1

DSP Development System

- Testing the software and hardware tools with Code Composer Studio
- Use of the TMS320C6713 DSK
- Programming examples to test the tools

Chapter 1 introduces several tools available for digital signal processing (DSP). These tools include the popular Code Composer Studio (CCS), which provides an integrated development environment (IDE), and the DSP starter kit (DSK) with the TMS320C6713 floating-point processor onboard and complete support for input and output. Three examples illustrate both the software and hardware tools included with the DSK. It is strongly suggested that you review these three examples before proceeding to subsequent chapters.

1.1 INTRODUCTION

Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing. The C6x notation is used to designate a member of Texas Instruments' (TI) TMS320C6000 family of digital signal processors. The architecture of the C6x digital signal processor is very well suited for numerically intensive calculations. Based on a very-long-instruction-word (VLIW) architecture, the C6x is considered to be TI's most powerful processor.

Digital signal processors are used for a wide range of applications, from communications and controls to speech and image processing. The general-purpose

Digital Signal Processing and Applications with the C6713 and C6416 DSK By Rulph Chassaing
ISBN 0-471-69007-4

digital signal processor is dominated by applications in communications (cellular). Applications embedded digital signal processors are dominated by consumer products. They are found in cellular phones, fax/modems, disk drives, radio, printers, hearing aids, MP3 players, high-definition television (HDTV), digital cameras, and so on. These processors have become the products of choice for a number of consumer applications, since they have become very cost-effective. They can handle different tasks, since they can be reprogrammed readily for a different application. DSP techniques have been very successful because of the development of low-cost software and hardware support. For example, modems and speech recognition can be less expensive using DSP techniques.

DSP processors are concerned primarily with real-time signal processing. Real-time processing requires the processing to keep pace with some external event, whereas non-real-time processing has no such timing constraint. The external event to keep pace with is usually the analog input. Whereas analog-based systems with discrete electronic components such as resistors can be more sensitive to temperature changes, DSP-based systems are less affected by environmental conditions. DSP processors enjoy the advantages of microprocessors. They are easy to use, flexible, and economical.

A number of books and articles address the importance of digital signal processors for a number of applications [1–22]. Various technologies have been used for real-time processing, from fiberoptics for very high frequency to DSPs very suitable for the audio-frequency range. Common applications using these processors have been for frequencies from 0 to 96 kHz. Speech can be sampled at 8 kHz (the rate at which samples are acquired), which implies that each value sampled is acquired at a rate of 1/(8 kHz) or 0.125 ms. A commonly used sample rate of a compact disk is 44.1 kHz. Analog/digital (A/D)-based boards in the megahertz sampling rate range are currently available.

The basic system consists of an analog-to-digital converter (ADC) to capture an input signal. The resulting digital representation of the captured signal is then processed by a digital signal processor such as the C6x and then output through a digital-to-analog converter (DAC). Also included within the basic system are a special input filter for anti-aliasing to eliminate erroneous signals and an output filter to smooth or reconstruct the processed output signal.

1.2 DSK SUPPORT TOOLS

Most of the work presented in this book involves the design of a program to implement a DSP application. To perform the experiments, the following tools are used:

1. *TI's 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 board, shown in Figure 1.1, that contains the TMS320C6713 (C6713) floating-point digital signal processor as well as a 32-bit stereo codec for input and output (I/O) support.

(c) A universal synchronous bus (USB) cable that connects the DSK board to a PC.

(d) A 5 V power supply for the DSK board.

2. *An IBM-compatible PC*. The DSK board connects to the USB port of the PC through the USB cable included with the DSK package.
3. *An oscilloscope, signal generator, and speakers*. A signal/spectrum analyzer is optional. Shareware utilities are available that utilize the PC and a sound card to create a virtual instrument such as an oscilloscope, a function generator, or a spectrum analyzer.

All the files/programs listed and discussed in this book (except some student project files in Chapter 10) are included on the accompanying CD. Most of the examples (with some minor modifications) can also run on the fixed-point C6416-based DSK. See Appendix H for the appropriate support files along with five illustrative examples. Reference 1 contains examples implemented on the C6711-based DSK (which has been discontinued). A list of all the examples is given on pages xv–xviii.

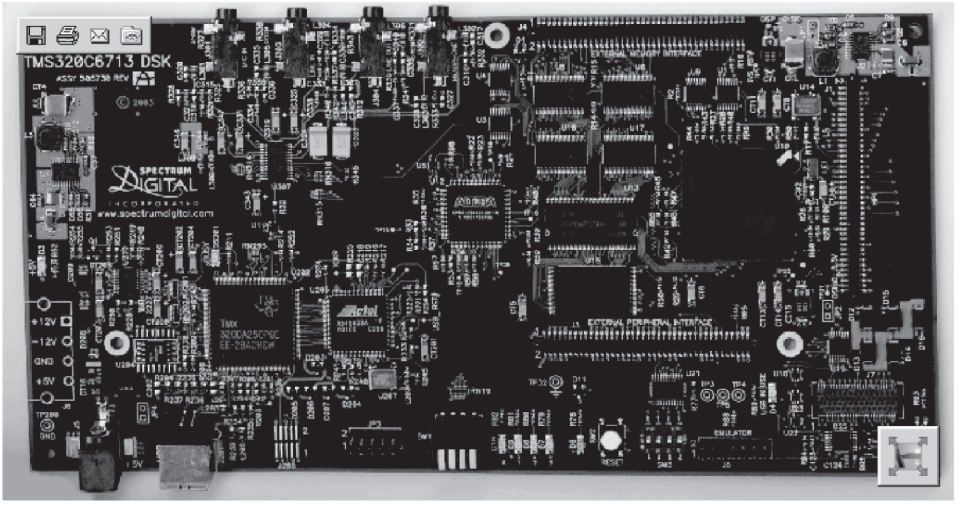
1.2.1 DSK Board

The DSK package is powerful, yet relatively inexpensive ($395), with the necessary hardware and software support tools for real-time signal processing [23–43]. It is a complete DSP system. The DSK board, with an approximate size of 5×8 in., includes the C6713 floating-point digital signal processor and a 32-bit stereo codec TLV320AIC23 (AIC23) for input and output.

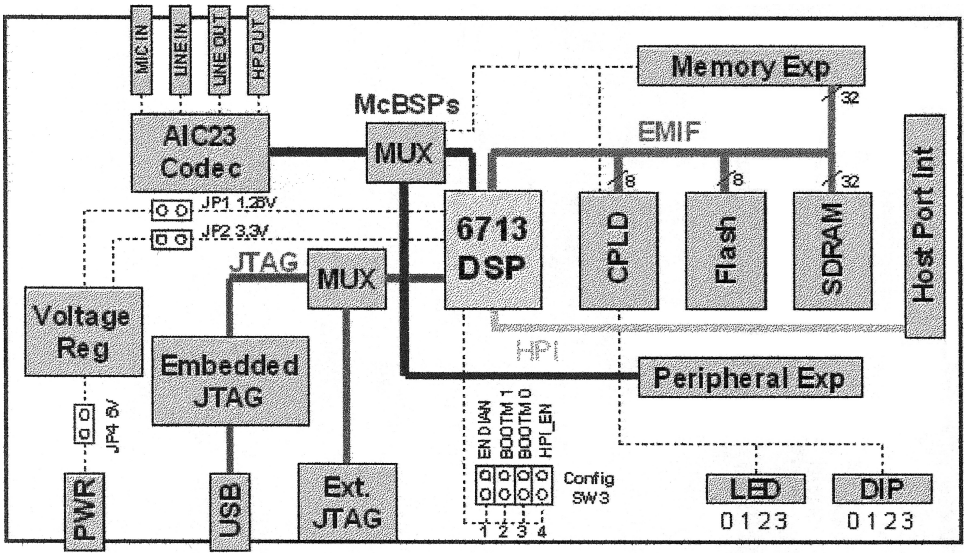
The onboard codec AIC23 [37] uses a sigma–delta technology that provides ADC and DAC. It connects to a 12-MHz system clock. Variable sampling rates from 8 to 96 kHz can be set readily.

A daughter card expansion is also provided on the DSK board. Two 80-pin connectors provide for external peripheral and external memory interfaces. Two project examples in Chapter 10 illustrate the use of the external memory interface (EMIF) with light-emitting diodes (LEDs) and liquid-crystal displays (LCDs) for spectrum display.

The DSK board includes 16 MB (megabytes) of synchronous dynamic random access memory (SDRAM) and 256 kB (kilobytes) of flash memory. Four connectors on the board provide 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 the four user dip switches on the DSK board can be read from a program and provides the user with a feedback control interface. The DSK operates at 225 MHz. Also onboard the DSK are voltage



(*a*)



(*b*)

FIGURE 1.1. TMS320C6713-based DSK board: (*a*) board; (*b*) diagram. (Courtesy of Texas Instruments)

regulators that provide 1.26 V for the C6713 core and 3.3 V for its memory and peripherals.

Appendix H illustrates a DSK based on the fixed-point processor C6416.

1.2.2 TMS320C6713 Digital Signal Processor

The TMS320C6713 (C6713) is based on the 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 arithmetic-logic units (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 (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.

1.3 CODE COMPOSER STUDIO

CCS provides an IDE to incorporate the software tools. CCS includes tools for code generation, such as a C compiler, an assembler, and a linker. It has graphical capabilities and supports real-time debugging. It provides an easy-to-use software tool to build and debug programs.

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 [25]. This executable file can be loaded and run directly on the C6713 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).

To create an application project, one can “add” the appropriate files to the project. Compiler/linker options can readily 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 monitor-

ing execution time. One can step through a program in different ways (step into, over, or out).

Real-time analysis can be performed using real-time data exchange (RTDX) (Chapter 9). RTDX allows for data exchange between the host PC and the target DSK, as well as analysis in real time without stopping the target. Key statistics and performance can be monitored in real time. Through the joint team action group (JTAG), communication with on-chip emulation support occurs to control and monitor program execution. The C6713 DSK board includes a JTAG interface through the USB port.

1.3.1 CCS Installation and Support

Use the USB cable to connect the DSK board to the USB port on the PC. Use the 5-V power supply included with the DSK package to connect to the +5-V power connector on the DSK to turn it on. Install CCS with the CD-ROM included with the DSK, preferably using the `c:\C6713` structure (in lieu of `c:\ti` as the default).

The CCS icon should be on the desktop as "C6713DSK CCS" and is used to launch CCS. The code generation tools (C compiler, assembler, linker) are used with CCS version 2.x.

CCS provides useful documentations included with the DSK package on the following (see the Help icon):

1. Code generation tools (compiler, assembler, linker, etc.)
2. Tutorials on CCS, compiler, RTDX
3. DSP instructions and registers
4. Tools on RTDX, DSP/basic input/output system (DSP/BIOS), and so on.

An extensive amount of support material (`pdf` files) is included with CCS. There are also examples included with CCS within the folder `c:\C6713\examples`. They illustrate the board and chip support library files, DSP/BIOS, and so on. CCS Version 2.x was used to build and test the examples included in this book. A number of files included in the following subfolders/directories within `c:\C6713` (suggested structure during CCS installation) can be very useful:

1. `myprojects`: a folder supplied only for your projects. All the folders in the accompanying book CD should be placed within this subdirectory.
2. `bin`: contains many utilities.
3. `docs`: contains documentation and manuals.
4. `c6000\cgtools`: contains code generation tools.
5. `c6000\RTDX`: contains support files for real-time data transfer.
6. `c6000\bios`: contains support files for DSP/BIOS.
7. `examples`: contains examples included with CCS.
8. `tutorial`: contains additional examples supplied with CCS.

Note that all the folders containing the programs and support files in the accompanying book CD should be transferred to the subdirectory `myprojects`. Change the properties of all the files included so that they are not read-only (all the folders can be highlighted to change the properties of their contents at once).

1.3.2 Useful Types of Files

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 processor
10. `file.cdb`: configuration file when using DSP/BIOS

1.4 QUICK TEST OF DSK

1. On power, a program *post.c* (Power On Self Test), stored in onboard flash memory, uses the board support library (BSL) to test the DSK. It tests the internal, external, and flash memories, the two multichannel buffered serial ports (McBSP), direct memory access (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 sec.
2. Launch CCS from the icon on the desktop. A USB enumeration process takes place. Then CCS will be opened and the LEDs will turn off. Press GEL → Check DSK → Quick Test. The Quick Test can be used for confirmation of correct operation and installation. The following message is then displayed:

   Switches: 15

   Board Revision: 1

   CPLD Revision: 2

This assumes that the four dip switches (0, 1, 2, 3) are all in the up position. Change the switches to $(1110)_2$ so that the first three switches (0, 1, 2) are up and press the

fourth switch (3) down. Repeat the procedure to select GEL → Check DSK → Quick Test and verify that the value of the switches is now 7 (with the display "Switches: 7"). You can set the value of the four user switches from 0 to 15. Within your program you can then direct the execution of your code based on these 16 values.

Alternative Quick Test of DSK

1. Open/launch CCS from the icon on the desktop if this has not been done already. Select File → Load Program. Click on the folder *sine8_LED\Debug* within myprojects to load the file *sine8_LED.out*. This loads the executable file *sine8_LED.out* into the C6713 processor. This assumes that you have already copied all the folders on the accompanying CD into your folder: *c:\c6713\myprojects*.
2. Select Debug → Run. Press the dip switch #0, which should light LED #0 on and generate a 1-kHz tone. Connect the LINE OUT (or the HEADPHONE) on the DSK board to a speaker or to an oscilloscope and verify the generation of the 1-kHz tone. The four connectors on the DSK board for I/O (MIC, LINE IN, LINE OUT, and HEADPHONE) use a 3.5-mm jack audio cable.

1.5 SUPPORT FILES

The following support files located in the folder `support` (except the library files) are used for most of the examples and projects discussed in this book:

1. `C6713dskinit.c`: contains functions to initialize the DSK, the codec, the serial ports, and for I/O. It is not included with CCS.
2. `C6713dskinit.h`: header file with function prototypes. Features such as those used to select the mic input in lieu of line input (by default), input gain, and so on are obtained from this header file (modified from a similar file included with CCS).
3. `C6713dsk.cmd`: sample linker command file. This generic file can be changed when using external memory in lieu of internal memory.
4. `Vectors_intr.asm`: a modified version of a vector file included with CCS to handle interrupts. Twelve interrupts, INT4 through INT15, are available, and INT11 is selected within this vector file. They are used for interrupt-driven programs.
5. `Vectors_poll.asm`: vector file for programs using polling.
6. *rts6700.lib*, *dsk6713bsl.lib*, *csl6713.lib*: run-time, board, and chip support library files, respectively. These files are included with CCS and are located in *C6000\cgtools\lib*, *C6000\dsk6713\lib*, and *c6000\bios\lib*, respectively.

1.6 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 primary focus is to become familiar with both the software and hardware tools. It is strongly suggested that you complete these three examples before proceeding to subsequent chapters.

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

This example generates a sinusoid using a table lookup method. More important, it illustrates some features of CCS for editing, building a project, accessing the code generation tools, and running a program on the C6713 processor. The C source program *`sine8_LED.c`* shown in Figure 1.2 implements the sine generation and is included in the folder *`sine8_LED`*.

Program Consideration

Although the purpose is to illustrate some of the tools, it is useful to understand the program *`sine8_LED.c`*. A table or buffer `sine_table` is created and filled with eight points representing $\sin(t)$, where $t = 0, 45, 90, 135, 180, 225, 270,$ and 315 degrees

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

#include "dsk6713_aic23.h"                //support file for codec,DSK
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;      //set sampling rate
short loop = 0;                           //table index
short gain = 10;                          //gain factor
short sine_table[8]={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)               //=0 if switch #0 pressed
  {
    DSK6713_LED_on(0);                    //turn LED #0 ON
    output_sample(sine_table[loop]*gain); //output every Ts (SW0 on)
    if (++loop > 7)  loop = 0;            //check for end of table
  }
  else DSK6713_LED_off(0);                //LED #0 off
 }                                        //end of while (1)
}                                         //end of main
```

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

(scaled by 1000). Within the function *main*, another function, *comm_poll*, is called that is located in the communication and initialization support file *c6713dskinit.c*. It initializes the DSK, the AIC23 codec onboard the DSK, and the two McBSPs on the C6713 processor. Within *c6713dskinit.c*, the function *DSK6713_init* initializes the BSL file, which must be called before the two subsequent BSL functions, *DSK6713_LED_init* and *DSK6713_DIP_init*, are invoked that initialize the four LEDs and the four dip switches.

The statement `while (1)` within the function *main* creates an infinite loop. When dip switch #0 is pressed, LED #0 turns on and the sinusoid is generated. Otherwise, *DSK6713_DIP_get(0)* will be false (true if the switch is pressed) and LED #0 will be off.

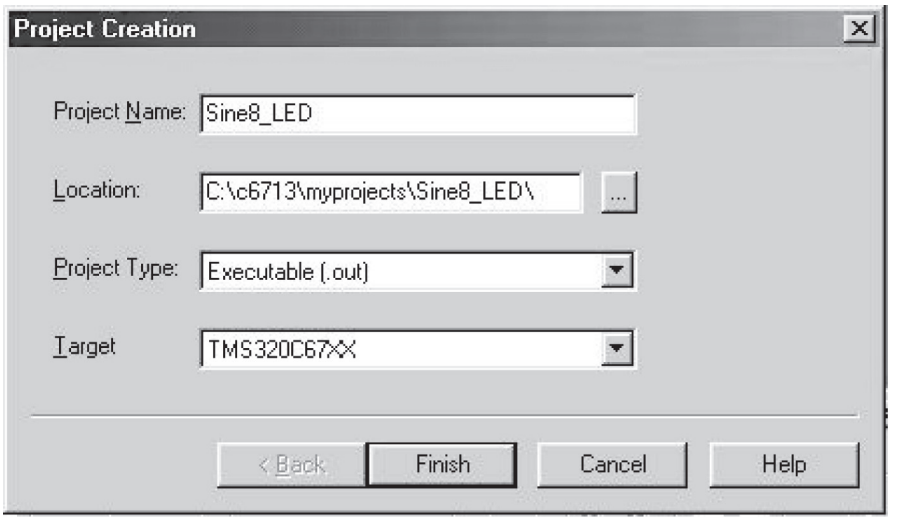
The function `output_sample`, located in the communication support file *C6713dskinit.c*, is called to output the first data value in the buffer or table *sine_table[0]* = 0. The loop index is incremented until the end of the table is reached, after which it is reinitialized to zero.

Every sample period $T = 1/F_s = 1/8000 = 0.125$ ms, the value of dip switch #0 is tested, and a subsequent data value in `sine_table` (scaled by *gain* = 10) is sent for output. Within one period, eight data values (0.125 ms apart) are output to generate a sinusoidal signal. The period of the output signal is $T = 8(0.125\,\text{ms}) = 1$ ms, corresponding to a frequency of $f = 1/T = 1$ kHz.

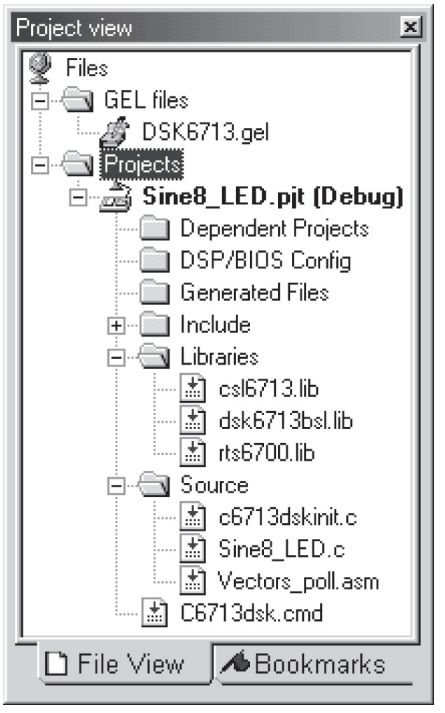
Create Project

In this section we illustrate how to create a project, adding the necessary files for building the project **`sine8_LED`**. Back up the folder *sine8_LED* (change its name) or delete its content (which can be retrieved from the book CD if needed), keeping only the C source file *sine8_LED.c* and the file *gain.gel* in order to recreate the content of that folder. Access CCS (from the desktop).

1. To create the project file *sine8_LED.pjt*. Select Project → New. Type *sine8_LED* for the project name, as shown in Figure 1.3. This project file is saved in the folder *sine8_LED* (within `c:\c6713\myprojects`). The `.pjt` file stores project information on build options, source filenames, and dependencies.
2. To add files to the project. Select Project → Add Files to Project. Look in the folder *support*, Files of type C Source Files. Double-click on the C source file *C6713dskinit.c* to add it to the project. Click on the "+" symbol to the left of the Project Files window within CCS to expand and verify that this C source file has been added to the project.
3. Repeat step 2, use the pull-down menu for Files of type, and select ASM Source Files. Double-click on the assembly source vector file *vectors_poll.asm* to add it to the project. Repeat again and select Files of type: Linker Command File, and add *C6713dsk.cmd* to the project.



(*a*)



(*b*)

FIGURE 1.3. CCS Project windows for `sine8_LED`: (*a*) project creation; (*b*) project view files.

4. To add the library support files to the project. Repeat the previous step, but select files of type: Object and Library Files. Look in `c:\c6713\c6000\cgtools\lib` and select the run-time support library file *`rts6700.lib`* (which supports the C67x architecture) to add to the project. Continue this process to add the BSL file *`dsk6713bsl.lib`* located in c*:\c6713\c6000\dsk6713\lib*, and the chip support library (CSL) file *`csl6713.lib`* located in c*:\c6713\c6000\bios\lib*.
5. Verify from the Files 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. The GEL file *`dsk6713.gel`* is added automatically when you create the project. It initializes the C6713 DSK invoking the BSL to use the phase-locked loop (PLL) to set the central processing unit (CPU) clock to 225 MHz (otherwise, the C6713 runs at 50 MHz by default).
6. Note that there are no "include" files yet. Select Project → Scan All File Dependencies. This adds/includes the header files *`c6713dskinit.h`*, along with several board and chip support header files included with CCS.

The Files window in CCS should look as in Figure 1.3*b*. Any of the files (except the library files) from CCS's Files window can be displayed by clicking on it. 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.)

It is also possible to add files to a project simply by "dragging" the file (from a different window) and dropping it into the CCS Project window.

Code Generation and Options

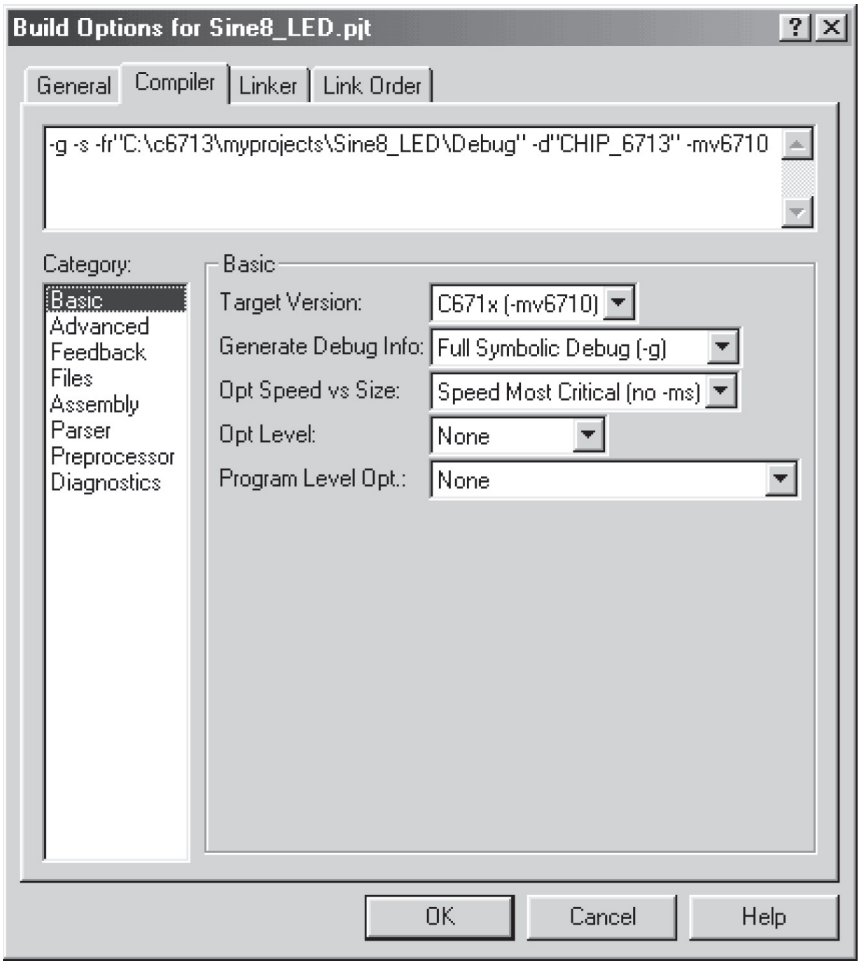
Various options are associated with the code generation tools: C compiler and linker to build a project.

Compiler Option

Select Project → Build Options. Figure 1.4*a* shows the CCS window Build Options for the compiler. Select the following for the compiler option with Basic (for Category): (1) c671x{-mv6710} (for Target Version), (2) Full Symbolic Debug (for Generate Debug Info), (3) Speed most critical (for Opt Speed vs. Size), and (4) None (for Opt Level and Program Level Opt). Select the Preprocessor Category and type for Define Symbols{d}: CHIP_6713, and from the Feedback Category, select for Interlisting: OPT/C and ASM{-s}. The resulting compiler option is

```
-g -s
```

The `-g` option is used to enable symbolic debugging information, useful during the debugging process, and is used in conjunction with the option `-s` to interlist the C



(*a*)

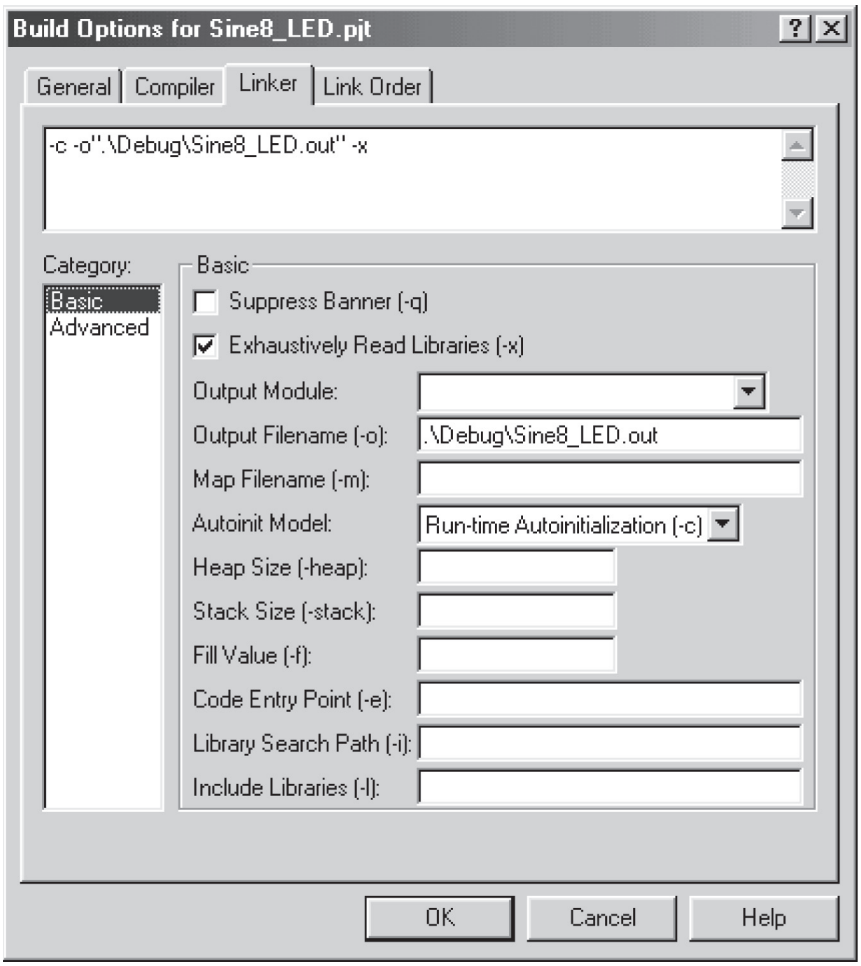
FIGURE 1.4. CCS Build options: (*a*) compiler; (*b*) linker.

source file with the assembly source file *sine8_LED.asm* generated (an additional option, `-k`, can be used to retain the assembly source file). The `-g` option disables many code optimizations to facilitate the debugging process. Press OK.

Selecting C621x or C64xx for Target Version invokes a fixed-point implementation. The C6713-based DSK can use either fixed- or floating-point processing. Most examples implemented in this book can run using fixed-point processing. Selecting C671x as Target Version invokes a floating-point implementation.

If No Debug is selected (for Generate Debug Info) and `-o3:File` is selected (for Opt Level), the Compiler option is automatically changed to

```
-s -o3
```



(*b*)

FIGURE 1.4. (*Continued*)

The `-o3` option invokes the highest level of optimization for performance or execution speed. For now, speed is not critical (neither is debugging). Use the compiler options `-gs` (which you can also type directly in the compiler command window). Initially, one would not optimize for speed but to facilitate debugging. A number of compiler options are described in Ref. 28.

Linker Option

Click on Linker (from CCS Build Options). The output filename `sine8_LED.out` defaults to the name of the `.pjt` filename, and Run-time Autoinitialization defaults for Autoinit Model. The linker option should be displayed as in Figure 1.4*b*. The map file can provide useful information for debugging (memory locations of func-

tions, etc.). The `-c` option is used to initialize variables at run time, and the `-o` option is used to name the linked executable output file `sine8_LED.out`. Press OK.

Note that you can/should choose to store the executable file in the subfolder "Debug," within the folder `sine8_LED`, especially during the debugging stage of a project.

Again, these various compiler and linker options can be typed directly within the appropriate command windows.

In lieu of adding the three library files to the project by retrieving them from their specific locations, it is more convenient to add them within the linker option window Include Libraries{-l}, typing them directly, separated by a comma. However, they will not be shown in the Files window.

Building and Running the Project

The project `sine8_LED` can now be built and run.

1. Build this project as **`sine8_LED`**. Select Project → Rebuild All or press the toolbar with the three down 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. Figure 1.5 shows several windows within CCS for the project `sine8_LED`. The building process causes all the dependent files to be included (in case one forgets to scan for all the file dependencies).
2. Select File → Load Program in order to load `sine_LED.out` by clicking on it (CCS includes an option to load the program automatically after a build). It should be in the folder `sine8_LED\Debug`. Select Debug → Run or use the toolbar with the "running man." Connect a speaker to the LINE OUT connector on the DSK. Press the dip switch #0. You should hear a tone. You can also use the headphone output at the same time.

The sampling rate F_s of the codec is set at 8 kHz. The frequency generated is $f = F_s$/(number of points) = 8 kHz/8 = 1 kHz. Connect the output of the DSK to an oscilloscope to verify a 1-kHz sinusoidal signal with an approximate amplitude of 0.8 V p-p (peak to peak).

Correcting Program Errors

1. Delete the semicolon in the statement

```
short gain = 10;
```

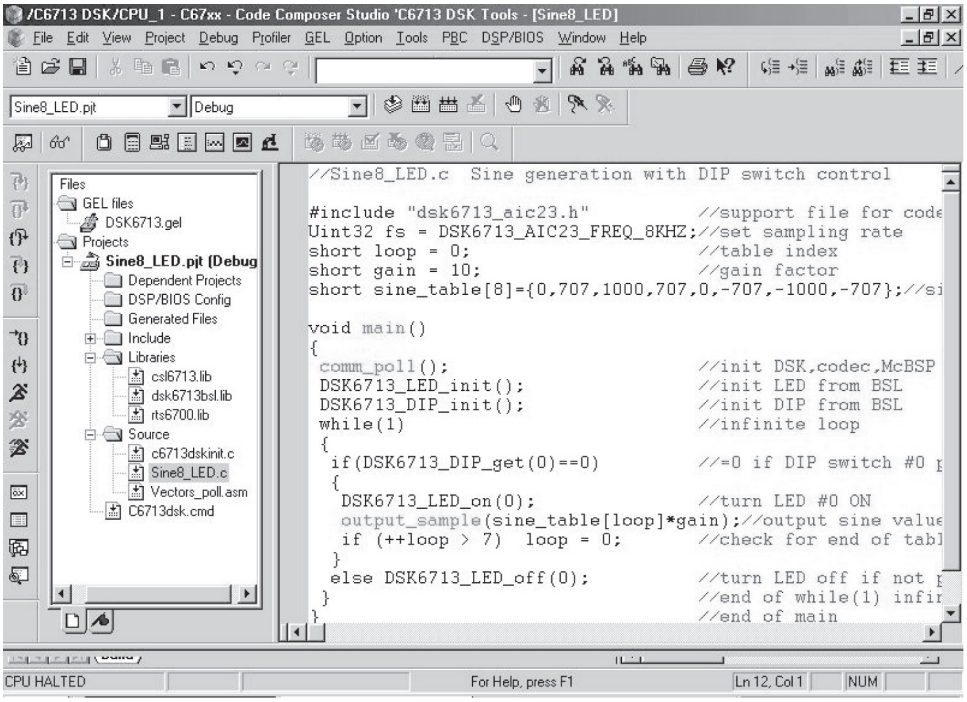


FIGURE 1.5. CCS windows for project `sine8_LED`.

in the C source file `sine8_LED.c`. If it is not displayed, double-click on it (from the Files window).

2. Select Project → Build to perform an incremental build or use the toolbar with the two (not three) arrows. The incremental build is chosen so that only the C source file `sine8_LED.c` is compiled. With the Rebuild option (toolbar with three arrows), files compiled and/or assembled previously would again go through this unnecessary process.
3. An error message, highlighted in red, stating that a ";" is expected, should appear in the Build window of CCS (lower left). You may need to scroll up the Build window for a better display of this error message. Double-click on the highlighted error message line. This should bring the cursor to the section of code where the error occurs. Make the appropriate correction, Build again, load, and run the program to verify your previous results.

Monitoring the Watch Window

Verify that the processor is still running (and dip switch #0 is pressed) . Note the indicator "DSP RUNNING" 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 window, which should be displayed on the lower section of CCS. 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 volume of the generated tone has increased (with the processor still running and dip switch #0 is pressed). The amplitude of the sine wave has increased from approximately 0.8 V p-p to approximately 2.5 V p-p.
3. Change *`gain`* to 33 (as in step 2). Verify that a higher-pitched tone exists, which implies that the frequency of the sine wave has changed just by changing its amplitude. This is not so. You have exceeded the range of the codec AIC23. Since the values in the table are scaled by 33, the range of these values is now between ±33,000. The range of output values is limited from -2^{15} to $(2^{15} - 1)$, or from –32,768 to +32,767.

   Since the AIC23 is a stereo codec, we can send data to both 16-bit channels within each sampling period. This is introduced in Chapter 2. This can be useful to experiment with the stereo effects of output signals. In Chapter 7, we use both channels for adaptive filtering where it is necessary to input one type of signal (such as noise) on one 16-bit channel and another signal (such as a desired signal) on the other 16-bit channel. In this book, we will mostly use the codec as a mono device without the need to use an adapter that is required when using both channels.

Applying the Slider Gel File

The General Extension Language (GEL) is an interpretive language similar to (a subset of) C. It allows you to change a variable such as gain, sliding through different values while the processor is running. All variables must first be defined in your source program.

1. Select File → Load GEL and open the file `gain.gel`, which you retained from the original folder, `sine8_LED` (that you backed up). Double-click on the file *`gain.gel`* to view it within CCS. It should be displayed in the Files window. This file is shown in Figure 1.6. By creating the slider function *`gain`* shown in Figure 1.6, you can start with an initial value of 10 (first value) for the variable `gain` that is set in the C program, up to a value of 35 (second value), incremented by 5 (third value).
2. Select GEL → Sine Gain → Gain. This should bring out the Slider window shown in Figure 1.7, with the minimum value of 10 set for the gain.
3. Press the up-arrow key to increase the gain value from 10 to 15, as displayed in the Slider window. Verify that the volume of the sine wave generated has increased. Press the up-arrow key again to continue increasing the slider, incrementing by 5 up to 30. The amplitude of the sine wave should be about 2.5 V p-p with a *`gain`* value set at 30. Now use the mouse to click directly on the Slider window and slowly increase the slider position to 31, then 32, and

```
/*gain.gel Create slider and vary amplitude (gain) of sinewave*/

menuitem "Sine Gain"

slider Gain(10,35,5,1,gain_parameter) /*incr by 5,up to 35*/
{
        gain = gain_parameter;        /*vary gain of sine*/
}
```

FIGURE 1.6. GEL file to slide through different gain values in the sine generation program (`gain.gel`).

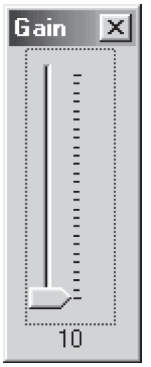


FIGURE 1.7. Slider window for varying the gain of generated sine wave.

verify that the frequency generated is still 1 kHz. Increase the slider to 33 and verify that you are no longer generating a 1-kHz sine wave. The table values, scaled by the gain value, are now between ±33,000 (beyond the acceptable range by the codec).

Changing the Frequency of the Generated Sinusoid

1. Change the sampling frequency from 8 to 16 kHz by setting f_s in the C source program to *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 points in the lookup table to four points in lieu of eight points—for example, {0, 1000, 0, –1000}. The size of the array `sine_table` and the loop index also need to be changed. Verify that the generated frequency is $f = F_s/$(number of points).

   Note that the sinusoid is no longer generated if the dip switch #0 is not pressed. If a different dip switch such as switch #3 is desired (in lieu of switch #0), the BSL functions *DSK6713_DIP_get(3)*, *DSK6713_LED_on(3)*, and *DSK6713_LED_off(3)* can be substituted in the C source program.

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 loop index 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 the Sinusoid and Plotting with CCS (`sine8_buf`)

This example generates a sinusoid with eight points, as in Example 1.1. More important, it illustrates CCS capabilities for plotting in both time and frequency domains. The program *`sine8_buf.c`*, shown in Figure 1.8, implements this project. This program creates a buffer to store the output data in memory.

Create this project as *`sine8_buf.pjt`*, and add the necessary files to the project, as in Example 1.1 (use the C source program *`sine8_buf.c`* in lieu of *`sine8_LED.c`*). Note that the necessary header support files are added to the project by selecting Project → Scan All File Dependencies. The necessary

```
//sine8_buf Sine generation. Output buffer plotted within CCS

#include "dsk6713_aic23.h"                //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;        //set sampling rate
int loop = 0;                             //table index
short gain = 10;                          //gain factor
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707};//sine values
short out_buffer[256];                    //output buffer
const short BUFFERLENGTH = 256;           //size of output buffer
int i = 0;                                //for buffer count

interrupt void c_int11()                  //interrupt service routine
{
 output_sample(sine_table[loop]*gain);    //output sine values
 out_buffer[i] = sine_table[loop]*gain;   //output to buffer
 i++;                                     //increment buffer count
 if(i==BUFFERLENGTH) i=0;                 //if @ bottom reinit count
 if (++loop > 7) loop = 0;                //check for end of table
 return;                                  //return from interrupt
}

void main()
{
  comm_intr();                            //init DSK, codec, McBSP
  while(1);                               //infinite loop
}
```

FIGURE 1.8. Sine generation with output stored in memory as well (`sine8_buf.c`).

support files for this project, `c6713dskinit.c`, `vectors_intr.asm` and `C6713dsk.cmd`, are in the folder `support`, and the three library support files can be added using Project → Build Options and selecting the linker option (Include Libraries). Type them, separating each by a comma. Note that since this program is interrupt-driven (in lieu of polling), the vector file `vectors_intr.asm` (in lieu of `vectors_poll.asm`) is added to the project.

Within the function *main*, *comm._intr* (in lieu of *comm_poll* in Example 1.1) is called. This function resides in `c6713dskinit.c` to support interrupt-driven programs. The statement `while(1)` within the function `main` creates an infinite loop to wait for an interrupt to occur. On interrupt, execution proceeds to the interrupt service routine (ISR) *c_int11*. This ISR address is specified in the file `vectors_intr.asm` with a branch instruction to this address, using interrupt INT11. Interrupts are discussed in more detail in Chapter 3.

Within the ISR, the function `output_sample`, located in the communication and initialization file `c6713dskinit.c`, is called to output the first data value in `sine_table`. The loop index is incremented until the end of the table is reached; after that, it is reinitialized to zero. An output buffer is created to capture a total of 256 (specified by *BUFFERLENGTH*) sine data values. Execution returns from ISR to the while (1) infinite loop to wait for each subsequent interrupt.

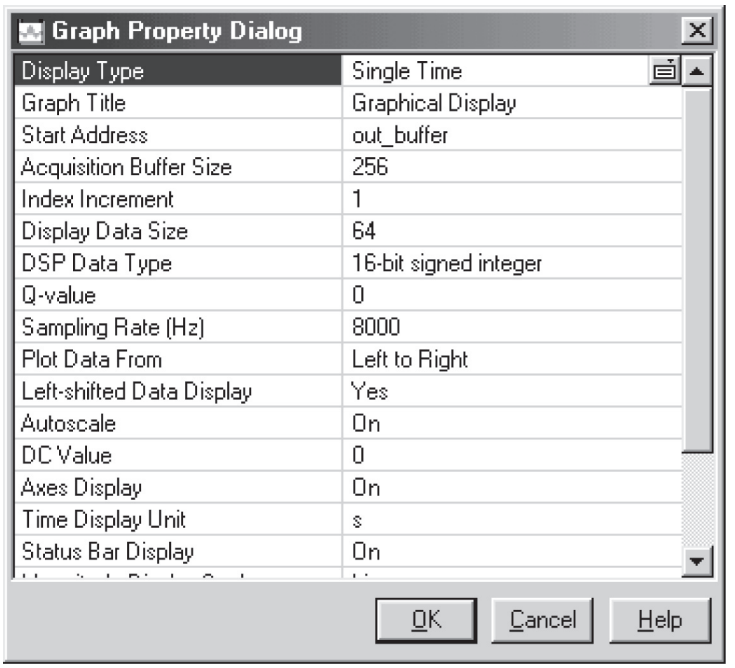
Build this project as **`sine8_buf`**. Load and run the executable file *`sine8_buf.out`* and verify that a 1-kHz sinusoid is generated with the output connected to a speaker or a scope (as in Example 1.1).

Plotting with CCS

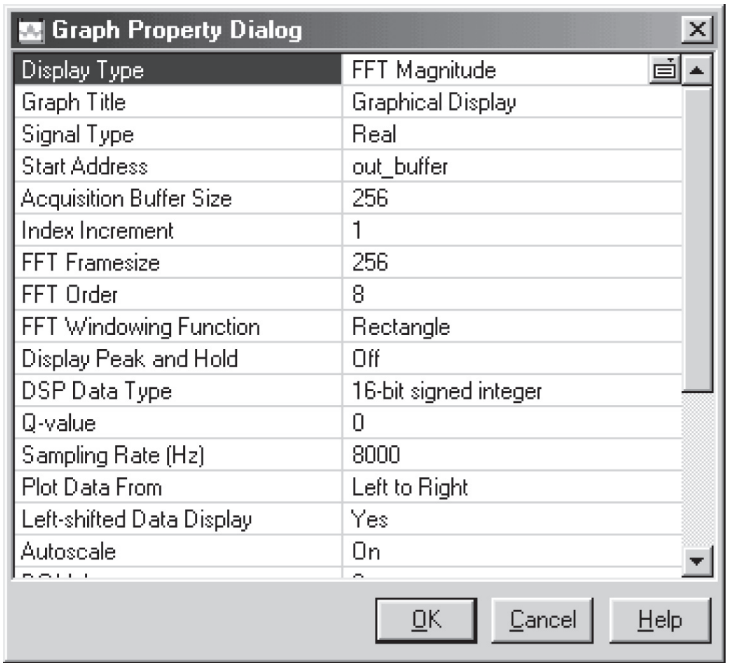
The output buffer is being updated continuously every 256 points (you can readily change the buffer size). Use CCS to plot the current output data stored in the buffer *`out_buffer`*.

1. Select View → Graph → Time/Frequency. Change the Graph Property Dialog so that the options in Figure 1.9*a* are selected for a time-domain plot (use the pull-down menu when appropriate). The starting address of the output buffer is `out_buffer`. The other options can be left as default. Figure 1.10 shows a time-domain plot of the sinusoidal signal within CCS.
2. Figure 1.9*b* shows CCS's Graph Property Display for a frequency-domain plot. Choose a fast Fourier transform (FFT) order so that the frame size is 2^{order}. Press OK and verify that the FFT magnitude plot is as shown in Figure 1.10. The spike at 1000 Hz represents the frequency of the sinusoid generated.

You can obtain many different windows within CCS. From the Build window, right-click and select Float In Main Window. To change the screen size, right-click on the Build window and deselect Allow Docking. For example, you can get the time-domain plot (separated). Right-click on the time-domain plot, select Float In Main Window, and again right-click on the same time-domain plot window and deselect Allow Docking. You can then move it.



(*a*)



(*b*)

FIGURE 1.9. CCS Graph Property Dialog for `sine8_buf`: (*a*) for time-domain plot; (*b*) for frequency-domain plot.

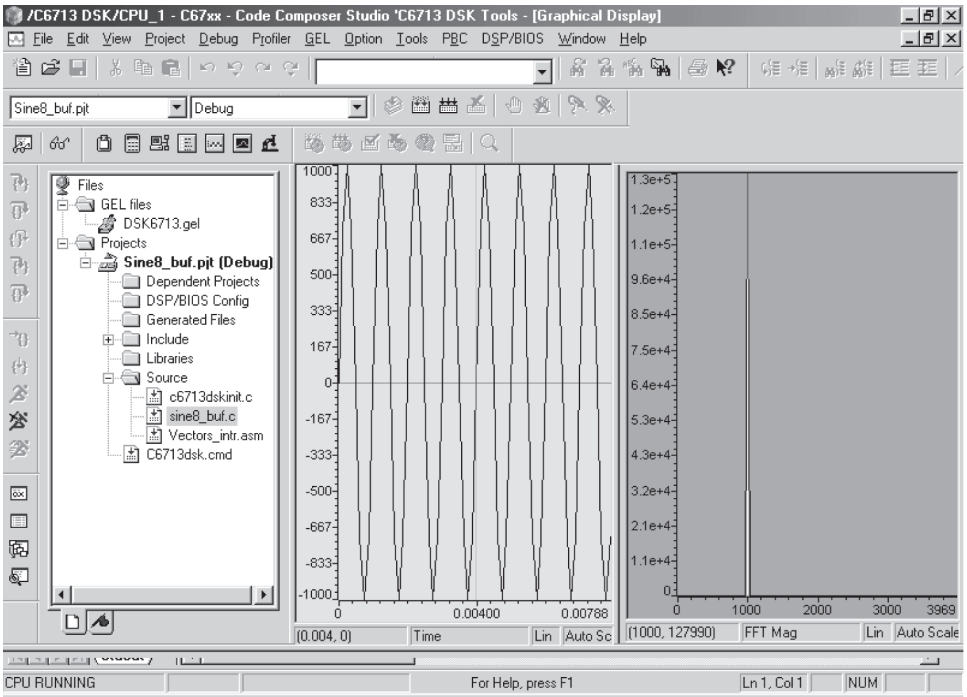


FIGURE 1.10. CCS windows for sine8_buf showing both time- and frequency-domain plots of a generated 1-kHz sine wave.

Viewing and Saving Data from Memory in a File

To view the content of that buffer, select View → Memory and specify `out_buffer` for the address, and select the 16-bit signed integer (or hex, etc.) for the format.

To save the content of the output buffer in a file, select File → Data → Save. Save the file as sine8_buf.dat (as type hex, for example) in the folder sine8_buf. From the Storing Memory window, use `out_buffer` as the buffer's address with length 256. You can then plot this data [with MATLAB for example] and verify the 1-kHz sinusoidal waveform (with 8 kHz as the sampling rate).

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

Operations such as addition/subtraction and multiplication are the key operations in a DSP. A very important operation is multiply/accumulate, which is useful in a number of applications requiring digital filtering, correlation, and spectrum analysis. Since the multiplication operation is executed commonly and is essential for most DSP algorithms, it is important that it executes in a single cycle. With the C6713 we can actually perform two multiply/accumulate operations within a single cycle.

This example illustrates additional features of CCS, such as single-stepping, setting breakpoints, and profiling for the benchmark. Again, the purpose here is to

```
//Dotp4.c Multiplies two arrays, each array with 4 numbers

int dotp(short *a,short *b,int ncount);//function prototype
#include <stdio.h>                     //for printf
#include "dotp4.h"                     //header file with data
#define count 4                        //# data in each array
short x[count] = {x_array};            //declaration of 1st array
short y[count] = {y_array};            //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 sum = 0;                         //init sum
  int i;

  for (i = 0; i < ncount; i++)
     sum += a[i] * b[i];               //sum of products
  return(sum);                         //return sum as result
}
```

FIGURE 1.11. Sum-of-products program using C code (`dotp4.c`).

```
//dotp4.h  Header file with two arrays of numbers

#define x_array 1,2,3,4

#define y_array 0,2,4,6
```

FIGURE 1.12. Header file with two arrays each with four numbers (`dotp4.h`).

become more familiar with the tools. We invoke C compiler optimization to see how performance or execution speed can be drastically increased.

The C source file `dotp4.c` in Figure 1.11 takes the sum of products of two arrays, each with four numbers, contained in the header file `dotp4.h` in Figure 1.12. The first array contains the four numbers 1, 2, 3, and 4, and the second array contains the four numbers 0, 2, 4, and 6. The sum of products is $(1 \times 0) + (2 \times 2) + (3 \times 4) + (4 \times 6) = 40$.

The program can be readily modified to handle a larger set of data. No real-time implementation is used in this example, and no real-time I/O support files are needed. The support functions for interrupts are not needed here.

Create this project as **`dotp4`** and add the following files to the project (see Example 1.1):

1. `dotp4.c`: C source file
2. `vectors_poll.asm`: vector file defining the entry address *`c_int00`*
3. `C6713dsk.cmd`: generic linker command file
4. `rts6700.lib`: library file

Do not add any "include" files using "Add Files to Project" since they are added by selecting Project → Scan All File Dependencies. The header file `stdio.h` is needed due to the `printf` statement in the program `dotp4.c` to print the result.

Implementing a Variable Watch

1. Select Project → Options with `-gs` as the compiler option and the default linker option with no optimization.
2. Rebuild All by selecting the toolbar with the three arrows (or select Project → Rebuild All). Load the executable file *dotp4.out* within the folder *dotp4\Debug*.
3. Select View → Quick Watch. Type *`sum`* to watch the variable *`sum`* and click on "Add to Watch." The message "identifier not found" associated with *`sum`* is displayed (as Value) because this local variable does not exist yet.
4. Set a breakpoint at the line of code

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

   by placing the mouse cursor (clicking) on that line, then right-click and select the Toggle breakpoint. Or, preferably, with the cursor on that line of code (at the extreme left), double-click. A red circle to the left of that line of code should appear. (Note: placing the cursor on a line of code with a set breakpoint and double clicking will remove the breakpoint.)
5. Select Debug → Run (or use the "running man" toolbar). The program executes up to (excluding) the line of code with the set breakpoint. A yellow arrow will also point to that line of code.
6. Single-step using F8. Repeat or continue to single-step and observe/watch the variable *`sum`* in the Watch window change in value to 0, 4, 16, 40. Select Debug → Run and verify that the resulting value of `sum` is printed as

   ```
   sum = 40 (decimal)
   ```

7. Note the `printf` statement in the C program `dotp4.c` for printing the result. This statement (while excellent for debugging) should be avoided after the debugging stage, since it takes over 6000 cycles to execute.

Animating

1. Select File → Reload Program to reload the executable file `dotp4.out`. Or, preferably, select Debug → Restart. Note that after the executable file is

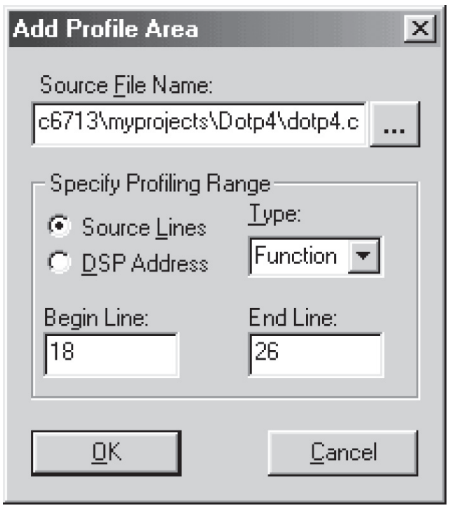
loaded, the entry address for execution is `c_int00`, as can be verified by the disassembled file.

2. The same breakpoint should be set already at the same line of code as before. Select Debug → Animate or use the equivalent toolbar in the left window (below the Halt running man). Observe the variable *sum* change in values through the Watch window. The speed of animation can be controlled by selecting Option → Customize → Animate Speed (the maximum speed is set to default at 0 second).

Benchmarking (Profiling) without Optimization

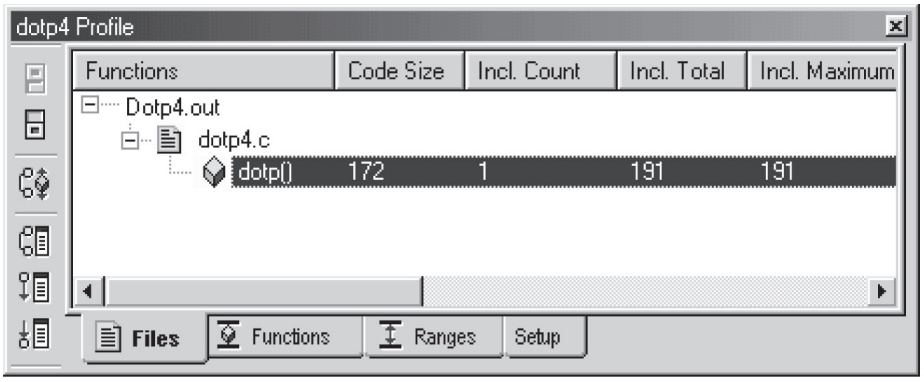
In this section we illustrate how to benchmark a section of code: in this case, the *dotp* function. Verify that the options for the compiler (`-g`) and linker (`-c -o dotp4.out`) are still set. To profile code, you must use the compiler option `-g` for symbolic debugging information. Remove any breakpoint by double-clicking on the line of code with the set breakpoint (or right-click and select the Toggle breakpoint).

1. Select Debug → Restart.
2. Select Profiler → Start New Session and enter `dotp4` as the Profile Session Name. Then press OK.
3. Click on the icon to "Create Profile Area" (see Figure 1.13*a*). This icon is the third icon from the bottom left in Figure 1.13*b*. Figure 1.13*b* shows the added profile area for the function dotp within the C source file `dotp4.c`.

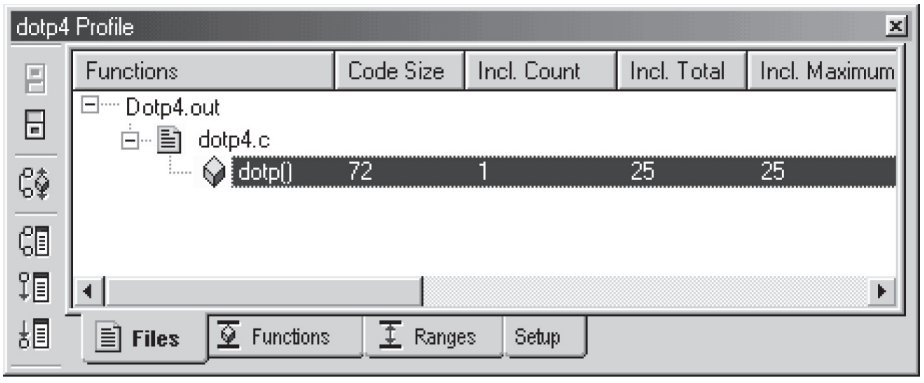


(*a*)

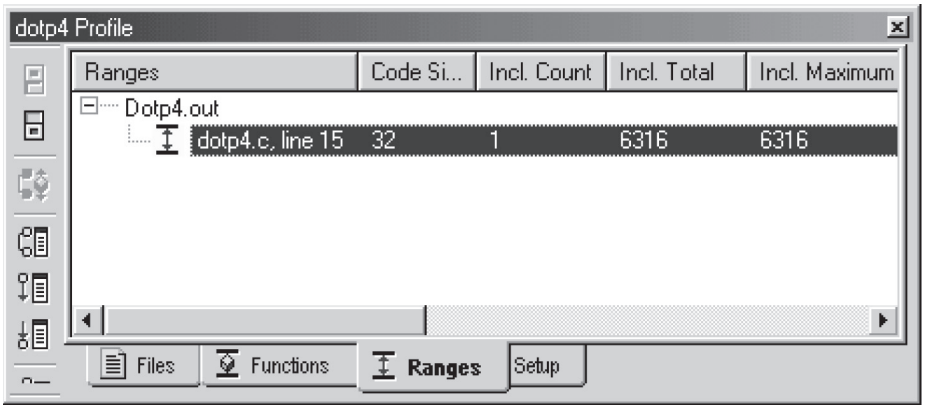
FIGURE 1.13. CCS display of project `dotp4` for profiling: (*a*) profile area for function *dotp*; (*b*) profiling function dotp with no optimization; (*c*) profiling function dotp with level 3 optimization; (d) profiling printf.



(*b*)



(*c*)



(*d*)

FIGURE 1.13. (*Continued*)

4. Run the program. Verify the results shown in Figure 1.13*b*. This indicates that it takes 191 cycles to execute the function dotp (with no optimization).

Benchmarking (Profiling) with Optimization

In this section we illustrate how to optimize the program using one of the optimization options, `-o3`. The program's execution speed can be increased using the optimizing C compiler. Change the compiler option (select Project → Build Options) to

```
-g -o3
```

and use the same linker options as before (you can type this option directly). The option `-o3` invokes the highest level of compiler optimization. Various compiler options are described in Ref. 28. Rebuild All (toolbar with three arrows) and load the executable file `dotp4.out` (or select File → Reload Program). Re-create the Profile Area as in Figure 1.13*a*.

Select Debug → Run. Verify that it takes now 25 cycles (from 191) to execute the `dotp` function, as shown in Figure 1.13*c*. This is a considerable improvement using the C compiler optimizer. The code size is reduced from 172 to 72. The dot product example can be also optimized using an intrinsic function or the code optimization techniques discussed in Chapter 8.

Profiling Printf

Again restart the program (Debug → Restart). Click on the icon *Ranges* at the bottom of the profile area. Highlight *printf* from the C source program, drag it to the profiling area window, and drop it by releasing the cursor. Verify that the code size of *printf* is 32 and that it takes 6316 cycles to execute, as shown in Figure 1.13*d*.

Note that in lieu of using Figure 1.13*a* to profile the function dotp, you can highlight it, drag it, and drop it with your mouse in the profiling area.

1.7 SUPPORT PROGRAMS/FILES CONSIDERATIONS

The following support files are used for practically all the examples in this book: (1) *`c6713dskinit.c`*, (2) *`vectors_intr.asm`* or *`vectors_poll.asm`*, and (3) *`c6713dsk.cmd`*. For now, the emphasis associated with these files should be on using them.

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

Several BSL and CSL support functions are included in the initialization and communication (init/comm) file *`c6713dskinit.c`*. A partial listing is shown in Figure 1.14. It includes functions to initialize the DSK and provide for input and output.

```
//C6713dskinit.c Partial list of init/comm file.Includes CSL/BSL funct
...
void c6713_dsk_init()               //dsp-peripheral init
{
DSK6713_init();                     //BSL to init DSK-EMIF,PLL
hAIC23_handle=DSK6713_AIC23_openCodec(0, &config);//handle to codec
DSK6713_AIC23_setFreq(hAIC23_handle, fs);  //set sample rate
MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData);//32bits interface
MCBSP_start(DSK6713_AIC23_DATAHANDLE,MCBSP_XMIT_START | MCBSP_RCV_START
 | MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC,220); //start data channel
}

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

void comm_intr()                    //for comm/init using interrupt
{
 poll = 0;                          //0 since not polling
 IRQ_globalDisable();               //disable interrupts
 c6713_dsk_init();                  //init DSP and codec
CODECEventId=MCBSP_getXmtEventId(DSK6713_AIC23_codecdatahandle);//Xmit
 ...
 IRQ_setVecs(vectors);              //point to the IRQ vector
 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 interrup out a sample
}

void output_sample(int out_data)    //out to Left and Right channels
{
 short CHANNEL_data;
 AIC_data.uint=0;                   //clear data structure
 AIC_data.uint=out_data;            //32-bit data -->data structure
 ...
 if(poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));//ready to Xmit?
   MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);//write data
}

void output_left_sample(short out_data)  //for output->left channel
{
 AIC_data.uint=0;                  //clear data structure
 AIC_data.channel[LEFT]=out_data;  //data->Left channel->data structure
 if(poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));//ready to Xmit?
 MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);//out->leftchannel
}

void output_right_sample(short out_data)  //for output->right channel
...
Uint32 input_sample()                     //for 32-bit input
{
 short CHANNEL_data;
```

FIGURE 1.14. Partial listing of communication/initialization support program (C6713dskinit.c).

```
 if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));//receiveready?
   AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE); //read data
 ...
 return(AIC_data.uint);
}

short input_left_sample()                    //input to left channel
{
 if(poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));//receiveready?
   AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);//read->left chan
 return(AIC_data.channel[LEFT]);             //return left channel data
}

short input_right_sample()                   //input to right channel
...
```

FIGURE 1.14. (*Continued*)

The function *comm_intr()* in an interrupt-driven program or *comm_poll()* in a polling-based program calls the appropriate functions to initialize the DSK. These two functions are located in the init/comm. file. When using an interrupt-driven program, interrupt #11 (INT11) is configured and enabled (selected). The nonmaskable interrupt bit must be enabled as well as the global interrupt enable (GIE) bit. A different interrupt, such as INT12, can be selected readily by modifying slightly the init/comm. file and the vector file that contains the branching address to the corresponding ISR in the main C source program. INT11 is generated via the serial port (McBSP).

The function *input_sample()* is used to input data and the function *output_sample()* to output data. Most of the examples throughout the book utilize the AIC23 codec in a mono format, defaulting to the left channel to read or write a 16-bit data. The example `loop_stereo.c` in Chapter 2 illustrates the stereo capability of the codec to input 16-bit data into each (left and right) channel and output a 16-bit data from each channel. Some adaptive filtering examples in Chapter 7 use both input channels to acquire two different 16-bit input data signals.

The code `input = input_sample();`, casting `input` as a short, acquires 16-bit data through the left (default) channel. Similarly, *output_sample ((short) . . .)*; outputs 16-bit data from the left (default) channel.

A polling-based program (non-interrupt-driven) continuously polls or tests whether or not data are ready to be received or transmitted. This scheme is in general less efficient than the interrupt scheme. For input, the content of the serial port control register (SPCR) bit 1 [the second least significant bit (LSB)], as shown in Figure B.8 (Appendix B), is continuously tested to determine when data are available to be received or read. For output, the content of SPCR bit 17 is tested (Figure B.8) to determine when data are available to be transmitted. An input data value is accessed through the data receive register of the McBSP. An output data value is sent through the data transmit register of McBSP.

The MCBSP1 transmit interrupt is used and INT11 is selected in the examples throughout the book. If the program is polling-based, the McBSP is continuously tested before reading (for input) or writing (for output).

Within the function `output_sample()` used for output, in the code segment

```
If (poll) while(!MCBSP_xrdy(...)); MCBSP_write(...);
```

the first line of code continuously tests (if polling-based) the transmit ready *xrdy* register bit. If it is a 1, then the subsequent line of code is executed to write (output). If the transmit ready bit is a 0 (not ready), then the *while()* statement becomes *while (true)* and execution remains in an infinite loop until the transmit ready bit becomes a 1 (ready). If the program is not polling-based, then the transmit ready bit is not tested and writing (output) occurs every sample period.

Similarly, within the function `input_sample()` used for input, in the code segment

```
If (poll) while(!MCBSP_rrdy(...)); MCBSP_read(...);
```

the first line of code continuously tests (if polling-based) the receive ready *rrdy* register bit. If it is a 1 (ready), the subsequent line of code reads the data. If it is a 0 (not ready), the *while ( )* statement causes execution to remain in an infinite loop until the receive ready bit register becomes a 1. If the program is not polling-based, the receive ready bit is not tested and reading occurs every sample period.

The examples throughout the book use both interrupt-driven and polling-based programs. A polling-based program can be readily changed to interrupt-driven and vice versa. Interrupts are discussed further in Chapter 3.

Header File (`c6713dskinit.h`)

The corresponding header support file *c6713dskinit.h* contains the function prototypes as well as various register settings associated with the AIC23 codec. For example (see *c6713dskinit.h*):

1. The mic input can be set in lieu of the line input by changing the value of register 4 from the (default) value of 0x0011 to 0x0015.
2. In Chapter 2, a loop program yields an output that is the delayed input, with the same frequency but attenuated (by default). To increase the gain of the (default) left line input channel, change the value of register 0 from 0x0017 to 0x001c. This value will produce an output of the same amplitude as the input. Note that either the line input or the mic input can be made active.

1.7.2 Vector File (`vectors_intr.asm/vectors_poll.asm`)

To select interrupt INT11, a branch instruction to the ISR `c_int11` located in the C program (see `sine8_buf.c`) is placed at the address INT11 in `vectors_intr.asm`. A listing of the file `vectors_intr.asm` is shown in Figure 1.15. Note

```
*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 ".vecs"                     ;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 1.15. Vector file for an interrupt-driven program (`vectors_intr.asm`).

the underscore preceding the name of the routine or function being called. The ISR is also referenced in `vectors_intr.asm` using *`.ref _c_int11`*.

For a non-interrupt-driven or polling-based program, a separate file `vectors_poll.asm` is used, in lieu of `vectors_intr.asm`, by

1. Deleting the reference to the interrupt service routine (ISR) *`.ref _c_int11`*
2. Replacing the branch instruction to the ISR for interrupt INT11 by (NOP), which is a no operation instruction.

1.7.3 Linker Command File (`c6713dsk.cmd`)

The linker command file *`C6713dsk.cmd`* is listed in Figure 1.16. It shows that sections such as *`.text`* reside in internal RAM (IRAM), which is mapped to the internal memory of the C6713 digital signal processor. It can be used as a generic sample linker command file even though some portion of it is not necessary. Chapter 2 contains an example illustrating the use of the *pragma* directive to specify a section such as EXT_RAM in synchronous DRAM (SDRAM). SDRAM is a section in external memory that starts at the address `0x80000000`. Chapter 2 contains an example illustrating the use of the onboard flash memory (burning the flash) that starts at address `0x90000000`. In Chapter 4, we illustrate the implementation of a digital filter is assembly code using external memory SDRAM. Chapter 10 contains

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

MEMORY
{
  IVECS:    org=0h,          len=0x220
  IRAM:     org=0x00000220, len=0x0002FDE0 /*internal memory*/
  SDRAM:    org=0x80000000, len=0x00100000 /*external memory*/
  FLASH:    org=0x90000000, len=0x00020000 /*flash memory*/
}
SECTIONS
{
  .EXT_RAM :> SDRAM
  .vectors :> IVECS     /*in vector file*/
  .text    :> IRAM
  .bss     :> IRAM
  .cinit   :> IRAM
  .stack   :> IRAM
  .sysmem  :> IRAM
  .const   :> IRAM
  .switch  :> IRAM
  .far     :> IRAM
  .cio     :> IRAM
  .csldata :> IRAM
}
```

FIGURE 1.16. Generic linker command file (`C6713dsk.cmd`).

two projects that utilize the EMIF 80-pin connector on the DSK, which starts at address `0xA0000000`, to interface to external LEDs and LCDs.

Linker options include *`-heap size`* to specify the heap size in bytes for dynamic memory allocation (default is 1 kB) and the option *`-stack size`* to specify the C system stack size in bytes. Other linker options can be found in Ref. 26.

The linker allocates the program in memory using a default location algorithm. It places the various sections into appropriate memory locations, where code and data reside. By using a linker command file with extension `.cmd`, one can customize the allocation process, specifying MEMORY and SECTIONS directives within the linker command file. The linker directive MEMORY (uppercase) defines a memory model and designates the origin and length of various available memory spaces. The directive SECTIONS (uppercase) allocate the output sections into defined memory and designate the various code sections to available memory spaces.

Most of the examples in the book invoke internal memory. The generic sample linker command file, shown in Figure 1.16, can be used for almost all of the examples in the book, even if neither external nor flash memory is utilized.

1.8 COMPILER/ASSEMBLER/LINKER SHELL

In previous examples the code generation tools for compiling, assembling, and linking were invoked within CCS while building a project. The tools may also be invoked directly outside CCS using a DOS shell.

1.8.1 Compiler

The compiler shell can be invoked using

```
cl6x [options] [files]
```

to compile and assemble files that can be C files with extension `.c`, assembly files with extension `.asm`, and linear assembly (introduced in Chapter 3) with extension `.sa`. A linear assembly program file is a cross between C and assembly that can provide a compromise between the more versatile C program and the most efficient assembly program. For example, the command

```
Cl6x -gks -o3 file1.c, file2, file3.asm, file4.sa
```

invokes the C compiler to compile *`file1`* and *`file2`* (defaults to extension `.c`) and generates the assembly files *`file1.asm`* and *`file2.asm`*. This also invokes the assembler optimizer to optimize *`file4.sa`* and create *`file4.asm`*. Then the assembler (invoked with the shell command `cl6x`) assembles the four assembly source files and creates the four object files *`file1.obj`*, . . . , *`file4.obj`*. The option `-gs` adds debugger-specific information for debugging purposes and interlists C

statements into assembly files, respectively. The `-k` option is used to keep the assembly source files generated.

Four levels of compiler optimizations are available, with `-o3` to invoke the highest level of optimization. Level 0 allocates variables to registers. Level 1 performs all level 0 optimizations, eliminates local common expressions, and removes unused assignments. Level 2 performs all the level 1 optimizations plus loop optimizations and rolling. Level 3 performs all level 2 optimizations and removes functions that are not called. There are also compiler optimizations to minimize code size (with possible degradation in execution speed).

Note that full optimization may change memory locations that can affect the functionality of a program. In such cases, these memory locations must be declared as volatile. The compiler does not optimize volatile variables. A volatile variable is allocated to an uninitialized section in lieu of a register. Volatiles can be used when memory access is to be exactly as specified in the C code.

Initially, the functionality of a program is of primary importance. One should *not* invoke any (or too-high-level) optimization option initially while debugging, since additional debugger-specific information is provided to enhance the debugging process. Such additional information suppresses the level of performance. It is also difficult to debug a program after optimization, since the lines of code are usually no longer arranged in a serial fashion. Compiler options can also be set using the environment variable with `C_OPTION`.

1.8.2 Assembler

An assembly-coded source file *`file3.asm`* can also be assembled using

```
asm6x file3.asm
```

to create `file3.obj`. The `.asm` extension is optional. The resulting object file is then linked with a run-time support library to create an executable COFF file with extension `.out` that can be loaded directly and run on the DSp. Examples using assembly-coded source files are introduced in Chapter 3.

1.8.3 Linker

The linker can be invoked using

```
lnk6x -c prog1.obj -o prog1.out -l rts6700.lib
```

The `-c` option tells the linker to use special conventions defined by the C environment for automatic variable initialization at run time (another linker option, `-cr`, initializes the variables at load time). The `-l` option invokes a library file such as the run-time support library file `rts6700.lib`. These options [`-c` (or `-cr`) and

-l] must be used when linking. The object file prog1.obj is linked with the library file(s) and creates the executable file prog1.out. Without the -o option, the executable file *a.out* (by default) is created.

The linker can also be invoked with the compiler shell command with the -z option

```
Cl6x -gks -o3 prog1.c prog2.asm -z -o prog.out -m prog.map
-l rts6700.lib
```

to create the executable file *prog.out*. The -m option creates a map file that provides a list of all the addresses of sections, symbols, and labels that can be useful for debugging.

The linker also links automatically a *boot* program when using C programs to initialize the run-time environment, setting the entry point to *c_int00*. The symbol *_c_int00* is defined automatically when the linker option -c (or -cr) is invoked. The function *_c_int00*, included in the run-time support library, is the entry point in the boot program that sets up the stack and calls *main*. The run-time library support program *boot.c* is used to auto-initialize variables. The linker option -c invokes the initialization process with boot.c. Note that it is defined in the vector files *vectors_intr.asm* and *vectors_poll.asm*.

The book CD contains all the main source files used in this book, located in separate folders, and some support files necessary for many examples and projects are located in the folder *support*. Other needed support files are included with CCS within *c:\C6713*.

1.9 ASSIGNMENTS

1. Write a program to generate a cosine with a frequency of 666.66 Hz. Verify your output result using LINE OUT, as well as plotting the generated cosine in both time and frequency domains.
2. Write a polling-based program so that once dip switch #3 is pressed, LED #3 turns on and a 666.66 Hz cosine is generated for approximately 5 seconds. [*Hint*: also use (incorporate) the delay associated with turning a LED on.]
3. Write a program to multiply two arrays, each containing the five numbers 1, 2, 3, 4, and 5 (i.e., $1^2 + 2^2 + 3^2 + 4^2 + 5^2$). Verify your result using a watch window and printing it within CCS in the Build window.
4. Write an interrupt-driven program to capture an input sinusoidal signal of amplitude 3 V p-p and a frequency of 1 kHz, and output that sampled signal every 0.0625 ms. Use the function input_sample in a similar fashion as the function output_sample used in Examples 1.1 and 1.2—for example,

   input = input_sample();

casting input as short (16-bit). Verify that the output signal has the same frequency as the input signal but is reduced in amplitude. Increase the input signal frequency until the output is reduced drastically. What is the approximate frequency at which this occurs? This represents the bandwidth of the onboard AIC23 codec (as illustrated in Chapter 2).

REFERENCES

Note: References 23 to 43 are included with the DSK package.

1. R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK*, Wiley, New York, 2002.
2. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
3. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
4. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 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, New York, 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*, SPRS186, Texas Instruments, Dallas, TX.

37. *TLV320AIC23 Stereo Audio Codec, 8- to 96-kHz, with Integrated Headphone Amplifier Data Manual*, SLWS106G, Texas Instruments, Dallas, TX, 2003.

38. *TMS320C6000 DSP Phase-Locked Loop (PLL) Controller Peripheral Reference Guide*, SPRU233, Texas Instruments, Dallas, TX.

39. *Migrating from TMS320C6211/C6711 to TMS320C6713*, SPRA851, Texas Instruments, Dallas, TX, 2003.

40. *How to begin Development Today with the TMS320C6713 Floating-Point DSP*, SPRA809, Texas Instruments, Dallas, TX, 2003.

41. *TMS320C6000 DSP/BIOS User's Guide*, SPRU423, Texas Instruments, Dallas, TX, 2002.

42. *TMS320C6000 Optimizing C Compiler Tutorial*, SPRU425A, Texas Instruments, Dallas, TX, 2002.

43. *TMS320C6000 Chip Support Library API User's Guide*, SPRU401F, Texas Instruments, Dallas, TX, 2003.

44. B. W. Kernigan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Upper Saddle River, NJ, 1988.

45. G. R. Gircys, *Understanding and Using COFF*, O'Reilly & Associates, Newton, MA, 1988.