1

DSP Development System

- Installing and testing Code Composer Studio Version 3.1
- Use of the TMS320C6713 or TMS320C6416 DSK
- Programming examples

This chapter describes how to install and test Texas Instruments' integrated development environment (IDE), Code Composer Studio (CCS), for either the TMS320C6713 or the TMS320C6416 Digital Signal Processing Starter Kit (DSK). Three example programs that demonstrate hardware and software features of the DSK and CCS are presented. It is recommended strongly that you review these examples before proceeding to subsequent chapters. The detailed instructions contained in this chapter are specific to CCS Version 3.1.

1.1 INTRODUCTION

The Texas Instruments TMS320C6713 and TMS320C6416 Digital Signal Processing Starter Kits are low cost development platforms for real-time digital signal processing applications. Each comprises a small circuit board containing either a TMS320C6713 floating-point digital signal processor or a TMS320C6416 fixed-point digital signal processor and a TLV320AIC23 analog interface circuit (codec) and connects to a host PC via a USB port. PC software in the form of Code Composer Studio (CCS) is provided in order to enable software written in C or assembly

Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK, Second Edition By Rulph Chassaing and Donald Reay
Copyright © 2008 John Wiley & Sons, Inc.

language to be compiled and/or assembled, linked, and downloaded to run on the DSK. Details of the TMS320C6713, TMS320C6416, TLV320AIC23, DSK, and CCS can be found in their associated datasheets [36–38]. The purpose of this chapter is to introduce the installation and use of either DSK.

A digital signal processor (DSP) is a specialized form of microprocessor. The architecture and instruction set of a DSP are optimized for real-time digital signal processing. Typical optimizations include hardware multiply-accumulate (MAC) provision, hardware circular and bit-reversed addressing capabilities (for efficient implementation of data buffers and fast Fourier transform computation), and Harvard architecture (independent program and data memory systems). In many cases, DSPs resemble microcontrollers insofar as they provide single chip computer solutions incorporating onboard volatile and nonvolatile memory and a range of peripheral interfaces and have a small footprint, making them ideal for embedded applications. In addition, DSPs tend to have low power consumption requirements. This attribute has been extremely important in establishing the use of DSPs in cellular handsets. As may be apparent from the foregoing, the distinctions between DSPs and other, more general purpose, microprocessors are blurred. No strict definition of a DSP exists. Semiconductor manufacturers bestow the name DSP on products exhibiting some, but not necessarily all, of the above characteristics as they see fit.

The C6x notation is used to designate a member of the Texas Instruments (TI) TMS320C6000 family of digital signal processors. The architecture of the C6x digital signal processor is very well suited to numerically intensive calculations. Based on a very-long-instruction-word (VLIW) architecture, the C6x is considered to be TI's most powerful processor family.

Digital signal processors are used for a wide range of applications, from communications and control to speech and image processing. They are found in cellular phones, fax/modems, disk drives, radios, printers, hearing aids, MP3 players, HDTV, digital cameras, and so on. Specialized (particularly in terms of their onboard peripherals) DSPs are used in electric motor drives and a range of associated automotive and industrial applications. Overall, DSPs are concerned primarily with real-time signal processing. Real-time processing means that the processing must keep pace with some external event; whereas nonreal-time processing has no such timing constraint. The external event to keep pace with is usually the analog input. While analog-based systems with discrete electronic components including resistors and capacitors are sensitive to temperature changes, DSP-based systems are less affected by environmental conditions such as temperature. DSPs enjoy the major advantages of microprocessors. They are easy to use, flexible, and economical.

A number of books and articles have been published that address the importance of digital signal processors for a number of applications [1–22]. Various technologies have been used for real-time processing, from fiber optics for very high frequency applications to DSPs suitable for the audio frequency range. Common applications using these processors have been for frequencies from 0 to 96 kHz. It is standard within telecommunications systems to sample speech at 8kHz (one sample every 0.125ms). Audio systems commonly use sample rates of 44.1kHz (compact disk) or 48kHz. Analog/digital (A/D)-based data-logging boards in the megahertz sampling rate range are currently available.

1.2 DSK SUPPORT TOOLS

Most of the work presented in this book involves the development and testing of short programs to demonstrate DSP concepts. To perform the experiments described in the book, the following tools are used:

1. *A Texas Instruments DSP starter kit (DSK).* The DSK package includes:
   (a) Code Composer Studio (CCS), which provides the necessary software support tools. CCS provides an integrated development environment (IDE), bringing together the C compiler, assembler, linker, debugger, and so on.
   (b) A circuit board (the TMS320C6713 DSK is shown in Figure 1.1) containing a digital signal processor and a 16-bit stereo codec for analog signal input and output.
   (c) A universal synchronous bus (USB) cable that connects the DSK board to a PC.
   (d) A +5V universal power supply for the DSK board.
2. *A PC.* The DSK board connects to the USB port of the PC through the USB cable included with the DSK package.
3. *An oscilloscope, spectrum analyzer, signal generator, headphones, microphone, and speakers.* The experiments presented in subsequent chapters of this book are intended to demonstrate digital signal processing concepts in real-time, using audio frequency analog input and output signals. In order to appreciate those concepts and to get the greatest benefit from the experiments, some forms of signal source and sink are required. As a bare minimum, a microphone and either headphones or speakers are required. A far greater benefit will be acquired if a signal generator is used to generate sinusoidal, and other, test signals and an oscilloscope and spectrum analyzer are used to display, measure, and analyze input and output signals. Many modern digital oscilloscopes incorporate FFT functions, allowing the frequency content of signals to be displayed. Alternatively, a number of software packages that use a PC equipped with a soundcard to implement virtual instruments are available.

All the files and programs listed and discussed in this book (apart from some of the student project files in Chapter 10) are included on the accompanying CD. A list of all the examples is given on pages xxi–xxvi.

1.2.1 C6713 and C6416 DSK Boards

The DSK packages are powerful, yet relatively inexpensive, with the necessary hardware and software support tools for real-time signal processing [23–43]. They are complete DSP systems. The DSK boards, which measure approximately 5×8 inches, include either a 225-MHz C6713 floating-point digital signal processor or a 1-GHz C6416 fixed-point digital signal processor and a 16-bit stereo codec TLV320AIC23 (AIC23) for analog input and output.

The onboard codec AIC23 [38] uses sigma–delta technology that provides analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) functions. It uses a 12-MHz system clock and its sampling rate can be selected from a range of alternative settings from 8 to 96 kHz.

A daughter card expansion facility is also provided on the DSK boards. Two 80-pin connectors provide for external peripheral and external memory interfaces.

The DSK boards each include 16 MB (megabytes) of synchronous dynamic RAM (SDRAM) and 512 kB (kilobytes) of flash memory. Four connectors on the boards provide analog input and output: MIC IN for microphone input, LINE IN for line input, LINE OUT for line output, and HEADPHONE for a headphone output (multiplexed with line output). The status of four user DIP switches on the DSK board can be read from within a program running on the DSP and provide the user with a feedback control interface. The states of four LEDs on the DSK board can be controlled from within a program running on the DSP. Also onboard the DSKs are voltage regulators that provide 1.26 V for the DSP cores and 3.3 V for their memory and peripherals.

1.2.2 TMS320C6713 Digital Signal Processor

The TMS320C6713 (C6713) is based on the very-long-instruction-word (VLIW) architecture, which is very well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. For example, with a clock rate of 225 MHz, the C6713 is capable of fetching eight 32-bit instructions every 1/(225 MHz) or 4.44 ns.

Features of the C6713 include 264 kB of internal memory (8 kB as L1P and L1D Cache and 256 kB as L2 memory shared between program and data space), eight functional or execution units composed of six ALUs and two multiplier units, a 32-bit address bus to address 4 GB (gigabytes), and two sets of 32-bit general-purpose registers.

The C67xx processors (such as the C6701, C6711, and C6713) belong to the family of the C6x floating-point processors; whereas the C62xx and C64xx belong to the family of the C6x fixed-point processors. The C6713 is capable of both fixed- and floating-point processing. The architecture and instruction set of the C6713 are discussed in Chapter 3.

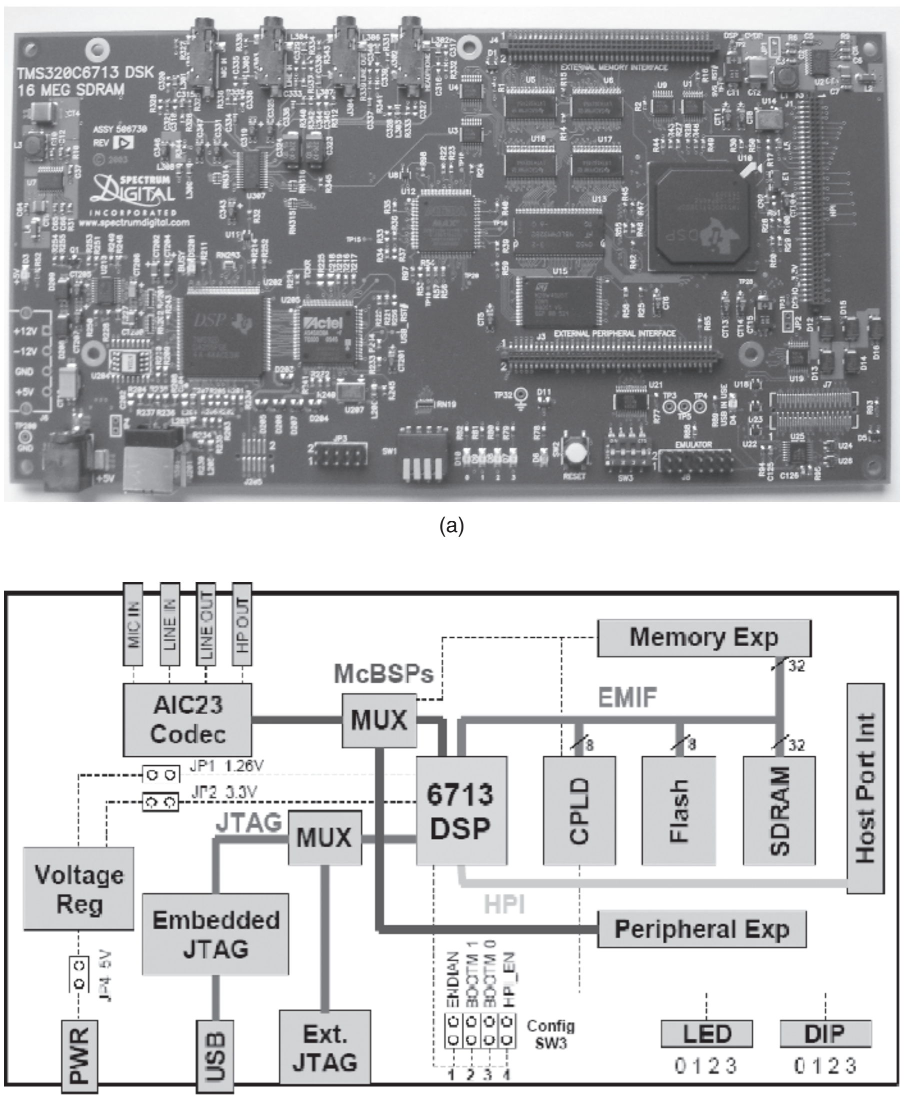


FIGURE 1.1. TMS3206713-based DSK board: (a) board and (b) block diagram. (Courtesy of Texas Instruments.)

1.2.3 TMS320C6416 Digital Signal Processor

The TMS320C6416 (C6416) is based on the VELOCITI advanced very-long-instruction-word (VLIW) architecture, which is very well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. For example, with a clock rate of 1 GHz, the C6416 is capable of fetching eight 32-bit instructions every 1/(1 GHz) or 1.0 ns.

Features of the C6416 include 1056 kB of internal memory (32 kB as L1P and L1D cache and 1024 kB as L2 memory shared between program and data space), eight functional or execution units composed of six ALUs and two multiplier units, a 32-bit address bus to address 4 GB (gigabytes), and two sets of 32-bit general-purpose registers.

1.3 CODE COMPOSER STUDIO

Code Composer Studio (CCS) provides an integrated development environment (IDE) for real-time digital signal processing applications based on the C programming language. It incorporates a C compiler, an assembler, and a linker. It has graphical capabilities and supports real-time debugging.

The C compiler compiles a C source program with extension `.c` to produce an assembly source file with extension *`.asm`*. The assembler assembles an *`.asm`* source file to produce a machine language object file with extension *`.obj`*. The linker combines object files and object libraries as input to produce an executable file with extension *`.out`*. This executable file represents a linked common object file format (COFF), popular in Unix-based systems and adopted by several makers of digital signal processors [44]. This executable file can be loaded and run directly on the digital signal processor. Chapter 3 introduces the linear assembly source file with extension `.sa`, which is a "cross" between C and assembly code. A linear optimizer optimizes this source file to create an assembly file with extension `.asm` (similar to the task of the C compiler).

A Code Composer Studio project comprises all of the files (or links to all of the files) required in order to generate an executable file. A variety of options enabling files of different types to be added to or removed from a project are provided. In addition, a Code Composer Studio project contains information about exactly how files are to be used in order to generate an executable file. Compiler/linker options can be specified. A number of debugging features are available, including setting breakpoints and watching variables, viewing memory, registers, and mixed C and assembly code, graphing results, and monitoring execution time. One can step through a program in different ways (step into, or over, or out).

Real-time analysis can be performed using CCS's real-time data exchange (RTDX) facility. This allows for data exchange between the host PC and the target DSK as well as analysis in real-time without halting the target. The use of RTDX is illustrated in Chapter 9.

1.3.1 CCS Version 3.1 Installation and Support

Instructions for installation of CCS Version 3.1 are supplied with the DSKs. The default location for CCS files is *`c:\CCStudio_v3.1`* and the following instructions assume that that you have used this default. An icon with the label *6713 DSK CCStudio v3.1* (or *6416 DSK CCStudio v3.1*) should appear on the desktop.

CCS Version 3.1 provides extensive help facilities and a number of examples and tutorials are included with the DSK package. Further information (e.g., data sheets and application notes) are available on the Texas Instruments website `http://www.ti.com`.

1.3.2 Installation of Files Supplied with This Book

The great majority of the examples described in this book will run on either the C6713 or the C6416 DSK. However, there are differences, particularly concerning the library files used by the different processors, and for that reason a complete set of files is provided on the CD for each DSK. Depending on whether you are using a C6713 or a C6416 DSK, copy all of the subfolders, and their contents, supplied on the CD accompanying this book in folders C6416 or C6713 into the folder *c:\CCStudio_v3.1\MyProjects* so that, for example, the source file *sine8_LED.c* will be located at *c:\CCStudio_v3.1\MyProjects\sine8_LED\sine8_LED.c*.

Change the properties of all the files copied so that they are not read-only (all the folders can be highlighted to change the properties of their contents at once).

1.3.3 File Types

You will be working with a number of files with different extensions. They include:

1. `file.pjt`: to create and build a project named file.
2. `file.c`: C source program.
3. `file.asm`: assembly source program created by the user, by the C compiler, or by the linear optimizer.
4. `file.sa`: linear assembly source program. The linear optimizer uses *file.sa* as input to produce an assembly program *file.asm*.
5. `file.h`: header support file.
6. `file.lib`: library file, such as the run-time support library file `rts6700.lib`.
7. `file.cmd`: linker command file that maps sections to memory.
8. `file.obj`: object file created by the assembler.
9. `file.out`: executable file created by the linker to be loaded and run on the C6713 or C6416 processor.
10. `file.cdb`: configuration file when using DSP/BIOS.

1.4 QUICK TESTS OF THE DSK (ON POWER ON AND USING CCS)

1. On power on, a power on self-test (POST) program, stored by default in the onboard flash memory, uses routines from the board support library (BSL) to test the DSK. The source file for this program, `post.c`, is stored in folder

`c:\CCStudio_v3.1\examples\dsk6713\bsl\post`. It tests the internal, external, and flash memory, the two multichannel buffered serial ports (McBSP), DMA, the onboard codec, and the LEDs. If all tests are successful, all four LEDs blink three times and stop (with all LEDs on). During the testing of the codec, a 1-kHz tone is generated for 1 second.

2. Launch CCS from the icon on the desktop. A USB enumeration process will take place and the Code Composer Studio window will open.
3. Click on *Debug→Connect* and you should see the message "The target is now connected" appear (for a few seconds) in the bottom left-hand corner of the CCS window.
4. Click on *GEL→Check DSK→QuickTest*. The Quick Test can be used for confirmation of correct operation and installation. A message of the following form should then be displayed in a new window within CCS:

Switches: 15 *Board Revision*: 2 *CPLD Revision*: 2

The value displayed following the label Switches reflects the state of the four DIP switches on the edge of the DSK circuit board. A value of 15 corresponds to all four switches in the up position. Change the switches to $(1110)_2$, that is, the first three switches (0,1,2) up and the fourth switch (3) down. Click again on *GEL→Check DSK→QuickTest* and verify that the value displayed is now 7 ("Switches: 7"). You can set the value represented by the four user switches from 0 to 15. Programs running on the DSK can test the state of the DIP switches and react accordingly. The values displayed following the labels Board Revision and CPLD Revision depend on the type and revision of the DSK circuit board.

Alternative Quick Test of DSK Using Code Supplied with This Book

1. Open/launch CCS from the icon on the desktop if not done already.
2. Select *Debug→Connect* and check that the symbol in the bottom left-hand corner of the CCS window indicates connection to the DSK.
3. Select *File→Load Program* and load the file `c:\CCStudio_v3.1\MyProjects\sine8_LED\Debug\`*`sine8_LED.out`*. This loads the executable file *`sine8_LED.out`* into the digital signal processor. (This assumes that you have already copied all the folders on the accompanying CD into the folder: `c:\CCStudio_v3.1\MyProjects`.)
4. Select *Debug→Run*.

Check that the DSP is running. The word RUNNING should be displayed in the bottom left-hand corner of the CCS window.

Press DIP switch #0 down. LED #0 should light and a 1-kHz tone should be generated by the codec. Connect the LINE OUT (or the HEADPHONE) socket on the DSK board to a speaker, an oscilloscope, or headphones and verify the generation of the 1-kHz tone. The four connectors on the DSK board for input and output (MIC, LINE IN, LINE OUT, and HEADPHONE) each use a 3.5-mm jack audio cable. halt execution of program `sine8_LED.out` by selecting *Debug→Halt*.

1.5 PROGRAMMING EXAMPLES TO TEST THE DSK TOOLS

Three programming examples are introduced to illustrate some of the features of CCS and the DSK board. The aim of these examples is to enable the reader to become familiar with both the software and hardware tools that will be used throughout this book. It is strongly suggested that you complete these three examples before proceeding to subsequent chapters. The examples will be described assuming that a C6713 DSK is being used.

Example 1.1: Sine Wave Generation Using Eight Points with DIP Switch Control (`sine8_LED`)

This example generates a sinusoidal analog output waveform using a table-lookup method. More importantly, it illustrates some of the features of CCS for editing source files, building a project, accessing the code generation tools, and running a program on the C6713 processor. The C source file *`sine8_LED.c`* listed in Figure 1.2 is included in the folder *`sine8_LED`*.

Program Description

The operation of program *`sine8_LED.c`* is as follows. An array, `sine_table`, of eight 16-bit signed integers is declared and initialized to contain eight samples of exactly one cycle of a sinusoid. The value of `sine_table[i]` is equal to

$$1000\sin(2\pi i/8) \quad \text{for } i = 1, 2, 3, \ldots, 7$$

Within function *`main()`*, calls to functions *`comm_poll()`*, *`DSK6713_LED_init()`*, and *`DSK6713_DIP_init()`* initialize the DSK, the AIC23 codec onboard the DSK, and the two multichannel buffered serial ports (McBSPs) on the C6713 processor. Function *`comm_poll()`* is defined in the file *`c6713dskinit.c`*, and functions *`DSK6713_LED_init()`* and *`DSK6713_DIP_init()`* are supplied in the board support library (BSL) file `dsk6713bsl.lib`.

The program statement `while(1)` within the function *`main()`* creates an infinite loop. Within that loop, the state of DIP switch #0 is tested and if it is pressed down, LED #0 is switched on and a sample from the lookup table is output. If DIP switch #0 is not pressed down then LED #0 is switched off. As long as DIP switch #0 is pressed down, sample values read from the array `sine_table` will be output and a sinusoidal analog output waveform will be generated via the left-hand channel of the AIC23 codec and the LINE OUT and HEADPHONE sockets. Each time a sample value is read from the array `sine_table`, multiplied by the value of the variable `gain`, and written to the codec, the index, `loopindex`, into the array

```
//sine8_LED.c  sine generation with DIP switch control

#include "dsk6713_aic23.h"              //codec support
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;    //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; //select input
#define LOOPLENGTH 8
short loopindex = 0;                    //table index
short gain = 10;                        //gain factor
short sine_table[LOOPLENGTH]=
  {0,707,1000,707,0,-707,-1000,-707};   //sine values

void main()
{
  comm_poll();                          //init DSK,codec,McBSP
  DSK6713_LED_init();                   //init LED from BSL
  DSK6713_DIP_init();                   //init DIP from BSL
  while(1)                              //infinite loop
  {
    if(DSK6713_DIP_get(0)==0)           //if DIP #0 pressed
    {
      DSK6713_LED_on();                 //turn LED #0 ON
      output_left_sample(sine_table[loopindex++]*gain); //output
      if (loopindex >= LOOPLENGTH) loopindex = 0; //reset index
    }
    else DSK6713_LED_off(0);            //else turn LED #0 OFF
  }                                     //end of while(1)
}                                       //end of main
```

FIGURE 1.2. Sine wave generation program using eight points with DIP switch control (`sine8_LED.c`).

is incremented and when its value exceeds the allowable range for the array (`LOOPLENGTH-1`), it is reset to zero.

Each time the function `output_left_sample()`, defined in source file *`C6713dskinit.c`*, is called to output a sample value, it waits until the codec, initialized by the function `comm_poll()` to output samples at a rate of 8kHz, is ready for the next sample. In this way, once DIP switch #0 has been pressed down it will be tested at a rate of 8kHz. The sampling rate at which the codec operates is set by the program statement

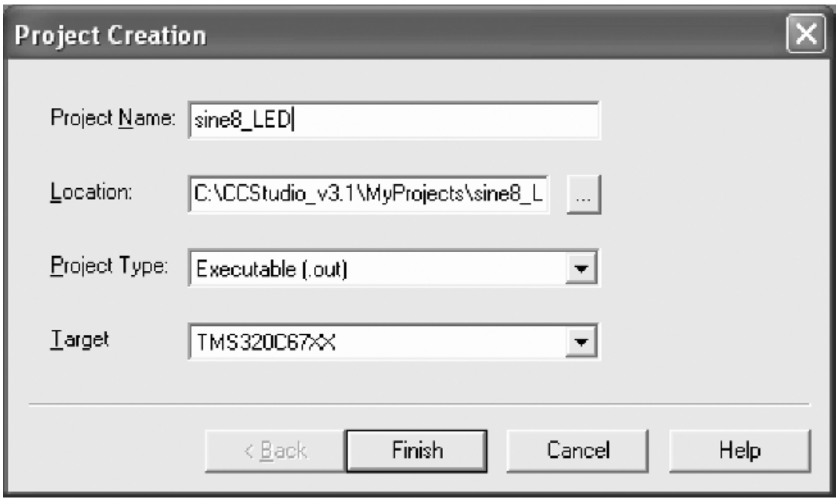
```
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;
```

One cycle of the sinusoidal analog output waveform corresponds to eight output samples and hence the frequency of the sinusoidal analog output waveform is equal to the codec sampling rate (8kHz) divided by eight, that is, 1kHz.

Creating a Project

This section illustrates how to create a project, adding the necessary files to generate an executable file **`sine8_LED.out`**. As supplied on the CD, folder `sine8_LED` contains a suitable project file named `sine8_LED.pjt`. However, for the purposes of gaining familiarity with CCS, this section will illustrate how to create that project file from scratch.

1. **Delete the existing project file `sine8_LED.pjt`** in folder `c:\CCStudio_v3.1\myprojects\sine8_LED`. Do this from outside CCS. Remember, a copy of the file `sine8_LED.pjt` still exists on the CD.
2. **Launch CCS** by double-clicking on its desktop icon.
3. **Create a new project file `sine8_LED.pjt`** by selecting *Project→New* and typing *`sine8_LED`* as the project name, as shown in Figure 1.3. **Set *Target* to *TMS320C67XX*** before clicking on *Finish*. The new project file will be saved in the folder `c:\CCStudio_v3.1\myprojects\`*`sine8_LED`*. The `.pjt` file stores project information on build options, source filenames, and dependencies. The names of the files used by a project are displayed in the *Project View* window, which, by default, appears at the left-hand side of the Code Composer window.
4. **Add the source file `sine8_LED.c` to the project.** `sine8_LED.c` is the top level C source file containing the definition of function `main()`. This source file is stored in the folder `sine8_LED` and must be added to the project if it is to be used to generate the executable file `sine8_LED.out`. Select *Project→Add Files to Project* and look for *Files of Type C Source Files (*.c, *.ccc). Open*,


Project Creation

Project Name: sine8_LED

Location: C:\CCStudio_v3.1\MyProjects\sine8_L

Project Type: Executable (.out)

Target: TMS320C67XX

FIGURE 1.3. CCS *Project Creation* window for project `sine8_LED`.

or double-click on, `sine8_LED.c`. It should appear in the *Project View* window in the *Source* folder.

5. **Add the source file `c6713dskinit.c` to the project.** `c6713dskinit.c` contains the function definitions for a number of low level routines including `comm._poll()` and `output_left_sample()`. This source file is stored in the folder `c:\CCStudio_v3.1\myprojects\Support`. Select *Project→Add Files to Project* and look for *Files of Type C Source Files (*.c, *.ccc). Open*, or double-click on, `c6713dskinit.c`. It should appear in the *Project View* window in the *Source* folder.
6. **Add the source file `vectors__poll.asm` to the project.** `vectors_poll.asm` contains the interrupt service table for the C6713. This source file is stored in the folder `c:\CCStudio_v3.1\myprojects\Support`. Select *Project→Add Files to Project* and look for *Files of Type ASM Source Files (*.a*). Open*, or double-click on, `vectors_poll.asm`. It should appear in the *Project View* window in the *Source* folder.
7. **Add library support files *`rts6700.lib`*, *`dsk6713bsl.lib`*, and *`csl6713.lib`* to the project.** Three more times, select *Project→Add Files to Project* and look for *Files of Type Object and Library Files (*.o*, *.l*)* The three library files are stored in folders `c:\CCStudio_v3.1\c6000\cgtools\lib`, `c:\CCStudio_v3.1\c6000\dsk6713\lib`, and `c:\CCStudio_v3.1\c6000\csl\lib`, respectively. These are the run-time support (for C67x architecture), board support (for C6713 DSK), and chip support (for C6713 processor) library files.
8. **Add the linker command file `c6713dsk.cmd` to the project.** This file is stored in the folder `c:\CCStudio_v3.1\myprojects\Support`. Select *Project→Add Files to Project* and look for *Files of Type Linker Command File (*.cmd;*.lcf)*. Open, or double-click on, `c6713dsk.cmd`. It should then appear in the *Project View* window.
9. No header files will be shown in the *Project View* window at this stage. Selecting *Project→Scan All File Dependencies* will rectify this. You should now be able to see header files `c6713dskinit.h`, `dsk6713.h`, and `dsk6713_aic23.h`, in the *Project View* window.
10. **The *Project View* window in CCS should look as shown in Figure 1.4.** The GEL file `dsk6713.gel` is added automatically when you create the project. It initializes the C6713 DSK invoking the board support library to use the PLL to set the CPU clock to 225 MHz (otherwise the C6713 runs at 50 MHz by default). Any of the files (except the library files) listed in the *Project View* window can be displayed (and edited) by double-clicking on their name in the *Project View* window. You should not add header or include files to the project. They are added to the project automatically when you select *Scan All File Dependencies*. (They are also added when you build the project.)

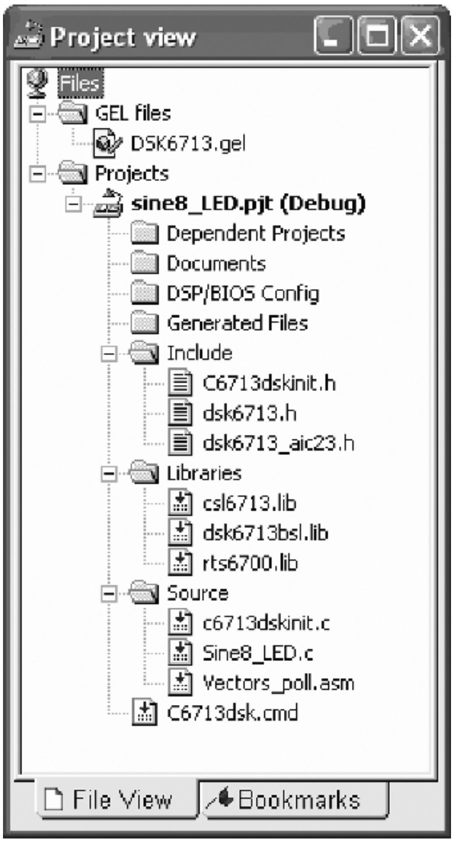


FIGURE 1.4. *Project View* window showing files added at step 10.

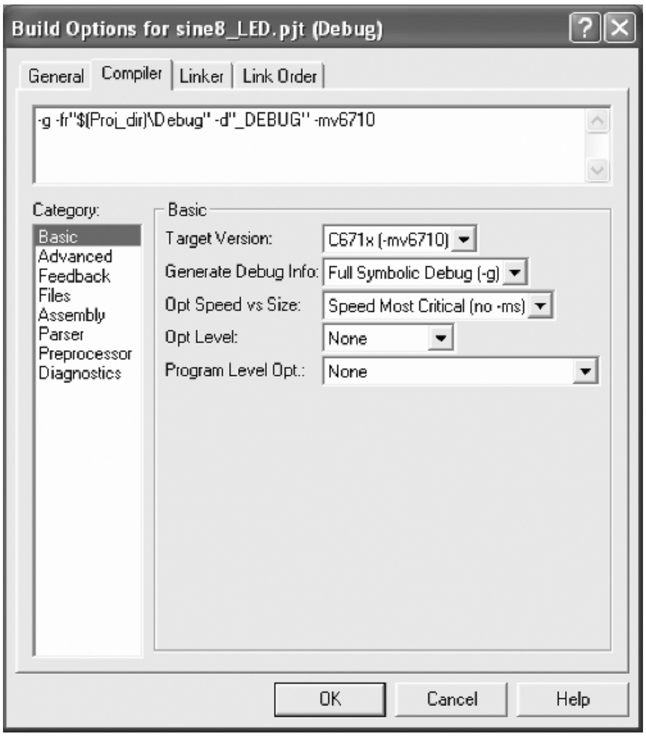
Verify from the *Project View* window that the project (`.pjt`) file, the linker command (`.cmd`) file, the three library (`.lib`) files, the two C source (`.c`) files, and the assembly (`.asm`) file have been added to the project.

Code Generation and Build Options

The code generation tools underlying CCS, that is, C compiler, assembler, and linker, have a number of options associated with each of them. These options must be set appropriately before attempting to build a project. Once set, these options will be stored in the project file.

Setting Compiler Options

Select *Project→Build Options* and click on the *Compiler* tab. Set the following options, as shown in Figures 1.5, 1.6, and 1.7. In the *Basic* category set *Target Version* to *C671x (-mv6710)*. In the *Advanced* category set *Memory Models* to *Far (–mem_*



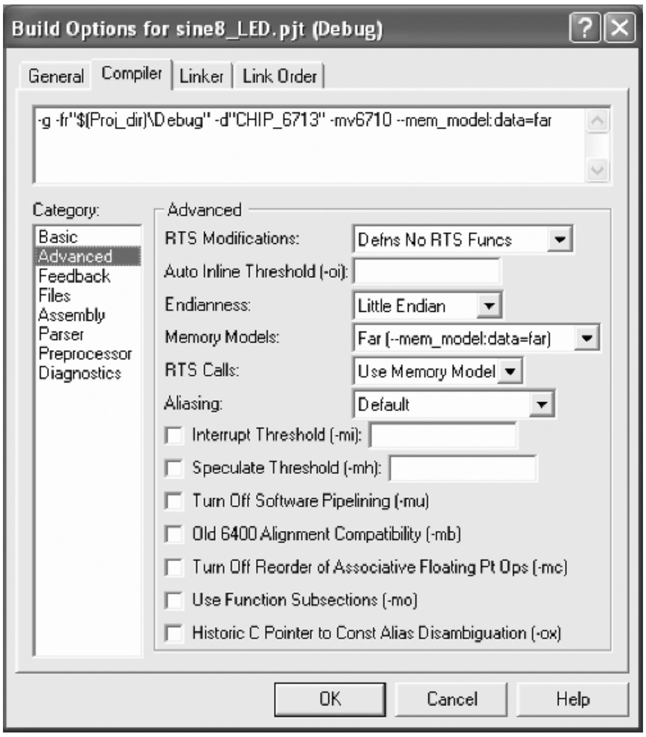
Build Options for sine8_LED.pjt (Debug)

```
-g -fr"$(Proj_dir)\Debug" -d"_DEBUG" -mv6710
```

Basic

Target Version:	C671x (-mv6710)
Generate Debug Info:	Full Symbolic Debug (-g)
Opt Speed vs Size:	Speed Most Critical (no -ms)
Opt Level:	None
Program Level Opt.:	None

FIGURE 1.5. CCS Build Options: Basic compiler settings.



Build Options for sine8_LED.pjt (Debug)

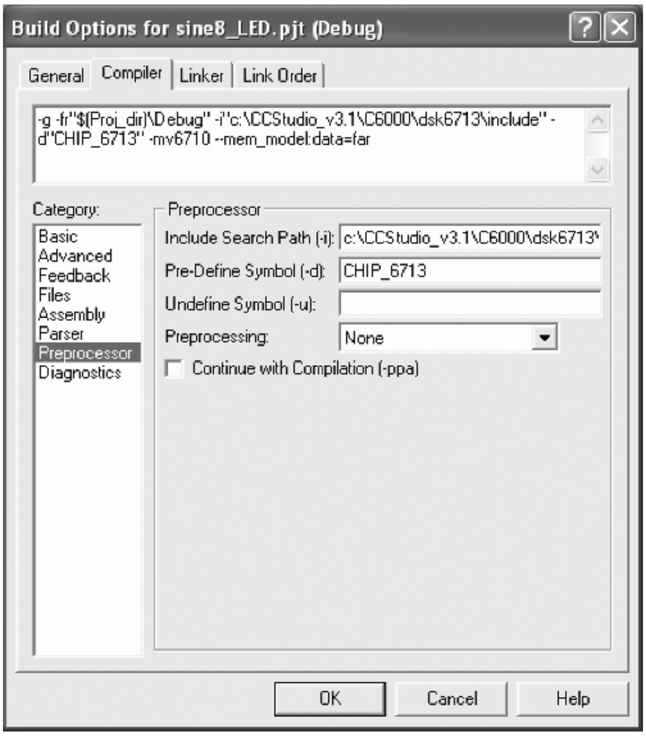
```
-g -fr"$(Proj_dir)\Debug" -d"CHIP_6713" -mv6710 --mem_model:data=far
```

Advanced

RTS Modifications:	Defns No RTS Funcs
Auto Inline Threshold (-oi):	
Endianness:	Little Endian
Memory Models:	Far (--mem_model:data=far)
RTS Calls:	Use Memory Model
Aliasing:	Default

- ☐ Interrupt Threshold (-mi):
- ☐ Speculate Threshold (-mh):
- ☐ Turn Off Software Pipelining (-mu)
- ☐ Old 6400 Alignment Compatibility (-mb)
- ☐ Turn Off Reorder of Associative Floating Pt Ops (-mc)
- ☐ Use Function Subsections (-mo)
- ☐ Historic C Pointer to Const Alias Disambiguation (-ox)

FIGURE 1.6. CCS Build Options: Advanced compiler settings.


Build Options for sine8_LED.pjt (Debug)

General | Compiler | Linker | Link Order

```
-g -fr"$(Proj_dir)\Debug" -i"c:\CCStudio_v3.1\C6000\dsk6713\include" -d"CHIP_6713" -mv6710 --mem_model:data=far
```

Category: Basic, Advanced, Feedback, Files, Assembly, Parser, Preprocessor, Diagnostics

Preprocessor

Include Search Path (-i): c:\CCStudio_v3.1\C6000\dsk6713\

Pre-Define Symbol (-d): CHIP_6713

Undefine Symbol (-u):

Preprocessing: None

☐ Continue with Compilation (-ppa)

OK | Cancel | Help

FIGURE 1.7. CCS Build Options: Preprocessor compiler settings.

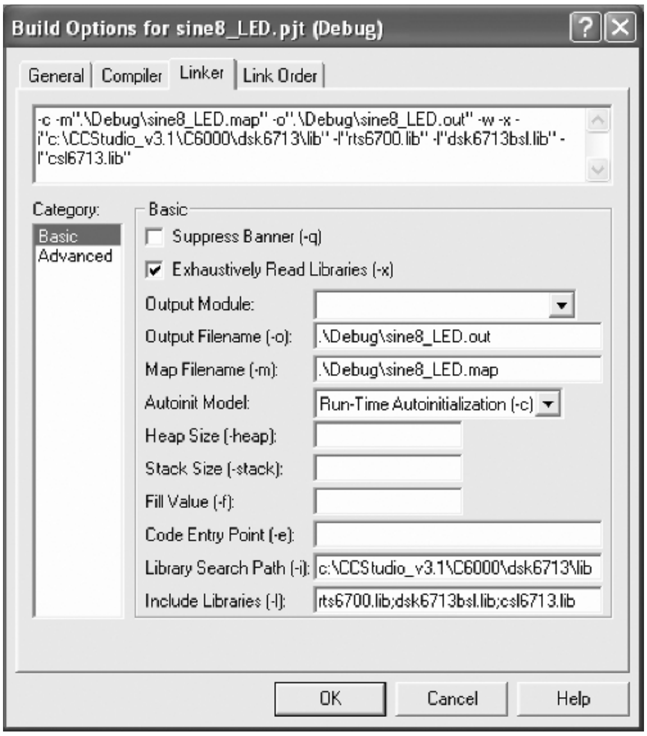
model:data=far). In the *Preprocessor* category set *Pre-Define Symbol* to *CHIP_6713* and *Include Search Path* to *c:\CCStudio_v3.1\C6000\dsk6713\include*. Compiler options are described in more detail in Ref. 28. Click on *OK*.

Setting Linker Options

Click on the *Linker* tab in the *Build Options* window, as shown in Figure 1.8. The *Output Filename* should default to *.\Debug\sine8_LED.out* based on the name of the project file and the *Autoinit Model* should default to *Run-Time Autoinitialization*. Set the following options (all in the *Basic* category). Set *Library Search Path* to *c:\CCStudio_v3.1\C6000\dsk6713\lib* and set *Include Libraries* to *rts6700.lib; dsk6713bsl.lib;csl6713.lib*. The map file can provide useful information for debugging (memory locations of functions, etc.). The `-c` option is used to initialize variables at run time, and the `-o` option is to name the linked executable output file `sine8_LED.out`. Click on *OK*.

Building, Downloading, and Running the Project

The project `sine8_LED` can now be built, and the executable file `sine8_LED.out` can be downloaded to the DSK and run.



```
-c -m".\Debug\sine8_LED.map" -o".\Debug\sine8_LED.out" -w -x -
i"c:\CCStudio_v3.1\C6000\dsk6713\lib" -l"rts6700.lib" -l"dsk6713bsl.lib" -
l"csl6713.lib"
```

Basic

- [ ] Suppress Banner (-q)
- [✓] Exhaustively Read Libraries (-x)

Output Module:	
Output Filename (-o):	.\Debug\sine8_LED.out
Map Filename (-m):	.\Debug\sine8_LED.map
Autoinit Model:	Run-Time Autoinitialization (-c)
Heap Size (-heap):	
Stack Size (-stack):	
Fill Value (-f):	
Code Entry Point (-e):	
Library Search Path (-i):	c:\CCStudio_v3.1\C6000\dsk6713\lib
Include Libraries (-l):	rts6700.lib;dsk6713bsl.lib;csl6713.lib

FIGURE 1.8. CCS Build Options: Basic Linker settings.

1. Build this project as **`sine8_LED`**. Select *Project→Rebuild All*. Or press the toolbar button with the three downward arrows. This compiles and assembles all the C files using `cl6x` and assembles the assembly file `vectors_poll.asm` using `asm6x`. The resulting object files are then linked with the library files using `lnk6x`. This creates an executable file `sine8_LED.out` that can be loaded into the C6713 processor and run. Note that the commands for compiling, assembling, and linking are performed with the Build option. A log file `cc_build_Debug.log` is created that shows the files that are compiled and assembled, along with the compiler options selected. It also lists the support functions that are used. The building process causes all the dependent files to be included (in case one forgets to scan for all the file dependencies). You should see a number of diagnostic messages, culminating in the message "Build Complete, 0 Errors, 0 Warnings, 0 Remarks" appear in an output window in the bottom left-hand side of the CCS window. It is possible that a warning about the Stack Size will have appeared. This can be ignored or can be suppressed by unchecking the *Warn About Output Sections* option in the *Advanced* category of *Linker Build Options*. Alternatively, it can be eliminated by setting the *Stack*

Size option in the *Advanced* category of *Linker Build Options* to a suitable value (e.g., `0x1000`).

Connect to the DSK. Select *Debug→Connect* and check that the symbol in the bottom left-hand corner of the CCS window indicates connection to the DSK.

2. Select *File→Load Program* in order to load `sine8_LED.out`. It should be stored in the folder `c:\CCStudio_v3.1\MyProjects\sine8_LED\Debug`. Select *Debug→Run*. In order to verify that a sinusoidal output waveform with a frequency of 1 kHz is present at both the LINE OUT and HEADPHONE sockets on the DSK, when DIP switch #0 is pressed down, use an oscilloscope connected to the LINE OUT socket and a pair of headphones connected to the HEADPHONE socket.

Editing Source Files Within CCS

Carry out the following actions in order to practice editing source files.

1. Halt execution of the program (if it is running) by selecting *Debug→Halt*.
2. Double-click on the file `sine8_LED.c` in the *Project View* window. This should open a new window in CCS within which the source file is displayed and may be edited.
3. Delete the semicolon in the program statement

```
short gain = 10;
```

4. Select *Debug→Build* to perform an incremental build or use the toolbar button with the two (not three) downward arrows. The incremental build is chosen so that only the C source file `sine8_LED.c` is compiled. Using the Rebuild option (the toolbar button with three downward arrows), files compiled and/or assembled previously would again go through this unnecessary process.
5. Two error messages, highlighted in red, stating

```
"Sine8_LED.c", Line 11: error: expected a ";"
"Sine8_LED.c", Line 23: error: identifier "sine_table" is
undefined
```

should appear in the *Build* window of CCS (lower left). You may need to scroll-up the *Build* window for a better display of these error messages. Double-click on the first highlighted error message line. This should bring the cursor to the section of code where the error occurs. Make the appropriate correction (i.e. replace the semicolon) *Build* again, *Load*, and *Run* the program and verify your previous results.

Monitoring the Watch Window

Ensure that the processor is still running (and that DIP switch #0 is pressed down). Note the message "RUNNING" displayed at the bottom left of CCS. The *Watch* window allows you to change the value of a parameter or to monitor a variable:

1. Select *View→Quick Watch*. Type *gain*, then click on *Add to Watch*. The gain value of 10 set in the program in Figure 1.2 should appear in the Watch window.
2. Change *gain* from 10 to 30 in the *Watch* window. Press enter. Verify that the amplitude of the generated tone has increased (with the processor still running and DIP switch #0 pressed down). The amplitude of the sine wave should have increased from approximately 0.9 V p-p to approximately 2.5 V p-p.

Using a GEL Slider to Control the Gain

The General Extension Language (GEL) is an interpreted language similar to (a subset of) C. It allows you to change the value of a variable (e.g., gain) while the processor is running.

1. Select *File→Load GEL* and load the file gain.gel (in folder sine8_LED). Double-click on the filename *gain.gel* in the *Project View* window to view it within CCS. The file is listed in Figure 1.9. The format of a slider GEL function is

   ```
   slider param_definition( minVal, maxVal, increment,
   pageIncrement, paramName )
   {
    statements
   }
   ```

   where param_definition identifies the slider and is displayed as the name of the slider window, minVal is the value assigned to the GEL variable paramName when the slider is at its lowest level, maxVal is the value assigned to the

```
/*gain.gel GEL slider to vary amplitude of sine wave*/
/*generated by program sine8_LED.c*/

menuitem "Sine Gain"

slider Gain(0,30,4,1,gain_parameter) /*incr by 4, up to 30*/
{
  gain = gain_parameter;             /*vary gain of sine*/
}
```

FIGURE 1.9. Listing of GEL file gain.gel.

GEL variable `paramName` when the slider is at its highest level, increment specifies the incremental change to the value of the GEL variable `paramName` made using the up- or down-arrow keys, and `pageIncrement` specifies the incremental change to the value of the GEL variable `paramName` made by clicking in the slider window.

In the case of `gain.gel`, the statement

```
gain = gain_parameter;
```

assigns the value of the GEL variable `gain_parameter` to the variable `gain` in `program sine8_LED`. The line

```
menuitem "Sine Gain"
```

sets the text that will appear as an option in the CCS *GEL* menu when `gain.gel` is loaded.

2. Select *GEL→Sine Gain→Gain*. This should bring out the slider window shown in Figure 1.10, with the minimum value of 0 set for the gain.
3. Press the up-arrow key three times to increase the gain value from 0 to 12. Verify that the peak-to-peak value of the sine wave generated is approximately 1.05 V. Press the up-arrow key again to continue increasing the slider, incrementing by 4 each time. The amplitude of the sine wave should be about 2.5 V p-p with the value of `gain` set to 30. Clicking in the *Gain* slider window above or below the current position of the slider will increment or decrement its value by 1. The slider can also be dragged up and down. Changes to the value of `gain` made using the slider are reflected in the *Watch* window.

Figure 1.11 shows several windows within CCS for the project `sine8_LED`.

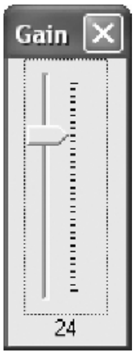
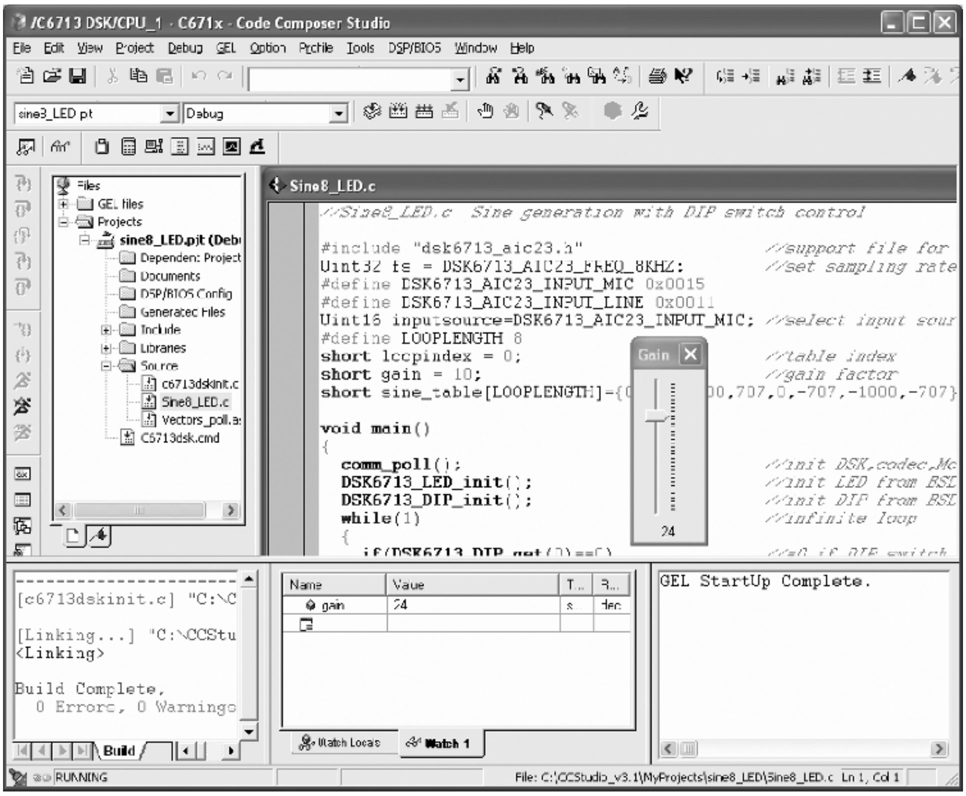


FIGURE 1.10. GEL slider used to vary gain in program `sine8_LED.c`.



```
//Sine8_LED.c  Sine generation with DIP switch control

#include "dsk6713_aic23.h"                          //support file for
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ:                //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; //select input sour
#define LOOPLENGTH 8
short loopindex = 0;                                //table index
short gain = 10;                                    //gain factor
short sine_table[LOOPLENGTH]={0          00,707,0,-707,-1000,-707}

void main()
{
  comm_poll();                                      //init DSK,codec,Mc
  DSK6713_LED_init();                               //init LED from BSL
  DSK6713_DIP_init();                               //init DIP from BSL
  while(1)                                          //infinite loop
  {
    if(DSK6713_DIP_get(0)==0)                       //=0 if DIP switch
```

```
------------------------
[c6713dskinit.c] "C:\C

[Linking...] "C:\CCStu
<Linking>

Build Complete,
  0 Errors, 0 Warnings
```

Name	Value	T...	R...
gain	24	s...	dec

```
GEL StartUp Complete.
```

FIGURE 1.11. CCS windows for project `sin8_LED`, including *Watch* window and GEL slider.

Changing the Frequency of the Generated Sinusoid

There are several different ways in which the frequency of the sinusoid generated by program `sine8_LED.c` can be altered.

1. Change the AIC23 codec sampling frequency from 8 kHz to 16 kHz by changing the line that reads

   ```
   Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;
   ```

   to read

   ```
   Uint32 fs = DSK6713_AIC23_FREQ_16KHZ;
   ```

   Rebuild (use incremental build) the project, load and run the new executable file, and verify that the frequency of the generated sinusoid is 2 kHz. The sampling frequencies supported by the AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96 kHz.

2. Change the number of samples stored in the lookup table to four. By changing the lines that read

```
#define LOOPLENGTH 8
short sine_table[LOOPLENGTH]={0,707,1000,707,0,-707,0,-1000,
-707};
```

to read

```
#define LOOPLENGTH 4
short sine_table[LOOPLENGTH]={0,1000,0,-1000};
```

Verify that the frequency of the sinusoid generated is 2 kHz (assuming an 8-kHz sampling frequency).

Remember that the sinusoid is no longer generated if the DIP switch #0 is not pressed down. A different DIP switch can be used to control whether or not a sinusoid is generated by changing the value of the parameter passed to the functions `DSK6713_DIP_get(), DSK6713_LED_on(), and DSK6713_LED_off()`. Suitable values are 0, 1, 2, and 3.

Two sliders can readily be used, one to change the gain and the other to change the frequency. A different signal frequency can be generated, by changing the incremental changes applied to the value of `loopindex` within the C program (e.g., stepping through every two points in the table). When you exit CCS after you build a project, all changes made to the project can be saved. You can later return to the project with the status as you left it before. For example, when returning to the project, after launching CCS, select *Project→Open* to open an existing project such as `sine8_LED.pjt` (with all the necessary files for the project already added).

Example 1.2: Generation of Sinusoid and Plotting with CCS (`sine8_buf`)

This example generates a sinusoidal analog output signal using eight precalculated and prestored sample values. However, it differs fundamentally from `sine8_LED` in that its operation is based on the use of interrupts. In addition, it uses a buffer to store the `BUFFERLENGTH` most recent output samples. It is used to illustrate the capabilities of CCS for plotting data in both time and frequency domains.

All the files necessary to build and run an executable file `sine8_BUF.out` are stored in folder `sine8_buf`. Program file *`sine8_buf.c`* is listed in Figure 1.12. Because a project file `sine8_buf.pjt` is supplied, there is no need to create a new

```
//sine8_buf.c sine generation with output stored in buffer

#include "DSK6713_AIC23.h"                  //codec support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;          //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; // select input
#define LOOPLENGTH 8
#define BUFFERLENGTH 256
int loopindex = 0;                          //table index
int bufindex = 0;                           //buffer index
short sine_table[LOOPLENGTH]={0,707,1000,707,0,-707,-1000,-707};
int out_buffer[BUFFERLENGTH];               //output buffer
short gain = 10;
interrupt void c_int11()                //interrupt service routine
  short out_sample;

  out_sample = sine_table[loopindex++]*gain;
  output_left_sample(out_sample);           //output sample value
  out_buffer[bufindex++] = out_sample;      //store in buffer
  if (loopindex >= LOOPLENGTH) loopindex = 0; //check end table
  if (bufindex >= BUFFERLENGTH) bufindex = 0; //check end buffer
  return;
}                                           //return from interrupt

void main()
{
  comm_intr();                              //initialise DSK
  while(1);                                 //infinite loop
}
```

FIGURE 1.12. Listing of program sine8_buf.c.

project file, add files to it, or alter compiler and linker build options. In order to build, download and run program sine8_buf.c.

1. Close any open projects in CCS.
2. Open project *sine8_buf.pjt* by selecting *Project→Open* and double-clicking on file sine8_buf.pjt in folder sine8_buf. Because this program uses interrupt-driven input/output rather than polling, the file vectors_intr.asm is used in place of vectors_poll.asm. The interrupt service table specified in vectors_intr.asm associates the interrupt service routine c_int11() with hardware interrupt INT11, which is asserted by the AIC23 codec on the DSK at each sampling instant.

Within function main(), function comm_intr() is used in place of comm_poll(). This function is defined in file c6713dskinit.c and is described in more detail in Chapter 2. Essentially, it initializes the DSK hardware, including the AIC23 codec, such that the codec sampling rate is set according to the value of the variable `fs` and the codec interrupts the processor at every sampling instant. The statement `while(1)` in function `main()` creates an infinite loop, during which the processor waits for interrupts. On interrupt, execution proceeds to the interrupt service routine (ISR) `c_int11()`, which reads a new sample value from the array `sine_table` and writes it both to the array `out_buffer` and to the DAC using function `output_left_sample()`. Interrupts are discussed in more detail in Chapter 3.

Build this project as **`sine8_buf`**. Load and run the executable file *`sine8_buf.out`* and verify that a 1-kHz sinusoid is generated at the LINE OUT and HEADPHONE sockets (as in Example 1.1).

Graphical Displays in CCS

The array `out_buffer` is used to store the `BUFFERLENGTH` most recently output sample values. Once program execution has been halted, the data stored in `out_buffer` can be displayed graphically in CCS.

1. Select *View→Graph→Time/Frequency* and set the *Graph Property Dialog* properties as shown in Figure 1.13a. Figure 1.13b shows the resultant *Graphical Display* window.
2. Figure 1.14a shows the *Graph Property Dialog* window that corresponds to the frequency domain representation of the contents of `out_buffer` shown in Figure 1.14b. The spike at 1 kHz represents the frequency of the sinusoid generated by program `sine8_buf.c`.

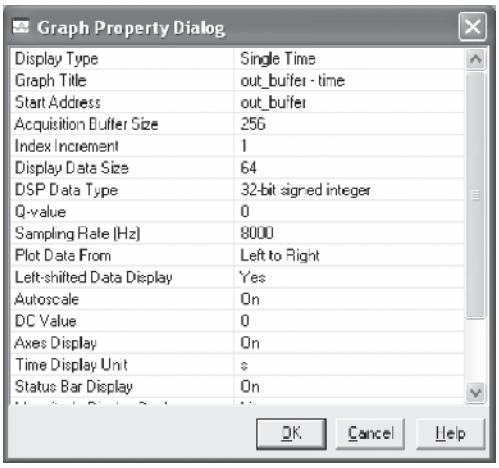
Viewing and Saving Data from Memory into File

To view the contents of `out_buffer`, select *View→Memory*. Specify `out_buffer` as the *Address* and select *32-bit Signed Integer* as the *Format*, as shown in Figure 1.15a. The resultant *Memory* window is shown in Figure 1.15b.

To save the contents of `out_buffer` to a file, select *File→Data→Save*. Save the file as `sine8_buf.dat`, selecting data type *Integer*, in the folder `sine8_buf`. In the *Storing Memory into File* window, specify `out_buffer` as the *Address* and a *Length* of 256. The resulting file is a text file and you can plot this data using other applications (e.g., MATLAB). Although the values stored in array `sine_table` and passed to function `output_left_sample()` are 16-bit signed integers, array `out_buffer` is declared as type `int` (32-bit signed integer) in program `sine8_buf.c` to allow for the fact that there is no 16-bit Signed Integer data type option in the *Save Data* facility in CCS.

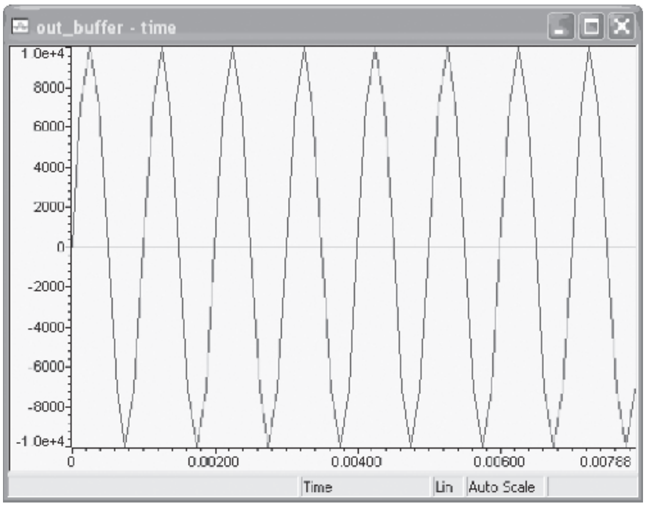
Example 1.3: Dot Product of Two Arrays (`dotp4`)

This example illustrates the use of breakpoints and single stepping within CCS. In addition, it illustrates the use of Code Composer's *Profile Clock* in order to estimate the time taken to execute a section of code.



Display Type	Single Time
Graph Title	out_buffer - time
Start Address	out_buffer
Acquisition Buffer Size	256
Index Increment	1
Display Data Size	64
DSP Data Type	32-bit signed integer
Q-value	0
Sampling Rate (Hz)	8000
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On

(a)

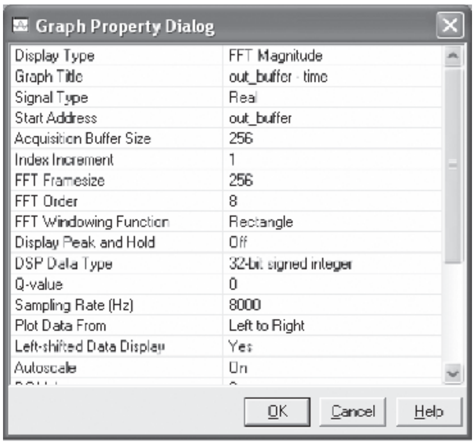


(b)

FIGURE 1.13. (a) *Graph Property* window and (b) Time domain plot of data stored in `out_buffer`.

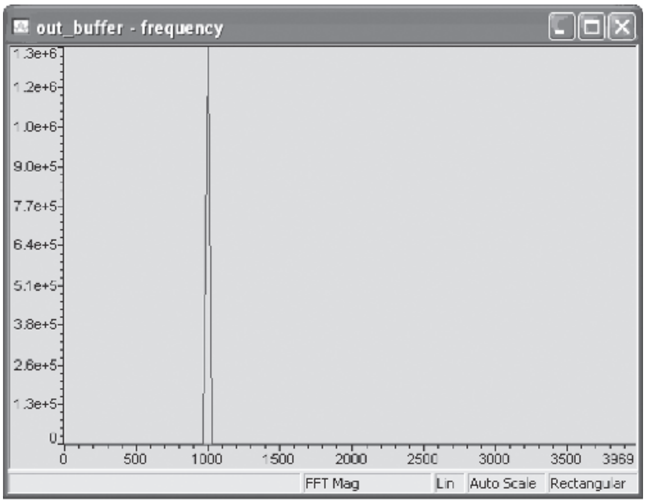
Multiply/accumulate is a very important operation in digital signal processing. It is a fundamental part of digital filtering, correlation, and fast Fourier transform algorithms. Since the multiplication operation is executed so commonly and is essential for most digital signal processing algorithms, it is important that it executes in a single instruction cycle. The C6713 and C6416 processors can perform two multiply/accumulate operations within a single instruction cycle.

The C source file `dotp4.c`, listed in Figure 1.16, calculates the dot products of two arrays of integer values. The first array is initialized using the four values 1, 2, 3, and 4, and the second array using the four values 0, 2, 4, and 6. The dot product is $(1 \times 0) + (2 \times 2) + (3 \times 4) + (4 \times 6) = 40$.



Display Type	FFT Magnitude
Graph Title	out_buffer - time
Signal Type	Real
Start Address	out_buffer
Acquisition Buffer Size	256
Index Increment	1
FFT Framesize	256
FFT Order	8
FFT Windowing Function	Rectangle
Display Peak and Hold	Off
DSP Data Type	32-bit signed integer
Q-value	0
Sampling Rate (Hz)	8000
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On

(a)



(b)

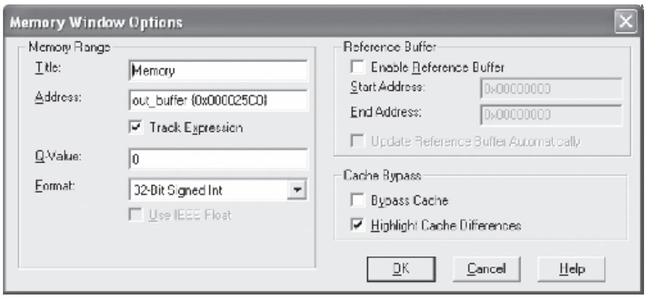
FIGURE 1.14. (a) *Graph Property* window and (b) Frequency domain plot of data stored in `out_buffer`.

The program can readily be modified to handle larger arrays. No real-time input or output is used in this example, and so the real-time support files `c6713dskinit.c` and `vectors_intr.asm` are not needed.

Build this project as **`dotp4`** ensuring that the following files are included in the project:

1. `dotp4.c`: C source file.
2. `6713dsk.cmd`: generic linker command file.
3. `rts6700.lib`: library file.

The *Project View* window should appear as shown in Figure 1.17.



Memory Window Options

Memory Range

Title: Memory

Address: out_buffer (0x000025C0)

☑ Track Expression

Q-Value: 0

Format: 32-Bit Signed Int

☐ Use IEEE Float

Reference Buffer

☐ Enable Reference Buffer

Start Address: 0x00000000

End Address: 0x00000000

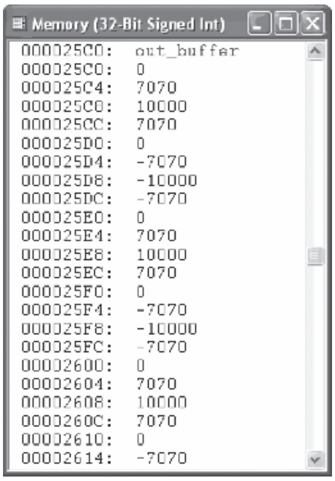
☐ Update Reference Buffer Automatically

Cache Bypass

☐ Bypass Cache

☑ Highlight Cache Differences

(a)



Memory (32-Bit Signed Int)

```
000025C0:  out_buffer
000025C0:  0
000025C4:  7070
000025C8:  10000
000025CC:  7070
000025D0:  0
000025D4:  -7070
000025D8:  -10000
000025DC:  -7070
000025E0:  0
000025E4:  7070
000025E8:  10000
000025EC:  7070
000025F0:  0
000025F4:  -7070
000025F8:  -10000
000025FC:  -7070
00002600:  0
00002604:  7070
00002608:  10000
0000260C:  7070
00002610:  0
00002614:  -7070
```

(b)

FIGURE 1.15. (a) *Memory* window settings and (b) *Memory* window view of data stored in `out_buffer`.

Implementing a Variable Watch

1. Select *Project→Build Options* and verify that the *Basic Compiler* settings are as shown in Figure 1.18. In this example it is important to **ensure that the optimization is disabled** (*Opt Level None*).
2. Build the project by clicking on the toolbar button with the three downward arrows (or select *Project→Build*). Load the executable file `dotp4.out`.
3. Select *View→Quick Watch*. Type *`sum`* to watch the variable `sum`, and click on *Add to Watch*. The message "identifier not found: sum" should be displayed in the *Watch* window. The variable `sum` is declared locally in function `dotp()` and until that function is called it does not exist.
4. Set a breakpoint at the line of code

   ```
   sum += a[i] * b[i];
   ```

   by clicking on that line in the source file `dotp4.c` and then either right-clicking and selecting *Toggle Software Breakpoint*, or clicking on the *Toggle Breakpoint* toolbar button. A red dot should appear to the left of that line of code.

```
//dotp4.c dot product of two vectors

int dotp(short *a, short *b, int ncount); //function prototype
#include <stdio.h>              //for printf
#define count 4                 //# of data in each array
short x[count] = {1,2,3,4};     //declaration of 1st array
short y[count] = {0,2,4,6};     //declaration of 2nd array

main()
{
  int result = 0;               //result sum of products

  result = dotp(x, y, count);   //call dotp function
  printf("result = %d (decimal) \n", result); //print result
}

int dotp(short *a, short *b, int ncount) //dot product function
{
  int i;
  int sum = 0;
  for (i = 0; i < ncount; i++)
    sum += a[i] * b[i];         //sum of products
  return(sum);                  //return sum as result
}
```

FIGURE 1.16. Listing of program `dotp4.c`.

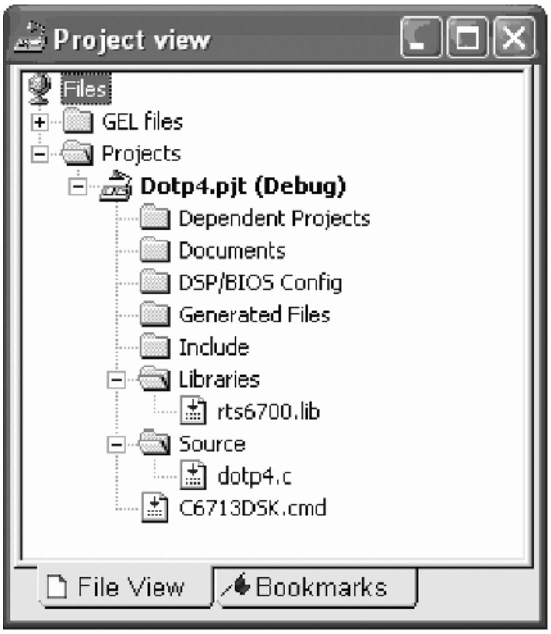
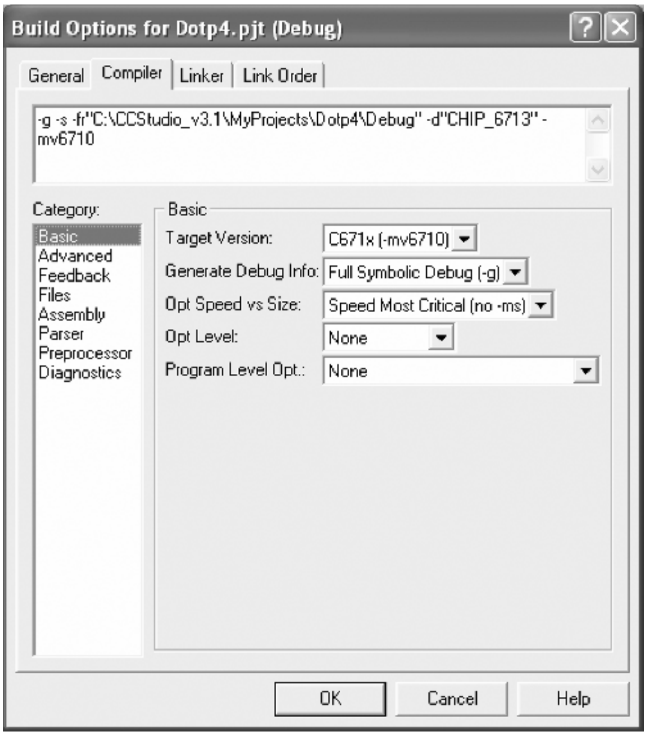


FIGURE 1.17. *Project View* window for project `dotp4`.



```
-g -s -fr"C:\CCStudio_v3.1\MyProjects\Dotp4\Debug" -d"CHIP_6713" -
mv6710
```

Category:
Basic
Advanced
Feedback
Files
Assembly
Parser
Preprocessor
Diagnostics

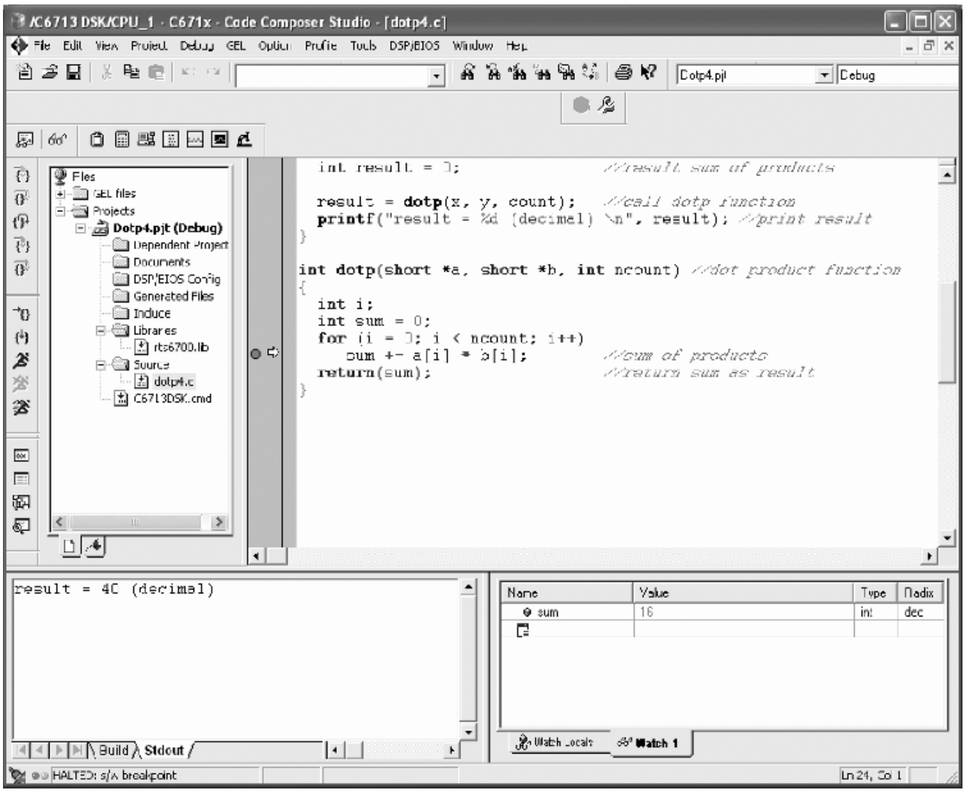
Basic
Target Version: C671x (-mv6710)
Generate Debug Info: Full Symbolic Debug (-g)
Opt Speed vs Size: Speed Most Critical (no -ms)
Opt Level: None
Program Level Opt.: None

FIGURE 1.18. *Build Options* for project `dotp4`.

5. Select *Debug→Run* (or use the "running man" toolbar button). The program will execute up to, but not including, the line of code at which the breakpoint has been set. A yellow arrow will appear to the left of that line of code. At this point, a value of 0 for the variable `sum` should appear in the *Watch* window. `sum` is a variable that is local to function `dotp()`. Now that the function is being executed, the variable exists and its value can be displayed.
6. Continue program execution by selecting *Debug→Step Into*, or by using function key F11. Continue to single-step and watch the variable `sum` in the *Watch* window change in value through 0, 4, 16, and 40 (See Figure 1.19.).
7. Once the value of the variable `sum` has reached 40, select *Debug→Run* in order to complete execution of the program, and verify that the value returned by function `dotp()` is displayed as

```
result = 40 (decimal)
```

in the *Stdout* window. At this point, the message "identifier not found: sum" should be displayed in the *Watch* window again, reflecting the fact that execution of function `dotp()` has ended and that the local variable `sum` no longer exists.



```
  int result = 0;                  //result sum of products

  result = dotp(x, y, count);      //call dotp function
  printf("result = %d (decimal) \n", result); //print result
}

int dotp(short *a, short *b, int ncount) //dot product function
{
  int i;
  int sum = 0;
  for (i = 0; i < ncount; i++)
     sum += a[i] * b[i];           //sum of products
  return(sum);                     //return sum as result
}
```

```
result = 40 (decimal)
```

Name	Value	Type	Radix
sum	16	int	dec

FIGURE 1.19. Various windows associated with program `dotp4.c`.

The `printf()` function is useful for debugging but its use should be avoided in real-time programs since it takes over 6000 instruction cycles to execute.

Animating

1. Select *File→Reload Program* to reload the executable file `dotp4.out` (alternatively, select *Debug→Restart*). After the executable file is loaded, or following restart, the program counter is set to the address labeled `c_int00`. This can be verified by looking at the *Disassembly* window.
2. The breakpoint set previously should still be set at the same line of code as before. Select *Debug→Animate* and watch the value of the variable *`sum`* displayed in the *Watch* window change. The speed of animation can be controlled by selecting *Option→Customize→Animate Speed* (by default, the maximum speed setting of 0 seconds is set).

Estimating Execution Time for Function `dotp()` Using the Profile Clock

The time taken to execute function `dotp()` can be estimated using Code Composer's *Profile Clock*.

1. Open project `dotp4.pjt`.
2. Select *Project→Build Options*. In the *Compiler* tab in the *Basic* category set the *Opt Level* to *none*.
3. Select *Project→Build* and then *File→Load Program* in order to create and load file `dotp4.out`.
4. Open source file `dotp4.c` and clear all breakpoints. Set breakpoints at the lines

   ```
   result = dotp(x, y, count);
   ```

   and

   ```
   printf("result = %d (decimal) \n", result);
   ```

5. Select *Profile→Clock→Enable*.
6. Select *Profile→Clock View*. A small clock icon and the number of processor instruction cycles that the *Profile Clock* has counted should appear in the bottom right-hand corner of the Code Composer window.
7. Run the program. It should halt at the first breakpoint.
8. Reset the *Profile Clock* by double-clicking on its icon in the bottom right-hand corner of the Code Composer window.
9. Run the program. It should stop at the second breakpoint.

The number of instruction cycles counted by the *Profile Clock* between the two breakpoints, that is, during execution of function `dotp()`, should be displayed next to the icon. On a 225-MHz C6713 processor, each instruction cycle takes 4.44 ns. Repeat the experiment having set the compiler optimization level to *Function (–o2)* and you should see a reduction in the number of instruction cycles used by function `dotp()` by a factor of approximately 2. Using breakpoints and the *Profile Clock* can give an indication of the execution times of sections of program but it does not always work with higher levels of compiler optimization, for example, *File (-o3)*. More detailed profiling of program execution can be achieved using a simulator.

1.6 SUPPORT FILES

The support files *`c6713dskinit.c`*, *`vectors_intr.asm`* or *`vectors_poll.asm`*, and *`c6713dsk.cmd`* are used by nearly all of the examples in this book.

1.6.1 Initialization/Communication File (*`c6713dskinit.c`*)

Source file *`c6713dskinit.c`*, supplied on the CD accompanying this book and listed in Figure 1.20, contains the definitions of a number of functions used to initialize

```
//c6713dskinit.c
//includes functions from TI in the C6713 CSL and C6713DSK BSL

#include "C6713dskinit.h"
#define using_bios
extern Uint32 fs;             //sampling frequency
extern Uint16 inputsource;    //input source (MIC or LINE)

void c6713_dsk_init()         //initialize DSK
{
  DSK6713_init();             //BSL routine to init DSK

  hAIC23_handle=DSK6713_AIC23_openCodec(0, &config);
  DSK6713_AIC23_setFreq(hAIC23_handle, fs); //set sampling rate
  // choose MIC or LINE IN on AIC23
  DSK6713_AIC23_rset(hAIC23_handle, 0x0004, inputsource);
  MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData);
  MCBSP_start(DSK6713_AIC23_DATAHANDLE, MCBSP_XMIT_START |
              MCBSP_RCV_START | MCBSP_SRGR_START |
              MCBSP_SRGR_FRAMESYNC, 220); //restart data channel
}

void comm_poll()              //for communication using polling
{
  poll=1;                     //1 if using polling
  c6713_dsk_init();           //init DSP and codec
}

void comm_intr()              //for communication using interrupt
{
  poll=0;                     //0 since not polling
  IRQ_globalDisable();        //globally disable interrupts
  c6713_dsk_init();           //init DSP and codec
  CODECEventId=MCBSP_getXmtEventId(DSK6713_AIC23_codecdatahandle);

#ifndef using_bios            //if not using DSP/BIOS
  IRQ_setVecs(vectors);       //use interrupt vector table
#endif                        //set up in vectors_intr.asm

  IRQ_map(CODECEventId, 11); //map McBSP1 Xmit to INT11
  IRQ_reset(CODECEventId);    //reset codec INT 11
  IRQ_globalEnable();         //globally enable interrupts
  IRQ_nmiEnable();            //enable NMI interrupt
  IRQ_enable(CODECEventId);   //enable CODEC eventXmit INT11

  output_sample(0);           //start McBSP by outputting a sample
}

void output_sample(int out_data) //output to both channels
{
  short CHANNEL_data;

  AIC_data.uint=0;            //clear data structure
```

FIGURE 1.20. Listing of support file `c6713dskinit.c`.

```
  AIC_data.uint=out_data;    //write 32-bit data

//The existing interface defaults to right channel.
//To default instead to the left channel and use
//output_sample(short), left and right channels are swapped.
//In main source program use LEFT 0 and RIGHT 1
//(opposite of what is used here)

  CHANNEL_data=AIC_data.channel[RIGHT]; //swap channels
  AIC_data.channel[RIGHT]=AIC_data.channel[LEFT];
  AIC_data.channel[LEFT]=CHANNEL_data;
  // if polling, wait for ready to transmit
  if (poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));
  // write data to AIC23 via MCBSP
  MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);
}

void output_left_sample(short out_data) //output to left channel
{
  AIC_data.uint=0;                  //clear data structure
  AIC_data.channel[LEFT]=out_data; //write 16-bit data
  // if polling, wait for ready to transmit
  if (poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));
  // write data to AIC23 via MCBSP
  MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);
}

void output_right_sample(short out_data)//output to right channel
{
  AIC_data.uint=0;                   //clear data structure
  AIC_data.channel[RIGHT]=out_data; //write 16-bit data
  // if polling, wait for ready to transmit
  if (poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));
  // write data to AIC23 via MCBSP
  MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);
}

Uint32 input_sample()        //input from both channels
{
  short CHANNEL_data;

  // if polling, wait for ready to receive
  if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));
  //read data from AIC23 via MCBSP
  AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);

  //Swap left and right channels (see comments in output_sample())
  CHANNEL_data=AIC_data.channel[RIGHT]; //swap channels
  AIC_data.channel[RIGHT]=AIC_data.channel[LEFT];
  AIC_data.channel[LEFT]=CHANNEL_data;
  return(AIC_data.uint);
}
```

FIGURE 1.20. (*Continued*)

```
short input_left_sample()    //input from left channel
{
  // if polling, wait for ready to receive
  if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));
  //read data from AIC23 via MCBSP
  AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);
  return(AIC_data.channel[LEFT]); //return left channel data
}
short input_right_sample()   //input from right channel
{
  // if polling, wait for ready to receive
  if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));
  //read data from AIC23 via MCBSP
  AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);
  return(AIC_data.channel[RIGHT]); //return right channel data
}
```

FIGURE 1.20. (*Continued*)

the DSK. Calls are made from these functions to lower level functions provided with CCS in the board support library (BSL) and chip support library (CSL) files `dsk6713bsl.lib` and `csl6713.lib`.

Functions *`comm_intr()`* and *`comm_poll()`* initialize communications between the C6713 processor and the AIC23 codec for either interrupt-driven or polling-based input and output. In the case of interrupt-driven input and output, interrupt #11 (INT11), generated by the codec via the serial port (McBSP), is configured and enabled (selected). The nonmaskable interrupt bit must be enabled as well as the global interrupt enable (GIE) bit.

Functions *`input_sample()`*, *`input_left_sample()`*, *`input_right_sample()`*, *`output_sample()`*, *`output_left_sample()`*, and *`input_right_sample()`* are used to read and write data to and from the codec. In the case of polling-based input and output, these functions wait until the next sampling instant (determined by the codec) before reading or writing, using the lower level functions `MCBSP_read()` or `MCBSP_write()`. They do this by polling (testing) the receive ready (RRDY) or transmit ready (XRDY) bits of the McBSP control register (SPCR). In the case of interrupt-driven input and output, the processor is interrupted by the codec at each sampling instant and when either *`input_sample()`* or *`output_sample()`* is called from within the interrupt service routine, reading or writing proceeds without RRDY or XRDY being tested. Interrupts are discussed further in Chapter 3.

1.6.2 Header File (*`c6713dskinit.h`*)

The corresponding header support file *`c6713dskinit.h`* contains function prototypes as well as initial settings for the control registers of the AIC23 codec. Nearly all of the example programs in this book use the same AIC23 control register settings. However, two codec parameters—namely, sampling frequency and selection of ADC input (LINE IN or MIC IN)—are changed more often, from one program example to another, and for that reason the following mechanism has been adopted. During initialization of the DSK (in function `dsk_init()`, defined in file `c6713dskinit.c`), the AIC23 codec control registers are initialized using the `DSK6713_AIC23_Config` type data structure `config` defined in header file `c6713dskinit.h`. Immediately following this initialization, two functions `DSK6713_AIC23_setFreq()` and `DSK6713_AIC23_rset()` are called and these set the sampling frequency and select the input source according to the values of the variables `fs` and `inputsource`. These values are set in the first few lines of every top level source file; for example,

```
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; //select input source
```

In this way, the sampling frequency and input source can be changed without having to edit either `c6713sdkinit.h` or `c6713dskinit.c`. (See Figure 1.21.)

1.6.3 Vector Files (`vectors_intr.asm, vectors_poll.asm`)

To make use of interrupt INT11, a branch instruction (jump) to the interrupt service routine (ISR) `c_int11()` defined in a C program, for example, `sine8_buf.c`, must be placed at the appropriate point in the interrupt service table (IST). Assembly language file `vectors_intr.asm`, which sets up the IST, is listed in Figure 1.22. Note the underscore preceding the name of the routine or function being called. By convention, this indicates a C function.

For a polling-based program, file `vectors_poll.asm` is used, in place of `vectors_intr.asm`. The main difference between these files is that there is no branch to `c_int11()` in the IST set up by `vectors_poll.asm`. Common to both files is a branch to `c_int00()`, the start of a C program, associated with reset. (See Figure 1.23.)

1.6.4 Linker Command File (`c6713dsk.cmd`)

Linker command file *`C6713dsk.cmd`* is listed in Figure 1.24. It specifies the memory configuration of the internal and external memory available on the DSK and the mapping of sections of code and data to absolute addresses in that memory. For example, the *`.text`* section, produced by the C compiler, is mapped into IRAM, that is, the internal memory of the C6713 digital signal processor, starting at address `0x00000220`. The section `.vectors` created by `vectors_intr.asm` or by `vectors_poll.asm` is mapped into IVECS, that is, internal memory starting at address

```
/*c6713dskinit.h include file for c6713dskinit.c */

#include "dsk6713.h"\
#include "dsk6713_aic23.h"

#define LEFT  1
#define RIGHT 0

union {
        Uint32 uint;
        short channel[2];
        } AIC_data;

extern far void vectors();       //external function

static Uint32 CODECEventId, poll;

// needed to modify the BSL data channel McBSP configuration
MCBSP_Config AIC23CfgData = {
        MCBSP_FMKS(SPCR, FREE, NO)              |
        MCBSP_FMKS(SPCR, SOFT, NO)              |
        MCBSP_FMKS(SPCR, FRST, YES)             |
        MCBSP_FMKS(SPCR, GRST, YES)             |
        MCBSP_FMKS(SPCR, XINTM, XRDY)           |
        MCBSP_FMKS(SPCR, XSYNCERR, NO)          |
        MCBSP_FMKS(SPCR, XRST, YES)             |
        MCBSP_FMKS(SPCR, DLB, OFF)              |
        MCBSP_FMKS(SPCR, RJUST, RZF)            |
        MCBSP_FMKS(SPCR, CLKSTP, DISABLE)       |
        MCBSP_FMKS(SPCR, DXENA, OFF)            |
        MCBSP_FMKS(SPCR, RINTM, RRDY)           |
        MCBSP_FMKS(SPCR, RSYNCERR, NO)          |
        MCBSP_FMKS(SPCR, RRST, YES),

        MCBSP_FMKS(RCR, RPHASE, SINGLE)         |
        MCBSP_FMKS(RCR, RFRLEN2, DEFAULT)       |
        MCBSP_FMKS(RCR, RWDLEN2, DEFAULT)       |
        MCBSP_FMKS(RCR, RCOMPAND, MSB)          |
        MCBSP_FMKS(RCR, RFIG, NO)               |
        MCBSP_FMKS(RCR, RDATDLY, 0BIT)          |
        MCBSP_FMKS(RCR, RFRLEN1, OF(0))         |
        MCBSP_FMKS(RCR, RWDLEN1, 32BIT)         |
        MCBSP_FMKS(RCR, RWDREVRS, DISABLE),

        MCBSP_FMKS(XCR, XPHASE, SINGLE)         |
        MCBSP_FMKS(XCR, XFRLEN2, DEFAULT)       |
        MCBSP_FMKS(XCR, XWDLEN2, DEFAULT)       |
```

FIGURE 1.21. Listing of support header file c6713dskinit.h.

```
        MCBSP_FMKS(XCR, XCOMPAND, MSB)          |
        MCBSP_FMKS(XCR, XFIG, NO)               |
        MCBSP_FMKS(XCR, XDATDLY, 0BIT)          |
        MCBSP_FMKS(XCR, XFRLEN1, OF(0))         |
        MCBSP_FMKS(XCR, XWDLEN1, 32BIT)         |
        MCBSP_FMKS(XCR, XWDREVRS, DISABLE),

        MCBSP_FMKS(SRGR, GSYNC, DEFAULT)        |
        MCBSP_FMKS(SRGR, CLKSP, DEFAULT)        |
        MCBSP_FMKS(SRGR, CLKSM, DEFAULT)        |
        MCBSP_FMKS(SRGR, FSGM, DEFAULT)         |
        MCBSP_FMKS(SRGR, FPER, DEFAULT)         |
        MCBSP_FMKS(SRGR, FWID, DEFAULT)         |
        MCBSP_FMKS(SRGR, CLKGDV, DEFAULT),

        MCBSP_MCR_DEFAULT,
        MCBSP_RCER_DEFAULT,
        MCBSP_XCER_DEFAULT,

        MCBSP_FMKS(PCR, XIOEN, SP)              |
        MCBSP_FMKS(PCR, RIOEN, SP)              |
        MCBSP_FMKS(PCR, FSXM, EXTERNAL)         |
        MCBSP_FMKS(PCR, FSRM, EXTERNAL)         |
        MCBSP_FMKS(PCR, CLKXM, INPUT)           |
        MCBSP_FMKS(PCR, CLKRM, INPUT)           |
        MCBSP_FMKS(PCR, CLKSSTAT, DEFAULT)      |
        MCBSP_FMKS(PCR, DXSTAT, DEFAULT)        |
        MCBSP_FMKS(PCR, FSXP, ACTIVEHIGH)       |
        MCBSP_FMKS(PCR, FSRP, ACTIVEHIGH)       |
        MCBSP_FMKS(PCR, CLKXP, FALLING)         |
        MCBSP_FMKS(PCR, CLKRP, RISING)
};

DSK6713_AIC23_Config config = {                                 \
0x0017, /* Set-Up Reg 0     Left line in volume control */      \
        /* LRS     0        simultaneous l/r volume: disabled */\
        /* LIM     0        left line input mute: disabled */   \
        /* XX      00       reserved */                         \
        /* LIV     10111    left line input volume: 0 dB */     \
                                                                \
0x0017, /* Set-Up Reg 1     Right line in volume control */     \
        /* RLS     0        simultaneous r/l volume: disabled */\
        /* RIM     0        right line input mute: disabled */  \
        /* XX      00       reserved */                         \
        /* RIV     10111    right line input volume: 0 dB */    \
                                                                \
```

FIGURE 1.21. (*Continued*)

```
0x01f9, /* Set-Up Reg 2     Left channel headphone volume */    \
        /* LRS      1        simultaneous l/r volume: enabled */ \
        /* LZC      1        zero-cross detect: enabled */       \
        /* LHV      1111001  left headphone volume: 0 dB */      \
                                                                 \
0x01f9, /* Set-Up Reg 3     Right channel headphone volume */   \
        /* RLS      1        simultaneous r/l volume: enabled */ \
        /* RZC      1        zero-cross detect: enabled */       \
        /* RHV      1111001  right headphone volume: 0 dB */     \
                                                                 \
0x0015, /* Set-Up Reg 4     Analog audio path control */        \
        /* X        0        reserved */                         \
        /* STA      00       sidetone attenuation: -6 dB */      \
        /* STE      0        sidetone: disabled */               \
        /* DAC      1        DAC: selected */                    \
        /* BYP      0        bypass: off */                      \
        /* INSEL    0        input select for ADC: line */       \
        /* MICM     0        microphone mute: disabled */        \
        /* MICB     1        microphone boost: enabled */        \
                                                                 \
0x0000, /* Set-Up Reg 5     Digital audio path control */       \
        /* XXXXX    00000    reserved */                         \
        /* DACM     0        DAC soft mute: disabled */          \
        /* DEEMP    00       deemphasis control: disabled */     \
        /* ADCHP    0        ADC high-pass filter: disabled */   \
                                                                 \
0x0000, /* Set-Up Reg 6     Power down control */               \
        /* X        0        reserved */                         \
        /* OFF      0        device power: on (i.e. not off) */  \
        /* CLK      0        clock: on */                        \
        /* OSC      0        oscillator: on */                   \
        /* OUT      0        outputs: on */                      \
        /* DAC      0        DAC: on */                          \
        /* ADC      0        ADC: on */                          \
        /* MIC      0        microphone: on */                   \
        /* LINE     0        line input: on */                   \
                                                                 \
0x0043, /* Set-Up Reg 7     Digital audio interface format */   \
        /* XX       00       reserved */                         \
        /* MS       1        master/slave mode: master */        \
        /* LRSWAP   0        DAC left/right swap: disabled */    \
        /* LRP      0        DAC lrp: MSB on 1st BCLK */         \
        /* IWL      00       input bit length: 16 bit */         \
        /* FOR      11       data format: DSP format */          \
                                                                 \
0x0081, /* Set-Up Reg 8     Sample rate control */              \
        /* X        0        reserved */                         \
        /* CLKOUT   1        clock output divider: 2 (MCLK/2) */ \
```

FIGURE 1.21. (*Continued*)

```
        /* CLKIN    0        clock input divider: 2 (MCLK/2) */  \
        /* SR,BOSR 00000    sample rate: ADC 48kHz DAC 48kHz */ \
        /* USB/N    1        clock mode select : USB */          \
                                                                 \
0x0001  /* Set-Up Reg 9     Digital interface activation */     \
        /* XX..X    00000000 reserved */                         \
        /* ACT      1        active */                           \
};

DSK6713_AIC23_CodecHandle hAIC23_handle;

void c6713_dsk_init();
void comm_poll();
void comm_intr();
void output_sample(int);
void output_left_sample(short);
void output_right_sample(short);
Uint32 input_sample();
short input_left_sample();
short input_right_sample();
```

FIGURE 1.21. (*Continued*)

0x00000000 (the interrupt service table). Chapter 2 contains an example illustrating the use of the *pragma* directive to create a section named EXT_RAM to be mapped into external memory starting at address 0x80000000 (SDRAM). Chapter 2 also contains an example to illustrate the use of the non-volatile flash memory that starts at address 0x90000000 (FLASH). In Chapter 4, we illustrate the implementation of a digital filter in assembly language using external memory. Chapter 10 contains two projects that utilize the external memory interface (EMIF) 80-pin connector on the DSK, which starts at address 0xA0000000, to interface to external LEDs and LCDs.

1.7 ASSIGNMENTS

1. Modify program sine8_buf.c to generate a sine wave with a frequency of 3000 Hz. Verify your result using an oscilloscope connected to the LINE OUT socket on the DSK as well as using Code Composer to plot the 32 most recently output samples in both the time and frequency domains.
2. Write a polling-based program such that when DIP switch #3 is pressed down, LED #3 turns on and a 500-Hz cosine wave is generated for 5 seconds.
3. Write an interrupt-driven program that maintains a buffer containing the 128 most recent input samples read at a sampling frequency of 16 kHz from the AIC23 codec, using the MIC IN socket on the DSK. Halt the program and plot the buffer contents using Code Composer.

```
*Vectors_intr.asm Vector file for interrupt INT11
   .global _vectors           ;global symbols
   .global _c_int00
   .global _vector1
   .global _vector2
   .global _vector3
   .global _vector4
   .global _vector5
   .global _vector6
   .global _vector7
   .global _vector8
   .global _vector9
   .global _vector10
   .global _c_int11                ;for INT11
   .global _vector12
   .global _vector13
   .global _vector14
   .global _vector15

   .ref _c_int00                   ;entry address

VEC_ENTRY .macro addr              ;macro for ISR
    STW   B0,*--B15\
    MVKL  addr,B0
    MVKH  addr,B0
    B     B0
    LDW   *B15++,B0
    NOP   2
    NOP
    NOP
   .endm

_vec_dummy:
  B    B3
  NOP  5

   .sect ".vectors"                ;aligned IST section
   .align 1024
_vectors:
_vector0:   VEC_ENTRY _c_int00     ;RESET
_vector1:   VEC_ENTRY _vec_dummy   ;NMI
_vector2:   VEC_ENTRY _vec_dummy   ;RSVD
_vector3:   VEC_ENTRY _vec_dummy
_vector4:   VEC_ENTRY _vec_dummy
_vector5:   VEC_ENTRY _vec_dummy
_vector6:   VEC_ENTRY _vec_dummy
_vector7:   VEC_ENTRY _vec_dummy
_vector8:   VEC_ENTRY _vec_dummy
_vector9:   VEC_ENTRY _vec_dummy
_vector10:  VEC_ENTRY _vec_dummy
_vector11:  VEC_ENTRY _c_int11     ;ISR address
_vector12:  VEC_ENTRY _vec_dummy
_vector13:  VEC_ENTRY _vec_dummy
_vector14:  VEC_ENTRY _vec_dummy
_vector15:  VEC_ENTRY _vec_dummy
```

FIGURE 1.22. Listing of vector file `vectors_intr.asm`.

```
*Vectors_poll.asm  Vector file for polling
   .global _vectors
   .global _c_int00
   .global _vector1
   .global _vector2
   .global _vector3
   .global _vector4
   .global _vector5
   .global _vector6
   .global _vector7
   .global _vector8
   .global _vector9
   .global _vector10
   .global _vector11
   .global _vector12
   .global _vector13
   .global _vector14
   .global _vector15

   .ref _c_int00                   ;entry address

VEC_ENTRY .macro addr
    STW   B0,*--B15
    MVKL  addr,B0
    MVKH  addr,B0
    B     B0
    LDW   *B15++,B0
    NOP   2
    NOP
    NOP
   .endm

_vec_dummy:
  B    B3
  NOP  5

 .sect ".vectors"
 .align 1024

_vectors:
_vector0:   VEC_ENTRY _c_int00     ;RESET
_vector1:   VEC_ENTRY _vec_dummy   ;NMI
_vector2:   VEC_ENTRY _vec_dummy   ;RSVD
_vector3:   VEC_ENTRY _vec_dummy
_vector4:   VEC_ENTRY _vec_dummy
_vector5:   VEC_ENTRY _vec_dummy
_vector6:   VEC_ENTRY _vec_dummy
_vector7:   VEC_ENTRY _vec_dummy
_vector8:   VEC_ENTRY _vec_dummy
_vector9:   VEC_ENTRY _vec_dummy
_vector10:  VEC_ENTRY _vec_dummy
_vector11:  VEC_ENTRY _vec_dummy
_vector12:  VEC_ENTRY _vec_dummy
_vector13:  VEC_ENTRY _vec_dummy
_vector14:  VEC_ENTRY _vec_dummy
_vector15:  VEC_ENTRY _vec_dummy
```

FIGURE 1.23. Listing of vector file `vectors_poll.asm`.

```
/*C6713dsk.cmd  Linker command file*/

MEMORY
{
  IVECS:    org=0h,         len=0x220
  IRAM:     org=0x00000220, len=0x0002FDE0 /*internal memory*/
  SDRAM:    org=0x80000000, len=0x01000000 /*external memory*/
  FLASH:    org=0x90000000, len=0x00020000 /*flash memory*/
}

SECTIONS
{
  .EXT_RAM :> SDRAM
  .vectors :> IVECS /*in vector file*/
  .text    :> IRAM  /*Created by C Compiler*/
  .bss     :> IRAM
  .cinit   :> IRAM
  .stack   :> IRAM
  .sysmem  :> IRAM
  .const   :> IRAM
  .switch  :> IRAM
  .far     :> IRAM
  .cio     :> IRAM
  .csldata :> IRAM
}
```

FIGURE 1.24. Listing of linker command file `c6713dsk.cmd`.

4. Write a program that reads input samples from the left-hand channel of the AIC23 codec ADC at a sampling frequency of 16 kHz using function `input_left_sample()` and, just after it has been read, writes each sample value to the right-hand channel of the AIC23 codec DAC using function `output_right_sample()`. Verify the effective connection of the left-hand channel of the LINE IN socket to the right-hand channel of the LINE OUT socket using a signal generator and an oscilloscope. Gradually increase the frequency of the input signal until the amplitude of the output signal is reduced drastically. This frequency corresponds to the bandwidth of the DSP system (illustrated in more detail in Chapter 2).

REFERENCES

1. R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK*, Wiley, Hoboken, NJ, 2002.
2. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, Hoboken, NJ, 1999.

3. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, Hoboken NJ, 1992.
4. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, Hoboken NJ, 1990.
5. N. Kehtarnavaz and M. Keramat, *DSP System Design Using the TMS320C6000*, Prentice Hall, Upper Saddle River, NJ, 2001.
6. N. Kehtarnavaz and B. Simsek, *C6x-Based Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 2000.
7. N. Dahnoun, *DSP Implementation Using the TMS320C6x Processors*, Prentice Hall, Upper Saddle River, NJ, 2000.
8. Steven A. Tretter, *Communication System Design Using DSP Algorithms With Laboratory Experiments for the TMS320C6701 and TMS320C6711*, Kluwer Academic, New York, 2003.
9. J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*, Prentice Hall, Upper Saddle River, NJ, 1998.
10. C. Marven and G. Ewers, *A Simple Approach to Digital Signal Processing*, Wiley, Hoboken NJ, 1996.
11. J. Chen and H. V. Sorensen, *A Digital Signal Processing Laboratory Using the TMS320C30*, Prentice Hall, Upper Saddle River, NJ, 1997.
12. S. A. Tretter, *Communication System Design Using DSP Algorithms*, Plenum Press, New York, 1995.
13. A. Bateman and W. Yates, *Digital Signal Processing Design*, Computer Science Press, New York, 1991.
14. Y. Dote, *Servo Motor and Motion Control Using Digital Signal Processors*, Prentice Hall, Upper Saddle River, NJ, 1990.
15. J. Eyre, The newest breed trade off speed, energy consumption, and cost to vie for an ever bigger piece of the action, *IEEE Spectrum*, June 2001.
16. J. M. Rabaey, Ed., VLSI design and implementation fuels the signal-processing revolution, *IEEE Signal Processing*, Jan. 1998.
17. P. Lapsley, J. Bier, A. Shoham, and E. Lee, *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology, Berkeley, CA, 1996.
18. R. M. Piedra and A. Fritsh, Digital signal processing comes of age, *IEEE Spectrum*, May 1996.
19. R. Chassaing, The need for a laboratory component in DSP education: a personal glimpse, *Digital Signal Processing*, Jan. 1993.
20. R. Chassaing, W. Anakwa, and A. Richardson, Real-time digital signal processing in education, *Proceedings of the 1993 International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 1993.

21. S. H. Leibson, DSP development software, *EDN Magazine*, Nov. 8, 1990.

22. D. W. Horning, An undergraduate digital signal processing laboratory, *Proceedings of the 1987 ASEE Annual Conference*, June 1987.

23. *TMS320C6000 Programmer's Guide*, SPRU198G, Texas Instruments, Dallas, TX, 2002.

24. *TMS320C6211 Fixed-Point Digital Signal Processor–TMS320C6711 Floating-Point Digital Signal Processor*, SPRS073C, Texas Instruments, Dallas, TX, 2000.

25. *TMS320C6000 CPU and Instruction Set Reference Guide*, SPRU189F, Texas Instruments, Dallas, TX, 2000.

26. *TMS320C6000 Assembly Language Tools User's Guide*, SPRU186K, Texas Instruments, Dallas, TX, 2002.

27. *TMS320C6000 Peripherals Reference Guide*, SPRU190D, Texas Instruments, Dallas, TX, 2001.

28. *TMS320C6000 Optimizing C Compiler User's Guide*, SPRU187K, Texas Instruments, Dallas, TX, 2002.

29. *TMS320C6000 Technical Brief*, SPRU197D, Texas Instruments, Dallas, TX, 1999.

30. *TMS320C64x Technical Overview*, SPRU395, Texas Instruments, Dallas, TX, 2000.

31. *TMS320C6x Peripheral Support Library Programmer's Reference*, SPRU273B, Texas Instruments, Dallas, TX, 1998.

32. *Code Composer Studio User's Guide*, SPRU328B, Texas Instruments, Dallas, TX, 2000.

33. *Code Composer Studio Getting Started Guide*, SPRU509, Texas Instruments, Dallas, TX, 2001.

34. *TMS320C6000 Code Composer Studio Tutorial*, SPRU301C, Texas Instruments, Dallas, TX, 2000.

35. *TLC320AD535C/I Data Manual Dual Channel Voice/Data Codec*, SLAS202A, Texas Instruments, Dallas, TX, 1999.

36. *TMS320C6713 Floating-Point Digital Signal Processor*, SPRS186L, 2005.

37. *TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-Point Digital Signal Processors (Rev. J)*, SPRS226J, 2006.

38. TLV320AIC23 Stereo Audio Codec, 8- to 96-kHz, with Integrated Headphone Amplifier Data Manual, SLWS106G, 2003.

39. TMS320C6000 DSP Phase-Locked Loop (PLL) Controller Peripheral Reference Guide, SPRU233.

40. *Migrating from TMS320C6211/C6711 to TMS320C6713*, SPRA851.

41. *How to Begin Development Today with the TMS320C6713 Floating-Point DSP*, SPRA809.
42. TMS320C6000 DSP/BIOS User's Guide, SPRU423, 2002.
43. TMS320C6000 Optimizing C Compiler Tutorial, SPRU425A, 2002.
44. G. R. Gircys, *Understanding and Using COFF*, O'Reilly & Associates, Newton, MA, 1988.