NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE

(Accredited by NAAC, Approved by AICTE New Delhi, Affiliated to APJKTU)

Pampady, Thiruvilwamala(PO), Thrissur(DT), Kerala 680 588

DEPARTMENT OF ELECTRONICS & COMMUNICATION

FACULTY MANUAL

ECL 333

DIGITAL SIGNAL PROCESSING LAB

VISION OF THE INSTITUTION
To mould true citizens who are millennium leaders and catalysts of change through excellence in education.
MISSION OF THE INSTITUTION
NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values. We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Electronics and Communication Engineering

  M.Tech in VLSI
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of Dr. A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Providing Universal Communicative Electronics Engineers with corporate and social relevance towards sustainable developments through quality education.

DEPARTMENT MISSION

MD 1: Imparting Quality education by providing excellent teaching, learning environment.

MD 2: Transforming and adopting students in this knowledgeable era, where the electronic gadgets (things) are getting obsolete in short span.

MD 3: To initiate multi-disciplinary activities to students at earliest and apply in their respective fields of interest later.

MD 4: Promoting leading edge Research & Development through collaboration with academia & industry.

PROGRAMME EDUCATIONAL OBJECTIVES

PEOI. To prepare students to excel in postgraduate programmes or to succeed in industry / technical profession through global, rigorous education and prepare the students to practice and innovate recent fields in the specified program/ industry environment.

PEO2. To provide students with a solid foundation in mathematical, Scientific and engineering fundamentals required to solve engineering problems and to have strong practical knowledge required to design and test the system.

PEO3. To train students with good scientific and engineering breadth so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

PEO4. To provide student with an academic environment aware of excellence, effective communication skills, leadership, multidisciplinary approach, written ethical codes and the life-long learning needed for a successful professional career.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality
System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software
products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

PROGRAM OUTCOMES (PO'S)

Engineering Graduates will be able to:

PO 1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO 2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO 3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO 11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO'S)

PSO 1: Design and develop Mechatronics systems to solve the complex engineering problem by integrating electronics, mechanical and control systems.

PSO 2: Apply the engineering knowledge to conduct investigations of complex engineering problem related to instrumentation, control, automation, robotics and provide solutions.

SYLLABUS

ECL333	DIGITAL SIGNAL PROCESSING LABORATORY	CATEGORY	L	T	P	CREDIT
		PCC	0	0	3	2

Preamble:

The following experiments are designed to make the student do real time DSP computing.

Dedicated DSP hardware (such as TI or Analog Devices development/evaluation boards) will be used for realization.

Prerequisites:

- ECT 303 Digital Signal Processing
- EST 102 Programming in C

Course Outcomes: The student will be able to

CO 1	Simulate digital signals.
CO 2	verify the properties of DFT computationally
CO 3	Familiarize the DSP hardware and interface with computer.
CO 4	Implement LTI systems with linear convolution.
CO 5	Implement FFT and IFFT and use it on real time signals.
CO 6	Implement FIR low pass filter.
CO 7	Implement real time LTI systems with block convolution and FFT.

Mapping of Course Outcomes with Program Outcomes

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO10	PO11	PO12
CO1	3	3	1	2	3	0	0	0	3	0	0	1
CO2	3	3	1	2	3	0	0	0	3	0	0	1
CO3	3	3	3	2	3	0	0	0	3	1	0	1
CO4	3	3	1	2	3	0	0	0	3	0	0	1
CO5	3	3	1	1	3	0	0	0	0	0	0	1
CO6	3	3	1	1	3	0	0	0	0	0	0	1
CO7	3	3	1	3	3	0	0	0	3	3	0	0

Assessment Pattern Mark

Distribution:

Total Mark	CIE	ESE
150	50	100

Continuous Internal Evaluation Pattern:

Each experiment will be evaluated out of 50 credits continuously as

Attribute	Mark
Attendance	15
Continuous assessment	30
Internal Test (Immediately before the second series test)	30

End Semester Examination Pattern: The following guidelines should be followed regarding award of marks

Attribute	Mark
Preliminary work	15
Implementing the work/ Conducting the experiment	10
Performance, result and inference (usage of equipments and trouble shooting)	25
Viva voce	20
Record	5

Course Level Assessment

QuestionsCO1-Simulation

of Signals

1. Write a Python/MATLAB/SCILAB function to generate a rectangular pulse.
2. Write a Python/MATLAB/SCILAB function to generate a triangular pulse.

CO2-Verfication of the Properties of DFT

1. Write a Python/MATLAB/SCILAB function to compute the *N* -point DFT

matrix and plot its real and imaginary parts.

2. Write a Python/MATLAB/SCILAB function to verify Parseval's theoremfor $N = 1024$.

CO3-Familarization of DSP Hardware

1. Write a C function to control the output LEDs with input switches.
2. Write a C function to connect the analog input port to the output port and test witha microphone.

CO4-LTI System with Linear Convolution

1. Write a function to compute the linear convolution and download to the hardware target and test with some signals.

CO5-FFT Computation

1. Write and download a function to compute N point FFT to the DSP hardwaretarget and test it on real time signal.
2. Write a C function to compute IFFT with FFT function and test in on DSP hardware.

CO6-Implementation of FIR Filter

1. Design and implement an FIR low pass filter for a cut off frequency of 0.1π and test it with an AF signal generator.

CO7-LTI Systems by Block Convolution

1. Implement an overlap add block convolution for speech signals on DSPtarget.

List of Experiments

(All experiments are mandatory.)

Experiment 1. Simulation of Signals Simulate the following signals using Python/ Scilab/MATLAB.

1. Unit impulse signal
2. Unit pulse signal
3. Unit ramp signal
4. Bipolar pulse
5. Triangular signal

Experiment 2. Verification of the Properties of DFT

- Generate and appreciate a DFT matrix.
  1. Write a function that returns the N point DFT matrix $\mathbf{V_N}$ for a given N.
  2. Plot its real and imaginary parts of $\mathbf{V_N}$ as images using *matshow* or *imshow* commands (in Python) for $N = 16$, $N = 64$ and $N = 1024$
  3. Compute the DFTs of 16 point, 64 point and 1024 point random sequences using the above matrices.
  4. Observe the time of computations for $N = 2^{\gamma}$ for $2 \leq \gamma \leq 18$ (You may use the *time* module in Python).
  5. Use some iterations to plot the times of computation against γ. Plot and understand this curve. Plot the times of computation for the *fft* function over this curve and appreciate the computational saving with FFT.
- Circular Convolution.
  1. Write a python function *circcon.py* that returns the circular convoluton of an N_1 point sequence and an N_2 point sequence given at the input. The easiest way is to convert a linear convolution into circular convolution with $N = max(N_1, N_2)$.
- Parseval's Theorem

For the complex random sequences $x_1[n]$ and $x_2[n]$,

$$\sum_{n=0}^{N-1} x_1[n]x_2^*[n] = \frac{1}{N}\sum_{k=0}^{N-1} X_1[k]X_2^*[k]$$

1. Generate two random complex sequences of say 5000 values.
2. Prove the theorem for these signals.

Experiment 3. Familarization of DSP Hardware

1. Familiarization of the code composer studio (in the case of TI hard- ware) or Visual DSP (in the case of Analog Devices hardware) or any equivalent cross compiler for DSP programming.
2. Familiarization of the analog and digital input and output ports of the DSP board.
3. Generation and cross compilation and execution of the C code to con- nectthe input digital switches to the output LEDs.
4. Generation and cross compilation and execution of the C code to con- nect the input analog port to the output. Connect a microphone, speak into itand observe the output electrical signal on a DSO and store it.
5. Document the work.

Experiment 4. Linear convolution

1. Write a C function for the linear convolution of two arrays.
2. The arrays may be kept in different files and downloaded to the DSP hardware.
3. Store the result as a file and observe the output.
4. Document the work.

Experiment 5. FFT of signals

1. Write a C function for N - point FFT.
2. Connect a precision signal generator and apply 1 *mV* , 1 *kHz* sinusoid atthe analog port.
3. Apply the FFT on the input signal with appropriate window size and observe the result.
4. Connect microphone to the analog port and read in real time speech.
5. Observe and store the FFT values.
6. Document the work.

Experiment 6. IFFT with FFT

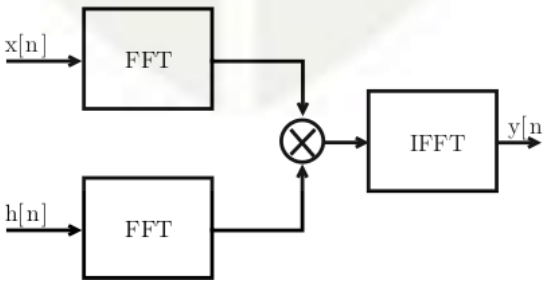
1. Use the FFT function in the previous experiment to compute the IFFT of the input signal.
2. Apply IFFT on the stored FFT values from the previous experiments and observe the reconstruction.
3. Document the work.

Experiment 7. FIR low pass filter

1. Use Python/scilab to implement the FIR filter response $h[n] = \frac{\sin(\omega_c n)}{\pi n}$ for a filter size $N = 50$, $\omega_c = 0.1\pi$ and $\omega_c = 0.3\pi$.
2. Realize the hamming($w_H[n]$) and kaiser ($w_K[n]$) windows.
3. Compute $h[n]w[n]$ in both cases and store as file.
4. Observe the low pass response in the simulator.
5. Download the filter on to the DSP target board and test with 1 *mV* sinusoid from a signal generator connected to the analog port.
6. Test the operation of the filters with speech signals.
7. Document the work.

Experiment 8. Overlap Save Block Convolution

1. Use the file of filter coefficients From the previos experiment.
2. Realize the system shown below for the input speech signal $x[n]$.



3. Segment the signal values into blocks of length $N = 2000$. Pad the last

block with zeros, if necessary.

4. Implement the *overlap save* block convolution method
5. Document the work.

Experiment 9. Overlap Add Block Convolution

1. Use the file of filter coefficients from the previous experiment.
2. Realize the system shown in the previous experiment for the input speechsignal $x[n]$.
3. Segment the signal values into blocks of length $N = 2000$. Pad the lastblock with zeros, if necessary.
4. Implement the *overlap add* block convolution method
5. Document the work.

Schedule of Experiments: Every experiment should be completed in three hours.

Textbooks

1. Vinay K. Ingle, John G. Proakis, “Digital Signal Processing UsingMATLAB.”
2. Allen B. Downey, “Think DSP: Digital Signal Processing using Python.”
3. Rulph Chassaing, “DSP Applications Using C and the TMS320C6x DSK(Topics in Digital Signal Processing)”

PREPARATION FOR THE LABORATORY SESSION
GENERAL INSTRUCTIONS TO STUDENTS

1. Read carefully and understand the description of the experiment in the lab manual. You may go to the lab at an earlier date to look at the experimental facility and understand it better. Consult the appropriate references to be completely familiar with the concepts and hardware.

2. Make sure that your observation for previous week experiment is evaluated by the faculty member and you have transferred all the contents to your record before entering to the lab/workshop.

3. At the beginning of the class, if the faculty or the instructor finds that a student is not adequately prepared, they will be marked as absent and not be allowed to perform the experiment.

4. Bring necessary material needed (writing materials, graphs, calculators, etc.) to perform the required preliminary analysis. It is a good idea to do sample calculations and as much of the analysis as possible during the session. Faculty help will be available. Errors in the procedure may thus be easily detected and rectified.

5. Please actively participate in class and don't hesitate to ask questions. Please utilize the teaching assistants fully. To encourage you to be prepared and to read the lab manual before coming to the laboratory, unannounced questions may be asked at any time during the lab.

6. Carelessness in personal conduct or in handling equipment may result in serious injury to the individual or the equipment. Do not run near moving machinery/equipment. Always be on the alert for strange sounds. Guard against entangling clothes in moving parts of machinery.

7. Students must follow the proper dress code inside the laboratory. To protect clothing from dirt, wear a lab coat. Long hair should be tied back. Shoes covering the whole foot will have to be worn.

8. In performing the experiments, please proceed carefully to minimize any water spills, especially on the electric circuits and wire.

9. Maintain silence, order and discipline inside the lab. Don't use cell phones inside the laboratory.

10. Any injury no matter how small must be reported to the instructor immediately.

11. Check with faculty members one week before the experiment to make sure that you have the handout for that experiment and all the apparatus.

AFTER THE LABORATORY SESSION

1. Clean up your work area.

2. Check with the technician before you leave.

3. Make sure you understand what kind of report is to be prepared and due submission of record is next lab class.

4. Do sample calculations and some preliminary work to verify that the experiment was successful

MAKE-UPS AND LATE WORK

Students must participate in all laboratory exercises as scheduled. They must obtain permission from the faculty member for absence, which would be granted only under justifiable circumstances. In such an event, a student must make arrangements for a make-up laboratory, which will be scheduled when the time is available after completing one cycle. Late submission will be awarded less mark for record and internals and zero in worst cases.

LABORATORY POLICIES

1. Food, beverages & mobile phones are not allowed in the laboratory at any time.

2. Do not sit or place anything on instrument benches.

3. Organizing laboratory experiments requires the help of laboratory technicians and staff. Be punctual.

1. Simulation of signals:

In this experiment, we will generate some elementary signals using Python. First import numpy and matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt
```

If you are using a Jupyter notebook, also give the command `%matplotlib inline` to generate the plots in the notebook itself, rather than in a separate figure window

a. Generate and plot a discrete-time impulse using the code below:

```
n = np.arange(-5, 6) # n=[-5, -4, …., 5]
x = np.zeros_like(n) # x: array of 0s with same no. of elements as
n
x[n == 0] = 1 # set x[0] = 1
plt.stem(n, x)
plt.xticks(n);
```

b. Generate a discrete-time pulse signal $x[n] = \begin{cases} 1; & 0 \le n \le 4 \\ 0; & otherwiwse \end{cases}$

```
n = np.arange(-5, 6)
x = np.zeros_like(n)
x[(n >= 0) & (n <= 4) ] = 1
plt.stem(n, x)
plt.xticks(n);
```

c. Generate and stem the discrete-time bipolar pulse signal $x[n] = \begin{cases} -1; -3 \le n < 0 \\ 1; 0 \le n < 3 \end{cases}$

d. Generate and plot a discrete-time ramp signal.

e. Generate a triangular signal.

```
# x[n] = {0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0}
a = np.arange(6)
b = np.arange(4, -1, -1)
x = np.concatenate([a, b])
plt.stem(x)
plt.xticks(np.arange(11));
```

f. Plot the discrete-time signal x[n]={ 2, -1 , 4, 1} using `plt.stem()`. Use `np.array` to create `x`

g. Generate the complex exponential sequence $x[n] = e^{\left(-\frac{1}{12}+j\frac{\pi}{6}n\right)}$. Display both the real and imaginary parts of the signal from n=0 to n=40 using `plt.subplot()`. *Hint:* Use `np.exp, np.pi, np.real, np.imag.` You have to write `1j` to represent j

h. Generate $x[n] = (0 \cdot 95)^n \cos(0.1\pi n)$ for n=0 to 50. *Note: Exponentiation operator in Python is* **

i. Generate 50 samples of the following discrete time sequences and display using stem: $x[n] = 20(0.9)^n$, $x[n] = 0.2(1.2)^n$, $x[n] = (-0.8)^n$, $x[n] = -4(0.8)^n$, $x[n]=2[n(.9)^n]$

j. Generate the discrete-time sinusoids: x1[n]=sin(0.2πn), x2[n]=sin(1.8.πn), x3[n]=sin(2.2πn). Compare the plots generated for the three cases and comment on your result.

k. Even though all signals that we generate on a digital computer are necessarily discrete, we can simulate a continuous time signal by sampling at a high rate and graphing it as a continuous curve using `plt.plot()`.

```
# Continuous time sinusoids
t = np.arange(0, 1, .02) # Fs= 50Hz
x = np.sin(2*np.pi*1*t) # 1 Hz
y = np.sin(2*np.pi*2*t) # 2 Hz
plt.plot(t, x, label = '1 Hz')
plt.plot(t, y, label = '2 Hz')
plt.legend();
```

l. Generate and plot a sine wave and the full wave rectified version of it from -2π to +2π.

m. Random signals: A random signal of length N with samples uniformly distributed in the interval [0,1) can be generated by using the command `x = np.random.random_sample(N)`. Generate and display a random signal of length 100 whose elements are uniformly distributed in the interval [−2, 2).

n. Likewise, a random signal x[n] of length N with samples normally distributed with zero mean and unity std. dev. can be generated by using the following command

```
x = np.random.normal(0.0, 1.0, N)
```

Generate and display a Gaussian random signal of length 75 whose elements are normally distributed with zero mean and a variance of 3. (Check your result with `np.mean()` and `np.var()`.

o. Generate and plot a sinewave corrupted with zero mean Gaussian noise.

```
t = np.arange(0, 2, .01)
signal = np.sin(2*np.pi*t)
noise = np.random.normal(0, 0.1, t.size)
noisy_signal = signal + noise
plt.plot(noisy_signal);
```

Plot the noisy signal for various values of std dev for the noise

2. DFT:

The DFT X[k] of a finite length sequence x[n] defined for n=0… N-1 can be obtained by sampling its DTFT $X(e^{j\omega})$ on the ω axis between $0 \le \omega < 2\pi$ at $\omega_k = \frac{2\pi k}{N}$, $k = 0 \ldots N-1$. i.e. $DFT\{x[n]\} = X[k] = X(e^{j\omega})\Big|\omega = \frac{2\pi k}{N} = \sum_{n=0}^{N-1} x[n]\, e^{-\frac{j2\pi kn}{N}}$

Using the commonly used notation $W_N = e^{\frac{-j2\pi}{N}}$, $X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{kn}$, $k = 0 \ldots N-1$.

It is possible to view the DFT equation as a linear transformation on the sequence $x[n]$ as: $\boldsymbol{X} = \boldsymbol{D_N x}$ where $\boldsymbol{X}$ is the vector composed of N DFT samples $= \left[X[0],\ X[1], -\cdots X[N-1]\right]^{\boldsymbol{T}}$, $\boldsymbol{x}$ is the vector of N input samples $\boldsymbol{x} = \left[x[0],\ x[1], -\cdots x[N-1]\right]^{\boldsymbol{T}}$ and $\boldsymbol{D_N}$ is the NxN DFT matrix given by $\boldsymbol{D_N} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & W_N^1 & W_N^2 & \ldots & W_N^{(N-1)} \\ 1 & W_N^2 & W_N^4 & \ldots & W_N^{2(N-1)} \\ . & \ldots & \ldots & \ldots & \ldots \\ 1 & W_N^{(N-1)} & W_N^{2(N-1)} & \ldots & W_N^{(N-1)(N-!)} \end{bmatrix}$

To understand how this is correct, consider a general linear transformation of the form $\boldsymbol{y} = \boldsymbol{Ax}$. We can write this matrix equation in scalar form as $y_i = \sum_j a_{ij}\, x_j$. If we rearrange the DFT equation as $X[k] = \sum_{n=0}^{N-1} W_N^{kn} x[n]$, we see that the two are similar in form. Just as the (i,j)th element of $\boldsymbol{A}$ is a_{ij}, the (k,n)th element (counting from 0) of $\boldsymbol{D_N}$ is W_N^{kn}. This is exactly how the matrix $\boldsymbol{D_N}$ is arranged.

a. DFT matrix: We can generate the DFT matrix in a straightforward manner using two for loops:

```
N = 4;
D = np.empty((N, N), dtype=np.cdouble); # NxN complex matrix
W = np.exp(-1j*2*np.pi/N)
for k in np.arange(N):
    for n in np.arange(N):
        D[k, n]= W**(k*n)
np.round(D)
```

One for loop can be eliminated if we define k as an array:

```
N = 4;
D = np.empty((N, N), dtype=np.cdouble);
W = np.exp(-1j*2*np.pi/N)
k = np.arange(N)
for n in np.arange(N):
    D[:, n]= W**(k*n)
```

```
np.round(D)
```

b. Compute the DFT of the sequence $x[n] = \{1, 2, 3, 4\}$ using the matrix method.

```
x=np.array([[1, 2, 3, 4]]).T #column vector
X=D@x; # @ is the matrix multiplication operator
np.round(X)
```

We can verify the answer using the `scipy.fft.fft()` function which implements the fast FFT algorithm for DFT computation:

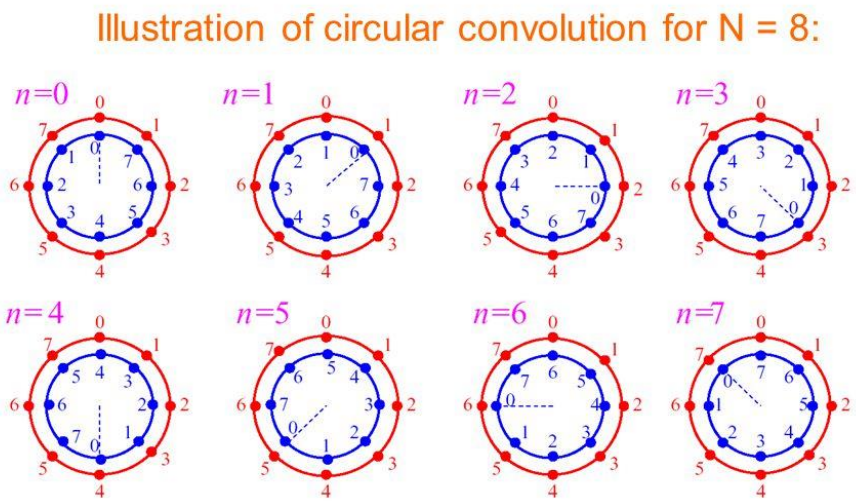
```
from scipy import fft
fft.fft(x, axis=0)
```

c. Compute the DFT of the sequence in part b by direct evaluation of the DFT equation.

d. Compare the computation time for the three methods using the `%%timeit` cell magic in Jupyter notebook

e. Compute a 64 point DFT matrix and plot it's real and imaginary parts using `plt.imshow(D.real)` and `plt.imshow(D.imag)`

f. Circular convolution: The circular convolution of two N-point sequences g[n] and h[n] is another N point sequence y[n] defined as $y[n] = \sum_{k=0}^{N-1} g[k]h[(n-k)_N]$. To perform circular convolution graphically, N samples of g(n) are equally spaced around the outer circle in the clockwise direction, and N samples of h(n) are displayed on the inner circle in the counterclockwise direction starting at the same point. Corresponding samples on the two circles are multiplied, and the products are summed to form an output. The successive value of the circular convolution is obtained by rotating the inner circle of one sample in the clockwise direction, and repeating the operation of computing the sum of corresponding products. This process is repeated until the first sample of inner circle lines up with the first sample of the exterior circle again.



This procedure can be coded as a function:

```
def circonv(g, h):
    if g.size != h.size:
        raise Exception("Sequences not of same length")
    N = g.size
    htr=np.concatenate([[h[0]], h[:0:-1]]) #circular time-
reversal
    y=np.zeros(N)
    for n in np.arange(N):
        y[n]=np.sum(g*htr)
        htr=np.roll(htr,1) #circular shift by 1 unit
    return y
```

g. The above function raises an exception when the sequences are not of the same length. Modify the function such that if the two sequences are not of the same length, the shorter sequence is padded with zeros and circular convolution is computed.

h. Using your function, compute the circular convolution of two sequences g[n] = [1 2 3 4 5] and h[n] = [2 2 0 1 1].

i. If y[n] is the N point circular convolution of the sequences g[n] and h[n] defined as $y[n] = g[n] \otimes h[n]$, then $Y[k] = G[k].H[k]$, where Y[k], G[k] and H[k] are the N-point DFTs of y[n], g[n] and h[n]. We can use this property to compute circular convolution as: $y[n] = IDFT\{ G[k] * H[k] \}$. Use this property to verify your circular convolution result in the previous part. Use `fft.fft()` and `fft.ifft()` functions in the `scipy` module to compute the DFTs and IDFT.

j. The N-point circular convolution operation can be written in matrix form as

$$\begin{bmatrix} y[0] \\ y[1] \\ y][2 \\ .. \\ y[N-1] \end{bmatrix} = \begin{bmatrix} h[0] & h[N-1] & h[N-2] & .. & h[1] \\ h[1] & h[0] & h[N-1] & .. & h[2] \\ h[2] & h[1] & h[0] & .. & h[3] \\ .. & .. & .. & .. & .. \\ h[N-1] & h[N-2] & h[N-3] & .. & h[0] \end{bmatrix} \begin{bmatrix} g[0] \\ g[1] \\ g[2] \\ .. \\ g[N-1] \end{bmatrix}$$

The elements in each row of the matrix above are obtained by circularly rotating the elements of the previous row to the right by 1 position. Such a matrix is called a *circulant matrix*. A circulant matrix can be generated using `scipy.linalg.circulant(c)`. The argument `c` is the first column of the matrix. Verify the circular convolution result using the matrix method.

k. Parseval's relation: If G[k] denotes the N-point DFT of the length N sequence g[n], then: $\sum_{n=0}^{N-1} |g[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |G[k]|^2$. The code below verifies the relation for a sequence g[n]:

```
# Verify Parseval's relation for a sequence g[n]
g = np.concatenate([np.arange(128), np.arange(128,-1,-1)])
```

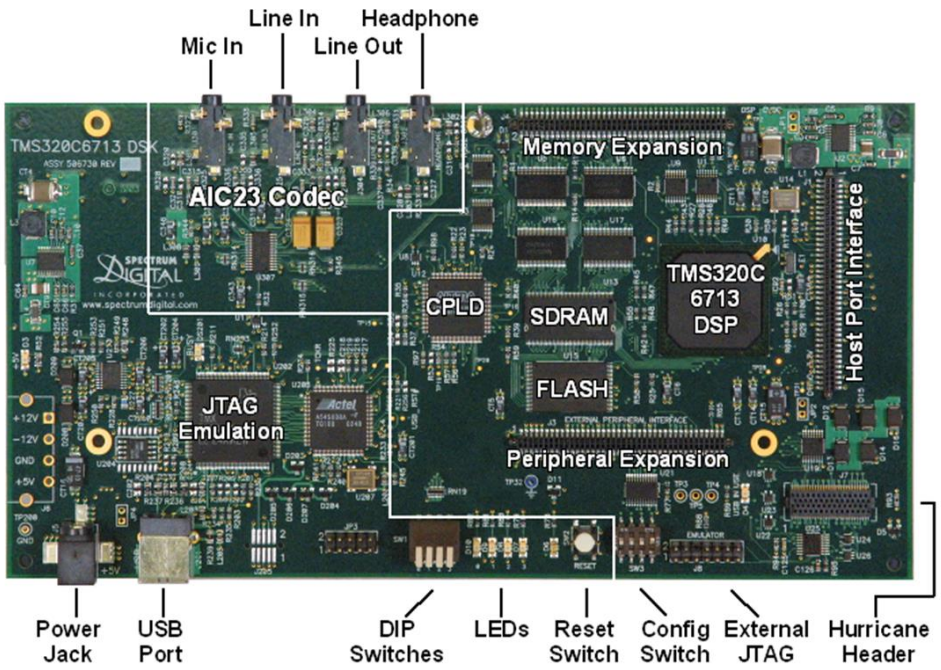
```
LHS = np.sum(g**2)
G = fft.fft(g)
RHS = 1/G.size * np.sum(np.abs(G)**2)
print(LHS, RHS)
```

I. Generate a random complex sequence of 500 values. Verify Parseval's relation for the sequence.

3. Familiarization of DSP Hardware and Software

TMS320C6713 DSP Starter Kit (DSK)

The C6713 DSK is a board that enables users to develop real-time DSP applications. The heart of the DSK is the Texas Instruments TMS320C6713 32-bit floating point Digital Signal Processor. DSP's differ from ordinary microprocessors in that they are specifically designed to rapidly perform the sum of products operation required in many discrete-time signal processing algorithms. They contain hardware parallel multipliers, and functions implemented by microcode in ordinary microprocessors are implemented by high-speed hardware in DSP's. Compared to fixed-point processors, floating-point processors are easier to program since issues like underflow, overflow, dynamic range etc can be ignored. The board is programmed using the TI



Code Composer Studio (CCS) software, which connects to the board through a USB.

The DSK comes with a full complement of on-board devices that suit a wide variety of application environments. Key features include:

- A Texas Instruments TMS320C6713 DSP operating at 225 MHz.
- An AIC23 stereo codec
- 16 Mbytes of synchronous DRAM
- 512 Kbytes of non-volatile Flash memory (256 Kbytes usable in default configuration)
- 4 user accessible LEDs and DIP switches
- Software board configuration through registers implemented in CPLD
- Configurable boot options
- Standard expansion connectors for daughter card use
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator
- Single voltage power supply (+5V)

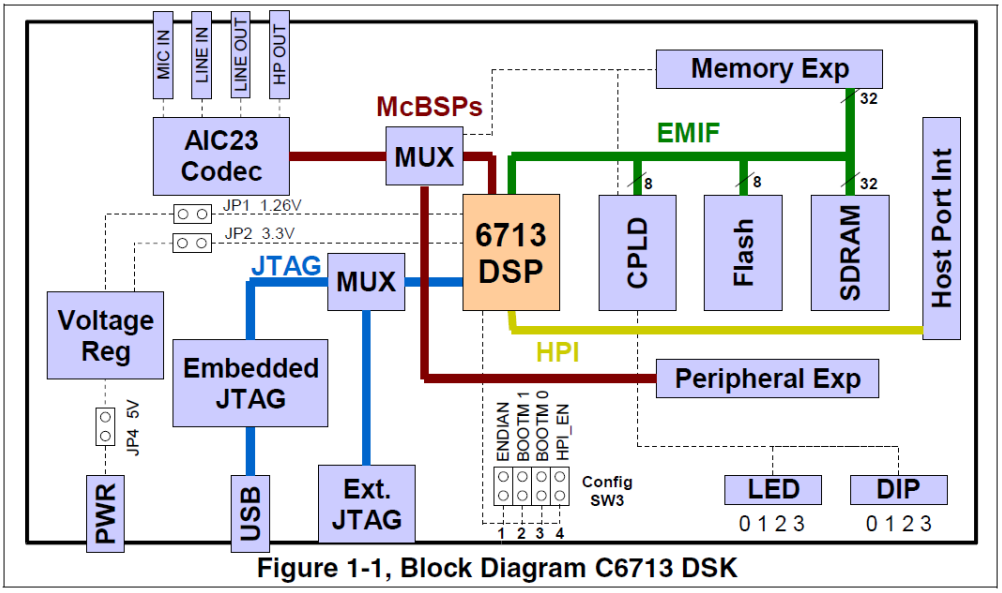


Figure 1-1, Block Diagram C6713 DSK

Functional Overview of the TMS320C6713 DSK

The DSP on the 6713 DSK interfaces to on-board peripherals through a 32-bit wide EMIF (External Memory InterFace). The SDRAM, Flash and CPLD are all connected to the bus. EMIF signals are also connected daughter card expansion connectors which are used for third party add-in boards.

The DSP interfaces to analog audio signals through an on-board AIC23 codec and four 3.5 mm audio jacks (microphone input, line input, line output, and headphone output). The codec can select the microphone or the line input as the active input. The analog output is driven to both the line out (fixed gain) and headphone (adjustable gain) connectors. Multichannel Buffered Serial Port 0 (McBSP0) is used to send commands to the codec control interface while McBSP1 is used for digital audio data. McBSP0 and McBSP1 can be re-routed to the expansion connectors in software.

A programmable logic device called a CPLD is used to implement glue logic that ties the board components together. The CPLD has a register-based user interface that lets the user configure the board by reading and writing to its registers.

The DSK includes 4 LEDs and a 4 position DIP switch as a simple way to provide the user with interactive feedback. Both are accessed by reading and writing to the CPLD registers.

A 5V external power supply is used to power the board. On-board switching voltage regulators provide the +1.26V DSP core voltage and +3.3V I/O supplies. The board is held in reset until these supplies are within operating specifications.

Code Composer communicates with the DSK through an embedded JTAG emulator with a USB host interface. The DSK can also be used with an external emulator through the external JTAG connector.

Memory Map

The C67xx family of DSPs has a large byte addressable address space. Program code and data can be placed anywhere in the unified address space. Addresses are always 32-bits wide. The memory map shows the address space of a generic 6713 processor on the left with specific details of how each region is used on the right. By default, the internal memory sits at the beginning of the address space. Portions of the internal memory can be reconfigured in software as L2 cache rather than fixed RAM. The EMIF has 4 separate addressable regions called chip enable spaces (CE0-CE3). The SDRAM occupies CE0 while the Flash and CPLD share CE1. CE2 and CE3 are generally reserved for daughtercards.

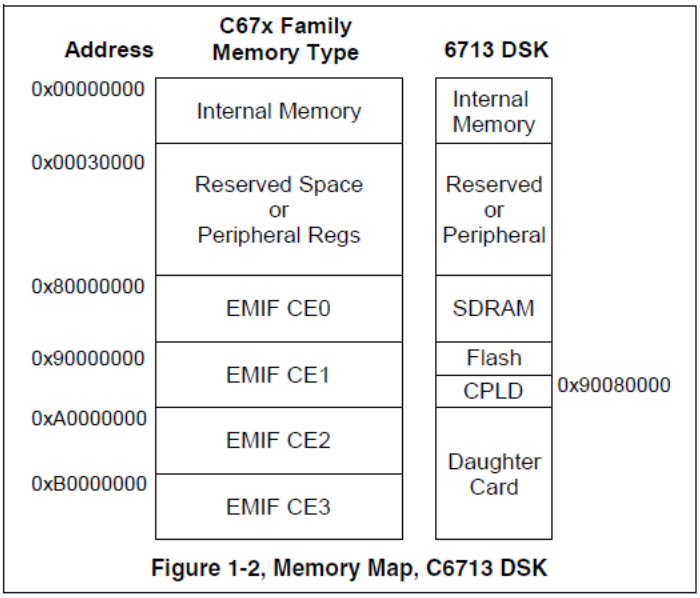


Figure 1-2, Memory Map, C6713 DSK

AIC23 Codec

The DSK uses a Texas Instruments AIC23 (part #TLV320AIC23) stereo codec for input and output of audio signals. The codec samples analog signals on the microphone or line inputs and converts them into digital data so it can be processed by the DSP. When the DSP is finished with the data it uses the codec to convert the samples back into analog signals on the line and headphone outputs so the user can hear the output.

The codec communicates using two serial channels, one to control the codec's internal configuration registers and one to send and receive digital audio samples. McBSP0 is used as the unidirectional control channel. It should be programmed to send a 16-bit control word to the AIC23 in SPI format. The top 7 bits of the control word should specify the register to be modified and the lower 9 should contain the register value. The control channel is only used when configuring the codec, it is generally idle when audio data is being transmitted, McBSP1 is used as the bi-directional data channel. All audio data flows through the data channel. Many data formats are supported based on the three variables of sample width, clock signal source and serial data format.

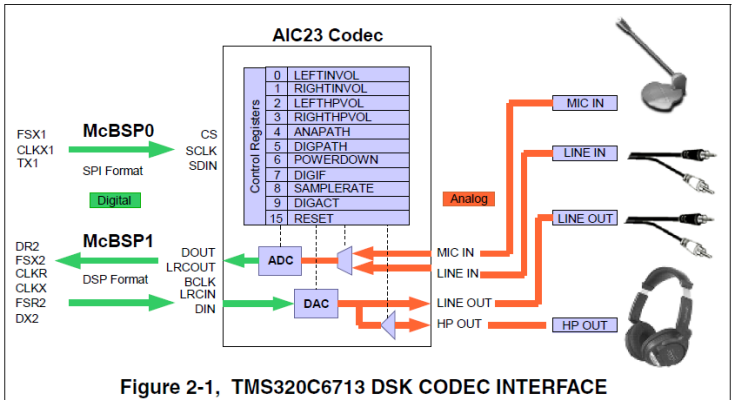


Figure 2-1, TMS320C6713 DSK CODEC INTERFACE

The codec has a 12MHz system clock. The 12MHz system clock corresponds to USB sample rate mode, named because many USB systems use a 12MHz clock and can use the same clock for both the codec and USB controller. The AIC23 can divide down the 12 MHz clock frequency to provide sampling rates of 8, 16, 24, 32, 44.1, 48 and 96 KHz.

Software

Texas Instruments' Code Composer Studio (CCS) Integrated Development Environment (IDE) incorporates a C compiler, an assembler, and a linker. It is a development tool that allows users to create, edit and build programs, load them into the processor memory and monitor program execution. CCS communicates with the DSK via a USB connection. It supports real - time debugging has graphical capabilities. CCS is based on Eclipse, which is a Linux based open source software. CCSv7 and later does not require a paid license. The latest version is v11. But it does not support the debug probe on the 6713DSK. Older versions do not run on Windows 10. We will be using version 7.

A special Board Support Library (BSL) *dsk6713bsl32.lib* is supplied with the TMS320C6713 DSK. The BSL provides C-language functions for configuring and controlling all the on-board devices. The library includes modules for general board initialization, access to the AIC23 codec, reading the DIP switches, controlling the LED's, and programming and erasing the Flash memory. TI also provides a Chip Support Library (CSL) *csl6713.lib* that contains C functions and macros for configuring and interfacing with all the 'C6713 on-chip peripherals. The pre-compiled board support and chip support libraries are provided to you in the *dsplab* folder. The C source code for BSL functions are also provided. The folder also contains the required header files for using BSL and CSL functions.

On power on, a power on self - test (POST) program, stored by default in the onboard flash memory, uses routines from the board support library (BSL) to test the DSK. It tests the internal, external, and flash memory, the two multichannel buffered serial ports (McBSP), DMA, the onboard codec, and the LEDs. If all tests are successful, all four LEDs blink three times and stop (with all LEDs on). During the testing of the codec, a 1kHz tone is generated for 1 second.

CCS Project

A very readable and useful user guide for CCS is available online at

https://software-dl.ti.com/ccs/esd/documents/users_guide/index.html

All work in CCS is based on projects, which are typically a collection of files and folders required for an application to be run on the DSK. Project folders are stored and organized in workspace folder. A workspace is the main working folder for CCS. When CCS is launched, it will prompt for the workspace folder location.

A Code Composer Studio project comprises all of the files (or links to all of the files) required in order to generate an executable file. A variety of options enabling files of different types to be added to or removed from a project are provided. In addition, a Code Composer Studio project contains information about exactly how files are to be used in order to generate an executable file. Compiler/linker options can be specified.

To create a new CCS project, follow the steps below:

Go to menu *Project → New CCS Project…* or *File → New → CCS Project*.

In the New CCS Project wizard:

Type or select the *Target device*: Select *Unclassified Devices* and *DSK6713*

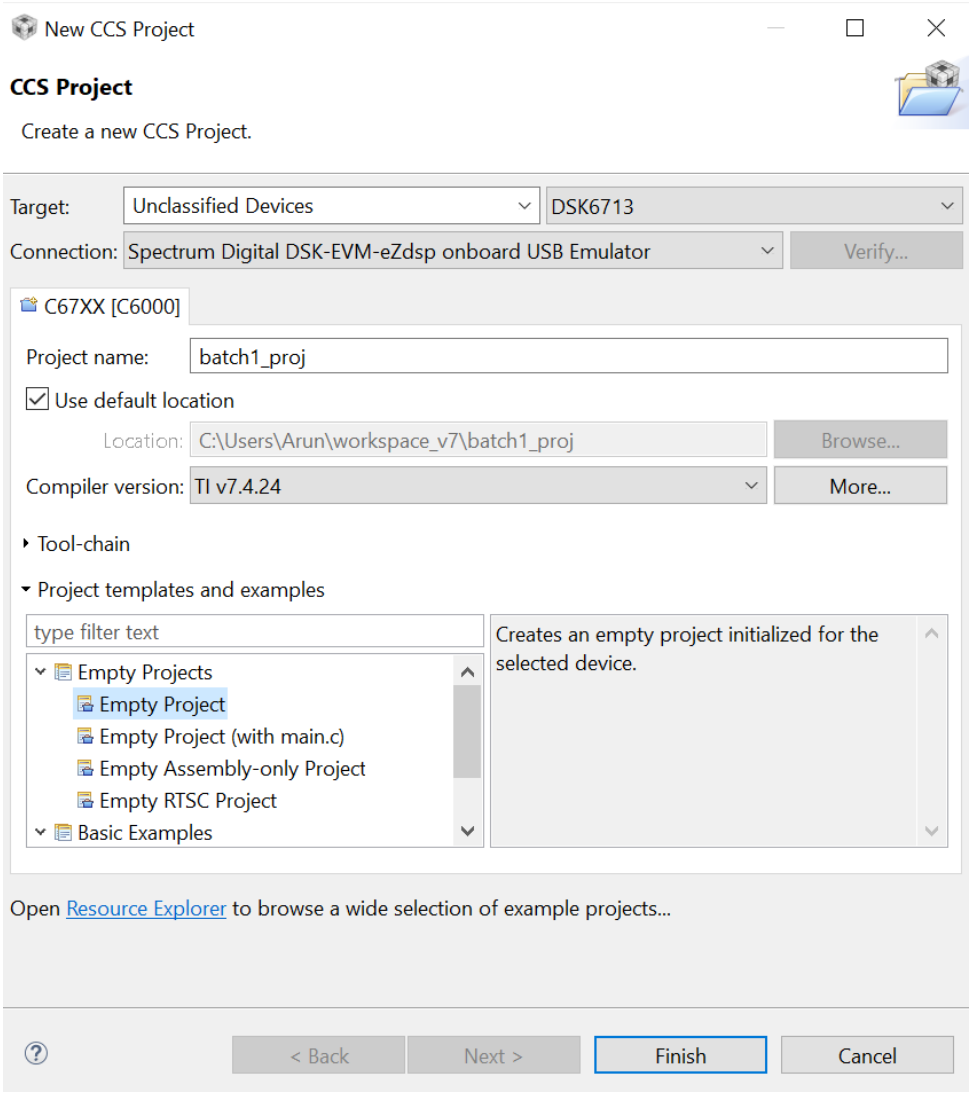
Connection: *Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator*

This automatically creates a Target Configuration File *DSK6713.ccxml*. The Target Configuration File is a plain text XML file, with a .ccxml extension, that contains all the necessary information for a debug session: the type of Debug Probe, the target board or device , and (optionally) a path to a

GEL (General Extension Language) script, which is responsible for performing device and/or hardware initialization.

Project Name: Give your project a name, such as *batch1_proj*. The default location will be a folder with the name of your project within the *workspace_v7* folder

In project templates, select Empty project

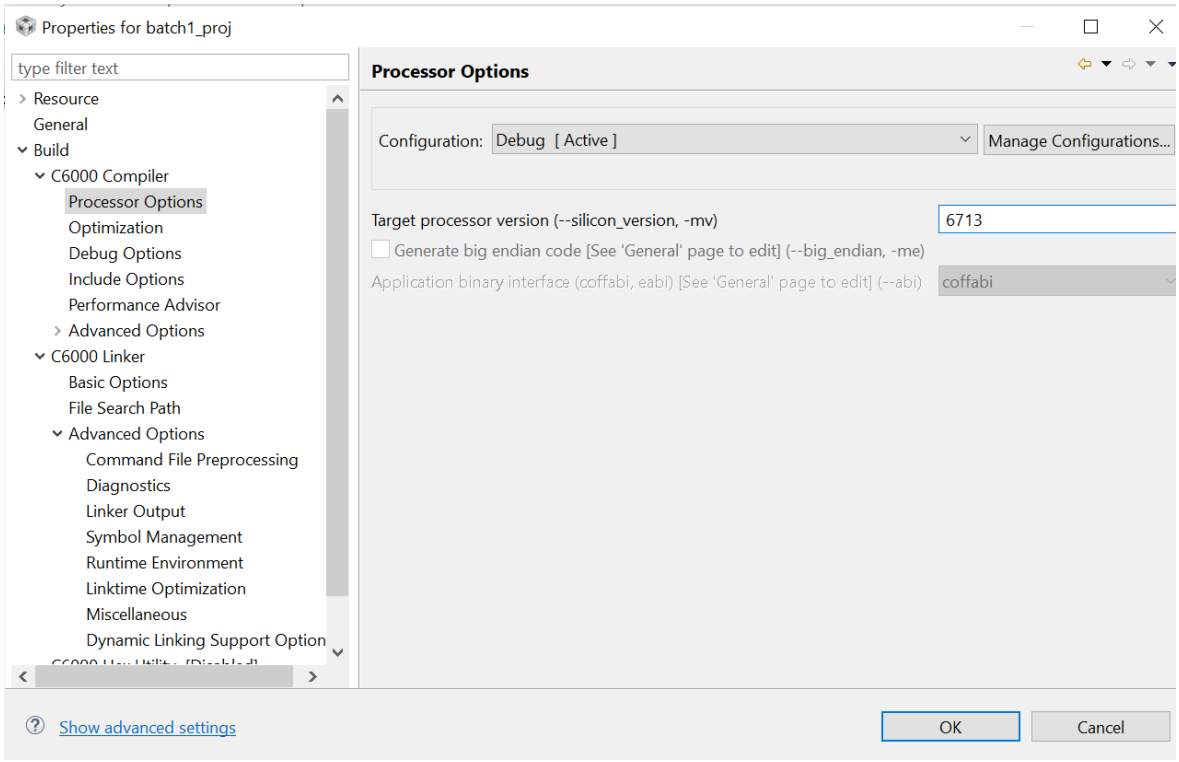


Click Finish

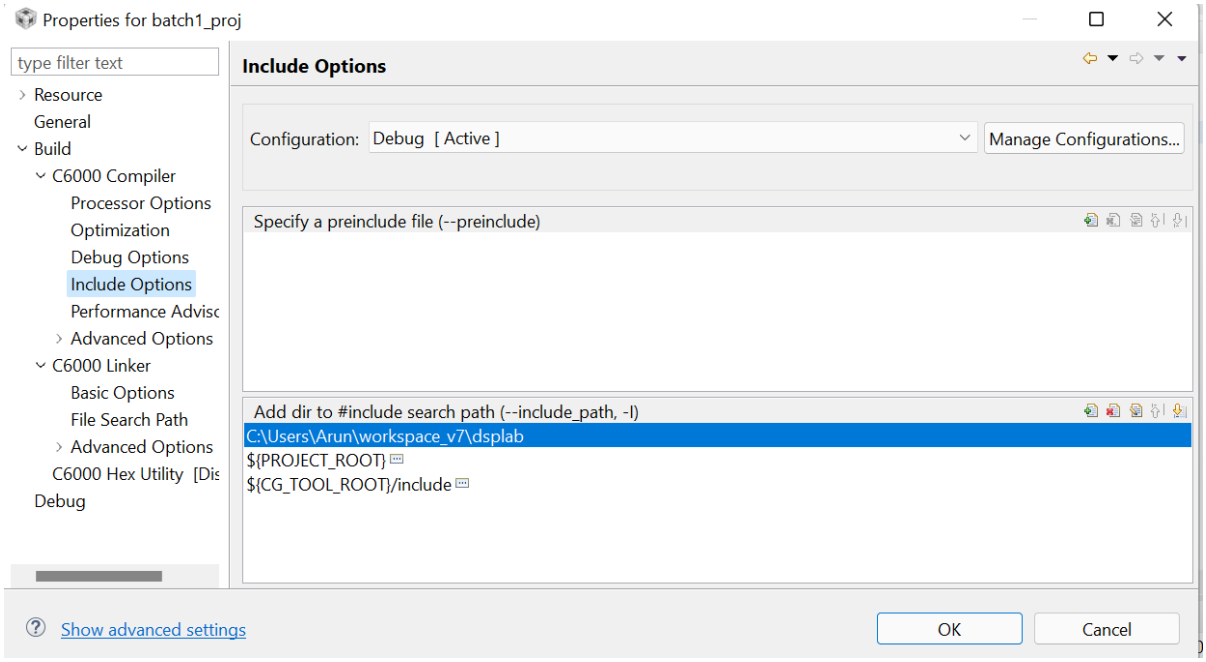
Your project should now show in the Project Explorer window.

First, we need to set some properties for our project. Right-click on the project and go to Properties (Or from menu *Project > Properties*)

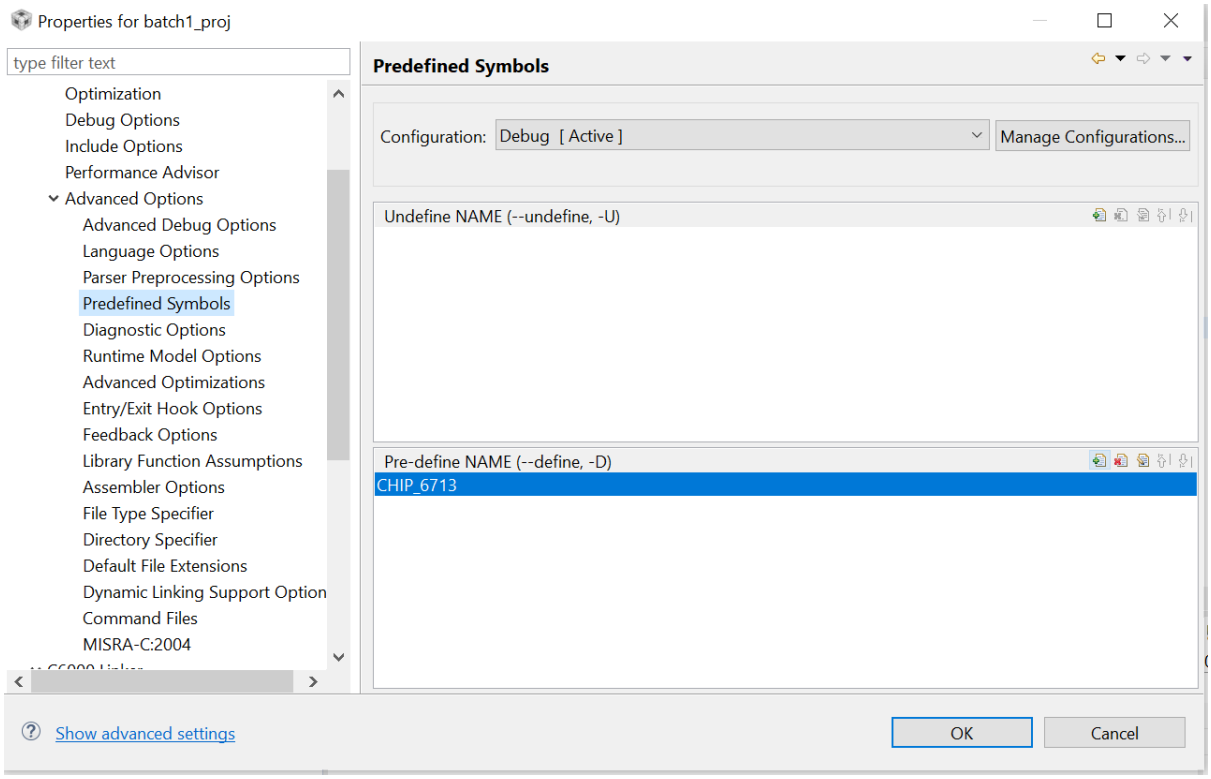
Under *Build>C6000 Compiler>Processor Options*, set *Target processor version* as 6713



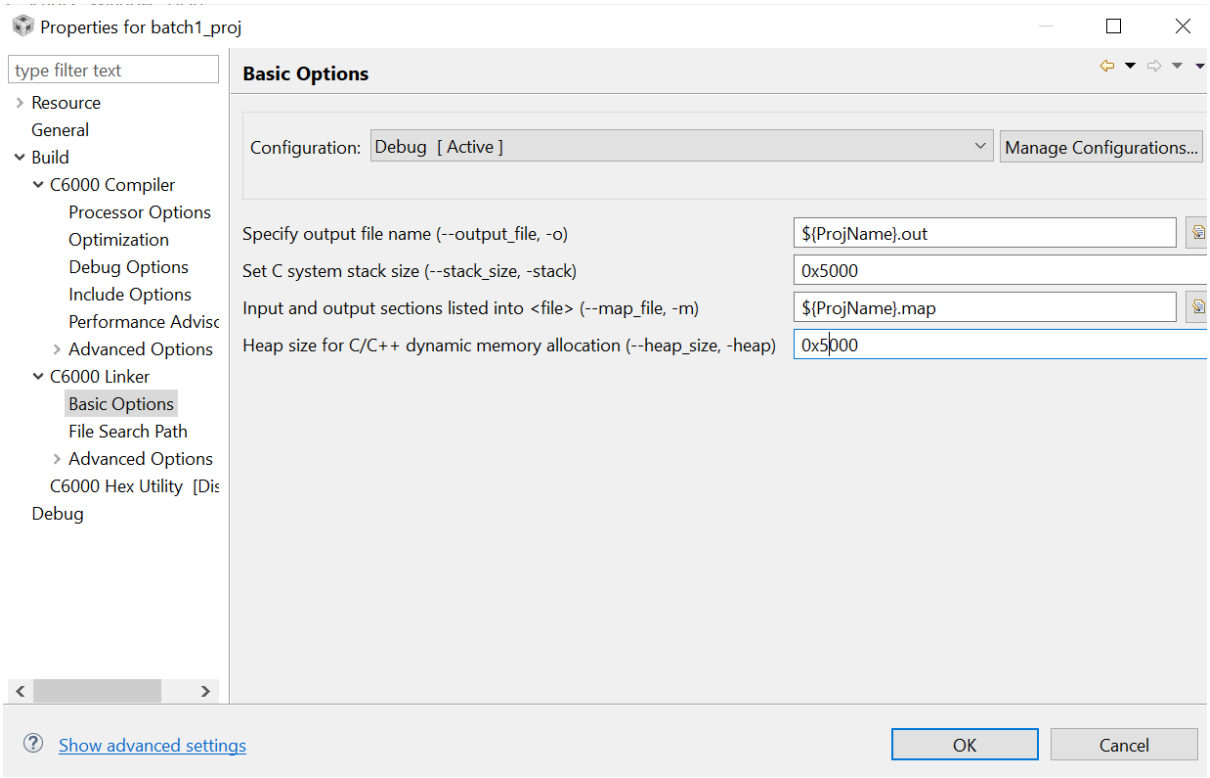
Under *Include Options*, go *to Add dir to #include search path*, click on the file icon with a green + mark and Browse to the folder *dsplab* provided to you and click OK. This folder contains the header files for the board support library and chip support library functions.



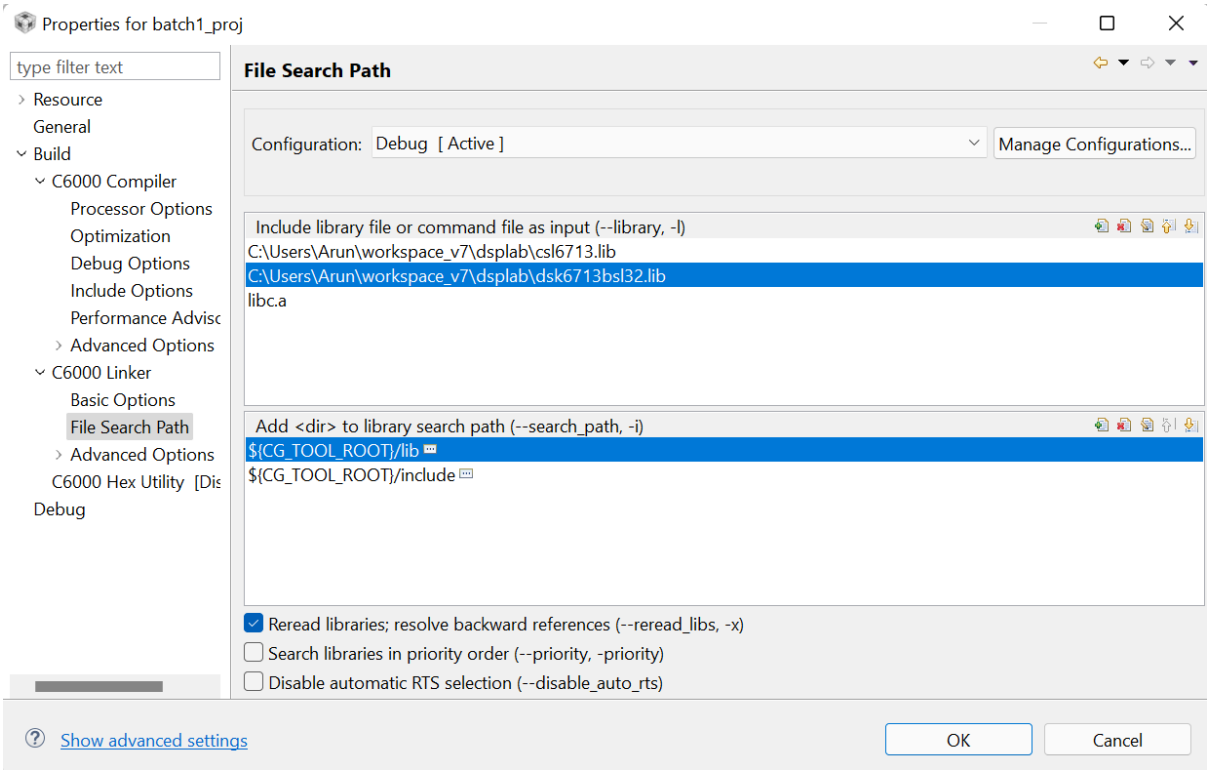
Under *Advanced Options>Predefined Symbols*, in the *Pre-define NAME* window, click on the file icon with green + , and enter CHIP_6713. This symbol is used for conditional compilation. If you don't do this step, you will have to type the line #define CHIP_6713 in your source file.



Under *C6000 Linker> Basic options*, enter a suitable value (eg: 0x5000) as the size for stack and heap



Under *C6000 Linker>File search path*, *Include library file or command file as input* window should already contain the file *libc.a* which is the standard C library. We need to add the chip support and board support libraries. Click on the file icon with green +, browse to *dsplab* folder, select the file *csl6713.lib*, click Open, then OK. Similarly add the file *dsk6713bsl32.lib*



a. LED Blink: In this first experiment, we will add source code that blinks LED #0 at a rate of about 2.5 times per second using the LED module of the DSK6713 Board Support Library. The example also reads the state of DIP switch #3 and lights LED #3 if the switch is depressed or turns it off if the switch is not depressed. The purpose of this experiment is to demonstrate basic Board Support Library usage as well as provide a project base for your own code. The BSL is divided into several modules, each of which has its own include file. The file *dsk6713.h* must be included in every program that uses the BSL. This example also includes *dsk6713_led.h* and *dsk6713_dip.h* because it uses the LED and DIP modules. To add a source file, right click on the project name and New>File and give file name as *led.c*. Click Finish. (You can also add from menu *File> New*). Type the following code in the *led.c* file:

```
/*  ======== led.c ========*/
#include <dsk6713.h>
#include <dsk6713_led.h>
#include <dsk6713_dip.h>
void main()
{
     /* Initialize the board support library, must be first BSL call */
     DSK6713_init();

      /* Initialize the LED and DIP switch modules of the BSL */
     DSK6713_LED_init();
     DSK6713_DIP_init();

     while(1)
     {
```

```
                /* Toggle LED #0 */
                DSK6713_LED_toggle(0);

            /* Check DIP switch #3 and light LED #3 accordingly, */
            //0 = switch pressed
                if (DSK6713_DIP_get(3) == 0)
                /* Switch pressed, turn LED #3 on */
                    DSK6713_LED_on(3);
                else
                /* Switch not pressed, turn LED #3 off */
                    DSK6713_LED_off(3);

                /* Spin in a software delay loop for about 200ms */
                DSK6713_waitusec(200000);
            }
    }
```

Save and Build the program (by clicking the hammer icon). Connect the DSK to the PC and start Debug (by clicking the bug icon). CCS will automatically perform a series of steps. It will switch to Debug perspective, connect to the debug probe, Load the project's executable file (.out) to the device memory, and will run to the function main(). Click Resume to continue execution. Verify that the program works as described. A number of debugging features are available in CCS, including setting breakpoints and watching variables, viewing memory, registers, and mixing C and assembly code, graphing results, and monitoring execution time. One can step through a program in different ways (step into, or over, or out). Since we will be using CCS for the rest of this lab course, you are strongly advised to read the CCS user manual online at https://software-dl.ti.com/ccs/esd/documents/users_guide/index.html and familiarize with its operation and features.

b. Ring counter: In this experiment, you will write a program to set up a ring counter using the four LEDs on the DSK. The counting speed should increase when dip switch 0 is pressed. You do not need to create a new project for this experiment. In the project explorer, right click on the file *led.c* and click Exclude from build. Now add a new source file to the project, name it *led_dip.c* and write your code there. This way, by swapping in and out files, you will be using the same project for all your programs.

c. Tone Generation: The code below will output a 1KHz tone by sending samples to the AIC23 codec onboard the DSK. The DAC converts the samples to an analog signal and outputs on the line-out and headphones out of the DSK. Modify your project by excluding the earlier file led_dip.c and adding the code below in a new file *tone.c*.

```
#include <math.h>
#include <dsk6713.h>
#include <dsk6713_aic23.h>
int main()
{
    float Fs = 8000.;
    float F0 = 1000.;
    float pi = 3.141592653589;
    float theta = 0.;
    float delta = 2. * pi * F0 / Fs; // increment for theta
```

```
    float sample;
    unsigned out_sample;
    /* Initialize the board support library, must be called first */
    DSK6713_init();

    DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
    DSK6713_AIC23_CodecHandle hCodec;
    /* Start the codec */
    hCodec = DSK6713_AIC23_openCodec(0, &config);

    /* Change the sampling rate to 16 kHz */
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_8KHZ);

    for (;;)
    { /* Infinite loop */
        sample = 15000.0 * sin(theta); /* Scale for DAC */
        out_sample = (int)sample & 0x0000ffff; // Put in lower half (R)

        /* Poll XRDY bit until true, then write to DXR */
        while (!DSK6713_AIC23_write(hCodec, out_sample))
            ;
        theta += delta;
        if (theta > 2 * pi)
            theta -= 2 * pi;

    }
}
```

In the code, right click on DSK6713_AIC23_DEFAULTCONFIG and click *Open Declaration*. This will open the header file which contains detailed information about how the codec is configured. (eg: how to decrease headphone volume?). The codec is started by calling the BSL function `DSK6713_AIC23_openCodec()`.

Each iteration of the infinite loop generates a sample and writes it to the codec. Note that the float value is converted to int and the upper 16 bits are set to 0 before outputting. This is because the BSL function that configures the codec sets McBSP1 to send and receive 32-bit words, with the left sample in the upper 16 bits and right sample in the lower 16 bits. The 16-bit samples are in signed 2s complement form. Since the upper 16 bits of out_sample are set to 0, the tone will be heard on the right channel only. If we want the output to be on the left channel, we can use the statement `out_sample = (int)sample << 16;` instead, which puts the 16-bit value in the top half, and sets the lower 16 bits to 0s. We can also pack two 16-bit samples in out_sample for output on both L and R channels.

The function `DSK6713_AIC23_write()` is used to write a pair of samples to the DAC. The function uses polling to write samples and returns 0 if codec is not ready and returns 1 if write is successful. The while loop continues till write is successful. Build and Debug the program. Connect your headphones to the headphone and listen to the tone(beware of volume). You should hear the tone on R channel only. Using bitwise operators in C, try to output the tone on both channels. Change frequency F0 to 500 Hz and listen. Next, change F0 to 7500 Hz and listen. Explain what you observe.

d. DIP controlled tone: Write a program that outputs a tone only if DIP switch #0 is pressed and held down. When pressed down, LED #0 should also light up.

e. Table lookup tone generation: An alternative approach to generate a sine wave is to store the samples for one period in an array and to lookup the array for each sample. Although lookup table approach is not very flexible to generate different frequencies, it has the advantage of less computational effort since the sine values are computed only once. Generate a 1KHz tone assuming a sampling frequency of 8KHz using the table lookup method.

f. Audio Loopback: Code below reads pairs of left and right channel samples from the codec ADC and loops them back out to the codec DAC. The BSL function `DSK6713_AIC23_read()` is used to read a pair of samples from ADC and the function `DSK6713_AIC23_write()` is used to write a pair of samples to the DAC. Both functions use polling to read/write samples and returns 0 if codec is not ready and returns 1 if read/write is successful. The while loop continues till read/write is successful. Note that the function DSK6713_AIC23_read() uses a pointer variable. Build and Debug the program. Connect a 3.5mm aux cable from the headphone out of your PC (or phone). Connect a pair of headphones to the headphone out of the DSK. Verify that loopback is working.

```
// loopback.c
#include <dsk6713.h>
#include <dsk6713_aic23.h>

void main(void)
{
    DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
    DSK6713_AIC23_CodecHandle hCodec;
    Uint32 sample_pair = 0;
    DSK6713_init(); /* In the BSL library */
    /* Start the codec */
    hCodec = DSK6713_AIC23_openCodec(0, &config);

    /* Change the sampling rate to 16 kHz */
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_16KHZ);

    /* Read left and right channel samples from the ADC and loop */
    /* them back out to the DAC.                                 */
    for (;;)

        while (!DSK6713_AIC23_read(hCodec, &sample_pair))
            ;
        while (!DSK6713_AIC23_write(hCodec, sample_pair))
            ;
    }
}
```

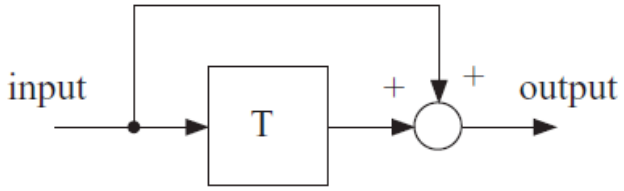
g. Quantization: This experiment studies quantization effects. The DSK's codec is a 16-bit ADC/DAC with each sample represented by a two's complement integer. The range of representable integers is: $-32768 \leq x \leq 32767$. For high-fidelity audio at least 16 bits are required to match the dynamic range of human hearing; for speech, 8 bits are sufficient. If the audio or speech samples are quantized to less than 8 bits, quantization noise will

become audible. The 16-bit samples can be requantized to fewer bits by a right/left bit-shifting operation. For example, right shifting by 3 bits will knock out the last 3 bits, then left shifting by 3 bits will result in a 16-bit number whose last three bits are zero, that is, a 13-bit integer. Quantization can also be introduced by bitwise AND. For example

```
sample_pair = sample_pair & 0xfffcfffc;
```

will set the last two bits of L and R channels to zero. Insert this statement between the codec read and write operations in the loopback example and listen. Modify the statement so that the output samples are quantized to 12 bits, 10 bits, 8 bits…. You should hear an increase in the quantization noise as more number of bits are set to zero.

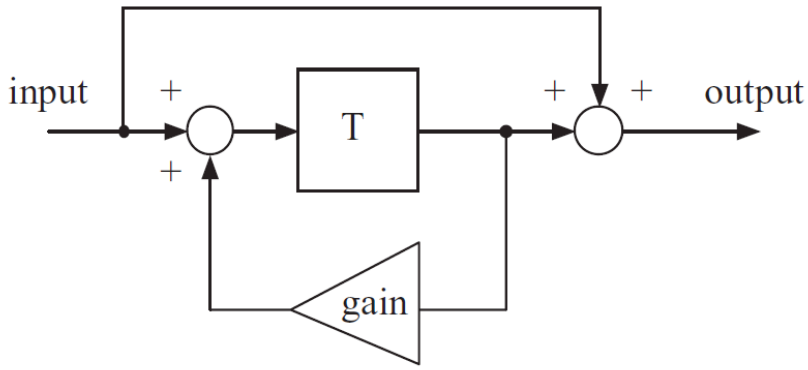
h. Delay: In the loopback experiment, we simply connected the input to the output. Typically, we will do some processing on the input sample and then output the processed sample. Some simple, yet striking, effects can be achieved simply by delaying the samples as they pass from input to output. Program delay.c, listed below, demonstrates this. A delay line is implemented using the array buffer to store samples as they are read from the codec. Once the array is full, the program overwrites the oldest stored input sample with the current, or newest, input sample. Just prior to overwriting the oldest stored input sample in buffer , that sample is retrieved, added to the current input sample, and output to the codec. Figure below shows a block diagram representation of the operation of program delay.c in which the block labeled T represents a delay of T seconds. Note that the sampling rate is set to 8 kHz, therefore, the delay of 8000 samples corresponds to a delay of 1 sec. Build and run the project, using line-in and headphones to verify its operation.



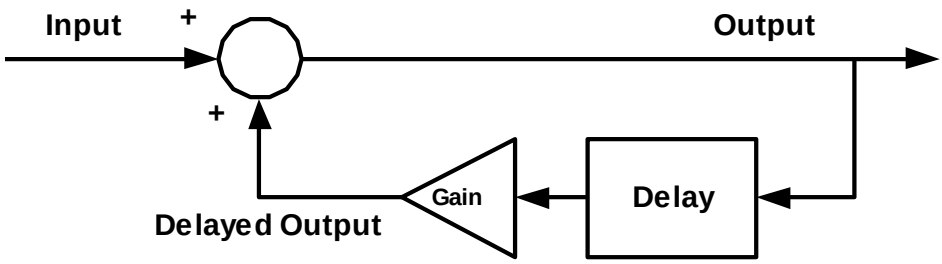

```
#include <dsk6713.h>
#include <dsk6713_aic23.h>
#define BUF_SIZE 8000
void main()
{

	Uint32 sample_pair; // both channels packed in 32-bits
	short i = 0, left, buffer[BUF_SIZE] = { 0 }, delayed, output;
	DSK6713_init();
	DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
	DSK6713_AIC23_CodecHandle hCodec;
	hCodec = DSK6713_AIC23_openCodec(0, &config);
	DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_8KHZ);
	while (1)
	{
		while (!DSK6713_AIC23_read(hCodec, &sample_pair))
			;
		//extract left sample and put in 16-bits
```

```
                left = (int)sample_pair >> 16; //
                delayed = buffer[i]; //read oldest sample
                output = left + delayed; //output sum of new and delayed
                buffer[i] = left; //replace oldest sample with input

                //increment i to point to the oldest sample
                if (++i >= BUF_SIZE)
                      i = 0;

                //put 16-bit sample in top-half
                sample_pair = (int)output << 16;

                while (!DSK6713_AIC23_write(hCodec, sample_pair))
                      ;

          }
}
```

i. Echo: Modify the delay program to create an echo. By feeding back a fraction of the output of the delay line to its input, a fading echo effect can be realized. A block diagram representation of the required program is given below. Note that the output is input + delayed as in the previous experiment, but the input to the delay line is now input + gain*delayed. Modify the previous program to realize an echo effect. Experiment with different values for gain (between 0.0 and 1.0) and also with different delays.



j. Reverberation: Whereas an echo is a single reflection of a soundwave off a distance surface, reverberation is the reflection of sound waves created by the superposition of such echoes caused by multiple reflections. A simplified model for reverb is given below. Realize the model in code an generate a reverberation effect. Set delay buffer length to 2500 and gain=0.5. Play some speech on your PC/phone and listen to the effect. Pause the play in between to observe the effect better. Change gain to 0.25 and 0.75 and note the effect of the change. For more realistic reverb effects, see [8]



4. Linear Convolution:

a. A program to compute the linear convolution of two finite duration sequences x[n] and h[n] is given below. It is assumed that x[n] extends from n=0 to L-1 and h[n] extends from n=0 to M-1. Length of y[n] is then L+M-1. The function **conv()** implements the convolution operation given by $y[n] = \sum_{k=0}^{M-1} h[k]x[n-k]$. The sequence x[n] is reflected and shifted. For each n, the range over which both sequences overlap needs to be determined. In the function, conditional operators are used to determine kmin and kmax.

```
#include <stdio.h>
void conv(float *x, float *h, float *y, short l, short m);
void main()
{
    float x[] = { 1, 2, 3, 4 };
    float h[] = { 1, 1, 1 };
    float y[6]; //length of y = L+M-1
    short n;
    conv(x, h, y, 4, 3);
    for (n = 0; n < 6; n++)
        printf("%.2f ", y[n]);
}

void conv(float *x, float *h, float *y, short l, short m)
{
    short k, kmin, kmax, n;
```

```
        for (n = 0; n < l + m - 1; n++)
        {
            y[n] = 0;
            kmin = (n > l - 1) ? n - l + 1 : 0;
            kmax = (n > m - 1) ? m - 1 : n;
            //printf("%d %d\n", kmin, kmax);
            for (k = kmin; k <= kmax; k++)
            {
                y[n] = y[n] + h[k] * x[n - k];

            }

        }

    }
```

Build and Debug. During a debug session, we can visualize the signals using the Graph tool in CCS. Step Over the program line by line and halt when y has been computed. Click on *Tools>Graph>Single-Time*. Set the Graph Properties as follows:

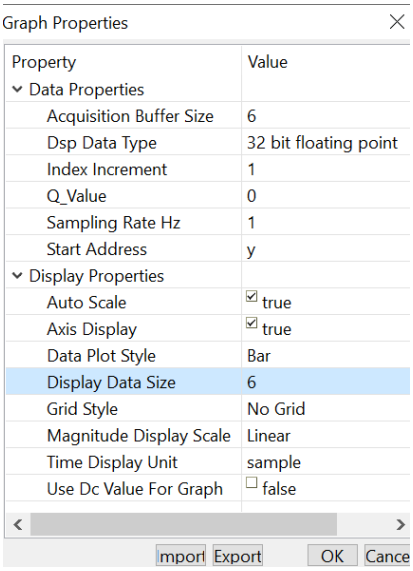
Acquisition buffer size: 6 (since y[n] has 6 elements)

DSP Data type: 32-bit floating point (since we are using float type variable)

Start Address: y (array name by itself is treated as the address of its first element)

Display Data Size: 6

Click Ok. The graph of y[n] should appear.

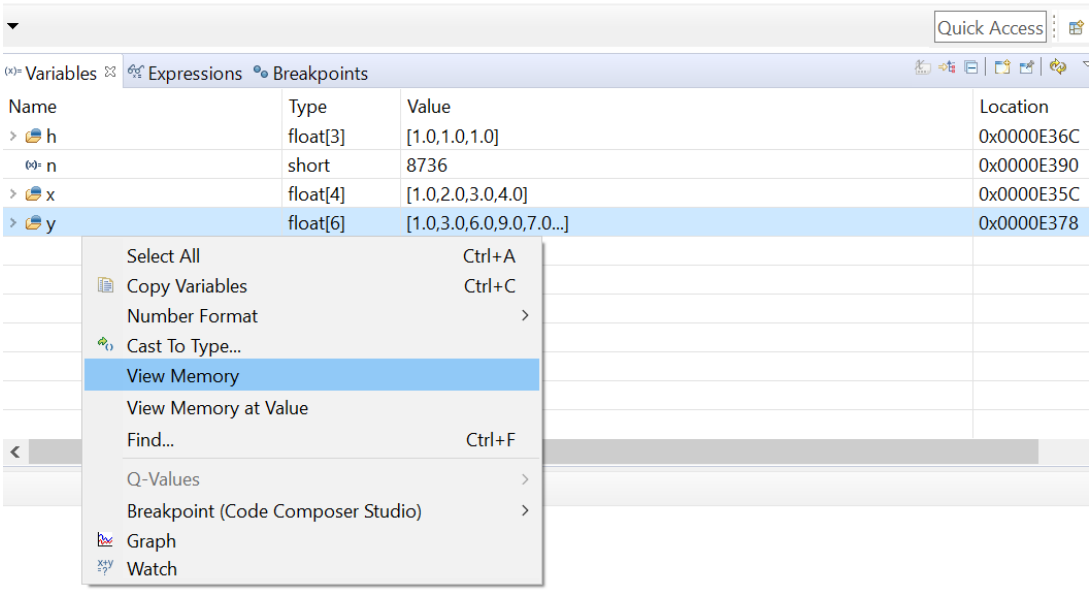


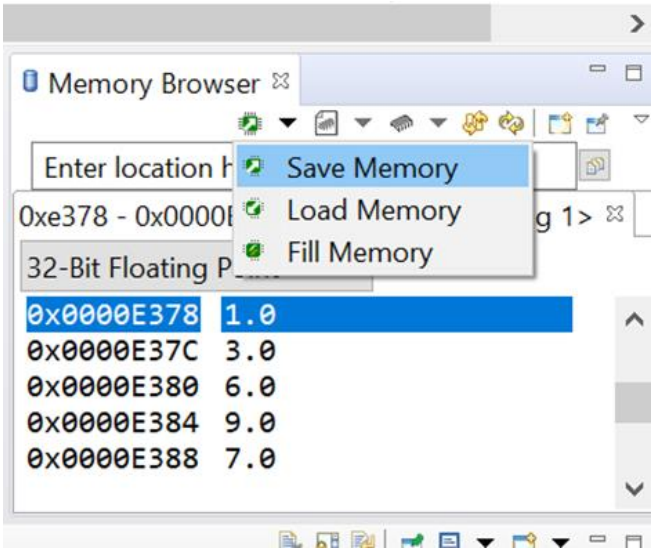
Instead of the above steps, you can go to the *Variables* view (from the cluster of tabbed views named Variables, Expressions, and Registers), right click on y and click Graph. To emphasize the discrete nature of y[n], click on *Show The Graph Properties* button in the graph window toolbar and change the *Data Plot Style* to Bar. Also, right click anywhere on the graph, on the context menu that opens, select *Display Properties*, click *Axes* and change *Y-axis Display format* to *Decimal*

b. Memory Save/ Load: In the previous experiment, the elements of the arrays x and h were hard-coded in the program itself. We might need to process input data saved in files (for example, when simulating real-time input with predigitized data captured at another time/place). CCS can read data from a file on the host computer and put the data in target processor memory. CCS can also write the processed data samples from the target processor to the host computer as an output file for analysis.

In this experiment, we will first save the output y[n] from the previous part in a file and then use that file as the input x[n] and perform convolution again. In effect, we are simulating the cascade of two identical systems with impulse response h[n].

First, start Debugging the program in part a once again and Step over line by line. Once the conv() function returns, go to Variables view, right click on variable y and click *View Memory*.





In the *Memory Browser* window that opens, you can see the memory addresses and data in y. In the memory browser toolbar, click on *Save memory*.

Browse to your workspace folder and give some file name. Click *Save*.

Leave *File type* as *TI data*.

Click *Next*, select *Format* as *32-bit floating point*.

Find out the starting address for y in the Memory Browser and type the correct start address in the *Start Address* field.

Specify the number of memory words as 6 (since y has 6 floats, and a float as well as the word-size on the 6713 is 32 bits, 6 words mean 6 floats).

Click Finish.

Your workspace folder should now have a file with .dat extension, containing the data in y. Terminate the Debug session.

Next, we are going to use the saved file as the input x[n] for convolution. In the program, change the size of x array to 6, delete the initializer list, change size of y array to 8(=6+3-1).

Also make the necessary change in the length of x in call to the function conv(). Rebuild the program with the changes and start stepping over.

Once memory for x is allocated (when the line float x[6]; is executed), right click on x from the Variables view, and select *View Memory*.

From the memory browser toolbar, click *Load Memory*. Browse to the previously saved file, give file type, start address and length and click Finish.

The array x should now be filled with data from the previously saved file. Resume debugging and observe the new output.

c. C Library File IO: CCS also supports standard C library file I/O functions such as fopen( ), fclose( ), fread( ), fwrite( ), and so on. These functions not only provide the ability of operating on different file formats, but also allow users to directly use the functions on computers. Comparing with the memory load/save method introduced in the previous part, these file I/O functions are portable to other development environments. An example of C program that uses fopen( ), fclose( ), fread( ), and fwrite( ) functions is included below. Verify the working of the program in CCS.

```
#include <stdio.h>
void writedatafile(void);
void conv(float *x, float *h, float *y, short l, short m);
void main()
{
    writedatafile();//put some data in a file
    FILE *fp;
    fp = fopen("../../myfilebin", "rb");//open file for binary read
    float x[5];
    //read 5 floats from file and store in x
    fread(x, sizeof(float), 5, fp);
    float h[3]={1, 1, 1};
    float y[7];
    conv(x, h, y, 5, 3);

}

void writedatafile()
{
    float a[] = { 1, 2, 3, 4, 5}; //some data
    FILE *fp;
    fp = fopen("../../myfilebin", "wb");//open for binary write

    //write 5 floats in array a to myfilebin
    fwrite(a, sizeof(float), 5, fp);
    fclose(fp);

}

void conv(float *x, float *h, float *y, short l, short m)
{
    short k, kmin, kmax, n;
    for (n = 0; n < l + m - 1; n++)
    {
        y[n] = 0;
        kmin = (n > l - 1) ? n - l + 1 : 0;
        kmax = (n > m - 1) ? m - 1 : n;
        // printf("%d %d\n", kmin, kmax);
        for (k = kmin; k <= kmax; k++)
        {
            y[n] = y[n] + h[k] * x[n - k];
        }

    }
```

```
}
```

5. FFT:

The DFT X[k] of a finite length sequence x[n] defined for n=0… N-1 can be obtained by sampling its DTFT $X(e^{j\omega})$ on the ω axis between $0 \le \omega < 2\pi$ at $\omega_k = \frac{2\pi k}{N}$, $k = 0 \ldots N-1$. Ie. $DFT\{x[n]\} = X[k] = X(e^{j\omega})\Big|\omega = \frac{2\pi k}{N} = \sum_{n=0}^{N-1} x[n]\, e^{-\frac{j2\pi kn}{N}}$. Using the commonly used notation $W_N = e^{\frac{-j2\pi}{N}}$, $X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{kn}$, $k = 0 \ldots N-1$

Using Euler's relation $e^{-j\theta} = cos\theta - jsin\theta$, the real and imaginary parts of X[k] are:

$$ReX(k) = \sum_{k=0}^{N-1} \big(Rex(n)cos(2\pi kn/N) + Imx(n)sin(2\pi kn/N)\big)$$

$$ImX(k) = \sum_{k=0}^{N-1} \big(Imx(n)cos(2\pi kn/N) - Rex(n)sin(2\pi kn/N)\big)$$

The function dft() in the program below implements the above two equations to compute DFT. A structure COMPLEX array is used to store the real and imaginary parts of x[n] and X[k]. The computations are performed in-place with the input array over-written by the output array. The program computes the 64-point DFT on the 64 samples of a 1KHz signal sampled at 8000Hz.

```
#include <math.h>
#define PI 3.1415926535897
#define M 64 //signal length
#define N 64 //DFT length
typedef struct
{
    float real;
    float imag;
} COMPLEX;
void dft(COMPLEX *);

void main()
{
    int n;
    float F = 1000.0, Fs = 8000.0;
    COMPLEX samples[N]={0.0};

     //Generate time-domain signal
    for (n = 0; n < M; n++) //M samples of x[n]
    {
        samples[n].real = cos(2 * PI * F * n / Fs);
        samples[n].imag = 0.0;
    }
```

```
    dft(samples); //call DFT function

}

void dft(COMPLEX *x)
{
    COMPLEX result[N];
    int k, n;
    for (k = 0; k < N; k++) // N point DFT
    {
        result[k].real = 0.0;
        result[k].imag = 0.0;
        for (n = 0; n < N; n++)
        {
            result[k].real += x[n].real * cos(2 * PI * k * n / N) +
                              x[n].imag * sin(2 * PI * k * n / N);
            result[k].imag += x[n].imag * cos(2 * PI * k * n / N) -
                              x[n].real * sin(2 * PI * k * n / N);
        }
    }
    for (k = 0; k < N; k++)
    {
        x[k] = result[k];
    }
}
```

Procedure: Insert a breakpoint in the code on the line calling the dft() function (To add a breakpoint, double click on the line number OR right-click anywhere on the line and click Breakpoint(CCS) >Breakpoint. You should see a small blue mark on the line where the breakpoint is inserted).

Build the project and Debug.

The program will first halt at entry to main().Click Resume and program will halt at the breakpoint. The array `samples` now contain the time-domain signal. We can visualize the signal using the Graph tool in CCS. Click on Tools>Graph>Single-Time. Set the Graph Properties as follows:

Acquisition buffer size: 64 (since we have stored 64 samples of x[n]
DSP Data type: 32 bit floating point
Index increment: 2 (The nature of the structure array samples is such that it comprises 2N float values ordered so that the first value is the real part of x[0], the second is the imaginary part of x[0], the third is the real part of x[1], and so on. Since x[n] is purely real, we take alternate values only)
Start Address: samples
Data plot style: Bar

Display Data Size: 64
Click Ok. The graph of x[n] should appear.

Click *Step Over* and the program and will now halt after the function dft() returns. At this point, the array `samples` contain the 64 DFT coefficients X[k]. The graph now displays the real part of the X[k]. If needed, click on Reset the Graph button and then Refresh button in the graph window. You should see two distinct peaks at k= 8 and k=56.

The spike at k=8 with amplitude =32 corresponds to the frequency $= \frac{k}{NT} = \frac{kF_S}{N} = \frac{8x8000}{64} = 1KHz$. The spike at k=56 is a consequence of the fact that DFT of a real x[n] is conjugate symmetric; $\mathrm{X[k]} = X^*[N-k]$. So $\mathrm{X}[56] = X^*[64-56] = X^*[8] = X[8]$.

Other than these two real components, all other DFT coefficients are zero for this example.

To get a better understanding of the spectrum of x[n], we can compute a DFT with N > 64, say N=256. Change the line `#define N 64` to `#define N 256`. Rebuild and Debug. At the first breakpoint, open graph properties and change Acquisition Buffer Size and Display Data Size to 256 (other properties as before). Since we haven't changed M, we should see zeros after the first 64 samples (zero-padding). At the second break point, you should see the real part of the 256-point DFT of x[n].

In order to display the imaginary (rather than the real) parts of the sequence X[k], the Start Address must be set to the address of the second value of type float in the array samples. That address can be found by moving the cursor over an occurrence of the identifier samples in the source file. (The address can also be found from the variables window.) Its hexadecimal address will appear in a pop - up box. Entering this value in the Start Address field of the Graph Property Dialog window in place of the identifier samples will result in the same Graphical Display. Adding four (the number of bytes used to store one 32 - bit floating point value) to the Start Address value will result in the imaginary parts of the sequence of complex values being displayed.

FFT Magnitude Graph: The magnitude and phase of the DFT can be computed from the real and imaginary parts with some effort, but more easily, the graph tool in CCS can be used to compute and display the FFT magnitude and phase of a time-domain data array. Restart the program and at the first breakpoint when the array samples contain the time-domain samples, click on Tools>Graph>FFT magnitude, set Acquisition Buffer Size to 256, DSP data type to 32-bit floating point, Index increment to 2 (since imaginary part of x[n] is 0), Sampling Rate to 8000, Signal Type to Real, Start Address to samples, Data Plot Style to Bar, FFT order to 8 (so that FFT Frame Size is 256). In the FFT magnitude graph that is displayed, there should be a peak at 1000 Hz. The graph shows frequencies upto Fs/2 only, since the information in the other half is redundant.

a. Modify the program above to sample the signal $x(t) = cos(2\pi 1000t) + 0.75cos(2\pi 500t)$ at 8 KHz and save 64 samples. Compute its DFT and display the real and imaginary parts. Using the FFT magnitude graph tool in CCS, display the magnitude of its 64-point DFT (Remember, the FFT Magnitude graph tool in CCS

computes and displays the DFT magnitude of a time-domain input). Explain the location of the peaks. Also display its 128-point DFT magnitude and explain.

b. Direct computation of a single DFT point using the dft() function requires N − 1 additions and N multiplications. Thus, direct computation of all N points requires N(N − 1) complex additions and N^2 complex multiplications. The computational complexity can be reduced to the order of N $\log_2$ N by algorithms known as fast Fourier transforms (FFT's). One FFT algorithm is called the decimation-in-time algorithm. A C function fft() for computing a complex, radix-2, decimation-in-time FFT is included below (adapted from [4]). Replace the dft() function in part a with the fft() function given below and repeat the procedure in part a.

```
void fft(COMPLEX *X)
/* X is an array of N = 2**M complex points. */
{
    COMPLEX temp1; /* temporary storage complex variable */
    COMPLEX W; /* e**(-j 2 pi/ N) */
    COMPLEX U; /* Twiddle factor W**k */
    int i, j, k; /* loop indexes */
    int id; /* Index of lower point in butterfly */
    int Nt, num_stages = 0; /* Number of stages */
    int N2 = N / 2;
    int L; /* FFT stage */
    int LE; /* Number of points in sub DFT at stage L, */
    /* and offset to next DFT in stage */
    int LE1; /* Number of butterflies in one DFT at*/
    /* stage L. Also is offset to lower */
    /* point in butterfly at stage L */
    float pi = 3.1415926535897;
    /*================================================================*/
    /* Rearrange input array in bit-reversed order */
    /* */
    /* The index j is the bit reversed value of i. Since 0 -> 0 */
    /* and N-1 -> N-1 under bit-reversal, these two reversals are */
    /* skipped. */
    j = 0;
    for (i = 1; i < (N - 1); i++)
    {
        /*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
        /* Increment bit-reversed counter for j by adding 1 to msb and */
        /* propagating carries from left to right. */
        k = N2; /* k is 1 in msb, 0 elsewhere */
        /*--------------------------------------------------------------*/
        /* Propagate carry from left to right */
        while (k <= j) /* Propagate carry if bit is 1 */
        {
            j = j - k; /* Bit tested is 1, so clear it. */
            k = k / 2; /* Set up 1 for next bit to right. */
        }
        j = j + k; /* Change 1st 0 from left to 1 */
        /*--------------------------------------------------------------*/
        /*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
        /* Swap samples at locations i and j if not previously swapped.*/
        if (i < j) /* Test if previously swapped. */
        {
            temp1.real = (X[j]).real;
            temp1.imag = (X[j]).imag;
            (X[j]).real = (X[i]).real;
            (X[j]).imag = (X[i]).imag;
            (X[i]).real = temp1.real;
            (X[i]).imag = temp1.imag;
        }
        /*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
    }
    /*================================================================*/
    /* Do M stages of butterflies */
```

```
    Nt = N;
    while (Nt >>= 1)
        ++num_stages;    //num_stages=log2(Nt)
    for (L = 1; L <= num_stages; L++)
    {
        LE = 1 << L; /* LE = 2**L = points in sub DFT */
        LE1 = LE / 2; /* Number of butterflies in sub-DFT */
        U.real = 1.0;
        U.imag = 0.0; /* U = 1 + j 0 */
        W.real = cos(pi / LE1);
        W.imag = -sin(pi / LE1); /* W = e**(-j 2 pi/LE) */
        /*---------------------------------------------------------------*/
        /* Do butterflies for L-th stage */
        for (j = 0; j < LE1; j++) /* Do the LE1 butterflies per sub DFT*/
        {
            /*.............................................................*/
            /* Compute butterflies that use same W**k */
            for (i = j; i < N; i += LE)
            {
                id = i + LE1; /* Index of lower point in butterfly */
                temp1.real = (X[id]).real * U.real - (X[id]).imag * U.imag;
                temp1.imag = (X[id]).imag * U.real + (X[id]).real * U.imag;
                (X[id]).real = (X[i]).real - temp1.real;
                (X[id]).imag = (X[i]).imag - temp1.imag;
                (X[i]).real = (X[i]).real + temp1.real;
                (X[i]).imag = (X[i]).imag + temp1.imag;
            }
            /*.............................................................*/
            /* Recursively compute W**k as W*W**(k-1) = W*U */
            temp1.real = U.real * W.real - U.imag * W.imag;
            U.imag = U.real * W.imag + U.imag * W.real;
            U.real = temp1.real;
            /*.............................................................*/
        }
        /*---------------------------------------------------------------*/
    }
    return;
}
```

c. Real-time FFT: Rather than processing one sample at a time, the DFT and the FFT algorithms process blocks, or frames, of samples. Frame - based processing divides continuous sequences of input and output samples into frames of N samples. In this experiment, the DSK will collect blocks of N=1024 samples taken at 8KHz from the codec. N=1024 samples will be read from the codec and stored. The function fft() will compute the 1024-point DFT of the block. The results will be displayed on the PC by using Code Composer Studio's graphing capabilities. Another block of data is then read from the codec, its DFT computed and visualized, and the process is repeated. Your program should:

i. Initialize the DSK and codec as usual, setting the sampling rate to 8KHz

ii. Using a loop, read N sample_pairs (codec outputs L and R samples as a pair) from codec, extract one half of each pair (corresponding to L or R), convert to

float and store in the real part of the structure array samples. These can be done with the following code fragment:

```
for (i = 0; i < N; i++) //read N samples from codec and store
{
    while (!DSK6713_AIC23_read(hCodec, &sample_pair))
       ;
     samples[i].real = (int)sample_pair>>16;
     samples[i].imag = 0.0;

}
```

To reduce effects of signal truncation, multiply the frame by a Hamming window sequence. The Hamming window array should be stored beforehand:

```
float hamming[N];
for(i=0; i<N; i++)
    hamming[i]=0.54-0.46*cos(2*pi*i/(N-1));
```

iii. Call fft() function to compute the N-point DFT of the frame

iv. Steps ii and iii should be enclosed in an infinite loop to repeat the frame read and DFT computation

v. Insert two breakpoints before and after fft() is called.

vi. Connect the headphone out of your PC to the line-in input of the DSK using a 3.5mm aux cable. Play a 1KHz tone from your PC (search youtube for 1KHz tone)

vii. Use Graph tool in CCS to display the time-domain signal at the first breakpoint and real part of the DFT at the second breakpoint. You might need to Reset the Graph and Refresh to auto-scale the graph at each breakpoint when the program is halted.

viii. Play tones of different frequencies from youtube (< 4KHz) and observe the signal and spectrum

ix. Play any audio signal from your PC and observe the signal and spectrum.

In the above code, the DSP spends nearly all the time waiting to receive samples from the codec. It processes the frame only when all the N samples have been acquired. A much more efficient approach is to let the DSP perform some other task in the background (such as computing the DFT of a previously acquired frame) and have the serial port interrupt the background tasks when a sample has been received. The interrupt service routine is called a foreground task. Such an approach requires two buffers(called ping pong buffers). While a new frame of input samples is being collected in one buffer using interrupts, a previously collected frame of input samples in the other buffer is processed. After the tasks are completed, the ping and pong buffers interchange their roles.

The TMS320C6713 has an enhanced direct memory access controller (EDMA) that can transfer data between any locations in the DSP's 32-bit address space independently of the CPU. Efficiency can also be improved by configuring the DMA controller to send/receive an entire block of data.

6. IFFT with FFT:

In Q2, we wrote the matrix form of the DFT equation as $\boldsymbol{X = D_N x}$, from which $\boldsymbol{x = D_N^{-1} X}$. The inverse of the DFT matrix is given by $\boldsymbol{D_N^{-1} = \frac{1}{N} D_N^*}$. So we have $\boldsymbol{x = \frac{1}{N} D_N^* X}$ (which is nothing but the IDFT equation in matrix form). If we take complex conjugate on both sides, we have $\boldsymbol{x^* = \frac{1}{N} D_N X^*}$. This relation suggests a method to find the IDFT of X[k] using a function to compute dft. If we apply the complex conjugate of a DFT sequence as the input to the fft() function and scale the output by 1/N, the result is the complex conjugate of the time-domain sequence. This method is illustrated below:

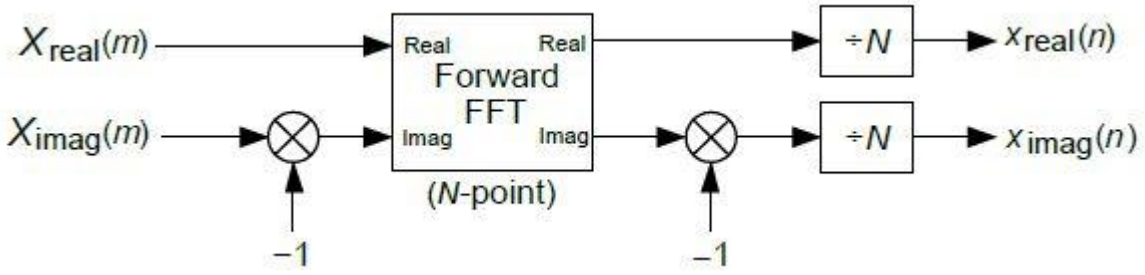


Image courtesy: https://www.dsprelated.com/showarticle/800.php

Modify the code in Q5 to find the IDFT of a sequence using this method. Generate a 1024-point sequence (or capture a 1024-point frame from the codec), take its FFT and apply this method to recover the time-domain sequence. Compare the graphs of the original sequence with the recovered sequence.

7. FIR Filter:

Design of filter: The objective of designing an FIR filter is to determine a set of filter coefficients that satisfies the given specifications. A variety of techniques have been developed for designing FIR filters. The window method is one of the oldest and simplest. The window method involves multiplying the impulse response of an ideal filter with a window sequence that tapers smoothly to zero at each end. i.e. $h[n] = h_d[n\,]w[n]$.

In order to design a low-pass FIR filter using the window method, we start with the impulse response of an ideal discrete-time low pass filter with linear phase response given by $h_d[n] = \frac{\sin\omega_c(n-\alpha)}{\pi(n-\alpha)} \quad -\infty \le n \le \infty$, where ω_c is the cut-off frequency in radians/sample. Choose $\alpha = \frac{L-1}{2}$, so that $h_d[n\,]$ is symmetric about the point $\frac{L-1}{2}$ (either full-sample symmetric or half-sample symmetric depending on whether the length L of the filter is odd or even). Since all window sequences of length L are also symmetric about $\frac{L-1}{2}$, the resulting $h[n] = h_d[n\,]w[n]$ is also symmetric and the filter will have a linear phase response.

Once the filter coefficients are determined, the filter can be implemented by the convolution formula $y[n] = \sum_{k=0}^{L-1} h[k]x[n-k]$, where the h[k]s are the coefficients of the length L FIR filter. In a software implementation, the filter coefficients and input samples are stored in arrays of length L. For each new input sample, an output sample is computed. This is called sample-by-sample processing in contrast with block processing. At some time instant n_0, the output $y[n_0] = h[0]x[n_0] + h[1]x[n_0-1] + h[2]x[n_0-2] + .. + h[L-1]x[n_0 - L + 1]$.

Note that the oldest input sample $x[n_0\text{-}L+1]$ is multiplied with the filter coefficient h[L-1] and the newest input sample $x[n_0]$ is multiplied with the filter coefficient h[0].

At this instant, the contents of the arrays h and x will be as follows:

h[0]	h[1]	..	..	..	h[L-1]

$x[n_0]$	$x[n_0\text{-}1]$				$x[n_0\text{-}L+1]$

To compute $y[n_0]$, the corresponding elements of the arrays are multiplied and the products are summed.

At the next time instant n_0+1, the output:

$y[n_0+1] = h[0]x[n_0+1] + h[1]x[n_0] + h[2]x[n_0-1] + .. + h[L-1]x[n_0 - L + 2]$

and therefore the array contents should be:

h[0]	h[1]	..	..	..	h[L-1]

$x[n_0+1]$	$x[n_0]$	$x[n_0\text{-}1]$			$x[n_0\text{-}L+2]$

We see that filter coefficient array remains constant, but the input array is refreshed at every sampling instant. The oldest sample is discarded , the rest samples are shifted one location to the right in the buffer, and a new sample (from ADC in real-time applications) is inserted to the left. To compute the output at an instant, the corresponding elements of the arrays are multiplied and the products are summed.

a. In the code below, we use the window method to design an FIR low pass filter for a cut-off frequency of 1 KHz and test it on real-time signals on the DSK. To convert the cut-off frequency in Hz to cut-off frequency in rad/sample, we use the relation $\omega = \Omega T = \frac{2\pi F}{F_s}$. Assuming a sampling frequency $F_s = 8KHz$, the cut-off frequency in rad/sample is $\omega_c = \frac{2\pi 1000}{8000} = \frac{\pi}{4}$.

```
#include <dsk6713.h>
#include <dsk6713_aic23.h>
#include <math.h>
#define L 11 //length of filter
int main(void)
{
    Uint32 sample_pair;
    float pi = 3.141592653589;
    float hamming[L], h[L], x[L] = { 0 }, y;
    int i;

    //Generate Hamming window sequence
    for (i = 0; i < L; i++)
        hamming[i] = 0.54 - 0.46 * cos(2 * pi * i / (L - 1));

    //cut-off frequency of filter in rad/sample
    float wc = pi / 4.0;

    //compute filter coeffs
    for (i = 0; i < L; i++)
        //avoid division by 0 when i=(L-1)/2
        if (i == (L - 1) / 2)
            h[i] = wc / pi * hamming[i];
        else
            h[i] = sin(wc * (i - (L - 1) / 2.0)) / (pi*(i - (L - 1)/2.0))
                    * hamming[i];

    DSK6713_init();
    DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
    DSK6713_AIC23_CodecHandle hCodec;
    hCodec = DSK6713_AIC23_openCodec(0, &config);

    /* Change the sampling rate to 8 kHz */
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_8KHZ);

    while (1)
    {
        while (!DSK6713_AIC23_read(hCodec, &sample_pair))
            ;
        //store top-half of sample from codec in x[0]
        x[0] = (int)sample_pair >>16;
```

```
            //process input sample:
            y = 0.0;

            for (i = 0; i < L; i++) //compute filter output
                y += h[i] * x[i];

            //shift delay line contents
            for (i = (L - 1); i > 0; i--)
                x[i] = x[i - 1];

            //output y to left channel
            sample_pair = (int)y <<16;
            while (!DSK6713_AIC23_write(hCodec, sample_pair))
                ;

        }

        /* Close the codec */
        DSK6713_AIC23_closeCodec(hCodec);

        return 0;
    }
```

Test the filter with different frequencies in the pass-band and stop-band (connect headphone out of PC to line-in of DSK using aux cable and play tones from youtube). Observe the effect of the filter on a music signal

b. More advanced FIR filters can be easily designed using the MATLAB GUI tool *filterDesigner*. It can be opened by typing *filterDesigner* at the Matlab command prompt. Design a bandpass FIR filter with the following specifications:
Sampling frequency = 8000 Hz.
Lower stopband cutoff frequency (Fstop1) = 1200 Hz
Lower passband cutoff frequency (Fpass1) = 1400 Hz
Upper passband cutoff frequency (Fpass2) = 1600 Hz
Upper stopband cutoff frequency (Fstop2) = 1800 Hz
Passband ripple = 1 dB
Stopband (both lower and upper) attenuation = 60 dB

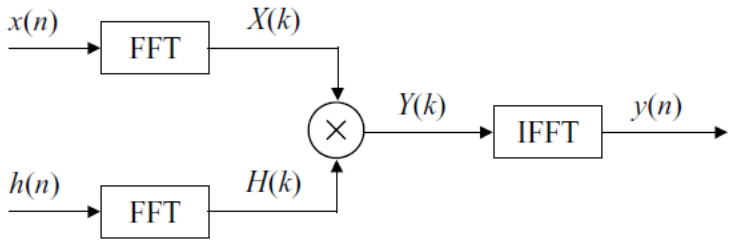
Use the default Equiripple method to design. We have two options for determining the filter order: we can specify the filter order by Specify Order, or use the default Minimum Order. Use the default minimum order. Pressing Design Filter button computes the filter coefficients. The GUI will show the Magnitude Response in dB. Note that order = 101 means the length of FIR filter is L = 102. We can analyze different characteristics of the designed filter from the Toolbar or by clicking the Analysis menu (or from toolbar). For example, selecting the Impulse Response displays the designed FIR filter coefficients. To create a C header file containing filter coefficients, select Generate C header from the Targets menu. Select Export As and select Single Precision Float. Click on Generate and save the file. Open the file in notepad, copy the array elements and paste as elements of

h array in CCS. Modify the code in part a above to implement the band pass filter and test with real-time signals on the DSK.

c. Circular buffer implementation of FIR filter: In the previous implementation, the entire contents of the input array are shifted at each instant. The shifting is done from right to left to prevent over-writing of needed contents. The oldest sample is over-written by the second oldest and the new sample enters the left. For large values of L, this becomes an inefficient operation because it involves the shifting of large amounts of data from one memory location to the next. An alternative approach is to keep the data unshifted but to shift the beginning address of the buffer. This leads to the concept of a circular buffer. Implement the filter in part a using circular buffer.

8. Overlap-Save

Ref. [7]. FIR filtering is a linear convolution of filter impulse response h(n) with the input sequence x(n). If the FIR filter has L coefficients, we need L real multiplications and L − 1 real additions to compute each output y(n). To obtain L output samples, the number of operations (multiplication and addition) needed is proportional to L^2. To take advantage of efficient FFT and IFFT algorithms, we can use the fast convolution algorithm illustrated in Figure below for FIR filtering.



Fast convolution provides a significant reduction in computational requirements for higher order FIR filters, thus it is often used to implement FIR filtering in applications having a large number of data samples.

It is important to note that the fast convolution shown in Figure above produces the circular convolution of x(n) and h(n). In order to produce a linear convolution, it is necessary to append zeros to the signals. If the data sequence x(n) has finite duration M, the first step is to pad data sequence and coefficients with zeros to a length corresponding to an allowable FFT size N (≥ L + M − 1), where L is the length of h(n). The FFT is computed for both sequences to obtain X(k) and H(k), the corresponding complex products Y (k) = X(k)H(k) are calculated, and the IFFT of Y (k) is used to obtain y(n). The desired linear convolution is contained in the first L + M − 1 terms of these results. Since the filter impulse response h(n) is known as a priori, the FFT of h(n) can be precalculated and stored as fixed coefficients.

For many applications, the input sequence is very long as compared to the FIR filter length L. This is especially true in real-time applications, such as in audio signal processing. In order to use the efficient FFT and IFFT algorithms, the input sequence must be partitioned into

segments of N (N > L and N is a size supported by the FFT algorithm) samples, process each segment using the FFT, and finally assemble the output sequence from the outputs of each segment. This procedure is called the block-processing operation. The cost of using this efficient block processing is the buffering delay. There are two techniques for the segmentation and recombination of the data: the overlap-save and overlap-add algorithms.
In this experiment we implement the Overlap-Save Technique.

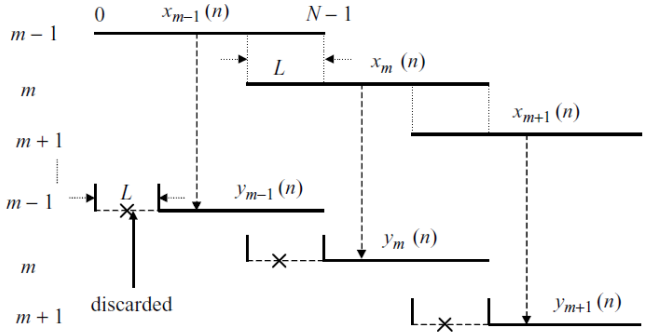


Image courtesy: [7]

The overlap-save process overlaps L input samples on each segment. The output segments are truncated to be nonoverlapping and then concatenated. The following steps describe the process illustrated in Figure above

1. Apply N-point FFT to the expanded (zero-padded) impulse response sequence to obtain H(k), where k = 0, 1, . . . , N − 1. This process can be precalculated off-line and stored in memory.
2. Select N signal samples $x_m(n)$ (where m is the segment index) from the input sequence x(n) based on the overlap illustrated in Figure above, and then use N-point FFT to obtain $X_m(k)$.
3. Multiply the stored H(k) (obtained in Step 1) by the $X_m(k)$ (obtained in Step 2) to get $Y_m(k) = H(k)X_m(k)$, k = 0, 1, . . . , N − 1.
4. Perform N-point IFFT of $Y_m(k)$ to obtain $y_m(n)$ for n = 0, 1, . . . , N − 1.
5. Discard the first L samples from each IFFT output. The resulting segments of (N − L) samples are concatenated to produce y(n).

Implement the steps above in a program to low pass filter a real-time speech signal applied to the DSK via line-in.

9. Overlap-Add

Ref: [7]. The overlap-add process divides the input sequence x(n) into nonoverlapping segments of length (N − L). Each segment is zero-padded to produce $x_m(n)$ of length N. Follow the Steps 2–4 of the overlap-save method to obtain N-point segment $y_m(n)$. Since the convolution is the linear operation, the output sequence y(n) is the summation of all segments.

Implement the overlap-add method to filter a real-time speech signal.

References

1. A. V. Oppenheim and R. W. Schafer, Discrete-Time Signal Processing, PHI
2. S. K. Mitra, Digital Signal Processing - A Computer-Based approach, TMH
3. R. Chassaing and D. Reay, Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK
4. S. A. Tretter, Communication System Design Using DSP Algorithms
5. Spectrum Digital, TMS320C6713 DSK Technical Reference
6. Texas Instruments, TMS320C67x DSP Library Programmer's Reference Guide
7. S. M Kuo, B. H. Lee, Real-Time Digital Signal Processing, Implementations and Applications, Wiley
8. S. J. Orfanidis, DSP Lab Manual, Rutgers University
9. Paul Embree, C Algorithms for Real-time DSP