AN-NAJAH NATIONAL UNIVERSITY

FACULTY OF ENGINEERING

TELECOMMUNICATION ENGINEERING DEPARTMENT



Digital Signal Processing LAB

69443

For

Telecommunication Engineering

Reviewed and prepared by:

Dr. Falah Mohammed

Eng. Nuha Odeh

Eng. Monir Aghbar

*2013/2014*

An-Najah National University

**جامعة النجاح الوطنية**

**كلية الهندسة  
هندسة الإتصالات**

Faculty of Engineering

Telecommunication  
  
 Engineering Department

|  |
| --- |
| Telecommunication Engineering Department  Digital Signal Processing Lab (69443)  Report form |

|  |  |
| --- | --- |
| Instructors: Falah Mohammed,  Monir Aghbar | Student Names: |
| Academic Year:2013/2014 | Section: |
| Semester: Spring | Report mark: 10 |
| Credit Hours: 1 | Reports weight of final mark: 40% |
| Date: | Lab duration:3 hours |

Report notes:

1. The report structure must contain title, an abstract, objectives, results, discussion and conclusion
2. Try to support your results and analysis with graphs and equations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Assessment criteria | Points | ILO’s | ILO’s % | Assessment criteria grade | Required Time |
| Writing skills | 4 | 3 | 100% |  | One week |
| Team work | 4 | 2 | 100% |  |
| Result analysis and discussion | 2 | 1 | 100% |  |
| Student grade | | | |  |  |

Good luck

|  |  |
| --- | --- |
| Department of Telecommunication Engineering | |
| DSP Lab. (69443) | |
| Total Credits | 1 |
| major compulsory | |
| Prerequisites | P1 : Digital Signal Processing (69441) |
| 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, PCM encoding, delta modulation, digital modulation schemes (ASK, PSK, FSK), error correcting codes, read write from CODEC, fast Fourier transform, FIR Filter implementation (Low Pass, High Pass Band Stop), IIR Filter implementation, linear convolution auto correlation and power spectral density. | |
| |  |  |  |  | | --- | --- | --- | --- | | Intended Learning Outcomes (ILO's) | | Student Outcomes  (SO's) | Contribution | | 1 | An ability to use DSP processor in the design of communication systems, speech processing, image processing, digital filtering (FIR, IIR), spectral estimation. | C | 30 % | | 2 | An ability to function on multidisciplinary teams | D | 25 % | | 3 | Ability to communicate effectively through the various lab reports | G | 25 % | | 4 | Ability to design real system using DSP processor via Matlab Simulink interface Software | K | 20 % | | |
| Textbook and/ or Refrences | |
| Digital Signal Processing and Applications with the C6713 and C6416 DSK, Rulph chassaing, John Wiley, 2005 | |
| |  |  | | --- | --- | | Assessment Criteria | Percent (%) | | Laboratory Work | 65 % | | Final Exam | 35 % | | |
| Course Plan | |
| |  |  | | --- | --- | | Week | Topic | | 1 | Introduction and project creation | | 2 | Waveform generation and sampling | | 3 | ASK, BPSK and FSK modulation and demodulation | | 4 | QPSK modulation and demodulation | | 5 | FIR filtering | | 6 | IIR filtering | | 7 | Adaptive filtering and noise cancellation spectral estimation | | 8 | Audio Effects (Echo and Reverb, Harmonics, and Distortion) | | 9 | Echo cancelling and voice scrambling | | 10 | DFT and FFT implementation | | 11 | Hamming Codes | | 12 | Dual-Tone Multi frequency DTMF | | 13 | Final exam | | |

Table of Contents

[1 Digital signal processing Lab 1](#_Toc366763937)

[1.1 Objectives 1](#_Toc366763938)

[1.2 Introduction 1](#_Toc366763939)

[1.3 Advantages of Digital Signal Processing 3](#_Toc366763940)

[1.3.1 What is Real-Time Processing? 3](#_Toc366763941)

[1.4 Digital Signal Processing Systems and Applications 4](#_Toc366763942)

[1.4.1 The TMS320 Family: 4](#_Toc366763943)

[1.5 Texas Instruments’ TMS320 Family 5](#_Toc366763944)

[1.6 DSK Board 5](#_Toc366763945)

[1.6.1 TLV320AIC23 (AIC23) ONBOARD STEREO CODEC FOR INPUT AND OUTPUT 7](#_Toc366763946)

[1.7 Programming the TMS320DSK6713 9](#_Toc366763947)

[1.7.1 CODE COMPOSER STUDIO 9](#_Toc366763948)

[1.7.2 Support files 10](#_Toc366763949)

[1.8 PROGRAMMING EXAMPLES TO TEST THE DSK TOOLS 12](#_Toc366763950)

[1.8.1 How to create a project 13](#_Toc366763951)

[1.8.2 Code Generation and Options 16](#_Toc366763952)

[2 Experiment 2 wave generation and sampling 31](#_Toc366763953)

[2.1 Objectives 31](#_Toc366763954)

[2.2 Wave generation 31](#_Toc366763955)

[2.2.1 Square wave using Fourier series 31](#_Toc366763956)

[2.2.2 Square wave generation using direct synthesis 32](#_Toc366763957)

[2.2.3 Ramp signal generation 33](#_Toc366763958)

[2.3 Sampling 33](#_Toc366763959)

[2.4 Experimental procedures 33](#_Toc366763960)

[2.4.1 Square wave generation using Fourier series 33](#_Toc366763961)

[2.4.2 Square wave generation using direct thesis 35](#_Toc366763962)

[2.4.3 Ramp signal generation procedure 36](#_Toc366763963)

[2.4.4 Basic sampling 36](#_Toc366763964)

[2.4.5 Down sampling and antialias filtering 37](#_Toc366763965)

[3 Experiment 3 Amplitude Shift Keying (ASK) and Frequency shift keying (FSK) 40](#_Toc366763966)

[3.1 Objectives 40](#_Toc366763967)

[3.2 Introduction 40](#_Toc366763968)

[3.3 Theory of ASK 40](#_Toc366763969)

[3.4 Theory of FSK 41](#_Toc366763970)

[3.5 Experimental procedures 42](#_Toc366763971)

[4 Experiment 4 QPSK 46](#_Toc366763972)

[4.1 Objectives 46](#_Toc366763973)

[4.2 Theory 46](#_Toc366763974)

[4.2.1 Implementation 47](#_Toc366763975)

[4.2.2 Generation and demodulation of QPSK signal 47](#_Toc366763976)

[4.3 Experimental procedures 48](#_Toc366763977)

[5 Experiment 5 FIR filter 52](#_Toc366763978)

[5.1 Objectives 52](#_Toc366763979)

[5.2 Experimental procedures 52](#_Toc366763980)

[5.2.1 Low pass filter 52](#_Toc366763981)

[5.2.2 Low pass, high pass, band pass and pass band filter 55](#_Toc366763982)

[5.2.3 Effects on voice using three FIR low pass filters (FIR3LP) 58](#_Toc366763983)

[6 Experiment 6 IIR filter 61](#_Toc366763984)

[6.1 Objectives 61](#_Toc366763985)

[6.2 Theory 61](#_Toc366763986)

[6.2.1 Cascade form structure 62](#_Toc366763987)

[6.2.2 Parallel form structure 63](#_Toc366763988)

[6.3 Experimental procedures 63](#_Toc366763989)

[6.3.1 IIR Filter Implementation Using Second-Order Stages in Cascade (IIR) 63](#_Toc366763990)

[6.3.2 IIR Inverse Filter (IIR inverse) 68](#_Toc366763991)

[7 Experiment 7 Adaptive filters 72](#_Toc366763992)

[7.1 Theory 72](#_Toc366763993)

[7.2 Applications of adaptive filters 73](#_Toc366763994)

[7.3 Experimental procedures 75](#_Toc366763995)

[7.3.1 Adaptive Filter for Sinusoidal Noise Cancellation 75](#_Toc366763996)

[7.4 Adaptive FIR filter for noise cancellation using external inputs 78](#_Toc366763997)

[7.4.1 Adaptive FIR Filter for System ID of a Fixed FIR as an Unknown System 81](#_Toc366763998)

[8 Experiment 8 audio effects 86](#_Toc366763999)

[8.1 Objectives 86](#_Toc366764000)

[8.2 Introduction 86](#_Toc366764001)

[8.2.1 Delay 86](#_Toc366764002)

[8.2.2 Echo 86](#_Toc366764003)

[8.2.3 Reverberation 87](#_Toc366764004)

[8.3 Experiment procedure 88](#_Toc366764005)

[8.3.1 Echo generation 88](#_Toc366764006)

[8.3.2 Echo with Control for Different Effects 89](#_Toc366764007)

[9 Experiment 9 echo cancelling and voice scrambling 92](#_Toc366764008)

[9.1 Objectives 92](#_Toc366764009)

[9.2 Echo canceling using adaptive algorithm 92](#_Toc366764010)

[9.3 Voice Scrambling Using Filtering and Modulation (Scrambler) 94](#_Toc366764011)

[9.3.1 Experiment procedures 97](#_Toc366764012)

[10 Experiment 10 DFT and FFT 98](#_Toc366764013)

[10.1 Objectives 98](#_Toc366764014)

[10.2 Introduction 98](#_Toc366764015)

[10.3 FFT of a Real-Time Input Signal Using an FFT Function in C 100](#_Toc366764016)

[10.3.1 Testing procedures 102](#_Toc366764017)

[10.4 Fast Convolution 102](#_Toc366764018)

[10.4.1 Testing procedure 105](#_Toc366764019)

[11 Experiment 11 Hamming Codes 107](#_Toc366764020)

[11.1 Objectives 107](#_Toc366764021)

[11.2 Theory of Hamming Code 107](#_Toc366764022)

[11.3 Experimental procedures 110](#_Toc366764023)

[12 Dual-Tone Multi frequency DTMF 113](#_Toc366764024)

[12.1 Objectives 113](#_Toc366764025)

[12.2 Theory of ASK 113](#_Toc366764026)

[12.3 Experimental procedures 113](#_Toc366764027)

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.https://encrypted-tbn3.gstatic.com/images?q=tbn:ANd9GcTCALPxB6BgbOulAeoKU4mtGI4dC2TEYWGAHoPR-B57Wh-hONPU2g |

1. *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.

|  |
| --- |
| NEVER use 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.

1. Return parts and all equipment have to correct locations when you are finished with them.
2. Give suspected defective parts to the Lab technician for testing or disposal.
3. Report all equipment problems to Lab Instructor or Lab technician.
4. Most experiments have several parts; students must alternate in doing these parts as they are expected to work in group.
5. 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.
6. 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 notebook during the experiment. Do not use scratch paper for recording data. Do not trust your memory to fill in the details at a 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 them and make 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 (2-3 lines), 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, a report 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 data sheets can 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 may be 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.

# Digital signal processing Lab

## Objectives

The objectives of this experiment are

1. Brief students with the operation of the DSK6713 kit
2. 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 1 Block 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 signal in 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

Digital Signal Processor (DSP) is a microcontroller designed specifically for signal processing applications. This is achieved as specified in . 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 2 Block 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. shows the basic components of a DSP system.

|  |
| --- |
|  |

Figure 3 Main 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.

### 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 time lines 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.

### The TMS320 Family:

Texas instrument introduced the first DSP processor, the TMS32010 is 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 C67x generations.

|  |
| --- |
|  |

## 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

1. Audio applications

* Audio watermarking
* Coding and decoding
* Effects generator
* Surround sound processing
* Three dimensional audio

1. 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 codec TLV320AIC23 (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 to 96 kHz can be set readily. A daughter card expansion is also provided on the DSK board. Two 80-pin connectors provide for external peripheral and external memory interfaces.

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

|  |
| --- |
| (a) |
| (b) |

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

### 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 is 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 . It performs all the functions required for ADC and DAC, low pass filtering, oversampling, and so on. 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 5 TLV320AIC23 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 , 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 and , an exact audio sample rate of 48 kHz (12MHz/250) and a CD rate of 44.1kHz (12MHz/272) can be obtained. The sampling rate is set by the codec’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

### 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

### 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 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) and LED #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 table sine\_table[0] = 0. The loop index is incremented until the end of the table is reached, after which it is reinitialized to zero. Every sample period , the value of dip switch #0 is tested, and a subsequent data value in sine\_table (scaled by gain = 10) is sent for output. Within one period, eight data values (0.125 ms apart) are output to generate a sinusoidal signal. The period of the output signal is , corresponding to a frequency of .

|  |
| --- |
|  |

Figure 6 Sine generation program using eight points with dip switch control (sine8\_LED.c).

### 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 Project → New. Type sine8\_LED for the project name, as shown in Figure 1.3. This project file is saved in the folder sine8\_LED (choose your directory).The .pjt file stores project information on build options,
2. Copy the files under support folder (you can find it in desktop) into your project folder.
3. To add files to the project. Select Project→Add Files to Project. Look in the folder support, Files of type C Source Files. Double-click on the C source file C6713dskinit.c to add it to the project. Click on the “+” symbol to the left of the Project Files window within CCS to expand and verify that this C source file has been added to the project, see b.
4. Repeat step 3, use the pull-down menu for files of type, and select ASM Source Files. Double-click on the assembly source vector file vectors\_poll.asm to add it to the project. Repeat again and select files of type: Linker Command File, and add C6713dsk.cmd to the project.
5. To add the library support files to the project. Repeat the previous step, but select files of type: Object and Library Files. Look in c:\ccstudiov3.1\c6000\cgtools\lib and select the run-time support library file *rts6700.lib* (which supports the C67x architecture) to add to the project. Continue this process to add the BSL file *dsk6713bsl.lib* located in c*:\ccstudiov3.1\c6000\dsk6713\lib*, and the chip support library (CSL) file *csl6713.lib* located in c*:\ ccstudiv3.1\c6000\csl\lib.*
6. Verify from the Files window that the project (.pjt) file, the linker command (.cmd) file, the three library (.lib) files, the two C source (.c) files, and the assembly (.asm) file have been added to the project as shown in b. The GEL file *dsk6713.gel* is added automatically when you create the project. It initializes the C6713 DSK invoking the BSL to use the phase-locked loop (PLL) to set the central processing unit (CPU) clock to 225 MHz (otherwise, the C6713 runs at 50 MHz by default).
7. Note that there are no “include” files yet. Select Project→ Scan All File Dependencies. This adds/includes the header files *c6713dskinit.h*, along with several board and chip support header files included with CCS.
8. Add your source file ("sin8\_led.c) to the project.

The Files window in CCS should look as in b. Any of the files (except the library files) from CCS’s Files window can be displayed by clicking on it. You should not add header or include files to the project. They are added to the project automatically when you select: Scan All File Dependencies. (They are also added when you build the project.)

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

|  |
| --- |
| (a) |
| (b) |

Figure 7 CCS Project windows for sine8\_LED: (a) project creation; (b) project view files.

### Code Generation and Options

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

*Compiler Option*

Select Project → Build Options. shows the CCS window Build Options for the compiler. Select the following for the compiler option with Basic (for Category): (1) c671x{-mv6710} (for Target Version), (2) Full Symbolic Debug (for Generate Debug Info), (3) Speed most critical (for Opt Speed vs. Size), (4) None (for Opt Level and Program Level Opt). Select the Preprocessor Category and type for Define Symbols {d}: CHIP\_6713, and (5) form Advanced Category change Memory Model to "Far(--mem\_model:data=far),and from the Feedback Category, select for Interlisting: OPT/C and ASM{-s}. The resulting compiler option is

-g -s

The -g option is used to enable symbolic debugging information, useful during the debugging process, and is used in conjunction with the option -s to interlist the C source file with the assembly source file *sine8\_LED.asm* generated (an additional option, -k, can be used to retain the assembly source file). The -g option disables many code optimizations to facilitate the debugging process. Press OK.

|  |
| --- |
|  |

Figure 8 CCS Build options: compiler

Selecting C621x or C64xx for Target Version invokes a fixed-point implementation. The C6713-based DSK can use either fixed- or floating-point processing. Most examples implemented in this book can run using fixed-point processing. SelectingC671x as Target Version invokes a floating-point implementation. If No Debug is selected (for Generate Debug Info) and -o3: File is selected (for Opt Level), the Compiler option is automatically changed to

-s -o3

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

Linker Option

Click on Linker (from CCS Build Options).The output filename sine8\_LED.out defaults to the name of the .pjt filename, and Run-time Auto initialization defaults for Auto init Model. The linker option should be displayed as in . 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.

|  |
| --- |
|  |

Figure 9 Linker options

*Building and Running the Project*

The project sine8\_LED can now be built and run.

1. Build this project as sine8\_LED. Select Project → Rebuild All or press the toolbar with the three down arrows. This compiles and assembles all the C files using cl6x and assembles the assembly file vectors\_poll.asm usingasm6x. The resulting object files are then linked with the library files usinglnk6x. This creates an executable file sine8\_LED.out that can be loaded into the C6713 processor and run. Note that the commands for compiling, assembling, and linking are performed with the Build option. A log filecc\_build\_Debug.log is created that shows the files that are compiled and assembled, along with the compiler options selected. It also lists the support functions that are used. shows several windows within CCS for the project sine8\_LED.The building process causes all the dependent files to be included (in case one forgets to scan for all the file dependencies).
2. Select File → Load Program in order to load sine\_LED.out by clicking on it (CCS includes an option to load the program automatically after a build).It should be in the folder sine8\_LED\Debug. Select Debug → Run or use the toolbar with the “running man.” Connect the LINE OUT terminal to the oscilloscope and observe the resulting sine wave when the dip switch #0 is pressed on the DSK board.

The sampling rate of the codec is set at 8 kHz. The generated frequency . Note that the amplitude of the generated signal is approximately 0.8 V p-p (peak to peak).

|  |
| --- |
|  |

Figure 10 CCS 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 → Quick Watch window, which should be displayed on the lower section of CCS. Type gain, then click on “Add to Watch.” The gain value of 10 set in the program in should appear in the Watch window.
2. Change gain from 10 to 30 in the Watch window. Press Enter. Verify that the volume of the generated tone has increased (with the processor still running and dip switch #0 is pressed). The amplitude of the sine wave has increased from approximately 0.8 V p-p to approximately 2.5V p-p.
3. Change gain to 33 (as in step 2). Verify that a higher-pitched tone exists, which implies that the frequency of the sine wave has changed just by changing its amplitude. This is not so. You have exceeded the range of the 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 to , 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. Select File →new and create new file. Write the code shown in in the new file and save this file in your project directory (sine8\_LED) as gain.gel
2. Select File → Load GEL and open the file gain.gel. By creating the slider function gain shown in , 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 GEL → Sine Gain → Gain. This should bring out the Slider window shown in , 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).

|  |
| --- |
|  |

Figure 11 GEL file to slide through different gain values in the sine generation program (gain.gel).

|  |
| --- |
|  |

Figure 12 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 in the C source program to DSK6713\_AIC23\_FREQ\_16KHZ. Rebuild (use incremental build) the project, load and run the new executable file, and verify that the frequency of the generated sinusoid is 2 kHz. The sampling frequencies supported by the AIC23 codec are 8, 16, 24, 32, 44.1, 48, and 96 kHz.
2. Change the number of points in the lookup table to four points in lieu of eight points for example, {0, 1000, 0, -1000}. The size of the array sine\_table and the loop index also need to be changed. Verify that the generated frequency is .

Note that the sinusoid is no longer generated if the dip switch #0 is not pressed. If a different dip switch such as switch #3 is desired (in lieu of switch #0), the BSL functions DSK6713\_DIP\_get(3), DSK6713\_LED\_on(3), and DSK6713\_LED\_off(3) can be substituted in the C source program.

Two sliders can readily be used, one to change the gain and the other to change the frequency. A different signal frequency can be generated by changing the loop index within the C program (e.g., stepping through every two points in the table).When you exit CCS after you build a project, all changes made to the project can be saved. You can later return to the project with the status as you left it before. For example, when returning to the project after launching CCS, select Project →Open to open an existing project such as sine8\_LED.pjt (with all the necessary files for the project already added).

Example 2: Loop Program Using Interrupt (loop\_intr)

This example illustrates input and output with the AIC23 codec. Figure 2.4 shows the C source program loop\_intr.c, which implements the loop program. It is interrupt-driven using INT11.

This program example is very important since it can be used as a base program to build on. For example, to implement a digital filter, one would need to insert the appropriate algorithm between the input and output functions. The two functions input\_sample () and output\_sample(), as well as the function comm\_intr(), are included in the communication support file C6713dskinit.c.This is done so that the C source program is kept as small as possible. The file C6713dskinit.c can be used as a “black box program” since it is used in many examples throughout this lab.

After the initialization and selection/enabling of an interrupt, execution waits within the infinite while loop until an interrupt occurs. Upon interrupt, execution proceeds to the ISR c\_int11, as specified in the vector file vectors\_intr.asm. An interrupt occurs every sample period , at which time an input sample value is read from the codec’s ADC and then sent as output to the codec’s DAC.

Execution returns from interrupt to the while(1) statement waiting for a subsequent interrupt. [Note that in lieu of waiting within the while(1) infinite loop, one could be processing code.] Upon interrupt, execution proceeds to Interrupt Service Routine (ISR), “services” the necessary task dictated by ISR, then returns to the calling function waiting for the occurrence of a subsequent interrupt.

|  |
| --- |
|  |

Figure 13 Loop program using interrupt .

Comments about this example

1. The function output\_sample() be default send the a 16-bit sample data to the 16-bit left channel (LINE OUT). If data needs to be sent to the right channel you need to use the function output\_right\_sample().
2. Within the function comm\_intr(), the following tasks are performed.
3. Initialize the DSK.
4. Configure/select INT11.
5. Enable the specific interrupt.
6. Enable the global enable interrupt (GIE) bit and the nonmaskable interrupt.
7. Initiate communication.

Execution steps and results

1. Create and build this project as loop\_intr. The main C source file is in the folder loop\_intr. Note that the vector file for the interrupt-driven and linker command file located in the folder support, and the runtime support, board support, and chip support library files that can be added with the building option for the linker.
2. Input a sinusoidal waveform to the LINE IN connector on the DSK, with an amplitude of approximately 2 V p-p and a frequency between approximately 1 and 3kHz. Connect the output of the DSK, LINE OUT to a speaker or to an oscilloscope and verify a tone of the same input frequency, but attenuated to approximately 0.8 V p-p. Using an oscilloscope, the output is a delayed version of the input signal
3. Increase the amplitude of the input sinusoidal waveform beyond 6V p-p and observe that the output signal becomes distorted.
4. *Input with Gain*

To adjust the gain of the left line-input channel, the corresponding header support file c6713dskinit.h of the communication/init “black box” file needs to be modified slightly. First, copy this header file AND c6713dskinit.c from the support folder into the folder loop\_intr so that you do not modify the original header file. Remove the init file from the project and replace it with the one in the folder loop\_intr.This will keep the original init support file unchanged in the folder support. Modify the setup register 0, which controls the left input volume, from 0x0017 to 0x001c in order to increase the left line-input volume.

Rebuild the project, making sure that you are adding c6713dskinit.c from the folder loop\_intr (and not from the folder support). In this fashion, the corresponding header file c6713dskinit.h that will be included will come from that same folder.

Load/run the executable file loop\_intr.out, and verify that the output amplitude is not attenuated and is the same as the input amplitude of 2V p-p. Values for the set-up register 0 from 0x0018 to 0x001c will cause the output amplitude to increase from 0.8 to 2V p-p.

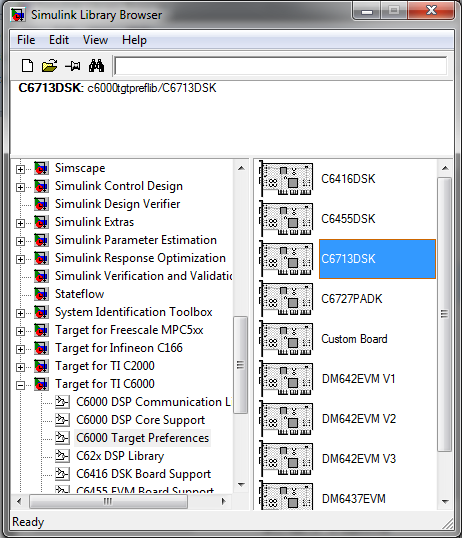
The left input channel was selected since input\_sample and output\_sample default to the left channel. Otherwise, if the right line-input volume is to be increased by modifying the set-up register 1, an adapter/connector with two inputs and one single-ended output connections would be needed. See Example 2.3 (loop\_stereo/ sine\_stereo).

1. *Input from a Microphone*

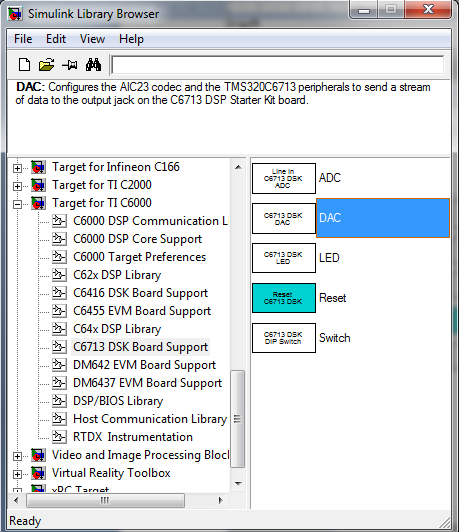
To select an input from a microphone in lieu of line input, modify the header file set-up register 4 from 0x0011 to 0x0015 (third LSB as a 1) so that the ADC gets its input from MIC IN. The microphone input and line input are multiplexed, and only one is active at a time. Rebuild the project to verify your output, with the input to the MIC IN connector.

Sine wave Generation ( Simulink Matlab):

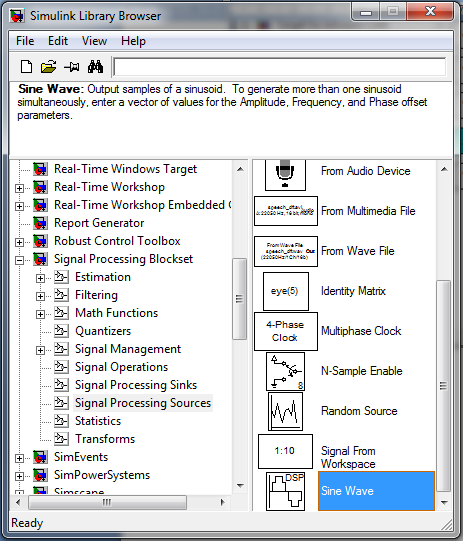
1. Open MATLAB R007b , Then choose .
2. Choose , choose Target for TI C6000 library then C6000 Target Preferences then drag C6713DSK to your file.



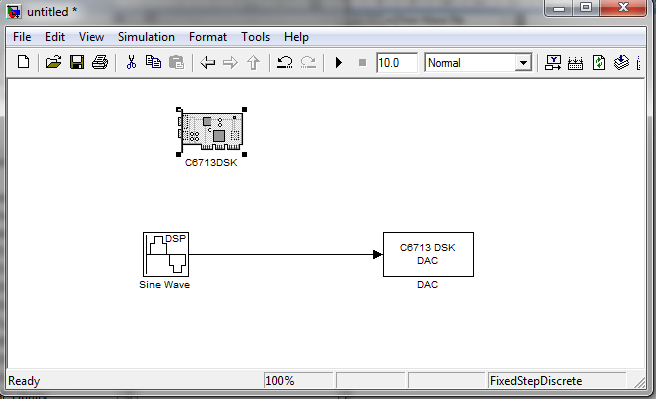
1. Choose Target for TI C6000 library again ,then choose C6713 DSK Board Support, then drag DAC (digital to analog convertor) to your file



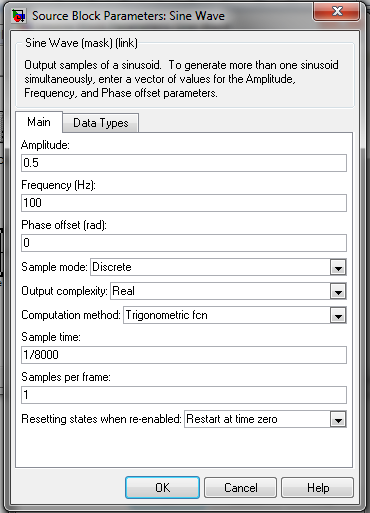
1. Choose Signal Processing Blockset Library, then choose Signal Processing Sources and drag Sine Wave to your file.



1. Connect between Sine Wave and DAC



1. Double Click on Sine Wave Block and Choose the required parameters:

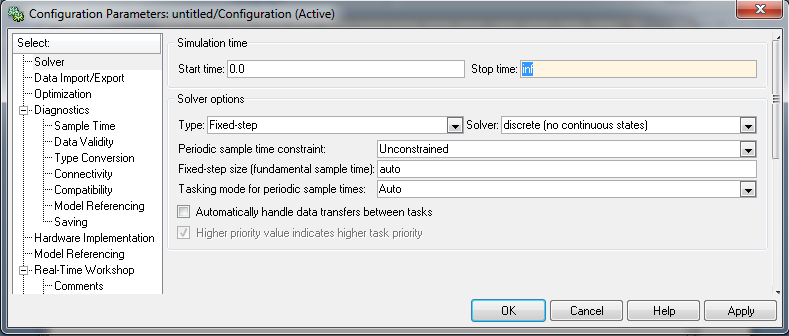


1. Double Click on Sine Wave Block and choose the required parameters:



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

1. Save your file in any directory , and select this directory before execution ( change Current Directory).
2. From Simulation→Simulation Parameters→Solver , change stop time to inf.



1. Choose incremental build icon (Notice the execution steps that done automatically). Verify the results on CRO.
2. Try to modify the frequency and amplitude.

# Experiment 2 wave generation and sampling

## Objectives

The objectives of this experiment are

1. Show the students how to use DSP processors to generate real time signals such as square wave, ramp signal
2. 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

### 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

In equation (1) represent the square wave, is the number of harmonics and 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 20 Square wave generated by the addition of two harmonics.

If the number of harmonics is increased to , then the signal gets closer to the perfect square as shown in

|  |
| --- |
|  |

Figure 21 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 22 Square wave generated by the addition of harmonics where the number of harmonics .

### 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 ( and the second half of the table is filled by minimum amplitude (), then a square wave will be generated.

### 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

2

Where 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.

### Square wave generation using Fourier series

1. Open the project that you have made in the previous lab
2. Go to project→Add files to project and select vectors\_intr.asm and vectors.asm
3. Select Project→ Scan All File Dependencies
4. Open new source file and save it as sintosquare.c
5. 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 N 9 // no. of Harmonics  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.  Uint32 f0 =200; //define the fundamental frequency of your square wave  short square\_table[N]={0}; //a look table used to store sample values for the square wave  int i,j, gain=1000, N=9; // variable declaration, N is the number of harmonics  float pi = 3.14159; // variable declaration  interrupt void c\_int11() // ISR call, At each Interrupt, program execution goes to the interrupt service routine  {  for (j=0;j<n;n++) //This loop counts advances the sampling time (nTs) which is used in computing the samples  {  for (i=1;i<=N;i+=2) // This for loop computes sample from each harmonic and adds them up all the samples in one memory location  {  square\_table[j]=square\_table[j]+(21\*gain/i)\*sin(2\*pi\*f0\*i\*j\*Ts);  } //end the inner loop  } //ends the outer loop  return; // program execution goes back to while(1) and then again starts listening for next interrupt and this process goes on  }  void main()  {  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.  } |

1. Exercise add slider to your program in order to change the number of harmincs from 1 to 11 and see the effect on the ripple and the.
2. What happens to the ripple if the number of harmonics is increased? What happens to the ripple if the number of harmonics is reduced?

### 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 for assistance

|  |
| --- |
|  |

Figure 23 Square-wave generation program (squarewave.c).

1. What is the frequency of the generated square wave?
2. explain how the frequency of the generated wave can be modified
3. change the frequency of the square wave to 2 kHz and explain what if you still  
    square wave shape or not

### Ramp signal generation procedure

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

|  |
| --- |
|  |

Figure 24 Ramp generation program without a lookup table (ramp.c).

Observer the generated signal you see on the oscilloscope and explain how the frequency and slope of the ramp signal can be changed

### 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 to help you performing this task

|  |
| --- |
|  |

Figure 25 Loop program using interrupt (loop\_intr.c).

1. After compiling and running your program, connect a signal from the function generator to the LINE IN terminal
2. Set the function generator to sine wave, the amplitude of the signal to 1 VPP, and the frequency to 1 kHz and plot the signal as it appears on the screen of the oscilloscope
3. Repeat step 3 for a square wave
4. 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
5. Determine the frequency of the reconstructed signal if the input signal frequency is and the sampling frequency is 8 kHz

### Down sampling and antialias filtering

In this part of the experiment you will take samples from a signal applied to LINE IN taken from the function generator; at a frequency of 8 kHz, then investigate the effect of down sampling.

Recall that from the DSP course that down sampling was performed on a given signal in order to reduce the processing time taken by the DSP processor. Down sampling is illustrated by

|  |
| --- |
|  |

Figure 26 Block diagram to illustrate down sampling

If a signal down sampled by factor , then its spectrum is increased t by a factor of . This increase in the spectrum may lead to aliasing as illustrated by the

|  |
| --- |
|  |

Figure 27 Pictorial illustration of down frequency sampling.

However, the effect of aliasing resulting from the down sampling process can be reduced by using a low pass filter before the down sampler. This filter is known as antialias filter. The cut off frequency of the antialias filter is selected such that .

In this part of the experiment students will examine the effect of down sampling on a given signal, then down sample the captured samples and investigate the effect of aliasing and the use of antialis filter

1. Open new MATLAB Simulink design
2. In your design bring a wave form generator
3. Set the waveform square with a frequency of 1000 HZ
4. Add to your design ADC for the TI C6000
5. Set the sampling frequency of the ADC to 8 kHz
6. Connect the ADC to DAC component for the TI C6000 and observe the signal measured at the DAC output using oscilloscope
7. Insert in your design a down sampling, set the down sampling rate to 2 and observe explain what happens to the signal
8. Insert a low pass anti alias filter with cutoff frequency of 2 kHz and observe what happens to the signal measured at the oscilloscope
9. Determine the frequency of the reconstructed signal if the input signal frequency is and the sampling frequency is 8 kHz

# 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

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. illustrates a binary bit stream and the ASK modulated signal. Neither signal is band limited.

|  |
| --- |
|  |

Figure 28an 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 .

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

If for example a given bit stream is to be transmitted at a bit rate of 1200 bits/s, then the bit duration is . If the carrier is to have 4 cycles per bit then, then the carrier frequency must be selected such that .

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 . Therefore we select a sampling frequency of 48 kHz to satisfy the Nyquist criteria. The number of samples is determined from

## 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 29 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

Where is the frequency used to represent logic 1 while is the frequency used to represent logic 0, and is called the deviation ratio.

If , 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 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 is made larger, the demodulation of the FSK signal became simpler. However larger values of 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

## Experimental procedures

In order to generate an ASK signal follow these steps

1. Set the sampling frequency to and the carrier frequency
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;

}

}

}

1. What is the number of cycles per bit?
2. What is the bit rate of the above transmitted stream?
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 by using simulink in the receiving DSK6713 kit. And verify that this circuit demodulates the ASK signal

|  |
| --- |
|  |

Figure 30 Ask demodulator block in Simulink

Note that sampling frequency must be 32kHz (as in modulator), samples/frame=1, cut off frequency of the low pass filter =2 kHz.

1. Is there any alternative way to demodulate the ASK signal? Explain in your report
2. Modify the above code in order to generate an FSK signal
3. How can the FSK signal be demodulated?
4. 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?

# 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 Error! Reference source not found.. With 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 31 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.

### 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

This yields the four phases of the carrier , , and . The previous equation can be further simplified by expanding the cosine term using trigonometric identities as shown below

This results in a two-dimensional signal space diagram with unit basis functions. The first basis function; ; is used as the in-phase component of the signal and the second basis function; ; 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

### Generation and demodulation of QPSK signal

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

|  |
| --- |
|  |

Figure 32 Block diagram for generating QPSK signal.

|  |
| --- |
|  |

Figure 33 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

1. Set the sampling frequency to and the carrier frequency
2. Set the number of samples to 16 samples
3. 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

1. What is the bit rate of the above transmitted stream?
2. What is the bandwidth required for the transmission of this QPSK signal
3. 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 by using simulink in the receiving DSK6713 kit. Verify that this circuit demodulates the QPSK signal

|  |
| --- |
|  |

Figure 34 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:

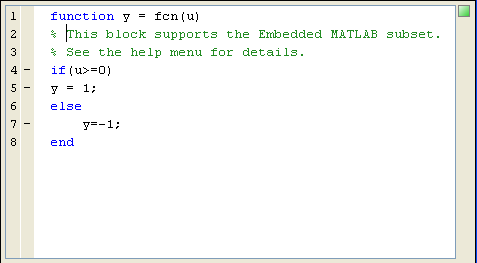


Figure 5 Implementing of a comparator using Embedded MATLAB Function

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

# Experiment 5 FIR filter

## Objectives

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

## 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.

### Low pass filter

In this experiment you will design a low pass filter whose cutoff frequency is . 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 will appear on the screen.

|  |
| --- |
|  |

Figure 36 Filter design tool in Matlab.

1. Set the response type to Low pass filter, the design method to FIR window, filter order 100, window Hamming, sampling frequency to and the carrier frequency
2. Press on the design button to design the filter.
3. 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 will appear. Select export to coefficient File (ASCII) and press on the export button. Name your file as “lpf.cof” and save it under your project directory. The file contains the filter coefficient that is needed by project

|  |
| --- |
|  |

Figure 37 Export dialog in MATLAB.

1. Modify the filter coefficient file by adding the following two statements

|  |
| --- |
| #define N 101 //number of coefficients  float h[N]={ add the coeffients of the filter here} |

1. 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  } |

1. After you build the project, load it on the DSK6713 kit, then run the program.
2. To test the designed LPF filter, connect a sine wave from the function generator to the LINE IN terminal of the DSK6713 kit.
3. Set the amplitude of the sign wave to 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

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

Table 1

Plot the amplitude versus frequency and comment on the results.

1. 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 38 Random noise generator in MATLAB

1. Set the sample time for the noise to and it type is Gaussian.
2. Connect the output of the DSK kit to your pico-scope and adjust the scope for spectrum operation
3. Sketch the frequency response of the filter as it appears on the screen of the spectrum analyzer

### 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 , the cut off frequency of the High pass filter as , the center frequency of the bandpass filter as , and the band stop of the band stop filter as .

|  |
| --- |
| \* Description :This Program tells about the Finite Impulse Response.  FIR filter:  lowpass 1500 Hz,  High pass,2200hz,  bandpass 1750 hz,  bandstop 790 hz.  we can select desires filter by varying slider in gel file  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #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 >> 15); //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\*/  } |

### 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, 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- 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 >> 15); //output filter  return; //return from interrupt  }  void main()  {  short i;  for (i=0; i<N; i++)  {  dly[i] = 0; //init buffer  h[0][i] = hlp600[i]; //start 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  } |

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. Use a .wav file from your PC 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\*/  } |

# Experiment 6 IIR filter

## Objectives

1. The aim of this experiment is to design and implement different IIR filters type
2. 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

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 and . 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 .

|  |  |
| --- | --- |
| (a) | (b) |

Figure 39 (a) direct form I IIR structure. (b) Direct form II IIR structure.

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.

### 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

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

|  |
| --- |
|  |

Figure 40 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 41 Fourth-order IIR filter with two direct form II sections in cascade.

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

### Parallel form structure

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

This can be obtained by using a partial fraction expansion. The block diagram of the parallel form structure is shown in . Each of the transfer functions , , …., etc.

|  |
| --- |
|  |

Figure 42 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, can be expressed as

## 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.

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

In this experiment you will design a low pass filter whose cutoff frequency is . 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 will appear on the screen.

|  |
| --- |
|  |

Figure 43 SPTool in Matlab.

1. From the startup window; second column in ; select new. A design window such as the one shown in 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 and the cut off frequency

|  |
| --- |
|  |

Figure 44 Filter design tool in MATLAB.

1. Access the startup window in SPTool again. Select →Edit→Name. Change the name (enter new variable name) to lp2400
2. Select File →Export → Export to workspace the lp2400 design.
3. 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

|  |
| --- |
|  |

These 30 coefficients represent the numerator coefficients , , and and the denominator coefficients , , and .They represent six coefficients per stage, with normalized to 1 and scaled by .

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 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  }; |

1. The IIR filter can be implemented by using the following two equations associated with each stage :
2. The loop section of code within the program is processed five times (the number of stages) for each value of , or sample period. For the first stage, is the newly acquired input sample. However, for the other stages, the input is the output of the preceding stage.
3. The coefficients and correspond to and , respectively; where i represents each stage. The delays and correspond to and , respectively.
4. 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  } |

1. After you build the project, load it on the DSK6713 kit, then run the program.
2. To test the designed LPF filter, connect a sine wave from the function generator to the LINE IN terminal of the DSK6713 kit.
3. Set the amplitude of the sign wave to 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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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

1. 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

### 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

The output sequence of the IIR filter is

where represents the input sequence. The input sequence can then be recovered using as an estimate of , or

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 [9].

|  |
| --- |
| #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  } |

# 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 . This is a basic adaptive filter structure in which the adaptive filter’s output is compared with a desired signal to yield an error signal , 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 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 , and what to make the error .

|  |
| --- |
|  |

Figure 45 Basic 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 is

Where represent weights or coefficients for a specific time . 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 and the adaptive filter’s output .

The weights or coefficients are adjusted such that a mean squared error function is minimized. This mean squared error function is , where represents the expected value. Since there are weights or coefficients, a gradient of the mean squared error function is required. An estimate can be found instead using the gradient of yielding

* 1. Applications of adaptive filters

Adaptive filters have been used for different applications such as:

1. For noise cancellation as illustrated in . The desired signal is corrupted by uncorrelated additive noise . The input to the adaptive filter is a noise that is correlated with the noise . The noise could come from the same source as but modified by the environment. The adaptive filter’s output is adapted to the noise . When this happens, the error signal approaches the desired signal . The overall output is this error signal and not the adaptive filter’s output . If is uncorrelated with , the strategy is to minimize , where 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,, will be unaffected and onlyits noise component will be minimized.

|  |
| --- |
|  |

Figure 46 Adaptive filter structure for noise cancellation.

1. For system identification. 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 is the difference between the response of the unknown system and the response of the adaptive filter . This error signal is fed back to the adaptive filter and is used to update the adaptive filter’s coefficients until the overall output .When this happens, the adaptation process is finished, and 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 47 Adaptive filter structure for system identification.

1. Inverse system modeling (e.g. channel equalization in modems) which is shown in

|  |
| --- |
| C:\Users\laura\AppData\Local\Temp\Rar$DI19.080\img004.jpg |

Figure 48 adaptive filter for system modeling.

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

|  |
| --- |
|  |

Figure 49: Adaptive predictor structure.

## Experimental procedures

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

### 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 , 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 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, .

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); | |

1. 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  interrupt void c\_int11() //ISR  {  short i;  static short buffercount=0; //init count of # out samples  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  } |

1. Write a slider code to change the parameter out\_type from 1 to 2
2. Build and execute the project
3. Plot the signal that as you see on the screen of the oscilloscope if the out\_type variable is set to 1.
4. What signal you are measuring on the oscilloscope screen?
5. Plot the signal that as you see on the screen of the oscilloscope if the out\_type variable is set to 2.
6. What signal you are measuring on the oscilloscope screen?
7. Repeat the experiment by using Gaussian error signal rather than using a sinusoidal signal
   1. Adaptive FIR filter for noise cancellation using external inputs

This example extends the previous one to cancel an undesirable sinusoidal noise using external inputs. The source program shown below allows two external inputs: a desired signal and a sinusoidal interference. The program uses the union structure introduced in Chapter 2 with the project example loop\_stereo. A 32-bit signal is captured using this structure that allows an external 16-bit input signal through each channel. The 16-bit desired signal is input through the left channel and the undesirable 16-bit signal through the right channel. An adapter with two connectors at one end for each input signal and one connector at the other end, which connects to the DSK, was introduced in Chapter 2 with the loop\_stereo project and is required to implement this example. The basic adaptive structure in is applied here along with the LMS algorithm.

|  |
| --- |
| #include "DSK6713\_AIC23.h" //codec-DSK support file  uint32 fs=DSK6713\_AIC23\_FREQ\_48KHZ; //set sampling rate  #define beta 1E-13 //rate of convergence  #define N 30 //# of weights (coefficients)  #define LEFT 0 //left channel  #define RIGHT 1 //right channel  float w[N]; //weights for adapt filter  float delay[N];//input buffer to adapt filter  short output;//overall output  short out\_type = 1;//output type for slider  volatile union{unsigned intuint; short channel[2];}AIC23\_data;  interrupt void c\_int11()//ISR  {  short i;  float yn=0, E=0, dplusn=0, desired=0, noise=0;  AIC23\_data.uint = input\_sample();//input 32-bit from both channels  dplusn =(AIC23\_data.channel[LEFT]);//input left channel  noise = (AIC23\_data.channel[RIGHT]);//input right channel  delay[0] = noise; //noise as input to adapt FIR  for (i = 0; i < N; i++) //to calculate out of adapt FIR  yn += (w[i] \* delay[i]); //output of adaptive filter  E = (dplusn) - 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  }  if(out\_type == 1) //if slider in position 1  output=((short)E); //error signal as overall output  else if(out\_type==2) //if slider in position 2  output=((short)dplusn); //output (desired+noise)  output\_sample(output); //overall output result  return;  }  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  } |
|  |

1. Using your PC soundcard and the MATLAB. Generate 1.5 kHz sinusoidal signal from the function generator to represent the desired signal and another sinusoidal signal with 2 kHz; which represents the additive noise signal; and another cosine 2 kHz signal to represent reference noise as shown in :

|  |
| --- |
| Figure 50 |

1. Run the program. Verify that the 2-kHz noise signal is being canceled gradually. You can adjust the rate of convergence by changing beta by a factor of 10 in the program.
2. Access/load the slider program adaptnoise\_2IN.gel and change the slider position from 1 to 2. Verify the output as the two original sinusoidal signals at 1.5 kHz and at 2 kHz.
3. Desired: wideband random noise; undesired: 2 kHz. Input random noise (from a noise generator, MATLAB.) as the desired wideband signal into the left input channel and the undesired 2-kHz sinusoidal noise signal into the right input channel. Restart/run the program. Verify that the 2-kHz sinusoidal noise signal is being canceled gradually, with the wideband random noise signal remaining. With the slider in position 2, observe that both the undesired and desired input signals are.

### 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 .

To test the adaptive scheme, the unknown system to be identified is chosen as an FIR bandpass filter with 55 coefficients centered at . The coefficients of this fixed FIR filter are in the file bp55.cof, introduced in Chapter 4. A 60-coefficient adaptive FIR filter models the fixed unknown FIR band pass filter.

A pseudorandom noise sequence is generated within the program (see examples 2.16 and 4.4 in Rulph chassing) 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 (see method B in Example 4.8 in Rulph chassing).

1. Design an FIR banpass filter with a center frequency of , bandwidth of 400 Hz. Use 55 coefficients in designing the filter. Export the filter coefficients to a file called bp55.cof
2. Use the following header file required for the random noise generation in your project directory

|  |
| --- |
| //Noise\_gen.h 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; |

1. 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).
2. 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.h" //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  int fb; //feedback variable  shift\_reg sreg; //shift register  int prand(void) //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  }  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] = prand(); //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  sreg.regval=0xFFFF; //initial seed value  fb = 1; //initial feedback value  comm\_intr(); //init DSK, codec, McBSP  while (1); //infinite loop  } |

# Experiment 8 audio 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.

### 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 where 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 , we demonstrate the delay operation using 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 will reach to the end of the array cycles later.

|  |
| --- |
|  |

Figure 51 block diagram illustrating delay.

### 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 52 echo block diagram

The difference equation which describes the echo system is ; where is the delay mix parameter, 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.

### 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 below.

|  |
| --- |
|  |

Figure 53 The block diagram of a multi-echo reverberation system

The difference equation for this system is . The value of 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.

### 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  } |

1. Vary the buffer size from 1000 to 8000 and listen to the played recorded wave. What effect do you observe on heard signal?
2. 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?
3. 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 by 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.  } |

### 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 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\*/  } |

# Experiment 9 echo cancelling and voice scrambling

## Objectives

The main objectives of this experiment are

1. Use the adaptive filter for echo cancelation.
2. Illustrate the concept of voice scrambling.
   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 54 echo canceling using adaptive filter.

The adaptive filter takes the un-echoed signal from the far end microphone as the desired signal . The error signal is defined as the difference between the adaptive filter output and the echoed signal as given by

The adaptive filter tries to generate a delayed version of the original un-echoed signal 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 30 // no. of samples  #define beta 0.3 //# of weights (coefficients)  short w[N]; //weights for adapt filter  short delay[N],gain=1; //input buffer to adapt filter  short E, yn, echo1, lcecho\_output, rcecho\_output; // variable declaration  short lecho\_output,recho\_output;  int a;  short input, output, output2;  short echo1,lecho\_output;  short bufferlength = 8000; //buffer size for delay  short buffer[8000]; //create buffer  short i = 0, out\_type=1;  short amplitude = 5; //to vary amplitude of echo  short Echo\_cancellation(short lin\_input, short echo\_output);  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()  {  comm\_intr(); //init DSK, codec, McBSP  while(1); //infinite loop  }  short Echo\_cancellation(short input,short echo\_output) //the control passes to the function Echo\_cancellation( ), called by above.& its call by value function  {  // store the input signal into a variable  echo1=lecho\_output; // store the echo signal into echo1 variable  delay[0]=echo1; //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 = (input - echo1)- yn\*gain ;  }    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  }  lcecho\_output = E;  return lcecho\_output; // program execution goes back to the function called and then again starts listening for next call and this process goes on    } |

* 1. 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.

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 55 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\_8KHZ; //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  filtmodfilt(input); //process sample twice(upsample)  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 ) 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, , 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 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

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

|  |
| --- |
| i=0:199;  desired= round(10000\*sin(2\*pi\*(i)\*4000/16000)); %sin(1500)addnoise= round(100\*sin(2\*pi\*(i)\*312/8000)); %sin(312)  fid=fopen('sin200.h','w'); %desired sin(1500)  fprintf(fid,'short sin200[]={');  fprintf(fid,'%d, ' ,desired(1:199));  fprintf(fid,'%d' ,desired(200));  fprintf(fid,'};\n');      fclose(fid); |

Add these commands sin200.h :

#ifndef NSINE

#define NSINE 200

.

.

.

#endif

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.

An optional up-sampling (by a factor of 2) scheme is used to obtain a 16-kHzsampling rate. This scheme is achieved by “processing” the input data twice while retaining only the second result.

This allows for a wider input signal bandwidth to be scrambled, resulting in a better performance.

* + 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 .
4. Note that the upper sideband signal of 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 , 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.

# Experiment 10 DFT and FFT

## Objectives

The main objective of this experiment is to familiarize students with the basic implementation of the FFT algorithm.

## Introduction

In this part of the experiment the implementation of the DFT for a sine wave using look up table will be illustrated. The discrete Fourier transform for a given input sequence is defined by

Where is the twiddle constant. The series of can be decomposed into a sum of real components and a sum of imaginary components as illustrated below

Using a sequence of real numbers with an integer number of cycles m, for all k, except at and at .

The input is a cosine wave with data points. The program that is used to compute the DFT for the input signal is illustrated below.

|  |
| --- |
| #include <stdio.h>  #include <math.h>  void dft(short \*x, short k, int \*out); //function prototype  #define N 8 //number of data values  float pi = 3.1416;  int out[2] = {0,0}; //init Re and Im results  short x[N] = {1000,707,0,-707,-1000,-707,0,707}; //1-cycle cosine  //short x[N]={0,602,974,974,602,0,-602,-974,-974,-602,  // 0,602,974,974,602,0,-602,-974,-974,-602};//2-cycles sine  void dft(short \*x, short k, int \*out) //DFT function  {  int sumRe = 0, sumIm = 0; //init real/imag components  float cs = 0, sn = 0; //init cosine/sine components  int i = 0;  for (i = 0; i < N; i++) //for N-point DFT  {  cs = cos(2\*pi\*(k)\*i/N); //real component  sn = sin(2\*pi\*(k)\*i/N); //imaginary component  sumRe = sumRe + x[i]\*cs; //sum of real components  sumIm = sumIm - x[i]\*sn; //sum of imaginary components  }  out[0] = sumRe; //sum of real components  out[1] = sumIm; //sum of imaginary components  }  void main()  {  int j;  for (j = 0; j < N; j++)  {  dft(x,j,out); //call DFT function  } |

To test the results, build the project, load the program and follow these steps

1. Select View →Watch Window and insert the two expressions and (right click on the Watch window). Click on to expand and view and , which represent the real and imaginary components, respectively.
2. Place a breakpoint at the bracket “}” that follows the DFT function call.
3. Select Debug →Animate (Animation speed can be controlled through Options). Verify that the real component value is large (3996) at and at , while small otherwise. Since is a one-cycle sequence, . Since the number of points is a “spike” occurs at and at .
4. Use the two-cycle sine data table (in the program) with 20 points as input . Within the program, change , comment the table that corresponds to the cosine (first input), and instead use the sine table values. Rebuild and animate again. Verify a large negative value at and a large positive value at . For a real-time implementation, the magnitude of can be found. With , the frequency generated would correspond to .

## FFT of a Real-Time Input Signal Using an FFT Function in C

In this part of the experiment a 256 point FFT in real time will be implemented in real time using an external input signal. The implementation of this part of the experiment relies on the FFT function and on the code shown below

|  |
| --- |
| #include "dsk6713\_aic23.h"  Uint32 fs=DSK6713\_AIC23\_FREQ\_8KHZ;  #include <math.h>  #define PTS 256 //# of points for FFT  #define PI 3.14159265358979  typedef struct {float real,imag;} COMPLEX;  void FFT(COMPLEX \*Y, int n); //FFT prototype  float iobuffer[PTS]; //as input and output buffer  float x1[PTS]; //intermediate buffer  short i; //general purpose index variable  short buffercount = 0; //number of new samples in iobuffer  short flag = 0; //set to 1 by ISR when iobuffer full  COMPLEX w[PTS]; //twiddle constants stored in w  COMPLEX samples[PTS]; //primary working buffer  main()  {  for (i = 0 ; i<PTS ; i++) // set up twiddle constants in w  {  w[i].real = cos(2\*PI\*i/512.0); //Re component of twiddle constants  w[i].imag =-sin(2\*PI\*i/512.0); //Im component of twiddle constants  }  comm\_intr(); //init DSK, codec, McBSP  while(1) //infinite loop  {  while (flag == 0) ; //wait until iobuffer is full  flag = 0; //reset flag  for (i = 0 ; i < PTS ; i++) //swap buffers  {  samples[i].real=iobuffer[i]; //buffer with new data  iobuffer[i] = x1[i]; //processed frame to iobuffer  }  for (i = 0 ; i < PTS ; i++)  samples[i].imag = 0.0; //imag components = 0  FFT(samples,PTS); //call function FFT.c  for (i = 0 ; i < PTS ; i++) //compute magnitude  {  x1[i] = sqrt(samples[i].real\*samples[i].real  + samples[i].imag\*samples[i].imag)/16;  }  x1[0] = 32000.0; //negative spike for reference  } //end of infinite loop  } //end of main  interrupt void c\_int11() //ISR  {  output\_sample((short)(iobuffer[buffercount]));//output from iobuffer  iobuffer[buffercount++]=(float)((short)input\_sample());//input>iobuffer  if (buffercount >= PTS) //if iobuffer full  {  buffercount = 0; //reinit buffercount  flag = 1; //set flag  }  } |

The twiddle constants are generated within the program. The imaginary components of the input data are set to zero to illustrate this implementation. The magnitude of the resulting FFT (scaled) is taken for output to the codec. Three buffers are used:

1. samples: contains the data to be transformed
2. iobuffer: used to output processed data as well as acquiring new input sampled data
3. x1: contains the magnitude (scaled) of the transformed (processed) data

In every sample period, an output value from a buffer (iobuffer) is sent to the codec’s DAC and an input value is acquired and stored into the same buffer. An index (buffercount) to this buffer is used to set a flag when this buffer is full.

When this buffer is full, it is copied to another buffer (samples), which will be used when calling the FFT function. The magnitude (scaled) of the processed FFT data, contained in a buffer x1, can now be copied to the I/O buffer, iobuffer, for output. In a filtering algorithm, processing can be done as each new sample is acquired. On the other hand, an FFT algorithm requires that an entire frame of data be available for processing.

### Testing procedures

1. To test the program input a 2-kHz sine wave with amplitude of approximately .
2. Connect the LINE OUT of the DSK kit to the oscilloscope and observe the measured signal
3. Measure the frequency of the first and the second positive spikes and explain the result you got
4. Measure the time interval between the two negative spikes. What does this interval represent?

## Fast Convolution

The following example shows how the FFT enables signals to be processed in the frequency domain. Fast convolution takes less computational effort and is potentially more accurate than time-domain implementation of FIR filters having very large numbers of coefficients.

To implement an FIR filter and illustrate the fast convolution’s overlap-add scheme. TI’s floating-point FFT support functions, bitrev, digitrev\_index, and cfftr2\_dit were introduced in examples 6.3 and 6.4 in Rulph Chassing). In addition, TI’s inverse complex FFT function icfftr2\_dif (radix-2, DIF) is used here. This function expects its input to be scrambled or to be in bit-reversed order. As a result, the bit-reversed output of the complex FFT function cfftr2\_d it need not be reordered, and the support files for bit reversal, digitrev\_index.c and bitrev.sa, are not needed after the FFT section of the program. Both data (samples) and filter coefficients (h) are in bit-reversed order and may be multiplied together in that order.

The code for implementing this project is shown below

|  |
| --- |
| #include "DSK6713\_AIC23.h" //codec-DSK support file  Uint32 fs=DSK6713\_AIC23\_FREQ\_8KHZ;//set sampling rate  #include <math.h>  #include "coeffs.h" //time domain FIR coefficients  #define PI 3.14159265358979  #define PTS 256 //number of points for FFT  #define SQRT\_PTS 16 //used in twiddle factor calc.  #define RADIX 2 //passed to TI FFT routines  #define DELTA (2\*PI)/PTS  typedef struct Complex\_tag {float real, imag;} COMPLEX ;  #pragma DATA\_ALIGN(W, sizeof(COMPLEX))  #pragma DATA\_ALIGN(samples, sizeof(COMPLEX))  #pragma DATA\_ALIGN(h, sizeof(COMPLEX))  COMPLEX W[PTS/RADIX] ; //twiddle factor array  COMPLEX samples[PTS]; //processing buffer  COMPLEX h[PTS]; //FIR filter coefficients  short buffercount = 0; //buffer count for iobuffer samples  float iobuffer[PTS/2]; //primary input/output buffer  float overlap[PTS/2]; //intermediate result buffer  short i; //index variable  short flag = 0; //set to indicate iobuffer full  float a, b; //variables used in complex multiply  short NUMCOEFFS = sizeof(coeffs)/sizeof(float);  short iTwid[SQRT\_PTS] ; //PTS/2 + 1 > sqrt(PTS)  interrupt void c\_int11(void) //ISR  {  output\_sample((short)(iobuffer[buffercount]));  iobuffer[buffercount++] = (float)((short)input\_sample());  if (buffercount >= PTS/2) //for overlap-add method iobuffer  { //is half size of FFT used  buffercount = 0;  flag = 1;  }  }  main()  { //set up array of twiddle factors  digitrev\_index(iTwid, PTS/RADIX, RADIX);  for(i = 0 ; i < PTS/RADIX ; i++)  {  W[i].real = cos(DELTA\*i);  W[i].imag = sin(DELTA\*i);  }  bitrev(W, iTwid, PTS/RADIX); //bit reverse order W  for (i = 0 ; i<PTS ; i++) //initialise PTS element  { //of COMPLEX to hold real-valued  h[i].real = 0.0; //time domain FIR filter coefficients  h[i].imag = 0.0;  }  for (i = 0 ; i < NUMCOEFFS ; i++)  { //read FIR filter coeffs  h[i].real = coeffs[i]; //NUMCOEFFS should be less than PTS/2  }  cfftr2\_dit(h,W,PTS); //transform filter coeffs  comm\_intr(); //initialise DSK, codec, McBSP  while(1) //frame processing infinite loop  {  while (flag == 0); //wait for iobuffer full  flag = 0;  for (i = 0 ; i<PTS/2 ; i++) //iobuffer into first half of  { //samples buffer  samples[i].real = iobuffer[i];  iobuffer[i] = overlap[i]; //previously processed output  } //to iobuffer  for (i = 0 ; i<PTS/2 ; i++)  { //second half of samples to overlap  overlap[i] = samples[i+PTS/2].real;  samples[i+PTS/2].real = 0.0;//zero-pad input from iobuffer  }  for (i=0 ; i<PTS ; i++)  samples[i].imag = 0.0; //init imag parts in samples buffer  cfftr2\_dit(samples,W,PTS); //complex FFT function from TI  for (i=0 ; i<PTS ; i++) //frequency-domain representation  { //complex multiply samples by h  a = samples[i].real;  b = samples[i].imag;  samples[i].real = h[i].real\*a - h[i].imag\*b;  samples[i].imag = h[i].real\*b + h[i].imag\*a;  }  icfftr2\_dif(samples,W,PTS); //inverse FFT function from TI  for (i=0 ; i<PTS ; i++)  samples[i].real /= PTS;  for (i=0 ; i<PTS/2 ; i++) //add first half of samples  overlap[i] += samples[i].real; //to overlap  } //end of while(1)  } //end of main() |

### Testing procedure

1. Build this project as Fastconvo
2. Verify that the time-domain filter coefficients are implementing a 2-kHz bandpass filter.

Several buffers are used, and iobuffer is the primary input/output buffer. At each sampling interval, the ISR is executed. The next output value is read from iobuffer, output to the codec, and then replaced by a new input sample. After PTS/2 sampling instants, iobuffer contains a new frame of PTS/2 input samples.

This situation is signaled by setting flag to 1. The main program waits for this flag signal using

while (flag == 0);

and subsequently carries out the following operations:

1. Resets flag to 0
2. Copies the contents of the buffer iobuffer (frame of new input samples) to the first PTS/2 locations of the buffer samples
3. Copies the contents of the buffer overlap (previously computed frame of output samples) to the buffer iobuffer
4. Processes the new frame of input samples to compute the next frame of output samples

The frame processing operation (within an infinite loop) has PTS/2 sampling periods in which to execute and comprises the following steps:

1. The contents of the last PTS/2 locations of the samples buffer (real parts) are copied to the overlap buffer. These time-domain data may be thought of as the overlapping latter half (PTS/2 samples) of the previous frame processing operation.
2. The last PTS/2 locations of the buffer samples are zero-padded. The buffer samples now contains PTS/2 new samples followed by PTS/2 zeros.
3. The buffer samples is transformed in-place into the frequency domain using a PTS-point FFT.
4. The complex frequency-domain sample values are multiplied by the complex frequency-domain filter coefficients stored in h.
5. The results are transformed back into the time domain by applying a PTSpoint IFFT to the contents of the samples buffer. The resulting PTS time domain samples will be real-valued.
6. The contents of the first PTS/2 locations of the buffer samples (i.e., the former half of the current frame processing result) are added to the contents of the overlap buffer.

Since the input and output signals are real-valued, so are the buffers iobuffer and overlap. However, since the frequency-domain representation of these signals is complex, the buffer samples and the array of filter coefficients h are complex, requiring two floating-point values (real and imaginary parts) per sample. 240 Fast Fourier Transform

A faster and more efficient implementation of buffering is possible using pointers rather than copying data from one buffer to another, but the latter approach is adopted for purposes of clarity.

# Experiment 11 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 there is a code with *m* parity bits and 2m− *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).

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

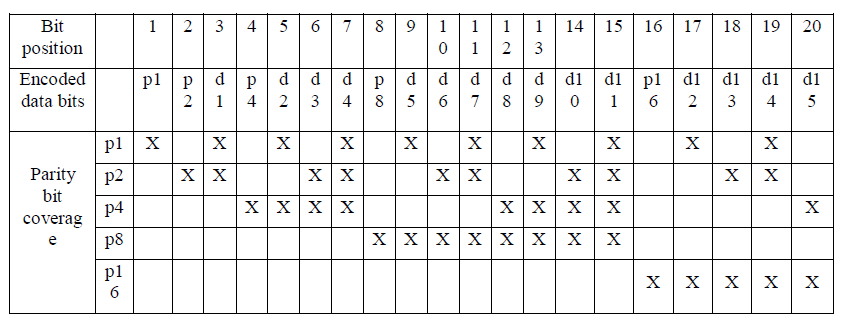
position, are data bits.

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.

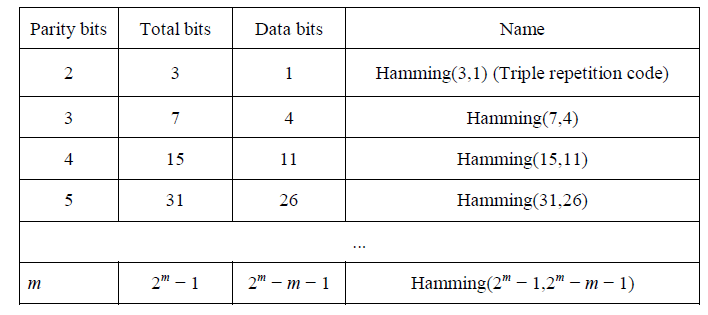
This general rule can be shown visually:



As you can see, if you have *m* parity bits, it can cover bits from 1 up to . If we

subtract out the parity bits, we are left with we can use for the data. As

*m* varies, we get all the possible Hamming codes:



Calculating the Hamming Code

The key to the Hamming Code is the use of extra parity bits to allow the identification

of 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,...)

1. 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.

Detection of errors in the hamming code

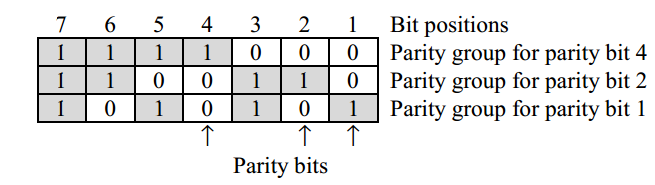
Every integer there is a () bit Hamming code which contains parity bits and

information bits.The parity bits are intermixed with the information bits as follows:

If we number the bit positions from 1 to , the bits in position , where

, 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 , we have a Hamming code as shown below:



So to find the bit error position:

Then, the position of bit error is .

## Experimental procedures

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

1. Set the sampling frequency to
2. 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];

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];

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

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

}

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

}

1. 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", eb);

}

else

printf("\nthere is no error\n");

printf("\n The corrected recieved data\n");

printf("%d",c[2]);

printf("%d",c[4]);

printf("%d",c[5]);

printf("%d",c[6]);

}

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

}

# Dual-Tone Multi frequency DTMF

## Objectives

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

## Theory of ASK

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; ; 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 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 1 Extended 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
2. Set the number of sample
3. Write a program to read the DIP switches and generates the required tones as per
4. You may consider the binary codes that corresponds to each number as shown in

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 2

1. Use the following code to generate an ASK modulated signal.

|  |
| --- |
| #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;  }  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.  } |

#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

#include"dsk6713\_aic23.h"

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)==1)&&(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

}

if((DSK6713\_DIP\_get(0)==0)&&(DSK6713\_DIP\_get(1)==1)&&(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\_table6[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==1)&&(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\_table7[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

}

if((DSK6713\_DIP\_get(0)==0)&&(DSK6713\_DIP\_get(1)==0)&&(DSK6713\_DIP\_get(2)==1)&&(DSK6713\_DIP\_get(3)==1))

{

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\_table8[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

}

if((DSK6713\_DIP\_get(0)==0)&&(DSK6713\_DIP\_get(1)==0)&&(DSK6713\_DIP\_get(2)==1)&&(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\_table2[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==0)&&(DSK6713\_DIP\_get(2)==1)&&(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\_table2[i] + sine\_table6[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

}

if((DSK6713\_DIP\_get(0)==0)&&(DSK6713\_DIP\_get(1)==1)&&(DSK6713\_DIP\_get(2)==1)&&(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\_table2[i] + sine\_table7[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==0)&&(DSK6713\_DIP\_get(2)==1)&&(DSK6713\_DIP\_get(3)==1))

{

output\_sample(output); // the value in the buffer ouput indexed by the variable loop is written on to the codec.

output = sine\_table2[i] + sine\_table8[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==1)&&(DSK6713\_DIP\_get(2)==1)&&(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\_table3[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

}

if((DSK6713\_DIP\_get(0)==0)&&(DSK6713\_DIP\_get(1)==0)&&(DSK6713\_DIP\_get(2)==0)&&(DSK6713\_DIP\_get(3)==1))

{

output\_sample(output); // the value in the buffer ouput indexed by the variable loop is written on to the codec.

output = sine\_table3[i] + sine\_table6[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==0)&&(DSK6713\_DIP\_get(2)==0)&&(DSK6713\_DIP\_get(3)==1))

{

output\_sample(output); // the value in the buffer ouput indexed by the variable loop is written on to the codec.

output = sine\_table3[i] + sine\_table7[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

}

if((DSK6713\_DIP\_get(0)==0)&&(DSK6713\_DIP\_get(1)==1)&&(DSK6713\_DIP\_get(2)==1)&&(DSK6713\_DIP\_get(3)==1))

{

output\_sample(output); // the value in the buffer ouput indexed by the variable loop is written on to the codec.

output = sine\_table3[i] + sine\_table8[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==1)&&(DSK6713\_DIP\_get(2)==1)&&(DSK6713\_DIP\_get(3)==1))

{

output\_sample(output); // the value in the buffer ouput indexed by the variable loop is written on to the codec.

output = sine\_table4[i] + sine\_table8[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

}

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\_table4[i] + sine\_table6[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==1)&&(DSK6713\_DIP\_get(2)==1)&&(DSK6713\_DIP\_get(3)==1))

{

output\_sample(output); // the value in the buffer ouput indexed by the variable loop is written on to the codec.

output = sine\_table4[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

}

if((DSK6713\_DIP\_get(0)==1)&&(DSK6713\_DIP\_get(1)==1)&&(DSK6713\_DIP\_get(2)==0)&&(DSK6713\_DIP\_get(3)==1))

{

output\_sample(output); // the value in the buffer ouput indexed by the variable loop is written on to the codec.

output = sine\_table4[i] + sine\_table7[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.

}