AN-NAJAH NATIONAL UNIVERSITY

FACULTY OF ENGINEERING

TELECOMMUNICATION ENGINEERING DEPARTMENT

Digital Signal Processing LAB

10646443

For

Telecommunication Engineering Students

Reviewed and prepared by:

Dr. Falah Mohammed

Inst. Nuha Odeh

Eng. Monir Aghbar

2018/2019

Department Name: Telecommunication Engineering **Course Name: Digital signal processing Lab** **Number: 10646443** **Report Grading Sheet**	
Instructor Name:	Experiment #:
Academic Year: 2018/2019	Performed on:
Semester:	Submitted on:
Experiment Name:	
Students:	
1-	2-
3-	4-
5-	6-

Report's Outcomes

ILO _1_ =(40) %	ILO _3_ =(37) %	ILO _4_ =(23) %	ILO __ =( ) %	ILO __ =( ) %

Evaluation Criterion	Grade	Points
Introduction Sufficient, Clear and complete statement of objectives.		
Theory Presents sufficiently the theoretical basis.	1	
Apparatus/ Procedure Apparatus sufficiently described to enable another experimenter to identify the equipment needed to conduct the experiment. Procedure sufficiently described.	1	
Experimental Results and Calculations Results analyzed correctly. Experimental findings adequately and specifically summarized, in graphical, tabular, and/or written form.	3	
Discussion Crisp explanation of experimental results. Comparison of theoretical predictions to experimental results, including discussion of accuracy and error analysis in some cases.	2	
Conclusions and Recommendations Conclusions summarize the major findings from the experimental results with adequate specificity. Recommendations appropriate in light of conclusions. Correct grammar.	1	
Appendices Appropriate information, organized and annotated. Includes all calculations and raw data Sheet.		
AppearanceTitle page is complete, page numbers applied, content is well organized, correct spelling, fonts are consistent, good visual appeal.	2	
Total	10	

Department of Telecomunication Engineering	
Digital signal processing Lab (10646443)	
Total Credits	**1**
major compulsory	
Prerequisites	P1 : Digital signal processing (10646441)

Course Contents

The DSP Lab is equipped with complete set of hardware and software to perform DSP experiments in this course. Real -Time DSP is used to understand the real-time DSP systems principles and Real-world applications. It also includes sampling and waveform generation, quantization, digital modulation schemes (ASK, PSK, FSK), error correcting codes, FIR Filter implementation (Low Pass, High Pass Band Stop), IIR Filter implementation, Adaptive filtering applications, Audio effects (multi echo generation, fading), Dual tone multi frequency generation (DTMF) and Image processing principles.

	Intended Learning Outcomes (ILO's)	Student Outcomes (SO's)	Contribution
1	An ability to apply knowledge of digital communications and digital signal processing theory in practice.	B	25 %
2	An ability to use DSP processor in the design of communication systems, speech processing, digital filtering (FIR, IIR), error correction codes and DTMF.	C	30 %
3	An ability to function on multidisciplinary teams	D	15 %
4	An Ability to use Matlab software (Simulink and DSP toolbox) to design real system using DSP processor and investigate image processing principles (including arithmetic operation and filtering).	K	30 %

Textbook and/ or Refrences

1) DSP laboratory manual.
2) Digital Signal Processing and Applications with the C6713 and C6416 DSK, Rulph chassaing, John Wiley, 2005.

Assessment Criteria	Percent (%)
Laboratory Work	70 %
Final Exam	30 %

Course Plan

Week	Topic
1	Introduction to DSP lab (hardware and software tools).
2	Waveform generation and sampling.
3	ASK and FSK modulation and demodulation.

4	QPSK modulation and demodulation.
5	FIR filtering.
6	IIR filtering.
7	Adaptive filtering (Noise cancellation and System identification).
8	Audio Effects (Echo, Multi Echo, Reverb, Fading generation).
9	Hamming Codes.
10	Digital Image Processing using MATLAB -1
11	Digital Image Processing using MATLAB -2
12	Echo cancelling and voice scrambling.
13	Dual-Tone Multi frequency DTMF.
14	Final exam.

Table of Contents

ELECTRICAL SAFETY GUIDELINES

1) Be familiar with the electrical hazards associated with your workplace.

2) You may enter the laboratory only when authorized to do so and only during authorized hours of operation.

3) Be as careful for the safety of others as for yourself. Think before you act, be tidy and systematic.

4) Avoid bulky, loose or trailing clothes. Avoid long loose hair.

5) Food, beverages and other substances are strictly prohibited in the laboratory at all times. Avoid working with wet hands and clothing.

6) Use extension cords only when necessary and only on a temporary basis.

7) Request new outlets if your work requires equipment in an area without an outlet.

8) Discard damaged cords, cords that become hot, or cords with exposed wiring.

9) Before equipment is energized ensure, (1) circuit connections and layout have been checked by a laboratory technician and (2) all colleagues in your group give their assent.

10) Know the correct handling, storage and disposal procedures for batteries, cells, capacitors, inductors and other high energy-storage devices.

11) Experiments left unattended should be isolated from the power supplies. If for a special reason, it must be left on, a barrier and a warning notice are required.

12) Equipment found to be faulty in any way should be reported to the laboratory technician immediately and taken out of service until inspected and declared safe.

13) Voltages above 50 V rms AC and 120 V DC are always dangerous. Extra precautions should be considered as voltage levels are increased.

14) Never make any changes to circuits or mechanical layout without first isolating the circuit by switching off and removing connections to power supplies.

15) Know what you must do in an emergency, i.e. Emergency Power Off

Electrical Emergency Response

The following instructions provide guidelines for handling two types of electrical emergencies:

1. *Electric Shock*:

When someone suffers serious electrical shock, he or she may be knocked unconscious. If the victim is still in contact with the electrical current, immediately turn off the electrical power source. If you cannot disconnect the power source, depress the Emergency Power Off switch.

IMPORTANT:

Do not touch a victim that is still in contact with a live power source; you could be electrocuted.

Have someone call for emergency medical assistance immediately. Administer first-aid, as appropriate.

2. *Electrical Fire:*

If an electrical fire occurs, try to disconnect the electrical power source, if possible. If the fire is small and you are not in immediate danger; and you have been properly trained in fighting fires, use the correct type of fire extinguisher to extinguish the fire. When in doubt, push in the Emergency Power Off button.

NEVERuse water to extinguish an electrical fire.

Laboratory Guidelines (Laboratory procedures)

Every week before lab, each student should read over the laboratory experiment and work out the various calculations, etc. that are outlined in the prelab. The student should refer to Digital signal processing and Applications with C6713 and C6416 DSK, by Rulph Chassaing.

a) Return parts and all equipment have to correct locations when you are finished with them.
b) Give suspected defective parts to the Lab technician for testing or disposal.
c) Report all equipment problems to Lab Instructor or Lab technician.
d) Most experiments have several parts; students must alternate in doing these parts as they are expected to work in group.
e) Laboratory and equipment maintenance is the responsibility of not only the Lab technician, but also the students. A concerted effort to keep the equipment in excellent condition and the working environment well-organized will result in a productive and safe laboratory.
f) Each student must have a laboratory notebook. The notebook should be a permanent document that is maintained and witnessed properly, and that contains accurate records of all lab sessions.

Laboratory Notebook

The laboratory notebook is a record of all work pertaining to the experiment. This record should be sufficiently complete so that you or anyone else of similar technical background can duplicate the experiment and data by simply following your laboratory notebook. Record everything directly into the note book during the experiment. Do not use scratch paper for recording data. Do not trust your memory to fill in the details later time.

GUIDELINES FOR LABORATORY NOTEBOOK

- State the objective of the experiment.
- Draw the block diagrams or any related signal processing block set and mention the values of resistances etc. which are used.
- Make a note of all the measuring instruments you have used.
- Mention the formulas used.
- Create a table and write down the readings, including the units.
- Show all your calculation neatly and SYSTEMATICALLY. Do this is an organized manner.
- Attach graph if any.
- Be concise. Complete sentences are not necessary as long as the context is clear.
- If mistakes are made, they should not be erased. Just bracket the handmake a short note explaining the problem.
- Make entries as the lab progresses; don't assume you can fill it in later. The instructor will ask to see it during the lab.
- Date every page.
- All-important results must be underlined.
- Attach simulation and hand calculation to your note book.
- Draw the figure using pencil before you come to the lab so that you can make corrections to it in case you need to do so by erasing and redrawing. This will ensure tidy and neat work.
- Prepare the READING TABLE using pencil and ruler and not just by sketching lines. Sketching gives rise to crooked lines and gives the lab notebook a haphazard look.
- Take a few short notes(2-3lines), which explains some of the problems you encountered while doing the experiment. This will help you write better reports.

General Lab Report Format

Following the completion of each laboratory exercise in Electrical Engineering courses, are port must be written and submitted for grading. The purpose of the report is to completely document the activities of the design and demonstration in the laboratory. Reports should be complete in the sense that all information required to reproduce the experiment is contained within. Writing useful reports is a very essential part of becoming an engineer. In both academic and industrial environments, reports are the primary means of communication between engineers.

There is no one best format for all technical reports but there are a few simple rules concerning technical presentations which should be followed. Adapted to this laboratory they may be summarized in the following recommended report format:

- Title page
- Introduction
- Experimental procedure
- Experimental data
- Discussion
- Conclusions

Detailed descriptions of these items are given below.

- Title Page:

  The title page should be prepared according to the ABET form included at the beginning of this lab manual. The title page should contain the following informations

  - Your name
  - ID
  - Course number (including section)
  - Experiment number and title
  - Date submitted
  - Instructors Name
- Introduction:

  It should contain a brief statement in which you state the objectives, or goals of the experiment. It should also help guide the reader through the report by stating, for example, that experiments were done with different DSP algorithms or consisted of two parts etc. or that additional calculations or datasheet scan be found in the appendix, or at the end of the report.

- The Procedure

  It describes the experimental setup and how the measurements were made. Include here block diagrams or flow charts. Mention instruments used and describe any special measurement procedure that was used.

- Results/Questions:

  This section of the report should be used to answer any questions presented in the lab handout. Any tables and/or circuit diagrams representing results of the experiment should be referred to and discussed/explained with detail. All questions should be answered very clearly in paragraph form. Any unanswered questions from the lab handout will result in loss of marks on the report.

  The best form of presentation of some of the data is graphical. In engineering presentations, a figure is often worth more than a thousand words. There are some simple rules concerning graphs and figures which should always be followed. If there is more than one figure in the report, the figures should be numbered. Each figure must have a caption following the number. For example, *"Figure 1.1: TTL Inverter"* In addition, it will greatly help you to learn how to use headers and figures in MS Word.

- The Discussion

  It is a critical part of the report which testifies to the student's understanding of the experiments and its purpose. In this part of the report you should compare the expected outcome of the experiment, such as derived from theory or computer simulation, with the measured value. Before you can make such comparison, you may have to do some data analysis or manipulation.

  When comparing experimental data with numbers obtained from theory or simulation, make very clear which is which. It does not necessarily mean that your experiment was a failure. The results will be accepted, provided that you can account for the discrepancy. Your ability to read the scales maybe one limitation. The value of some circuit components may not be well known and a nominal value given by the manufacturer does not always correspond to reality. Very often, however, the reason for the difference between the expected and measured values lies in the experimental procedure or in not taking into account all factors that enter into analysis.

- Conclusion:

  A brief conclusion summarizing the work done, theory applied, and the results of the completed work should be included here. Data and analyses are not appropriate for the conclusion.

Notes

Typed Reports are required. Any drawings done by hand must be done with neatness, using a straight edge and drawing guides wherever possible. Free hand drawings will not be accepted.

Prelab results should be reported in the provided sheets at the end of the manual. It is your responsibility to obtain the instructor's signature and to include the signed sheet with your final experiment report.

1 Experiment 1 Introduction to DSP Lab

Objectives

The objectives of this experiment are

a) Brief students with the operation of the DSK6713 kit
b) Starting a simple project to familiarize students with the code composer environment

Introduction

A signal is a physical quantity that is usually a function of time, position, pressure, etc. For example, the voltage output from a microphone represents sound pressure as a function of time. Signals that we encounter frequently in our daily life include speech, music, data, images, video signals. The objective of signal processing is to transmit or store signals, to enhance desired signal components, and to extract useful information carried by the signals.

Signal Processing is a method of extracting information from the signal which in turn depends on the type of signal and the nature of information it carries.

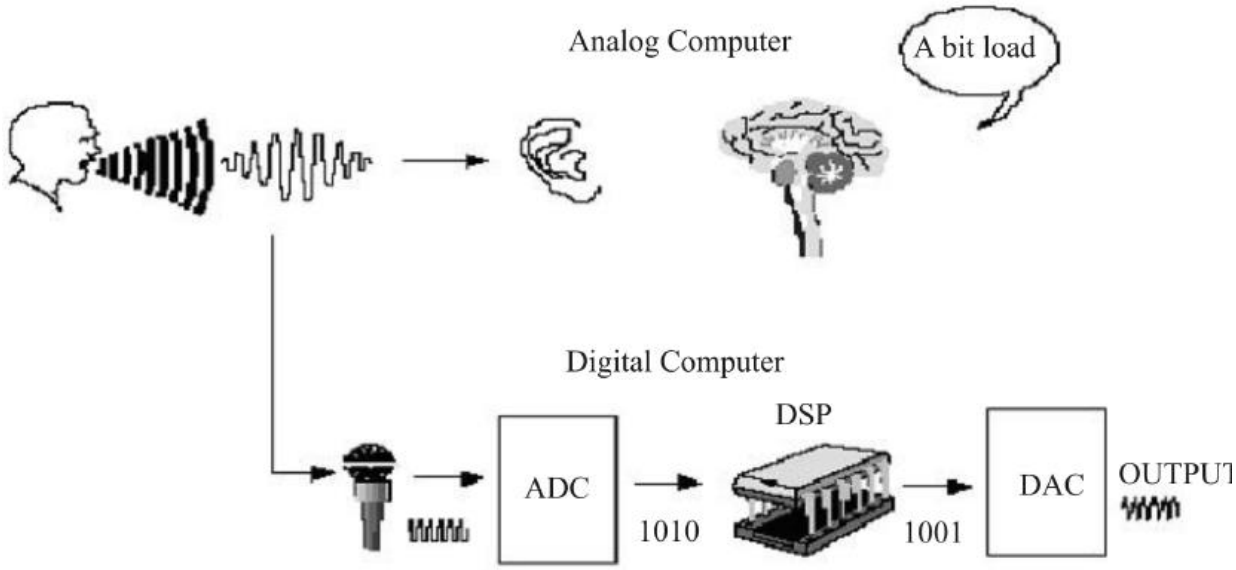


Figure 1Block Diagram of Signal Processing

Digital signal processing is concerned with the digital representation of signals and use of digital processors to analyze, modify or extract information from signals. Most signals in nature are analog. Analog signals are varying with time, and represent the variations of physical quantities such as sound waves. The signals used in most popular forms of DSP are derived from analog signals which have been sampled at regular intervals and converted into digital form. The specific

mean for processing a digital signal may be, for example, to remove interference of noise from the signal, to obtain the spectrum of the data or to transform the signal from the signalin to more suitable form. DSP is now used in many areas where analog methods were previously used and in entirely new applications which were difficult with analog methods. Basic building block of digital signal processing is shown in Figure 1

Digital Signal Processor (DSP) is a microcontroller designed specifically for signal processing applications. This is achieved as specified in Figure 2. Commonly used operations in signal processing applications are convolution, filtering, and frequency to time domain conversions. These operations need recursive multiplication and additions. In other words, they need multiply and accumulate (MAC) operations. Standard microprocessors execute the multiplication operation as a recursive addition operation. This means for a standard microprocessor, the MAC operation is processed by excessive number of addition operations. This takes time. However, DSPs contain special MAC units that can execute the same operation in a single machine cycle. For example, a 150 MIPS DSP can process approximately 32 million data samples per second. For a standard 150 MIPS microprocessor, this reduces to 2 million data samples per second. Like microcontrollers, DSPs are equipped with different peripheral devices according to their usage area. TMS320C6713 does not contain any ADC or DAC but it contains SPI and I2C interfaces. Therefore, its development kit, ST6000 [TMS320C6713 DSK], contains an AIC23 codec chip working as an ADC and DAC interface. It communicates with the DSP over the SPI.

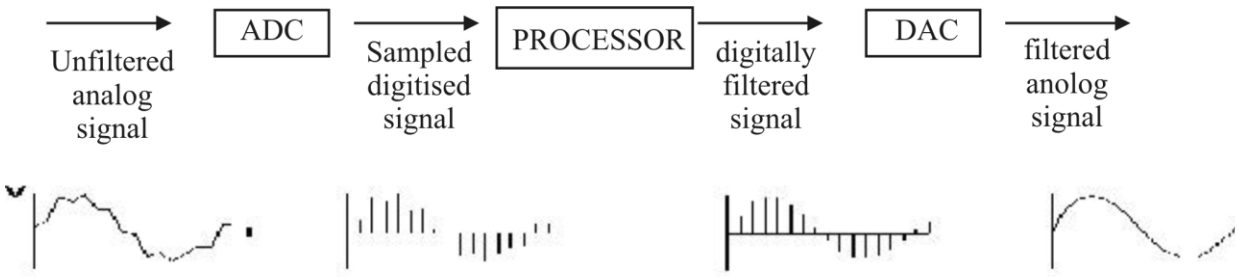


Figure 2Block Diagram of Real-Time DSP System

Mostly sensors generate analog signals in response to various phenomena. Signal processing can be carried out either in analog or digital domain. To do processing of analog signals in digital domain, first digital signal is obtained by sampling and followed by quantization (digitization). The digitization can be obtained by analog to digital converter (ADC). The role of digital signal processor (DSP) is the manipulation of digital signals so as to extract desired information. In order to interface DSP with analog world, digital to analog converters (DAC) are used. Figure 3shows the basic components of a DSP system.

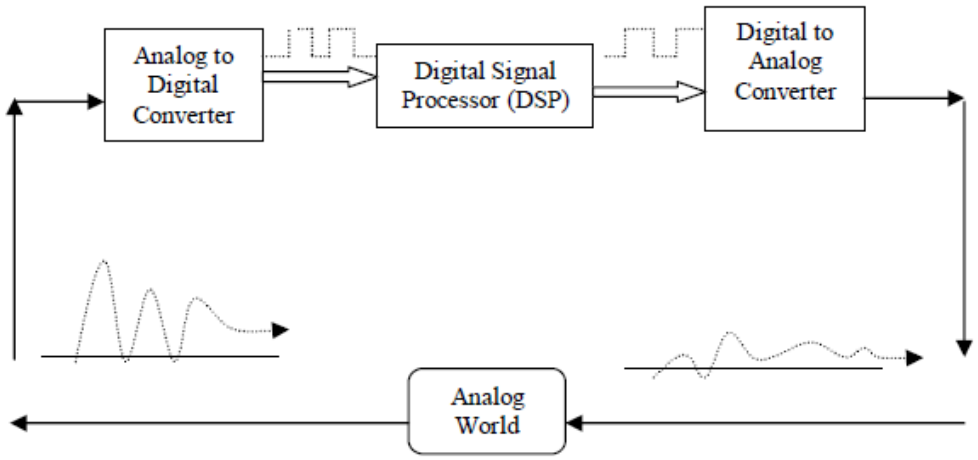


Figure 3Main components of a DSP system

ADC captures and inputs the signal. The resulting digital representation of the input signal is processed by DSP such as C6x and then output through DAC. Within in the basic DSP system, anti-aliasing filter at input to remove erroneous signals and output filter to smooth the processed data is also used.

Advantages of Digital Signal Processing

The advantages of DSP system are

- Guaranteed Accuracy: Accuracy is only determined by the number of bits used.
- Perfect Reproducibility: Identical performance from unit to unit is obtained since there is no variation due to component tolerances. For example, using DSP technique, a digital recording can be copied to or reproduced several times over without any degradation in the signal quality.
- Greater Flexibility: DSP system can be programmed and reprogrammed to perform a variety of a function, without modifying the hardware.
- Superior Performance: DSP can be used to perform functions not possible with analog signal processing. For example, linear phase response can be achieved. And complex adaptive filtering algorithms can be implemented using DSP technique.

1.1.1 What is Real-Time Processing?

Consider a software system in which the inputs represent digital data from hardware such as imaging devices or other software system's and the output share digital data that control external hardware such as displays. The time between the presentation of a set of inputs and the appearance of all the associated outputs is called the response time. A real-time system is one

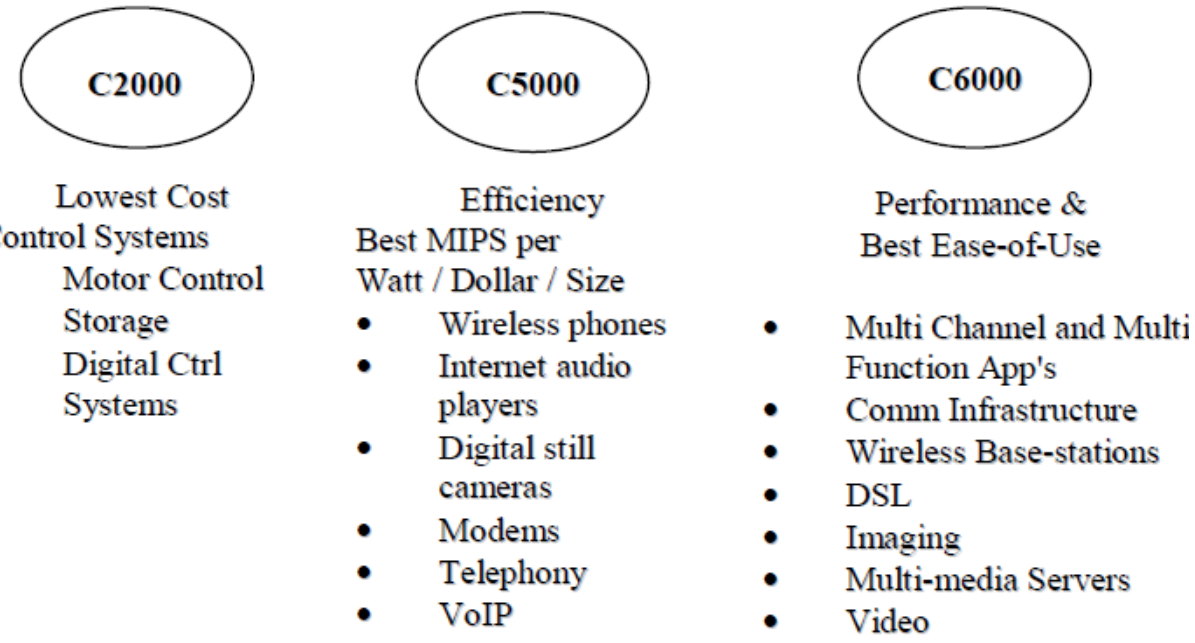
that must satisfy explicit bounded response time constraints to avoid failure. Equivalently, a real-time system is one whose logical correctness is based both on the correctness of the outputs and their timeliness. Notice that response time of, for example, microseconds are not needed to characterize a real-time system. It simply must have response times that are constrained and thus predictable. In fact, the misconception that real-time systems must be "fast" is because in most instances, the deadlines are on the order of microseconds. But the timelines constraints or deadlines are generally a reflection of the underlying physical process being controlled. For example, in image processing involving screen update for viewing continuous motion, the deadlines are on the order of 30 microseconds. In practical situations, the main difference between real-time and non-real-time systems is an emphasis on response time prediction and its reduction. Upon reflection, one realizes that every system can be made to conform to the real-time definition simply are setting deadlines (arbitrary or otherwise). For example, a one-time image filtration algorithm for medical imaging, which might not be regarded as real-time, really is real-time if the procedure is related to an illness in which diagnosis and treatment have some realistic

Digital Signal Processing Systems and Applications

DSP systems are often embedded in larger systems to perform specialized DSP operations, thus allowing the overall systems to handle general purpose tasks. For example, a DSP processor is a modem used for data transmission in the embedded DSP system of a computer. Often this type of a DSP system runs only one application and is not programmed by the end user.

1.1.2 The TMS320 Family:

Texas instrument introduced the first DSP processor, the TMS32010is the first DSP processor of TMS320 family. Today it consists of fixed point and floating point processors. The 16 bit fixed point processor includes the TMS320C2000 (C2x and C28x), C5000 (the C54x and C55x) and C6000 (the C62x and C64x) generations. The 32 bit floating point processors consist of C3x, C4x and C67xgenerations.



Texas Instruments' TMS320 Family

The applications of the digital signal processing will include the following main applications.

1. General Purpose applications
    - waveform generation
    - Convolution and correlation
    - Digital filtering
    - Adaptive filtering
    - FFTs and fast cosine transform
2. Audio applications
    - Audio watermarking
    - Coding and decoding
    - Effects generator
    - Surround sound processing
    - Three dimensional audio
3. Communications:
    - Communication security
    - Detection
    - Encoding and Decoding
    - Software radios

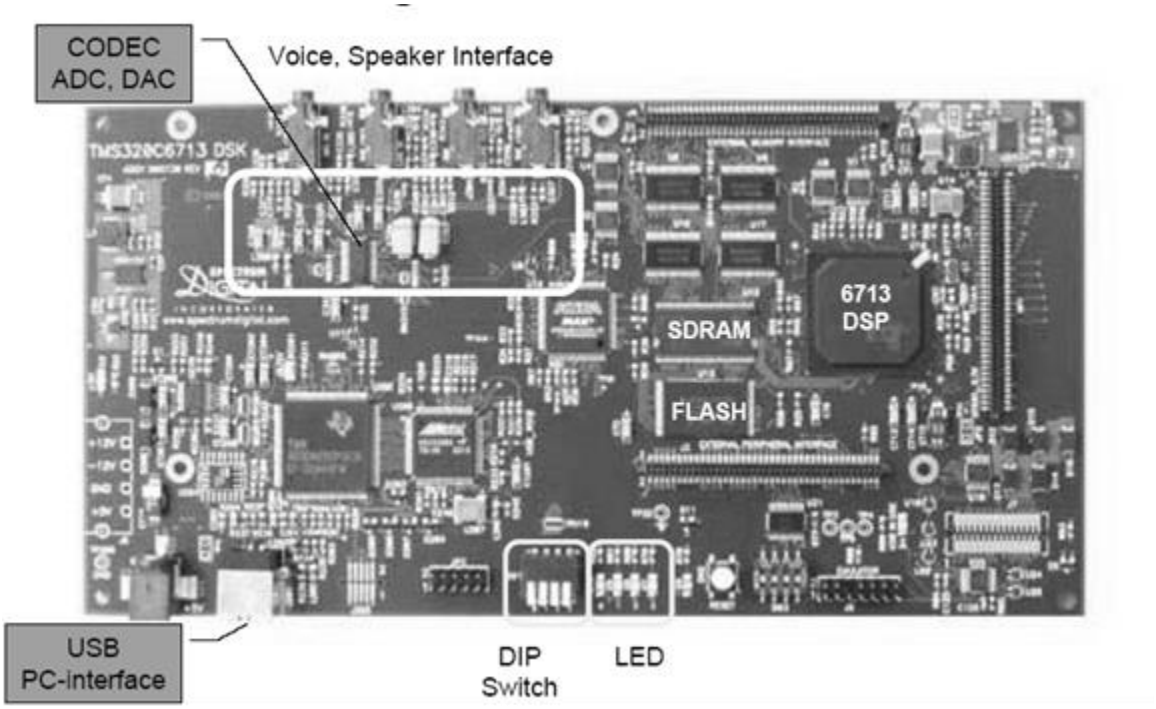
DSK Board

The DSK package is powerful, with the necessary hardware and software support tools for real-time signal processing. It is a complete DSP system. The DSK board, with an approximate size of

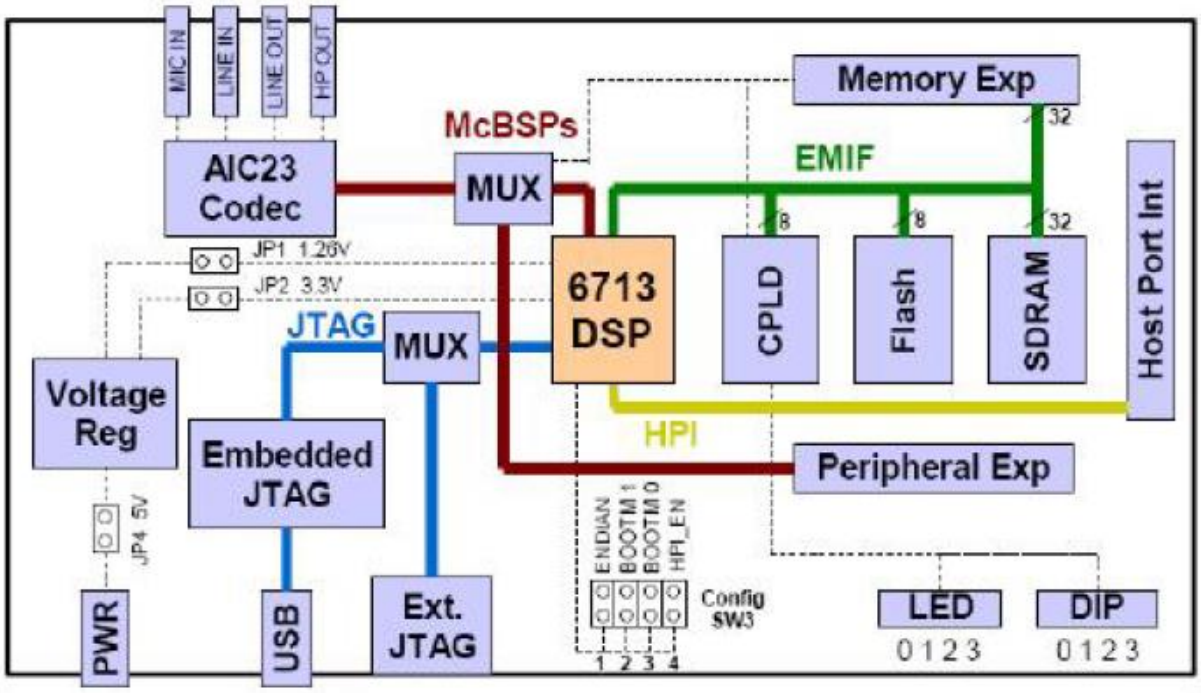
5 × 8 inches, includes the C6713 floating-point digital signal processor and a 32-bit stereo codecTLV320AIC23 (AIC23) for input and output.

The onboard codec AIC23 uses a sigma–delta technology that provides ADC and DAC. It is connected to a 12-MHz system clock. Variable sampling rates from 8 to96 kHz can be set readily. A daughter card expansion is also provided on the DSK board. Two 80-pin connectors provide for external peripheral and external memory interfaces.

The DSK board includes 16MB (megabytes) of synchronous dynamic random-access memory (SDRAM) and 256kB (kilobytes) of flash memory. Four connectors on the board provide input and output: MIC IN for microphone input, LINE IN for line input, LINE OUT for line output, and HEADPHONE for a headphone output (multiplexed with line output). The status of the four user dip switches on the DSK board can be read from a program and provides the user with a feedback control interface. The DSK operates at 225 MHz. Also onboard the DSK are voltage regulators that provide 1.26 V for the C6713 core and 3.3 V for its memory and peripherals.
The TMS320DSK6713 board and block diagram are shown in Figure 4



(a)



(b)

Figure 4 TMS320C6713-based DSK board: (a) board; (b) diagram. (Courtesy of Texas Instruments)

1.1.3 TLV320AIC23 (AIC23) ONBOARD STEREO CODEC FOR INPUT AND OUTPUT

The DSK board includes the TLV320AIC23 (AIC23) codec for input and output. The ADC circuitry on the codec converts the input analog signal to a digital representation to be processed by the DSP. The maximum level of the input signal to be converted is determined by the specific ADC circuitry on the codec, which is 6Vp-p with the onboard codec. After the captured signal is processed, the result needs to be sent to the outside world. Along the output path in Figure 3is a DAC, which performs the reverse operation of the ADC. An output filter smooth's out or reconstructs the output signal. ADC, DAC, and all required filtering functions are performed by the single-chip codec AIC23 on board the DSK.

The AIC23 is a stereo audio codec based on sigma–delta technology. The functional block diagram of the AIC23 codec is shown in Figure 5. It performs all the functions required for ADC and DAC, low pass filtering, oversampling, and soon. The AIC23 codec contains specifications for data transfer of words with length16, 20, 24, and 32 bits.

Sigma–delta converters can achieve high resolution with high over sampling ratios but with lower sampling rates. They belong to a category in which the sampling rate can be much higher

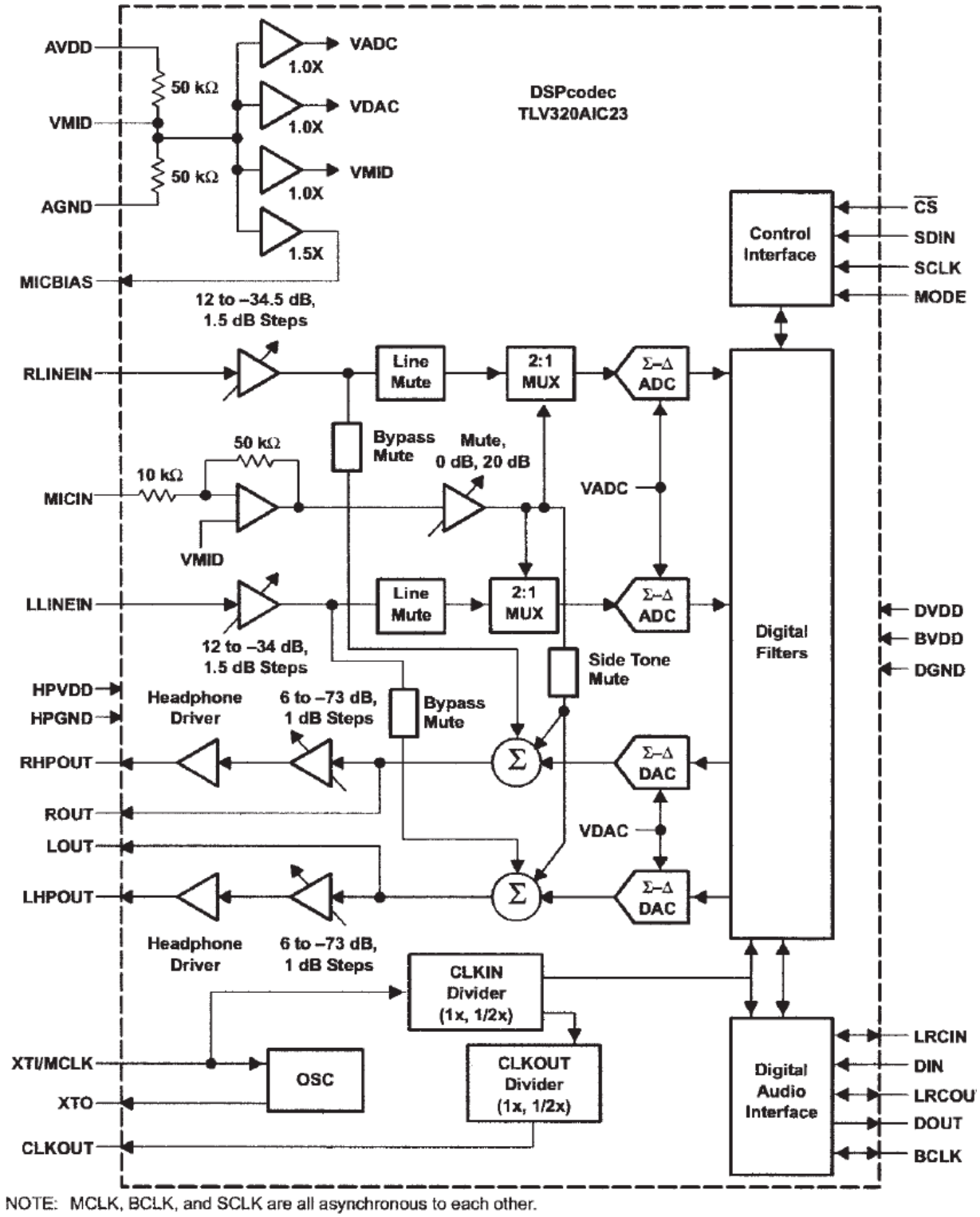


Figure 5TLV320AIC23 codec block diagram (Courtesy of Texas Instruments)

than the Nyquist rate. Sample rates of 8, 16, 24, 32, 44.1, 48, and 96 kHz are supported and can be readily set in the program.

A digital interpolation filter produces the oversampling. The quantization noise power in such devices is independent of the sampling rate. A modulator is included to shape the noise so that it is spread beyond the range of interest. The noise spectrum is distributed between 0 and $\frac{F_s}{2}$, so that only a small amount of noise is within the signal frequency band. Therefore, within the actual band of interest, the noise power is considerably lower. A digital filter is also included to remove the out-of band noise.

A 12-MHz crystal supplies the clocking to the AIC23 codec (as well as to the DSP and the USB interface). Using this 12-MHz master clock, with over sampling rates of $\mathbf{250} \times \boldsymbol{F_s}$and $\mathbf{272} \times \boldsymbol{F_s}$, an exact audio sample rate of 48 kHz (12MHz/250) and aCD rate of 44.1kHz (12MHz/272) can be obtained. The sampling rate is set by thecodec's register SAMPLERATE.

The ADC converts an input signal into discrete output digital words in a 2'scomplementformat that corresponds to the analog signal value. The DAC includes an interpolation filter and a digital modulator. A decimation filter reduces the digital data rate to the sampling rate. The DAC's output is first passed through an internal low pass reconstruction filter to produce an output analog signal. Low noise performance for both ADC and DAC is achieved using oversampling techniques with noise shaping provided by sigma–delta modulators. 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. Alternative I/O daughter cards can be used for input and output. Such cards can plug into the DSK through the external peripheral interface 80-pin connector J3 on the DSK board.

Programming the TMS320DSK6713

In order to program the DSK kit to perform a certain signal processing task we can use C code via a program called code composer studio or via MATLAB using simulink. In this lab we will use both programing methods, but the code composer studio will be considered first

1.1.4 CODE COMPOSER STUDIO

CCS provides an IDE to incorporate the software tools. CCS includes tools for code generation, such as a C compiler, an assembler, and a linker. It has graphical capabilities and supports real-time debugging. It provides an easy-to-use software tool to build and debug programs. The C compiler compiles a C source program with extension.c to produce an assembly source file with extension.asm. The assembler assembles an.asm source file to produce a machine language object file with extension.obj. The linker combines object files and object libraries as input to produce an executable file with extension.out. This executable file represents a linked common

object file format (COFF), popular in Unix-based systems and adopted by several makers of digital signal processors. This executable file can be loaded and run directly on the C6713 processor.

Any project created by the code composer must contain several files with different file types, these file types are listed below

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 filerts6700.lib
7. file.cmd: linker command file that maps sections to memory
8. file.obj: object file created by the assembler
9. file.out: executable file created by the linker to be loaded and run on the C6713 processor
10. file.cdb: configuration file when using DSP/BIOS

1.1.5 Support files

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

1. C6713dskinit.c: contains functions to initialize the DSK, the codec, the serial ports, and for I/O. you can find this file under the support folder provided by this lab
2. C6713dskinit.h: header file with function prototypes. Features such as those used to select the mic input in lieu of line input (by default), input gain, and so on are obtained from this header file (modified from a similar file included with CCS).
3. C6713dsk.cmd: sample linker command file. This generic file can be changed when using external memory in lieu of internal memory.
4. Vectors_intr.asm: a modified version of a vector file included with CCS to handle interrupts. Twelve interrupts, INT4 through INT15, are available, and INT11 is selected within this vector file. They are used for interrupt-driven programs.
5. Vectors_poll.asm: vector file for programs using polling.
6. rts6700.lib,dsk6713bsl.lib,csl6713.lib: run-time, board, and chip support library files, respectively. These files are included with CCS and are located in C6000\cgtools\lib, C6000\dsk6713\lib, and c6000\bios\lib, respectively.

PROGRAMMING EXAMPLES TO TEST THE DSK TOOLS

Three programming examples are introduced to illustrate some of the features of CCS and the DSK board. The primary focus is to become familiar with both the software and hardware tools. It is strongly suggested that you complete these three examples before proceeding to subsequent chapters

Example 1: Sine Generation Using Eight Points with DIP Switch Control (sine8_LED)

This example generates a sinusoid using a table lookup method. More important, it illustrates some features of CCS for editing, building a project, accessing the code generation tools, and running a program on the C6713 processor. The C source program sine8_LED.c shown in Figure 6 implements the sine generation and is included in the folder sine8_LED.

Program Consideration

Although the purpose is to illustrate some of the tools, it is useful to understand the program sine8_LED.c. A table or buffer sine_table is created and filled with eight points representing sin(t), where t = 0, 45, 90, 135, 180, 225, 270, and 315 degrees (scaled by 1000).

Within the function main, another function, comm_poll(), is called that is located in the communication and initialization support file c6713dskinit.c. It initializes the DSK, the AIC23 codec onboard the DSK, and the two McBSPs on the C6713 processor.

Within c6713dskinit.c, the function DSK6713_init initializes the BSL file, which must be called before the two subsequent BSL functions, DSK6713_LED_init()and DSK6713_DIP_init(), are invoked that initialize the four LEDs and the four dip switches.

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

The function output_sample( ), located in the communication support fileC6713dskinit.c, is called to output the first data value in the buffer or tablesine_table[0] = 0. The loop index is incremented until the end of the table isreached, after which it is reinitialized to zero.Every sample period $\boldsymbol{T} = \frac{\mathbf{1}}{\boldsymbol{F_s}} = \frac{\mathbf{1}}{\mathbf{8000}} = \mathbf{0.125}\boldsymbol{ms}$, the value of dip switch #0 istested, and a subsequent data value in sine_table (scaled by gain = 10) is sentfor output.Within one period, eight data values (0.125 ms apart) are output to generatea sinusoidal signal. The period of the output signal is $\boldsymbol{T} = \mathbf{8}(\mathbf{0.125}\boldsymbol{ms}) = \mathbf{1}\boldsymbol{ms}$,corresponding to a frequency of $\boldsymbol{f} = \frac{\mathbf{1}}{\boldsymbol{T}} = \mathbf{1}\boldsymbol{kHz}$.

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

#include "dsk6713_aic23.h"                  //support file for codec,DSK
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;        //set sampling rate
short loop = 0;                             //table index
short gain = 10;                            //gain factor
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707};//sine values

void main()
{
 comm_poll();                               //init DSK, codec, McBSP
 DSK6713_LED_init();                        //init LED from BSL
 DSK6713_DIP_init();                        //init DIP from BSL
 while(1)                                   //infinite loop
 {
  if(DSK6713_DIP_get(0)==0)                 //=0 if switch #0 pressed
  {
    DSK6713_LED_on(0);                      //turn LED #0 ON
    output_sample(sine_table[loop]*gain); //output every Ts (SW0 on)
    if (++loop > 7)  loop = 0;              //check for end of table
  }
  else DSK6713_LED_off(0);                  //LED #0 off
 }                                          //end of while (1)
}                                           //end of main
```

Figure 6 Sine generation program using eight points with dip switch control (sine8_LED.c).

1.1.6 How to create a project

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

1. To create the project file sine8_LED.pjt. Select File → New → CCS Project. Type sine8_LED for the project name, as shown in Figure 7. This project file is saved in the folder sine8_LED (choose your directory).The .pjt file stores project information on build options.
2. Choose the project type as C6000 then click next. In Project Setting window, choose the settings as shown in Figure 8.

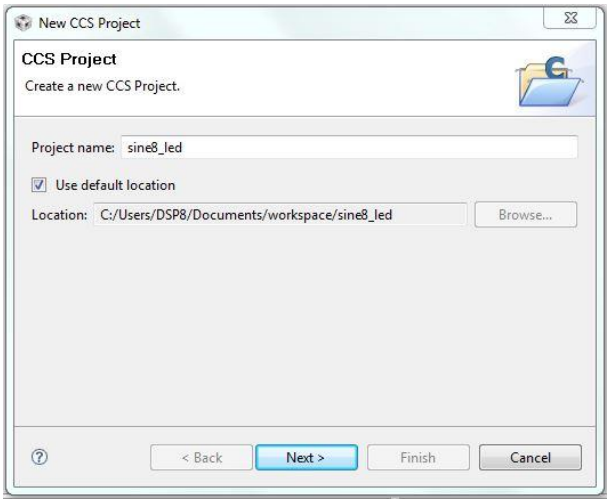


Figure 7 CCS Project windows for sine8_LED project creation

3. Add the command file C6713DSK.cmd from the support folder in the extension "C:\Users\DSP8\Documents\workspace\support\C6713DSK.cmd" to the Linker Command File.
4. Choose rts6700.lib for Run Support Library. Click Next.
5. In Project Template window, choose DSP/BIOSv5.xx Examples → Empty Example.
6. Copy the files under support folder (except C6713DSK.cmd) into your project folder and choose one of the files vector_intr.asm or vector_poll.asm as your code requires.

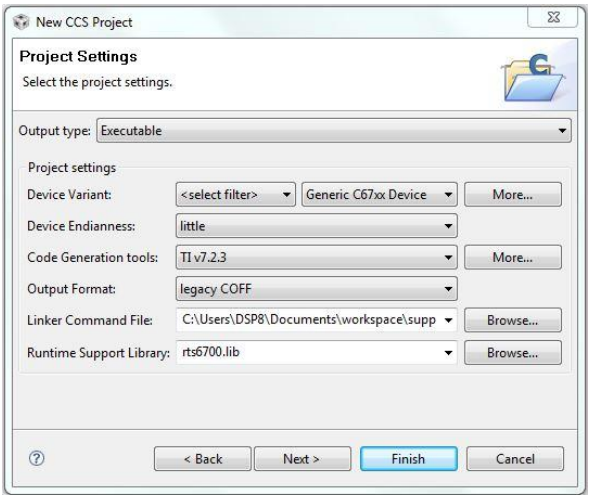


Figure 8CCS project settings configuration

7. To add CSL file, right click on the project name and choose Build Properties → Include options and select Add item to add "lib_3x" and "include"folder from the extension "C:\Program Files\C6xCSL" as shown in Figure 9. From Build Properties → C6000 Linker → File Search Path add CSL6713.lib file from the extension "C:\Program Files\C6xCSL\lib_3x".

8. To add dsk6713bsl.lib file, Select Build Properties → C6000 Linker → File Search Path then select add item. You can find this file in the extension "C:\Users\DSP8\Documents\workspace\dsk6713revc_files\CCStudio\c6000\dsk6713\lib\dsk6713bsl".
9. To add target configuration file, right click to project name and choose New → Target Configuration File then click Finish. Choose the settings as shown in Figure 10. Note that this step is to define the connection between DSK and the program using USB.
10. To add DSP/BIOS configuration file from New→ DSP/BIOSv5.x Configuration File. Click Next and choose dsk6713 as shown in Figure 11 then click Finish. Note that this step configures DSK memory.
11. To add you source file to the project, right click to project name and choose New → File.

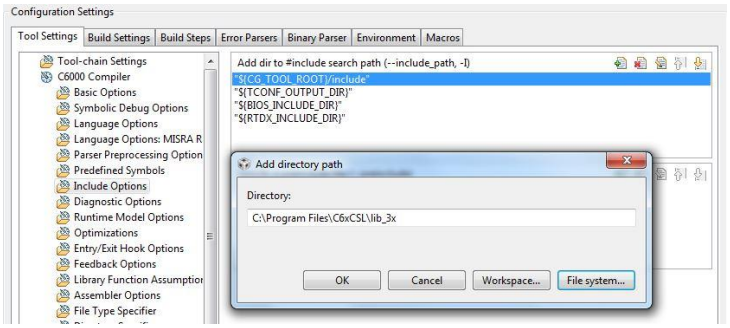


Figure 9CCS project include options settings

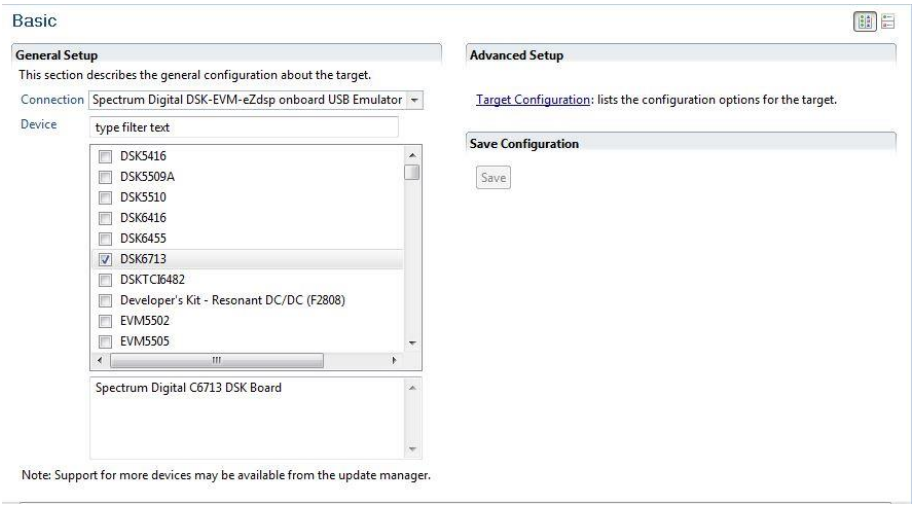


Figure 10CCS project Target Configurations

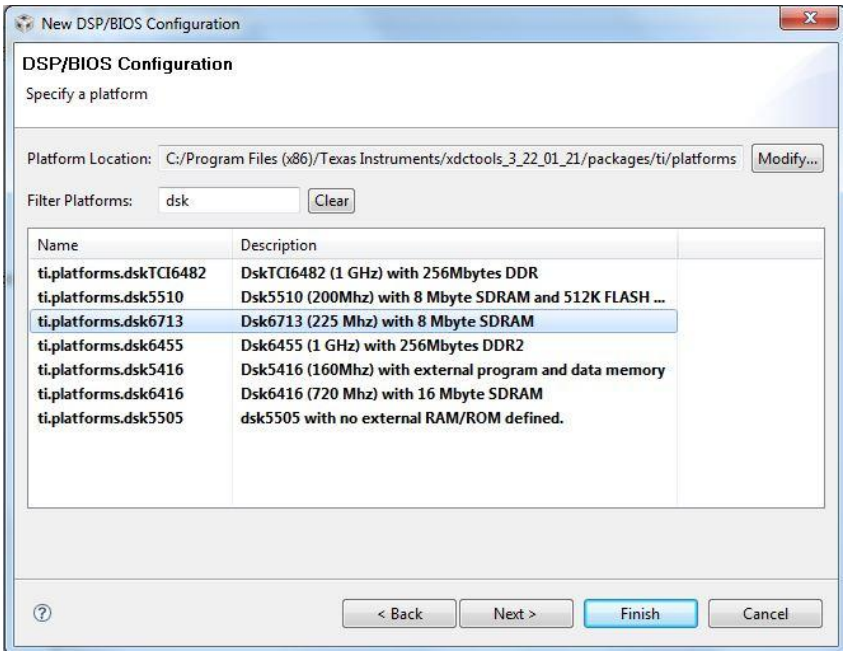


Figure 11DSP/BIOS Cofigurations

1.1.7 Code Generation and Options

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

Compiler Option

Select Project → Build Properties → C6000 Compiler. Select Predefined Symbols and type for Define Symbols“CHIP_6713”. Then select Runtime Model Options and change Data Access Model to Far as shown in Figure 12.

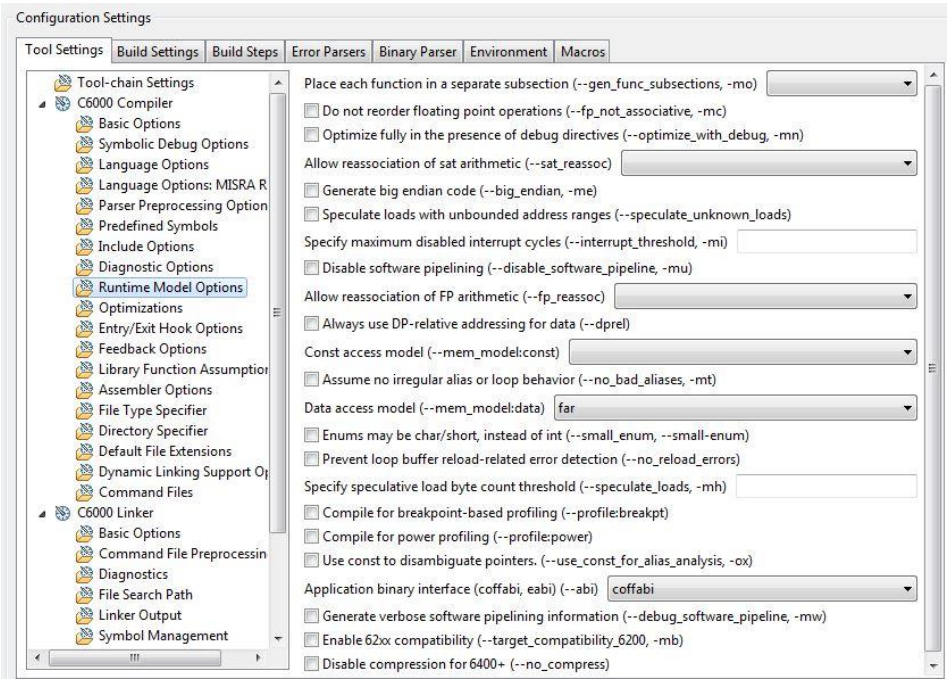


Figure 12Compiler Runtime Configurations

Linker Option

Click on C6000 Linker (from CCS Build Properties). In Basic options, the output filename sine8_LED.out defaults to the name of the .pjt filename. Set the stack size and heap size to 0x400. Run-time Environment defaults for Auto initialization Model. 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 used to name the linked executable output file sine8_LED.out. Press OK.

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

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

Building and Running the Project

The project sine8_LED can now be built and run.

1. Build this project as sine8_LED. Select Project → Rebuild All or press the toolbar with the icon that says "Rebuild Active Project". This compiles and assembles all the C files using cl6x and assembles the assembly file vectors_poll.asm usingasm6x. The resulting object files are then linked with the library files. 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. Figure 13 shows several windows within CCS for the project sine8_LED.The building process causes all the dependent files to be included.
2. Select Target → Debug Active Project in order to load sine_LED.out or use the toolbar with the icon that says Debug Launch (CCS includes an option to load the program automatically after a build).It should be in the folder sine8_LED\Debug. Select Target → Run or use the toolbar with the icon that says Run. Connect the LINE OUT terminal to the picoscope and observe the resulting sine wave when the dip switch #0 is pressed on the DSK board.

The sampling rate $\boldsymbol{F_s}$ of the codec is set at 8 kHz. The generated frequency $\boldsymbol{isf} = \frac{\boldsymbol{Fs}}{\boldsymbol{numberofpoints}} = \boldsymbol{8}\frac{\boldsymbol{kHz}}{\boldsymbol{8}} = \boldsymbol{1kHz}$. Note that the amplitude of the generated signal is approximately 0.8 V p-p (peak to peak).

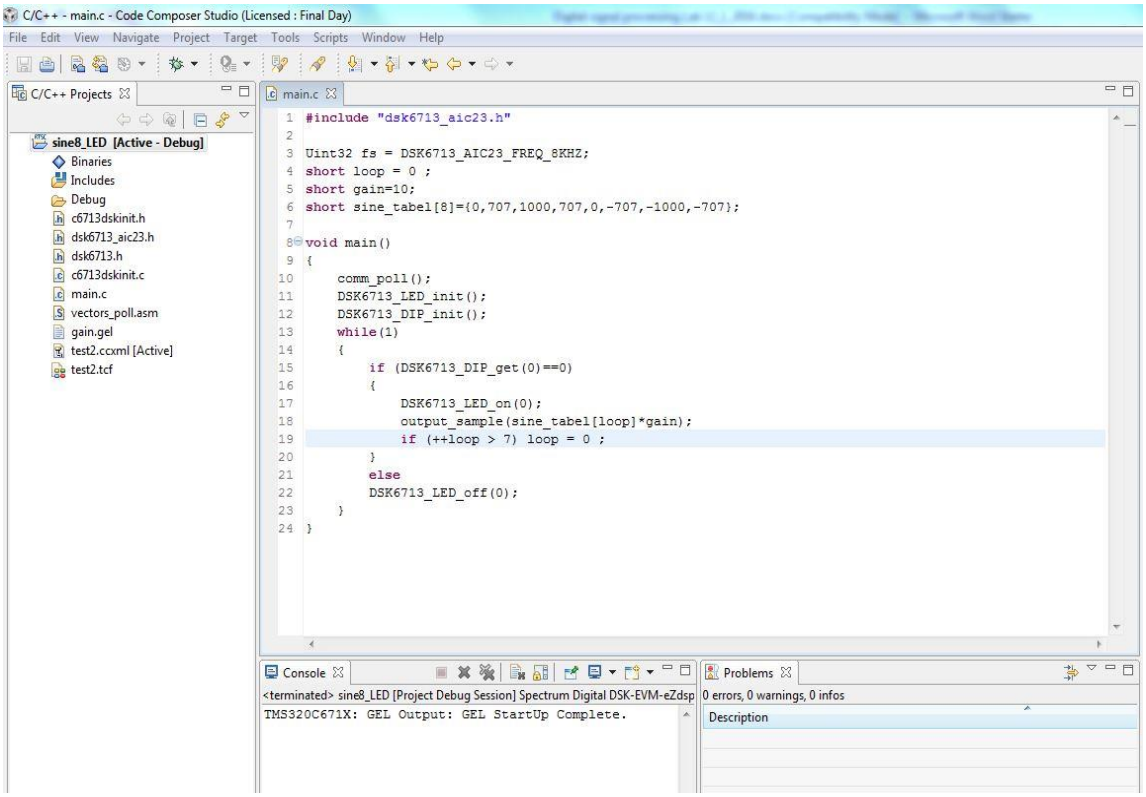


Figure 13CCS windows for project sine8_LED.

Monitoring the Watch Window

Verify that the processor is still running (and dip switch #0 is pressed). Note the indicator "DSP RUNNING" at the bottom left of CCS. The Watch window allows you to change the value of a parameter or to monitor a variable:

1. Select View → Watch, which should be displayed on the upper section of CCS. Type gain, then click enter. The gain value of 10 set in the program should appear in the Watch window.
2. Change gain from 10 to 30 in the Watch window. Press Enter. Verify that the volume of the generated tone has increased (with the processor still running and dip switch #0 is pressed). The amplitude of the sine wave has increased from approximately 0.8 Vp-p to approximately 2.5Vp-p.
3. Change gain to 33 (as in step 2). Verify that a higher-pitched tone exists, which implies that the frequency of the sine wave has changed just by changing its amplitude. This is not so. You have exceeded the range of the codecAIC23. Since the values in the table are scaled by 33, the range of these values is now between ±33,000. The range of output values is limited from $-\mathbf{2^{15}}$to ($\mathbf{2^{15}-1}$) , or from -32,768 to +32,767.

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

Applying the Slider Gel File

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

1. Before debug the project, select File →new →File and create new file. Write the code shown in Figure 14 in the new file and save this file in your project directory (sine8_LED) as gain.gel
2. Press Debug Launch icon, then select Tools → GEL Files then from the new window that will be opened, right click and choose Load GEL and then select the file gain.gel. By creating the slider function gain shown in Figure 14, you can start with an initial value of 10 (first value) for the variable gain that is set in the C program, up to a value of 35 (second value), incremented by 5 (third value).

3. Select Scripts → Sine Gain → Gain. This should bring out the Slider window shown in Figure 15, with the minimum value of 10 set for the gain.
4. Press the up-arrow key to increase the gain value from 10 to 15, as displayed in the Slider window. Verify that the volume of the sine wave generated has increased. Press the up-arrow key again to continue increasing the slider, incrementing by 5 up to 30. The amplitude of the sine wave should be about 2.5 V p-p with a gain value set at 30. Now use the mouse to click directly on the Slider window and slowly increase the slider position to 31, then 32, and verify that the frequency generated is still 1 kHz. Increase the slider to 33 and verify that you are no longer generating a 1-kHz sine wave. The table values, scaled by the gain value, are now between ±33,000 (beyond the acceptable range by the codec).

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

menuitem "Sine Gain"

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

Figure 14 GEL file to slide through different gain values in the sine generation program (gain.gel).

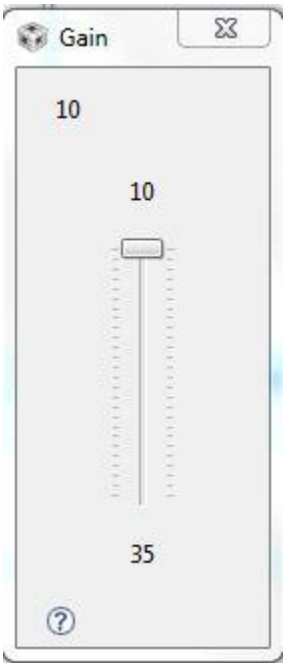


Figure 15 Slider window for varying the gain of generated sine wave.

Changing the Frequency of the Generated Sinusoid:

1. Change the sampling frequency from 8 kHz to 16 kHz by setting $\boldsymbol{f_s}$ in the C source program to DSK6713_AIC23_FREQ_16KHZ. Rebuild (use Rebuild Active Project) the project, load and run the new executable file, and verify that the frequency of the generated sinusoid is 2 kHz. The sampling frequencies supported by the AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96 kHz.

2. Change the number of points in the lookup table to four points in lieu of eight points for example, {0, 1000, 0, -1000}. The size of the array sine_table and the loop index also need to be changed. Verify that the generated frequency is $\boldsymbol{f} = \frac{\boldsymbol{F_s}}{\boldsymbol{numberofpoints}}$.

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

Two sliders can readily be used, one to change the gain and the other to change the frequency. A different signal frequency can be generated by changing the loop index within the C program (e.g., stepping through every two points in the table).When you exit CCS after you build a project, all changes made to the project can be saved. You can later return to the project with the status as you left it before.

Sine wave Generation (Simulink Matlab):

1- Open MATLAB R2012b , Then choose ***Home → New → Simulink Model***.
2- Choose → ***LibraryBrowser*** from toolbar, choose Embedded Coder thenEmbedded Targets then drag Target Preferences to your file and choose the properties of this block as in the figure below and finally click yes.

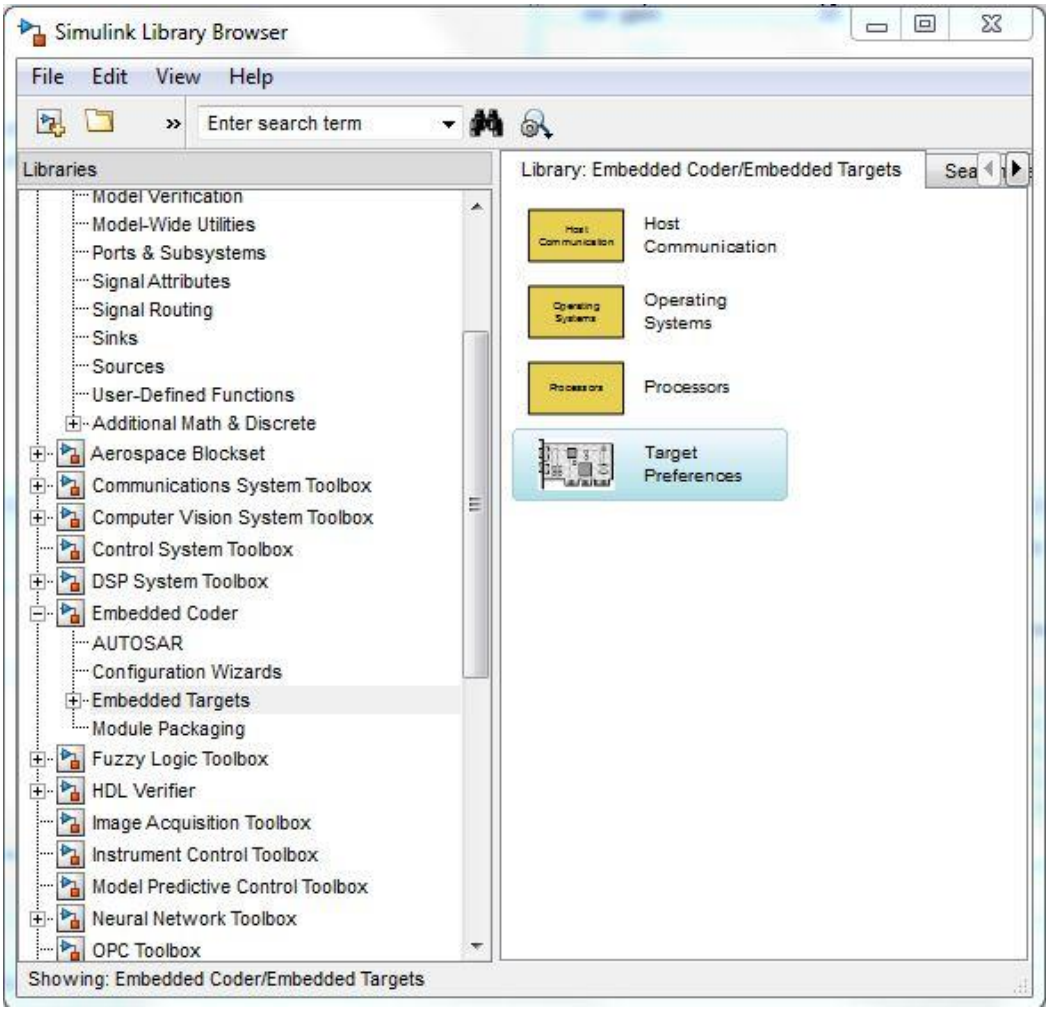


Figure 16 Simulink library browserfor DSK boards

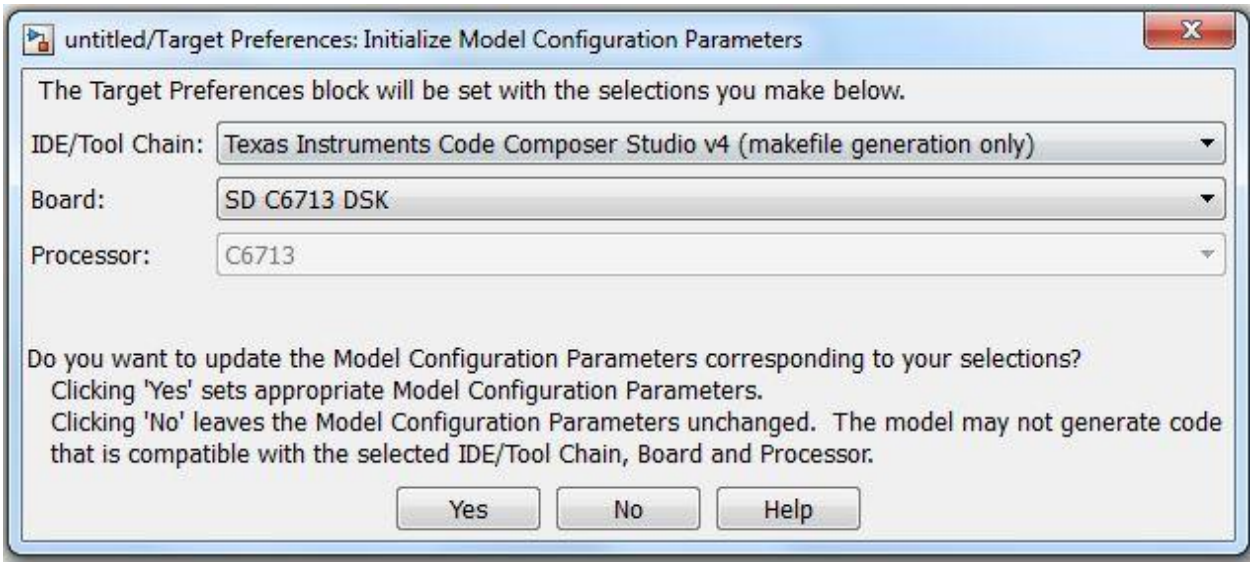


Figure 17Target Preferences window

3- Choose Embedded Coder again, then choose embedded targets, then choose Processors, then choose C6713 DSK, then drag DAC (digital to analog convertor) to your file.

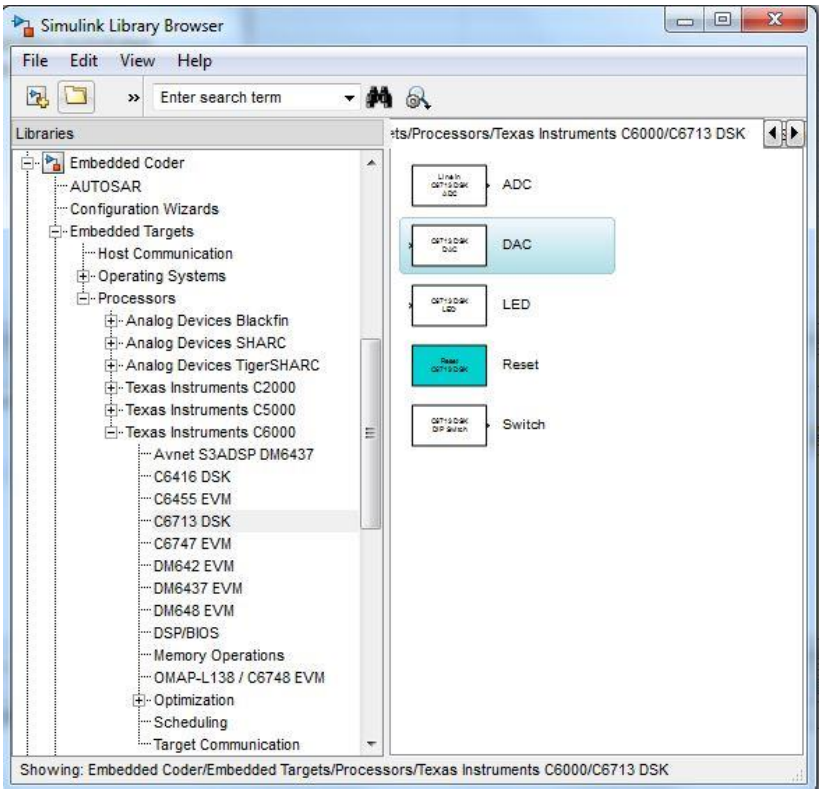


Figure 18 Simulink library browser for DSK6713 board

4- Choose DSP System Toolbox, then choose Sources and drag Sine Wave to your file.

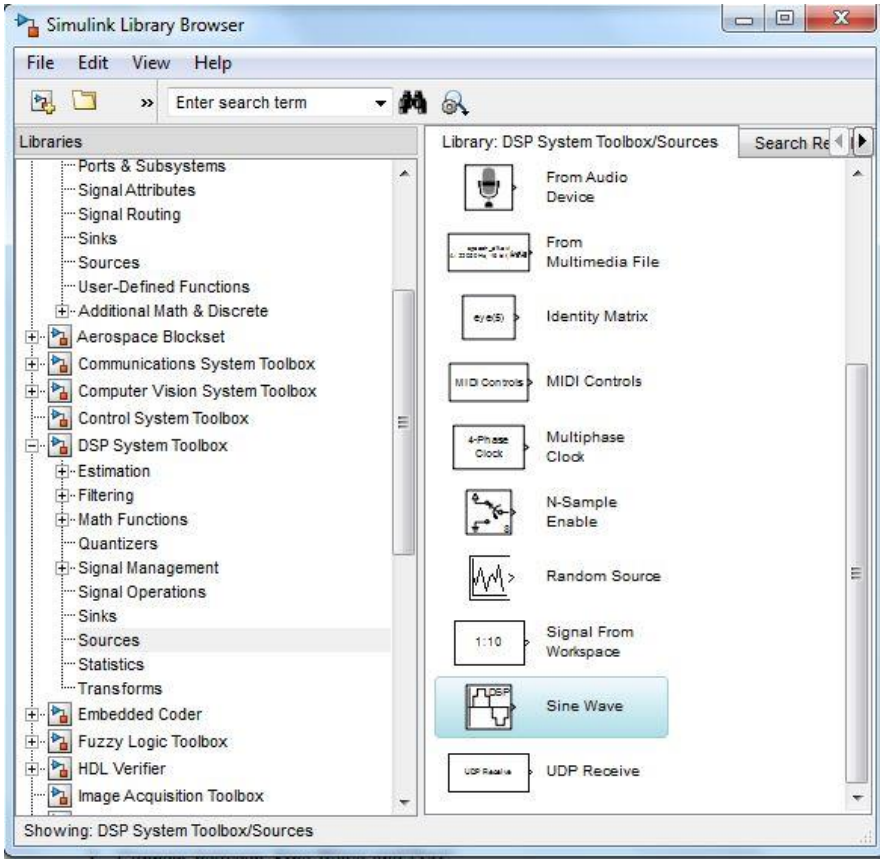


Figure 19 Simulink library browser for signal processing sources

5- Connect between Sine Wave and DAC

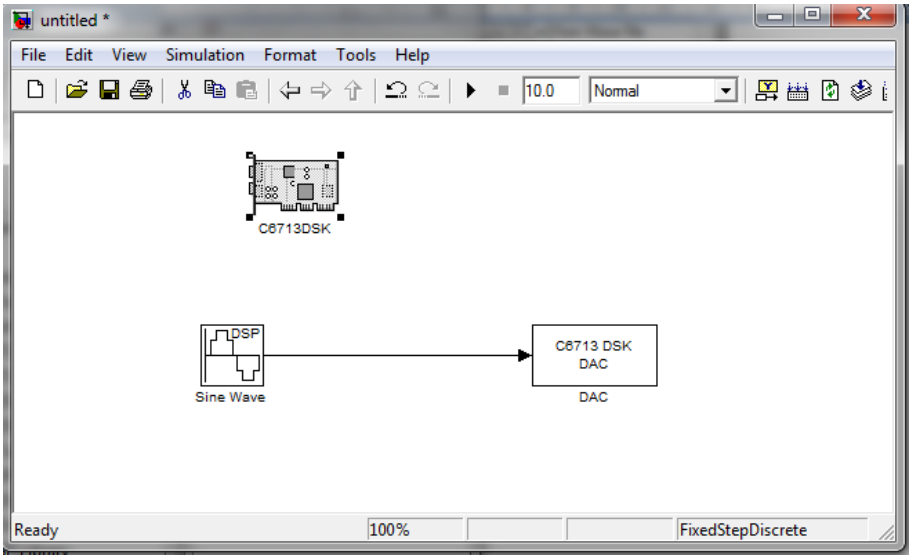


Figure 20 DSK6713 model connection

6- Double Click on Sine Wave Block and Choose the required parameters:

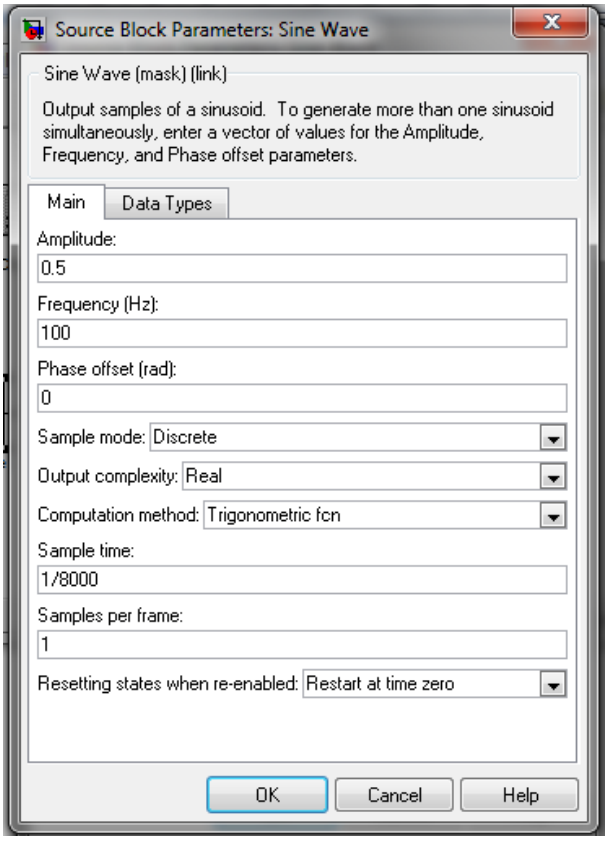


Figure 21 Sine wave block parameters

7- Double Click on DAC Block and choose the required parameters:

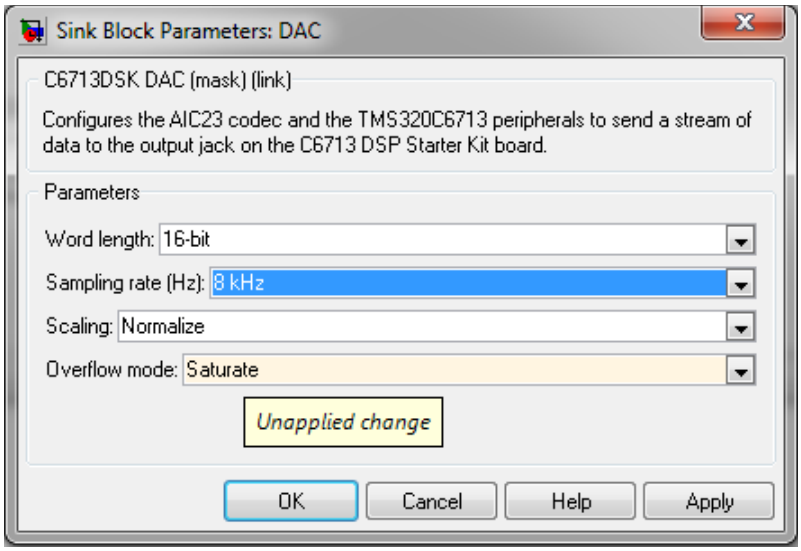


Figure 22 C6713DSK DAC parameters

Note that the sampling frequency for both Sine wave and DAC must be the same.

8- Save your file in any directory.

9- From Simulation→Model Configuration Parameters→Solver, change stop time to inf.

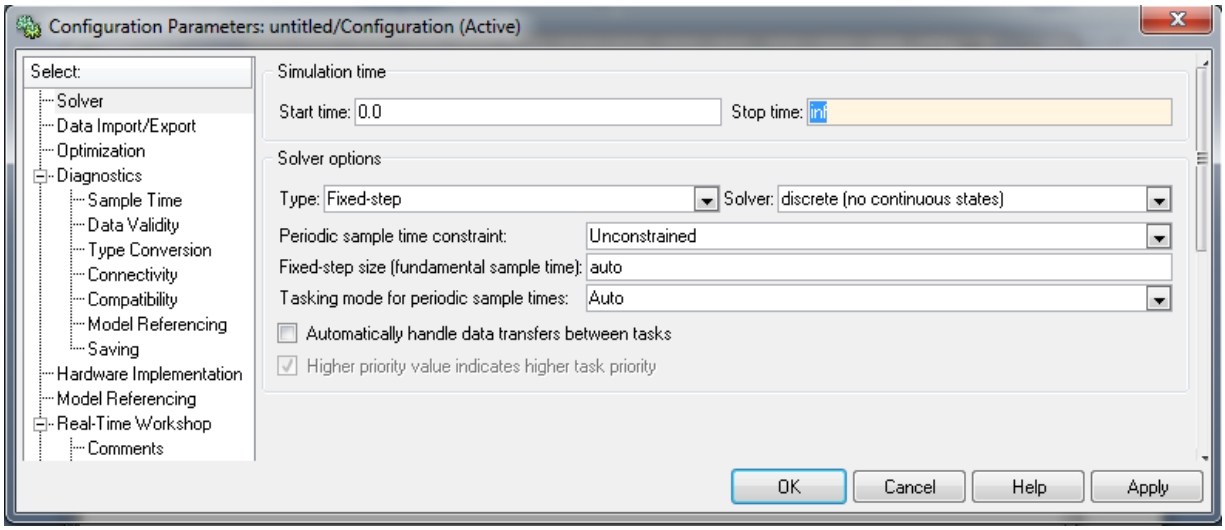


Figure 23 Simulation parameters configurations

10- Choose Build Model icon (Notice the execution steps that done automatically). Verify the results on CRO.

11- Try to modify the frequency and amplitude.

2 Experiment 2 Wave Generation and Sampling

Objectives

The objectives of this experiment are

a) Show the students how to use DSP processors to generate real time signals such as square wave, ramp signal
b) Verify the sampling theory and aliasing

Wave generation

Square wave can be generated either by using either the Fourier series expansion or by using direct synthesis. Each of these methods will be described briefly in the next subsection

2.1.1 Square wave using Fourier series

Square wave can be generated as a an infinite sum of odd harmonics according to Fourier series expansion as described mathematically by

$$s(t) = \sum_{\substack{n=1 \\ n\ odd}}^{n=\infty} \frac{1}{n} \sin(n\omega_0 t) \qquad (1)$$

In equation (1),$\mathbf{s(t)}$ represents the square wave,$\boldsymbol{n}$ is the number of harmonics and $\boldsymbol{\omega_0}$ is the fundamental radian frequency of the generated wave.

However it is practically impossible to include an infinite number of harmonics to synthesize the square wave. Only few numbers of harmonics can be used to synthesize the square wave.

If for example we used two harmonics to generate the square wave then we would have a square wave as shown Figure

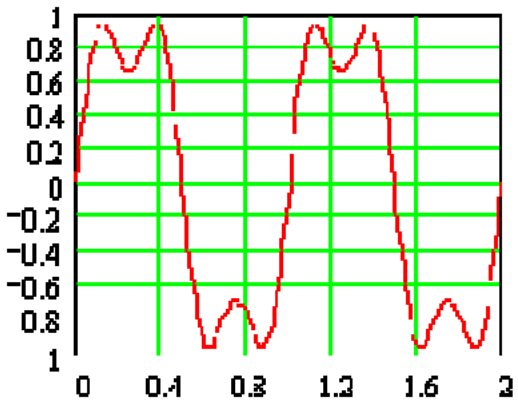


Figure 2.1 Square wave generated by the addition of two harmonics

If the number of harmonics $\boldsymbol{n}$ is increased to $\boldsymbol{n = 11}$, then the signal gets closer to the perfect square as shown in Figure

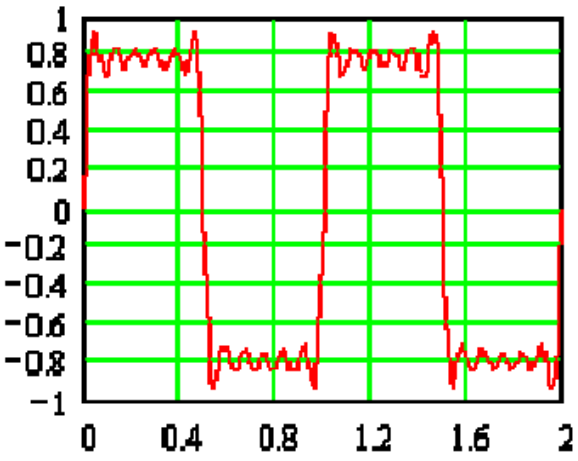


Figure 2.2 Square wave generated by the addition of eleven harmonics

If the number of harmonics is increased more, then the synthesized square wave gets closer to an ideal square wave as shown by Figure

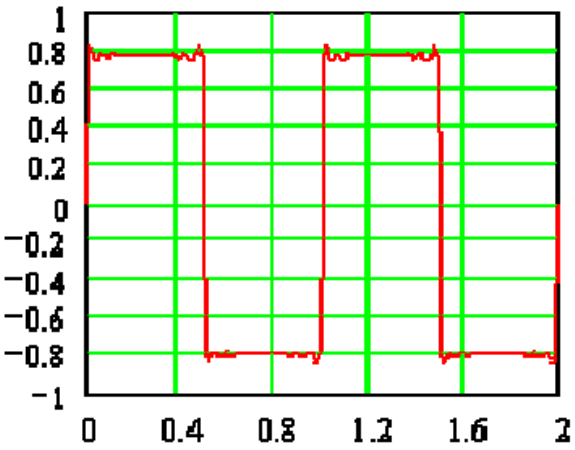


Figure 2.3 Square wave generated by the addition of harmonics where the number of harmonics $n = 79$

2.1.2 Square wave generation using direct synthesis

It can be noticed from the previous graph that the ripple present in the generated wave from, and both the rise and the fall time became smaller as the number of the harmonics used is increased more. The ripple is known as the Gibbs phenomenon.

The second way to generate square wave is to use a lookup table with a finite length. If half of the look up table is filled by the maximum DAC amplitude of the AIC23 codec ($\boldsymbol{maximumamplitude} = \mathbf{2^{15}} - \mathbf{1} = \mathbf{32767}$) and the second half of the table is filled by minimum amplitude ($\boldsymbol{minimumamplitude} = -\mathbf{2^{15}} = -\mathbf{32768}$), then a square wave will be generated.

2.1.3 Ramp signal generation

Ramp signals are useful type of signal used in measurement systems to perform sweep operation. This kind of signals can be used in the design of spectrum analyzers, GSM jammers and many other applications.

Sampling

Sampling is the process of converting a continuous time signal into discrete time signal. As we know from the sampling theory, the sampling frequency must be at least greater than twice the bandwidth of the analog signal as explained by the sampling theory

$$f_s \geq 2f_m \qquad (2)$$

Where $\boldsymbol{f_m}$ is the maximum frequency content of the analog signal.

In this experiment a simple program for sampling a signal from the function generator will be considered

Experimental procedures

In this section the various methods for square wave and ramp signal generation will be produced using the procedures detailed below.

2.1.4 Square wave generation using Fourier series

1. Open the project that you have made in the previous lab.
2. Add vectors_intr.asm file to the project.
3. Open new source file and save it as sintosquare.c
4. Use the following code in your c file

```
#include"DSK6713_AIC23.h"              // this file is added to initialize the DSK6713
#include "math.h"                                  // header file used when mathematical
instructions are executed
#define N 8000                                     // no. of samples
#define numHarm 9
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;                    // set the sampling frequency,
Different sampling frequencies supported by AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96
kHz.
short y[numHarm]={0},x=0;
short sine_table[numHarm][N];

short i,j,k,gain=1000.0,y[numHarm]={0};     // variable declaration

interrupt void c_int11()            // ISR call, At each Interrupt, program execution goes to the
interrupt service routine
{

        for(k=0;k<numHarm;k++){
                if(k==0)
                        y[k] = 21.0*sine_table[k][i];
                else
                        y[k] = y[k-1] + (21.0/((2.0*k)+1))*sine_table[k][i];

        }

        if(i< N-1) ++i;                             // the index loop is incremented by an
amount equal to N
        else i = 0;
   output_sample(y[x]);
```

```
        return;                                          // program execution goes back to
while(1) and then again starts listening for next interrupt and this process goes on
}

void main()
{
        float pi = 3.14159;                    // variable declaration

        for(j = 0; j <= numHarm; j++){
                for(i = 0;i< N;i++)
                {
                        sine_table[j][i] = gain*sin((2.0*pi*i/8000.0)*200.0*((2.0*j)+1));
                }
        }
        i = 0;
        comm_intr();                    // ISR function is called, using the given command
        while(1);                              //program execution halts and it starts
listening for the interrupt which occur at every sampling period Ts.
}
```

5. Exercise add slider to your program in order to change the number of harmonics from 1 to 9 and see the effect on the ripple and the.

Question 1. What happens to the ripple if the number of harmonics is increased? What happens to the ripple if the number of harmonics is reduced?

2.1.5 Square wave generation using direct thesis

1. Write a c code to generate a square wave using interrupt technique. You may use the code in Figure for assistance

```
//Squarewave.c Generates a squarewave using a look-up table

#include "dsk6713_aic23.h"          //codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;  //set sampling rate
#define table_size (int)0x40        //size of table=64
short data_table[table_size];       //data table array
int i;

interrupt void c_int11()            //interrupt service routine
{
 output_sample(data_table[i]);      //output value each Ts
 if (i < table_size) ++i;           //if table size is reached
 else i = 0;                        //reinitialize counter
 return;                            //return from interrupt
}

main()
{
for(i=0; i<table_size/2; i++)        //set 1st half of buffer
   data_table[i] = 0x7FFF;           //with max value (2^15)-1
for(i=table_size/2;i<table_size;i++) //set 2nd half of buffer
   data_table[i] = -0x8000;          //with -(2^15)
i = 0;                               //reinit counter
comm_intr();                         //init DSK, codec, McBSP
while (1);                           //infinite loop
}
```

Figure 2.4 Square-wave generation program (squarewave.c)

Question 2. What is the frequency of the generated square wave?

Question 3. explain how the frequency of the generated wave can be modified

Question 4. change the frequency of the square wave to 2 kHz and explain what if you still square wave shape or not.

2.1.6 Ramp signal generation procedure

1. Write a c code to generate a ramp wave using the interrupt technique. You may use the code in **Error! Reference source not found.** for assistance

```
//Ramp.c Generates a ramp

#include “dsk6713_aic23.h”                  //codec-DSK support file
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;        //set sampling frequency
short output;
```

```
interrupt void c_int11(){                          //interrupt service routine
        output_sample(output);          //output each sample period
        output += 512;                             //increment output value
        if(output >= 20480)           //if peak is reached
        output = 0;                          //reinitialize
        return;                             // return from interrupt
}

void main(){
        output = 0;                           // init output to zero
        comm._intr();                       // init DSK, codec, McBSP
        while(1);                           // infinite loop
}
```

1- Observer the generated signal you see on the oscilloscope. Verify that a ramp with negative slope is generated and measure its frequency.
2- Explain how you can calculate the frequency of the ramp wave depending on the above code.
3- To obtain a ramp with a positive slope, change output to output -= 512;Also change the if statement to reinitialize output, or if (output <= -20480). Verify the result.
4- Explain how the frequency and slope of the ramp signal can be changed

2.1.7 Basic sampling

In this part of the circuit a small program will be written to read data from the function generator connected to line in of the DSK6713 kit. The sampling frequency used for reading the data is 8 kHz

1. Write a c program to read the signal from the LINE IN input of the DSK6713 and write the data back to the line out. You can use the code shown in Figure to help you performing this task

```
//Loop_intr.c Loop program using interrupt.Output=delayed inp

#include "dsk6713_aic23.h"          //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;  //set sampling rate

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

   sample_data = input_sample();    //input data
   output_sample(sample_data);      //output data
   return;
}

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

Figure 2.5 Loop program using interrupt (loop_intr.c)

2. After compiling and running your program, connect a signal from the function generator to the LINE IN terminal
3. Set the function generator to sine wave, the amplitude of the signal to 1 V_{PP}, and the frequency to 1 kHz and plot the signal as it appears on the screen of the oscilloscope
4. Repeat step 3 for a square wave
5. Increase the frequency of the modulated signal from 1 kHz to 5 kHz, then plot the signal as it appears on the oscilloscope. Explain the signal shape of the signal you see on the oscilloscope

Question 5. Determine the frequency of the reconstructed signal if the input signal frequency is $\boldsymbol{f = 7kHz}$ and the sampling frequency is 8 kHz

3 Experiment 3Amplitude Shift Keying (ASK) and Frequency shift keying (FSK)

Objectives

The aim of this experiment to show that a complex modulation schemes such as ASK or FSK can be generated by DSP processors before transmitting data such as voice through a wireless channel

Introduction

Amplitude Shift Keying (ASK) and frequency shift keying are two fundamental digital modulation techniques. The generation of these two modulation schemes will be considered in this experiment

Theory of ASK

In amplitude shift keying, the amplitude of the carrier signal is varied to create signal elements. Both frequency and phase remain constant while the amplitude changes. In ASK, the amplitude of the carrier assumes one of two amplitudes depending on the logic states of the input bit stream. This modulated signal can be expressed as

$$s(t) = \begin{cases} A\cos(2\pi f_c t) & logicone \\ 0 & logiczero \end{cases}$$

Amplitude shift keying in the context of digital signal communications is a modulation process, which imparts to a sinusoid two or more discrete amplitude levels. These are related to the number of levels adopted by the digital message. For a binary message sequence there are two levels, one of which is typically zero. Thus the modulated waveform consists of bursts of a sinusoid. Figure 3.1 illustrates a binary bit stream and the ASK modulated signal. Neither signal is band limited.

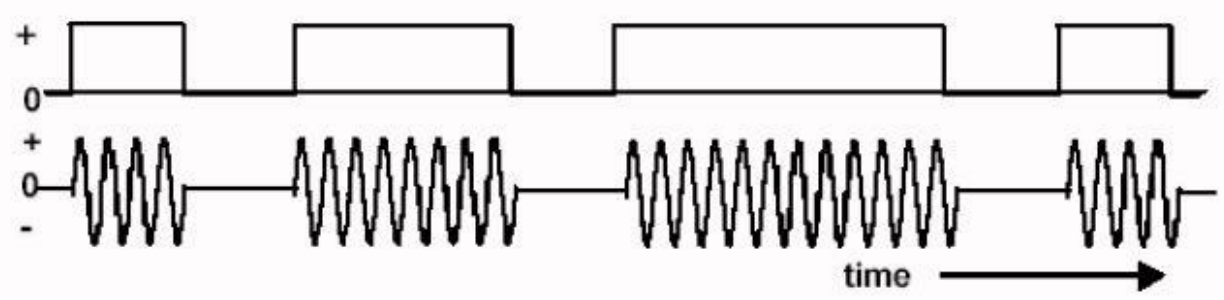


Figure 3.1 an ASK signal (below) and the message (above)

There are sharp discontinuities shown at the transition points. These result in the signal having an unnecessarily wide bandwidth. Band limiting is generally introduced before transmission, in which case these discontinuities would be 'rounded off'. The band limiting may be applied to the digital message, or

the modulated signal itself. The data rate is often made a sub-multiple of the carrier frequency. This has been done as illustrated in Figure 3.1.

Recall that ASK signal is generated by multiplying the bit stream by the carrier signal. The carrier frequency and the number of cycles per bit are related by

$$\boldsymbol{f_c = \frac{numbrofcycles}{T_b}}$$

If for example a given bit stream is to be transmitted at a bit rate of 1200 bits/s, then the bit duration is $\boldsymbol{T_b = \frac{1}{1200} = 833.33\mu s}$. If the carrier is to have 4 cycles per bit then, then the carrier frequency must be selected such that $\boldsymbol{f_c = \frac{4}{833.33\times10^{-6}} = 4800Hz}$.

The other parameters to specify when implementing ASK is to specify the sampling frequency and the number of samples. For the example mentioned in this experiment, the sampling frequency must be $\boldsymbol{f_s \geq 2f_m \geq 2 \times 4800 \geq 9600Hz}$. Therefore we select a sampling frequency of 48 kHz to satisfy the Nyquist criteria. The number of samples is determined from $\boldsymbol{\frac{nf_c}{f_s} = numberofcycles \Longrightarrow n \times \frac{4800}{48000} = 4 \Rightarrow n = 40samples}$

Theory of FSK

Frequency shift keying is a frequency modulation scheme in which digital information is transmitted through discrete frequency changes of a carrier wave. The simplest FSK is *binary FSK* (BFSK). BFSK uses a pair of discrete frequencies to transmit binary (0s and 1s) information. With this scheme, the "1" is called the mark frequency and the "0" is called the space frequency. The time domain of an FSK modulated carrier is illustrated in Figure 3.2

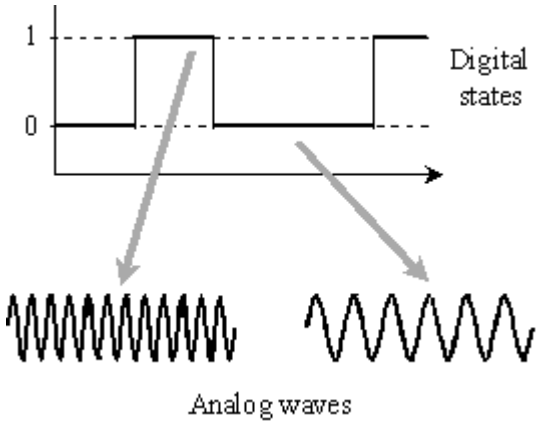


Figure 3.2 time domain representation of FSK signals

Frequency-shift keying (FSK) is a method of transmitting digital signals. The two binary states, logic 0 (low) and 1 (high), are each represented by an analog waveform. Logic 0 is represented by a wave at a specific frequency, and logic 1 is represented by a wave at a different frequency. A modem converts the binary data from a computer to FSK for transmission over telephone lines, cables,

optical fiber, or wireless media. The modem also converts incoming FSK signals to digital low and high states, which the computer can "understand."

The relation between the carrier frequencies used to represent logic 1 and logic 0 are determined from

$$f_1 - f_2 = \frac{h}{T_b}$$

Where f_1 is the frequency used to represent logic 1 while f_2 is the frequency used to represent logic 0, and $h = 0.5, 1, 1.5, \dots, etc$ is called the deviation ratio.

If $h = 0.5$, then the we have a special kind of FSK called minimum shift keying (MSK). In MSK minimum means that the minimum frequency spacing between logic 1 and logic where the carriers are still orthogonal. If $h < 0.5$ then the carriers representing logic 1 and logic 0 are no longer orthogonal, this will degrade the average probability of error performance of the FSK system.

if$h$ is made larger, the demodulation of the FSK signal became simpler. However larger values of h means that the peak frequency deviation is larger, which means that a larger bandwidth requirements for the FSK signal.

The approximate bandwidth (using Carson's rule) required for the transmission of the FSK signal is given by

$$BW = 2\Delta f + 2R_b$$

Experimental procedures

In order to generate an ASK signal follow these steps

1. Set the sampling frequency to $f_s = 32kHz$ and the carrier frequency $f_c = 4000Hz$
2. Set the number of samples to 16 samples
3. Use the following code to generate an ASK modulated signal.

```
#include<stdio.h>                    //for input/output display
#include"DSK6713_AIC23.h"     // this file is added to initialize the DSK6713
#include "dsk6713.h"
#include "math.h"                    // header file used when mathematical instructions are
executed
#define N 16                         // no. of samples
Uint32 fs = DSK6713_AIC23_FREQ_32KHZ;//support file for codec,DSK
int cnt=0,data[8]={0},k;
short i,j,sine_table[N],cos_table[N];
float pi = 3.14159;                                  // variable declaration
```

```
void main()
{


        DSK6713_init(); // Initialize the board support library, must be called first

comm_poll();
                                // Set the codec sample rate frequency

        printf("Enter the Binary Elements of Sequence (0 or 1)\n");         //prints the line on CCS
window
        for(k=0;k<=7;k++)
        {
                scanf("%u",&data[k]);     //ask the user to enter 8 input binary digits
                printf("Entered values are \t\t\t%u\n",data[k]);  //displays it on CCS window
        }

                        for(i = 0;i<N;i++)
        // write the sample values of waveform  at every sampling instant
                        {
                                sine_table[i] = 10000*sin((2.0*pi*i/32000)*4000);          //
generation of sine-wave signal using formula, value is taken in a loop.
                                if(i>N) i = 0;
                        }

        while(1)
        //program execution halts and it starts listening for the interrupt which occur at every
sampling period Ts.
        {
                if(cnt<=7)
                {
                        if(data[cnt]==1)                        //if input is 1
                        {
                                for(j=0;j<N;j++)
                                {
                                output_sample(sine_table[j]);
                                }
                        }
                        else for(j=0;j<N;j++)
                        {
```

```
                        output_sample(0);

                    }

                    if(j>=N)j=0;
                    cnt++;
                    if(cnt>7) cnt=0;
                }
            }
        }
```

Question 1. What is the number of cycles per bit?

Question 2. What is the bit rate of the above transmitted stream?

Question 3. What is the bandwidth required for the transmission of this ASK signal

4. Modify the code in you experiment by replacing `output_sample(0);` by `output_sample(-1*sine_table[j]);` and explain what is the modulation scheme is shown on the screen of the oscilloscope
5. In order to demodulate the ASK signal the modulated signal is multiplied by the carrier again then passed through a LPF. The output of the low pass filter is further processed by a decision device. In this part of the experiment connect the LINE OUT of your DSK6713 to LINE IN of DSK6713 of your colleague's kit. Setup the design shown in Figure 3.3 by using simulink in the receiving DSK6713 kit. And verify that this circuit demodulates the ASK signal

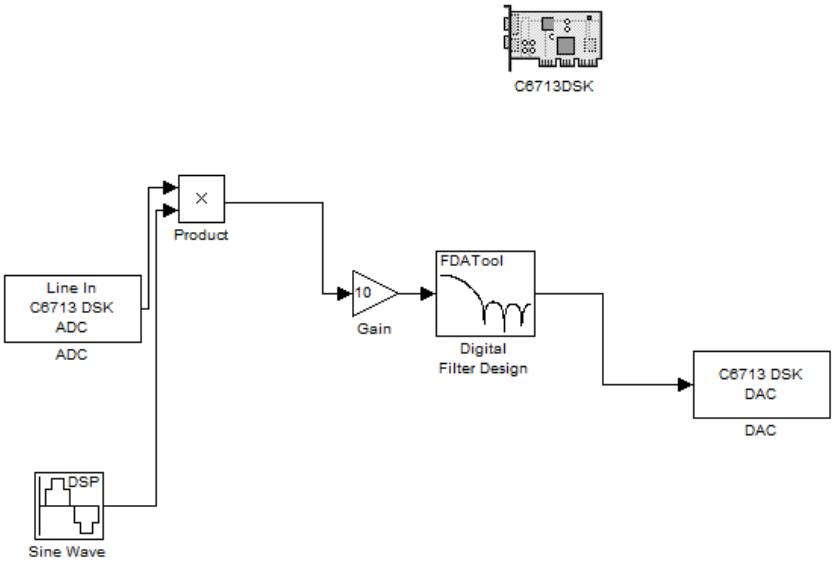


Figure 3.3 Ask demodulator block in Simulink

Note:

a- The sampling frequency must be 32kHz (as in modulator), samples/frame=1.

b- Set the amplitude of the carrier to 0.5 and the frequency to 4000.

c- Cut off frequency of the low pass filter =2 kHz, FIR filter with the setting illustrated in Figure 3.4.

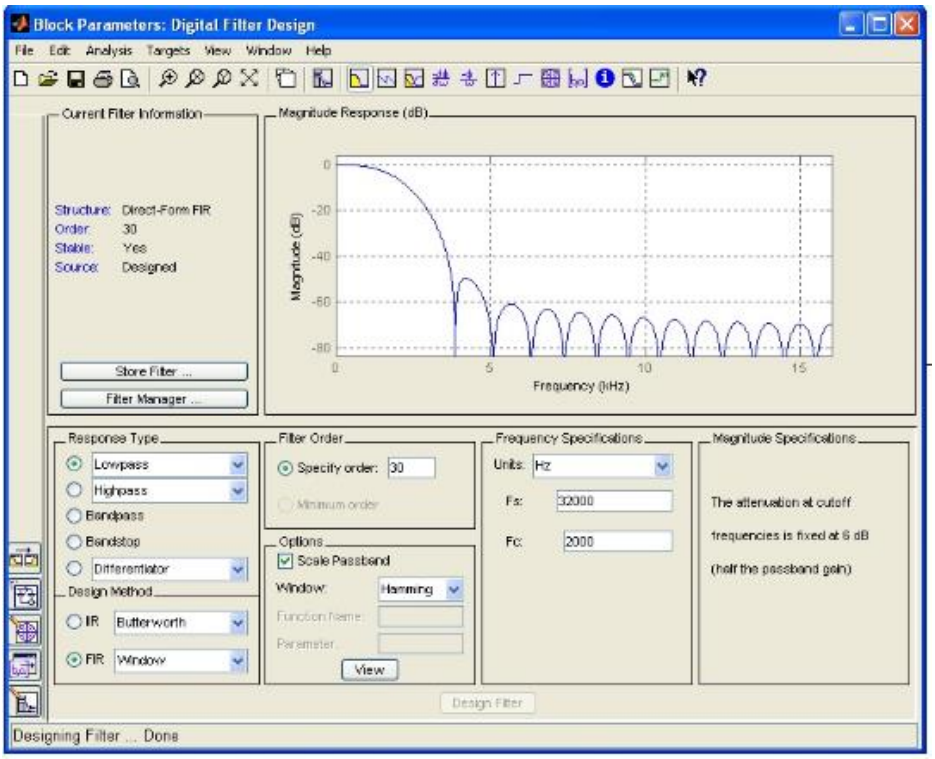


Figure 3.4 Filter Design settings

Question 4. Explain the problem of synchronization between transmitter and receiver as appeared in your results.

6. To avoid this problem, we can multiply The ASK signal by itself then pass it to low pass filter.

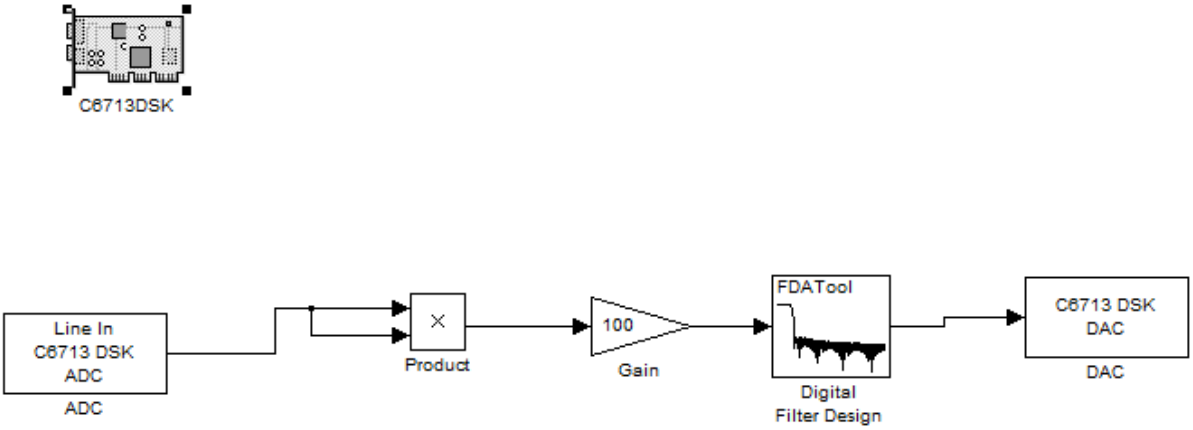


Figure 3.5 Ask demodulator modified block in Simulink

Note: Change the order of the low pass filter to 50.

Question 5. What is the name of this type of demodulation?

7. Modify the above code in order to generate an FSK signal
8. In order to demodulate the FSK signal, we can use synchronous demodulation where the FSK signal is multiplied by f_1 and f_2 then passed to low pass filter followed by a multiplexer.

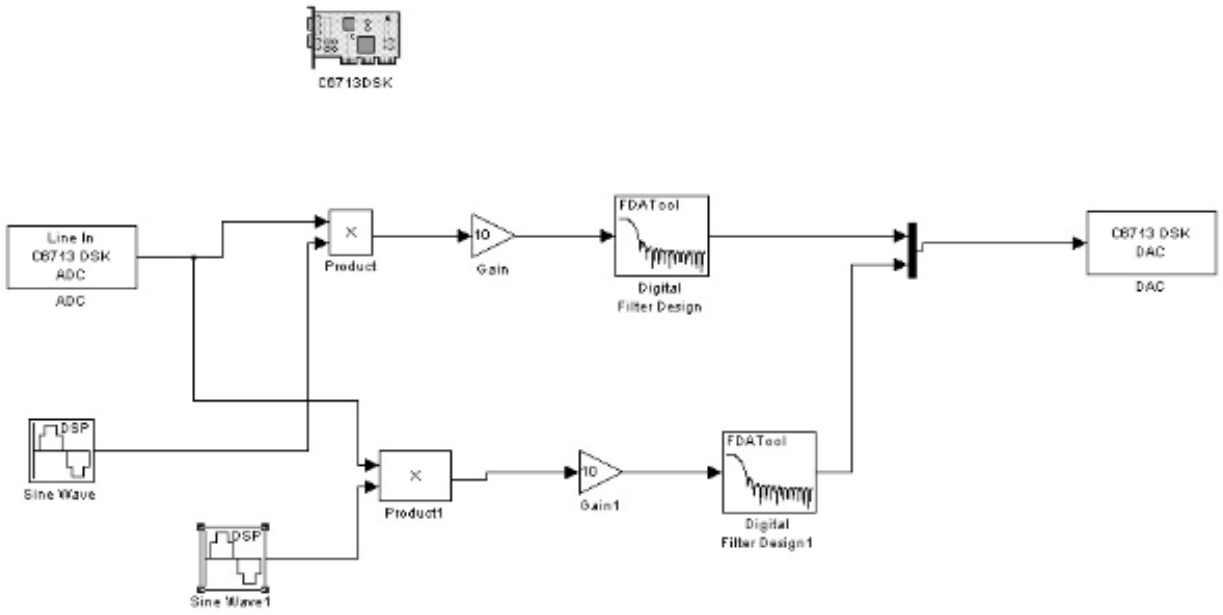


Figure 3.6 FSK demodulator block in Simulink

9. To avoid synchronization problem between transmitter and receiver, we can convert FSK signal to ASK signal by using band stop filter (notch filter) to eliminate either f_1 or f_2 , then multiply the resulting ASK signal by itself and pass it to low pass filter.

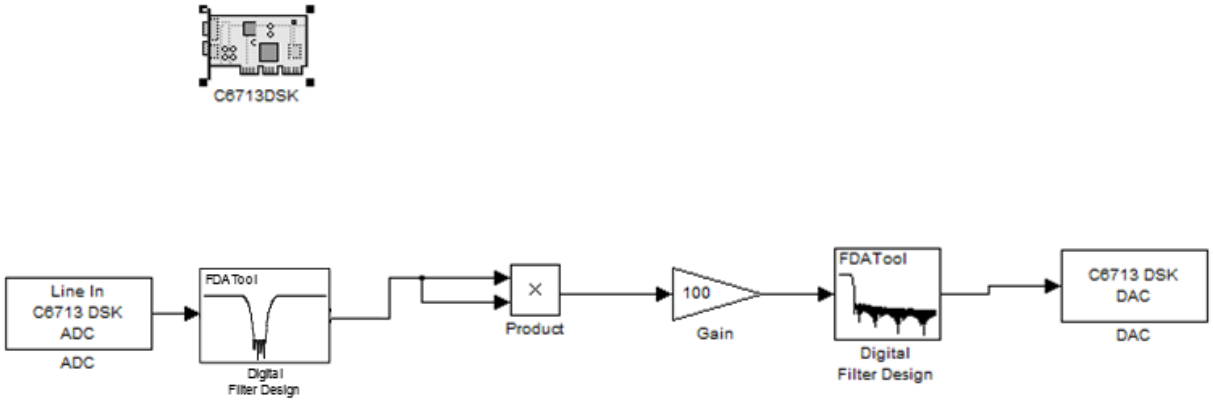


Figure 3.7 FSK demodulator block- method 2

Note: Set the order of the band stop filter to 20 and cut off frequencies to 7500 Hz& 8500 Hz.Set the order of the low pass filter to 20 and cut off frequency to 2 KHz.

Question 6. Suppose that the carrier frequency used to generate logic 1 is 4 kHz and that used to generate logic zero is 8 kHz, what would be the bandwidth required for this FSK signal? Is there any special name given this FSK modulation scheme?

4 Experiment 4 QPSK

Objectives

The objectives of this experiment is to show students how to generate QPSK signal using DSP processors

Theory

Quadrature Phase Shift Keying (QPSK) can be interpreted as two independent BPSK systems (one on the I-channel and one on Q-channel). Therefore it has twice the bandwidth efficiency compared with BPSK. Quadrature Phase Shift Keying has twice the bandwidth efficiency of BPSK since two bits are transmitted in a single modulation symbol.

In QPSK there are large envelope variations occur due to abrupt phase transitions, thus QPSK signal requires linear amplification in the receiver.

QPSK uses four points on the constellation diagram, equally spaced around a circle as shown in Figure 35with ith four phases, QPSK can encode two bits per symbol, shown in the diagram with gray coding to minimize the bit error rate (BER) sometimes misperceived as twice the BER of BPSK.

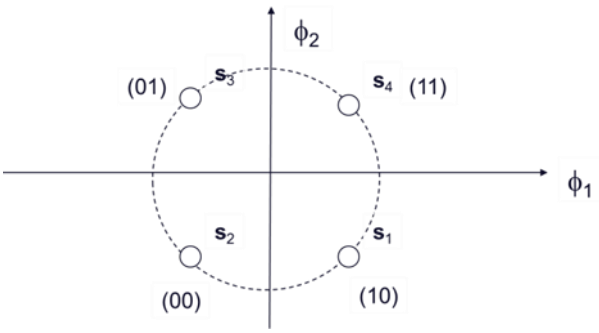


Figure 4.1 Constellation diagram of QPSK

The mathematical analysis shows that QPSK can be used either to double the data rate compared with a BPSK system while maintaining the same bandwidth of the signal, or to maintain the data-rate of BPSK but halving the bandwidth needed. In this latter case, the BER of QPSK is exactly the same as the BER of BPSK and deciding differently is a common confusion when considering or describing QPSK.

Given that radio communication channels are allocated by agencies such as the Federal Communication Commission giving a prescribed (maximum) bandwidth, the advantage of QPSK over BPSK becomes evident: QPSK transmits twice the data rate in a given bandwidth compared to BPSK at the same BER. The engineering penalty that is paid is that QPSK transmitters and receivers are more complicated than the ones for BPSK. However, with modern electronics technology, the penalty in cost is very moderate.

4.1.1 Implementation

The implementation of QPSK is more general than that of BPSK and also indicates the implementation of M-ary PSK. QPSK is described mathematically by

$$s_i(t) = \sqrt{\left(\frac{2E_s}{T_s}\right)}\cos\left(2\pi f_c t + \frac{2(i-1)\pi}{4}\right), \qquad i = 1, 2, 3, 4$$

This yields the four phases of the carrier $\frac{\pi}{4}$, $\frac{3\pi}{4}$, $\frac{5\pi}{4}$ and $\frac{7\pi}{4}$. The previous equation can be further simplified by expanding the cosine term using trigonometric identities as shown below

$$s_i(t) = \sqrt{E_s}\cos\left((2i-1)\frac{\pi}{4}\right)\sqrt{\frac{2}{T_s}}\cos(2\pi f_c t) - \sqrt{E_s}\sin\left((2i-1)\frac{\pi}{4}\right)\sqrt{\frac{2}{T_s}}\sin(2\pi f_c t)$$

This results in a two-dimensional signal space diagram with unit basis functions. The first basis function; $\phi_1(t) = \sqrt{\frac{2}{T_s}}\cos(2\pi f_c t)$; is used as the in-phase component of the signal and the second basis function; $\phi_2(t) = \sqrt{\frac{2}{T_s}}\sin(2\pi f_c t)$; as the quadrature component of the signal.

Hence, the signal constellation consists of the signal-space 4 points. The four symbol points in vector form are given by

$$s_i = \begin{bmatrix} \sqrt{E_s}\cos\left((2i-1)\frac{\pi}{4}\right) \\ -\sqrt{E_s}\sin\left((2i-1)\frac{\pi}{4}\right) \end{bmatrix}$$

4.1.2 Generation and demodulation of QPSK signal

The block diagram for generating QPSK is shown in Figure 4.2, while the block diagram for demodulating QPSK signal is shown in Figure 4.3 respectively.

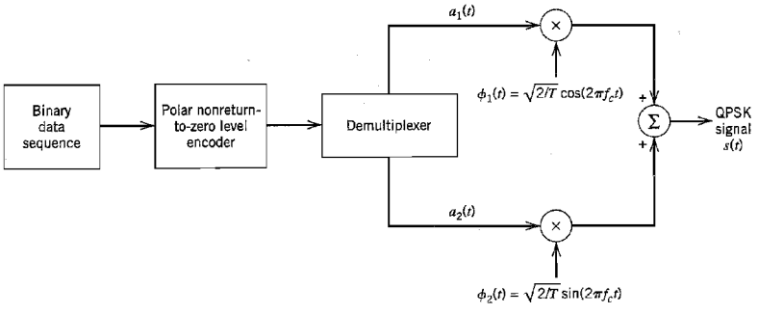


Figure 4.2 Block diagram for generating QPSK signal.

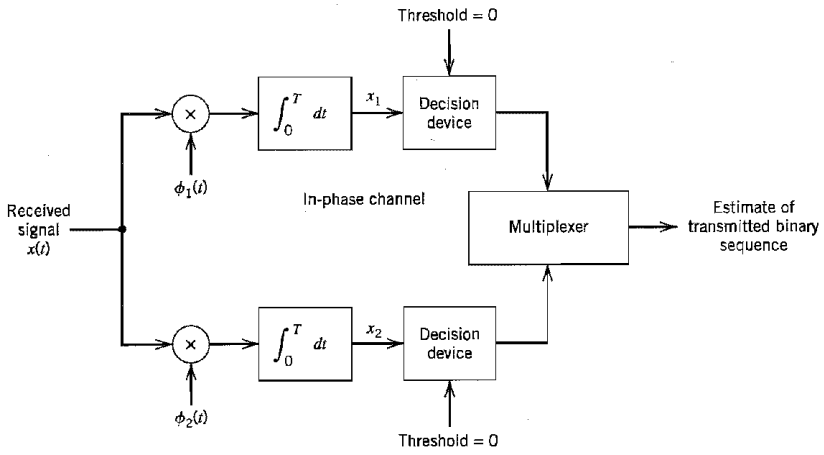


Figure 4.3 Block diagram for QPSK demodulator.

In this experiment both modulator and demodulator for QPSK will be implemented by using the DSK6713 starter kit.

QPSK signal can be generated in the code by using the following procedure

1. Initialize temporary variables to hold a sample values from both the in-phase and quadrature phase components.
2. If the first bit in the data sequence is one multiply the in-phase carrier by one and add the result to the temporary in-phase component defined in step 1 else multiply the carrier by -1 and add the result to the temporary in-phase component defined in step 1.
3. If the second bit in the sequence is 1 multiply the quadrature- phase carrier by one and add the result to the temporary quadrature-phase component else multiply the quadri-phase carrier by -1 and add the result to the temporary quadrature-phase.
4. Add the in-phase and quadrature-phase components, then send the result to the DAC output using output_sample ( ) function.
5. Repeat step 4 until the 16th carrier samples are sent.
6. Reset the sample counter to zero.
7. Get the next two bits from the data buffer and repeat steps 1-5 again.
8. Repeat step 7 until all the data bits are being transmitted.

Experimental procedures

In order to generate a QPSK signal follow these steps

9. Set the sampling frequency to $\boldsymbol{f_s = 32kHz}$ and the carrier frequency $\boldsymbol{f_c = 4000Hz}$
10. Set the number of samples to 16 samples
11. Use the following code to generate an QPSK modulated signal.

```
#include<stdio.h>                    //for input/output display
#include"DSK6713_AIC23.h"            // this file is added to initialize the DSK6713
#include "dsk6713.h"
```

```
#include "math.h"                       // header file used when mathematical instructions are
                                        // executed
#define N 16                            // no. of samples
uint32 fs = DSK6713_AIC23_FREQ_32KHZ;           // Set the codec sample rate frequency
int cnt=0,data[8]={0},k;
short i,j,sine_table[N],cos_table[N];
float pi = 3.14159;                             // variable declaration

void main()
{
        DSK6713_init();         // Initialize the board support library, must be called first
        comm_poll();            //support file for codec,DSK
        printf("Enter the Binary Elements of Sequence (0 or 1)\n");     //prints the line on CCS
window
        for(k=0;k<=7;k++)
        {
                scanf("%u",&data[k]);   //ask the user to enter 8 input binary digits
                printf("Entered values are \t\t\t%u\n",data[k]);  //displays it on CCS window
        }

                        for(i = 0;i<N;i++)
        // write the sample values of waveform  at every sampling instant
                        {
                        sine_table[i] = 10000*sin((2.0*pi*i/32000)*4000);       // generation of
                        sine-wave signal using formula, value is taken in a loop.
                                if(i>N) i = 0;
                                cos_table[i] = 10000*cos((2.0*pi*i/32000)*4000);        //
                        generation of cos-wave signal using formula, value is taken in a loop.
                        }

        while(1)
        {


Write your QPSK modulator code here

                        if(j>=N)
                        {
                        j=0;
                        cnt++;
                        }
```

```
                if(cnt>7) cnt=0;
            }                                   // ends the while loop
    }                                           // ends the main function
```

Question 5. What is the bit rate of the above transmitted stream?

Question 6. What is the bandwidth required for the transmission of this QPSK signal

12. In order to draw QPSK constellation diagram, send the first bit of each symbol to the DAC right channel and the second bit to the DAC left channel , taking into account that the data must be changed from unipolar to bipolar NRZ format.
13. Use the following code to generate the constellation diagram:

```
// constellation.c
#include<stdio.h>                       //for input/output display
#include"DSK6713_AIC23.h"               // this file is added to initialize the DSK6713
#include "dsk6713.h"
#include "math.h"       // header file used when mathematical instructions are //
executed
#define N 16                            // no. of samples
Uint32 fs = DSK6713_AIC23_FREQ_32KHZ;               // Set the codec sample rate
frequency

#define LEFT 0
#define RIGHT 1
union {Uint32 combo; short channel[2];} AIC23_data;


short gain=10000;
int cnt=0,data[8]={0},k;

void main()
{
        DSK6713_init();         // Initialize the board support library, must be called
first
        comm_poll();            //support file for codec,DSK
        printf("Enter the Binary Elements of Sequence (0 or 1)\n");     //prints the line
on CCS window
        for(k=0;k<=7;k++)
        {
                scanf("%u",&data[k]);   //ask the user to enter 8 input binary digits
                printf("Entered values are \t\t\t%u\n",data[k]);  //displays it on CCS
window
```

```
}



while(1){


    if(data[cnt]==0){
        if(data[cnt+1]==0){

    AIC23_data.channel[RIGHT]=-1*gain; //for right channel
    AIC23_data.channel[LEFT]=-1*gain;
     for(i = 0;i<N;i++)                                                    // write
the sample values of waveform  at every sampling instant
        {
      output_sample(AIC23_data.combo);

        }

                cnt=cnt+2;
        }
        else{
        AIC23_data.channel[RIGHT]=-1*gain; //for right channel
   AIC23_data.channel[LEFT]=1*gain;

    for(i = 0;i<N;i++)                                                    // write
the sample values of waveform  at every sampling instant
        {
      output_sample(AIC23_data.combo);

        }

cnt=cnt+2;
        }
    }
    else{
        if(data[cnt+1]==0){

    AIC23_data.channel[RIGHT]=1*gain; //for right channel
    AIC23_data.channel[LEFT]=-1*gain;
```

```
                for(i = 0;i<N;i++)                                          // write
the sample values of waveform  at every sampling instant
                    {
                output_sample(AIC23_data.combo);

                    }

            cnt=cnt+2;
                    }
                    else{
                    AIC23_data.channel[RIGHT]=1*gain; //for right channel
             AIC23_data.channel[LEFT]=1*gain;
             for(i = 0;i<N;i++)                                             // write
the sample values of waveform  at every sampling instant
                    {
                output_sample(AIC23_data.combo);

                    }

                    cnt=cnt+2;
                    }
                }

          if(cnt>7) cnt=0;
        }           // ends the while loop
         }                      // ends the main function
```

14. Connect line out right channel of the DSK to channel A of the picoscope, and connect line out left channel to channel B of the picoscope.
15. Run the code and enter the following data: (11100001).
16. On your picoscope screen, choose View-----XY Mode.
17. Explain your results.

18. In order to demodulate the QPSK signal the modulated signal is multiplied by the carrier again then passed through a LPF. The output of the low pass filter is further processed by a decision device. In this part of the experiment connect the LINE OUT of your DSK6713 to LINE IN of DSK6713 of your colleague's kit. Setup the design shown in Figure 4.4by using simulink in the receiving DSK6713 kit. Verify that this circuit demodulates the QPSK signal

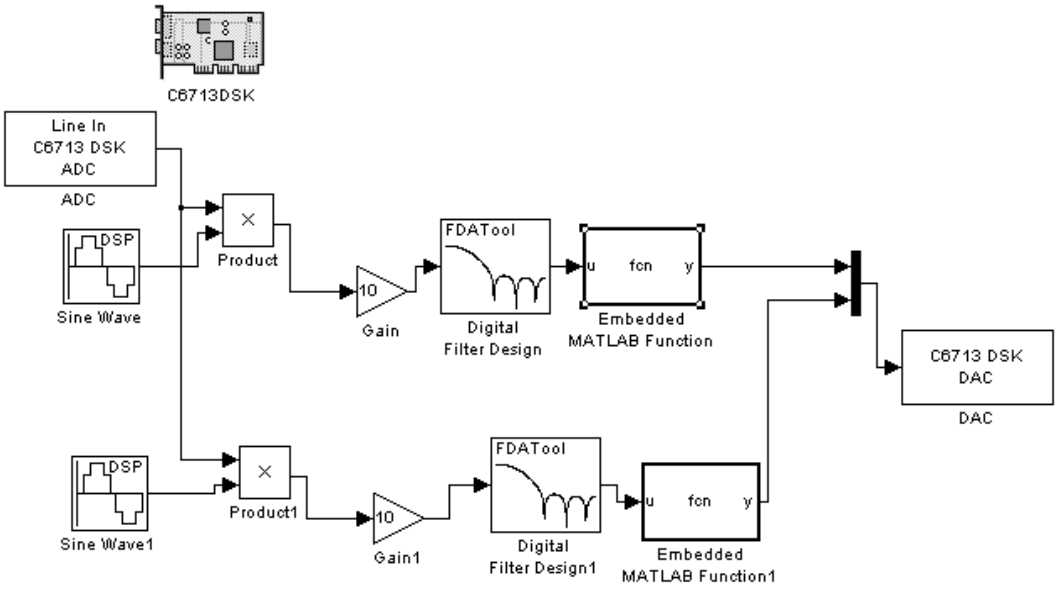


Figure 4.4 QPSK demodulator block in Simulink

Note that sampling frequency must be 32 kHz (as in modulator), samples/frame=1, cut off frequency of the low pass filter =2 kHz. Embedded MATLAB Function is used as comparator device , you can write the following code inside:

```
function y = fcn(u)
% This block supports the Embedded MATLAB subset.
% See the help menu for details.
if(u>=0)
y = 1;
else
    y=-1;
end
```

Figure 4.5Implementing of a comparator using Embedded MATLAB Function

Question 7. Explain in your report how to achieve synchronization between the carriers in both the transmitter and receiver.

5 Experiment 5 FIR filter

Objectives

The aim of this experiment is to design and implement several FIR filters

Theory

Filters are essential components in many communication systems. Filters can be used in samplers and down samplers as an anti-alias filters. Also filters can be used in modulators and demodulators, reconstruction filters in in receivers. Also filters can be used in some control systems and in almost every signal processing components.

Filters can be classified as either an analogue or digital filters. In this experiment we will be interested in digital filters only.

Digital filters can also be classified as either a Finite Impulse Response (FIR) or an Infinite Impulse Response (IIR) filters.

A finite impulse response (FIR) filter is a type of a signal processing filter whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time. However the response time for an IIR filter will extends to an infinite.

A comparison between FIR and IIR filters show that IIR requires less coefficient; hence faster computation of the filtered signal; compared with FIR filters. However IIR filters have a non-linear phase distortion whereas FIR filters has a linear phase response.

In this experiment only FIR filters will be considered while IIR filters will be discussed in more details in the next experiment.

To design an FIR filter, it is necessary to compute the filter coefficients by either windowing or using one the various algorithms presented by digital signal processing course. However the calculation of the filter coefficients is greatly simplified by using commercial software such as MATLAB and digifilter.

Recall that FIR filters are described by the system function defined by

$$\frac{Y(z)}{X(z)} = \sum_{k=0}^{N-1} h[k] z^{-k}$$

Where $h[k]$ are the filter coefficients to be determined and N is the number of the filter coefficients.

FIR filters can be implemented in hardware by using the structure shown in Figure 5.1.

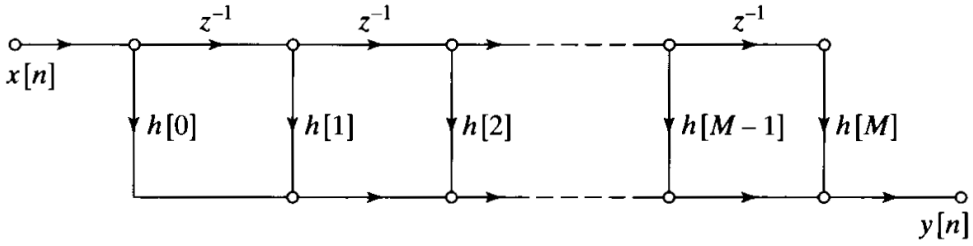


Figure 5.1 Structure for FIR filter

Note that filters are considered linear time invariant system. This means that if an input signal $x_i[n]$is passed through the input terminal of a given filter then the output of that filter is given by the convolution sum described by the general difference equation which is given by

$$\boldsymbol{y[n] = \sum_{k=0}^{N-1} h(k)x[n-k]}$$

We can arrange the impulse response coefficients within a buffer (array) so that the first coefficient, $h(0)$, is at the beginning (first location) of the buffer (lower- memory address). The last coefficient, $h(N-1)$, can reside at the end (last location) of the coefficients buffer (higher-memory address). The delay samples are organized in memory so that the newest sample, $x(n)$, is at the beginning of the samples buffer, while the oldest sample, $x(n-(N-1))$, is at the end of the buffer. The coefficients and the samples can be arranged in memory as shown in Table 1. Initially, all the samples are set to zero.

i	Coefficients	Samples
0	h(0)	x(n)
1	h(1)	x(n - 1)
2	h(2)	x(n - 2)
.	.	.
.	.	.
.	.	.
N-1	h(N - 1)	x(n - (N - 1))

Table 1.Memory Organization for Coefficients and Samples (Initially)

Experimental procedures

In this experiment we will implement and test low pass filter, high pass filter, bandpass filter and band stop (notch) filter as illustrated by the following exercises.

5.1.1 Low pass filter

In this experiment you will design a low pass filter whose cutoff frequency is $f_c = 2400Hz$. In order to design the filters follow these steps

1. In MATLAB open the filter design tool by typing the fdatool in the MATLAB command window. A design window such as the one shown in Figure 5.2 will appear on the screen.

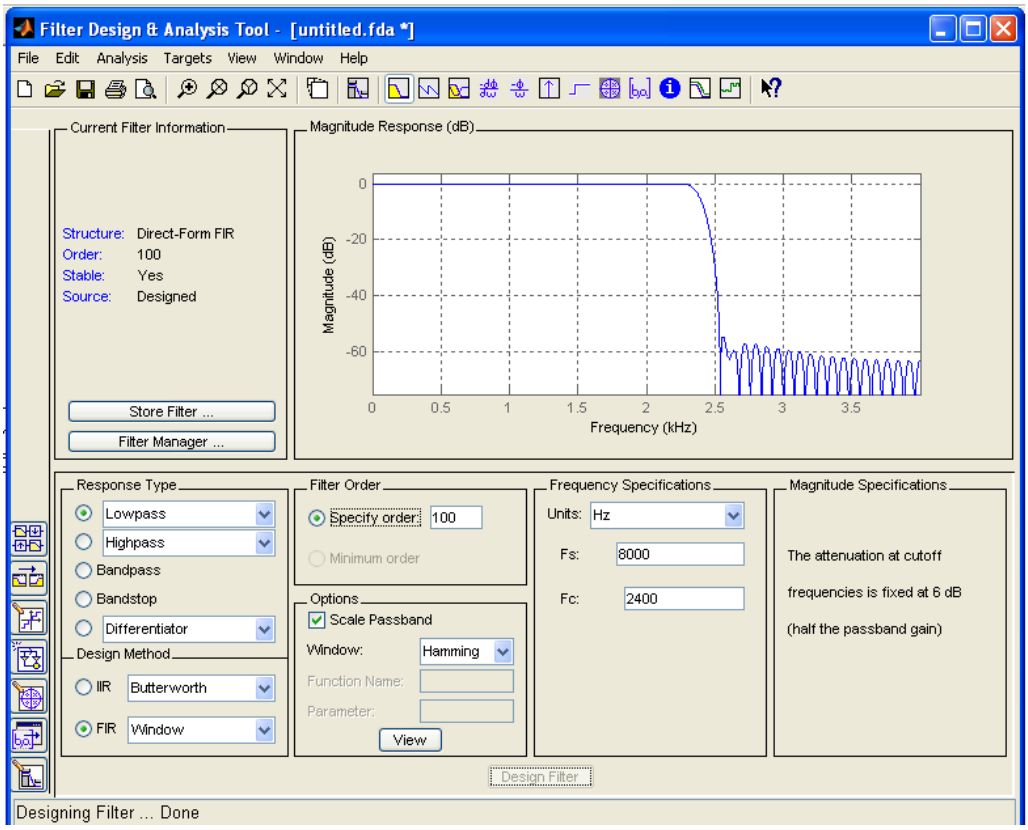


Figure 5.2 Filter design tool in Matlab

2. Set the response type to Low pass filter, the design method to FIR window, filter order 100, window Hamming, sampling frequency to $f_s = 8kHz$ and the carrier frequency $f_c = 2400Hz$
3. Press on the design button to design the filter.
4. In order to obtain the filter coefficients go to the file menu then select export. A small dialog box such as the one shown in Figure 5.3 will appear. Select export to Workspace and export as Coeffiecients and press on the export button. Name your Variable as “lp2400” and it will be saved under your project directory.

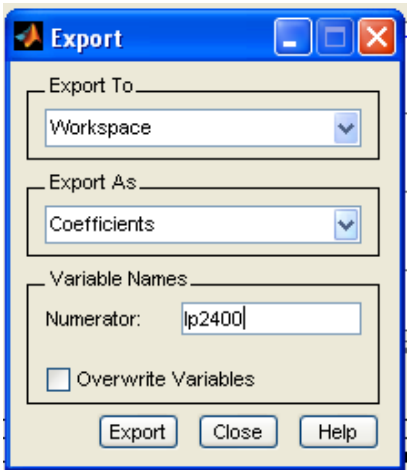


Figure 5.3 Export dialog in MATLAB

5. Create the filter coefficient file by running the matlab code below:

```
fid=fopen('lpf.cof','w');
fprintf(fid,'#define N 101 \n');
fprintf(fid,'float h[N]={');
fprintf(fid,'%f, ' ,lp2400(1:100));
fprintf(fid,'%f' ,lp2400(101));
fprintf(fid,'};\n');
fclose(fid);
```

This Matlab code will create lpf.cof file with the following format:

```
#define N 101 //number of coefficiets
float h[N]={ the coeffients of the filter here}
```

Copy this file to your project folder

6. To implement the filter on the DSK6713 kit open the existing project that you worked on last time and add the following c code to it.

```
#include "lpf.cof"                          //coefficient file

#include "dsk6713_aic23.h"                  //codec-dsk support file

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;          //set sampling rate

signed int yn = 0;                                  //initialize filter's output

short dly[N];                                       //delay samples
```

```
interrupt void c_int11()                    //ISR
{
short i;
dly[0]=input_sample();                      //input newest sample
yn = 0;                                     //initialize filter's output
for (i = 0; i< N; i++)
yn += (h[i] * dly[i]);                      //y(n) += h(i)* x(n-i)
for (i = N-1; i > 0; i--)                   //starting @ end of buffer
dly[i] = dly[i-1];                          //update delays with data move
output_sample(yn);                          //output filter sample
return;
}
void main()
{
comm_intr();                        //init DSK, codec, McBSP
while(1);                           //infinite loop
}
```

7. After you build the project, load it on the DSK6713 kit, then run the program.
8. To test the designed LPF filter, connect a sine wave from the function generator to the LINE IN terminal of the DSK6713 kit.
9. Set the amplitude of the sign wave to $\mathbf{1}\boldsymbol{V_{pp}}$ and vary the frequency of the sine wave from 50 Hz to 3.4 kHz in steps of 200 Hz. Measure the amplitude of the output wave at every frequency point and tabulate your results as shown in Table

freq	50	400	600	1000	1400	1600	1800	2000	2200	2400	2600	3000	3200	3400
Am														

Table 2

Plot the amplitude versus frequency and comment on the results.

10. An alternative way to test your filter is to apply an input random signal, then connect the filter output to the spectrum analyzer. This can be achieved by generated the random signal through the MATLAB Simulink as shown in Figure 5.4

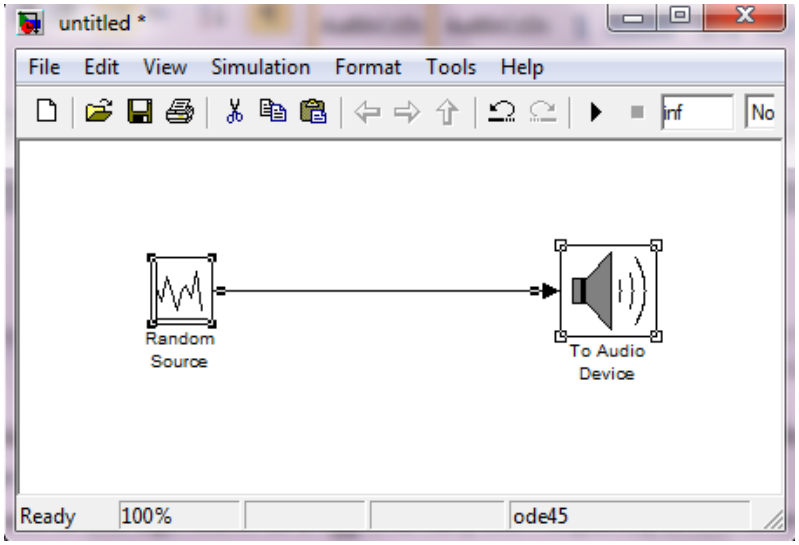


Figure 5.4 Random noise generator in MATLAB

11. Set the sample time for the noise to $\mathbf{1/8000}$ and it type is Gaussian.
12. Connect the output of the DSK kit to your pico-scope and adjust the scope for spectrum operation
13. Sketch the frequency response of the filter as it appears on the screen of the spectrum analyzer

5.1.2 Low pass, high pass, band pass and pass band filter

In this exercise we will design four filters using the same procedure used in the previous section. The student is supposed to design the four filter types in MATLAB then export the filters coefficients to a different file depending on the filter type.

A slider can be used to select which filter to use as illustrated by the following c source file. Select the cutoff frequency of the low pass filter as $\boldsymbol{f_c = 1500Hz}$, the cut off frequency of the High pass filter as $\boldsymbol{f_c = 2200Hz}$, the center frequency of the bandpass filter as $\boldsymbol{f_c = 1750Hz}$, and the band stop of the band stop filter as $\boldsymbol{f_{bs} = 790Hz}$.

```
#include "DSK6713_AIC23.h"            //this file is added to initialize the DSK6713

#include "lowp1500.cof"                    // coefficient of low-pass filter file calculated from MATLAB


#include "highp2200.cof"              // coefficient of high-pass filter file calculated from MATLAB

#include "bpass1750.cof"              // coefficient of band-pass filter file calculated from MATLAB
```

```
#include "bstop790.cof"                    // coefficient of band-stop filter file calculated from MATLAB

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; // set the sampling frequency, Different sampling frequencies supported by
AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96 kHz.


short FIR_number = 0;   //filter number

signed int yn = 0;                                  //variable declaration

short dly[N];                       //declare delay buffer of n values

float h[4][N];                      //co-efficients of 4 different filters


interrupt void c_int11() // ISR call, At each Interrupt, program execution goes to the interrupt service routine


{
        short i;                                                    //variable declaration

        dly[0] = input_sample();                    //newest input @ top of buffer

        yn = 0;                     //initialize filter output

        for (i = 0; i< N; i++)                                  //for loop takes in the value of i from 0 to N

        yn +=(h[FIR_number][i]*dly[i]);  //y(n) += h(LP#,i)*x(n-i)

        for (i = N-1; i > 0; i--)               //starting @ bottom of buffer

        dly[i] = dly[i-1];                  //update delays with data move

        output_sample(yn);                          //output filter,the value in the buffer yn indexed by the variable
loop is written on to the codec.

        return;                                                     // program execution goes back to while(1)
and then again starts listening for next interrupt and this process goes on

}

void main()

{
```

```
        short i;                                        //variable declaration
        for (i=0; i<N; i++)                             //for loop which takes in the value of i from 0 to
 N=4 and switches to corresponding filter co-efficients
        {
                dly[i] = 0;                     //init buffer
                h[0][i] = hlp[i];                       //start addr of lp1500 coeff
                h[1][i] = hhp[i];                       //start addr of hp2200 coeff
                h[2][i] = hbp[i];                       //start addr of bp1750 coeff
                h[3][i] = hbs[i];                       //start addr of bs790 coeff
        }
        comm_intr();                    // ISR function is called, using the given command
        while(1);                                       //program execution halts and it starts listening for
 the interrupt which occur at every sampling period Ts.
 }
```

You need also to add the following GEL file to your project

```
/*FIR4types.gel Gel file for 4 different filters: LP,HP,BP,BS*/


menuitem "Filter Characteristics"


slider Filter(0,3,1,1,filterparameter) /*from 0 to 3,incr by 1*/
{
        FIR_number = filterparameter;    /*for 4 FIR filters*/
}
```

1. To test the designed filters, connect a sine wave from the function generator to the LINE IN terminal of the DSK6713 kit. Set the amplitude of the sign wave to $\mathbf{1}\boldsymbol{V_{pp}}$ and vary the frequency of the sine wave to check the functionality of your designed filters.
2. Also to test your filters, apply an input random signal, and then connect the filter output to the spectrum analyzer. Check the frequency response of the 4 types.

5.1.3 Effects on voice using three FIR low pass filters (FIR3LP)

In this exercise you will implement three FIR low pass filters with cutoff frequencies at 600, 1500, and 3000Hz, respectively. The three low pass filters were designed with MATLAB's fdatool to yield the corresponding three sets of coefficients.

If for example LP_number is set to 0, $\boldsymbol{h}[\mathbf{0}][\boldsymbol{i}]$ is equal to hlp600[i] (within the "for" loop in the function main), which is the address of the first set of coefficients. The coefficients file LP600.cof represents an 81-coefficient FIR low pass filter with a 600-Hz cutoff frequency, using the Kaiser window function. your program may appear as the one shown below. LP_number can be changed to 1 or 2 to implement the 1500 Hz or 3000 Hz low pass filter, respectively.

```
//Fir3LP.c FIR using 3 low pass coefficients with three 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

short LP_number = 0; //start with 1st LP filter

signed int yn = 0; //initialize filter's output

short dly[N]; //delay samples

float h[3][N]; //filter characteristics 3xN

interrupt void c_int11() //ISR

{

short i;

dly[0] = input_sample(); //newest input @ top of buffer

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--) //starting @ bottom of buffer
```

```
dly[i] = dly[i-1]; //update delays with data move

output_sample(yn ); //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 addr of LP600 coeff

h[1][i] = hlp1500[i]; //start addr of LP1500 coeff

h[2][i] = hlp3000[i]; //start addr of LP3000 coeff

}


comm_intr(); //init DSK, codec, McBSP

while(1); //infinite loop

}
```

1. With the GEL file FIR3LP.gel, one can vary LP_number from 0 to 2 and slide through the three different filters. Build this project as FIR3LP.
2. Use a .wav file from your PC or test your voice through a mic as input and observe the effects of the three low pass filters on the input voice. Connect the LINE OUT of the DSK6713 to a speaker and observe what happens when the different filters are used

Your GEL file may appear as shown below

```
/*FIR3LP.gel Gel file to step through three different LP filters*/
menuitem "Filter Characteristics"
slider Filter(0,2,1,1,filterparameter) /*from 0 to 2,incr by 1*/
{
LP_number = filterparameter; /*for 3 LP filters*/
}
```

6 Experiment 6 IIR filter

Objectives

a) The aim of this experiment is to design and implement different IIR filters type
b) Observe the effect of the nonlinear phase of IIR filter

Theory

Infinite impulse response (IIR) filters are described by the general difference equation which is given by

$$y[n] = \sum_{k=0}^{N-1} a_k x[n-k] - \sum_{j=1}^{M-1} b_j y[n-j]$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N-1} a_k z^{-k}}{\sum_{j=1}^{M-1} b_j z^{-j}} = \frac{N(z)}{D(z)}$$

This difference equation can be solved recursively to produce the output samples from the input samples.

The design of an IIR filter require the computation of the filter coefficients $a_k$and b_k. These filter coefficients can be determined either by using the impulse invariance method or by using the bilinear transformation design techniques presented by the DSP course. However the calculation of the filter coefficients is greatly simplified by using commercial software such as MATLAB or digifilter.

IIR filters can be implemented by using either direct form I or direct form II structure as illustrated in Figure 6.1.

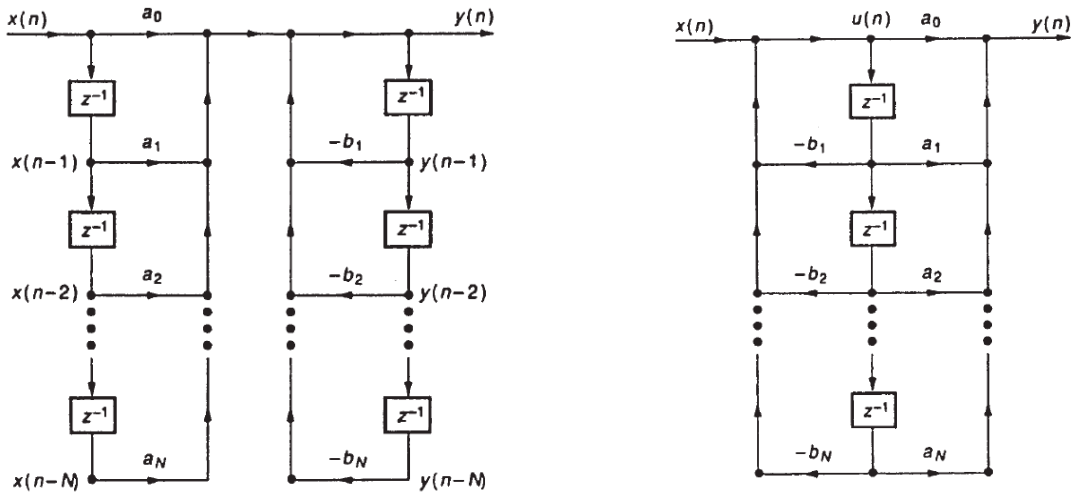


Figure 6.1 (a) direct form I IIR structure. (b) Direct form II IIR structure.

In Direct form I:

$$y(n) = a_0x(n) + a_1x(n-1) + \ldots\ldots + a_Nx(n-N) - b_1y(n-1) - b_2y(n-2) - \ldots\ldots - b_My(n-M)$$

$$If\ M = N, then\ number\ of\ delay\ elements = 2N$$

In Direct form II:

$$Y(z) = \frac{N(z)X(z)}{D(z)}$$

$$If\ U(z) = \frac{X(z)}{D(z)}, then\ Y(z) = N(z)U(z)$$

$$X(z) = D(z)U(z) \quad \rightarrow \quad x(n) = u(n) + b_1u(n-1) + \ldots\ldots + b_Mu(n-M)$$

As a result,

$$u(n) = x(n) - b_1u(n-1) - \ldots\ldots + b_Mu(n-M)$$

$$y(n) = a_0u(n) + a_1u(n-1) + \ldots\ldots + a_Nu(n-N)$$

So,

$$If\ M = N, then\ number\ of\ delay\ elements = N$$

Direct form II requires almost half the delay elements compared with direct form I, therefore direct form II is the commonly used structure used to implement IIR filters.

The transfer function of the filter can be implemented by using structures other than direct form I and direct form II structures. Some of these structures are the transposed form structure, cascade form structure, parallel form structure and the lattice structure.

In this experiment only cascade form structure and parallel form structure will be considered.

6.1.1 Cascade form structure

In the cascade form structure the transfer function of the IIR filter can be factored into first order or second order terms as expressed mathematically by

$$\boldsymbol{H(z) = CH_1(z)H_2(z) \ldots . H\_r(z)}$$

The cascade (series) connection of the first or second order sections is illustrated by the block diagram shown in Figure 6.2.

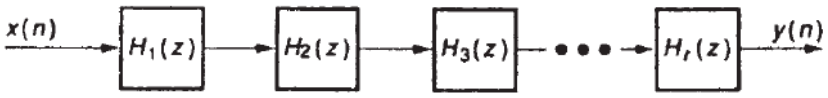


Figure 6.2 Cascade form IIR filter structure.

For example, a fourth-order IIR structure can be implemented in terms of two second-order sections in cascade is shown in Figure 6.3.

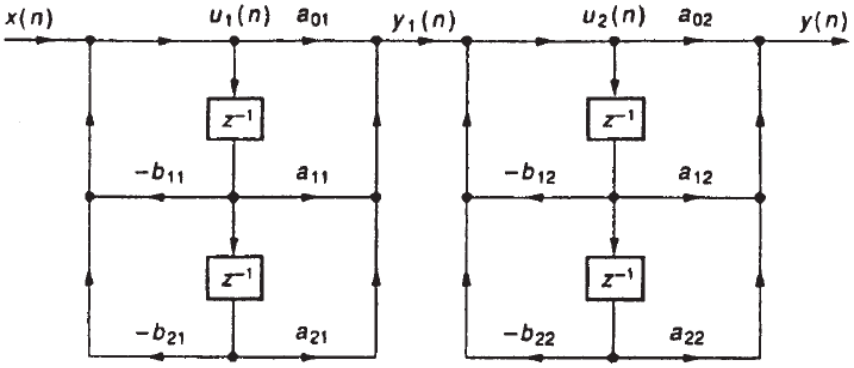


Figure 6.3 Fourth-order IIR filter with two direct form II sections in cascade.

The transfer function of the cascaded second order IIR filter is given mathematically by:

$$H(z) = \prod_{i=0}^{N/2} \frac{(a_{i0} + a_{i1}z^{-1} + a_{i2}z^{-2})}{(1 + b_{i1}z^{-1} + b_{i2}z^{-2})}$$

The transfer function of the above mentioned fourth order IIR filter is given mathematically by

$$H(z) = \frac{(a_{01} + a_{11}z^{-1} + a_{21}z^{-2})(a_{02} + a_{12}z^{-1} + a_{22}z^{-2})}{(1 + b_{11}z^{-1} + b_{21}z^{-2})(1 + b_{12}z^{-1} + b_{22}z^{-2})}$$

6.1.2 Parallel form structure

In parallel form structure, the transfer function of an IIR filter can be represented mathematically by

$$H(z) = C + H_1(z) + H_2(z) + \cdots + H_3(z)$$

This can be obtained by using a partial fraction expansion. The block diagram of the parallel form structure is shown in Figure 6.4. Each of the transfer functions$\mathbf{H_1(z)}$, $\mathbf{H_2(z)}$, …., etc.

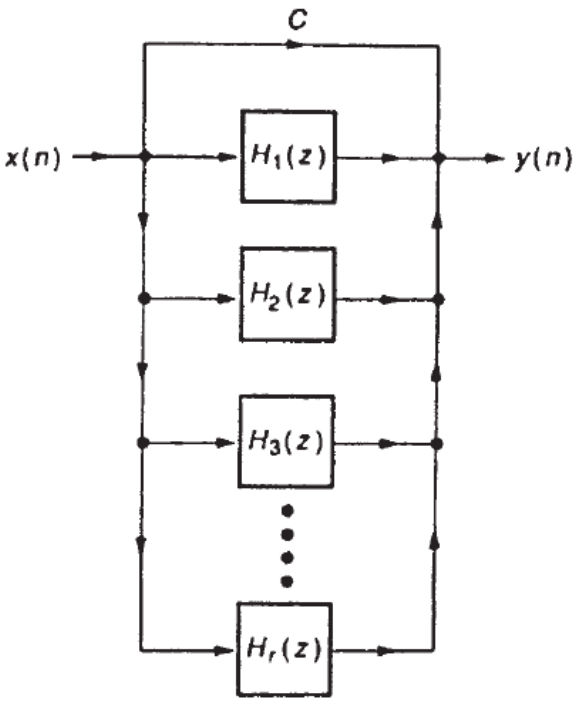


Figure 6.4 Parallel form IIR filter structure.

As with the cascade structure, the parallel form can be efficiently represented in terms of second-order direct form II structure sections. For example, for a fourth-order transfer function, $H(z)$ can be expressed as

$$H(z) = C + \frac{(a_{01} + a_{11}z^{-1} + a_{21}Z^{-2})}{(1 + b_{11}z^{-1} + b_{21}z^{-2})} + \frac{(a_{02} + a_{12}z^{-1} + a_{22}z^{-2})}{(1 + b_{12}z^{-1} + b_{22}z^{-2})}$$

Experimental procedures

In this experiment a low pass filter, high pass filter, band-pass filter and band stop (notch) filter will be implemented and tested as illustrated by the following exercises.

6.1.3 IIR Filter Implementation Using Second-Order Stages in Cascade (IIR)

In this experiment you will design a low pass filter whose cutoff frequency is $\boldsymbol{f_c = 2400Hz}$. In order to design this filter; filter coefficients must be determined first by using the SPTool in MATLAB as described below

1. In MATLAB command window open the SPTool filter design tool by typing SPTool in the MATLAB command window. A design window such as the one shown in Figure 6.5will appear on the screen.

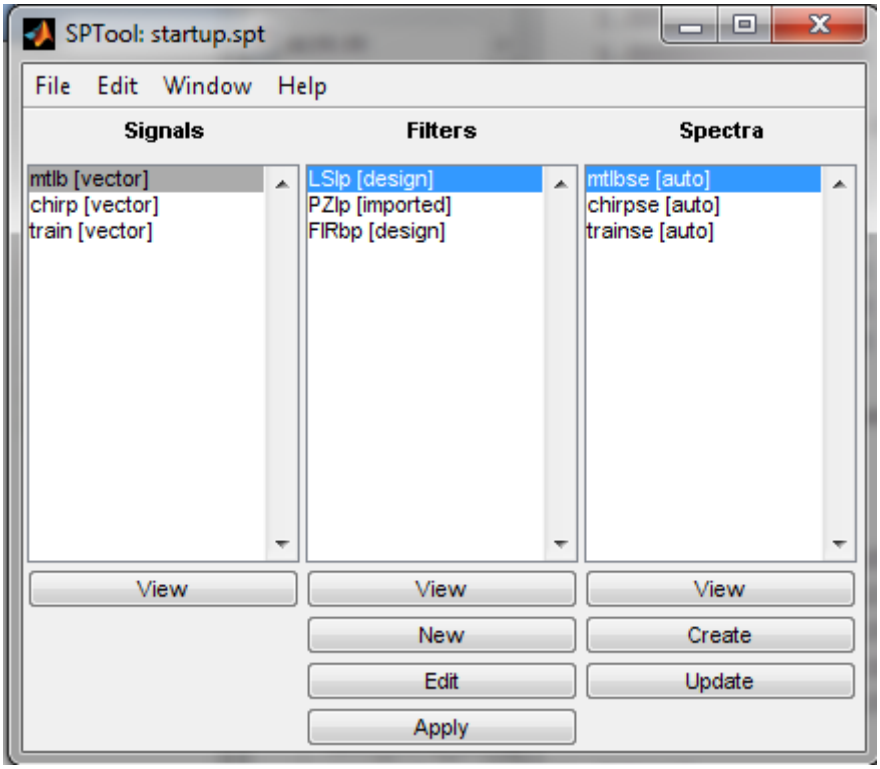


Figure 6.5 SPTool in Matlab

2. From the startup window; second column in Figure 6.5; select new. A design window such as the one shown in Figure 6.6 will pop up. From the design window select an IIR filter Elliptic, set the response to low pass and specify the filter order to 10. Set the sampling frequency to $\boldsymbol{f_s = 8kHz}$ and the cut off frequency $\boldsymbol{f_c = 2400Hz}$

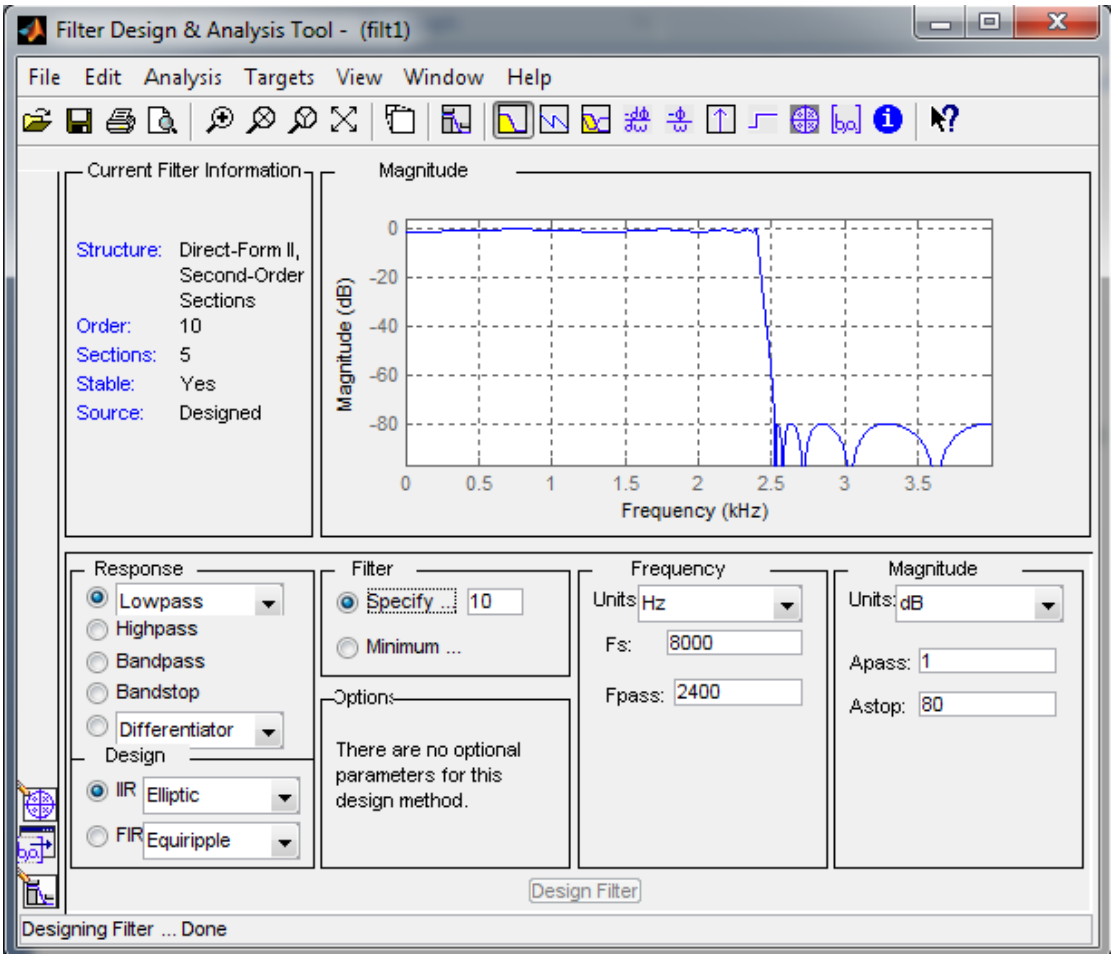


Figure 6.6 Filter design tool in MATLAB.

3. Access the startup window in SPTool again. Select →Edit→Name. Change the name (enter new variable name) to lp2400
4. Select File →Export → Export to workspace the lp2400 design.
5. Access MATLAB's workspace and type the following commands:

```
>>[z,p,k] = tf2zp(lp2400.tf.num, lp2400.tf.den);

>>sec_ord_sec = zp2sos(z,p,k);

>>sec_ord_sec = round(sec_ord_sec*2^15)
```

The first command finds the roots of the numerator and the denominator (zeros and poles). The second command converts the resulting floating-point coefficients into a format for implementation as second-order sections. The third command scales these coefficients for a fixed-point implementation. The resulting numerator and denominator coefficients should be listed as

27585	-10772	27585	32768	-11329	25257
32768	-12180	32768	32768	-9065	31465
32768	-13408	32768	32768	-15948	31492
32768	-11823	32768	32768	-10215	32554
32768	-13762	32768	32768	-15253	32557

These 30 coefficients represent the numerator coefficients $\boldsymbol{a_0}$, $\boldsymbol{a_1}$, and $\boldsymbol{a_2}$ and the denominator coefficients $\boldsymbol{b_0}$, $\boldsymbol{b_1}$, and $\boldsymbol{b_2}$.They represent six coefficients per stage, with $\boldsymbol{b_0}$ normalized to 1 and scaled by $\boldsymbol{2^{15} = 32,768}$.

The coefficients using SPTool should be contained in the file lp2400.cof, listed below. This file shows 25 coefficients (in lieu of 30).

Since the coefficient $\boldsymbol{b_0}$ is always normalized to 1, it is not used in the program.

```
//lp200.cof  IIR low pass coefficient file, with cut off at 2400Hz


#define stages 5                          //number of 2nd-order stages
int a[stages][3]= {                       //numerator coefficients
{27585, -10772, 27585},                   //a10, a11, a12 for 1st stage
{32768,-12180, 32768},                    //a20, a21, a22 for 2nd stage
{32768, -13408, 32768},                   //a30, a31, a32 for 3rd stage
{32768, -11823, 32768},                   //a40, a41, a42 for 4th stage
{32768, -13762, 32768}
 };
int b[stages][2]= {                       //denominator coefficients
{-11329    , 25257},                      //b11, b12 for 1st stage
 {-9065, 31465},                          //b21, b22 for 2nd stage
{-15948    , 31492},                      //b31, b32 for 3rd stage
{-10215   ,  32554},                      //b41, b42 for 4th stage
{-15253,  32557}                          //b51, b52 for 5th stage
 };
```

6. The IIR filter can be implemented by using the following two equations associated with each stage :

$$\boldsymbol{u(n) = x(n) - b_1 u(n-1) - b_2 u(n-2)}$$

$$\boldsymbol{y(n) = a_0 u(n) + a_1 u(n-1) + a_2 u(n-2)}$$

7. The loop section of code within the program is processed five times (the number of stages) for each value of $\boldsymbol{n}$, or sample period. For the first stage, $\boldsymbol{x(n)}$ is the newly acquired input sample. However, for the other stages, the input $\boldsymbol{x(n)}$ is the output $\boldsymbol{y(n)}$ of the preceding stage.
8. The coefficients$\boldsymbol{b[i][0]}$ and$\boldsymbol{b[i][1]}$ correspond to $\boldsymbol{b_1}$and $\boldsymbol{b_2}$, respectively; where i represents each stage. The delays $\boldsymbol{dly[i][0]}$ and $\boldsymbol{dly[i][1]}$ correspond to $\boldsymbol{u(n-1)}$and $\boldsymbol{u(n-2)}$, respectively.
9. To implement the filter IIR filter on the DSK6713 kit open the existing project that you worked on last time and add the following c code to it.

```
//IIR.c IIR filter using cascaded Direct Form II

//Coefficients a's and b's correspond to b's and a's from MATLAB

#include "DSK6713_AIC23.h"                          //codec-DSK support file

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;                  //set sampling rate

#include "lp2400.cof"                               //low pass @ 2400 Hz coefficient
file

short dly[stages][2] = {0};                         //delay samples per stage

interrupt void c_int11()                            //ISR

{

short i, input;

int un, yn;

input = input_sample();                             //input to 1st stage

for (i = 0; i < stages; i++)                        //repeat for each stage

{

un=input-((b[i][0]*dly[i][0])>>15) - ((b[i][1]*dly[i][1])>>15);

yn=((a[i][0]*un)>>15)+((a[i][1]*dly[i][0])>>15)+((a[i][2]*dly[i][1])>>15);

dly[i][1] = dly[i][0];                              //update delays

dly[i][0] = un;                                     //update delays
```

```
input = yn;                                  //intermed out->in to next stage
}
output_sample((short)yn);                    //output final result for time n
return;                                      //return from ISR
}
void main()
{
comm_intr();                                 //init DSK, codec, McBSP
while(1);                                    //infinite loop
}
```

10. After you build the project, load it on the DSK6713 kit, then run the program.
11. To test the designed LPF filter, connect a sine wave from the function generator to the LINE IN terminal of the DSK6713 kit.
12. Set the amplitude of the sign wave to $\mathbf{1}\boldsymbol{V_{pp}}$ and vary the frequency of the sine wave from 50 Hz to 3.4 kHz in steps of 200 Hz. Measure the amplitude of the output wave at every frequency point and tabulate your results as shown in Table

freq	50	400	600	1000	1400	1600	1800	2000	2200	2400	2600	3000	3200	3400
Am														

Table 3

Plot the amplitude versus frequency and comment on the results

14. Apply an input random signal, then connect the filter output to the spectrum analyzer. This can be achieved by generated the random signal through the MATLAB Simulink as shown in Figure below

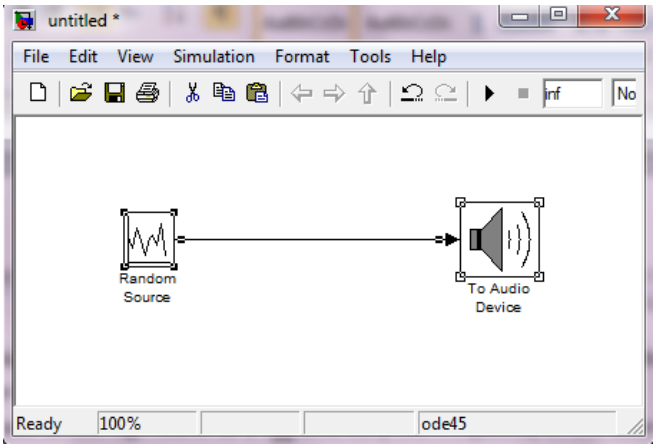


Figure 6.7 Random noise generator in MATLAB

15. Set the sample time for the noise to $\mathbf{1/8000}$ and it type is Gaussian.
16. Connect the output of the DSK kit to your pico-scope and adjust the scope for spectrum operation
17. Sketch the frequency response of the filter as it appears on the screen of the spectrum analyzer

13. Repeat the process and design high pass with cutoff frequency of 2200 Hz, band pass filter with a center frequency of 1750 Hz and a bandstop filter with band stop centered at 1750 Hz frequency of

6.1.4 IIR Inverse Filter (IIR inverse)

This example illustrates an IIR inverse filter. With noise as input, a forward IIR filter is calculated. The output of the forward filter becomes the input to an inverse IIR filter. The output of the inverse filter is the original input noise sequence.

The transfer function of an IIR filter is

$$\boldsymbol{H(z) = \frac{\sum_{i=0}^{N-1} a_i z^{-i}}{\sum_{j=1}^{M-1} b_j z^{-i}}}$$

The output sequence of the IIR filter is

$$\boldsymbol{y(n) = \sum_{i=0}^{N-1} a_i x(n-i) - \sum_{j=1}^{M-1} b_j x(n-j)}$$

where$\boldsymbol{x(n-i)}$ represents the input sequence. The input sequence $\boldsymbol{x(n)}$ can then be recovered using $\boldsymbol{\hat{x}(n)}$ as an estimate of $\boldsymbol{x(n)}$, or

$$\hat{x}(n) = \frac{\left(y(n) + \sum_{j=1}^{M-1} b_j y(n-j) - \sum_{i=1}^{N-1} a_i \hat{x}(n-i)\right)}{a_0}$$

In terms of second order cascaded Direct form II:

$$u(n) = y(n) - a_1 u(n-1) - a_2 u(n-2)$$

$$\hat{x}(n) = \frac{[u(n) + b_1 u(n-1) + b_2 u(n-2)]}{a_0}$$

The program IIRinverse.c implements the inverse IIR filter. Build this project as IIRinverse. Use noise as input to the system. Run the program and verify that the resulting output is the input noise (with the slider in the default position 1).

Change the slider and verify that the output of the forward IIR filter is an IIR bandpass filter centered at 2 kHz. The coefficient file bp2000.cof was used in Example 5.1 to implement an IIR filter. With the slider in position 3, verify that the output of the inverse IIR filter is the original input noise.

In this example, the forward filter's characteristics are known. This example can be extended so that the filter's characteristics are unknown. In such a case, the unknown forward filter's coefficients, a's and b's, can be estimated using Prony's method.

```
#include "dsk6713_AIC23.h" //codec-DSK support file

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#include "bp2000.cof" //BP @ 2 kHz coefficient file

short dly[stages][2] = {0}; //delay samples per stage

short out_type = 1; //type of output for slider

short a0, a1, a2, b1, b2; //coefficients

interrupt void c_int11() //ISR

{

short i, input, input1;

int un1, yn1, un2, input2, yn2;

input1 = input_sample(); //input to 1st stage

input = input1; //original input
```

```
for(i = 0; i < stages; i++) //repeat for each stage
{
a1 = ((a[i][1]*dly[i][0])>>15); //a1*u(n-1)
a2 = ((a[i][2]*dly[i][1])>>15); //a2*u(n-2)
b1 = ((b[i][0]*dly[i][0])>>15); //b1*u(n-1)
b2 = ((b[i][1]*dly[i][1])>>15); //b2*u(n-2)
un1 = input1 - b1 - b2;
a0=((a[i][0]*un1)>>15);
yn1 = a0 + a1 + a2; //stage output
input1 = yn1; //intermediate out->in next stage
dly[i][1] = dly[i][0]; //update delays u(n-2) = u(n-1)
dly[i][0] = un1; //update delays u(n-1) = u(n)
}
input2 = yn1; //out forward=in reverse filter
for(i = stages; i > 0; i--) //for inverse IIR filter
{
a1 = ((a[i][1]*dly[i][0])>>15); //a1u(n-1)
a2 = ((a[i][2]*dly[i][1])>>15); //a2u(n-2)
b1 = ((b[i][0]*dly[i][0])>>15); //b1u(n-1)
b2 = ((b[i][1]*dly[i][1])>>15); //b2u(n-2)
un2 = input2 - a1 - a2;
yn2 = (un2 + b1 + b2);
input2 = (yn2<<15)/a[i][0]; //intermediate out->in next stage
}
if(out_type == 1) //if slider in position 1
```

```
output_sample(input); //original input signal
if(out_type == 2) output_sample((short)yn1);//forward filter
if(out_type == 3) output_sample((short)(yn2>>3));//inverse filter
return; //return from ISR
}
void main()
{
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```

7 Experiment 7 Adaptive filters

Theory

Adaptive filters are filters with varying coefficients. These filters are used in applications where a given system is changing its coefficients in an unknown manner. A typical system can be either a communication channel whose coefficients are changing with either temperature or number of users.

In such cases it is highly desirable to design the filter to be self-learning so that it can adapt itself to the situation at hand.

The coefficients of an adaptive filter are adjusted to compensate for changes in input signal, output signal, or system parameters. Instead of being rigid, an adaptive system can learn the signal characteristics and track slow changes. An adaptive filter can be very useful when there is uncertainty about the characteristics of a signal or when these characteristics change.

Conceptually, the adaptive scheme is fairly simple. Most of the adaptive schemes can be described by the structure shown in Figure 7.1. This is a basic adaptive filter structure in which the adaptive filter's output $\boldsymbol{y}$ is compared with a desired signal $\boldsymbol{d}$to yield an error signal $\boldsymbol{e}$, which is fed back to the adaptive filter.

The error signal is input to the adaptive algorithm, which adjusts the filter's coefficients to satisfy some predetermined criteria or rules. The desired signal is usually the most difficult one to obtain. One of the first questions that probably come to mind is: Why are we trying to generate the desired signal at $\boldsymbol{y}$ if we already know it? Surprisingly, in many applicationsthe desired signal does exist somewhere in the system or is known a priori. The challenge in applying adaptive techniques is to figure out where to get the desired signal, what to make the output $\boldsymbol{y}$, and what to make the error $\boldsymbol{e}$.

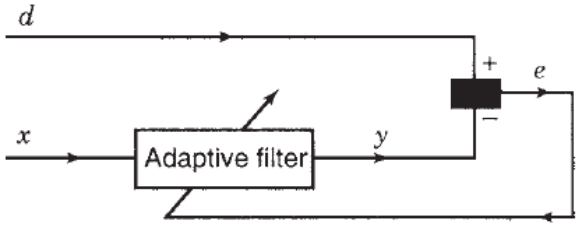


Figure 7.1Basic adaptive filter structure

The coefficients of the adaptive filter are adjusted, or optimized, using an LMS algorithm based on the error signal. Here we discuss only the LMS searching algorithm with a linear combiner (FIR

filter), although there are several strategies for performing adaptive filtering. The output of the adaptive filter in Figure is

$$\boldsymbol{y[n] = \sum_{k=0}^{N-1} w_k[n]x[n-k]}$$

Where $\boldsymbol{w_k[n]}$ represent $\boldsymbol{N}$ weights or coefficients for a specific time $\boldsymbol{n}$. The above mentioned equation represents the convolution which was implemented for the FIR experiment.

The error signal is defined as the difference between the desired signal $\boldsymbol{d[n]}$ and the adaptive filter's output $\boldsymbol{y[n]}$.

$$\boldsymbol{e[n] = d[n] - y[n]}$$

The weights or coefficients $\boldsymbol{w_k[n]}$ are adjusted such that a mean squared error function is minimized. This mean squared error function is $\boldsymbol{E[e^2[n]]}$, where $\boldsymbol{E}$ represents the expected value. Since there are $\boldsymbol{k}$ weights or coefficients, a gradient of the mean squared error function is required. An estimate can be found instead using the gradient of $\boldsymbol{e^2[n]}$, yielding

$$\boldsymbol{w_k[n+1] = w_k[n] + 2\beta e[n]x[n-k]k = 0, 1, \dots, N-1}$$

7.1 Applications of adaptive filters

Adaptive filters have been used for different applications such as:

1. For noise cancellation as illustrated in Figure 7.2. The desired signal $\boldsymbol{d}$ is corrupted by uncorrelated additive noise $\boldsymbol{n}$. The input to the adaptive filter is a noise $\boldsymbol{n'}$that is correlated with the noise $\boldsymbol{n}$. The noise $\boldsymbol{n'}$ could come from the same source as $\boldsymbol{n}$ but modified by the environment. The adaptive filter's output $\boldsymbol{y}$ is adapted to the noise $\boldsymbol{n}$. When this happens, the error signal approaches the desired signal $\boldsymbol{d}$. The overall output is this error signal and not the adaptive filter's output $\boldsymbol{y}$. If $\boldsymbol{d}$ is uncorrelated with $\boldsymbol{n}$, the strategy is to minimize $\boldsymbol{E(e^2)}$, where $\boldsymbol{E()}$ is the expected value. The expected value is generally unknown; therefore, it is usually approximated with a running average or with the instantaneous function itself. Its signal component,$\boldsymbol{E(d^2)}$, will be unaffected and onlyits noise component $\boldsymbol{E[(n - y)^2]}$ will be minimized.

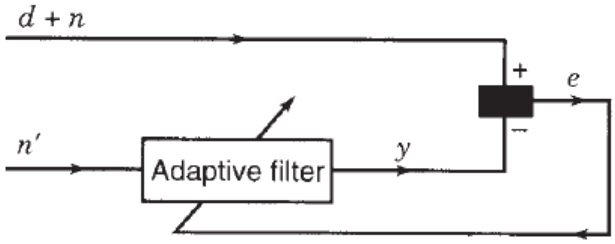


Figure 7.2 Adaptive filter structure for noise cancellation

2. For system identification. Figure 7.3 shows an adaptive filter structure that can be used for system identification or modeling. The same input is to an unknown system in parallel with an adaptive filter. The error signal $\boldsymbol{e}$ is the difference between the response of the unknown system $\boldsymbol{d}$ and the response of the adaptive filter $\boldsymbol{y}$. This error signal is fed back to the adaptive filter and is used to update the adaptive filter's coefficients until the overall output $\boldsymbol{y} = \boldsymbol{d}$.When this happens, the adaptation process is finished, and $\boldsymbol{e}$ approaches zero. If the unknown system is linear and not time-varying, then after the adaptation is complete, the filter's characteristics no longer change. In this scheme, the adaptive filter models the unknown system. This structure is illustrated later with three programming examples.

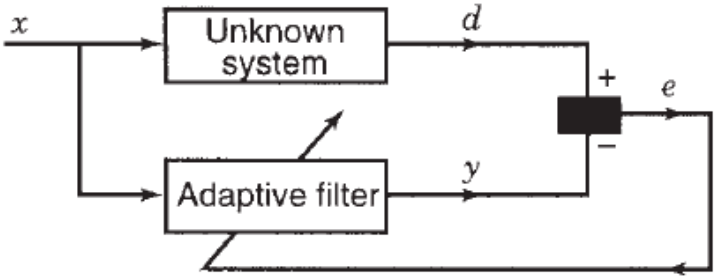


Figure 7.3 Adaptive filter structure for system identification

3. Inverse system modeling (e.g. channel equalization in modems) which is shown in Figure 7.4

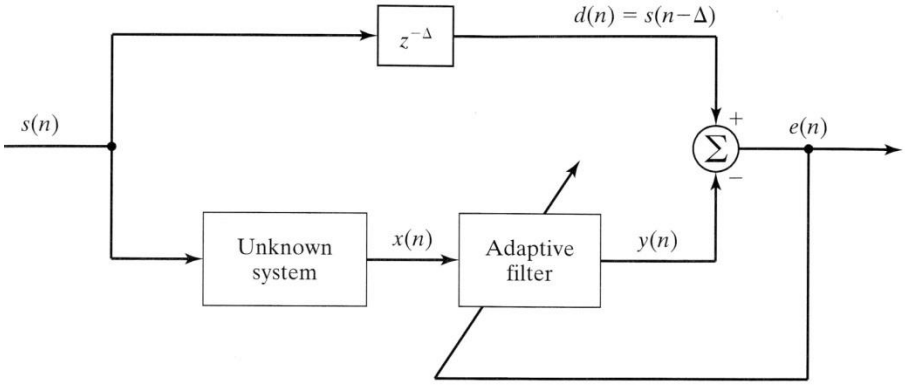


Figure 7.4 adaptive filter for system modeling

4. Adaptive predictor. Figure 7.5 shows an adaptive predictor structure that can provide an estimate of an input. This structure is illustrated later with a programming example.

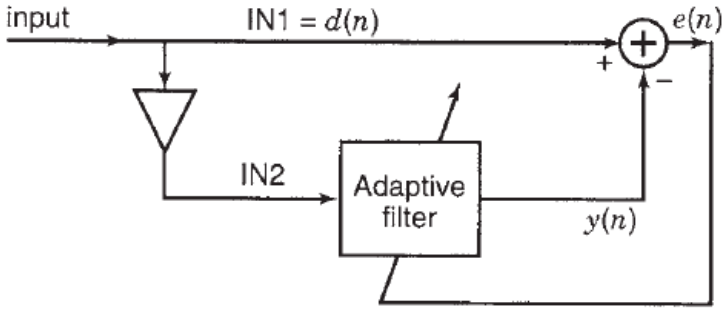


Figure 7.5 Adaptive predictor structure

Experimental procedures

In this experiment an adaptive filter for noise cancellation and system identification will be designed and implemented.

7.1.1 Adaptive Filter for Sinusoidal Noise Cancellation

This example illustrates the application of the LMS criterion to cancel an undesirable sinusoidal noise.

A desired sine wave of 1500 Hz with an additive (undesired) sine wave noise of 312 Hz forms one of two inputs to the adaptive filter structure. A reference (template) cosine signal, with a frequency of 312Hz, is the input to a 30-coefficient adaptive FIR filter. The 312-Hz reference cosine signal is correlated with the 312-Hz additive sine noise but not with the 1500-Hz desired sine signal.

For each time $\boldsymbol{n}$, the output of the adaptive FIR filter is calculated and the 30 weights or coefficients are updated along with the delay samples. The error signal $\boldsymbol{e}$ is the overall desired output of the adaptive structure. This error signal is the difference between the desired signal and additive noise (dplusn) and the adaptive filter's output, $\boldsymbol{y(n)}$.

To perform these parts of the experiment follow these steps

1. In MATLAB,generate the desired signal, the noise plus the desired signal and the reference noise according to the following MATLAB commands

```
%Adaptnoise.m Generates: dplusn.h, refnoise.h, and sin1500.h

for i=1:128

desired(i) = round(100*sin(2*pi*(i-1)*1500/8000)); %sin(1500)

addnoise(i) = round(100*sin(2*pi*(i-1)*312/8000)); %sin(312)

refnoise(i) = round(100*cos(2*pi*(i-1)*312/8000)); %cos(312)

end

dplusn= desired+addnoise;
```

```
fid=fopen('sin1500.h','w'); %desired sin(1500)
fprintf(fid,'short sin1500[128]={');
fprintf(fid,'%d, ' ,desired(1:127));
fprintf(fid,'%d' ,desired(128));
fprintf(fid,'};\n');
fclose(fid);
fid=fopen('dplusn.h','w'); %desired + noise
fprintf(fid,'short dplusn[128]={');
fprintf(fid,'%d, ' ,dplusn(1:127));
fprintf(fid,'%d' ,dplusn(128));
fprintf(fid,'};\n');
fclose(fid);
fid=fopen('refnoise.h','w'); %reference noise
fprintf(fid,'short refnoise[128]={');
fprintf(fid,'%d, ' ,refnoise(1:127));
fprintf(fid,'%d' ,refnoise(128));
fprintf(fid,'};\n');
fclose(fid);
```

2. In your code composer studio use the following code to implement an adaptive filter

```
#include "DSK6713_AIC23.h" //codec-DSK support file
Uint32 fs= DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#include "refnoise.h" //cosine 312 Hz
#include "dplusn.h" //sin(1500) + sin(312)
#define beta 1E-10 //rate of convergence
#define N 30 //# of weights (coefficients)
#define NS 128 //# of output sample points
```

```
float w[N]; //buffer weights of adapt filter
float delay[N]; //input buffer to adapt filter
short output; //overall output
short out_type = 1; //output type for slider
short buffercount=0;
interrupt void c_int11() //ISR
{
short i;
float yn, E; //output filter/"error" signal
delay[0] = refnoise[buffercount]; //cos(312Hz) input to adapt FIR
yn = 0; //init output of adapt filter
for (i = 0; i < N; i++) //to calculate out of adapt FIR
yn += (w[i] * delay[i]); //output of adaptive filter
E = dplusn[buffercount] - yn; //"error" signal=(d+n)-yn
for (i = N-1; i >= 0; i--) //to update weights and delays
{
w[i] = w[i] + beta*E*delay[i]; //update weights
delay[i] = delay[i-1]; //update delay samples
}
buffercount++; //increment buffer count
if (buffercount>= NS) //if buffercount=# out samples
buffercount = 0; //reinit count
if (out_type == 1) //if slider in position 1
output = ((short)E*10); //"error" signal overall output
else if (out_type == 2) //if slider in position 2
output=dplusn[buffercount]*10; //desired(1500)+noise(312)
```

```
output_sample(output); //overall output result
return; //return from ISR
}
void main()
{
short T=0;
for (T = 0; T < 30; T++)
{
w[T] = 0; //init buffer for weights
delay[T] = 0; //init buffer for delay samples
}
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```

3. Write a slider code to change the parameter out_type from 1 to 2
4. Build and execute the project
5. Plot the signal that as you see on the screen of the oscilloscope if the out_type variable is set to 1.

   Question 1. What signal you are measuring on the oscilloscope screen?
6. Plot the signal that as you see on the screen of the oscilloscope if the out_type variable is set to 2.

   Question 2. What signal you are measuring on the oscilloscope screen?
7. Test the effect of the rate of convergence by changing beta by a factor of 10 in the program.
8. Repeat the experiment by using Gaussian error signal as the noise rather than using a sinusoidal signal. Use following MATLAB commands

```
%Adaptnoise.m Generates: dplusn.h, refnoise.h
for i=1:128
desired(i) = round(100*sin(2*pi*(i-1)*1500/8000)); %sin(1500)
end
```

```
snr = 5; % SNR in dB

   n1 = 100/sqrt(2)*[randn(1,128)]; % white gaussian noise,
0dB variance

addnoise= 10^(-snr/20)*n1; %  Noise addition

      n2 = 100/sqrt(2)*[randn(1,128)]; % white gaussian noise,
0dB variance


refnoise=10^(-snr/20)*n2; % Noise addition

dplusn= desired+addnoise;

fid=fopen('dplusn.h','w'); %desired + noise

fprintf(fid,'short dplusn[128]={');

fprintf(fid,'%d, ' ,dplusn(1:127));

fprintf(fid,'%d' ,dplusn(128));

fprintf(fid,'};\n');

fclose(fid);

fid=fopen('refnoise.h','w'); %reference noise

fprintf(fid,'short refnoise[128]={');

fprintf(fid,'%d, ' ,refnoise(1:127));

fprintf(fid,'%d' ,refnoise(128));

fprintf(fid,'};\n');

fclose(fid);
```

7.1.2 Adaptive FIR Filter for System ID of a Fixed FIR as an Unknown System

Adaptive filters can be used for system identification as illustrated in Figure 7.3.

To test the adaptive scheme, the unknown system to be identified is chosen as an FIR bandpass filter with 55 coefficients centered at $\boldsymbol{Fs/4 = 2kHz}$. The coefficients of this fixed FIR filter are in the file bp55.cof. A 60-coefficient adaptive FIR filter models the fixed unknown FIR band pass filter.

A pseudorandom noise sequence is generated within the program and becomes the input to both the fixed (unknown) and the adaptive FIR filters. This input signal represents a training signal. The adaptation process continues until the error signal is minimized. This feedback error signal is the difference between the output of the fixed unknown FIR filter and the output of the adaptive FIR filter.

An extra memory location is used in each of the two delay sample buffers (fixedand adaptive FIR). This is used to update the delay samples.

1. Design an FIR bandpass filter with a center frequency of $\boldsymbol{f_c = 2kHz}$, bandwidth of 400 Hz. Use 55 coefficients in designing the filter. Export the filter coefficients to a file called bp55.cof (refer to experiment 5 and follow the steps).
2. Use the following c file required for the random noise generation in your project directory

```
//Noise_gen.c header file for pseudo-random noise sequence

typedef struct BITVAL    //register bits to be packed as integer
{
 unsigned int b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1;
 unsigned int b7:1, b8:1, b9:1, b10:1, b11:1, b12:1,b13:1;
 unsigned int dweebie:2; //Fills the 2 bit hole - bits 14-15
} bitval;

typedef union SHIFT_REG
{
unsigned int regval;
bitval bt;
} shift_reg;

int fb = 1;                         //feedback variable
shift_reg sreg=0xFFFF;                    //shift register

int noiseGen()                    //pseudo-random sequence {-1,1}
{
int prnseq;
if(sreg.bt.b0) prnseq = -8000;       //scaled negative noise level
else prnseq = 8000;                 //scaled positive 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;                     //return noise sequence
}
```

3. Build and run this project as adaptIDFIR. Verify that the output (adaptfir_out) of the adaptive FIR filter converges to a bandpass filter centered at 2kHz(with the slider in position 1 by default).
4. With the slider in position 2, verify the output (fir_out) of the fixed FIR bandpass filter centered at 2 kHz and represented by the coefficient file bp55.cof. It can be observed that this output is practically identical to the adaptive filter's output.

```
#include "DSK6713_AIC23.h"                                    //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;    //set sampling rate
#include "bp55.cof"              //fixed FIR filter coefficients
#include "Noise_gen.c"            //support noise generation file
#define beta 1E-14               //rate of convergence
#define WLENGTH 60                //# of coeff for adaptive FIR
float w[WLENGTH+1];               //buffer coeff for adaptive FIR
int dly_adapt[WLENGTH+1];          //buffer samples of adaptive FIR
int dly_fix[N+1];                 //buffer samples of fixed FIR
short out_type = 1;               //output for adaptive/fixed FIR
interrupt void c_int11()          //ISR
{
int i;
int fir_out = 0;                  //init output of fixed FIR
int adaptfir_out = 0;             //init output of adapt FIR
float E;                          //error=diff of fixed/adapt out
dly_fix[0] = noiseGen();          //input noise to fixed FIR
dly_adapt[0]=dly_fix[0];          //as well as to adaptive FIR
for (i = N-1; i>= 0; i--)
{
fir_out +=(h[i]*dly_fix[i]);      //fixed FIR filter output
dly_fix[i+1] = dly_fix[i];        //update samples of fixed FIR
```

```
}
for (i = 0; i < WLENGTH; i++)
adaptfir_out +=(w[i]*dly_adapt[i]);  //adaptive FIR filter output
E = fir_out - adaptfir_out;         //error signal
for (i = WLENGTH-1; i >= 0; i--)
{
w[i] = w[i]+(beta*E*dly_adapt[i]);   //update weights of adaptive FIR
dly_adapt[i+1] = dly_adapt[i];       //update samples of adaptive FIR
}
if (out_type == 1)                  //slider position for adapt FIR
output_sample((short)adaptfir_out);  //output of adaptive FIR filter
else if (out_type == 2)             //slider position for fixed FIR
output_sample((short)fir_out);       //output of fixed FIR filter
return;}
void main() {
int T=0, i=0;
for (i = 0; i < WLENGTH; i++) {
w[i] = 0.0;                         //init coeff for adaptive FIR
dly_adapt[i] = 0;                   //init buffer for adaptive FIR
}
for (T = 0; T < N; T++)
dly_fix[T] = 0;                     //init buffer for fixed FIR
                                    //initial feedback value
comm_intr();                        //init DSK, codec, McBSP
while (1);                          //infinite loop
}
```

8 Experiment 8Audio Effects

Objectives

The aim of this experiment is to generate some audio effects such as echo, multi echo and reverberation

Introduction

In natural environments, sounds we perceive depend on the listening conditions and in particular on the acoustic environment. The same sound signal played in a concert hall; bathroom or a small room will not be “perceived” the same. Since these effects are important for musicians, the digital technology can be used to simulate them.

8.1.1 Delay

Delay is the simplest audio effect which holds input signal and then plays it back after a period of time. Delay is used very often by musicians. It is also the main block for other audio effects such as reverb, chorus, and flanging.

The difference equation for the delay operation is $\boldsymbol{y[n] = x[n - n_0]}$ where $\boldsymbol{n_0}$ is the delay amount. Since the difference equation between the output and the input is specified, it can be directly coded in C language. To implement the delay operation, the best way is defining an array which stores input audio signals. In Figure 8.1, we demonstrate the delay operation using $\boldsymbol{n_0 + 1}$ length array which should be defined beforehand. To feed the delayed audio signal to output, first we should store the audio signal on the first entry of the array. At each operation cycle, each entry in the array should be shifted towards right, to open space to the new input. This way, an input which is taken at time $\boldsymbol{n}$ will reach to the end of the array $\boldsymbol{n_0}$ cycles later.

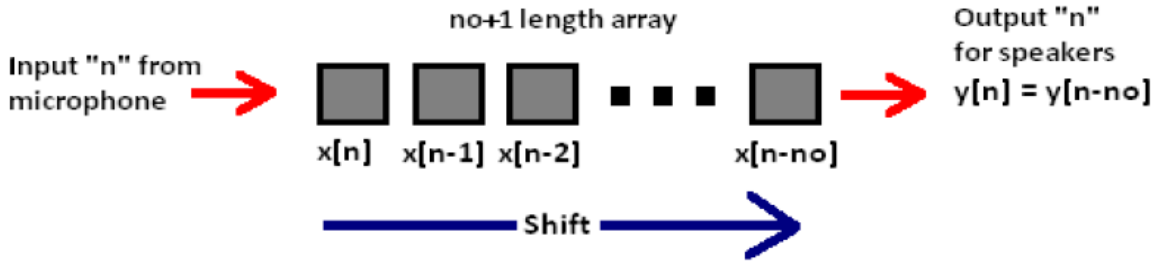


Figure 8.1 block diagram illustrating delay

8.1.2 Echo

The echo block simply takes an audio signal and plays it after summing with a delayed version of the same signal. The delay time can range from several milliseconds to several seconds.

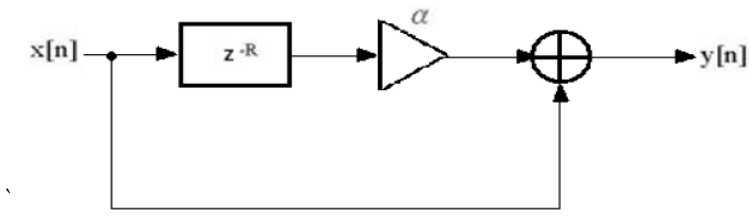


Figure 8.2 echo block diagram

The difference equation which describes the echo system is $\boldsymbol{y[n] = x[n] + \alpha x[n - n_0]}$; where $\boldsymbol{\alpha}$ is the delay mix parameter, $\boldsymbol{n_0}$ is the delay amount.

In order to generate the echo effect, both the present and the delayed signal are needed. In order to access to the delayed signal, we should store the input audio signal in an array to generate the delayed version of the signal as explained in the previous sub-section. Using this Objectives

The objectives of this experiment is show students how can an adaptive filter be used for different applications such as noise cancelation and system identifications

array, generate the echo as illustrated by the echo equation.

8.1.3 Reverberation

Reverberation is also one of the most heavily used effects in music. The effects of combining an original signal with a very short (<20ms) time-delayed version of itself results is reverberation. In fact, when a signal is delayed for a very short time and then added to the original, the ear perceives a fused sound rather than two separate sounds. If a number of very short delays (that are fed back) are mixed together, a crude form of reverberation can be achieved. The block diagram of a basic reverberation system is given in Figure 8.3.

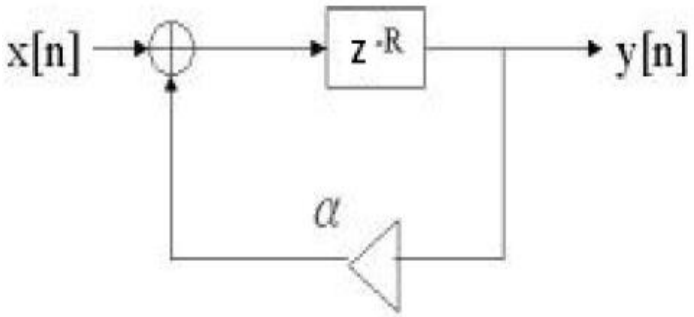


Figure 8.3The block diagram of a multi-echo reverberation system

The difference equation for this system is $\boldsymbol{y[n] = x[n] + \alpha y[n - n_0]}$. The value of $\boldsymbol{\alpha}$ normally falls between 0 and 1.

Experiment procedure

In this experiment echo generation and cancelation will be demonstrated by the procedure explained in the next subsections. Reverberation will be considered in section 2.2.

8.1.4 Echo generation

1. In this part of the experiment you are going to generate an echo signal by reading an audio samples form the LINE IN terminal then delay and process these signals according to the echo equation. Your code may appear as shown below

```
#include "dsk6713_aic23.h" //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
short input, output;
short bufferlength = 3000; //buffer size for delay
short buffer[3000]; //create buffer
short i = 0;
short amplitude = 5; //to vary amplitude of echo
interrupt void c_int11() //ISR
{
input = input_sample(); //newest input sample data
output=input + 0.1*amplitude*buffer[i];//newest + oldest samples
output_sample(output); //output sample
buffer[i] = input; //store newest input sample
i++; //increment buffer count
if (i >= bufferlength) i = 0; //if end of buffer reinit
}
main()
{
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```

2. Vary the buffer size from 1000 to 8000 and listen to the played recorded wave. What effect do you observe on heard signal?
3. In your code change the statement buffer[i]=input into buffer[i]=output and listen to the recorded signal. What effect to the signal happens here?
4. Modify the buffer size to 160 and keep the statement of buffer[i]=output, and listen to the recorded signal. What effect to the signal happens here? Calculate the delay in ms.
5. Try to implement multi echo by using different buffers with different buffer lengths and modify your code as shown below

```
* File Name :  echogeneration.c
* Target   :  TMS320C6713
* Version  :  3.1
```

```
* Description : This Program tells about the Echo generation.
  Input is taken from Mic-in using Mic, n output can be analysed
by using headphone.
*******************************************************************************

#include"dsk6713_aic23.h"                    //this file is added to initialize the DSK6713
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;         // set sampling frequency

short input,output;                                          // variable initialization
short bufferlength = 4000;                                   // buffer initialization of different length for storage of
input analog-signal
short buffer[4000];
short bufferlength1 = 8000;
short buffer1[8000];
short bufferlength2 = 12000;
short buffer2[12000];
short bufferlength3 = 16000;
short buffer3[16000];
short i = 0,j = 0,k = 0,l=0;
short amplitude =1;

interrupt void c_int11()                             // ISR call, At each Interrupt, program execution goes to the
interrupt service routine
{
        input = input_sample();                              // input from Mic
        output =input + 0.4*amplitude*buffer[i]+0.3*amplitude*buffer1[j]+ 0.2*amplitude*buffer2[k]+
0.1*amplitude*buffer3[l];
        output_sample(output);                                       // // the value in the buffer sine_table indexed
the variable loop is written on to the codec.
        buffer[i]  = input;
        buffer1[j] = buffer[i];
        buffer2[k] = buffer1[j];
        buffer3[l] = buffer2[k];
        i++;
        j++;
        k++;
        l++;
        if(i >= bufferlength)  i = 0;
        if(j >= bufferlength1) j = 0;
        if(k >= bufferlength2) k = 0;
        if(l >= bufferlength3) l = 0;
}
main()                                       // main routin call
{
        comm_intr();              // ISR function is called, using the given command
        while(1);                            // program execution halts and it starts listening for the interrupt which
occur at every sampling period Ts.
```

```
}
```

8.1.5 Echo with Control for Different Effects

In this part of the experiment you are to generate an echo signal and control the echo parameters using three sliders.

1. If the echo_type is set by the slider to 1 then fading is selected otherwise normal echo is used
2. The delay of the echo also can be made variable by using a delay parameter called delay. The delay parameter either increase or decrease the length of the buffer from 1000 to 8000 in steps of 1000
3. The amplitude parameter α of the delayed version of the input also can be made variable, therefore generating different effects on the echo signal
4. Type the program shown below and the subsequent gel files to accomplish this task

```
//Echo_control.c Echo effects with fading

//3 sliders to control effects: buffer size, amplitude, fading
#include "DSK6713_AIC23.h" //codec-DSK file support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
short input, output;
short buffer[8000]; //max size of buffer
short bufferlength = 1000; //initial buffer size
short i = 0; //buffer index
short delay = 3; //determines size of buffer
short delay_flag = 1; //flag if buffer size changes
short amplitude = 5; //amplitude control by slider
short echo_type = 1; //1 for fading(0 with no fading)
main()
{
short new_count; //count for new buffer
comm_poll(); //init DSK, codec, McBSP
while(1) //infinite loop
{
output=input+0.1*amplitude*buffer[i];//newest + oldest samples
if (echo_type == 1) //if fading is desired
{
new_count = (i-1) % bufferlength; //previous buffer location
buffer[new_count] = output; //to store most recent output
}
output_sample(output); //output delayed sample
input = input_sample(); //newest input sample data
if (delay_flag != delay) //if delay has changed
{ //new buffer size
delay_flag = delay; //reint for future change
```

```
bufferlength = 1000*delay; //new buffer length
i = 0; //reinit buffer count
}
buffer[i] = input; //store input sample
i++; //increment buffer index
if (i == bufferlength) i=0; //if @ end of buffer reinit
}
}
```

The slider files may appear as shown below

```
//Echo_control.gel Sliders vary time delay,amplitude,and type of echo
menuitem "Echo with Fading"
slider Amplitude(1,8,1,1,amplitude_parameter) /*incr by 1, up to 8*/
{
amplitude = amplitude_parameter; /*vary amplit of echo*/
}
slider Delay(1,8,1,1,delay_parameter) /*incr by 1, up to 8*/
{
delay = delay_parameter; /*vary buffer size*/
}
slider Type(0,1,1,1,echo_typeparameter) /*incr by 1, up to 1*/
{
echo_type = echo_typeparameter; /*echo type for fading*/
}
```

9 Experiment 9 Hamming Codes

Objectives

To brief student with hamming codes for error correction and detection

Theory of Hamming Code

In telecommunication, a Hamming code is a linear error-correcting code named after its inventor, Richard Hamming. Hamming codes can detect up to two contiguous bit errors, and correct single-bit errors; thus, reliable communication is possible when the Hamming distance between the transmitted and received bit patterns is less than or equal to one.

In mathematical terms, Hamming codes are a class of binary linear codes. For each integer $\boldsymbol{m \geq 2}$there is a code with *m* parity bits and $2^m - m - 1$ data bits. The parity check matrix of a Hamming code is constructed by listing all columns of length *m* that are pair wise independent. Hamming codes are an example of perfect codes, codes that exactly match the theoretical upper bound on the number of distinct code words for a given number of bits and ability to correct errors. Because of the simplicity of Hamming codes, they are widely used in computer memory (RAM).

9.1.1 General algorithm

The following general algorithm generates a single-error correcting (SEC) code for any number of bits.

1. Number the bits starting from 1: bit 1, 2, 3, 4, 5, etc.
2. Write the bit numbers in binary. 1, 10, 11, 100, 101, etc.
3. All bit positions that are powers of two (have only one 1 bit in the binary form of their position) are parity bits.
4. All other bit positions, with two or more 1 bits in the binary form of their position, are data bits.
5. Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position.
    1. Parity bit 1 covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself), 3, 5, 7, 9, etc.
    2. Parity bit 2 covers all bit positions which have the second least significant bit set: bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.
    3. Parity bit 4 covers all bit positions which have the third least significant bit set: bits 4–7, 12–15, 20–23, etc.

This general rule can be shown visually:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X	
	p2		X	X			X	X			X	X			X	X			X	X	
	p4				X	X	X	X					X	X	X	X					X
	p8								X	X	X	X	X	X	X	X					
	p16																X	X	X	X	X

As you can see, if you have *m* parity bits, it can cover bits from 1 up to $\mathbf{2^m - 1}$. If we subtract out the parity bits, we are left with $\mathbf{2^m - m - 1}$ we can use for the data. As *m*varies, we get all the possible Hamming codes:

Parity bits	Total bits	Data bits	Name
2	3	1	Hamming(3,1) (Triple repetition code)
3	7	4	Hamming(7,4)
4	15	11	Hamming(15,11)
5	31	26	Hamming(31,26)
...			
m	$2^m - 1$	$2^m - m - 1$	Hamming$(2^m - 1, 2^m - m - 1)$

9.1.2 Calculating the Hamming Code

The key to the Hamming Code is the use of extra parity bits to allow the identificationof a single error. Create the code word as follows:

1. Mark all bit positions that are powers of two as parity bits. (Positions 1, 2, 4, 8, 16, 32, 64, etc.)
2. All other bit positions are for the data to be encoded. (Positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)
3. Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.

Position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc.
(1,3,5,7,9,11,13,15,...)
Position 2: check 2 bits, skip 2 bits, check 2 bits, skip 2 bits, etc.
(2,3,6,7,10,11,14,15,...)
Position 4: check 4 bits, skip 4 bits, check 4 bits, skip 4 bits, etc.
(4,5,6,7,12,13,14,15,20,21,22,23,...)
Position 8: check 8 bits, skip 8 bits, check 8 bits, skip 8 bits, etc.
(8-15,24-31,40-47,...)

$$\boldsymbol{p1 = P3 \oplus P5 \oplus P7 \oplus P9 \oplus \dots\dots\dots}$$
$$\boldsymbol{p2 = P3 \oplus P6 \oplus P7 \oplus P10 \oplus P11 \oplus \dots\dots\dots}$$

4. Set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

9.1.3 Detection of errors in the hamming code

Every integer $\boldsymbol{m}$ there is a $(\boldsymbol{2^m - 1})$ bit Hamming code which contains $\boldsymbol{m}$ parity bits and $\boldsymbol{2^m - 1 - m}$information bits.The parity bits are intermixed with the information bits as follows:

If we number the bit positions from 1 to $\boldsymbol{2^m - 1}$, the bits in position $\boldsymbol{2^k}$, where $\boldsymbol{0 \le k \le m - 1}$, are the parity bits, and the bits in the remaining positions are information bits. The value of each parity bit is chosen so that the total number of 1's in a specific group of bit positions is even.

For $\boldsymbol{m = 3}$, we have a $(\boldsymbol{2^m - 1 = 7bits})$ Hamming code as shown below:

7	6	5	4	3	2	1	Bit positions
1	1	1	1	0	0	0	Parity group for parity bit 4
1	1	0	0	1	1	0	Parity group for parity bit 2
1	0	1	0	1	0	1	Parity group for parity bit 1
			↑		↑	↑	
			Parity bits				

So to find the bit error position:

$$\boldsymbol{c1 = P1 \oplus P3 \oplus P5 \oplus P7}$$

$$\boldsymbol{c2 = P2 \oplus P3 \oplus P6 \oplus P7}$$

$$\boldsymbol{c3 = P4 \oplus P5 \oplus P6 \oplus P7}$$

Then, the position of bit error is $\boldsymbol{C = (c3c2c1)}$.

Experimental procedures

In order to generate the hamming code and detect hamming code errors follow these steps

19. Set the sampling frequency to $f_s = 8\ kHz$
20. Use the following code to generate a (7,4) hamming code.

```
#include "dsk6713_aic23.h"                  //this file is added to initialize the DSK6713
#include<stdio.h>                                   // stdio.h, which stands for "standard
input/output header", is the header in the C standard library that contains macro definitions,
constants, and declarations of functions and types used for various standard input and output
operations.
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;          // set sampling frequency

void main()
{
        int d[4],c[7],i,cod[7];
        printf("Data  is ");
        for(i=0;i<4;i++)
        scanf("%d",&d[i]);          // takes in values from the user in the form of '0' and '1'

        for(i=0;i<4;i++)
        c[i]=d[i];
        c[4]=(d[0]+d[1]+d[3])%2;
        c[5]=(d[0]+d[2]+d[3])%2;
        c[6]=(d[1]+d[2]+d[3])%2;

        cod[0]=c[4];
        cod[1]=c[5];
        cod[2]=c[0];
        cod[3]=c[6];
        cod[4]=c[1];
        cod[5]=c[2];
```

```
        cod[6]=c[3];

        for(i=0;i<7;i++)                    // arranges` the 7 bits appended 3 bits with the user
input 4 bits and generates the Hamming Code
        printf("%d",cod[i]);                // displays the output 7-bit result

        printf("\n This is data bit appended with correction \n");      // displays the line on CCS
screen

}

interrupt void c_int11() // ISR call, At each Interrupt, program execution goes to the interrupt
service routine
{
        comm_poll();                                // ISR function is called, using the given
                        command,it uses a continuous procedure of
                        testing when the data is ready
}
```

Exercise: Modify your code to generate hamming code for six data bits.

21. Use the following code to detect a (7,4) hamming code errors and correct it if there is one bit error.

```
#include "dsk6713_aic23.h"              //this file is added to initialize the DSK6713
#include<stdio.h>                               // stdio.h, which stands for "standard
input/output header", is the header in the C standard library that contains macro definitions,
constants, and declarations of functions and types used for various standard input and output
operations.
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;        // set sampling frequency
void main()
{
        int c[7],A1,A2,A3,eb,i;
        printf("\n Enter the 7 bit information: \n");       // these line is printed on the CCS
screen
        for(i=0;i<7;i++)
        scanf("%d",&c[i]);          // Enter the value of 7-bit entered by the user

        A1=(c[0]+c[2]+c[4]+c[6])%2;
        A2=(c[1]+c[2]+c[5]+c[6])%2;
        A3=(c[3]+c[4]+c[5]+c[6])%2;
```

```
        eb=A1+2*A2+4*A3;

        if (eb!=0)
         {
        c[eb-1]=(c[eb-1]+1)%2;
        printf("\n The error is in bit number  %d \n", eb);
         }
        else
        printf("\nthere is no error\n");
        printf("%d",c[2]);
        printf("%d",c[4]);
        printf("%d",c[5]);
        printf("%d",c[6]);

        printf("\n This is the corrected received data\n");
}

interrupt void c_int11() // ISR call, At each Interrupt, program execution goes to the interrupt
service routine
{
        comm_poll();                    // ISR function is called, using the given command,it
uses a continuous procedure of testing when the data
is ready
}
```

Exercise:Modify your code to detect hamming code errors for six data bits and correct it if there is one bit error.

10 Experiment 10Digital Image Processing using MATLAB - 1

Objectives:

1- To be familiarized with image reading and displaying using MATLAB
2- To apply different arithmetic operations on images using MATLAB
3- To apply different spatial filters on images using MATLAB

Introduction

An image is a two-dimensional function f(x,y), where x and y are the spatial (plane) coordinates, and the amplitude of at any pair of coordinates (x,y) is called the intensity of the image at that level.

If x,y and the amplitude values of f are finite and discrete quantities, we call the image a digital image. A digital image is composed of a finite number of elements called pixels, each of which has a particular location and value. The field of digital image processing refers to processing digital images by means of a digital computer.

Aspects of image processing:

It is convenient to subdivide different image processing algorithms into broad subclasses. There are algorithms for different tasks and problems, and often we would like to distinguish the nature of the task at hand.

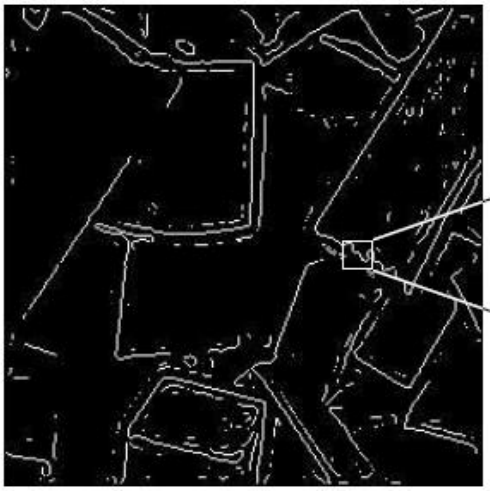
1- Image enhancement. This refers to processing an image so that the result is more suitable for a particular application. Example include:
   a- sharpening or de-blurring an out of focus image, highlighting edges,
   b- improving image contrast, or brightening an image,
   c- Removing noise.
2- Image restoration. This may be considered as reversing the damage done to an image by aknown cause, for example:
   a- removing of blur caused by linear motion,
   b- removal of optical distortions,
   c- Removing periodic interference.
3- Image segmentation. This involves subdividing an image into constituent parts, or isolating certain aspects of an image:
   a- finding lines, circles, or particular shapes in an image,
   b- Identifying cars, trees, buildings, or roads.

Types of digital images

We shall consider threemain types of images:

1- Binary. Each pixel is just black or white. Since there are only two possible values for each pixel,we only need one bit per pixel. Such images can therefore be very efficient in terms ofstorage. Images for which a binary representation may be suitable include text (printed or handwriting), Fingerprints, or architectural plans.

In this image, we have only the two colours: white for the edges, and black for the background.



1	1	0	0	0	0
0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	0	0	0	1

2- Grey scale: Each pixel is a shade of grey, normally from (black) to (white). This rang means that each pixel can be represented by eight bits, or exactly one byte. This is a very natural range for image file handling. Such images arise in medicine (X-rays), images of printed works, and indeed 256 different grey levels are sufficient for the recognition of most natural objects.

230	229	232	234	235	232	148
237	236	236	234	233	234	152
255	255	255	251	230	236	161
99	90	67	37	94	247	130
222	152	255	129	129	246	132
154	199	255	150	189	241	147
216	132	162	163	170	239	122

3- True colour, or RGB:

Here each pixel has a particular colour; that colour being described by theamount of red, green and blue in it. If each of these components has a range 0_255, this givesa total of 255^3= 16,777,216 different possible colours in the image. This is enough coloursfor any image. Since the total number of bits required for each pixel is 24 bits, such images arealso called 24-bit colour images.
Such an image may be considered as consisting of a stack of three matrices; representing thered, green and blue values for each pixel. This means that for every pixel there correspondthree values.

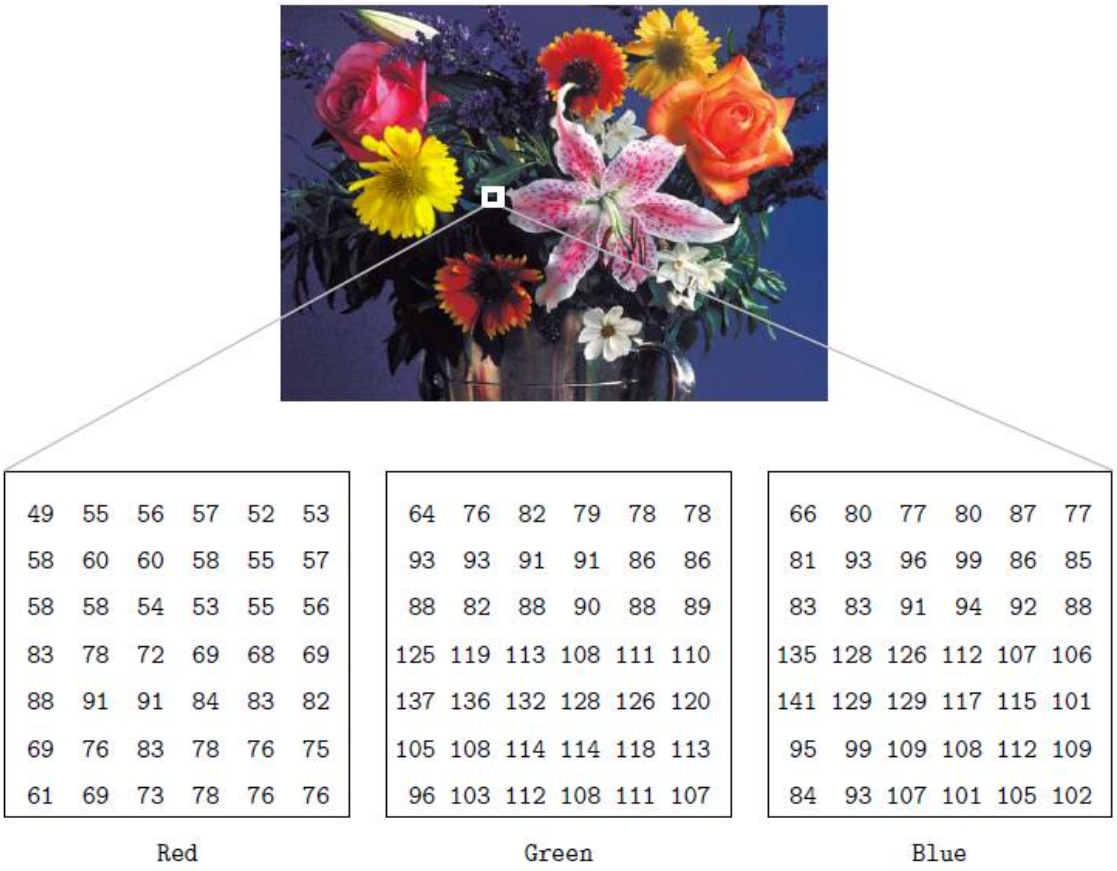


Image File Sizes

Image files tend to be large. We shall investigate the amount of information used in different imagetype of varying sizes. For example, suppose we consider a 512x512 binary image. The number ofbits used in this image (assuming no compression, and neglecting, for the sake of discussion, anyheader information) is

$$\begin{aligned} 512 \times 512 \times 1 &= 262,144 \\ &= 32768 \text{ bytes} \\ &= 32.768 \text{ Kb} \\ &\approx 0.033\text{Mb.} \end{aligned}$$

A grey scale image of the same size requires:

$$\begin{aligned} 512 \times 512 \times 1 &= 262,144 \text{ bytes} \\ &= 262.14 \text{ Kb} \\ &\approx 0.262 \text{ Mb.} \end{aligned}$$

If we now turn our attention to colour images, each pixel is associated with 3 bytes of colourinformation. A 512x512 image thus requires:

$$\begin{aligned} 512 \times 512 \times 3 &= 786,432 \text{ bytes} \\ &= 786.43 \text{ Kb} \\ &\approx 0.786 \text{ Mb.} \end{aligned}$$

Images and Matlab:

10.1.1 Grayscale images

Type in the command
>> w=imread('cameraman.tif');

This takes the grey values of all the pixels in the grayscale image cameraman.tif and puts them all into a matrix w. This matrix w is now a Matlab variable, and so we can perform various matrix operations on it. In general the imread function reads the pixel values from an image and returns a matrix of all the pixel values.

Now we can display this matrix as a gray scale image:
>>figure,imshow(w),impixelinfo

This is really three commands on the one line.
figure, which creates a figure on the screen. imshow(g), which displays the matrix g as an image. impixelinfo, which turns on the pixel values in our figure. This is a display of the grey values of the pixels in the image. They appear at the bottom of the figure in the form

$$c \times r = p$$

Where $c$is the column value of the given pixel; r its row value, and $p$its grey value.

10.1.2 RGB Images

Matlab handles 24-bit RGB images in much the same way as gray scale. We can save the colour values to a matrix and view the result:
>> a=imread('autumn.tif');
>>figure,imshow(a),impixelinfo
Note now that the pixel values now consist of a list of three values, giving the red, green and blue components of the colour of the given pixel.
An important deference between this type of image and a gray scale image can be seen by the command
>>size(a)
which returns three values: the number of rows, columns, and pagesof a,
A useful function for obtaining RGB values is impixel; the command
>>impixel(a,200,100)
returns the red, green, and blue values of the pixel at column 200, row 100.

Information about your image
A great deal of information can be obtained with the imfinfo function. For example, suppose we take our indexed image emu.tif from above.
>>imfinfo('flowers.tif')

```
ans =

                Filename: 'C:\Users\yafa\Desktop\images\flowers.tif'
FileModDate: '07-Apr-2016 20:55:34'
                FileSize: 543962
Format: 'tif'
FormatVersion: []
Width: 500
Height: 362
                BitDepth: 24
ColorType: 'truecolor'
FormatSignature: [73 73 42 0]
ByteOrder: 'little-endian'
```

```
NewSubFileType: 0

BitsPerSample: [8 8 8]

Compression: 'Uncompressed'

PhotometricInterpretation: 'RGB'

StripOffsets: [73x1 double]

SamplesPerPixel: 3

RowsPerStrip: 5

StripByteCounts: [73x1 double]

XResolution: 72

YResolution: 72

ResolutionUnit: 'None'

                Colormap: []

PlanarConfiguration: 'Chunky'

TileWidth: []

TileLength: []

TileOffsets: []
```

10.1.3 Data types and conversions

Elements in Matlab matrices may have a number of different numeric data types; the most common are listed in table 3.1. There are others; see the help for data types. These data types are also

Data type	Description	Range
int8	8-bit integer	−128 — 127
uint8	8-bit unsigned integer	0 — 255
int16	16-bit integer	−32768 — 32767
uint16	16-bit unsigned integer	0 — 65535
double	Double precision real number	Machine specific

They are of different data types. An important consideration (of which we shall more) is that arithmetic operations are not permitted with the data types int8, int16, uint8 and uint16.

Image Display

10.1.4 The imshow function

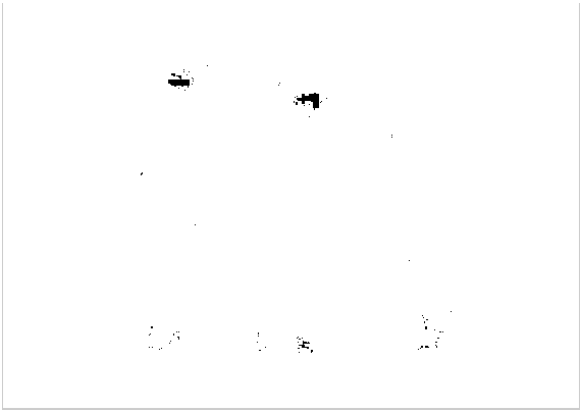
We have seen that if x is a matrix of type uint8, then the command
>>imshow(x)
Lots of Matlab image processing commands produces output matrices which are of type double. We have two choices with a matrix of this type:

1. Convert to type uint8 and then display,
2. Display the matrix directly.

The second option is possible because imshow will display a matrix of type double as a gray scale image as long as the matrix elements are between 0 and 1. Suppose we take an image and convert it to type double:

```
>> c=imread('Penguins.tif');
>>cd=double(c);
>>imshow(c),figure,imshow(cd)
```

a) The original image



b) After conversion to type double

However, as you can see, the figure doesn't look much like the original picture at all! This is because for a matrix of type double, the imshow function expects the values to be between 0 and 1, where 0 is displayed as black, and 1 is displayed as white.
Conversely, values greater than 1 will be displayed as 1 (white) and values less than 0 will be displayed as zero (black). In the Penguins image, every pixel has value greater than or equal to 1 (in fact the minimum value is 21), so that every pixel will be displayed as white. To display the matrix cd, we need to scale it to the range 0-1. This is easily done simply by dividing all values by 255:

```
>>imshow(cd/255)
```

We can vary the display by changing the scaling of the matrix. Results of the commands:

>>imshow(cd/512)
>>imshow(cd/128)
Dividing by 512 darkens the image, as all matrix values are now between 0 and 0.5, so that the brightest pixel in the image is a mid-grey. Dividing by 128 means that the range is 0-2, and all pixels in the range 1-2 will be displayed as white. Thus the image has an over-exposed, washed-out appearance.
We can convert the original image to double more properly using the function im2double. This applies correct scaling so that the output values are between 0 and 1. So the commands
>>cd=im2double(c);
>>imshow(cd)

10.1.5 Spatial Resolution

Spatial resolution is the density of pixels over the image: the greater the spatial resolution, the more pixels are used to display the image. We can experiment with spatial resolution with Matlab's imresize function. Suppose we have a 256x256 8-bit grayscale image saved to the matrix x. Then the command

>>imresize(x,1/2);

will halve the size of the image. It does this by taking out every odd row and every odd column, thus leaving only those matrix elements whose row and column indices are even:

$$\begin{matrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & \cdots \\ x_{21} & \boxed{x_{22}} & x_{23} & \boxed{x_{24}} & x_{25} & \boxed{x_{26}} & \cdots \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & \cdots \\ x_{41} & \boxed{x_{42}} & x_{43} & \boxed{x_{44}} & x_{45} & \boxed{x_{46}} & \cdots \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & \cdots \\ x_{61} & \boxed{x_{62}} & x_{63} & \boxed{x_{64}} & x_{65} & \boxed{x_{66}} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{matrix} \longrightarrow \texttt{imresize(x,1/2)} \longrightarrow \begin{matrix} x_{22} & x_{24} & x_{26} & \cdots \\ x_{42} & x_{44} & x_{46} & \cdots \\ x_{62} & x_{64} & x_{66} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

The effective resolution of this new image is only 128x128. We can do all this in one line:

>>x2=imresize(imresize(x,1/2),2);

Command	Effective resolution
`imresize(imresize(x,1/4),4);`	64×64
`imresize(imresize(x,1/8),8);`	32×32
`imresize(imresize(x,1/16),16);`	16×16
`imresize(imresize(x,1/32),32);`	8×8

To see the effects of these commands, suppose we apply them to the image newborn.tif:

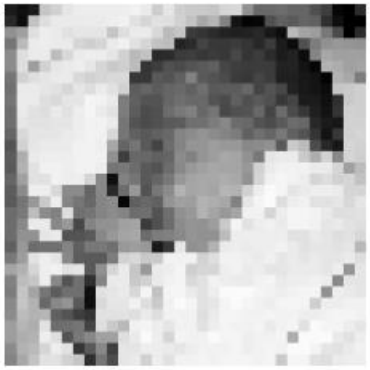
The effects of increasing blockiness or pixelization become quite pronounced as the resolution decreases; even at 128x128 resolution fine detail, such as the edges of the baby's fingers, are less

clear, and at 64x64 all edges are now quit blocky. At 32x32 the image is barely recognizable and at 16x16 and 8x8 the image becomes unrecognizable.

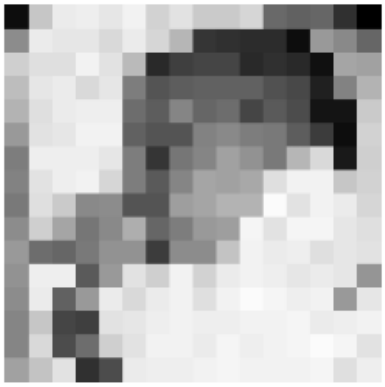
(a) The original image

(b) at 128 × 128 resolution

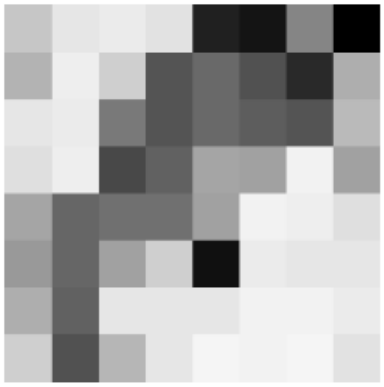
(a) At 64 × 64 resolution



(b) At 32 × 32 resolution



(a) At 16 × 16 resolution



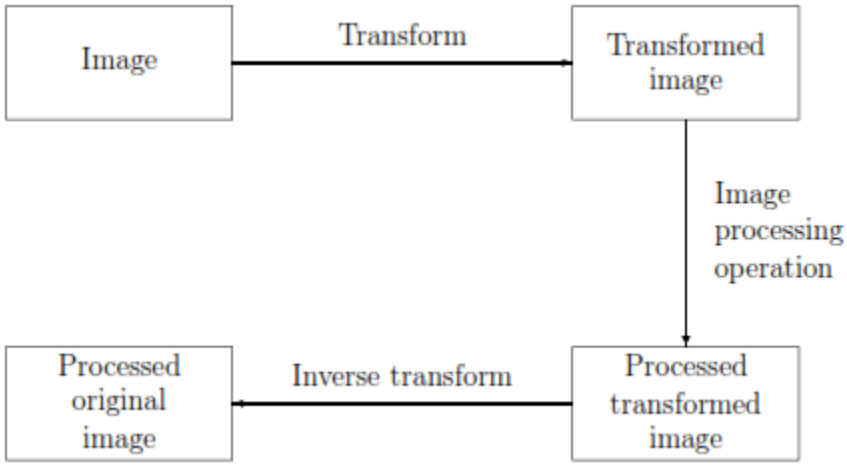
(b) at 8 × 8 resolution

Exercise: See the effect on the image 'girl.tif'

Point Processing

Any image processing operation transforms the gray values of the pixels. However, image processing operations may be divided into three classes based on the information required to perform the transformation. From the most complex to the simplest, they are:

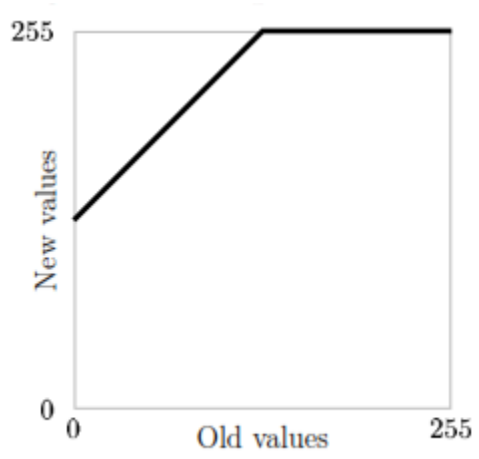
1. Transforms. We require knowledge of all the grey levels in the entire image to transform the image. In other words, the entire image is processed as a single large block. This may be illustrated by the diagram shown



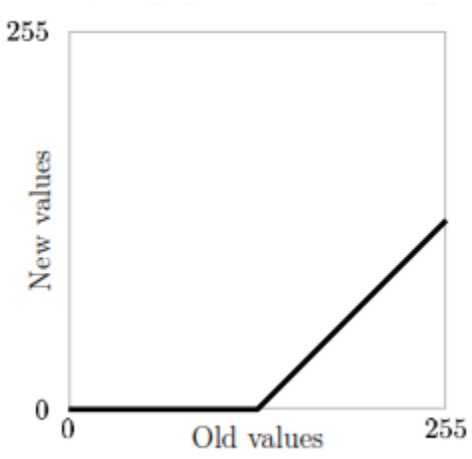
2. Spatial filters. To change the grey level of a given pixel we need only know the value of the gray levels in a small neighborhood of pixels around the given pixel.
3. Point operations. A pixel's grey value is changed without any knowledge of its surrounds.

10.1.6 Arithmetic operations

We can obtain an understanding of how these operations affect an image by looking at the graph of old grey values against new values. Figure below shows the result of adding or subtracting 128 from each pixel in the image. Notice that when we add 128, all grey values of 127 or greater will be 255.



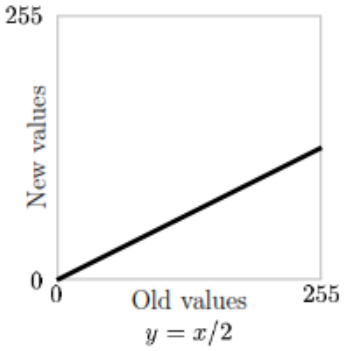
Adding 128 to each pixel



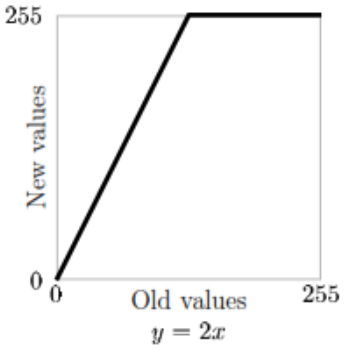
Subtracting 128 from each pixel

We can test this on the 'rice' image rice.png; we start by reading the image in:
>> b=imread('rice.png');
>> b1=imadd(b,128);
Subtraction is similar using the imsubtract function:
>> b2=imsubtract(b,128);
And now we can view them:
>>imshow(b1),figure,imshow(b2)

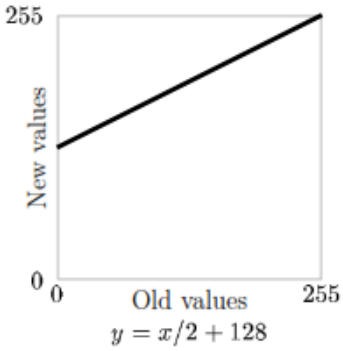
We can also perform lightening or darkening of an image by multiplication; figure 5.4 shows some examples of functions which will have these effects.



$y = x/2$



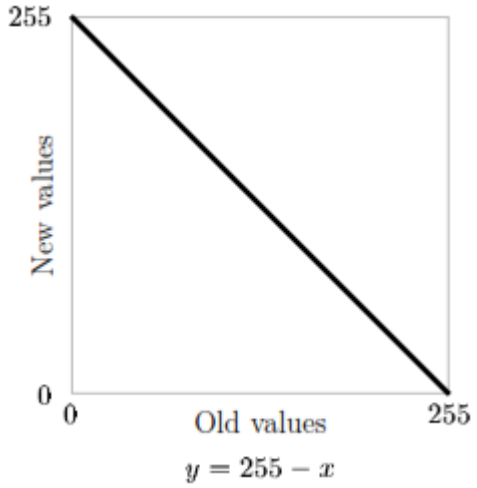
$y = 2x$

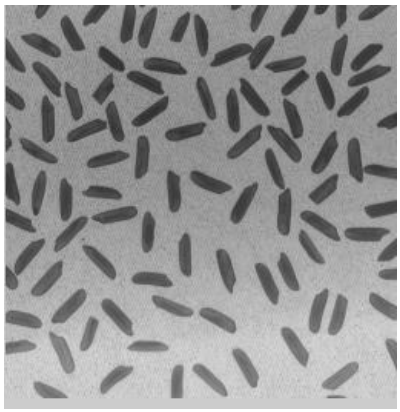


$y = x/2 + 128$

>> b3=immultiply(b,0.5); or b3=imdivide(b,2)
>> b4=immultiply(b,2);
>>b5=imadd(immultiply(b,0.5),128); or imadd(imdivide(b,2),128);

Complements:
The complement of a grayscale image is its photographic negative.
If the image is of type uint8, the best approach is the imcomplement function. Figure below shows the complement function y=255-x, and the result of the commands
>>bc=imcomplement(b);
>>imshow(bc)



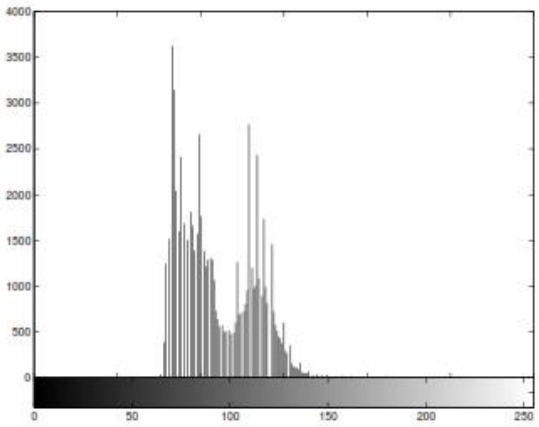


10.1.7 Histograms and Histogram equalization

Given a grayscale image, its histogram consists of the histogram of its grey levels; that is, a graph indicating the number of times each grey level occurs in the image. We can infer a great deal about the appearance of an image from its histogram, as the following examples indicate:

- In a dark image, the grey levels (and hence the histogram) would be clustered at the lower end:
- In a uniformly bright image, the grey levels would be clustered at the upper end:
- In a well contrasted image, the grey levels would be well spread out over much of the range:

We can view the histogram of an image in Matlab by using the imhist function:
>> p=imread('pout.tif');
>>imshow(p),figure,imhist(p)



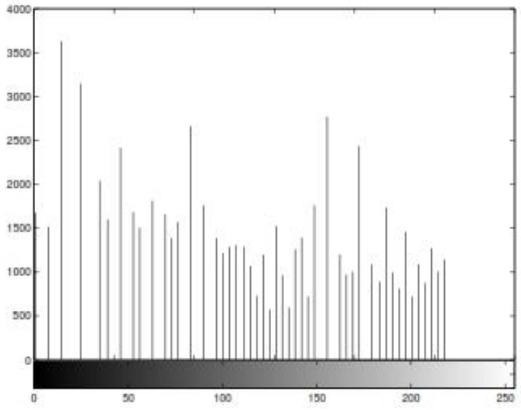
Since the gray values are all clustered together in the centre of the histogram, we would expect the image to be poorly contrasted, as indeed it is.
Given a poorly contrasted image, we would like to enhance its contrast, by spreading out its histogram using histogram equalization.

Histogram equalization is an entirely automatic procedure.
Suppose our image has L different grey levels 0,1,2 …. L-1 and that grey level i occurs n_i times in the image. Suppose also that the total number of pixels in the image is n so that ($n_0 + n_1 + n_2 + \cdots n_{L-1} = n$). To transform the grey levels to obtain a better contrasted image, we change grey level i to

$$\left(\frac{n_0 + n_1 + \cdots + n_i}{n}\right)(L-1).$$

And this number is rounded to the nearest integer.
The command:

```
>>ch=histeq(p);
>>imshow(ch),figure,imhist(ch)
```



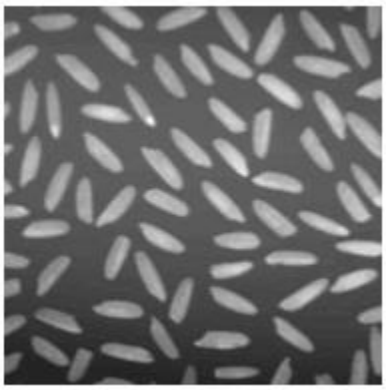
10.1.8 Thresholding

A gray scale image is turned into a binary (black and white) image by first choosing a grey level T in the original image, and then turning every pixel black or white according to whether its grey value is greater than or less than T:

$$\text{A pixel becomes } \begin{cases} \text{white if its grey level is } > T, \\ \text{black if its grey level is } \leq T. \end{cases}$$

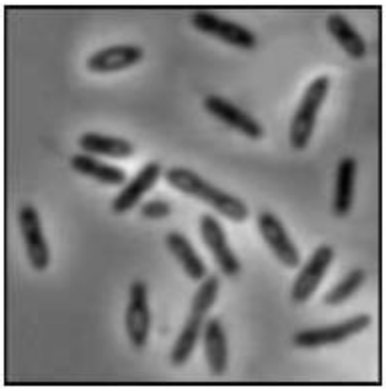
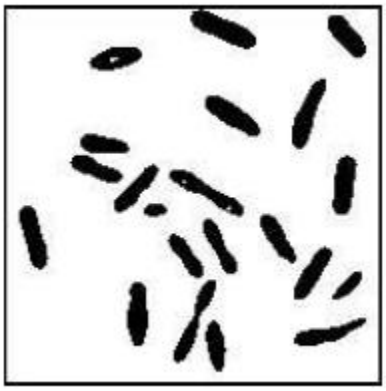
Thresholding is a vital part of image segmentation, where we wish to isolate objects from the background.

Thresholding can be done very simply in Matlab, the command
X>T
will perform the thresholding. We can view the result with imshow. For example, the commands:
>> r=imread('rice.png');
>>imshow(r),figure,imshow(r>120)




The rice image shown above has light grains on a dark background; an image with dark objects over a light background may be treated the same:

>> b=imread('bacteria.tif');
>>imshow(b),figure,imshow(b>80)

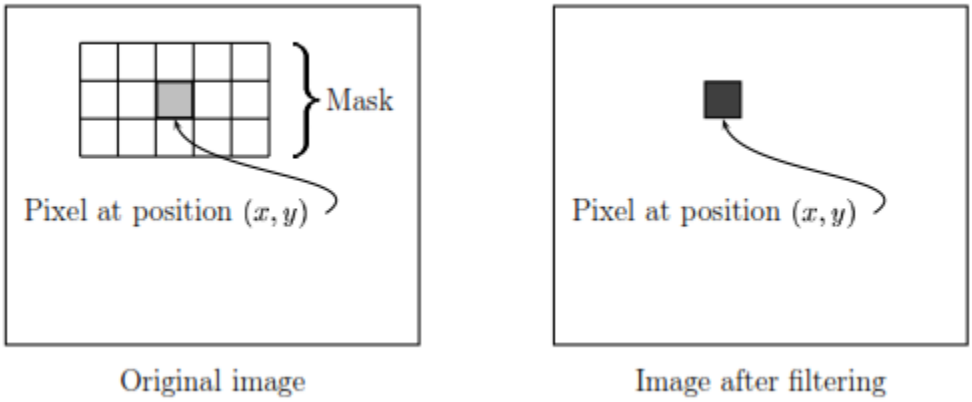
As well as the previous method, Matlab has the im2bw function, which thresholds an image of any data type, using the general syntax
>>im2bw(image,level)
where level is a value between 0 and 1 (inclusive), indicating the fraction of grey values to be turned white. For example, the thresholded rice and bacteria images above could be obtained using

>> im2bw(r,0.43);
>>im2bw(b,0.39);

Exercise: test on 'txt.tif'

Spatial Filtering

We have seen in part 3 that an image can be modified by applying a particular function to each pixel value. Spatial filtering may be considered as an extension of this, where we apply a function to a neighborhood of each pixel.
The idea is to move a mask: a rectangle (usually with sides of odd length) or other shape overthe given image. As we do this, we create a new image whose pixels have grey values calculated from the grey values under the mask, as shown in figure below. The combination of mask and function is called a filter. If the function by which the new grey value is calculated is a linear function of all the grey values in the mask, then the filter is called a linear filter.

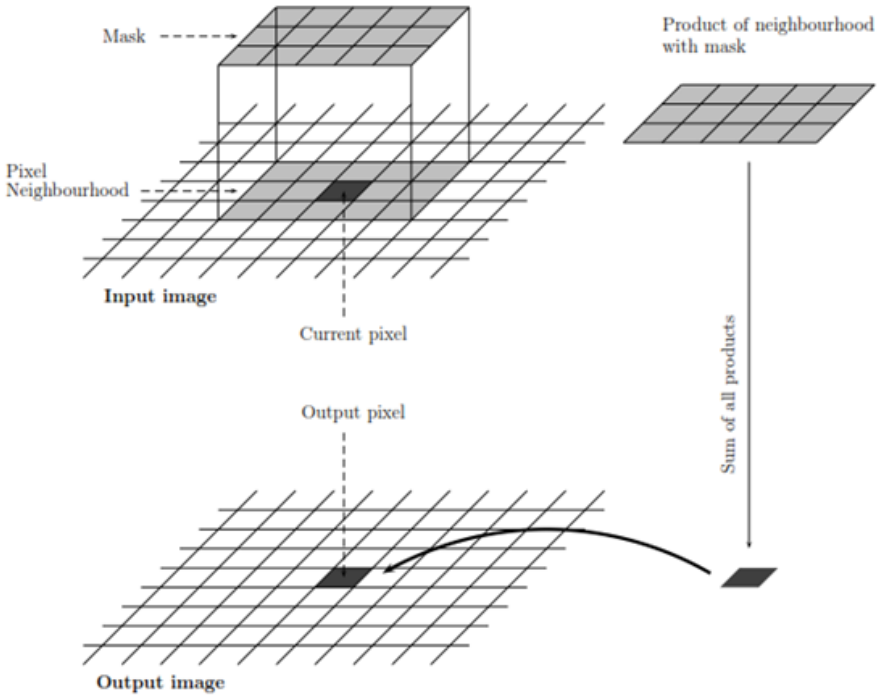


Using a spatial mask on an image

We can implement a linear filter by multiplying all elements in the mask by corresponding elements in the neighborhood, and adding up all these products.
A diagram illustrating the process for performing spatial filtering is given in next figure. We see that spatial filtering requires three steps:

1. Position the mask over the current pixel,
2. Form all products of filter elements with the corresponding elements of the neighborhood,
3. Add up all the products.

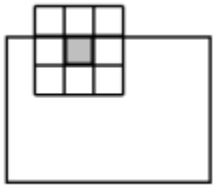
This must be repeated for every pixel in the image.



Performing spatial filtering

Edges of the image

There is an obvious problem in applying a filter what happens at the edge of the image, where the mask partly falls outside the image? In such a case, as illustrated in figure below there will be a lack of grey values to use in the filter function.



A mask at the edge of an image

There are a number of different approaches to deal with this problem:

1- Ignore the edges. That is, we only apply the mask to those pixels in the image for with the mask will lie fully within the image. This means all pixels except for the edges, and results in an output image which is smaller than the original. If the mask is very large, we may lose a significant amount of information by this method.

2- Pad with zeros. We assume that all necessary values outside the image are zero. This gives us all values to work with, and will return an output image of the same size as the original, but may have the effect of introducing unwanted artifacts (for example, edges) around the image.

10.1.9 Filtering in Matlab

The filter2 function does the job of linear filtering for us; its use is
>>filter2(filter,image,shape)
and the result is a matrix of data type double. The parameter shape is optional; it describes the method for dealing with the edges (by default it uses zero padding method).

We can create our filters by hand, or by using the fspecial function; this has many options which makes for easy creation of many different filters. We shall use the average option, which produces averaging filters of given size; thus

>>fspecial('average',[5,7])
will return an averaging filter of size 5x7 , more simply:
>>fspecial('average',11)
will return an averaging filter of size 11x11 . If we leave out the final number or vector, the 3x3 averaging filter is returned.

For Example, suppose we apply 3x3 averaging filter to an image as follows:

```
>> c=imread('cameraman.tif');
>> f1=fspecial('average');
>> cf1=filter2(f1,c);
>>figure,imshow(c),figure,imshow(cf1/255)
```

The averaging filter blurs the image. The image can be further blurred by using an averaging filter of larger size.

(a) Original image (b) Average filtering

(c) Using a 9×9 filter (d) Using a 25×25 filter

Notice how the zero padding used at the edges has resulted in a dark border appearing around the image. This is especially noticeable when a large filter is being used.

10.1.10 Frequencies; low and high pass filters

Fundamentally, the frequencies of an image are the amount by which grey values change with distance. High frequency components are characterized by large changes in grey values over small distances; examples of high frequency components are edges and noise. Low frequency components, on the other hand, are parts of the image characterized by little change in the grey values. These may include backgrounds, skin textures.

- **high pass filter:** if it passes over the high frequency components, and reduces or eliminates low frequency components,
- **low pass filter:** if it passes over the low frequency components, and reduces or eliminates high frequency components,

For example, the 3x3 averaging filter is low pass filter since it tends to blur edges.
The filter below is a high pass filter:

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

We note that the sum of the coefficients (that is, the sum of all e elements in the matrix), in the high pass filter is zero. This means that in a low frequency part of an image, where the grey values are similar, the result of using this filter is that the corresponding grey values in the new image will be close to zero. High pass filters are of particular value in edge detection and edge enhancement

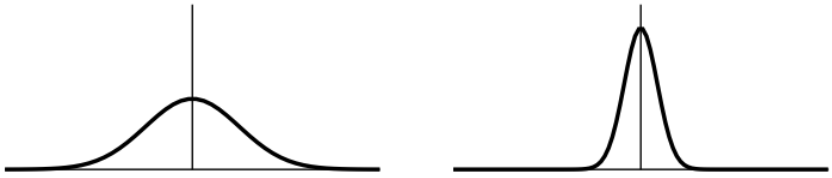
Exercise: Apply high pass filter matrix and averaging filter to 'cameraman.tif' image and 'girl.jpg'.

10.1.11 Gaussian filters

Gaussian filters are a class of low-pass filters, all based on the Gaussian probability distribution function

$$f(x) = e^{-\frac{x^2}{2\sigma^2}}$$

Where σ the standard deviation is: A large σ produces to a flatter curve, and a small value leads to a pointier curve. The figure shows examples of such one dimensional Gaussian.



Large value of σ Small value of σ

A two dimensional Gaussian function is given by

$$f(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Gaussian filters have a blurring effect which looks very similar to that produced by neighborhood averaging. Let's experiment with the cameraman image, and some different Gaussian filters.

```
>> g1=fspecial('gaussian',[5,5]);

>> g2=fspecial('gaussian',[5,5],2);

>> g3=fspecial('gaussian',[11,11],1);

>> g4=fspecial('gaussian',[11,11],5);

>> imshow(filter2(g1,c)/256), figure,imshow(filter2(g2,c)/256)

>>figure,imshow(filter2(g3,c)/256), figure,imshow(filter2(g4,c)/256)
```

11 Experiment 11 Digital Image Processing using MATLAB - 2

Objectives:

4- To define different types of noise that corrupts the images using MATLAB.
5- To apply different methods of image restoration depending on the noise type affected the image using MATLAB
6- To use GUI in MATLAB for image processing applications.

Part1: Noise and Image Restoration

Introduction

We may define noise to be any degradation in the image signal, caused by external disturbance. If an image is being sent electronically from one place to another, via satellite or wireless transmission, or through networked cable, we may expect errors to occur in the image signal. These errors will appear on the image output in different ways depending on the type of disturbance in the signal. Usually we know what type of errors to expect, and hence the type of noise on the image; hence we can choose the most appropriate method for reducing the effects. Cleaning an image corrupted by noise is thus an important area of image restoration.

Types of noise:

1- Salt and pepper noise: Also called impulse noise, shot noise, or binary noise. This degradation can be caused by sharp, sudden disturbances in the image signal; its appearance is randomly scattered white or black (or both) pixels over the image.

2- Gaussian noise: Gaussian noise is an idealized form of white noise, which is caused by random fluctuations in the signal. We can observe white noise by watching a television which is slightly mistuned to a particular channel. Gaussian noise is white noise which is normally distributed. If the image is represented as I, and Gaussian noise by N, then we can model a noisy image by simply adding the two:

$$I + N$$

Here we may assume that I is a matrix whose elements are the pixel values of our image, and N is a matrix whose elements are normally distributed. It can be shown that this is an appropriate model for noise.

3- Speckle noise: Whereas Gaussian noise can be modelled by random values added to an image; speckle noise (or more simply just speckle) can be modelled by random values multiplied by pixel

values, hence it is also called multiplicative noise. Speckle noise is a major problem in some radar applications.

Although Gaussian noise and speckle noise appear superficially similar, they are formed by two totally different methods, and, as we shall see, require different approaches for their removal.

4- Periodic noise: If the image signal is subject to a periodic, rather than a random disturbance, we might obtain an image corrupted by periodic noise.

Salt and pepper noise, Gaussian noise and speckle noise can all be cleaned by using spatial filtering techniques. Periodic noise, however, requires image transforms for best results.

Image Restoration:

1- Cleaning salt and pepper noise:

a- Low pass filtering: Given that pixels corrupted by salt and pepper noise are high frequency components of an image, we should expect a low-pass filter should reduce them.

b- Median filtering: Median filtering seems almost tailor-made for removal of salt and pepper noise. Recall that the median of a set is the middle value when they are sorted. If there are an even number of values, the median is the mean of the middle two. A median filter is an example of a non-linear spatial filter; using a 3x3 mask, the output value is the median of the values in the mask. For example:

50	65	52
63	255	58
61	60	57

→ 50 52 57 58 **60** 61 63 65 255 → 60

We see that very large or very small values (noisy values) will end up at the top or bottom of the sorted list. Thus the median will in general replace a noisy value with one closer to its surroundings.

c- Rank-order filtering.

d- An outlier method.

2- Cleaning Gaussian noise:

a- Image averaging: It may sometimes happen that instead of just one image corrupted with Gaussian noise, we have many different copies of it. An example is satellite imaging; if a satellite passes over the same spot many times, we will obtain many different images of the same place. Another example is in microscopy: we might take many different images of the same object. In such a case a very simple approach to cleaning Gaussian noise is to simply take the average (the mean) of all the images.

To see why this works, suppose that we have 100 copies of our image; each with noise; then the i-th noisy image will be: $M + N_i$

where M is the matrix of original values, and N_i is a matrix of normally distributed random values with mean 0. We can find the mean M' of these images by the usual add and divide method:

$$M' = \frac{1}{100}\sum_{i=1}^{100}(M + N_i)$$

$$= \frac{1}{100}\sum_{i=1}^{100} M \; + \frac{1}{100}\sum_{i=1}^{100} N_i$$

$$= M \; + \frac{1}{100}\sum_{i=1}^{100} N_i$$

Since N_i is normally distributed with mean 0, it can be shown that the mean of all N_i's will be closed to zero- the greater the number of N_i's ; the closer to zero. Thus $M' \approx M$.

b- Average filtering: If the Gaussian noise has mean 0, then we would expect that an average filter would average the noise to 0. The larger the size of the filter mask, the closer to zero. Unfortunately, averaging tends to blur an image. However, if we are prepared to trade off blurring for noise reduction, then we can reduce noise significantly by this method.

c- Wiener filtering: Before we describe this method, we shall discuss a more general question: given a degraded imageM' of some original image M and a restored version R, what measure can we use to say whether our restoration has done a good job? Clearly we would like$R$ to be as closed as possible to the correct image M. One way of measuring the closeness of$R$ to M is by adding the squares of all differences:

$$\sum (m_{ij} - r_{ij})^2$$

where the sum is taken over all pixels of$R$ and M, this sum can be taken as a measure of the closeness of$R$ to M. If we can minimize this value, we may be sure that our procedure has done as good a job as possible. Filters which operate on this principle of least squares are called Wiener

filters. They come in many guises; we shall look at the particular filter which is designed to reduce Gaussian noise.

This filter is a (non-linear) spatial filter; we move a mask across the noisy image, pixel by pixel, and as we move, we create an output image the grey values of whose pixels are based on the values under the mask.

Experimental Procedure:

1- To demonstrate salt and pepper noise appearance, we will first generate a grey-scale image, starting with a colour image:

>>tw=imread('twins.tif');

>> t=rgb2gray(tw);

2- To add noise, we use the Matlab function imnoise, which takes a number of different parameters. To add salt and pepper noise:

>> t_sp=imnoise(t,'salt & pepper');

The amount of noise added defaults to 10%; to add more or less noise we include an optional parameter, being a value between 0 and 1 indicating the fraction of pixels to be corrupted. Thus, for example:

>>imnoise(t,'salt & pepper',0.2);

would produce an image with 20% of its pixels corrupted by salt and pepper noise.

3- To add Guassian noise, using imnoise function:

>> t_ga=imnoise(t,'gaussian');

The gaussian parameter also can take optional values, giving themean and variance of the noise. The default values are 0 and 0.01.

4- To add Speckle noise:

>> t_spk=imnoise(t,'speckle');

Note: In Matlab, spekle noise is implemented by: $I(1+N)$ where I is the image matrix, and N consists of normally distributed values with mean 0. An optional parameter gives the variance of N, its default value is 0.04.

5- To demonstrate periodic noise, we add a periodic matrix (using a trigonometric function), to our image:

>> s=size(t);

>> [x,y]=meshgrid(1:s(1),1:s(2));

>> p=sin(x/3+y/5)+1;

>> t_pn=(im2double(t)+p/2)/2;

6- To reduce salt and pepper noise using low pass filtering, we will try filtering with an average filter:

>> a3=fspecial('average');

>> t_sp_a3=filter2(a3,t_sp)/255;

Notice, however, that the noise is not so much removed over the image; the result is not noticeably "better"than the noisy image.

7- The effect is even more pronounced if we use a larger averaging filter:

>> a7=fspecial('average',[7,7]);

>> t_sp_a7=filter2(a7,t_sp)/255;

8- Toclean salt and pepper noise usingMedian filtering, median filter is implemented by the medfilt2 function:

>> t_sp_m3=medfilt2(t_sp);

9- If we corrupt more pixels with noise:

>> t_sp2=imnoise(t,'salt & pepper',0.2);

then medfilt2 still does a remarkably good job, check:

>> t_sp_m3=medfilt2(t_sp2);

10- To remove salt and pepper noise completely, we can either try a second application of the 3x3 median filter -repeat the last command- , or try a 5x5 median filter on the original noisy image:

>> t_sp2_m5=medfilt2(t_sp2,[5,5]);

11-To demonstrate cleaning Gaussian noise using image averaging with the twins image, We first need to create different versions withGaussian noise, and then take the average of them. We shall create 10 versions. One way is to create an empty three-dimensional array of depth 10, and fill each "level" with a noisy image:

>> s=size(t);

>> t_ga10=zeros(s(1),s(2),10);

>>for i=1:10

t_ga10(:,:,i)=imnoise(t,'gaussian');

end

Note here that the gaussian option of imnoise calls the random number generator randn, which creates normally distributed random numbers. Each time randn is called, it creates a different sequence of numbers. So we may be sure that all levels in our three-dimensional array do indeed contain different images. Then, take the average:

>> t_ga10_av=mean(t_ga10,3)/255;

12- Repeat step 11 by creating 100 versions of noisy images and taking the average of them. Discuss

13- To demonstrate cleaning Gaussian noise using average filtering, apply 3x3 and 5x5 average filer to the noisy image:

>> a3=fspecial('average');

>> a5=fspecial('average',[5,5]);

>> tg3=filter2(a3,t_ga)/255;

>> tg5=filter2(a5,t_ga)/255;

Note that the results are not really particularly pleasing; although there has been some noise reduction, the nature of the resulting images is unattractive.

13- To demonstrate cleaning Gaussian noise using Wiener filtering,we will use the wiener2 function, which can take an optional parameter indicating the size of the mask to be used. The default size is3x3. We shall create four images:

>> t1=wiener2(t_ga);

>> t2=wiener2(t_ga,[5,5]);

>> t3=wiener2(t_ga,[7,7]);

>> t4=wiener2(t_ga,[9,9]);

Note that Wiener filtering does tend to blur edges and high frequency components of the image. But it does a far better job than using a low pass blurring filter.

14- We can achieve very good results for noise where the variance is not as high as that in our current image:

>> t2=imnoise(t,'gaussian',0,0.005);

>>imshow(t2)

>> t2w=wiener2(t2,[7,7]);

>>figure,imshow(t2w)

Part2: GUI for image processing application

Introduction

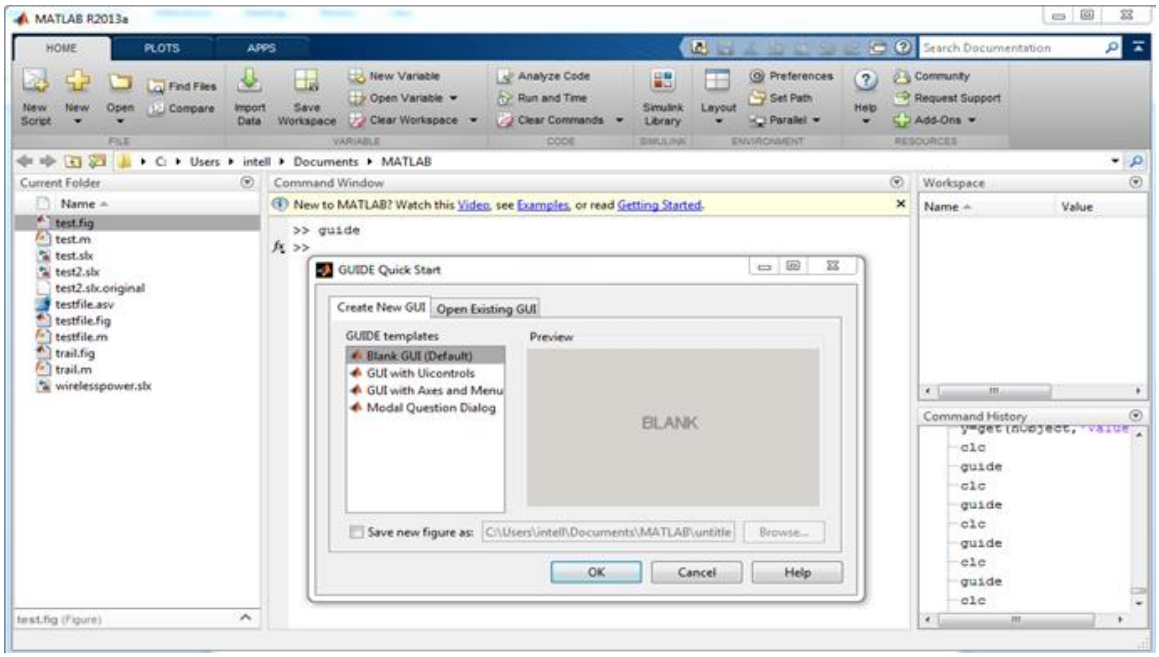
Graphical user interface provides the user an interactive environment.Once the GUI is created the user need not know anything about the coding section. Using GUI we can perform any computations, communicate with any other UI's, create tables etc. MATLAB GUI contains several user interface tools like radio buttons,axes,check box, tables, sliders,list box,panels..etc. GUI interface is an event driven programming. Flow of the application is based on events. Events may be click,double click,key press etc. Callback functions will be executed once an event occurs. We can edit the properties of each callback functions for making suitable response from the GUI as the user interacts.

GUIDE generates two files for each GUI:

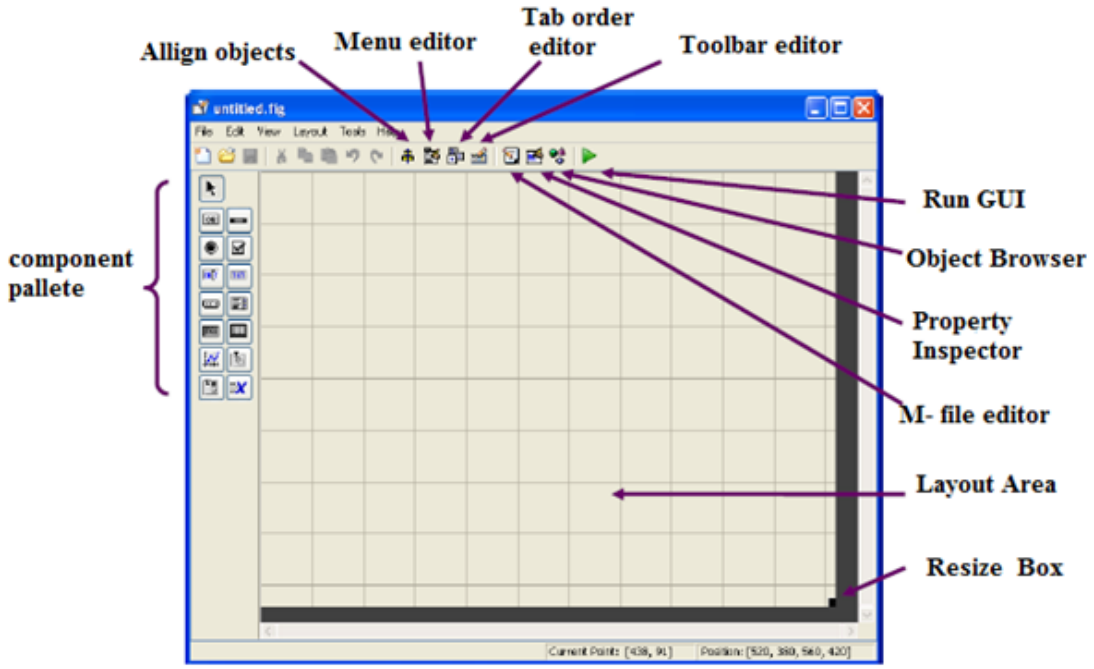
- .fig file: it contains the layout of the GUI.
- .m file: it contains the code that is needed to control GUI behavior.

Experimental Procedure:

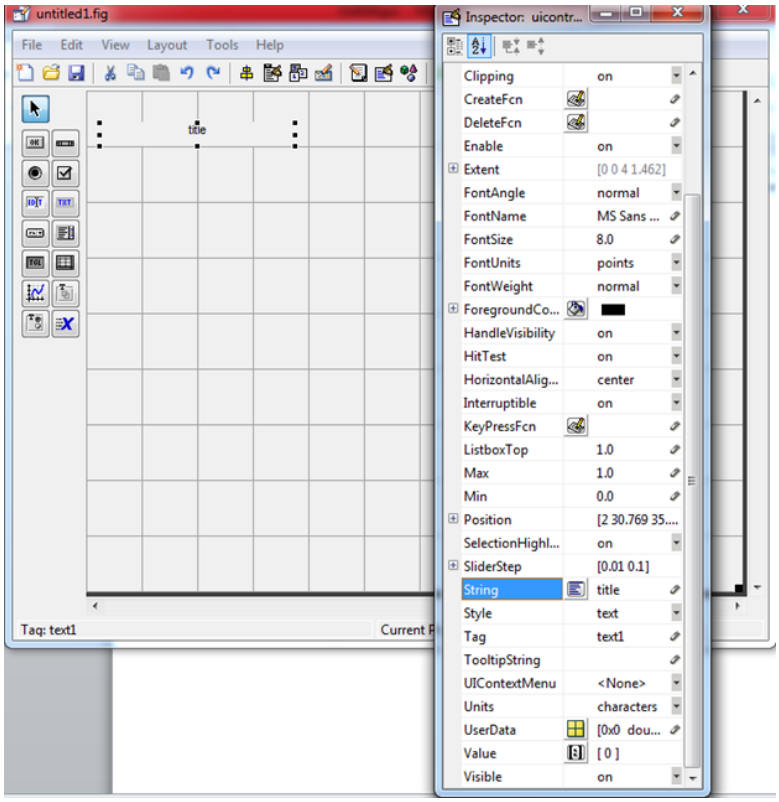
1- Open MATLAB .Type "guide " in Command Window .



2- Select the type of GUI: For that, choose "Blank GUI (Default)" option in the 'GUIDE Quick Start' dialogue box. Click "OK". Following workspace will be displayed.

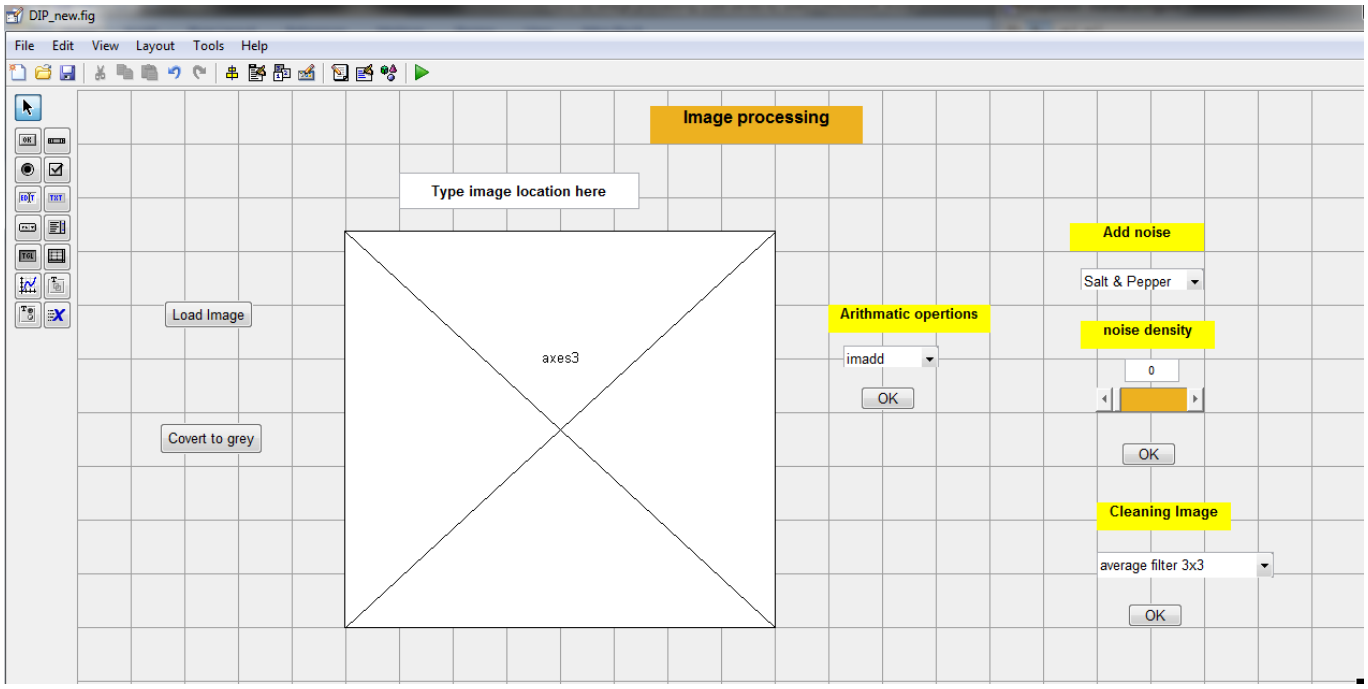


3- Add components: Simply drag and drop items from the component palette to layout area to create your GUI. To edit the properties of the components, right click on the component and select "Property Inspector" or double click on the component.



Note that Tag indicates the name of the component.

4- Let's create a simple image processing tool box using GUI. This tutorial helps you to perform image processing on image selected. Once if you enter the file location of the image to be loaded and click on the LOAD IMAGE push button , selected image will be loaded on the 'Axes ' box on the grid. Effect of different types of noise on the image can be analyzed using this GUI. Image arithmetic operations are also included in this GUI.

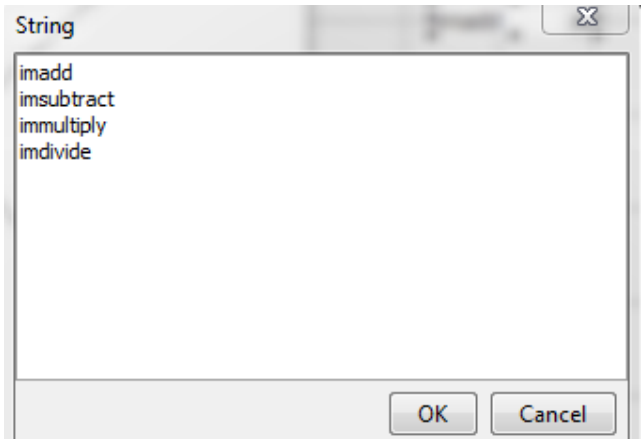


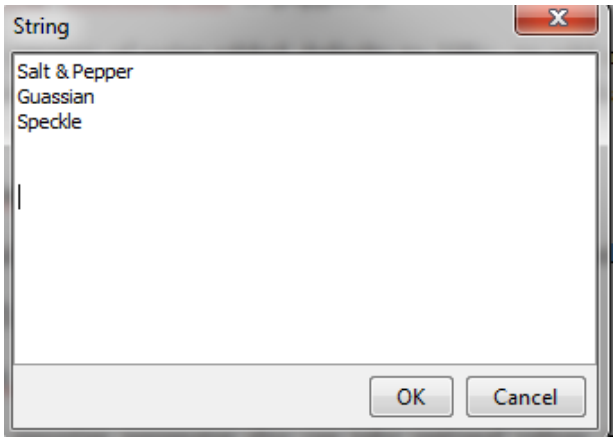
5- Drag and drop static text for the main and partial titles.

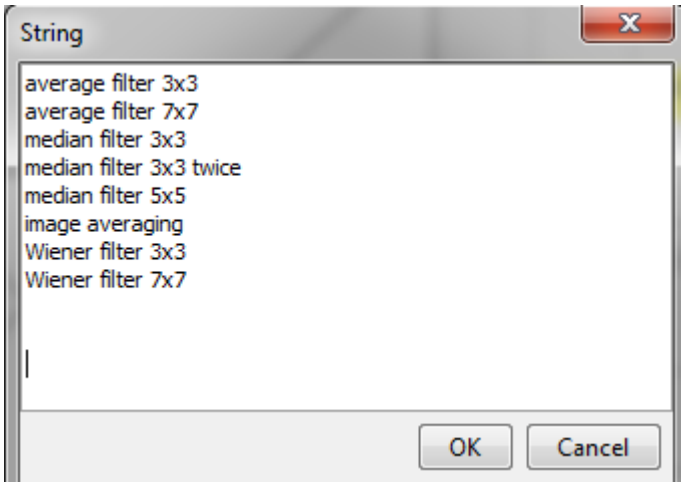
6- Drag and drop edit text to type image location and another one to type slider value.

7- Drag and drop push buttons for loading image, converting to gray and applying operations (arithmetic, adding noise, cleaning noise).

8- Drag and drop Pop-up Menus for arithmetic operations, noise types and cleaning image methods. Add the following list using String property of each.







9- Drag and drop slider for Salt & Pepper noise density . Set MIN to 0 and MAX to 0.5.

10-Save your GUI layout. It will automatically create an callback function corresponding to each components which we added in our design. Callback functions controls the component behavior. By writing an appropriate callback functions we can make the components to respond to user interactions.

11- To Program the push button (Load Image), right click to the button and choose view callbacks → callback. Then add the following commands to handle image location for the edit text and show it in the axes.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
image_file = get(handles.edit1,'String'); %store the text entered in edit
text box1 to  variable image_file.
imshow(image_file);
```

12- To Program the push button (Convert to gray), right click to the button and choose view callbacks → callback. Then add the following commands:

```
function pushbutton2_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
image_file = get(handles.edit1,'String'); %store the text entered in edit
text box1 to  variable image_file.
xx=imread(image_file);
[r c v]=size(xx);
if v==1
    x=xx;
else
x=rgb2gray(xx);
end
imshow(x);
```

13- To Program the push button (OK) for execution of the chosen arithmetic opertion from the related pop-up menu, callback function of that button must read the pop-up menu chosen index and do the required operation:

```
function pushbutton3_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
image_file = get(handles.edit1,'String'); %store the text entered in edit
text box1 to  variable image_file.
xx=imread(image_file);
[r c v]=size(xx);
if v==1
    x=xx;
else
x=rgb2gray(xx);
end
index_selected=get(handles.popupmenu1,'Value');
if index_selected==1
    x_arith=imadd(x,128);
elseif index_selected==2
    x_arith=imsubtract(x,128);
elseif index_selected==3
    x_arith=immultiply(x,2);
else
     x_arith=imdivide(x,2);
end
imshow(x_arith)
```

14- To program the edit text box to handle the changes in the slider value which represent salt & pepper noise density, right click to the slider and choose view callbacks → callback:

```
% --- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
% hObject    handle to slider1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of
slider
z=get(handles.slider1,'Value');
set(handles.edit2,'String', num2str(z));
```

15- To Program the push button (OK) for execution of the chosen noise type from the related pop-up menu and noise density from the slider, callback function of that button must read the pop-up menu chosen index and do the required operation:

```
image_file = get(handles.edit1,'String'); %store the text entered in edit
text box1 to  variable image_file.
xx=imread(image_file);
[r c v]=size(xx);
if v==1
    x=xx;
else
x=rgb2gray(xx);
end
z=get(handles.slider1,'Value');
index_selected=get(handles.popupmenu2,'Value');
if index_selected==1
    x_noise=imnoise(x,'salt & pepper',z);
elseif index_selected==2
    x_noise=imnoise(x,'gaussian');
else
    x_noise=imnoise(x,'speckle');;
end
imshow(x_noise)
```

16- Complete your GUI layout by writing the required commands to execute the different image cleaning methods that listed in the last pop-up menu. Write the callback commands for the last push button.

17- Run and test your image processing layout.

12 Experiment 12Echo Cancelling and Voice Scrambling

Objectives

The main objectives of this experiment are

a) Use the adaptive filter for echo cancelation.
b) Illustrate the concept of voice scrambling.

12.1 Echo canceling using adaptive algorithm

Echo occurs in telephone systems or in acoustic room recording when the voice of a speaker at the far end picked up by a microphone at the near end. Sometimes echo represents undesirable phenomena. One way to cancel this echo is to use an adaptive filter as illustrated in Figure 9.1

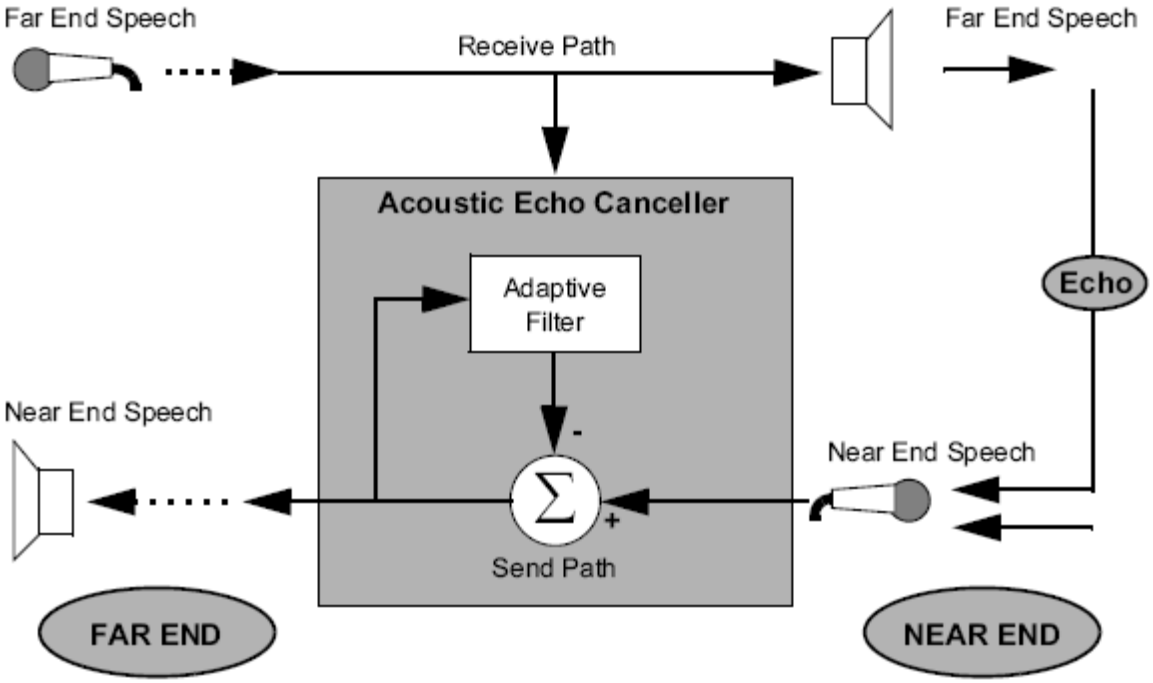


Figure 9.1 echo canceling using adaptive filter

The adaptive filter takes the un-echoed signal from the far end microphone as the desired signal $\boldsymbol{d}(\boldsymbol{n})$. The error signal is defined as the difference between the adaptive filter output and the echoed signal $\boldsymbol{x}[\boldsymbol{n}]$ as given by

$$\boldsymbol{e}[\boldsymbol{n}] = \boldsymbol{x}[\boldsymbol{n}] - \boldsymbol{y}[\boldsymbol{n}]$$

The adaptive filter tries to generate a delayed version of the original un-echoed signal $\boldsymbol{d}[\boldsymbol{n}]$ at it's output. When the error is computed, the error represents a clean signal with no echo.
In order to implement an echo cancelling using an adaptive filter first it is needed n next. This process can be illustrated by the code listed below

```
#include "dsk6713_aic23.h" //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#define N 100                                   // no. of samples
#define beta 1E-8       //# of weights (coefficients)


short w[N];             //weights for adapt filter
short delay[N],gain=1;          //input buffer to adapt filter
short E, yn, echo_output;               // variable declaration
int a;



short input, output, output2;
short bufferlength = 5000; //buffer size for delay
short buffer[5000]; //create buffer
short i = 0, out_type=1;
short amplitude = 5; //to vary amplitude of echo
short Echo_cancellation(short input,short echo_output) //the control passes to the function
Echo_cancellation( ), called by above.& its call by value function
{
        yn=0;                           // store the input signal into a variable
                                // store the echo signal into echo1 variable
        delay[0]=input;                 //noise as input to adapt FIR

        for (a = 0; a < N; a++)       //to calculate out of adapt FIR
        {
                yn += (w[a] * delay[a]);
                }       //echo multiplied with the weights for adapt filter
                E =echo_output-yn ;


        for (a = N-1; a >= 0; a--)          //to update weights and delays
        {
                w[a] = w[a] + beta*E*delay[a];  //update weights
                delay[a] = delay[a-1];          //update delay samples
        }


        return E;       // program execution goes back to the function called and then again starts
listening for next call and this process goes on

}
interrupt void c_int11() //ISR
```

```
{
input = input_sample(); //newest input sample data
output=input + 0.1*amplitude*buffer[i];//newest + oldest samples
output2=Echo_cancellation(input,output);
if (out_type==1)
output_sample(output); //output sample
else if(out_type==2)

output_sample(output2);
buffer[i] = input; //store newest input sample
i++; //increment buffer count
if (i >= bufferlength) i = 0; //if end of buffer reinit
}
main()
{
short T=0;
for (T = 0; T < 100; T++)
{
w[T] = 0; //init buffer for weights
delay[T] = 0; //init buffer for delay samples
}

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

12.2 Voice Scrambling Using Filtering and Modulation (Scrambler)

This exercise 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 unscrambled 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, represented by the band 0.3 to 3 kHz, and “folds” it about a 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. On the lower sideband that represents the audible speech range, the low tones are high tones, and vice versa.

Figure 9.2 is a block diagram of the scrambling scheme. At point A we have a band-limited signal 0 to 3.7 kHz. At point B we have a double-sideband signal with suppressed carrier. At point C the upper sideband is filtered out. Its attractiveness comes from its simplicity, since only simple DSP algorithms are utilized: filtering, and sine generation and modulation.

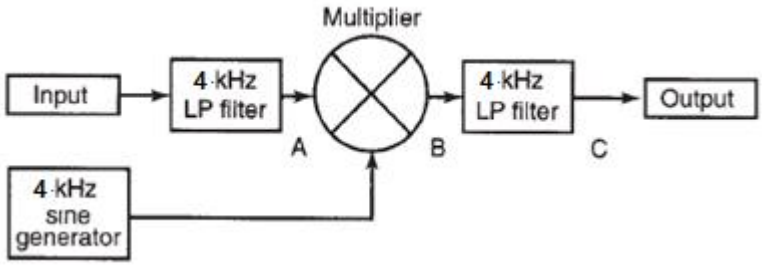


Figure 9.2 Block diagram of scrambler/descrambler scheme

In order to implement voice scrambling you may use the following code

```
//Scrambler.cVoice scrambler/de-scrambler program
#include "dsk6713_aic23.h" //codec-dsk support file
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ; //set sampling rate
#include "sine160.h" //sine data values
#include "lp114.cof" //filter coefficient file
short filtmodfilt(short data);
short filter(short inp,short *dly);
short sinemod(short input);
static short filter1[N],filter2[N];
short input, output;
void main()
{
short i;
comm_poll(); //init DSK using polling
for (i=0; i< N; i++)
{
filter1[i] = 0; //init 1st filter buffer
filter2[i] = 0; //init 2nd filter buffer
}
while(1)
{
input=input_sample(); //input new sample data

output=filtmodfilt(input); //and throw away 1st result
output_sample(output); //then output
}
}
short filtmodfilt(short data) //filtering & modulating
{
data = filter(data,filter1); //newest in ->1st filter
data = sinemod(data); //modulate with 1st filter out
data = filter(data,filter2); //2nd LP filter
return data;
```

```
}
short filter(short inp,short *dly) //implements FIR
{
short i;
int yn;
dly[N-1] = inp; //newest sample @bottom buffer
yn = dly[0] * h[N-1]; //y(0)=x(n-(N-1))*h(N-1)
for (i = 1; i < N; i++) //loop for the rest
{
yn += dly[i] * h[N-(i+1)]; //y(n)=x[n-(N-1-i)]*h[N-1-i]
dly[i-1] = dly[i]; //data up to update delays
}
yn = (yn>>15); //filter's output
return yn; //return y(n) at time n
}
short sinemod(short input) //sine generation/modulation
{
static short i=0;
input=(input*sine160[i++])>>11; //(input)*(sine data)
if(i>= NSINE) i = 0; //if end of sine table
return input; //return modulated signal
}
```

The input signal is first lowpass-filtered and the resulting output (at point A in Figure) is multiplied (modulated) by a 4-kHz sine wave with data values in a buffer (lookup table).The modulated signal (at point B) is filtered again, and the overall output is a scrambled signal (at point C).

There are three functions in the code in addition to the function main. One of the functions, $\boldsymbol{filtmodfilt}()$, calls a filter function to implement the first lowpass filteras an antialiasing filter.The resulting output (filtered input) becomes the input to amultiplier/modulator. The function $\boldsymbol{sinemod}()$ modulates (multiplies) the filtered input with the 4-kHz sine data values. This produces higher and lower sidebandcomponents.The modulated output is again filtered, so that only the lower sidebandcomponents are kept.

A buffer is used to store the 114 coefficients that represent the lowpass filter.Thecoefficient file lp114.cof can be downloaded from your course container. Two other buffers are used for the delaysamples, one for each filter. The samples are arranged in memory as

$$\boldsymbol{x\big(n-(N-1)\big), x\big(n-(N-2)\big), x\big(n-(N-3)\big), \dots, x(n)}$$

with the oldest sample at the beginning of the buffer and the newest sample at theend (bottom) of the buffer.The file sine160.h with 160 data values over 40 cycles your course container. The frequency generated is $\boldsymbol{f = Fs\frac{numberofcycles}{numberofpoints} = \frac{16{,}000(40)}{160} = 4kHz}$.

```
i=1:160;
desired= round(5000*sin(2*pi*(i)*4000/16000)); %sin(4000)
fid=fopen('sine160.h','w');

fprintf(fid,'#ifndef NSINE\n');
fprintf(fid,'#define NSINE 160\n');

fprintf(fid,'short sine160[]={');
fprintf(fid,'%d, ' ,desired(1:159));
fprintf(fid,'%d' ,desired(160));
fprintf(fid,'};\n');
fprintf(fid,'#endif\n');

fclose(fid);
```

Using the resulting output as the input to a second DSK running the same algorithm, the original unscrambled input is recovered as the output of the second DSK.Note that the program can still run on the first DSK when the USB connector cable is removed from the DSK.

12.2.1 Experiment procedures

1. Build and run this project as Scrambler.
2. Connect a 3-kHz input sine wave at the input of the DSK signal.
3. Verify that the resulting output is a lower sideband signal of 1 kHz, obtained as ($\boldsymbol{4kHz - 3kHz}$).
4. Note that the upper sideband signal of ($\boldsymbol{4 + 3kHz}$) is filtered out by the second low pass filter (actually by the antialiasing filter on the codec).
5. Run the same program on a second DSK is used to recover/unscramble the original signal (simulating the receiving end).This produces the reverse procedure, yielding the original unscrambled signal
6. Use the output of the first DSK as the input to the second DSK.
7. Measure the frequency of the signal at the output of the receiving end and explain the results you got. Listen to the output using a speaker connected to the receiving end output.
8. Change the carrier frequency at the receiving end to $\boldsymbol{f_c - f_m}$ (modify the matlab code to generate the new carrier 40 cycles then measure the frequency of the signal at the output of the receiver. Listen to the signal detected at the receiving end output. Explain the results that you got.
9. Repeat the same steps for voice input data instead of the sine wave and explain the results you got.

13 Experiment 13Dual-Tone Multi frequency DTMF

Objectives

To brief student with the operation of the DTMF encoder and decoder

Theory of DTMF

Telephone touch-tone pads generate dual tone multiple frequency (DTMF) signals to dial a telephone. When any key is pressed, the sinusoids of the corresponding row and column frequencies; Table ; are generated and summed, hence dual tone. As an example, pressing the 5 key generates a signal containing the sum of the two tones at 770 Hz and 1336 Hz together. The frequencies in Table 4 were chosen (by the design engineers) to avoid harmonics. No frequency

Frequencies	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Table 4Extended DTMF encoding table for Touch Tone dialing

In this experiment it is desired to write a c program that reads the status of the on board DIP switches and generates a DTMF dial tone according to the following table

Experimental procedures

In order to generate the DTMF signal follow these steps

1. Set the sampling frequency to $f_s = 8\ kHz$
2. Set the number of sample $N = 8000$
3. Write a program to read the DIP switches and generates the required tones as per Table
4. You may consider the binary codes that corresponds to each number as shown in Table

Digit	DIP 0	DIP 1	DIP 2	DIP 3
1	On	Off	Off	Off
2	Off	On	Off	Off
3	On	On	Off	Off
4	Off	Off	On	Off
5	On	Off	On	Off
6	Off	On	On	Off
7	On	On	On	Off
8	Off	Off	Off	On
9	On	Off	Off	On
*	Off	On	Off	On

0	Off	Off	Off	Off
#	On	On	Off	On
A	Off	Off	On	On
B	On	Off	On	On
C	Off	On	On	On
D	On	On	On	On

Table 5

5. Use the following code to generate an ASK modulated signal.

```
    #include "dsk6713_aic23.h"          //this file is added to initialize the DSK6713
#include "math.h"                   //header file used when mathematical instructions are
executed
#include<stdio.h>                   //for input/output files
#define N 8000                          //define no. of samples
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;    //set the sampling frequency, Different sampling
frequencies supported by AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96 kHz.

short sine_table1[N],sine_table2[N],sine_table3[N],sine_table4[N];  //buffer initialization for
sine_wave
short sine_table5[N],sine_table6[N],sine_table7[N],sine_table8[N];
short output,i,gain= 1000;          //variable declaration

interrupt void c_int11()       // ISR call, At each Interrupt, program execution goes to the
interrupt service routine
{
    if((DSK6713_DIP_get(0)==0)&&(DSK6713_DIP_get(1)==0)&&(DSK6713_DIP_get(2)==0)&&
(DSK6713_DIP_get(3)==0))
    {
        output_sample(output);          // the value in the buffer ouput indexed by the
variable loop is written on to the codec.
        output = sine_table1[i] + sine_table5[i]; // 1
        if(i< N-1) ++i;             // the index loop is incremented by an amount equal to N
        else i = 0;                     // if i is greater than the N than make value of i=0
    }
//Add other combinations according to tables 1 &2
return;
}
Void main()
{
float pi = 3.14159;                 // variable declaration
        DSK6713_DIP_init();             // initialize DIP switches



        for(i = 0;i< N;i++)             // write the sample values of waveform on the codec at
every sampling instant
```

```
        {                                                   //each sine_wave have different
frequency
                sine_table1[i] = 1000*sin((2.0*pi*i/8000)*697); // generation of sine-wave signal
using formula, value is taken in a loop
                sine_table2[i] = 1000*sin((2.0*pi*i/8000)*770); // generation of sine-wave signal
using formula, value is taken in a loop
                sine_table3[i] = 1000*sin((2.0*pi*i/8000)*852); // generation of sine-wave signal
using formula, value is taken in a loop
                sine_table4[i] = 1000*sin((2.0*pi*i/8000)*941); // generation of sine-wave signal
using formula, value is taken in a loop
                sine_table5[i] = 1000*sin((2.0*pi*i/8000)*1209);        // generation of sine-
wave signal using formula, value is taken in a loop
                sine_table6[i] = 1000*sin((2.0*pi*i/8000)*1336);        // generation of sine-
wave signal using formula, value is taken in a loop
                sine_table7[i] = 1000*sin((2.0*pi*i/8000)*1477);        // generation of sine-
wave signal using formula, value is taken in a loop
                sine_table8[i] = 1000*sin((2.0*pi*i/8000)*1633);        // generation of sine-
wave signal using formula, value is taken in a loop
        }

        comm_intr();    // ISR function is called, using the given command
        while(1);               //program execution halts and it starts listening for the interrupt
which occur at every sampling period Ts.
}
```