

TMS320C6000

Code Composer Studio

Tutorial

Literature Number: SPRU301A
May 1999



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.

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 and the DSP/BIOS and RTDX APIs and plug-ins. 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 will introduce 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.

DSP/BIOS User's Guide (literature number SPRU303) 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

MS-DOS, Windows, and Windows NT are trademarks of Microsoft Corporation. Other Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft.

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, XDS, Code Composer Studio, Probe Point, Code Explorer, DSP/BIOS, RTDX, Online DSP Lab, BIOSuite, and SPOX.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

If You Need Assistance . . .**❑ World-Wide Web Sites**

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580	
TI Literature Response Center U.S.A.	(800) 477-8924	
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285	
U.S. Technical Training Organization	(972) 644-5580	
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324 Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323	
DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs		

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+49 8161 80 33 11	or +33 1 30 70 11 68	
English	+33 1 30 70 11 65		
Français	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline Fax:	+49 81 61 80 40 10		

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	
Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/		

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type "Go TIASP"	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
 Technical Documentation Services, MS 702
 P.O. Box 1443
 Houston, Texas 77251-1443

Email: dsph@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number.



Contents

1	Code Composer Studio Overview	1-1
	<i>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.</i>	
1.1	Code Composer Studio Development	1-2
1.2	Code Generation Tools	1-4
1.3	Code Composer Studio Integrated Development Environment	1-7
1.3.1	Program Code Editing Features	1-7
1.3.2	Application Building Features	1-8
1.3.3	Application Debugging Features	1-8
1.4	DSP/BIOS Plug-ins	1-9
1.4.1	DSP/BIOS Configuration	1-10
1.4.2	DSP/BIOS API Modules	1-11
1.5	Hardware Emulation and Real-Time Data Exchange	1-13
1.6	Third-Party Plug-ins	1-14
1.7	Code Composer Studio Files and Variables	1-15
1.7.1	Installation Folders	1-15
1.7.2	File Extensions	1-16
1.7.3	Environment Variables	1-17
1.7.4	Increasing DOS Environment Space	1-17
2	Developing a Simple Program	2-1
	<i>This chapter introduces Code Composer Studio and shows the basic process used to create, build, debug, and test programs.</i>	
2.1	Creating a New Project	2-2
2.2	Adding Files to a Project	2-3
2.3	Reviewing the Code	2-4
2.4	Building and Running the Program	2-6
2.5	Changing Program Options and Fixing Syntax Errors	2-8
2.6	Using Breakpoints and the Watch Window	2-10
2.7	Using the Watch Window with Structures	2-12
2.8	Profiling Code Execution Time	2-13
2.9	Things to Try	2-15
2.10	Learning More	2-15

3	Developing a DSP/BIOS Program	3-1
	<i>This chapter introduces DSP/BIOS and shows how to create, build, debug, and test programs that use DSP/BIOS.</i>	
3.1	Creating a Configuration File	3-2
3.2	Adding DSP/BIOS Files to a Project	3-4
3.3	Testing with Code Composer Studio	3-6
3.4	Profiling DSP/BIOS Code Execution Time	3-8
3.5	Things to Try	3-10
3.6	Learning More	3-10
4	Testing Algorithms and Data from a File	4-1
	<i>This chapter shows the process for creating and testing a simple algorithm and introduces additional Code Composer Studio features.</i>	
4.1	Opening and Examining the Project	4-2
4.2	Reviewing the Source Code	4-4
4.3	Adding a Probe Point for File I/O	4-6
4.4	Displaying Graphs	4-8
4.5	Animating the Program and Graphs	4-9
4.6	Adjusting the Gain	4-10
4.7	Viewing Out-of-Scope Variables	4-11
4.8	Using a GEL File	4-13
4.9	Adjusting and Profiling the Processing Load	4-14
4.10	Things to Try	4-16
4.11	Learning More	4-16
5	Debugging Program Behavior	5-1
	<i>This chapter introduces techniques for debugging a program and several DSP/BIOS plug-ins and modules.</i>	
5.1	Opening and Examining the Project	5-2
5.2	Reviewing the Source Code	5-3
5.3	Modifying the Configuration File	5-6
5.4	Viewing Task Execution with the Execution Graph	5-10
5.5	Changing and Viewing the Load	5-12
5.6	Analyzing Task Statistics	5-15
5.7	Adding Explicit STS Instrumentation	5-17
5.8	Viewing Explicit Instrumentation	5-18
5.9	Things to Try	5-20
5.10	Learning More	5-20
6	Analyzing Real-Time Behavior	6-1
	<i>This chapter introduces techniques for analyzing and correcting real-time program behavior.</i>	
6.1	Opening and Examining the Project	6-2
6.2	Modifying the Configuration File	6-3
6.3	Reviewing the Source Code Changes	6-5
6.4	Using the RTDX Control to Change the Load at Run Time	6-7
6.5	Modifying Software Interrupt Priorities	6-11

6.6	Things to Try.6-12
6.7	Learning More6-12
7	Connecting to I/O Devices7-1
	<i>This chapter introduces RTDX and DSP/BIOS techniques for implementing I/O.</i>	
7.1	Opening and Examining the Project.7-2
7.2	Reviewing the C Source Code7-3
7.3	Reviewing the Signalprog Application7-6
7.4	Running the Application7-7
7.5	Modifying the Source Code to Use Host Channels and Pipes7-10
7.6	More about Host Channels and Pipes7-12
7.7	Adding Channels and an SWI to the Configuration File7-13
7.8	Running the Modified Program.7-17
7.9	Learning More7-17



Code Composer Studio Overview

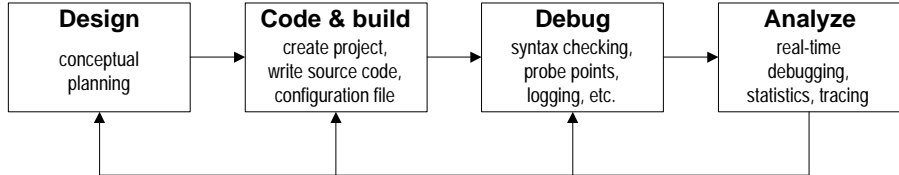
This chapter provides an overview of the Code Composer Studio software development process, the components of Code Composer Studio, and the files and variables used by Code Composer Studio.

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	1-2
1.2 Code Generation Tools	1-4
1.3 Code Composer Studio Integrated Development Environment . . .	1-7
1.4 DSP/BIOS Plug-ins	1-9
1.5 Hardware Emulation and Real-Time Data Exchange	1-13
1.6 Third-Party Plug-ins	1-14
1.7 Code Composer Studio Files and Variables	1-15

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:



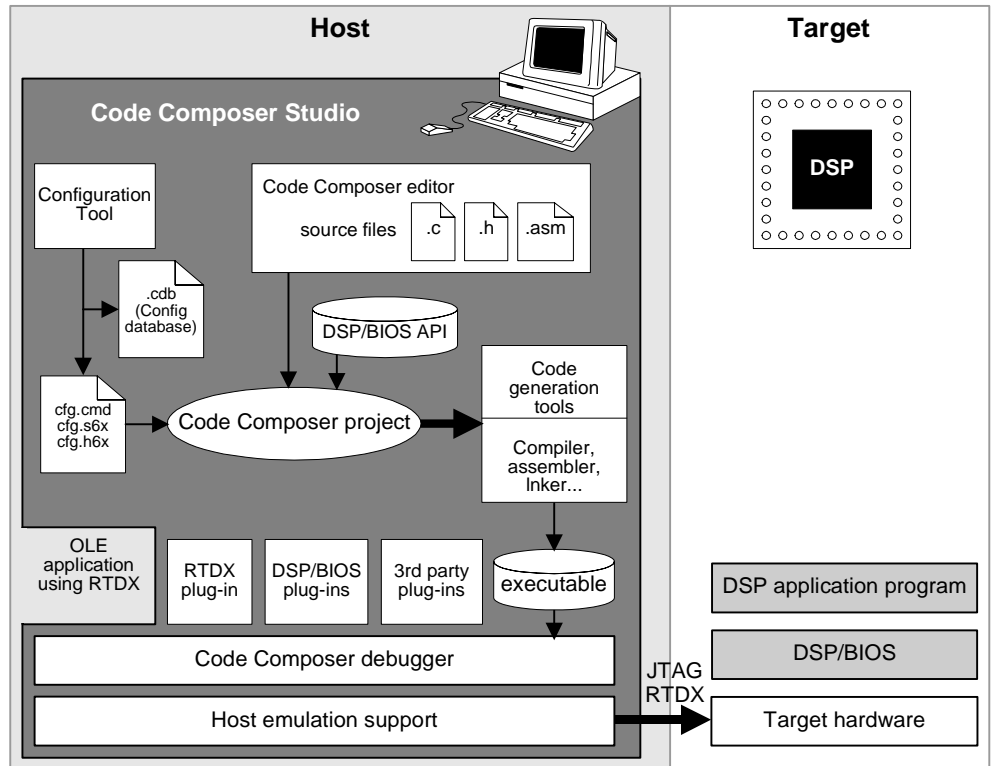
In order to use this tutorial, you should have already done the following:

- 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

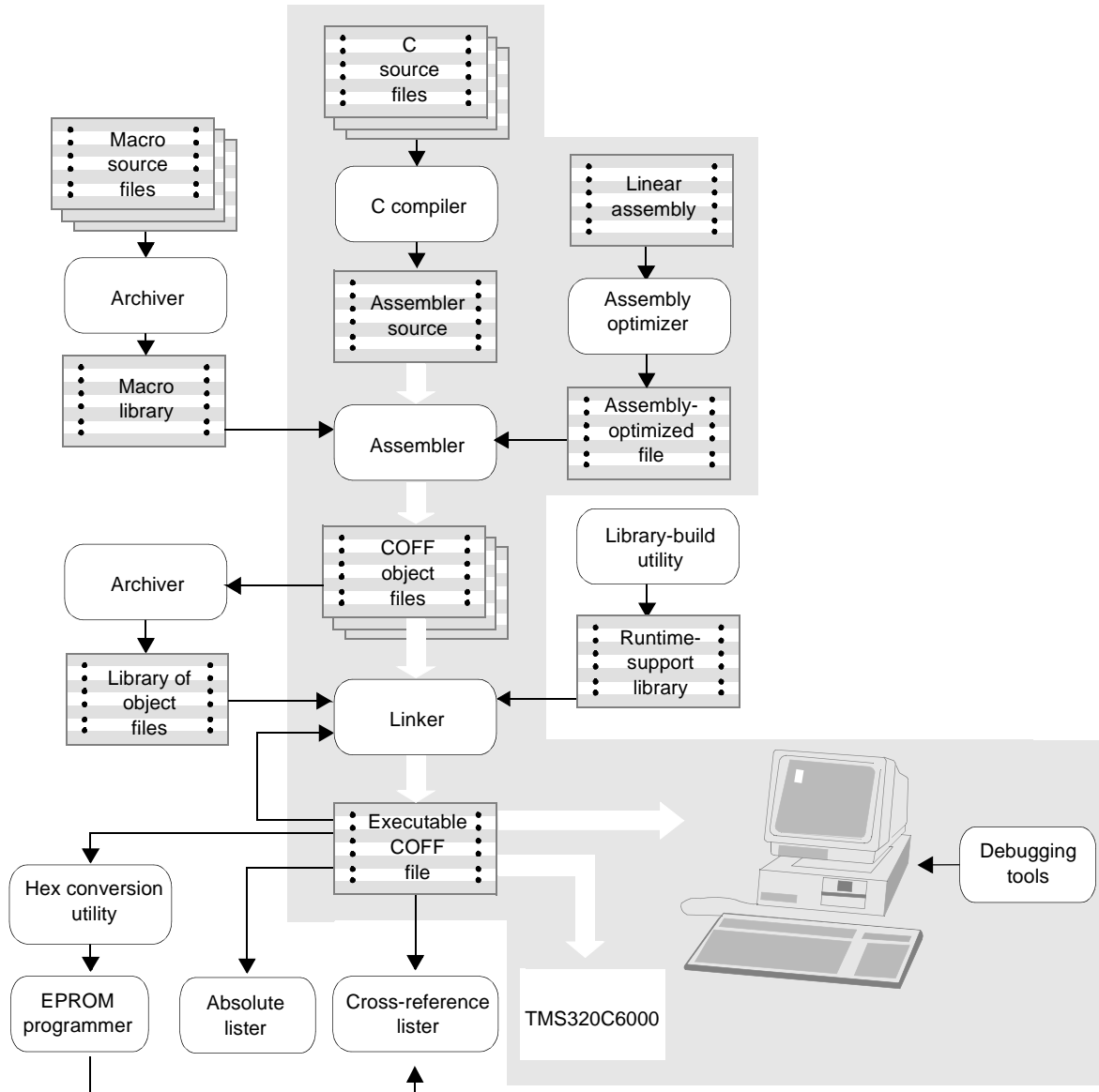
These components work together as shown here:



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*.
- ❑ 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 runtime-support library. See the *TMS320C6000 Optimizing C Compiler User's Guide* for details.
- ❑ The **runtime-support libraries** contain the ANSI standard runtime-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 produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. See the *TMS320C6000 Assembly Language Tools User's Guide* for details.

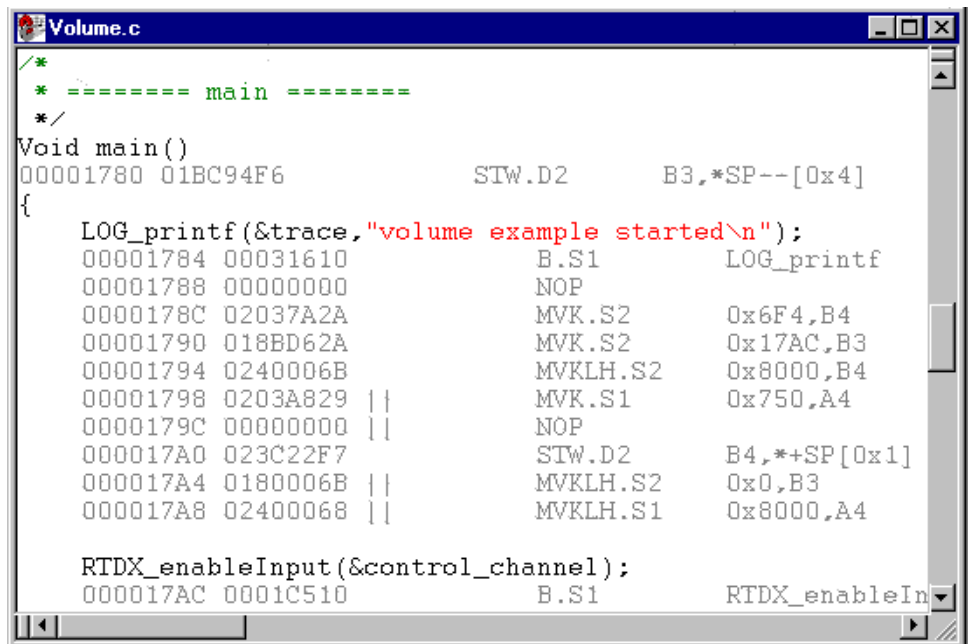
- ❑ The **absolute lister** accepts linked object files as input and creates .abs files as output. You assemble the .abs files to produce a listing that contains absolute addresses rather than relative addresses. Without the absolute lister, producing such a listing would be tedious and require many 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.



```

Volume.c
/*
 * ----- main -----
 */
Void main()
00001780 01BC94F6          STW.D2      B3,*SP--[0x4]
{
    LOG_printf(&trace,"volume example started\n");
00001784 00031610          B.S1      LOG_printf
00001788 00000000          NOP
0000178C 02037A2A          MVK.S2     0x6F4,B4
00001790 018BD62A          MVK.S2     0x17AC,B3
00001794 0240006B          MVKLH.S2  0x8000,B4
00001798 0203A829 ||          MVK.S1     0x750,A4
0000179C 00000000 ||          NOP
000017A0 023C22F7          STW.D2     B4,*+SP[0x1]
000017A4 0180006B ||          MVKLH.S2  0x0,B3
000017A8 02400068 ||          MVKLH.S1  0x8000,A4

    RTDX_enableInput(&control_channel);
000017AC 0001C510          B.S1      RTDX_enableIn

```

The integrated editor provides support for the following activities:

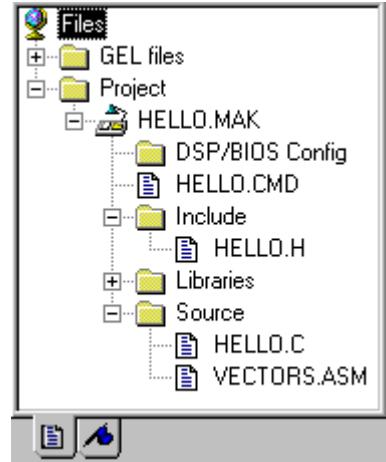
- Highlighting of keywords, comments, and strings in color
- Marking C blocks in parentheses and braces, finding matching or next parenthesis or brace
- Increasing and decreasing indentation level, customizable tab stops
- Finding and replacing in one or more files, find next and previous, quick search
- 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:

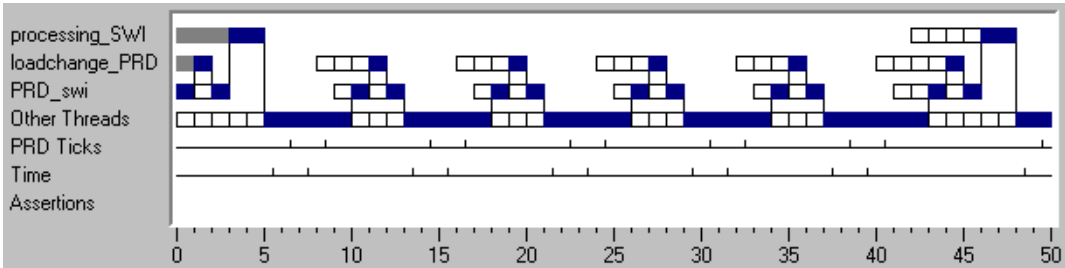
- Setting breakpoints with a number of stepping options
- Automatically updating windows at breakpoints
- Watching variables
- Viewing and editing memory and registers
- Viewing the call stack
- Using Probe Point tools to stream data to and from the target and to gather memory snapshots
- Graphing signals 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, e.g., a hardware ISR, a software interrupt, an idle function, or a periodic function.)



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.

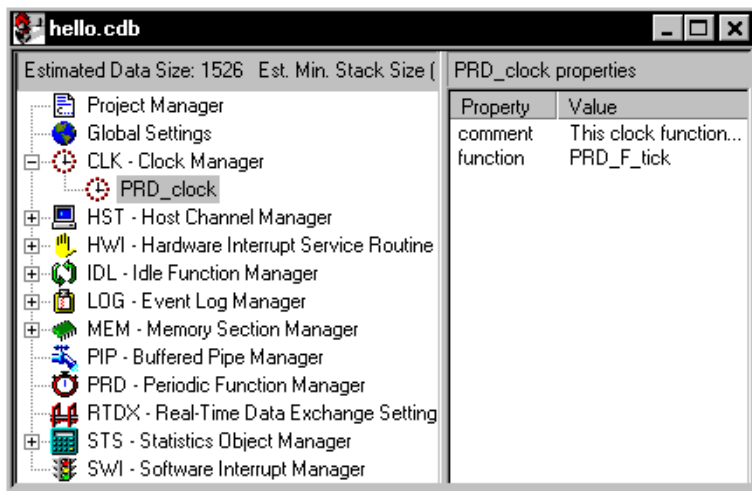
1.4.1 DSP/BIOS Configuration

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 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 codes shown here.

- ❑ **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.
- ❑ **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.
- ❑ **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.
- ❑ **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.
- ❑ **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.
- ❑ **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.
- ❑ **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

For details about each module, see the online help or the *TMS320C6000 DSP/BIOS User's 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.

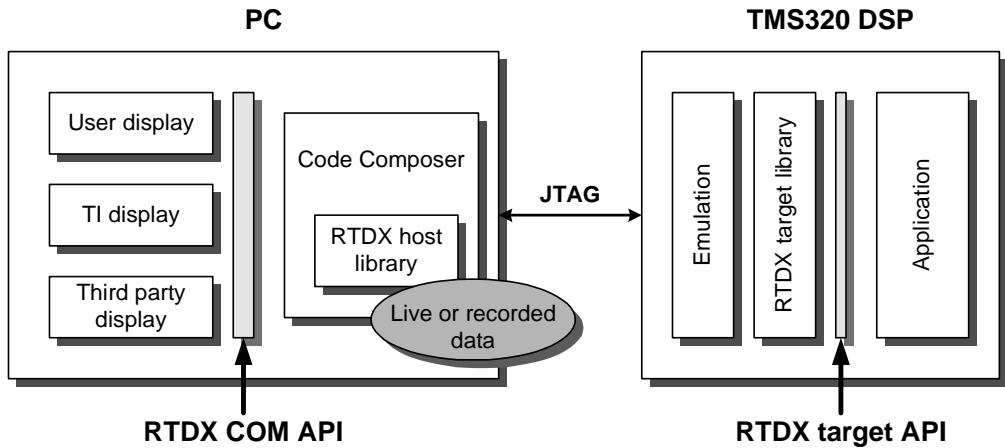
The on-chip emulation hardware provides a variety of capabilities:

- 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.

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

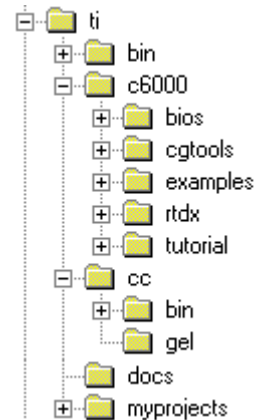
The following sections provide an overview of the folders that contain the Code Composer Studio files, the types of files you use, and the environment variables used by Code Composer Studio.

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
- 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 <i>TMS320C6000 Assembly Language Tools User's Guide</i> 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 <i>TMS320C6000 Optimizing C Compiler User's Guide</i> for details.
PATH	A list of folders is added to your PATH definition. The default is to add the <code>c:\ti\c6000\cgtools\bin</code> and <code>c:\windows\ti\plugins\bios</code> 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
```



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.

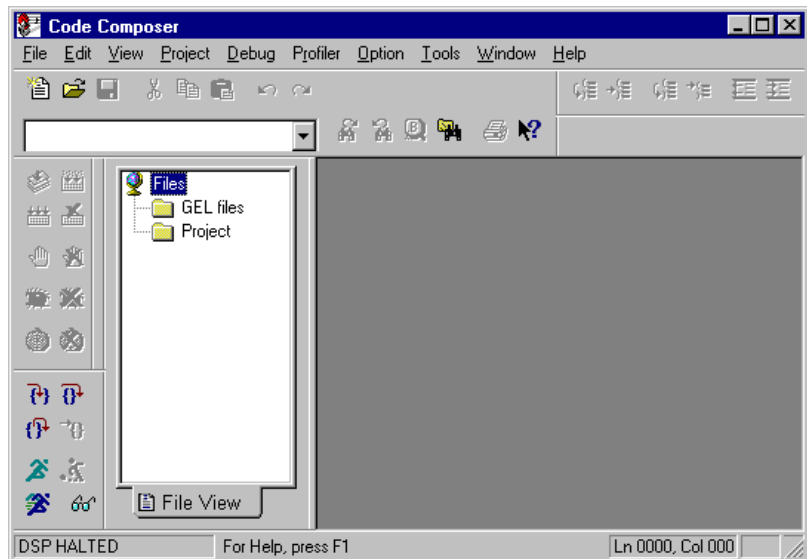
In order to use this tutorial, you should have already installed Code Composer Studio according to the installation instructions. It is recommended that you use Code Composer Studio with a target board rather than with the simulator. If you have Code Composer and the Code Generation Tools, but not Code 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	2-2
2.2 Adding Files to a Project	2-3
2.3 Reviewing the Code	2-4
2.4 Building and Running the Program	2-6
2.5 Changing Program Options and Fixing Syntax Errors	2-8
2.6 Using Breakpoints and the Watch Window	2-10
2.7 Using the Watch Window with Structures	2-12
2.8 Profiling Code Execution Time	2-13
2.9 Things to Try	2-15
2.10 Learning More	2-15

2.1 Creating a New Project

You begin by creating a project with Code Composer Studio and adding source code files and libraries to that project. In this example, you use 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→Code Composer Studio`. (Or, double-click the Code Composer Studio icon on your desktop.)



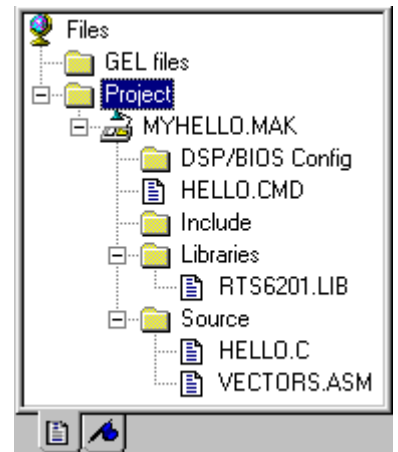
Note: Code Composer Studio Setup

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. Type `myhello` as the File name 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 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 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 Library Files (*.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.

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);
#endif
}
```

```
    /* open a file on the host and read char array */
    fopen("file.txt", "r");
    fseek(fp, 0L, SEEK_SET);
    readSize = fread(fileStr, sizeof(char), BUFSIZE, fp);
    printf("Read a %d byte char array: %s \n", readSize, fileStr);
    fclose(fp);
#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), 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 you can start Code Composer Studio the first time, but a subsequent time you get 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.


- 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.

You see the hello world! message in the Stdout tab.



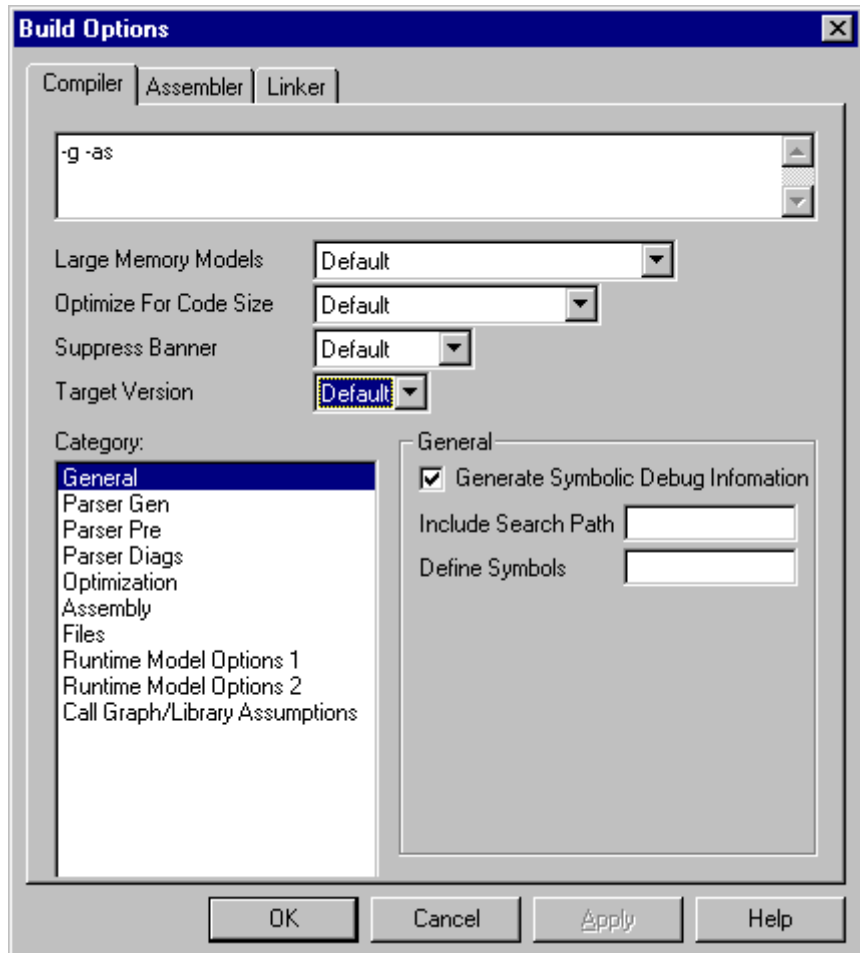
- 5) Click  (Halt) or press Shift F5 to stop the program.


2.5 Changing Program Options and Fixing Syntax Errors

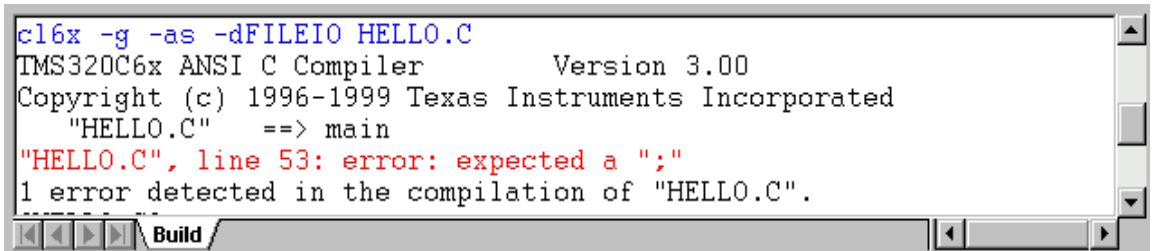
In the previous section, the portion of the program enclosed by the preprocessor commands (`#ifdef` and `#endif`) did not run because `FILEIO` was undefined. In this section, you set a preprocessor option with Code Composer Studio. You also find and correct a syntax error.

- 1) Choose Project→Options.
- 2) In the Compiler tab of the Build Options window, 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.



```

c16x -g -as -dFILEIO HELLO.C
TMS320C6x ANSI C Compiler      Version 3.00
Copyright (c) 1996-1999 Texas Instruments Incorporated
"HELLO.C" ==> main
"HELLO.C", line 53: error: expected a ";"
1 error detected in the compilation of "HELLO.C".


```

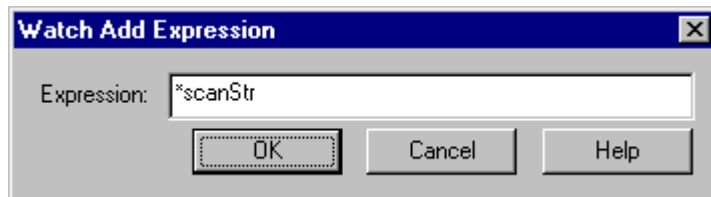
- 7) Double-click on the red text that describes the location of the syntax error. Notice that the hello.c source file opens, and your cursor is on the line that says `fileStr[i] = 0;`.
- 8) Fix the syntax error in the line above the cursor location. (The semicolon is missing.) Notice that an asterisk (*) appears next to the file name 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

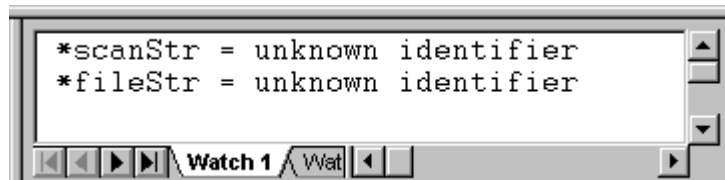
When you are developing and testing programs, you often need to check the value of a variable during program execution. In this section, you use breakpoints and the Watch Window to view such values. You also use the step commands after reaching the breakpoint.

- 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(fpPtr, "%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.

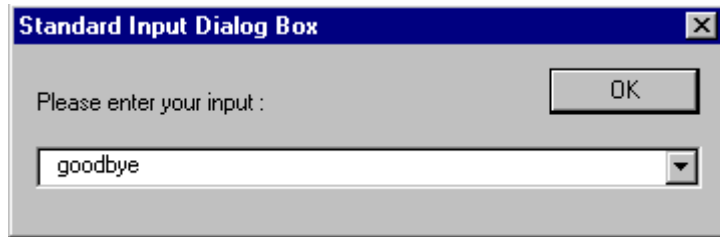


- 8) Choose Insert New Expression from the pop-up list again. This time, type `*fileStr` as the Expression and click OK. Both of the expressions you typed are listed in the Watch Window. Currently, the expressions are undefined because the program is not running the main function within which these variables are 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.



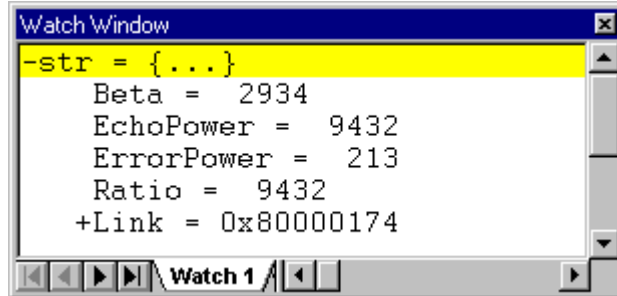
After you type an input string, the program runs and stops at the breakpoint. The next line to be executed is highlighted in yellow.


- 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 that a structure of type `PARAMS` 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) Click  (Halt) or press Shift F5 to stop the program.
- 7) Select the `str` variable in the Watch Window. Right-click in the Watch Window and choose Remove Current Expression from the pop-up list.
- 8) Choose Debug→Breakpoints. In the Breakpoints tab, click Delete All and then click OK.

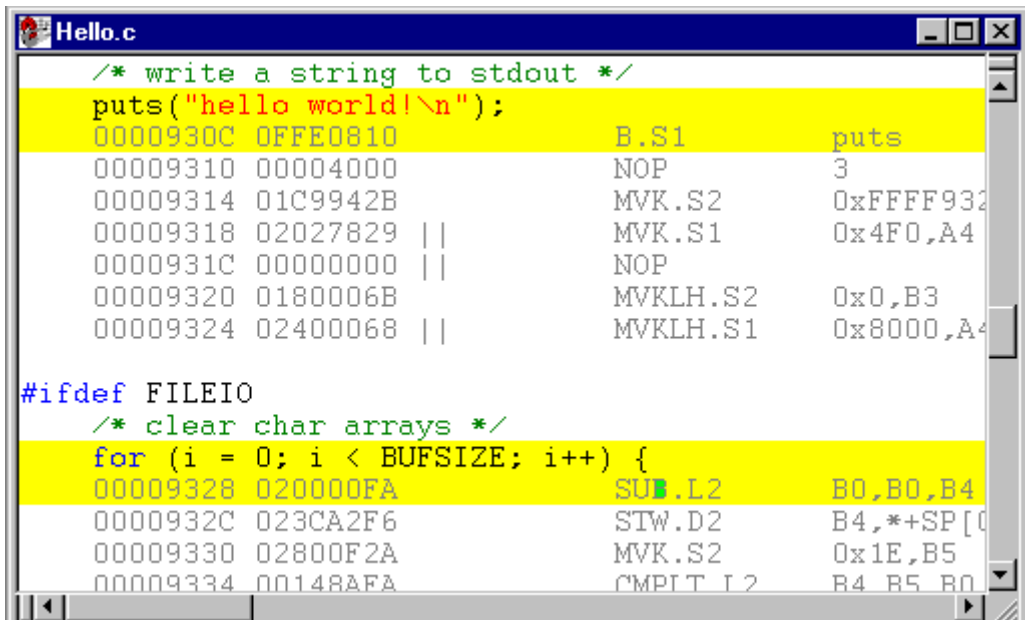
2.8 Profiling Code Execution Time

In this section, you use the profiling features of Code Composer Studio to gather statistics about the execution of the standard `puts()` function. In Chapter 3, page 3–10, 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.)



```

Hello.c
/* write a string to stdout */
puts("hello world!\n");
0000930C 0FFE0810          B.S1          puts
00009310 00004000          NOP          3
00009314 01C9942B          MVK.S2       0xFFFF932
00009318 02027829 ||          MVK.S1       0x4F0,A4
0000931C 00000000 ||          NOP
00009320 0180006B          MVKLH.S2     0x0,B3
00009324 02400068 ||          MVKLH.S1     0x8000,A4

#ifdef FILEIO
/* clear char arrays */
for (i = 0; i < BUFSIZE; i++) {
00009328 020000FA          SUB.L2       B0,B0,B4
0000932C 023CA2F6          STW.D2       B4,*+SP[0
00009330 02800F2A          MVK.S2       0x1E,B5
00009334 00148AFA          CMPIT.L2     B4 B5 B0

```


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

Location	Count	Average	Total	Maximum	Minimum
HELLO.C line 53	0	0.0	0	0	0
HELLO.C line 57	0	0.0	0	0	0


Note: Line Numbers May Vary

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

- 11) Click the  (Run) toolbar button or press F5 to run the program. Type an input string in the prompt window.
- 12) Notice the number of cycles shown for the second profile-point. It should be about 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 53	1	1407.0	1407	1407	1407
HELLO.C line 57	1	1710.0	1710	1710	1710

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

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

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.)

- 14) Before proceeding to the next chapter (after completing section 2.9), 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 Debug→Breakpoints. Select Delete All and click OK.
 - 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. 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).



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.

Topic	Page
3.1 Creating a Configuration File	3-2
3.2 Adding DSP/BIOS Files to a Project	3-4
3.3 Testing with Code Composer Studio	3-6
3.4 Profiling DSP/BIOS Code Execution Time	3-8
3.5 Things to Try	3-10
3.6 Learning More	3-10

3.1 Creating a Configuration File

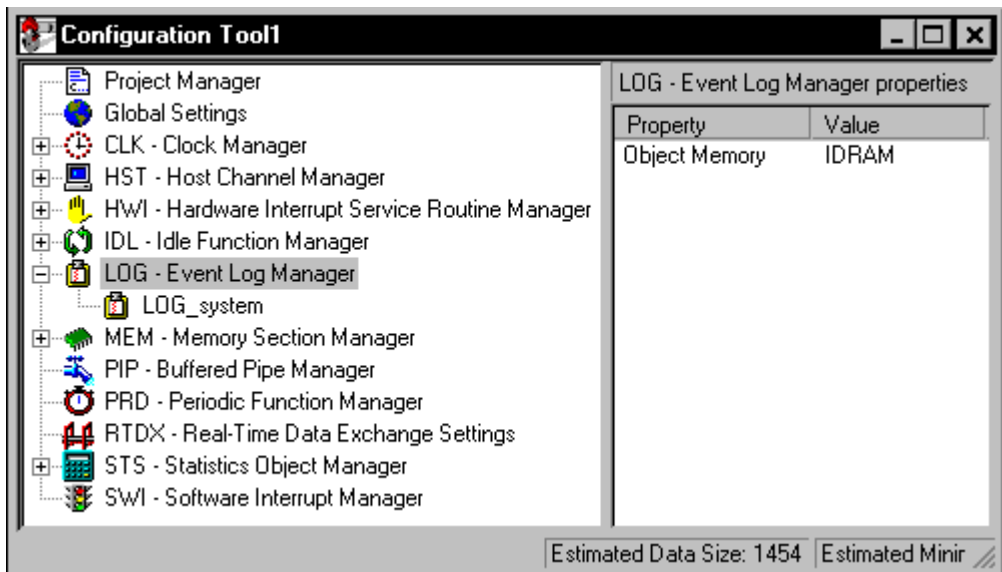
Another way to implement the hello world program is to use the LOG module provided with the DSP/BIOS API. You can use the DSP/BIOS API to provide basic run-time services within your embedded programs. This API is optimized for use on real-time DSPs. A program can use one or more of the DSP/BIOS modules as desired.

In this chapter, you modify the files you used in Chapter 2 to use the DSP/BIOS API. (If you skipped Chapter 2, create a folder called hello1 in the c:\ti\myprojects folder. Copy all files from the c:\ti\c6000\tutorial\hello1 folder to this new folder.)

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, create a folder within the myprojects folder in the location where you installed.)
- 2) Choose File→New→DSP/BIOS Configuration.
- 3) Select the template for your DSP board and click OK.

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 the 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.



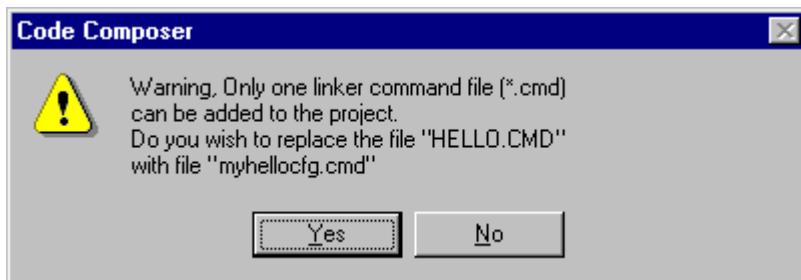
- 6) Choose File→Save. Move to your working folder (usually c:\ti\myprojects\hello1) and save the configuration as myhello.cdb. Saving this configuration actually creates the following files:

- **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

- 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.
- 2) The output file name must match the .cdb file name (myhello.out and myhello.cdb). Go to Project→Options and choose the Linker tab. In the Output Filename field, type myhello.out.
- 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 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:

- 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.
- The code then declares the `LOG` object you created in the configuration file.
- Within the main function, this example calls `LOG_printf` and passes it the address of the `LOG` object (`&trace`) and the `hello world!` message.
- 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`. 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→Reload Program.
- 2) Choose Debug→Go Main.
- 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, then right-click on the RTA Control Panel and choose Property Page.)
- 6) Choose Debug→Run or press F5.

The hello world! message appears in the Message Log area.



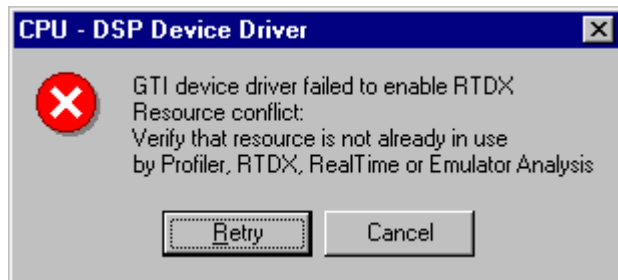
```
0 hello world!
```

- 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 to learn more about the idle loop.
- 8) Close the Message Log by right-clicking and selecting Close. This is necessary because in the next section we will be profiling.

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

On some DSP targets (for example, the TMS320C6201) you cannot use both profiling and RTDX at the same time. Close any tool that uses RTDX, such as the Message Log or any other DSP/BIOS plug-ins, before using profiling. To ensure that RTDX is disabled, especially after using DSP/BIOS plug-ins, go to Tools→RTDX to launch the RTDX plug-in. Select RTDX disable from the pull-down list, then right-click and select Hide. Conversely, after profiling, free the profiler resources before using RTDX, as described in section 2.8, page 2–15.



The following error message results when you attempt simultaneous use:




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) Click the  (Run) toolbar button or press F5 to run the program.
- 10) 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 31	1	36.0	36	36	36
HELLO.C line 27	1	6725.0	6725	6725	6725

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 36 instruction cycles compared to over 1700 for puts(). You can leave calls to LOG_printf() in your code for system status monitoring with very little impact on code execution.

- 11) Click  (Halt) or press Shift F5 to stop the program.
- 12) Before proceeding to the next chapter (after completing section 3.5), 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 Debug→Breakpoints. Select Delete All and click OK.
 - Go to Profiler→Profile-points. Select Delete All and click OK.
 - Go to the View menu and uncheck Mixed Source/ASM.

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 or software interrupt triggered to handle the application's real-time tasks. 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 the differences in memory sections and sizes when using stdio.h calls and DSP/BIOS.

3.6 Learning More

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

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.

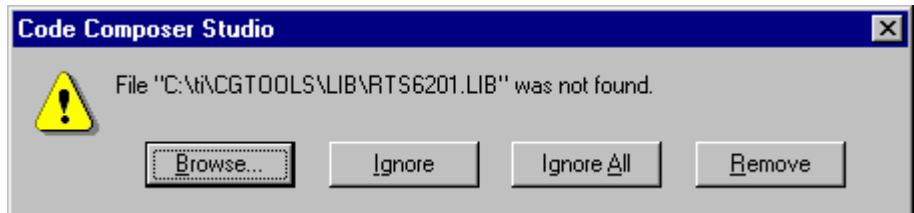
You create and test a simple algorithm using data stored in a file on your PC. You also use Probe Points, graphs, animation, and GEL files with Code Composer Studio.

Topic	Page
4.1 Opening and Examining the Project	4-2
4.2 Reviewing the Source Code	4-4
4.3 Adding a Probe Point for File I/O	4-6
4.4 Displaying Graphs	4-8
4.5 Animating the Program and Graphs	4-9
4.6 Adjusting the Gain	4-10
4.7 Viewing Out-of-Scope Variables	4-11
4.8 Using a GEL File	4-13
4.9 Adjusting and Profiling the Processing Load	4-14
4.10 Things to Try	4-16
4.11 Learning More	4-16

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.

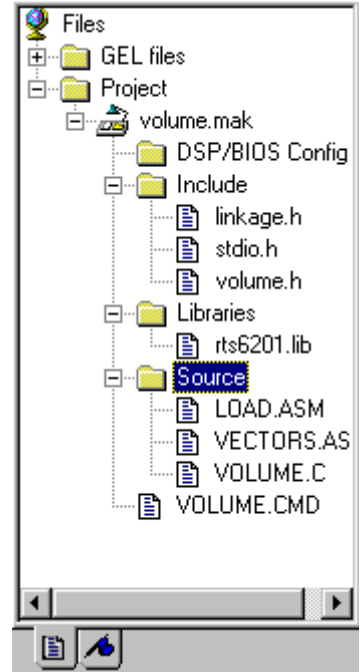
- 1) If you installed Code Composer Studio in c:\ti, create a folder called volume1 in the c:\ti\myprojects folder. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)
- 2) Copy all files from the c:\ti\c6000\tutorial\volume1 folder to this new folder.
- 3) If Code Composer Studio is not already running, from the Windows Start menu, choose Programs→Code Composer Studio 'C6000→Code Composer Studio.
- 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.



- 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 $(9 * \text{argument}) + 18$ 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. You also open a run-time graphical view of your input and output data.


Probe Points are a useful tool for algorithm development. You can use them to send data from a file on the PC to a buffer on the target that is acted upon by the algorithm. Chapter 7 shows two other ways to manage input and output streams.

You can also use Probe Points to write data from memory to a file or to cause a window to be updated at a particular point in program execution.

- 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.

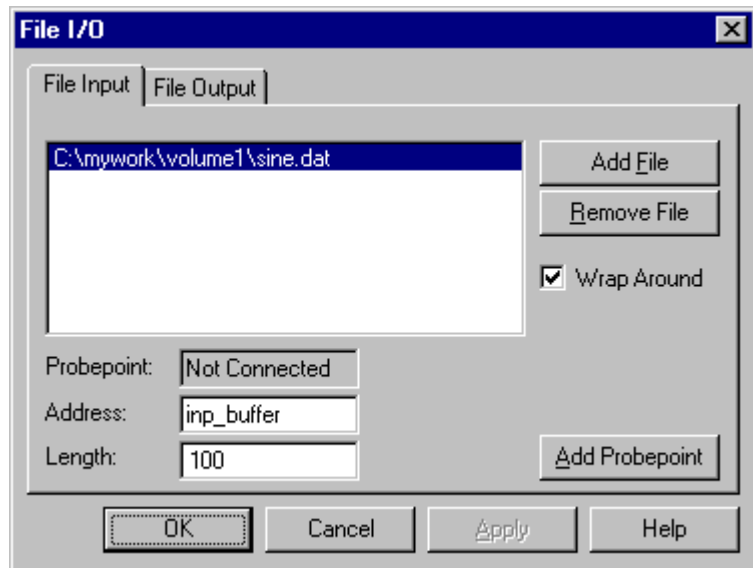
Notice that you can select the format of the data in the Files of Type box. The sine.dat file contains hex values for a sine waveform.

- 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.



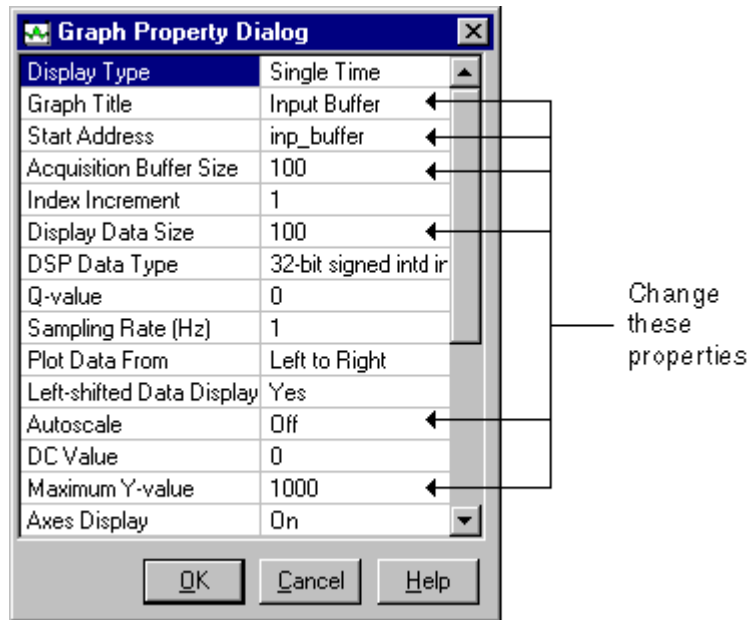
- The Address field specifies where the data from the file is to be placed. The `inp_buffer` is declared in `volume.c` as an integer array of `BUFSIZE`.
 - The Length field specifies how many samples from the data file are read each time the Probe Point is reached. You use 100 because that is the value 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 Probe Point. The Probe Points tab of the Break/Probe/Profile Points dialog appears.
- 12) Highlight the Probe Point you created in step 5.
- 13) In the Connect To field, click the down arrow and select the `sine.dat` file from the list.
- 14) Click Replace. The fields change 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 way to graph signals being processed by your program. 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.



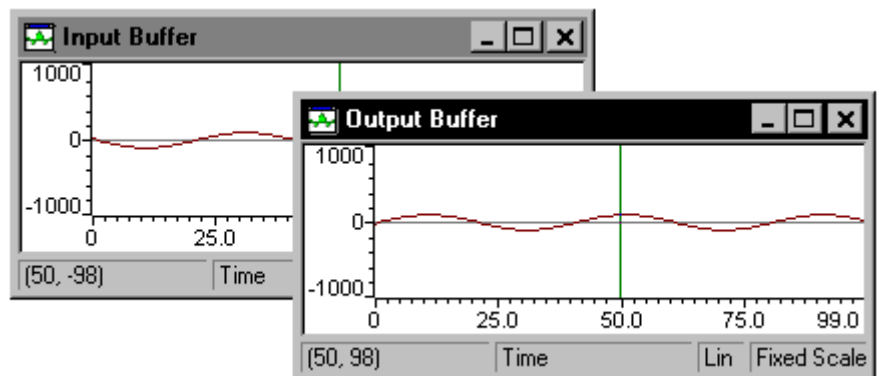
- 3) Click OK. A graph window for the Input Buffer appears.
- 4) Right-click on the Input Buffer window and choose Clear Display from the pop-up menu.
- 5) Choose View→Graph→Time/Frequency again.
- 6) This time, change the Graph Title to Output Buffer and the Start Address to `out_buffer`. All the other 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

Code Composer Studio can display a wide variety of graph types. In this example, you view a signal plotted against time.

- 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 to indicate that both a breakpoint and a Probe Point are set on this line.
- 3) Arrange the windows so that you can see both graphs.
- 4) Click the  (Animate) toolbar button or press F12 to run the program.

Each time the program reaches the Probe Point, Code Composer Studio updates the Probe Point by getting 100 values from the sine.dat file and writing them to the inp_buffer address. The graph windows are automatically updated when the program reaches a breakpoint. Then, the program runs to the next breakpoint. The graph windows show the input signal and the output signal. At this point, the signals are identical.



Notice that each buffer contains 2.5 sine waves and the signs are reversed in these graphs. This is 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.

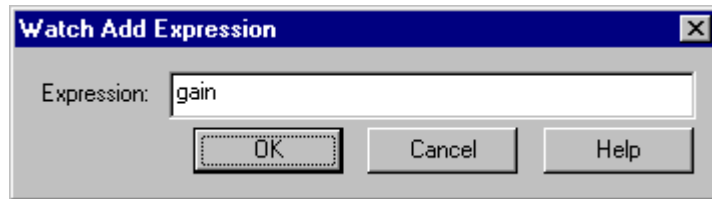
4.6 Adjusting the Gain

Recall from section 4.2 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;
```

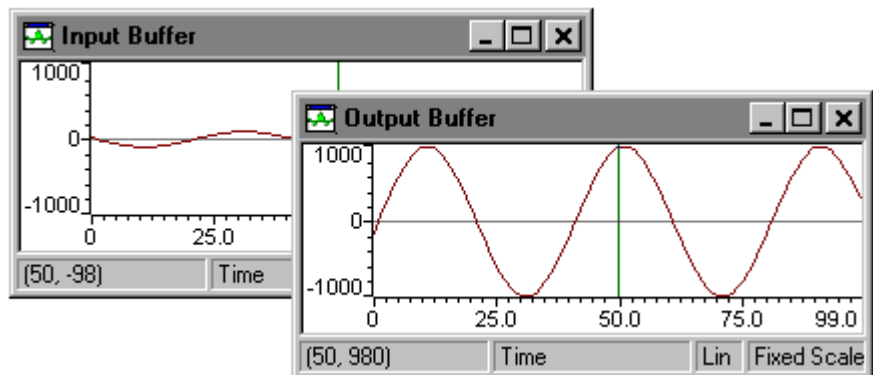
This statement multiplies a value in `inp_buffer` by the gain and places it in the corresponding location in the `out_buffer`. The gain is initially set to `MINGAIN`, which is defined as 1 in `volume.h`. To modify the output, you need to change gain. One way to do this is to use a watch variable.

- 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.

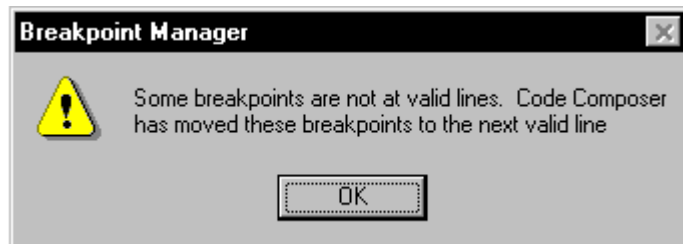
- 4) If you have halted the program, click the  (Animate) toolbar button to restart the program.
- 5) Double-click on gain in the Watch window area.
- 6) In the Edit Variable window, change the gain to 10 and click OK.
- 7) Notice that the amplitude of the signal in the Output Buffer graph changes to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



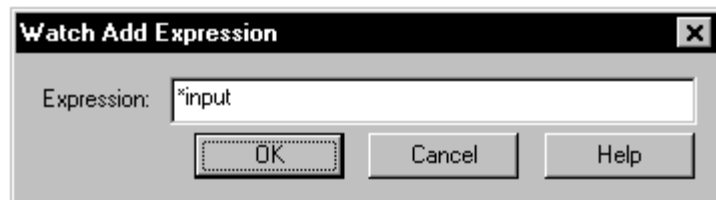
4.7 Viewing Out-of-Scope Variables

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.

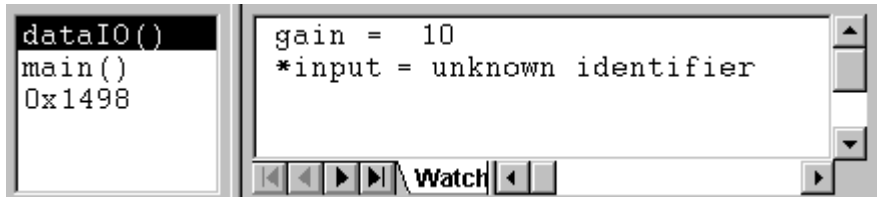
- 1) Click  (Halt) or press Shift F5 to stop the program.
- 2) Review the volume.c program within Code Composer Studio or by looking at section 4.2). Notice that *input is defined in both the main and processing functions. However, it is not defined within the dataIO function.
- 3) In the Volume.c window, 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.
- 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.



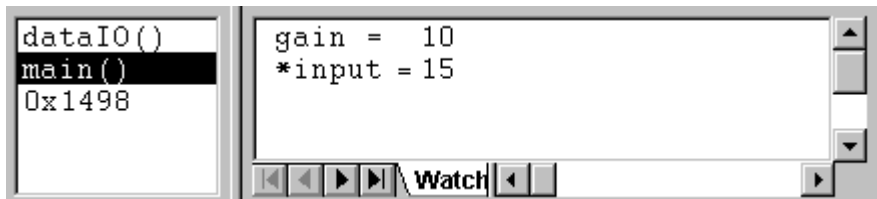
- 6) Click OK.
- 7) Press F5 until the program stops at the breakpoint inside the dataIO function (instead of at the breakpoint on the call to the dataIO function).
- 8) Right-click on the Watch window area and choose Insert New Expression from the pop-up list.
- 9) Type *input as the Expression and 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 actual value varies depending on your current location within the `sine.dat` file.)

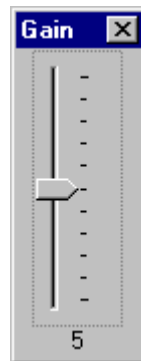


- 13) You can also click on the address at the bottom of the call stack, to see that `gain` is defined globally, but `*input` is not. (The address varies depending on the DSP you are using.)
- 14) Right-click on the call stack area and choose Hide from the pop-up menu.
- 15) Remove the breakpoint that you added at the end of the `dataIO()` subroutine in step 4):
 - a) Put the cursor on the line after `return;` in `dataIO()`.
 - b) Click the  (Toggle Breakpoint) toolbar button or press 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 Open 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.
- 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.

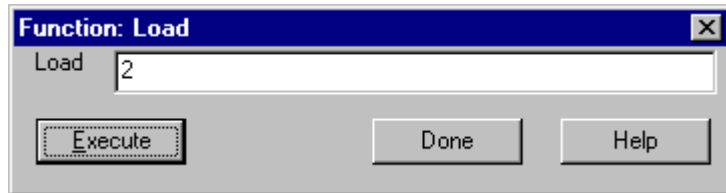
- 1) Choose Profiler→Enable Clock. Make sure you see a check mark next to this item in the Profiler menu.
- 2) Double-click on the volume.c file in the Project View.
- 3) Choose View→Mixed Source/ASM to view both the C source and the equivalent assembly instructions.
- 4) Put your cursor in the assembly instruction after the line that says:



```
load(processingLoad);
```
- 5) Click the  (Toggle Profile-point) toolbar button or right-click and select Toggle Profile Pt.
- 6) Put your cursor in the assembly instruction after the line that says:


```
return(TRUE);
```
- 7) Click the  (Toggle Profile-point) toolbar button.
- 8) Choose Profiler→View Statistics. 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.
- 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 27 cycles. (The actual numbers shown may vary.) This is the number of cycles required to execute the load routine when the processingLoad is 1.

Location	Count	Average	Total
VOLUME.C line 87	11	20.0	220
VOLUME.C line 84	11	2846.0	31306

- 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 36. The number of cycles increases by 9 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) Close the Load and Gain controls.
- 16) Before proceeding to Chapter 3 (after completing section 4.10), 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 Debug→Breakpoints. Select Delete All and click OK.
 - Go to Profiler→Profile-points. Select Delete All and click OK.
 - Go to the View menu and uncheck Mixed Source/ASM.

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 area 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` and rebuild, then reload the program. Change the Length in the File I/O dialog to `0x78` and the Acquisition Buffer Size for both graphs 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. 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.
- ❑ To practice using the project-building capabilities of Code Composer Studio, copy all files from 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. You need to use the Project→New menu item and the Project→Add Files to Project menu item. See section 4.1 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).

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.

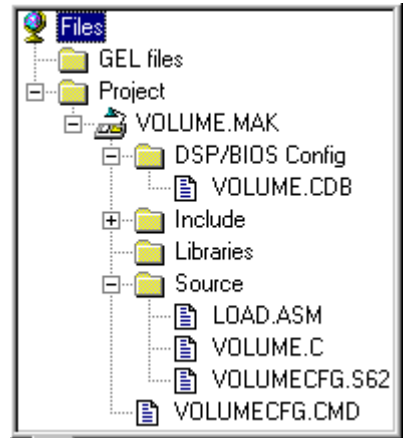
This chapter requires a physical board and cannot be carried out using a software simulator. Also, this chapter requires the DSP/BIOS components of Code Composer Studio.

Topic	Page
5.1 Opening and Examining the Project	5-2
5.2 Reviewing the Source Code	5-3
5.3 Modifying the Configuration File	5-6
5.4 Viewing Task Execution with the Execution Graph	5-10
5.5 Changing and Viewing the Load	5-12
5.6 Analyzing Task Statistics	5-15
5.7 Adding Explicit STS Instrumentation	5-17
5.8 Viewing Explicit Instrumentation	5-18
5.9 Things to Try	5-20
5.10 Learning More	5-20

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→Code Composer Studio.
- 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 datatypes 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);
}
```

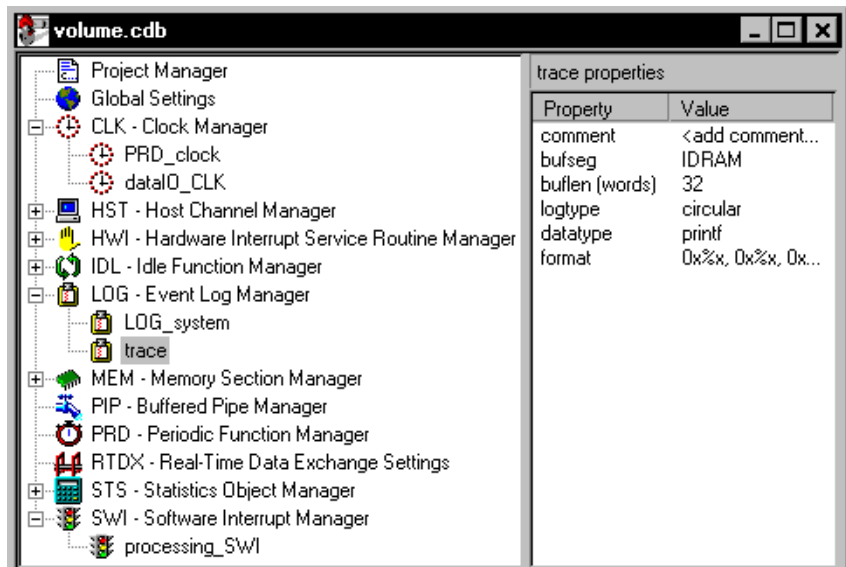
```
/* ===== dataIO ===== */
/* FUNCTION:      Called from timer ISR to fake a periodic
 *               hardware interrupt that reads in the input
 *               signal and outputs the processed signal.
 * PARAMETERS:   none
 * RETURN VALUE: none      */
Void dataIO()
{
    /* do data I/O */

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

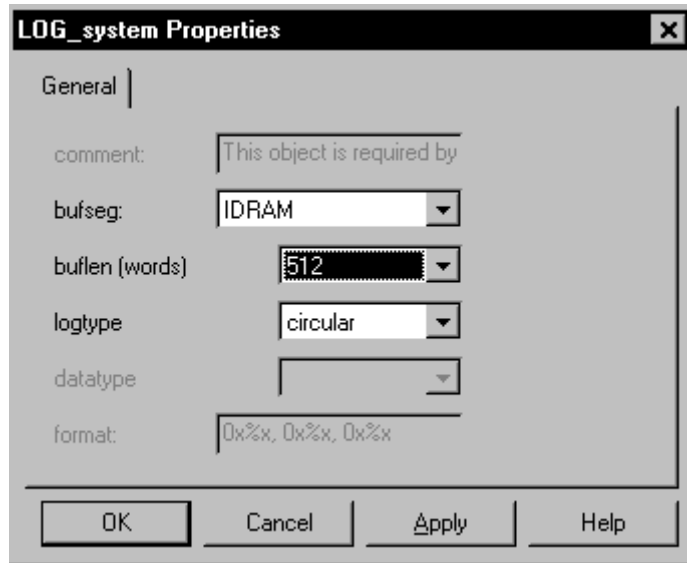
5.3 Modifying the Configuration File

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. 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.

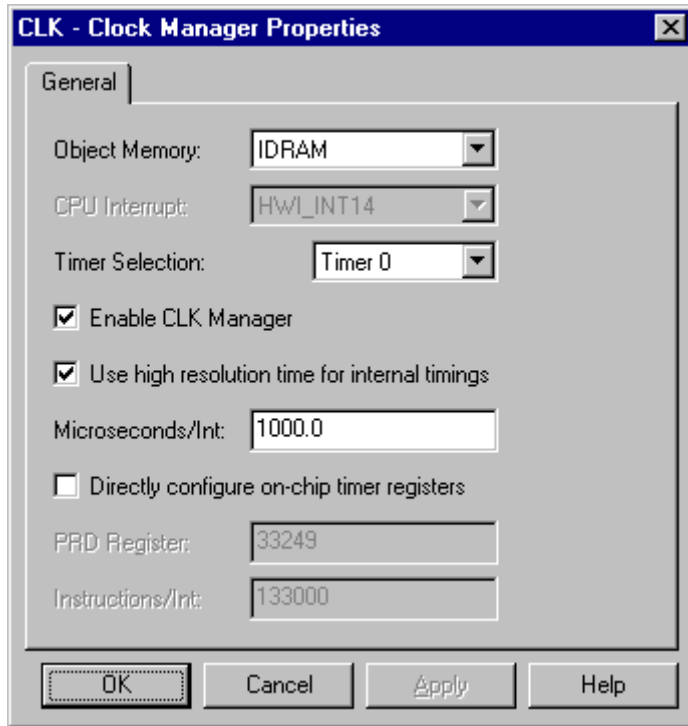
- 5) Change the buflen property to 512 words and click OK.
- 6) Highlight the CLK object called dataIO_CLK. Notice that the function called when this CLK object is activated is `_dataIO`. This is the `dataIO` function in `volume.c`.

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.

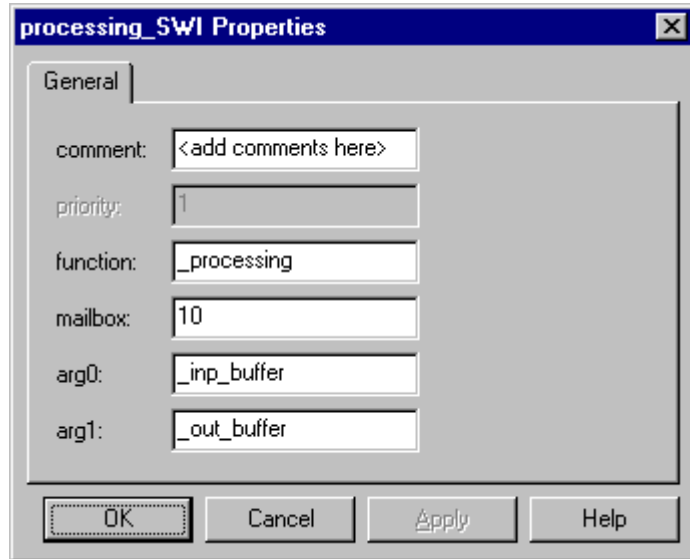



Notice that the CPU Interrupt for the Clock Manager is HWI_INT14. This property is gray because it is actually set by the HWI_INT14 object.

- 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 normally be required within a hardware ISR function.)

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

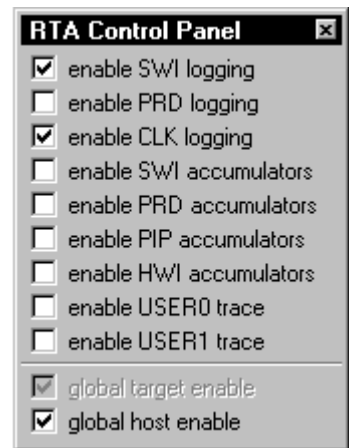



- **function.** When this software interrupt is activated, the processing function runs. This function is shown in section 5.2, page 5–4.
 - **mailbox.** The mailbox value can control when a software interrupt runs. Several API calls affect the value of the mailbox and may post the software interrupt depending on the resulting value. When a software interrupt is posted, it runs when it is the highest priority software or hardware interrupt task 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.

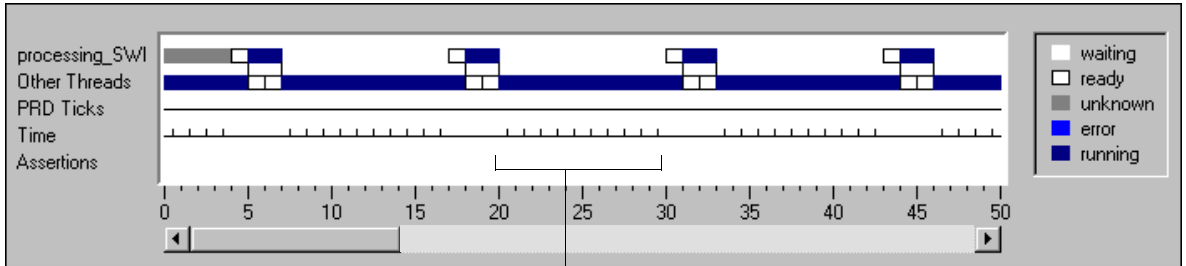
5.4 Viewing Task Execution with the Execution Graph

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 tasks can meet their real-time deadlines.

- 1) Choose File→Load Program. Select the program you just built, volume.out, 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 several check boxes 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 time marks

- 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.

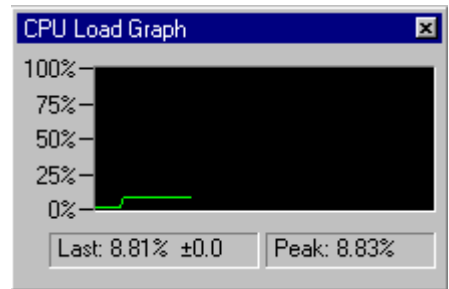
5.5 Changing and Viewing the Load

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 tasks than multiplying a value and copying it to another buffer. You can simulate such complex tasks 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.

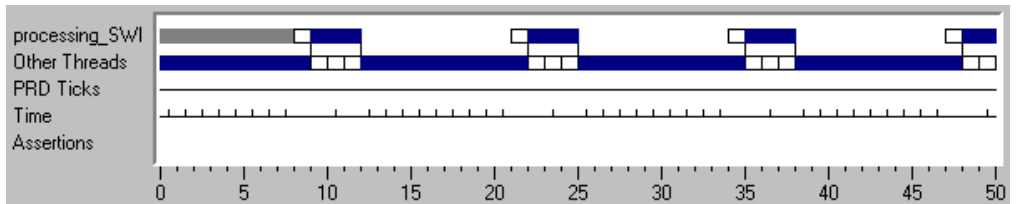
The Statistics View and the CPU Load transfer very little data from the target to the host, so you can set these windows to update frequently without causing a large effect on program execution. The Message Log and Execution Graph transfer the number of words specified for the buflen property of the 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 Open dialog, select the volume.gel file and click Open.
- 5) Choose GEL→Application Control→Load.
- 6) Type 10000 as the new load and click Execute. Notice that 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.

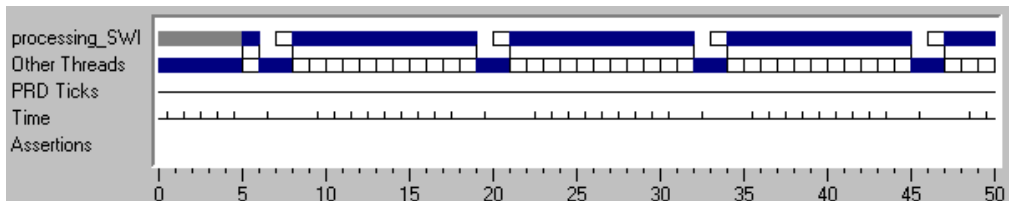
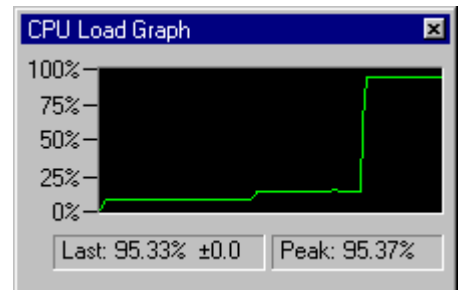


- 8) Using the GEL control, change the load to 20000 and click Execute.

- 9) Right-click on the Execution Graph and choose Clear from the pop-up menu. Notice that one of the time marks occurs 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 task before it needs to run again.

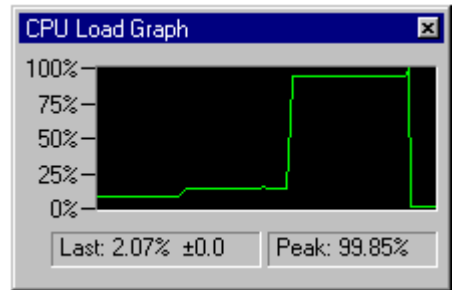


- 10) Using the GEL control, change the load to 140000 and click Execute. Notice that the CPU load increases to about 95%.
- 11) Right-click on the Execution Graph and choose Clear from the pop-up menu. Notice that the program still meets its real-time deadline because `processing_SWI` completes before 10 time marks have occurred.



- 12) Using the GEL control, change the load to 160000 and click Execute. Notice that the CPU Load Graph and the Execution Graph stop updating. This is because these updates are performed within an idle task, 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) Using the GEL control, change the load to 10 and click Execute. Notice that the CPU load and Execution Graph begin updating again.



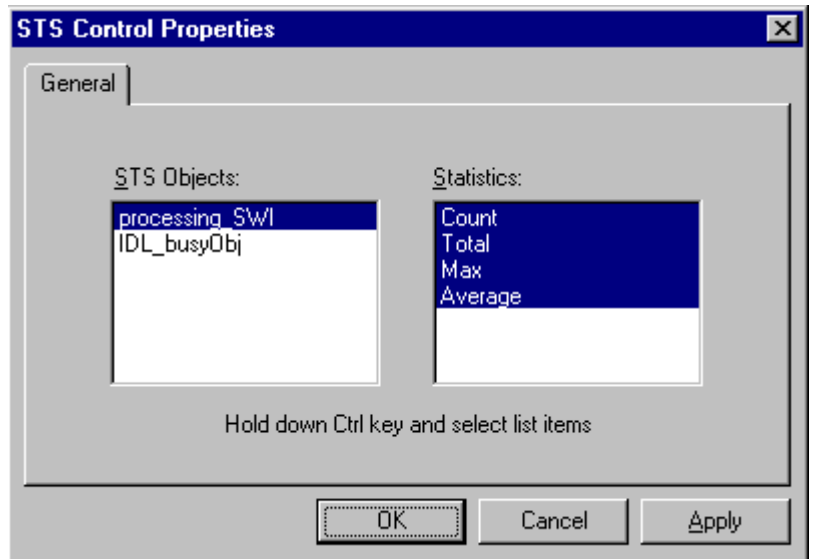
Note:

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.

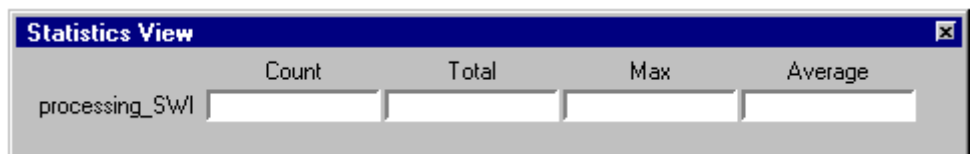
5.6 Analyzing Task Statistics

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.




- 3) Click OK. You see the statistics fields for the processing_SWI 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 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.
- 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_STIS. 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_STIS;
```

- 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_STIS, CLK_gethtime());
}
```

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

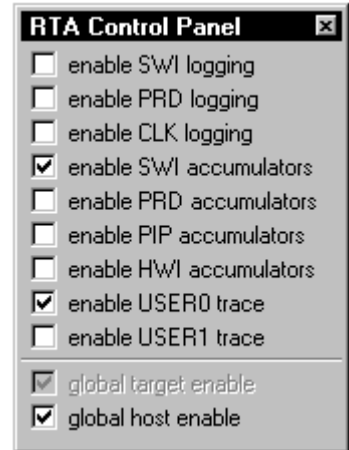
```
if (TRC_query(TRC_USER0) == 0) {
    STS_delta(&processingLoad_STIS, CLK_gethtime());
}
```

- 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

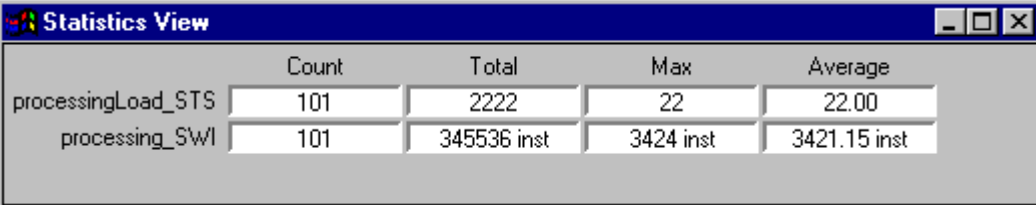
To view the information provided by the explicit instrumentation calls you added, you use the Statistics View and the RTA Control Panel.

- 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 and USER0 logging 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.



	Count	Total	Max	Average
processingLoad_STS	101	2222	22	22.00
processing_SWI	101	345536 inst	3424 inst	3421.15 inst

- 9) Notice the difference between the Max values for processing_SWI and processingLoad_STS. You used the CLK_gettime function to benchmark the processing load. On TMS320C6000 DSPs, the high-resolution time is incremented every 4 instruction cycles. Multiply the Max value for processingLoad_STS by 4 and subtract the result from the processing_SWI Max value. The result should be about 3308 instructions. (The actual numbers shown may vary.) These instructions are performed within the processing function, but not between the calls to STS_set and STS_delta.


For example, when the load is 10, the processingLoad_STS Max is about 49 and the processing_SWI Max is about 3504. (The actual numbers shown may vary.) To calculate the instruction cycles performed within the processing function but outside the calls to STS_set and STS_delta, the equation is:

$$3504 - (49 * 4) = 3308$$

- 10) Choose GEL→Application Control→Load. (If you have closed and restarted Code Composer Studio, you must reload the GEL file.)
- 11) Change the Load and click Execute.
- 12) 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.
- 13) Remove the check mark from the enable USER0 trace box in the RTA Control Panel.
- 14) Right-click on the Statistics View and choose Clear from the pop-up menu.
- 15) 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.

- 16) Before proceeding to the next chapter (after completing section 5.9), 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 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_gettime 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_gettime function instead of the CLK_gethtime function. Rebuild the program and notice how the values in the Statistics View change. The CLK_gettime 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_gettime.

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).

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	6-2
6.2 Modifying the Configuration File	6-3
6.3 Reviewing the Source Code Changes	6-5
6.4 Using the RTDX Control to Change the Load at Run Time	6-7
6.5 Modifying Software Interrupt Priorities	6-11
6.6 Things to Try	6-12
6.7 Learning More	6-12

6.1 Opening and Examining the Project

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

Note:

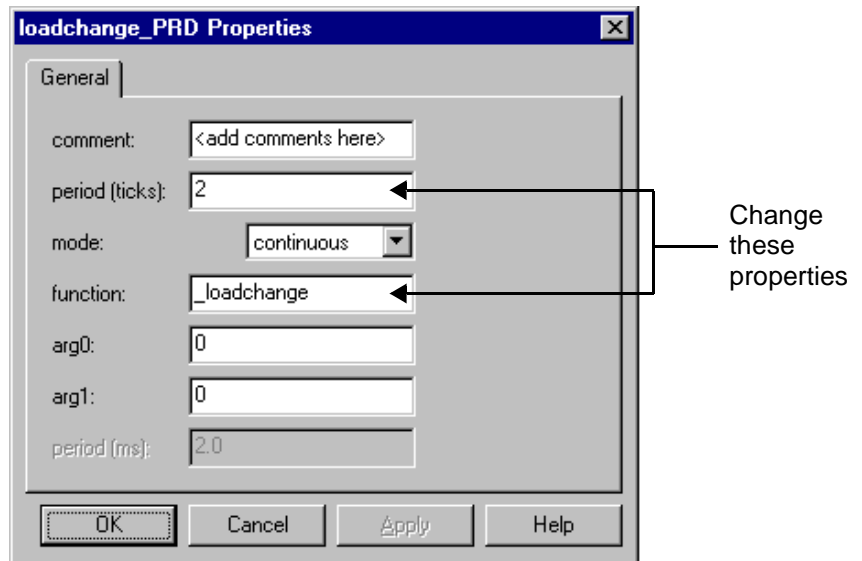
If you did not complete the previous chapter, you can copy example files from the volume3 folder that reflect the state of the example at the end of the previous chapter. Copy all the files from c:\ti\c6000\tutorial\volume3 (or the location where you installed Code Composer Studio) to your working folder.

- 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.
 - **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 application
- 2) From the Windows Start menu, choose Programs→Code Composer Studio 'C6000→Code Composer Studio.
- 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 on 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. Notice that 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.
- 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) Your own programs can call PRD_tick from some other event, such as a hardware interrupt, to drive periodic functions. Notice that the PRD manager has a property called Use CLK Manager to drive PRD. If you remove the check mark from this box, the PRD_clock object is automatically deleted. Make sure this box is checked for this example.
- 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 the previous section. 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 = BASELOAD << control;
            LOG_printf(&trace, "Load value = %d", processingLoad);
        }
    }
}
```

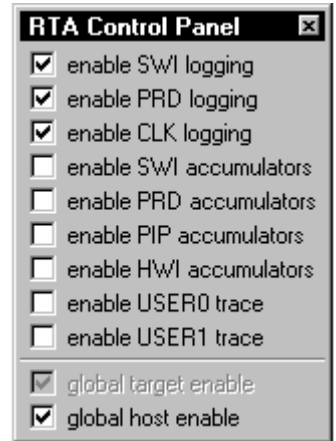
This function uses RTDX API functions to change the load of the processing signal in real time. Notice the following aspects of these changes:

- ❑ 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 = BASELOAD << control;` statement shifts the bits in `BASELOAD` (1) to the left by the number of digits specified by the `control`. Because `MAXCONTROL` is set to 31, the RTDX control allows you to set an exponentially scaled value for the load up to the maximum value that can be stored in a 32-bit variable.

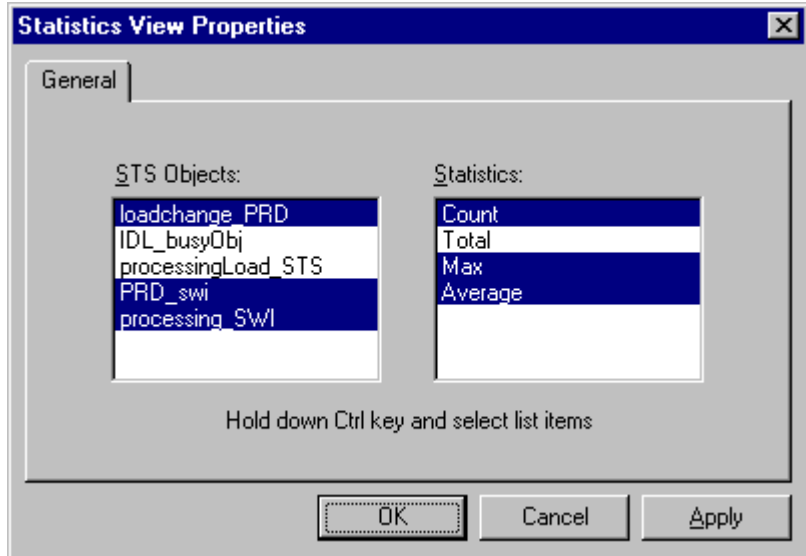
6.4 Using the RTDX Control to Change the Load at Run Time

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), you have already tested the signal processing algorithm. At this stage of development, your focus is on making sure the tasks 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.

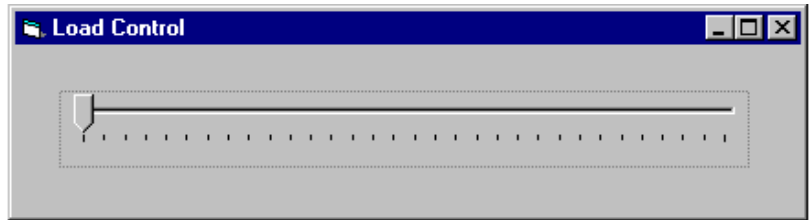


- 7) Right-click on the Statistics View area, which says Load program and/or set property to use control. Choose Property Page from the pop-up menu. Highlight the items shown here.



- 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.)

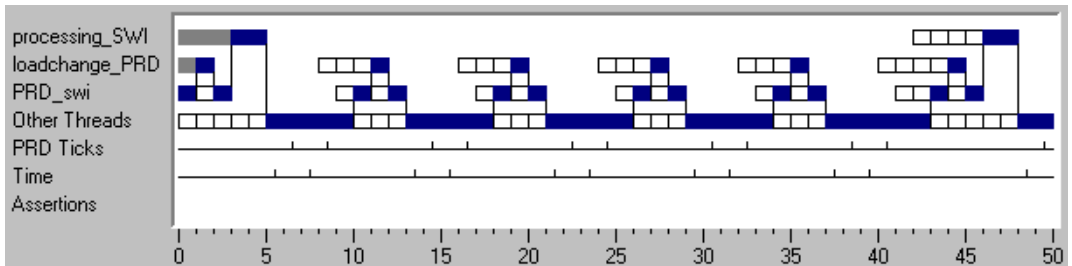
- 14) Using the Windows Explorer, run loadctrl.exe, which is located in your working folder. The Load Control window appears.



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 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.Writel4(data14, 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.

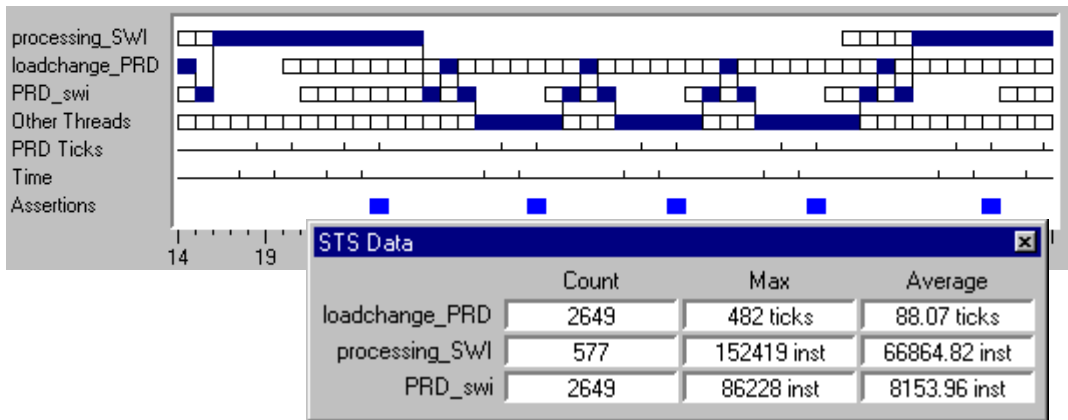


Notice that processing_SWI occurs once every 10 time ticks (and PRD ticks). PRD_swi runs once every 2 PRD ticks. The loadchange_PRD runs within the context of PRD_swi. These are the expected execution frequencies.

PRD statistics are measured in PRD ticks. SWI statistics are measured in on-chip timer counter increments. Notice that 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.)

STS Data			
	Count	Max	Average
loadchange_PRD	846	0 ticks	0.00 ticks
processing_SWI	170	3215 inst	3206.99 inst
PRD_swi	846	404 inst	338.40 inst

- 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.

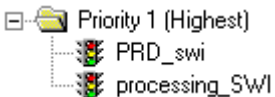
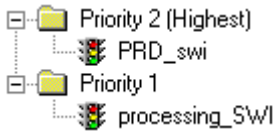



What is happening? Notice that the Max value for loadchange_PRD increases when you increase the load beyond a certain point. With the increased load, the processing_SWI takes so long to run that the loadchange_PRD cannot begin running until long past its real-time deadline.

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 tasks.

- 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.
 
- 4) Because PRD_swi and processing_SWI have the same priority, PRD_swi cannot preempt processing_SWI. 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.
- 5) To correct this problem, you need give PRD_swi the highest priority. Drag processing_SWI down. This adds a second priority level. Now, PRD_swi has the highest priority.
 
- 6) Select File→Save to save your changes.
- 7) Select File→Close to close volume.cdb.
- 8) Choose Project→Build or click the  (Incremental Build) toolbar button.
- 9) Run the example again using the RTDX-enabled Windows application loadctrl.exe to change the load at run time (as in section 6.4).
- 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 proceeding to the next chapter (after completing section 6.6), 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.

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? Hardware ISR run at a higher priority than the highest priority SWI object.
- ❑ View the CPU Load Graph. Use the RTA Control Panel to turn the statistics accumulators on and off. How much does this affect the CPU load? How much does this affect the statistics for `processing_SWI`?
- ❑ Add calls to `STS_set` and `STS_delta` in the `loadchange` function like the ones you added in section 5.7. 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 Code Composer Studio and RTDX online help or the *TMS320C6000 DSP/BIOS User's Guide* (which is provided as an Adobe Acrobat file).

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	7-2
7.2 Reviewing the C Source Code	7-3
7.3 Reviewing the Signalprog Application	7-6
7.4 Running the Application	7-7
7.5 Modifying the Source Code to Use Host Channels and Pipes ...	7-10
7.6 More about Host Channels and Pipes	7-12
7.7 Adding Channels and an SWI to the Configuration File	7-13
7.8 Running the Modified Program	7-17
7.9 Learning More	7-17

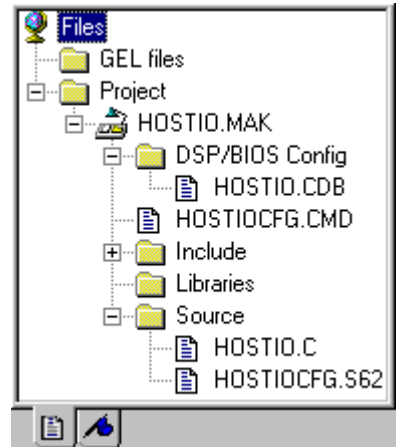
7.1 Opening and Examining the Project

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→Code Composer Studio.
- 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 don't 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.** A Visual Basic application that generates a sine wave and displays the input and output signals
- **slider.exe.** A Visual Basic application that 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 the next section.

- 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 the previous chapter, 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`.
 - 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.

- Notice that 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));

    /* 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));
}
}
```

7.3 Reviewing the Signalprog Application

The source code for the 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 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.

The following global declarations earlier in the Visual Basic source code connect 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 ReadSAI4 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.

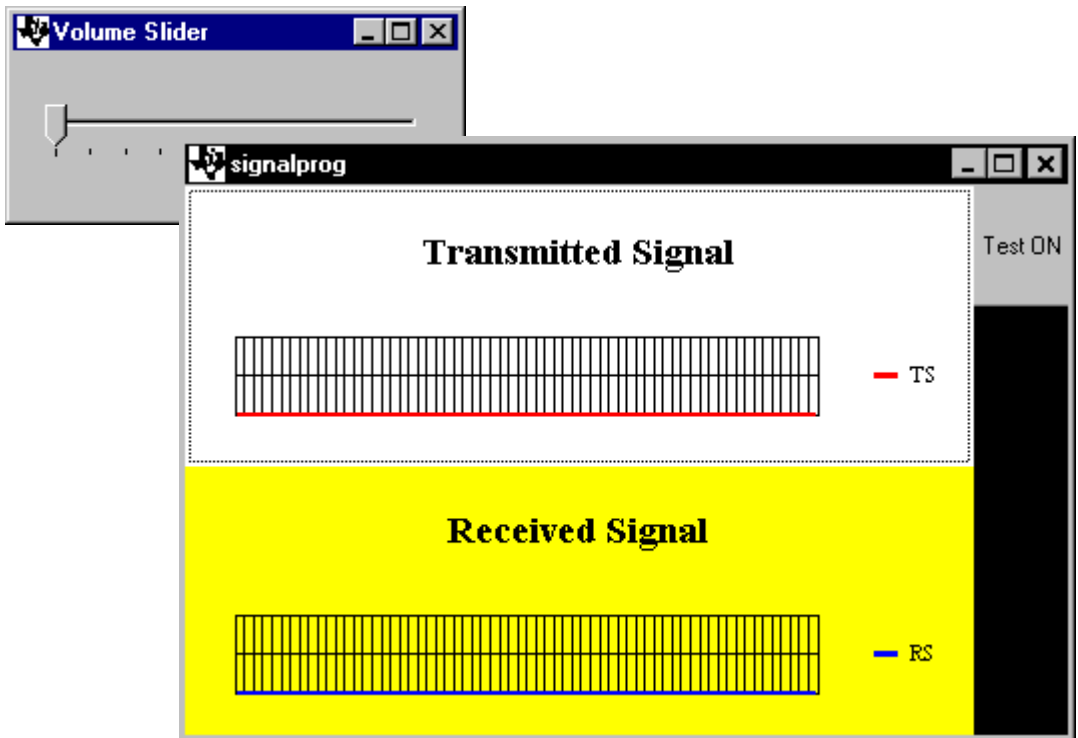
7.4 Running the Application

- 1) Choose File→Load Program. Select the program, hostio.out, and click Open.
- 2) Choose Tools→RTDX.
- 3) 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.
- 4) Change RTDX Disable to RTDX Enable in the RTDX area.

Note:

Changing RTDX Disable to RTDX Enable changes the Configure button to Diagnostics.

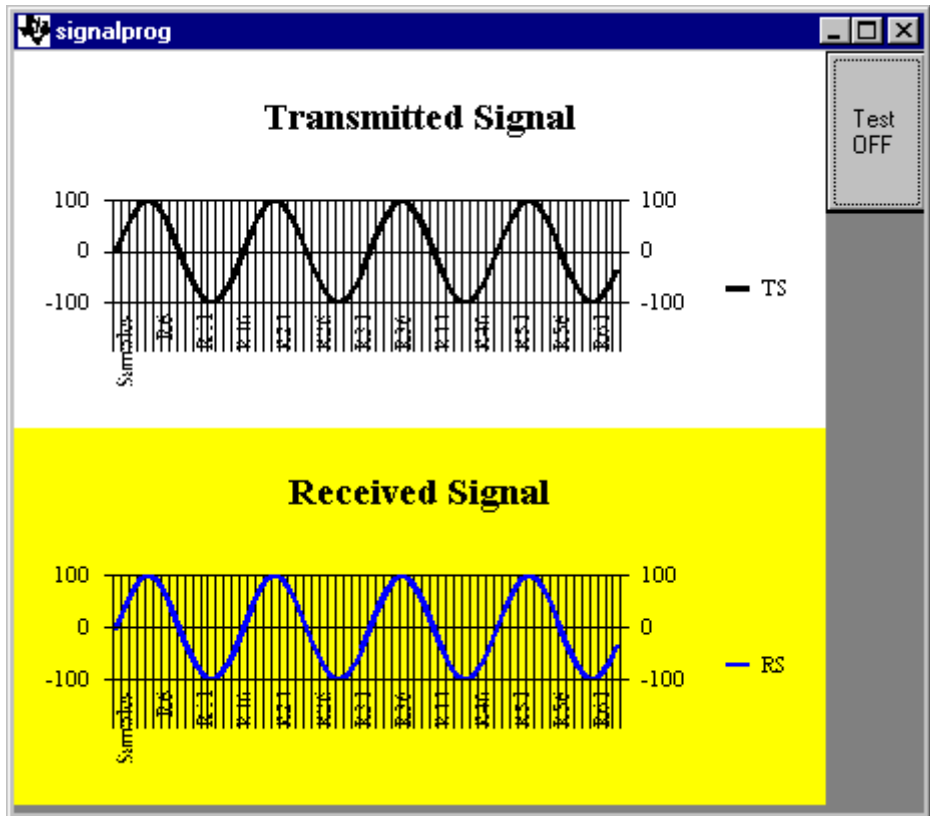
- 5) Choose Debug→Run or click the  (Run) toolbar button.
- 6) Using the Windows Explorer, run signalprog.exe and slider.exe. You see these two Visual Basic applications.



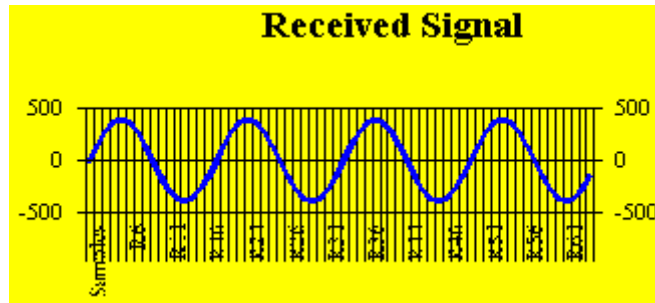
The slider.exe program must be started after RTDX is enabled and the program is running because it creates and opens the RTDX control channel when you run the program. If RTDX is not enabled at this point, slider.exe cannot open the channel.

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

- 7) Resize the signalprog window so that it is taller. This allows you to see the axis labels.
- 8) Click Test On in the signalprog window. This starts the input and output channels.




- 9) 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.



Note:

The received signal graph changes the scale labels rather than changing the displayed size of the sine waves.

- 10) Close the Volume Slider application. This stops the input and output channels.
- 11) Click Test OFF in the signalprog window. This closes the control channel.
- 12) Click  (Halt) or press Shift F5 to stop the program.

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. This function is called by the A2DscaleD2A_SWI object, which you create in the next section.

```
/* ===== 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 earlier in this chapter, 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

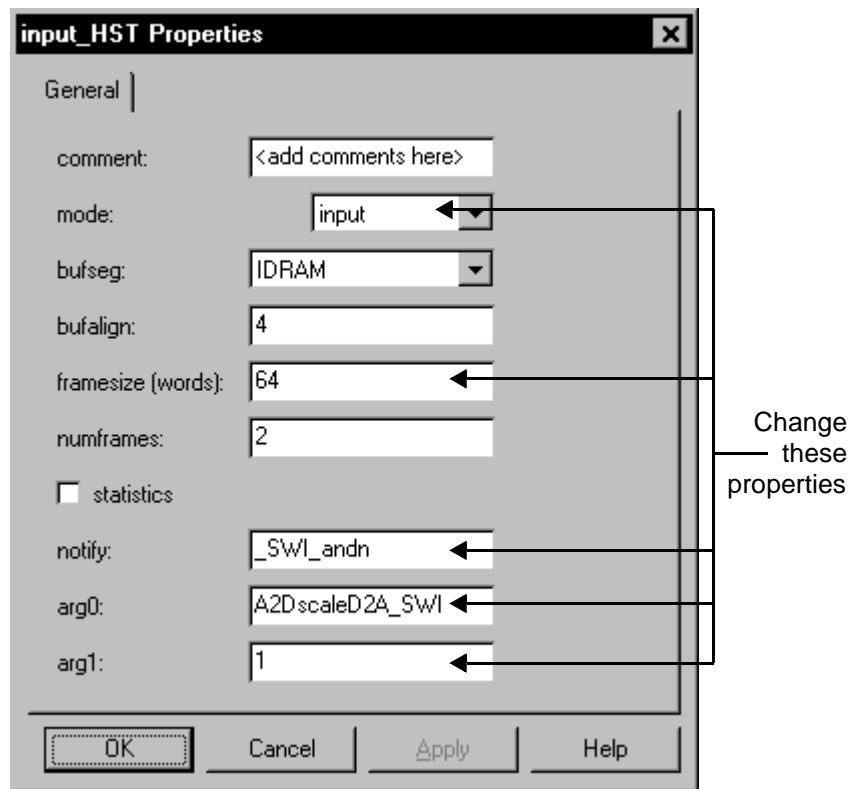
The A2DscaleD2A function is called by an SWI object and uses two HST objects. You create these objects in this section. (The hostio.cdb file in the c:\ti\c6000\tutorial\hostio2\ folder already contains these objects.)

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.

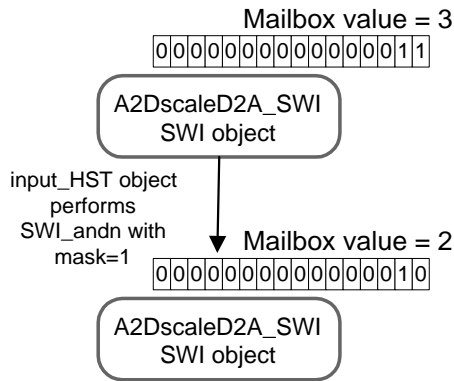
- 3) Rename the new HST0 object to input_HST.
- 4) Right-click on the input_HST object and choose Properties from the pop-up menu. Set the following properties for this object and click OK.



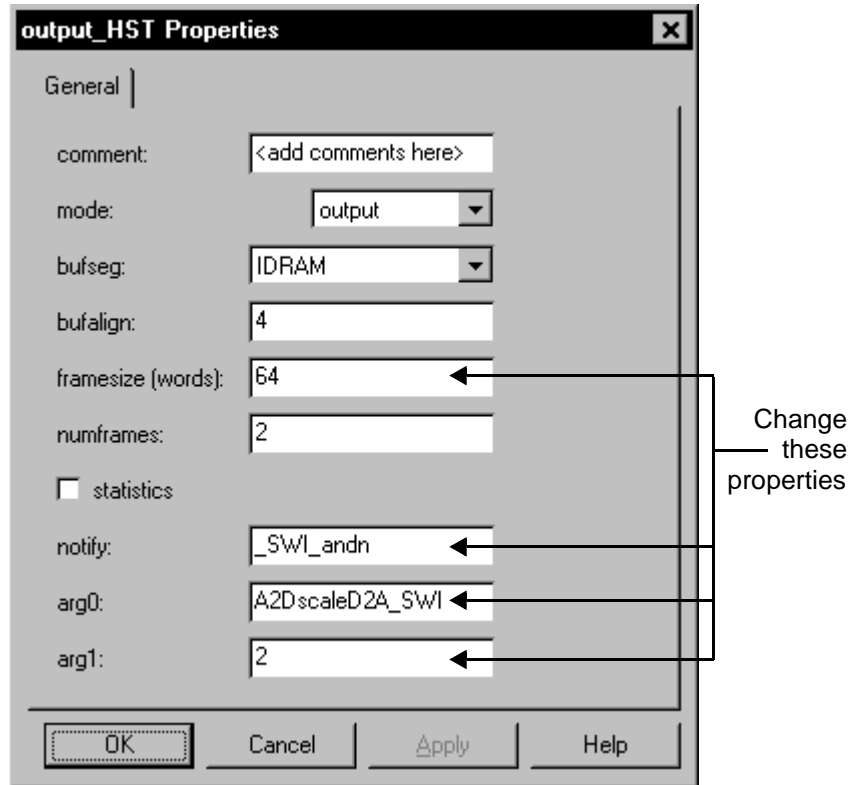
- **mode.** This property determines which end of the pipe the target program manages, and which end the Host Channel Control plug-in manages. An input channel sends data from the host to the target. An output channel sends data from the target to the host.
- **framesize.** This property sets the size of each frame in the channel. Use 64 words—the same value as the BUFSIZE used earlier in this chapter
- **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.

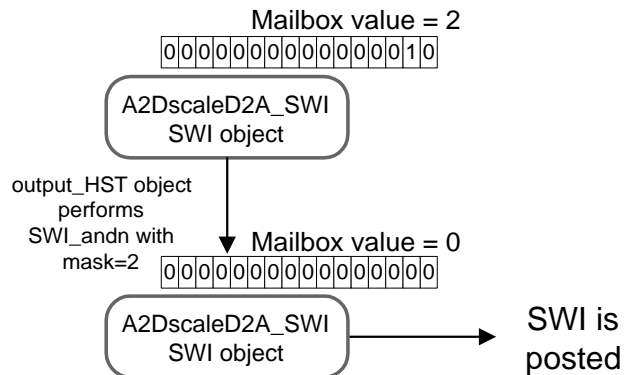
The SWI_andn function treats the mailbox value as a bitmask. It clears the bits specified by the second argument passed to the function. So, when this channel contains a full frame (because the target filled a frame), it calls SWI_andn for the A2DscaleD2A_SWI object and causes it to clear bit 1 of the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



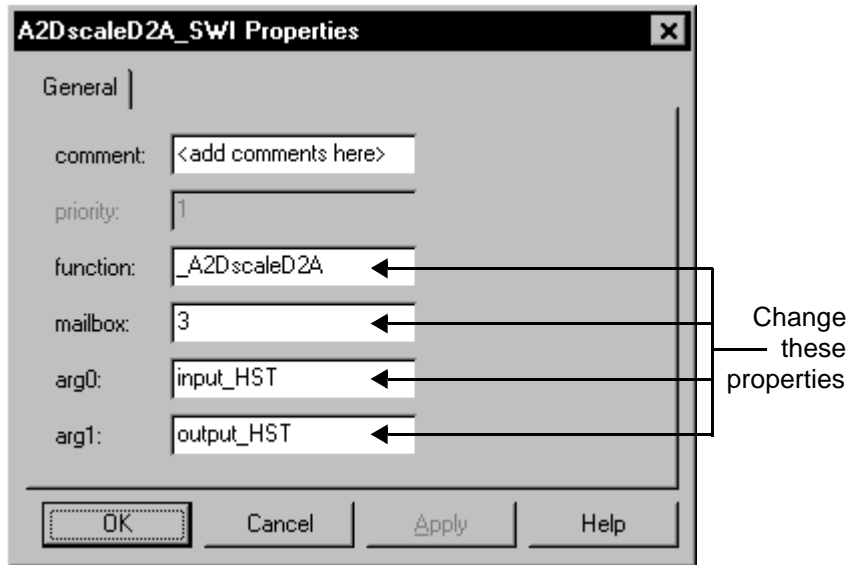
- 5) Insert another HST object and rename it output_HST.
- 6) Set the following properties for the output_HST object and click OK.



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.




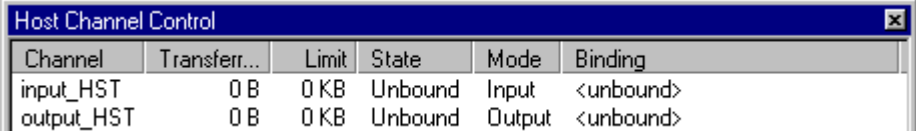
- 7) Right-click on the SWI manager and choose Insert SWI.
- 8) Rename the new SWI0 object to A2DscaleD2A_SWI.
- 9) Set the following properties for A2DscaleD2A_SWI and click OK.




- **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.

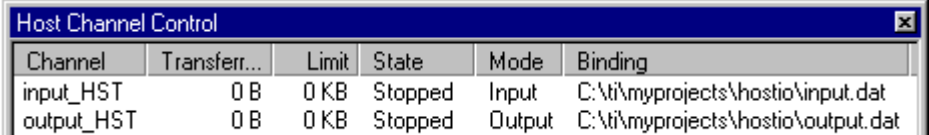
7.8 Running the Modified Program

- 1) Choose Project→Build or click the  (Incremental Build) 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.




Channel	Transferr...	Limit	State	Mode	Binding
input_HST	0 B	0 KB	Unbound	Input	<unbound>
output_HST	0 B	0 KB	Unbound	Output	<unbound>

- 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.



Channel	Transferr...	Limit	State	Mode	Binding
input_HST	0 B	0 KB	Stopped	Input	C:\ti\myprojects\hostio\input.dat
output_HST	0 B	0 KB	Stopped	Output	C:\ti\myprojects\hostio\output.dat

- 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).



A

- animating program 4-9
- archiver 1-5
- asm file 1-16
- assembler 1-5
- assembly optimizer 1-5
- assembly vs. C 5-7
- assembly, viewing with C source 2-13
- assertions 6-10

B

- bin folder 1-15
- bios folder 1-15
- breakpoint
 - creating 2-10
 - deleting 2-12
 - stepping from 2-11
- building application 2-6

C

- C compiler 1-5
- c file 1-16
- C vs. assembly 5-7
- c6201 vs. c6701 iv
- C6X_A_DIR environment variable 1-17, 2-3
- C6X_C_DIR environment variable 1-17, 2-3
- cdb file 1-16, 3-3
- cgtools folder 1-15
- chip type 3-3
- CLK module 1-11
 - manager properties 5-8
- CLK_gethtime function 5-17
- CLK_getltime function 5-20
- clock
 - viewing 4-16
- clock manager 5-8
- clock, enabling 2-13
- cmd file 1-16, 2-3, 3-3

- Code Composer Studio
 - vs. Code Composer Studio Simulator 1-2
- COFF file 1-5
- color, modifying 2-10
- command file 2-3
- configuration file
 - adding to project 3-4
 - creating 3-2
 - generated files 3-3
 - opening 5-6
 - saving 3-3
- CPU Load Graph 5-12
- cross-reference utility 1-5

D

- datatypes for DSP/BIOS 5-3
- development cycle 1-2
- development flow diagram 1-4
- directories 1-15
 - search path 2-3
- docs folder 1-15
- DOS environment space, increasing 1-17
- drivers folder 1-15
- DSP type 3-3
- DSP/BIOS
 - API modules 1-9
 - datatypes 5-3
 - header files 3-5

E

- edit variable 2-12
- enable clock 2-13
- environment variables
 - C6X_A_DIR 1-17
 - C6X_C_DIR 1-17
 - PATH 1-17
- EPROM programmer 1-5
- examples folder 1-15
- Execution Graph 5-10

explicit instrumentation 5-17

F

file I/O 4-6
file streaming 1-9
floating-point support iv
folders 1-15
 search path 2-3
font, setting 2-4
function names 5-7

G

GEL
 functions 4-13
gel folder 1-15
generated files 3-3
graph
 clearing 4-8
 viewing 4-8

H

h file 1-16
halting program 2-7
 at breakpoint 2-10
header files 3-5
hello1 example 2-2
hello2 example 3-2
hello.c 2-4, 3-4
hex conversion utility 1-5
Host Channel Control 7-17
host operation 5-20
hostio1 example 7-2
hostio2 example 7-10
hostio.c 7-3, 7-10
HST module 1-11, 7-10
 creating object 7-13
HST_getpipe function 7-11
HWI module 1-11
 object properties 5-8

I

IDL module 1-11, 3-10
idle loop 3-5, 5-3
ifdef symbols 2-8
implicit instrumentation 5-15
include files 3-5

inserting an object 3-3
instruction cycles 2-13
 profiling 2-14
integrated development environment 1-7
interrupts
 disabled vs. enabled 3-10
 software 5-3

L

lib file 1-16
libraries
 adding to project 2-3
 runtime-support 1-5
library-build utility 1-5
linker 1-5
linker command file 2-3, 3-3
loading program 2-6
LOG module 1-11
LOG object
 creating 3-3
 declaring 3-5
LOG_printf function 3-5

M

mailbox value 5-9
main function 3-5
mak file 1-16
MEM module 1-11
Message Log 3-6
mixed source/ASM 2-13
modules, list of 1-11
myprojects folder 1-15

N

naming conventions 5-7
new project 2-2

O

obj file 1-16
object
 editing properties 5-7
 inserting 3-3
 renaming 3-3
options
 color 2-10
 font 2-4

options for project 2-8
out file 1-16

P

PATH environment variable 1-17
performance monitoring 1-9
PIP module 1-11, 7-10
PIP_alloc function 7-11
PIP_free function 7-11
PIP_get function 7-11
PIP_getReaderAddr function 7-11
PIP_getReaderNumFrames function 7-11
PIP_getReaderSize function 7-11
PIP_getWriterAddr function 7-11
PIP_getWriterNumFrames function 7-11
PIP_put function 7-11
plug-ins
 DSP/BIOS 1-9
 RTDX 1-13
 third party 1-14
plugins folder 1-15
PRD module 1-12
 adding object 6-3
PRD_clock CLK object 6-4
PRD_swi object 6-4
PRD_tick function 6-4
preprocessor symbols 2-8
priorities of software interrupts 6-11
Probe Point
 connecting 4-7
 creating 4-6
profile clock 2-13
profile-point
 creating 2-13
 viewing statistics 2-14
program
 halting 2-7
 loading onto target 2-6
 running 2-6
program tracing 1-9
project
 adding files 2-3
 building 2-6
 creating new 2-2
 options 2-8
 viewing files in 2-3
project management 1-8
properties
 changing 5-7
 viewing 5-6

R

real-time deadlines 5-12, 6-10
renaming an object 3-3
resetting DSP 2-6
RTDX
 channel declaration 7-3
 rt dx folder 1-15
 RTDX host interface 1-13
 RTDX module 1-12
 configuring 6-8, 7-7
 RTDX_channelBusy function 6-5, 7-3
 RTDX_enableInput function 6-5, 7-4
 RTDX_read function 7-3
 RTDX_readNB function 6-5
 RTDX_write function 7-3
 rt dx.Close 6-9
 rt dx.Open 6-9
 rt dx.Writel4 6-9
 rts6201.lib 2-3
 rts6701.lib 2-3
 running 2-6
 animation 4-9
 to cursor 2-11
 to main 5-10
 running Code Composer Studio 2-2
 runtime-support libraries 1-5

S

saving 2-9
search path 2-3
sectti utility 3-10
signalprog.exe 7-6
simulator iii, 1-2
slider.exe 7-7
source files
 adding to project 2-3
 mixed C/assembly view 2-13
starting Code Composer Studio 2-2
statistics
 units 5-20, 6-10
 viewing with profile-points 2-14
Statistics View 5-15
step commands 2-11
stopping program 2-7
structure
 watch variables 2-12
STS module 1-12
 adding instrumentation 5-17
 Statistics View 5-15
STS_delta function 5-17
STS_set function 5-17
SWI module 1-12

- object properties 5-9
- priorities 6-11
- SWI_andn function 7-14
- SWI_dec function 5-3
- symbols, defining 2-8
- syntax errors 2-9

T

- time/frequency graph 4-8
- TRC module 1-12
- TRC_query function 5-17
- troubleshooting 2-6
- tutorial folder 1-15

U

- underscore 5-7
- uninstall folder 1-15
- USER0 tracing 5-18
- utility folder (bin) 1-15

V

- variables, editing 2-12
- vectors.asm 2-3
- view statistics 2-14
- Visual Basic application 6-2
- volume1 example 4-2
- volume2 example 5-2
- volume3 example 6-2
- volume4 example 6-2
- volume.c 4-4, 5-3, 6-5

W

- watch variable
 - adding 2-10
 - changing value 2-12
 - removing 2-12
 - structures 2-12
- wks file 1-16