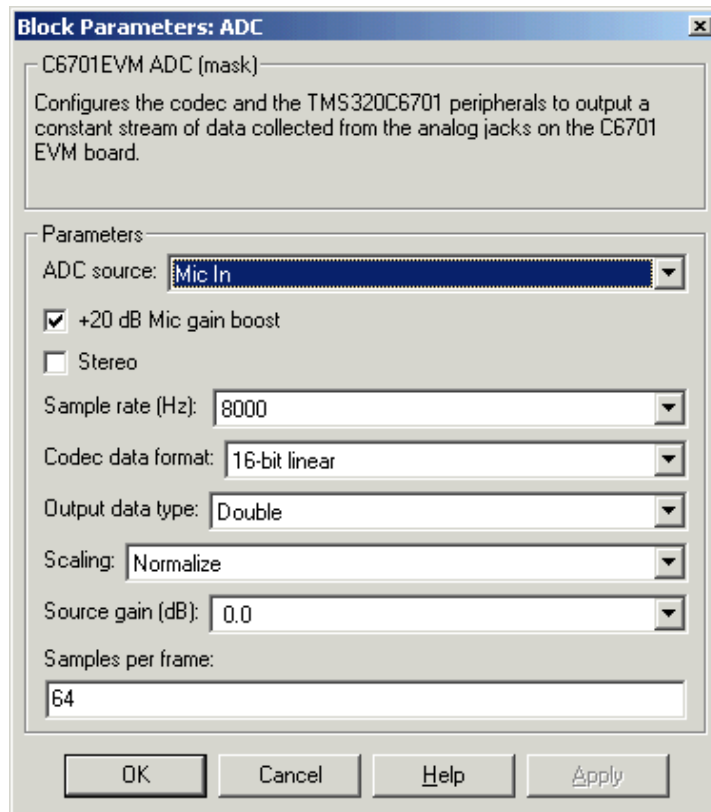
Figure 2-1: Block Parameters for C6701 EVM ADC Dialog

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; μ-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 μ law—used in the telephone industry most frequently. μ-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

2-25

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
Codec data format (C6701 EVM DAC only)	Codec
DAC attenuation	Codec
Overflow mode	Digital Signal Processor
Scaling	Digital Signal Processor

When you double-click the C6701 EVM DAC block, you see the dialog shown here.
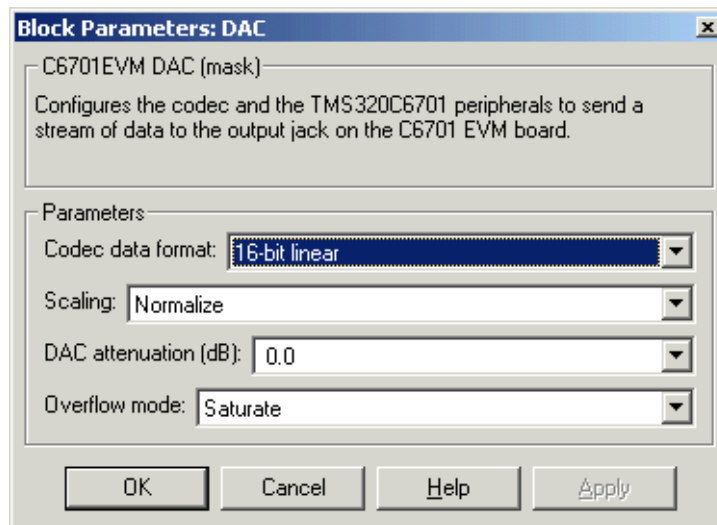
Choosing the Codec Data Format (C6701 EVM DAC Block Only)

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.
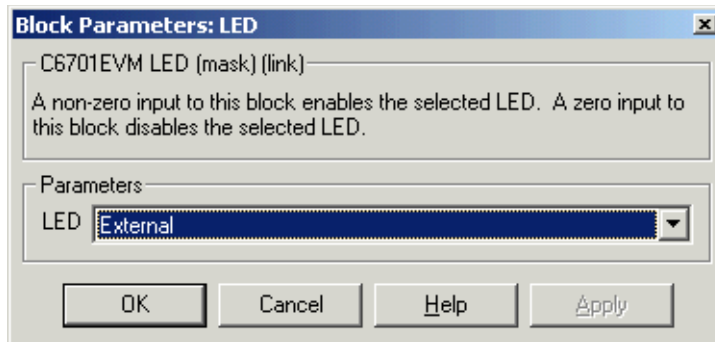
When you add an C6701 EVM LED block to your model, you use the **LED** list to select which LED the block controls—internal or external. Select External from the list to have the C6701 EVM LED block trigger the LED located on the C6701 EVM mounting bracket at the back of your PC. To trigger the internal 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 **Overrun notification method** selections that turn on the LEDs use the external LED or user-defined LEDs to signal model conditions. If you are using the overrun indicator, consider not using an LED block to trigger the external LED on the C6701 EVM, or the user LEDs on the C6711 DSK until you stop monitoring your process for overrun conditions.

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

- None—your model does not respond to overrun conditions during processing.
- Notify_and_continue—when your model runs out of clock cycles before completing enough of the processing algorithm, the overrun indicator executes the option you chose for **Overrun notification method**. The model continues to run without pause.
- Notify_and_halt—when your algorithm runs out of clock cycles before completing the required calculations and manipulations, the model stops executing and notifies you about the overrun using the method you select for the **Overrun notification method**.

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

- Print_message—print a message to stdout, or the message log when your application uses DSP/BIOS
- Turn_on_LEDs—illuminate the user LEDs on the target, either the external LED on the C6701 EVM or all the user LEDs on the C6711 DSK
- Print_message_and_turn_on_LEDs— light the LEDs and print a message

Configuring Reset Blocks

Each target library offers a block that performs a software reset of the appropriate board—a Reset block. While they are blocks, 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.

Configuring Target Preferences Blocks

One block is required in all C6000 targets models—a block from the Target Preferences library in the C6000 Block (c6000lib) library. Adding one of these blocks to your model provides direct access from the model to the **C6000 Target Preferences** dialog where you choose your target board and processor, define your target board memory map, and set data and code sections, such as compiler, DSP/BIOS, and user sections.
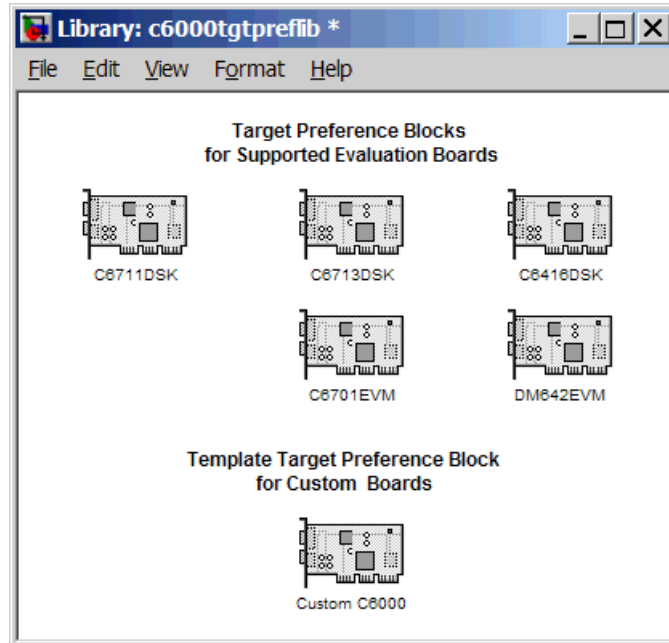
To target a model to a C6000 processor based target, add one of the following target preferences blocks to the model. The blocks with specific board names represent blocks that are preconfigured with default values to match the

attributes of the named Texas Instruments DSP starter kits and evaluation modules:

Target Preferences Block	Description
Custom C6000	Provides access to the hardware set up for targeting any C6000 processor-based board. Note that it does not set any default values. When you add this block to a model, you must set all the options on each available pane— board information, memory mapping, and section layout.
C6416DSK	Sets default values for targeting the C6416 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6701EVM	Sets default values for targeting the C6701 EVM. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6711DSK	Sets default values for targeting the C6711 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.

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

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

If the target preferences block in your model indicates a target name which does not exist in CCS Setup, Embedded Target for TI C6000 DSP places the new project into the CCS IDE window for board 0 as defined in the CCS Setup Utility. When you have more than one target configured for CCS, such as boards 0 and 1, this default behavior may not put the project in the target you intend. Warning messages such as

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

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

Double-clicking one of the above blocks in a model opens the **C6000 Target Preferences** dialog, shown here in the custom board configuration.
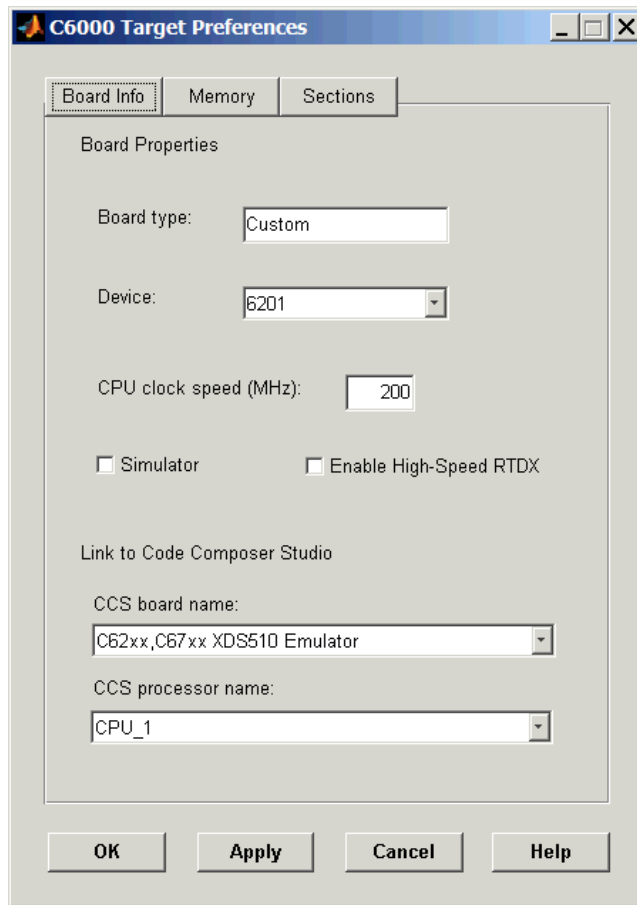
All target preferences block dialogs provide tabbed access to panes that include options you set for the target processor and target board:

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

Board Info Pane

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

Board Type

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

Device

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

CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results may be wrong, and code profiling results will not be correct.

You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

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

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. 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. Other C6000 processors allow different clock speeds.

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

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, `100 MHz` for example, the timer calculates the sine generator interrupt period as follows for the sine block:

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

To create sine block interrupts at 0.001 s/sample requires

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

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

Simulator

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

Enable High-Speed RTDX

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

CCS Board Name

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

CCS Processor Name

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

Memory Pane

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

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

- **Physical Memory**—specifies the processor and board memory map
- **Heap**—specifies whether you use a heap and determines the size in words

• **L2 Cache**—enables the L2 cache (where available) and sets the size in kB

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

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

Physical Memory Options

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

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

- Custom boards based on C670x processors provide IPRAM and IDRAM memory segments by default.
- C6701 EVM boards provide IPRAM, IDRAM, SBSRAM, SDDRAM0, and SDRAM1 memory segments by default
- C6711DSK boards provide SDRAM memory segment by default

Name

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

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

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

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

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

Address

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

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

Length

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

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

Contents

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

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

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

Add

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

Remove

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

Create Heap

If your processor supports using a heap, as do the C6711 or C6701, for example, selecting this option enables creating the heap, and enables the **Heap size** option. **Create heap** is not available on processors that either do not provide a heap or do not allow you to configure the heap.

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

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

Heap Size

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

Define Label

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

Heap Label

Enabled by selecting **Define label**, you use this option to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Enable L2 Cache

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

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

L2 Cache size

Once you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

Sections Pane

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

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

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

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

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

String	Section List	Description of the Section Contents
.stack	Compiler	The global stack.
.switch	Compiler	Jump tables for switch statements in the executable code.
.sysdata	DSP/BIOS	Data about DSP/BIOS.
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code.
.sysmem	Compiler	Dynamically allocated object in the code. Contains the heap.
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants.
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation.

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

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

- .cinit
- .const
- .switch
- .text—created by the assembler.

These sections are uninitialized:

- .bss—created by the assembler.

- .far
- .stack
- .sysmem

Other sections appear on the list as well:

- .data—created by the assembler. The C/C++ compiler does not use this section.
- .cio
- .pinit

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

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

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

DSP/BIOS Sections

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

Description

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

Placement

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

DSP/BIOS Object Placement

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

Custom Sections

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

Name

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

Placement

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

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

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

Configuring DM642 EVM Video ADC and Video DAC Blocks

Preparing the Video ADC and Video DAC blocks for the DM642 EVM is quite different from configuring the ADC and DAC blocks for the other supported hardware. Rather than being just analog-to-digital converters, or digital-to-analog, the DM642 EVM blocks allow you to capture and display video data. While conversion is a part of what they do, the configuration process sets up the video formats that the video ADC (capture) block accepts and the video DAC (output) block provides for display.

Configuring the Video ADC or Video Capture Block

To capture video from the video inputs on your DM642 EVM, add the DM642 EVM Video ADC block to capture the video input. Options in the block let you set the output format and output mode of video that leaves the block for processing in your model. Setting the options configures the peripherals on your board, and the SAA7115 or TVP5146/5150 video decoders to capture and reformat video for output from the block for further processing.
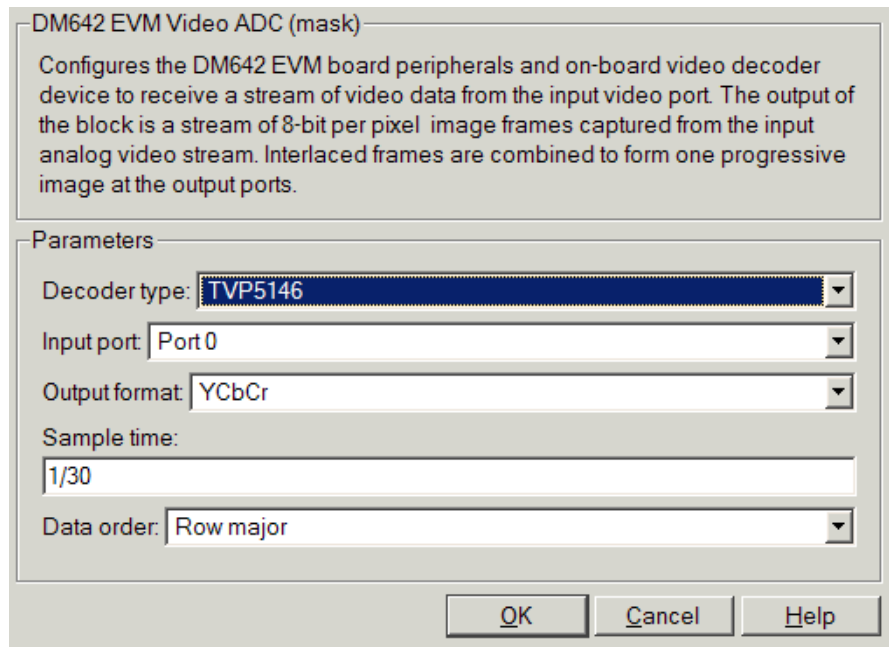
From the available options in the dialog, you see that you do not need to configure the block for the type of video input. The block accepts whatever supported video you provide at the video input ports. Options in the dialog let you specify how to output the video from the block to downstream model elements and which video decoder you DM642 EVM uses.

Option	Affected Hardware
Decoder type	Video decoder, either SAA7115 or TVP5146/5150
Input port	Video decoder
Output mode	SAA7115H video decoders (available when you set **Decoder type** to SAA7115)
Output format	SAA7115H video decoders

Option	Affected Hardware
Sample time	Clock rate and decoders
Data order	Buffers and decoder

Setting the Decoder type

Your selection from the list configures the block options to support either the TVP5146 Decoder on the DM642 EVM or the SAA7115 Decoder, depending on the model of your board. Choose one option from the list—TVP5146 or SAA7115. When you select SAA7115 for the type of decoder, the dialog adds an option—**Output Mode**. Generally, older DM642 EVM boards use the SAA7115 decoder option. Newer boards use the default setting TVP5146 decoder. Refer to "Identifying Your DM642 EVM Board Revision" on page A-4 for information about identifying the revision of your DM642 EVM.

Choosing the Input port

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

Selecting the Output Mode (SAA7115 Decoder only)

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

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

Output Mode	Description
NTSC 720x480	525 horizontal lines stacked on top of each other, with varying numbers of lines making up the horizontal resolution. There are 59.94 fields displayed per second. Each field is a set of even lines, or odd lines. Displaying the even and odd fields sequentially by interlacing them creates each full 60 field frame. One full frame is displayed about every 1/30 of a second (30 Hz refresh rate).
NTSC 640x480	Scales the output to standard (SDTV) mode.
PAL 720x576	625 horizontal lines making up the vertical resolution. 50 fields are displayed and interlaced per second, making for a 25 frame per second system (25 Hz refresh rate).

Setting the Output format

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

Digital Output Format	Description
RGB24	Output uses 8 bits each of red, green, and blue colors to represent the color of each pixel in the image. RGB color space is device-dependent.
YCbCr	Output from the block includes one luminance channel Y (essentially the black/white signal) and two chrominance (color) channels Cb and Cr to represent the color image data per pixel. This is the digital standard color space DVDs use.
Y	Black/White video. No color/chromaticity values.

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

Sample Time

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

Note that

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

Data Order

With data order, you control the way the the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

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

DM642 EVM Video ADC blocks store the image data in row major format because most video capture devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

MATLAB and Simulink use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video ADC blocks.

Configuring the Video DAC or Video Output Block

To provide video output from your running process on your target, add the Video DAC block to your model. Options for the block let you determine the video format for the output and center the image in the display.
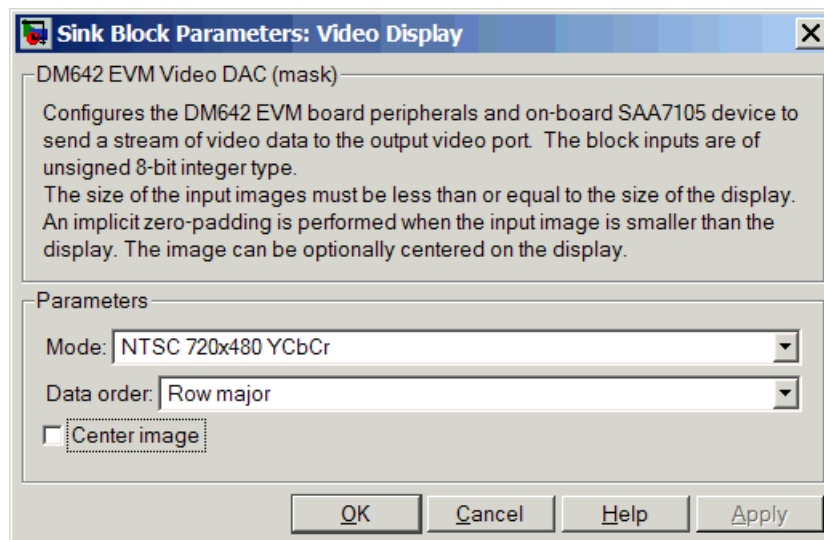
You see from the options that the block provides parameters that control the video encoder on the DM642 EVM.

Option	Affected Hardware
Mode	SAA7105 video encoder
Data order	SAA7105 video encoder
Center image	SAA7105 video encoder

Selecting the Mode

Unlike the DM642 EVM Video ADC block, this DAC block does not convert the video between formats. Nor does this block inherit any settings from the DM642 EVM Video ADC block, as some of the other C6000 DAC blocks do.

The **Mode** option specifies both the video format the block accepts and the format the block outputs to the video output ports on the EVM.

To be able to be displayed, images that you send to the block should be equal to or smaller than your target display size. If the input images are smaller than the target display size, the block pads the image by adding zeros to the image.

Analog Output Mode	Description
NTSC 720x480 YCbCr	Analog output of video data in 720-by-480 pixels format with full color
NTSC 640x480 Y	Analog video output in 640-by-480 pixels format with black and white only (luminance). No color data.
SVGA 800x600 RGB24	Full super VGA format 800-by-600 pixels with three color channels: 8-bit red, 8-bit green, and 8-bit blue data.

Data Order

With data order, you control the way the the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

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

DM642 EVM Video DAC blocks store the image data in row major format because most video display devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

MATLAB and Simulink use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video DAC blocks.

Selecting the **Center image** option instructs the block to center the output image on the display. Note that centering the image requires some computation by the processor so there are small time and CPU cycles penalties for choosing this option. For that reason, **Center image** is cleared by default.

Another note of interest—some cameras pad their video output with zeros to ensure that the display does not cut off the image on one side, usually the left. Images that include such padding may appear to be off-center on the display. In fact, while the displayed image may not appear centered, the electronic image (the data that compose the displayed image plus the padding which you can not see) is centered in the display area.

Creating DSP Application Models for Targeting

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

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

- Add a target preferences block from the Target Preferences library (`c6000tgtpreflib`) to configure your generated code for your target processor

Using Logging in Your DSP Applications

Simulink offers various data logging capabilities in the **Configuration Parameters** dialog for your model. Found on the `Data Import/Export` pane of the **Configuration 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 `Data Import/Export` pane in the **Configuration 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->Configuration Parameters** from the menu bar in your model.

2 Click `Data Import/Export` in the **Select** tree to access the `Data Import/Export` pane.

3 Clear the options in the **Load from workspace** and **Save to workspace** fields.

  - **Input**
  - **Initial state**
  - **Time**
  - **States**
  - **Output**
  - **Final states**

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

```
dspstartup
```

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

Generating Code from Real-Time Models

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

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

1 Real-Time Workshop invokes the function `make_rtw` to start the Real-Time Workshop build procedure for a block diagram. `make_rtw` invokes the Target Language Compiler to generate the code and then invokes the language specific make procedure.

2 `gmake` builds file `modelname.out`. Depending on the build options you select in the **Configuration Parameters** dialog, `gmake` can download and execute the model on your TI target board.

Setting Real-Time Workshop Options for C6000 Hardware

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

Note Embedded Target for TI C6000 does not support continuous states in Simulink models for code generation. In the `Solver options` in the **Configuration Parameters** dialog, 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 **Configuration Parameters** dialog lets you set numerous options for the real-time model. To open the **Configuration Parameters** dialog, select **Simulation -> Configuration Parameters** from the menu bar in your model. The following figure shows the Real-Time Workshop categories when you are using the Embedded Target for TI C6000 DSP.
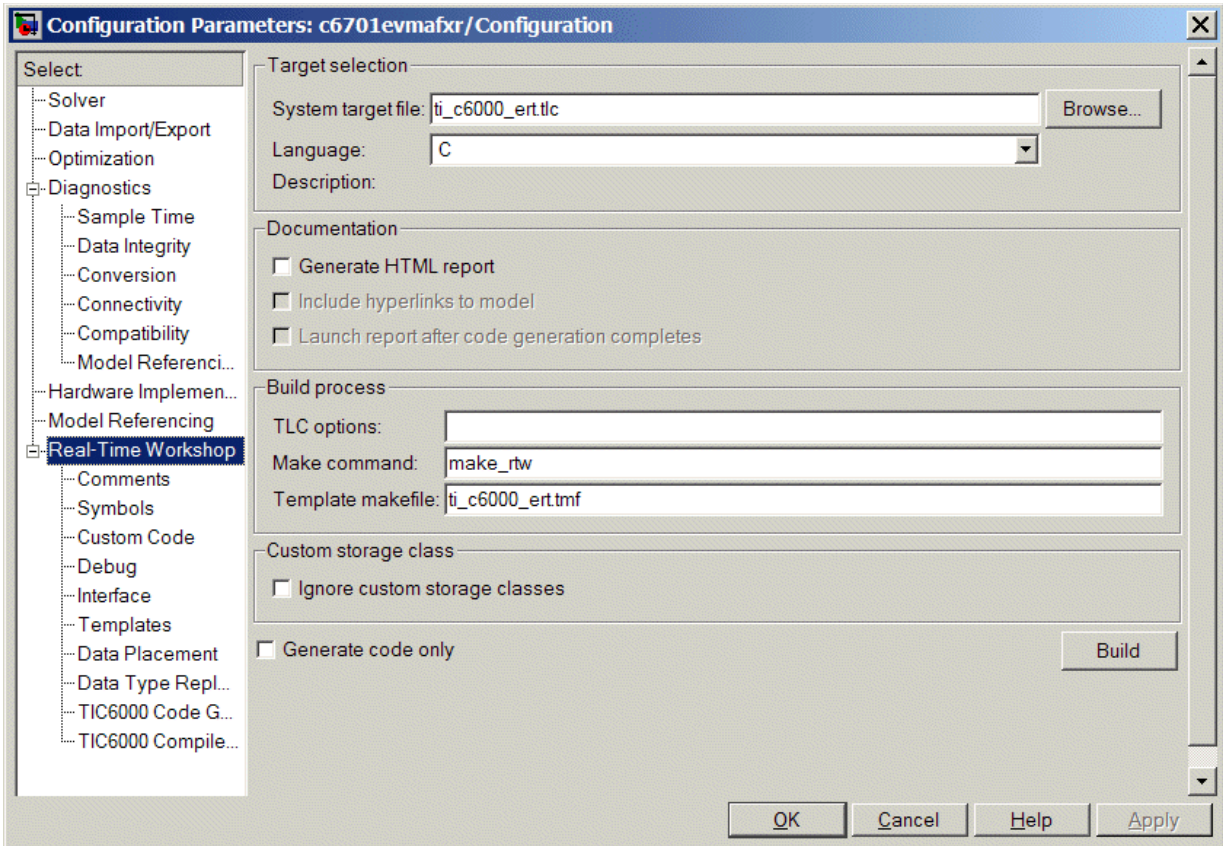
In the **Select** tree, the categories provide access to the options you use to control how Real-Time Workshop builds and runs your model. The first categories under Real-Time Workshop in the tree apply to all Real-Time Workshop targets including the target and always appear on the list.

The last categories under Real-Time Workshop are specific to the Embedded Target for TI C6000 DSP target ti_C6000.tlc and appear when you select any TI C6000 target.

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

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

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

Real-Time Workshop Pane Options

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

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

When you select the appropriate C6000 target (`ti_c6000.tlc`) in **System target file**, you enable the automatic board selection for your model. After that, opening the **Configuration Parameters** dialog 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.
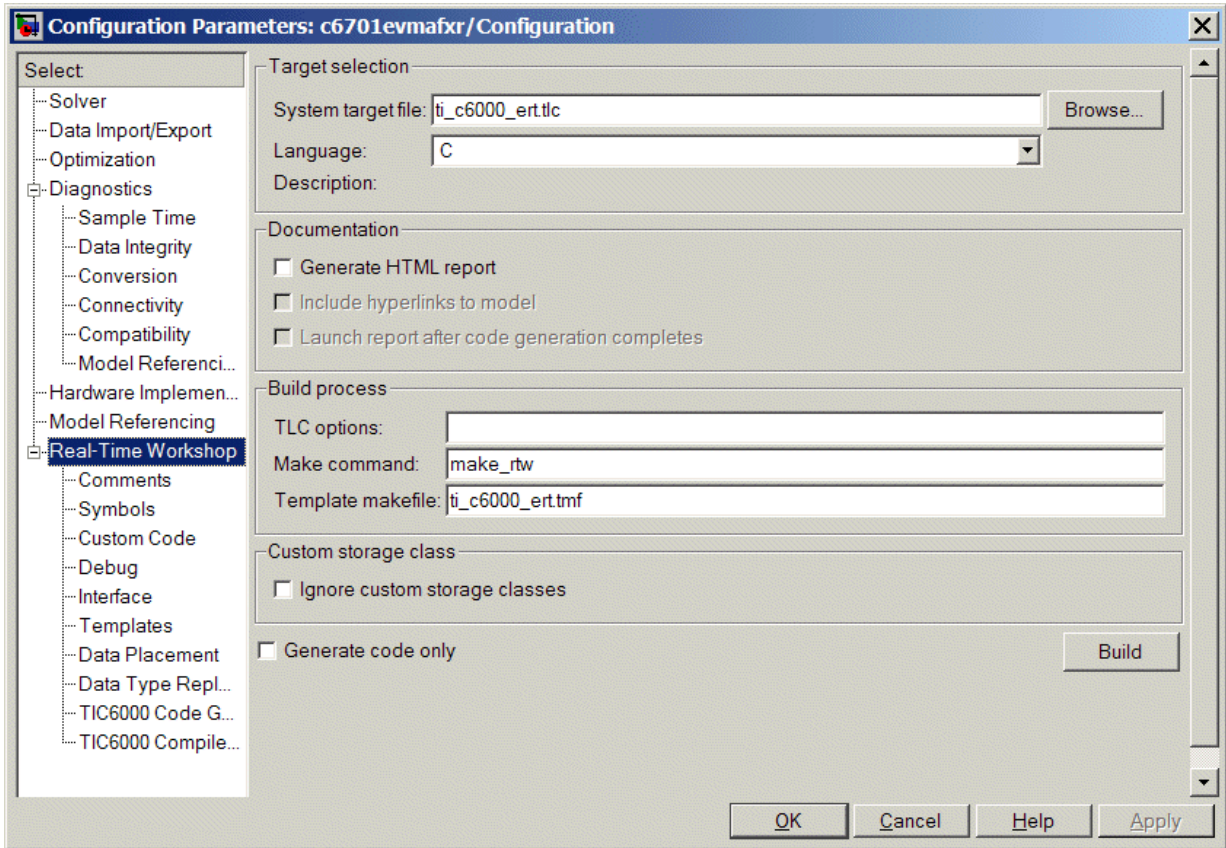
Target Selection

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

If you are using the Real-Time Workshop Embedded Coder software, select the `ti_c6000_ert.tlc` target here.

Documentation

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

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

at the MATLAB prompt.

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

Launch report after code generation completes. Automatically opens a MATLAB Web browser window and displays the code generation report. When you clear this option, you can open the code generation report (*modelname*_codegen_rpt.html or *subsystemname*_codegen_rpt.html) manually in a MATLAB Web browser window, or in another Web browser manually.

Build Process

Template makefile. Real-Time Workshop uses template makefiles to generate the makefile for building the executable file. During the automatic build process, MATLAB issues the `make_rtw` command. `make_rtw` extracts information from the template makefile `ti_c6000.tmf` and creates the actual makefile `c6000.mk`. When Real-Time Workshop compiles the model, it uses the actual makefile to generate the compiled code for the target.

Set the **Template makefile** option to `ti_c6000.tmf` when you build your application for the C6000 target. If the template makefile shown in the option is not `ti_c6000.tmf`, click **Browse** to open the list of available system target

files and select the correct file from the list. Real-Time Workshop then selects the appropriate template makefile.

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

Custom Storage Class

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

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

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

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

Generate code only. This option does not apply to targeting with the Embedded Target for TI C6000 DSP. To generate source code without building and executing the code on your target, select TI C6000 runtime from the **Category** list in the Select tree. Then, under **Runtime**, select Generate code only for **Build action**. You cannot use DSP/BIOS features when you use the Generate code only option for the **Build action**.

Debug Pane Options

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

- View the TLC call stack.

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

When you select Debug from the **Select** tree, you see the Debug options as shown in the next figure. Within this, you set options that are specific to Real-Time Workshop process and TLC debugging.
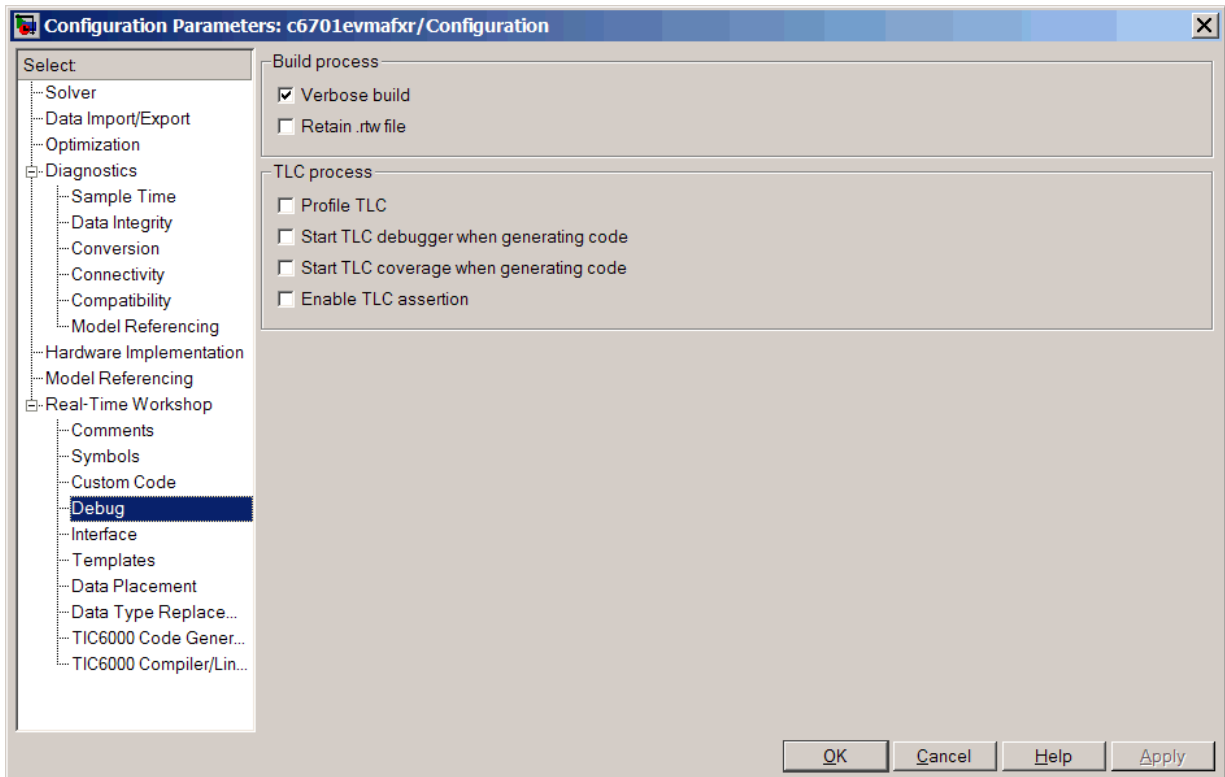
For details about using the options in Debug, refer to the section "About TLC Debugger" in your Real-Time Workshop documentation.

Optimization Pane Options

On the Optimization pane in the **Configuration Parameters** dialog, you set options for the code that Real-Time Workshop generates during the build

process. You use these options to tailor the generated code to your needs. Select `Optimization` from the **Select** tree on the **Configuration Parameters** dialog. The figure shows the `Optimization` pane when you select the system target file `ti_C6000.tlc` under **Real-Time Workshop system target file**.
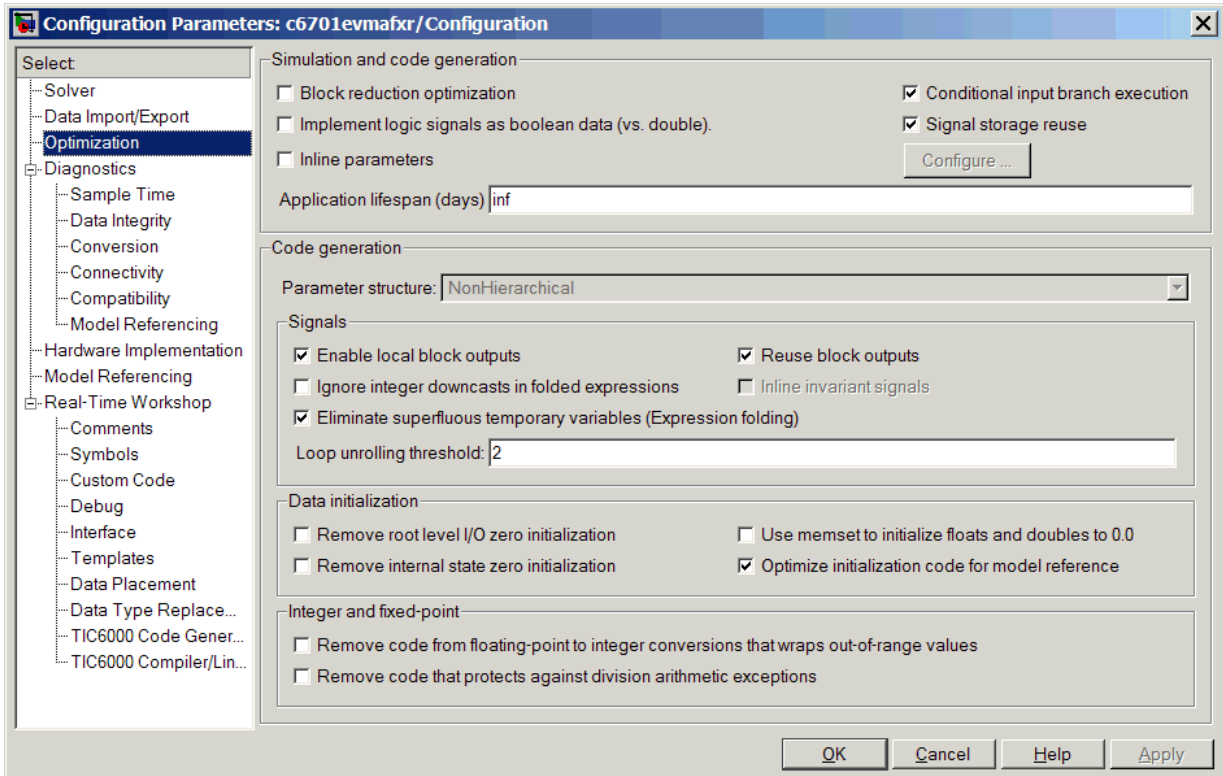
These are the options typically selected for Real-Time Workshop:

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

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

TI C6000 Code Generation Pane Options

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

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

Target Selection

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

Code Generation

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

- **Incorporate DSP/BIOS**
- **Profile performance at atomic subsystem boundaries**
- **Inline Signal Processing 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 includes instrumentation based on DSP/BIOS objects. "Introducing DSP/BIOS" on page 3-2 provides details about the changes that occur in your generated code when you include DSP/BIOS.

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

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

To allow you to specify whether the functions generated from blocks in your model are used inline or by pointers, **Inline Signal Processing Blockset functions** tells the compiler to inline each Signal Processing Blockset function. Inlining functions can make your code run more efficiently (better optimized) at the expense of using more memory. As shown in the figure, the default setting uses inlining to optimize your generated code.
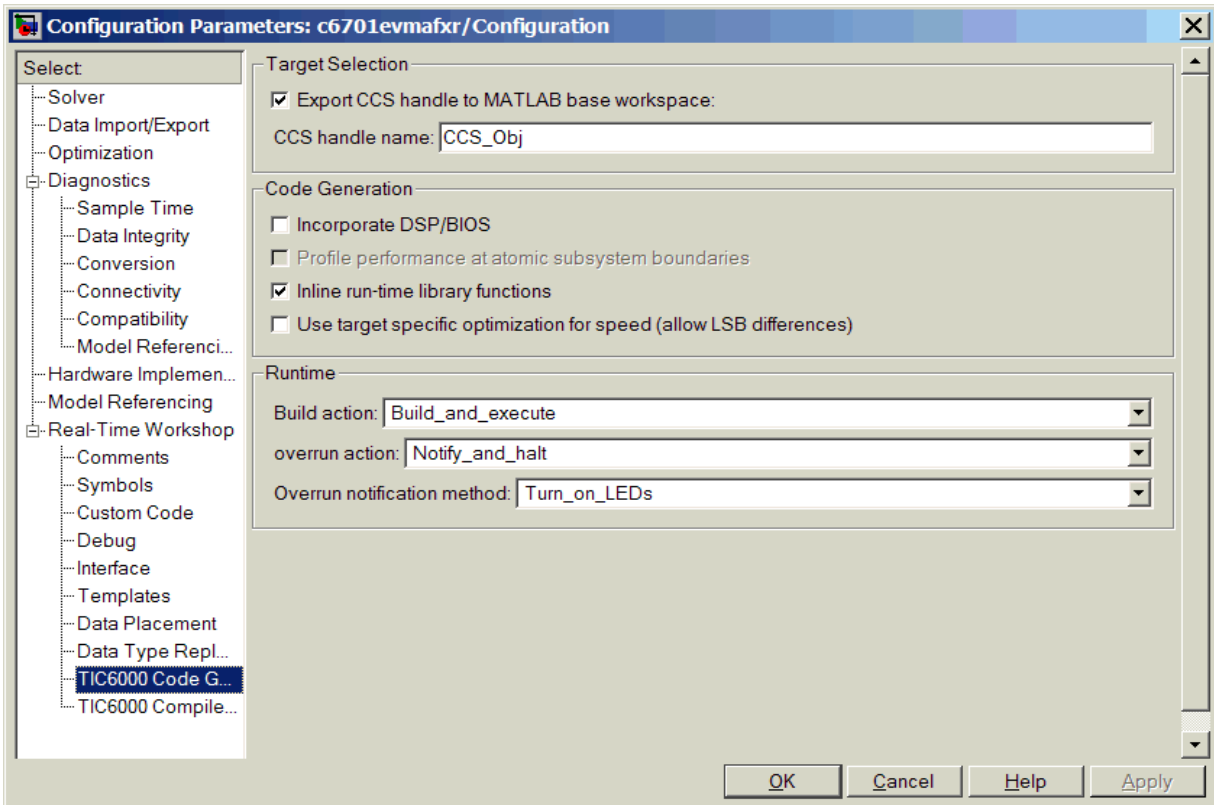
When you inline a block function, the compiler replaces each call to a block function with the equivalent function code from the static run-time library. If your model use the same block four times, your generated code contains four copies of the function. While this redundancy uses more memory, inline functions run more quickly than calls to the functions outside the generated code.

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

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

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

RunTime Options

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

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

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

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

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

`Generate_code_only` creates a file named `model.bat`—an MS-DOS batch file that contains the TI C6000 compiler command line (`cl6x`) you use to

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

The build process for a model also generates the files `modelname.c`, `modelname.cmd`, `modelname.bld`, and many others. It puts the files in a build directory named `modelname_c6000_rtw` in your MATLAB working directory. This file set contains many of the same files that Real-Time Workshop generates to populate a CCS project when you choose `Create_CCS_Project` for the build action.

- `Create_CCS_Project`—directs Real-Time Workshop to start CCS and populate a new project with the files from the build process. This option offers a convenient way to build projects in CCS.

- `Archive_CCS_Library`—directs Real-Time Workshop to archive the project for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your CCS projects for models that you use in model referencing.

- `Build`—builds the executable COFF file, but does not download the file to the target.

- `Build_and_execute`—directs Real-Time Workshop to download and run your generated code as an executable on your target.

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

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

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

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

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

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

- `Print_message`—when your application overruns, and **Overrun action** is `Notify_and_continue` or `Notify_and_halt`, the software prints a message to the standard output or the message log (for DSP/BIOS enabled projects).
- `Turn_on_LEDs`—when your application overruns, and **Overrun action** is `Notify_and_continue` or `Notify_and_halt`, the software turns on the 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 might not be able to determine whether the LED block enabled the external LED or user-defined LEDs, or an overrun condition caused the LEDs to light. Other target boards might not provide LEDs and the LED option does not apply.
- `Print_message_and_turn_on_LEDs`—in an overrun situation where you have selected a notification action for `Overrun action`, the software prints a message and turns on the LEDs. The same rules apply as for the individual notification actions.

Overrun Indicator and Software-Based Timer

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

- In multirate systems where the rate in the model is not the same as the base clock rate for your model. 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.

TI C6000 Compiler/Linker Options

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

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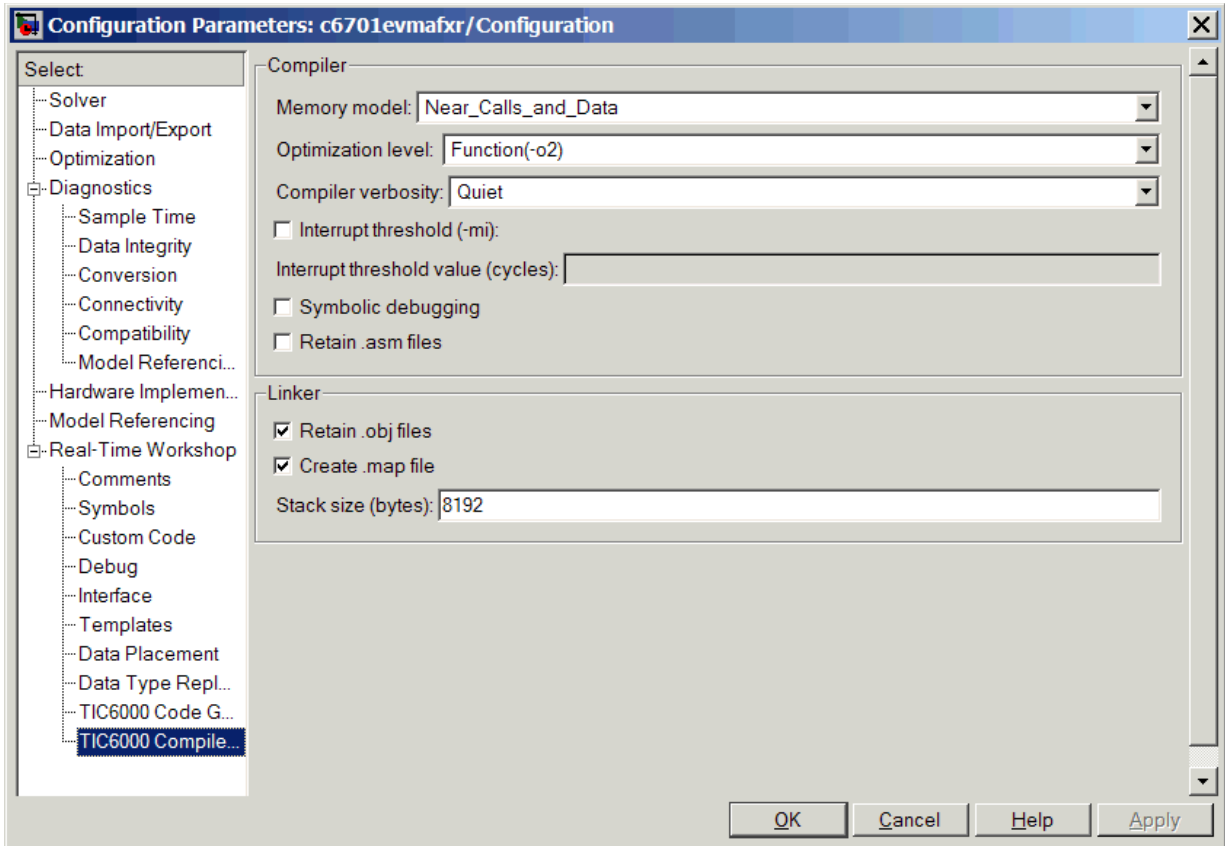
Compiler Options

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

Your selections for the memory model options affect how the Embedded Target for TI C6000 DSP handles *near* and *far* data 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. Accessing near data requires only one operation; far data requires more operations. As a consequence, programs and code that use far data run more

slowly. You should refer to your CCS documentation for details about near and far data and near and far function calls.

- `Near_Calls_and_Data`—tells the compiler to allocate calls and data as near calls. This is the default behavior.
- `Far_Aggregate_Data`—tells the compiler to allocate aggregate data, like arrays and structures, as far calls.
- `Far_Calls`—tells the compiler to allocate calls/functions as far calls.
- `Far_Calls_and_Aggregate_Data`—tells the compiler to allocate both calls and aggregate data as far calls.
- `Far_Calls_and_Data`—tells the compiler to allocate both calls and data as far calls.

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

If you select `Near_Calls_and_Data`, 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 does not fit in internal memory. To eliminate these errors, select `Far_Calls_and_Data` from the **Memory model** list.

Use the `Far_Calls_and_Data` selection when either or both of the following conditions are true:

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

2-73

In the two instances above, if you select `Near_Calls_and_Data` but your program meets the conditions specified, the TI linker issues the error message shown earlier.

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 `-mln` option in **Memory models** in the project build configuration in CCS). For more information on the `-mln` option, refer to your CCS documentation.

You can avoid the allocation error by selecting `Far_Calls_and_Data` for the **Memory model** 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 might 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.

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

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

- `Verbose`—returns all compiler messages
- `Quiet`—suppresses compiler progress messages
- `Super Quiet`—suppresses all compiler messages

Interrupt threshold (-mi). **Interrupt threshold (-mi)** enables an interrupt threshold that defines the maximum number of cycles over which the compiler

can disable interrupts. You can use this option in various ways by setting combinations for **Interrupt threshold** and **Interrupt threshold value**.

Interrupt Threshold Option	Interrupt Threshold Value (Cycles)	Effect of the Option Settings
Cleared	Not Applicable	(Default) Interrupts are explicitly disabled around software pipelined loops. The remainder of the code is interruptible.
Selected	Empty	Compiler assumes the code is never interrupted during execution.
Selected	1	Compiler generates fully interruptible code.
Selected	Integer = 2 or greater	The compiler analyzes the code for each loop and determines the maximum number of cycles required to execute the loop. If the maximum number of cycles is less than the threshold value, the compiler generates the optimal or fastest version of the loop. When the maximum is larger than the threshold, the compiler generates an interruptible loop that still generates correct output. In most cases this reduces the performance of the loop.

Using the Interrupt Threshold Option and Setting a Threshold Value

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

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

2-75

interrupts are disabled for 1 ms or longer, your profiling measurements will contain timing errors.

This profiling error condition can arise if you use the compiler optimization level -o2 or higher, because software pipelining causes interrupts to be disabled during pipelined loops. Also, some C62x/C64x DSP library assembly routines disable interrupts during execution. If your profiling measurements are affected by this interaction, measured and reported execution times might be shorter than the real execution times. To resolve this issue, consider one of the following recommendations:

- Generate fully interruptible code by selecting the **Interrupt threshold** option and entering 1 for the **Interrupt threshold value**. This disables software pipelining and lowers the execution speed of your model. If your model still runs without changing the desired sampling rate, this might be a good option to maintain your profiling.

- Avoid disabling interrupts for more than 1 ms. This ensures that the clock manager can always interrupt the code at 1 KHz (1 ms intervals). Remember that you need to translate 1 ms to cycles to enter a threshold value. The value in cycles varies depending on your CPU clock speed (value in cycles = 1/1000 x CPU speed (in cycles/second)). Use the calculated cycles value to set the **Interrupt threshold value**.

• If your program uses -o2 level compiler optimization (which is the default optimization level), setting the threshold interrupt value can ensure that critical interrupts happen as you expect. Using -o2 optimization mostly adds the benefit of software pipelined loops around which interrupts are disabled. When your application has time-critical interrupts that must be serviced, or in situations where preemption occurs, such as in multitasking, multirate models, disabling interrupts could cause problems. Consider the following techniques to resolve this problem.

- Generate fully interruptible code by selecting the **Interrupt threshold** option and entering 1 for the **Interrupt threshold value**. Setting 1 as the interrupt value prevents the compiler from disabling interrupts for more than one cycle—essentially allowing any interrupt to occur.

- Select **Interrupt threshold** and set **Interrupt threshold value** to something close to the base sampling time of your model. To determine the

recommended threshold value, use the base sampling time of your model and the CPU clock speed as shown here:

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

This value usually produces good results with -o2 optimization.

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

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

- Value = 0. A 0 value (or empty) for **Interrupt threshold value** disables the threshold and the compiler assumes the code is never interrupted.

- Value = 1. A 1 for the threshold value means the code is fully interruptible. The clock/interrupt service can interrupt any code segment at any time.

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

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

Linker Options

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 allocates code and data into memory on the target. Refer to Texas Instruments *TMS320C000 Optimizing C Compiler User's Guide* and to the online help in CCS for more details about using memory maps on TI processors.

Retain .obj files. The linker uses object (`.obj` extension) files to generate a single executable common object file format (COFF) file that you run on the 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.

Stack Size (Bytes)

You can enter any stack size in bytes, using decimal format. For more information about the stack and block output, refer to **Enable local block outputs** in the **Code Generation** options in the `Optimization` category in the **Select** tree. Also refer to the online Help system for more information about Real-Time Workshop options for configuring and building models and generating code.

Embedded Target for TI C6000 DSP Default Project Configuration—custom_MW

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

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

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

Default Compiler Build Options in custom_MW

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

For the compiler options, custom_MW uses the Function(-o2) compiler setting. The CCS default Release configuration uses File(-o3), a slightly more aggressive optimization model. For memory configuration, where Release uses the default memory model that specifies near functions and data, custom_MW specifies near functions and data—the -ml1 memory model— because some custom hardware might not support far calls or data or aggregate data. Your CCS documentation provides complete details on the compiler build options.

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

Model Reference and Embedded Target for TI C6000 DSP

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

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

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

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, here are some terms you need to know.

- Top model—the root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- Referenced models—blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections present a brief description of how model reference works. More details are available in your Real-Time Workshop documentation in the online Help system.

Model Reference in Simulation

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

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

Model Reference in Code Generation

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

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

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

Using Model Reference with Embedded Target for TI C6000 DSP

With few limitations or restrictions, Embedded Target for TI C6000 DSP provides full support for generating code from models that use model reference.

Build Action Setting

The most important requirement for using model reference with the TI targets is that you must set the **Build action** (go to **Configuration Parameters >TIC6000 Code Generation**) for all models referred to in the simulation to `Archive_CCS_Library`.

To set the build action

1 Open your model.

2 Select **Simulation—>Configuration Parameters** from the model menus.

The **Configuration Parameters** dailog opens.

3 From the **Select** tree, choose `TIC6000 Code Generation`.

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

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

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

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

Target Preferences Blocks in Reference Models

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

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

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

Other Block Limitations

Model reference with Embedded Target for TI C6000 DSP does not allow you to use certain blocks or S-functions in reference models:

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

- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Embedded Target for TI C6000 DSP library.

Targeting Your C6701 EVM and Other Hardware

Texas Instruments markets a complete set of tools for you to 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.

Typical Targeting Process

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

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

2 Add target-specific blocks to your model, such as ADC and DAC blocks, and configure the block parameters.

3 Add a target preferences block to your model. Select the block that best matches your target—one of the device specific blocks, like C6711 DSK, or the Custom C6000 block when none of the specific blocks is appropriate. All models that you target to a C6000-processor-based hardware must have a target preferences block at the top level of the model.

4 Configure the options on the target preferences block to select the target, map memory segments, allocate sections to the memory segments, and configure other target-specific options.

5 Set the configuration parameters for your model. Notice that you do this step after you add the target preferences block to your model.

6 Build your model to your target.

Targeting the C6701 Evaluation Module

After you install the C6701 EVM development board and supporting TI products on your PC, start MATLAB. At the MATLAB command prompt, enter `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 LEDs 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 blocks to your model.

With your model open, select **Simulation -> Configuration Parameters**. From this dialog, select `Real-Time Workshop` from the **Select** tree. 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

- **Real-Time Workshop system target file**—ti_c6000.tlc
- **Template makefile**—ti_c6ooo.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 inlines S-functions. Inlined S-functions offer speed advantages and simplify the generated code. For more information about inlining S-functions, refer to Target Language Compiler Reference documentation. For a complete discussion of S-functions, refer to your Writing S-Functions documentation.

During the same build operation, the template makefile and block parameter dialog entries are combined to form the target makefile for your TI evaluation module. This makefile invokes the TI compiler to build an executable file. If you select the `Build_and_execute` option, Real-Time Workshop automatically downloads the executable to the TI evaluation board via the peripheral component interface (PCI) bus. After downloading the executable file to the 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 process encounters a Stop block in the model code.
- The running application encounters an error condition that stops the process.

If you included a Reset 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)
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 configuration shown in the preceding table, reconfigure your C6701 EVM. After you change your board configuration, rerun the verification utility to check your new settings.

Testing Your 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.
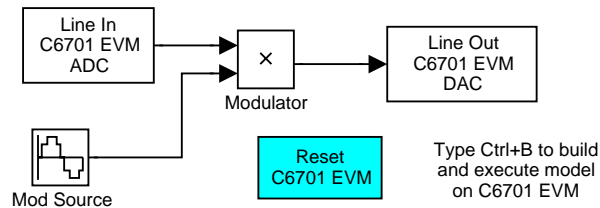
1 Enter c6701evmtest at the MATLAB command prompt.

The model opens in Simulink.

2 Select **Configuration 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 **Configuration Parameters** option selected.
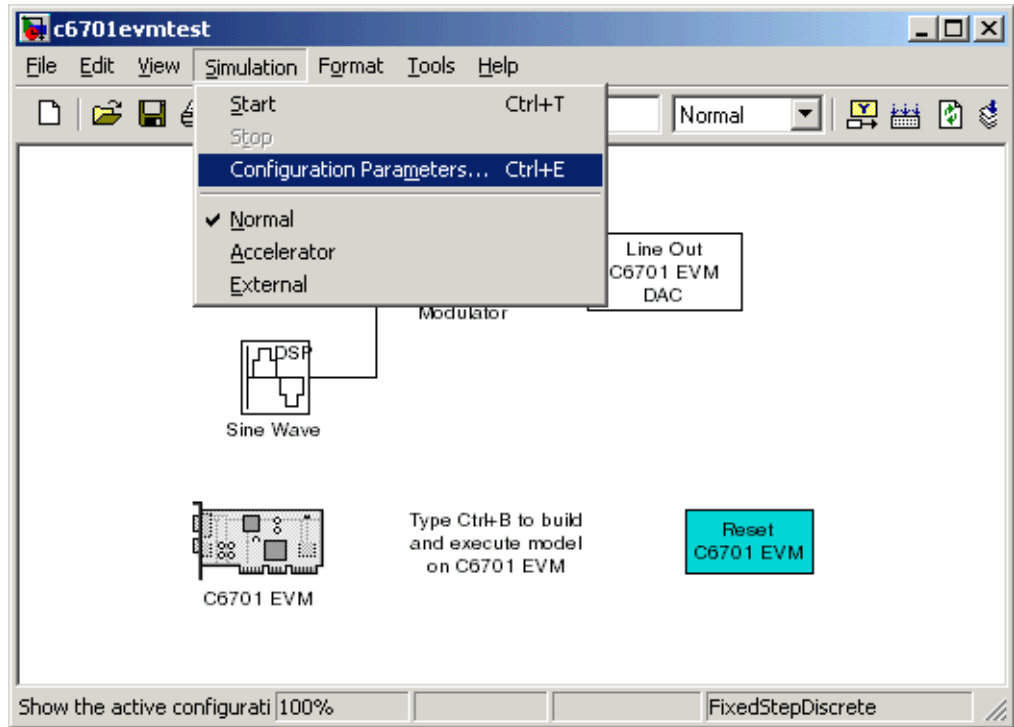
Figure 2-2: Using c6701evmtest to Test Your Embedded Target for TI C6000 DSP Installation

3 On the **Configuration Parameters** dialog, click Real-Time Workshop in the **Select** tree to view the Real-Time Workshop pane.

4 Click **Build** to run the model. Building the model provides a comprehensive test of the build, download, and run processes in the Embedded Target for TI C6000 DSP.

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

```
### Starting Real-Time Workshop build procedure for model:
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:O
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 Messages While Building C6701evmtest

If you receive an error message from the build and compile process, your board or the software may not be configured correctly. Reinstall the board and review the configurations listed in "Configuring Your C6701 EVM" on page 2-87. 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 C6701evmtest Is Running

To see that the model is running, turn on your signal generator and set the output to produce a sine wave at 8000 Hz. Connect your oscilloscope to display both the input signal from the signal generator and the output from your 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 C6701evmtest 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 Signal Processing Blockset
- Use blocks from the fixed-point blocks library TI C62x DSPLIB or TI C64x DSPLIB
- Use other Simulink discrete-time blocks
- Use the blocks provided in the C6000 blockset: ADC, DAC, LED and Reset blocks for specific supported target hardware
- Use blocks that provide the functions you need from any blockset installed on your computer
- Create and use custom blocks

Once you have designed and built your model, you generate C code and build the real-time executable by clicking **Build** on the **Real-Time Workshop** pane

of the **Configuration Parameters** dialog. 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.

Block Name/Category	Library	Description
Scope	Simulink, Signal Processing Blockset	Provides oscilloscope view of your output. Do not use the **Save data to workspace** option on the **Data history** pane in the **'Scope' parameters** dialog.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.

Block Name/Category	Library	Description
Spectrum Scope	Signal Processing Blockset	Compute and display the short-time FFT of a signal. It has internal buffering that can slow your process without adding value.
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	Signal Processing Blockset	Send data to your MATLAB workspace.
Signal To Workspace	Signal Processing Blockset	Send a signal to your MATLAB workspace.
Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
To Wave device	Signal Processing Blockset	Send data to a `.wav` device.
From Wave device	Signal Processing Blockset	Get data from a `.wav` device.
To Wave file	Signal Processing Blockset	Send data to a `.wav` file.
From Wave file	Signal Processing Blockset	Get data from a `.wav` file.

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

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

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

C6701 EVM Tutorial—Single Rate Application

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

You can choose to create the Simulink model for this tutorial from blocks in Signal Processing Blockset and Simulink block libraries, or you can find the model in the Embedded Target for TI C6000 DSP demos. For this example, we show the model as it appears in the demonstration program. The demonstration model name is `c6701evmafxr.mdl` as shown in the next figure. Open this model by entering `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, complete the following tasks:

1 Use Simulink blocks, Signal Processing Blockset blocks, and blocks from other blocksets to create your model application.

2 Add the Embedded Target for TI C6000 DSP blocks that let your signal sources and output devices communicate with your 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 Add the C6701EVM target preferences block from the C6000 Target Preferences library to your model. Verify and set the block parameters for your hardware. In most cases, the default settings work fine.

   If you are using a C6701 simulator target, select **Simulator** on the **Board info** pane of the target preferences block.

4 Set the configuration parameters for your model, including

   - Solver parameters such as simulation start and stop time and solver options

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

5 Build your model to the selected target.

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

Your target for this tutorial is your C6701 EVM installed on your PC. Be sure to configure and test your board as directed in "Configuring Your C6701 EVM" on page 2-87 in this guide before continuing this tutorial.

Building the Audio Reverberation Model

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

1 Start Simulink.

2 Create a new model by selecting **File -> New -> Model** from the **Simulink** menu bar.

3 Use Simulink blocks and Signal Processing Blockset blocks to create the following model.
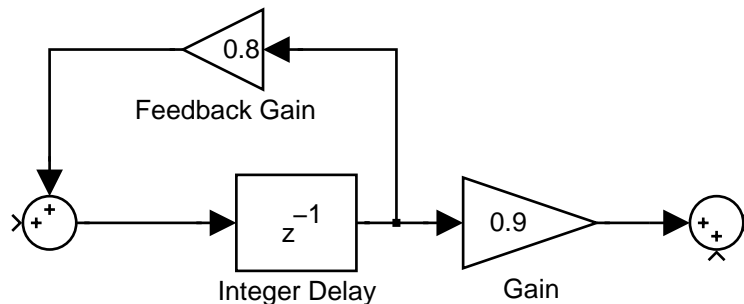
Look for the Integer Delay block in the Signal Operations library of the Signal Processing Blockset. You do not need to add the input and output signal lines at this time. When you add the C6701 EVM blocks in the next section, you add the input and output to the sum blocks.

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)

Entering c6701evmlib at the MATLAB prompt opens this window showing the library blocks.
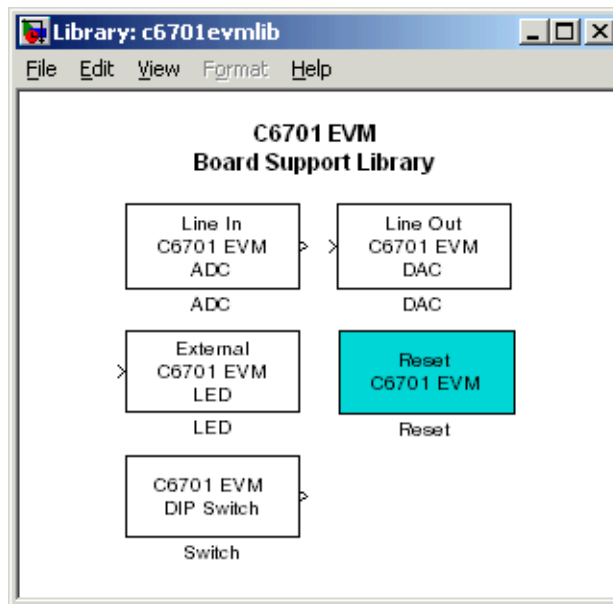
Embedded Target for TI C6000 DSP Target for C6701 EVM Block Library C6701evmlib

This block library is 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 generate code that configures 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.
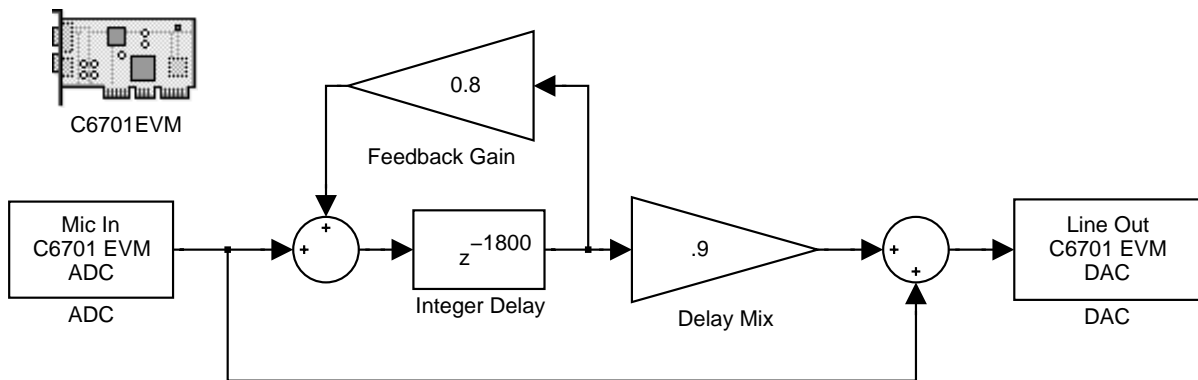
4 Connect new signal lines as shown in the figure.

5 Finally, from the TI C6000 Target Preferences block library, add the C6701EVM target preferences block to the model. Notice that it is not connected to any other block in the model.

Configuring the Embedded Target for TI C6000 DSP Blocks

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

1 Click the 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.

[1] 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.

8 Click the C6701EVM target preferences block.

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

10 Verify the parameter settings for the C6701 EVM target. The figures below show the proper values.
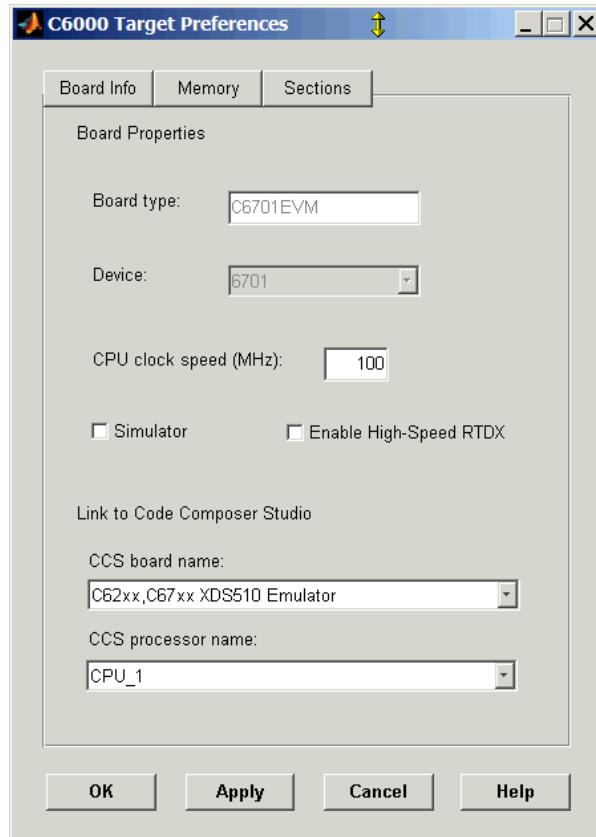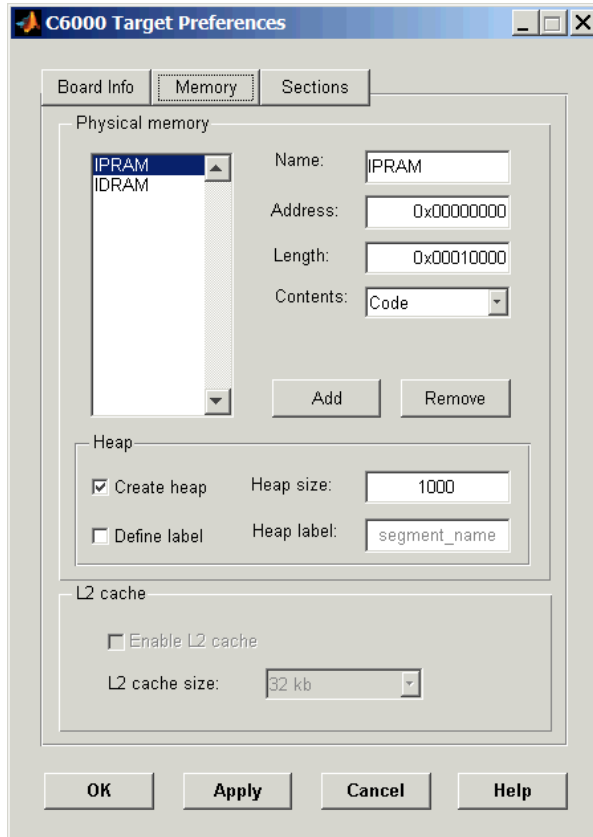
Board info Settings

Memory Settings

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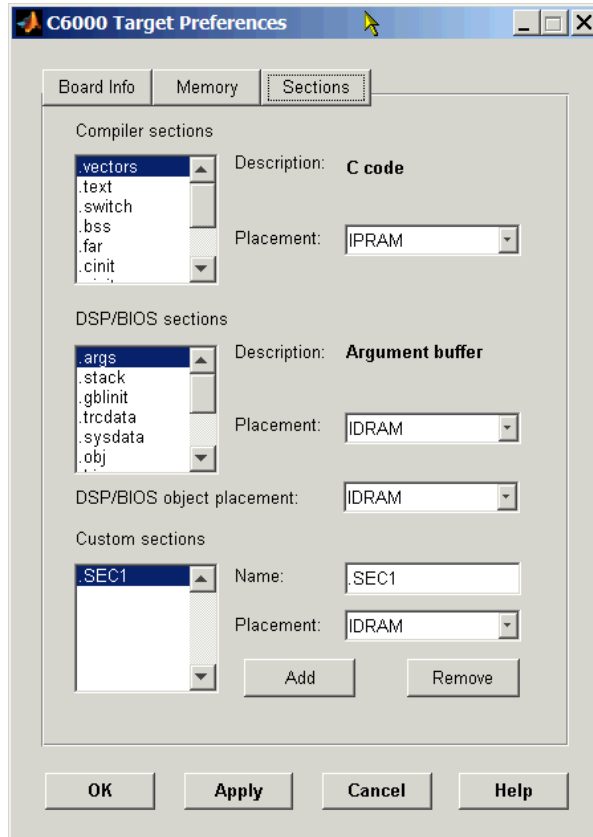
You have completed the model. Now configure the Real-Time Workshop options to build and download your new model to your C6701 EVM.

Specifying Configuration 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 **Configuration Parameters** dialog in Simulink.

2-103

Setting Simulink Configuration Parameters

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

1 Open the **Configuration Parameters** dialog and set the appropriate options on the Solver category for your model and for the Embedded Target for TI C6000 DSP.

- Set **Start time** to 0.0 and **Stop time** to inf (model runs without stopping). Generated code does not honor this setting if you set a stop time. Set this to inf for completeness.
- Under **Solver options**, select the fixed-step and discrete settings from the lists
- Set the **Fixed step size** to Auto and the **Tasking Mode** to Single Tasking

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

Setting Real-Time Workshop Target Build Options

To configure Real-Time Workshop to use the correct target files and to compile and run your model executable file, you set the options in the Real-Time Workshop category of the **Configuration Parameters** dialog. Follow these steps to set the Real-Time Workshop options to target your C6701 EVM:

1 Select Real-Time Workshop on the **Select** tree.

2 In Target selection, click **Browse** to select the system target file for C6000 targets—ti_c6000.tlc. It may already be the selected target.

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

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

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

4 From the **Select** tree, choose TI C6000 code generation to specify code generation options that apply to the C6701 EVM target.

5 Under **Code Generation**, select the **Inline Signal Processing Blockset functions** option. Clear the other options.

6 Under **Target Selection**, verify that **Export CCS handle to MATLAB base workspace** is selected and provide a name for the handle (optional).

7 Select `TI C6000 Compiler/Linker` on the **Select** tree to set the compiler options.

8 Set the following options in the dialog under **Compiler**:

- **Optimization level** should be `Function (-o2)`.
- Set **Compiler verbosity** to `Quiet`.

Clear the other options under **Compiler**.

9 Set the linker operation options by selecting the **Retain .obj files** check box.

10 Change the category on the **Select** tree to `Hardware Implementation`.

11 Set **Byte ordering** to `Little endian`.

12 Change the category again to `TI C6000 Code Generation`.

13 Set the following Real-Time Workshop run-time options:

- **Build action**: `Build_and_execute`.
- **Overrun action**: `Notify_and_halt`.
- **Overrun notification method**: `Turn_on_LEDs`.

You have configured the Real-Time Workshop options that let you target your C6701 EVM. You may have noticed that you did not configure a few Real-Time Workshop categories on the **Select** tree, such as `Comments`, `Symbols`, and `Optimization`.

For your new model, the default values for the options in these categories are correct. For other models you develop, you may want to set the options in these categories to provide information during the build and to run TLC debugging when you generate code.

Building and Executing Your Model on Your C6701 EVM

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

1 Change the category to `Real-Time Workshop` on the **Configuration Parameters** dialog.

2 Clear **Generate code only** and click **Build** to generate and build an executable file targeted to your C6701 EVM.

When you click **Build** with `Build_and_execute` selected for **Build action**, the automatic build process creates an executable file that can be run by the C6701 DSP on your C6701 EVM, and then downloads the executable file to the target and runs the file.

3 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—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 c6701evmwdnoisf. As with any model file, you can run this denoising demonstration by typing c6701evmwdnoisf at the MATLAB prompt. The model also appears in the MATLAB demos collection in the Help browser—under Simulink demos, in the Embedded Target for TI C6000 DSP category. Here is a picture of the model as it appears in the demonstration library.
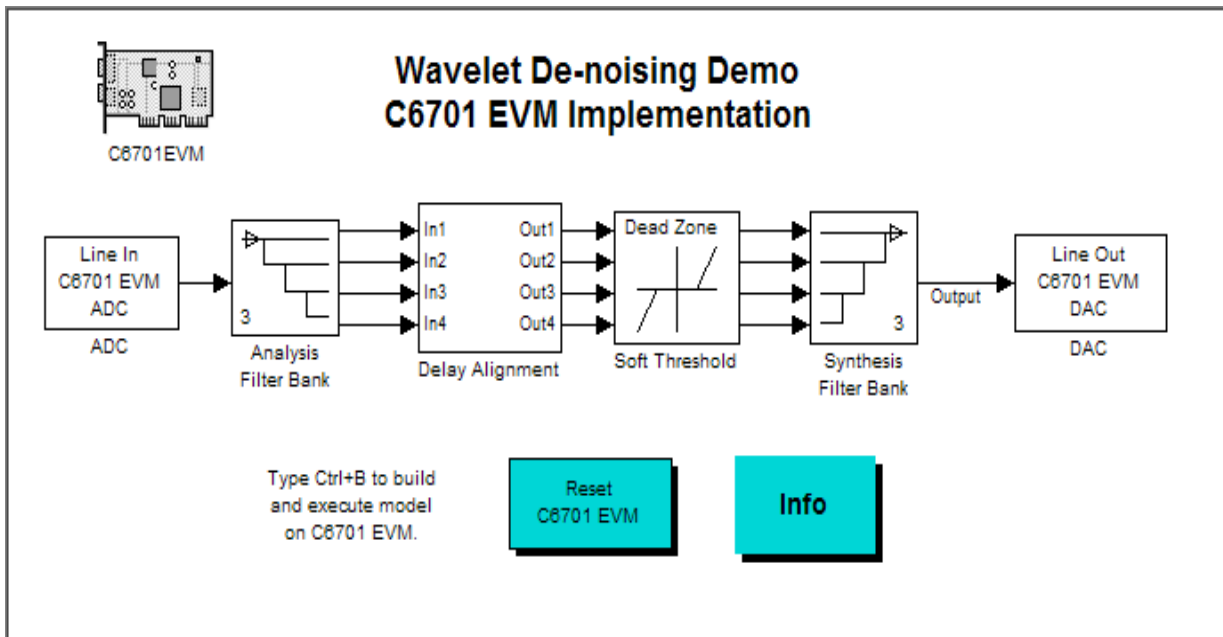
Unlike the audio reverberation demo, this model is difficult to build from blocks in Simulink. It uses complex subsystems for the Delay Alignment block and the Soft Threshold block. For this tutorial you work with a copy of the demonstration model, rather than creating the model.

This tutorial takes you through generating C code and building an executable program from the demonstration model. The resulting program runs on your C6701 EVM as an executable COFF file.

2-107

Working and Build Directories

It is convenient to work with a local copy of the c6701evmwdnoisf model, stored in its own directory, which you named (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 configuration parameters. In particular, Real-Time Workshop uses a fixed-step solver. To set the parameters, use the **Configuration Parameters** dialog as follows:

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

2 Click `Solver` and enter the following parameter values on the **Solver** pane. Note that Embedded Target for TI C6000 DSP does not honor a stop time if you set one here.

   Start Time: `0.0`

   Stop Time: `inf`

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

   Fixed step size: `auto`

   Tasking mode for periodic sample times: `Auto`

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

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

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

Configuration Parameters: c6701evmwdnoisf/Configuration

Select:
- **Solver**
- Data Import/Export
- Optimization
- Diagnostics
  - Sample Time
  - Data Integrity
  - Conversion
  - Connectivity
  - Compatibility
  - Model Referencing
- Hardware Implement...
- Model Referencing
- Real-Time Workshop
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface
  - Templates
  - Data Placement
  - Data Type Replac...
  - TIC6000 Code Ge...
  - TIC6000 Compiler/...

Simulation time

Start time: 0.0   Stop time: inf

Solver options

Type: Fixed-step   Solver: discrete (no continuous states)

Periodic sample time constraint: Unconstrained

Fixed-step size (fundamental sample time): auto

Tasking mode for periodic sample times: Auto

☐ Higher priority value indicates higher task priority

☐ Automatically handle data transfers between tasks

[OK] [Cancel] [Help] [Apply]

Selecting the Target Configuration

To specify the desired target configuration, you choose the

- System target file
- Template makefile
- `make` command

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

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

To target your C6701 EVM:

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

2 Click `Real-Time Workshop` on the **Select** tree. The `Real-Time Workshop` pane activates.
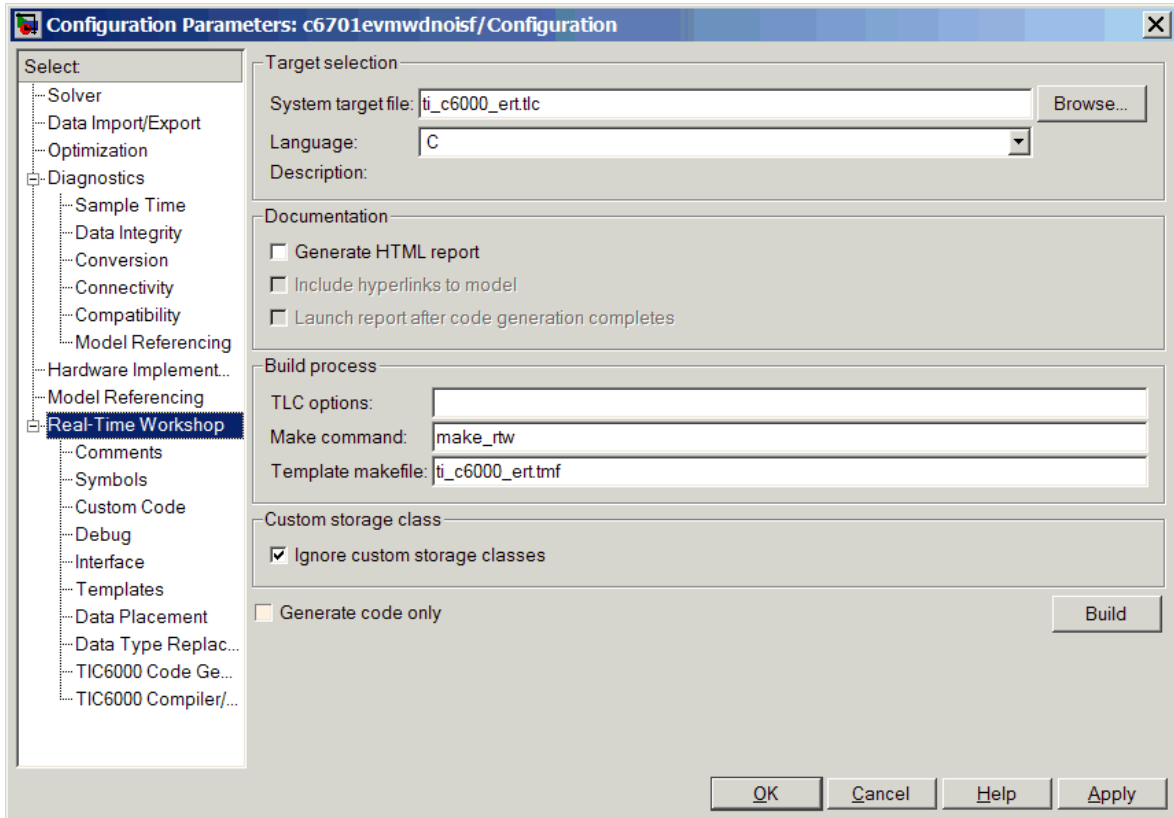
3 Click **Browse** next to the **System target file** field. This opens the **System Target File Browser**. The browser displays a list of available target configurations. When you select a target configuration, Real-Time Workshop automatically chooses the appropriate system target file, template makefile, and `make` command.
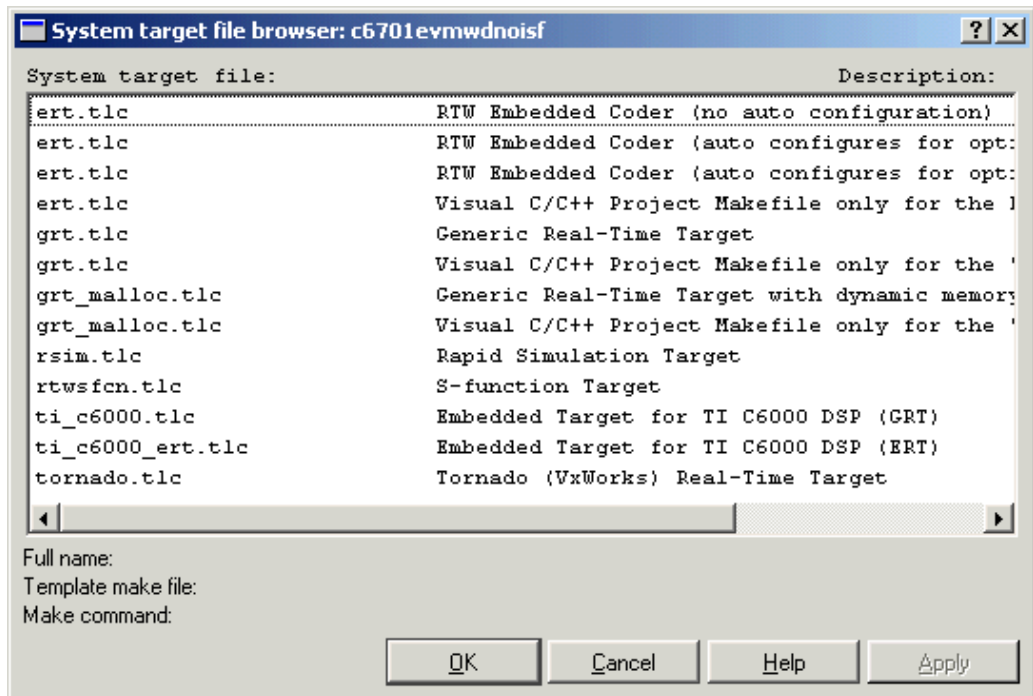
4 From the list of available configurations, select `ti_c6000.tlc` and click **OK**.

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

5 To decide whether to export a CCS handle to your MATLAB work space when you generate code, or run your model, select `TI C6000 Code Generation` from the **Select** tree.
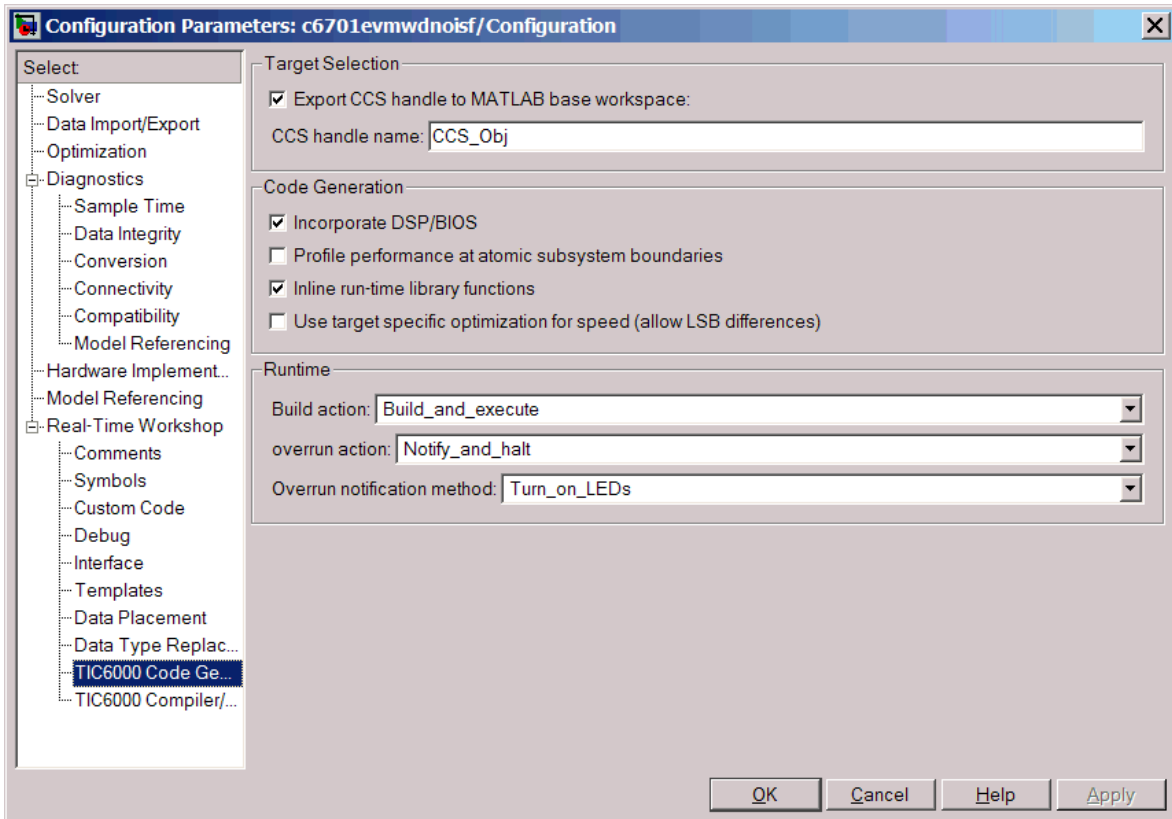
6 To export the handle (a variable) that CCS creates when you generate code from your model, select **Export CCS handle to MATLAB workspace** and enter a name for the handle in **CCS handle name**.

7 Select the **Inline Signal Processing Blockset functions** and the **Incorporate DSP/BIOS** options, as shown.

8 Select `Optimization` from the **Select** tree. A new set of options appears. The options displayed here are common to all target configurations. Make sure that all options are set to their defaults, as shown below.
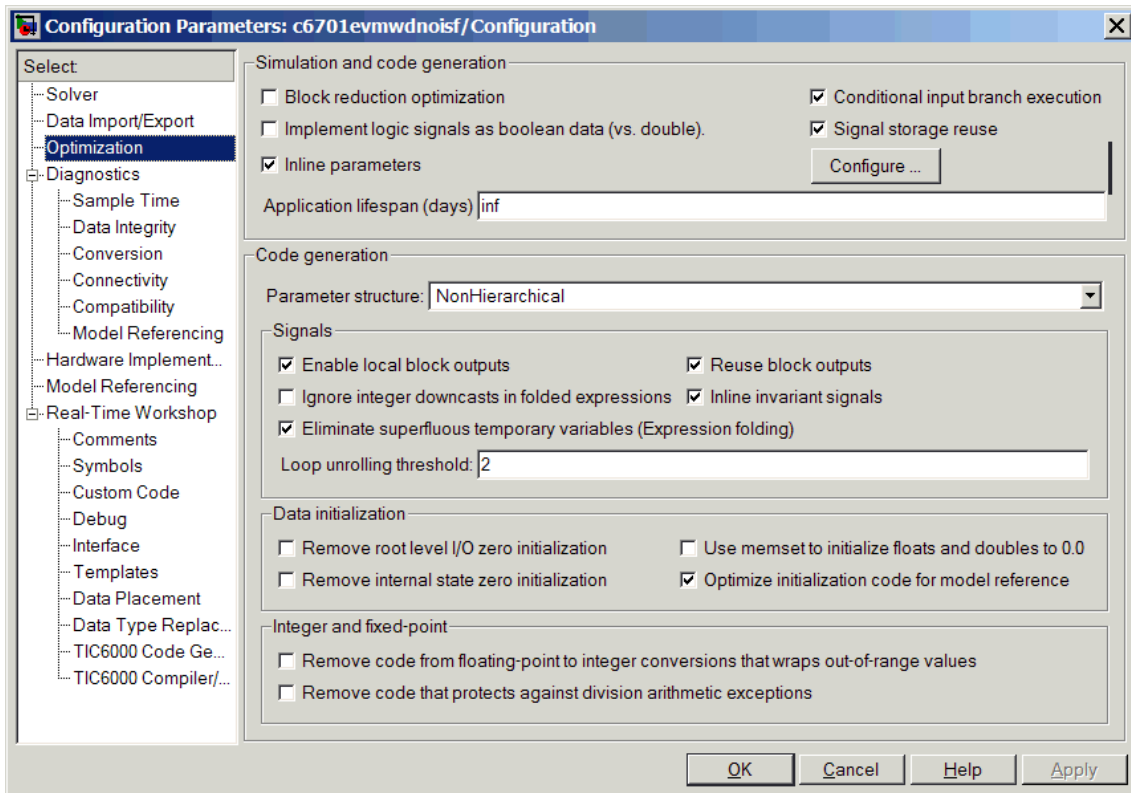
9 Select `Debug` from the **Select** tree to access the Real-Time Workshop Process and TLC process debugging options. Clear the check boxes on this pane. Check **Verbose build** to see all the messages that Real-Time Workshop issues while it generates your code or project. Selecting **Verbose build** is optional, but can be useful when you are new to the code generation process
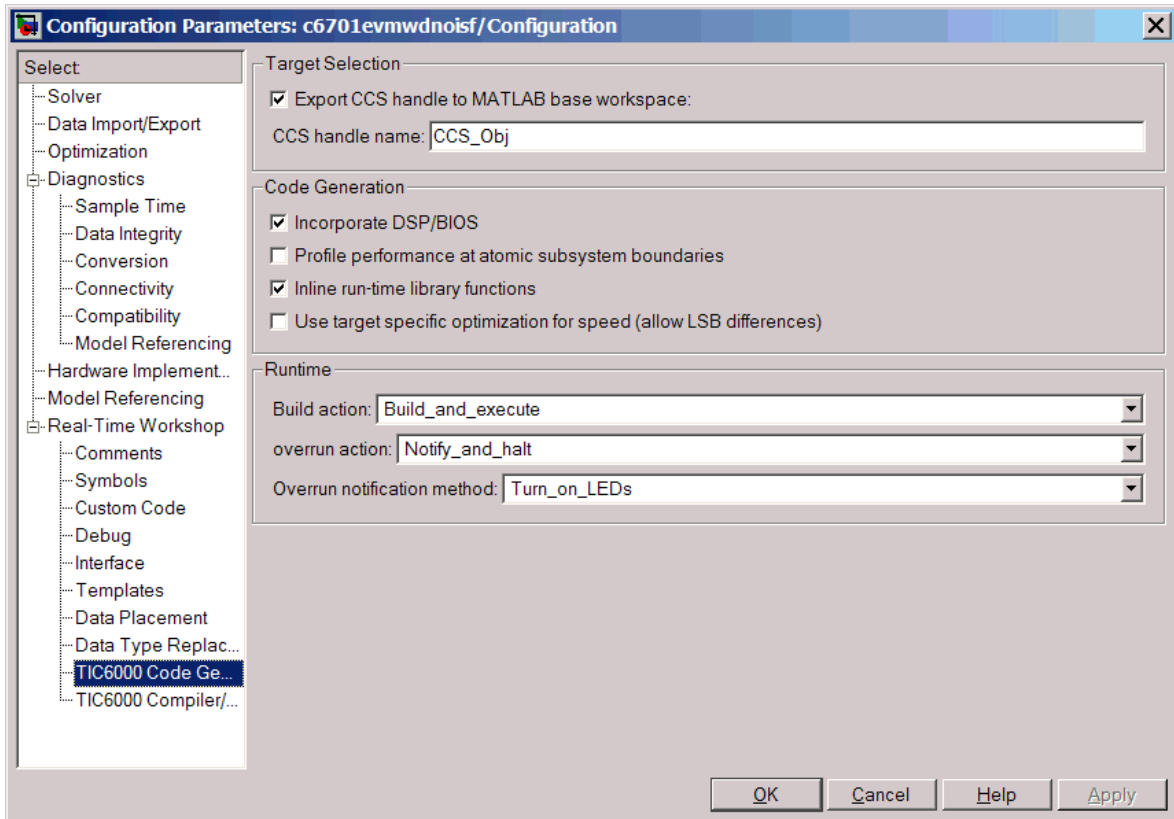
10 Select `TI C6000 Compiler/Linker` from the **Select** tree. The options displayed on the new pane are specific to the C6000 target and TI compiler. Check to make sure that the options are set as shown.
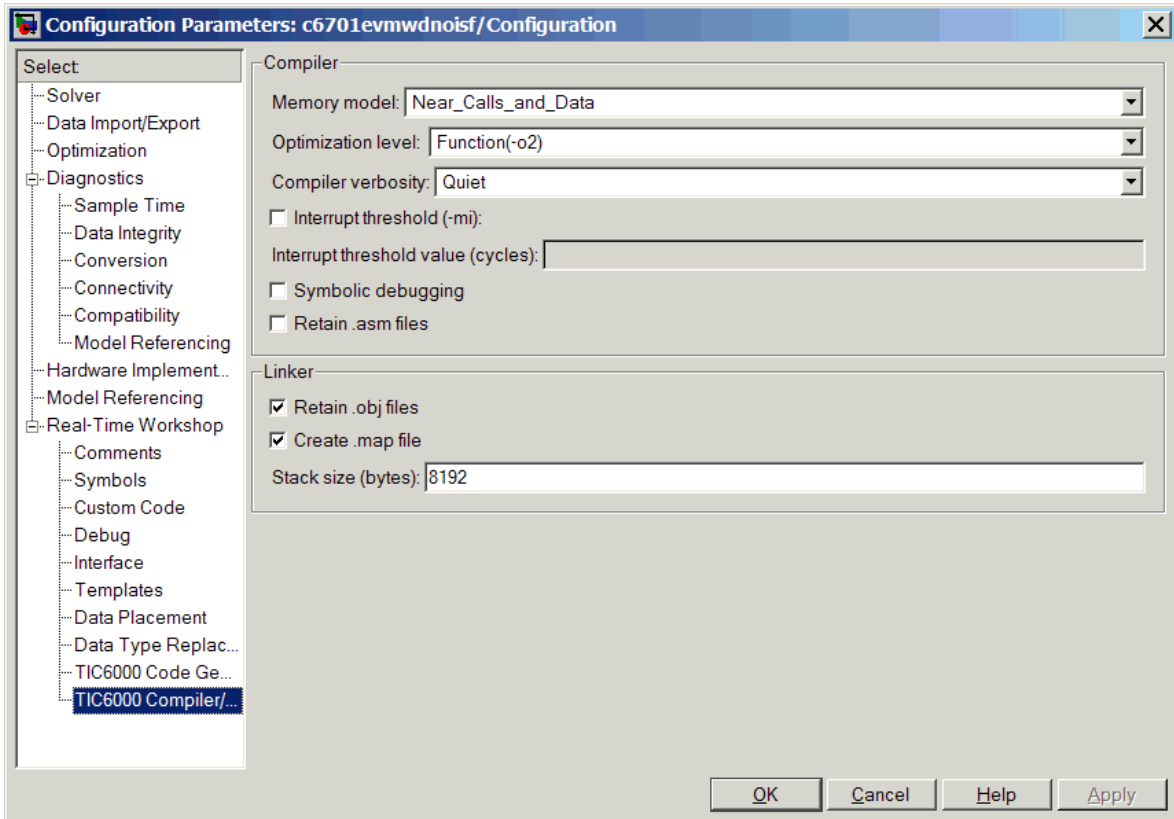
11 Select `TI C6000 Code Generation` on the **Select** tree to access the C6000 run-time options. Set the run-time options as shown.
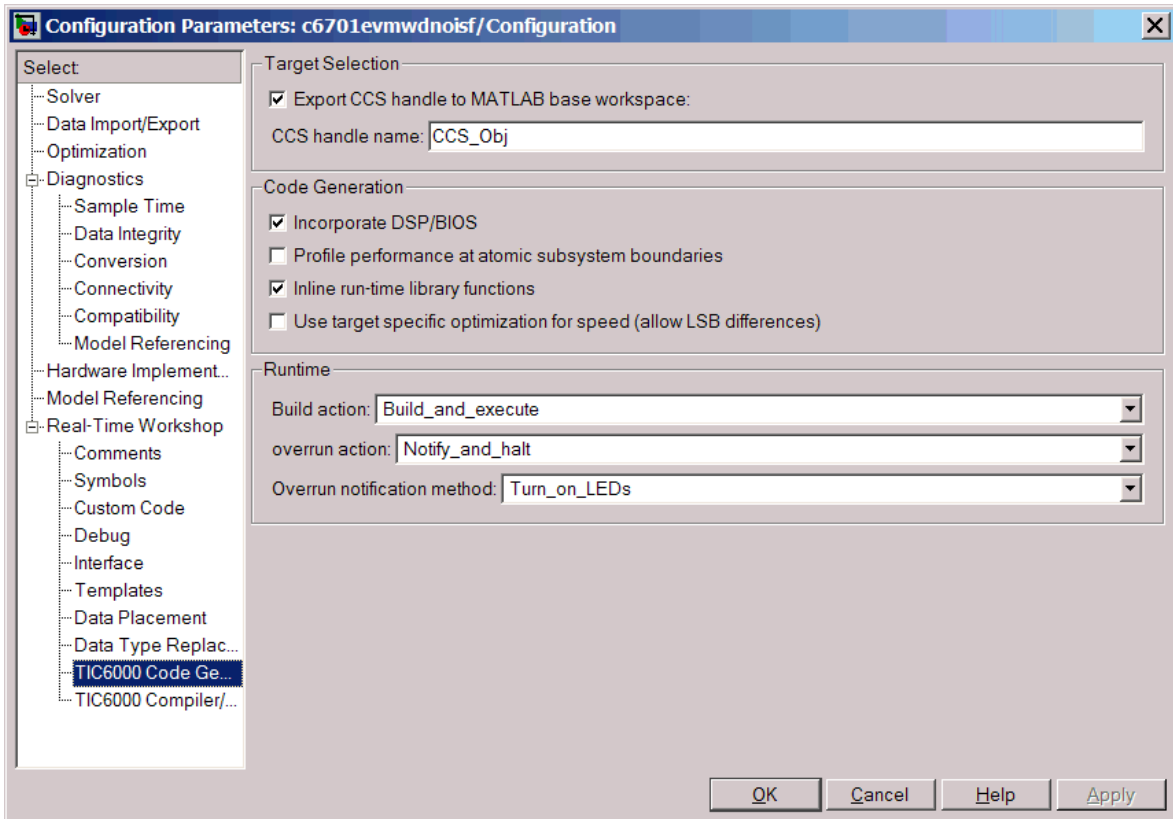
12 Click **OK** to close the **Configuration Parameters** dialog. Save the model to retain your new build settings.

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 **Configuration Parameters** dialog for your model.

2 Clear **Generate code only** and click **Build** in the Real-Time Workshop pane to start the build process.

3 A number of messages concerning code generation and compilation appear in the MATLAB Command Window. The initial messages are

```
### Starting Real-Time Workshop build procedure for model:
dnoisfrtw
### Generating code into build directory: .\dnoisfrtw_c6000_rtw
```

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

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

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

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

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

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

```
!dnoisfrtw
```

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

The program produces one line of output.

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

6 To see the contents of the build directory, type

```
dir dnoisfrtw_c6701_rtw
```

Contents of the Build Directory

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

`dnoisfrtw_c6701_rtw` contains these generated source code files:

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

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

Targeting Your C6711 DSK and Other Hardware

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

This chapter describes how to use the Embedded Target for TI C6000 DSP to create and execute applications on the C6711 DSK. To use the targeting software, you should be familiar with using Simulink to create models and with the basic concepts of Real-time Workshop automatic code generation. To read more about Real-Time Workshop, refer to your Real-Time Workshop documentation.

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

- Configuring your Embedded Target for TI C6000 DSP software, in "Real-Time Workshop Options for C6000 Hardware" on page 2-57
- Configuring your Texas Instruments TMS320C6711 DSP Starter Kit, in "Configuring Your C6711 DSK" on page 2-121
- Testing your hardware and software installation to be sure everything works, in "Confirming Your C6711 DSK Installation" on page 2-121 and "Testing Your C6711 DSK" on page 2-122

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\conftest`

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

3 Start the confidence test by typing `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.
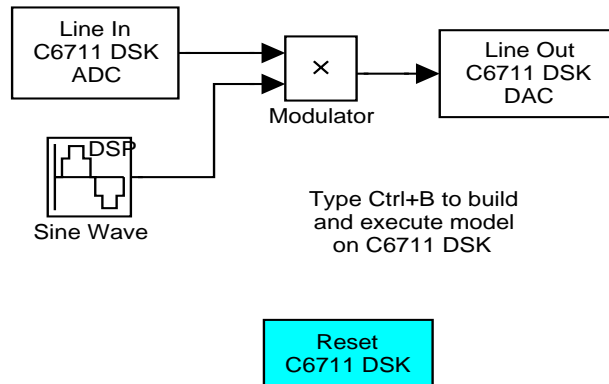
1 Enter c6711dsktest at the MATLAB command prompt.

Test model c6711dsktest opens in Simulink.

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

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

2-123
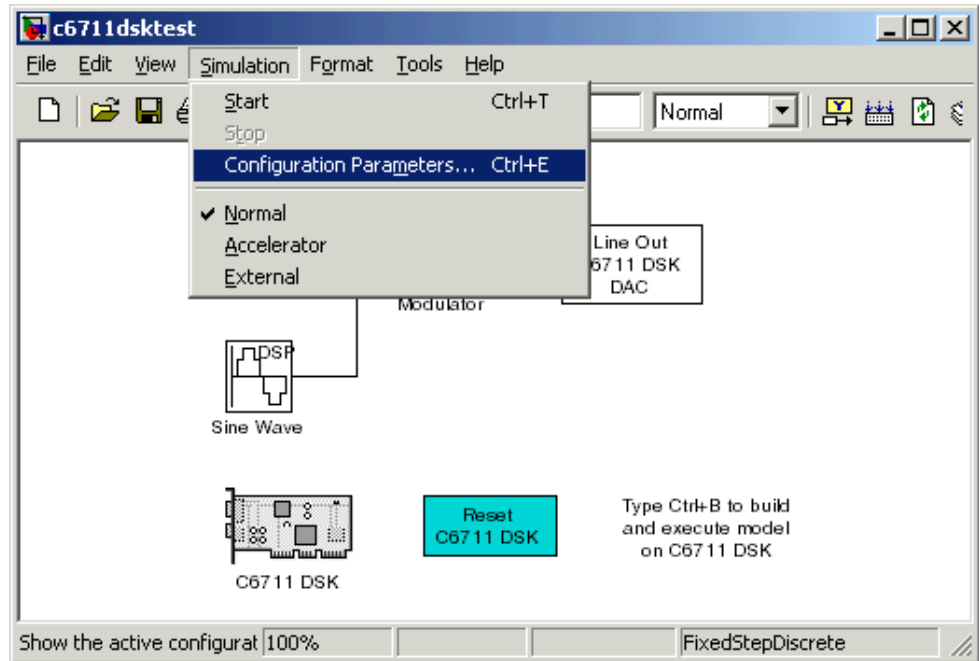
3 Click `Real-Time Workshop` in the **Select** tree on the **Configuration Parameters** dialog to view the `Real-Time Workshop` options.

4 Click **Build** to run the model. Building the model provides a comprehensive test of the build, download, and run processes in the Embedded Target for TI C6000 DSP.

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

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

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

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

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

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

Error Messages While Building c6711dsktest

If you receive an error message from the build and compile process, your board or the software may not be configured correctly. Reinstall the board and review the configurations listed in section "Configuring Your C6711 DSK" on page 2-121. You need to resolve errors that appear in this build before you start to develop and build your own models.

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

Verifying That c6711dsktest Is Running

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

Starting and Stopping c6711dsktest on the C6711 DSK

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

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

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

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

Running Models on Your C6711 DSK

Texas Instruments markets a complete set of tools for use with the C6711 DSK. These tools are primarily intended for rapid prototyping of control systems and hardware-in-the-loop applications.

This section provides a brief example of how the TI development tools work with Real-Time Workshop, the Embedded Target for TI C6000 DSP, and the C6711 DSK Board Support block library.

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

Other components, such as a communication link with Simulink, are required if you need the ability to download parameters on-the-fly to your target hardware.

Since these components are specific to particular hardware targets (in this case, the C6711 DSK), you must ensure that the target-specific components are compatible with the target hardware.

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

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

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

- C6711 DSK ADC—configures the analog to digital converter
- C6711 DSK DAC—configures the digital to analog converter
- C6711 DSK LED—controls the user-defined light emitting diodes (LED) on the C6711 DSK
- C6711 DSK DIP Switch—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 -> Configuration Parameters** from the menu bar to open the **Configuration Parameters** dialog. From this dialog, click **Real-Time Workshop** on the select tree. You must specify the appropriate versions of the system target file and template makefile. For the C6711 DSK, in the **Real-Time Workshop** pane of the dialog, specify

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

With this configuration, you can generate and download a real-time executable to your TI C6711 DSK. Start the Real-Time Workshop build process by clicking **Build** on the **Real-Time Workshop** pane. Real-Time Workshop automatically generates C code and inserts the I/O device drivers as specified by the ADC and DAC blocks in your block model.

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

During the same build operation, the template makefile and block parameter dialog 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
- Encountering a Stop block in the model.
- Clicking the C6711 DSK Reset block in your model (if you added one) or in the DSK block library

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

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

C6711 DSK Tutorial—Single Rate Application

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

You can choose to create the Simulink model for this tutorial from blocks in the Signal Processing Blockset and Simulink block libraries, or you can find the model in the Embedded Target for TI C6000 DSP demos. For this example, we show the model as it appears in the demonstration program. The demonstration model name is `c6711dskafxr.mdl` and is 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 Add the C6711 DSK target preferences block from the C6000 Target Preferences library to your model. Verify and set the block parameters for your hardware. In most cases, the default settings work fine.

If you are using a C6711 simulator target, select **Simulator** on the **Board info** pane of the target preferences block.

4 Set the configuration parameters for your model, including

- Configuration Parameters such as simulation start and stop time and solver options.

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

5 Build your model to the selected target.

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

Your target for this tutorial is your C6711 DSK installed on your PC. Be sure to configure and test your board as directed in "Configuring Your C6711 DSK" on page 2-121.

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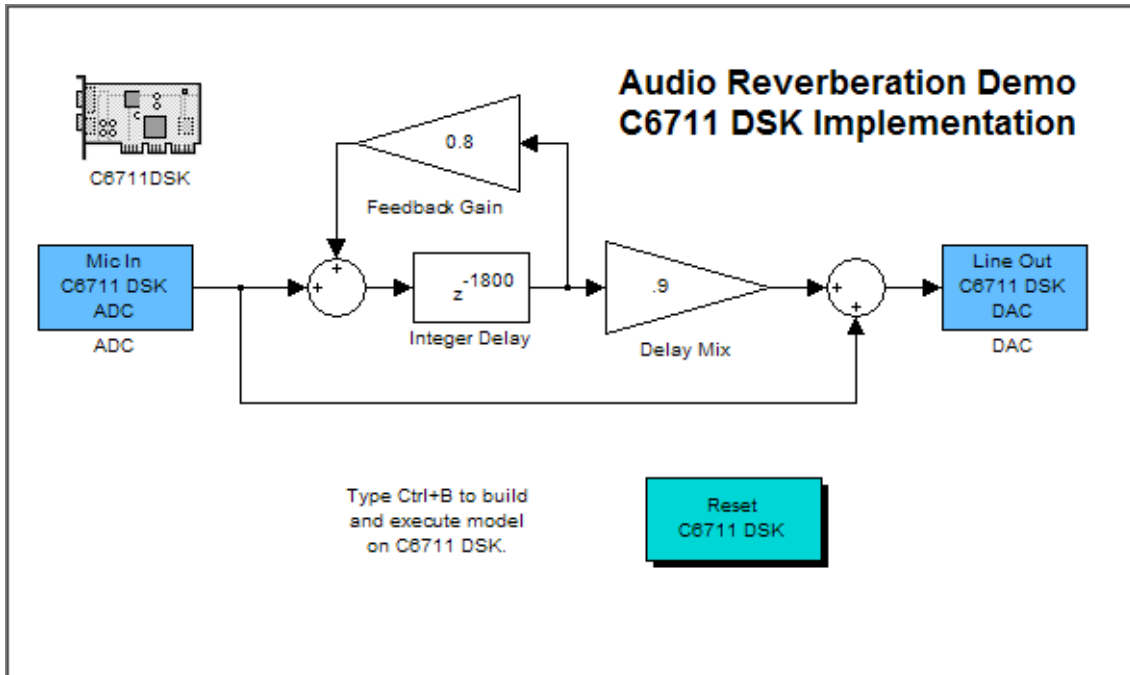
Look for the Integer Delay block in the Signal Operations library of Signal
Processing Blockset. You do not need to add the input and output signal
lines at this time. When you add the C6711 DSK blocks in the next section,
you add the input and output to the sum blocks.

4 Save your model with a suitable name before continuing.

Adding C6711 DSK Blocks to Your Model

So that you can send signals to your C6711 DSK and get signals back, 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.
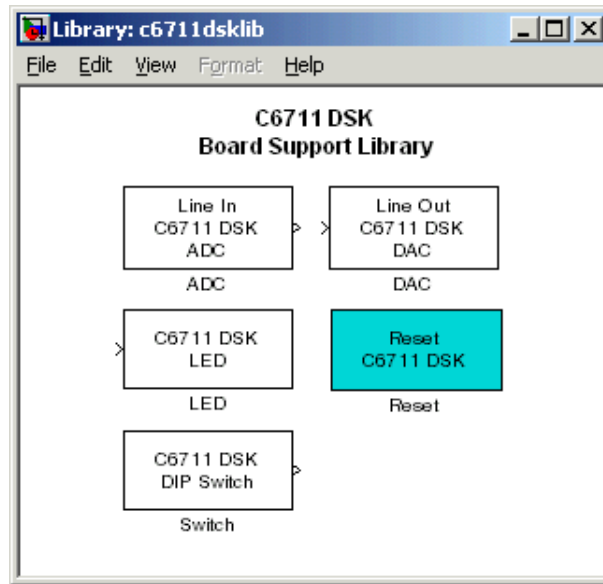
These blocks are included in the Embedded Target for TI C6000 DSP `c6000lib` blockset in the Simulink Library browser.

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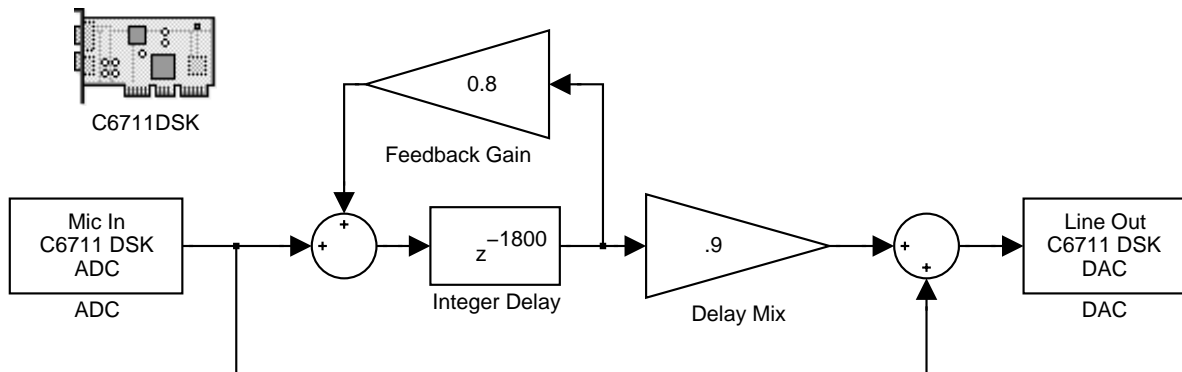
4 Connect new signal lines as shown in the figure.

5 Finally, from the TI C6000 Target Preferences block library, add the C6711DSK target preferences block. This block is not connected to any other block in the model.

Configuring the Embedded Target for TI C6000 DSP Blocks

To configure the C6711 DSK blocks in your model, follow these steps:
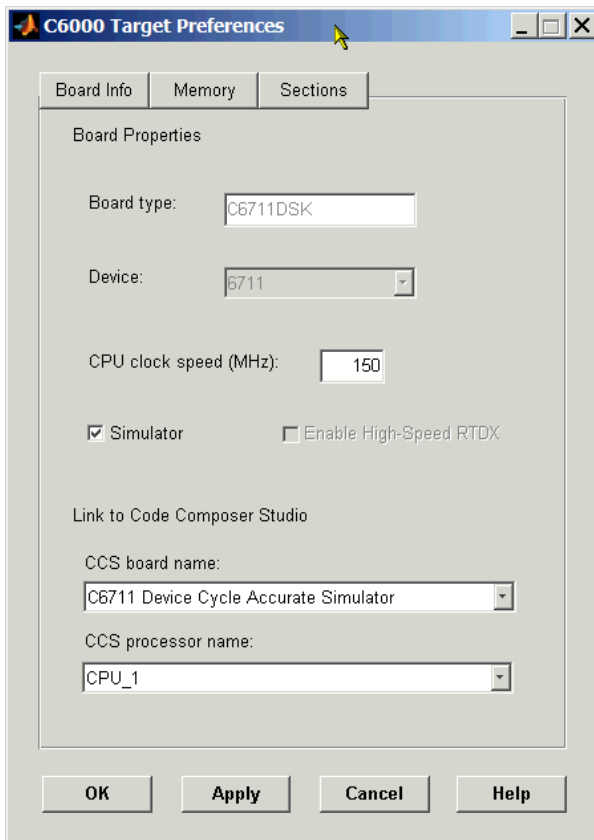
1 Click the C6711 DSK ADC block to select it.

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 directly 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.

8 Click the C6711DSK target preferences block.

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

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

Board info Settings

Memory Settings
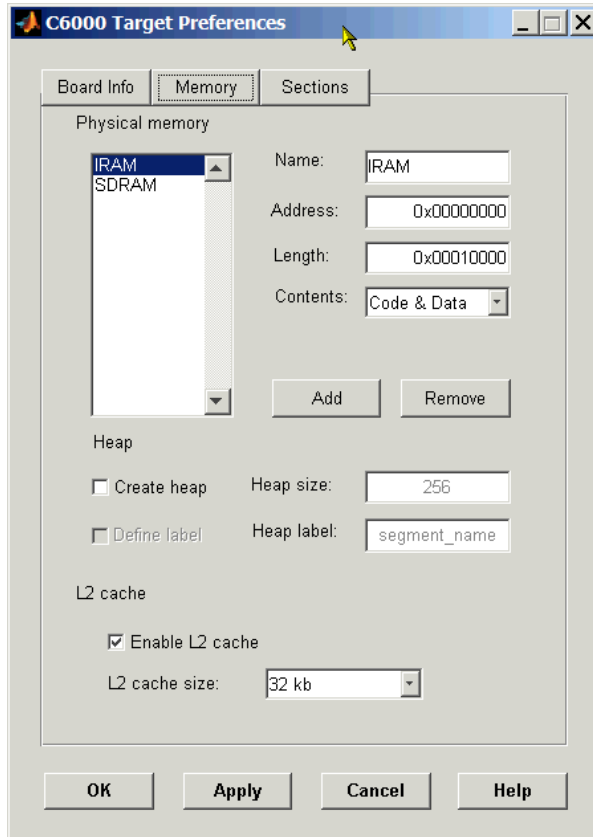
Section Settings

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

Configuring Configuration 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 **Configuration Parameters** dialog in Simulink.

Setting Simulink Configuration Parameters

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

1 Open the **Configuration Parameters** dialog, select Solver on the **Select** tree, and set the appropriate options on the **Solver** pane for your model and for the target.

- Set **Start time** to 0.0 and **Stop time** to inf (model runs without stopping). Note that the generated code ignores the stop time value.

- 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 **Tasking mode**.

Ignore the other general categories, such as Diagnostics or Data Import/Export, in the **Select** tree in the **Configuration 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 **Configuration 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 category in the **Select** tree.

2 In the **Target Selections** options, click **Browse** to select the system target file for the C6000 hardware targets.

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

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

4 Now, change the category again to TI C6000 Compiler/Linker to set the Real-Time Workshop compile and link options.

5 Set the following options in the **Compiler** options:

- Set **Compiler verbosity** to `Quiet`.
- Select the `Symbolic debugging` check box.

6 Set a **Linker** option by selecting the **Retain .obj files** check box.

7 Change categories on the **Select** tree to `TI C6000 Code Generation`.

8 Set the following run-time options:

- **Build action**: `Build_and_execute`.
- **Overrun action**: `Notify_and_halt`.
- **Overrun notification method**: `Turn_on_LEDs`.

You have configured the Real-Time Workshop options that let you target your C6711 DSK. Notice that you did not configure a number of Real-Time Workshop options on the **Select** tree: `Comments`, `Symbols`, `Custom Code`, `Interface`, and `Debug`. In addition, running this model does not require you to set all the Embedded Target for TI C6000 DSP specific options. For most of those, default values work fine.

For your model in this tutorial, the default values for options in these categories are appropriate. For other models you develop, you may want to set the options in these categories and others in the **Select** tree to provide information during the build and to launch target languange compiler (TLC) debugging when you generate code.

Building and Executing Your Model on Your C6711 DSK

After you set the configuration 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 Switch to the `Real-Time Workshop` category in the Configuration Parameters dialog. Clear **Generate code only** and 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.

C6711 DSK Tutorial—A More Complex Application

For this tutorial, we demonstrate an application that uses multiple stages—using wavelets to remove noise from a noisy signal. The model name is c6711dskwdnoisf. As with any model file, you can run this denoising demonstration by typing c6711dskwdnoisf at the MATLAB prompt.

The model also appears in the MATLAB demos collection in the Embedded Target for TI C6000 entry under the Simulink heading. Here is a picture of the model as it appears in the demonstration library.
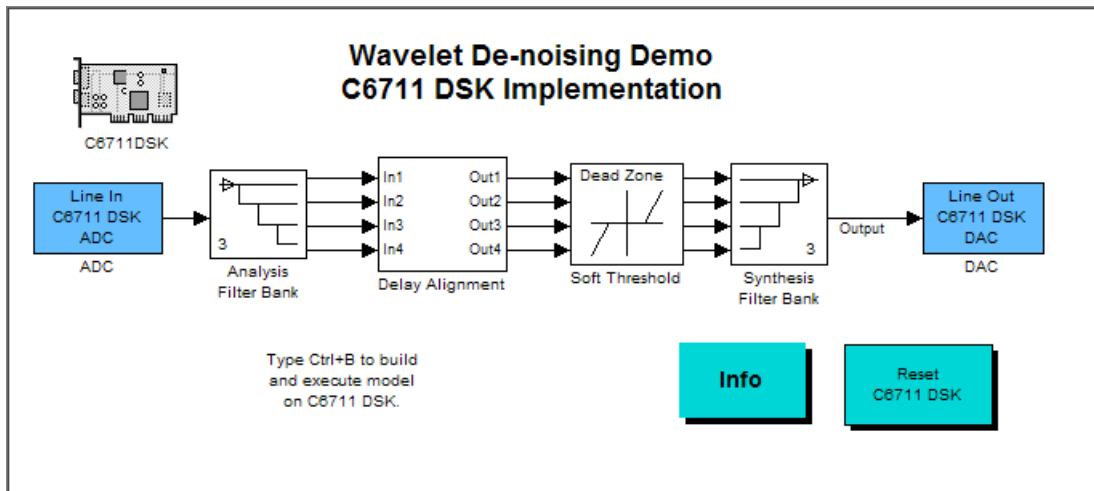
Unlike the audio reverberation tutorial model used in tutorial 2-3, this model is difficult to build from blocks in Simulink. It uses complicated subsystems for the Delay Alignment block and the Soft Threshold block. For this tutorial you work with a copy of the demonstration model, rather than creating 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

2-141

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 Configuration Parameters

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

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

2 Click `Solver` on the **Select** tree and enter the following parameter values on the `Solver` pane. Embedded Target for TI C6000 DSP ignores the stop time setting—the model runs continuously until you stop it.

Start Time: `0.0`

Stop Time: `inf`

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

Fixed step size: `auto`

Tasking mode: `Auto`

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

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

The next figure shows the `Solver` pane with the correct parameter settings.
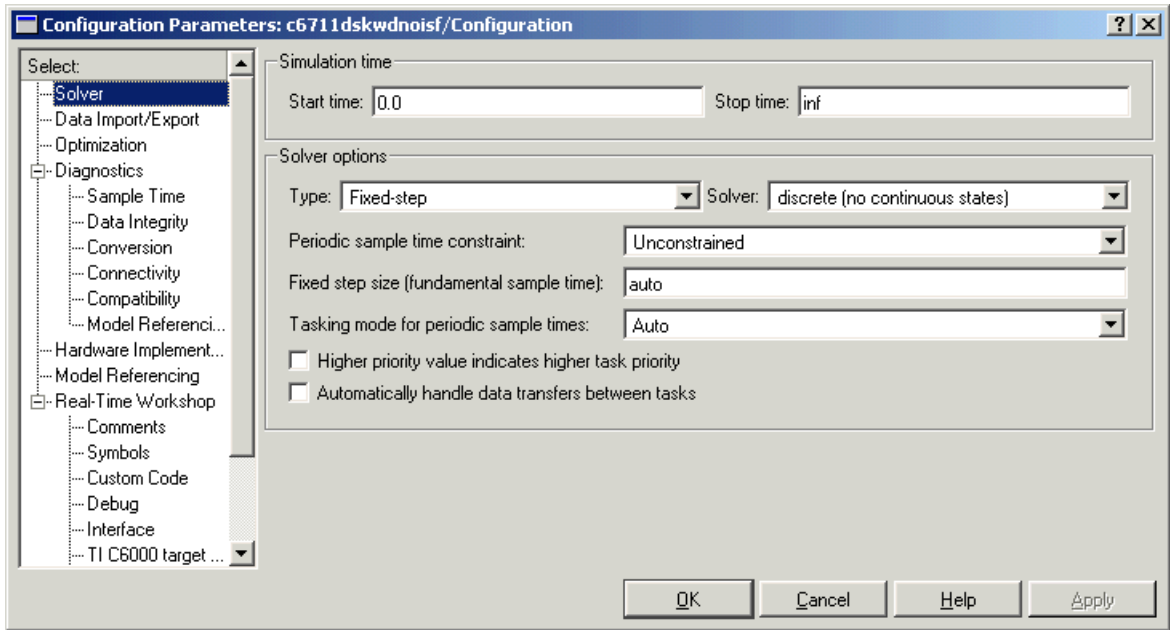
Selecting the Target Configuration

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 category has several subcategories, which you select using the **Select** tree. During this tutorial you change or review options on many categories in the tree.

To select the C6000 target:

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

2 Click `Real-Time Workshop` on the **Select** Tree of the **Configuration Parameters** dialog. The **Real-Time Workshop** pane activates.
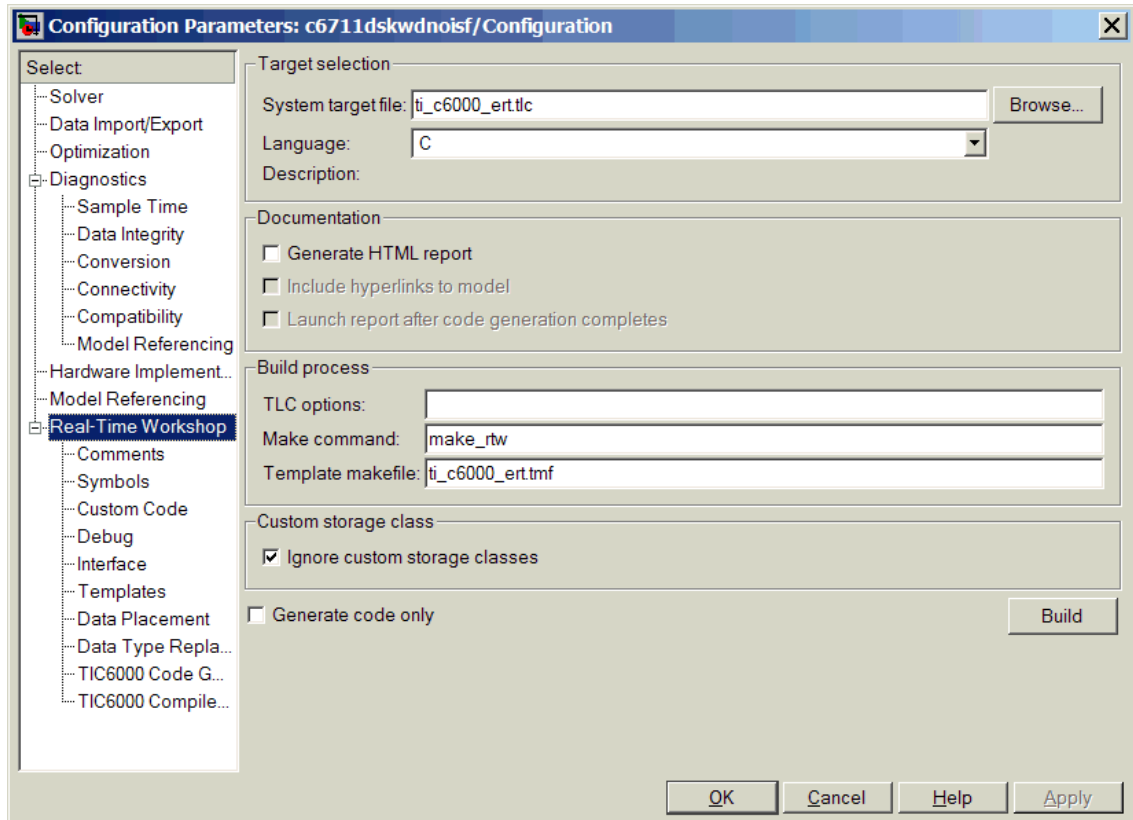
3 Click **Browse** next to the **System target file** field. This opens the **System Target File Browser**. The browser displays a list of available target configurations. When you select a target configuration, Real-Time Workshop automatically chooses the appropriate **System target file**, **Template makefile**, and **Make command**.
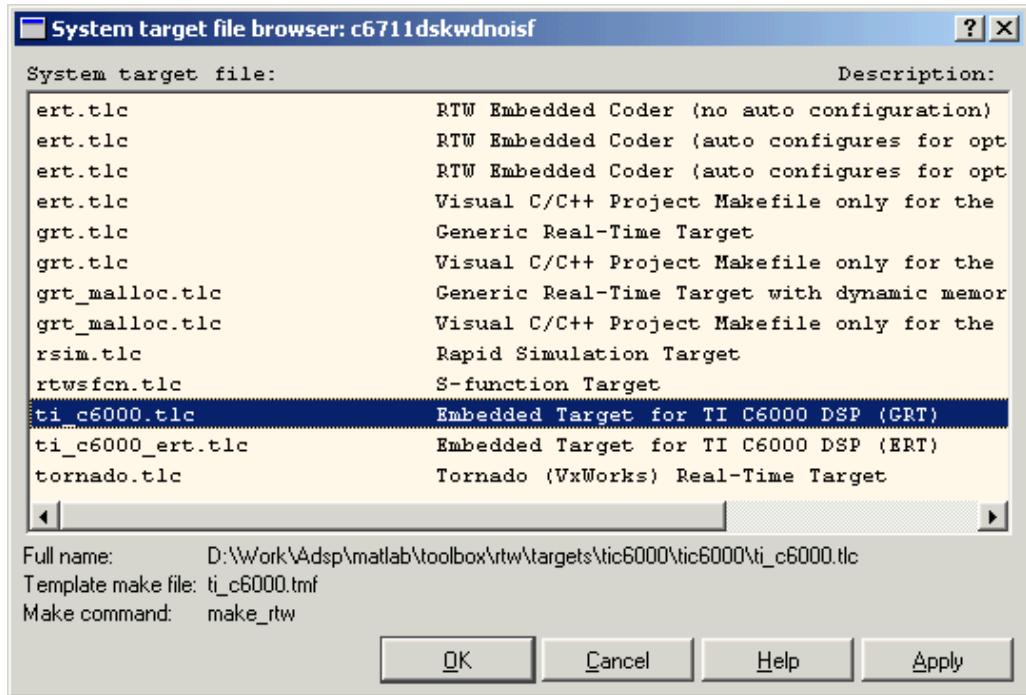
4 From the list of available configurations, select `ti_c6000.tlc` shown above and click **OK**.

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 the previous figure.

5 Select `Optimization` from the categories. The options displayed here are common to all target configurations. Check to make sure that all options are set to their defaults, as below.
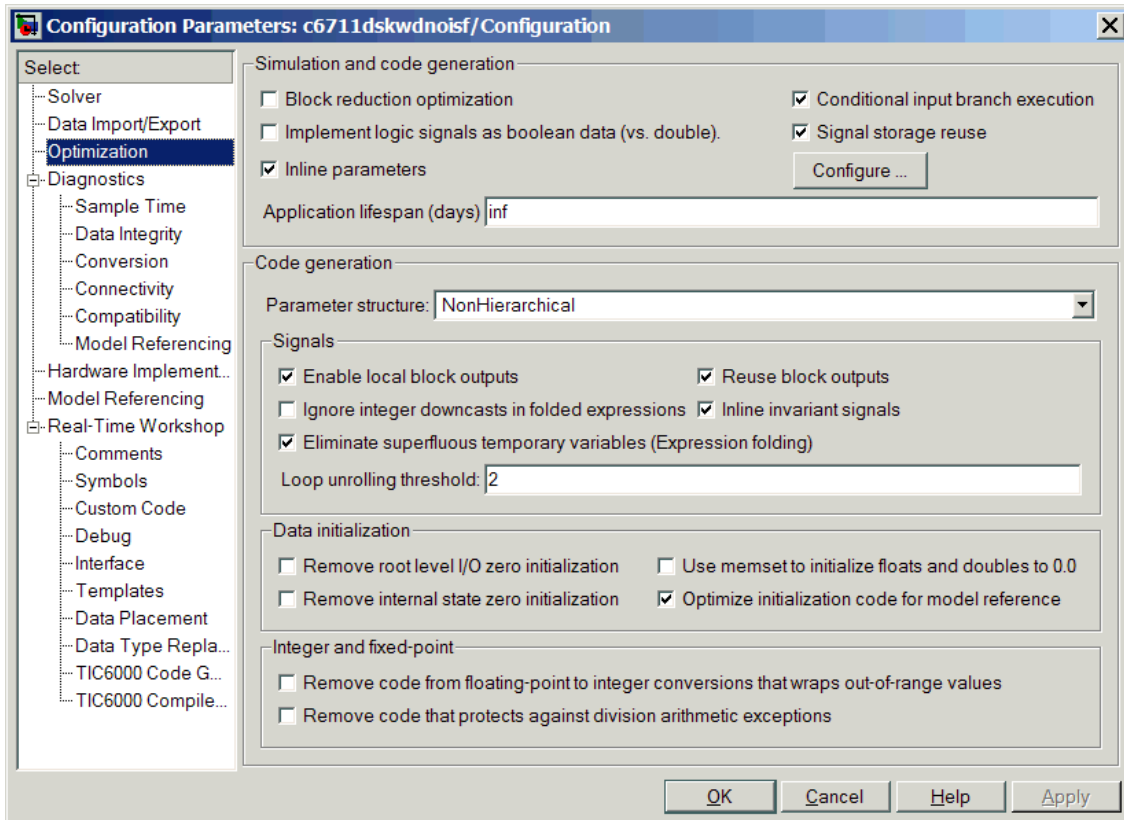
6 Select `Debug` from the **Select** tree. Clear all the check boxes in this pane. You can select the **Verbose build** option to see more build messages. Selecting **Verbose build** is optional but it can be helpful when you are new to generating code.

7 Select `TI C6000 Compiler/Linker` from the categories listed in the tree. The options displayed here are specific to the C6711 DSK target. Verify that the options are set as shown below.

2-147
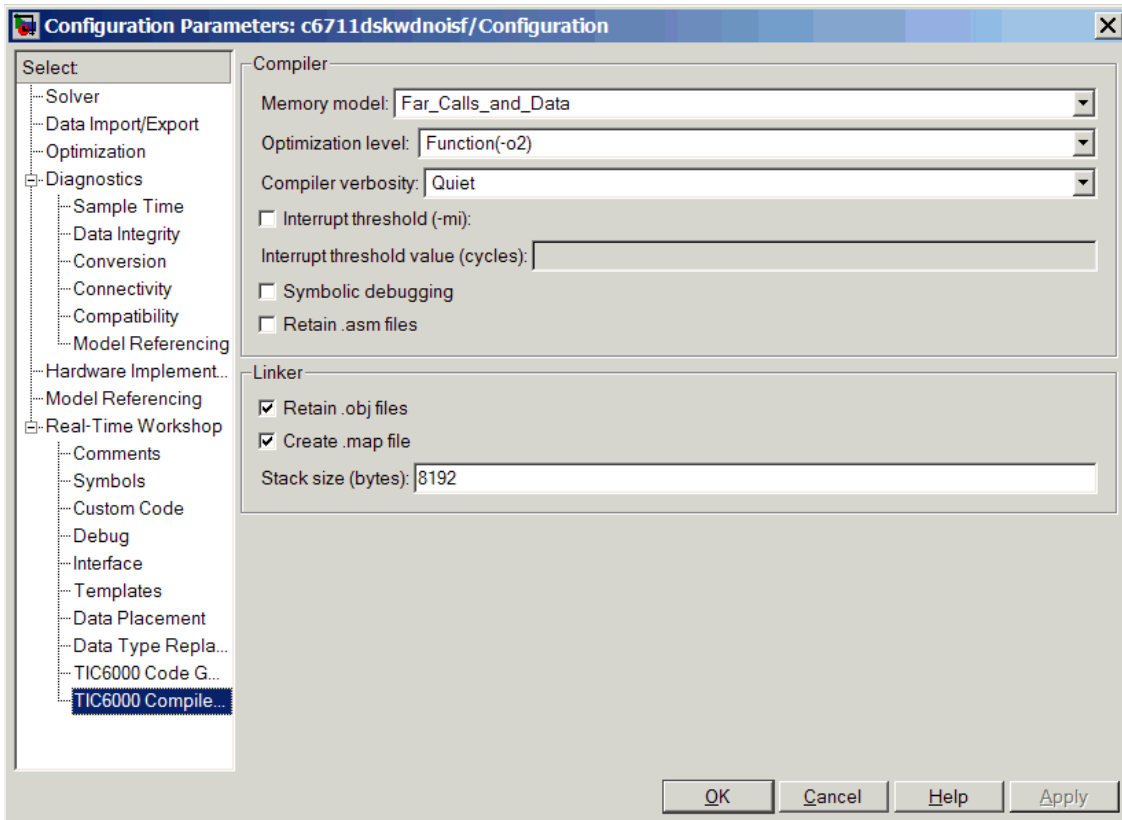
8 Set the **Linker** options as shown.

9 On the **Select** tree, choose TI C6000 Code Generation. Set the run-time options as shown under **Runtime**.
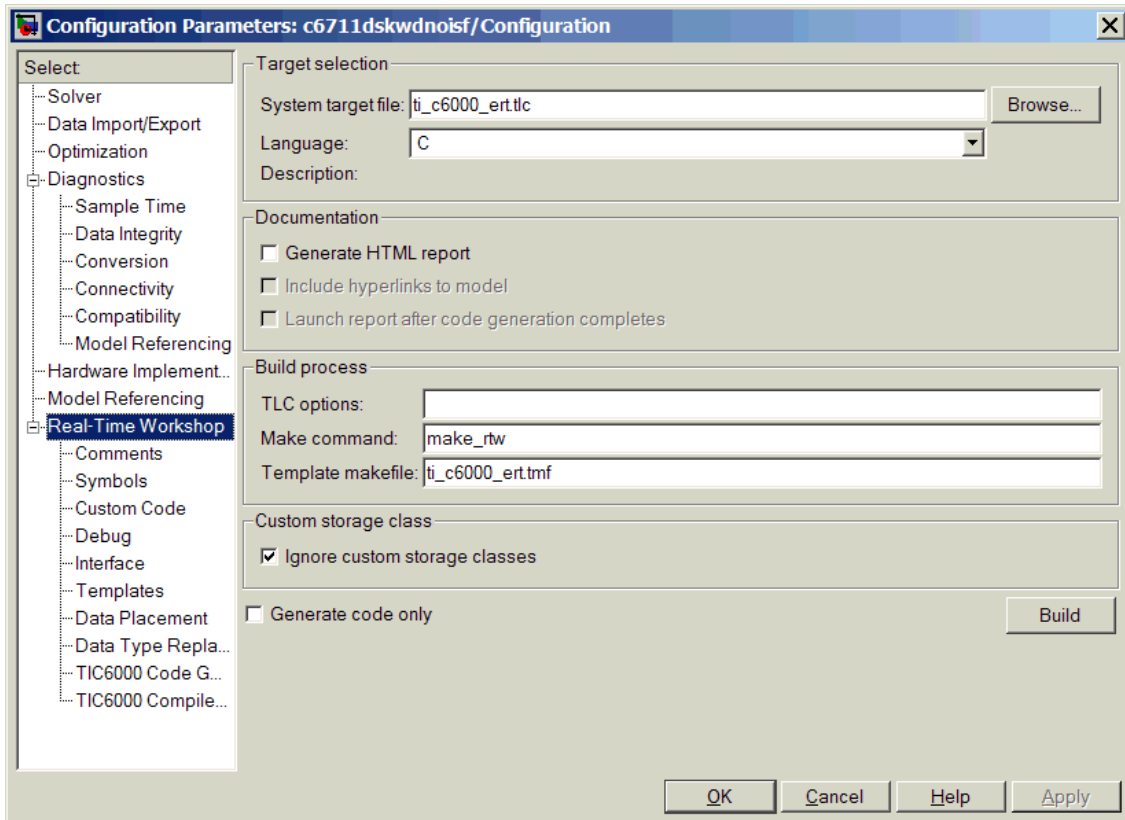
10 Click **OK** to close the **Configuration Parameters** dialog. Save the model to retain your new build settings.

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 configuration parameters for your model.

2 Clear **Generate code only** and click **Build** on the `Real-Time Workshop` pane of the **Configuration 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 **Select** tree in the **Configuration Parameters** dialog, select `TI C6000 Code Generation`. Select `Create_CCS_Project` for the **Build action**, as shown in the next figure. Note that the `Build` and `Build_and_execute` options create CCS projects as well. The `Generate_code_only` option does not create a CCS project. None of the other options has an effect here. Ignore them when you are creating a project in CCS rather than generating code.
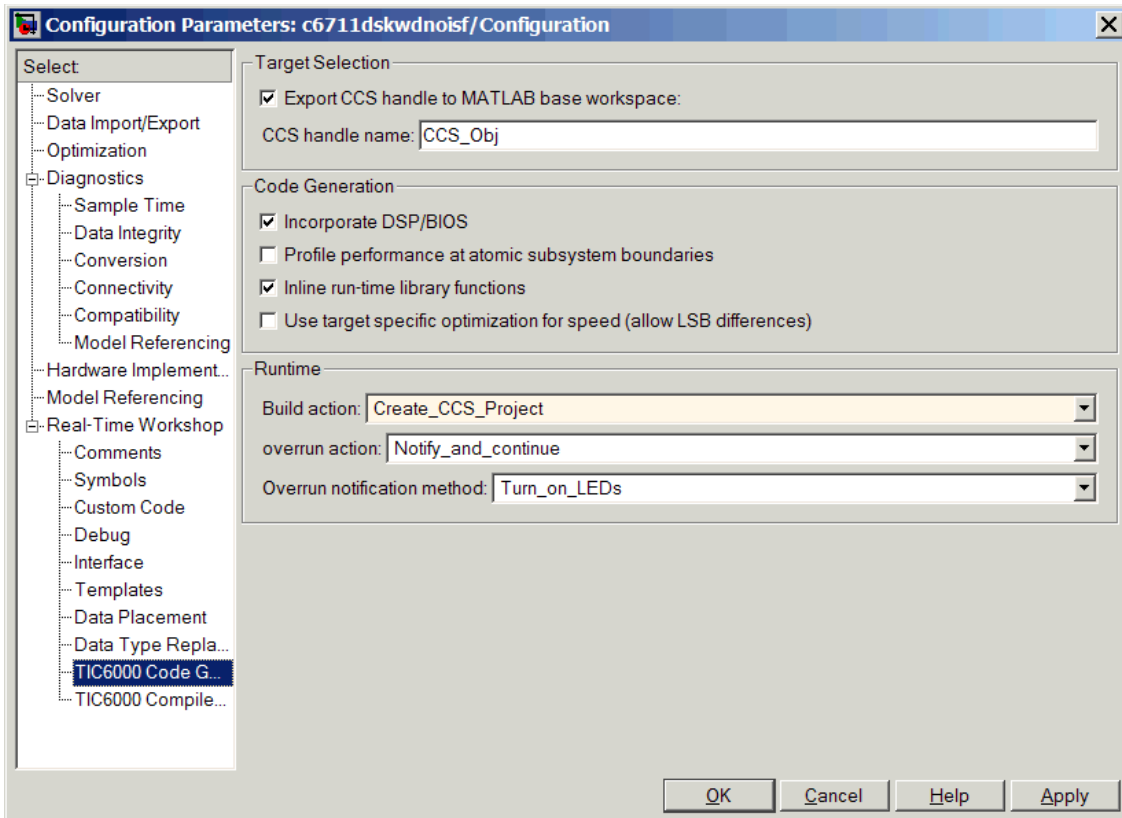
After you select Create_CCS_Project, set the options for the **Compiler** and **Linker** categories on the TI C6000 Compiler/Linker category on the **Select** tree.

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

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

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

Targeting Custom Hardware

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

Note that the blocks for the peripherals in the C6000 DSP Library, such as the C6701 EVM ADC or C6711 DSK DAC blocks, are specific to their hardware and will not work with your custom board. None of the board-specific blocks provided by this toolbox work with custom hardware. However, the RTDX and core support blocks should work for standard processors.

Custom hardware targeting currently supports all C6000 processors through target preferences blocks, either specific to the processor, or a general custom preferences block:

Target Preferences Block	Description
Custom C6000	Provides access to the hardware set up for targeting any C6000 processor-based board. Note that it does not set any default values. When you add this block to a model, you must set all the options on each available pane—board information, memory mapping, and section layout.
C6416DSK	Sets default values for targeting the C6416 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6701EVM	Sets default values for targeting the C6701 EVM. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.

Target Preferences Block	Description
C6711DSK	Sets default values for targeting the C6711 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
C6713DSK	Sets default values for targeting the C6713 DSK. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.
DM642EVM	Sets default values for targeting the DM642 EVM. After you add this block to your model, you can modify the default values as you require. Parameters in this block are set to match the board attributes.

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

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

1 Specify the memory allocation (memory mapping) using the **Memory** and **Sections** panes on the **C6000 Target Preferences** dialog. Set the memory mapping for your target that best matches your hardware. For example, if your custom target uses the C6713 processor, be sure your memory configuration is the same as the one on the supported C6713 DSK, such as has the same memory size, the same EMF settings, the same memory sections, and the same cache organization.

2 To use on-chip memory only for your target, choose the Near_Calls_and_Data setting for the **Memory model** in the **TI C6000 compiler** options. To use external memory that is specific to your board, choose the Far_Calls_and_Data setting for the **Memory model**. The other

2-155

selections in the **Memory model** list offer a combination of near and far allocation for calls, data, and aggregate data.

3 Do not use the existing ADC, DAC, DIP Switch, or LED blocks unless you are quite sure that your hardware is identical to the appropriate EVM or DSK in all important respects. Generally, the ADC, DAC, and other target-specific blocks are design specifically for their designated targets and can cause problems when you use them on hardware that is not identical.

4 Set the **Overrun notification method** in the **TI C6000 runtime** category to `Print_message` when you use the overrun notification feature. If you choose to use the LED notification option, verify that on your specialized target you access the LEDs in exactly the same way, and the LEDs respond in the same way, as the LEDs on the corresponding supported DSK or EVM.

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

Typical Targeting Process

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

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

2 Add target-specific blocks to your model, such as ADC and DAC blocks, and configure the block parameters. (Skip this step when you are targeting a processor on a custom board.)

3 Add a target preferences block to your model. Select the block that best matches your target: one of the device-specific blocks, like C6711DSK or the Custom C6000 block when none of the specific blocks is appropriate. All models that you target to C6000-processor-based must have a target preferences block at the top level of the model.

4 Configure the options on the target preferences block to select the target, map memory segments, allocate code and data sections to the memory segments, and other set other target-specific options.

5 Set the Simulink configuration parameters for your model. Notice that you do this after you add the target preferences block to your model.

6 Build your model to your target.

About Memory Maps

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

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

- Memory map— Map of the memory space for a target system. The memory space is partitioned into functional blocks.
- memory segment—Memory partition that corresponds to a physical range of memory on the target. The segment is named in some fashion, such as IPRAM or SDRAM.
- Memory section—the smallest unit of an object file. This is a block of data or code that, based on the memory map, resides in an area of contiguous memory on the target and in the memory map. Sections of object files are both distinct and separate. Memory sections come in two flavors:
  - Uninitialized sections that reserve memory space for uninitialized data. One example of an uninitialized section is `.bss`. The `.bss` section reserves space for variables that are not initialized.
  - Initialized sections contain code and data. `.text` (containing executable code) and `.data` (containing initialized data) are initialized sections.
- Memory management—process of specifying the memory segments that the various memory sections use for your application. A logical memory map of the hardware memory results from the process of managing memory.

During code generation, the linker and assembler work to allocate your code and data into the memory on your target according to the memory map specifications you provide. For more information about memory utilization and memory management, refer to the online help for CCS, using keywords like memory map, memory segment, and section. Note that the compiler does not interact with the memory map. It makes no assumptions about memory allocation and is not aware of the memory map. As far as the C6000 compiler

2-157

is concerned, the physical memory on your target is one continuous linear block of memory that is subdivided into smaller blocks containing code and/or data.

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

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

To Target a Custom C6000 Target

To use a board that has a TI C6000 processor but is not one of the supported boards, use the Custom C6000 target preferences block by adding it to your model. Configuring the block parameters tell Simulink, Embedded Target for TI C6000 DSP, and Real-Time Workshop about your target processor and how to generate code that will run on the target.

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

2 Click **Board Info** to access the board information pane shown here.
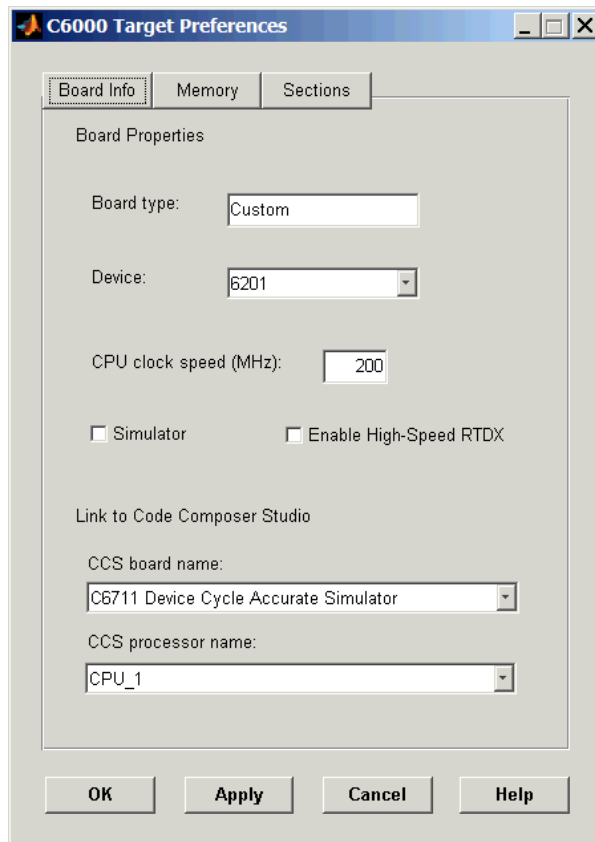
3 For **Board type**, enter Custom to tell the system you are targeting a board that Embedded Target for TI C6000 DSP does not explicitly support.

4 Select your target processor from the **Device** list. Most of the C6000 family of DSP processors are on the list. If the one you need is not listed, pick one that closely matches your target.

5 Set the actual CPU clock rate for the CPU on your target in CPU clock speed (MHz). Report the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results might be wrong, and code profiling results will

not be correct. You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

6 If your target is a simulator rather than a hardware target, select **Simulator**.

7 To enable high-speed RTDX, meaning that you are using a high-speed RTDX emulator or your hardware configuration supports high-speed RTDX, select **Enable High Speed RTDX**.

8 To enable Embedded Target for TI C6000 DSP to connect to CCS, select your target from the **CCS board name** list. On this list you see the names of the boards you have configured in the CCS Setup Utility. If your target board does not appear on the list, start CCS Setup and add your board to the **System Configuration** dialog.

9 Select the processor to target from the **CCS processor name** list. For the board you selected in **CCS board name**, **CCS processor name** lists all the processors on the board. The list comes from the processors you added to the board in the CCS Setup Utility.

Now you have completed the process of identifying your target to Embedded Target for TI C6000 DSP and Real-Time Workshop. While this process is necessary, it represents only one small part of enabling you to generate code to run on your custom board.

Perhaps the most important part of targeting custom hardware is providing the memory map configuration for your target to the linker and assembler.

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

Here are the **Memory** and **Sections** panes with the default values for all options.

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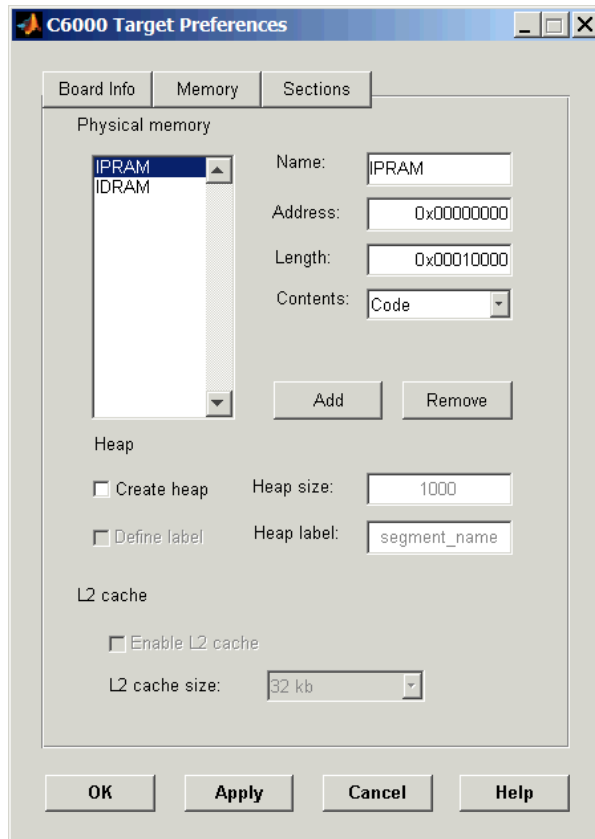
Sections Pane
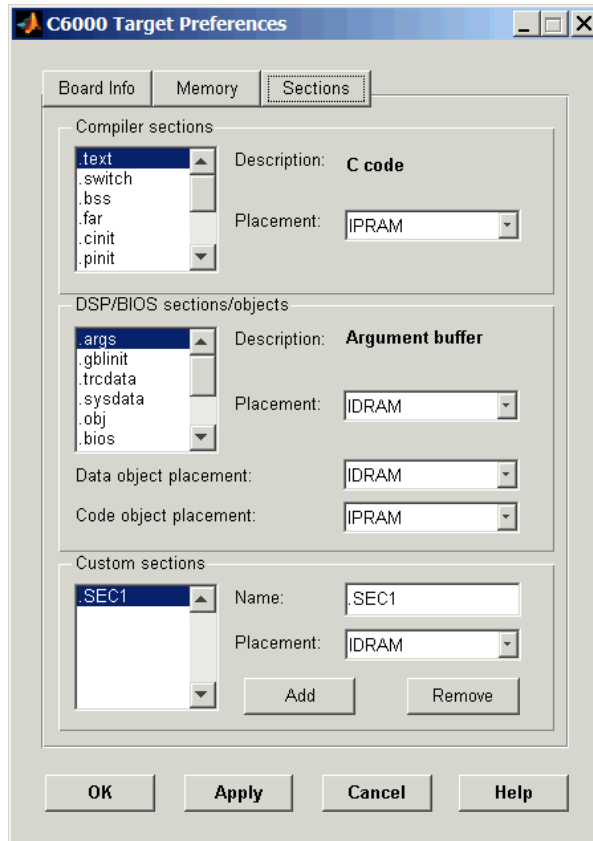
The information that follows describes the options on the panes in detail.

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

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

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

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

Physical Memory Options

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

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

- Custom boards based on C670x processors provide IPRAM and IDRAM memory segments by default.
- C6701 EVM boards provide IPRAM, IDRAM, SBSRAM, SDDRAM0, and SDRAM1 memory segments by default
- C6711DSK boards provide SDRAM memory segment by default

Name

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

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

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

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

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

Address

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

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

Length

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

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

Contents

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

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

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

Add

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

entering the new segment name. Entering the new name, or clicking **Apply** updates the temporary name on the list to the name you enter.

Remove

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

Create Heap

If your processor supports using a heap, as do the C6711 or C6701, for example, selecting this option enables creating the heap and enables the **Heap size** option. **Create heap** is not available on processors that either do not provide a heap or do not allow you to configure the heap.

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

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

Heap Size

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

Define Label

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

Heap Label

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

Enable L2 Cache

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

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

L2 Cache Size

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

Sections Pane

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

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

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

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

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

String	Section List	Description of the Section Contents
.switch	Compiler	Jump tables for switch statements in the executable code
.sysdata	DSP/BIOS	Data about DSP/BIOS
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code
.sysmem	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

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

- .cinit
- .const
- .switch
- .text—Created by the assembler

These sections are uninitialized:

- .bss—Created by the assemble.
- .far
- .stack

- `.sysmem`

Other sections appear on the list as well:

- `.data`—created by the assembler. The C/C++ compiler does not use this section.
- `.cio`
- `.pinit`

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

Description

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

Placement

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

DSP/BIOS Sections

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

Description

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

Placement

Shows where the selected **DSP/BIOS Sections** list entry is allocated in memory. You change the memory allocation by selecting a different location

from the **Placement** list. The list contains the memory segments available on C6000 processors, and changes based on the processor you are using.

DSP/BIOS Object Placement

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

Custom Sections

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

Name

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

Placement

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

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom Sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

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

2-171

To Create Memory Maps for Targets

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

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

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

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

Once you have configured the options in the **C6000 Target Preferences** dialog, you are ready to set the Simulink configuration parameters for your model and generate code.

Using Embedded Target for TI C6000 DSP with Real-Time Workshop Embedded Coder

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

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

If you simply choose the system target file `ti_c6000_ert.tlc` in the **System Target File Browser** directly to change the target for the model, all the TI C6000 code generation options are reset to default values by the switch. The C6000-specific options are the same between the two system target files.

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

To preserve the option values in the configuration set when you migrate your model to the ERT-based target (or back to the GRT-based target), use the function `switchc6000target.m`.

For example, the command

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

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

Conversely,

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

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

To Use the Embedded Coder Target File

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

1 From your model menu bar, select **Simulation—>Configuration Parameters**.

2 Click `Real-Time Workshop` on the **Select** tree to access the Real-Time Workshop options.

3 Click **Browse** to open the **System Target File Browser**.

4 On the **System Target File Browser**, find and select the file `ti_c6000_ert.tlc`.

5 Click **OK**.

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

1 Open your Simulink model to change.

2 At the MATLAB prompt, enter

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

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

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

3

Targeting with DSP/BIOS Options

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 3-22

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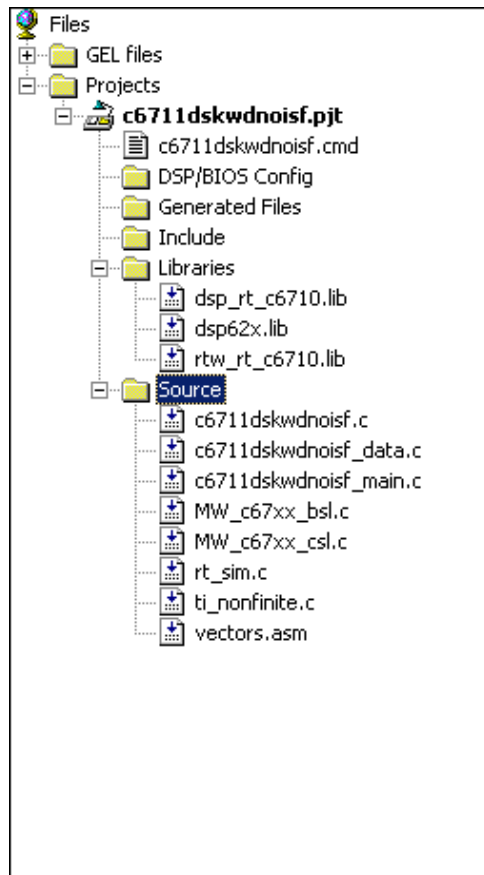
Example—c6711dskwdnoisf.pjt Code Including DSP/BIOS

If you now create a new project that includes DSP/BIOS, the directory structure for your project changes to look like the following figure.
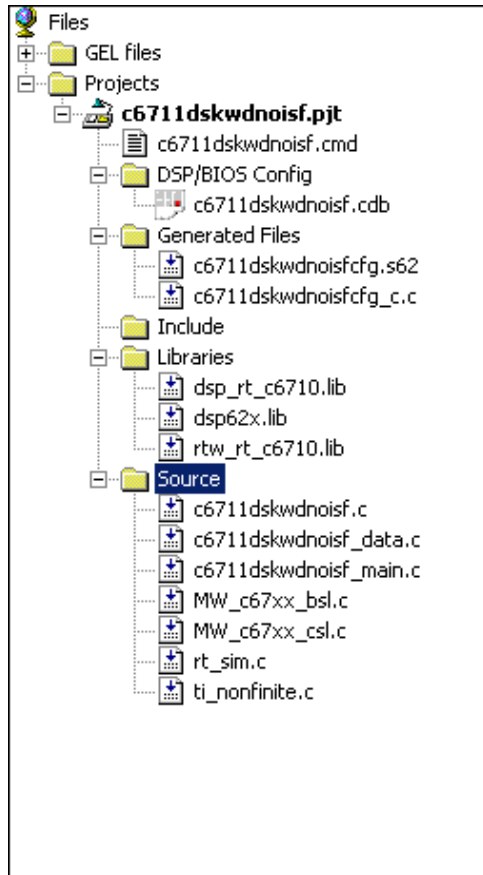
Notice that the new directory includes some new files, shown in the next table.

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 **HWI** section of the Configuration Tool.
Original linker command file— modelname.cmd	Assigns memory sections on the processor. This file is removed if the **SECTION** directive is empty because all of the section assignments moved to the configuration file. Otherwise, include call to the DSP/BIOS command file.
Some *.lib files	Provide access to libraries for the processor, and peripherals. These files are removed if their contents have been incorporated in the new compound linker command file.

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

Profiling Generated Code

When you use the Embedded Target for TI C6000 DSP to generate code that incorporates the DSP/BIOS options, you can easily profile your generated code to gauge performance and find bottlenecks. By selecting **Profile performance at atomic subsystem boundaries** in the Real-Time Workshop options, Real-Time Workshop inserts statistics (STS) object instrumentation at the beginning and end of the code for each atomic subsystem in your model. (For more about STS objects, refer to your DSP/BIOS documentation from Texas Instruments.)

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

Code profiling works only on atomic subsystems in your model. To allow Embedded Target for TI C6000 DSP to profile your model when you build it in Real-Time Workshop, you convert segments of your model into atomic subsystems using **Create subsystem**. By designating subsystems of your model as atomic, you force each subsystem to execute only when all of its inputs are available. Waiting for all the subsystem inputs to be available before running the subsystem allows the subsystem code to be profiled as a contiguous segment.

To enable the profile feature for your Simulink model, choose **Tools-> Real-Time Workshop -> Options** from the model menu bar. Navigate to the **TI C6000 code generation** category, and select the **Profile performance at atomic subsystem boundaries** check box.

Profiling Subsystems

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

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

must execute contiguously for the profiling timing measurements to be correct. Setting the **Tasking mode for periodic sample times** to Auto in the model configuration parameters does not guarantee contiguous execution for all code segments and subsystems.

Notice two things in your code:

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

The Embedded Target for TI C6000 DSP inserts and configures these objects specifically for profiling your code. You do not have to make changes to the STS objects. To see the statistics objects in use, download your generated application to your board, select **DSP/BIOS -> Statistics View** from the menu bar in CCS, and run the board for a few seconds. You see the statistics being accumulated by the STS objects.

Profiling Multitasking Systems

For a multitasking system, DSP/BIOS STS objects cannot reliably measure the time the processor spends in all tasks. When tasks can be preempted by other tasks (a result of multitasking operation), the profile timing measurements may be incorrect. For this reason, Embedded Target for TI C6000 DSP includes profiling instrumentation for atomic systems that run at the base sample rate only.

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

1 Select **Tools** -> **Real-Time-Workshop** -> **Options** from the model menu bar.

2 Under **Tasking** on the **Solver** pane, select SingleTasking for **Tasking mode for periodic sample times**.

3 Rebuild and execute your model on your C6000 hardware.

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

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

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

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

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

The Profiling Report

To help you to measure subsystem performance, Embedded Target for TI C6000 DSP provides a custom HTML report that analyzes and displays the profile statistics. The HTML page shows you the amount of time spent computing each subsystem, including both `Outputs` and `Update` code segments, and provides links to open the corresponding subsystem in the Simulink model.

To view the profiling report, enter

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

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

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

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

```
tempdir
```

at the MATLAB command prompt.

Note Each time you run the profiling process, Embedded Target for TI C6000 DSP replaces your existing report with a newer version. To save earlier reports, rename and save the report before you generate a new one, or change your destination temporary directory in MATLAB.

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

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

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

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

or

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

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

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

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

Interrupts and Profiling

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

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

The interrupt rate for a DSP/BIOS application created by the Embedded Target for TI C6000 is the fastest block execution rate in the model. The interrupt rate is usually, but not always, the same as the codec frame rate. When there is an upsampling operation or other rate increasing operation in your model, interrupts are triggered by a timer (PRD) object at the faster rate. You can determine the effective interrupt rate of the model by inverting the interrupt interval reported by the profiler.

Reading Your Profile Report

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

Sample of a Profile Report

When you click Sample Profile Report, the sample report opens in a new Help browser window. This opens the sample report in a new window so you can read the report and the descriptions of the report contents at the same time.

Running the model `c6711dskwdnoisf` with DSP/BIOS generates the sample profile report. The next sections explain the headings in the report—what they mean and how they are measured (where that applies).

Report Heading Information

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

Report Subsections and Contents

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

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

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

Definitions of Report Entries

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

System name

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

Number of Iterations Counted

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

CPU Clock Speed

The instruction cycle speed of your digital signal processor. On the C6701 EVM, you can adjust this speed to one of four values, where 100 MHz is the default—25, 33.25, 100, 133 MHz. If you change the speed to something other than the default setting of 100 MHz, you must specify the new speed in the Real-Time Workshop options. Use the **Current C6701EVM CPU clock rate** option on the TIC6000 runtime category on the RTW tab.

Set at a fixed 150 MHz, you cannot change the CPU clock rate on the C6711 DSK. You do not need to report the setting in the Real-Time Workshop options.

Maximum time spent in this subsystem per interrupt

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

Maximum Percent of base interval

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

STS Objects

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

```
stsSysi_segment
```

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

Profiling Your Generated Code

Before profiling your generated code, you must configure your model and Real-Time Workshop to support the profiling features in Embedded Target for TI C6000 DSP. Your model must use DSP/BIOS features for profiling to work fully.

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

1 Enable DSP/BIOS for your code.

2 Enable profiling in the Real-Time Workshop.

3 Create atomic subsystems to profile in your model.

4 Build, download, and run your model.

5 In MATLAB, use `profile` to view the profile report.

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

procedure. Simulators work as well, just choose the appropriate model for your simulator.

Begin by loading the model, entering

```
c6711dskwdnoisf
```

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

To Enable Profiling for Your Generated Code

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

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

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

2 Click **Real-Time Workshop** to display the configuration panes for setting your code generation options.

3 From the **Category** list, select TI C6000 Code Generation.

Your display changes to show the options you set to control code generation for TI C6000 targets, as shown here.
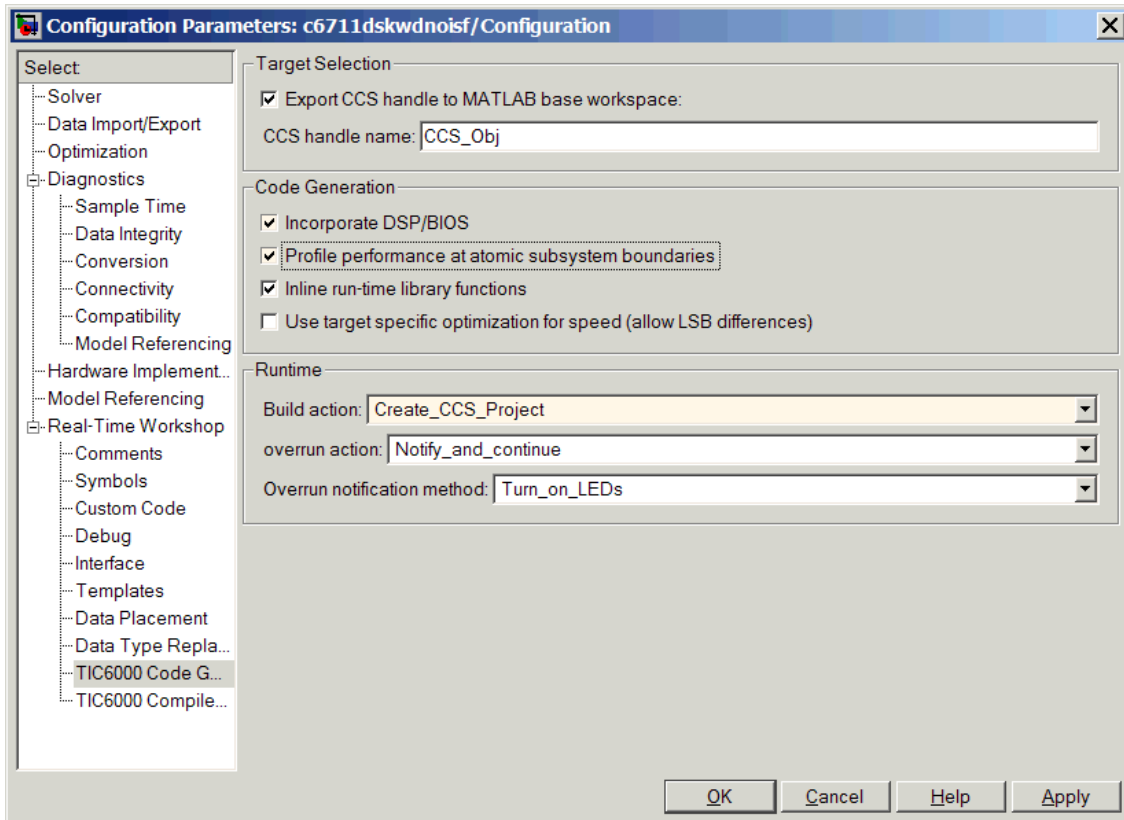
4 Select the **Profile performance at atomic subsystem boundaries** option. Selecting this option enables profiling in your generated code. However, you still need to configure your model to support the profiling process.

To Create Atomic Subsystems for Profiling

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

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

1 Select the Analysis Filter Bank block. Select **Edit -> Create subsystem** from the model menu bar. Note that the name of the block changes to subsystem. Repeat for the Soft Threshold block.

2 To convert your new subsystems to atomic subsystems, right-click on each subsystem and choose **Subsystem parameters...** from the context menu.

3 In the **Block Parameters: Subsystem** dialog for each subsystem, select the **Treat as atomic unit** option. Click **OK** to close the dialog. If you look closely you can see that the subsystems now have heavier borders to distinguish them from the other blocks in your model.

To Build and Profile Your Generated Code

You have enabled profiling in your model and configured two atomic subsystems in the model as well. Now, use the profiling feature in Embedded Target for TI C6000 DSP to see how your code runs and check the performance for bottlenecks and slowdowns as the code runs on your target.

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 = ccsdsp;
```

at the MATLAB prompt.

3 Enter

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

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

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

Profile Report

Simulink model: c6416dskprofile.mdl
Target: C6416DSK

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

Timing constants

Base sample time	250 ms
CPU Clock speed[1]	720 MHz

Profiled Simulink Subsystems

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

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

3-21

Using DSP/BIOS with Your Target Application

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

To Enable DSP/BIOS When You Generate Code

For any code you generate using Real-Time Workshop and the Embedded Target for TI C6000 DSP, you have the option of including DSP/BIOS features automatically when you generate the code. Incorporating the features requires you to select one option in the `TI C6000 Code Generation` settings— **Incorporate DSP/BIOS**.

1 Open the model to use to generate code.

2 From your model menu bar, select **Simulation -> Simulation parameters...** to start the **Simulation Parameters** dialog.

3 From the **Category** list, select `TI C6000 Code Generation`.

To provide access to the options, the display changes to show the following options.
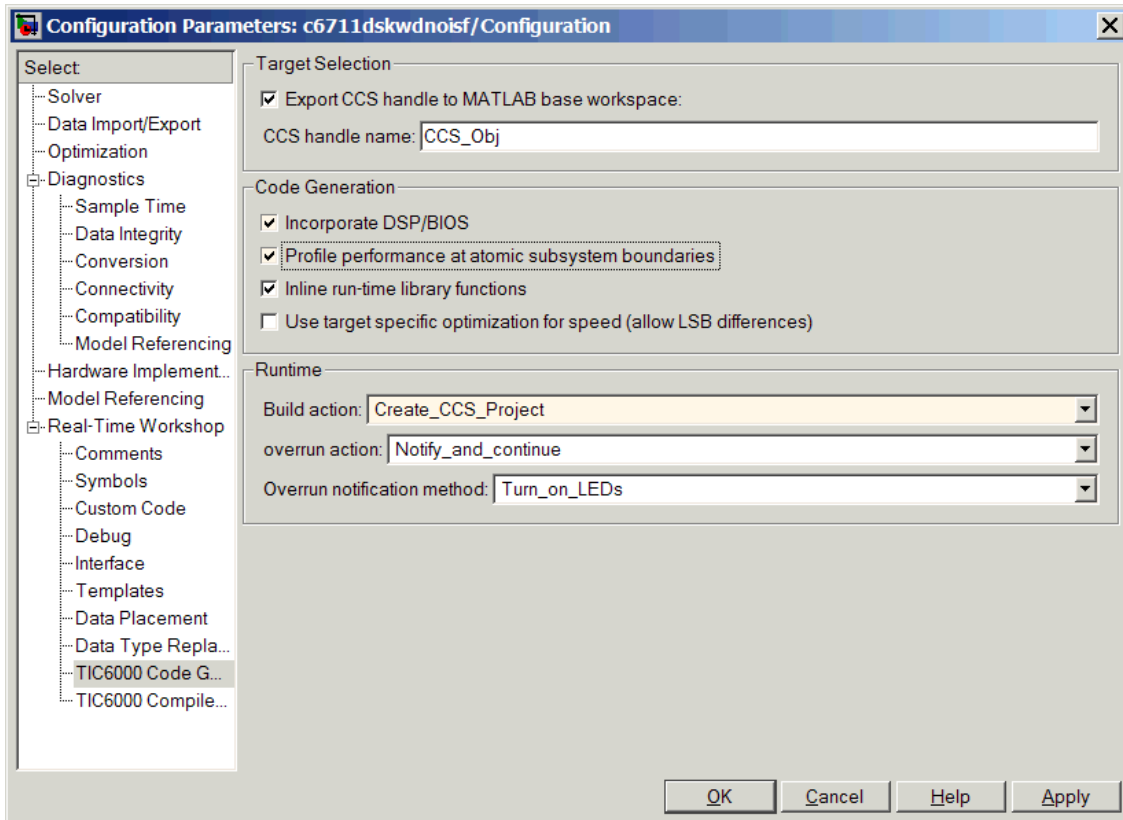
4 As shown in the figure, select **Incorporate DSP/BIOS**.

5 Using the other entries on the **Category** list, set other options as you require for your project.

6 For the **Build action** (under **Runtime**), select one of the following choices. Each option generates code that includes the DSP/BIOS instrumentation:

- `Create_CCS_project`
- `Build`

- Build_and_execute

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

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

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

4

Using the C62x and C64x DSP Libraries

About the C62x and C64x DSP Libraries

C62x DSP Library

Blocks in the C62x DSP library correspond to functions in the Texas Instruments TMS320C62x DSP Library assembly-code library, which target the TI C62x family of digital signal processors. Use these blocks to run simulations by building models in Simulink before generating code. Once you develop your model, you can invoke Real-Time Workshop to generate code that is optimized to run on the C6711 DSK or C6701 EVM development platforms or C62x hardware. (Fixed-point processing on C67x hardware is identical to C62x fixed point hardware and processing so you can develop on the C67x for the C62x.) During code generation, each C62x DSP Library block in your model is mapped to its corresponding TMS320C62x DSP Library assembly-code routine to create target-optimized code.

C62x DSP Library blocks generally input and output fixed-point data types. Chapter 5, "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 Toolbox documentation.

You can use C62x DSP Library blocks with certain blocks from the DSP Blockset and Simulink. To learn more about creating models that include both C62x DSP Library blocks and blocks from other blocksets, refer to "Building Models" on page 4-8.

C64x DSP Library

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

C64x DSP Library blocks generally input and output fixed-point data types. Chapter 5, "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 Toolbox documentation.

You can use C64x DSP Library blocks with certain certain blocks from the DSP Blockset and Simulink. To learn more about creating models that include both C64x DSP Library blocks and blocks from other blocksets, refer to "Building Models" on page 4-8.

Note While you can use C62x blocks on C64x targets, the generated code is not optimal for the C64x target. Using the appropriate C64x block creates better optimized code. (Embedded Target for TIC6000 generates a warning message when you try to do this but allows you to use the block.)

You cannot use the C64x blocks on your C62x target.

Characteristics Common to C62x and C64x Library Blocks

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

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

To learn more about characteristics particular to each block in the library, refer to Chapter 5, "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.

b_{ws-1}	b_{ws-2}	...	b_5	b_4	b_3	b_2	b_1	b_0

MSB   LSB
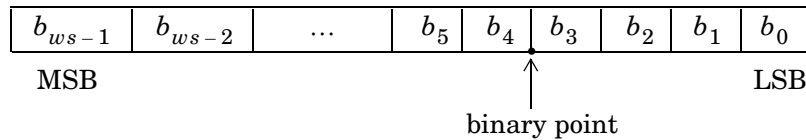
binary point

where

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

Signed Fixed-Point Numbers

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

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

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

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

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

results in the negative of 000101 being 111011.

Q Format Notation

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

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

$Qm.n$

where

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

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

Example—Q.15

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

    Q0.15

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

    Q.15

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

    sfrac16

or

    sfix16_En15

The Filter Design Toolbox expresses this data type as the vector

    [16 15]

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

Example—Q1.30

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

    m+n+1 = 1+30+1 = 32 bits total

Therefore this number is expressed as

    Q1.30

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

    sfix32_En30

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

    [32 30]

Example—Q-2.17

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

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

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

m+n+1 = -2+17+1 = 16 bits total

Therefore this number is expressed as

    Q-2.17

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

    sfix16_En17

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

    [16 17]

Example—Q17.-2

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

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

Therefore this number is expressed as

    Q17.-2

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

    sfix16_E2

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

    [16 -2]

Building Models

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

Converting Data Types

Any blocks you connect in a model have compatible input and output data types. In most cases, C62x or C64x DSP Library blocks handle only a limited number of specific data types. Refer to any block reference page in Chapter 5, "Block Reference" for a discussion of the data types that each block accept sand produces.

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

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

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

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

Using Sources and Sinks

The C62x DSP Library does not include source or sink blocks. Use source or sink blocks from the core Simulink library or 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.

5

Block Reference

Blocks—Categorical List

The following pages list the blocks included in the Embedded Target for C6000 DSP. After the tables, block reference pages provide details about every block, including 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.

Blocks in Library c6000tgtprefs

Block	Description
C6416DSK	Set target preferences and memory map to generate code for the C6416 DSP Starter Kit
C6701EVM	Set target preferences and memory map to generate code for the C6701 Evaluation Module
C6711DSK	Set target preferences and memory map to generate code for thet C6711 DSP Starter Kit
C6713DSK	Set target preferences and memory map to generate code for the C6713 DSP Starter Kit
Custom C6000	Set target preferences and memory map to generate code for C6000-processor-based custom hardware targets
DM642EVM	Set target preferences and memory map to generate code for the DM642 Evaluation Module

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

Blocks in Library c6701evmlib

Block	Description
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

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

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

Blocks in the C62x DSP Library

Block	Description
Conversions	
C62x Convert Floating-Point to Q.15	Convert a single-precision floating-point input signal to a Q.15 fixed-point signal
C62x Convert Q.15 to Floating-Point	Convert a Q.15 fixed-point signal to a single-precision floating-point signal
Filters	
C62xComplex FIR	Filter a complex input signal using a complex FIR filter
C62xGeneral Real FIR	Filter a real input signal using a real FIR filter
C62xLMS Adaptive FIR	Perform least-mean-square adaptive FIR filtering
C62xRadix-4 Real FIR	Filter a real input signal using a real FIR filter
C62xRadix-8 Real FIR	Filter a real input signal using a real FIR filter
C62xReal Forward Lattice All-Pole IIR	Filter a real input signal using an auto-regressive forward lattice filter
C62xReal IIR	Filter a real input signal using a real auto-regressive moving-average IIR filter

Blocks in the C62x DSP Library (Continued)

Block	Description
C62xSymmetric Real FIR	Filter a real input signal using a symmetric real FIR filter
Math and Matrices	
C62xAutocorrelation	Compute the autocorrelation of an input vector or frame-based matrix
C62xBlock Exponent	Return the minimum exponent (number of extra sign bits) found in each channel of an input
C62xMatrix Multiply	Perform matrix multiplication on two input signals
C62xMatrix Transpose	Compute the matrix transpose of an input signal
C62xReciprocal	Compute the fractional and exponential portions of the reciprocal of a real input signal
C62xVector Dot Product	Compute the vector dot product of two real input signals
C62xVector Maximum Index	Compute the zero-based index of the maximum value element in each channel of an input signal
C62xVector Maximum Value	Compute the maximum value for each channel of an input signal
C62xVector Minimum Value	Compute the minimum value for each channel of an input signal
C62xVector Multiply	Perform element-wise multiplication on two inputs
C62xVector Negate	Negate each element of an input signal
C62xVector Sum of Squares	Compute the sum of squares over each channel of a real input

Blocks in the C62x DSP Library (Continued)

Block	Description
C62xWeighted Vector Sum	Find the weighted sum of two input vectors
Transforms	
C62xBit Reverse	Bit-reverse the positions of the elements of each channel of a complex input signal
C62xFFT	Compute the decimation-in-frequency forward FFT of a complex input vector
C62xRadix-2 FFT	Compute the radix-2 decimation-in-frequency forward FFT of a complex input vector
C62xRadix-2 IFFT	Compute the radix-2 inverse FFT of a complex input vector

Blocks in the C64x DSP Library

Block	Description
Conversions	
C64xConvert Floating-Point to Q.15	Convert a floating-point signal to a Q.15 fixed-point signal
C64xConvert Q.15 to Floating-Point	Convert a Q.15 fixed-point signal to a single-precision floating-point signal
Filters	
C64xComplex FIR	Filter a complex input signal using a complex FIR filter
C64xGeneral Real FIR	Filter a real input signal using a real FIR filter
C64xLMS Adaptive FIR	Perform least-mean-square adaptive FIR filtering
C64xRadix-4 Real FIR	Filter a real input signal using a real FIR filter

Blocks in the C64x DSP Library (Continued)

Block	Description
C64xRadix-8 Real FIR	Filter a real input signal using a real FIR filter
C64xReal Forward Lattice All-Pole IIR	Filter a real input signal using an auto-regressive forward lattice filter
C64xReal IIR	Filter a real input signal using a real auto-regressive moving-average IIR filter
C64xSymmetric Real FIR	Filter a real input signal using a symmetric real FIR filter
Math and Matrices	
C64xAutocorrelation	Compute the autocorrelation of an input vector or frame-based matrix
C64xBlock Exponent	Return the minimum exponent (number of extra sign bits) found in each channel of an input
C64xMatrix Multiply	Perform matrix multiplication on two input signals
C64xMatrix Transpose	Compute the matrix transpose of an input signal
C64xReciprocal	Compute the fractional and exponential portions of the reciprocal of a real input signal
C64xVector Dot Product	Compute the vector dot product of two real input signals
C64xVector Maximum Index	Compute the zero-based index of the maximum value element in each channel of an input signal
C64xVector Maximum Value	Compute the maximum value for each channel of an input signal
C64xVector Minimum Value	Compute the minimum value for each channel of an input signal

Blocks in the C64x DSP Library (Continued)

Block	Description
C64xVector Multiply	Perform element-wise multiplication on two inputs
C64xVector Negate	Negate each element of an input signal
C64xVector Sum of Squares	Compute the sum of squares over each channel of a real input
C64xWeighted Vector Sum	Find the weighted sum of two input vectors
Transforms	
C64xBit Reverse	Bit-reverse the positions of the elements of each channel of a complex input signal
C64xFFT	Compute the decimation-in-frequency forward FFT of a complex input vector
C64xRadix-2 FFT	Compute the radix-2 decimation-in-frequency forward FFT of a complex input vector

Blocks in the C6416 DSP Library

Block	Description
C6416 DSK ADC	Configure digitized signal output from the codec to the processor
C6416 DSK DAC	Use and configure the codec to convert digital input to analog output
C6416 DSK DIP Switch	Simulate or read the user-defined DIP switches on the C6416 DSK

Blocks in the C6416 DSP Library

Block	Description
C6416 DSK LED	Control the user-configurable light emitting diodes on the C6416 DSK
C6416 DSK RESET	Reset the C6416 DSP Stater Kit to initial conditions

Blocks in the C6713 DSP Library

Block	Description
C6713 DSK ADC	Configure digitized signal output from the codec to the processor
C6713 DSK DAC	Use and configure the codec to convert digital input to analog output
C6713 DSK DIP Switch	Simulate or read the user-defined DIP switches on the C6713 DSK
C6713 DSK LED	Control the user-configurable light emitting diodes on the C6713 DSK
C6713 DSK RESET	Reset the C6713 DSP Starter Kit to initial conditions

DSP Blocks in the DM642 EVM Library

Block	Description
DM642 EVM Video ADC	Configure video capture capability (video decoders) to capture analog video input on the DM642 Evaluation Module
DM642 EVM Video DAC	Configure the video display capability (video encoder) on the DM642 Evaluation Module

Block	Description
DM642 EVM LED	Control the eight light-emitting diodes on the DM642 Evaluation Module
DM642 EVM Reset	Reset the DM642 Evaluation Module to initial conditions

Blocks in the C6000 DSP Core Support Library

Block	Description
From Memory	Get data from a specific memory location into your code running on the C6000 target
To Memory	Send data from your model running in the processor to memory on your C6000 target

Blocks in the TMDX326040A DSP Support Library

Block	Description
TMDX326040 ADC	Configure the codec on the TMDX326040A daughter card to generate a stream of digital data to the processor on the C6711 DSK
TMDX326040 DAC	Configure the codec on the TMDX326040A daughter card to send data to the analog output on the card

Blocks—Alphabetical List

The following pages list the blocks included in the Embedded Target for TI C6000 DSP block libraries, listed in alphabetical order by the block name.

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
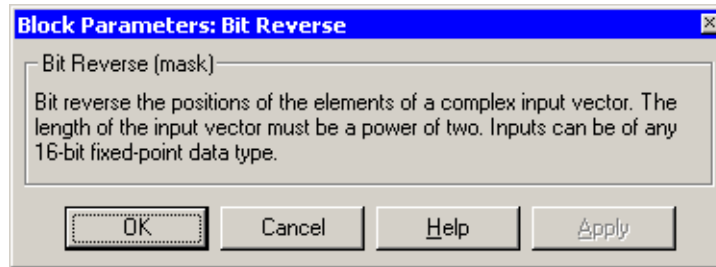
Algorithm   In simulation, the Bit Reverse block is equivalent to the
                    TMS320C62x DSP Library assembly code function `DSP_bitrev_cplx`. During
                    code generation, this block calls the `DSP_bitrev_cplx` routine to produce
                    optimized code.

Examples   The Bit Reverse block reorders the output of the C62xRadix-2 FFT in the model
                    below to natural order.

                    The following code calculates the same FFT in the workspace. The output from
                    this calculation, y2, is displayed side-by-side with the output from the model, c.

The outputs match, showing that the Bit Reverse block reorders the Radix-2 FFT output to natural order:

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

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

See Also   C62xRadix-2 FFT, C62xRadix-2 IFFT

C62x Block Exponent

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

Library   C62x DSP Library—Math and Matrices
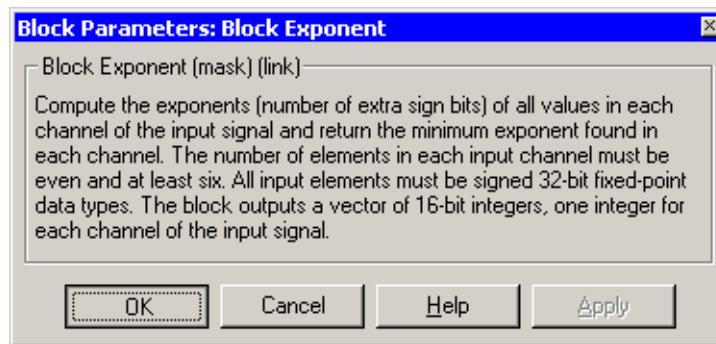
Description   The Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be even and at least six. All input elements 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

Block Parameters: Block Exponent

Block Exponent (mask) (link)

Compute the exponents (number of extra sign bits) of all values in each channel of the input signal and return the minimum exponent found in each channel. The number of elements in each input channel must be even and at least six. All input elements must be signed 32-bit fixed-point data types. The block outputs a vector of 16-bit integers, one integer for each channel of the input signal.

OK   Cancel   Help   Apply

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
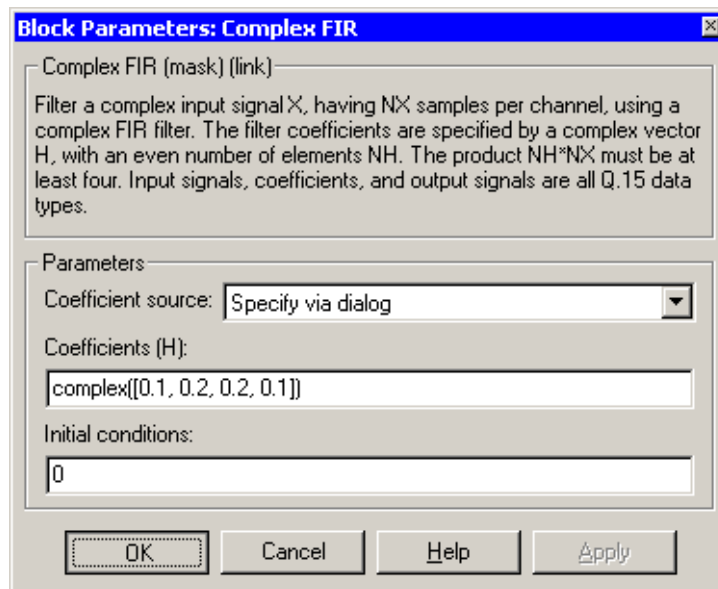
Coefficient source

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_cplx`. During code generation, this block calls the `DSP_fir_cplx` routine to produce optimized code.
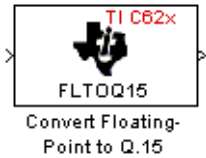
See Also   C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

Purpose   Convert a single-precision floating-point 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_fltoq15. During code generation, this block calls the DSP_fltoq15 routine to produce optimized code.

See Also   C62xConvert Q.15 to Floating Point

C62x Convert Q.15 to Floating-Point

Purpose   Convert a Q.15 fixed-point signal to a single-precision floating-point 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_q15tofl. During code
generation, this block calls the DSP_q15tofl routine to produce optimized code.

See Also   C62xConvert 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).

$$OutputFractionalBits \ = \ InputFractionalBits - (S - 1)$$

The FFT block supports both continuous and discrete sample times. This block
supports little-endian code generation.

C62x FFT

Dialog Box
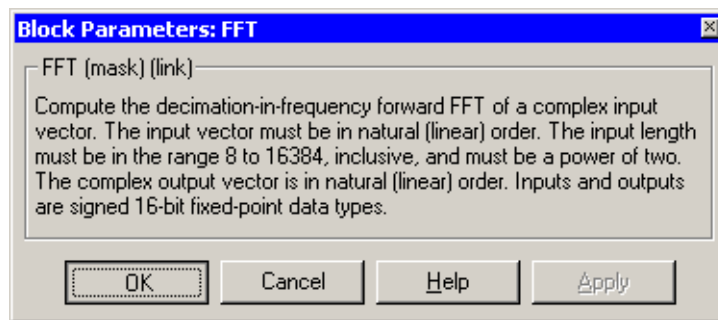
Algorithm   In simulation, the FFT block is equivalent to the TMS320C62x DSP Library assembly code function DSP_fft16x16r. During code generation, this block calls the DSP_fft16x16r routine to produce optimized code.

See Also   C62xRadix-2 FFT, C62xRadix-2 IFFT

Purpose   Filter 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
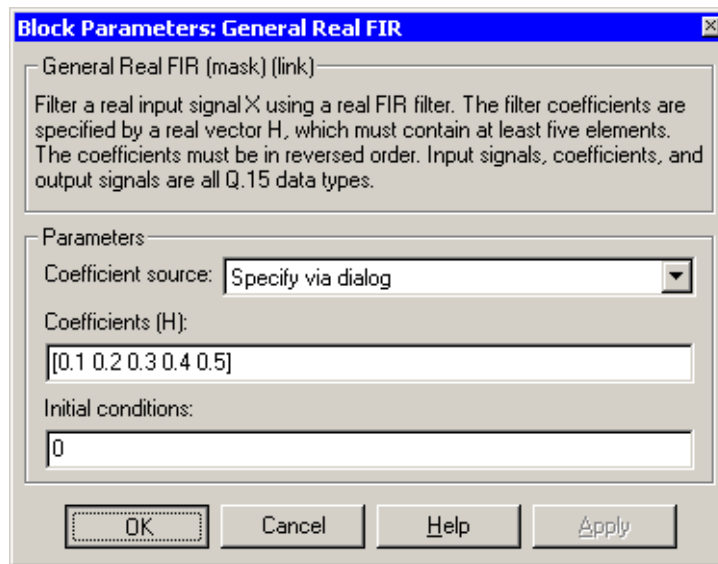
Coefficient source

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   C62xComplex FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real 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 $\overline{H}$ 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

$$e(n + 1) = d(n + 1) - [\overline{H}(n) \cdot \overline{X}(n + 1)]$$

and

$$\overline{H}(n + 1) = \overline{H}(n) + [\mu e(n + 1) \cdot \overline{X}(n + 1)]$$

where

- n designates the time step.
- $\overline{X}$ 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.
- $\overline{H}$ is a vector composed of the current set of filter taps.
- e is the error, or $d - [\overline{H}(n) \cdot \overline{X}(n + 1)]$.
- μ is the step size.

For this block, the input B and the output R are defined by

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

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

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

C62x LMS Adaptive FIR

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

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

d and B must be produced externally to the LMS Adaptive FIR block. Refer to Examples below for a sample model that does this.

The LMS Adaptive FIR block supports discrete sample times and both little-endian and big-endian code generation.

Dialog Box
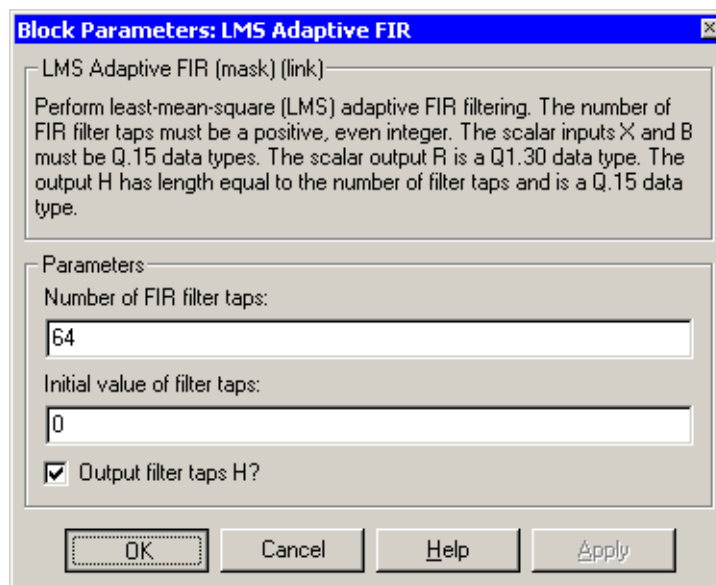
Number of FIR filter taps

    Designate the number of filter taps. The number of taps must be a positive, even integer.

Initial value of filter taps

    Enter the initial value of the filter taps.

Output filter coefficients H?

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

Algorithm   In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_firlms2`. During code generation, this block calls the `DSP_firlms2` routine to produce optimized code.

Examples   The following model uses the LMS Adaptive FIR block.

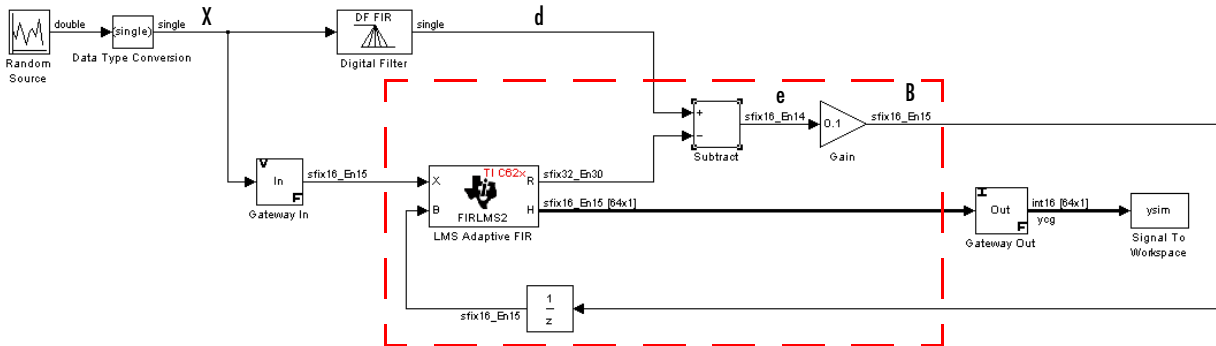The portion of the model enclosed by the dashed line produces the signal B and feeds it back into the LMS Adaptive FIR block. The inputs to this region are $\overline{X}$ and the desired signal d, and the output of this region is the vector of filter taps $\overline{H}$. Thus this region of the model acts as a canonical LMS adaptive filter. For example, compare this region to the `adaptfilt.lms` function in the Filter Design Toolbox. `adaptfilt.lms` performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

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

C62x Matrix Multiply

Purpose   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.
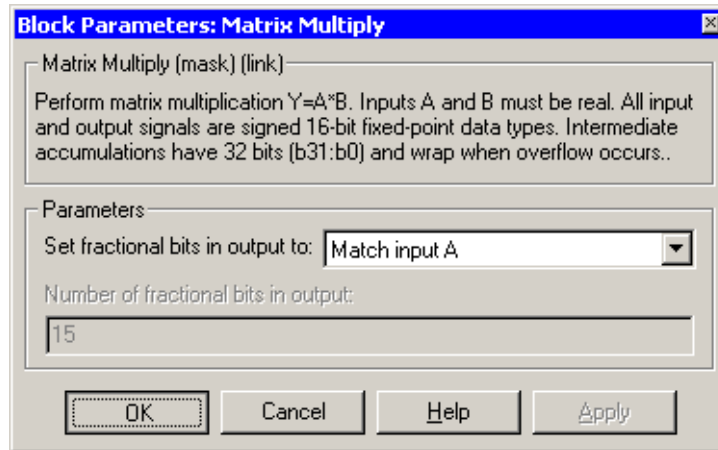
Alternatively, you can choose to output 16 bits according to how many fractional bits you want in the output. The number of fractional bits in the accumulator value is the sum of the fractional bits of the two inputs.

	Input A	Input B	Accumulator Value
Total Bits	16	16	32
Fractional Bits	R	S	$R + S$

Therefore $R+S$ is the location of the binary point in the accumulator value. You can select 16 bits in relation to this fixed position of the accumulator binary point to give the desired number of fractional bits in the output (see Examples below). You can either require the output to have the same number of fractional bits as one of the two inputs, or you can specify the number of output fractional bits in the **Number of fractional bits in output** parameter.

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

Dialog Box

Block Parameters: Matrix Multiply

─Matrix Multiply (mask) (link)─────────────────────────

Perform matrix multiplication Y=A*B. Inputs A and B must be real. All input and output signals are signed 16-bit fixed-point data types. Intermediate accumulations have 32 bits (b31:b0) and wrap when overflow occurs..

─Parameters──

Set fractional bits in output to: | Match input A ▼ |

Number of fractional bits in output:

| 15 |

| OK | Cancel | Help | Apply |

Set fractional bits in output to

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

- Match input A—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or R in the discussion above).

- Match input B—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or S in the discussion above).

- Match high bits of acc. (b31:b16)—Output the highest 16 bits of the accumulator value.

- Match high bits of prod. (b30:b15)—Output the second-highest 16 bits of the accumulator value.

- User-defined—Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

Number of fractional bits in output

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

C62x Matrix Multiply

Algorithm   In simulation, the Matrix Multiply block is equivalent to the TMS320C62x DSP Library assembly code function DSP_mat_mul. During code generation, this block calls the DSP_mat_mul routine to produce optimized code.

Examples   **Example 1** Suppose A and B are both Q.15. The data type of the resulting accumulator value is therefore the 32-bit data type Q1.30 ($R + S = 30$). In the accumulator, bits 31:30 are the sign and integer bits, and bits 29:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

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

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

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

See Also   C62xVector Multiply

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
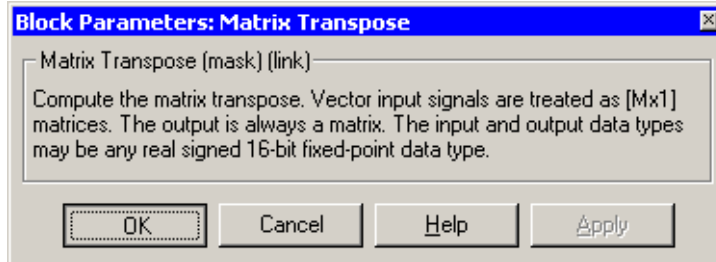
Algorithm   In simulation, the Matrix Transpose block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mat_trans`. During code generation, this block calls the `DSP_mat_trans` routine to produce optimized code.

C62x Radix-2 FFT

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

Library   C62x DSP Library—Transforms
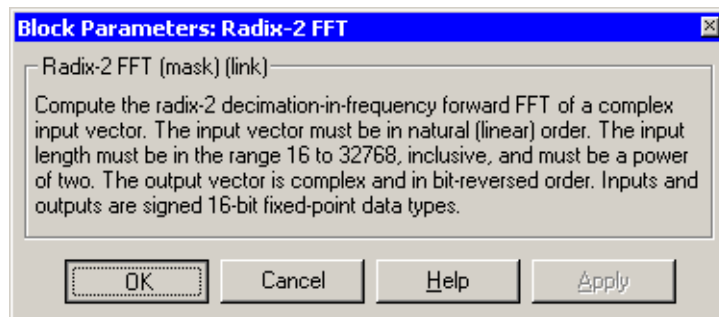
Description   The Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of 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 C62x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

The Radix-2 FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.
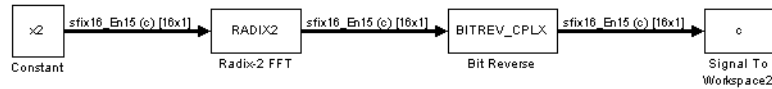
Dialog Box

Algorithm   In simulation, the Radix-2 FFT block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

Examples   The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the C62x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

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

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

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

See Also   C62x Bit Reverse, C62x FFT, C62x Radix-2 IFFT

C62x Radix-2 IFFT

Purpose   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
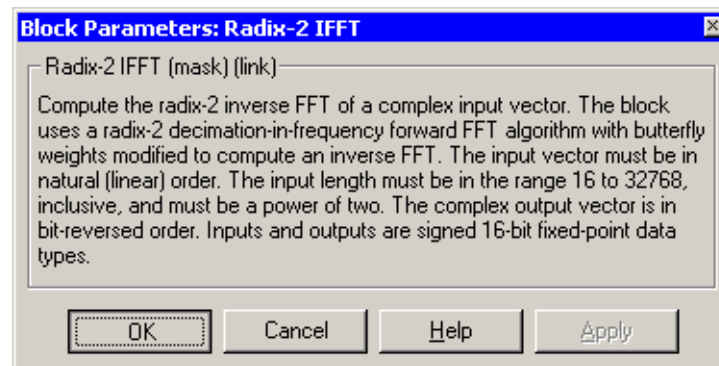outputs are signed 16-bit fixed-point data types.

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 C62x Bit Reverse block to reorder the output of the Radix-2
IFFT block to natural order.

The Radix-2 IFFT block supports both continuous and discrete sample times.
This block supports little-endian code generation.

Dialog Box

Algorithm   In simulation, the Radix-2 IFFT block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

See Also   C62x Bit Reverse, C62x FFT, C62x Radix-2 FFT

C62x Radix-4 Real FIR

Purpose   Filter a real input signal using a real FIR filter

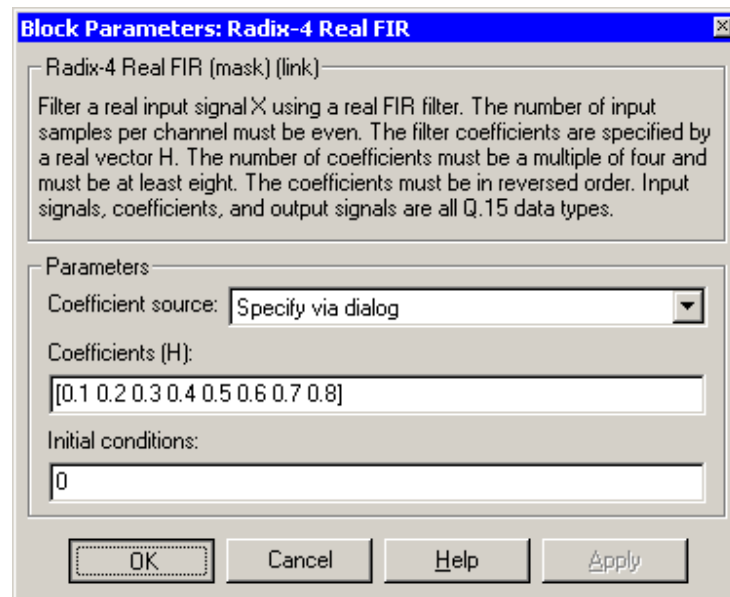**Library**   C62x DSP Library—Filtering

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

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

Coefficients (H)

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

Initial conditions

If the initial conditions are

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

Initial conditions must be real.

Algorithm   In simulation, the Radix-4 Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function DSP_fir_r4. During code generation, this block calls the DSP_fir_r4 routine to produce optimized code.

See Also   C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

C62x Radix-8 Real FIR

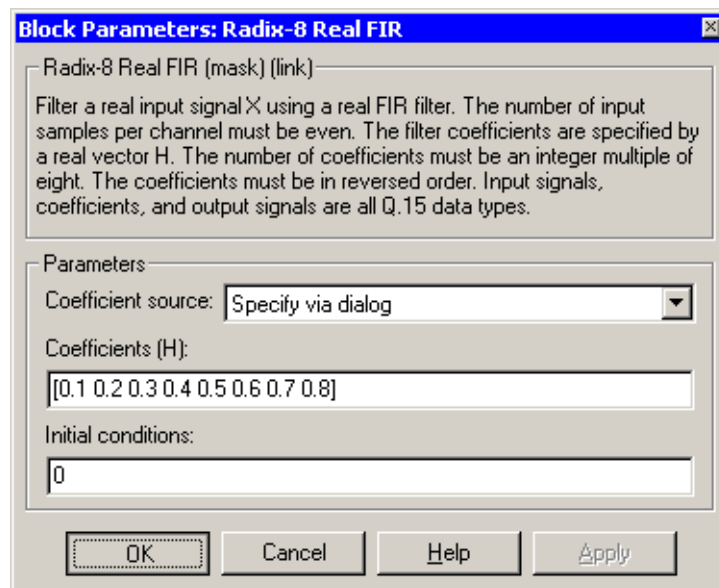|	
Purpose	Filter a real input signal using a real FIR filter
Library	C62x DSP Library—Filtering

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

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

Coefficients (H)

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

Initial conditions

If the initial conditions are

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

Initial conditions must be real.

Algorithm   In simulation, the Radix-8 Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_r8`. During code generation, this block calls the `DSP_fir_r8` routine to produce optimized code.

See Also   C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xSymmetric Real FIR

C62x Real Forward Lattice All-Pole IIR

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

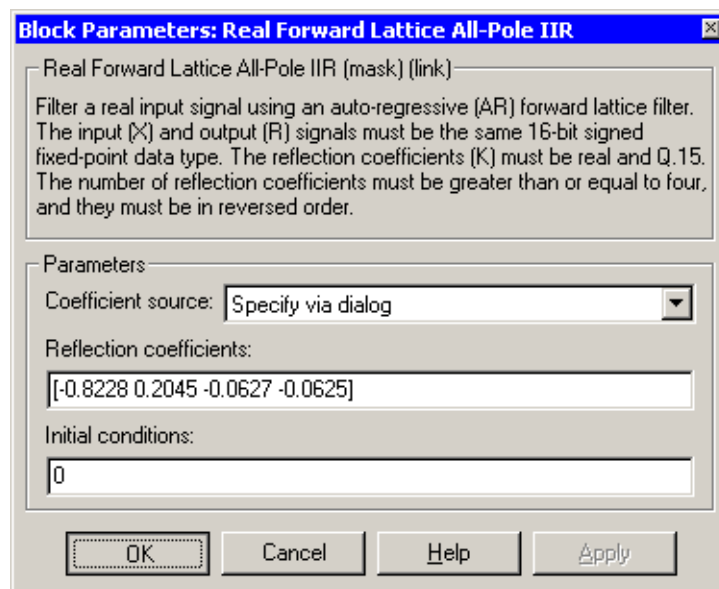**Library**   C62x DSP Library—Filtering

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

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

Reflection coefficients

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

Initial conditions

If your block initial conditions are

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

Algorithm   In simulation, the Real Forward Lattice All-Pole IIR block is equivalent to the TMS320C62x DSP Library assembly code function DSP_iirlat. During code generation, this block calls the DSP_iirlat routine to produce optimized code.
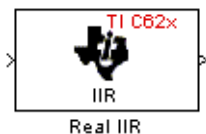
See Also   C62xReal IIR

C62x Real IIR

Purpose   Filter 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

Coefficient sources

Specify the source of the filter coefficients:

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

MA (numerator) coefficients

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

AR (denominator) coefficients

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

Input state initial conditions

If the input state initial conditions are

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

Output state initial conditions

If the output state initial conditions are

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

C62x Real IIR

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

Algorithm   In simulation, the Real IIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_iir`. During code generation, this block calls the `DSP_iir` routine to produce optimized code.

See Also   C62xReal Forward Lattice All-Pole IIR

Purpose   Compute the fractional and exponential portions of the reciprocal of a real input signal
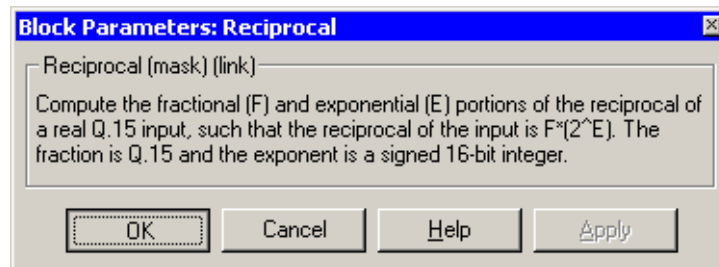
Library   C62x DSP Library—Math and Matrices

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 $F*(2^E)$. 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

Block Parameters: Reciprocal

Reciprocal (mask) (link)

Compute 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*(2^E). The fraction is Q.15 and the exponent is a signed 16-bit integer.

| OK | Cancel | Help | Apply |

Algorithm   In simulation, the Reciprocal block is equivalent to the TMS320C62x DSP Library assembly code function DSP_recip16. During code generation, this block calls the DSP_recip16 routine to produce optimized code.

C62x Symmetric Real FIR

Purpose
Filter 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.
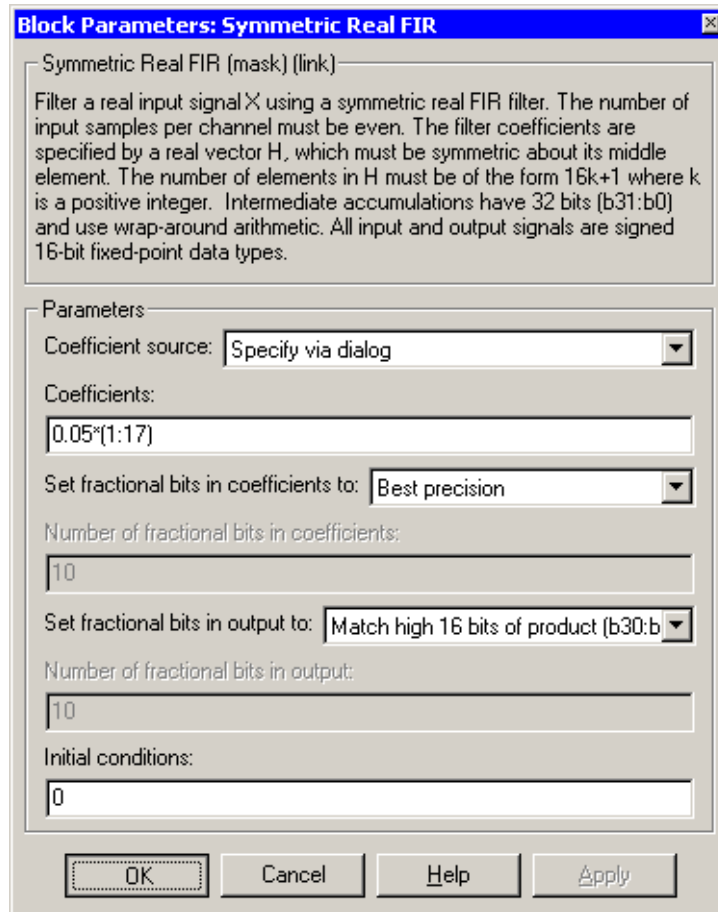
The number of input samples per channel must be even. The filter coefficients are specified by a real vector H, which must be symmetric about its middle element. The number of coefficients must be of the form 16k + 1, where k is a positive integer. This block wraps overflows that occur. The input, coefficients, and output are 16-bit signed fixed-point data types.

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

Match input x	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the input
Match coefficients h	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the coefficients
Match high 16 bits of acc.	Output bits 31 - 16 of the accumulator value
Match high 16 bits of prod.	Output bits 30 - 15 of the accumulator value
User-defined	Output 16 bits of the accumulator value such that the output has the number of fractional bits specified in the **Number of fractional bits in output** parameter

The Symmetric Real FIR block supports discrete sample times and only little-endian code generation.

Dialog Box

Coefficient source

Specify the source of the filter coefficients:

- Specify via dialog—Enter the coefficients in the **Coefficients** parameter in the dialog
- Input port—Accept the coefficients from port H

Coefficients

Enter the coefficients in vector format. This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter. This parameter is tunable in simulation.

Set fractional bits in coefficients to

Specify the number of fractional bits in the filter coefficients:

- `Match input X`—Sets the coefficients to have the same number of fractional bits as the input
- `Best precision`—Sets the number of fractional bits of the coefficients such that the coefficients are represented to the best precision possible
- `User-defined`—Sets the number of fractional bits in the coefficients with the **Number of fractional bits in coefficients** parameter

This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter.

Number of fractional bits in coefficients

Specify the number of bits to the right of the binary point in the filter coefficients. This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter, and is only enabled if `User-defined` is specified for the **Set fractional bits in coefficients to** parameter.

Set fractional bits in output to

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

- `Match input X`—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input X
- `Match coefficients H`—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in coefficients H
- `Match high bits of acc. (b31:b16)`—Output the highest 16 bits of the accumulator value
- `Match high bits of prod. (b30:b15)`—Output the second-highest 16 bits of the accumulator value

- • `User-defined`—Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter

See Matrix Multiply "Examples" on page 5-30 for demonstrations of these selections.

Number of fractional bits in output

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

Initial conditions

If the initial conditions are

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

Algorithm   In simulation, the Symmetric Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_sym`. During code generation, this block calls the `DSP_fir_sym` routine to produce optimized code.

See Also   C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR

C62x Vector Dot Product

Purpose   Compute the vector dot product of two real input signals

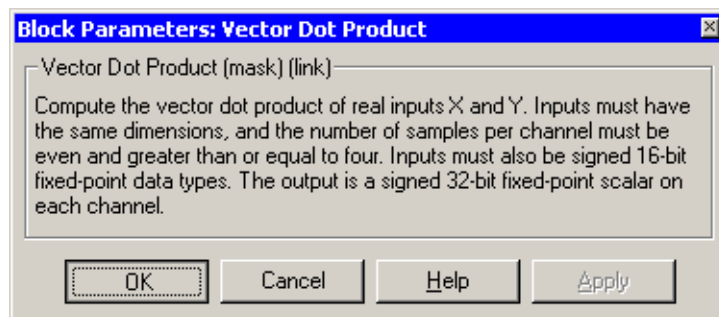**Library**   C62x DSP Library—Math and Matrices

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.

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

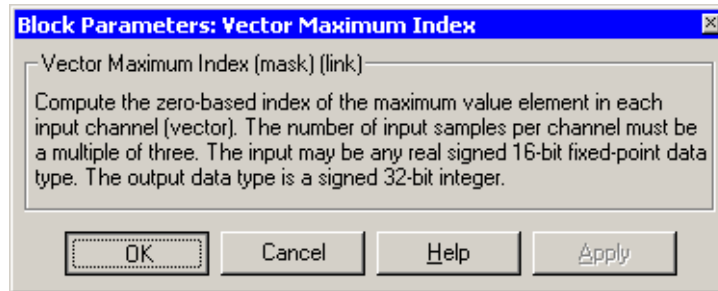**Library**   C62x DSP Library—Math and Matrices

Description   The Vector Maximum Index block computes the zero-based index of the maximum value element in each channel (vector) of the input signal. The input may be any real, 16-bit, signed fixed-point data type, and the number of samples per input channel must be an integer multiple of three. The output data type is a 32-bit signed integer.

The Vector Maximum Index block supports both continuous and discrete sample times. This block also supports both little-endian and big-endian code generation.

Dialog Box

Algorithm   In simulation, the Vector Maximum Index block is equivalent to the TMS320C62x DSP Library assembly code function DSP_maxidx. During code generation, this block calls the DSP_maxidx routine to produce optimized code.

C62x Vector Maximum Value

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

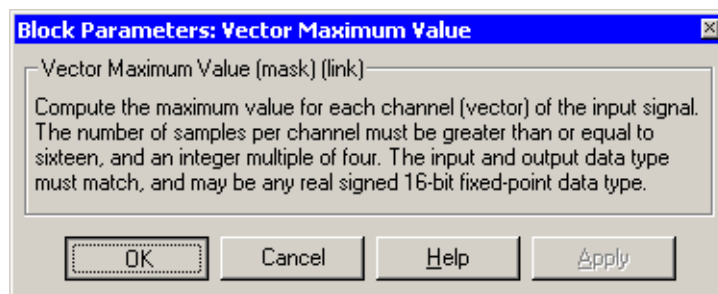**Library**   C62x DSP Library—Math and Matrices

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   C62xVector Minimum Value

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

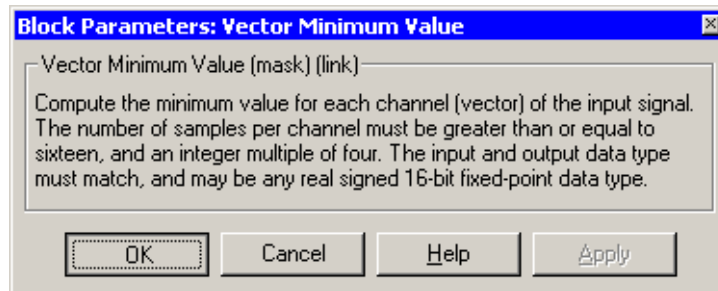**Library**   C62x DSP Library—Math and Matrices

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   C62xVector Maximum Value

C62x Vector Multiply

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
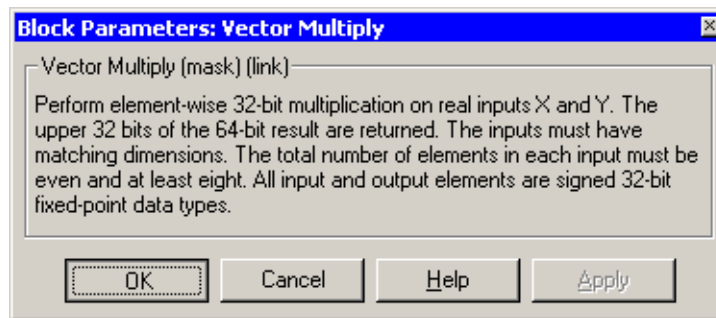
Algorithm   In simulation, the Vector Multiply block is equivalent to the TMS320C62x DSP Library assembly code function DSP_mul32. During code generation, this block calls the DSP_mul32 routine to produce optimized code.

See Also   C62xMatrix Multiply

Purpose   Negate each element of an input signal

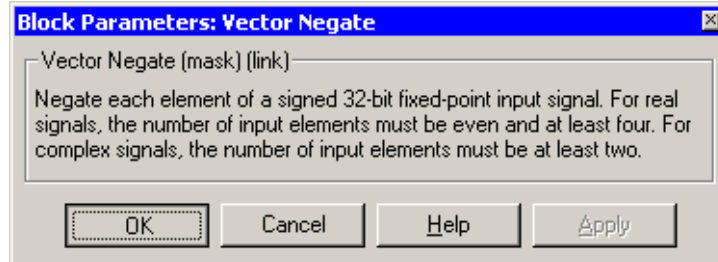**Library**   C62x DSP Library—Math and Matrices

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

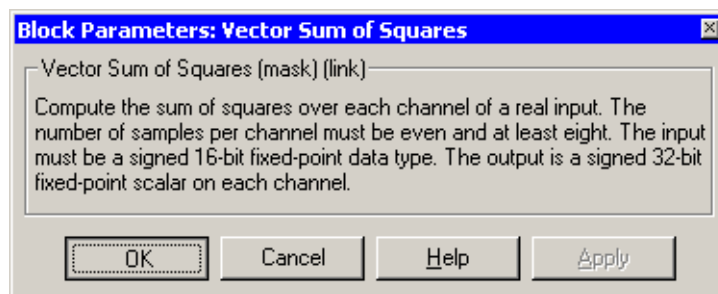**Library**   C62x DSP Library—Math and Matrices

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.

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.
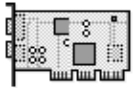
Purpose   Set target preferences and memory map to generate code for the C6416 DSP
                   Starter Kit

Library   Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

Description   Options on the block mask let you set features of code generation for your

C6416DSK

C6416 DSP Starter Kit target. Adding this block to your Simulink model
provides access to the processor hardware settings you need to configure when
you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6416 DSK must include this block, or the
Custom C6000 target preferences block. Real-Time Workshop returns an error
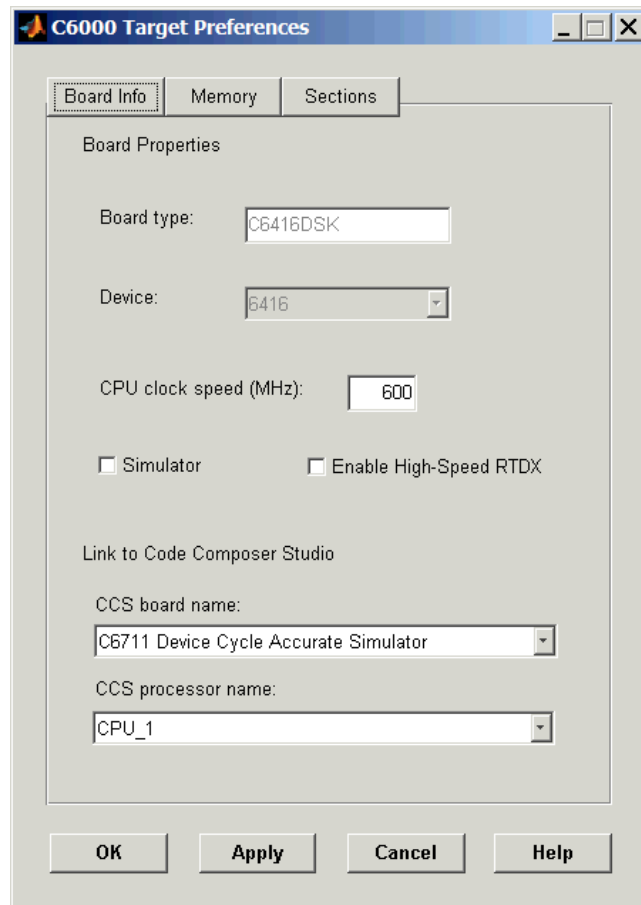message if a target preferences block is not present in your model.

Note This block must be in your model at the top level and not in a
subsystem. It does not connect to any other blocks, but stands alone to set the
target preferences for the model.

Among the processor and target options you specify here are the target board
information, memory mapping and layout, and how to allocate the various code
sections, such as compiler, DSP/BIOS, and custom sections.

Setting the options included in this dialog results in identifying your target to
Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and
configuring the memory map for your target. Both are essential steps in the
process of targeting any board, custom or explicitly supported like the C6711
DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog for this block unless
you add the block to a model. When you try to open the block dialog, the block
attempts to connect to your target. It cannot make the connection when the
block is in the library and returns an error message.

C6416DSK

Dialog Box

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

- **Board info**—select the target board and processor, set the clock speed, and identify the target.
- **Memory**—set the memory allocation and layout on the target processor (memory mapping).

- **Sections**—determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS information and where to put compiler information.

Board Info Pane

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

Board Type

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

Device

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

CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results may be wrong, and code profiling results will not be correct.

You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

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

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. 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 cannot change the clock rate. Other C6000 processors allow different clock speeds.

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

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

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

To create sine block interrupts at 0.001 s/sample requires

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

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

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

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

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

CCS Processor Name
Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one

processor. In the multiprocessor case, you select the processor by name from the list.

Memory Pane

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

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

C6416DSK

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

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

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

Physical Memory Options

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

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

- Custom boards based on C670x processors provide IPRAM and IDRAM memory segments by default.
- C6701 EVM boards provide IPRAM, IDRAM, SBSRAM, SDDRAM0, and SDRAM1 memory segments by default
- C6711DSK boards provide SDRAM memory segments by default.

Name

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

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

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

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

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

Address

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

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

Length

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

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

Contents

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

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

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

Add

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

Remove

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

Create Heap

If your processor supports using a heap, as do the C6711 or C6701, for example, selecting this option enables creating the heap, and enables the **Heap size** option. **Create heap** is not available on processors that either do not provide a heap or do not allow you to configure the heap.

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

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

Heap Size

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

Define Label

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

Heap Label

Enabled by selecting **Define label**, you use this option to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Enable L2 Cache

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

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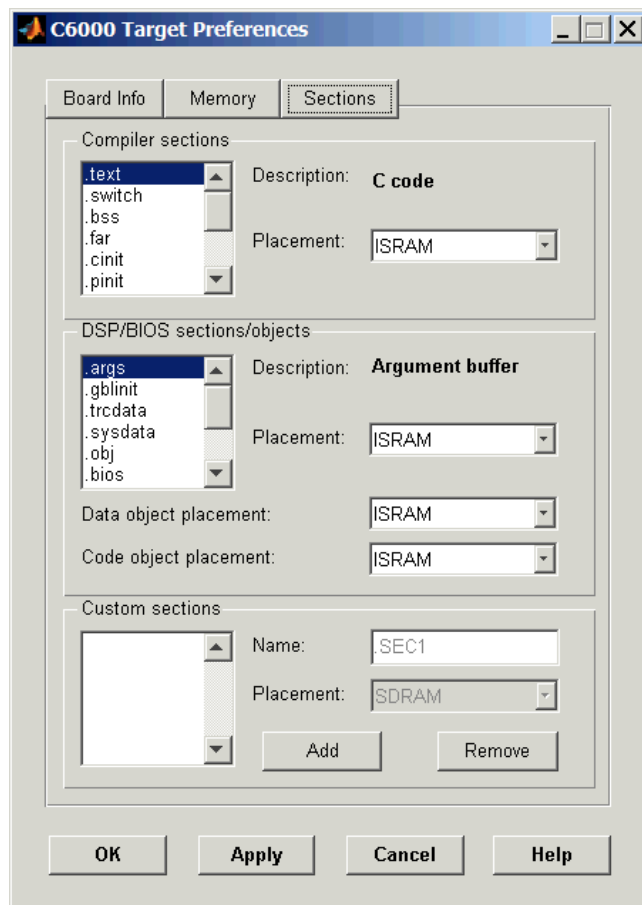
L2 Cache size

Once you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

Sections Pane

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

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

C6416DSK

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

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

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

String	Section List	Description of the Section Contents
.switch	Compiler	Jump tables for switch statements in the executable code
.sysdata	DSP/BIOS	Data about DSP/BIOS
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code
.sysmem	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

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

- .cinit
- .const
- .switch
- .text—created by the assembler.

These sections are uninitialized:

- .bss—created by the assembler.
- .far
- .stack

- .sysmem

Other sections appear on the list as well:

- .data—created by the assembler. The C/C++ compiler does not use this section.
- .cio
- .pinit

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

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

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

DSP/BIOS Sections

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

Description

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

Placement

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

DSP/BIOS Object Placement

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

Custom Sections

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

Name

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

Placement

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

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

To remove a section from the **Custom sections** list, select the section to remove and click **Remove**. The selected section disappears from the list.
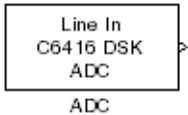
See Also   Custom C6000

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

Line In
C6416 DSK
ADC

ADC

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.

5-73

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.

Dialog Box
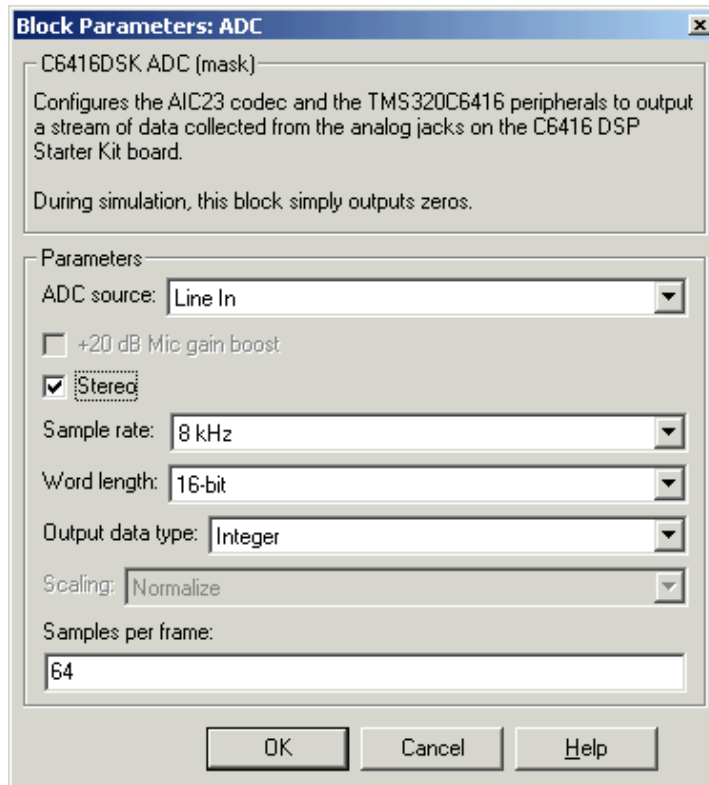
ADC source

> 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.

C6416 DSK ADC

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 ±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.
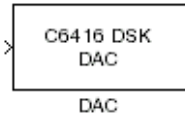
See Also   C6416 DSK DAC

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.

Option	Affected Hardware
Overflow mode	TMS320C6416 Digital Signal Processor
Scaling	TMS320C6416 Digital Signal Processor
Word Length	Codec

C6416 DSK DAC

Dialog Box
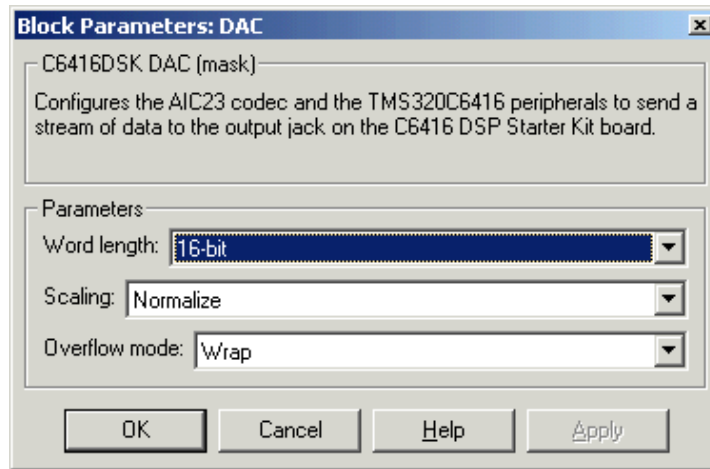
Word length

    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 ±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.
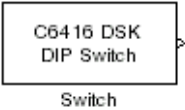
See Also   C6416 DSK ADC

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.

```
C6416 DSK
DIP Switch

   Switch
```

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

Output Values From The User DIP Switches on the C6416 DSK (Continued)

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
On	On	Off	Off	0011	3
Off	Off	On	Off	0100	4
On	Off	On	Off	0101	5
Off	On	On	Off	0110	6
On	On	On	Off	0111	7
Off	Off	Off	On	1000	8
On	Off	Off	On	1001	9
Off	On	Off	On	1010	10
On	On	Off	On	1011	11
Off	Off	On	On	1100	12
On	Off	On	On	1101	13
Off	On	On	On	1110	14
On	On	On	On	1111	15

C6416 DSK DIP Switch

Dialog Box
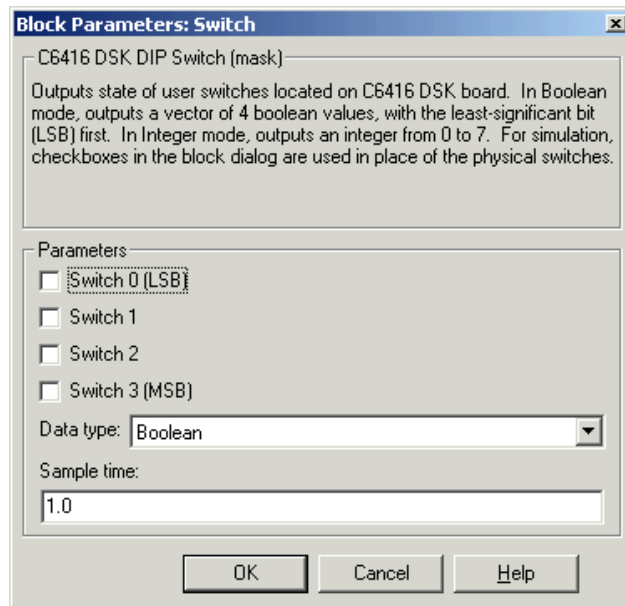
Opening this dialog causes a running simulation to pause. Refer to "Changing Source Block Parameters" in your online Simulink documentation for details.

Switch 0

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

Switch 1

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

Switch 2

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

Switch 3

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

Data type

Determines how the block reports the status of the user-defined DIP switches. Boolean is the default, indicating that the output is a vector of four logical values.

Each vector element represents the status of one DIP switch; the first is **Switch 0** and the fourth is **Switch 3**. The data type `Integer` converts the logical string to an equivalent unsigned 8-bit (`uint8`) value. For example, when the logical string generated by the switches is 0101, the conversion yields 5—the MSB is 0 and the LSB is 1.

Sample time

Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

C6416 DSK LED

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

C6416 DSK
LED

LED
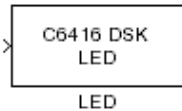
Adding the C6416 DSK LED block to your Simulink block diagram lets you trigger the user light emitting diodes (LED) on the C6416 DSK. To use the block, send a nonzero real scalar to the block. The C6416 DSK LED block controls all four user LEDs located on the C6416 DSK.

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:
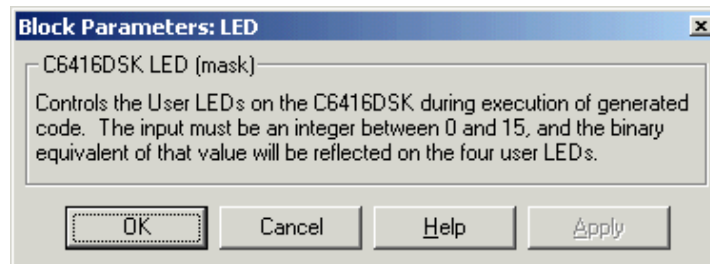
• When the block receives an input value equal to 0, the specified LEDs are turned off (disabled), 0000

• 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

Block Parameters: LED

C6416DSK LED (mask)

Controls the User LEDs on the C6416DSK during execution of generated code. The input must be an integer between 0 and 15, and the binary equivalent of that value will be reflected on the four user LEDs.

| OK | Cancel | Help | Apply |

This dialog does not have any user-selectable options.

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.

C64x Autocorrelation

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

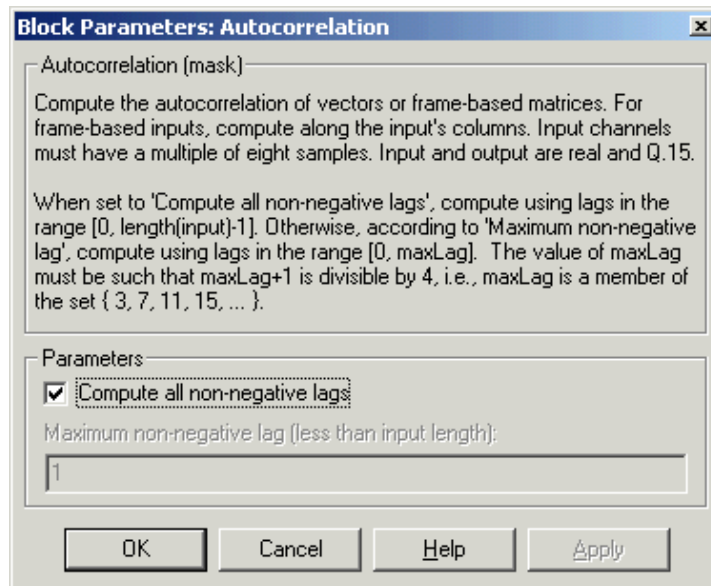**Library**   C64x DSP Library—Math and Matrices

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, 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, and (maxLag+1) must be divisible by 4, such as maxLag equal to 3, 7, or 19. This parameter is enabled when you clear the **Compute all non-negative lags** parameter.

Algorithm

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

C64x Bit Reverse

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

Block Parameters: Bit Reverse

Bit Reverse (mask)

Bit reverse the positions of the elements of a complex input vector. The length of the input vector must be a power of two. Inputs can be any 16-bit fixed-point data type.

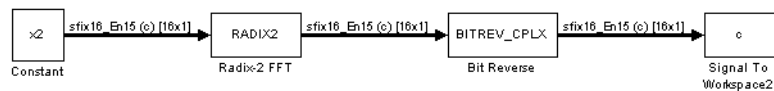   OK   Cancel   Help   Apply
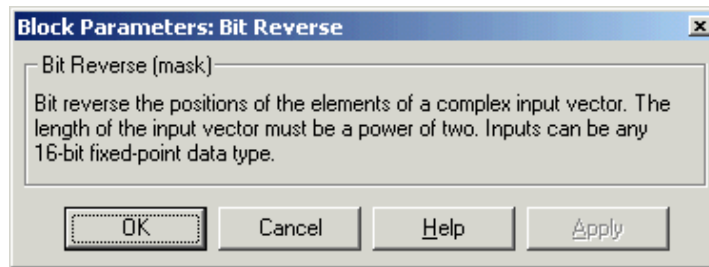
Algorithm   In simulation, the Bit Reverse block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_bitrev_cplx`. During code generation, this block calls the `DSP_bitrev_cplx` routine to produce optimized code.

Examples   The Bit Reverse block reorders the output of the C64x Radix-2 FFT in the model below to natural order.

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

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

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

See Also   C64x Radix-2 FFT, C64x Radix-2 IFFT

C64x Block Exponent

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

Library   C64x DSP Library—Math and Matrices
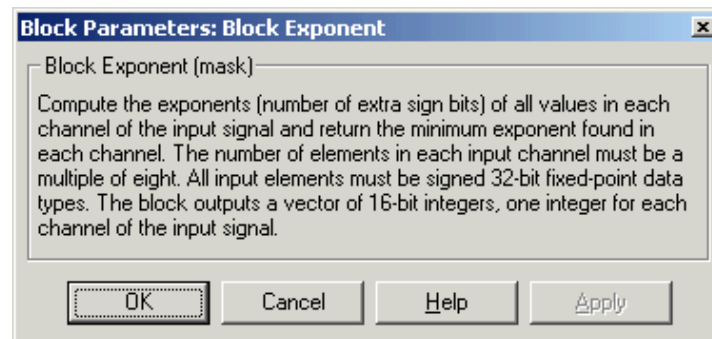
Description   The C64x Block Exponent block first computes the number of extra sign bits of
                   all values in each channel of an input signal, and then returns the minimum
                   number of sign bits found in each channel. The number of elements in each
                   input 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

Block Parameters: Block Exponent

─ Block Exponent (mask) ─

Compute the exponents (number of extra sign bits) of all values in each
channel of the input signal and return the minimum exponent found in
each channel. The number of elements in each input channel must be a
multiple of eight. All input elements must be signed 32-bit fixed-point data
types. The block outputs a vector of 16-bit integers, one integer for each
channel of the input signal.

OK   Cancel   Help   Apply

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

Purpose   Filter a complex input signal using a complex FIR filter

Library   C64x DSP Library—Filtering
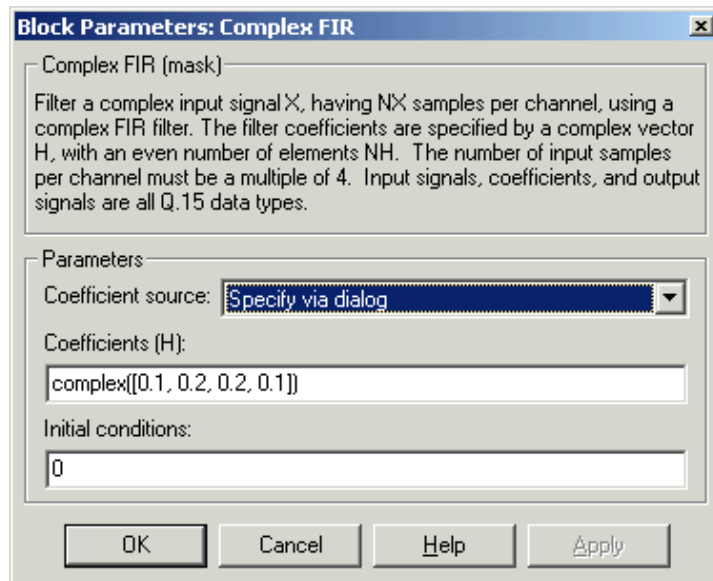
Description   The C64x Complex FIR block filters a complex input signal X using a complex
                    FIR filter. This filter is implemented using a direct form structure. Each input
                    channel must contain an integer multiple of four samples, with four samples as
                    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

Block Parameters: Complex FIR

Complex FIR (mask)

Filter a complex input signal X, having NX samples per channel, using a
complex FIR filter. The filter coefficients are specified by a complex vector
H, with an even number of elements NH. The number of input samples
per channel must be a multiple of 4. Input signals, coefficients, and output
signals are all Q.15 data types.

Parameters

Coefficient source: Specify via dialog

Coefficients (H):

complex([0.1, 0.2, 0.2, 0.1])

Initial conditions:

0

| OK | Cancel | Help | Apply |

Coefficient source
    Specify the source of the filter coefficients:

C64x Complex FIR

- `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_cplx`. During code generation, this block calls the `DSP_fir_cplx` routine to produce optimized code.
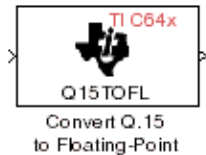
See Also

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

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_fltoq15. During code
generation, this block calls the DSP_fltoq15 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_q15tofl`. During code
generation, this block calls the `DSP_q15tofl` routine to produce optimized code.

See Also   C64x Convert Floating-Point to Q.15

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).

$$OutputFractionalBits = InputFractionalBits - (S - 1)$$

The FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

5-95

C64x FFT

Dialog Box

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

See Also   C64x Radix-2 FFT, C64x Radix-2 IFFT

Purpose   Filter a real input signal using a real FIR filter
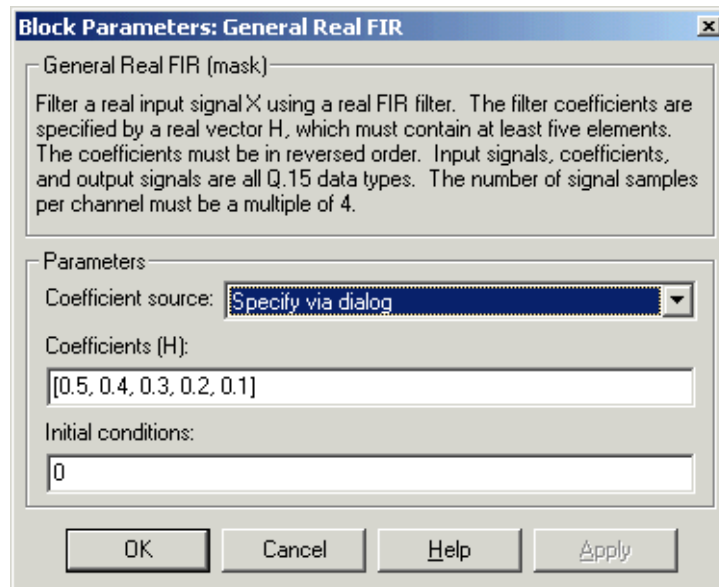
Library   C64x DSP Library—Filtering

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

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 $\overline{H}$ 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

$$e(n+1) = d(n+1) - [\overline{H}(n) \cdot \overline{X}(n+1)]$$

and

$$\overline{H}(n+1) = \overline{H}(n) + [\mu e(n+1) \cdot \overline{X}(n+1)]$$

where

- n designates the time step.
- $\overline{X}$ 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.
- $\overline{H}$ is a vector composed of the current set of filter taps.
- e is the error, or $d - [\overline{H}(n) \cdot \overline{X}(n+1)]$.
- μ is the step size.

For this block, the input B and the output R are defined by

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

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

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

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

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

d and B must be produced externally to the LMS Adaptive FIR block. See "Examples" below for a sample model where this is done.

The LMS Adaptive FIR block supports discrete sample times and both little-endian and big-endian code generation.

Dialog Box
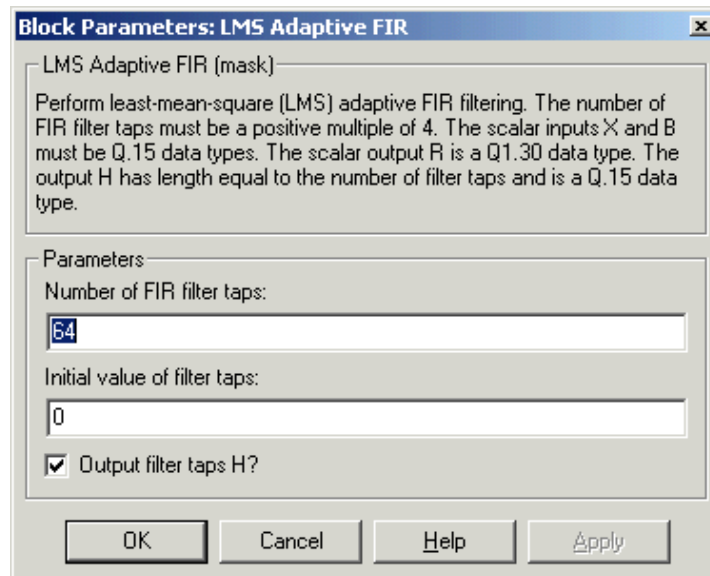
Number of FIR filter taps

Designate the number of filter taps. The number of taps must be a positive integer that is also a multiple of four.

Initial value of filter taps

Enter the initial value of the filter taps.

Output filter coefficients H?

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

Algorithm   In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_firlms2`. During code generation, this block calls the `DSP_firlms2` routine to produce optimized code.

Examples   The following model uses the LMS Adaptive FIR block.

The portion of the model enclosed by the dashed line produces the signal B and feeds it back into the LMS Adaptive FIR block. The inputs to this region are $\overline{X}$ and the desired signal d, and the output of this region is the vector of filter taps $\overline{H}$. Thus this region of the model acts as a canonical LMS adaptive filter. For example, compare this region to the `adaptfilt.lms` function in the Filter Design Toolbox. `adaptfilt.lms` performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

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

C64x Matrix Multiply

Purpose   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.

	Input A	Input B	Accumulator Value
Total Bits	16	16	32
Fractional Bits	R	S	$R + S$

Therefore $R+S$ is the location of the binary point in the accumulator value. You can select 16 bits in relation to this fixed position of the accumulator binary point to give the desired number of fractional bits in the output (see "Examples" below). You can either require the output to have the same number of fractional bits as one of the two inputs, or you can specify the number of output fractional bits in the **Number of fractional bits in output** parameter.

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

Dialog Box

Block Parameters: Matrix Multiply

Matrix Multiply (mask)

Perform matrix multiplication Y=A*B. Inputs A and B must be real. All input and output signals are signed 16-bit fixed-point data types. Intermediate accumulations have 32 bits (b31:b0) and wrap when overflow occurs..

Parameters

Set fractional bits in output to: Match input A

Number of fractional bits in output:

15

OK   Cancel   Help   Apply

Set fractional bits in output to

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

- Match input A—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or R in the discussion above).

- Match input B—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or S in the discussion above).

- Match high bits of acc. (b31:b16)—Output the highest 16 bits of the accumulator value.

- Match high bits of prod. (b30:b15)—Output the second-highest 16 bits of the accumulator value.

- User-defined—Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is enabled only when you select User-defined for **Set fractional bits in output to**.
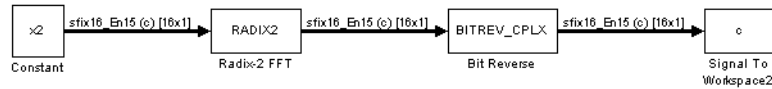
C64x Matrix Multiply

Algorithm   In simulation, the Matrix Multiply block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_mat_mul`. During code generation, this block calls the `DSP_mat_mul` routine to produce optimized code.

Examples   **Example 1** Suppose A and B are both Q.15. The data type of the resulting accumulator value is therefore the 32-bit data type Q1.30 ($R + S = 30$). In the accumulator, bits 31:30 are the sign and integer bits, and bits 29:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

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

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

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

See Also   C64x Vector Multiply

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
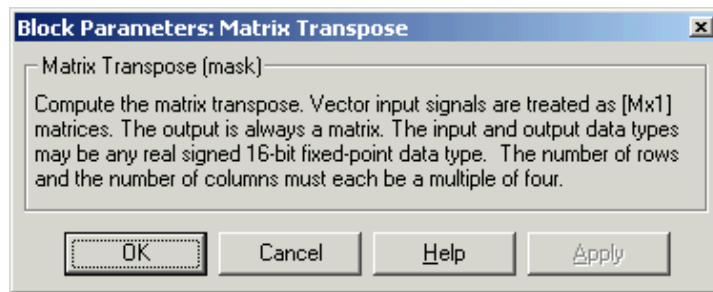
Algorithm   In simulation, the Matrix Transpose block is equivalent to the TMS320C64x DSP Library assembly code function DSP_mat_trans. During code generation, this block calls the DSP_mat_trans routine to produce optimized code.

C64x Radix-2 FFT

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.
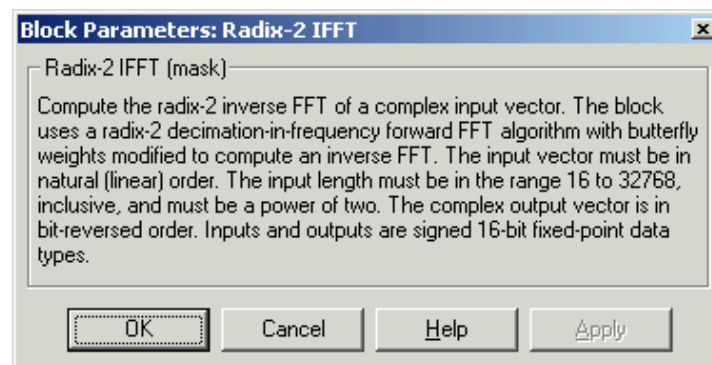
You can use the C64x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

The Radix-2 FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box

Block Parameters: Radix-2 FFT

┌─ Radix-2 FFT (mask) ─────────────────────────────────┐
Compute the radix-2 decimation-in-frequency forward FFT of a complex input vector. The input vector must be in natural (linear) order. The input length must be in the range 16 to 32768, inclusive, and must be a power of two. The output vector is complex and in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types.
└──┘

           OK   Cancel   Help   Apply

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

Examples   The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the C64x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

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

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

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

See Also   C64x Bit Reverse, C64x FFT, C64x Radix-2 IFFT

C64x Radix-2 IFFT

Purpose　　　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 C64x Bit Reverse block to reorder the output of the Radix-2
IFFT block to natural order.

The Radix-2 IFFT block supports both continuous and discrete sample times.
This block supports little-endian code generation.

Dialog Box

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

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

C64x Radix-4 Real FIR

Purpose   Filter a real input signal using a real FIR filter

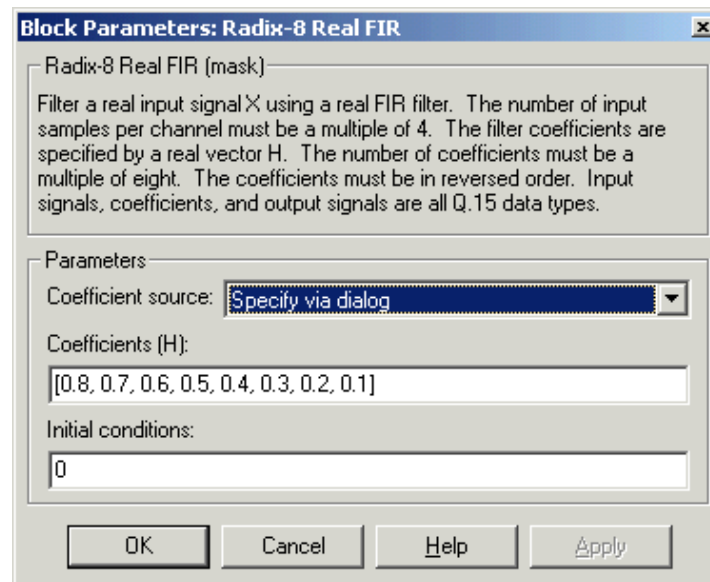**Library**   C64x DSP Library—Filtering

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),…,(b(0)}. All inputs, coefficients, and outputs are Q.15 signals.

The Radix-4 Real FIR block supports discrete sample times and both little-endian and big-endian code generation.

Dialog Box

Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog`—Enter the coefficients in the **Coefficients** parameter in the dialog

- `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),...,(b(0)*. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, enter a scalar.

- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.

- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

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

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

C64x Radix-8 Real FIR

Purpose   Filter 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),…,(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

- 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),…, k(0). Using an even number of reflection coefficients maximizes the speed of your generated code.

The Real Forward Lattice All-Pole IIR block supports discrete sample times and both little-endian and big-endian code generation.

Dialog Box
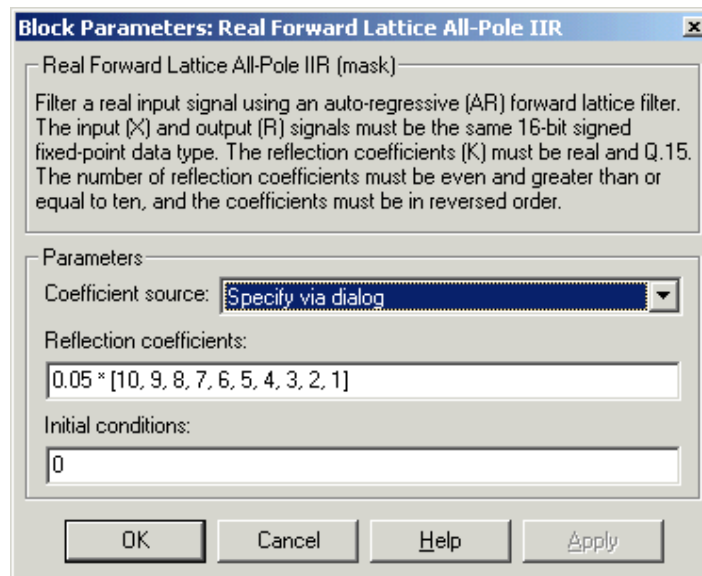
Coefficient source

   Specify the source of the filter coefficients:

- `Specify via dialog`—Enter the coefficients in the **Reflection coefficients** parameter in the dialog
- `Input port`—Accept the coefficients from port K

Reflection coefficients

Designate the reflection coefficients of the filter in vector format. The number of coefficients must be greater than or equal to ten and be even. Enter the coefficients in reverse order from k(n) to k(0). Using an even number of reflection coefficients maximizes the speed of your generated code. This parameter is visible when you select `Specify via dialog` for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If your block initial conditions are

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

Algorithm

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

See Also

C64x Real IIR

C64x Real IIR

Purpose   Filter 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

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 $F*(2^E)$. 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

Block Parameters: Reciprocal

Reciprocal (mask)

Compute 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*(2^E). The fraction is Q.15 and the exponent is a signed 16-bit integer.

OK   Cancel   Help   Apply
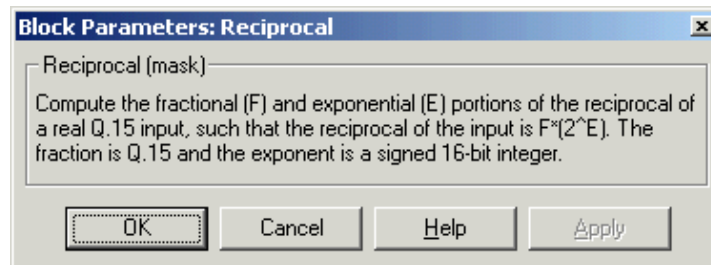
Algorithm   In simulation, the Reciprocal block is equivalent to the TMS320C64x DSP Library assembly code function DSP_recip16. During code generation, this block calls the DSP_recip16 routine to produce optimized code.

C64x Symmetric Real FIR

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

Match input x	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the input
Match coefficients h	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the coefficients
Match high 16 bits of acc.	Output bits 31 - 16 of the accumulator value
Match high 16 bits of prod.	Output bits 30 - 15 of the accumulator value
User-defined	Output 16 bits of the accumulator value such that the output has the number of fractional bits specified in the **Number of fractional bits in output** parameter

The Symmetric Real FIR block supports discrete sample times and only little-endian code generation.

Dialog Box

Filter a real input signal X using a symmetric real FIR filter. The number of input samples per channel must be a multiple of four. The filter coefficients are specified by a real vector H, which must be symmetric about its middle element. The number of elements in H must be of the form 16k+1 where k is a positive integer. Intermediate accumulations have 32 bits (b31:b0) and use wrap-around arithmetic. All input and output signals are signed 16-bit fixed-point data types.

Parameters

Coefficient source: Specify via dialog

Coefficients:

0.05 * [1, 2, 3, 4, 5, 6, 7, 8, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Set fractional bits in coefficients to: Best precision

Number of fractional bits in coefficients:

10

Set fractional bits in output to: Match high 16 bits of product (b30:b)

Number of fractional bits in output:

10

Initial conditions:

0

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

C64x Symmetric Real FIR

Coefficients

Enter the coefficients in vector format. Coefficients must be symmetric about the middle element of the vector, so the number of coefficients must be odd. This parameter is visible when `Specify via dialog` is specified for the **Coefficient source** parameter. This parameter is tunable in simulation.

Set fractional bits in coefficients to

Specify the number of fractional bits in the filter coefficients:

- `Match input X`—Sets the coefficients to have the same number of fractional bits as the input
- `Best precision`—Sets the number of fractional bits of the coefficients such that the coefficients are represented to the best precision possible
- `User-defined`—Sets the number of fractional bits in the coefficients with the **Number of fractional bits in coefficients** parameter

This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter.

Number of fractional bits in coefficients

Specify the number of bits to the right of the binary point in the filter coefficients. This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter, and is only enabled if `User-defined` is specified for the **Set fractional bits in coefficients to** parameter.

Set fractional bits in output to

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

- `Match input X`—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input X
- `Match coefficients H`—Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in coefficients H
- `Match high bits of acc. (b31:b16)`—Output the highest 16 bits of the accumulator value

- `Match high bits of prod. (b30:b15)`—Output the second-highest 16 bits of the accumulator value
- `User-defined`—Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter

See Matrix Multiply "Examples" on page 5-104 for demonstrations of these selections.

Number of fractional bits in output

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

Initial conditions

If the initial conditions are

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

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

See Also   C64x Complex FIR, C64x General Real FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR

C64x Vector Dot Product

Purpose   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
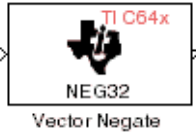   generation.

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.

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
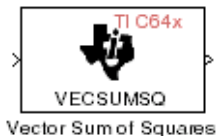   generation.

Dialog Box
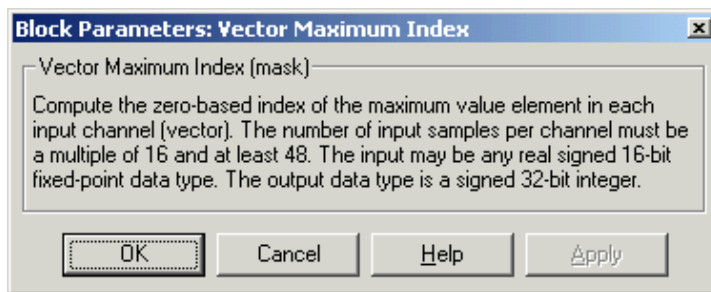
Algorithm   In simulation, the Vector Maximum Index block is equivalent to the
               TMS320C64x DSP Library assembly code function DSP_maxidx. During code
               generation, this block calls the DSP_maxidx routine to produce optimized code.

C64x Vector Maximum Value

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

Library   C64x DSP Library—Math and Matrices
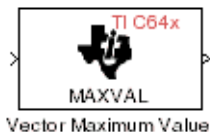
Description   The C64x Vector Maximum Value block returns the maximum value in each
channel (vector) of the input signal. The input can be any real, 16-bit, signed
fixed-point data type. The number of samples on each input channel must be
an integer multiple of 8 and must be at least 32. The output data 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

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
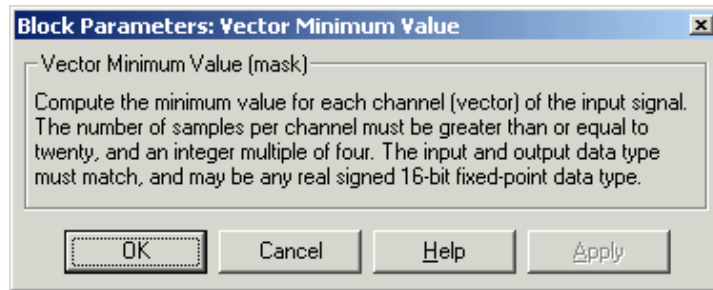
Algorithm   In simulation, the Vector Minimum Value block is equivalent to the TMS320C64x DSP Library assembly code function DSP_minval. During code generation, this block calls the DSP_minval routine to produce optimized code.

See Also   C64x Vector Maximum Value

C64x Vector Multiply

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 or 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
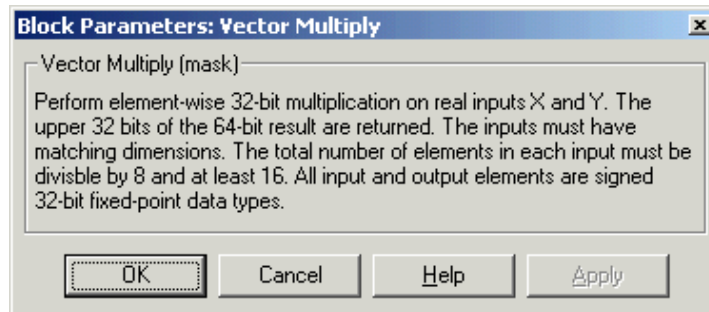
Algorithm   In simulation, the Vector Multiply block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_mul32`. During code generation, this block calls the `DSP_mul32` routine to produce optimized code.
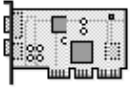
See Also   C64x Matrix Multiply

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

Block Parameters: Vector Negate

─Vector Negate (mask)─

Negate each element of a signed 32-bit fixed-point input signal. For real signals, the number of input elements must a multiple of four and at least eight. For complex signals, the number of input elements must be even and at least four.

OK   Cancel   Help   Apply

Algorithm   In simulation, the Vector Negate block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_neg32`. During code generation, this block calls the `DSP_neg32` routine to produce optimized code.

C64x Vector Sum of Squares

Purpose   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.

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.

Purpose   Set target preferences and memory map to generate code for the C6701 Evaluation Module

Library   Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

Description

C6701EVM

Options on the block mask let you set features of code generation for your C6701 Evaluation Module target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6701 evaluation module must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

Note This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

Among the processor and target options you specify here are the target board information, memory mapping and layout, and how to allocate the various code sections, such as compiler, DSP/BIOS, and custom sections.

Setting the options included in this dialog results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both are essential steps in the process of targeting any board, custom or explicitly supported like the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog for this block unless you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

C6701EVM

Dialog Box

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

- **Board info**—select the target board and processor, set the clock speed, and identify the target.
- **Memory**—set the memory allocation and layout on the target processor (memory mapping).

• **Sections**—determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS information and where to put compiler information.

Board Info Pane

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

Board Type

Lets you enter the type of board you are targeting with the model. You can enter Custom to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as C6711DSK. This block specifies the C6701EVM by default.

Device

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

CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results may be wrong, and code profiling results will not be correct.

You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

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

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. 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 cannot change the clock rate. Other C6000 processors allow different clock speeds.

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

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

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

To create sine block interrupts at 0.001 s/sample requires

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

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

Simulator

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

Enable High-Speed RTDX

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

CCS Board Name

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

CCS Processor Name

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

Memory Pane

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

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

- **Physical Memory**—specifies the processor and board memory map
- **Heap**—specifies whether you use a heap and determines the size in words

• **L2 Cache**—enables the L2 cache (where available) and sets the size in kB

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

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

Physical Memory Options

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

The default memory segments for C6701 EVM boards are:

- IPRAM
- IDRAM
- SBSRAM
- SDDRAM0
- SDRAM1

Name

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

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

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

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

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

Address

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

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

Length

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

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

Contents

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

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

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

Add

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

Remove

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

Create Heap

Selecting this option enables creating the heap, and enables the **Heap size** option.

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

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

Heap Size

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

Define Label

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

Heap Label

Enabled by selecting **Define label**, you use this option to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Enable L2 Cache

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

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

L2 Cache size

Once you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

Sections Pane

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

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

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

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

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

String	Section List	Description of the Section Contents
.switch	Compiler	Jump tables for switch statements in the executable code
.sysdata	DSP/BIOS	Data about DSP/BIOS
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code
.sysmem	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

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

- .cinit
- .const
- .switch
- .text—created by the assembler.

These sections are uninitialized:

- .bss—created by the assembler.
- .far
- .stack

- `.sysmem`

Other sections appear on the list as well:

- `.data`—created by the assembler. The C/C++ compiler does not use this section.
- `.cio`
- `.pinit`

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

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

Shows you where the selected **Compiler sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments available on the C6701 EVM:

- IPRAM
- IDRAM
- SBSRAM
- SDDRAM0
- SDRAM1

DSP/BIOS Sections

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

Description

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

Placement

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

DSP/BIOS Object Placement

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

Custom Sections

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

Name

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

Placement

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

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

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

See Also   Custom C6000

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

```
Line In
C6701 EVM
ADC

ADC
```
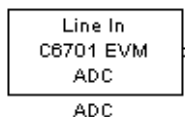
Use the C6701 EVM ADC (analog-to-digital converter) block to capture and digitize analog signals from external sources, such as signal generators, frequency generators or audio devices. Placing an C6701 EVM ADC block in your Simulink block diagram lets you use the multimedia audio coder-decoder module (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.

Audio Word Byte Order for Mono and Stereo Inputs

Format	Left and Mono Channel (first 16 bits of data word)	Right and Stereo Channel (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 define the output ranges based on your selections for the **Data type, Scaling,** and **Codec data format** parameters in the **Block Parameters** dialog.

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
	μ-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.

- μ-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
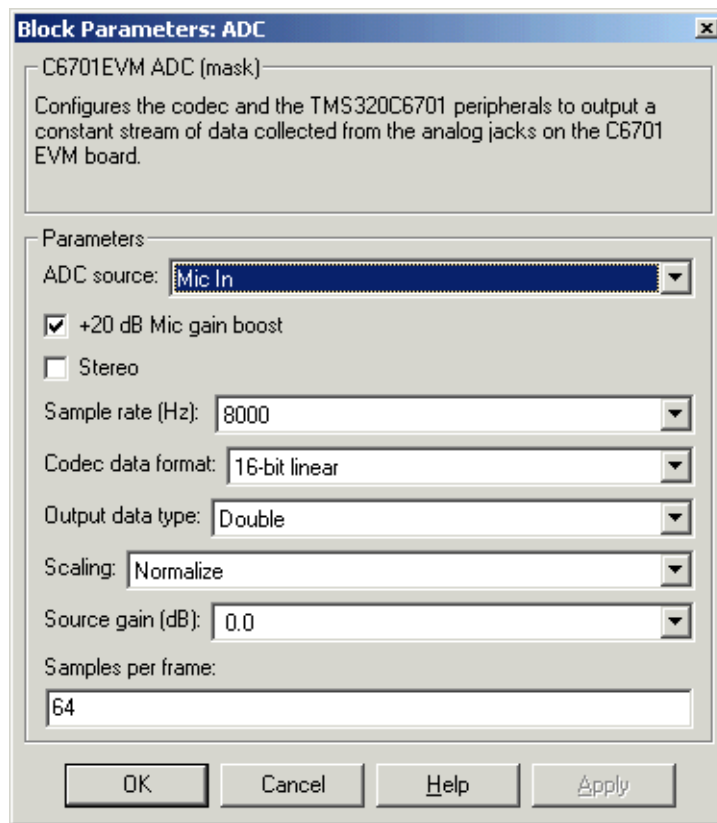
ADC source

> 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 ±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
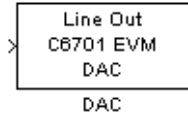
Description   Adding the C6701 EVM DAC (digital-to-analog converter) block to your
Simulink model provides the means to output an analog signal to the LINE
OUT connection on the C6701 EVM mounting bracket. When you add the
C6701 EVM DAC block, the digital signal received by the codec is converted to
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.

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
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
	μ-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` 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.

- µ-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
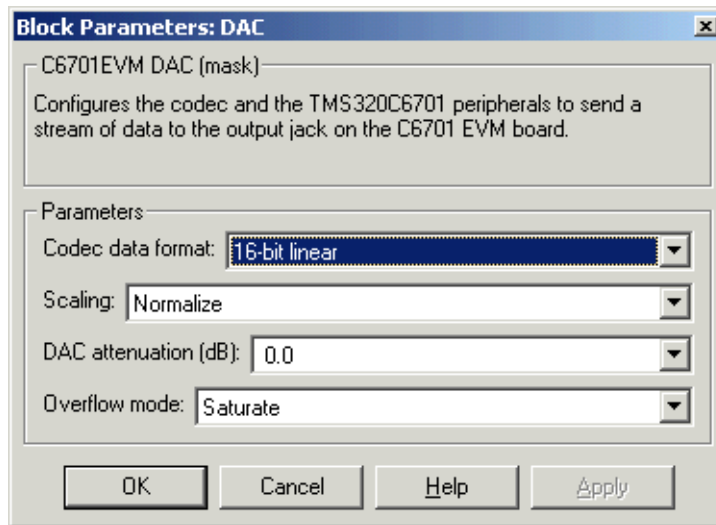
Codec data format

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 ±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

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.

```
C6701 EVM
DIP Switch

  Switch
```
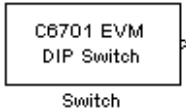
In simulation—the options **USER0**, **USER1**, and **USER2** generate output to
simulate the settings of the user-defined dual inline pin (DIP) switches on your
C6701 EVM. Each option turns the associated DIP switch on when you select
it. The switches are independent of one another.

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.

Option Settings to Simulate the User DIP Switches on the C6701 EVM

USER0 (LSB)	USER1	USER2 (MSB)	Boolean Output	Integer Output
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 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 . 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.

Output Values From the User DIP Switches on the C6701 EVM

USER0 (LSB)	USER1	USER2 (MSB)	Boolean Output	Integer Output
Off	Off	Off	000	0
On	Off	Off	001	1
Off	On	Off	010	2
On	On	Off	011	3
Off	Off	On	100	4
On	Off	On	101	5
Off	On	On	110	6
On	On	On	111	7

Dialog Box
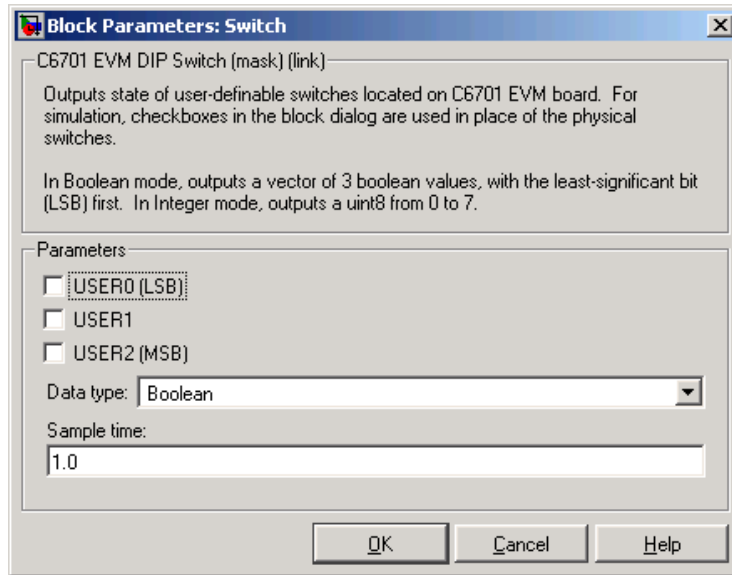
Opening this dialog causes a running simulation to pause. Refer to "Changing Source Block Parameters" in your online Simulink documentation for details.

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 logical string to an equivalent unsigned 8-bit (uint8) decimal value. For example, if the logical string is 101, the decimal conversion yields 5.

C6701 EVM DIP Switch

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
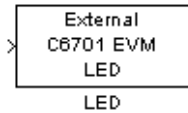
Description   Adding an C6701 EVM LED block to your Simulink block diagram lets you

```
    External
   C6701 EVM
      LED
      LED
```

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
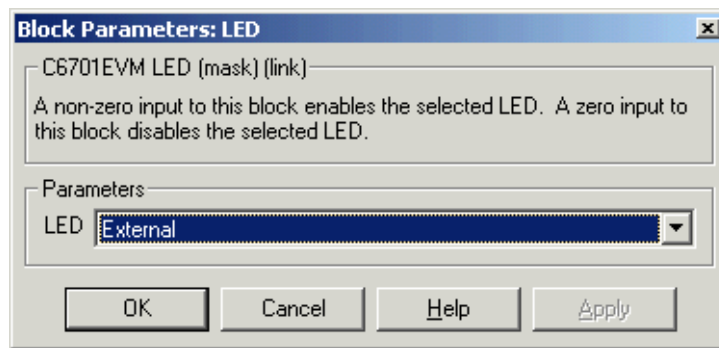
LED

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.

C6711DSK

Purpose   Set target preferences and memory map to generate code for the C6711 DSP Starter Kit

Library   Target Preferences in Embedded Target for TI C6000 DSP for TI DSP

Description
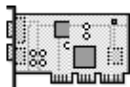
C6711DSK

Options on the block mask let you set features of code generation for your C6711 DSP Starter Kit target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the C6711 DSK must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

Note This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

Among the processor and target options you specify here are the target board information, memory mapping and layout, and how to allocate the various code sections, such as compiler, DSP/BIOS, and custom sections.

Setting the options included in this dialog results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both are essential steps in the process of targeting any board, custom or explicitly supported like the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog for this block unless you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

Dialog Box

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

- **Board info**—select the target board and processor, set the clock speed, and identify the target.
- **Memory**—set the memory allocation and layout on the target processor (memory mapping).

- **Sections**—determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS information and where to put compiler information.

Board Info Pane

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

Board Type

Lets you enter the type of board you are targeting with the model. The C6711DSK block comes with C6711DSK defined as the default board type.

Device

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

CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results may be wrong, and code profiling results will not be correct.

You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

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

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. 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 cannot change the clock rate. Other C6000 processors allow different clock speeds.

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

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

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

To create sine block interrupts at 0.001 s/sample requires

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

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

Simulator

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

Enable High-Speed RTDX

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

CCS Board Name

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

CCS Processor Name

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

Memory Pane

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

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

- **Physical Memory**—specifies the processor and board memory map
- **Heap**—specifies whether you use a heap and determines the size in words

• **L2 Cache**—enables the L2 cache (where available) and sets the size in kB

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

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

Physical Memory Options

This list shows the physical memory segments avaliable on the board and processor. By default, target preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured target preferences blocks shows the memory segments available on the board, but off of the processor. C6711DSK boards provide SDRAM memory segments by default

Name

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

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

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

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

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

Address

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

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

Length

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

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

Contents

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

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

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

Add

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

Remove

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

Create Heap

Selecting this option enables creating the heap, and enables the **Heap size** option.

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

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

Heap Size

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

Define Label

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

Heap Label

Enabled by selecting **Define label**, you use this option to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Enable L2 Cache

C6711 processors support an L2 cache memory structure that you can configure as SRAM and partial cache. Both the data memory and the program share this second-level memory.

L2 Cache size

Once you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

Sections Pane

Options on this pane let you specify where various program sections should go in memory. Program sections are distinct from memory segments—sections are portions of the executable code stored in contiguous memory locations.

Among the sections used generally are .text, .bss, .data, and .stack. Some sections relate to the compiler, some to DSP/BIOS, and some can be custom sections as you require.

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

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

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

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

String	Section List	Description of the Section Contents
.switch	Compiler	Jump tables for switch statements in the executable code
.sysdata	DSP/BIOS	Data about DSP/BIOS
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code
.sysmem	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

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

- .cinit
- .const
- .switch
- .text—created by the assembler.

These sections are uninitialized:

- .bss—created by the assembler.
- .far
- .stack

- `.sysmem`

Other sections appear on the list as well:

- `.data`—created by the assembler. The C/C++ compiler does not use this section.
- `.cio`
- `.pinit`

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

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

Shows you where the selected **Compiler sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. C6711 DSK boards provide IRAM and SDRAM.

DSP/BIOS Sections

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

Description

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

Placement

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

DSP/BIOS Object Placement

Distinct from the entries on the **DSP/BIOS sections** list, DSP/BIOS objects like STS or LOG, if your project uses them, get placed in the memory segment you

select from the **DSP/BIOS Object Placement** list. All DSP/BIOS objects use the same memory segment. You cannot select the location for individual objects.

Custom Sections

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

Name

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

Placement

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

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

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

See Also   Custom C6000

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.

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

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

Source gain (dB) lets you add gain to the input signal before the A/D conversion. When you select `Loopback` as the **ADC source**, your specified source gain is not added to the input signal. Select the appropriate gain from the list.

Dialog Box
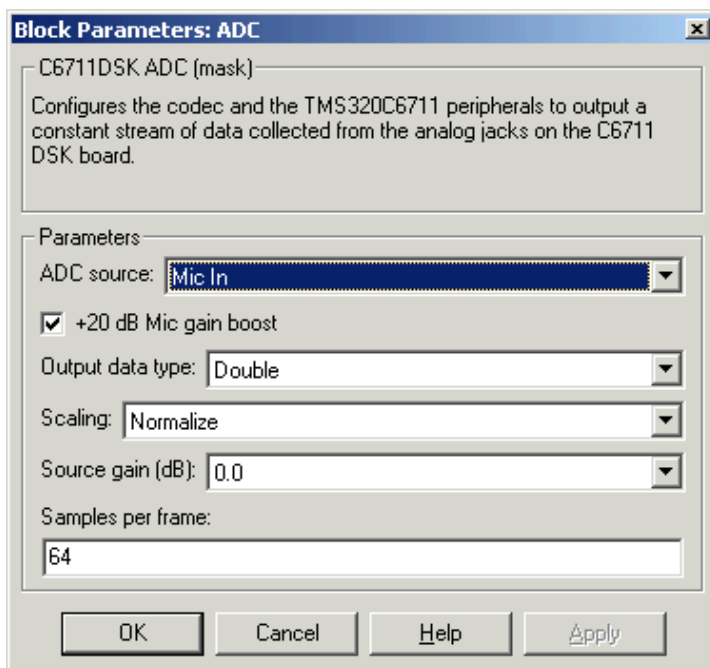
ADC source
    The input source to the codec. `Line In` is the default.

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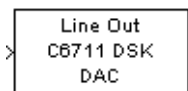
Description   Adding the C6711 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the LINE OUT connection on the C6711 DSK mounting bracket. When you add the C6711 DSK DAC block, the digital signal received by the codec is converted to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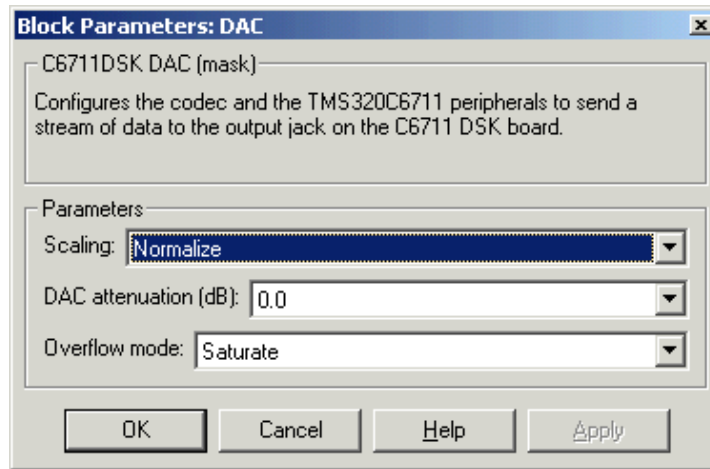
Scaling

Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range ±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.

C6711 DSK
DIP Switch

Switch
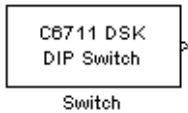
Simulation—the options **USER_SW1**, **USER_SW2**, and **USER_SW3** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6711 DSK. Each option turns the associated DIP switch on when you select it. The switches are independent of one another.

By defining the switches to represent actions on your target, DIP switches let you modify the operation of your process by reconfiguring the switch settings.

Use the **Data type** to specify whether the DIP switch options output an integer or a logical string of bits to represent the status of the switches. The table that follows presents all the option setting combinations with the result of your **Data type** selection.

Option Settings to Simulate the User DIP Switches on the C6711 DSK

USER_SW1 (LSB)	USER_SW2	USER_SW3 (MSB)	Boolean Output	Integer Output
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 . Your process uses the result in the same way whether in simulation or in code generation. In code generation and when running your application, the block code ignores the settings for **USER_SW1**, **USER_SW2**, and **USER_SW3** in favor of reading the hardware switch settings. When the block reads the DIP switches, it reports the results as either a Boolean string or an integer value as Output Values From The User DIP Switches on the C6711 DSK shows

Output Values From The User DIP Switches on the C6711 DSK

USER_SW1 (LSB)	USER_SW2	USER_SW3 (MSB)	Boolean Output	Integer Output
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
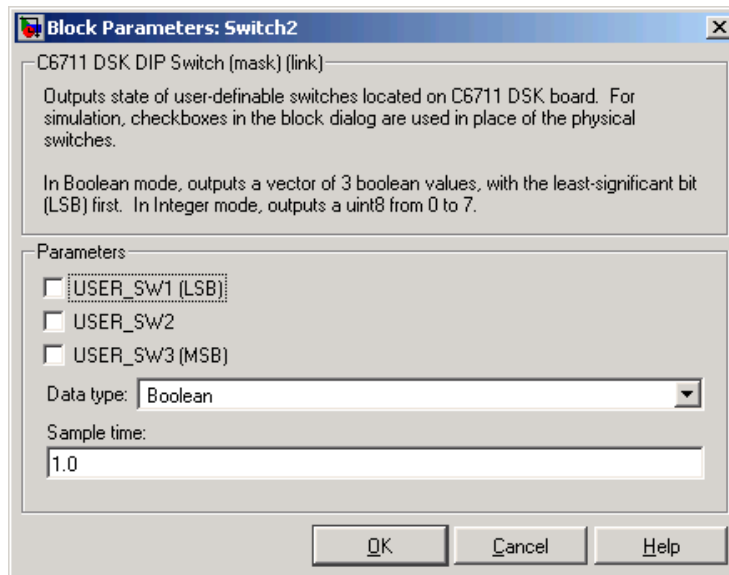
Opening this dialog causes a running simulation to pause. Refer to "Changing Source Block Parameters" in your online Simulink documentation for details.

USER_SW1

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

USER_SW2

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

USER_SW3

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

Data type

Determines how the block reports the status of the user-defined DIP switches. **Boolean** is the default, indicating that the output is a logical string of three bits.

Each bit represents the status of one DIP switch; the LSB is switch **USER_SW1** and the MSB is switch **USER_SW3**. The other data type, **Integer**, converts the logical string to an equivalent unsigned 8-bit (uint8)

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
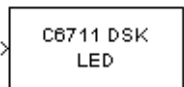
Description   Adding the C6711 DSK LED block to your Simulink block diagram lets you
trigger all three of the user red light emitting diodes (LED) on the C6711 DSK.
To use the block, send a nonzero real scalar to the block. The C6711 DSK LED
block triggers all three user LEDs located on the C6711 DSK.

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
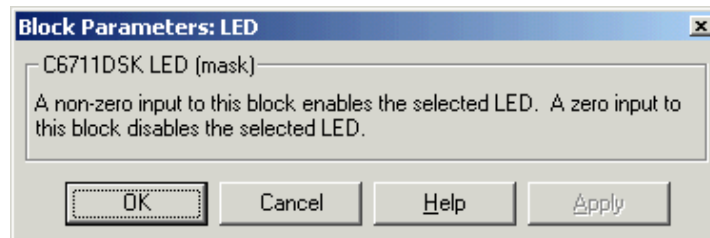
This dialog does not have any user-selectable options.

Purpose   Reset the C6711 DSK to initial conditions

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

Description
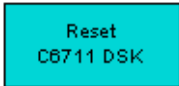
Reset
C6711 DSK

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.

C6713DSK

Purpose   Set target preferences and memory map to generate code for the C6713 DSP Starter Kit

Library   Target Preferences in Embedded Target for TI C6000 DSP for TI DSP
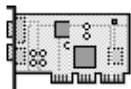
Description   Options on the block mask let you set features of code generation for your C6713 DSP Starter Kit target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

C6713DSK

Any model that you target to the C6713 DSK must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

Note This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

Among the processor and target options you specify here are the target board information, memory mapping and layout, and how to allocate the various code sections, such as compiler, DSP/BIOS, and custom sections.

Setting the options included in this dialog results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both are essential steps in the process of targeting any board, custom or explicitly supported like the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog for this block unless you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

Dialog Box

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

- **Board info**—select the target board and processor, set the clock speed, and identify the target.
- **Memory**—set the memory allocation and layout on the target processor (memory mapping).

C6713DSK

- **Sections**—determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS information and where to put compiler information.

Board Info Pane

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

Board Type

Lets you enter the type of board you are targeting with the model. You can enter Custom to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as C6711DSK. **Board type** for this block is set to C6713 DSK by default.

Device

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

CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results may be wrong, and code profiling results will not be correct.

You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

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

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. 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 cannot change the clock rate. Other C6000 processors allow different clock speeds.

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

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

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

To create sine block interrupts at 0.001 s/sample requires

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

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

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

Enable High-Speed RTDX
Select this option to tell the code generation process to enable high-speed RTDX for code generated from this model.

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

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

C6713DSK

Memory Pane

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

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

- **Physical Memory**—specifies the processor and board memory map
- **Heap**—specifies whether you use a heap and determines the size in words

• **L2 Cache**—enables the L2 cache (where available) and sets the size in kB

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

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

Physical Memory Options

This list shows the physical memory segments avaliable on the board and processor. By default, target preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured target preferences blocks shows the memory segments available on the board, but off of the processor. C6713 DSK boards provide IRAM and SDRAM memory segments by default

Name

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

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

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

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

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

Address

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

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

Length

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

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

Contents

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

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

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

Add

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

Remove

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

Create Heap

Selecting this option enables creating the heap, and enables the **Heap size** option.

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

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

Heap Size

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

Define Label

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

Heap Label

Enabled by selecting **Define label**, you use this option to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Enable L2 Cache

C6713 processors support an L2 cache memory structure that you can configure as SRAM and partial cache.

L2 Cache size

Once you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

Sections Pane

Options on this pane let you specify where various program sections should go in memory. Program sections are distinct from memory segments—sections are portions of the executable code stored in contiguous memory locations. Among the sections used generally are .text, .bss, .data, and .stack. Some

sections relate to the compiler, some to DSP/BIOS, and some can be custom sections as you require.

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

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

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

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

String	Section List	Description of the Section Contents
.switch	Compiler	Jump tables for switch statements in the executable code
.sysdata	DSP/BIOS	Data about DSP/BIOS
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code
.sysmem	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

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

- .cinit
- .const
- .switch
- .text—created by the assembler.

These sections are uninitialized:

- .bss—created by the assembler.
- .far
- .stack

- `.sysmem`

Other sections appear on the list as well:

- `.data`—created by the assembler. The C/C++ compiler does not use this section.
- `.cio`
- `.pinit`

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

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

Shows you where the selected **Compiler sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. For C6713 DSK targets, the list include IRAM and SDRAM segments.

DSP/BIOS Sections

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

Description

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

Placement

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

DSP/BIOS Object Placement

Distinct from the entries on the **DSP/BIOS sections** list, DSP/BIOS objects like STS or LOG, if your project uses them, get placed in the memory segment you

C6713DSK

select from the **DSP/BIOS Object Placement** list. All DSP/BIOS objects use the same memory segment. You cannot select the location for individual objects.

Custom Sections

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

Name

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

Placement

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

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

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

See Also   Custom C6000

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.

```
Line In
C6713 DSK
ADC
ADC
```

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.

C6713 DSK ADC

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.

Dialog Box
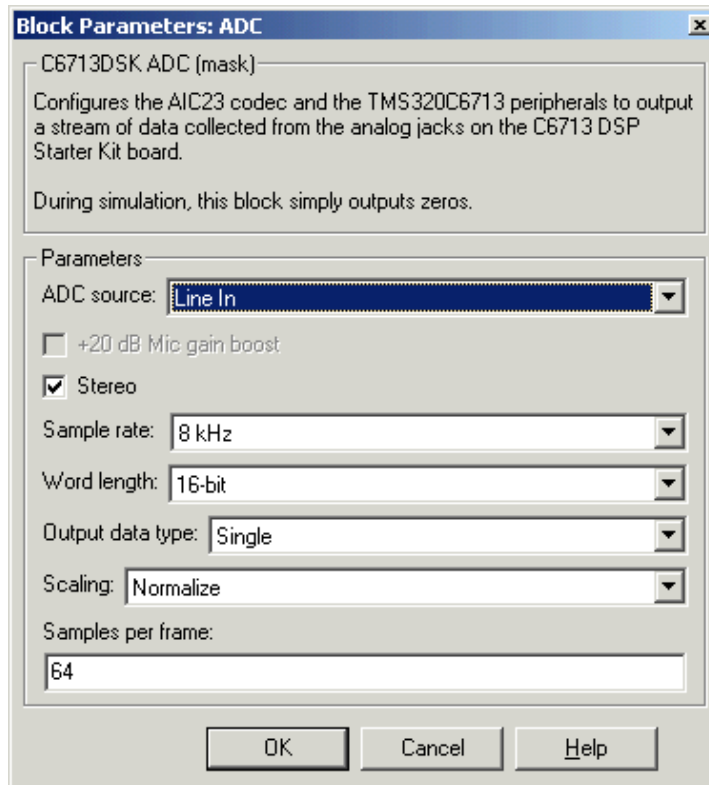
ADC source

    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.

C6713 DSK ADC

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 ±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

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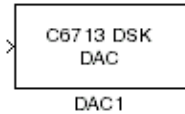
Description   Adding the C6713 DSK DAC (digital-to-analog converter) block to your Simulink model provides the means to output an analog signal to the analog output jack on the C6713 DSK. When you add the C6713 DSK DAC block, the digital signal received by the codec is converted to an analog signal.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

```
C6713 DSK
  DAC

   DAC1
```

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.

Option	Affected Hardware
Overflow mode	TMS320C6713 Digital Signal Processor
Scaling	TMS320C6713 Digital Signal Processor
Word length	Codec

C6713 DSK DAC

Dialog Box
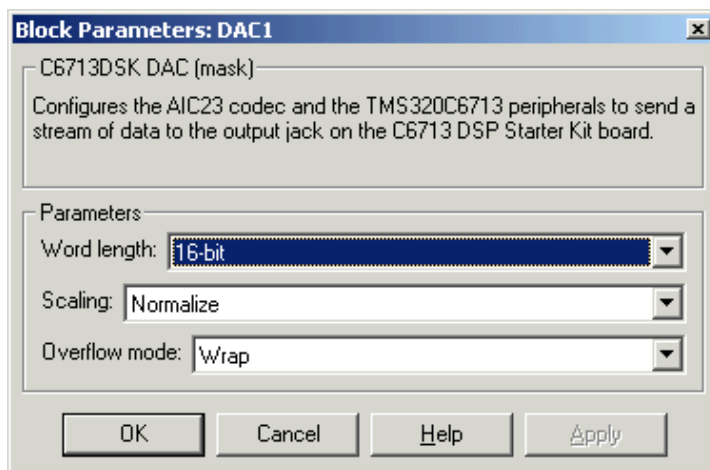
Word length

    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 ±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

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.

```
C6713 DSK
DIP Switch

   Switch
```
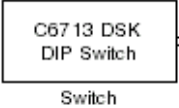
In Simulation—the options **Switch 0**, **Switch 1**, **Switch 2,** and **Switch 3** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6713 DSK. Each option turns the associated DIP switch on when you select it. The switches are independent of one another.

By defining the switches to represent actions on your target, DIP switches let you modify the operation of your process by reconfiguring the switch settings.

Use the **Data type** to specify whether the DIP switch options output an integer or a logical string of bits to represent the status of the switches. The table that follows presents all the option setting combinations with the result of your **Data type** selection.

Option Settings to Simulate the User DIP Switches on the C6713 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Cleared	Cleared	Cleared	0000	0
Selected	Cleared	Cleared	Cleared	0001	1
Cleared	Selected	Cleared	Cleared	0010	2
Selected	Selected	Cleared	Cleared	0011	3
Cleared	Cleared	Selected	Cleared	0100	4
Selected	Cleared	Selected	Cleared	0101	5
Cleared	Selected	Selected	Cleared	0110	6
Selected	Selected	Selected	Cleared	0111	7
Cleared	Cleared	Cleared	Selected	1000	8
Selected	Cleared	Cleared	Selected	1001	9

C6713 DSK DIP Switch

Option Settings to Simulate the User DIP Switches on the C6713 DSK (Continued)

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Selected	Cleared	Selected	1010	10
Selected	Selected	Cleared	Selected	1011	11
Cleared	Cleared	Selected	Selected	1100	12
Selected	Cleared	Selected	Selected	1101	13
Cleared	Selected	Selected	Selected	1110	14
Selected	Selected	Selected	Selected	1111	15

Selecting the Integer data type results in the switch settings generating integers in the range from 0 to 15 (uint8), corresponding to converting the string of individual switch settings to a decimal value. In the Boolean data type, the output string presents the separate switch setting for each switch, with the **Switch 0** status represented by the least significant bit (LSB) and the status of **Switch 3** represented by the most significant bit (MSB).

In Code generation and targeting—the code generated by the block reads the physical switch settings of the user switches on the board and reports them as shown above. Your process uses the result in the same way whether in simulation or in code generation. In code generation and when running your application, the block code ignores the settings for **Switch 0**, **Switch 1**, **Switch 2** and **Switch 3** in favor of reading the hardware switch settings. When the block reads the DIP switches, it reports the results as either a Boolean string or an integer value as the table below shows.

Output Values From The User DIP Switches on the C6713 DSK

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

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
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

C6713 DSK DIP Switch

Dialog Box
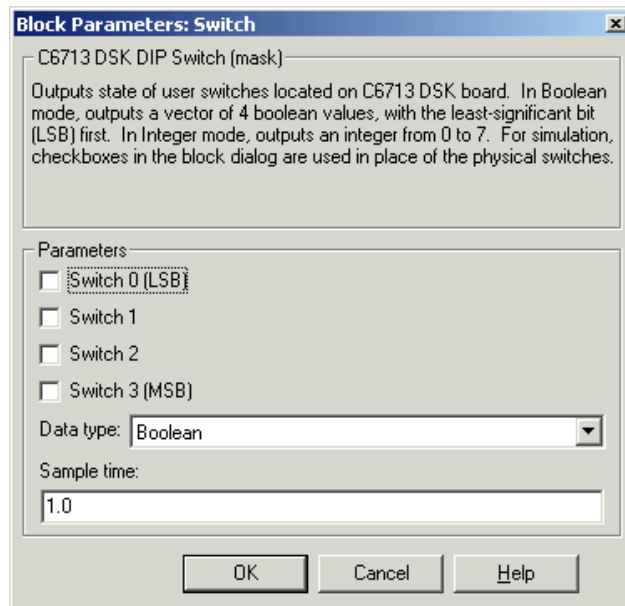
Opening this dialog causes a running simulation to pause. Refer to "Changing Source Block Parameters" in your online Simulink documentation for details.

Switch 0

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

Switch 1

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

Switch 2

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

Switch 3

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

Data type

Determines how the block reports the status of the user-defined DIP switches. Boolean is the default, indicating that the output is a vector of four logical values, either 0 or 1.

Each vector element represents the status of one DIP switch; the first switch is switch **Switch 0** and the fourth is switch **Switch 3**. The data type `Integer` converts the logical string to an equivalent unsigned 8-bit (`uint8`) value. For example, when the logical string generated by the switches is 0101, the conversion yields 5—the LSB is 1 and the MSB is 0.

Sample time

Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

C6713 DSK LED

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

Library   C6713 DSK Board Support in Embedded Target for TI C6000 DSP
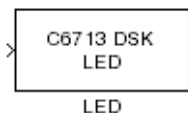
Description Adding the C6713 DSK LED block to your Simulink block diagram lets you trigger all four of the user light emitting diodes (LED) on the C6713 DSK. To use the block, send a nonzero real scalar to the block. The C6713 DSK LED block controls all four user LEDs located on the C6713 DSK.

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
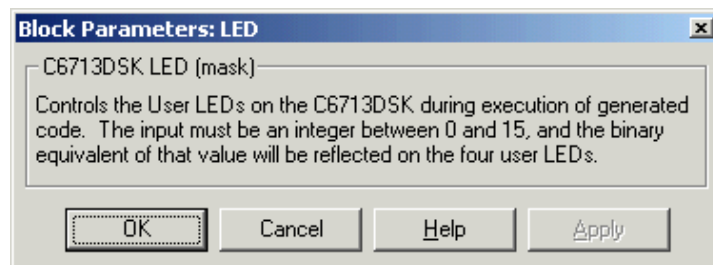
This dialog does not have any user-selectable options.

Purpose   Reset the C6713 DSK to initial conditions

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

Description   Double-clicking this block in a Simulink model window resets the C6713 DSK

Reset
C6713 DSK

Reset

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.

Custom C6000

Purpose   Set target preferences to generate code for C6000-processor-based custom hardware targets

Library   Target Preferences in Embedded Target for TI C6000 DSP for TI DSP
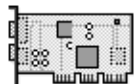
Description   Options on the block mask let you set features of code generation for your custom C6000 processor-based target. Adding this block to your Simulink model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to custom hardware must include this block or the target preferences block that best matches your processor, such as the C6416DSK target preferences block to target custom hardware based on the C6416 processor. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

Note   This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model. Simulink returns an error when your model does not include a target preferences block or has more than one.

Among the processor and target options you specify here are the target board information, memory mapping and layout, and how to allocate the various code sections, such as compiler, DSP/BIOS, and custom sections.

Setting the options included in this dialog results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both are essential steps in the process of targeting any board, custom or explicitly supported, like the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog for this block unless you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

Dialog Box

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

- **Board info**—select the target board and processor, set the clock speed, and identify the target.
- **Memory**—set the memory allocation and layout on the target processor (memory mapping).

- **Sections**—determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS information and where to put compiler information.

Board Info Pane

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

Board Type

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

Device

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

CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results may be wrong, and code profiling results will not be correct.

You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

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

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. 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 cannot change the clock rate. Other C6000 processors allow different clock speeds.

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

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

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

To create sine block interrupts at 0.001 s/sample requires

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

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

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

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

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

CCS Processor Name
Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one

processor. In the multiprocessor case, you select the processor by name from the list.

Memory Pane

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

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

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

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

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

Physical Memory Options

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

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

- Custom boards based on C670x processors provide IPRAM and IDRAM memory segments by default.
- C6701 EVM boards provide IPRAM, IDRAM, SBSRAM, SDDRAM0, and SDRAM1 memory segments by default
- C6711DSK boards provide SDRAM memory segments by default.

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

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

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

Custom C6000

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

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

Address

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

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

Length

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

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

Contents

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

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

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

Add

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

Remove

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

Create Heap

If your processor supports using a heap, as do the C6711 or C6701, for example, selecting this option enables creating the heap, and enables the **Heap size** option. **Create heap** is not available on processors that either do not provide a heap or do not allow you to configure the heap.

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

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

Heap Size

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

Define Label

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

Custom C6000

Heap Label

Enabled by selecting **Define label**, you use this option to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Enable L2 Cache

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

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

L2 Cache size

Once you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

Sections Pane

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

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

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

Custom C6000

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

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

String	Section List	Description of the Section Contents
`.switch`	`Compiler`	Jump tables for switch statements in the executable code
`.sysdata`	`DSP/BIOS`	Data about DSP/BIOS
`.sysinit`	`DSP/BIOS`	DSP/BIOS initialization startup code
`.sysmem`	`Compiler`	Dynamically allocated object in the code containing the heap
`.text`	`Compiler`	Load allocation for the literal strings, executable code, and compiler generated constants
`.trcdata`	`DSP/BIOS`	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

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

- `.cinit`
- `.const`
- `.switch`
- `.text`—created by the assembler.

These sections are uninitialized:

- `.bss`—created by the assembler.
- `.far`
- `.stack`

- `.sysmem`

Other sections appear on the list as well:

- `.data`—created by the assembler. The C/C++ compiler does not use this section.
- `.cio`
- `.pinit`

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

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

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

DSP/BIOS Sections

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

Description

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

Placement

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

DSP/BIOS Object Placement

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

Custom Sections

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

Name

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

Placement

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

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

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

DM642EVM

Purpose   Set target preferences to generate code for the DM642 Evaluation Module

Library   Target Preferences in Embedded Target for TI C6000 DSP for TI DSP
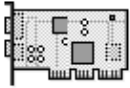
Description   Options on the block mask let you set features of code generation for your DM642 Evaluation Module target. Adding this block to your Simulink model provides access to the processor hardware settings to configure when you generate code from Real-Time Workshop to run on the target.

Any model that you target to the DM642 evaluation module must include this block, or the Custom C6000 target preferences block. Real-Time Workshop returns an error message if a target preferences block is not present in your model.

Note This block must be in your model at the top level and not in a subsystem. It does not connect to any other blocks, but stands alone to set the target preferences for the model.

Among the processor and target options you specify here are the target board information, memory mapping and layout, and how to allocate the various code sections, such as compiler, DSP/BIOS, and custom sections.

Setting the options included in this dialog results in identifying your target to Real-Time Workshop, Embedded Target for TI C6000 DSP, and Simulink, and configuring the memory map for your target. Both are essential steps in the process of targeting any board, custom or explicitly supported like the C6711 DSK or the DM642 EVM.

Unlike most other blocks, you cannot open the block dialog for this block unless you add the block to a model. When you try to open the block dialog, the block attempts to connect to your target. It cannot make the connection when the block is in the library and returns an error message.

Dialog Box

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

- **Board info**—select the target board and processor, set the clock speed, and identify the target.
- **Memory**—set the memory allocation and layout on the target processor (memory mapping).

- **Sections**—determine the arrangement and location of the sections on the target processor such as where to put the DSP/BIOS information and where to put compiler information.

Board Info Pane

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

Board Type

Lets you enter the type of board you are targeting with the model. You can enter Custom to support any board based on one of the supported processors, or enter the name of one of the supported boards, such as C6711DSK. By default, the DM642EVM block specifies the DM642EVM for the board type.

Device

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

CPU Clock Speed (MHz)

Shows the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results may be wrong, and code profiling results will not be correct.

You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock speed** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

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

Correctly generating interrupts for your model depends on the clock rate of the CPU on your target. 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 cannot change the clock rate. Other C6000 processors allow different clock speeds.

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

The timer uses the CPU clock rate you specify in **CPU clock speed** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 KHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. Using the clock rate you choose, 100 MHz for example, the timer calculates the sine generator interrupt period as follows for the sine block:

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

To create sine block interrupts at 0.001 s/sample requires

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

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

Simulator

Select this option when you are targeting a simulator rather than a hardware target. You must select **Simulator** to target your code to a C6000 simulator.

Enable High-Speed RTDX

Select this option to tell the code generation process to enable high-speed RTDX for this model.

CCS Board Name

Contains a list of all the boards defined in CCS Setup. From the list of available boards, select the one that you are targeting your code for.

CCS Processor Name

Lists the processors on the board you selected for targeting in **CCS board name**. In most cases, only one name appears because the board has one processor. In the multiprocessor case, you select the processor by name from the list.

Memory Pane

When you target any board, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map.

The **Memory** pane contains memory options in three areas:

- **Physical Memory**—specifies the processor and board memory map
- **Heap**—specifies whether you use a heap and determines the size in words

• **L2 Cache**—enables the L2 cache (where available) and sets the size in kB

Be aware that these options may affect the options on the **Sections** pane. You can make selections here that change how you configure options on the **Sections** pane.

Most of the information about memory segments and memory allocation is available from the online help system for Code Composer Studio.

Physical Memory Options

This list shows the physical memory segments avaliable on the board and processor. By default, target preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured target preferences blocks shows the memory segments available on the board, but off of the processor. Target preferences blocks set default starting addresses, lengths, and contents of the default memory segments. DM642EVM boards provide ISRAM and SDRAM memory segments by default.

Name

When you highlight an entry on the **Physical memory** list, the name of the entry appears here. To change the name of the existing memory sgment, select it in the Physical memory list and then type the new name here.

Note You cannot change the names of default processor memory segments.

To add a new physical memory segment to the list, click **Add**, replace the temporary label in **Name** with the one to use, and press **Return**. Your new segment appears on the list.

After you add the segment, you can configure the starting address, length, and contents for the new segment. New segments start with code and data as the type of content that can be stored in the segment (refer to the **Contents** option).

Names are case sensitive. `NewSegment` is not the same as `newsegment` or `newSegment`.

Address

Address reports the starting address for the memory segment showing in **Name**. Address entries are in hexadecimal format and limited only by the board or processor memory.

When you are using a processor-specific preferences block, the starting address shown is the default value. You can change the starting value by entering the new value directly in **Address** when you select the memory segment to change.

Length

From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs). For the C6000 processor family, the MADU is 8 bytes, one word.

When you are using a processor-specific preferences block, the length shown is the default value. You can change the value by entering the new value directly in this option.

Contents

Contents details the kind of program sections that you can store in the memory segment in **Name**. As the processor type for the target preferences block changes, the kinds of information you store in listed memory segments may change. Generally, the **Contents** list contains these strings:

- Code—allow code to be stored in the memory segment in **Name**.
- Data—allow data to be stored in the memory segment in **Name**.
- Code and Data—allow code and data to be stored in the memory segment in **Name**. When you add a new memory segment, this is the default setting for the contents of the new element.

You may add or use as many segments of each type as you need, within the limits of the memory on your processor.

Add

Click **Add** to add a new memory segment to the target memory map. When you click **Add**, a new segment name appears, for example NEWMEM1, in **Name** and on the **Physical memory** list. In **Name**, change the temporary name NEWMEM1 by entering the new segment name. Entering the new name, or clicking **Apply** updates the temporary name on the list to the name you enter.

Remove

This option lets you remove a memory segment from the memory map. Select the segment to remove on the **Physical memory** list and click **Remove** to delete the segment.

Create Heap

Selecting this option enables creating the heap, and enables the **Heap size** option.

Using this option you can create a heap in any memory segment on the **Physical memory** list. Select the memory segment on the list and then select **Create heap** to create a heap in the select segment. After you create the heap, use the **Heap size** and **Define label** options to configure the heap.

The location of the heap in the memory segment is not under your control. The only way to control the location of the heap in a segment is to make the segment and the heap the same size. Otherwise, the compiler determines the location of the heap in the segment.

Heap Size

After you select **Create heap**, this option lets you specify the size of the heap in words. Enter the number of words in decimal format. When you enter the heap size in decimal words, the system converts the decimal value to hexadecimal format. You can enter the value directly in hexadecimal format as well. Processors may support different maximum heap sizes.

Define Label

Selecting **Create heap** enables this option that allows you to name the heap. Enter your label for the heap in the **Heap label** option.

Heap Label

Enabled by selecting **Define label**, you use this option to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Enable L2 Cache

DM642 processors support an L2 cache memory structure that you can configure as ISRAM and partial cache.

L2 Cache size

Once you enable the L2 cache, use this list to determine the size of the cache allotted. Select the size of the cache from the list.

Sections Pane

Options on this pane let you specify where various program sections should go in memory. Program sections are distinct from memory segments—sections

are portions of the executable code stored in contiguous memory locations. Among the sections used generally are `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler, some to DSP/BIOS, and some can be custom sections as you require.

For more information about program sections and objects, refer to the CCS online help. Most of the definitions and descriptions in this section come from CCS.

Within this pane, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

Here are brief definitions of the various kinds of sections in the lists. All sections do not appear on both lists. The list on which the string appears is shown in the table.

String	Section List	Description of the Section Contents
.args	DSP/BIOS	Argument buffers
.bss	Compiler	Static and global C variables in the code
.bios	DSP/BIOS	DSP/BIOS code if you are using DSP/BIOS options in your program
.cinit	Compiler	Tables for initializing global and static variables and constants
.cio	Compiler	Standard I/O buffer for C programs
.const	Compiler	Data defined with the C qualifier and string constants
.data	Compiler	Program data for execution
.far	Compiler	Variables, both static and global, defined as far variables
.gblinit	DSP/BIOS	Load allocation of the DSP/BIOS startup initialization tables section
.hwi	DSP/BIOS	Dispatch code for interrupt service routines
.hwi_vec	DSP/BIOS	Interrupt Service Table
.obj	DSP/BIOS	Configuration properties that the target program can read
.pinit	Compiler	Load allocation of the table of global object constructors section.
.rtdx_text	DSP/BIOS	Code sections for the RTDX program modules
.stack	Compiler	The global stack

String	Section List	Description of the Section Contents
`.switch`	`Compiler`	Jump tables for switch statements in the executable code
`.sysdata`	`DSP/BIOS`	Data about DSP/BIOS
`.sysinit`	`DSP/BIOS`	DSP/BIOS initialization startup code
`.sysmem`	`Compiler`	Dynamically allocated object in the code containing the heap
`.text`	`Compiler`	Load allocation for the literal strings, executable code, and compiler generated constants
`.trcdata`	`DSP/BIOS`	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help. Most of the definitions and descriptions in this section come from the online help for CCS.

Compiler Sections

During program compilation, the C6000 compiler produces both uninitialized and initialized blocks of data and code. These blocks get allocated into memory as required by the configuration of your system. On the **Compiler sections** list you find both initialized (sections that contain data or executable code) and uninitialized (sections that reserve space in memory) sections. The initialized sections are:

- `.cinit`
- `.const`
- `.switch`
- `.text`—created by the assembler.

These sections are uninitialized:

- `.bss`—created by the assembler.
- `.far`
- `.stack`

- `.sysmem`

Other sections appear on the list as well:

- `.data`—created by the assembler. The C/C++ compiler does not use this section.
- `.cio`
- `.pinit`

When you highlight a section on the list, **Description** shows a brief description of the section. Also, **Placement** shows you where the section is presently allocated in memory.

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

Shows you where the selected **Compiler sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains ISRAM and SDRAM when you use this block.

DSP/BIOS Sections

During program compilation, DSP/BIOS produces both uninitialized and initialized blocks of data and code. These blocks get allocated into memory as required by the configuration of your system. On the **DSP/BIOS sections** list you find both initialized (sections that contain data or executable code) and uninitialized (sections that reserve space in memory) sections.

Description

Provides a brief explanation of the contents of the selected **DSP/BIOS sections** list entry

Placement

Shows where the selected **DSP/BIOS sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments available on C6000 processors, and changes based on the processor you are using.

DSP/BIOS Object Placement

Distinct from the entries on the **DSP/BIOS sections** list, DSP/BIOS objects like STS or LOG, if your project uses them, get placed in the memory segment you

select from the **DSP/BIOS Object Placement** list. All DSP/BIOS objects use the same memory segment. You cannot select the location for individual objects.

Custom Sections

When your program uses code or data sections that are not included in either the **Compiler sections** or **DSP/BIOS sections** lists, you add the new sections to this list. Initially, the **Custom sections** list contains no fixed entries, just a placeholder for a section for you to define.

Name

You enter the name for your new section here. To add a new section, click **Add**. Then replace the temporary name with the name to use. Although the temporary name includes a period at the beginning you do not need to include the period in your new name. Names are case sensitive. `NewSection` is not the same as `newsection`, or `newSection`.

Placement

With your new section added to the **Name** list, select the memory segment to which to add your new section. Within the restrictions imposed by the hardware and compiler, you can select any segment that appears on the list.

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog.

Remove

To remove a section from the **Custom sections** list, select the section to remove and click **Remove**. The selected section disappears from the list.

Purpose

Configure video capture capability (video decoders) to capture analog video input on the DM642 EVM

Library

DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

Description

Adding this block to a model enables code generated from your model to perform the following tasks:
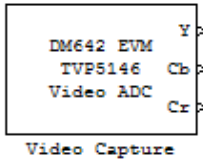
1 Capture analog video data from the video input ports on the DM642 EVM.

2 Convert the input to a format and mode you define in the block.

3 Output the converted digital video for further downstream processing.

Adding two of these blocks to a model lets you capture two separate video data streams and prepare them for display simultaneously, such as in picture-in-picture mode.

The block captures and buffers one frame (two fields for NTSC standard) of analog input video from the input ports, converts the buffered video to the specified format, and then outputs the converted video frame as 8-bit unsigned integer data for further processing.

Input to the DM642 EVM must be analog National Television Standards Committee (NTSC) video format. The block captures and processes data in frames, not fields.

To configure the format for the output video, the block offers output format options that control how the block handles color data. The block also offers a sample time option to let you set the frame rate for video output from the block.

Note This block does not provide output video for display. Use the DM642 EVM Video DAC to generate video data to output to the board video output connectors.

When you add this block to a Simulink model, it has no affect in your simulation—it outputs a string of zeros. Generating code from a model that includes this block produces the code needed for capturing data on your evaluation module by adding

5-243

- Video device configuration code for the chosen mode
- Code used to copy the run-time buffer

To use video in a Simulink model, use one of the available video source blocks to introduce video data to your model.

Options for the block let you configure the digital video format and video mode for the data output by the block.

NTSC TV systems use interlaced scanning to create TV frames from fields. The even and odd TV lines are separated into even and odd fields that combine to make a complete TV frame image. For output, the block always provides complete frames, consisting of two fields, which are available at any instant. When the sample time you specify for the block is different from the NTSC frame rate of 30Hz, you may encounter visible anomalies in the video stream from the block.

Notes About Converting NTSC Video Input From YCbCr to RGB24

When you choose to convert your NTSC YCbCr-defined video input to RGB24 (8:8:8 RGB) for output from the block, the block performs an intermediate conversion step that follows a standard process for conversion (as described by Graphical Device Interface (GDI) color space conversions documentation from the International Color Consortium (ICC)).

First, the block converts your YCbCr input signal to 5:6:5 RGB format where the red and blue channels of the source use a 5-bit representation and the green channel uses 6 bits.

Now the block converts your 5:6:5 RGB to 8:8:8 RGB using the following conventions:

1 For the red and blue 5-bit channels, it copies the three most significant bits (MSB) from the 5-bit source word and append them to the lower order end of the target word.

2 For the green 6-bit channel, it copies the two MSBs from the green source word and append them to the lower order end of the target green word.

The results is to output three RGB channels—red, green, and blue—each with 8-bit words.

For example, to convert hexadecimal values by this algorithm, 5:5:5 RGB data of (0x19, 0x33, 0x1A) becomes (0xCE, 0xCF, 0xD6) of 8:8:8 RGB output.

To do the conversion in the binary case for 5:5:5 RGB data:

1   blue data 1 1101 converts to 11101111

2   for the green channel, conversion takes 11 0011 to 1100 1111

3   red data 1 0101 becomes 1010 1101 (same algorithm as blue data)

To maximize the speed of the RGB conversion, the Video ADC block provides color space conversion using a routine hand-written in assembly language and optimized for the DM64x processor core. Using the optimized color space conversion code replaces the Color Space Conversion block available from the Video and Image Processing (VIP) Blockset. While you can use any compatible VIP blockset block with the DM642, this particular color space conversion operation is handled better by the conversion code included in the ADC block.

Dialog Box
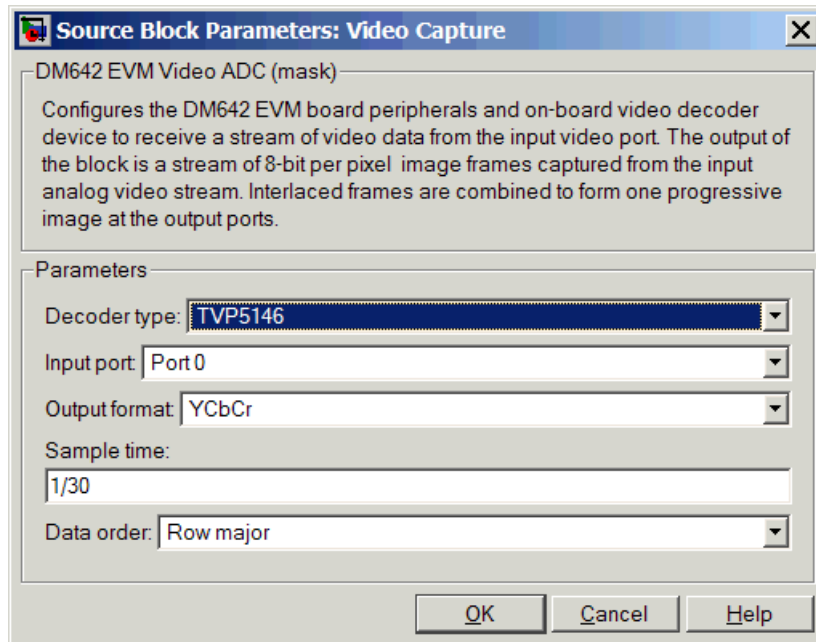
Decoder type

Configures the block options to support either the TVP5146 Decoder on the DM642 EVM or the SAA7115 Decoder, depending on the model of your board. Choose one option from the list—TVP5146 or SAA7115. When you select SAA7115 for the type of decoder, the dialog adds a new option—**Output Mode**. Generally, older DM642 EVM boards use the SAA7115 decoder. Newer boards use the default setting TVP5146 decoder. Refer to "Identifying Your DM642 EVM Board Revision" on page A-4 for information about identifying the revision of your DM642 EVM.

Input port

Directs the block to capture video from either the 0 or 1 video input port on the DM642 EVM. The block does not support port 2 for video input.

Output mode (available only when Decoder type is SAA7115)

Because all input video to the board is in analog NTSC 720-by-480 mode, this option scales the output from the block to different dimensions. Output modes for the block include various modes, as shown in this table. The important information in the table is the size of the images. Though the input to the block is always analog NTSC video, the output from the block uses the scaling capability of the video decoder to scale the digitized output image to one of the available sizes listed in the table

Digital Output Mode	Description
NTSC 720x480	Scales the output to higher definition TV mode.
NTSC 640x480	Scales the output to standard (SDTV) mode.
NTSC SIF 320x240	Scales the output to standard interchange format NTSC. Derived from CCIR 601 video (most often).
NTSC QCIF 176x144	Scales the output to 1/4 the resolution of CIF video.
NTSC SQCIF 128x96	Scales the output to 1/4 the resolution of QCIF video.

This option appears in the dialog when you select SAA7115 for the **Decoder type**.

Web cameras, PAL format video, and S-Video inputs are not supported.

Output format

Determines how the block represents color data in the output. Choose one of the following color representations according to what your model and algorithm require.

Digital Output Format	Description
RGB24	Output uses 8 bits each of red, green, and blue colors to represent the color of each pixel in the image. RGB color space is device-dependent.
YCbCr	Output from the block includes one luminance channel Y (essentially the black/white signal) and two chrominance (color) channels Cb and Cr to represent the color image data per pixel. This is the digital standard color space DVDs use.
Y	Black/White video. No color/chromaticity values.

Sample time

Tells the block how often to take frames from the video decoder and buffers. While NTSC video runs at 30 frames/s (1/30 s sample time), you can sample at any rate below or above the TV rate. Remember that sampling times that are not 1/30th of a second may either capture incomplete frames, when **Sample time** < frame time, or miss frames/fields when **Sample time** > frame time.

Also note that

- The sample time you specify becomes the DM642 timer period that drives the execution of your model.

- Your generated application is not synchronized with the input video signal—the application always runs on the processor timer.

Data order

With data order, you control the way the the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

- `Row major`—store video data in row major order. This is the default setting and matches most video data.
- `Column major`—store video data in column major order. Simulink® and MATLAB both use this format to store images and matrices.

DM642 EVM Video ADC blocks store the image data in row major format because most video capture devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

MATLAB and Simulink use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video ADC blocks.

To address this problem, the Video ADC blocks include an option **Data order** to let you select either row major or the column major storage formats. By default, these blocks use row major data format.

When the column major data ordering option is selected, the block performs an explicit transposition on the image data to map the data format from row major to column major order. To minimize the processor time spent on the transposition, the block uses optimized assembly routines to accomplish the image transposition.

See Also   DM642 EVM Video DAC

Purpose

Configure the video display capability (video encoder) on the DM642 EVM

Library

DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

Description

```
  >Y
        DM642 EVM
  >Cb   SAA7105
        Video DAC
  >Cr

    Video Display
```
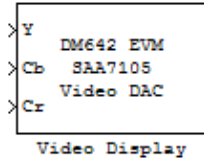
In the project generated from a model, this block provides the code to gather video from another block in the model, and direct the video stream to the video output port on the board.

You should input unsigned 8-bit integers to the block in the specified mode.

Adding this block to a model enables code generated from your model to perform the following tasks:

1 Capture digital video data from the application on your DM642 EVM.

2 Buffer the captured video into frames for NTSC display—two fields per frame and 30 frames per second, or SVGA display—RGB24 color with noninterlaced frames.

3 Convert to analog video.

4 Output the converted analog video to the EVM Video Out ports.

Unlike the DM642 EVM Video ADC block, this DAC block does not convert the video between formats. Nor does this block inherit any settings from the DM642 EVM Video ADC block, as some of the other C6000 DAC blocks do.

The **Mode** option specifies both the video format the block accepts and the format the block outputs to the video output ports on the EVM.

To be able to be displayed, images that you send to the block should be equal to or smaller than the target display size. If the input images are smaller than the target display size, the block pads the image by adding zeros to the image.

When you add this block to your Simulink model, it has no affect on your simulation—it outputs a string of zeros. In code generation, the block creates the device code needed to buffer, convert, and send video to the output port on the EVM.

Dialog Box
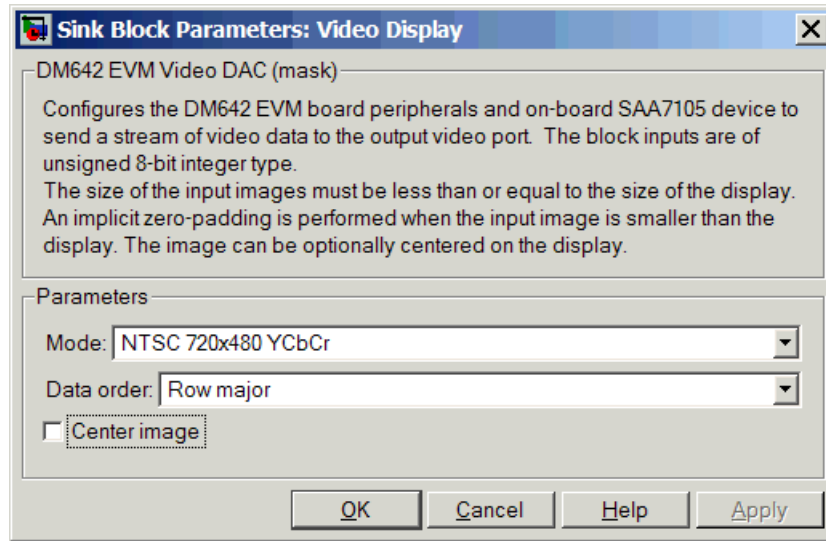
Mode

Specifies the video format for the block. The block then sends video in this format to the video output port on the EVM. The **Mode** parameter offers the following options:

Analog Output Mode	Description
NTSC 720x480 YCbCr	Analog output of video data in 720-by-480 pixels format with full color
NTSC 640x480 Y	Analog video output in 640-by-480 pixels format with black and white only (luminance). No color data.
SVGA 800x600 RGB24	Full super VGA format 800-by-600 pixels with three color channels: 8-bit red, 8-bit green, and 8-bit blue data.

Data order

With data order, you control the way the the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

- Row major—store video data in row major order. This is the default setting and matches most video data.
- Column major—store video data in column major order. Simulink® and MATLAB both use this format to store images and matrices.

DM642 EVM Video DAC blocks store the image data in row major format because most video display devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

MATLAB and Simulink use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video DAC blocks.

To address this problem, the Video DAC blocks include an option **Data order** to let you select either row major or the column major storage formats. By default, these blocks use row major data format.

When the column major data ordering option is selected, the block performs an explicit transposition on the image data to map the data format from row major to column major order. To minimize the processor time spent on the transposition, the block uses optimized assembly routines to accomplish the image transposition.

Center Image

Directs the block to center the output image on the display. Note that centering the image requires some computation by the processor so there are small time and CPU cycles penalties for choosing this option. For that reason, **Center image** is cleared by default.

Another note of interest—some cameras pad their video output with zeros to ensure that the display does not cut off the image on one side, usually the left. Images that include such padding may appear to be off-center on the display. In fact, while the displayed image may not appear centered, the electronic image (the data that compose the displayed image plus the padding which you cannot see) is centered in the display area.

DM642 EVM Video DAC

Purpose   Control the eight light-emitting diodes (LED) on the DM642 Evaluation Module

Library   DM642 EVM Board Support Library in Embedded Target for TI C6000 DSP

Description   Controls the user LEDs on the DM642 EVM while the processor executes your generated code. To trigger the LEDs, input an unsigned 8-bit integer to the block. In response, the eight user-controlled LEDs reflect the binary equivalent of that input value—turning off an LED is 0 and turning on an LED is 1.
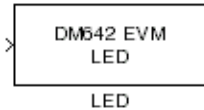
During operation, the LED block inherits the sample time from the upstream block in the model. Therefor, each time the model operation encounters the LED block, the block writes the desired output value to the LEDs.

Dialog Box
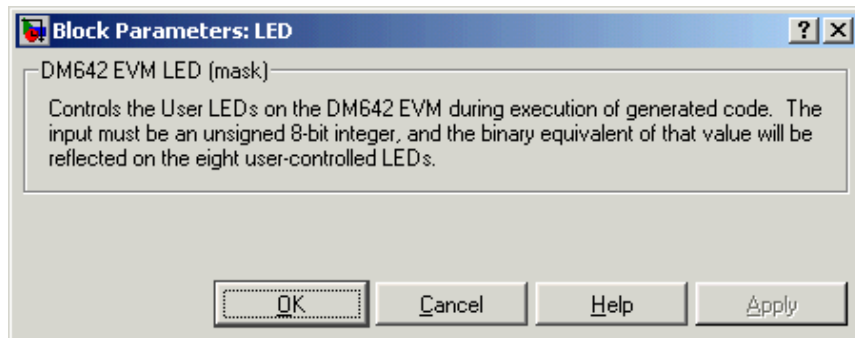
You see the block does not provide user options. Adding the block to your model adds the ability to control the LEDs.

DM642 EVM RESET

Purpose   Reset the DM642 EVM to initial conditions

Library   DM642 EVM Board Support in Embedded Target for TI C6000 DSP

Description   Double-clicking this block in a Simulink model window resets the DM642 EVM that is running the executable code built from the model. When you double-click the Reset block, the block runs the software reset function provided by CCS that resets the processor on your DM642 EVM. Applications running on the board stop and the signal processor returns to the initial conditions you defined.

Before you build and download your model, add the block to the model as a stand-alone block. You do not need to connect the block to any block in the model. When you double-click this block in the block library it resets your DM642 EVM. In other words, anytime you double-click a DM642 EVM RESET block you reset your DM642 EVM.

Dialog Box   This block does not have settable options and does not provide a user interface dialog.

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

```
From
Memory
```
From Memory
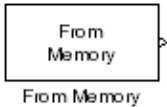
When you generate code from your Simulink model in Real-Time Workshop with this block in place, code generation inserts the C commands to create a read process that gets data from memory on the target. The inserted code reads the specified memory location in **Memory address** and returns the data stored there. Any valid memory location on the target works with the block.

When you look at your generated code, you find lines of code like the following that represent the From Memory block operation:

```
/* S-Function Block: <Root>/From Memory (c6000mem_src) */
  {
    /* Memory Mapped Input */
    rtB.From_Memory = (real_T)(*((volatile int *)(4096U)));
  }
```

In simulations this block does not perform any operations, with the exception that the block does output port checking. From Memory blocks work only in code generation and when your model runs on your target.

Using From Memory Blocks
Be careful when you use From Memory blocks in your models in combination with To Memory blocks. Because the To Memory blocks give you control over where the target stores information in memory, pay attention to how you use the From Memory block to retrieve data from memory. You can return data that is not what you expect.

Using the From Memory block itself does not cause problems in generated code on your target.

When you use the options in the To Memory block to specify where the project writes data in memory, you might be writing to memory locations that are reserved for the compiler or for other uses. Reading from those locations could return the wrong answer.

To prevent your model from encountering memory errors like these, generate your code once without loading the COFF file to the target. Look at the

From Memory

generated file *projectname*.map, where *projectname* is the name of your project, to see the memory range that the compiler uses.

From this list of allocated memory, determine the memory ranges that the compiler uses and the locations of free memory or the memory to read with your From Memory block. Determine the memory locations from which to read your data from the .map file listings.

You should examine the .map file for your project each time you change the Simulink model associated with your project.

Dialog Box
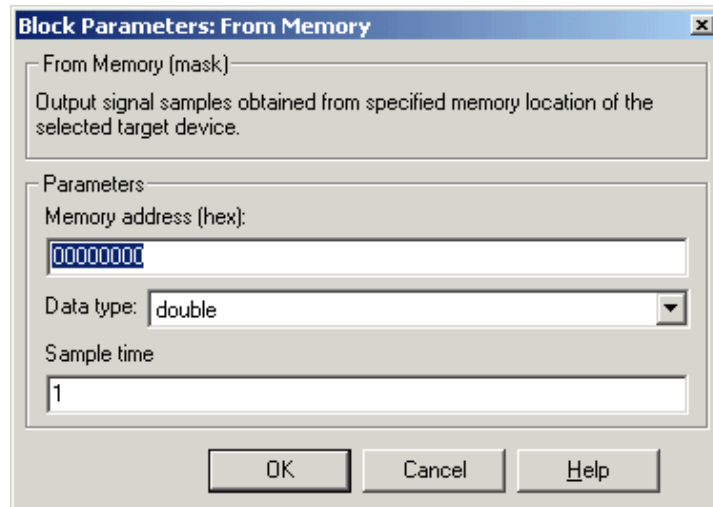
Memory address (hex)

Enter the address of the memory location that contains the data to return. Note that you do not need to start the address with 0x to indicate that it is hexadecimal.

Data type

Sets the type for the data coming from the block. Select one of the following types:

• double—double-precision floating-point values. This is the default setting.

- `single`—single-precision floating-point values.
- `uint8`—8-bit unsigned integers. Output values range from 0 to 255.
- `int16`—16-bit signed integers. With the sign, the values range from -32768 to 32767.
- `int32`—32-bit signed integers. Values range from -2^{31} to $(2^{31}-1)$.

Sample time

Specifies the time between samples of the signal. The default is 1 second between samples, for a sample rate of one sample per second (1/**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

From RTDX
ichan1

When you generate code from your Simulink model in Real-Time Workshop with this block in place, code generation inserts the C commands to create an RTDX input channel on the target. The inserted code opens and enables the channel with the name you specify in **Channel name** in the block parameters. You can open, close, disable, and enable the channel from the host side afterwards, overriding the target side status.

In the generated code, you see a command like the following

```
RTDX_enableInput(&channelname)
```

where channelname is the name you enter in **Channel name.**

In simulations this block does not perform any operations with the exception that the block will generate an output matching your specified initial conditions. From Rtdx blocks work only in code generation and when your model runs on your target.

If you are using the Link for Code Composer Studio Development Tools, you need to configure and cleanup RTDX properly before and after executing your model or code. Refer to the RTDX tutorials in the Link for Code Composer Studio documentation in the online Help system to see an example of how to do this housekeeping task.

The initial conditions you set in the block parameters determine the output from the block to the target for the first read attempt. Specify the initial conditions in one of the following ways:

• Scalar value—the block generates one output sample with the value of the scalar. For a value of 0, the block outputs a zero to the processor. When **Output dimension** specifies an array, every element in the array has the same scalar value.

• Null array ([ ])—same output as a scalar with the value zero for every sample.

Using RTDX in your model involves:

- Adding one or more To Rtdx or From Rtdx blocks to your model to prepare your target
- Downloading and running your model on your target
- Enabling the RTDX channels from MATLAB or using **Enable RTDX channel on start-up** on the block dialog
- Using the `readmsg` and `writemsg` functions in MATLAB to send and retrieve data from the target over RTDX

To see more details about using RTDX in your model, refer to Using Links in your Link for Code Composer Studio Development Tools documentation in the online Help system.

From Rtdx

Dialog Box

Block Parameters: From RTDX

From RTDX (mask)

Use specified RTDX channel to send data from host to target DSP. In blocking mode, the DSP waits for new data from the block. In non-blocking mode, the DSP uses previous data when new data is not available from the block.

Parameters

Channel name

ichan1

☐ Enable blocking mode

Initial conditions:

0

Sample Time

1

Output dimensions

[1 64]

☑ Frame-based

Data type: double

☐ Enable RTDX channel on start-up

OK   Cancel   Help   Apply

Channel name

Defines the name of the input channel to be created by the generated code. Recall that input channels refer to transferring data from the host to the target (input to the target). To use this RTDX channel, you enable and open the channel with the name, and send data from the host to the target across this channel. Specify any name as long as it meets C syntax requirements for length and character content.

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:

From Rtdx

- Double—double-precision floating-point values. This is the default setting. Values range from -1 to 1.

- Single—single-precision floating-point values ranging from -1 to 1.

- Uint8—8-bit unsigned integers. Output values range from 0 to 255.

- Int16—16-bit signed integers. With the sign, the values range from -32768 to 32767.

- Int32—32-bit signed integers. Values range from -2^{31} to $(2^{31}-1)$.

Enable RTDX channel on start-up

When your application code includes RTDX channel definitions, selecting this option enables the channels when you start the channel from MATLAB. With this selected, you do not need to use the Link for Code Composer Studio Development Tools `enable` function to prepare your RTDX channels. Note that the option applies only to the channel you specify in **Channel name**. You do have to open the channel.

See Also　　ccsdsp, readmsg, To Rtdx, writemsg

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
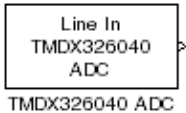
Description With the TMDX326040A daughter card installed on your C6711 DSK, you use this block to configure the codec on the card. The daughter card codec replaces the codec on the C6711 DSK, taking the analog input from the analog ports on the DSK and converting them to digital data. This block configures the analog-to-digital conversion performed by the daughter card.

```
Line In
TMDX326040
ADC

TMDX326040 ADC
```

Note This card is also known as the PCM3003 Audio Daughter Card.

Both the sampling rate and data format for the daughter card codec are fixed:

• Sampling rate is 48 kHz.

• Data format is linear 16-bit words.

Data leaving the codec on the daughter card goes to the COM port on the C6711 DSK and then to the C6711 digital signal processor.

You have the choice of using either stereo or monaural input to the card. The **Stereo** option tells the codec whether the input is in stereo or mono format. When you use the block, the **Stereo** option is selected by default.

Other codec options help you configure the digital data from the daughter card, such as setting the data type (double, single, or integer) and selecting whether the output data should be unmodified or scaled to the range between -1 and 1.

TMDX326040 ADC

Dialog Box
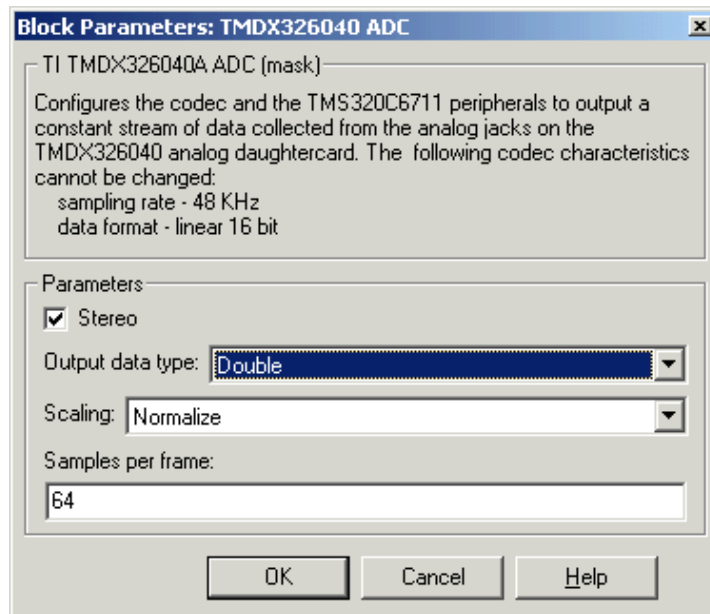
Stereo

Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels.

Output data type

Selects the word length and shape of the data from the codec. By default, double is selected. Options are Double, Single, and Integer

Scaling

Selects whether the codec data is unmodified, or normalized to the output range to ±1.0, based on the codec data format. Select either Normalize or Integer for the scaling. Normalize is the default setting. Scaling applies only to the floating-point data types double and single. When you use integer data, data values get scaled to be between -32768 to 32767.

Samples per frame

Creates frame-based outputs from sample-based inputs. This parameter specifies the number of samples of the signal the block buffers internally before it sends the digitized signals, as a frame vector, to the next block in the model. 64 samples per frame is the default setting. Notice that the frame rate depends on the sample rate and frame size. For example, if your input is 32 samples per second, and you select 64 samples per frame, the frame rate is one frame every two seconds. The throughput remains the same at 32 samples per second.

See Also   TMDX326040 DAC

TMDX326040 DAC

Purpose   Configure 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
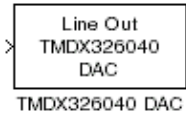
Description With the TMDX326040A daughter card installed on your C6711 DSK, you use this block to configure the codec on the card. The daughter card codec replaces the codec on the C6711 DSK, sending its output to the output connectors on the card. This block configures the digital-to-analog conversion performed by the daughter card.

```
     Line Out
>   TMDX326040
        DAC

  TMDX326040 DAC
```

Note This card is also known as the PCM3003 Audio Daughter Card.

Both the sampling rate and data format for the daughter card codec are fixed:

• Sampling rate is 48 kHz.
• Data format is linear 16-bit words.

Analog data leaving the codec on the daughter card outputs on the card and then to the C6711 DSK output connectors. Whether the output is in monaural or stereo depends on the setting of the TMDX326040 ADC block in your model.

To work properly, you must be sure the input signal is a column vector when you use the monaural mode, or an N-by-2 matrix in stereo mode. This input format must match the ADC block mode—when you select the **Stereo** option for the C6711 ADC block operating mode, you must format your input data as an N-by-2 matrix here.

Other codec options help you configure the digital data from the daughter card—scaling and overflow mode. The scaling option determines whether the input data remains unmodified or is scaled to the range between -1 and 1.

Dialog Box
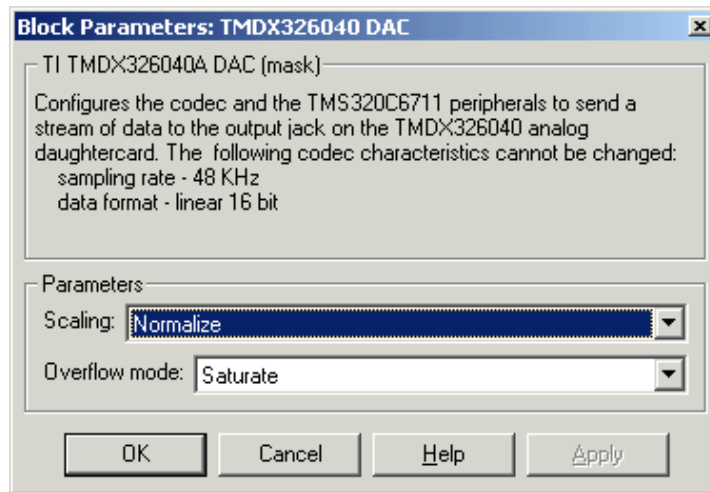
Scaling

    Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range ±1.0. Matching the setting for the TMDX326040 ADC block is appropriate here. Normalized scaling is the default setting.

    When the data type is integer, this scaling option does not apply. Scaling applies only to floating-point data types single and double. You can select different data types and scaling for the ADC and DAC blocks within your model.

Overflow mode

    Determines how the codec responds to data that is outside the range specified by the **Scaling** parameter. You choose either Wrap or Saturate from the list to specify how to handle the result of an overflow in an operation. Saturate mode is slightly less efficient because of the logic executed for each sample to determine whether to saturate the value.

See Also   TMDX326040 ADC

To Memory

Purpose   Send data from your model running in the processor to memory on your C6000 target

Library   C6000 DSP Core Support in Embedded Target for TI C6000 DSP for TI DSP
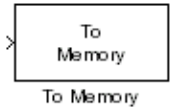
Description   When your Simulink model has this block in place, Real-Time Workshop code generation inserts the C commands to write data to the specified memory location on the target. The inserted code takes the value you send to the block input port and writes it to the location in **Memory address**.

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.

Using To Memory Blocks

You must take care when you use To Memory blocks in your models. Because the To Memory blocks give you control over where the target stores information in memory, and provide full flexibility to copy any section of target data and write to any memory sections, pay attention to how you use the block memory options. This flexibility can lead to unexpected behavior caused by memory operations in your model.

When you use the optional features in the To Memory block to write data to specified locations in memory, you might be writing to memory locations that are reserved for the compiler to use. Writing to those locations can cause CCS to crash.

To prevent your model from encountering memory errors like these, generate your code once without loading the COFF file to the target. Look at the

generated file *projectname*.map, where *projectname* is the name of your project, to see the memory range that the compiler uses.

From this list of allocated memory, you can determine the memory ranges that you can use safely without overwriting the reserved compiler sections.

You should examine the .map file for your project each time you change the Simulink model associated with your project.

Dialog Box
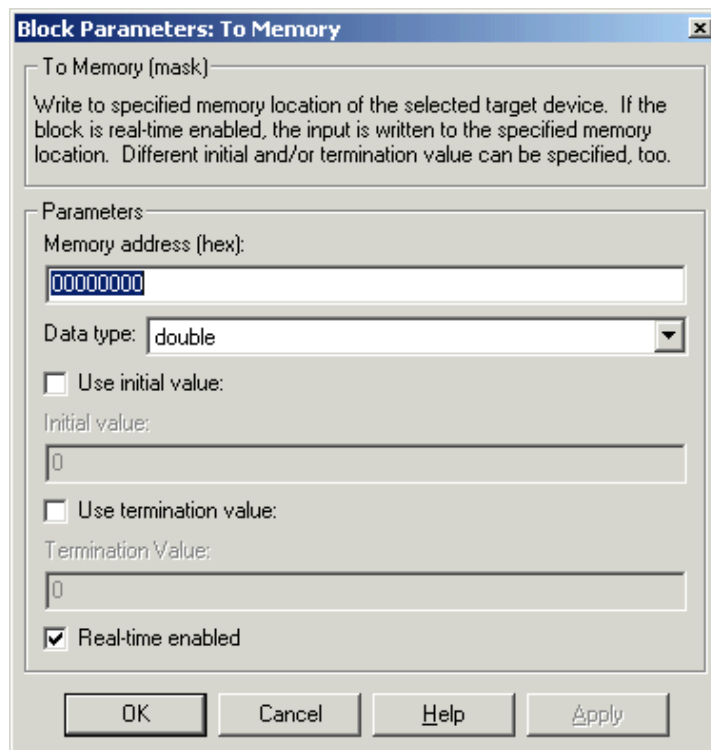
Memory address (hex)

Specifies the address to which you are sending data from the code. Enter the address as a hexadecimal value, without the leading 0x indicator. Any valid memory address works, as long as the processor can write to it.

Data type

To Memory

Sets the type for the data going to memory. Select one of the following types:

- double—double-precision floating-point values. This is the default setting and allows the full range of values representable in double-precision arithmetic as defined by the IEEE specification.
- single—single-precision floating-point values whose range is defined by the IEEE specification on single-precision values.
- uint8—8-bit unsigned integers. Input values range from 0 to 255.
- int16—16-bit signed integers. With the sign, the values range from -32768 to 32767.
- int32—32-bit signed integers. Values range from -2^{31} to $(2^{31}-1)$.

Use initial value

Select this option when you want to send a specific value to memory during the first execution on your model. Enter your desired value in **Initial value**.

Initial value

Enter the value to send to memory on the first execution of this code. Enter a floating-point integer here. The block interprets the value you enter as an integer. For example, to place the integer value 100 in memory, enter 100 here. Note that the block does not support MATLAB integer data types.

Use termination value

Select this option when you want to send a specific value to memory during the last or final execution of your model. Enter your desired value in **Termination value**.

Termination value

Enter the value to send to memory on the last execution of this code. Enter a floating-point integer here. The block interprets the value you enter as an integer. For example, to place the integer value 100 in memory on the final execution pass, enter 100 here.

Real-time enabled

In basic terms, generated code executes as follows:

1 Initialize

2 Start execution (your initial value is written to memory)

3 Output (execute loop for each time step)

4 Terminate execution (your termination value is written to memory)

Selecting **Real-time enabled** causes the code to write data to memory during the output phase of code execution. When you clear this option, the code does nothing during the output phase—it does not write data to the memory address you specify in **Memory address**. You might clear **Real-time enabled** when you want to write a value to memory only during the start phase (or only during the terminate phase, or during both phases) but not during output execution). The block input port disappears when you clear this check box.

See Also   From Memory

To Rtdx

Purpose   Add 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 1-2—Using Links for RTDX in your Link for Code Composer Studio documentation.

One mistake is to connect a To Rtdx block directly to a ADC block, or another source block. Due to current RTDX timing constraints, the generated code from this arrangement does not work as you expect. Look at the following model for an example that does not properly transfer data.
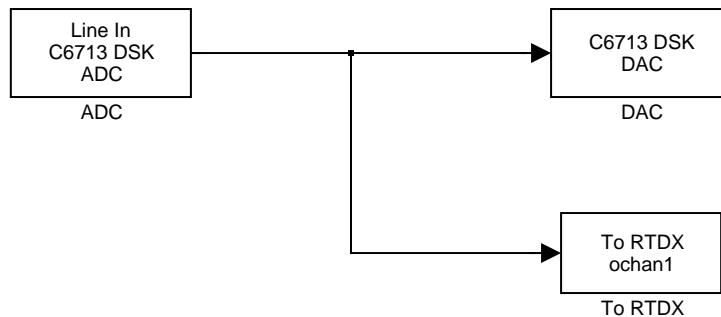
Applications that you generate from models that contain the directly connected blocks are likely to overrun because the sampling time of the codec or source is much faster than RTDX processing time. RTDX will not be able to keep up. This is true even if you do not generate your application from Simulink.

Adding additional blocks can fix the problem. In the next model, adding the Downsample block with 32 for **K**, the **Downsample factor**, allows RTDX to return messages as expected.
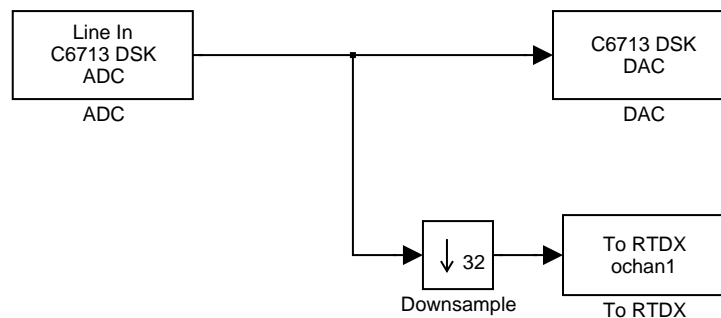
When you are using the Link for CCS to transfer data from the target to the host (MATLAB), this overrun is manifested by cc.rtdx.msgcount eventually decreasing to zero.

You need to modify your application such that:

• Data going to the RTDX block channel is slowed down. For example, use the Downsample block. Downsampling to 32 usually works fine.

To Rtdx

- Give the RTDX more time to process a message transfer by decreasing the size of the data (such as using `short` instead of `int` data types).

- Depending on the application type you are developing, you can use the standard RTDX or the high-speed RTDX. If you are using video processing using standard RTDX, you will not get the desired output. Please refer to the TI documentation for more information.

  To enable high-speed RTDX in Simulink:

  - In your model, go to **Simulation**->**Configuration Parameters**.

  - Select **Real-Time Workshop** on the left pane. Highlight the **RTW system target file** options of the TI 6000 target.

  - Select **TIC6000 target selection** on the left pane and on the right pane, select **Enable High-Speed RTDX**.

If you are using the Link for Code Composer Studio Development Tools, you need to configure and cleanup RTDX properly before and after executing your model or code. Refer to the RTDX tutorials in the Link for Code Composer Studio documentation in the online Help system to see an example of how to do this housekeeping task.

Dialog Box
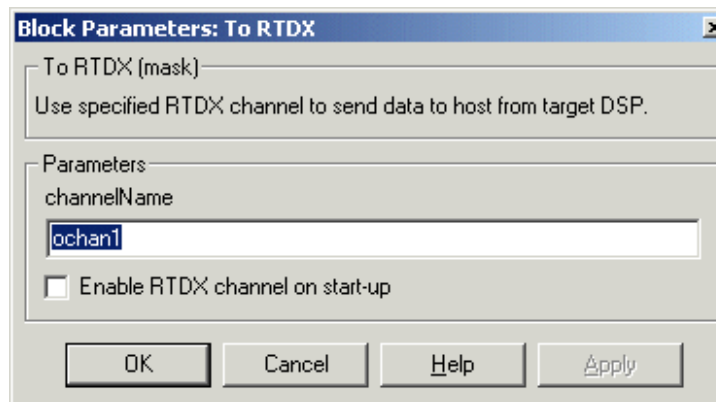
Channel name

Defines the name of the output channel on the target DSP. Recall that output channels refer to transferring data from the target to the host (output from the target). To use this RTDX channel, you enable and open the channel with the name, and send data from the target to the host across

this channel. Specify any name as long as it meets C syntax requirements for length and character content.

Enable RTDX channel on start-up

When your application code includes RTDX channel definitions, selecting this option enables the channels when you start the channel from MATLAB. With this selected, you do not need to use the Link for Code Composer Studio Development Tools `enable` function to prepare your RTDX channels. Note that the option applies only to the channel you specify in **Channel name**. You do have to open the channel.

See Also ccsdsp, From Rtdx, readmsg, writemsg

To Rtdx

Supported Hardware and Issues

Supported Hardware for Targeting (p. A-2)

Lists the hardware that the Embedded Target for TI C6000 DSP supports. Includes comments about supported operating systems where needed.

Requirements for the DM642 EVM (p. A-4)

Points out some details about using the DM642 target.

Continuing Issues with Embedded Target for TI C6000 DSP (p. A-9)

Describes some important features about targeting particular hardware.

Supported Hardware for Targeting

Using the C6000 target in Real-Time Workshop, the Embedded Target for TI C6000 DSP supports the following boards produced by TI and other manufacturers.

Supported Board Designation	Board Description
TMS320C6416 DSK	C6416 DSP Starter Kit. Does not work on Microsoft Windows™ NT platforms.
DM642 EVM	DM642 Evaluation module for developing video processing algorithms and applications
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.
Custom boards based on supported C6000 processors	Code generation for targets that are not explicitly supported, but that use supported DSPs, such as C6701 or C6416.

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.

Requirements for the DM642 EVM

Certain requirements for the DM642 EVM differ from the other supported targets. This section provides details about using both the DM642 EVM hardware target and the simulator. Using the DM642 requires the following:

- DM642 EVM version identification
- Separate CCS installation version 2.2
- DM642 EVM patch to upgrade CCS version to 2.20.18
- XDS560 (high speed RTDX emulator) or XDS510 (Regular RTDX emulator, if your model does not require high-speed RTDX capability)
- Device Driver Development Kit (DDK) patch
- TMS320DM642 Digital Media Development Kit (DMDK)
- Projects must enable DSP/BIOS. Embedded Target for TI C6000 DSP does not support operations on the DM642 EVM without DSP/BIOS.

About DM642 EVM Board Revisions

Working with DM642 EVM boards requires that you identify the board revision that you own.

Identifying Your DM642 EVM Board Revision

Spectrum Digital has released three different versions of the DM642 EVM board. DM642 EVM board Versions 1 and 2 are the same except for the CPU clock speed. Version 2 uses a 720 MHz clock, rather than the 600 MHz clock on Version 1. Both versions use Phillips SAA7115 video decoders.

Version 3 is a redesign of the board that uses TI TVP5146/5150 video decoders and a CPU clock speed of 720 MHz. To use Version 3 boards, you must install updated TI video drivers (included with TI Device Driver Developer's Kit (DDK)) to match the new decoders.

Here is how you identify the correct version number of your board:

- **Version 1**—Original board with 600 MHz DM642, Philips SAA7115 video decoders. ASSY 506840 Rev. D on back of board, 50 MHz oscillator.
- **Version 2**—Original board revised to use 720 MHz DM642, Philips SAA7115 video decoders. ASSY 506840 Rev. D on back of board, 60 MHz oscillator.

- **Version 3**—Revised board with 720 MHz DM642, TI TVP5146/5150 video decoders and HD filters. ASSY 507340 Rev. B on back of board, 60 MHz oscillator.

Configuring the Target Preferences Block for Your DM642 EVM

When you use the DM642EVM Target Preferences block, make sure that you enter the CPU clock speed that matches the CPU clock on your board. The figure below shows the correct setting of 600 for Version 1 boards in **CPU clock speed (MHz)**. For Version 2 and 3 boards, change the clock speed to 720.
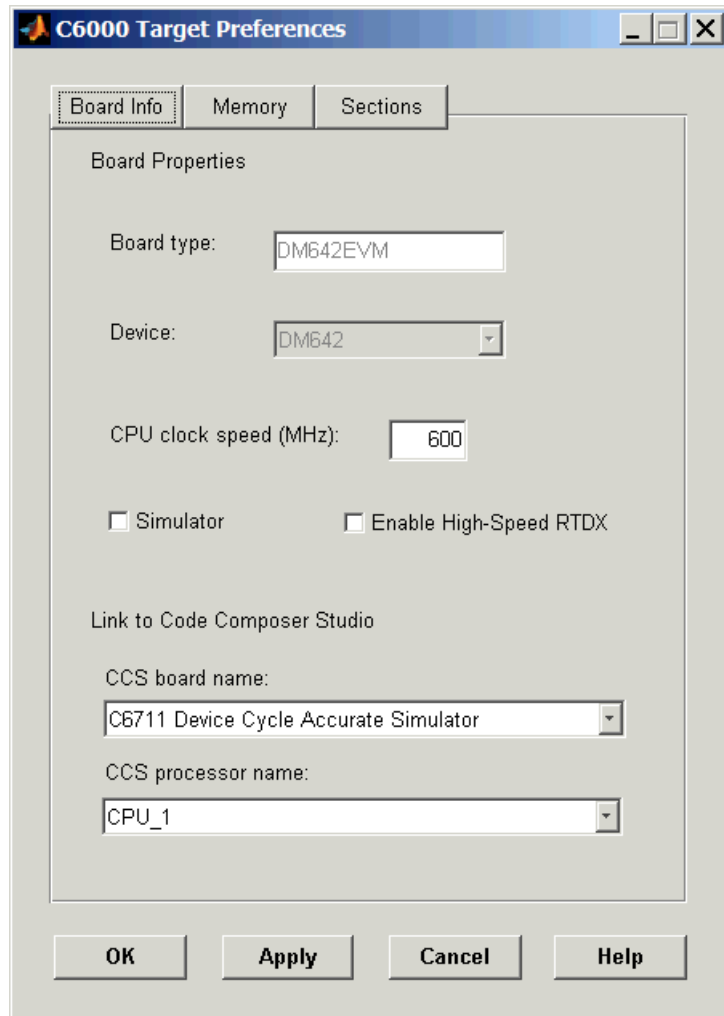
Configuring the DM642 EVM ADC Block

If you have a DM642 EVM Version 3 board, make sure that you have the updated video drivers in your CCS installation directory and that you select the correct decoder type when you use DM642 EVM Video ADC blocks as shown in the following figure.
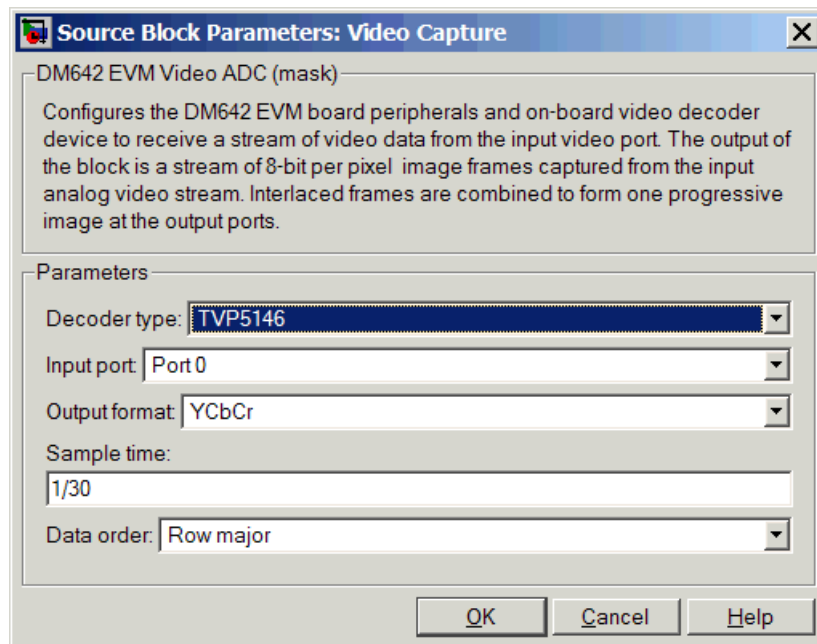
Setting Up Code Composer Studio for the DM642 EVM

Your DM642 EVM requires a separate Code Composer Studio installation. To use the EVM when you have more than one CCS installation, you need to install the CCS for the DM642 EVM in a separate location. You cannot merge your DM642 CCS installation with existing or other CCS installations. Follow the installation guidelines provided by Texas Instruments when you install CCS, to use your DM642 EVM.

Install the patch `C6000-2.20.00-FULL-to-C6000-2.20.18-FULL.EXE` as directed by Texas Instruments.

Finally, install the Device Driver Development Kit patch `ddk-v1-10-00-23.exe`.

About the Device Driver Development Kit

To use the Embedded Target for TI C6000 DSP software with your DM642, you need to install the Device Driver Developer Kit (DDK) patch that you get from Texas Instruments.

While the DDK is optional for some DM642 operations, the Embedded Target for TI C6000 DSP requires the DDK for code generation. According to TI, the DDK is the TI Device Driver Development Kit. Version 1.1 of the DDK includes device drivers for the DM642 EVM peripherals that are used by many of the examples and demos.

While the DDK is not required to run precompiled code, it is needed to rebuild or develop code. You should install the DDK in the folder `TI_DIR`.

The DDK patch for CCS is an optional patch that the Embedded Target for TI C6000 DSP requires.

About the XDS560 PCI-Bus JTAG Scan-Based Emulator

You need the XDS560 Emulator to use the DM642 with the Embedded Target for TI C6000 DSP. While the XDS510 Emulator might work, the target software has not been tested with it.

Continuing Issues with Embedded Target for TI C6000 DSP

This section details some target operations that you should know about as you use the Embedded Target for TI C6000 DSP.

- "Setting Up Code Composer Studio for the DM642 EVM" on page A-7
- "Setting the Clock Speed on the C6713 DSK" on page A-9
- "Simulink Stop Block Works Differently When Not Using DSP/BIOS Features" on page A-10

Setting the Clock Speed on the C6713 DSK

The C6713DSK PLL is not automatically set to the correct CPU Clock frequency when you try to target the board. When you power-up your DSK, it runs at a clock speed of 50 MHz. However, the C6713 is capable of running at 225 MHz.

If you generate code incorporating the DSP/BIOS real-time operating system, the PLL is automatically configured for you at run-time tjo use the correct clock speed. If you are not using DSP/BIOS in your project, you must manually configure the PLL to the correct clock rate before running your code.

Setting the PLL to Drive the CPU at 225 MHz

To set the C6713 DSK PLL to drive the CPU at 225 MHz, perform the following steps. Be sure you have defined your GEL file for your DSK in the Setup Utility for CCS.

1 Launch Code Composer Studio.

2 Open your C6713 DSK project with the GEL file.

3 Select **GEL**->**Resets**->**InitPLL** from the menu bar in CCS.

To make this happen whenever you open Code Composer Studio to use your C6713 DSK, edit the file \ti\cc\gel\dsk6713.gel. Add the following command to the StartUp() function:

```
init_pll();
```

This tells the GEL file to initialize the PLL to operate at 225 MHz.

On the DM642 EVM, ADC-DAC Loopback Does Not Display An RGB Image Correctly After Power-Up

When you set up the DM642 EVM to use loopback from the ADC to the DAC, the DAC block does not reproduce the captured image correctly immediately after you power up the board. Colors in the image are not shown correctly.

To get a clean image, reload the program to the target and run the program again. This also happens with the examples Texas Instruments ships with the DM642 EVM product.

Simulink Stop Block Works Differently When Not Using DSP/BIOS Features

If you are using the Simulink Stop block in your model, but you are not using DSP/BIOS features, your model might take longer to stop when it is running on the target than if you are using DSP/BIOS.

The condition the model uses to detect the stop processing flag is different when you do not use DSP/BIOS. The result is that the model may not detect and respond to the flag as promptly, taking longer to stop the running model on the target.

W
working directory 2-108
`wrap` 2-27