TMS320C6000
Code Composer Studio
Tutorial

Literature Number: SPRU301C
February 2000

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserves the right to make changes to their products or to discontinue any product or service without notice, and advises customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current and complete. All products are sold subject to the terms and conditions of sale at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Preface

Read This First

About This Manual

Code Composer Studio speeds and enhances the development process for programmers who create and test real-time, embedded signal processing applications. Code Composer Studio extends the capabilities of the Code Composer Integrated Development Environment (IDE) to include full awareness of the DSP target by the host and real-time analysis tools.

This tutorial assumes that you have Code Composer Studio, which includes the TMS320C6000 code generation tools along with the APIs and plug-ins for both DSP/BIOS and RTDX. This manual also assumes that you have installed a target board in your PC containing the DSP device.

If you only have Code Composer Studio Simulator and the code generation tools, but not the complete Code Composer Studio, you can perform the steps in Chapter 2 and Chapter 4.

If you are using the DSP simulator instead of a board, you are also limited to performing the steps in Chapter 2 and Chapter 4.

This tutorial introduces you to some of the key features of Code Composer Studio. The intention is not to provide an exhaustive description of every feature. Instead, the objective is to prepare you to begin DSP development with Code Composer Studio.

Notational Conventions

This document uses the following conventions:

- The TMS320C6000 core is also referred to as 'C6000.
- Code Composer Studio generates files with extensions of .s62 and .h62. These files can also be used with both the TMS320C6201 and the TMS320C6701. DSP/BIOS does not use the floating-point instructions that are supported by the TMS320C6701.
- Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
Void copy(HST_Obj *input, HST_Obj *output)
{
    PIP_Obj     *in, *out;
    Uns         *src, *dst;
    Uns         size;
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered. Syntax that is entered on a command line is centered. Syntax that is used in a text file is left-justified.
- Square brackets ([ ]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

Related Documentation from Texas Instruments

The following books describe the devices, related support tools, and Code Composer Studio.

Most of these documents are available in Adobe Acrobat format after you install Code Composer Studio. To open a document, from the Windows Start menu, choose Programs–>Code Composer Studio 'C6000–>Documentation.

To obtain a printed copy of any of these documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

Code Composer Studio User's Guide (literature number SPRU328) explains how to use the Code Composer Studio development environment to build and debug embedded real-time DSP applications.

TMS320C6000 DSP/BIOS User's Guide (literature number SPRU303a) describes how to use DSP/BIOS tools and APIs to analyze embedded real-time DSP applications.

TMS320C6000 DSP/BIOS API Reference Guide (literature number SPRU403) describes how to use DSP/BIOS tools and APIs to analyze embedded real-time DSP applications.

TMS320C6000 Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C62x/C67x Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C62x/C67x DSPs and includes application program examples.

TMS320C62x/C67x CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C62x/C67x CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/'C6701 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, multichannel buffered serial

ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C62x Technical Brief (literature number SPRU197) gives an introduction to the digital signal processor, development tools, and third-party support.

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6201 and provides pinouts, electrical specifications, and timings for the device.

TMS320C6701 Digital Signal Processor Data Sheet (literature number SPRS067) describes the features of the TMS320C6701 and provides pinouts, electrical specifications, and timings for the device.

TMS320 DSP Designer's Notebook: Volume 1 (literature number SPRT125) presents solutions to common design problems using 'C2x, 'C3x, 'C4x, 'C5x, and other TI DSPs.

Related Documentation

You can use the following books to supplement this user's guide:

American National Standard for Information Systems-Programming Language C X3.159-1989, American National Standards Institute.

The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall Press, 1988.

Programming in ANSI C, Kochan, Steve G. Sams Publishing, 1994.

C: A Reference Manual, Harbison, Samuel and Guy Steele. Prentice Hall Computer Books, 1994.

Trademarks

Contents

Chapter 1

Code Composer Studio Overview

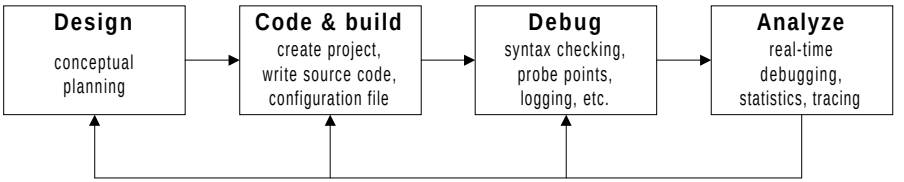
This chapter provides an overview of the Code Composer Studio software development process, the components of Code Composer Studio, and the files and variables used by Code Composer Studio.

Code Composer Studio speeds and enhances the development process for programmers who create and test real-time, embedded signal processing applications. It provides tools for configuring, building, debugging, tracing, and analyzing programs.

Topic **Page**

1.1 Code Composer Studio Development

Code Composer Studio extends the basic code generation tools with a set of debugging and real-time analysis capabilities. Code Composer Studio supports all phases of the development cycle shown here:



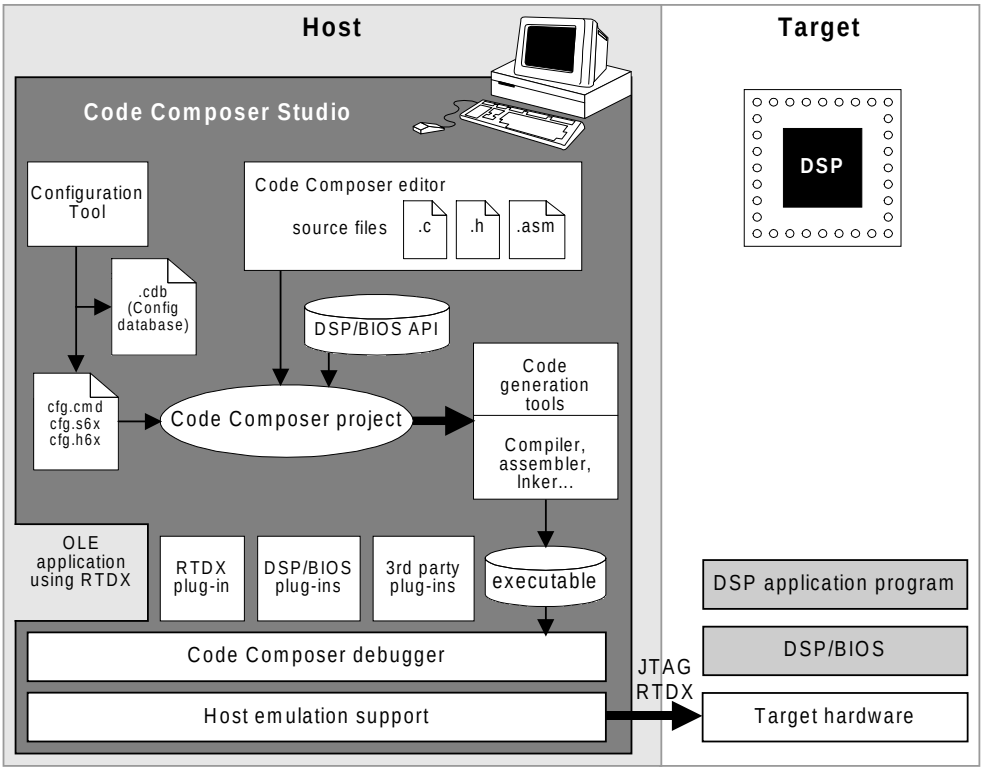
In order to use this tutorial, you should have already done the following:

- ❏ **Install target board and driver software**. Follow the installation instructions provided with the board. If you are using the simulator or a target board whose driver software is provided with this product you can perform the steps in the Installation Guide for this product.
- ❏ **Install Code Composer Studio**. Follow the installation instructions. If you have Code Composer Studio Simulator and the TMS320C6000 code generation tools, but not the full Code Composer Studio, you can perform the steps in Chapter 2 and in Chapter 4.
- ❏ **Run Code Composer Studio Setup**. The setup program allows Code Composer Studio to use the drivers installed for the target board.

Code Composer Studio includes the following components:

- ❏ TMS320C6000 code generation tools: see section 1.2
- ❏ Code Composer Studio Integrated Development Environment (IDE): see section 1.3
- ❏ DSP/BIOS plug-ins and API: see section 1.4
- ❏ RTDX plug-in, host interface, and API: see section 1.5

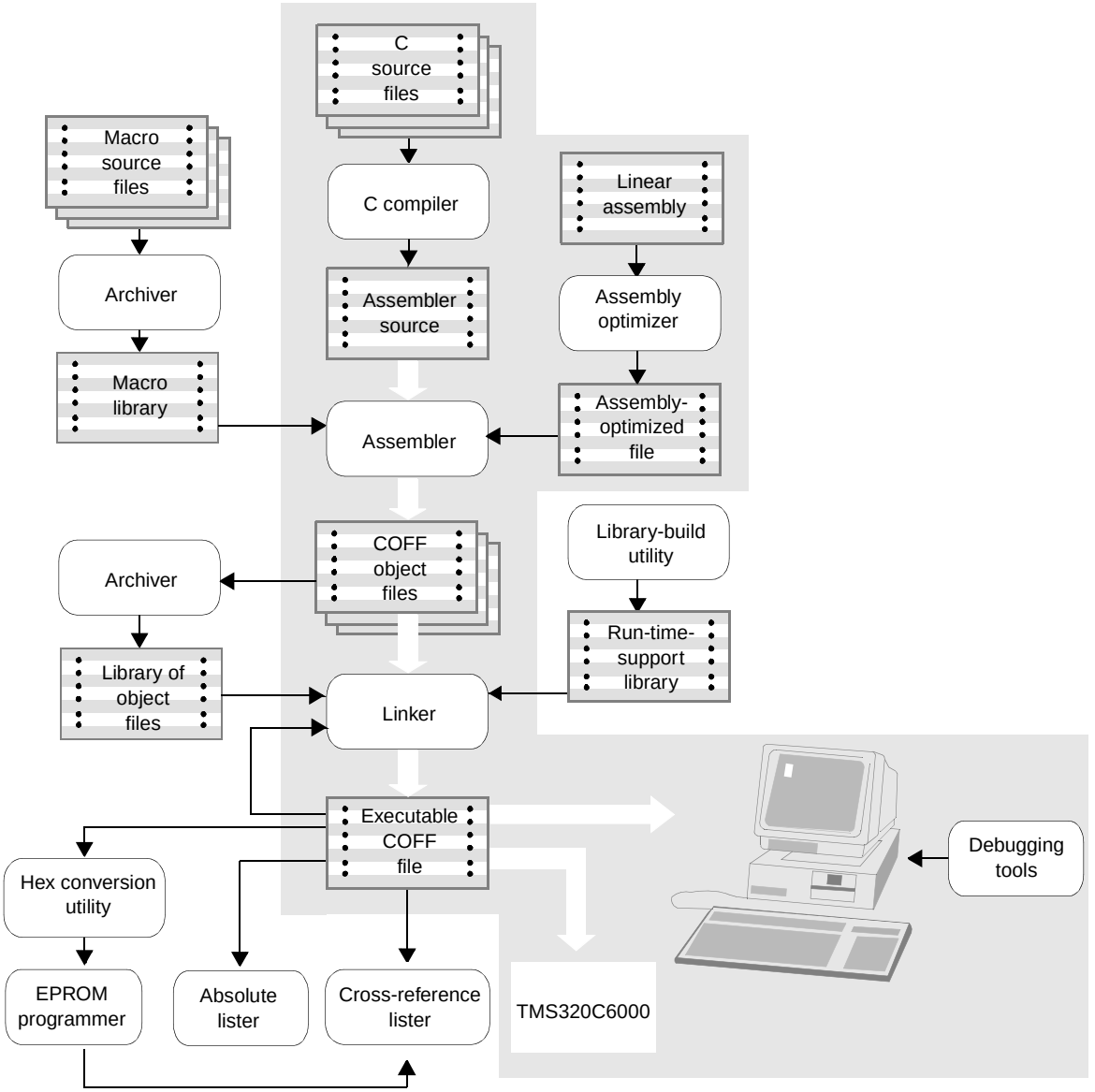
These components work together as shown here:



1.2 Code Generation Tools

The code generation tools provide the foundation for the development environment provided by Code Composer Studio. Figure 1–1 shows a typical software development flow. The most common software development path for C language programs is shaded. Other portions are peripheral functions that enhance the development process.

Figure 1–1 Software Development Flow



The following list describes the tools shown in Figure 1-1:

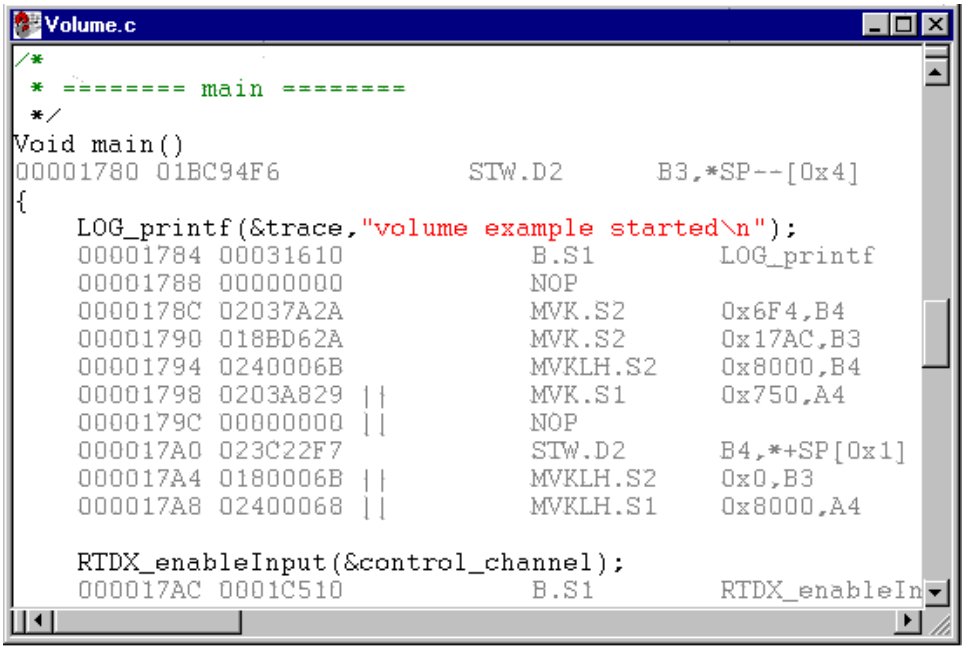
- The **C compiler** accepts C source code and produces assembly language source code. See the *TMS320C6000 Optimizing C Compiler User's Guide* for details.
- The **assembler** translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). See the *TMS320C6000 Assembly Language Tools User's Guide* for details.
- The **assembly optimizer** allows you to write linear assembly code without being concerned with the pipeline structure or with assigning registers. It assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly that takes advantage of software pipelining. See the *TMS320C6000 Optimizing C Compiler User's Guide* and the *TMS320C62x/C67x Programmer's Guide* for details.
- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. See the *TMS320C6000 Optimizing C Compiler User's Guide* and the *TMS320C6000 Assembly Language Tools User's Guide* for details.
- The **archiver** allows you to collect a group of files into a single archive file, called a library. The archiver also allows you to modify a library by deleting, replacing, extracting, or adding members. See the *TMS320C6000 Assembly Language Tools User's Guide* for details.
- You can use the **library-build utility** to build your own customized run-time-support library. See the *TMS320C6000 Optimizing C Compiler User's Guide* for details.
- The **run-time-support libraries** contain ANSI standard run-time-support functions, compiler-utility functions, floating-point arithmetic functions, and I/O functions that are supported by the C compiler. See the *TMS320C6000 Optimizing C Compiler User's Guide* for details.
- The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. You can download the converted file to an EPROM programmer. See the *TMS320C6000 Assembly Language Tools User's Guide* for details.
- The **cross-reference lister** uses object files to cross-reference symbols, their definitions, and their references in the linked source files. See the *TMS320C6000 Assembly Language Tools User's Guide* for details.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You assemble the .abs files to produce a listing that contains absolute addresses rather than relative addresses. Without the absolute lister, producing such a listing would be tedious and require many manual operations.

1.3 Code Composer Studio Integrated Development Environment

The Code Composer Studio Integrated Development Environment (IDE) is designed to allow you to edit, build, and debug DSP target programs.

1.3.1 Program Code Editing Features

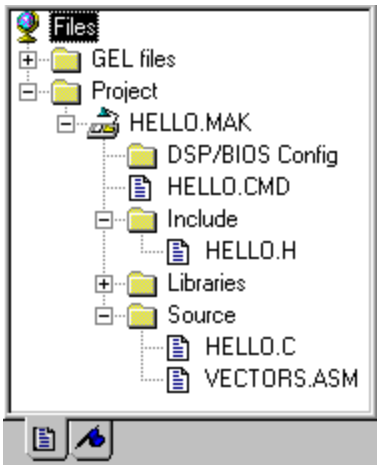
Code Composer Studio allows you to edit C and assembly source code. You can also view C source code with the corresponding assembly instructions shown after the C statements.



The integrated editor provides support for the following activities:

- ❏ Highlighting of keywords, comments, and strings in color
- ❏ Marking C blocks in parentheses and braces, finding matching or next parenthesis or brace
- ❏ Increasing and decreasing indentation level, customizable tab stops
- ❏ Finding and replacing in one or more files, find next and previous, quick search
- ❏ Undoing and redoing multiple actions
- ❏ Getting context-sensitive help
- ❏ Customizing keyboard command assignments

1.3.2 Application Building Features



Within Code Composer Studio, you create an application by adding files to a project. The project file is used to build the application. Files in a project can include C source files, assembly source files, object files, libraries, linker command files, and include files.

You can use a window to specify the options you want to use when compiling, assembling, and linking a project.

Using a project, Code Composer Studio can create a full build or an incremental build and can compile individual files. It can also scan files to build an include file dependency tree for the entire project.

Code Composer Studio's build facilities can be used as an alternative to traditional makefiles. If you want to continue using traditional makefiles for your project, Code Composer Studio also permits that.

1.3.3 Application Debugging Features

Code Composer Studio provides support for the following debugging activities:

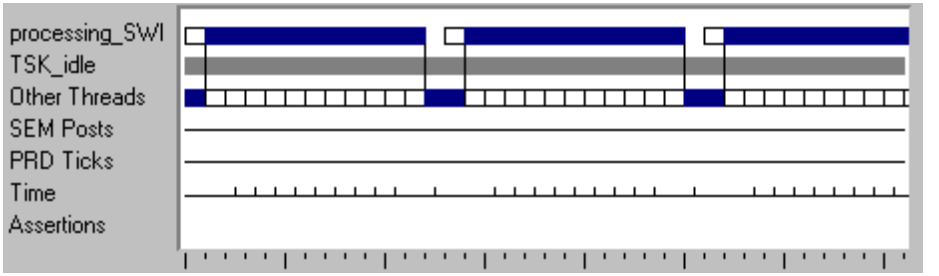
- ❏ Setting breakpoints with a number of stepping options
- ❏ Automatically updating windows at breakpoints
- ❏ Watching variables
- ❏ Viewing and editing memory and registers
- ❏ Viewing the call stack
- ❏ Using Probe Point tools to stream data to and from the target and to gather memory snapshots
- ❏ Graphing signals on the target
- ❏ Profiling execution statistics
- ❏ Viewing disassembled and C instructions executing on target

Code Composer Studio also provides the GEL language, which allows developers to add functions to the Code Composer Studio menus for commonly performed tasks.

1.4 DSP/BIOS Plug-ins

During the analysis phase of the software development cycle, traditional debugging features are ineffective for diagnosing subtle problems that arise from time-dependent interactions.

The Code Composer Studio plug-ins provided with DSP/BIOS support such real-time analysis. You can use them to visually probe, trace, and monitor a DSP application with minimal impact on real-time performance. For example, the Execution Graph shown below displays the sequence in which various program threads execute. (Threads is a general term used to refer to any thread of execution. For example, hardware ISRs, software interrupts, tasks, idle functions, and periodic functions are all threads.)



The DSP/BIOS API provides the following real-time analysis capabilities:

- **Program tracing**. Displaying events written to target logs and reflecting dynamic control flow during program execution
- **Performance monitoring**. Tracking statistics that reflect the use of target resources, such as processor loading and thread timing
- **File streaming**. Binding target-resident I/O objects to host files

DSP/BIOS also provides a priority-based scheduler that you can choose to use in your applications. This scheduler supports periodic execution of functions and multi-priority threading.

1.4.1 DSP/BIOS Configuration

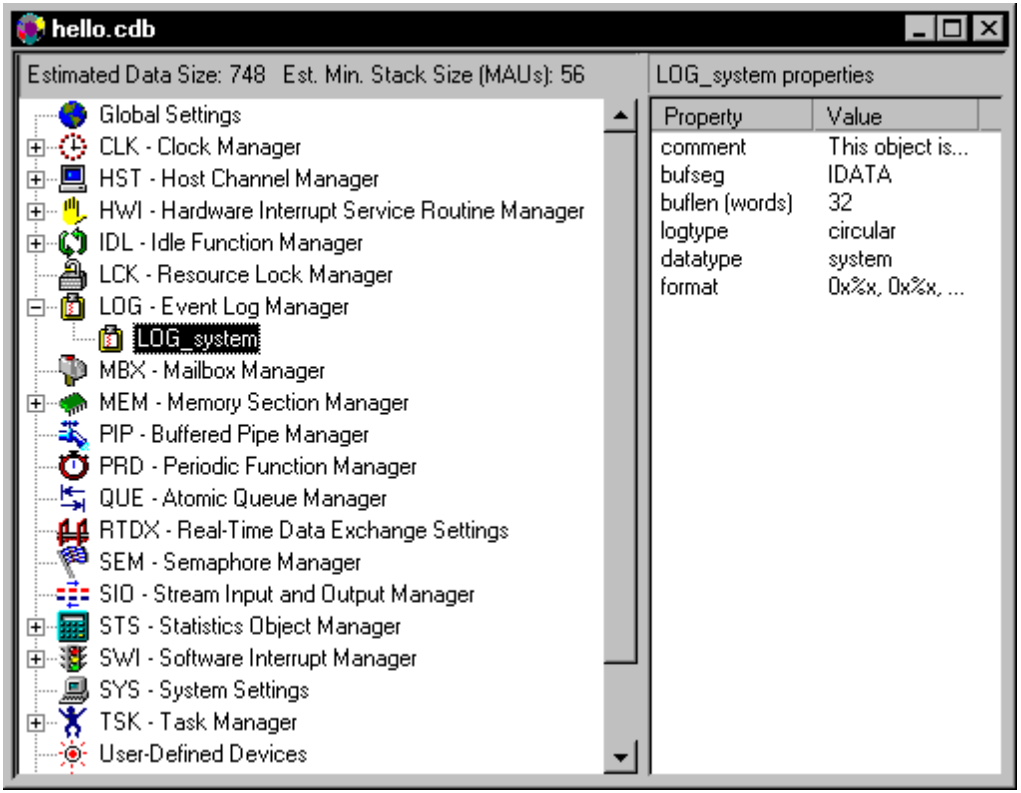
You can create configuration files in the Code Composer Studio environment that define objects used by the DSP/BIOS API. This file also simplifies memory mapping and hardware ISR vector mapping, so you may want to use it even if you are not using the DSP/BIOS API.

A configuration file has two roles:

- It lets you set global run-time parameters.

- It serves as a visual editor for creating and setting properties for run-time objects that are used by the target application's DSP/BIOS API calls. These objects include software interrupts, I/O pipes, and event logs.

When you open a configuration file in Code Composer Studio, a window like the following one appears.



Unlike systems that create objects at run time with API calls that require extra target overhead (especially code space), all DSP/BIOS objects are statically configured and bound into an executable program image. In addition to minimizing the target memory footprint by eliminating run-time code and optimizing internal data structures, this static configuration strategy detects errors earlier by validating object properties before program execution.

You can use a configuration file in both programs that use the DSP/BIOS API and in programs that do not. A configuration file simplifies ISR vector and memory section addressing for all programs.

Saving a configuration file generates several files that you must link with your application. See section 1.7.2, page 1–16 for details about these files.

1.4.2 DSP/BIOS API Modules

Unlike traditional debugging, which is external to the executing program, the DSP/BIOS features require the target program to be linked with certain DSP/BIOS API modules.

A program can use one or more DSP/BIOS modules by defining DSP/BIOS objects in a configuration file, declaring these objects as external, and calling DSP/BIOS API functions in the source code. Each module has a separate C header file or assembly macro file you can include in your program. This allows you to minimize the program size in a program that uses some, but not all, DSP/BIOS modules.

The DSP/BIOS API calls (in C and assembly) are optimized to use minimal resources on your target DSP.

The DSP/BIOS API is divided into the following modules. All the API calls within a module begin with the letter codes shown here.

- **ATM**. This module provides atomic functions that can be used to manipulate shared data.
- **C62**. This module provides DSP-specific functions to manage interrupts.
- **CLK**. The on-chip timer module controls the on-chip timer and provides a logical 32-bit real-time clock with a high-resolution interrupt rate as fine as the resolution of the on-chip timer register (4 instruction cycles) and a low-resolution interrupt rate as long as several milliseconds or longer.
- **DEV**. This module allows you to create and use your own device drivers.
- **HST**. The host input/output module manages host channel objects, which allow an application to stream data between the target and the host. Host channels are statically configured for input or output.
- **HWI**. The hardware interrupt module provides support for hardware interrupt routines. In a configuration file, you can assign functions that run when hardware interrupts occur.
- **IDL**. The idle function module manages idle functions, which are run in a loop when the target program has no higher priority functions to perform.
- **LCK**. The lock module manages shared global resources, and is used to arbitrate access to this resource among several competing tasks.
- **LOG**. The log module manages LOG objects, which capture events in real time while the target program executes. You can use system logs or define your own logs. You can view messages in these logs in real time with Code Composer Studio.
- **MBX**. The mailbox module manages objects that pass messages from one task to another. Tasks block when waiting for a mailbox message.

- **MEM**. The memory module allows you to specify the memory segments required to locate the various code and data sections of a target program.
- **PIP**. The data pipe module manages data pipes, which are used to buffer streams of input and output data. These data pipes provide a consistent software data structure you can use to drive I/O between the DSP device and other real-time peripheral devices.
- **PRD**. The periodic function module manages periodic objects, which trigger cyclic execution of program functions. The execution rate of these objects can be controlled by the clock rate maintained by the CLK module or by regular calls to PRD_tick, usually in response to hardware interrupts from peripherals that produce or consume streams of data.
- **QUE**. The queue module manages data queue structures.
- **RTDX**. Real-Time Data Exchange permits the data to be exchanged between the host and target in real time, and then to be analyzed and displayed on the host using any OLE automation client. See section 1.5 for more information.
- **SEM**. The semaphore module manages counting semaphores that may be used for task synchronization and mutual exclusion.
- **SIO**. The stream module manages objects that provide efficient real-time device-independent I/O.
- **STS**. The statistics module manages statistics accumulators, which store key statistics while a program runs. You can view these statistics in real time with Code Composer Studio.
- **SWI**. The software interrupt module manages software interrupts, which are patterned after hardware interrupt service routines (ISRs). When a target program posts an SWI object with an API call, the SWI module schedules execution of the corresponding function. Software interrupts can have up to 15 priority levels; all levels are below the priority level of hardware ISRs.
- **SYS**. The system services module provides general-purpose functions that perform basic system services, such as halting program execution and printing formatted text.
- **TRC**. The trace module manages a set of trace control bits which control the real-time capture of program information through event logs and statistics accumulators. There are no TRC objects, so the trace module is not listed in configuration files.
- **TSK**. The task module manages task threads, which are blocking threads with lower priority than software interrupts.

For details, see the online help or the *TMS320C6000 DSP/BIOS User's Guide* and *TMS320C6000 DSP/BIOS API Reference Guide*.

1.5 Hardware Emulation and Real-Time Data Exchange

TI DSPs provide on-chip emulation support that enables Code Composer Studio to control program execution and monitor real-time program activity. Communication with this on-chip emulation support occurs via an enhanced JTAG link. This link is a low-intrusion way of connecting into any DSP system. An emulator interface, like the TI XDS510, provides the host side of the JTAG connection. Evaluation boards like the C6x EVM provide an on-board JTAG emulator interface for convenience.

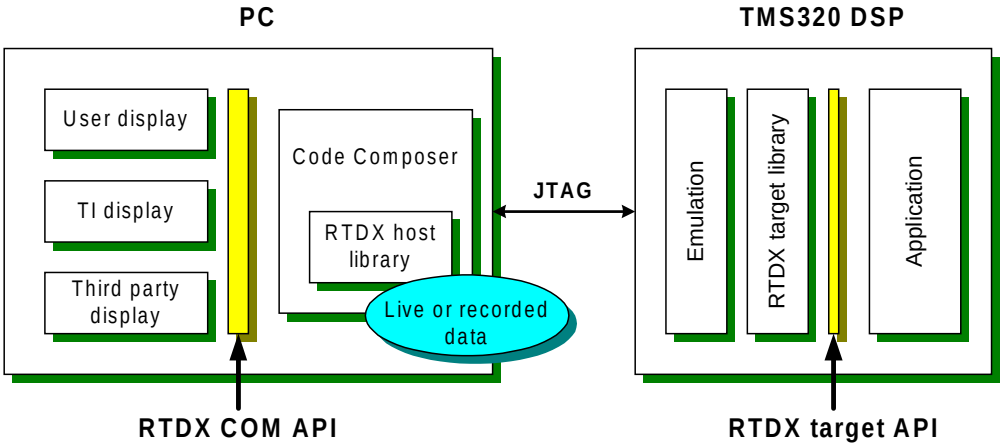
The on-chip emulation hardware provides a variety of capabilities:

- ❏ Starting, stopping, or resetting the DSP
- ❏ Loading code or data into the DSP
- ❏ Examining the registers or memory of the DSP
- ❏ Hardware instruction or data-dependent breakpoints
- ❏ A variety of counting capabilities including cycle-accurate profiling
- ❏ Real-time data exchange (RTDX) between the host and the DSP

Code Composer Studio provides built-in support for these on-chip capabilities. In addition, RTDX capability is exposed through host and DSP APIs, allowing for bi-directional real-time communications between the host and DSP.

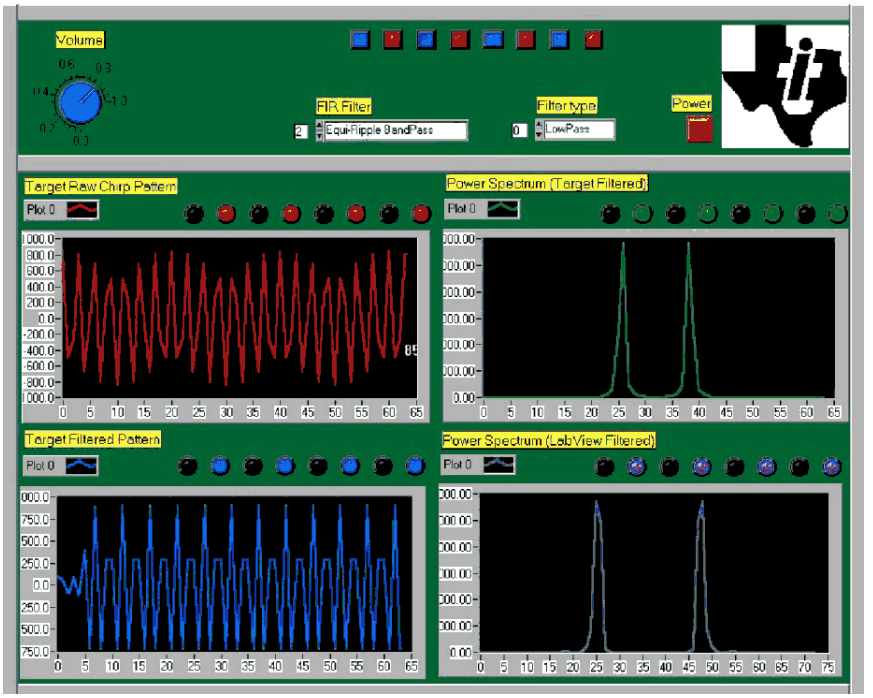
RTDX provides real-time, continuous visibility into the way DSP applications operate in the real world. RTDX allows system developers to transfer data between a host computer and DSP devices without stopping their target application. The data can be analyzed and visualized on the host using any OLE automation client. This shortens development time by giving designers a realistic representation of the way their systems actually operate.

RTDX consists of both target and host components. A small RTDX software library runs on the target DSP. The designer's DSP application makes function calls to this library's API in order to pass data to or from it. This library uses the on-chip emulation hardware to move data to or from the host platform via an enhanced JTAG interface. Data transfer to the host occurs in real time while the DSP application is running.



On the host platform, an RTDX library operates in conjunction with Code Composer Studio. Display and analysis tools can communicate with RTDX via an easy-to-use COM API to obtain the target data or send data to the DSP application. Designers may use standard software display packages, such as National Instruments' LabVIEW, Quinn-Curtis' Real-Time Graphics Tools, or Microsoft Excel. Alternatively, designers can develop their own Visual Basic or Visual C++ applications.

RTDX can also record real-time data and play it back for non-real-time analysis. The following sample display features National Instruments' LabVIEW. On the target a raw signal pattern is run through an FIR filter, and both the raw and filtered signals are sent to the host via RTDX. On the host the LabVIEW display obtains the signal data via the RTDX COM API. These two signals appear on the left half of the display. To confirm that the target's FIR filter is operating correctly, power spectrums are produced. The target's filtered signal is run through a LabVIEW power spectrum and displayed on the top right. The target's raw signal is run through a LabVIEW FIR filter followed by a LabVIEW power spectrum and displayed on the bottom right. Comparison of these two power spectrums validates the target FIR filter.



RTDX is well-suited for a variety of control, servo, and audio applications. For example, wireless telecommunications manufacturers can capture the outputs of their vocoder algorithms to check the implementations of speech applications. Embedded control systems also benefit. Hard disk drive designers can test their applications without crashing the drive with improper signals to the servo motor, and engine control designers can analyze changing factors like heat and environmental conditions while the control application is running. For all of these applications, users can select visualization tools that display information in a way that is most meaningful to them. Future TI DSPs will enable RTDX bandwidth increases, providing greater system visibility to an even larger number of applications. For more information on RTDX, see the RTDX online help available within Code Composer Studio.

1.6 Third-Party Plug-ins

Third-party software providers can create ActiveX plug-ins that complement the functionality of Code Composer Studio. A number of third-party plug-ins are available for a variety of purposes.

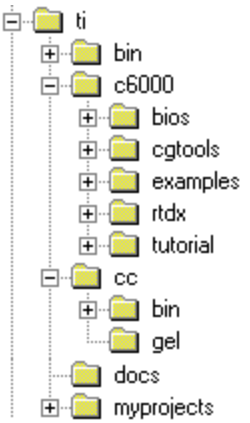
1.7 Code Composer Studio Files and Variables

The following sections provide an overview of the folders that contain the Code Composer Studio files, the types of files you use, and the environment variables used by Code Composer Studio.

1.7.1 Installation Folders

The installation process creates the subfolders shown here in the folder where you install Code Composer Studio (typically c:\ti). Additionally, subfolders are created in the Windows directory (c:\windows or c:\winnt).

The c:\ti structure contains the following directories:



- **bin**. Various utility programs
- **c6000\bios**. Files used when building programs that use the DSP/BIOS API
- **c6000\cgtools**. The Texas Instruments code generation tools
- **c6000\examples**. Code examples
- **c6000\rtdx**. Files for use with RTDX
- **c6000\tutorial**. The examples you use in this manual
- **cc\bin**. Program files for the Code Composer Studio environment
- **cc\gel**. GEL files for use with Code Composer Studio
- **docs**. Documentation and manuals in PDF format. If you did not choose to install the complete documentation, see the CD-ROM for manuals in PDF format.
- **myprojects**. Location provided for your copies of the tutorial examples and your project files

The following directory structure is added to the Windows directory:

- **ti\drivers**. Files for various DSP board drivers
- **ti\plugins**. Plug-ins for use with Code Composer Studio
- **ti\uninstall**. Files supporting Code Composer Studio software removal

1.7.2 File Extensions

While using Code Composer Studio, you work with files that have the following file-naming conventions:

- **project.mak**. Project file used by Code Composer Studio to define a project and build a program
- **program.c**. C program source file(s)
- **program.asm**. Assembly program source file(s)
- **filename.h**. Header files for C programs, including header files for DSP/BIOS API modules
- **filename.lib**. Library files
- **project.cmd**. Linker command files
- **program.obj**. Object files compiled or assembled from your source files
- **program.out**. An executable program for the target (fully compiled, assembled, and linked). You can load and run this program with Code Composer Studio.
- **project.wks**. Workspace file used by Code Composer Studio to store information about your environment settings
- **program.cdb**. Configuration database file created within Code Composer Studio. This file is required for applications that use the DSP/BIOS API, and is optional for other applications. The following files are also generated when you save a configuration file:
  - **programcfg.cmd**. Linker command file
  - **programcfg.h62**. Header file
  - **programcfg.s62**. Assembly source file

  Although these files have extensions of .s62 and .h62, they can also be used with the TMS320C6701. DSP/BIOS does not need to use the floating-point instructions supported by the TMS320C6701, therefore only one version of the software is required to support both DSPs.

1.7.3 Environment Variables

The installation procedure defines the following variables in your autoexec.bat file (for Windows 95 and 98) or as environment variables (for Windows NT):

Table 1–1 Environment Variables

Variable	Description
C6X_A_DIR	A search list used by the assembler to find library and include files for DSP/BIOS, RTDX, and the code generation tools. See the *TMS320C6000 Assembly Language Tools User's Guide* for details.
C6X_C_DIR	A search list used by the compiler and linker to find library and include files for DSP/BIOS, RTDX, and the code generation tools. See the *TMS320C6000 Optimizing C Compiler User's Guide* for details.
PATH	A list of folders is added to your PATH definition. The default is to add the c:\ti\c6000\cgtools\bin and c:\ti\bin folders.

1.7.4 Increasing DOS Environment Space

If you are using Windows 95, you may need to increase your DOS shell environment space to support the environment variables required to build Code Composer Studio applications.

Add the following line to the config.sys file and then restart your computer:

```
shell=c:\windows\command.com /e:4096 /p
```

Chapter 2

Developing a Simple Program

This chapter introduces Code Composer Studio and shows the basic process used to create, build, debug, and test programs.

In this chapter, you create and test a simple program that displays a hello world message.

This tutorial introduces you to some of the key features of Code Composer Studio. The intention is not to provide an exhaustive description of every feature. Instead, the objective is to prepare you to begin DSP software development with Code Composer Studio.

In order to use this tutorial, you should have already installed Code Composer Studio according to the installation instructions. It is recommended that you use Code Composer Studio with a target board rather than with the simulator. If you have Code Composer and the Code Generation Tools, but not Code Composer Studio, or if you are using the simulator, you can perform the steps in Chapter 2 and Chapter 4 only.

Topic **Page**

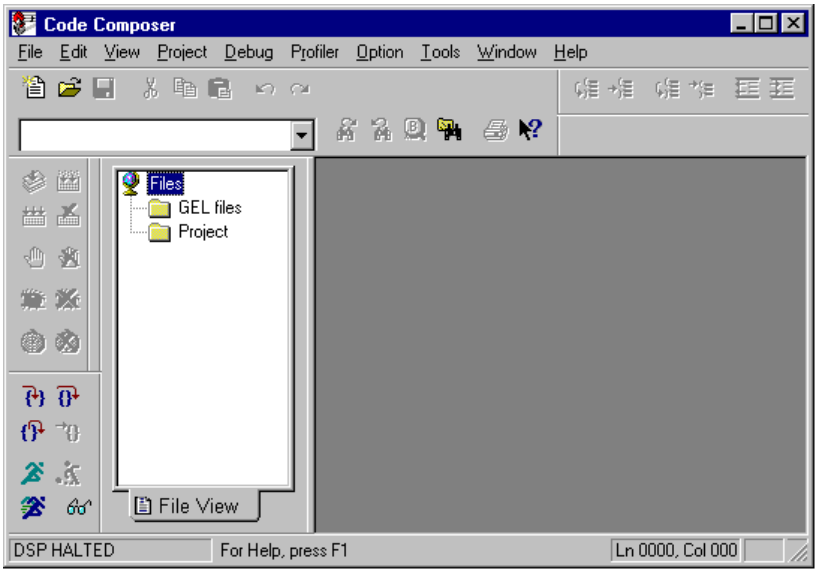
2.1 Creating a New Project

In this chapter, you create a project with Code Composer Studio and add source code files and libraries to the project. Your project uses standard C library functions to display a hello world message.

1) If you installed Code Composer Studio in c:\ti, create a folder called hello1 in the c:\ti\myprojects folder. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)

2) Copy all files from the c:\ti\c6000\tutorial\hello1 folder to this new folder.

3) From the Windows Start menu, choose Programs→Code Composer Studio 'C6000→CCStudio. (Or, double-click the Code Composer Studio icon on your desktop.)



Note: Code Composer Studio Setup

If you get an error message the first time you try to start Code Composer Studio, make sure you ran Code Composer Setup after installing Code Composer Studio. If you have a target board other than the ones mentioned in the instructions provided with the CD-ROM, see the documentation provided with your target board for the correct I/O port address.

4) Choose the Project→New menu item.

5) In the Save New Project As window, select the working folder you created and click Open. Type myhello as the filename and click Save. Code Composer Studio creates a project file called myhello.mak. This file stores your project settings and references the various files used by your project.

2.2 Adding Files to a Project

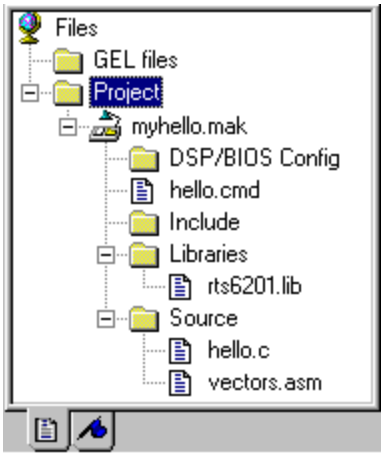
1) Choose Project→Add Files to Project. Select hello.c and click Open.

2) Choose Project→Add Files to Project. Select Asm Source Files (*.a*, *.s*) in the Files of type box. Select vectors.asm and click Open. This file contains assembly instructions needed to set the RESET interrupt service fetch packets (ISFPs) to branch to the program's C entry point, c_int00. (For more complex programs, you can define additional interrupt vectors in vectors.asm, or you can use DSP/BIOS as shown in section 3.1, page 3-2 to define all the interrupt vectors automatically.)

3) Choose Project→Add Files to Project. Select Linker Command File (*.cmd) in the Files of type box. Select hello.cmd and click Open. This file maps sections to memory.

4) Choose Project→Add Files to Project. Go to the compiler library folder (C:\ti\c6000\cgtools\lib). Select Object and Library Files (*.o*, *.lib) in the Files of type box. Select rts6201.lib and click Open. This library provides run-time support for the target DSP.(If you are using the TMS320C6701 and floating point values, select rts6701.lib instead.)

5) Expand the Project list by clicking the + signs next to Project, myhello.mak, Libraries, and Source. This list is called the Project View.



Note: Opening Project View

If you do not see the Project View, choose View→Project. Click the File icon at the bottom of the Project View if the Bookmarks icon is selected.

6) Notice that include files do not yet appear in your Project View. You do not need to manually add include files to your project, because Code Composer Studio finds them automatically when it scans for dependencies as part of the build process. After you build your project, the include files appear in the Project View.

If you need to remove a file from the project, right click on the file in the Project View and choose Remove from project in the pop-up menu.

When building the program, Code Composer Studio finds files by searching for project files in the following path order:

- ❏ The folder that contains the source file.
- ❏ The folders listed in the Include Search Path for the compiler or assembler options (from left to right).
- ❏ The folders listed in the definitions of the C6X_C_DIR (compiler) and C6X_A_DIR (assembler) environment variables (from left to right). The C6X_C_DIR environment variable defined by the installation points to the folder that contains the rts6201.lib file.

2.3 Reviewing the Code

1) Double-click on the HELLO.C file in the Project View. You see the source code in the right half of the window.

2) You may want to make the window larger so that you can see more of the source code at once. You can also choose a smaller font for this window by choosing Option→Font.

```
/* ======== hello.c ======== */
#include <stdio.h>
#include "hello.h"

#define BUFSIZE 30

struct PARMS str =
{
    2934,
    9432,
    213,
    9432,
    &str
};

/*
 * ======== main ========
 */
void main()
{
#ifdef FILEIO
    int     i;
    char    scanStr[BUFSIZE];
    char    fileStr[BUFSIZE];
    size_t  readSize;
    FILE    *fptr;
#endif

    /* write a string to stdout */
    puts("hello world!\n");
```

```
#ifdef FILEIO
    /* clear char arrays */
    for (i = 0; i < BUFSIZE; i++) {
        scanStr[i] = 0   /* deliberate syntax error */
        fileStr[i] = 0;
    }

    /* read a string from stdin */
    scanf("%s", scanStr);

    /* open a file on the host and write char array */
    fptr = fopen("file.txt", "w");
    fprintf(fptr, "%s", scanStr);
    fclose(fptr);

     /* open a file on the host and read char array */
    fptr = fopen("file.txt", "r");
    fseek(fptr, 0L, SEEK_SET);
    readSize = fread(fileStr, sizeof(char), BUFSIZE, fptr);
    printf("Read a %d byte char array: %s \n", readSize, fileStr);
    fclose(fptr);
#endif
}
```

When FILEIO is undefined, this is a simple program that uses the standard puts() function to display a hello world message. When you define FILEIO (as you do in section 2.5, page 2-7), this program prompts for a string and prints it to a file. It then reads the string from the file and prints it and a message about its length to standard output.

2.4 Building and Running the Program

Code Composer Studio automatically saves changes to the project setup as you make them. In case you exited from Code Composer Studio after the previous section, you can return to the point where you stopped working by restarting Code Composer Studio and using Project→Open.

Note: Resetting the Target DSP

If Code Composer Studio displays an error message that says it cannot initialize the target DSP, choose the Debug→Reset DSP menu item. If this does not correct the problem, you may need to run a reset utility provided with your target board.

To build and run the program, follow these steps:

1) Choose Project→Rebuild All or click the (Rebuild All) toolbar button. Code Composer Studio recompiles, reassembles, and relinks all the files in the project. Messages about this process are shown in a frame at the bottom of the window.

2) Choose File→Load Program. Select the program you just rebuilt, myhello.out, and click Open. (It should be in the c:\ti\myprojects\hello1 folder unless you installed Code Composer Studio elsewhere.) Code Composer Studio loads the program onto the target DSP and opens a Dis-Assembly window that shows the disassembled instructions that make up the program. (Notice that Code Composer Studio also automatically opens a tabbed area at the bottom of the window to show output the program sends to stdout.)

3) Click on an assembly instruction in the Dis-Assembly window. (Click on the actual instruction, not the address of the instruction or the fields passed to the instruction.) Press the F1 key. Code Composer Studio searches for help on that instruction. This is a good way to get help on an unfamiliar assembly instruction.

4) Choose Debug→Run or click the (Run) toolbar button.

Note: Screen Size and Resolution

Depending on the size and resolution of your screen, part of the toolbar may be hidden by the Build window. To view the entire toolbar, right-click in the Build window and deselect Allow Docking.

When you run the program, you see the hello world message in the Stdout tab.

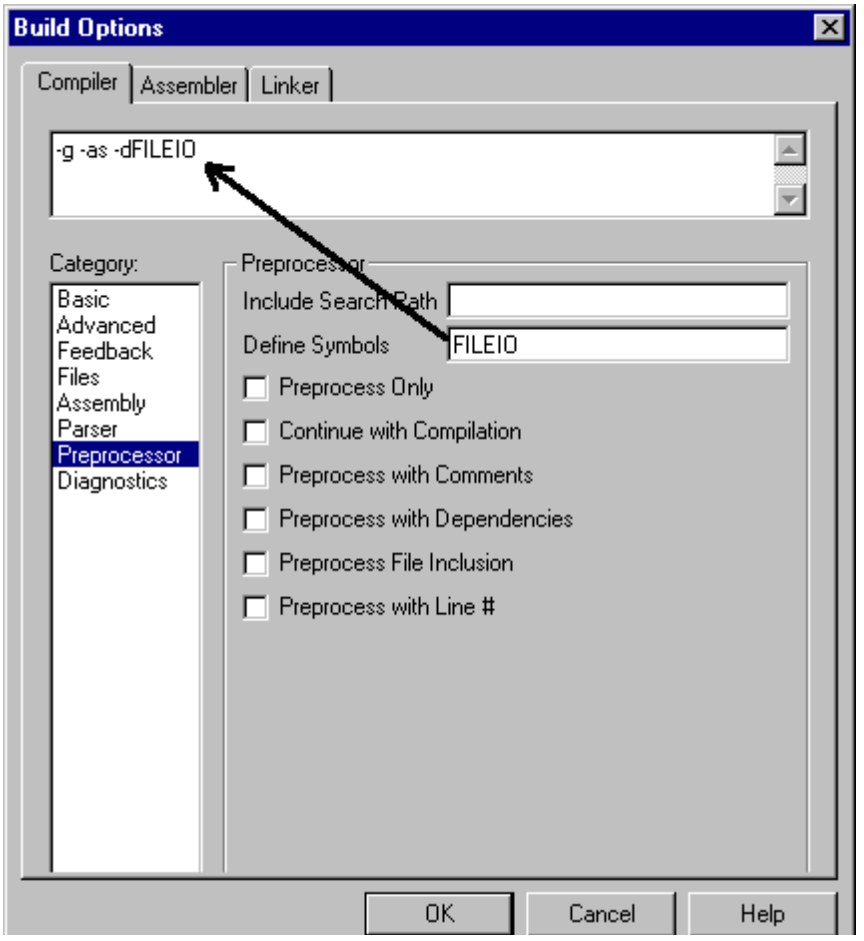
2.5 Changing Program Options and Fixing Syntax Errors

In the previous section, the portion of the program enclosed by the preprocessor commands (#ifdef and #endif) did not run because FILEIO was undefined. In this section, you set a preprocessor option with Code Composer Studio. You also find and correct a syntax error.

1) Choose Project→Options.

2) In the Compiler tab of the Build Options window, select Preprocessor from the Category list. Type **`FILEIO`** in the Define Symbols box. Press the Tab key.

   Notice that the compiler command at the top of the window now includes the -d option. The code after the #ifdef FILEIO statement in the program is now included when you recompile the program. (The other options may vary depending on the DSP board you are using.)

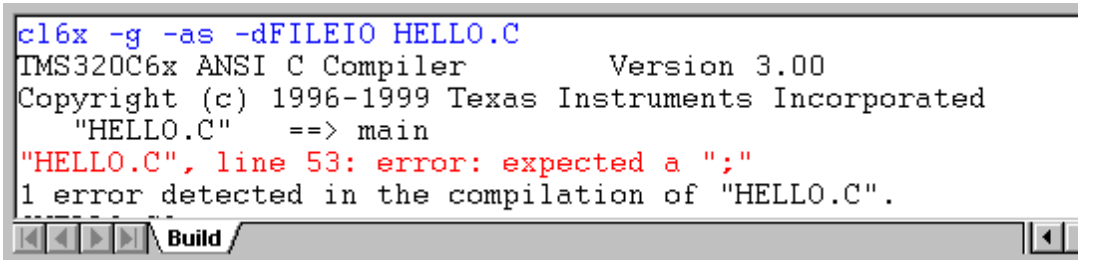


3) If you are programming for the TMS320C6701 and your program uses floating point values, go to the Target Version field and select 67xx from the pull-down list.

4) Click OK to save your new option settings.

5) Choose Project→Rebuild All or click the (Rebuild All) toolbar button. You need to rebuild all the files whenever the project options change.

6) A message says the program contains compile errors. Click Cancel. Scroll up in the Build tab area. You see a syntax error message.



7) Double-click on the red text that describes the location of the syntax error. Notice that the hello.c source file opens, and your cursor is on the following line:

```
fileStr[i] = 0;
```

8) Fix the syntax error in the line above the cursor location. (The semicolon is missing.) Notice that an asterisk (*) appears next to the filename in the Edit window's title bar, indicating that the source file has been modified. The asterisk disappears when the file is saved.

9) Choose File→Save or press Ctrl+S to save your changes to hello.c.

10) Choose Project→Build or click the (Incremental Build) toolbar button. Code Composer Studio rebuilds files that have been updated.

2.6 Using Breakpoints and the Watch Window

When you are developing and testing programs, you often need to check the value of a variable during program execution. In this section, you use breakpoints and the Watch Window to view such values. You also use the step commands after reaching the breakpoint.

1) Choose File→Reload Program.

2) Double-click on the hello.c file in the Project View. You may want to make the window larger so that you can see more of the source code at once.

3) Put your cursor in the line that says:

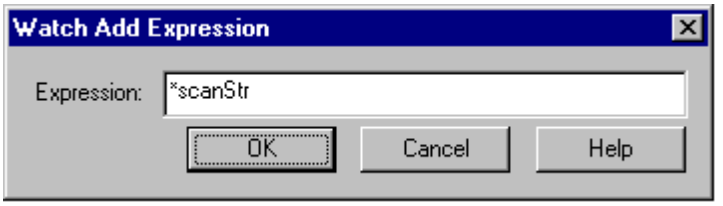
```
fprintf(fptr, "%s", scanStr);
```

4) Click the  (Toggle Breakpoint) toolbar button or press F9. The line is highlighted in magenta. (If you like, you can change this color using Option→Color.)

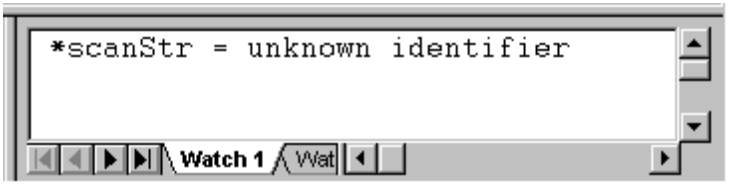
5) Choose View→Watch Window. A separate area in the lower-right corner of the Code Composer Studio window appears. At run time, this area shows the values of watched variables.

6) Right-click on the Watch Window area and choose Insert New Expression from the pop-up list.

7) Type **`*scanStr`** as the Expression and click OK.

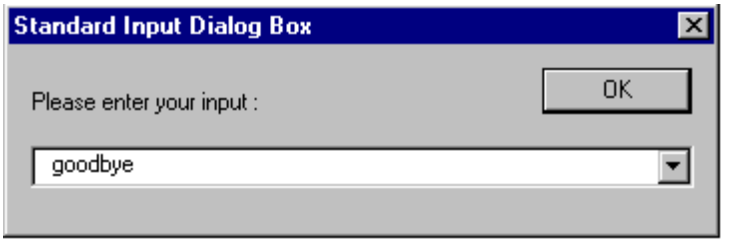


8) Notice that *scanStr is listed in the Watch Window but is undefined. This is because the program is currently not running in the main() function where this variable is declared locally.



9) Choose Debug→Run or press F5.

10) At the prompt, type **goodbye** and click OK. Notice that the Stdout tab shows the input text in blue.



Also notice that the Watch Window now shows the value of *scanStr.

After you type an input string, the program runs and stops at the breakpoint. The next line to be executed is highlighted in yellow.

11) Click the (Step Over) toolbar button or press F10 to step over the call to fprintf().

12) Experiment with the step commands Code Composer Studio provides:

- Step Into (F8)
- Step Over (F10)
- Step Out (Shift F7)
- Run to Cursor (Ctrl F10)

13) Click (Run) or press F5 to finish running the program when you have finished experimenting.

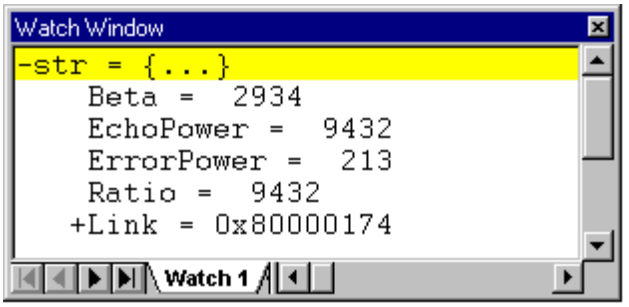
2.7 Using the Watch Window with Structures

In addition to watching the value of a simple variable, you can watch the values of the elements of a structure.

1) Right-click on the Watch Window area and choose Insert New Expression from the pop-up list.

2) Type **str** as the Expression and click OK. A line that says `+str = {...}` appears in the Watch Window. The + sign indicates that this is a structure. Recall from section 2.3, page 2-4 that a structure of type PARMS was declared globally and initialized in hello.c. The structure type is defined in hello.h.

3) Click once on the + sign. Code Composer Studio expands this line to list all the elements of the structure and their values. (The address shown for Link may vary.)



4) Double-click on any element in the structure to open the Edit Variable window for that element.

5) Change the value of the variable and click OK. Notice that the value changes in the Watch Window. The value also changes color to indicate that you have changed it manually.

6) Select the str variable in the Watch Window. Right-click in the Watch Window and choose Remove Current Expression from the pop-up list. Repeat this step for all expressions in the Watch Window.

7) Right-click on the Watch Window and choose Hide from the pop-up menu to hide the window.

8) Choose Debug→Breakpoints. In the Breakpoints tab, click Delete All and then click OK.

2.8 Profiling Code Execution Time

In this section, you use the profiling features of Code Composer Studio to gather statistics about the execution of the standard puts() function. In section 3.4, page 3-8, you compare these results to the results for using the DSP/BIOS API to display the hello world message.

1) Choose File→Reload Program.

2) Choose Profiler→Enable Clock. A check mark appears next to this item in the Profiler menu. This clock counts instruction cycles. It must be enabled for profile-points to count instruction cycles.

3) Double-click on the hello.c file in the Project View.

4) Choose View→Mixed Source/ASM. Assembly instructions are listed in gray following each C source code line.

5) Put your cursor in the line that says:

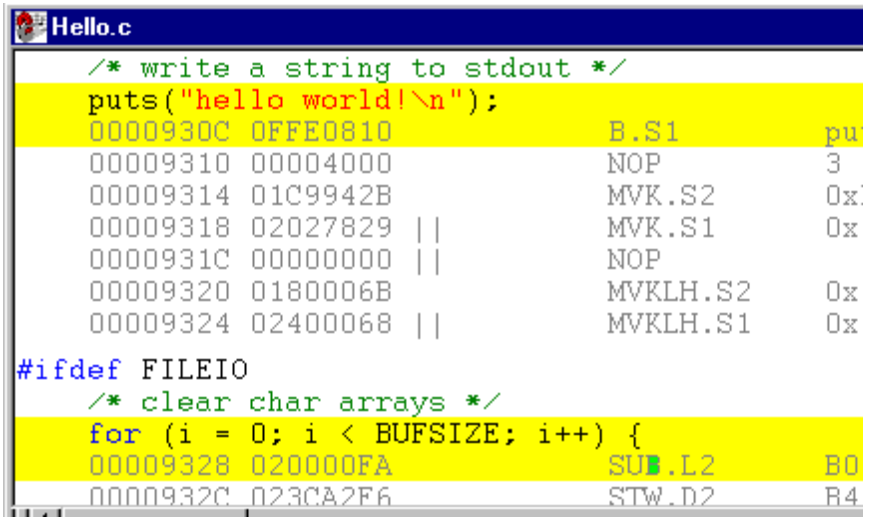
```
puts("hello world!\n");
```

6) Click the  (Toggle Profile-point) toolbar button. The C source code line and the first assembly instruction are highlighted in green.

7) Scroll down and put your cursor in the line that says:

```
for (i = 0; i < BUFSIZE; i++) {
```

8) Click the  (Toggle Profile-point) toolbar button. (Or, right-click on the code line and choose Toggle Profile Pt from the pop-up menu.)



Profile-points are handled before the profile-point line is executed. They report the number of instruction cycles since the previous profile-point or since the program started running. As a result, the statistics for the second profile-point report the number of cycles from when puts() started executing until it finished executing.

9) Choose Profiler→View Statistics. An area appears at the bottom of the window that displays statistics about the profile-points.

10) Make sure the line numbers are in ascending order. If they are in the reverse order, click the Location column heading once.

11) Resize this area by dragging its edges so you can see all the columns.

Location	Count	Average	Total	Maximum	Minimum
hello.c line 47	0	0.0	0	0	0
hello.c line 51	0	0.0	0	0	0

Note: Line Numbers May Vary

Line numbers displayed in this manual may vary from those displayed in the current release of the software.

12) Click the  (Run) toolbar button or press F5 to run the program. Type an input string in the prompt window.

13) Notice the number of cycles shown for the second profile-point. It should be about 1600 to 1700 cycles. (The actual numbers shown may vary.) This is the number of cycles required to execute the call to puts().

Location	Count	Average	Total	Maximum	Minimum
hello.c line 47	1	57280.0	57280	57280	57280
hello.c line 51	1	1625.0	1625	1625	1625

The average, total, maximum, and minimum are the same for these profile-points because these instructions are executed only one time.

Note: Target Halts at Profile-Points

Code Composer Studio temporarily halts the target whenever it reaches a profile-point. Therefore, the target application may not be able to meet real-time deadlines when you are using profile-points. (Real-time monitoring can be performed using RTDX. See section 1.5, page 1-12.)

14) Before continuing to the next chapter (after completing section 2.9, page 2-14), perform the following steps to free the resources used in your profiling session:

- Go to the Profiler menu and uncheck Enable Clock.
- Close the Profile Statistics window by right-clicking and choosing Hide from the pop-up menu.
- Go to Profiler→Profile-points. Select Delete All and click OK.
- Go to the View menu and uncheck Mixed Source/ASM.

2.9 Things to Try

To further explore Code Composer Studio, try the following:

- ❏ In the Build Options window, examine the fields on the Compiler, Assembler, and Linker tabs. Choose the various Categories to see all the options. Notice how changing the values in the field affects the command line shown. You can see the online help to learn about the various command line switches.
- ❏ Set some breakpoints. Choose Debug→Breakpoints. In the Breakpoint type box, notice that you can also set conditional breakpoints that break only if an expression is true. You can also set a variety of hardware breakpoints.

2.10 Learning More

To learn more about using Code Composer Studio, see the online help for Code Composer Studio or the *Code Composer Studio User's Guide* (which is provided as an Adobe Acrobat file).

Chapter 3

Developing a DSP/BIOS Program

This chapter introduces DSP/BIOS and shows how to create, build, debug, and test programs that use DSP/BIOS.

In this chapter, you optimize the hello world program you created in Chapter 2 by using DSP/BIOS.

This chapter requires a physical board and cannot be carried out using a software simulator. Also, this chapter requires the DSP/BIOS components of Code Composer Studio.

Topic **Page**

3.1 Creating a Configuration File

Another way to implement the hello world program is to use the LOG module provided with the DSP/BIOS API. You can use the DSP/BIOS API to provide basic run-time services within your embedded programs. The API modules are optimized for use on real-time DSPs. Unlike C library calls such as puts(), DSP/BIOS enables real-time analysis without halting your target hardware. Additionally, the API code consumes less space and runs faster than C standard I/O. A program can use one or more DSP/BIOS modules as desired.

In this chapter, you modify the files from Chapter 2 to use the DSP/BIOS API. (If you skipped Chapter 2, follow the steps in section 2.1, page 2-2 and section 2.2, page 2-3.)

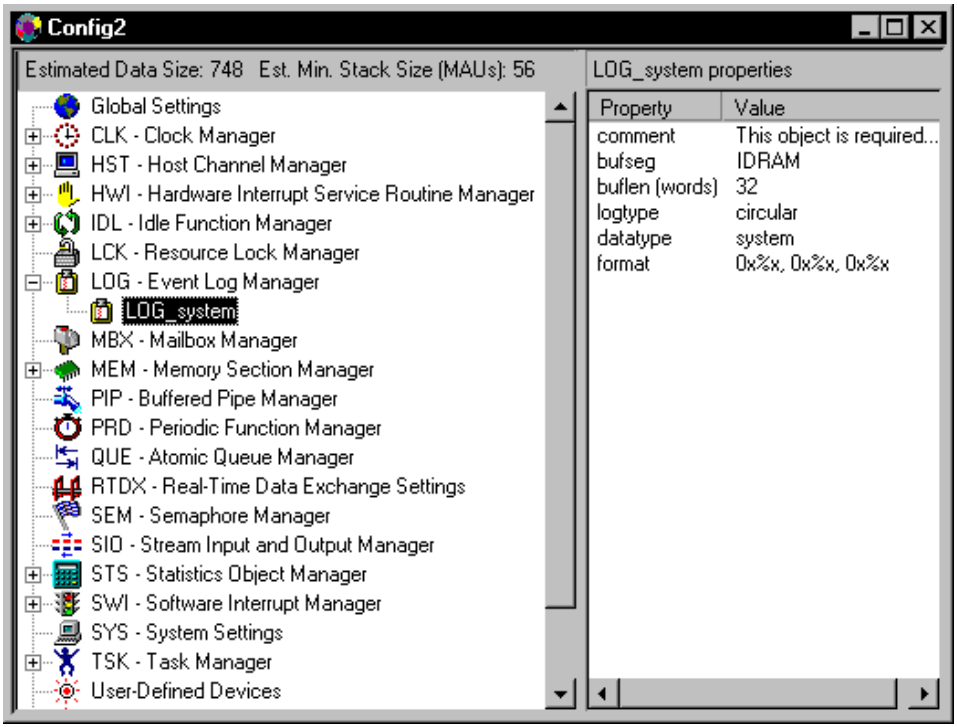
In order to use the DSP/BIOS API, a program must have a configuration file that defines the DSP/BIOS objects used by the program. In this section, you create a configuration file.

1) If you have closed Code Composer Studio, restart it and use Project→Open to reopen the myhello.mak project in the c:\ti\myprojects\hello1 folder. (If you installed elsewhere, open the folder within the myprojects folder in the location where you installed.)

2) Choose File→New→DSP/BIOS Config.

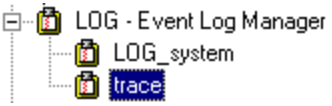
3) Select the template for your DSP board and click OK. (The *TMS320C6000 DSP/BIOS User's Guide* explains how to create a custom template.)

You see a window like the following. You can expand and contract the list by clicking the + and - symbols on the left. The right side of the window shows properties of the object you select in the left side of the window.



4) Right-click on the LOG - Event Log Manager and choose Insert LOG from the pop-up menu. This creates a LOG object called LOG0.

5) Right-click on the name of the LOG0 object and choose Rename from the pop-up menu. Change the object's name to trace.



6) Choose File→Save. Move to your working folder (usually c:\ti\myprojects\hello1) and save this configuration file with the name myhello.cdb. Saving this configuration actually creates the following files:

- **myhello.cdb**. Stores configuration settings
- **myhellocfg.cmd**. Linker command file
- **myhellocfg.s62**. Assembly language source file
- **myhellocfg.h62**. Assembly language header file included by myhellocfg.s62

Although these files have extensions of .s62 and .h62, they can also be used with the TMS320C6701. DSP/BIOS does not need to use the floating-point instructions supported by the TMS320C6701, therefore only one version of the software is required to support both DSPs. If you are using the TMS320C6701 with DSP/BIOS, open the Global Settings property page in the configuration and change the DSP Type property. This controls the libraries with which the program is linked.

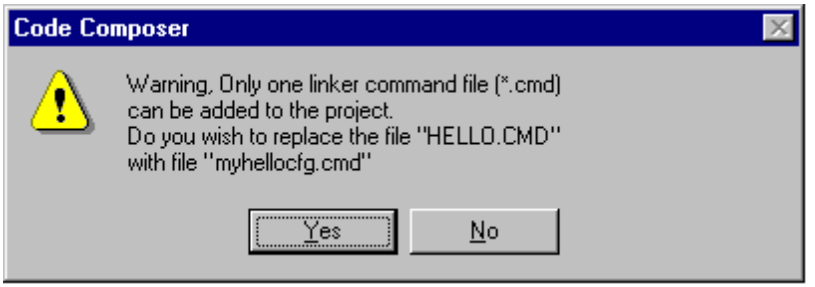
3.2 Adding DSP/BIOS Files to a Project

Recall that the configuration file you made in the previous section actually resulted in the creation of four new files: myhello.cdb, myhellocfg.cmd, myhellocfg.s62, and myhellocfg.h62. In this section, you add these files to your project and remove the files which they replace.

1) Choose Project→Add Files to Project. Select Configuration File (*.cdb) in the Files of type box. Select the myhello.cdb file and click Open. Notice that the Project View now contains myhello.cdb in a folder called DSP/BIOS Config. In addition, the myhellocfg.s62 file is now listed as a source file. Remember that Code Composer Studio automatically adds include files (in this case, the myhellocfg.h62 file) to the project when it scans for dependencies during a project build.

2) The output filename must match the .cdb filename (myhello.out and myhello.cdb). Choose Project→Options and go to the Linker tab. In the Output Filename field, verify that myhello.out is the filename and click OK.

3) Choose Project→Add Files to Project again. Select Linker Command File (*.cmd) in the Files of type box. Select the myhellocfg.cmd file and click Open. This causes Code Composer Studio to display the following warning:



4) Click Yes. This replaces the previous command file (HELLO.CMD) with the new one that was generated when you saved the configuration.

5) In the Project View area, right-click on the vectors.asm source file and choose Remove from project in the pop-up menu. The hardware interrupt vectors are automatically defined by the DSP/BIOS configuration file.

6) Right-click on the RTS6201.lib library file and remove it from the project. This library is automatically included by the myhellocfg.cmd file.

7) Double-click on the hello.c program to open it for editing. If the assembly instructions are shown, choose View→Mixed Source/ASM to hide the assembly code.

8) Change the source file's contents to the following. (You can copy and paste this code from c:\ti\c6000\tutorial\hello2\hello.c if you like.) Make sure you replace the existing main function (which has the puts() function) with the main shown below, because puts() and LOG_printf use the same resources.

```
/* ======== hello.c ======== */

/* DSP/BIOS header files*/
#include <std.h>
#include <log.h>

/* Objects created by the Configuration Tool */
extern far LOG_Obj trace;

/*  ======== main ======== */
Void main()
{
    LOG_printf(&trace, "hello world!");

    /* fall into DSP/BIOS idle loop */
    return;
}
```

9) Notice the following parts of this code:

   a) The code includes the std.h and log.h header files. All programs that use the DSP/BIOS API must include the std.h file and header files for any modules the program uses. The log.h header file defines the LOG_Obj structure and declares the API operations in the LOG module. The std.h file must be included first. The order of the remaining modules you include is not important.

   b) The code then declares the LOG object you created in the configuration file.

   c) Within the main function, this example calls LOG_printf and passes it the address of the LOG object (&trace) and the hello world message.

   d) Finally main returns, which causes the program to enter the DSP/BIOS idle loop. Within this loop, DSP/BIOS waits for software

interrupts and hardware interrupts to occur. Chapter 5 through Chapter 7 explain these types of events.

10) Choose File→Save or press Ctrl+S to save your changes to hello.c.

11) Choose Project→Options. Choose the Preprocessor category. Remove FILEIO from the Define Symbols box in the Compiler tab. Then click OK.

12) Choose Project→Rebuild All or click the  (Rebuild All) toolbar button.

3.3 Testing with Code Composer Studio

Now you can test the program. Since the program writes only one line to a LOG, there is not much to analyze. Chapter 5 through Chapter 7 show more ways to analyze program behavior.

1) Choose File→Load Program. Select the program you just rebuilt, myhello.out, and click Open.

2) Choose Debug→Go Main. A window shows the hello.c file with the first line of the main function highlighted. The highlighting indicates that program execution is paused at this location.

3) Choose Tools→DSP/BIOS→Message Log. A Message Log area appears at the bottom of the Code Composer Studio window.

4) Right-click on the Message Log area and choose Property Page from the pop-up window.

5) Select trace as the name of the log to monitor and click OK. The default refresh rate is once per second. (To change refresh rates, choose Tools→DSP/BIOS→RTA Control Panel. Right-click on the RTA Control Panel area and choose Property Page. Choose a new refresh rate and click OK.)

6) Choose Debug→Run or press F5.

   The hello world message appears in the Message Log area.

```
0   hello world!
```

7) Choose Debug→Halt or press Shift F5 to stop the program. After the main function returns, your program is in the DSP/BIOS idle loop, waiting for an event to occur. See section 3.5, page 3-10 to learn more about the idle loop.

8) Close the Message Log by right-clicking and selecting Close. This is necessary because you will use the Profiler in the next section.

9) Choose Tools→RTDX to open the RTDX control window. Select RTDX Disable from the pull-down list, then right-click on the RTDX area and select Hide.

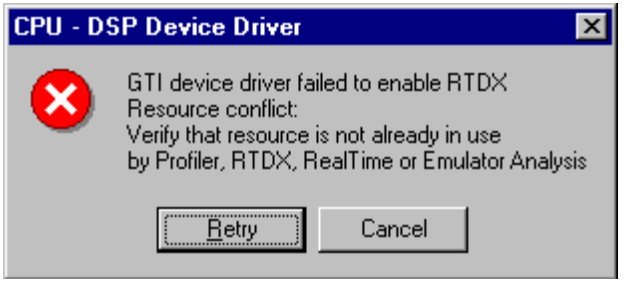
Note: Profiling and RTDX Cannot Be Used Together on Some Targets

The DSP/BIOS plug-ins use RTDX for host/target communication. On some DSP targets (for example, the TMS320C6201) you cannot use both profiling and RTDX at the same time. Close all tools that uses RTDX, such as the Message Log and other DSP/BIOS plug-ins, before using profiling.

To ensure that RTDX is disabled, especially after using DSP/BIOS plug-ins, choose Tools→RTDX to open the RTDX plug-in. Select RTDX Disable from the pull-down list, then right-click and select Hide.

Conversely, after using profiling, free the profiler resources before using RTDX, as described in section 2.8, page 2–14.

An error message similar to the following is shown if you attempt to use RTDX and profiling at the same time:



3.4 Profiling DSP/BIOS Code Execution Time

Earlier, you used the profiling features of Code Composer Studio to find the number of cycles required to call puts(). Now, you can do the same for the call to LOG_printf.

1) Choose File→Reload Program.
2) Choose Profiler→Enable Clock. Make sure you see a check mark next to this item in the Profiler menu.
3) Double-click on the hello.c file in the Project View.
4) Choose View→Mixed Source/ASM. Assembly instructions are listed in gray following each C source code line.
5) Put your cursor in the line that says:

   ```
   LOG_printf(&trace, "hello world!");
   ```

6) Click the  (Toggle Profile-point) toolbar button. This line and the assembly language instruction that follows it are highlighted in green.
7) Scroll down and put your cursor in the line for the final curly brace of the program, and click the  (Toggle Profile-point) toolbar button.

   You might think that you should set the second profile-point on the line that says `return;` since that is the last statement in the program. However, notice that there are no assembly language equivalents shown until after the curly brace. If you set the profile-point at the line that says `return;`, Code Composer Studio automatically corrects the problem at run time.
8) Choose Profiler→View Statistics.
9) Make sure the line numbers are in ascending order. If they are in the reverse order, click the Location column heading once.
10) Click the  (Run) toolbar button or press F5 to run the program.
11) Notice the number of instruction cycles shown for the second profile-point. It should be about 36. (The actual numbers shown may vary.) This is the number of cycles required to execute the call to LOG_printf.

Location	Count	Average	Total	Maximum	Minimum
HELLO.C line 29	1	6725.0	6725	6725	6725
HELLO.C line 33	1	36.0	36	36	36

Calls to LOG_printf are efficient because the string formatting is performed on the host PC rather than on the target DSP. LOG_printf takes about 36 instruction cycles compared to about 1700 for puts() measured at the end of Chapter 2. You can leave calls to LOG_printf in your code for system status monitoring with very little impact on code execution.

12) Click  (Halt) or press Shift F5 to stop the program.

13) Before proceeding to the next chapter (after completing section 3.5, page 3-10), perform the following steps to free the resources used in your profiling session:

- In the Profiler menu, uncheck Enable Clock.
- Close the Profile Statistics window by right-clicking and choosing Hide from the pop-up menu.
- Choose Profiler→Profile-points. Select Delete All and click OK.
- In the View menu, uncheck Mixed Source/ASM.
- Close all source and configuration windows.
- Choose Project→Close to close the project.

3.5 Things to Try

To explore Code Composer Studio, try the following:

- ❏ Load myhello.out and put a breakpoint on the line that calls LOG_printf. Use Debug→Breakpoints to add a breakpoint at IDL_F_loop. (Type IDL_F_loop in the Location box and click Add.)

  Run the program. At the first breakpoint, use View→CPU Registers→CPU Register to see a list of register values. Notice that GIE is 0, indicating that interrupts are disabled while the main function is executing.

  Run to the next breakpoint. Notice that GIE is now 1, indicating that interrupts are now enabled. Notice that if you run the program, you hit this breakpoint over and over.

  After the startup process and main are completed, a DSP/BIOS application drops into a background thread called the idle loop. This loop is managed by the IDL module and continues until you halt the program. The idle loop runs with interrupts enabled and can be preempted at any point by any ISR, software interrupt, or task triggered to handle the application's real-time processing. Chapter 5 through Chapter 7 explain more about using ISRs and software interrupts with DSP/BIOS

- ❏ In an MS-DOS window, run the sectti.exe utility by typing the following command lines. Change the directory locations if you installed Code Composer Studio in a location other than c:\ti.

  ```
  cd c:\ti\c6000\tutorial\hello1
  sectti hello.out > hello1.prn
  cd ..\hello2
  sectti hello.out > hello2.prn
  ```

  Compare the hello1.prn and hello2.prn files to see differences in memory sections and sizes when using stdio.h calls and DSP/BIOS. Notice the size of the .text section is smaller for the DSP/BIOS call to LOG_printf, as compared to linking the stdio when using puts(). See the *TMS320C6000 DSP/BIOS API Reference Guide* for more information on the sectti utility.

3.6 Learning More

To learn more about Code Composer Studio and DSP/BIOS, see the online help for Code Composer Studio. In addition, see the *Code Composer Studio User's Guide,* the *TMS320C6000 DSP/BIOS User's Guide*, and the *TMS320C6000 DSP/BIOS API Reference Guide* (which are provided as Adobe Acrobat files).

Chapter 4

Testing Algorithms and Data from a File

This chapter shows the process for creating and testing a simple algorithm and introduces additional Code Composer Studio features.

In this chapter, you create a program that performs basic signal processing. You expand on this example in the following two chapters.

You create and test a simple algorithm using data stored in a file on your PC. You also use Probe Points, graphs, animation, and GEL files with Code Composer Studio.

Topic **Page**

4.1 Opening and Examining the Project

You begin by opening a project with Code Composer Studio and examining the source code files and libraries used in that project.

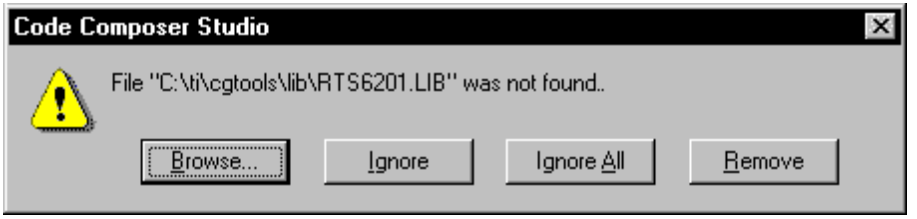
1) If you installed Code Composer Studio in c:\ti, create a folder called volume1 in the c:\ti\myprojects folder. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)

2) Copy all files from the c:\ti\c6000\tutorial\volume1 folder to this new folder.

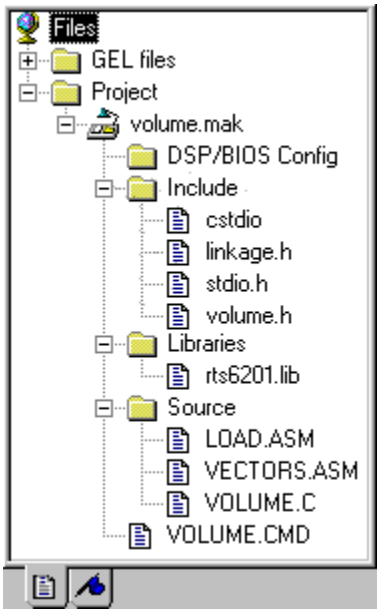
3) If Code Composer Studio is not already running, from the Windows Start menu, choose Programs→Code Composer Studio 'C6000→CCStudio.

4) Choose Project→Open. Select the volume.mak file in the folder you created and click Open.

5) Code Composer Studio displays a dialog box indicating the library file was not found. This is because the project was moved. To locate this file, click the Browse button, navigate to c:\ti\c6000\cgtools\lib, and select rts6201.lib. (If you installed somewhere other than c:\ti, navigate to the \c6000\cgtools\lib folder within the folder where you installed.)



6) Expand the Project View by clicking the + signs next to Project, VOLUME.MAK, Include, Libraries, and Source.



The files used in this project are:

- **volume.c**. This is the source code for the main program. You examine the source code in the next section.
- **volume.h**. This is a header file included by volume.c to define various constants and structures.
- **load.asm**. This file contains the load routine, a simple assembly loop routine that is callable from C with one argument. It consumes about 1000**argument* instruction cycles.
- **vectors.asm**. This is the same file used in Chapter 2 to define a reset entry point in the DSP's interrupt vector table.
- **volume.cmd**. This linker command file maps sections to memory.
- **rts6201.lib**. This library provides run-time support for the target DSP.

4.2 Reviewing the Source Code

Double-click on the volume.c file in the Project View to see the source code in the right half of the Code Composer Studio window.

Notice the following parts of this example:

- ❏ After the main function prints a message, it enters an infinite loop. Within this loop, it calls the dataIO and processing functions.
- ❏ The processing function multiplies each value in the input buffer by the gain and puts the resulting values into the output buffer. It also calls the assembly load routine, which consumes instruction cycles based on the processingLoad value passed to the routine.
- ❏ The dataIO function in this example does not perform any actions other than to return. Rather than using C code to perform I/O, you can use a Probe Point within Code Composer Studio to read data from a file on the host into the inp_buffer location.

```
#include <stdio.h>

#include "volume.h"

/* Global declarations */
int inp_buffer[BUFSIZE];          /* processing data buffers */
int out_buffer[BUFSIZE];

int gain = MINGAIN;               /* volume control variable */
unsigned int processingLoad = BASELOAD;   /* processing load */

/* Functions */
extern void load(unsigned int loadValue);

static int processing(int *input, int *output);
static void dataIO(void);

/* ======== main ======== */
void main()
{
    int *input = &inp_buffer[0];
    int *output = &out_buffer[0];

    puts("volume example started\n");
```

```
    /* loop forever */
    while(TRUE)
    {
       /* Read using a Probe Point connected to a host file. */
       dataIO();

       /* apply gain */
       processing(input, output);
    }
}

/*  ======== processing ======== *
 * FUNCTION: apply signal processing transform to input signal.
 * PARAMETERS: address of input and output buffers.
 * RETURN VALUE: TRUE. */
static int processing(int *input, int *output)
{
    int size = BUFSIZE;

    while(size--){
        *output++ = *input++ * gain;
    }

    /* additional processing load */
    load(processingLoad);

    return(TRUE);
}

/*  ======== dataIO ======== *
 * FUNCTION: read input signal and write output signal.
 * PARAMETERS: none.
 * RETURN VALUE: none. */
static void dataIO()
{
    /* do data I/O */
    return;
}
```

4.3 Adding a Probe Point for File I/O

In this section, you add a Probe Point, which reads data from a file on your PC. Probe Points are a useful tool for algorithm development. You can use them in the following ways:

- ❏ To transfer input data from a file on the host PC to a buffer on the target for use by the algorithm
- ❏ To transfer output data from a buffer on the target to a file on the host PC for analysis
- ❏ To update a window, such as a graph, with data

Probe Points are similar to breakpoints in that they both halt the target to perform their action. However, Probe Points differ from breakpoints in the following ways:

- ❏ Probe Points halt the target momentarily, perform a single action, and resume target execution.
- ❏ Breakpoints halt the CPU until execution is manually resumed and cause all open windows to be updated.
- ❏ Probe Points permit automatic file input or output to be performed; breakpoints do not.

This chapter shows how to use a Probe Point to transfer the contents of a PC file to the target for use as test data. It also uses a breakpoint to update all the open windows when the Probe Point is reached. These windows include graphs of the input and output data. Chapter 7 shows two other ways to manage input and output streams.

1) Choose Project→Rebuild All or click the  (Rebuild All) toolbar button.
2) Choose File→Load Program. Select the program you just rebuilt, volume.out, and click Open.
3) Double-click on the volume.c file in the Project View.
4) Put your cursor in the line of the main function that says:

   ```
   dataIO();
   ```

   The dataIO function acts as a placeholder. You add to it later. For now, it is a convenient place to connect a Probe Point that injects data from a PC file.
5) Click the  (Toggle Probe Point) toolbar button. The line is highlighted in blue.

6) Choose File→File I/O. The File I/O dialog appears so that you can select input and output files.

7) In the File Input tab, click Add File.

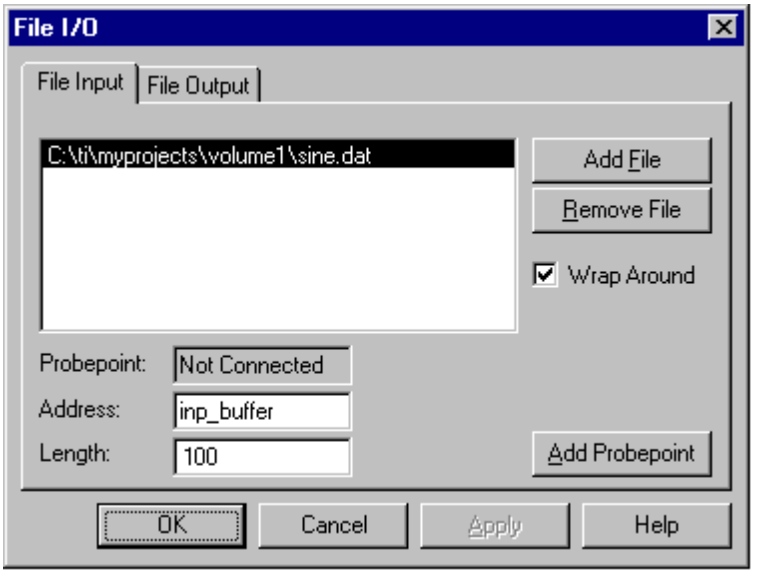
8) Choose the sine.dat file.

   Notice that you can select the format of the data in the Files of Type box. The sine.dat file contains hex values for a sine waveform.

9) Click Open to add this file to the list in the File I/O dialog.

   A control window for the sine.dat file appears. (It may be covered by the Code Composer Studio window.) Later, when you run the program, you can use this window to start, stop, rewind, or fast forward within the data file.

10) In the File I/O dialog, change the Address to inp_buffer and the Length to 100. Also, put a check mark in the Wrap Around box.



- The Address field specifies where the data from the file is to be placed. The inp_buffer is declared in volume.c as an integer array of BUFSIZE.

- The Length field specifies how many samples from the data file are read each time the Probe Point is reached. You use 100 because that is the value set for the BUFSIZE constant in volume.h (0x64).
- The Wrap Around option causes Code Composer Studio to start reading from the beginning of the file when it reaches the end of the file. This allows the data file to be treated as a continuous stream of data even though it contains only 1000 values and 100 values are read each time the Probe Point is reached.

11) Click Add Probepoint. The Probe Points tab of the Break/Probe/Profile Points dialog appears.

12) In the Probe Point list, highlight the line that says VOLUME.C line 53 --> No Connection.

13) In the Connect To field, click the down arrow and select the sine.dat file from the list.

14) Click Replace. The Probe Point list changes to show that this Probe Point is connected to the sine.dat file.

15) Click OK. The File I/O dialog shows that the file is now connected to a Probe Point.

16) Click OK in the File I/O dialog.

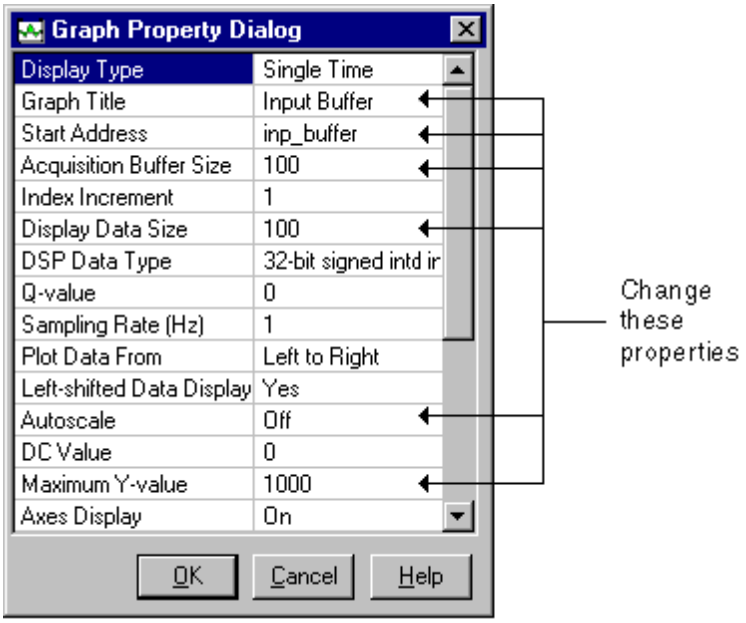
4.4 Displaying Graphs

If you ran the program now, you would not see much information about what the program was doing. You could set watch variables on addresses within the inp_buffer and out_buffer arrays, but you would need to watch a lot of variables and the display would be numeric rather than visual.

Code Composer Studio provides a variety of ways to graph data processed by your program. In this example, you view a signal plotted against time. You open the graphs in this section and run the program in the next section.

1) Choose View→Graph→Time/Frequency.

2) In the Graph Property Dialog, change the Graph Title, Start Address, Acquisition Buffer Size, Display Data Size, Autoscale, and Maximum Y-value properties to the values shown here. Scroll down or resize the dialog box to see all the properties.



3) Click OK. A graph window for the Input Buffer appears.

4) Right-click on the Input Buffer window and choose Clear Display from the pop-up menu.

5) Choose View→Graph→Time/Frequency again.

6) This time, change the Graph Title to Output Buffer and the Start Address to out_buffer. All the other settings are correct.

7) Click OK to display the graph window for the Output Buffer. Right-click on the graph window and choose Clear Display from the pop-up menu.

4.5 Animating the Program and Graphs

So far, you have placed a Probe Point, which temporarily halts the target, transfers data from the host PC to the target, and resumes execution of the target application. However, the Probe Point does not cause the graphs to be updated. In this section, you create a breakpoint that causes the graphs to be updated and use the Animate command to resume execution automatically after the breakpoint is reached.

1) In the Volume.c window, put your cursor in the line that calls dataIO.

2) Click the (Toggle Breakpoint) toolbar button or press F9. The line is highlighted in both magenta and blue (unless you changed either color using Option→Color) to indicate that both a breakpoint and a Probe Point are set on this line.

   You put the breakpoint on the same line as the Probe Point so that the target is halted only once to perform both operations—transferring the data and updating the graphs.

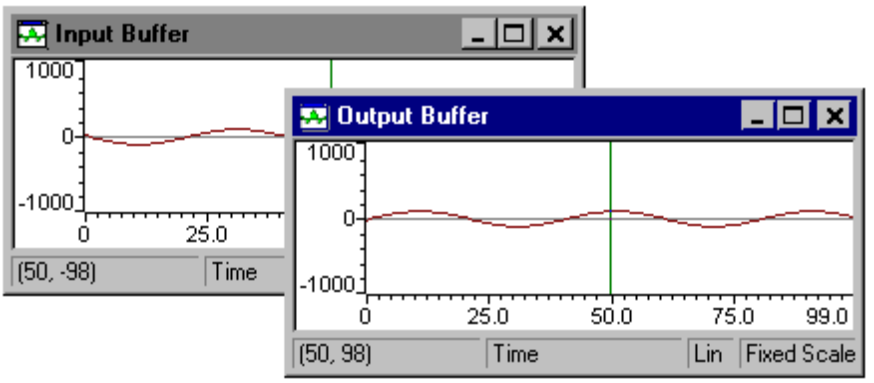
3) Arrange the windows so that you can see both graphs.

4) Click the (Animate) toolbar button or press F12 to run the program.

   The Animate command is similar to the Run command. It causes the target application to run until it reaches a breakpoint. The target is then halted and the windows are updated. However, unlike the Run command, the Animate command then resumes execution until it reaches another breakpoint. This process continues until the target is manually halted. Think of the Animate command as a run-break-continue process.

5) Notice that each graph contains 2.5 sine waves and the signs are reversed in these graphs. Each time the Probe Point is reached, Code Composer Studio gets 100 values from the sine.dat file and writes them to the inp_buffer address. The signs are reversed because the input

buffer contains the values just read from sine.dat, while the output buffer contains the last set of values processed by the processing function.



Note: Target Halts at Probe Points

Code Composer Studio briefly halts the target whenever it reaches a Probe Point. Therefore, the target application may not meet real-time deadlines if you are using Probe Points. At this stage of development, you are testing the algorithm. Later, you analyze real-time behavior using RTDX and DSP/BIOS.

The graphs can also be updated using only Probe Points and the Run command. See section 4.10, page 4-18 at the end of this chapter to try animating the graphs using Probe Points only.

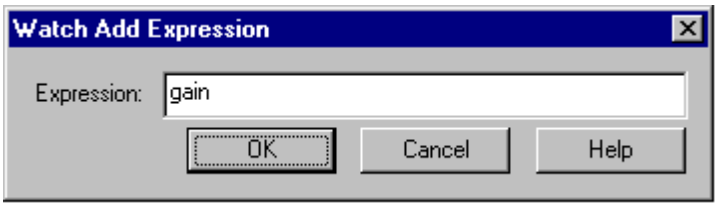
4.6 Adjusting the Gain

Recall from section 4.2, page 4-4 that the processing function multiplies each value in the input buffer by the gain and puts the resulting values into the output buffer. It does this by performing the following statement within a while loop:

```
*output++ = *input++ * gain;
```

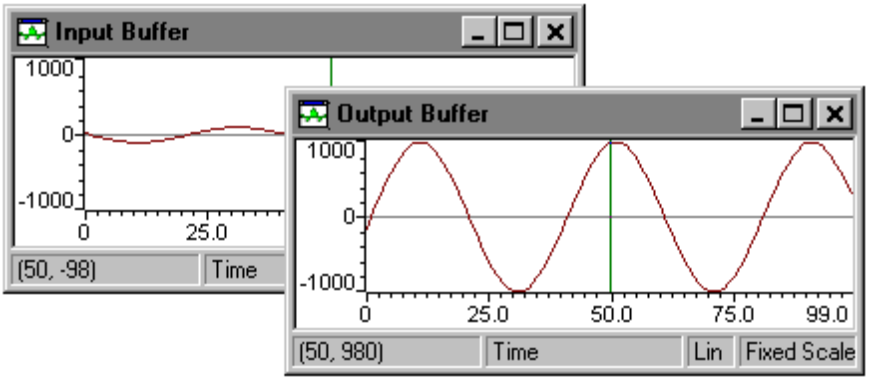
This statement multiplies a value in inp_buffer by the gain and places it in the corresponding location in the out_buffer. The gain is initially set to MINGAIN, which is defined as 1 in volume.h. To modify the output, you need to change gain. One way to do this is to use a watch variable.

1) Choose View→Watch Window.
2) Right-click on the Watch window area and choose Insert New Expression from the pop-up list.
3) Type **`gain`** as the Expression and click OK.



The value of this variable appears in the Watch window area.

4) If you have halted the program, click the (Animate) toolbar button to restart the program.
5) Double-click on gain in the Watch window area.
6) In the Edit Variable window, change the gain to 10 and click OK.
7) Notice that the amplitude of the signal in the Output Buffer graph changes to reflect the increased gain.



4.7 Viewing Out-of-Scope Variables

You have used the Watch Window to view and change the values of variables. Sometimes you want to examine variables when they are not currently in scope at the current breakpoint. You can do this by using the call stack.

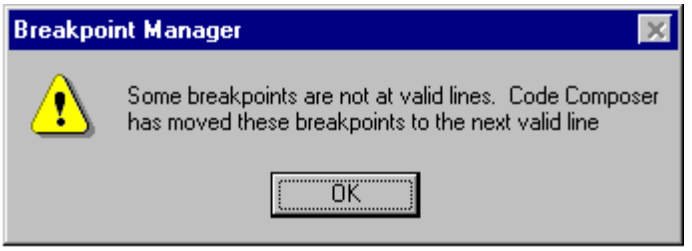
1) Click (Halt) or press Shift F5 to stop the program.

2) Review the volume.c program within Code Composer Studio (or by looking at section 4.2, page 4-4). Notice that *input is defined in both the main and processing functions. However, it is not defined within the dataIO function.

3) In the Volume.c window, put your cursor on the line that says `return;` within the dataIO function.

4) Click the (Toggle Breakpoint) toolbar button or press F9. The line is highlighted in magenta to indicate that a breakpoint is set (unless you changed the color using Option→Color).

5) Press F5 to run the program. Code Composer Studio automatically moves the breakpoint to the next line within that function that corresponds to an assembly instruction. It displays a message telling you that it has moved the breakpoint.

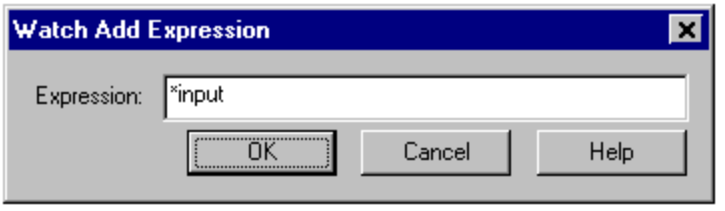


6) Click OK.

7) Press F5 until the program stops at the breakpoint at the end of the dataIO function (instead of at the breakpoint on the call to the dataIO function).

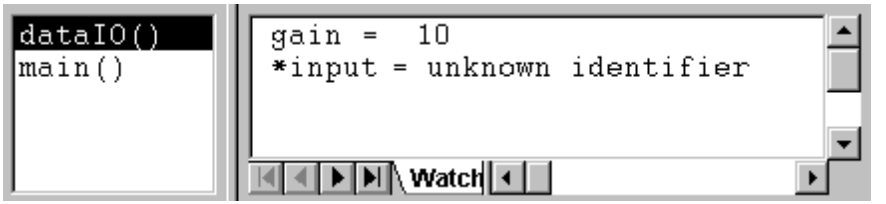
8) Right-click on the Watch window area and choose Insert New Expression from the pop-up list.

9) Type **`*input`** as the Expression and click OK.

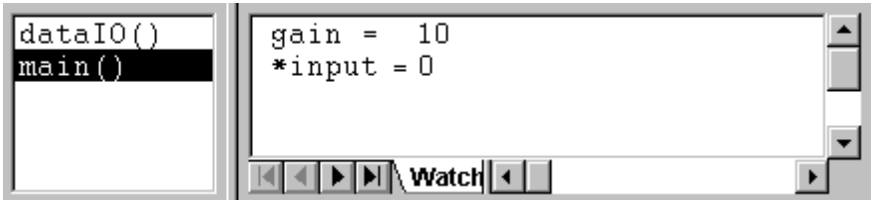


10) Notice that the Watch window area says this variable is an unknown identifier. This shows that *input is not defined within the scope of the dataIO function.

11) Choose View→Call Stack. You see the call stack area next to the Watch window area.



12) Click on main() in the call stack area to see the value of *input within the scope of the main function. (The value should be 0, but may differ if you have modified the sine.dat file.)

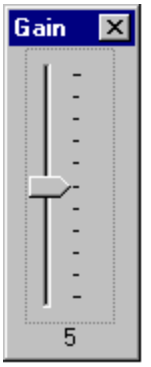


13) Right-click on the call stack area and choose Hide from the pop-up menu.

14) Remove the breakpoint that you added in step 4 by putting he cursor on the line after `return;` in dataIO() and clicking the (Toggle Breakpoint) toolbar button or pressing F9.

4.8 Using a GEL File

Code Composer Studio provides another way of modifying a variable. This method uses GEL, an extension language, to create small windows that allow you to modify variables.



1) Choose File→Load GEL. In the Load GEL File dialog box, select the volume.gel file and click Open.
2) Choose GEL→Application Control→Gain. This item was added to your menus when you loaded the GEL file.
3) If you have halted the program, click the (Animate) toolbar button. Notice that even though the gain is at zero your previous gain is still the same. The Gain slider does not send a value until it is changed.
4) In the Gain window, use the slider to change the gain. The amplitude changes in the Output Buffer window. In addition, the value of the gain variable in the Watch window area changes whenever you move the slider.
5) Click (Halt) or press Shift F5 to stop the program.
6) To see how the Gain GEL function works, click the + sign next to GEL Files in the Project View. Then, double-click on the VOLUME.GEL file to see its contents:

```
menuitem "Application Control"
dialog Load(loadParm "Load")
{
    processingLoad = loadParm;
}
slider Gain(0, 10 ,1, 1, gainParm)
{
    gain = gainParm;
}
```

The Gain function defines a slider with a minimum value of 0, a maximum value of 10, and an increment and page up/down value of 1. When you move the slider, the gain variable is changed to the new value of the slider (gainParm).

4.9 Adjusting and Profiling the Processing Load

In Chapter 2, you used profile-points to measure the number of cycles required to call puts(). Now, you use profile-points to see the effect of changing the processingLoad variable, which is passed to the assembly load routine. The processingLoad is initially set to BASELOAD, which is defined as 1 in volume.h.

1) Choose Profiler→Enable Clock. Make sure you see a check mark next to this item in the Profiler menu. (If you see a resource conflict message, close any DSP/BIOS plug-in windows. Then choose Tools→RTDX and select RTDX Disable from the pull-down list.)

2) Double-click on the volume.c file in the Project View.

3) Put your cursor in the line that says:

```
load(processingLoad);
```

4) Click the  (Toggle Profile-point) toolbar button or right-click and select Toggle Profile Pt.

5) Put your cursor in the line that says:

```
return(TRUE);
```

6) Click the  (Toggle Profile-point) toolbar button.

7) Choose Profiler→View Statistics. The locations shown identify the line number where you added the profile-points. You may want to resize the Statistics area so that you can see the Maximum column. Or, you can right-click on the Statistics area and deselect Allow Docking to display the statistics in a separate window.

8) If the locations are not listed in ascending order by line number, click the Location column header.

9) Click the  (Animate) toolbar button or press F12.

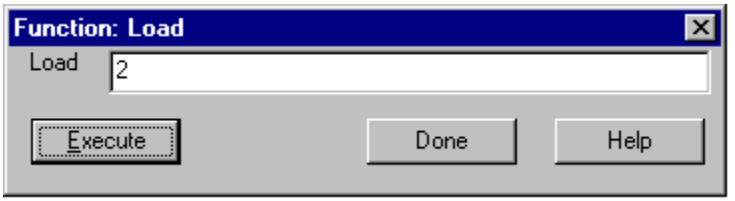
10) Notice the maximum number of cycles shown for the second profile-point. It should be about 1018 cycles. (The actual numbers shown may vary, especially if you are using a non-generic development board.) This is the number of cycles required to execute the load routine when the processingLoad is 1.

Profile Statistics

Location	Count	Average	Total	Maximum	Minimum
VOLUME.C line 78	133	2343.2	311652	2354	2343
VOLUME.C line 80	133	1018.0	135804	1018	1018

11) Choose GEL→Application Control→Load.

12) Type **2** as the new load and click Execute. The maximum number of cycles for the second profile-point changes to 2018. The number of cycles increases by about 1000 when you increment the processingLoad by 1. These instruction cycles are performed within the load function, which is stored in load.asm.



13) Right-click on the Profile Statistics area and choose Clear All from the pop-up menu. This resets the statistics to 0. The average, maximum, and minimum are equal to the number of instruction cycles for the current processingLoad.

14) Click (Halt) or press Shift F5 to stop the program.

15) Before continuing to Chapter 5 (after completing section 4.10, page 4-18), perform the following steps to free the resources used in your profiling session:

- Close the Load and Gain controls, the sine.dat control window, and the Time/Frequency graphs.
- Choose File→File I/O and click Remove File to delete the sine.dat file.
- In the Profiler menu, uncheck Enable Clock.
- Close the Profile Statistics window by right-clicking and choosing Hide from the pop-up menu.
- Choose Debug→Breakpoints. Select Delete All and click OK.
- Choose Debug→Probe Points. Select Delete All and click OK.
- Choose Profiler→Profile-points. Select Delete All and click OK.
- Close open windows and the project (Project→Close).
- Right-click on volume.gel in the Project View and select Remove.
- Delete all expressions from the Watch Window and hide the Watch Window.

4.10 Things to Try

To explore using Code Composer Studio, try the following:

- ❏ Add processingLoad to the Watch window. When you use the Load GEL control, the processingLoad value is updated in the Watch window.
- ❏ Right-click on the Watch window and choose Insert New Expression. Click the Help button and read about the display formats you can use. Experiment with various display formats. For example, you can type **`*input,x`** as the expression to view the sine input in hexadecimal format.
- ❏ Change BUFSIZE in volume.h to 0x78 (or 120) and rebuild, then reload the program. Change the Length in the File I/O dialog to 0x78. For both graphs, change the Acquisition Buffer Size and the Display Data Size to 0x78. This causes a buffer to contain 3 full sine waves rather than 2.5 waves. Animate the program and notice that the input and output buffer graphs are now in phase. (You may need to halt the program to see whether the graphs are in phase.)
- ❏ Instead of using profile-points to gather statistics, try using the clock as an alternative. Replace the profile-points with breakpoints. Choose Profiler→View Clock. Run the program to the first breakpoint. Double-click on the clock area to clear it. Run the program again. The clock shows the number of cycles it takes to reach the second breakpoint.
- ❏ Experiment with Probe Points by repeating section 4.3, page 4-6 through section 4.5, page 4-10. This time, use only Probe Points and the Run command. Note that you need to create three Probe Points. This is because a Probe Point can be associated with only one action. There are two graphs to be updated and one file to use as input. Each of these actions requires its own Probe Point.

  Also notice that each Probe Point must go on a different line of code. As a result, the target is halted three times as often and actions are not performed at the same point in target execution. For these reasons, the combination of a Probe Point and a breakpoint used in this lesson is more efficient than using Probe Points only.
- ❏ To practice building projects with Code Composer Studio, copy all the files in the c:\ti\c6000\tutorial\volume1 folder to a new folder. Delete the volume.mak file. Then, use Code Composer Studio to recreate the project using the Project→New and Project→Add Files to Project menu items. See section 4.1, page 4-2 for a list of the files to add to the project.

4.11 Learning More

To learn more about Probe Points, graphs, animation, and GEL files, see the online help for Code Composer Studio or the *Code Composer Studio User's Guide* (which is provided as an Adobe Acrobat file).

Chapter 5

Debugging Program Behavior

This chapter introduces techniques for debugging a program and several DSP/BIOS plug-ins and modules.

In this chapter, you modify the example from Chapter 4 to create a program that schedules its functions and allows for multi-threading. You view performance information for debugging purposes. You also use more features of DSP/BIOS, including the Execution Graph, the real-time analysis control panel (RTA Control Panel), the Statistics View, and the CLK, SWI, STS, and TRC modules.

This chapter requires a physical board and cannot be carried out using a software simulator. Also, this chapter requires the DSP/BIOS components of Code Composer Studio.

Topic **Page**

5.1 Opening and Examining the Project

You begin by opening a project with Code Composer Studio and examining the source code files and libraries used in that project.

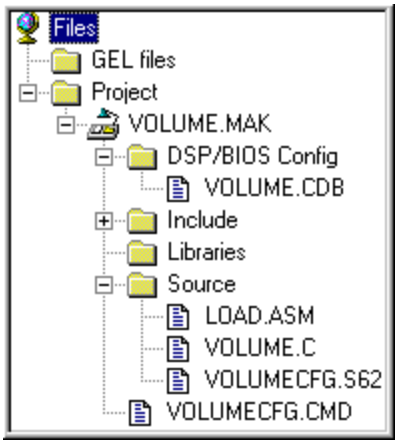
1) If you installed Code Composer Studio in c:\ti, create a folder called volume2 in the c:\ti\myprojects folder. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)

2) Copy all files from the c:\ti\c6000\tutorial\volume2 folder to this new folder.

3) From the Windows Start menu, choose Programs→Code Composer Studio 'C6000→CCStudio.

4) Choose Project→Open. Select the volume.mak file in the folder you created and click Open.

5) Expand the Project View by clicking the + signs next to Project, VOLUME.MAK, DSP/BIOS Config, and Source. The volumecfg.cmd file, which was created along with a configuration file, includes a large number of DSP/BIOS header files. (You do not need to examine all these header files.)



The files used in this project include:

- **volume.cdb**. This is the configuration file for the project.
- **volume.c**. This is the source code for the main program. It has been revised from the version you used in the previous chapter to support using DSP/BIOS in this program. You examine the source code in the next section.
- **volume.h**. This is a header file included by volume.c to define various constants and structures. It is identical to the volume.h file used in the previous chapter.
- **load.asm**. This file contains the load routine, a simple assembly loop routine that is callable from C with one argument. It is identical to the load.asm file used in the previous chapter.
- **volumecfg.cmd**. This linker command file is created when saving the configuration file.
- **volumecfg.s62**. This assembly file is created when saving the configuration file.
- **volumecfg.h62**. This header file is created when saving the configuration file.

5.2 Reviewing the Source Code

This example modifies the example from Chapter 4 to introduce real-time behavior. Rather than having the main function loop forever, the data I/O in the real application is likely to happen as a result of a periodic external interrupt. A simple way to simulate a periodic external interrupt in the example is to use the timer interrupt from the on-chip timer.

1) Double-click on the volume.c file in the Project View to see the source code in the right half of the Code Composer Studio window.

2) Notice the following aspects of the example:

   - The data types for declarations have changed. DSP/BIOS provides data types that are portable to other processors. Most types used by DSP/BIOS are capitalized versions of the corresponding C types.
   - The code uses #include to reference three DSP/BIOS header files: std.h, log.h, and swi.h. The std.h file must be included before other DSP/BIOS header files.
   - The objects created in the configuration file are declared as external. You examine the configuration file in the next section.
   - The main function no longer calls the dataIO and processing functions. Instead, the main function simply returns after calling LOG_printf to display a message. This drops the program into the DSP/BIOS idle loop. At this point, the DSP/BIOS scheduler manages thread execution.
   - The processing function is now called by a software interrupt called processing_SWI, which yields to all hardware interrupts. Alternatively, a hardware ISR could perform the signal processing directly. However, signal processing may require a large number of cycles, possibly more than the time until the next interrupt. Such processing would prevent the interrupt from being handled.

- The dataIO function calls SWI_dec, which decrements a counter associated with a software interrupt object. When the counter reaches 0, the software interrupt schedules its function for execution and resets the counter.

  The dataIO function simulates hardware-based data I/O. A typical program accumulates data in a buffer until it has enough data to process. In this example, the dataIO function is performed 10 times for each time the processing function is performed. The counter decremented by SWI_dec controls this.

```
#include <std.h>
#include <log.h>
#include <swi.h>

#include "volume.h"

/* Global declarations */
Int inp_buffer[BUFSIZE];          /* processing data buffers */
Int out_buffer[BUFSIZE];

Int gain = MINGAIN;               /* volume control variable */
Uns processingLoad = BASELOAD; /* processing load value */

/* Objects created by the Configuration Tool */
extern far LOG_Obj trace;
extern far SWI_Obj processing_SWI;

/* Functions */
extern Void load(Uns loadValue);

Int processing(Int *input, Int *output);
Void dataIO(Void);

/* ======== main ======== */
Void main()
{
    LOG_printf(&trace,"volume example started\n");
    /* fall into DSP/BIOS idle loop */
    return;
}
```

```
/* ======== processing ======== *
 * FUNCTION:     Called from processing_SWI to apply signal
 *               processing transform to input signal.
 * PARAMETERS:   Address of input and output buffers.
 * RETURN VALUE: TRUE.    */
Int processing(Int *input, Int *output)
{
    Int size = BUFSIZE;
    while(size--){
        *output++ = *input++ * gain;
    }
    /* additional processing load */
    load(processingLoad);

    return(TRUE);
}

/* ======== dataIO ======== *
 * FUNCTION:     Called from timer ISR to fake a periodic
 *               hardware interrupt that reads in the input
 *               signal and outputs the processed signal.
 * PARAMETERS:   none
 * RETURN VALUE: none     */
Void dataIO()
{
    /* do data I/O */

    /* post processing_SWI software interrupt */
    SWI_dec(&processing_SWI);
}
```

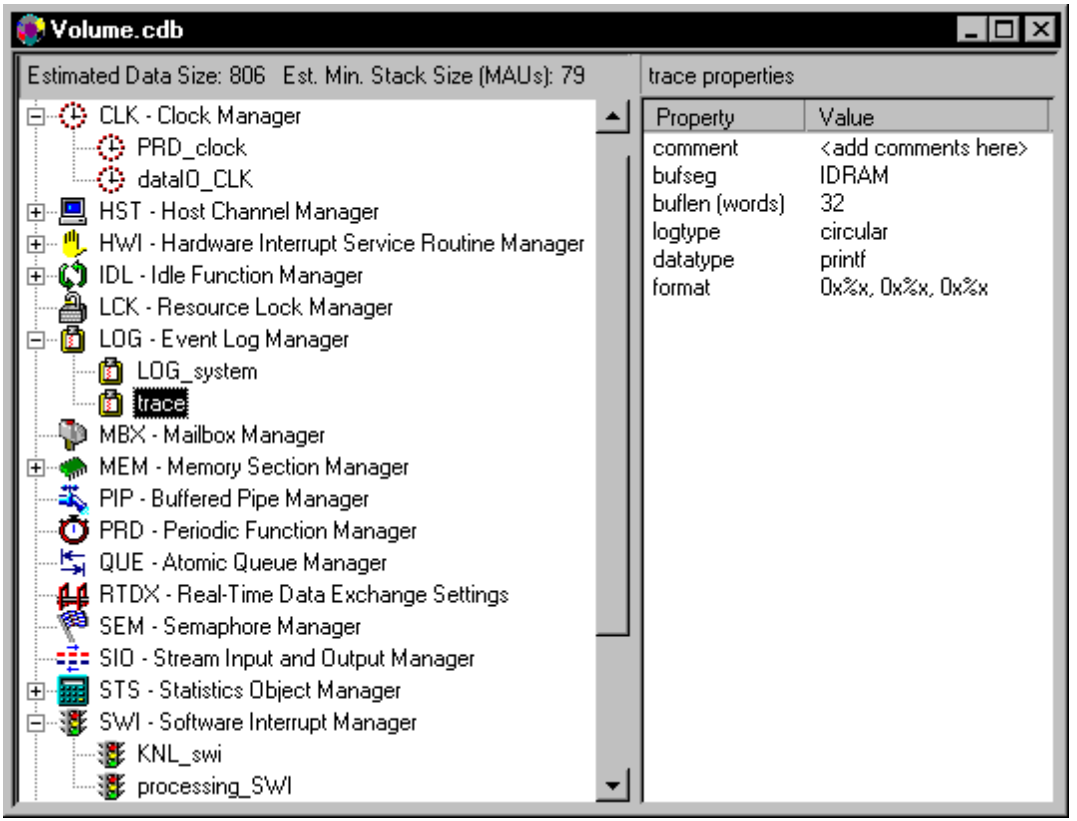
5.3 Modifying the Configuration File

A DSP/BIOS configuration file has already been created for this example. In this section, you examine the objects that have been added to the default configuration.

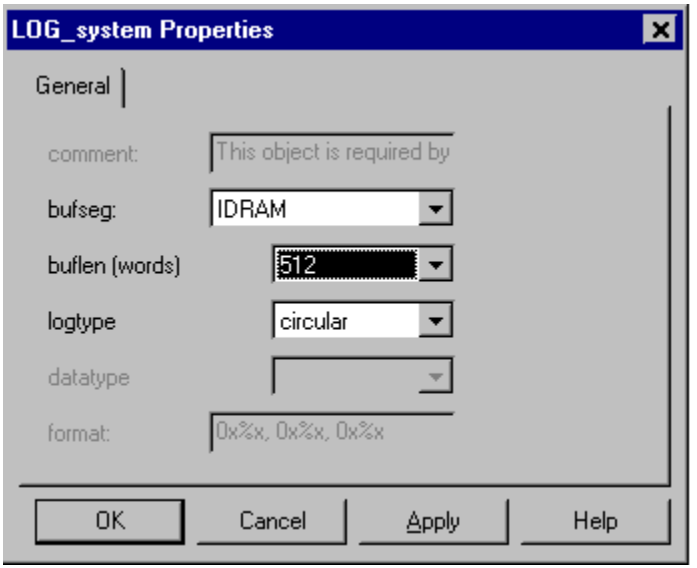
1) In the Project View, double-click on the volume.cdb file (in the DSP/BIOS Config folder) to open it.

2) Click the + signs next to the CLK, LOG, and SWI managers. This configuration file contains objects for these modules in addition to the default set of objects in the configuration file.

3) Highlight the LOG object called trace. You see the properties for this log in the right half of the window. This object's properties are the same as those of the trace LOG you created in section 3.1, page 3-2. The volume.c program calls LOG_printf to write `volume example started` to this log.



4) Right-click on the LOG object called LOG_system. From the pop-up menu, select Properties.



You see the properties dialog for this object. At run time, this log stores events traced by the system for various DSP/BIOS modules.

5) Change the buflen property to 512 words and click OK.

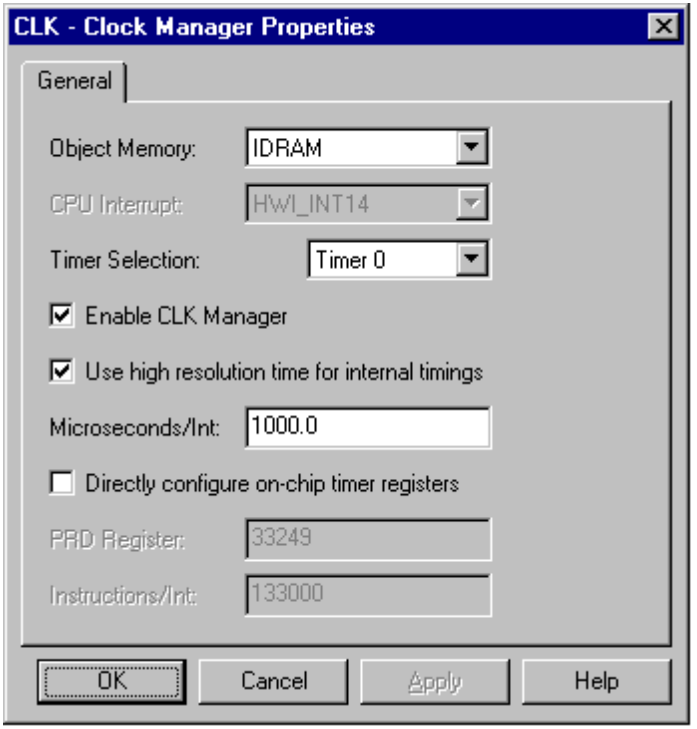
6) Highlight the CLK object called dataIO_CLK. Notice that the function called when this CLK object is activated is _dataIO. This is the dataIO function in volume.c.

Note: Underscores and C Function Names

This C function name is prefixed by an underscore because saving the configuration generates assembly language files. The underscore prefix is the convention for accessing C functions from assembly. (See the section on interfacing C with assembly language in the *TMS320C6000 Optimizing C Compiler User's Guide* for more information.)

This rule applies only to C functions you write. You do not need to use the underscore prefix with configuration-generated objects or DSP/BIOS API calls because two names are automatically created for each object: one prefixed with an underscore and one without.

7) Since the dataIO function is no longer run within main, what causes this CLK object to run its function? To find out, right-click on the CLK - Clock Manager object. From the pop-up menu, select Properties. You see the Clock Manager Properties dialog.



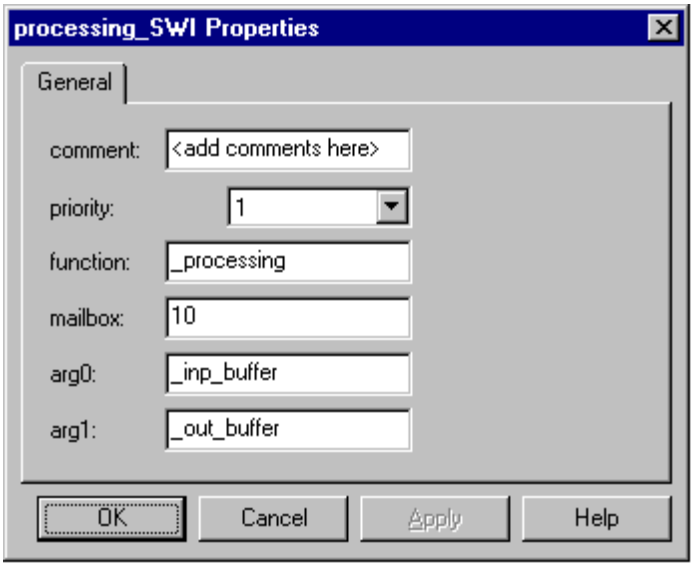
Notice that the CPU Interrupt for the Clock Manager is HWI_INT14. This property is gray because it is actually set by the HWI_INT14 object.

8) Click Cancel to close the Clock Manager Properties dialog without making any changes.

9) Expand the list of HWI objects and examine the properties of the HWI_INT14 object. Its interrupt source is Timer 0 on the DSP and it runs a function called CLK_F_isr when the on-chip timer causes an interrupt.

   The CLK object functions run from the context of the CLK_F_isr hardware interrupt function. Therefore, they run to completion without yielding and have higher priority than any software interrupts. (The CLK_F_isr saves the register context, so the CLK functions do not need to save and restore context as would be required normally within a hardware ISR function.)

10) Right-click on the processing_SWI software interrupt object. From the pop-up menu, select Properties.



- **function**. When this software interrupt is activated, the processing function runs. This function is shown in section 5.2, page 5–5.
- **mailbox**. The mailbox value can control when a software interrupt runs. Several API calls affect the value of the mailbox and can post the software interrupt depending on the resulting value. When a software interrupt is posted, it runs when it is the highest priority software or hardware interrupt thread that has been posted.
- **arg0, arg1**. The inp_buffer and out_buffer addresses are passed to the processing function.

11) Click Cancel to close this properties dialog without making any changes.

12) Since the processing function is no longer run within main, what causes this SWI object to run its function? In volume.c, the dataIO function calls SWI_dec, which decrements the mailbox value and posts the software interrupt if the new mailbox value is 0. So, this SWI object runs its function every tenth time the dataIO_CLK object runs the dataIO function.

13) Choose File→Close. You are asked whether you want to save your changes to volume.cdb. Click Yes. Saving this file also generates volumecfg.cmd, volumecfg.s62, and volumecfg.h62.

14) Choose Project→Build or click the (Incremental Build) toolbar button.

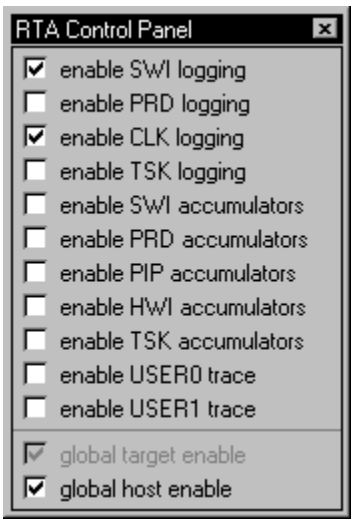
5.4 Viewing Thread Execution with the Execution Graph

While you could test the program by putting a Probe Point within the processing function and view graphs of input and output data (as you did in the previous chapter), you have already tested the signal processing algorithm. At this stage of development, your focus is on making sure the threads can meet their real-time deadlines.

1) Choose File→Load Program. Select the program you just built, volume.out in the volume2 folder, and click Open.

2) Choose Debug→Go Main. The program runs to the first statement in the main function.

3) Choose Tools→DSP/BIOS→RTA Control Panel. You see a list of instrumentation types at the bottom of the Code Composer Studio window.



4) Right-click on the area that contains the check boxes and deselect Allow Docking, or select Float in Main Window, to display the RTA Control Panel in a separate window. Resize the window so that you can see all of the check boxes shown here.

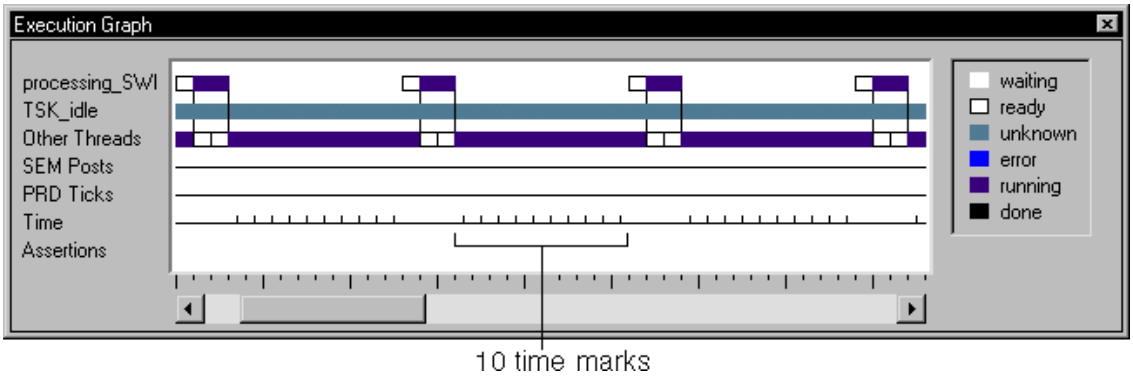
5) Put check marks in the boxes shown here to enable SWI and CLK logging and to globally enable tracing on the host.

6) Choose Tools→DSP/BIOS→Execution Graph. The Execution Graph appears at the bottom of the Code Composer Studio window. You may want to resize this area or display it as a separate window.

7) Right-click on the RTA Control Panel and choose Property Page from the pop-up menu.

8) Verify that the Refresh Rate for Message Log/Execution Graph is 1 second and click OK.

9) Choose Debug→Run or click the (Run) toolbar button. The Execution Graph should look similar to this:



10) The marks in the Time row show each time the Clock Manager ran the CLK functions. Count the marks between times the processing_SWI object was running. There should be 10 marks. This indicates that the processing_SWI object ran its function every tenth time the dataIO_CLK object ran its function. This is as expected because the mailbox value that is decremented by the dataIO function starts at 10.

5.5 Changing and Viewing the Load

Using the Execution Graph, you saw that the program meets its real-time deadlines. However, the signal processing functions in a typical program must perform more complex and cycle consuming work than multiplying a value and copying it to another buffer. You can simulate such complex threads by increasing the cycles consumed by the load function.

1) Choose Tools→DSP/BIOS→CPU Load Graph. A blank CPU Load Graph appears.

2) Right-click on the RTA Control Panel and choose Property Page from the pop-up menu.

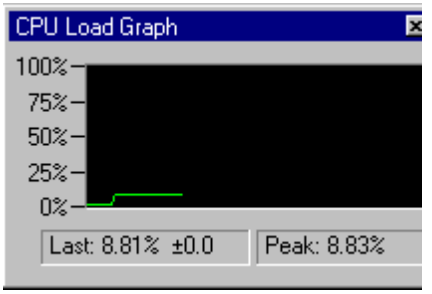
3) Change the Refresh Rate for Statistics View/CPU Load Graph to 0.5 seconds and click OK. Notice that the CPU load is currently very low.

   The Statistics View and the CPU Load transfer very little data from the target to the host, so you can set these windows to update frequently without causing a large effect on program execution. The Message Log and Execution Graph transfer the number of words specified for the buflen property of the corresponding LOG object in the configuration file. Since more data is transferred, you may want to make these windows update less frequently.

4) Choose File→Load GEL. In the Load GEL File dialog, select the volume.gel file and click Open.

5) Choose GEL→Application Control→Load.

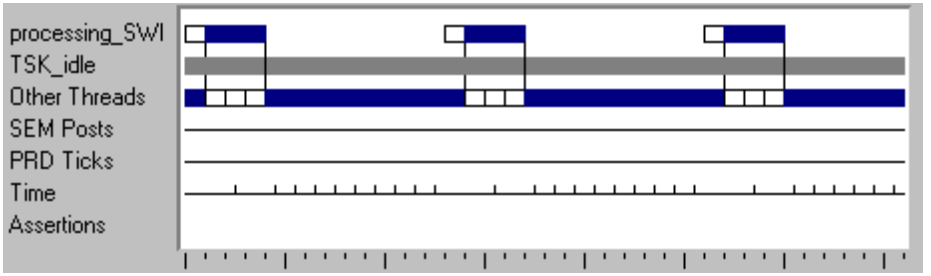
6) Type **`100`** as the new load and click Execute. The CPU load increases to about 9%.



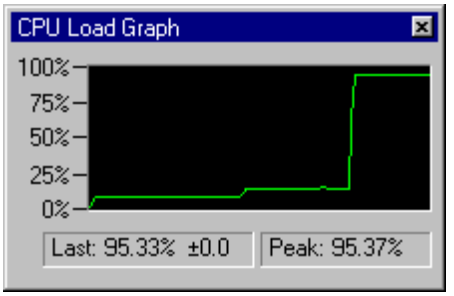
7) Right-click on the Execution Graph and choose Clear from the pop-up menu. Notice that the program still meets its real-time deadline. There are 10 time marks between each execution of the processing_SWI function.

8) Using the GEL control, change the load to **`200`** and click Execute.

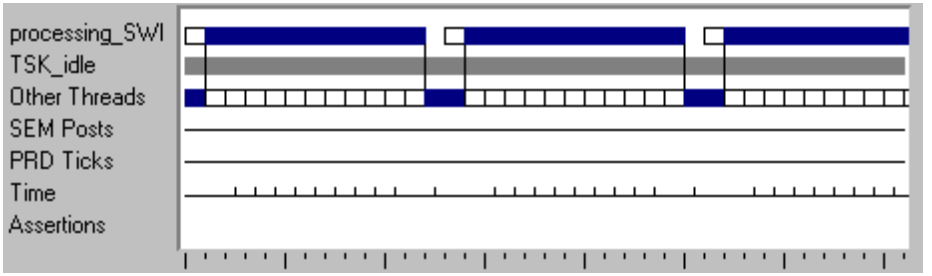
9) Right-click on the Execution Graph and choose Clear from the pop-up menu. One or two of the time marks occur while the processing_SWI function is executing. Does this mean the program is missing its real-time deadline? No, it shows that the program is functioning correctly. The hardware interrupt that runs the CLK object functions can interrupt the software interrupt processing, and the software interrupt still completes its work before it needs to run again.



10) Using the GEL control, change the load to **`1250`** and click Execute. The CPU load increases to about 95% and the Execution Graph and CPU load are updated less frequently.



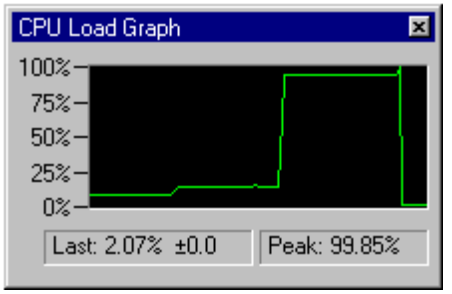
11) Right-click on the Execution Graph and choose Clear from the pop-up menu. The program still meets its real-time deadline because processing_SWI completes before 10 time marks have occurred.



12) Using the GEL control, change the load to **`1500`** and click Execute. The CPU Load Graph and the Execution Graph stop updating. This is because Execution Graph data is sent to the host within an idle thread, which has the lowest execution priority within the program. Because the higher-priority threads are using all the processing time, there is not enough time for the host control updates to be performed. The program is now missing its real-time deadline.

13) Choose Debug→Halt. This stops the program and updates the Execution Graph. You may see squares in the Assertions row. These squares indicate that the Code Composer Studio detected that the application missed a real-time deadline.

14) Using the GEL control, change the load to 10 and click Execute. The CPU load and Execution Graph begin updating again.



Note: Modifying the Load Using RTDX

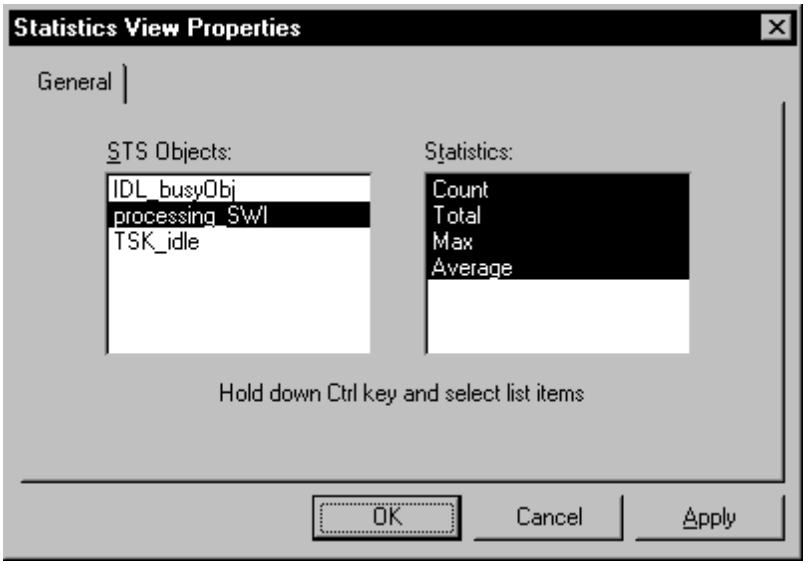
Using the Load GEL control temporarily halts the target. If you are analyzing a real-time system and do not want to affect system performance, modify the load using a Real-Time Data Exchange (RTDX) application. The next chapter shows how to modify the load in real-time using RTDX.

5.6 Analyzing Thread Statistics

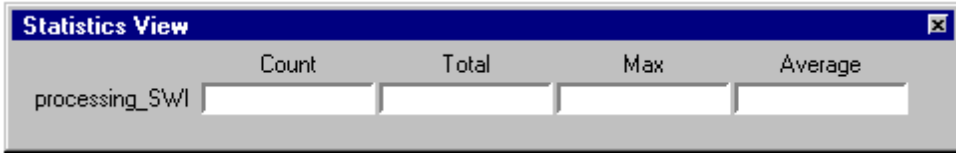
You can use other DSP/BIOS controls to examine the load on the DSP and the processing statistics for the processing_SWI object.

1) Choose Tools→DSP/BIOS→Statistics View. A Statistics View area that says Load DSP/BIOS program and/or set property to use control appears. It says this because you need to select the statistics you want to view.

2) Right-click on the Statistics View area and choose Property Page from the pop-up menu. Highlight the items shown here and click OK. (You can hold down the Ctrl key to select individual items or the Shift key to select a continuous range of items.)



3) You see the statistics fields for the processing_SWI objects. You may want to make this area a separate window (by right-clicking on it and deselecting Allow Docking in the pop-up menu) and resize the window so that you can see all four fields.



4) In the RTA Control Panel, put a check mark in the enable SWI accumulators box.

5) If you have halted the program, click the  (Run) toolbar button.

6) Notice the Max value in the Statistics View. SWI statistics are measured in instruction cycles.

7) Using the GEL control, increase the load and click Execute. Notice the change in the Max value for the number of instructions performed from the beginning to the end of processing_SWI increases.

8) Experiment with different load values. If you decrease the load, right-click on the Statistics View and select Clear from the pop-up menu.

   This resets the fields to their lowest possible values, allowing you to see the current number of instruction cycles in the Max field.

9) Click the  (Halt) toolbar button and close all the DSP/BIOS and GEL controls you have opened.

5.7 Adding Explicit STS Instrumentation

In the previous section, you used the Statistics View to see the number of instructions performed during a software interrupt's execution. If you use a configuration file, DSP/BIOS supports such statistics automatically. This is called implicit instrumentation. You can also use API calls to gather other statistics. This is called explicit instrumentation.

1) In the Project View, double-click on the volume.cdb file (in the DSP/BIOS Config folder) to open it.

2) Right-click on the STS manager and choose Insert STS from the pop-up menu.

3) Rename the new STS0 object to processingLoad_STS. The default properties for this object are all correct.

4) Choose File→Close. You are asked whether you want to save your changes to volume.cdb. Click Yes.

5) In the Project View, double-click on the volume.c program to open it for editing. Make the following changes to the program:

   - Add the following lines below the line that includes the swi.h file:

     ```
     #include <clk.h>
     #include <sts.h>
     #include <trc.h>
     ```

   - Add the following to the declarations in the section labeled with the comment "`Objects created by the Configuration Tool`":

     ```
     extern far STS_Obj processingLoad_STS;
     ```

   - Add the following lines within the processing function before the call to the load function:

     ```
     /* enable instrumentation only if TRC_USER0 is set */
     if (TRC_query(TRC_USER0) == 0) {
         STS_set(&processingLoad_STS, CLK_gethtime());
     }
     ```

   - Add the following lines within the processing function after the call to the load function:

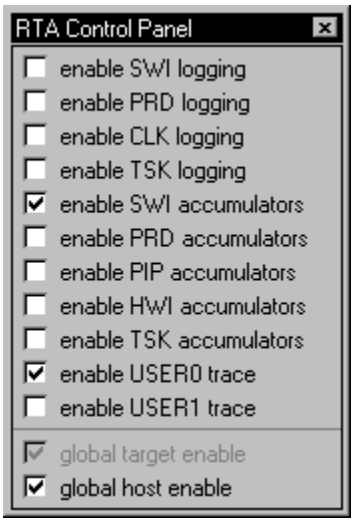
     ```
     if (TRC_query(TRC_USER0) == 0) {
         STS_delta(&processingLoad_STS, CLK_gethtime());
     }
     ```

6) Choose File→Save to save your changes to volume.c.

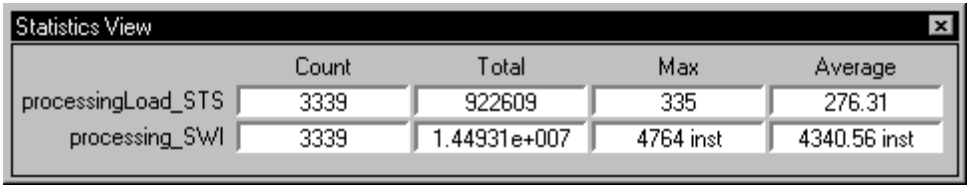
7) Choose Project→Build or click the  (Incremental Build) toolbar button.

5.8 Viewing Explicit Instrumentation

To view information provided by the explicit instrumentation calls you added, you use the Statistics View and the RTA Control Panel.



1) Choose File→Load Program. Select the program you just rebuilt, volume.out, and click Open.
2) Choose Tools→DSP/BIOS→RTA Control Panel.
3) Right-click on the RTA Control Panel area and deselect Allow Docking to display the RTA Control Panel in a separate window. Resize the window so that you can see all of the check boxes shown here.
4) Put check marks in the boxes shown here to enable SWI accumulators, USER0 trace, and to globally enable tracing on the host. Enabling USER0 tracing causes the calls to TRC_query(TRC_USER0) to return 0.
5) Choose Tools→DSP/BIOS→Statistics View.
6) Right-click on the Statistics View area and choose Property Page from the pop-up menu. Highlight the processing_SWI and processingLoad_STS objects. Also, highlight all four statistics.
7) Click OK. You see the statistics fields for both objects. You may want to make this area a separate window (by deselecting Allow Docking in the pop-up menu) and resize the window so that you can see all the fields.
8) Choose Debug→Run or click the (Run) toolbar button.

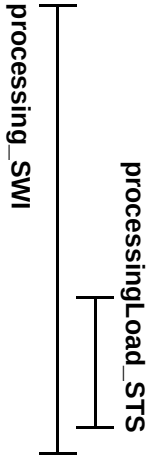


9) Multiply the Max value for processingLoad_STS by 4.

   Because you used the CLK_gethtime function to benchmark the processing load, the statistics for processingLoad_STS are measured in on-chip timer counter increments. SWI statistics are measured in instruction cycles. On TMS320C6000 DSPs, the high-resolution time is incremented every 4 instruction cycles. Therefore, to convert the processingLoad_STS units to instruction cycles, you multiply by 4.

10) Subtract the resulting processingLoad_STS Max value in instruction cycles from the processing_SWI Max value. The result should be about 3420

11) instructions. (The actual numbers shown may vary, especially if you are using a non-generic development board.) These instructions are performed within the processing function, but not between the calls to STS_set and STS_delta, as shown below.




```
/*  ======== processing ======== */
Int processing(Int *input, Int *output)
{
    Int size = BUFSIZE;
    while(size--){
        *output++ = *input++ * gain;
    }
    /* enable instrumentation if TRC_USER0 is set */
    if (TRC_query(TRC_USER0) == 0) {
        STS_set(&processingLoad_STS, CLK_gethtime());
    }
    /* additional processing load */
    load(processingLoad);
    if (TRC_query(TRC_USER0) == 0) {
        STS_delta(&processingLoad_STS, CLK_gethtime());
    }
    return(TRUE);
}
```

   For example, if the load is 10, the processingLoad_STS Max is about 2604 and the processing_SWI Max is about 13832. To calculate the instruction cycles performed within the processing function but outside the calls to STS_set and STS_delta, the equation is:

```
13832 - (2604 * 4) = 3416
```

12) Choose GEL→Application Control→Load. (If you have closed and restarted Code Composer Studio, you must reload the GEL file.)

13) Change the Load and click Execute.

14) Notice that while both Max values increase, the difference between the two Max values (after you multiply the processingLoad_STS Max by 4) stays the same.

15) Remove the check mark from the enable USER0 trace box in the RTA Control Panel.

16) Right-click on the Statistics View and choose Clear from the pop-up menu.

17) Notice that no values are updated for processingLoad_STS. This is because disabling the USER0 trace causes the following statement in the program to be false:

```
if (TRC_query(TRC_USER0) == 0)
```

As a result, the calls to STS_set and STS_delta are not performed.

18) Before continuing to the next chapter (after completing section 5.9, page 5-20), perform the following steps to prepare for the next chapter:

- Click  (Halt) or press Shift F5 to stop the program.
- Close all GEL dialog boxes, DSP/BIOS plug-ins, and source windows.

5.9 Things to Try

To further explore DSP/BIOS, try the following:

❏ Change the Statistics Units property of the SWI Manager in the configuration file to milliseconds or microseconds. Rebuild and reload the program and notice how the values in the Statistics View change.

❏ Change the Host Operation property of the processingLoad_STS object in the configuration file to A * x and the A property to 4. This Host Operation multiplies the statistics by 4. The calls to CLK_gethtime cause statistics to be measured in high-resolution timer increments, which occur every 4 CPU cycles. So, changing the Host Operation converts the timer increments to CPU cycles. Rebuild the program and notice how the values in the Statistics View change.

❏ Modify volume.c by using the CLK_getltime function instead of the CLK_gethtime function. Rebuild the program and notice how the values in the Statistics View change. The CLK_getltime function gets a low-resolution time that corresponds to the timer marks you saw in the Execution Graph. You must increase the load significantly to change the Statistics View values when using CLK_getltime.

5.10 Learning More

To learn more about the CLK, SWI, STS, and TRC modules, see the online help or the *TMS320C6000 DSP/BIOS User's Guide* (which is provided as an Adobe Acrobat file).

Chapter 6

Analyzing Real-Time Behavior

This chapter introduces techniques for analyzing and correcting real-time program behavior.

In this chapter, you analyze real-time behavior and correct scheduling problems using the example from Chapter 5. You use RTDX (Real-Time Data Exchange) to make real-time changes to the target, use DSP/BIOS periodic functions, and set software interrupt priorities.

This chapter requires a physical board and cannot be carried out using a software simulator. Also, this chapter requires the DSP/BIOS and RTDX components of Code Composer Studio.

Topic **Page**

6.1 Opening and Examining the Project

In this chapter, you modify the example you worked on in the previous chapter.

Note: Copy Example Files if Previous Chapter Not Completed

If you did not complete the previous chapter, you can copy example files from the volume3 folder that reflect the state of the example at the end of the previous chapter. Copy all the files from c:\ti\c6000\tutorial\volume3 (or the location where you installed Code Composer Studio) to your working folder.

1) Copy *only* the following files from the c:\ti\c6000\tutorial\volume4\ folder to your working folder. (Note that you should *not* copy all the files from the volume4 folder. In particular, do not copy the volume.cdb file.)

   - **volume.c**. The source code has been modified to allow you to use the RTDX module to change the load without stopping the target program. You examine the changes in section 6.3, page 6-5.
   - **loadctrl.exe**. This is a simple Windows application written in Visual Basic 5.0. It sends load values to the target in real time using RTDX.
   - **loadctrl.frm, loadctrl.frx, loadctrl.vbp**. If you have Visual Basic, you can use it to examine these source files for the loadctrl.exe application.

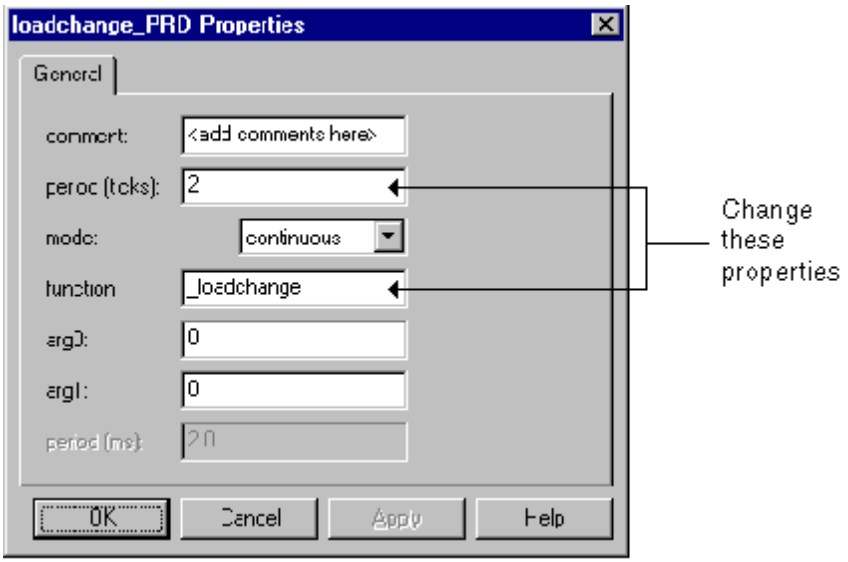
2) From the Windows Start menu, choose Programs→Code Composer Studio 'C6000→CCStudio.

3) Choose Project→Open. Select the volume.mak file in your working folder and click Open.

6.2 Modifying the Configuration File

For this example, you need to add one new object to the configuration file. (The volume.cdb file in the c:\ti\c6000\tutorial\volume4\ folder already contains this object.)

1) In the Project View, double-click on the volume.cdb file to open it.
2) Select LOG_system, change the buflen property to 512 words, and click OK (as you did in section 5.3, page 5–7).
3) Right-click on the PRD manager and choose Insert PRD from the pop-up menu.
4) Rename the PRD0 object to loadchange_PRD.
5) Right-click on the loadchange_PRD object and choose Properties from the pop-up menu.
6) Set the following properties for the loadchange_PRD object and click OK.



- Change the period to 2. By default, the PRD manager uses the CLK manager to drive PRD execution. The default properties for the CLK class make a clock interrupt trigger a PRD tick each millisecond. So, this PRD object runs its function every 2 milliseconds.
- Change the function to _loadchange. This PRD object executes the loadchange C function each time the period you chose elapses. (Recall that you need to use an underscore prefix for C functions in the configuration file.) You look at this function in the next section.

7) Click the + sign next to the SWI manager. A SWI object called PRD_swi was added automatically. This software interrupt executes periodic functions at run time. Therefore, all PRD functions are called within the context of a software interrupt and can yield to hardware interrupts. In contrast, CLK functions run in the context of a hardware interrupt. (The KNL_swi object runs a function that runs the TSK manager. See the *TMS320C6000 DSP/BIOS User's Guide* and the online help for information about tasks, which are not used in this tutorial.)

8) Click the + sign next to the CLK manager. Notice that the CLK object called PRD_clock runs a function called PRD_F_tick. This function causes the DSP/BIOS system clock to tick (by calling the PRD_tick API function) and the PRD_swi software interrupt to be posted if any PRD functions need to run. PRD_swi runs the functions for all the PRD objects whose period has elapsed.

9) Right click on the PRD manager, and choose Properties from the pop-up menu. The PRD manager has a property called Use CLK Manager to drive PRD. Make sure this box is checked for this example.

   In your own projects, if you remove the check mark from this box, the PRD_clock object would be deleted automatically. Your program could then call PRD_tick from some other event, such as a hardware interrupt, to drive periodic functions.

10) Recall that the processing_SWI object has a mailbox value of 10 and that the mailbox value is decremented by the dataIO_CLK object, which runs every millisecond. As a result, the processing_SWI runs its function every 10 milliseconds. In contrast, the loadchange_PRD object should run its function every 2 milliseconds.

11) Choose File→Close. You are asked whether you want to save your changes to volume.cdb. Click Yes. Saving this file also generates volumecfg.cmd, volumecfg.s62, and volumecfg.h62.

12) Choose Project→Rebuild All or click the  (Rebuild All) toolbar button.

6.3 Reviewing the Source Code Changes

Double-click on the volume.c file in the Project View to see the source code in the right half of the Code Composer Studio window.

Since you copied the volume.c file from the c:\ti\c6000\tutorial\volume4\ folder to your working folder, the source code now contains the following differences from the source code used in the previous chapter:

- ❏ Added the following to the list of included header files:

```
#include <rtdx.h>
```

- ❏ Added the following to the declarations:

```
RTDX_CreateInputChannel(control_channel);

Void loadchange(Void);
```

- ❏ Added the following call to the main function:

```
RTDX_enableInput(&control_channel);
```

- ❏ The following function is called by the PRD object you created in section 6.2, page 6-3. This is where the processor is controlled.

```
/* ======== loadchange ========
 * FUNCTION: Called from loadchange_PRD to
 * periodically update load value.
 * PARAMETERS: none.
 * RETURN VALUE: none.
 */
Void loadchange()
{
    static Int control = MINCONTROL;

    /* Read new load control when host sends it */
    if (!RTDX_channelBusy(&control_channel)) {
        RTDX_readNB(&control_channel, &control,
                    sizeof(control));
        if ((control < MINCONTROL) || (control > MAXCONTROL)) {
            LOG_printf(&trace,"Control value out of range");
        }
        else {
            processingLoad = control;
            LOG_printf(&trace,"Load value = %d",processingLoad);
        }
    }
}
```

This function uses RTDX API functions to change the load of the processing signal in real time. Notice the following aspects of these changes:

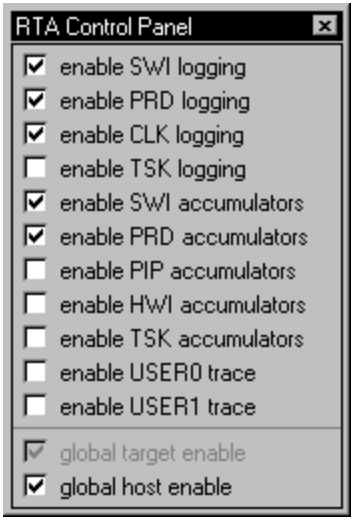
- The call to RTDX_enableInput enables the input channel called control_channel so that data can flow on it from the host to the target. At run time, a Visual Basic host client writes a load control value on that channel, thereby sending it to the target application.
- The call to RTDX_readNB asks the host to send a load control value on the control_channel and stores it in the variable called control. This call is non-blocking; it returns without waiting for the host to send the data. The data is delivered when the host client writes to control_channel. From the time of the call to RTDX_readNB until the data is written to the variable control, this channel is busy, and no additional requests can be posted on this channel (that is, calls to RTDX_readNB do not succeed). During that time, the call to RTDX_channelBusy returns TRUE for control_channel.
- The processingLoad = control; statement sets the processing load to the value specified by the control.

6.4 Using the RTDX Control to Change the Load at Run Time

While you could test the program by putting a Probe Point within the processing function and view graphs of input and output data (as you did in section 4.3, page 4-6), you have already tested the signal processing algorithm. At this stage of development, your focus is on making sure the threads you have added can still meet their real-time deadlines. Also, Probe Points halt the target and interfere with the real-time aspects of the test.

1) Choose File→Load Program. Select the program you just rebuilt, volume.out, and click Open.

2) Choose Tools→DSP/BIOS→RTA Control Panel.



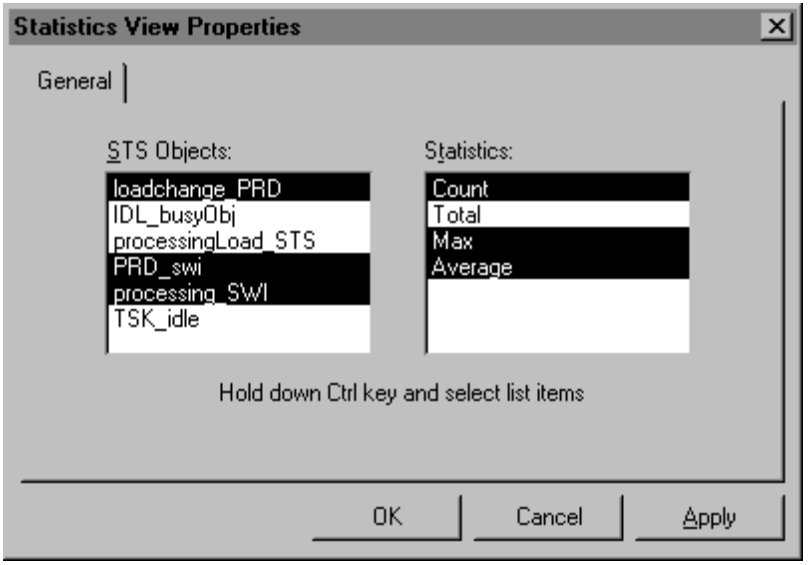
3) Right-click on the RTA Control Panel and deselect Allow Docking to display the RTA Control Panel in a separate window. Resize the window so that you can see all of the check boxes shown here.

4) Put check marks in the boxes shown here to enable SWI, PRD, and CLK logging; SWI and PRD accumulators; and global tracing on the host.

5) Choose Tools→DSP/BIOS→Execution Graph. The Execution Graph area appears at the bottom of the Code Composer Studio window. You may want to resize this area or display it as a separate window.

6) Choose Tools→DSP/BIOS→Statistics View.

7) Right-click on the Statistics View area and choose Property Page from the pop-up menu. Highlight the items shown here.



8) Click OK.

9) Resize the Statistics area to see the fields for the statistics you selected.

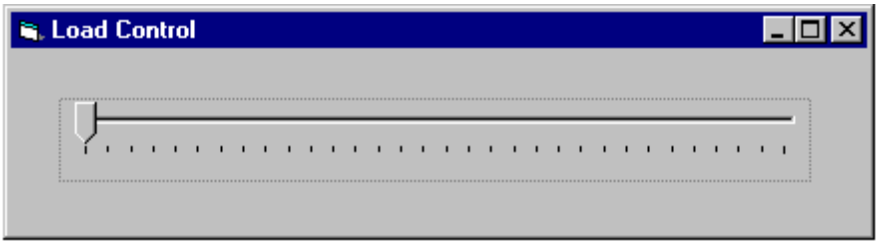
10) Right-click on the RTA Control Panel and choose Property Page from the pop-up menu.

11) Set the Refresh Rate for Message Log/Execution Graph to 1 second and the Refresh Rate for Statistics View/CPU Load Graph to 0.5 seconds. Then click OK.

12) Choose Tools→RTDX.

13) Notice that RTDX is already enabled. This happened behind-the-scenes in Step 2 when you opened a DSP/BIOS control. DSP/BIOS controls configure and enable RTDX in continuous mode. In continuous mode, RTDX does not record data received from the target in a log file (as it does in non-continuous mode). This allows continuous data flow. (If your program does not use DSP/BIOS, you can use the RTDX area to configure and enable RTDX directly.)

14) Using the Windows Explorer, run loadctrl.exe, which is located in your working folder. The Load Control window appears.

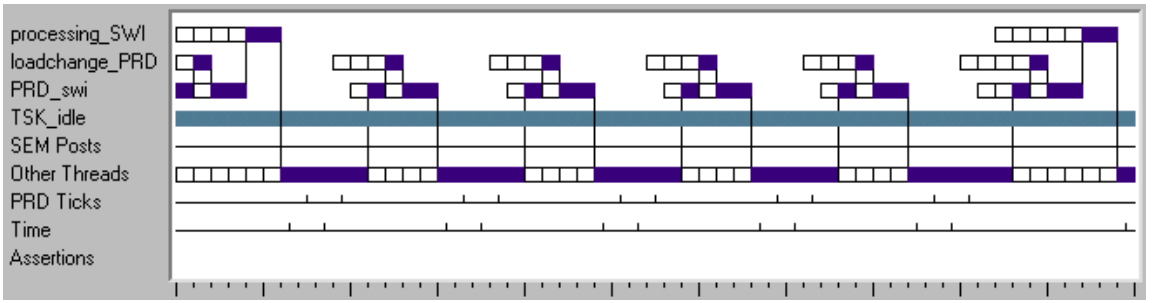


Each mark on the slider changes the load value by 50, or about 50,000 instructions.

This simple Windows application was written using Visual Basic and RTDX. If you have Visual Basic, you can examine the source files for the loadctrl.exe application stored in the c:\ti\c6000\tutorial\volume4\ folder. This application uses the following RTDX functions:

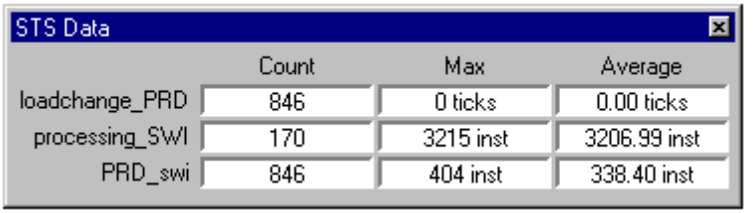
- **rtdx.Open("control_channel", "W")**. Opens a control channel to write information to the target when you open the application
- **rtdx.Close()**. Closes the control channel when you close the application
- **rtdx.Writel2(datal2, bufstate)**. Writes the current value of the slider control to control_channel so that the target program can read this value and use it to update the load

15) Choose Debug→Run or click the (Run) toolbar button.



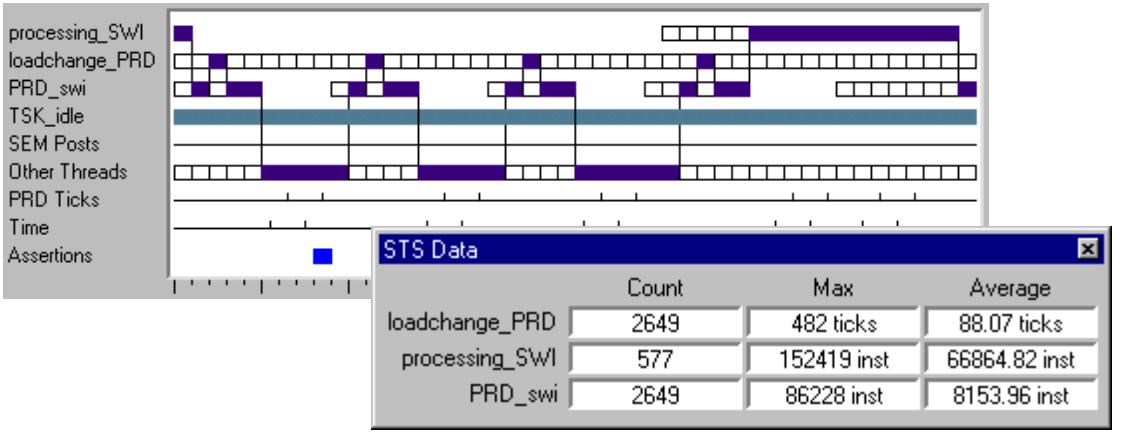
Notice that processing_SWI occurs once every 10 time ticks (and PRD ticks). PRD_swi runs once every 2 PRD ticks. The loadchange_PRD runs within the context of PRD_swi. These are the expected execution frequencies.

PRD statistics are measured in PRD ticks. SWI statistics are measured in instruction cycles. The Max and Average fields for loadchange_PRD show that there is less than a full PRD tick between the time this function needs to start running and its completion. (The actual numbers shown may vary.)



16) Use the Load Control window to gradually increase the processing load. (If you move the slider in the Load Control window while the DSP program is halted, the new load control values are buffered on the host by RTDX. These have no effect until the DSP application runs again and calls RTDX_readNB to request updated load values from the host.)

17) Repeat step 16 until you see the Max and Average values for loadchange_PRD increase and blue squares appear in the Assertions row of the Execution Graph. Assertions indicate that a thread is not meeting its real-time deadline.



What is happening? The Max value for loadchange_PRD increases when you increase the load beyond a certain point. With the increased load, the processing_SWI takes so long to run that the loadchange_PRD cannot begin running until long past its real-time deadline.

When you increase the load so much that the low-priority idle loop is no longer executed, the host stops receiving real-time analysis data and the DSP/BIOS plug-ins stop updating. Halting the target updates the plug-ins with the queued data.

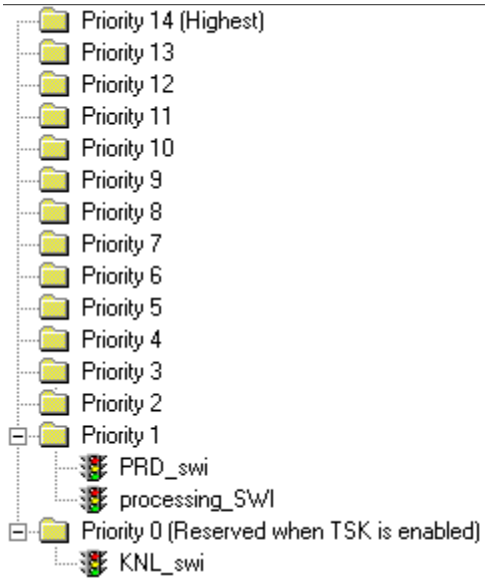
6.5 Modifying Software Interrupt Priorities

To understand why the program is not meeting its real-time deadline, you need to examine the priorities of the software interrupt threads.

1) Select Debug→Halt to halt the target.

2) In the Project View, double-click on the volume.cdb file to open it.

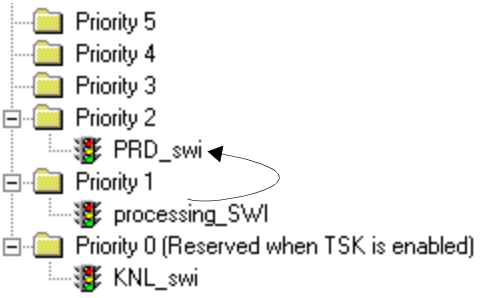
3) Highlight the SWI manager. Notice the SWI object priorities shown in the right half of the window. (The KNL_swi object runs a function that runs the TSK manager. This object must always have the lowest SWI priority. Tasks are not used in this lesson.)



   Because the PRD_swi and processing_SWI objects both have the same priority level, the PRD_swi cannot preempt the processing_SWI while it is running.

   The processing_SWI needs to run once every 10 milliseconds and PRD_swi needs to run every 2 milliseconds. When the load is high, processing_SWI takes longer than 2 milliseconds to run, and so it prevents PRD_swi from meeting its real-time deadline.

4) To correct this problem, use your mouse to select and drag PRD_swi to a higher priority level, such as Priority 2.



5) Select File→Save to save your changes.

6) Select File→Close to close volume.cdb.

7) Choose Project→Build or click the (Incremental Build) toolbar button.

8) Select File→Reload Program.

9) Select Debug→Run to run the example again. Use the RTDX-enabled Windows application loadctrl.exe to change the load at run time (as in section 6.4, page 6-7).

10) Notice that you can now increase the load without causing PRD_swi to miss its real-time deadline.

Note: Starving Idle Loop

It is still possible to starve the idle loop by increasing the processing load to maximum.

11) Before continuing to the next chapter (after completing section 6.6, page 6-12), perform the following steps to prepare for the next chapter:

- Click  (Halt) or press Shift F5 to stop the program.
- Close all GEL dialogs, DSP/BIOS plug-ins, and source windows.

6.6 Things to Try

To further explore DSP/BIOS, try the following:

❏ When you increase the load, the Execution Graph shows that processing_SWI requires more than one PRD tick to run. Does this mean that processing_SWI is missing its real-time deadline? Recall that processing_SWI must run every 10 milliseconds, while PRD ticks occur every millisecond.

❏ What would happen if the processing function were called directly from a hardware ISR rather than being deferred to a software interrupt? That would cause the program to miss its real-time deadline because hardware ISRs run at higher priority than the highest priority SWI object. Recall that when the load is high, PRD_swi needs to preempt processing_SWI. If processing_SWI is a hardware interrupt, it cannot be preempted by PRD_swi.

❏ View the CPU Load Graph. Use the RTA Control Panel to turn the statistics accumulators on and off. Notice that the CPU Load Graph appears unaffected. This demonstrates that the statistics accumulators place a very small load on the processor.

  How much do the statistics accumulators affect the statistics for processing_SWI? Watch the statistics view as you turn the statistics accumulators on and off. The difference is a precise measurement of the number of instructions each accumulator requires. Remember to right-click and clear the statistics view to see the effect.

❏ Add calls to STS_set and STS_delta in the loadchange function like the ones you added in section 5.7, page 5-17. How does this change affect

the CPU load? Now, add calls to STS_set and STS_delta in the dataIO function. How does this change affect the CPU load? Why? Consider the frequency at which each function is executed. Even small increases to the processing requirements for functions that run frequently can have dramatic effects on CPU load.

6.7 Learning More

To learn more about the software interrupt priorities and the RTDX and PRD modules, see the online help and the *TMS320C6000 DSP/BIOS API Reference Guide* (which is provided as an Adobe Acrobat file).

Chapter 7

Connecting to I/O Devices

This chapter introduces RTDX and DSP/BIOS techniques for implementing I/O.

In this chapter, you connect a program to an I/O device using RTDX and DSP/BIOS. You also use the HST, PIP, and SWI modules of the DSP/BIOS API.

This chapter requires a physical board and cannot be carried out using a software simulator. Also, this chapter requires the DSP/BIOS and RTDX components of Code Composer Studio.

Topic **Page**

7.1 Opening and Examining the Project

You begin by opening a project with Code Composer Studio and examining the source code files and libraries used in that project.

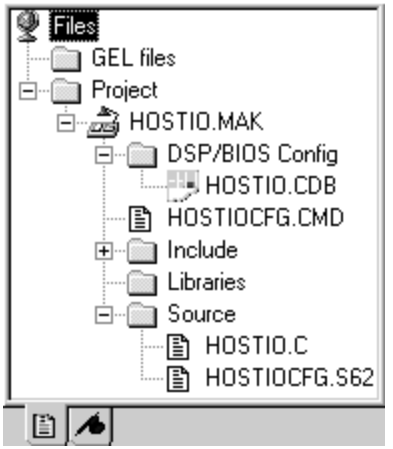
1) If you installed Code Composer Studio in c:\ti, create a folder called hostio in the c:\ti\myprojects folder. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)

2) Copy all files from the c:\ti\c6000\tutorial\hostio1 folder to this new folder.

3) From the Windows Start menu, choose Programs→Code Composer Studio 'C6000→CCStudio.

4) Choose Project→Open. Select the hostio.mak file in the folder you created and click Open.

5) Expand the Project View by clicking the + signs next to Project, HOSTIO.MAK, and Source. The hostiocfg.cmd file, which was created when the configuration was saved, includes a large number of DSP/BIOS header files. You do not need to examine all these header files.



The files used in this project include:

- **hostio.c**. This is the source code for the main program. You examine the source code in the next section.
- **signalprog.exe**. This Visual Basic application generates a sine wave and displays the input and output signals.
- **slider.exe**. This Visual Basic application allows you to control the volume of the output signal.
- **hostiocfg.cmd**. This linker command file is created when saving the configuration file. The only object that has been added to the default configuration is a LOG object called trace.
- **hostiocfg.s62**. This assembly file is created when saving the configuration file.
- **hostiocfg.h62**. This header file is created when saving the configuration file.

7.2 Reviewing the C Source Code

The example in this chapter simulates a DSP application that digitizes an audio signal, adjusts its volume, and produces an analog output at the adjusted volume.

For simplicity, no actual device is used to send and receive analog data in this example. Instead, the example tests the algorithm using host-generated digital data. Input and output data and volume control are transferred between the host and the target using RTDX.

A Visual Basic application running on the host uses RTDX to generate the input signal and display the input and output signals. This application allows developers to test the algorithm without stopping the target. Similar methods can be used to create display controls for real-time testing of other applications. You examine the Visual Basic application in section 7.3, page 7-6.

1) Double-click on the hostio.c file in the Project View to see the source code.

2) Notice the following aspects of this example:

   - Three RTDX channels are declared globally. The first input channel controls the volume. The second input channel receives the input signal from the host. The output channel sends the output signal from the target to the host. (Input and output channels are named from the perspective of the target application: input channels receive data from the host, and output channels send data to the host.)
   - The call to RTDX_channelBusy returns FALSE if the channel is not currently waiting for input. This indicates that the data has arrived and can be read. As in Chapter 6, the call to RTDX_readNB is non-blocking; it returns control to the DSP application without waiting to receive the data from the host. The data is delivered asynchronously when the host client writes it to the control_channel.
   - Calls to RTDX_Poll are used to communicate with the underlying RTDX layer to read and write data.
   - The call to RTDX_read waits for data if the channel is enabled.
   - The call to RTDX_write writes the contents of the buffer to the output RTDX channel if the channel is enabled.

- While control_channel is enabled by the target via a call to RTDX_enableInput, the other RTDX channels are not enabled from this program. Instead, a host program described in the next section enables these channels. This is because the slider control, which uses the control_channel, is viewed as an integral part of the application. By enabling this channel in the target program, you know the channel is enabled while the application is running. In contrast, the A2D and D2A channels are used to test the algorithm. Hence, these channels are enabled and disabled by the host application.

```
#include <std.h>
#include <log.h>
#include <rtdx.h>

#include "target.h"

#define BUFSIZE 64
#define MINVOLUME 1

typedef Int sample;     /* representation of a data sample from A2D */

/* Global declarations */
sample inp_buffer[BUFSIZE];
sample out_buffer[BUFSIZE];

Int volume = MINVOLUME; /* the scaling factor for volume control */

/* RTDX channels */
RTDX_CreateInputChannel(control_channel);
RTDX_CreateInputChannel(A2D_channel);
RTDX_CreateOutputChannel(D2A_channel);

/* Objects created by the Configuration Tool */
extern far LOG_Obj trace;

/*
 * ======== main ========
 */
Void main()
{
    sample *input = inp_buffer;
    sample *output = out_buffer;
    Uns size = BUFSIZE;

    TARGET_INITIALIZE();               /* Enable RTDX interrupt */

    LOG_printf(&trace,"hostio example started");

    /* enable volume control input channel */
    RTDX_enableInput(&control_channel);
```

```
    while (TRUE) {
        /* Read a new volume when the hosts send it */
        if (!RTDX_channelBusy(&control_channel)){
            RTDX_readNB(&control_channel, &volume, sizeof(volume));
        }

        while (!RTDX_isInputEnabled(&A2D_channel)){
            RTDX_Poll();             /* poll comm channel for input */
        }

        /*
         * A2D: get digitized input (get signal from the host through
         * RTDX). If A2D_channel is enabled, read data from the host.
         */
        RTDX_read(&A2D_channel, input, size*sizeof(sample));

        /*
         * Vector Scale: Scale the input signal by the volume factor to
         * produce the output signal.
         */
        while(size--){
            *output++ = *input++ * volume;
        }
        size = BUFSIZE;
        input = inp_buffer;
        output = out_buffer;

        /*
         * D2A: produce analog output (send signal to the host through
         * RTDX). If D2A_channel is enabled, write data to the host.
         */
        RTDX_write(&D2A_channel, output, size*sizeof(sample));

        while(RTDX_writing){
            RTDX_Poll();            /* poll comm channel for output */
        }
    }
}
```

7.3 Reviewing the Signalprog Application

The source code for the Visual Basic signalprog.exe application is available in the signalfrm.frm file. Details about this application are provided in the signalprog.pdf Adobe Acrobat file. In this section, you examine a few of the routines and functions that are important for this example.

- **Test_ON**. This routine runs when you click the Test_ON button. It creates instances of the RTDX exported interface for the input channel (toDSP) and for the output channel (fromDsp). Then it opens and enables both of these channels. The channels in the signalprog.exe application are the same channels declared globally in the hostio.c source code.

  This routine also clears the graphs and starts the timer used to call the Transmit_Signal and Receive_Signal functions.

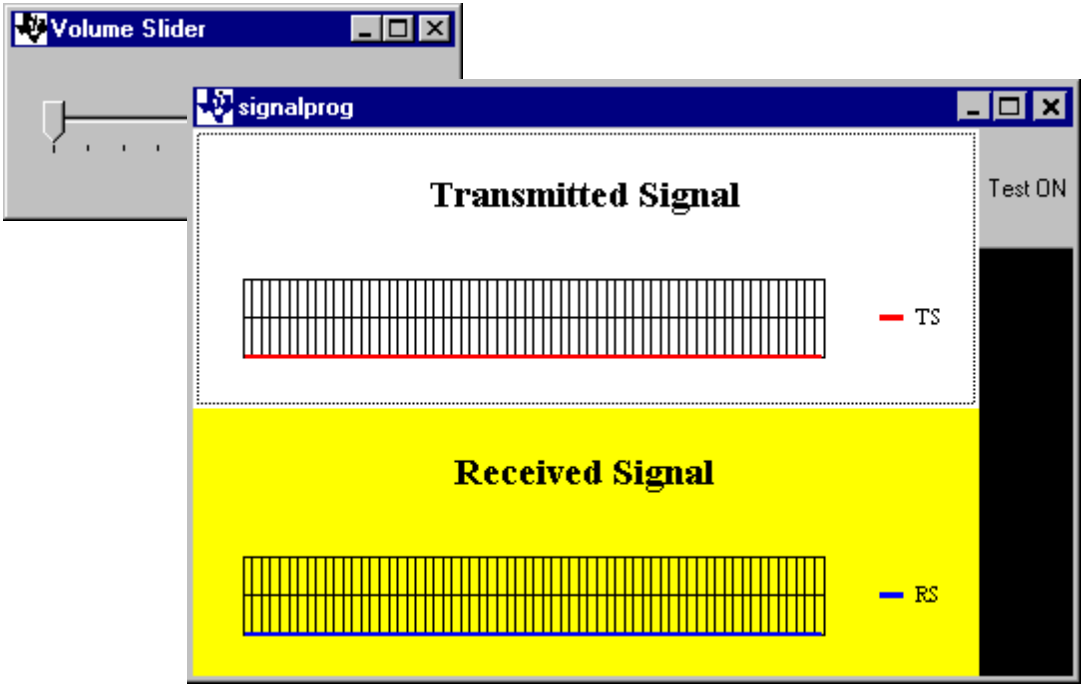
  These global declarations made earlier in the Visual Basic source code connected the READ_CHANNEL and WRITE_CHANNEL used in the Test_ON routine to the D2A_channel and A2D_channel used in hostio.c:

  ```
  ' Channel name constants
  Const READ_CHANNEL = "D2A_channel"
  Const WRITE_CHANNEL = "A2D_channel"
  ```

- **Test_OFF**. This routine disables, closes, and releases the RTDX objects created by Test_ON. It also disables the timer.
- **Transmit_Signal**. This function generates a sine wave signal and displays it in the Transmitted Signal graph. Then, the function attempts to transmit the signal to the target using the Write method of the toDSP channel.
- **Receive_signal**. This function uses the ReadSAI2 method of the fromDSP channel to read a signal from the target. It displays the signal in the Received Signal graph.
- **tmr_MethodDispatch_Timer**. This routine calls the Transmit_Signal and Receive_Signal functions. This routine is called at 1 millisecond intervals after the timer object is enabled by the Test_ON routine.

7.4 Running the Application

1) Choose Project→Build or click the (Incremental Build) toolbar button.
2) Choose File→Load Program. Select hostio.out and click Open.
3) Choose Tools→RTDX.
4) Click Configure in the RTDX area of the window. In the General Settings tab of the RTDX Properties dialog, select Continuous RTDX mode. Then, click OK.
5) Change RTDX Disable to RTDX Enable in the RTDX area. This changes the Configure button to Diagnostics.
6) Choose Tools→DSP/BIOS→Message Log. Right-click on the Message Log area and choose Property Page from the pop-up window. Select trace as the name of the log to monitor and click OK.
7) Choose Debug→Run or click the (Run) toolbar button.
8) Using Windows Explorer, run signalprog.exe and slider.exe. You see these two Visual Basic applications.

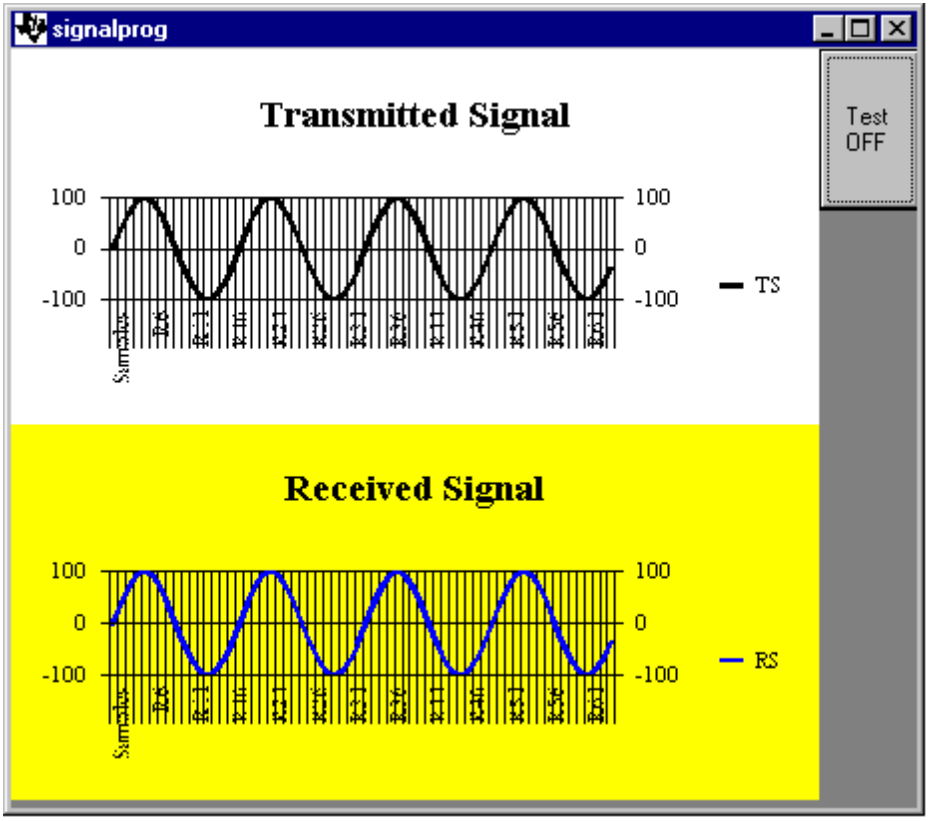


The slider.exe program must be started after RTDX is enabled and the program is running because it creates and opens the RTDX control channel when you run the program. If RTDX is not enabled at this point, slider.exe cannot open the channel.

The signalprog.exe program can be started at any point. It does not use RTDX until you click the Test On button.

9) Resize the signalprog window so that it is taller. This allows you to see the axis labels.

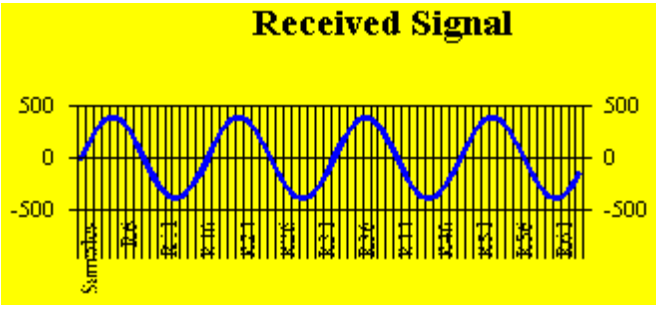
10) Click Test On in the signalprog window. This starts the input and output channels.



11) Slide the control in the Volume Slider window. This changes the volume of the output signal. Watch the amplitude of the Received Signal graph change. (Only the scale values to the left and right of the graph change. The graph changes scale automatically to accommodate the size of the sine wave and the size of the window.)

Note: Initial Setting of Volume Slider

The initial setting of the Volume Slider bar is not synchronized with the application. They are synchronized the first time you move the slider bar.



12) Close the Volume Slider application. This stops the input and output channels.

13) Click Test OFF in the signalprog window. This closes the control channel.

14) Click (Halt) or press Shift F5 to stop the program.

15) You now see the “hostio example started” message from the call to LOG_printf in the Message Log area. You did not see this message earlier because the entire program runs within the main function. DSP/BIOS communicates with the host PC within the idle loop. Until a program returns from main, it never enters the idle loop. Therefore, if you want to see the effects of DSP/BIOS calls at run-time, your program should perform its functions after returning from main. The modified version of hostio.c used in the next section shows this technique.

7.5 Modifying the Source Code to Use Host Channels and Pipes

Now you modify the example to use the host channels and pipes provided with DSP/BIOS. The modified example still tests your DSP algorithm in real time. Rather than generating a sine wave on the host, this time the data comes from a host file.

The HST module provides a more direct path toward implementing I/O with peripheral devices. The HST module uses the PIP module for host I/O. You can use the PIP module API with minimal modifications to the source code once the I/O devices and ISRs are ready for testing.

1) Copy *only* the following files from the c:\ti\c6000\tutorial\hostio2\ folder to your working folder. (Note that you should *not* copy all the files from the hostio2 folder. In particular, do not copy the hostio.cdb file.)

   - **hostio.c**. The source code has been modified to use the HST and PIP modules of the DSP/BIOS API instead of RTDX to transfer the input and output signals
   - **input.dat**. This file contains input data

2) Double-click on the hostio.c file in the Project View to see the source code in the right half of the Code Composer Studio window. The source code now contains the following differences from the source code used earlier in this chapter:

   - Added the following to the list of included header files:

     ```
     #include <hst.h>
     #include <pip.h>
     ```

   - Removed the BUFSIZE definition, the global declarations of inp_buffer and out_buffer, and the RTDX input and output channel declarations. This example retains the RTDX channel used to control the volume.
   - Moved the input and output functionality from a while loop in the main function to the A2DscaleD2A function.

```
/* ======== A2DscaleD2A ======== */
/* FUNCTION: Called from A2DscaleD2A_SWI to get digitized data
 *           from a host file through an HST input channel,
 *           scale the data by the volume factor, and send
 *           output data back to the host through an HST
 *           output channel.
 * PARAMETERS: Address of input and output HST channels.
 * RETURN VALUE: None. */
```

```
Void A2DscaleD2A(HST_Obj *inpChannel, HST_Obj *outChannel)
{
    PIP_Obj *inp_PIP;
    PIP_Obj *out_PIP;
    sample *input;
    sample *output;
    Uns size;

    inp_PIP = HST_getpipe(inpChannel);
    out_PIP = HST_getpipe(outChannel);

    if ((PIP_getReaderNumFrames(inp_PIP) <= 0) ||
        (PIP_getWriterNumFrames(out_PIP) <= 0)) {
       /* Software interrupt should not have been triggered! */
       error();
    }

    /* Read a new volume when the hosts send it */
    if (!RTDX_channelBusy(&control_channel))
       RTDX_readNB(&control_channel, &volume, sizeof(volume));

    /* A2D: get digitized input (get signal from the host
     * through HST). Obtain input frame and allocate output
     * frame from the host pipes. */

    PIP_get(inp_PIP);
    PIP_alloc(out_PIP);

    input = PIP_getReaderAddr(inp_PIP);
    output = PIP_getWriterAddr(out_PIP);
    size = PIP_getReaderSize(inp_PIP);

    /* Vector Scale: Scale the input signal by the volume
     * factor to produce the output signal. */
    while(size--){
        *output++ = *input++ * volume;
    }

    /* D2A: produce analog output (send signal to the host
     * through HST). Send output data to the host pipe and
     * free the frame from the input pipe. */
    PIP_put(out_PIP);
    PIP_free(inp_PIP);
}
```

The A2DscaleD2A function is called by the A2DscaleD2A_SWI object. You create this SWI object in the next section and make it call the A2DscaleD2A function.

The A2DscaleD2A_SWI object passes two HST objects to this function. This function then calls HST_getpipe to get the address of the internal PIP object used by each HST object.

Calls to PIP_getReaderNumFrames and PIP_getWriterNumFrames then determine whether there is at least one frame in the input pipe that is ready to be read and one frame in the output pipe that can be written to.

Using the same RTDX calls used in section 7.2, page 7-3, the function gets the volume setting from the RTDX control channel.

The call to PIP_get gets a full frame from the input pipe. The call to PIP_getReaderAddr gets a pointer to the beginning of the data in the input pipe frame and PIP_getReaderSize gets the number of words in the input pipe frame.

The call to PIP_alloc gets an empty frame from the output pipe. The call to PIP_getWriterAddr gets a pointer to the location to begin writing data to in the output pipe frame.

The function then multiplies the input signal by the volume and writes the results to the frame using the pointer provided by PIP_getWriterAddr.

The call to PIP_put puts the full frame into the output pipe. The call to PIP_free recycles the input frame so that it can be reused the next time this function runs.

- Added an error function, which writes an error message to the trace log and then puts the program in an infinite loop. This function runs if A2DscaleD2A runs when there are no frames of data available for processing.

7.6 More about Host Channels and Pipes

Each host channel uses a pipe internally. When you are using a host channel, your target program manages one end of the pipe and the Host Channel Control plug-in manages the other end of the pipe.

When you are ready to modify your program to use peripheral devices other than the host PC, you can retain the code that manages the target's end of the pipe and add code in functions that handle device I/O to manage the other end of the pipe.

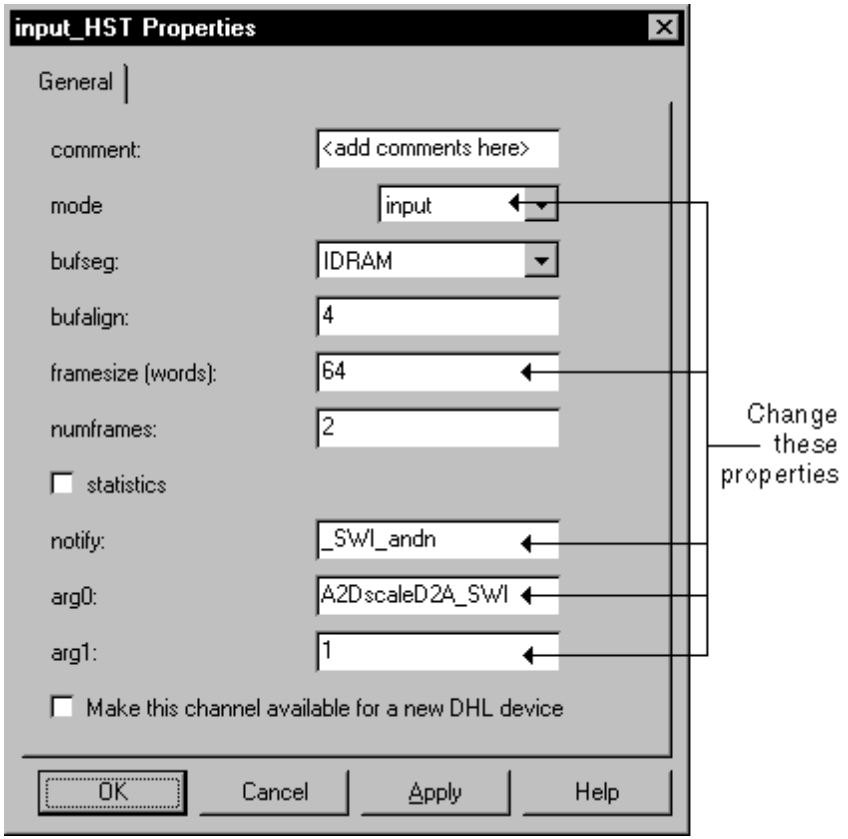
7.7 Adding Channels and an SWI to the Configuration File

The A2DscaleD2A function is called by an SWI object and uses two HST objects. You create these objects in this section. (The hostio.cdb file in the c:\ti\c6000\tutorial\hostio2\ folder already contains these objects.)

The A2DscaleD2A function also references two PIP objects, but these objects are created internally when you create the HST objects. The HST_getpipe function gets the address of the internal PIP object that corresponds to each HST object.

1) In the Project View, double-click on the HOSTIO.CDB file to open it.
2) Right-click on the HST manager and choose Insert HST.

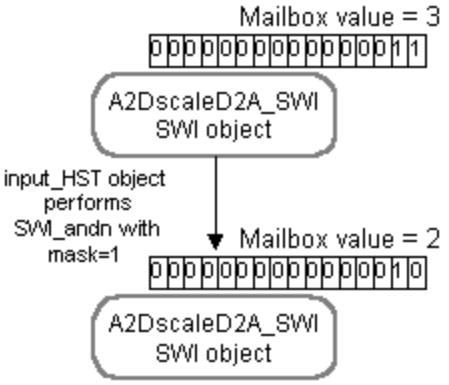
   Notice that there are HST objects called RTA_fromHost and RTA_toHost. These objects are used internally to update the DSP/BIOS controls.
3) Rename the new HST0 object to input_HST.
4) Right-click on the input_HST object and choose Properties from the pop-up menu. Set the following properties for this object and click OK.



- **mode**. This property determines which end of the pipe the target program manages, and which end the Host Channel Control plug-in manages. An input channel sends data from the host to the target. An output channel sends data from the target to the host.
- **framesize**. This property sets the size of each frame in the channel. Use 64 words—the same value as the BUFSIZE defined in section 7.2, page 7-3.
- **notify, arg0, arg1**. These properties specify the function to run when this input channel contains a full frame of data and the arguments to pass to that function. The SWI_andn function provides another way to manipulate a SWI object's mailbox.

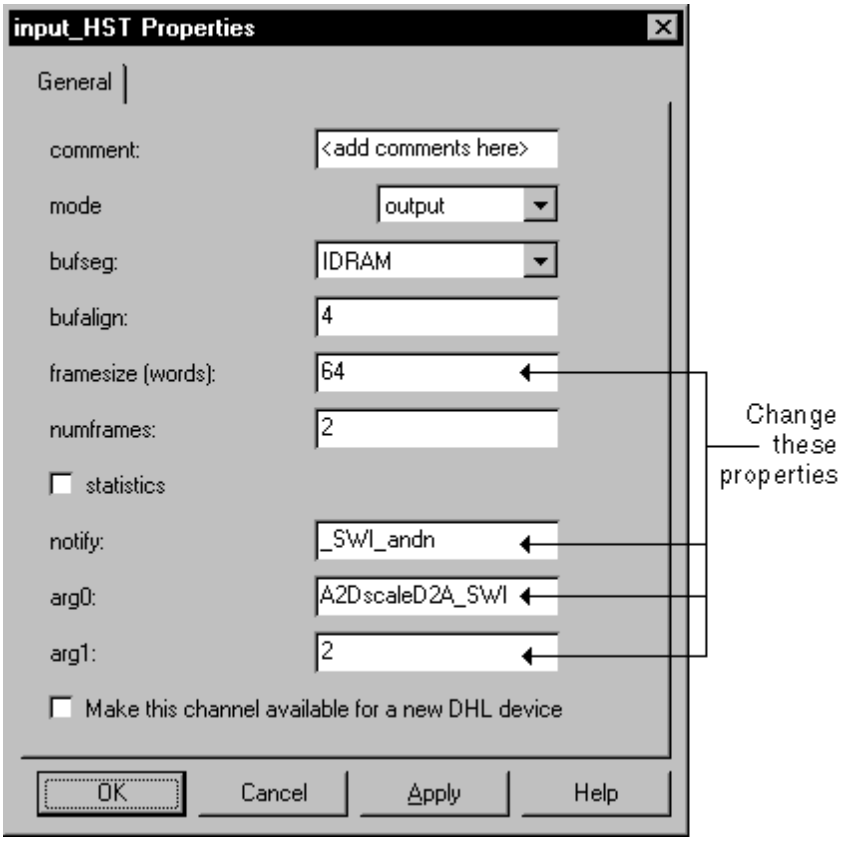
  In Chapter 5, you used the SWI_dec function to decrement the mailbox value and run the SWI object's function when the mailbox value reached zero.

  The SWI_andn function treats the mailbox value as a bitmask. It clears the bits specified by the second argument passed to the function. So, when this channel contains a full frame (because the target filled a frame), it calls SWI_andn for the A2DscaleD2A_SWI object and causes it to clear bit 1 of the mailbox.

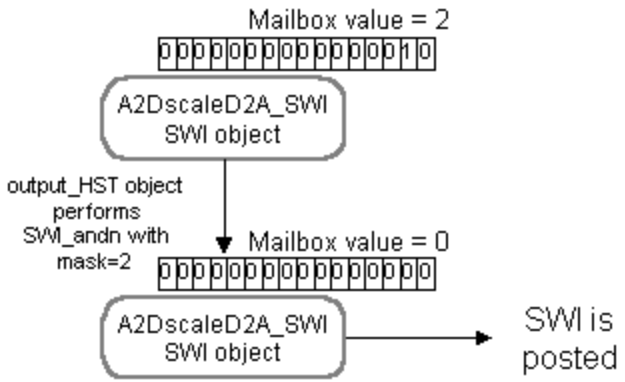


5) Insert another HST object and rename it output_HST.

6) Set the following properties for the output_HST object and click OK.



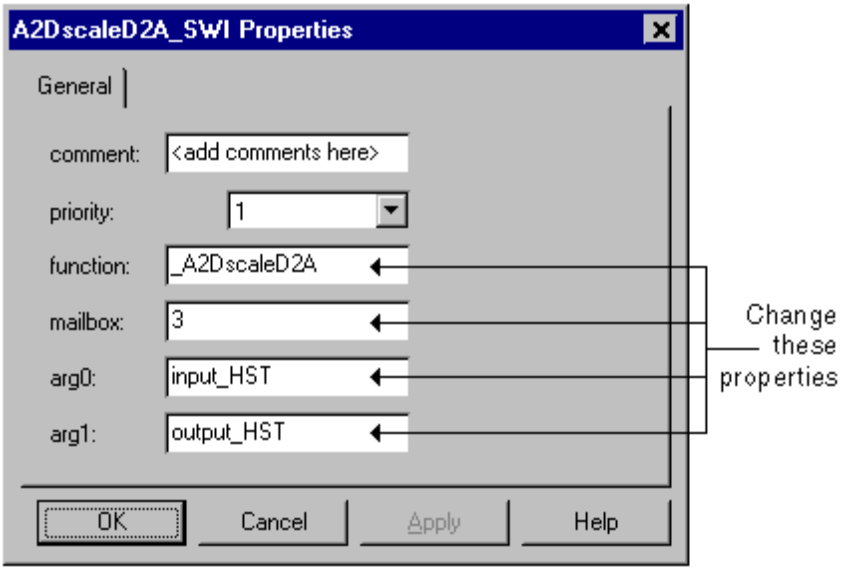
When this output channel contains an empty frame (because the target read and released a frame), it uses SWI_andn to clear the second bit of the mailbox.



7) Right-click on the SWI manager and choose Insert SWI.

8) Rename the new SWI0 object to A2DscaleD2A_SWI.

9) Set the following properties for A2DscaleD2A_SWI and click OK.

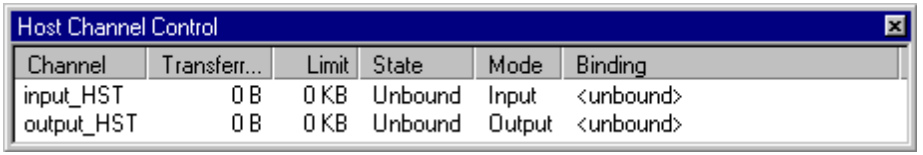


- **function**. This property causes the object to call the A2DscaleD2A function when this software interrupt is posted and runs.
- **mailbox**. This is the initial value of the mailbox for this object. The input_HST object clears the first bit of the mask and the output_HST object clears the second bit of the mask. When this object runs the A2DscaleD2A function, the mailbox value is reset to 3.
- **arg0, arg1**. The names of the two HST objects are passed to the A2DscaleD2A function.

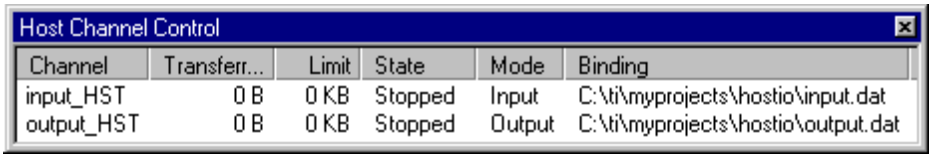
10) Choose File→Close. You are asked whether you want to save your changes to hostio.cdb. Click Yes. Saving the configuration also generates hostiocfg.cmd, hostiocfg.s62, and hostiocfg.h62.

7.8 Running the Modified Program

1) Choose Project→Rebuild All or click the (Rebuild All) toolbar button.
2) Choose File→Load Program. Select the program you just rebuilt, hostio.out, and click Open.
3) Choose Tools→DSP/BIOS→Host Channel Control. The Host Channel Control lists the HST objects and allows you to bind them to files on the host PC and to start and stop the channels.



4) Choose Debug→Run or click the (Run) toolbar button.
5) Right-click on the input_HST channel and choose Bind from the pop-up menu.
6) Select the input.dat file in your working folder and click Bind.
7) Right-click on the output_HST channel and choose Bind from the pop-up menu.
8) Type **`output.dat`** in the File Name box and click Bind.



9) Right-click on the input_HST channel and choose Start from the pop-up menu.
10) Right-click on the output_HST channel and choose Start from the pop-up menu. Notice that the Transferred column shows that data is being transferred.
11) When the data has been transferred, click (Halt) or press Shift F5 to stop the program.

7.9 Learning More

To learn more about the RTDX, HST, PIP, and SWI modules, see the online help or the *TMS320C6000 DSP/BIOS User's Guide* (which is provided as an Adobe Acrobat file).

Index

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W