



Jordan University of Science & Technology
Department of Electrical Engineering

Digital Signal Processing Laboratory

Using

TMS320C6713 DSP Starter Kit

(EE462)

PREFACE

This Booklet contains a selected and edited laboratory experiments that were taken with some minor modifications from the book “Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK”, second edition, John Wiley & Sons, 2008 written by Rulph Chassaing and Donald Reay. The book has a wide variety of digital signal processing and filter design experiments that can be run on the TMS320C6713 and TMS320C6416 digital signal processing kit (DSK). Our selection of the experiments was made to achieve some objectives: the ability to analyze signals and systems in the time and frequency domain, implement real-time signal processing system and design of FIR and IIR digital filters. In addition, we have included one experiment based on Matlab in order to have a simulation tool which can be used for comparison purposes and to start the design of some FIR and IIR design experiments. Another experiment was included to get the students familiar with the DSK kit.

Dr. Jehad Ababneh
Eng. Yara Obeidat

Labs

Lab 1:

Experiment 1: Waveform Generation and Digital Filter Design and Analysis in Time and Frequency Domains Using Matlab.

Lab 2:

Experiment 2A: Introduction to the Digital Signal Processing Kit (DSK) and the Code Composer Studio (CCS).

Experiment 2B: Sine Wave Generation Using Eight points with DIP Switch Control.

Experiment 2C: Generation of sinusoid and Plotting with CCS (*sine8_buf*).

Lab 3:

Experiment 3A: Basic Input and Output Using Polling (loop_poll).

Experiment 3B: Basic Input and Output Using Interrupts (loop_intr).

Experiment 3C: Sine Wave Generation Using sin() Function, Reconstruction, Aliasing, and the Properties of the AIC 23 Codec.

Lab 4:

Experiment 4: FIR Filter I

Lab 5:

Experiment 5: FIR Filter II.

Lab 6:

Experiment 6: IIR Filter I.

Lab 7:

Experiment 7: IIR Filter II.

Lab 8:

Experiment 8: Discrete Time and Fast Fourier Transform

Experiment 1

Waveform Generation and Digital Filter Design and Analysis in Time and Frequency Domains Using Matlab

Objectives:

- 1- Generation and analysis of different basic signals in time and frequency domain.
- 2- Design and analysis of typical FIR and IIR digital filters.
- 3- Filters application and implementation.

Lab Equipments:

- 1- PC with Matlab software installed.
- 2- Headphone.

Description:

In this experiment, we will use signal processing toolbox commands and analysis tools in Matlab to visualize signals in time and frequency domains, compute FFTs for spectral analysis of signals and filters, design FIR and IIR filters. Most toolbox functions require you to begin with a vector representing a time base. Consider generating data with a 1000 Hz sample frequency, for example. An appropriate time vector is $t = (0:0.001:1)'$; where the MATLAB colon operator creates a 1001-element row vector that represents time running from 0 to 1 s in steps of 1 ms. The transpose operator (') changes the row vector into a column; the semicolon (;) tells MATLAB to compute, but not display the result. Given t , you can create a sample signal y consisting of two sinusoids, one at 50 Hz and one at 120 Hz with twice the amplitude. $y = \sin(2\pi \cdot 50 \cdot t) + 2 \cdot \sin(2\pi \cdot 120 \cdot t)$; You may also generate discrete-time signals by first generating a sample axis using the command $n = (0:1:1024)$; Then, to generate a sinusoidal signal sampled at twice the Nyquist rate (or a signal that has a frequency that is one forth the sampling frequency), use the command: $X = \cos(n \cdot \pi / 2)$; You may plot the signal in the time domain using the command: `plot(n,X)`. Since MATLAB is a programming language, an endless variety of different signals is possible. Here are some statements that generate several commonly used sequences, including the unit impulse, unit step, and unit ramp functions: $t = (0:0.001:1)'$;

```
y = [1; zeros(99,1)]; % impulse
y = ones(100,1);      % step (filter assumes 0 initial cond.)
y = t;                % ramp
```

Some applications, however, may need to import data from outside MATLAB. To load data from an ASCII file or MAT-file, use the MATLAB *load* command. You may also use this command to load wave files.

The single sided amplitude spectrum of a signal can be evaluated using the FFT function which computes the Fast Fourier Transform. A simple Matlab function named *single_sided_amplitude_spectrum* was written for this purpose. This function is stored in the directory C:\DSPLaboratory. To calculate and plot single sided amplitude spectrum of the signal Y sampled at FS frequency, type the command:

`HY= Single_Sided_Amplitude_Spectrum(Y,FS);`

We will also learn how to graphically design and implement digital filters using Signal Processing Toolbox. Filter design is the process of creating the filter coefficients to meet specific frequency specifications. Although many methods exist for designing the filter coefficients, this experiment focuses on using the basic features of the Filter Design and Analysis Tool (FDATool) GUI. This experiment includes a brief discussion of applying the completed filter design and filter implementation using MATLAB command line functions, such as *filter*. The analysis and design process in this experiment will be used in later experiments for design and analysis of real time filters implemented on the **TMS320C6713 DSP starter kit**.

LAB WORK:

1- Waveform Generation and Analysis

1. **Launch Matlab** by double - clicking on its desktop icon
2. Generate 1024 samples of 1kHz sinusoidal (*cos*) signal sampled at 8kHz with the command: `n=(0:1023);X=cos(2*n*pi*1000/8000);`
3. Plot 100 samples of the generated signal in the time domain using both the *plot* and *stem* Matlab functions using the commands: `plot(n(1:100),X(1:100))`, `stem(n(1:100),X(1:100))`. Use appropriate title and axis labeling.
4. Evaluate and plot the amplitude spectrum of the generated signal using *fft* Matlab function with the command: `HX= Single_Sided_Amplitude_Spectrum(X,8000);`
5. Use the Matlab function *load* to load the word “Aspect” uttered by male speaker with the command: `[Y,FS,NBITS]=wavread('aspect11');`
6. Plot three 250 samples of three different segments (frames) of the loaded signal in the time domain using the *plot* Matlab function with the commands:

```
plot(Y(1000:1250))
plot(Y(3200:3450))
plot(Y(5000:5250))
```

Use appropriate title and axis labeling

7. Evaluate and plot the amplitude spectrum of these different segments using the commands:

```
HY= Single_Sided_Amplitude_Spectrum(Y(1000:1250),FS);
HY= Single_Sided_Amplitude_Spectrum(Y(3200:3450),FS);
HY= Single_Sided_Amplitude_Spectrum(Y(5000:5250),FS);
```

Use appropriate title and axis labeling.

8. Compare and discuss the results obtained in steps 3 through 7 in your lab report.
9. Generate and analyze 100 samples of unit impulse and unit step function in the time and frequency domain using the same procedure

2- Filters Design with FDATool GUI

In this part, first we will exactly follow and use the Matlab example to design and analyze an octave-band filter in the Getting Started With Signal Processing Toolbox to get familiar with this powerful tool. Then, we will use similar procedure to design a notch filter and export its coefficients to be used in the last part of this experiment to suppress a single tone from a corrupted spoken word.

An octave is the interval between two frequencies having a ratio of 2:1. An octave-band filter is a bandpass filter with high cutoff frequency approximately twice that of the low cutoff frequency. The class of an octave filter is determined by its allowable passband ripple and its stopband attenuation.

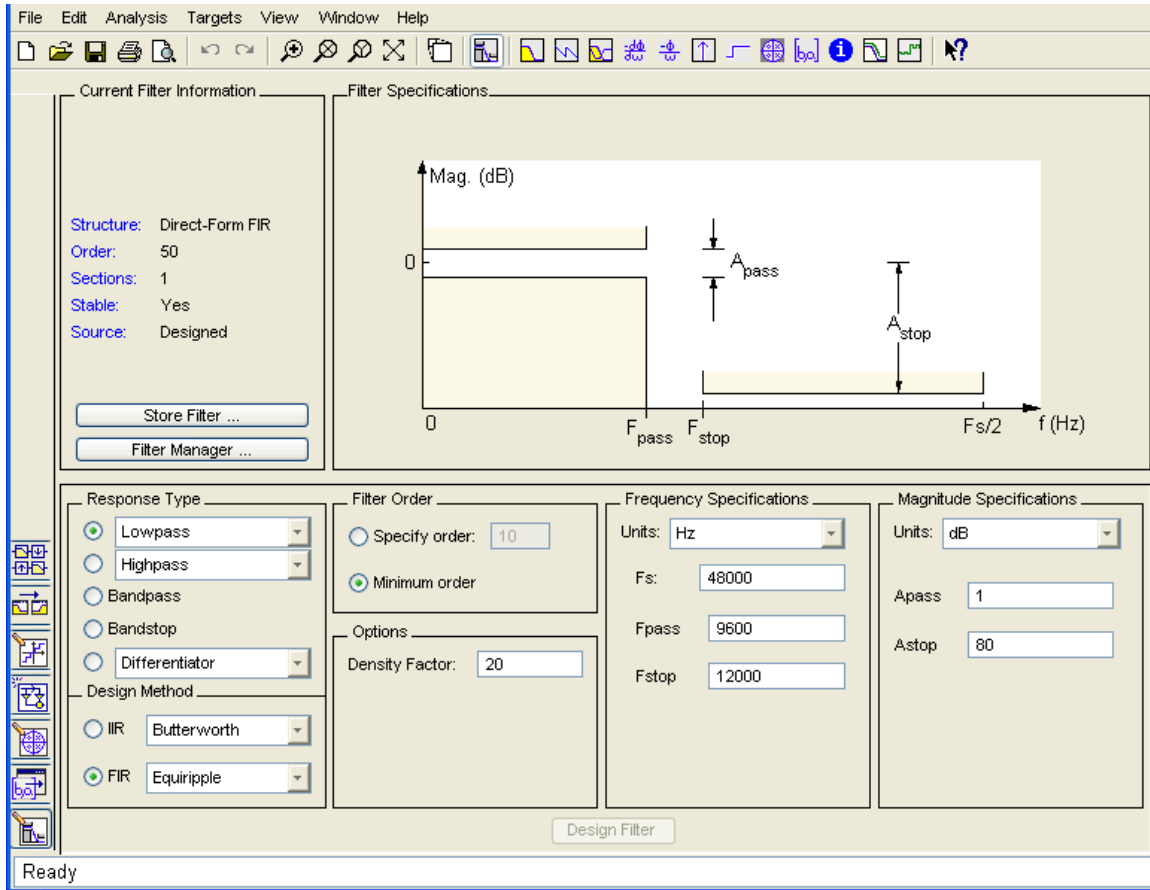
2.1. Designing the Octave-band Filter

10. Start FDATool from the MATLAB command line by typing:

Fdatool

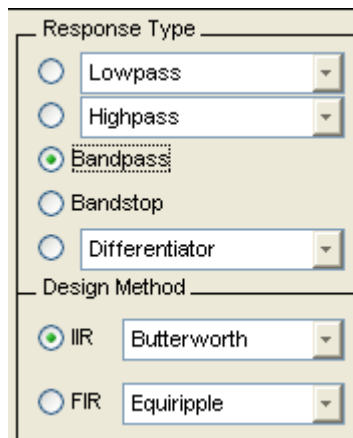
The FDATool dialog opens with a default filter. Its filter information is summarized in the upper left (Current Filter Information) and its filter specifications are depicted in the upper right. In addition to displaying filter specification, this upper right pane displays filter responses and filter coefficients.

The bottom half of FDATool shows the Filter Design panel, where you specify the filter parameters. Other panels, such as Import filter from workspace and Pole/Zero Editor, which you access with the buttons on the lower left, are also displayed in this area. If you have other products installed, you may see additional buttons. Note that when you open FDATool, Design Filter is not enabled. You must make a change to the default filter design in order to enable Design Filter. This is true each time you want to change the filter design. Changes to radio button items or drop down menu items such as those under Response Type or Filter Order enable Design Filter immediately. Changes to specifications in text boxes such as F_s , F_{pass} , and F_{stop} require you to click outside the text box to enable Design Filter.

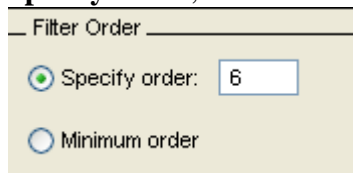


11. In the **Response Type** pane, select **Bandpass**.

12. In the Design Method pane, select IIR, and then select Butterworth.



13. For the Filter Order, select **Specify order**, and then enter 6.



14. Set the Frequency Specifications as follows:

| Parameter | Setting | Description |
|-----------|---------|--|
| Units | Hz | Units for the parameters |
| Fs | 48000 | Sampling frequency |
| Fc1 | 22 | First cutoff frequency (i.e., the frequency preceding the passband at which the magnitude response is 3 dB below the passband gain) |
| Fc2 | 45 | Second cutoff frequency (i.e., the frequency following the passband at which the magnitude response is 3 dB below the passband gain) |

Frequency Specifications

Units:

Fs:

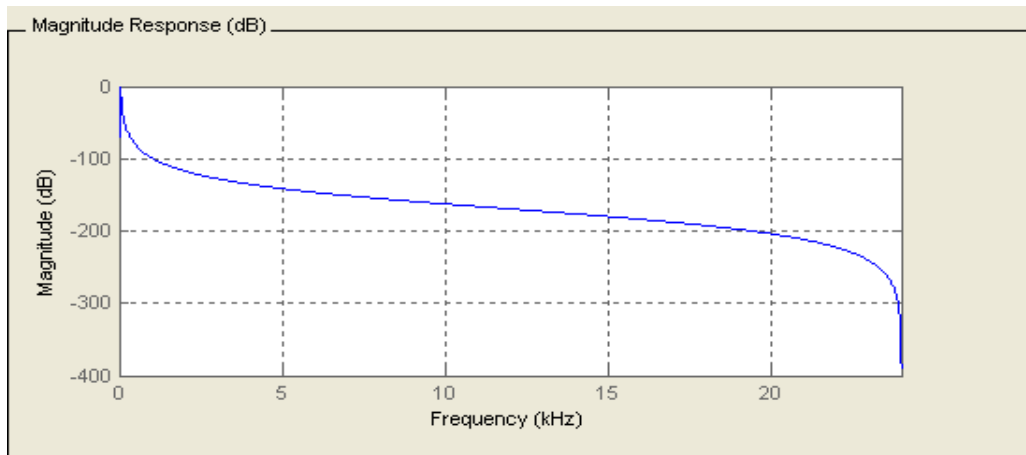
Fc1:

Fc2:

Magnitude Specifications

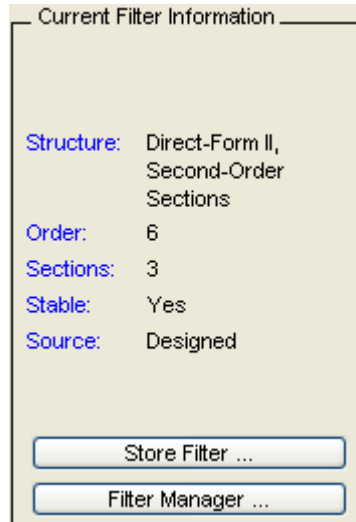
The attenuation at cutoff frequencies is fixed at 3 dB (half the passband gain)

15. After specifying the filter design parameters, click the Design Filter button at the bottom of the design panel to compute the filter coefficients. The display updates to show the magnitude response of the designed filter.

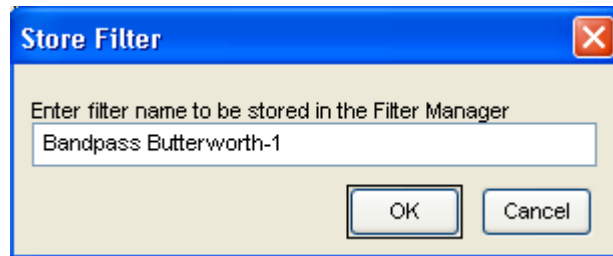


Notice that the Design Filter button is disabled after you compute the coefficients for your filter design. This button is enabled again if you make any changes to the filter specifications.

16. Click the **Store Filter** button.














17. In the Store Filter dialog, change the filter name to Bandpass Butterworth-1 and click OK to save the filter in the Filter Manager.

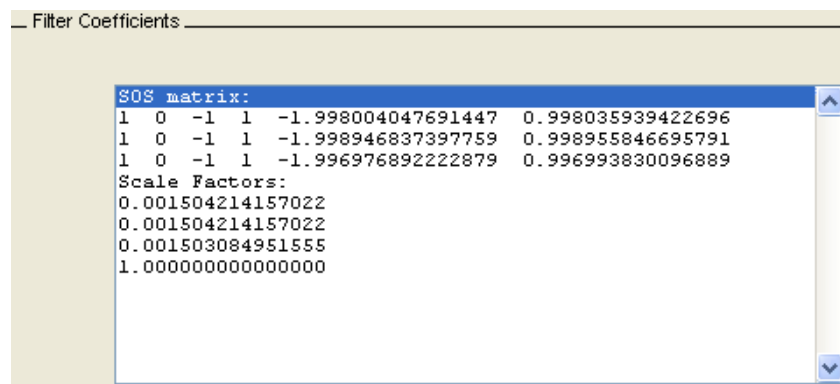


2.2. Analyzing the Octave-band Filter

After designing the filter, you can view the following filter responses in the display region by clicking on the associated toolbar button or by selecting the desired response from the Analysis menu.

| Response | Toolbar Button Image |
|-------------------------------|--|
| Filter specifications |  |
| Magnitude response |  |
| Phase response |  |
| Magnitude and Phase responses |  |
| Group delay |  |
| Phase delay |  |
| Impulse response |  |
| Step response |  |
| Pole-zero plot |  |
| Filter coefficients |  |
| Filter information |  |

18. Examine the displayed magnitude response of the filter.
19. Display other responses, as desired. Click the appropriate buttons, shown in the table above or select the desired response from the Analysis menu.
20. Click the Filter coefficients button to display the filter coefficients.



2.3. Designing and Analyzing the Notch Filter

21. Start FDATool from the MATLAB command line by typing fdatool.

22. In the **Response Type** pane, select **Notching**.
23. In the Design Method pane, select IIR, and then select single notch.
24. In the frequency specifications pane, type 11025 for Fs and 3000 for Fnotch, select Bandwidth and type 200.
25. Click the Design Filter button at the bottom of the design panel to compute the filter coefficients.
26. From the View menu, select filter visualization too, a new window is opened and showing the magnitude response. You need to print or save this figure to include it in your report.
27. In the filter visualization window, select phase response, impulse response and filter coefficients and save or print this information to include it in your report.
28. In the filter design and analysis tool window, select Export from the File menu, a small window with Export title will be opened.
29. In the Export window, select Workspace for Export to, Coefficients for Export As.
30. In the Variable Names, type NumNotch for Numerator and type DenNotch for Denominator, then click Export. The filter coefficients are now available in the present Workspace and can be verified by typing *whos*.

3- Filter Application

In this part, we will use the notch filter designed in the previous part to suppress a single tone from a corrupted speech signal.

31. Load the speech signal stored as a wave signal using the command:
`[Y,FS,NBITS]=wavread('aspect11');`
32. Listen to this speech signal using the command:
`sound(Y,FS)`
33. Generate a 3kHz single tone sinusoidal signal with the same length of the speech signal with the command:
`n=(0:length(Y)-1);X=cos(2*n*pi*3000/FS);`
34. Mix the speech signal with the single tone signal with the command:
`Mix=0.05*X+Y';`
35. Listen to the Mixed speech signal using the command:
`sound(Mix,FS)`

36. Evaluate and plot the amplitude spectrum of both the original speech signal and the mixed signal with the command:
- ```
HY= Single_Sided_Amplitude_Spectrum(Y,FS);
HMix= Single_Sided_Amplitude_Spectrum(Mix,FS);
```
- Notice the presence of high spike at frequency 3kHz in the later spectrum.
37. To suppress the single tone from the corrupted speech signal use the notch filter designed in the previous part. First, verify the response of the filter using the command:
- ```
freqz(NumNotch,DenNotch)
```
- A figure of both the amplitude and phase response of the filter will be created. Then, use the following command to apply the notch filter to the mixed signal:
- ```
YF=filter(NumNotch,DenNotch,Mix);
```
38. Verify the suppression of the single tone from the mixed signal by plotting and listening to the filtered signal YF using the *Single\_Sided\_Amplitude\_Spectrum* and the *sound* functions

## Experiment 2A

### Introduction to the Digital Signal Processing Kit (DSK) and the Code Composer Studio (CCS)

#### Objectives:

- 1- To become familiar with the DSK and its audio connections.
- 2- Test the operation of the DSK and CCS.

#### Lab Equipments:

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphone.

#### Description:

##### 1- The C6713 DSK Board

The DSK packages are powerful, yet relatively inexpensive, with the necessary hardware and software support tools for real - time signal processing. They are complete DSP systems. A simplified block diagram of the DSK is shown in Figure 2A.1.

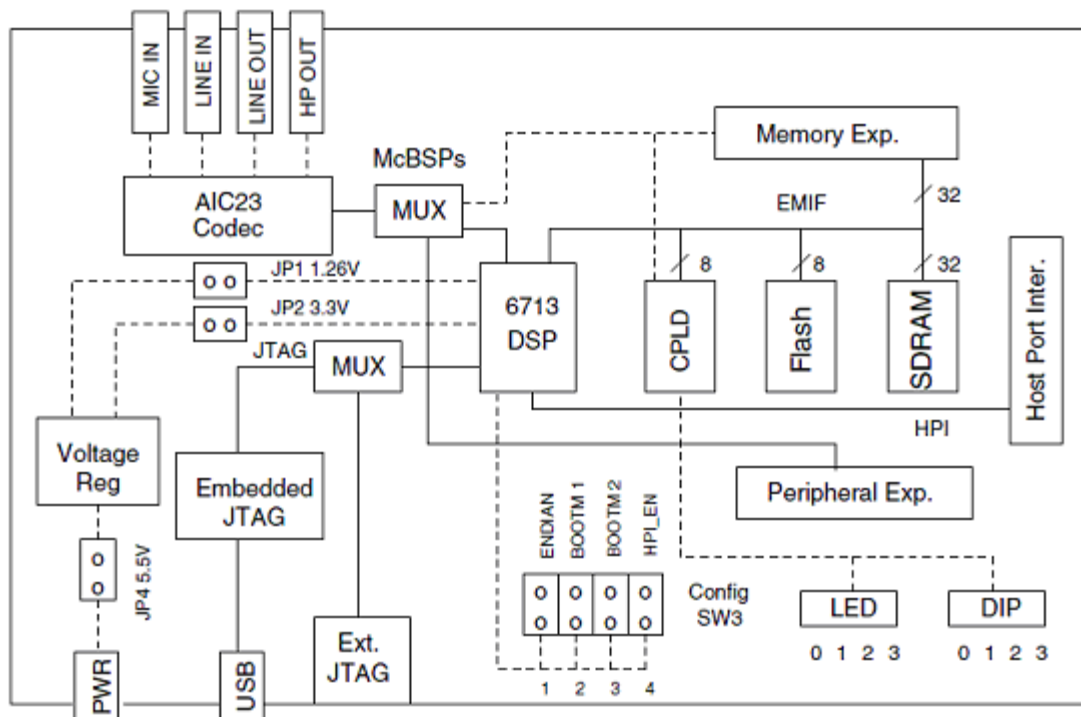


Figure 2A.1 Block Diagram of the TMS320C6713 DSP Starter Kit (DSK)

The major DSK hardware features are:

- A TMS320C6713 DSP operating at 225 MHz.

- 16 - bit stereo codec TLV320AIC23 (AIC23) for analog input and output. The onboard codec AIC23 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
- Two 80 - pin connectors provide for external peripheral and external memory interfaces.
- 16 MB (megabytes) of synchronous dynamic RAM (SDRAM).
- 512 Kbytes of Flash memory (256 Kbytes usable in default configuration).
- 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).
- Four user accessible LEDs and DIP switches.
- Voltage regulators that provide 1.26 V for the DSP cores and 3.3 V for their memory and peripherals.

## **2- Software Support for the DSK Board and 'C6x DSP's**

### **2.1 The Board Support Library (BSL)**

A special Board Support Library (BSL) is supplied with the TMS320C6713 DSK. The BSL provides C-language functions for configuring and controlling all the on-board devices. The library includes modules for general board initialization, access to the AIC23 codec, reading the DIP switches, controlling the LED's, and programming and erasing the Flash memory. The source code for this library is also included. The version of Code Composer supplied with the DSK is set up to automatically use the BSL. You can get complete documentation for the BSL by connecting the DSK to your PC, bring up Code Composer, and going to Help, Contents, TMS320C6713DSK, Software, Board Support Library.

### **2.2 The Chip Support Library (CSL)**

Chip Support Library contains C functions and macros for configuring and interfacing with all the 'C6713 on-chip peripherals and CPU interrupt controller. This library is loaded onto the PC when the DSK software is installed. The CSL header files provide a complete symbolic description of all peripheral registers and register fields.

## **3- Code Composer Studio (CCS)**

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. This executable file can be loaded and run directly on the digital signal processor.

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.

### **3.1 Project Files and Building Programs**

You can build a project in CCS to easily manage an application involving multiple source files, libraries, memory maps, and special command files. The file containing all the project information is given the extension *.pj*. By clicking on the Rebuild All or Incremental build task bar buttons or by menu selections.

### **3.2 The Optimizing Compiler and Assembler**

Code Composer Studio includes a C/C++ optimizing compiler that converts standard ANSI C source programs into C6000 assembly language source. The compiler has several extensions to ANSI C. Assembly statements can be included inline with the C source code. This is useful for manipulating registers in the DSP and using special hardware features that are not efficiently accessible through C. TI has created a language called linear assembly that is part way between pure assembly language and C. Linear assembly source files have the extension *.sa*. In linear assembly you do not have to be concerned with assigning registers. Symbolic names can be used for registers. The assembly optimizer assigns registers and optimizes loops to generate highly parallel assembly code. The assembly source code files generated by the compiler and optimizing assembler must then be passed through the assembler to generate relocatable object modules.

### **3.3 The Linker**

The final step in building a program is to link all the relocatable modules together. The linker, *lnk6x.exe*, combines relocatable object modules to form an executable output

program. The default extension for executable programs is out. In addition, the linker can generate a map file showing the absolute memory addresses of all global variables. A very important input to the linker is a linker command file which has the extension cmd. The command file can contain names of additional object modules to link, paths to libraries, names for the map and out files, a memory map for the target hardware system, and commands describing where to put specific program sections in memory.

#### **4- 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 processor.
10. file.cdb : configuration file when using DSP/BIOS.

### **LAB WORK:**

#### **1- QUICK TESTS OF THE DSK (ON POWER ON AND USING CCS)**

1. Check out the hardware. Find the three audio connectors for the DSK. They are MIC IN, LINE IN, and LINE OUT. The MIC IN jack is for low level signals from a microphone.

You will be using the commercial signal generator for this course and should use only the LINE IN and LINE OUT connectors for these larger signal levels. Beware that a common mistake of lab students is to make the input too large and saturate the input amplifiers resulting in strange outputs.

2. 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. 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.
3. Launch CCS from the icon on the desktop. A USB enumeration process will take place and the Code Composer Studio window will open.



4. 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.
5. 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    CPLDRevision: 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.

6. Click on *Debug* → ☐ *Disconnect*

## **2- Alternative Quick Test of DSK**

7. Open/launch CCS from the icon on the desktop if not done already.
8. Select *Debug* → ☐ *Connect* and check that the symbol in the bottom left – hand corner of the CCS window indicates connection to the DSK.
9. 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.
10. 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*.

11. Select *Debug* → ☐ *Disconnect*.
12. Close the CCS program.

## Experiment 2B

### Sine Wave Generation Using Eight points with DIP Switch Control

#### Objectives:

- 1- To generate a sinusoidal analog output waveform using a table-lookup method.
- 2- To illustrate some of the features of the CCS for editing source files, building a project, accessing the code generation tools, and running a program on the C6713 processor.

#### Lab Equipments:

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphone.

#### Description:

The C source file *sine8\_LED.c* listed in Figure 2B.1 is included in the folder *sine8\_LED*.

```
//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
 }
}
```

**FIGURE 2B.1** Sine wave generation program using eight points with DIP switch control (*sine8\_LED.c*).

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

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

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 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 8 kHz, 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 8 kHz. 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 (8 kHz) divided by eight, that is, 1 kHz.

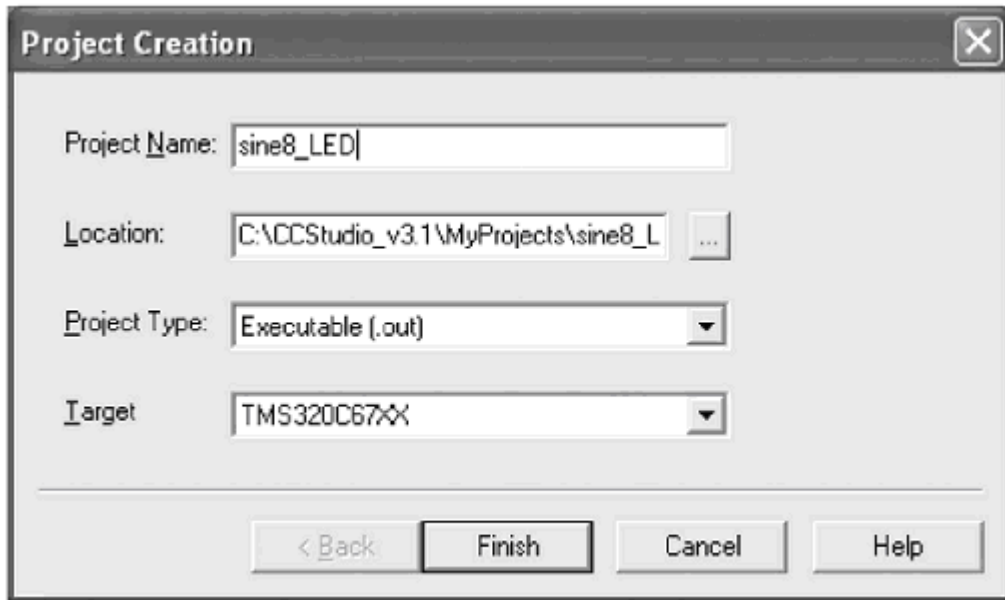
## LAB WORK:

### 1- Creating a Project

This experiment illustrates how to create a project, adding the necessary files to generate an executable file **sine8\_LED.out**. a file named *sine8\_LED.pjt* already exist at *c:\CCStudio\_v3.1\MyProjects\sine8\_LED*. However, for the purposes of gaining familiarity with CCS, this experiment 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.
2. **Launch CCS** by double - clicking on its desktop icon.

3. Make a quick test on the DSK.
4. **Create a new project file `sine8_LED.pjt`** by selecting *Project*→☐New and typing *sine8\_LED* as the project name, as shown in Figure 2B.2. **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.



**FIGURE 2B.2** CCS *Project Creation* window for project *sine8\_LED*.

5. **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*, or double - click on, `sine8_LED.c`. It should appear in the *Project View* window *Source* folder.
6. **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.
7. **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.

8. **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.
9. **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.
10. 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.
11. **The *Project View* window in CCS should look as shown in Figure 2B.3.** 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.)  
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.

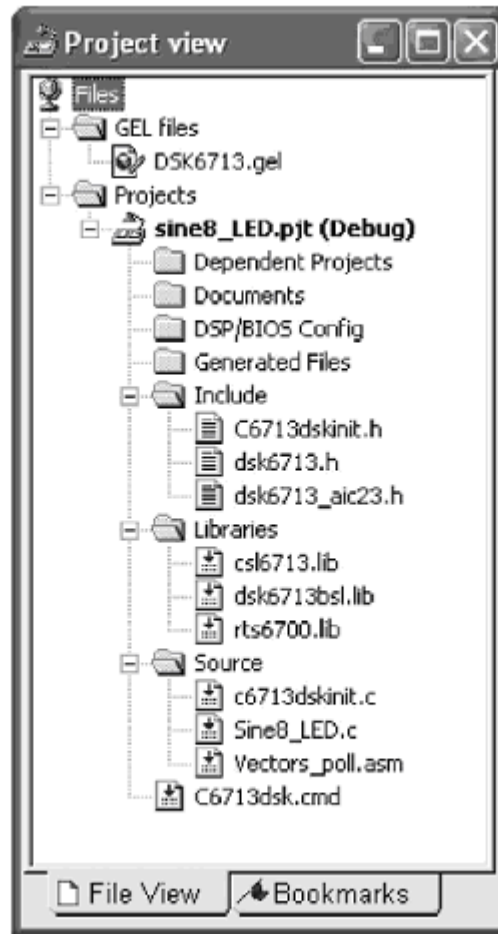


FIGURE 2B.3 Project View window showing files added at step 10.

## 2- 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.

### 2.1. Setting Compiler Options

Select *Project* → ☐ *Build Options* and click on the *Compiler* tab. Set the following options, as shown in Figures 2B.4, 2B.5, and 2B.6. In the *Basic* category set *Target Version* to *C671x (- mv6710)*. In the *Advanced* category set *Memory Models* to *Far ( - mem\_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*. Click on *OK*.

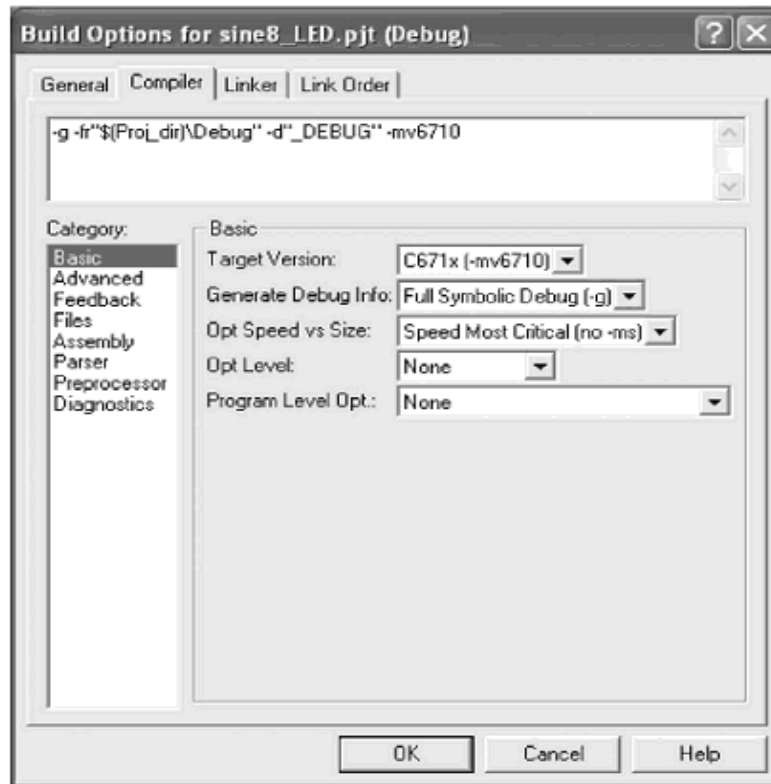


FIGURE 2B.4 CCS Build Options: Basic compiler settings.

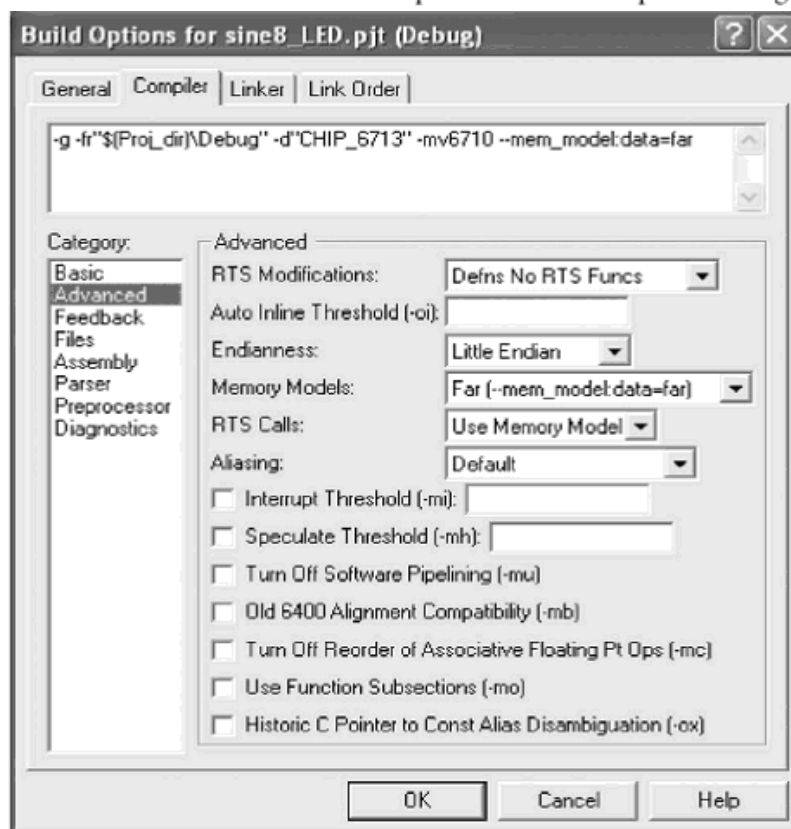
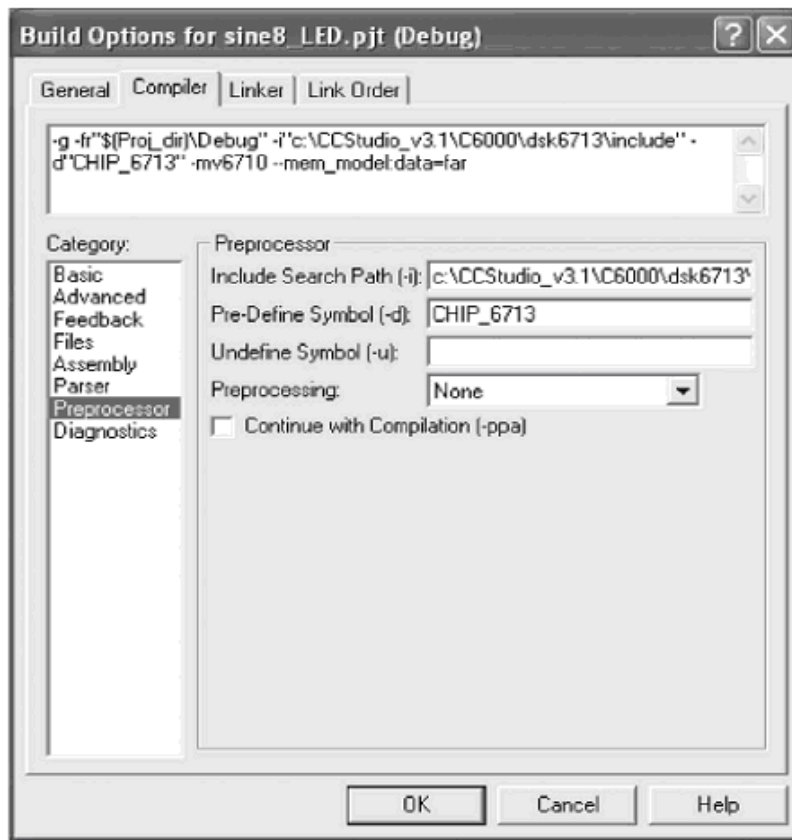


FIGURE 2B.5 CCS Build Options: Advanced compiler settings.



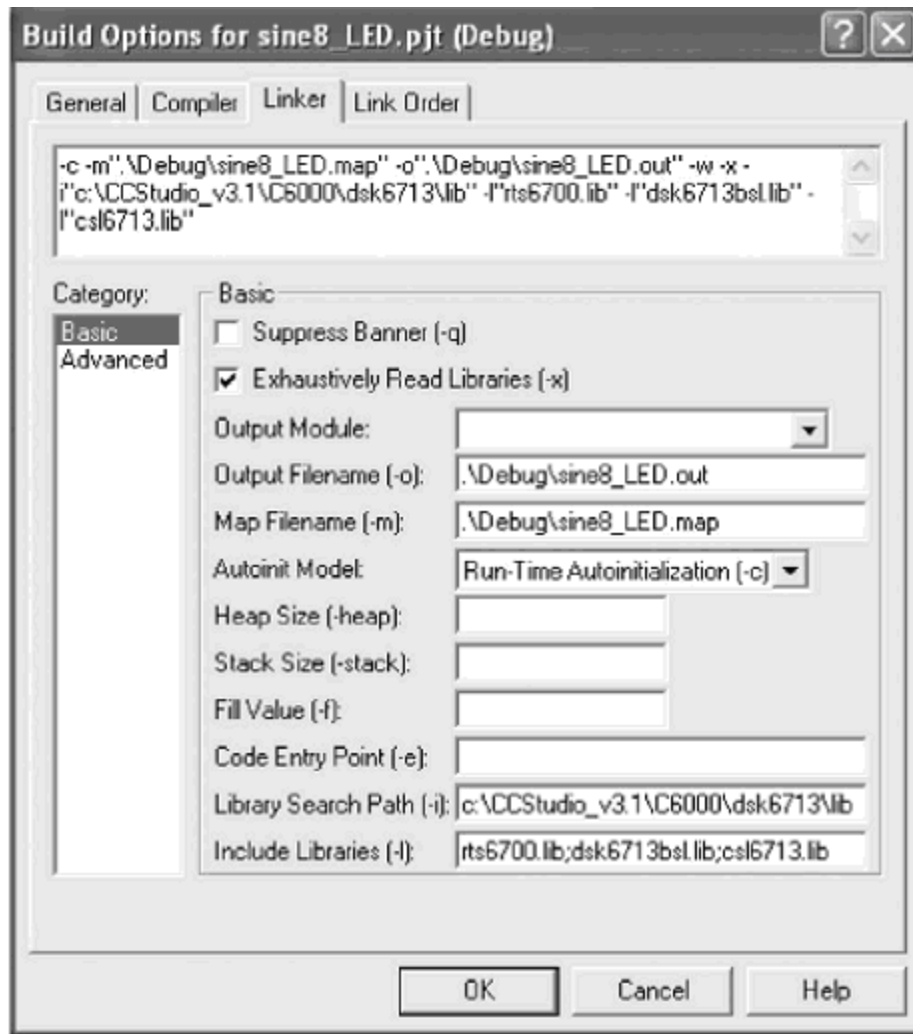


**FIGURE 2B.6** CCS Build Options: Preprocessor compiler settings.

## 2.2. Setting Linker Options

Click on the *Linker* tab in the *Build Options* window, as shown in Figure 2B.7. 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;cs16713.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*.





**FIGURE 2B.7** CCS Build Options: Basic Linker settings.

### 3- 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.

12. 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.

13. 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.

#### 4- 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:

14. Select *View* →☐ *Quick Watch*. Type *gain*, and then click on *Add to Watch*. The gain value of 10 set in the program in Figure 2B.1 should appear in the Watch window.
15. 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.

#### 5- 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.

16. Select *File* →☐ *Load GEL* and load the file gain.gel (in folder sine8\_LED ).
17. Double - click on the filename *gain.gel* in the *Project View* window to view it within CCS. The file is listed in Figure 2B.8. The format of a slider GEL function is

```

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

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

menuitem "Sine Gain"

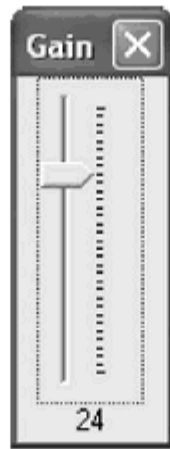
slider Gain(0,30,4,1,gain_parameter) /*incr by 4, up to 30*/
{
 gain = gain_parameter; /*vary gain of sine*/
}

```

**FIGURE 2B.8** Listing of GEL file gain.gel.

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 param-Name when the slider is at its lowest level, maxVal is the value assigned to the 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.

18. Select *GEL* → ☐ *Sine Gain* → ☐ *Gain*. This should bring out the slider window shown in Figure 2B.9, with the minimum value of 0 set for the gain.
19. 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 2B.9** GEL slider used to vary gain in program `sine8_LED.c`.

## 6- 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.

20. 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.
21. 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).

## **Experiment 2C**

### **Generation of sinusoid and Plotting with CCS (*sine8\_buf*)**

#### **Objectives:**

- 1- To generate a sinusoidal analog output waveform using eight pre-calculated and pre-stored sample values.
- 2- To illustrate the capabilities of CCS for plotting data in both time and frequency domains.

#### **Lab Equipments:**

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphone.

#### **Description:**

This example generates a sinusoidal analog output signal using eight pre[calculated and pre]stored 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 2C.1.

This program uses interrupt- driven input/output rather than polling, so that 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`. 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()`.

Because a project file `sine8_buf.pjt` is supplied, there is no need to create a new project file, add files to it, or alter compiler and linker build options. In order to build, download and run program `sine8_buf.c`.

```

//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 2C.1** Listing of program `sine8_buf.c`.

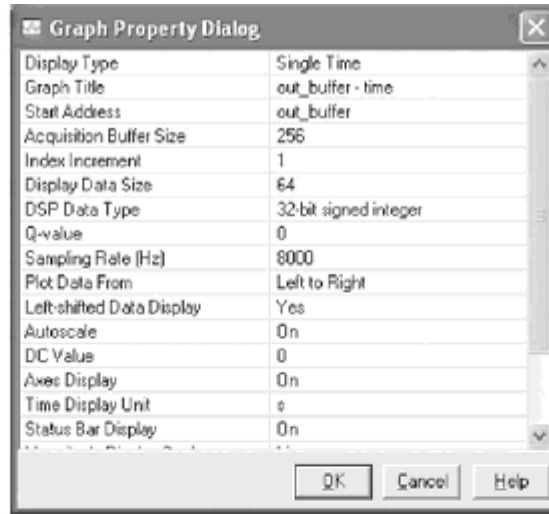
## LAB WORK:

1. **Launch CCS** by double - clicking on its desktop icon.
2. Make a quick test on the DSK.
3. Open project *sine8\_buf.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *sine8\_buf.pjt* in folder *sine8\_buf*.
4. 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

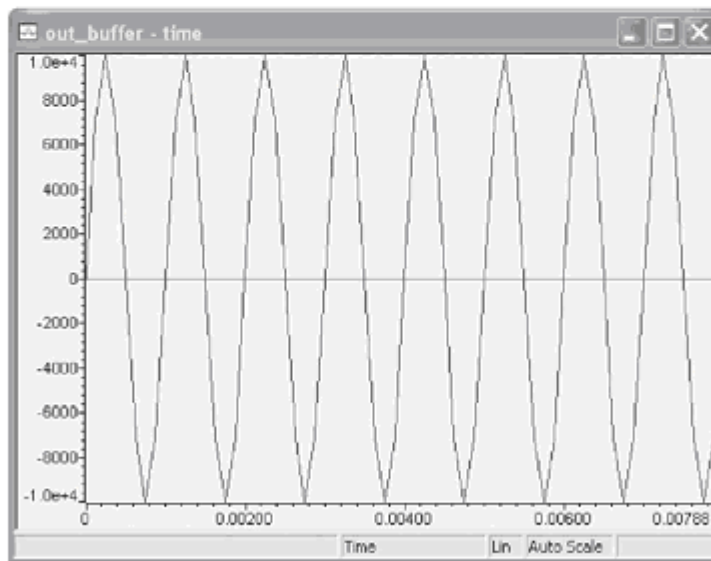
## Graphical Displays in CCS

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.

5. Select *View* → ☐ *Graph* → ☐ *Time/Frequency* and set the *Graph Property Dialog* properties as shown in Figure 2C.2.a. Figure 2C.2.b. shows the resultant *Graphical Display* window.
6. Figure 2C.3.a shows the *Graph Property Dialog* window that corresponds to the frequency domain representation of the contents of *out\_buffer* shown in Figure 2C.3.b. The spike at 1 kHz represents the frequency of the sinusoid generated by program *sine8\_buf.c*.



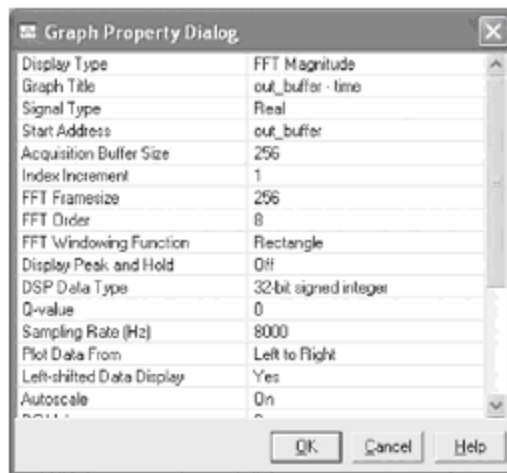
(a)



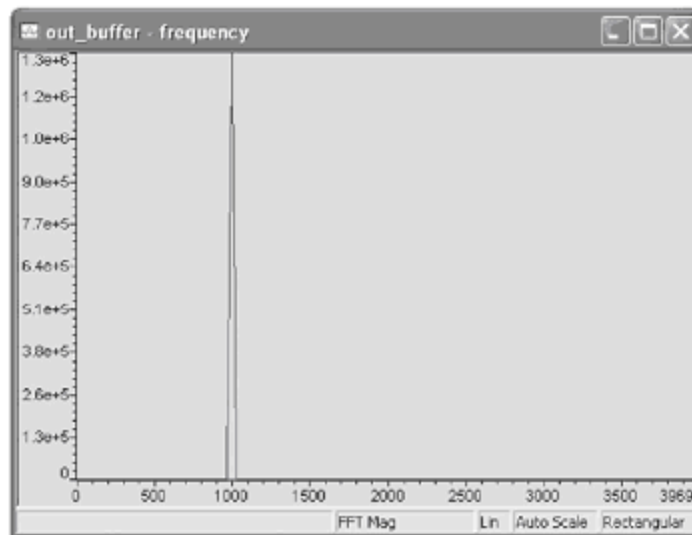
(b)

**FIGURE 2C.2** (a) *Graph Property* window and (b) *Time domain* plot of data stored in *out\_buffer*.





(a)



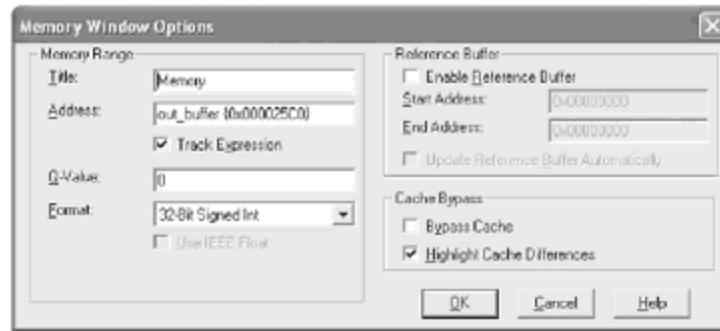
(b)

**FIGURE 2C.3** (a) *Graph Property* window and (b) Frequency domain plot of data stored in `out_buffer`.

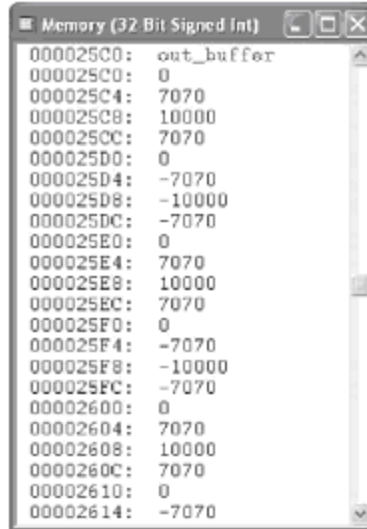
### **Viewing and Saving Data from Memory into File**

7. 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 2C.4.a. The resultant *Memory* window is shown in Figure 2C.4.b.
8. 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.



(a)



(b)

**FIGURE 2C.4** (a) *Memory* window settings and (b) *Memory* window view of data stored in `out_buffer`.

## Experiment 3A

### Basic Input and Output Using Polling (loop\_poll)

#### Objectives:

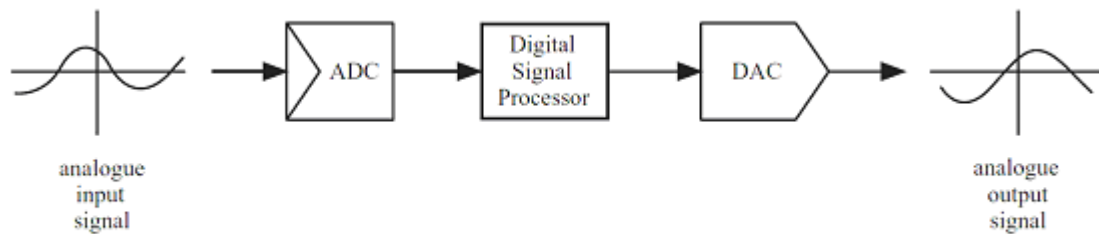
- 1- Sampling and reconstruction of real time signals.
- 2- Gain change of the line input.

#### Lab Equipments:

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphone.
- 5- Signal Generator.

#### Description:

A basic DSP system, suitable for processing audio frequency signals, comprises a digital signal processor and analog interfaces as shown in Figure 2A.1. The C6713 provides just such a system, using the TMS320C6713 (C6713) floating - point processor and the TLV320AIC23 (AIC23) codec. The term codec refers to the *coding* of analog waveforms as digital signals and the *decoding* of digital signals as analog waveforms. The AIC23 codec performs both the analog - to - digital conversion (ADC) and digital - to - analog conversion (DAC) functions shown in Figure 3A.1. Alternatively, I/O daughter cards, plugged into the External Peripheral Interface 80 - pin connector J3 on the DSK board can be used for analog input and output.



**FIGURE 3A.1** Basic digital signal processing system.

Within digital signal processors, signals are represented as sequences of discrete samples and whenever signals are sampled, the possibility of aliasing arises. Aliasing is undesirable phenomena and it may be avoided by the use of an antialiasing filter placed at the input to the system shown in Figure 3A.1 and by suitable design of the DAC. In a baseband system, an effective antialiasing filter is one that allows frequency components below half of the sampling frequency to pass but which attenuates greatly, or stops, frequency components equal to or greater than half of the sampling frequency. A suitable DAC for a baseband system essentially comprises a lowpass filter having characteristics similar to the aforementioned antialiasing filter. The AIC23 codec contains digital antialiasing and reconstruction filters.

The C6713 DSK makes use of the TLV320AIC23 (AIC23) codec for analog input and output. The analog - to - digital converter (ADC), or coder, part of the codec converts an analog input signal into a sequence of sample values (16 - bit signed integer) to be processed by the digital signal processor. The digital - to - analog converter (DAC), or decoder, part of the codec reconstructs an analog output signal from a sequence of sample values (16 - bit signed integer) that have been processed by the digital signal processor. The AIC23 is a stereo audio codec based on sigma - delta technology.

A 12 - MHz crystal supplies the clock to the AIC23 codec (also to the DSP and the USB interface). Using this 12 - MHz master clock, with oversampling rates of  $250F_s$  and  $272F_s$ , exact audio sample rates of 48 kHz (12 MHz/250) and the CD rate of 44.1 kHz (12 MHz/272) can be obtained. The sampling rate of the AIC23 can be configured to be 8, 16, 24, 32, 44.1, 48, or 96 kHz.

Communication with the AIC23 codec for input and output uses two multichannel buffered serial ports (McBSPs) on the C6713. McBSP0 is used as a unidirectional channel to send a 16 - bit control word to the AIC23. McBSP1 is used as a bidirectional channel to send and receive audio data. The codec can be configured for data - transfer word-lengths of 16, 20, 24, or 32 bits. The LINE IN and HEADPHONE OUT signal paths within the codec contain configurable gain elements with ranges of 12 to - 34 dB in steps of 1.5 dB, and 6 to - 73 dB in steps of 1 dB, respectively. Most of the programming examples in this booklet configure the codec for a sampling rate of 8 kHz, 32 - bit data transfer, and 0 - dB gain in the LINE IN and HEADPHONE OUT signal paths. The maximum allowable input signal level at the LINE IN inputs to the codec is 1 V rms. However, the C6713 DSK contain a potential divider circuit with a gain of 0.5 between their LINE IN sockets and the codec itself with the effect that the maximum allowable input signal level at the LINE IN sockets on the DSK is 2 V rms. Above this level, input signals will be distorted. Input and output sockets on the DSK are ac coupled to the codec.

The C language source file for a program, `loop_poll.c`, that simply copies input samples read from the AIC23 codec ADC back to the AIC23 codec DAC as output samples is listed in Figure3A.2. Effectively, the MIC input socket is connected straight through to the HEADPHONE OUT socket on the DSK via the AIC23 codec and the digital signal processor. `loop_poll.c` uses the same polling technique for real - time input and output as program `sine8_LED.c`, presented in previous experiment .

```

//loop_poll.c loop program using polling

#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

void main()
{
 short sample_data;

 comm_poll(); //init DSK, codec, McBSP
 while(1) //infinite loop
 {
 sample_data = input_left_sample(); //input sample
 output_left_sample(sample_data); //output sample
 }
}

```

**FIGURE 3A.2** Loop program using polling (loop\_poll.c).

### Input and Output Functions Defined in Support File c6713dskinit.c

The functions `input_left_sample()`, `output_left_sample()`, and `comm_poll()` are defined in the support file `c6713dskinit.c`. This way the C source file `loop_poll.c` is kept as small as possible and potentially distracting low level detail is hidden. The implementation details of these, and other, functions defined in `c6713dskinit.c` need not be studied in detail in order to carry out the examples presented in this booklet but are described here for completeness.

Further calls are made by `input_left_sample()` and `output_left_sample()` to lower level functions contained in the board support library `DSK6713bsl.lib`. Function `comm_poll()` initializes the DSK and, in particular, the AIC23 codec such that its sample rate is set according to the value of the variable `fs` (assigned in `loop_poll.c`), its input source according to the value of the variable `inputsource` (assigned in `loop_poll.c`), and polling mode is selected. Other AIC23 configuration settings are determined by the parameters specified in file `c6713dskinit.h`. These parameters include the gain settings in the LINE IN and HEADPHONE out signal paths, the digital audio interface format, and so on. Similar values for all of these parameters are used by almost all of the program examples in this booklet. Only rarely will they be changed and so it is convenient to hide them out of the way in file `c6713dskinit.h`.

The two settings, sampling rate and input source, are changed sufficiently frequently, from one program example to another, that their values are set in each example program by initializing the values of the variables `fs` and `inputsource`. In function `dsk6713_init()` in file `c6713dskinit.c`, these values are used by functions `DSK6713_AIC23_setFreq()` and `DSK6713_AIC23_rset()`, respectively. In polling mode, function `input_left_sample()`

polls, or tests, the receive ready bit ( RRDY ) of the McBSP serial port control register ( SPCR ) until this indicates that newly converted data is available to be read using function MCBSP\_read(). Function output\_left\_sample() polls, or tests, the transmit ready bit ( XRDY ) of the McBSP serial port control register ( SPCR ) until this indicates that the codec is ready to receive a new output sample. A new output sample is sent to the codec using function McBSP\_write() .

Although polling is simpler than the interrupt technique used in sine8\_buf.c, it is less efficient since the processor spends nearly all of its time repeatedly testing whether the codec is ready either to transmit or to receive data.

### **LAB WORK:**

1. **Launch CCS** by double - clicking on its desktop icon.
2. Make a quick test on the DSK.
3. Open project *loop\_poll.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *loop\_poll.pjt* in folder *loop\_poll*.
4. Load and run the executable file *loop\_poll.out*.
5. Use a microphone and headphones to verify that the program operates as intended.
6. Rebuild the program having changed the line that reads  
    `Uint16 inputsource=DSK6713_AIC23_INPUT_MIC;`  
to read  
    `Uint16 inputsource=DSK6713_AIC23_INPUT_LINE;`  
in order to select the LINE IN rather than the MIC socket on the DSK.
7. Input a sinusoidal waveform to the LINE IN connector on the DSK, with amplitude of approximately 2.0 V p - p and a frequency of approximately 1 kHz.
8. Connect the output of the DSK, LINE OUT, to an oscilloscope, and verify the presence of a tone of the same frequency, but attenuated to approximately 1.0 V p - p.
9. Explain the attenuation occurred.
10. Increase the amplitude of the input sinusoidal waveform (at the LINE IN socket) beyond 5V p – p. and verify that the output signal becomes distorted. Why?

### ***Changing the LINE IN Gain of the AIC23 Codec***

The AIC23 codec allows for the gain on left - and right - hand line - in input channels to be adjusted independently in steps of 1.5 dB by writing different values to the left and

right line input channel volume control registers. The values assigned to these registers by function `comm_poll()` are defined in the header file `c6713dskinit.h`. In order to change the values written, that file must be modified.

11. Copy the files `c6713dskinit.h` and `C6713dskinit.c` from the Support folder into the folder `loop_poll` so that you don't modify the original header file.
12. Remove these two files from the `loop_poll` project by right - clicking on `c6713dskinit.c` in the Project View window and then selecting **Project** → **Remove from Project**.
13. Add the copy of the file `c6713dskinit.c` in folder `loop_poll` to the project by selecting **Project** → **Add Files to Project**.
14. Check that you have added the copy of file `c6713dskinit.c` to the project by right - clicking on it in the Project View window and selecting **Properties**.
15. Select **Project** → **Scan all Dependencies** in order to replace the file `c6713dskinit.h` with the copy in folder `loop_poll`.
16. Edit the copy of file `c6713dskinit.h` included in the project (and stored in folder `loop_poll`), changing the line that reads  
    `0x0017 / * Set -Up Reg 0 Left line volume control */`  
    to read  
    `0x001B / * Set -Up Reg 0 Left line volume control */`  
This modifies the value written to the AIC23 left line input channel gain register from `0x0017` to `0x001B` and this increases the gain from 0 dB to 6 dB.
17. Build the project, making sure that the copy of the file `c6713dskinit.c` used in the project is the copy in folder `loop_poll`. The header file `c6713dskinit.h` that will be included will come from that same folder
18. Load and run the executable file `loop_poll.out` and verify that the output signal is not attenuated, but has the same amplitude as the input signal, that is,  $2 V_{p-p}$ .

## **Experiment 3B**

### **Basic Input and Output Using Interrupts (loop\_intr)**

#### **Objectives:**

- 1- Sampling and reconstruction of real time signals using the interrupt driven model.
- 2- Demonstration of echo and delay effects.

#### **Lab Equipments:**

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphone.
- 5- Signal Generator.

#### **Description:**

Program loop\_intr.c is functionally equivalent to program loop\_poll.c but makes use of interrupts. This simple program is important because many of the other example programs in this booklet are based on the same interrupt - driven model. Instead of simply copying the sequence of samples representing an input signal to the codec output, a digital filtering operation can be performed each time a new input sample is received. It is worth taking time to ensure that you understand how program loop\_intr.c works. In function main() , the initialization function comm\_intr() is called. comm\_intr() is very similar to comm\_poll() but in addition to initializing the DSK, codec, and McBSP, and not selecting polling mode, it sets up interrupts such that the AIC23 codec will sample the analog input signal and interrupt the C6713 processor, at the sampling frequency defined by the line

```
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
```

It also initiates communication with the codec via the McBSP. In this example, a sampling rate of 8 kHz is used and interrupts will occur every 0.125 ms. (Sampling rates of 16, 24, 32, 44.1, 48, and 96 kHz are also possible.)

Following initialization, function main() enters an endless while loop, doing nothing but waiting for interrupts. The functions that will act as interrupt service routines for the various different interrupts are specified in the interrupt service table contained in file vectors\_intr.asm. This assembly language file differs from the file vectors\_poll.asm in that function c\_int11() is specified as the interrupt service routine for interrupt INT11. On interrupt, the interrupt service routine (ISR) c\_int11() is called and it is within that routine that the most important program statements are executed. Function output\_left\_sample() is used to output a value read from the codec using function input\_left\_sample() .



### *Format of Data Transferred to and from AIC 23 Codec*

The AIC23 ADC converts left - and right - hand channel analog input signals into 16 - bit signed integers and the DAC converts 16 - bit signed integers to left - and right - hand channel analog output signals. Left - and right - hand channel samples are combined to form 32 - bit values that are communicated via the multichannel buffered serial port (McBSP) to and from the C6713. Access to the ADC and DAC from a C program is via the functions `Uint32 input_sample()`, `short input_left_sample()`, `short input_right_sample()`, `void output_sample(int out_data)`, `void output_left_sample(short out_data)`, and `void output_right_sample(short out_data)`.

The 32 - bit unsigned integers (`Uint32`) returned by `input_sample()` and passed to `output_sample()` contain both left and right channel samples. The statement

```
union {
 Uint32 uint;
 Short channel [2];
} AIC_data;

//loop_intr.c loop program using interrupts

#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

interrupt void c_int11() //interrupt service routine
{
 short sample_data;

 sample_data = input_left_sample(); //input data
 output_left_sample(sample_data); //output data
 return;
}

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

**FIGURE 3B.1** Loop program using interrupts (`loop_intr.c`).

In file `dsk6713init.h` declares a variable that may be handled either as one 32 - bit unsigned integer (`AIC_data.uint`) containing left and right channel sample values, or as two 16 - bit signed integers (`AIC_data.channel[0]` and `AIC_data.channel[1]`).

Most of the program examples in this booklet use only one channel for input and output and for clarity most use the functions `input_left_sample()` and `output_left_sample()`. These functions are defined in the file `c6713dskinit.c`, where the unpacking and packing

of the signed 16 - bit integer left - hand channel sample values out of and into the 32 - bit words received and transmitted from and to the codec are carried out.

### ***Modifying Program loop\_intr.c to Create a Delay ( delay)***

Some simple, yet striking, effects can be achieved simply by delaying the samples as they pass from input to output. Program delay.c, listed in Figure 3B.2 , demonstrates this. A delay line is implemented using the array buffer to store samples as they are read from the codec. Once the array is full, the program overwrites the oldest stored input sample with the current, or newest, input sample. Just prior to overwriting the oldest stored input sample in buffer , that sample is retrieved, added to the current input sample, and output to the codec. Figure 3B.3 shows a block diagram representation of the operation of program delay.c in which the block labeled T represents a delay of T seconds.

```
//delay.c Basic time delay

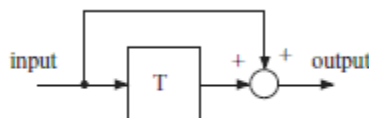
#include "DSK6713_AIC23.h" //codec support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; //select input

#define BUF_SIZE 8000
short input,output,delayed;
short buffer[BUF_SIZE];
int i;

interrupt void c_int11() //interrupt service routine
{
 input = input_left_sample(); //read new input sample
 delayed = buffer[i]; //read output of delay line
 output = input + delayed; //output sum of new and delayed
 buffer[i] = input; //replace delayed sample with
 if(++i >= BUF_SIZE) i=0; //new input sample then increment
 output_left_sample(output); //buffer index
 return; //return from ISR
}

void main()
{
 for(i=0 ; i<BUF_SIZE ; i++)
 buffer[i] = 0;
 comm_intr(); //init DSK, codec, McBSP
 while(1); //infinite loop
}
```

**FIGURE 3B.2** Delay program using interrupts (delay.c).



**FIGURE 3B.3** Block diagram representation of program delay.c.

### *Modifying Program loop\_intr.c to Create an Echo ( echo)*

By feeding back a fraction of the output of the delay line to its input, a fading echo effect can be realized. Program echo.c, listed in Figure 3B.4, does this. Figure 3B.5 shows a block diagram representation of the operation of program echo.c. The value of the constant BUF\_SIZE determines the number of samples stored in the array buffer and hence the duration of the delay. The value of the constant GAIN determines the fraction of the output that is fed back into the delay line and hence the rate at which the echo effect fades away. Setting the value of GAIN equal to or greater than unity would cause instability of the loop.

```
//echo.c echo with fixed delay and feedback

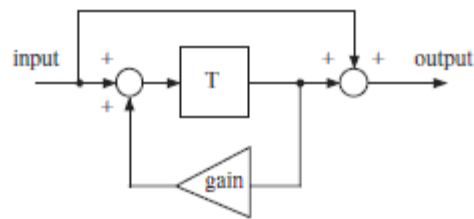
#include "DSK6713_AIC23.h" //codec support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; //select input

#define GAIN 0.6 //fraction of output fed back
#define BUF_SIZE 2000 //this sets length of delay
short buffer[BUF_SIZE]; //storage for previous samples
short input,output,delayed;
int i; //index into buffer

interrupt void c_int11() //interrupt service routine
{
 input = input_left_sample(); //read new input sample from ADC
 delayed = buffer[i]; //read delayed value from buffer
 output = input + delayed; //output sum of input and delayed
 output_left_sample(output);
 buffer[i] = input + delayed*GAIN; //store new input and
 //fraction of delayed value
 if(++i >= BUF_SIZE) i=0; //test for end of buffer
 return; //return from ISR
}

void main()
{
 comm_intr(); //init DSK, codec, McBSP
 for(i=0 ; i<BUF_SIZE ; i++) //clear buffer
 buffer[i] = 0;
 while(1); //infinite loop
}
```

**FIGURE 3B.4**Fading echo program (echo.c).



**FIGURE 3B.5**Block diagram representation of program echo.c.

Build and run this project as echo. Experiment with different values of GAIN (between 0.0 and 1.0) and BUF\_SIZE (between 100 and 8000). Source file echo.c must be edited and the project rebuilt in order to make these changes.

### ***Loop Program with Input Data Stored in a Buffer ( loop\_buf )***

Program loop\_buf.c , listed in Figure 3B.6 , is an interrupt - based program and is stored in folder loop\_buf . It is similar to program loop\_intr.c except that it maintains a circular buffer in array buffer containing the BUF\_SIZE most recent input sample values. Consequently, it is possible to display this data in CCS after halting the program.

```
//loop_buf.c loop program with storage

#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_LINE; //select input
#define BUFSIZE 512

int buffer[BUFSIZE];
int buf_ptr = 0;

interrupt void c_int11() //interrupt service routine
{
 int sample_data;

 sample_data = input_left_sample(); //read input sample
 buffer[buf_ptr] = sample_data; //store in buffer
 if(++buf_ptr >= BUFSIZE) buf_ptr = 0; //update buffer index
 output_left_sample(sample_data); // write output sample
 return;
}

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

**FIGURE 3B.6** Loop program with input data stored in memory (loop\_buf.c).

## LAB WORK:

1. **Launch CCS** by double - clicking on its desktop icon.
2. Make a quick test on the DSK.
3. Open project *loop\_intr.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *loop\_intr.pjt* in folder *loop\_intr*.
4. Load and run the executable file *loop\_intr.out*.
5. Use a microphone and headphones to verify that the program operates as intended.
6. Halt the executable file *loop\_intr.out*.
7. Open project *delay.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *delay.pjt* in folder *delay*.
8. Load and run the executable file *delay.out*.
9. Use a microphone and headphones to verify that the program operates as intended.
10. Halt the executable file *delay.out*.
11. Open project *echo.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *echo.pjt* in folder *delay*.
12. Load and run the executable file *echo.out*.
13. Experiment with different values of GAIN (between 0.0 and 1.0) and BUF\_SIZE (between 100 and 8000). Source file *echo.c* must be edited and the project rebuilt in order to make these changes.
14. Halt the executable file *echo.out*.
15. Use a signal generator connected to the LINE IN socket to input a sinusoidal signal with a frequency between 100 and 3500 Hz.
16. Open project *loop\_buf.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *loop\_buf.pjt* in folder *loop\_buf*.
17. Load and run the executable file *loop\_buf.out*.
18. Halt the program after a short time.
19. Select View → Graph → Time/Frequency in order to display the contents of array buffer.

## Experiment 3C

### Sine Wave Generation Using sin() Function,

### Reconstruction, Aliasing, and the Properties of the AIC 23

### Codec

#### Objectives:

- 1- To generate sinusoidal signal using mathematical function.
- 2- Demonstration of aliasing in real time system.
- 3- To visualize the step response of the AIC23 Codec Anti-aliasing Filter

#### Lab Equipments:

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Signal Generator.

#### Description:

Generating analog output signals using programs such as sine\_intr.c (Figure 3C.1) is a useful means of investigating the characteristics of the AIC23 codec.

```
//sine_intr.c Sine generation using sin() function

#include <math.h>
#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 SAMPLING_FREQ 8000
#define PI 3.14159265358979

float frequency = 1000.0;
float amplitude = 10000.0;
float theta_increment;
float theta = 0.0;

interrupt void c_int11()
{
 theta_increment = 2*PI*frequency/SAMPLING_FREQ;
 theta += theta_increment;
 if (theta > 2*PI) theta -= 2*PI;
 output_left_sample((short)(amplitude*sin(theta)));
 return;
}

void main()
{
 comm_intr();
 while(1);
}
```

**FIGURE 3C.1** Sine wave generation program using call to math function sin() (sine\_intr.c).

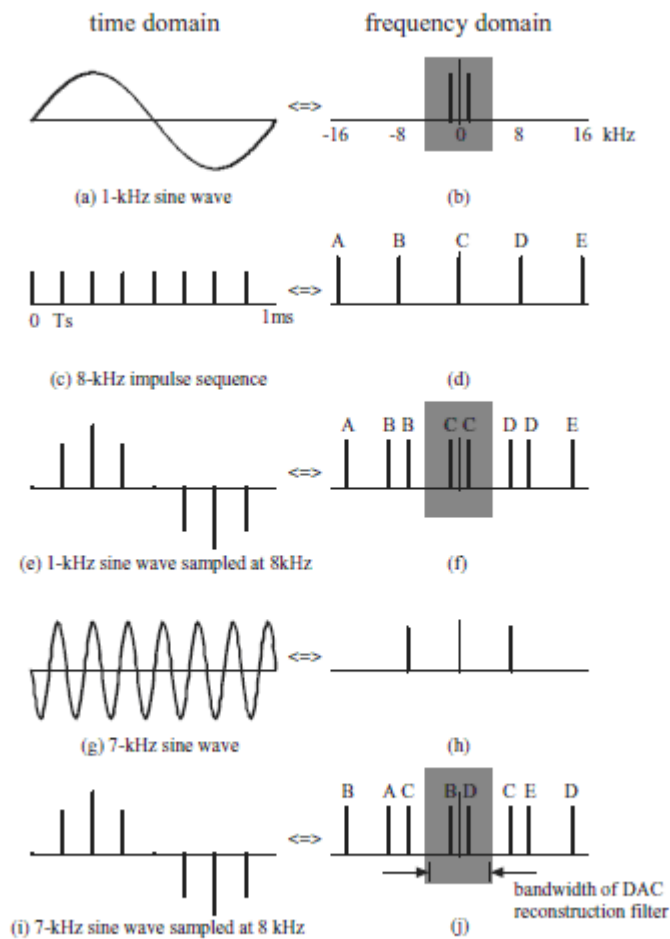
At each sampling instant, that is, within function `c_int11()`, a new output sample value is calculated using a call to the math library function `sin()`. The floating - point parameter, `theta`, passed to that function is incremented at each sampling instant by the value  $\text{theta\_increment} = 2 * \text{PI} * \text{frequency} / \text{SAMPLING\_FREQ}$  and when value of `theta` exceeds  $2\pi$  the value  $2\pi$  is subtracted from it.

Changing the value of the variable frequency in program `sine_intr.c` to an arbitrary value between 100.0 and 3500.0, you should find that a sine wave of that frequency is generated. While changing the value of the variable frequency to 7000.0, you will find that a 1 - kHz sine wave is generated. The same is true if the value of frequency is changed to 9000.0 or 15000.0. These effects are due to the phenomenon of aliasing. Sequences of samples calculated using function `sin()` at frequencies  $8000n \pm 1000$  Hz, where  $n = 0, \pm 1, \pm 2, \pm 3, \dots$  are identical and all are reconstructed by the codec as a 1 - kHz sine wave. A graphical representation of this is shown in Figure 3C.2.

In the time domain, the sampling process may be represented by multiplication of the analog input waveform  $\sin(2 * \text{pi} * 1000 * t)$  (Figure 3C.2a) by a sequence of impulses at intervals of  $T_s = 0.125$  ms (Figure 3C.2c), resulting in a sequence of weighted impulses (Figure 3C.2e).

In the frequency domain, the analog input waveform is represented by two discrete values at  $\pm 1$  kHz (Figure 3C.2b) and the sequence of time - domain impulses by a sequence of impulses in the frequency domain at intervals of  $1/T_s = 8$  kHz (Figure 3C.2d). Multiplication in the time domain is equivalent to convolution in the frequency domain. Convoluting the signals of Figures 3C.2b and 3C.2d, the frequency - domain representation of the sampled sinusoid (Figure 3C.2e) is an infinitely repeated sequence of copies of the two impulses at  $\pm 1$  kHz centered at 0 Hz,  $\pm 8$  kHz,  $\pm 16$  kHz,  $\dots$  (Figure 3C.2f). Next, consider the case of a 7 - kHz sine wave sampled at 8 kHz. Time - and frequency - domain representations of the analog input signal  $\sin(2 * \text{pi} * 7000 * t)$  are shown in Figures 3C.2g and 3C.2h.

Convoluting the signal shown in Figure 3C.2h with that shown in Figure 3C.2d results in the signal shown in Figure 3C.2j. This comprises an infinitely repeated sequence of copies of the two impulses at  $\pm 7$  kHz centered at 0 Hz,  $\pm 8$  kHz,  $\pm 16$  kHz,  $\dots$ . Despite their different derivations, Figures 3C.2f and 3C.2j are identical. This corresponds to the equivalence of the time - domain sample sequences shown in Figures 3C.2e and 3C.2i. The lowpass characteristic of the DAC can be represented by the attenuation, or blocking, of frequency components outside the range  $\pm 4$  kHz. This results, in this example, in the lowpass filtered or reconstructed versions of the signals in Figures 3C.2f and 3C.2j being identical to that shown in Figure 3C.2b.



**FIGURE 3C.2** Graphical representation of equivalence of signals generated by sampling 1-kHz and 7-kHz sine waves at 8-kHz.

Since the reconstruction (digital - to - analog conversion) process is one of lowpass filtering, it follows that the bandwidth of signals output by the codec is limited.

### LAB WORK:

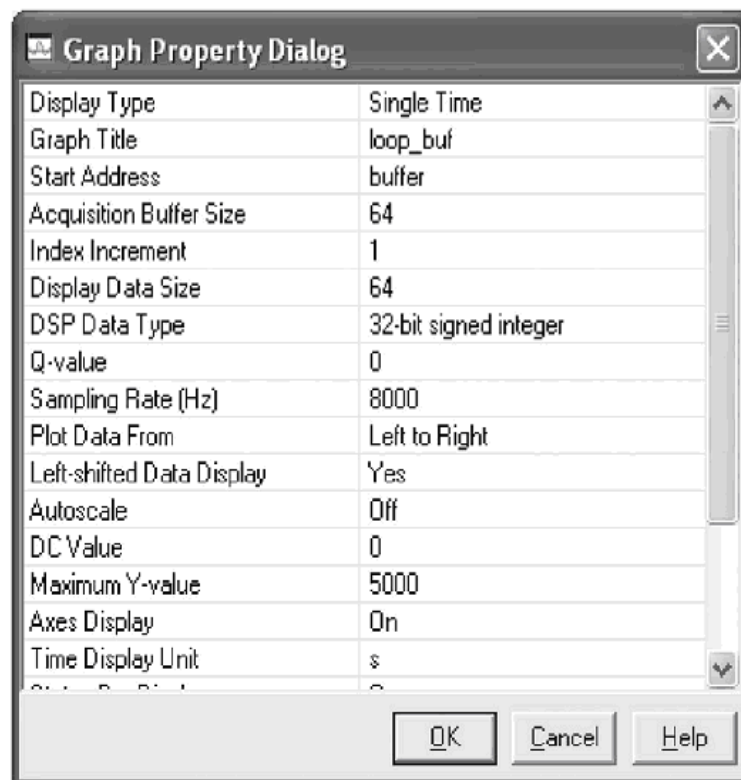
1. **Launch CCS** by double - clicking on its desktop icon.
2. Make a quick test on the DSK.
3. Connect the oscilloscope to LINE OUT on the DSK.
4. Open project *sine\_intr.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *sine\_intr.pjt* in folder *sine\_intr*.



5. Change the value of the variable frequency in program `sine_intr.c` to an arbitrary value between 100.0 and 3500.0.
6. Rebuild the project *sine\_intr.pjt*.
7. Load and run the executable file *sine\_intr.out*.
8. Record the frequency of the output sine wave.
9. Change the value of the variable frequency to 7000.0, then rebuild the project.
10. Load and run the executable file *sine\_intr.out*.
11. Record the frequency of the output sine wave.
12. Change the value of the variable frequency to 3500.0, then rebuild the project.
13. Load and run the executable file *sine\_intr.out*.
14. Record the frequency of the output sine wave.
15. Change the value of the variable frequency to 4500.0, then rebuild the project.
16. Load and run the executable file *sine\_intr.out*.
17. Record the frequency of the output sine wave.

***Step Response of the AIC23 Codec Antialiasing Filter (loop\_buf)***

18. Connect a signal generator to the DSK LINE IN socket.
19. Adjust the signal generator to give a square wave output of frequency 270 Hz and amplitude 0.2 V.
20. Load and run program `loopbuf.c` on the DSK.
21. Halting the DSP after a few seconds.
22. View the most recent 64 input sample values by selecting View→ Graph and setting the Graph Properties as shown in Figure 3C.3.



**FIGURE 3C.3** *Graph Property* settings for use with program `loop_buf.c`.

## Experiment 4

### FIR Filter I

#### Objectives:

- 1- To implement FIR filters in real time signal processing system.
- 2- To assess the magnitude frequency response of FIR filters.

#### Lab Equipments:

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphones.

#### Theory:

The moving average filter is widely used in DSP and arguably is the easiest of all digital filters to understand. It is particularly effective at removing (high frequency) random noise from a signal or at *smoothing* a signal.

The moving average filter operates by taking the arithmetic mean of a number of past input samples in order to produce each output sample. This may be represented by the equation

$$y(n) = \frac{1}{N} \sum_{i=0}^{N-1} x(n-i)$$

Where  $x(n)$  represents the  $n$ th sample of an input signal and  $y(n)$  the  $n$ th sample of the filter output. The moving average filter is an example of *convolution* using a very simple *filter kernel* or *impulse response* comprising  $N$  coefficients each of value  $1/N$ . The above equation may be thought of as a particularly simple case of the more general convolution sum implemented by a finite impulse response filter; that is,

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i)$$

where the FIR filter coefficients  $h(i)$  are samples of the filter impulse response and in the case of the moving average filter each is equal to  $1/N$ . As far as implementation is concerned, at the  $n$ th sampling instant we multiply  $N$  past input samples individually by  $1/N$  and sum the  $N$  products.

Program `average.c`, listed in Figure 4.1, uses this approach, even though it is not the most computationally efficient. The value of  $N$  defined near the start of the source file determines the number of previous input samples to be averaged.

A more rigorous method of assessing the magnitude frequency response of the filter is to use a signal generator and an oscilloscope or spectrum analyzer to measure its gain at different individual frequencies. By using this method, it is straightforward to identify the distinct notches in the magnitude frequency response at 1600 Hz (corresponding to the

tone in test file mefsin.wav that is stored in folder average.c ) and at 3200 Hz. The theoretical frequency response of the filter can be found using Matlab by running the following two lines:

```
>> [H W]=freqz([0.2 0.2 0.2 0.2 0.2],1);
>> plot(W*4000/pi,20*log10(abs(H)))
```

This frequency response is shown in Figure 4.2

```
//average.c

#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_LINE; //select input

#define N 5 //no of points averaged
float x[N]; //filter input delay line
float h[N]; //filter coefficients

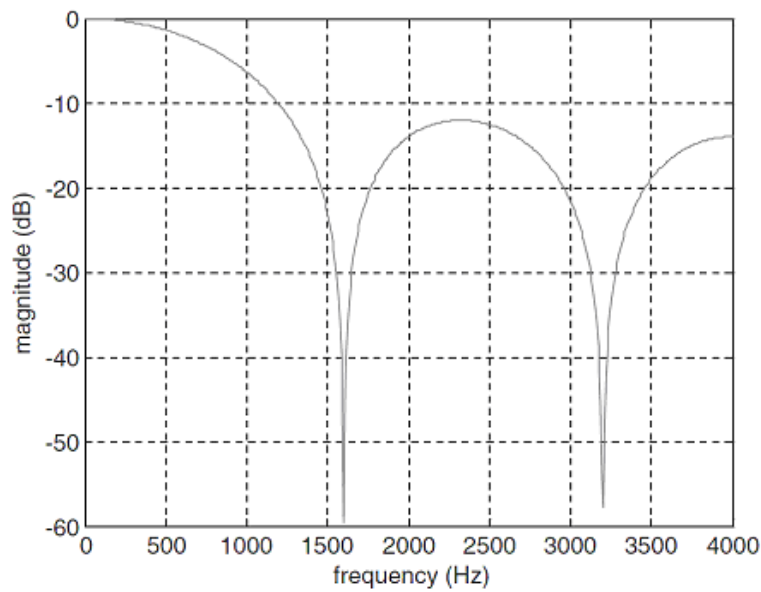
interrupt void c_int11() //interrupt service routine
{
 short i;
 float yn = 0.0;

 x[0]=(float)(input_left_sample()); //get new input sample
 for (i=0 ; i<N ; i++) //calculate filter output
 yn += h[i]*x[i];
 for (i=(N-1) ; i>0 ; i--) //shift delay line contents
 x[i] = x[i-1];
 output_left_sample((short)(yn)); //output to codec
 return;
}

void main()
{
 short i; //index variable

 for (i=0 ; i<N ; i++) //initialise coefficients
 h[i] = 1.0/N;
 comm_intr(); //initialise DSK
 while(1); //infinite loop
}
```

**FIGURE 4.1** Five point moving average filter program (average.c).



**FIGURE 4.2** Theoretical magnitude frequency response of five point moving average filter (sampling rate 8 kHz).

Another method of assessing the magnitude frequency response of a filter is to use wideband noise as an input signal. Program `averagen.c` demonstrates this technique. A pseudorandom binary sequence (PRBS) is generated within the program and used as an input to the filter in lieu of samples read from the ADC. The filtered noise can be viewed on a spectrum analyzer and whereas the frequency content of the PRBS input is uniform across all frequencies, the frequency content of the filtered noise will reflect the frequency response of the filter.

```

//averagen.c

#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_LINE; //select input

#include "noise_gen.h" //support file for noise
int fb;
shift_reg sreg;
#define NOISELEVEL 8000 //scale factor for noise
#define N 5 //no of points averaged
float x[N]; //filter input delay line
float h[N]; //filter coefficients

int prand(void) //pseudo-random noise
{
 int prnseq;
 if(sreg.bt.b0)
 prnseq = -NOISELEVEL; //scaled -ve noise level
 else
 prnseq = NOISELEVEL; //scaled +ve noise level
 fb=(sreg.bt.b0)^(sreg.bt.b1); //XOR bits 0,1
 fb^=(sreg.bt.b11)^(sreg.bt.b13); //with bits 11,13 -> fb
 sreg.regval<<=1;
 sreg.bt.b0=fb; //close feedback path
 return prnseq;
}

void resetreg(void) //reset shift register
{
 sreg.regval=0xFFFF; //initial seed value
 fb = 1; //initial feedback value
}

interrupt void c_int11() //interrupt service routine
{
 short i;
 float yn = 0.0;

 x[0] = (float)(prand()); //get new input sample
 for (i=0 ; i<N ; i++) //calculate filter output
 yn += h[i]*x[i];
 for (i=(N-1) ; i>0 ; i--) //shift delay line contents
 x[i] = x[i-1];
 output_left_sample((short)(yn)); //output to codec
 return;
}

void main()
{
 short i; //index variable
 resetreg();
 for (i=0 ; i<N ; i++) //initialise coefficients
 h[i] = 1.0/N;
 comm_intr(); //initialise DSK
 while(1); //infinite loop
}

```

**FIGURE 4.3** Five point moving average filter program with internally generated pseudo-random noise as input (averagen.c).

The frequency response of the moving average filter can be changed by altering the number of previous input samples that are averaged, or by altering the values of the coefficients.

## LAB WORK I:

1. **Launch CCS** by double - clicking on its desktop icon.
2. Make a quick test on the DSK.
3. Open project *average.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *average.pjt* in folder *average*.

4. Connect the output of a function generator to the LINE IN socket on the DSK.
5. Connect the LINE OUT socket on the DSK to an oscilloscope.
6. Load and run the executable file *average.out*.
7. Construct a table to assess the magnitude frequency response of the filter.
  - i. Set the output of the function generator to  $1 V_{p-p}$  sinusoidal waveform.
  - ii. Change the frequency of the sinusoidal signal at the output of a function from 100 Hz to 4000 Hz in steps and record the peak value of the signal on the scope.

| Frequency | Peak Value | dB value |
|-----------|------------|----------|
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |

8. A test file *mefsin.wav*, stored in folder *average*, contains a recording of speech corrupted by the addition of a sinusoidal tone. Listen to this file using Goldwave, Windows Media Player, or similar.
9. Connect the PC soundcard output to the LINE IN socket on the DSK and listen to the filtered test signal (LINE OUT or HEADPHONE).
10. Halt the executable file *average.out*.
11. Open project *averagen.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *averagen.pjt* in folder *averagen*.
12. Load and run the executable file *averagen.out*.
13. Using the FFT feature of the scope, capture the output of program *averagen.c*, compare the plot with figure 4.2.

14. Halt the executable file *averagen.out*.
15. Modify program *averagen.c* so that it implements an eleven point moving average filter; that is, change the line that reads  

```
#define N 5
```

to read  

```
#define N 11
```
16. Build and run the project and verify that the frequency response of the filter has changed using the FFT feature of the scope.
17. Build and run the project and verify that the frequency response of the filter has changed using the FFT feature of the scope.
18. Use Matlab to verify the frequency response of this new filter.
19. Halt the executable file *averagen.out*.
20. Modify program *averagen.c* again, changing the lines that read  

```
#define N 11
```

```
float h[N];
```

to read  

```
#define N 5
```

```
float h[N] = {0.0833, 0.2500, 0.3333, 0.2500, 0.0833};
```

and comment out the following line  

```
for (i=0 ; i < N ; i++) h[i] = 1.0/N;
```
21. Build and run the project and verify that the frequency response of the filter has changed using the FFT feature of the scope.  
Use Matlab to verify the frequency response of this new filter. Record your notes.

## **FIR Filter with Moving Average, Bandstop, and Bandpass Characteristics ( fir )**

### **Description**

The mechanism used by program *fir.c* (Figure 4.4) to calculate each output sample is identical to that employed by program *average.c*. Function *c\_int11()* has exactly the same definition in both programs. Whereas program *average.c* calculated the values of its coefficient in function *main()* , program *fir.c* reads the values of its coefficients from a separate file. Using this mechanism, we can implement any FIR filter with its coefficients stored in a separate file.

### **LAB WORK II:**

22. Open project *fir.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *fir.pjt* in folder *fir*.



23. Connect the output of a function generator to the LINE IN socket on the DSK.
24. Connect the LINE OUT socket on the DSK to an oscilloscope.
25. Load and run the executable file *fir.out*. Coefficient file *ave5f.cof* is listed in Figure 4.5. Using that file, program *fir.c* implements the same five point moving average filter implemented by program *average.c*. The number of filter coefficients is specified by the value of the constant *N*, defined in the *.cof* file and the coefficients are specified as the initial values in an *N* element array, *h*, of type float.
26. Run the program and verify that it implements a five point moving average filter using an assessment method similar to that in LAB WORK I above. You need only to take some selected frequencies as a test such as 1600 Hz. Record your note.
27. To implement a band-pass filter at 2700 Hz change the line that reads  
#include "ave5f.cof"  
To read  
#include "bs2700f.cof"
28. Build and run this project.
29. Input a sinusoidal signal and vary the input frequency slightly below and above 2700 Hz. Verify that the output is a minimum at 2700 Hz. The values of the coefficients for this filter were calculated using MATLAB's filter design and analysis tool, *fdatool*.
30. Edit program *fir.c* again to include the coefficient file *bp1750f.cof* in place of *bs2700f.cof*. File *bp1750f.cof* represents an FIR bandpass filter (81 coefficients) centered at 1750 Hz. Again, this filter was designed using MATLAB's *fdatool*.
31. Select Project → Build, and the new coefficient file *bp1750.cof* will automatically be included in the project. Run again and verify an FIR bandpass filter centered at 1750 Hz using an assessment method similar to that in LAB WORK I above. You need only to take some selected frequencies as a test such as 1600 Hz. Record your note.

```

//fir.c

#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_LINE; //select line in

#include "ave5f.cof" //filter coefficient file
float x[N]; //filter delay line

interrupt void c_int11() //interrupt service routine
{
 short i;
 float yn = 0.0;

 x[0]=(float)(input_left_sample()); //get new input sample
 for (i=0 ; i<N ; i++) //calculate filter output
 yn += h[i]*x[i];
 for (i=(N-1) ; i>0 ; i--) //shift delay line contents
 x[i] = x[i-1];
 output_left_sample((short)(yn)); //output to codec
 return;
}

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

```

**FIGURE 4.4** FIR filter program (fir.c).

```

// ave5f.cof
// this file was generated automatically using function dsk_fir67.m

#define N 5

float h[N] = {
 2.0000E-001,2.0000E-001,2.0000E-001,2.0000E-001,2.0000E-001
};

```

**FIGURE 4.5** Coefficient file ave5f.cof.

## Generating Filter Coefficient ( .cof) Files Using MATLAB

If the number of filter coefficients is small, a coefficient ( .cof) file can be edited by hand. For larger numbers of coefficients the MATLAB function `dsk_fir67()` , supplied on the CD accompanying the text book as file `dsk_fir67.m` , can be used. This function, listed in Figure 4.6, expects to be passed a MATLAB vector of coefficient values and prompts the user for an output filename.

For example, the coefficient file `ave5f.cof` was created by typing the following at the MATLAB command prompt:

```
>> x = [0.2, 0.2, 0.2, 0.2, 0.2];
>> dsk_fir67(x)
```

Enter filename for coefficients ave5f.cof

Note that the coefficient filename must be entered in full, including the suffix .cof.

```
% DSK_FIR67.M
% MATLAB function to write FIR filter coefficients
% in format suitable for use in C6713 DSK programs
% firnc.c and firprn.c
% written by Donald Reay
%
function dsk_fir67(coeff)
%
coefflen=length(coeff);
fname = input('enter filename for coefficients ','s');
fid = fopen(fname,'wt');
fprintf(fid,'// %s\n',fname);
fprintf(fid,'// this file was generated automatically using function
dsk_fir67.m\n',fname);
fprintf(fid,'\n#define N %d\n',coefflen);
fprintf(fid,'\nfloat h[N] = { \n');
% j is used to count coefficients written to current line
% in output file
j=0;
% i is used to count through coefficients
for i=1:coefflen
% if six coeffs have been written to current line
% then start new line
 if j>5
 j=0;
 fprintf(fid,'\n');
 end
% if this is the last coefficient then simply write
% its value to the current line
% else write coefficient value, followed by comma
 if i==coefflen
 fprintf(fid,'%2.4E',coeff(i));
 else
 fprintf(fid,'%2.4E,',coeff(i));
 j=j+1;
 end
end
fprintf(fid,'\n};\n');
fclose(fid);
```

**FIGURE 4.6** Listing of MATLAB function dsk\_fir67.m.

### LAB WORK III:

32. Open project *fir.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *fir.pjt* in folder *fir*.
33. Connect the output of the PC speaker to the LINE IN socket on the DSK.
34. Connect the HP OUT socket on the DSK to a headphone.
35. Design a bandstop filter to approximately reject 3 kHz tone from a corrupted speech signal using the Matlab filter design tool (fdatool).
36. In the filter design and analysis tool window, select *Export* from the *File* menu, a small window with *Export* title will be opened.
37. In the *Export* window, select *Workspace* for *Export to*, *Coefficients* for *Export As*.
38. In the *Variable Names*, type *Num3000*, then click *Export*. The filter coefficients are now available in the present *Workspace* and can be verified by typing *whos*.
39. Create a coefficient file by typing  
    `>> dsk_fir67(Num3000)`  
    enter filename for coefficients *Num3000f.cof*
40. To implement this filter edit file *fir.c* and change the line that reads  
    `#include "ave5f.cof"`  
    To read  
    `#include "Num3000f.cof"`
41. Build and run this project.
42. On Matlab, play the corrupted speech signal with the command  
    `>> sound(CorrAspect,8000)` or using *RealPlayer* while listening to the speech signal using the headphone. Record your notes.

## **Experiment 5**

### **FIR Filter II**

#### **Objectives:**

- 1- To implement real time signal processing system based on FIR filters.

#### **Lab Equipments:**

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphones.

#### **1- Effects on Voice or Music Using Three FIR Lowpass Filters (fir3LP)**

In this part of the experiment, three FIR lowpass filters with cutoff frequencies at 600, 1500, and 3000 Hz, respectively are implemented. The program `fir3lp.c` used in this part is listed in Figure 5.1. The three lowpass filters were designed using MATLAB. `LP_number` selects the desired lowpass filter to be implemented. For example, if `LP_number` is set to 0, `h[0][i]` is equal to `hlp600[i]` (within the for loop in function `main()`), which is the address of the first set of coefficients. The coefficient file `LP600.cof` represents an 81 - coefficient FIR lowpass filter with a 600 - Hz cutoff frequency, using the Kaiser window function. Figure 5.2 shows a listing of coefficient file `LP600.cof`. Note that the FIR filters in this experiment are implemented using fixed - point arithmetic and use 16 - bit integer type coefficients. Coefficient files `LP600.cof`, `LP1500.cof`, and `LP3000.cof` are incompatible with programs `fir.c`. The value of `LP_number` can be changed to 1 or 2 to implement the 1500 – or 3000 - Hz lowpass filter, respectively. With the GEL file `fir3lp.gel`, the value of `LP_number` can be varied while the program is running.

```

//fir3lp.c FIR using 3 lowpass coefficients with different BW

#include "lp600.cof" //coeff file LP @ 600 Hz
#include "lp1500.cof" //coeff file LP @ 1500 Hz
#include "lp3000.cof" //coeff file LP @ 3000 Hz
#include "dsk6713_aic23.h" //codec-dsk support file
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 MIC IN
short LP_number = 0; //start with 1st LP filter
int yn = 0; //initialize filter output
short dly[N]; //delay samples
short h[3][N]; //filter characteristics

interrupt void c_int11() //ISR
{
 short i;

 dly[0] = input_left_sample(); //newest input
 yn = 0; //initialize filter output
 for (i = 0; i < N; i++)
 yn += (h[LP_number][i]*dly[i]); //y(n) += h(LP#,i)*x(n-i)
 for (i = N-1; i > 0; i--) //start @ bottom of buffer
 dly[i] = dly[i-1]; //update delays
 output_left_sample(yn >> 15); //output filter
 return; //return from interrupt
}

void main()
{
 short i;

 for (i=0; i<N; i++)
 {
 dly[i] = 0; //init buffer
 h[0][i] = hlp600[i]; //start of LP600 coeffs
 h[1][i] = hlp1500[i]; //start of LP1500 coeffs
 h[2][i] = hlp3000[i]; //start of LP3000 coeffs
 }
 comm_intr(); //init DSK, codec, McBSP
 while(1); //infinite loop
}

```

**FIGURE 5.1** FIR program to implement three different lowpass filters using GEL slider for selection (fir3lp.c).

```
//LP600.cof FIR lowpass filter coefficients using Kaiser window

#define N 81 //length of filter

short hlp600[N] = {0,-6,-14,-22,-26,-24,-13,8,34,61,80,83,63,
19,-43,-113,-171,-201,-185,-117,0,146,292,398,428,355,174,-99,
-416,-712,-905,-921,-700,-218,511,1424,2425,3391,4196,4729,
4915,4729,4196,3391,2425,1424,511,-218,-700,-921,-905,-712,
-416,-99,174,355,428,398,292,146,0,-117,-185,-201,-171,-113,
-43,19,63,83,80,61,34,8,-13,-24,-26,-22,-14,-6,0};
```

**FIGURE 5.2** Coefficient file for FIR lowpass filter with 600-Hz cutoff frequency (LP600.cof).

## LAB WORK I:

1. **Launch CCS** by double - clicking on its desktop icon.
2. Make a quick test on the DSK.
3. Open project *fir3lp.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *fir3lp.pjt* in folder *fir3lp*.
4. Edit the file *fir3lp.c* and change the line that reads  
 Uint16 inputsource=DSK6713\_AIC23\_INPUT\_MIC;//select MIC IN  
 to read  
 Uint16 inputsource=DSK6713\_AIC23\_INPUT\_LINE; // select LINE IN
5. Connect the microphone to the LINE IN and headphone to the HP OUT sockets on the DSK.
6. Rebuild the project, load and run program *fir3lp.out*.
7. Select *File* → ☐ *Load GEL* and load the file *fir3lp.gel* (in folder *fir3lp*).
8. Double - click on the filename *fir3lp.gel* in the *Project View* window to view it within CCS.
9. Select *GEL* → ☐ *Filter Characteristics* → *filter*. The value of LP\_number can be varied while the program is running.
10. Vary the value of the LP\_number and note the effect of the three different lowpass filters while talking into a microphone and listening on the headphone.
11. Halt the executable file *fir3lp.out*.
12. The effect of the filters is particularly striking if applied to a voice signal input, connect the PC soundcard output to the LINE IN socket on the DSK. Run the file

lab5.wav which stored in the DSP laboratory directory and note the effect of the three different lowpass filters.

13. Use the file lab5.wav as a source connected to the LINE IN socket on the DSK. With the lower bandwidth of 600 Hz, using the first set of coefficients, the frequency components of the input signal above 600 Hz are suppressed. Connect the output to a speaker or a spectrum analyzer (use the FFT feature of the scope) to verify such results, and listen to the effect of the different bandwidths of the three FIR lowpass filters.
14. Alternatively, the effects of the filters can be illustrated using an oscilloscope and a signal generator set to input a 200 - Hz square wave to the LINE IN socket.
15. Monitor the effect of the different bandwidths of the three FIR lowpass filters on the input a 200 - Hz square wave by monitoring the output signal on the scope in the time and the frequency domain (use the FFT feature of the scope).

## **2- Two Notch Filters to Recover a Corrupted Speech Recording (notch2)**

This part of the experiments illustrates the use of two notch (bandstop) FIR filters in series to recover a speech recording corrupted by the addition of two sinusoidal signals at frequencies of 900 and 2700 Hz. Program notch2.c is listed in Figure 5.3. Two coefficient files, bs900.cof and bs2700.cof , each containing 89 coefficients and designed using MATLAB, are used by the program. They implement two FIR notch filters, centered at 900 and 2700 Hz, respectively. The output of the first notch filter, centered at 900 Hz, is used as the input to the second notch filter, centered at 2700 Hz.



```

//notch2.c Two FIR notch filters to remove sinusoidal noise

#include "DSK6713_AIC23.h" //codec-DSK support file
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_LINE; //select line in
#include "bs900.cof" //BS 900 Hz coefficient file
#include "bs2700.cof" //BS 2700 Hz coefficient file
short dly1[N]={0}; //delay for 1st filter
short dly2[N]={0}; //delay for 2nd filter
int y1out = 0, y2out = 0; //init output of each filter
short out_type = 2; //slider for output type

interrupt void c_int11() //ISR
{
 short i;

 dly1[0] = input_left_sample(); //new input @ top of buffer
 y1out = 0; //init output of 1st filter
 y2out = 0; //init output of 2nd filter
 for (i = 0; i < N; i++)
 y1out += h900[i]*dly1[i]; //y1(n)+=h900(i)*x(n-i)
 dly2[0]=y1out>>15;
 for (i = 0; i < N; i++)
 y2out += h2700[i]*dly2[i]; //y2(n)+=h2700(i)*x(n-i)
 for (i = N-1; i > 0; i--) //from bottom of buffer
 {
 dly1[i] = dly1[i-1]; //update samples in buffers
 dly2[i] = dly2[i-1];
 }
 if (out_type==1) //if slider is in position 1
 output_left_sample((short) (y2out>>15)); //output of 1st filter
 if (out_type==2)
 output_left_sample((short) (y1out>>15)); //output of 2nd filter
 return; //return from ISR
}

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

```

**FIGURE 5.3** . Program implementing two FIR notch filters in cascade to remove two undesired sinusoidal signals (notch2.c).

## LAB WORK II:

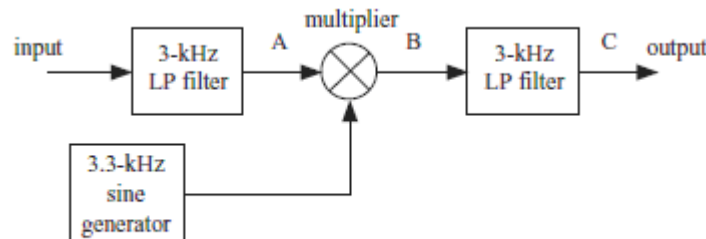
16. Open project *notch2.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *notch2.pjt* in folder *notch2*.

17. Connect the PC soundcard output to the LINE IN socket and the scope to the LINE OUT sockets on the DSK.
18. Load and run program *notch2.out*.
19. The file *corrupt.wav*, stored in folder *notch2*, contains a recording of speech corrupted by the addition of 900 - and 2700 - Hz sinusoidal tones. Use the FFT feature of the scope to see the 900 and 2700Hz before and after the filter.
20. A GEL slider ( *notch2.gel* ) can be used to select either the output of the two cascaded notch filters (default) or the output of the first notch filter. Make these different selections and record your notes.
21. Compare the results of this example with those obtained using a moving average filter.

### 3- Voice Scrambling Using Filtering and Modulation ( scrambler )

This part of the experiment illustrates a voice scrambling/descrambling scheme. The approach makes use of basic algorithms for filtering and modulation. With voice as input, the resulting output is scrambled voice. The original descrambled voice is recovered when the output of the DSK is used as the input to a second DSK running the same program. The scrambling method used is commonly referred to as frequency inversion. It takes an audio range, in this case 300 Hz to 3 kHz, and “ folds ” it about a 3.3 – kHz carrier signal. The frequency inversion is achieved by multiplying (modulating) the audio input by a carrier signal, causing a shift in the frequency spectrum with upper and lower sidebands. In the lower sideband that represents the audible speech range, the low tones are high tones, and vice versa.

Figure 5.4 is a block diagram of the scrambling scheme. At point A we have an input signal, band limited to 3 kHz. At point B we have a double - sideband signal with suppressed carrier. At point C the upper sideband and the section of the lower sideband between 3 and 3.3 kHz are filtered out. The scheme is attractive because of its simplicity. Only simple DSP algorithms — namely, filtering, sine wave generation, and amplitude modulation — are required for its implementation.



**FIGURE 5.4** Block diagram of scrambler system.

Figure 5.5 shows a listing of program `scrambler.c`, which operates at a sampling rate,  $f_s$ , of 16 kHz. The input signal is first lowpass filtered using an FIR filter with 65 coefficients.h, defined in the file `lp3k64.cof`. The filtering algorithm used is identical to that used in, for example, program `fir.c`. The filter delay line is implemented using array `x1` and the output is assigned to variable `yn1`. The filter output (at point A in Figure 5.4 ) is multiplied (modulated) by a 3.3 - kHz sinusoid stored as 160 samples (exactly 33 cycles) in array `sine160` (file `sine160.h` ) . Finally, the modulated signal (at point B) is lowpass filtered again, using the same set of filter coefficients.h (`lp3k64.cof`) but a different filter delay line implemented using array `x2` and the output variable `yn2` . The output is a scrambled signal (at point C).

```
//scrambler.c

#include "DSK6713_AIC23.h" // codec support
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE; //select line in
#include "sine160.h"
#include "lp3k64.cof" //filter coefficient file
float yn1, yn2; //filter outputs
float x1[N],x2[N]; //filter delay lines
int index = 0;

interrupt void c_int11()
{
 short i;

 // first filter input
 x1[0]=(float)(input_left_sample()); //get input into delay line
 yn1 = 0.0; //initialise filter output
 for (i=0 ; i<N ; i++) yn1 += h[i]*x1[i];
 for (i=(N-1) ; i>0 ; i--) x1[i] = x1[i-1];
 // next mix with 3300Hz
 yn1 *= sine160[index++];
 if (index >= NSINE) index = 0;

 // now filter again
 x2[0] = yn1; //get input into delay line
 yn2 = 0.0; //initialise filter output
 for (i=0 ; i<N ; i++) yn2 += h[i]*x2[i];
 for (i=(N-1) ; i>0 ; i--) x2[i] = x2[i-1];
 output_left_sample((short)(yn2)); //output to codec
 return;
}

void main()
{
 comm_intr(); //initialise McBSP, AD535
 while(1); //infinite loop
}
```

**FIGURE 5.5** Scrambler program `scrambler.c`.

Using this scrambled signal as the input to a second DSK running the same algorithm, the original descrambled input is recovered as the output of the second DSK.

### LAB WORK III:

22. Open project *scrambler.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *scrambler.pjt* in folder *scrambler*.
23. Connect the output of a function generator to the LINE IN socket on the DSK.
24. Connect the LINE OUT socket on the DSK to an oscilloscope.
25. Set the function generator output to 2 – kHz sine wave.
26. Edit the file *c6713dskinit.h* included in the project changing the line that reads  
0x0017 / \* Set -Up Reg 0 Left line volume control \*/  
to read  
0x001B / \* Set -Up Reg 0 Left line volume control \*/
27. Rebuild the project then load and run program *scrambler.out*. The resulting output is a lower sideband signal at 1.3 kHz. The upper sideband signal at 5.3 kHz is filtered out by the second lowpass filter.
28. Vary the frequency of the sinusoidal input in the range 300 – 3000 Hz and verify that output frequencies appear as in the inverted range 3000 to 300 Hz. A second DSK running the same program can be used to recover the original signal (simulating the receiving end). Use the output of the first DSK as the input to the second DSK.
29. Change the input source used by the program on each DSK from LINE IN to MIC. By editing the file *scrambler.c* and change the line that reads  
Uin16 inputsource=DSK6713\_AIC23\_INPUT\_LINE;//select LINE IN  
to read  
Uin16 inputsource=DSK6713\_AIC23\_INPUT\_MIC; // select MIC
30. Test the scrambler and descrambler using speech from a microphone as the input. Run exactly the same program on each DSK, and connect HEADPHONE on the first DSK (scrambler) to MIC IN on the second DSK (descrambler).

## **Experiment 6**

### **IIR Filter I**

#### **Objectives:**

- 1- To design an IIR filter using the impulse invariance transformation method.
- 2- To assess the magnitude frequency response of IIR filter.

#### **Lab Equipments:**

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphones.

#### **1- Design of a Simple IIR Low pass Filter Using Impulse Invariance Transformation Method**

Traditionally, IIR filter design is based on the concept of transforming a continuous-time, or analog, design into the discrete - time domain. Butterworth, Chebyshev, Bessel, and elliptical classes of analog filter are widely used. In this part of the experiment, we will design a second order, type 1 Chebyshev, lowpass continuous-time filter with 2 dB of passband ripple and a cutoff frequency of 1500 Hz (9425 rad/s). Then, the designed continuous-time filter is transformed into digital filter using the impulse invariance method. In impulse invariance method, the impulse response of the digital filter is the samples of the impulse response of the continuous-time filter (mathematically:  $h[n] = Th(nT)$ ), where T represents the sampling interval.

Program *iirsos.c*, stored in folder *iirsos* and listed in Figure 6.1, implements ageneric IIR filter using cascaded direct form II second order stages (sections) and coefficient values stored in a separate file. The program uses the following two expressions:

$$w(n) = x(n) - a_1w(n-1) - a_2w(n-2)$$
$$y(n) = b_0w(n) + b_1w(n-1) + b_2y(n-2)$$

Implemented by the lines

```
wn = input - a[section][0]*w[section][0] - [section][1]
 *w[section][1];
yn = b[section][0]*wn + b[section][1]*w[section][0] +
 a[section][2]*w[section][1];
```

```

//iirsos.c iir filter using cascaded second order sections

#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_LINE;
#include "impinv.cof"

float w[NUM_SECTIONS][2] = {0};

interrupt void c_int11() //interrupt service routine
{
 int section; //index for section number
 float input; //input to each section
 float wn,yn; //intermediate and output
 //values in each stage

 input = ((float)input_left_sample());

 for (section=0 ; section< NUM_SECTIONS ; section++)
 {
 wn = input - a[section][0]*w[section][0]
 - a[section][1]*w[section][1];
 yn = b[section][0]*wn + b[section][1]*w[section][0]
 + b[section][2]*w[section][1];
 w[section][1] = w[section][0];
 w[section][0] = wn;
 input = yn; //output of current section
 //will be input to next
 }
 output_left_sample((short)(yn)); //before writing to codec
 return; //return from ISR
}

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

```

**FIGURE 6.1** IIR filter program using second order stages in cascade (iirsos.c).

## LAB WORK I:

1. Design the continuous-time filter using the Matlab command:

```
>> [b,a] = cheby1(2,2 * pi * 1500, ' s ');
```

The continuous - time transfer function of such a filter is given by:

$$H(s) = \frac{58072962}{s^2 + 7576s + 73109527}$$

2. Draw the frequency response of this filters using the Matlab command:

```
>> freqs(b,a)
```

3. Transfer the designed continuous-time filter into digital filter using the Matlab command:

```
>> [bz,az] =impinvar(b,a,8000);. Here a sampling frequency of 8000 Hz is assumed.
```

The filter coefficients bz and az are used to create the coefficient filter file *impinv.cof* (listed in Figure 6.2) which is then used in the main program *iirsos.c*.

```
// impinv.cof
// second order type 1 Chebyshev LPF with 2dB passband ripple
// and cutoff frequency 1500Hz

#define NUM_SECTIONS 1

float b[NUM_SECTIONS][3]={ {0.0, 0.48255, 0.0} };
float a[NUM_SECTIONS][2]={ {-0.71624, 0.387913} };
```

**FIGURE 6.2** Listing of coefficient file *impinv.cof*.

4. Use Matlab to assess the magnitude frequency response of the digital filter by typing the command:

```
>> freqz(bz,az).
```

5. Compare the gain of the analog prototype filter (step 2) with that of the transformed digital filter.

6. **Launch CCS** by double - clicking on its desktop icon.

7. Make a quick test on the DSK.

8. Open file *iirsos.c* by selecting *File* → ☐ *Open* and double – clicking on file *iirsos.c* in folder *iirsos*.

9. Connect the output of a function generator to the LINE IN socket on the DSK.

10. Connect the LINE OUT socket on the DSK to an oscilloscope.

11. Load and run the executable file *iirsos.out*.

12. Construct a table to assess the magnitude frequency response of the filter.

- i. Set the output of the function generator to sinusoidal.
- ii. Change the frequency of the sinusoidal signal at the output of a function from 100 Hz to 4000 Hz in steps and record the peak value of the signal on the scope.
- iii. Make sure to include frequency 3000 Hz in your table.

| Frequency | Peak Value | dB value |
|-----------|------------|----------|
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |

You will find that the attenuation of frequencies above 2500 Hz is not very pronounced. That is due to the low order of the filter and to inherent shortcomings of the impulse invariant transformation method.

13. In your report, explain the difference between the designed digital filter and the analog prototype filter.
14. Alternatively, we can estimate the frequency response of the digital filter using Pseudorandom Noise as input. In real time, it generates a pseudorandom binary sequence and uses this wideband noise signal as the input to an IIR filter. The output of the filter is written to the DAC in the AIC23 codec and the resulting analog signal (filtered noise) can be analyzed using an oscilloscope or spectrum analyzer. Halt the executable file *iirsos.out*.
15. Open file *iirsosprn.c* by selecting *File* → ☐ *Open* and double – clicking on file *iirsosprn.c* in folder *iirsosprn*.
16. Load and run the executable file *iirsosprn.out*.
17. Using the FFT feature of the scope, capture the output of program *iirsosprn.out*, compare the plot with your results in steps 4 and 12.

## 2- Estimating the Frequency Response of an IIR Filter Using a Sequence of Impulses as Input (*iirsosdelta*)

Instead of a pseudorandom binary sequence, program *iirsosdelta.c* generates a sequence of discrete - time impulses as the input to an IIR filter. The resultant output is an approximation to a repetitive sequence of filter impulse responses. This relies on the filter impulse response decaying practically to zero within the period between successive input impulses. The filter output is written to the DAC in the AIC23 codec and the resulting analog signal can be analyzed using an oscilloscope, spectrum analyzer, Goldwave , or other instrument. In addition, program *iirsosdelta.c* stores BUFSIZE samples of the filter output in buffer response and we can use the View→ Graph facility in Code Composer to view that data in both time and frequency domains.



```

// iirsosdelta.c iir filter using cascaded second order sections
// input internally generated delta sequence, output to line out
// and save in buffer
// float coefficients read from included .cof file

#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_LINE;
#define BUFSIZE 256
#define AMPLITUDE 20000
#include "impinv.cof"

float w[NUM_SECTIONS][2] = {0};

float dimpulse[BUFSIZE];
float response[BUFSIZE];
int index = 0;

float w[NUM_SECTIONS][2] = {0};

interrupt void c_int11() //interrupt service routine
{
 int section; //index for section number
 float input; //input to each section
 float wn,yn; //intermediate and output
 //values in each stage

 input = dimpulse[index]; //input to first section is
 //read from impulse sequence

 for (section=0 ; section< NUM_SECTIONS ; section++)
 {
 wn = input - a[section][0]*w[section][0]
 - a[section][1]*w[section][1];
 yn = b[section][0]*wn + b[section][1]*w[section][0]
 + b[section][2]*w[section][1];
 w[section][1] = w[section][0];
 w[section][0] = wn;
 input = yn; //output of current section
 //will be input to next
 }
 output_left_sample((short)(yn)); //before writing to codec
 return; //return from ISR
}

void main()
{
 int i;

 for (i=0 ; i< BUFSIZE ; i++) dimpulse[i] = 0.0;
 dimpulse[0] = 1.0;
 comm_intr(); //init DSK, codec, McBSP
 while(1); //infinite loop
}

```

**FIGURE 6.3** IIR filter program using second order stages in cascade and internally gener-

## LAB WORK II:

18. Open project *iirsosdelta.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *iirsosdelta.pjt* in folder *iirsosdelta*.
19. Connect the LINE OUT socket on the DSK to an oscilloscope.
20. Load and run the executable file *iirsosdelta.out*.
21. Monitor the output (impulse response of the filter) on the scope and make your notes.  
The output waveform is shaped both by the IIR filter and by the AIC23 codec reconstruction filter.
22. Using the FFT feature of the scope, capture the magnitude of the frequency response of the filter.  
In the frequency domain, the codec reconstruction filter is responsible for the steep roll – off of gain at frequencies above 3500 Hz and the ac coupling of the codec output is responsible for the steep roll - off of gain at frequencies below 100 Hz.
23. Halt the program and select *View* → *Graph*. Set the Graph Properties as indicated in Figure 6.4 and you should see something similar to the right - hand graph shown in Figure 6.5. You need to set the Graph Properties differently to see something similar to the left - hand graph shown in Figure 6.5.

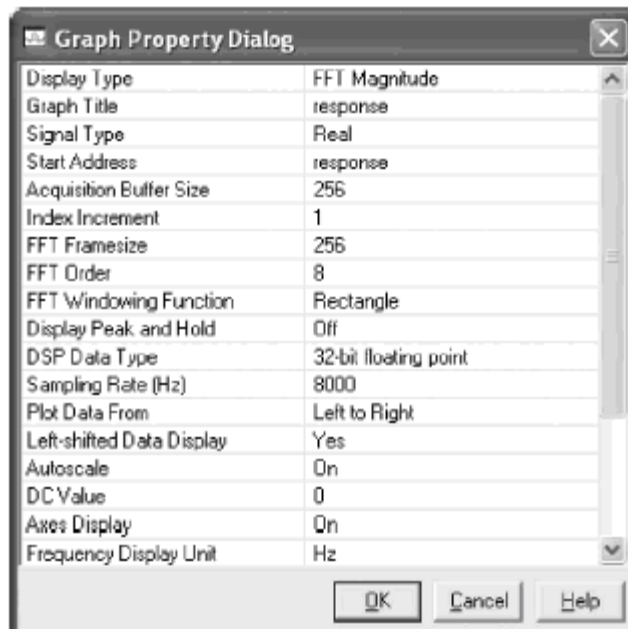
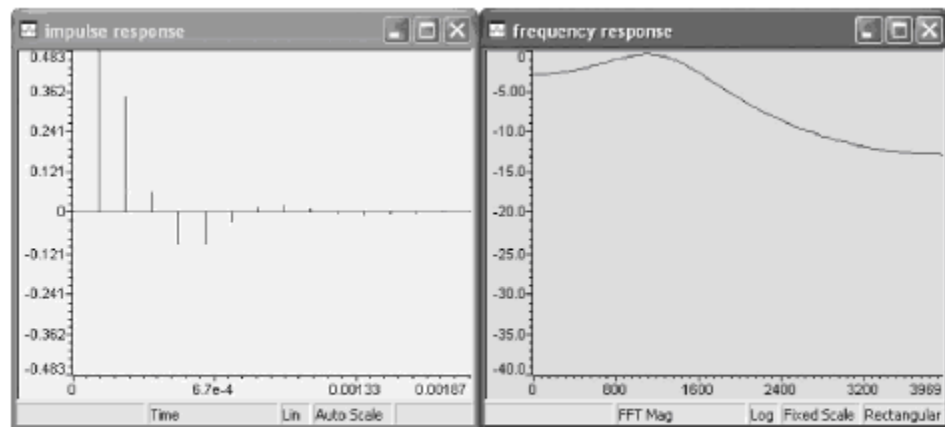


FIGURE 6.4 *Graph Property* settings for use with program *iirsosdelta.c*.



**FIGURE 6.5** Impulse and magnitude frequency response of example filter captured using Code Composer and program `iirsosdelta.c`.

## Experiment 7

### IIR Filter II

#### Objectives:

- 1- To design an IIR filter using the bilinear transformation method.
- 2- To assess the magnitude frequency response of IIR filter.

#### Lab Equipments:

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphones.

### 1- Bilinear Transform Method of Digital Filter Implementation.

The bilinear transform method of converting an analog filter design to discrete time is relatively straightforward, often involving less algebraic manipulation than the impulse invariant method. It is achieved by making the substitution

$$s = \frac{2(z-1)}{T(z+1)}$$

In  $H(s)$ , where  $T$  is the sampling period of the digital filter; that is,

$$H(z) = H(s) \Big|_{s=2(z-1)/T(z+1)}$$

The concept behind the bilinear transform is that of compressing the frequency response of an analog filter's design such that its response over the entire range of frequencies from zero to infinity is mapped into the frequency range zero to half the sampling frequency of the digital filter. This may be represented by

$$f_D = \frac{\arctan(\pi f_A T_s)}{\pi T_s} \quad \text{or} \quad \omega_D = \frac{2}{T_s} \arctan\left(\frac{\omega_A T_s}{2}\right)$$

And

$$f_A = \frac{\tan(\pi f_D T_s)}{\pi T_s} \quad \text{or} \quad \omega_A = \frac{2}{T_s} \tan\left(\frac{\omega_D T_s}{2}\right)$$

As a result of the frequency warping inherent in the bilinear transform, the cutoff frequency of the discrete - time filter obtained is not equal to the cutoff frequency of the analog filter. A technique called prewarping the prototype analog design (used by default in the MATLAB filter design and analysis tool *fdtool*) can be used in such a way that the bilinear transform maps an analog frequency  $\omega_A = \omega_c$ , in the range  $0$  to  $\omega_s/2$ , to exactly the same digital frequency  $\omega_D = \omega_c$ . This technique is based on the selection of  $T$  according to  $T = 2 \tan(\pi \omega_c / \omega_s) / \omega_c$ .

In this part of the experiment, we will design a second order, type 1 Chebyshev, lowpass continuous-time filter with 2 dB of passband ripple and a cutoff frequency of 1500 Hz (9425 rad/s). Then, the designed continuous-time filter is transformed into digital filter using the bilinear transformation method. We will also use the same programs in experiment 10 (*iirsos.c*, *iirsoprns.c* and *iirsosdelta.c*)

## LAB WORK I:

1. Design the continuous-time filter using the Matlab command:  
`>> [b,a] = cheby1(2,2,2 * pi * 1500, ' s ' );`
2. Draw the frequency response of this filters using the Matlab command:  
`>> freqs(b,a)`
3. Transfer the designed continuous-time filter into digital filter using the Matlab command:  
`>> [bd,ad] = bilinear(b,a,8000);`. Here a sampling frequency of 8000 Hz is assumed. The filter coefficients bz and az are used to create the coefficient filter file *bilinear.cof* which is then used in the main program *iirsos.c*.
4. Use Matlab to assess the magnitude frequency response of the digital filter by typing the command:  
`>> freqz(bz,az).`  
In your report, explain the difference between the designed digital filter and the analog prototype filter.
5. **Launch CCS** by double - clicking on its desktop icon.
6. Make a quick test on the DSK.
7. Connect the output of a function generator to the LINE IN socket on the DSK.
8. Connect the LINE OUT socket on the DSK to an oscilloscope.
9. Open file *iirsos.c* by selecting *File* → ☐ *Open* and double – clicking on file *iirsos.c* in folder *iirsos*.
10. Change the line that reads `#include "impinv.cof"` to read `#include "bilinear.cof"`.
11. Build the project.
12. Load and run the executable file *iirsos.out*.
13. Construct a table to assess the magnitude frequency response of the filter.
  - i. Set the output of the function generator to sinusoidal.
  - ii. Change the frequency of the sinusoidal signal at the output of a function from 100 Hz to 4000 Hz in steps and record the peak value of the signal on the scope.

| Frequency | Peak Value | dB value |
|-----------|------------|----------|
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |
|           |            |          |

You will find that the cutoff frequency of the discrete - time filter obtained is not 1500 Hz but 1356 Hz. In addition, you will find that the gain of the analog filter at a frequency of 4500 Hz is equal to the gain of the digital filter at a frequency of 2428 Hz and that the digital frequency 1500 Hz corresponds to an analog frequency of 1702 Hz.

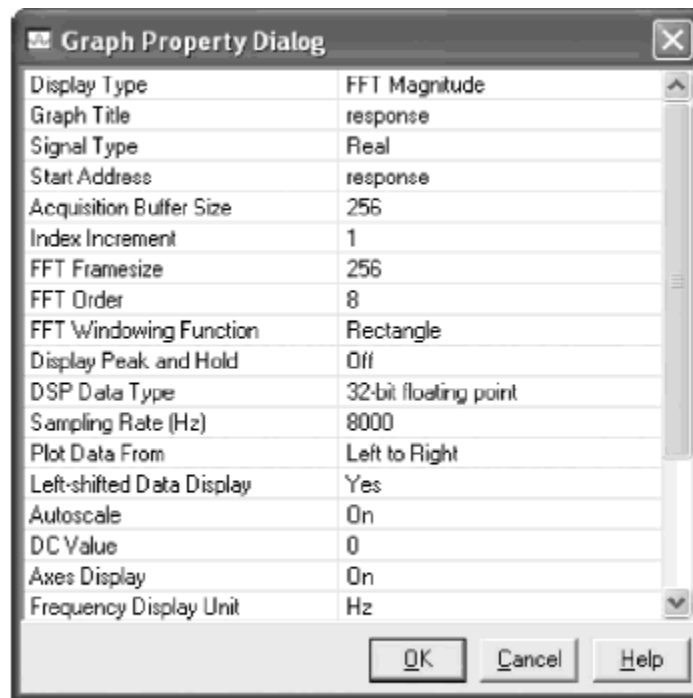
14. Alternatively, we can estimate the frequency response of the digital filter using Pseudorandom Noise as input. Halt the executable file *iirsos.out*.
15. Open file *iirsosprn.c* by selecting *File* → ☐ *Open* and double – clicking on file *iirsoprns.c* in folder *iirsosprn*.
16. Change the line that reads `#include "impinv.cof"` to read `#include "bilinear.cof"`.
17. Build the project.
18. Load and run the executable file *iirsosprn.out*.
19. Using the FFT feature of the scope, capture the output of program *iirsosprn.out*, compare the plot with your results in steps 4 and 13.

## 2- Estimating the Frequency Response of an IIR Filter Using a Sequence of Impulses as Input ( *iirsosdelta*).

### LAB WORK II:

20. **Launch CCS** by double - clicking on its desktop icon.
21. Connect the LINE OUT socket on the DSK to an oscilloscope.
22. Open project *iirsosdelta.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *iirsosdelta.pjt* in folder *iirsosdelta*.
23. Edit file *iirsosdelta.c* and change the line that reads `#include "impinv.cof"` to read `#include "bilinear.cof"`.
24. Build the project.

25. Load and run the executable file *iirsosdelta.out*.
26. Monitor the output (impulse response of the filter) on the scope and make your notes.  
The output waveform is shaped both by the IIR filter and by the AIC23 codec reconstruction filter.
27. Using the FFT feature of the scope, capture the magnitude of the frequency response of the filter.
28. Halt the program and select View→ Graph. Set the Graph Properties as indicated in Figure 7.1, save the obtained graph and used in your report.
29. You need to set the Graph Properties differently to see the impulse response of the filter in the time domain.



**FIGURE 7.1** Graph Property settings for use with program *iirsosdelta.c*.

### 3- Design of IIR Filters Using MATLAB 's Filter Design and Analysis Tool

MATLAB provides a filter design and analysis tool, *fdatool*, that makes the design of IIR filter coefficients simple. Coefficients can be exported in direct form II, second order section format and a MATLAB function *dsk\_sos\_iir67()* written by the author of the text book can be used to generate coefficient files compatible with the programs (*iirsos.c*, *iirsoprns.c* and *iirsosdelta.c*).

### LAB WORK III:

30. Start FDATool from the MATLAB command line by typing:  
    >>fdatool.
31. In the **Response Type** pane, select **Lowpass**.
32. In the Design Method pane, select IIR, and then select Butterworth
33. For the Filter Order, select **Specify order**, and then enter 6.
34. Set frequency specifications to Fs to 8000 and Fc to 1500.
35. Click the Design Filter button, you will shortly be able to see the magnitude response of the designed filter, make your not and save it to use it in your report.
36. Click on Export in the fdatool File menu.
37. Select Workspace, Coefficients, SOS , and G and click Export .
38. At the MATLAB command line, type `dsk_sos_iir67(SOS,G)` and enter a filename Butterworth.cof .
39. **Launch CCS** by double - clicking on its desktop icon.
40. Connect the LINE OUT socket on the DSK to an oscilloscope.
41. Open project *iirsosdelta.pjt* by selecting *Project* → ☐ *Open* and double – clicking on file *iirsosdelta.pjt* in folder *iirsosdelta*.
42. Edit file *iirsosdelta.c* and change the line that reads `#include “ impinv.cof ”` to read `#include “Butterworth.cof ”`.
43. Build the project.
44. Load and run the executable file *iirsosdelta.out*.
45. Monitor the output (impulse response of the filter) on the scope and make your notes.
46. Using the FFT feature of the scope, capture the magnitude of the frequency response of the filter.
47. Halt the program and select View→ Graph. Set the Graph Properties as indicated in Figure 7.1, save the obtained graph and compare it with that in step 35.
48. You need to set the Graph Properties differently to see the impulse response of the filter in the time domain.



## Experiment 8

### Discrete Time and Fast Fourier Transforms

#### Objectives:

- 1- To compute the Discrete Fourier Transform (DTF) of real signal.
- 2- To compute the Fast Fourier Transform (FFT) of real signal.
- 3- To estimate the execution time for the DFT and FFT functions

#### Lab Equipments:

- 1- DSK Board.
- 2- CCS software installed on the computer.
- 3- Oscilloscope.
- 4- Headphones.

#### 1- DFT of a Sequence of Real Numbers with Output in the CCS Graphical Display Window ( dft )

This part of the experiment illustrates the DFT of an N - point, real - valued sequence. Program dft.c, listed in Figure 8.1, calculates the complex DFT:

$$x(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N} \quad k = 0, 1, \dots, N-1$$

Using Euler ' s relation to represent a complex exponential

$$e^{-j\omega t} = \cos(\omega t) - j \sin(\omega t)$$

The real and imaginary parts of X (k) are computed by the program:

$$\text{Re}\{X(k)\} = \sum_{n=0}^{N-1} (\text{Re}\{x(n)\}\cos(2\pi kn/N) + \text{Im}\{x(n)\}\sin(2\pi kn/N))$$

$$\text{Im}\{X(k)\} = \sum_{n=0}^{N-1} (\text{Im}\{x(n)\}\cos(2\pi kn/N) - \text{Re}\{x(n)\}\sin(2\pi kn/N))$$

A structured data type COMPLEX is used by the program to represent the complex valued time - and frequency - domain values of X(k) and x(n).

The function dft() has been written such that it replaces the input samples x(n), stored in array samples with their frequency - domain representation X(k).

The time - domain sequence x(n) consists of exactly 10 cycles of a real - valued cosine wave (assuming a sampling frequency of 8 kHz, the frequency of the cosine wave is 800 Hz). The DFT of this sequence, X ( k ), is equal to zero for all k , except at k = 10 and at k = 90. These two real values correspond to frequency components at  $\pm 800$  Hz. Different time - domain input sequences can be used in the program, most readily by changing the value of the constant TESTFREQ .

```

//dft.c N-point DFT of sequence read from lookup table
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979
#define N 100
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0

typedef struct
{
 float real;
 float imag;
} COMPLEX;

COMPLEX samples[N];

void dft(COMPLEX *x)
{
 COMPLEX result[N];
 int k,n;

 for (k=0 ; k<N ; k++)
 {
 result[k].real=0.0;
 result[k].imag = 0.0;

 for (n=0 ; n<N ; n++)
 {
 result[k].real += x[n].real*cos(2*PI*k*n/N) +
x[n].imag*sin(2*PI*k*n/N);
 result[k].imag += x[n].imag*cos(2*PI*k*n/N) -
x[n].real*sin(2*PI*k*n/N);
 }
 }
 for (k=0 ; k<N ; k++)
 {
 x[k] = result[k];
 }
}


void main()
{
 int n;

 for(n=0 ; n<N ; n++)
 {
 samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
 samples[n].imag = 0.0;
 }
 printf("real input data stored in array samples[]\n");
 printf("\n"); // place breakpoint here
 dft(samples); //call DFT function
 printf("done!\n");
}

```

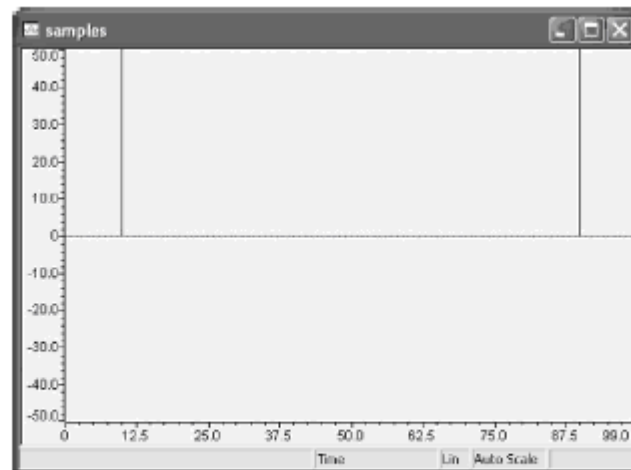
**FIGURE 8.1** Listing of program dft.c.

## LAB WORK I:

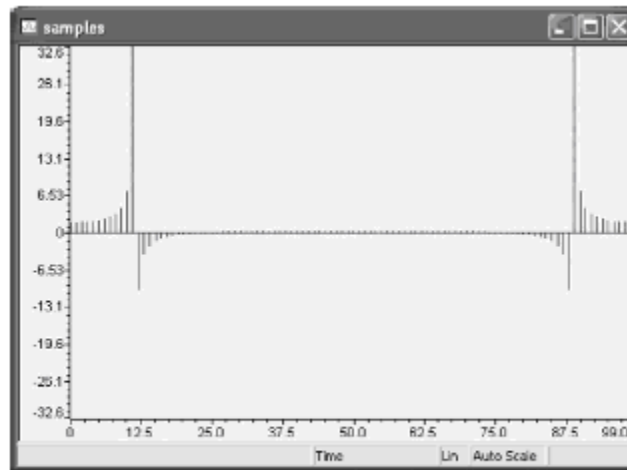
1. **Launch CCS** by double - clicking on its desktop icon.
2. Open project *dft.pjt* by selecting *Project* →  *Open* and double – clicking on file *dft.pjt* in folder *dft*.
3. Load the executable file *dft.out*.
4. Place a breakpoint at the line  
`printf("\n"); // place breakpoint here`  
by clicking on that line in the source file *dft.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.
5. Select *Debug*→ *Run*. The program should halt at the breakpoint just before calling function *dft()* and at this point the initial, time - domain contents of array samples will be displayed in the *Graphical Display* window.
6. Select *View*→ *Graph* → *Time/Frequency* and set the *Graph Properties* as shown in Figure 8.2. Note that this will display only the real part of the complex values stored in array samples. The *Graph Property Data Plot Style* is set to *Bar* in order to emphasize that the DFT operates on discrete data.
7. Select *Debug*→ *Run* again. The program should run to completion at which point the contents of array samples will be equal to the frequency – domain representation  $X(k)$  of the input data  $x(n)$ . The real part of  $X(k)$  will now be displayed in the *Graphical Display* window and you should be able to see two distinct spikes at  $k = 10$  and  $k = 90$ , representing frequency components at  $\pm 800$  Hz, as shown in Figure 8.3.
8. Change the frequency of the input waveform to 900 Hz (`#define TESTFREQ 900.0`) and repeat the procedure listed above. You should see a number of nonzero values in the frequency - domain sequence  $X(k)$ , as shown in Figure 8.4 . This effect is referred to as spectral leakage and is due to the fact that the  $N$  sample time - domain sequence stored in array samples does not now contain an integer number of cycles of a sinusoid. Correspondingly, the frequency of that sinusoid is not exactly equal to one of the  $N$  discrete frequency components, spaced at intervals of  $(8000.0/N)$  Hz in the frequency - domain representation  $X(k)$ .



**FIGURE 8.2** *Graph Properties* used to display real part of array `samples` in program `dft.c`.



**FIGURE 8.3** *Graphical Display* of real part of array `samples` produced by program `dft.c` (`TESTFREQ = 800`).



**FIGURE 8.4** Graphical Display of real part of array samples produced by program `dft.c` (TESTFREQ = 900).

## 2- DFT of a Sequence of Real Numbers Using Twiddle Factors with Output in the CCS Graphical Display Window ( `dftw` )

Whereas the radix - 2 FFT is applicable if  $N$  is an integer power of 2, the DFT can be applied to an arbitrary length sequence (e.g.,  $N = 100$ ), as illustrated by program `dft.c`. However, the FFT is widely used because of its computational efficiency. Part of that efficiency is due to the use of precalculated twiddle factors, stored in a lookup table, rather than the repeated evaluation of  $\sin()$  and  $\cos()$  functions during computation of the FFT. The use of precalculated twiddle factors can be applied to the function `dft()` to give significant efficiency improvements to program `dft.c`. Calls to the math library functions  $\sin()$  and  $\cos()$  are computationally very expensive and are made a total of  $4N^2$  times in function `dft()`. In program `dftw.c`, listed in Figure 8.5, these function calls are replaced by reading precalculated twiddle factors from array `twiddle`.

The source file `dftw.c` is stored in folder `dft` and can be substituted for source file `dft.c` in project `dft`. Verify that program `dftw.c` gives similar results. (Change the Output Filename to `dftw.out`.)

### LAB WORK II:

9. Open project `dft.pjt` by selecting *Project*  $\rightarrow$  ☐ *Open* and double – clicking on file `dft.pjt` in folder `dft`.
10. Remove file `dft.c` from the project by double – clicking on *Source* from the project view, then right click on the file `dft.c` and select *remove file from project*.
11. Add file `dftw.c` to the project by right clicking on *Source* and select *Add files to project* and select the file `dftw.c`.

```

//dftw.c N-point DFT of sequence read from lookup table
//using pre-computed twiddle factors

#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979
#define N 100
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0

typedef struct
{
 float real;
 float imag;
} COMPLEX;

COMPLEX samples[N];
COMPLEX twiddle[N];

void dftw(COMPLEX *x, COMPLEX *w)
{
 COMPLEX result[N];
 int k,n;

 for (k=0 ; k<N ; k++)
 {
 result[k].real=0.0;
 result[k].imag = 0.0;

 for (n=0 ; n<N ; n++)
 {
 result[k].real += x[n].real*w[(n*k)%N].real -
x[n].imag*w[(n*k)%N].imag;
 result[k].imag += x[n].imag*w[(n*k)%N].real +
x[n].real*w[(n*k)%N].imag;
 }
 for (k=0 ; k<N ; k++)
 {
 x[k] = result[k];
 }
 }

void main()
{
 int n;

 for(n=0 ; n<N ; n++)
 {
 twiddle[n].real = cos(2*PI*n/N);
 twiddle[n].imag = -sin(2*PI*n/N);
 }
 for(n=0 ; n<N ; n++)
 {
 samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
 samples[n].imag = 0.0;
 }
 printf("real input data stored in array samples[]\n");
 printf("\n"); // place breakpoint here
 dftw(samples,twiddle); //call DFT function
 printf("done!\n");
}

```

**FIGURE 8.5** Listing of program dftw.c.

12. Select Project→ Build Options. In the Compiler tab in the Basic category set the Opt Level to Function( o2) and in the Linker tab set the Output Filename to .\Debug\dftw.out .

13. Build the project.
14. Load the executable file *dftw.out*.
15. Place a breakpoint at the line  

```
printf("\n"); // place breakpoint here
```

 by clicking on that line in the source file *dftw.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.
16. Select Debug→ Run. The program should halt at the breakpoint just before calling function *dftw()* and at this point the initial, time - domain contents of array samples will be displayed in the Graphical Display window.
17. Select View→ Graph → Time/Frequency and set the Graph Properties as shown in Figure 8.2. Note that this will display only the real part of the complex values stored in array samples. The Graph Property Data Plot Style is set to Bar in order to emphasize that the DFT operates on discrete data.
18. Select Debug→ Run again. The program should run to completion at which point the contents of array samples will be equal to the frequency – domain representation  $X(k)$  of the input data  $x(n)$ . The real part of  $X(k)$  will now be displayed in the Graphical Display window and you should be able to see two distinct spikes at  $k = 10$  and  $k = 90$ , representing frequency components at  $\pm 800$  Hz, as shown in Figure 8.3.
19. Change the frequency of the input waveform to 900 Hz (`#define TESTFREQ 900.0`) and repeat the procedure listed above. You should see a number of nonzero values in the frequency - domain sequence  $X(k)$ , as shown in Figure 8.4 . This effect is referred to as spectral leakage and is due to the fact that the  $N$  sample time - domain sequence stored in array samples does not now contain an integer number of cycles of a sinusoid. Correspondingly, the frequency of that sinusoid is not exactly equal to one of the  $N$  discrete frequency components, spaced at intervals of  $(8000.0/N)$  Hz in the frequency - domain representation  $X(k)$ .

### 3- Estimating Execution Times for DFT and FFT Functions (fft)

The computational expense of function *dft()* can be illustrated using Code Composer's Profile Clock. In this part of the experiment, the functions *dft()* and *dftw()* used in Lab Work I and II are compared with a third function, *fft()* , which implements the FFT in C.

### LAB WORK III:

20. Edit the lines in programs *dft.c* and *dftw.c* that read  

```
#define N 100
```

 to read  

```
#define N 128
```

21. Ensure that source file `dft.c` and not `dftw.c` is present in the project.
22. Select Project→ Build Options. In the Compiler tab in the Basic category set the Opt Level to Function- (o2) and in the Linker tab set the Output Filename to `.\Debug\dft.out`.
23. Build the project and load `dft.out`.
24. Open source file `dft.c` by double - clicking on its name in the Project View window and set breakpoints at the lines  
`dft(samples);`  
and  
`printf("done!\n");` .
25. Select Profile → Clock → Enable.
26. 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.
27. Run the program. It should halt at the first breakpoint.
28. Reset the Profile Clock by double - clicking on its icon in the bottom right – hand corner of the CCS window.
29. Run the program. It should stop at the second breakpoint. The number of instruction cycles counted by the Profile Clock gives an indication of the computational expense of executing function `dft()`.
30. Repeat the preceding experiment (step 22 to step 27) substituting file `dftw.c` for file `dft.c`.
31. Finally, repeat the preceding experiment (step 22 to step 27) using file `fft.c` (also stored in folder `dft`) (see Figure 8.6). This program computes the FFT using a function written in C and defined in the file `fft.h` (Figure 8.7). The advantage, in terms of execution time, of the FFT over the DFT should increase with the number of points,  $N$ , used. Repeat this example using different values of  $N$  (e.g., 256 or 512).



|        | N=128                        |                | N=256                        |                | N=512                        |                |
|--------|------------------------------|----------------|------------------------------|----------------|------------------------------|----------------|
|        | Number of Instruction Cycles | Execution Time | Number of Instruction Cycles | Execution Time | Number of Instruction Cycles | Execution Time |
| dft.c  |                              |                |                              |                |                              |                |
| dftw.c |                              |                |                              |                |                              |                |
| fft.c  |                              |                |                              |                |                              |                |

```

//fft.c N-point FFT of sequence read from lookup table

#include <stdio.h>
#include <math.h>
#include "fft.h"

#define PI 3.14159265358979
#define N 128
#define TESTFREQ 800.0
#define SAMPLING_FREQ 8000.0

COMPLEX samples[N];
COMPLEX twiddle[N];

void main()
{
 int n;
 for (n=0 ; n<N ; n++) //set up DFT twiddle factors
 {
 twiddle[n].real = cos(PI*n/N);
 twiddle[n].imag = -sin(PI*n/N);
 }

 for(n=0 ; n<N ; n++)
 {
 samples[n].real = cos(2*PI*TESTFREQ*n/SAMPLING_FREQ);
 samples[n].imag = 0.0;
 }
 printf("real input data stored in array samples[]\n");
 printf("\n"); // place breakpoint here
 fft(samples,N,twiddle); //call DFT function
 printf("done!\n");
}

```

**FIGURE 8.6** Listing of program `fft.c`.

```

//fft.h complex FFT function taken from Rulph's C31 book
//this file contains definition of complex dat structure also

struct cmpx //complex data structure used by FFT
{
 float real;
 float imag;
};
typedef struct cmpx COMPLEX;

void fft(COMPLEX *Y, int M, COMPLEX *w)
{
 COMPLEX temp1,temp2; //temporary storage variables
 int i,j,k; //loop counter variables
 int upper_leg, lower_leg; //index of upper/lower butterfly leg
 int leg_diff; //difference between upper/lower leg
 int num_stages=0; //number of FFT stages, or iterations
 int index, step; //index and step between twiddle factor
 i=1; //log(2) of # of points = # of stages
 do
 {
 num_stages+=1;
 i=i*2;
 } while (i!=M);

 leg_diff=M/2; //difference between upper & lower legs
 step=2; //step between values in twiddle.h
 for (i=0;i<num_stages;i++)
 {
 index=0;
 for (j=0;j<leg_diff;j++)
 {
 for (upper_leg=j;upper_leg<M;upper_leg+=(2*leg_diff))
 {
 lower_leg=upper_leg+leg_diff;
 temp1.real=(Y[upper_leg]).real + (Y[lower_leg]).real;
 temp1.imag=(Y[upper_leg]).imag + (Y[lower_leg]).imag;
 temp2.real=(Y[upper_leg]).real - (Y[lower_leg]).real;
 temp2.imag=(Y[upper_leg]).imag - (Y[lower_leg]).imag;
 (Y[lower_leg]).real=temp2.real*(w[index]).real
 -temp2.imag*(w[index]).imag;
 (Y[lower_leg]).imag=temp2.real*(w[index]).imag
 +temp2.imag*(w[index]).real;
 (Y[upper_leg]).real=temp1.real;
 (Y[upper_leg]).imag=temp1.imag;
 }
 index+=step;
 }
 leg_diff=leg_diff/2;
 }
}

```

**FIGURE 8.7** Listing of header file `fft.h`.