DIGITAL SIGNAL PROCESSING LABORATORY

(ECL333)

Laboratory Manual

BTech Semester V

SREE BUDDHA COLLEGE OF ENGINEERING

PATTOR P.O, NOORNAD, ALAPPUZHA, KERALA -690529

Department

Of

Electronics and Communication Engineering

2022-23

This Page is left Intentionally

TABLE OF CONTENT

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

Mark Distribution:

Total Mark:150

CIE: 50

ESE:100

Introduction to GNU Octave

What is GNU Octave?

Octave is software featuring a high-level programming language, primarily intended for numerical computations. It was developed by John W. Eaton and written in C, C++, Fortran languages. It comes up with a text interface along with an experimental graphical interface. It is also used for various Machine Learning algorithms for solving various numeric problems. It is similar to MATLAB but performance is slow when compared to MATLAB.
GNU Octave is also freely redistributable software. You may redistribute it and/or modify it under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation.

What is MATLAB?

MATLAB is a multi-paradigm numerical computing environment and a high-performance language that is used for technical computing. The name MATLAB stands for matrix laboratory. It was developed by Cleve Molar of the company MathWorks.Inc in the year 1984. It is written in C, C++, Java. It allows matrix manipulations, plotting of functions, implementation of algorithms and creation of user interfaces.

Why we prefer GNU Octave over MATLAB in our lab?

S.No.	MATLAB	Octave
1	MATLAB is a matrix laboratory, referred to as language used for technical computations.	Octave is programming language used for numerical computations.
2	It was written in C, C++ and Java programming language.	It was written in C, C++ and Fortran Language.
3	It is not an open-source language.	It is an open-source language.
4	It consumes more memory when compared to octave.	It consumes less memory than MATLAB.
5	MATLAB has a very good interface. Hence, it is easy to operate.	Its interface is also as good as MATLAB.
6	In MATLAB, it is allowed to load the empty files.	In Octave, it is not allowed to load the empty files.
7	It does not support C-style auto-increment and assignment operators i.e. x++, ++x, etc.	It supports C-style auto-increment and assignment operators i.e. x++, ++x, etc.
8	The execution speed is faster than Octave.	The execution speed is slower than MATLAB.

a) Basic Operations in Octave

GNU Octave is a high-level programming language, primarily intended for numerical computations.
Below are the various basic functionalities of Octave:

1) Arithmetic Operations: Octave can be used to perform basic mathematical operations like addition, subtraction, multiplication, power operation etc.

Example:

```
close all;
clear all;
% Addition operation
23 + 65 + 8
% Subtraction operation
32 - 74
% Power operation
6 ^ 2
% Multiplication operation
45 * 7
% Division operation
5 / 6
```

Output:

```
ans = 96
ans = -42
ans = 36
ans = 315
ans = 0.8333
```

2) Logical Operations: Octave can be used to perform logical operations like AND, OR, NOT etc.

Example:

```
close all;
clear all;
% Logical AND
1 && 0
% Logical OR
1 || 0
% Logical NOT
~1
```

Output:

```
ans = 0
ans = 1
ans = 0
```

3) Relational Operations: Octave can be used to perform relational operations like greater than, less than etc.

Example:

```
close all;
clear all;
% Equal to
1 == 1
% Not equal to
0 ~= 0
% Greater than
1 > 0
% Less than
1 < 0
% Greater than equal to
1 >= 2
% Less than equal to
0 <= 0
```

Output:

```
ans = 1
ans = 0
ans = 1
ans = 0
ans = 0
ans = 1
```

4) Variables: Like other programming languages, Octave also has variables to temporarily store data.

Example:

```
close all;
clear all;
% Variable declaration and initialization
var1 = 2
% To create the variable and don't want to print it then put a semicolon at
the end of that command
var2 = 3;
% Variable of datatype char
ch = 'c'
% Storing the result of an operation in a variable
result = var1 + var2
% Storing the value of pi in a variable
var3 = pi;
% Printing a variable with disp() function
disp(var3);
% Using sprintf() function to print a string
disp(sprintf('3 decimal values : %0.3f', var3))
% Using format long to resize
format long
var3
% Using format short to resize
```

```
format short
var3
```

Output:

```
var1 = 2
ch = c
result = 5
3.1416
3 decimal values: 3.142
var3 = 3.141592653589793
var3 = 3.1416
```

5) Matrices and Vectors: Now let's learn how to deal with matrices and vectors in Octave. We can create matrix as shown below.

Example 1:

```
close all;
clear all;
% Creating matrix with 3 rows and 3 columns
matrix = [1 2 3; 4 5 6; 7 8 9]
```

Output:

```
matrix =
1   2   3
4   5   6
7   8   9
```

We can also make a vector; a vector is a matrix with n rows and 1 column (column vector) or 1 row with n columns (row vector).

Example 2:

```
close all;
clear all;
% Creating row vector
r_v = [1, 2, 3]
% Creating column vector
c_v = [1; 2; 3]
```

Output:

```
r_v =
1   2   3
c_v =
1
2
3
```

Here are some utility shortcuts to create matrices and vectors:

Creating vector using ":"

Example 3:

```
close all;
clear all;
% Creating vector using ":"
```

```
% The extreme end values denote the range and the middle value denotes the
step
v1 = 1 : 5 : 20
v2 = 1 : 0.5 : 5
% Without the step parameter
v3 = 1 : 10
```

Output:

```
v1 =
    1    6   11   16
v2 =
    1.0000  1.5000  2.0000  2.5000  3.0000  3.5000  4.0000  4.5000  5.0000
v3 =
    1    2    3    4    5    6    7    8    9   10
```

Creating matrix with all ones and zeros

Example 4:

```
close all;
clear all;
% Generate matrix of size 4x4 with all element as 1
ones_matrix = ones(4, 4)
% Generate matrix of size 4x4 with all element as 0
ones_matrix = zeros(4, 4)
% Generate matrix of size 4x4 with all element as 10
M = 10 * ones(4, 4)
% Generate row vector of size 5 with all elements 1
zeroes_vector = ones(1, 5)
% Generate row vector of size 5 with all elements 0
zeroes_vector = zeros(1, 5)
```

Output:

```
ones_matrix =
   1   1   1   1
   1   1   1   1
   1   1   1   1
   1   1   1   1
ones_matrix =
   0   0   0   0
   0   0   0   0
   0   0   0   0
   0   0   0   0
M =
   10   10   10   10
   10   10   10   10
   10   10   10   10
   10   10   10   10
zeroes_vector =
   1   1   1   1   1
zeroes_vector =
   0   0   0   0   0
```

Creating random matrix and random vectors

Example 5:

```
close all;
clear all;
% Generate row vector of some random numbers between 0 and 1
random_vector = rand(1, 5)
% Generate matrix of some random numbers between 0 and 1
random_matrix = rand(3, 4)
% Generate matrix with Gaussian distribution
% Where mean = 0 and variance and standard deviation = 1
gauss_matrix = randn(5, 5)
% Generate identity matrix with size 5x5
identity_matrix = eye(5)
```

Output:

```
random_vector =
   0.423893   0.352786   0.619899   0.269878   0.063018
random_matrix =
   0.240602   0.924415   0.157596   0.892195
   0.398675   0.931306   0.683808   0.423259
   0.083318   0.745773   0.888504   0.989097
gauss_matrix =
   1.552172   1.246823   2.759686  -1.171696   1.580425
  -1.314579   0.085221   0.125284   1.989425   0.511990
   0.020208   1.495984   0.472791  -0.497383   0.547206
  -0.451172   1.857004  -0.393860  -0.420161   0.973322
   0.383452  -0.532051   0.049422  -0.264828   0.487899
identity_matrix =
Diagonal Matrix
   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
```

b) How to take input in Octave GNU?

There are multiple library functions to take input from the user in Octave.

1) input() function: The input() function prints the prompt and waits for the user to enter a value, it takes an expression, evaluates it, and then returns it. The return arguments depend on the expression entered.

```
Syntax: input(prompt, “s”)
Parameters:
prompt: the text prompted on the terminal
“s”: indicates not to evaluate the entered expression
Returns:
depends on the input value
```

Example:

```
close all;
clear all;
%Without using the “s” attribute
var = input("Enter an expression : ");
% Enter expression 2 + 3
fprintf("Input is %d\n", var);
% Using the “s” attribute:
var = input("Enter an expression : ", "s");
% enter expression 2 + 3
fprintf("Input is %s\n", var);
```

Output:

```
Enter an expression: 2+3
Input is 5
Enter an expression: 2+3
Input is 2+3
```

2) yes_or_no () function: The yes_or_no () function accepts only two input values, either yes or no.

```
Syntax: yes_or_no(“prompt”)
Parameters:
prompt: the text prompted on the terminal
Returns: 1 if “yes”, 0 if “no”
```

Example:

```
close all;
clear all;
var1 = yes_or_no("Enter a value : ");
% entered value is yes
disp(var1)
var2 = yes_or_no("Enter a value : ");
% entered value is no
disp(var2)
```

Output:

```
Enter a value: (yes or no) yes
1
Enter a value: (yes or no) no
0
```

3) kbhit() function: The kbhit() function waits for any keypress, after the key is pressed, it returns that key.

```
Syntax: kbhit(argument)
Parameters:
argument: if called with an argument, it does not wait for a keypress
Returns: depends on the input value
```

Example:

```
close all;
clear all;
```

```
kbhit()
% enter the value as a
kbhit(1)
```

Output:

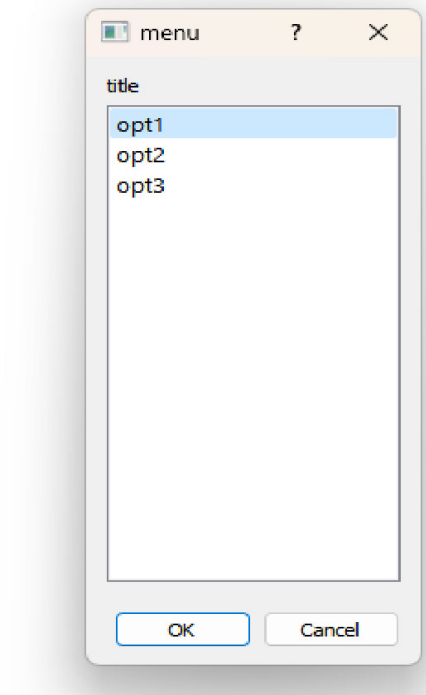
```
ans = j
ans =
```

4) menu(): The menu() function is used to display a menu with a heading title and options, and wait for the user input. If the GUI is running, the menu is displayed graphically Otherwise, the title and menu options are printed on the console.

```
Syntax: menu(title, opt1, …)
Parameters:
title: title of the menu window
opt1, opt2 …: list of options in the menu window
Returns: depends on the option selected
```

Example:

```
close all;
clear all;
% generating the menu
choice = menu("title", "opt1","opt2","opt3");
% displaying the choice
fprintf("The choice is : ");
disp(choice);
```



Note: Double click on any option and it will be displayed.

Output:

```
The choice is: 2
```

c) How to output in Octave GNU?

There are multiple library functions to display an output in Octave.

1) printf(): The printf() function prints the characters and returns the number of character printed, the format is a string starting with % and ends with conversion character (like c, i, f, d, etc.).

2) fprintf(): The fprintf() function is equivalent to the printf() function, except that the output is written to the file descriptor fid instead of stdout.

3) sprintf(): The sprintf() function is like the printf() function, except that the output is returned as a string.

4) disp(): The disp() function is used to either display the value on the console or assign the value to a variable.

Example:

```
close all;
clear all;
% using printf() without format specifiers
printf("Hello World!!\n");
x = 5;
y = 10;
% using printf() with format specifiers
printf("Welcome, value of x is %d \n", x);
% using sprintf() with format specifiers
result = sprintf("Welcome, value of y is %d \n", y);
printf ("%s\n",result);
% using fprintf() with format specifiers
fp1 = fopen('123.txt','w');
fprintf(fp1,"Welcome, value of x & y is %d and %d\n",x,y);
fclose(fp1);
type 123.txt
% using disp() with format specifiers
disp("Hello World!!");
x=20;
disp(x);
```

Output:

```
Hello World!!
Welcome, value of x is 5
Welcome, value of y is 10
Welcome, value of x & y is 5 and 10
Hello World!!
```

d) Loops (For and While) and Control Statements in Octave

Control statements are expressions used to control the execution and flow of the program based on the conditions provided in the statements.

1) if condition: This control structure checks the expression provided in parenthesis is true or not. If true, the execution of the statements continues.

2) if-else condition: It is similar to if condition but when the test expression in if condition fails, then statements in else condition are executed.

3) if-elseif condition: When the first if condition fails, we can use elseif to supply another if condition.

A loop is used to repeat a block of code until the specified condition is met.

1) for loop: It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

2) while loop: The while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of the loop.

Example of control statement:

```
close all;
clear all;
% Initializing variables variable1 and variable2
variable1 = 20;
variable2 = 20;
variable3 = 10;

% Check the if condition
if variable1 == variable2,
% If the condition is true this statement will execute
disp('The variables are Equal');
% end the if statement
endif;

% Check the if-else condition
if variable1 == variable3,
% If the condition is true this statement will execute
disp('The variables are Equal');
% If the condition is false else statement will execute
else
disp('The variables are Not Equal');
% End the if-else statement
end;

% Check the if-else condition
if variable1 == variable3,
% If the condition is true this statement will execute
disp('The variables 1 and Variable 3 are Equal');
```

```
% If the condition is false else statement will execute
elseif variable1 == variable2,
disp('The variables 1 and Variable 2 are Equal');
elseif  variable2== variable3,
disp('The variables 2 and Variable 3 are  Equal');
else
disp('The variables 1, variable 2 and Variable 3 are  different');
% End the if-else statement
end;
```

Output:

```
The variables are Equal
The variables are Not Equal
The variables 1 and Variable 2 are Equal
```

Example of loop statement:

```
close all;
clear all;
% Initializing for loop
disp('For loop running ');
for i = 1:5,
% Displays value of i
disp(i);
% End the for loop
endfor;

% Initializing while loop
disp('While loop running');
i = 1;
% While condition
while i <= 5
% Displaying the value of i
disp(i);
% Make an increment of 1 in the value of i
i = i + 1;
% End the while loop
Endwhile;
```

Output:

```
For loop running
1
2
3
4
5
While loop running
1
2
```

```
3
4
5
```

e) Basics of Plotting Data

Octave plotting features include choosing from various types of plots in 2D and 3D formats, decorating plots with additional information such as titles, labelled axes, grids, and labels for data, and writing equations and other important information about data. It is worth mentioning that plotting capabilities are essential to all simulation-based experiments since visual direction from the progressive steps give developers an intuitive understanding of the problem under consideration.

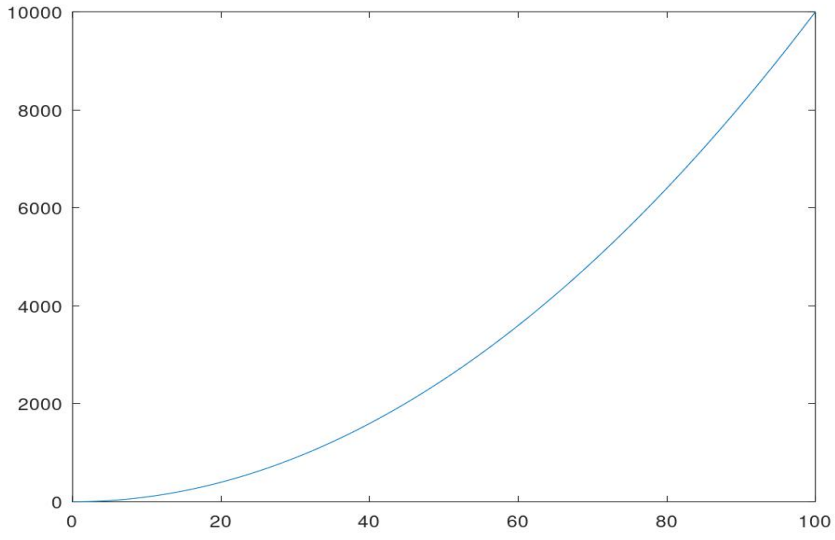
1) 2D Plotting: Octave presents a host of built-in functions for generating a variety of 2D plots.

a) The plot(x,y) Function

Example:

```
close all;
clear all;
x = linspace(0,100,100);
y = x.^2;
plot(x,y);
```

Output:



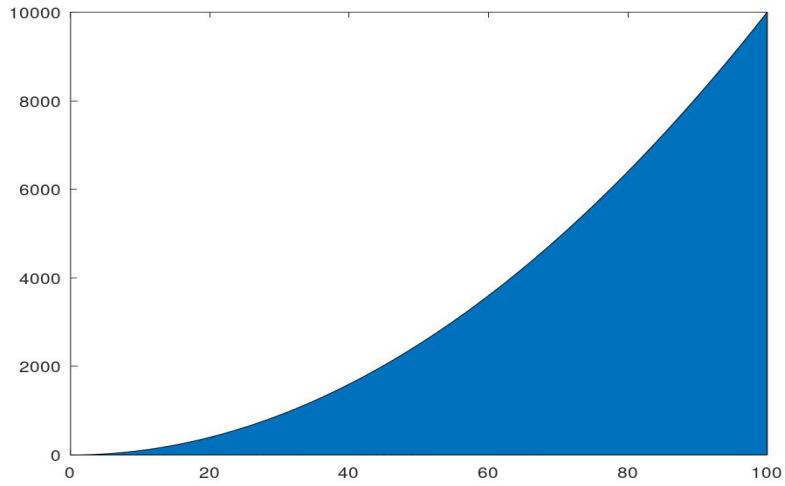
b) The area() Function

Example:

```
close all;
clear all;
x = linspace(0,100,100);
```

```
y = x.^2;
area(x,y);
```

Output:

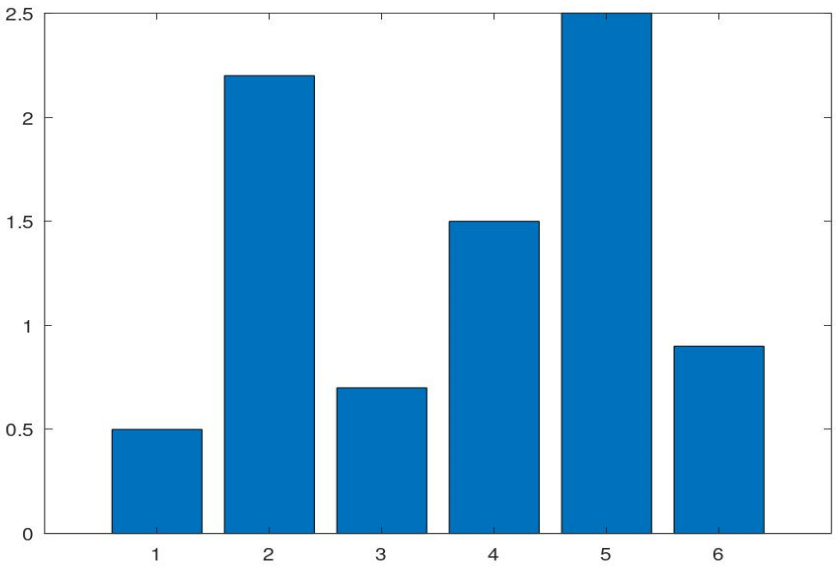


c) The bar() Function

Example:

```
close all;
clear all;
x = [1,2,3,4,5,6];
y = [0.5,2.2,0.7,1.5,2.5,0.9];
bar(x,y);
```

Output:



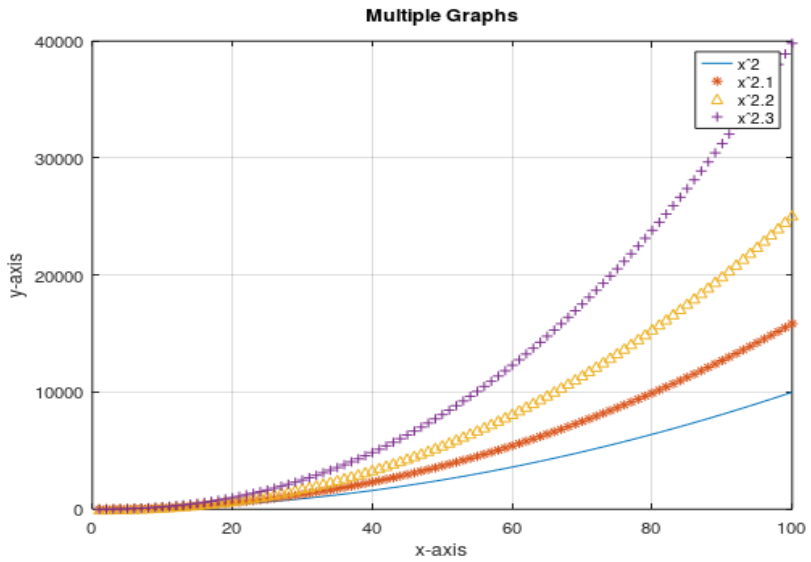
b) Plotting Multiple Plots on the Same Graph

Example:

```
close all;
clear all;
```

```
x = linspace(1,100,100);
y1 = x.^2.0;
y2 = x.^2.1;
y3 = x.^2.2;
y4 = x.^2.3;
plot(x,y1,"-",x,y2,"*",x,y3,"^",x,y4,"+");
grid on;
legend('x^2','x^{2.1}','x^{2.2}','x^{2.3}');
xlabel('x-axis');
ylabel('y-axis');
title('Multiple Graphs');
```

Output:



c) Plotting Multiple Plots Separately

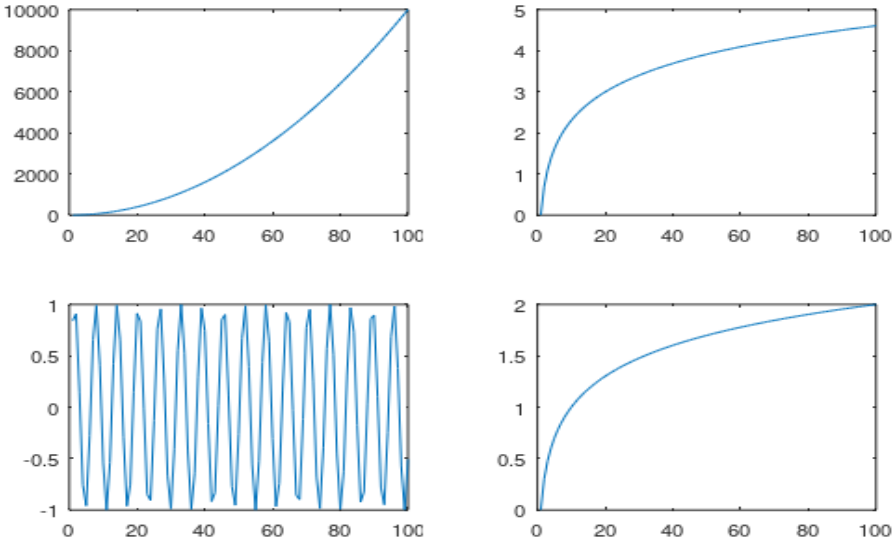
The subplot(row,column,index) command is used to plot multiple plots within the same figure separately. subplot(2,2,4) means that the plot will be on the second row, in the second column, and in the fourth index.

Example:

```
close all;
clear all;
x = linspace(1,100,100);
y1 = x.^2.0;
y2 = log(x);
y3 = sin(x);
y4 = log10(x);
subplot(2,2,1), plot(x,y1);
subplot(2,2,2), plot(x,y2);
subplot(2,2,3), plot(x,y3);
subplot(2,2,4), plot(x,y4);
gridon;
legend('x^2','x^{2.1}','x^{2.2}','x^{2.3}');
```

```
xlabel('x-axis');
ylabel('y-axis');
title('Multiple Graphs');
```

Output:

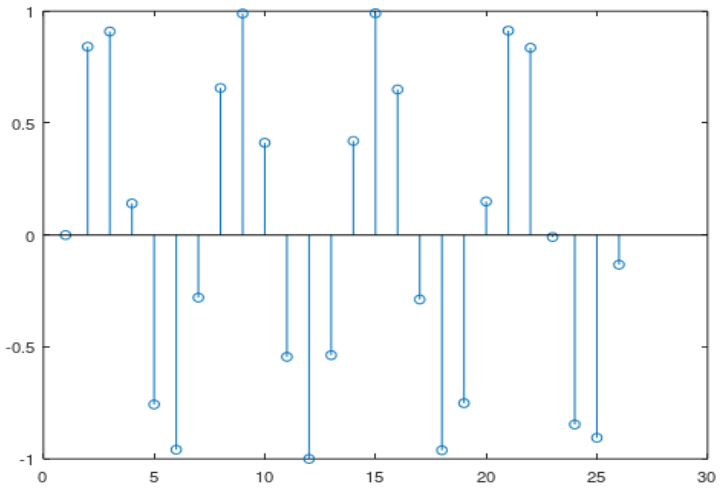


d) The stem() Function

Example:

```
close all;
clear all;
x = -4*pi:4*pi;
y = sin(x);
stem(y);
```

Output:



Experiment 1. Simulation of Signals

Aim: To simulate the following signals

1. Unit impulse signal

2. Unit pulse signal

3. Unit ramp signal

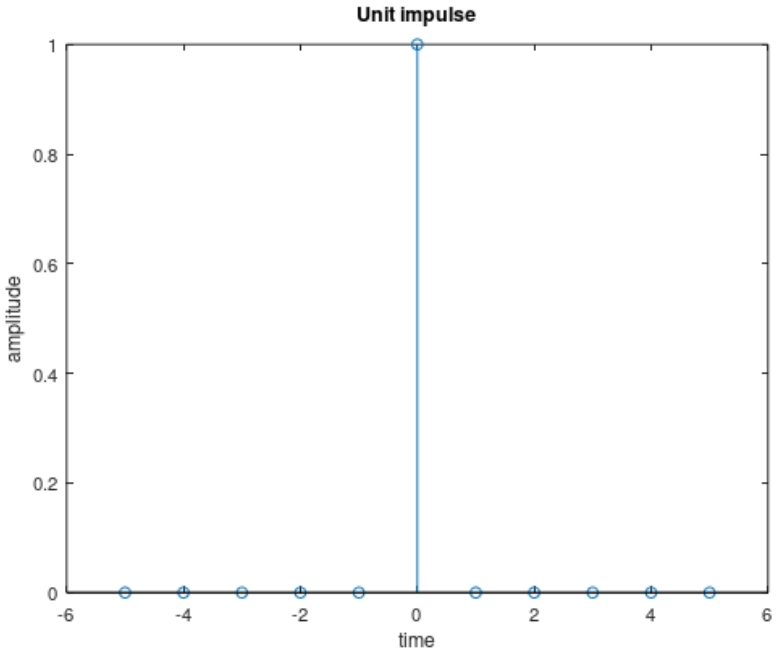
4. Bipolar pulse

5. Triangular signal

1. Unit impulse signal

```
clear all;
close all;
% Unit impulse
n=-5:5;
a = [zeros(1,5),ones(1,1),zeros(1,5)];
stem(n,a);
xlabel ('time');
ylabel ('amplitude');
title('Unit impulse');
```

Output:

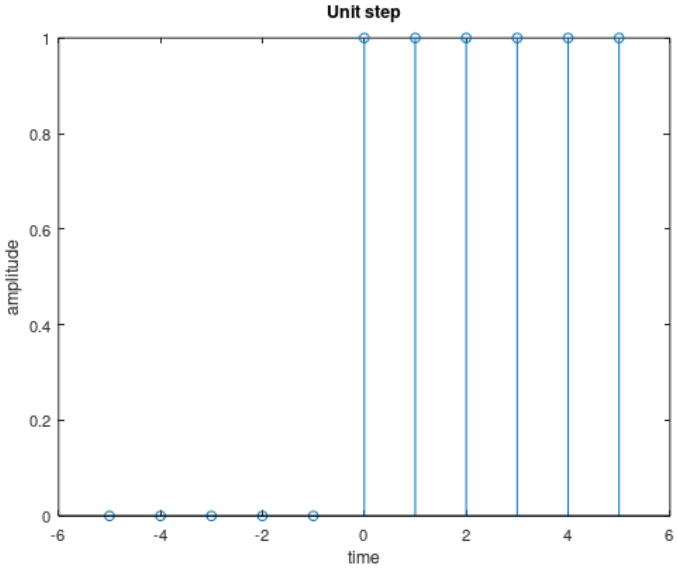


2. Unit pulse signal

```
clear all;
close all;
% Unit pulse signal
n=-5:5;
a = [zeros(1,5),ones(1,6)];
```

```
stem(n,a);
xlabel ('time');
ylabel ('amplitude');
title('Unit step');
```

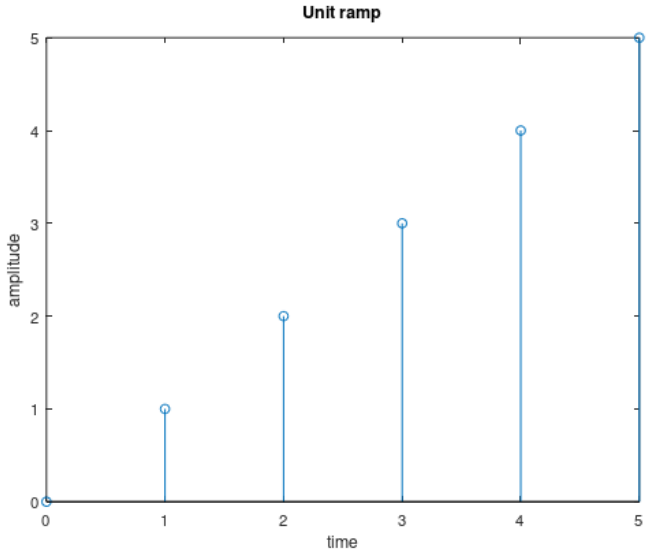
Output:



3. Unit ramp signal

```
clear all;
close all;
% Unit ramp signal
n= 0:5;
a = n;
stem(n,a);
xlabel ('time');
ylabel ('amplitude');
title('Unit ramp');
```

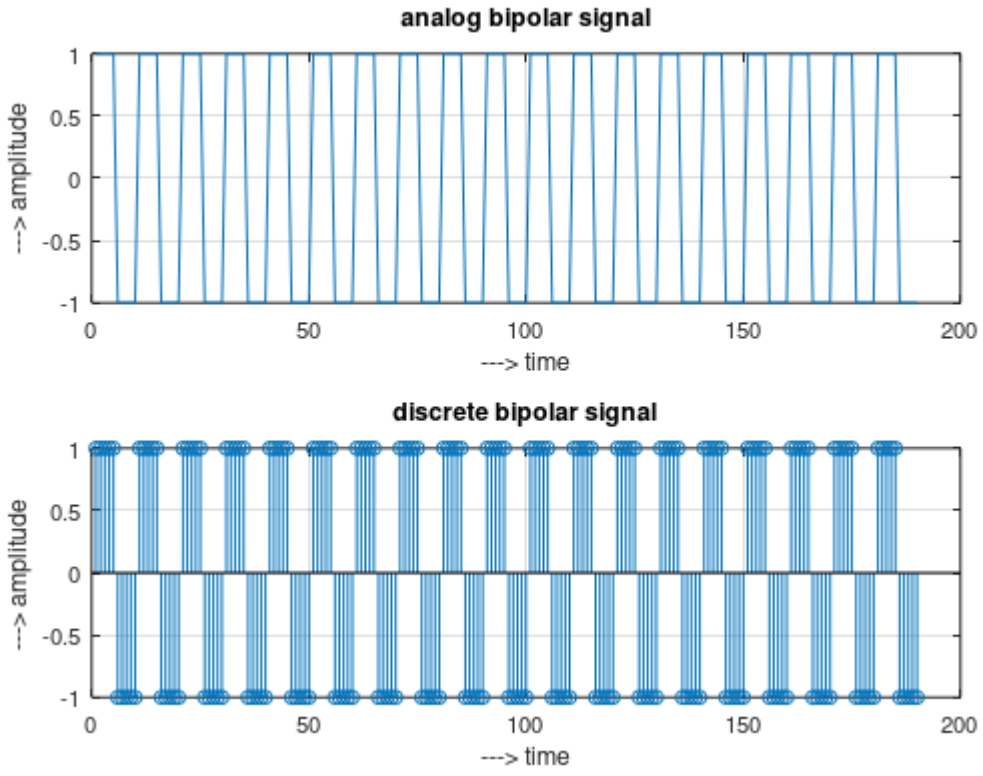
Output:



4. Bipolar pulse

```
% To generate a Bipolar signal
clc;
clear all;
close all;
N = input('enter the number of cycles....');
M = input('enter the amplitude....');
t1 = ones(1,5)
t2 = ones(1,5);
t = [];
for i = 1:0.5:N,
 t = [t,t1,-t2];
end;
subplot(2,1,1);
plot(t); grid on;
xlabel('---> time');
ylabel('---> amplitude');
title('analog bipolar signal');
subplot(2,1,2);
stem(t); grid on;
xlabel('---> time');
ylabel('---> amplitude');
title('discrete bipolar signal');
```

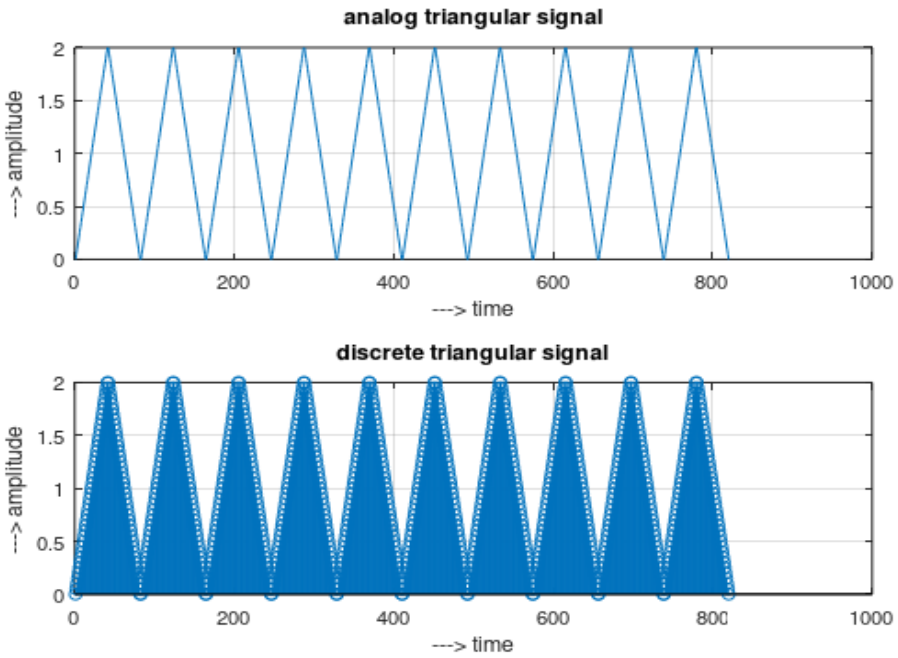
Output:



5. Triangular signal

```
% To generate a triangular signal
clear all;
close all;
N = input('enter the number of cycles....');
M = input('enter the amplitude....');
t1 = 0:0.05:M;
t2 = M:-0.05:0;
t = [];
for i = 1:N,
 t = [t,t1,t2];
end;
subplot(2,1,1);
plot(t); grid on;
xlabel('---> time');
ylabel('---> amplitude');
title('analog triangular signal');
subplot(2,1,2);
stem(t); grid on;
xlabel('---> time');
ylabel('---> amplitude');
title('discrete triangular signal');
```

Output:

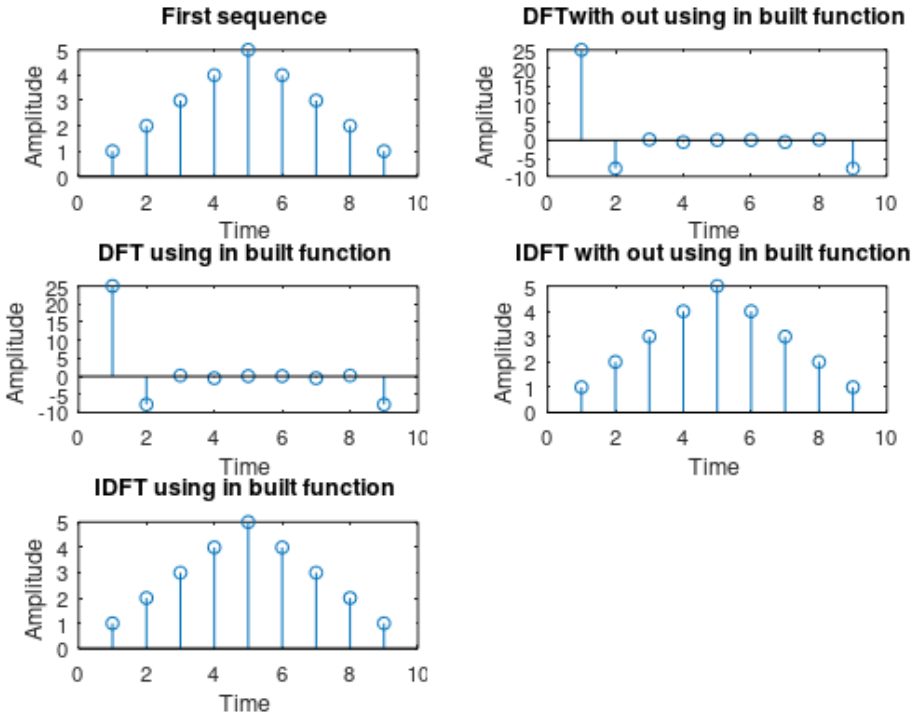


Experiment 2. Verification of the Properties of DFT

Aim1: Verify DFT pair

```
%% DFT AND IDFT OF A SEQUENCE
clear all;
close all;
x=input('Enter the First sequence = ');
N=input('Enter the length of DFT = ');
L = length(x);
x = [x zeros(1,N-L)];
%% Discrete Fourier Transform
f = zeros(1,N);
for k = 0:N-1
 for n = 0:N-1
     f(k+1) = f(k+1) + x(n+1)*exp(-j*2*pi*k*n/N);
  end
end
disp('DFT is: ');
disp(f);
x1=fft(x,N);%using inbuilt function
disp(x1);


%% Inverse Discrete Fourier Transform
r = zeros(1,N);

for n = 0:N-1
  for k = 0:N-1
      r(n+1) = r(n+1) + f(k+1)*exp(j*2*pi*k*n/N);
  end
  r(n+1) = r(n+1)/N;
end
disp('IDFT is: ');
disp(r);
x2=ifft(f,N);%using inbuilt function
disp(x2);
subplot(3,2,1);stem(x);xlabel('Time ');ylabel('Amplitude ');title('First
sequence ');
subplot(3,2,2);stem(f);xlabel('Time ');ylabel('Amplitude ');title('DFT with
out using in built function ');
subplot(3,2,3);stem(x1);xlabel('Time ');ylabel('Amplitude ');title('DFT using
in built function ');
subplot(3,2,4);stem(r);xlabel('Time ');ylabel('Amplitude ');title('IDFT with
out using in built function ');
subplot(3,2,5);stem(x2);xlabel('Time ');ylabel('Amplitude ');title('IDFT using
in built function ');
```

Output:



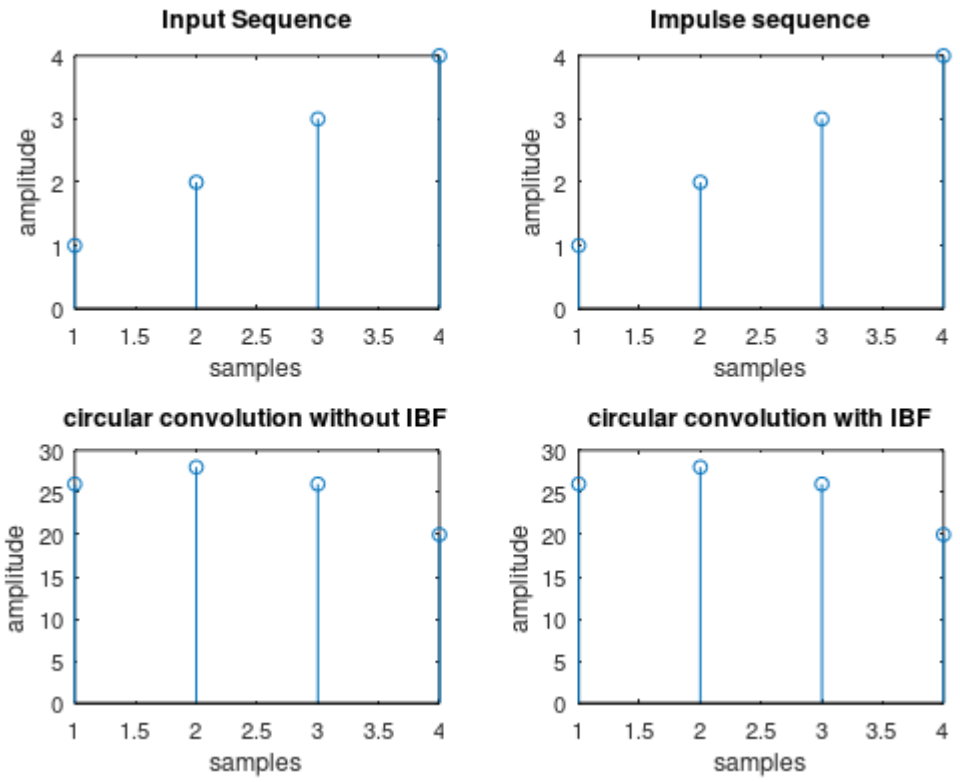
Aim2: Circular Convolution

Circular convolution: The circular convolution of two N-point sequences x[n] and h[n] is another N point sequence y[n] defined as $y[n] = \sum_{k=0}^{N-1} x[k]h[(n-k)_N]$. 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 counter clockwise 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.

```
%COMPUTATION OF CIRCULAR CONVOLUTION:
clc;
clear all;
close all;
x=input('Enter the sequence = ');
h=input('Enter the imp response = ');
% no of samples in x
n1=length(x);
n2=length(h);
n=max(n1,n2);
x=[x,zeros(1,n-n1)];
h=[h,zeros(1,n-n2)];
for i = 1:n
```

```
    y(i)=0;
    for j=1:n
        k=i-j+1;
        if(k<=0)
        k=n+k;
        end;
     y(i)=y(i)+[x(j)*h(k)];
     end;
end;
z = cconv(x,h,n);
t=1:n ;
subplot(2,2,3)
stem(t,y);
title('circular convolution without IBF');
xlabel('samples');
ylabel('amplitude');
subplot(2,2,4)
stem(t,z);
title('circular convolution with IBF');
xlabel('samples');
ylabel('amplitude');
```

Output:



Aim3: Parseval's Theorem

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

```
%Parseval's Theorem Verification
close all;
clear all;
clc;
numSamples = 5000;
x1 = randi(10,numSamples,1);
X1 = fft(x1);
% Summation by Parseval Theorem
out1 = sum(x1.* conj(x1));
out2 = sum(X1.*conj(X1)) / numSamples;
out2 = round(out2);
if (out1 == out2)
  printf ("LHS (%u) = RHS (%d) hence Parseval's Theorem verified
\n",out1,out2);
  else
  printf ("LHS (%u) != RHS (%d) hence Parseval's Theorem not verified
\n",out1,out2);
end
```

Output:

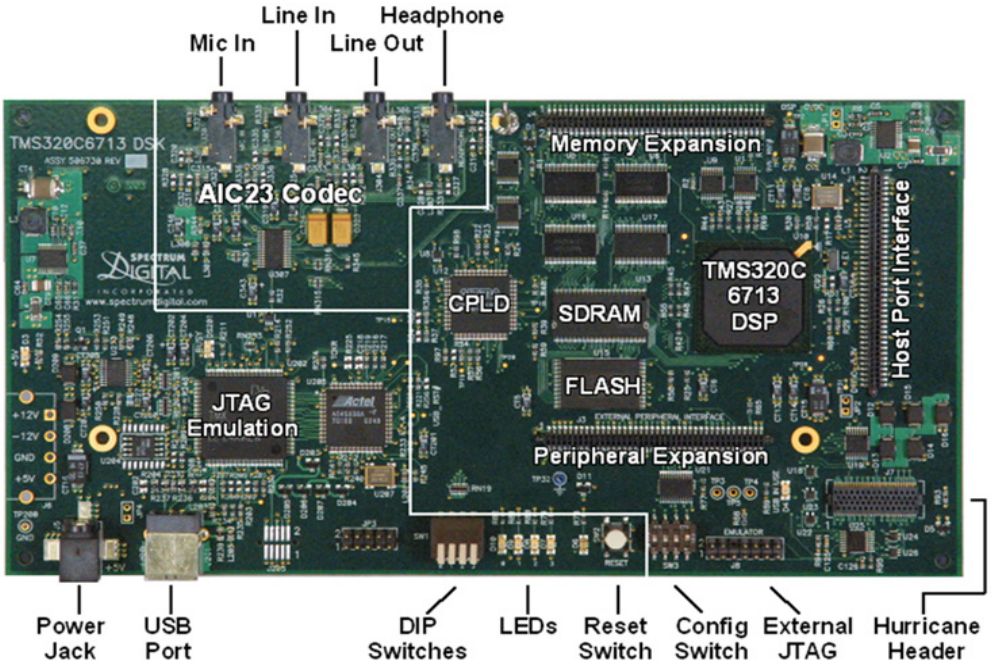
```
LHS (194699) = RHS (194699) hence Parseval's Theorem verified
```

Experiment 3. Familiarization of DSP Hardware (TMS320C6713)

Aim1: 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.

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. DSPs 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)

Aim2: Familiarization of the analog and digital input and output ports of the DSP board

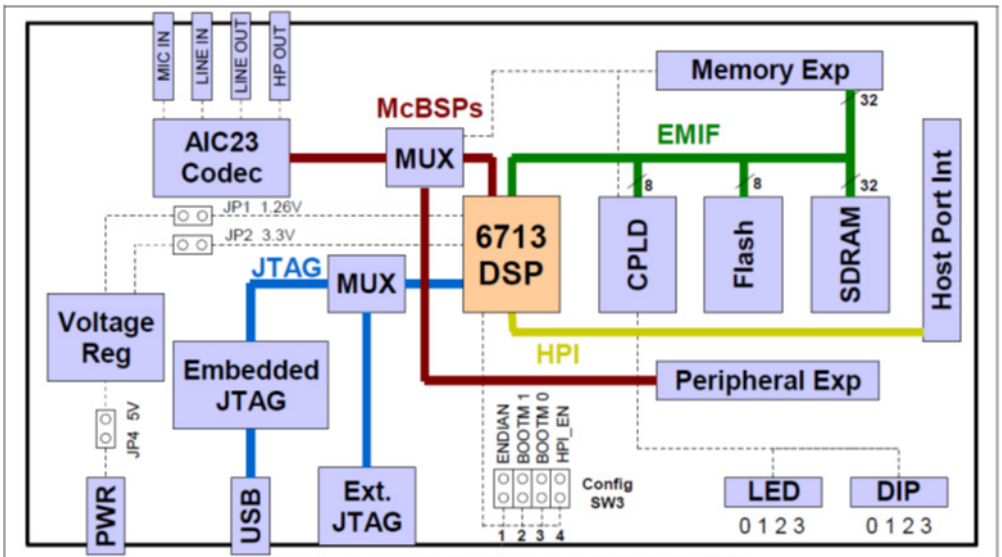


Figure 1-1, Block Diagram C6713 DSK

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.

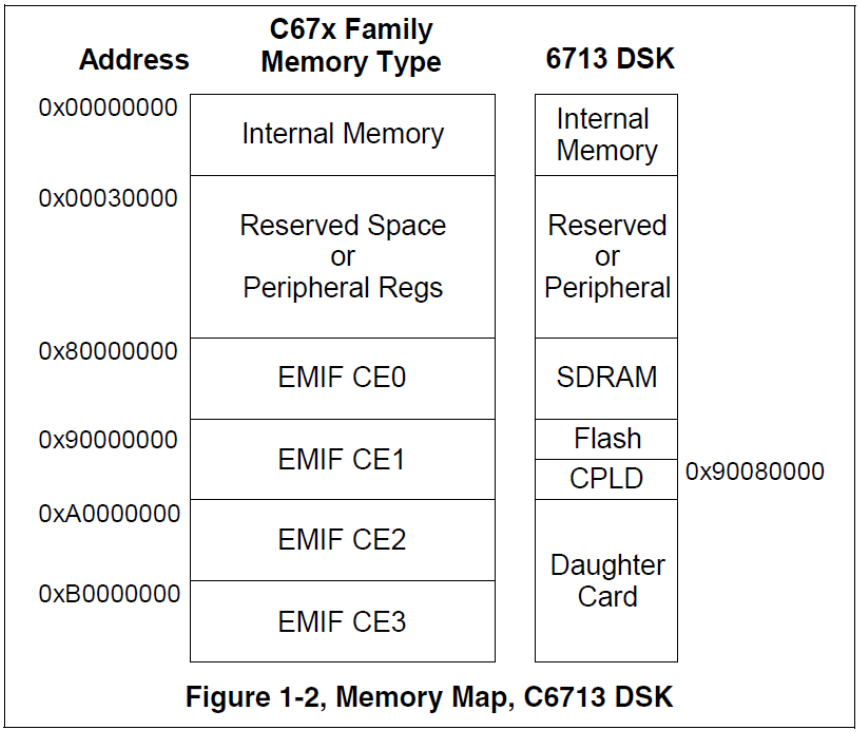


Figure 1-2, Memory Map, C6713 DSK

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.

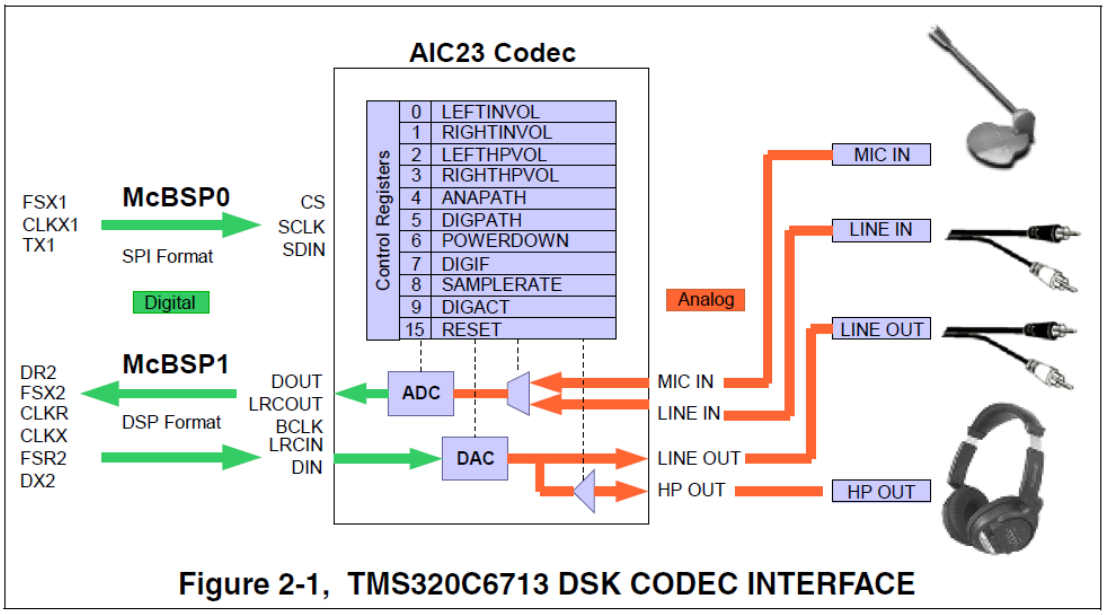


Figure 2-1, TMS320C6713 DSK CODEC INTERFACE

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 5.

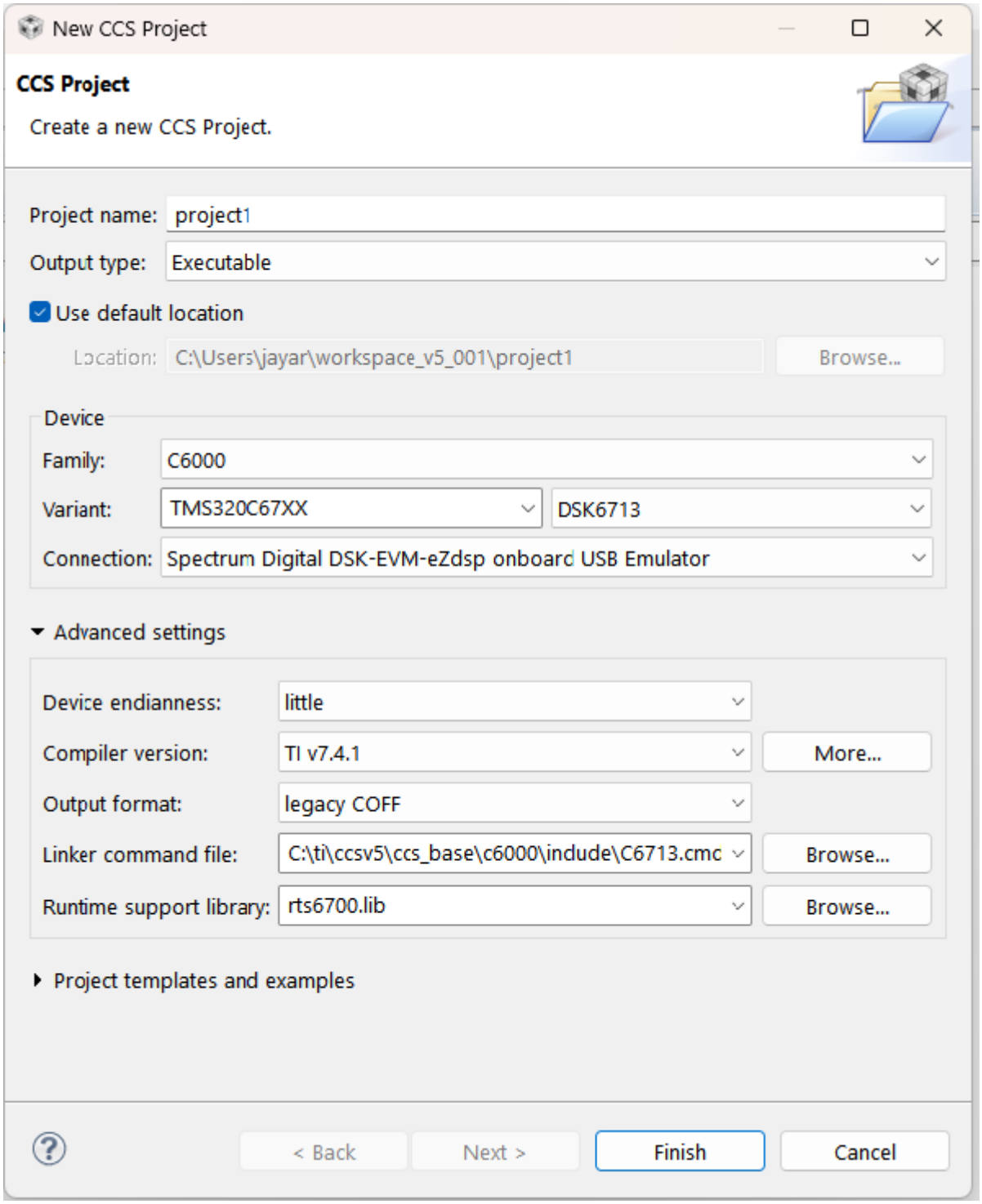
Creating a Project in CCS

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

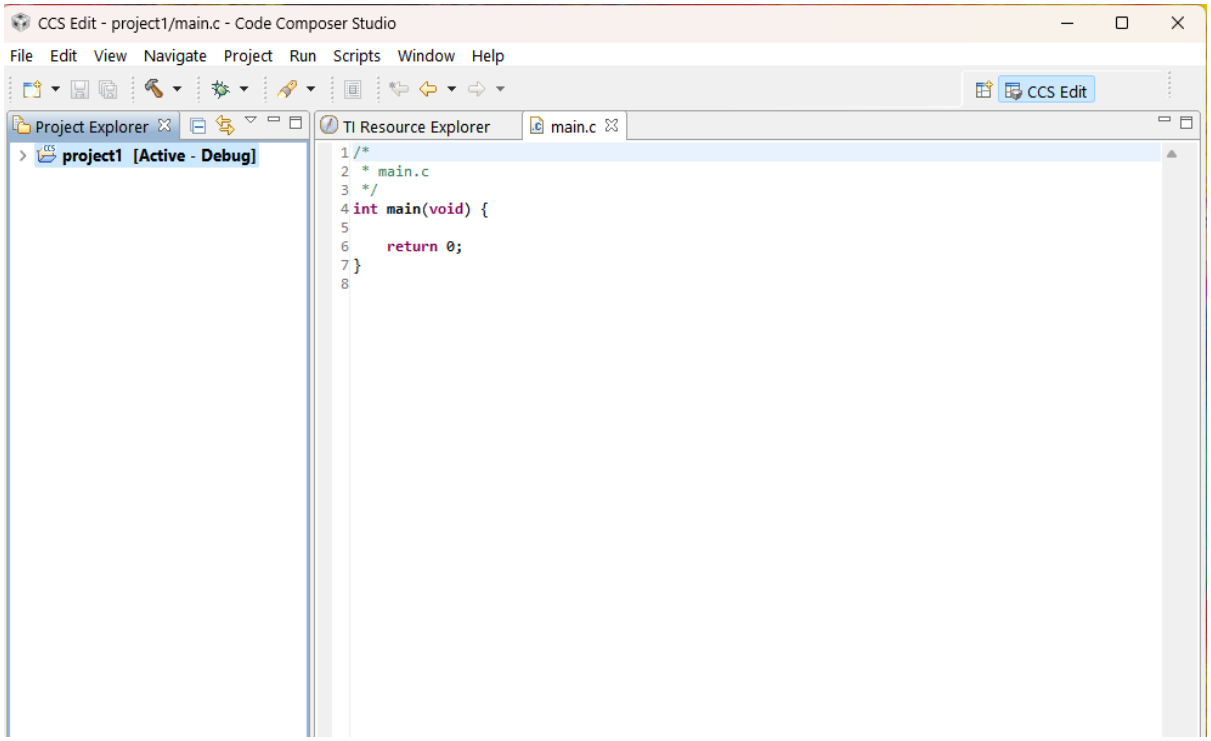
Go to menu *Project* → *New CCS Project*… or *File* → *New* → *CCS Project*.

In the New CCS Project wizard:

Make the following selections and press Finish button:

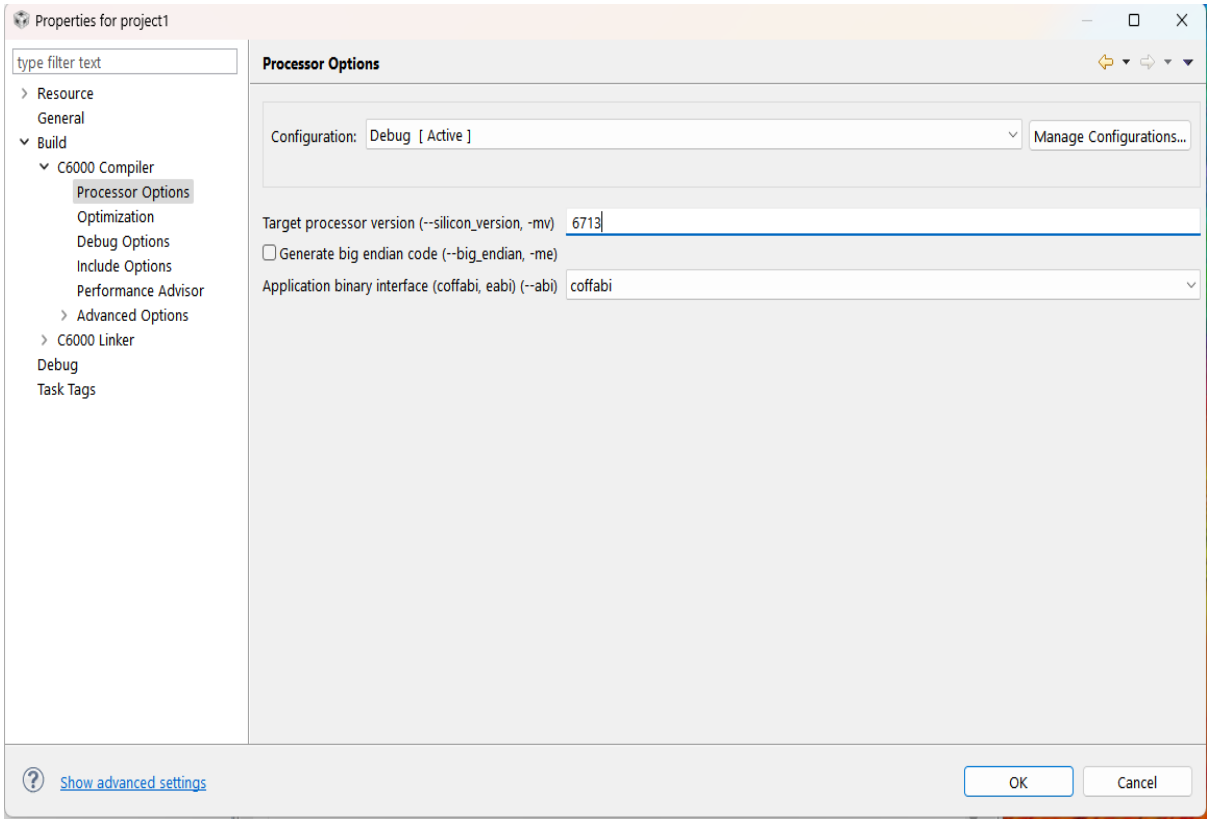


Your project should now show in the Project Explorer window.

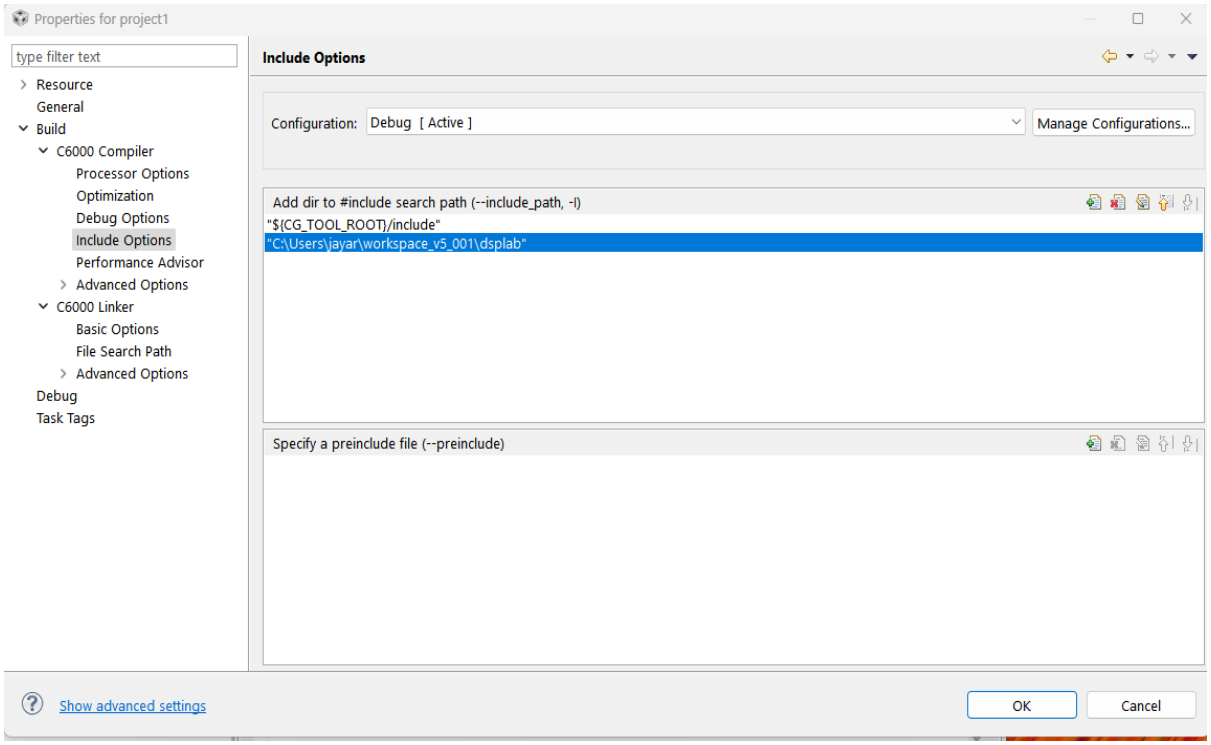


First, we need to set some properties for our project. Right-click on the project and go to Properties (Or from menu *Project* > *Properties*)

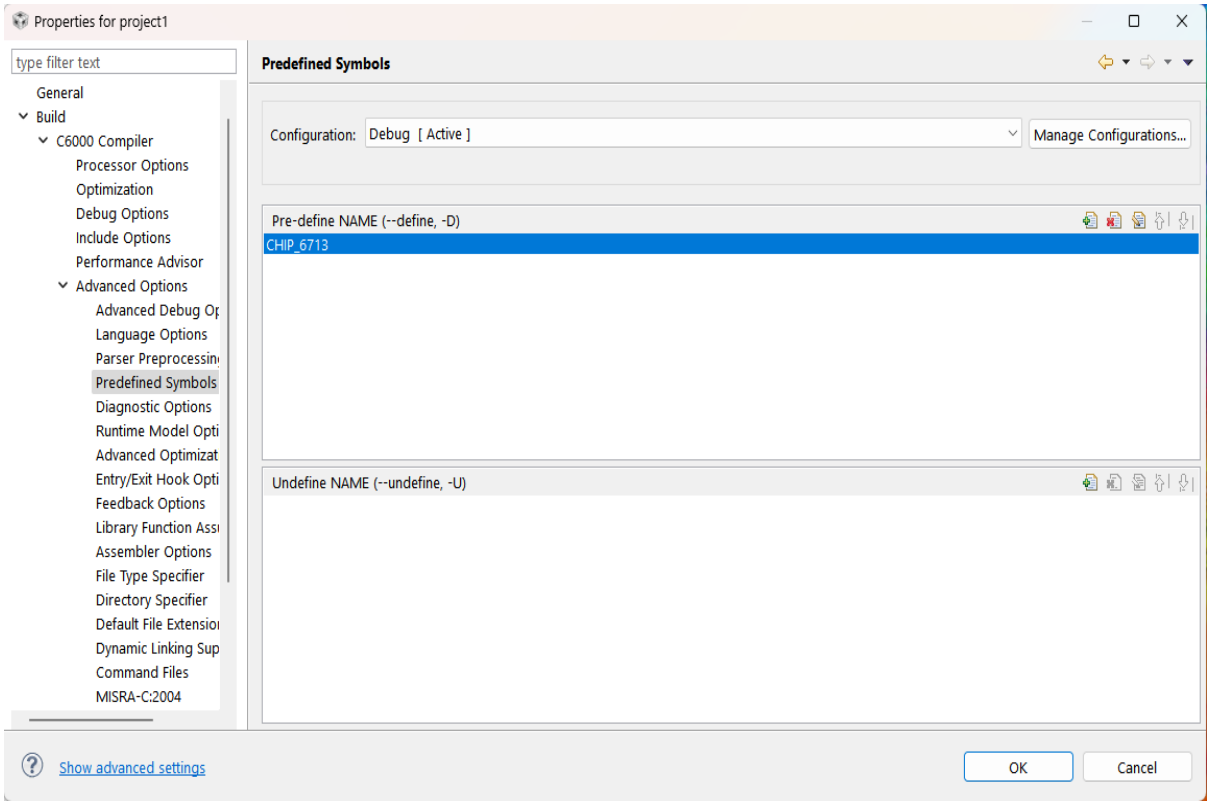
Under *Build>C6000 Compiler>Processor Options*, set *Target processor version* as 6713



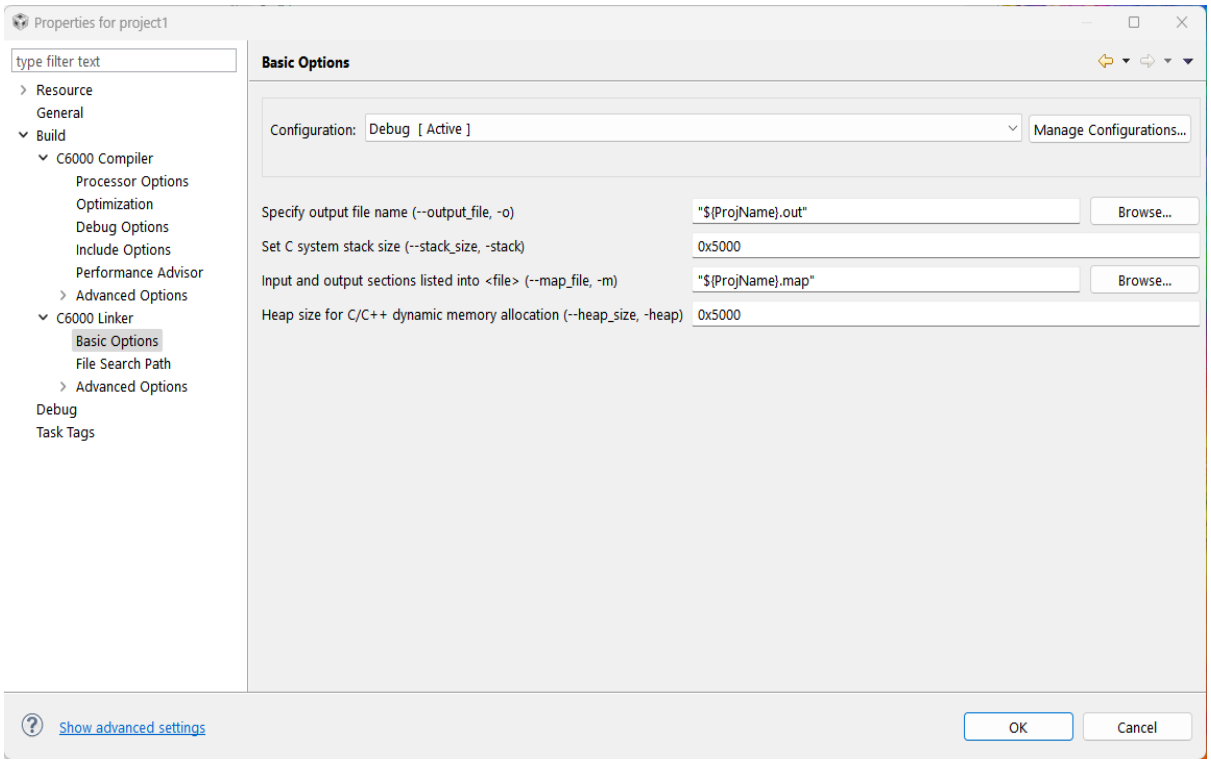
Under *Include Options*, go *to Add dir to #include search path*, click on the file icon with a green + mark and Browse to the folder *dsplab* provided to you and click OK. This folder contains the header files for the board support library and chip support library functions.



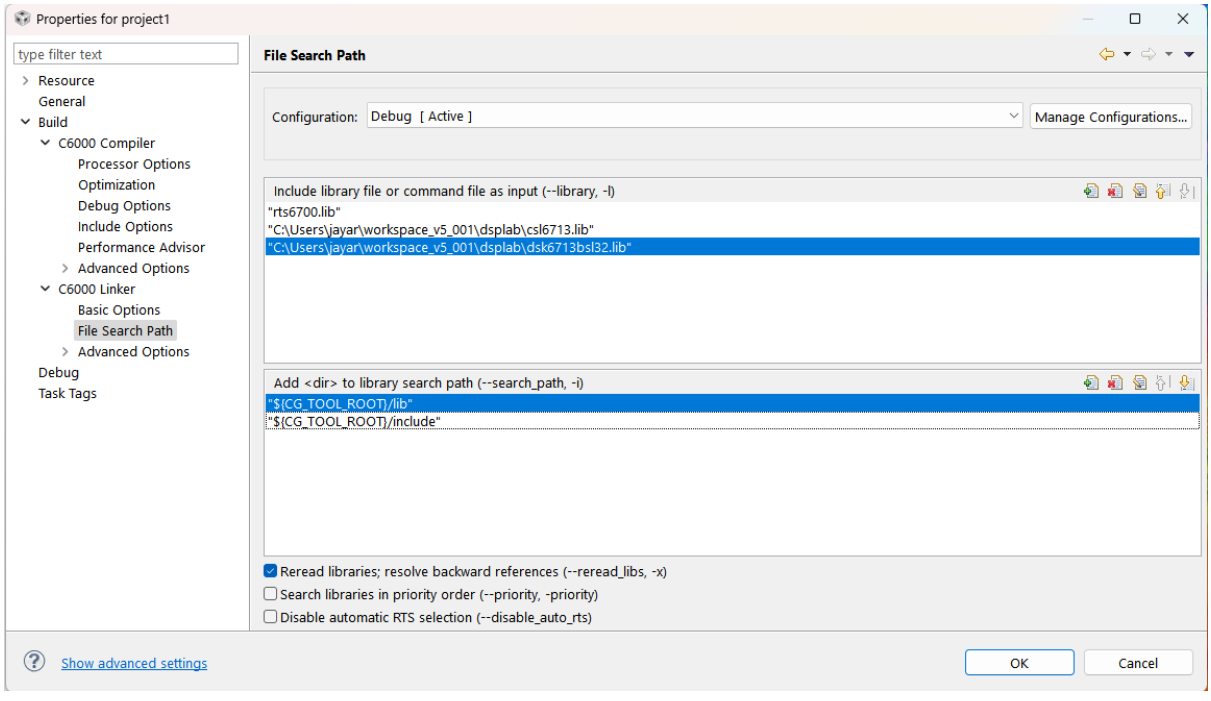
Under *Advanced Options>Predefined Symbols*, in the *Pre-define NAME* window, click on the file icon with green + , and enter CHIP_6713. This symbol is used for conditional compilation. If you don't do this step, you will have to type the line #define CHIP_6713 in your source file.



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* and click ok.



Aim3: Generation and cross compilation and execution of the C code to connect the input digital switches to the output LEDs.

Experiment 1: Led Blink control using Dip switches

```
#include "dsk6713.h"
#include "dsk6713_led.h"
#include "dsk6713_dip.h"
void main()
{
    while(1)
    {
        if(DSK6713_DIP_get(0)==0) // instruction showing how to use DIP
switches,
        {
            DSK6713_LED_on(0); // led0 turns ON
            DSK6713_waitusec(200000);
            DSK6713_LED_off(0); // led0 turns Off
            DSK6713_waitusec(200000);
        }
        else if(DSK6713_DIP_get(1)==0)
        {
            DSK6713_LED_on(1); //led1 turns ON
            DSK6713_waitusec(200000);
            DSK6713_LED_off(1); // led1 turns Off
            DSK6713_waitusec(200000);
        }
        else if(DSK6713_DIP_get(2)==0)
        {
            DSK6713_LED_on(2); //led2 turns ON
            DSK6713_waitusec(200000);
            DSK6713_LED_off(2); // led2 turns Off
            DSK6713_waitusec(200000);
        }
        else if(DSK6713_DIP_get(3)==0)
        {
            DSK6713_LED_on(3); //led3 turns ON when
            DSK6713_waitusec(200000);
            DSK6713_LED_off(3); // led3 turns Off when
            DSK6713_waitusec(200000);
        }
        else
        {
            DSK6713_LED_on(0); //led0,1,2,3 all turn ON
            DSK6713_LED_on(1);
            DSK6713_LED_on(2);
            DSK6713_LED_on(3);
            DSK6713_waitusec(200000);
            DSK6713_LED_off(0);//led0,1,2,3 all turn Off
```

```
            DSK6713_LED_off(1);
            DSK6713_LED_off(2);
            DSK6713_LED_off(3);
            DSK6713_waitusec(200000);
        }
    } // end of while
} // end of main loop
```

Experiment 2: Tone generation to demonstrate AIC23 Codec Line out/Headphone out.

```
#include <math.h>
#include <dsk6713_aic23.h>
#include <dsk6713.h>
#include "dsk6713_led.h"
#include "dsk6713_dip.h"
int main()
{
    float Fs = 8000.;
    float F0 = 500.;
    float pi = 3.14;
    float theta = 0;
    float delta = 2. * pi * F0 / Fs; // increment for theta
    float sample;
    unsigned out_sample,out_sample1;
    /* 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 48 kHz */
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_48KHZ);

      while(1)
        {
            if(DSK6713_DIP_get(0)==0) //if DIP #0 pressed
                {
                    DSK6713_LED_on(0);
                    sample = 15000.0 * sin(theta); /* Scale for DAC */
                    out_sample = (int)sample & 0x0000ffff; // Put in lower
half (R)
                    while (!DSK6713_AIC23_write(hCodec, out_sample));
                    theta += delta;
                    if (theta > 2 * pi)
                    {
                        theta -= 2 * pi;
                    }
                }
```

```
            else if(DSK6713_DIP_get(3)==0) //if DIP #3 pressed
                {
                    DSK6713_LED_on(3);
                    sample = 15000.0 * sin(theta); /* Scale for DAC */
                    out_sample1 = out_sample = (int)sample << 16;  // Put in
                                                                   upper half (L)
                    while (!DSK6713_AIC23_write(hCodec, out_sample1));//
                    theta += delta;
                    if (theta > 2 * pi)
                    {
                        theta -= 2 * pi;
                    }
                }

            else
                {
                    DSK6713_LED_on(0);
                    DSK6713_LED_on(3);
                    DSK6713_waitusec(200000);
                    DSK6713_LED_off(0);
                    DSK6713_LED_off(3);
                    DSK6713_waitusec(200000);
                }
        }
}
```

Experiment 3: Loopback to demonstrate AIC23 Codec Line out/Headphone out and Line in.

```
#include <math.h>
#include <dsk6713_aic23.h>
#include <dsk6713.h>
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define BUF_SIZE 8000
void main()
{
    Uint32 IN_L=0, IN_R=0;
    Uint32 OUT_L= 0,OUT_R=0;// both channels packed in 32-bits
    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 96 kHz */
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_96KHZ);
            while (1)
            {
```

```
                while (!DSK6713_AIC23_read(hCodec, &IN_L));
                while (!DSK6713_AIC23_read(hCodec, &IN_R));
                OUT_L = IN_L;
                OUT_R = IN_R;
                while (!DSK6713_AIC23_write(hCodec, OUT_L));
                while (!DSK6713_AIC23_write(hCodec, OUT_R));
            }
}
```

Experiment 4. Linear convolution

Aim: To perform linear convolution

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 **linconv()** 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.

```c
// Linear convolution
#include<stdio.h>
void linconv(int ,int );
int main()
{
    int length1,length2;
    // Entering the length of X and H
    printf("\n Enter length of first sequence  = ");
    scanf("%d",&length1);
    printf("\n Enter length of second sequence  = ");
    scanf("%d",&length2);
    linconv(length1,length2);
}
//creating a function linconv()
void linconv(int m,int n)
{
    int x[15],h[15],y[15];
    int i,j,k;
    // Entering the value of X and H
    printf("\n Enter values for first sequence x(n):\n");
    for(i=0;i<m;i++)
    {
        scanf("%d",&x[i]);
    }
    printf("\n Enter Values for second sequence h(n):\n");
    for(i=0;i<n; i++)
    {
        scanf("%d",&h[i]);
    }

    //Computing the length of
    int L=m+n-1;

    // Padding of zeros
    for(i=m;i<=L;i++)
    {
        x[i]=0;
    }
```

```
    for(i=n;i<=L;i++)
    {
        h[i]=0;
    }

    //Convolution operation
    for(i=0;i<L;i++)
    {
    y[i]=0;
        for(j=0;j<=i;j++)
        {
            y[i]=y[i]+(x[j]*h[i-j]);
        }
    }
    //Displaying the o/p
    for(k=0;k<L;k++)
    {
        printf("\n The Value of output y[%d]=%d",k,y[k]);
    }
}
```

Output:

```
Enter length of first sequence m = 4
Enter length of second sequence n = 4

Enter values for first sequence x(n):
1
2
3
4

Enter Values for second sequence h(n):
4
3
2
1

The Value of output y[0]=4
The Value of output y[1]=11
The Value of output y[2]=20
The Value of output y[3]=30
The Value of output y[4]=20
The Value of output y[5]=11
The Value of output y[6]=4
```

Experiment 5. FFT of signals

Aim1: Computing DFT and IDFT of a sequence

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 $\omega_k = \frac{2\pi k}{N}, k = 0 \dots N-1$.

For DFT

$DFT\{x[n]\} = X[k] = X(e^{j\omega})\Big|\omega = \frac{2\pi k}{N} = \sum_{n=0}^{N-1} x[n]\, e^{-\frac{j2\pi kn}{N}}$. Using the commonly used notation $W_N = e^{\frac{-j2\pi}{N}}$, $X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{kn}, k = 0 \dots N-1$

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

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

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

For IDFT

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

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

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

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

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
#define N 8
typedef struct
{
    float real;
    float imag;
} COMPLEX;
void dft(COMPLEX *);
void idft(COMPLEX *);

void main()
```

```
{
    int n,M;
    COMPLEX samples[N]={0.0};
    printf("Enter the length of the Input sequence = ");
       scanf("%d",&M);
//inputing the signal
    printf("Enter the Input sequence real part = ");
        for (n = 0; n < M; n++) //M samples of x[n]
        {
            scanf("%f",&samples[n].real);
        }
    printf("Enter the Input sequence imaginary  part = ");
        for (n = 0; n < M; n++) //M samples of x[n]
        {
            scanf("%f",&samples[n].imag);
        }
        printf("Display DFT Output:\n");
        dft(samples); //call DFT function
}

//Computing DFT
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++)
    {
        printf(" (%.1f) + (%.1f) \n",result[k].real, result[k].imag );
    }
    printf("Display IDFT Output:\n");
    idft(result);
}

//Computing IDFT
void idft(COMPLEX *X)
{
```

```
    COMPLEX result1[N];
    int k, n;
    for (k = 0; k < N; k++) // N point DFT
    {
        result1[k].real = 0.0;
        result1[k].imag = 0.0;
        for (n = 0; n < N; n++)
        {
            result1[k].real += (X[n].real * cos(2 * PI * k * n / N) -
                                X[n].imag * sin(2 * PI * k * n / N))/N;
            result1[k].imag += (X[n].imag * cos(2 * PI * k * n / N) +
                                X[n].real * sin(2 * PI * k * n / N))/N;
        }
    }
    for (k = 0; k < N; k++)
    {
        printf("(%.1f) + (%.1f) \n",result1[k].real, result1[k].imag);
    }
}
```

Output:

```
Enter the length of the Input sequence = 8
Enter the Input sequence real part =
1
2
3
4
5
6
7
8
Enter the Input sequence imaginary part =
0
0
0
0
0
0
0
0
Display DFT Output:
 (36.0) + (0.0)
 (-4.0) + (9.7)
 (-4.0) + (4.0)
 (-4.0) + (1.6)
 (-4.0) + (-0.1)
 (-4.0) + (-1.7)
```

```
 (-3.9) + (-4.1)
 (-3.8) + (-9.7)
Display IDFT Output:
 (1.0) + (-0.0)
 (2.0) + (-0.0)
 (3.0) + (-0.0)
 (4.0) + (-0.0)
 (5.0) + (-0.0)
 (6.0) + (0.0)
 (7.0) + (0.0)
 (8.0) + (0.0)
```

Aim2: FFT computation and storing the values in a .dat file.

```
#include<stdio.h>
#include<math.h>
#include <dsk6713.h>
#define N 8
#define PI 3.14
struct cmpx
{
float real,imag;
};
int main()
{
    int i;
    typedef struct cmpx complex;
    complex w[N];
    printf("The entered sequence is:\n ");
    printf("x[8]={1,0.0,2,0.0,3,0.0,4,0.0,5,0.0,6,0.0,7,0.0,8,0.0} \n");
    complex x[8]={1,0.0,2,0.0,3,0.0,4,0.0,5,0.0,6,0.0,7,0.0,8,0.0};
    complex temp1,temp2;
    int j,k,upper_leg,lower_leg,leg_diff,index,step;
    for(i=0;i<N;i++)
    {
    w[i].real=cos((2*PI*i)/(N*2.0));
    w[i].imag=-sin((2*PI*i)/(N*2.0));
    }
    leg_diff=N/2;
    step=2;
    for(i=0;i<3;i++)
    {
        index=0;
        for(j=0;j<leg_diff;j++)
        {
            for(upper_leg=j;upper_leg<N;upper_leg+=(2*leg_diff))
```

```
            {
            lower_leg=upper_leg+leg_diff;
            temp1.real=(x[upper_leg]).real+(x[lower_leg]).real;
            temp1.imag=(x[upper_leg]).imag+(x[lower_leg]).imag;
            temp2.real=(x[upper_leg]).real-(x[lower_leg]).real;
            temp2.imag=(x[upper_leg]).imag-(x[lower_leg]).imag;
            (x[lower_leg]).real=temp2.real*(w[index]).real-
                                temp2.imag*(w[index]).imag;
            (x[lower_leg]).imag=temp2.real*(w[index]).imag+
                                temp2.imag*(w[index]).real;
            (x[upper_leg]).real=temp1.real;
            (x[upper_leg]).imag=temp1.imag;
            }
            index+=step;
        }
        leg_diff=(leg_diff)/2;
        step=step*2;
    }
    j=0;
    for(i=1;i<(N-1);i++)
    {
        k=N/2;
        while(k<=j)
        {
        j=j-k;
        k=k/2;
        }
        j=j+k;
        if(i<j)
        {
        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;
        }
    }

    //*******storing in a file ***********
      FILE *fp1;
      fp1 = fopen("FFT_storage.dat","wb");//open for binary write
    //write N floats in array to FFT_storage.dat
      for(i= 0; i < N; i++)
        {
          fwrite(&x[i],sizeof(struct cmpx), 1, fp1);
        }
      fclose(fp1);
```

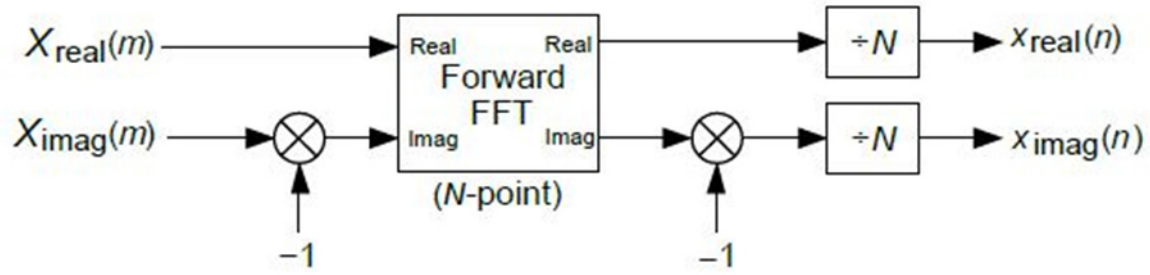
```
      DSK6713_waitusec(100000);

    // printing the values
      printf("The fft of the given input sequence is \n");
      for(i=0;i<N;i++)
        {
        printf("(%0.2f) + j(%0.2f) \n",(x[i]).real,(x[i]).imag);
        }

   //**********reading the value from storage location**************
      FILE *fp2;
      fp2 = fopen("FFT_storage.dat","rb");//open file for binary read
      printf("The stored sequence is \n");
      for(i = 0; i < 8; i++)
        {
         fread(&x[i], sizeof(struct cmpx), 1, fp2);
         printf(" %0.2f \t  %0.2f\n",(x[i]).real,(x[i]).imag);
        }
      fclose(fp2);
}
```

Output:

```
The entered sequence is:
x[8]={1,0.0,2,0.0,3,0.0,4,0.0,5,0.0,6,0.0,7,0.0,8,0.0}
The fft of the given input sequence is
(36.00) + j(0.00)
(-4.01) + j(9.66)
(-4.00) + j(4.00)
(-4.01) + j(1.65)
(-4.00) + j(0.00)
(-4.00) + j(-1.66)
(-4.00) + j(-4.00)
(-3.99) + j(-9.65)
The stored sequence is
 36.00    0.00
 -4.01    9.66
 -4.00    4.00
 -4.01    1.65
 -4.00    0.00
 -4.00    -1.66
 -4.00    -4.00
 -3.99    -9.65
```

Experiment 6. IFFT with FFT

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: We will be using the .dat file created in previous experiment to demonstrate the IFFT.



The following program demonstrate the above block diagram

```
#include<stdio.h>
#include<math.h>
#include <dsk6713.h>
#define N 8
#define PI 3.141
struct cmpx
{
float real,imag;
};
int main()
{
    int i;
    typedef struct cmpx complex;
        complex w[N];
        complex x[N];
        complex temp1,temp2;
//**********Reading the value from storage location**************
    FILE *fp2;
    fp2 = fopen("FFT_storage.dat","rb");//open file for binary read
    printf("The stored sequence is \n");
    for(i = 0; i < 8; i++)
        {
          fread(& x[i], sizeof(struct cmpx), 1, fp2);
          printf(" %0.2f \t  %0.2f\n", x[i].real, x[i].imag);
        }
    int j,k,upper_leg,lower_leg,leg_diff,index,step;
```

```
//**********printing the modified sequence*************
    printf("The modified sequence is \n");
    for(i=0;i<N;i++)
    {
        x[i].real =  x[i].real;
        x[i].imag = -1*( x[i].imag);
        printf(" %0.2f \t  %0.2f\n", x[i].real, x[i].imag);
    }
    for(i=0;i<N;i++)
    {
        w[i].real=cos((2*PI*i)/(N*2.0));
        w[i].imag=-sin((2*PI*i)/(N*2.0));
    }
    leg_diff=N/2;
    step=2;
    for(i=0;i<3;i++)
    {
    index=0;
    for(j=0;j<leg_diff;j++)
    {
        for(upper_leg=j;upper_leg<N;upper_leg+=(2*leg_diff))
        {
            lower_leg=upper_leg+leg_diff;
            temp1.real=(x[upper_leg]).real+(x[lower_leg]).real;
            temp1.imag=(x[upper_leg]).imag+(x[lower_leg]).imag;
            temp2.real=(x[upper_leg]).real-(x[lower_leg]).real;
            temp2.imag=(x[upper_leg]).imag-(x[lower_leg]).imag;
            (x[lower_leg]).real=temp2.real*(w[index]).real-
                                temp2.imag*(w[index]).imag;
            (x[lower_leg]).imag=temp2.real*(w[index]).imag+
                                temp2.imag*(w[index]).real;
            (x[upper_leg]).real=temp1.real;
            (x[upper_leg]).imag=temp1.imag;
        }
    index+=step;
    }
    leg_diff = (leg_diff)/2;
    step=step*2;
    }
    j=0;
    for(i=1;i<(N);i++)
    {
        k=N/2;
        while(k<=j)
        {
            j=j-k;
            k=k/2;
```

```
        }
        j=j+k;
        if(i<j)
        {
            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;
        }
    }
    for(i=0;i<(N);i++)
    {
        (x[i]).real=((x[i]).real)/N;
        (x[i]).imag=-1*((x[i]).imag)/N;
    }

    DSK6713_waitusec(100000);
        // printing the values
    printf("The Ifft of the given input sequence is \n");
        for(i=0;i<8;i++)
            {
              printf("(%0.2f) + j(%0.2f) \n",(x[i]).real,(x[i]).imag);
            }
}
```

Output:

```
The stored sequence is
 36.00    0.00
 -4.01    9.66
 -4.00    4.00
 -4.01    1.65
 -4.00    0.00
 -4.00    -1.66
 -4.00    -4.00
 -3.99    -9.65
The modified sequence is
 36.00    0.00
 -4.01    -9.66
 -4.00    -4.00
 -4.01    -1.65
 -4.00    0.00
 -4.00    1.66
 -4.00    4.00
 -3.99    9.65
```

```
The Ifft of the given input sequence is
(1.00) + j(0.00)
(2.00) + j(-0.00)
(3.00) + j(-0.00)
(4.00) + j(-0.00)
(5.00) + j(-0.00)
(6.00) + j(0.00)
(7.00) + j(0.00)
(8.00) + j(0.00)
```

Experiment 7. FIR low pass filter

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<math.h>
#include<dsk6713_aic23.h>
#include<dsk6713.h>
#include<dsk6713_led.h>
#include<dsk6713_dip.h>
#define L 11//length of filter
int main()

{
    Uint32 sample_pair;
    float pi = 3.14;
    float hamming[L],hanning[L], h[L]= { 0 },x[L] = { 0 },y=0.0;
    int i;
    //cut-off frequency of filter in rad/sample
    float wc = pi / 4.0;
    DSK6713_init();
    DSK6713_LED_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)
    {
        //compute filter coeffs USING HAMMING WINDOW
            if(DSK6713_DIP_get(1)==0)
                    {
                        DSK6713_LED_on(1);
                        //Generate Hamming window sequence
                      for (i = 0; i < L; i++)
                        {
                        hamming[i] = 0.54 - 0.46 * cos(2 * pi * i / (L - 1));
                            //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];
                        }
```

```
                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
                for (i = 0; i < L; i++)
                {
                     y += h[i] * x[i];//compute filter output
                }
                //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));
            }
        //compute filter coeffs USING HANNING WINDOW
        else if(DSK6713_DIP_get(2)==0)
            {
                DSK6713_LED_on(2);
                //Generate Hanning window sequence
                  for (i = 0; i < L; i++)
                  {
                   hanning[i] = 0.5 - 0.5 * cos(2 * pi * i / (L - 1));
                      //avoid division by 0 when i=(L-1)/2
                    if (i == (L - 1) / 2)
                          h[i] = wc / pi * hanning[i];
                    else
                          h[i] = sin(wc * (i - (L - 1) / 2.0)) /
                   (pi*(i - (L - 1)/2.0)) * hanning[i];
                  }
                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++)
                          y += h[i] * x[i];//compute filter output
                //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));
            }
```

```
            else if(DSK6713_DIP_get(3)==0)
                    {
                       DSK6713_LED_on(3);
                       while (!DSK6713_AIC23_read(hCodec, &sample_pair));
              //store top-half of sample from codec in x[0]
                        y = (int)sample_pair >>16;
                        sample_pair = (int)y <<16;
                         while (!DSK6713_AIC23_write(hCodec, sample_pair));
                     }
            else
                {
                    DSK6713_LED_on(1);
                    DSK6713_LED_on(2);
                    DSK6713_LED_on(3);
                    DSK6713_waitusec(20000);
                    DSK6713_LED_off(1);
                    DSK6713_LED_off(2);
                    DSK6713_LED_off(3);
                    DSK6713_waitusec(20000);
                }
    }
}
```

Experiment 8. Overlap Save Block Convolution

The overlap-save procedure cuts the signal up into equal length segments with some overlap. Then it takes the DFT of the segments and saves the parts of the convolution that correspond to the circular convolution. Because there are overlapping sections, it is like the input is copied therefore there is not lost information in throwing away parts of the linear convolution.

```
%Program for computing Block Convolution using Overlap Add Method
close all;
clear all;
x=input('Enter the sequence x(n) = ');
h=input('Enter the sequence h(n) = ');
g=conv(x,h);
disp('output using inbuilt function');
disp(g);
l1=length(x);
l2=length(h);
N = l1+l2-1;
x = [x zeros(1,N-l1)];
h1 = [h zeros(1,l2-1)];
l3 = length(h1);
y = zeros(1,N+l3-l2);
H = fft(h1);
for i=1:l2:l1
  if i<=l1
  x1=[x(i:i+l3-l2) zeros(1,l3-l2)];
  else
  x1=[x(i:l1) zeros(1,l3-l2)];
  end
  x2=fft(x1);
  x3=x2.*H;
  x4=round(ifft(x3));

  if (i==1)
      y(1:l3)=x4(1:l3);
  else
      y(i:i+l3-1)=y(i:i+l3-1)+x4(1:l3);
  end
end
disp('The output sequence y(n)=');
disp(y(1:N));
subplot(3,1,1);stem(x);xlabel('Time ');ylabel('Amplitude ');
title('First sequence ');
subplot(3,1,2);stem(h);xlabel('Time ');ylabel('Amplitude ');
title('Second sequence ');
subplot(3,1,3);stem((y(1:N)));xlabel('Time ');ylabel('Amplitude ');
title('Overlap Add Method ');
```

Output:

Enter the sequence x(n) = [1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1]

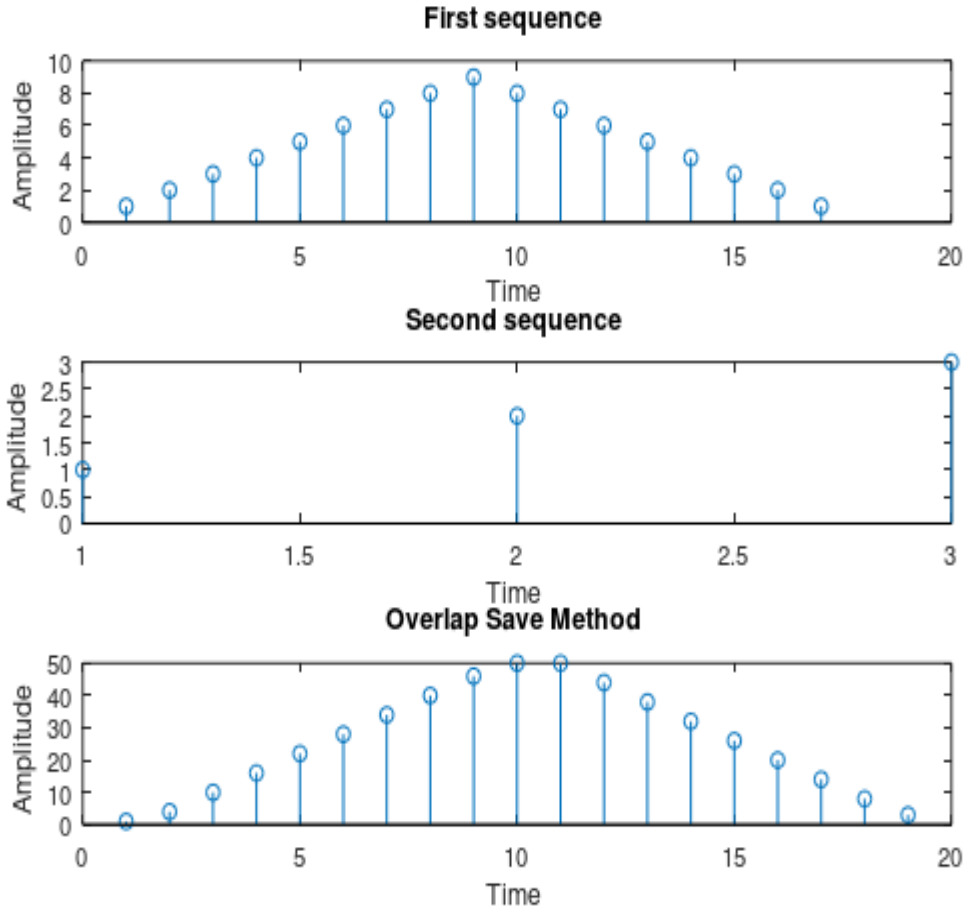
Enter the sequence h(n) = [1 2 3]

output using inbuilt function

1 4 10 16 22 28 34 40 46 50 50 44 38 32 26 20 14 8 3

The output sequence y(n)=

1 4 10 16 22 28 34 40 46 50 50 44 38 32 26 20 14 8 3



Experiment 9. Overlap Add Block Convolution

The overlap-add procedure cuts the signal up into equal length segments with no overlap. Then it zero-pads the segments and takes the DFT of the segments. Part of the convolution result corresponds to the circular convolution. The tails that do not correspond to the circular convolution are added to the adjoining tail of the previous and subsequent sequence. This addition results in the aliasing that occurs in circular convolution.

```
%Program for computing Block Convolution using Overlap Add Method
close all;
clear all;
x=input('Enter the sequence x(n) = ');
h=input('Enter the sequence h(n) = ');
g=conv(x,h);
disp('output using inbuilt function');
disp(g);
l1=length(x);
l2=length(h);
N = l1+l2-1;
x = [x zeros(1,N-l1)];
h1 = [h zeros(1,l2-1)];
l3 = length(h1);
y = zeros(1,N+l3-l2);
H = fft(h1);
for i=1:l2:l1
  if i<=l1
  x1=[x(i:i+l3-l2) zeros(1,l3-l2)];
  else
  x1=[x(i:l1) zeros(1,l3-l2)];
  end
  x2=fft(x1);
  x3=x2.*H;
  x4=round(ifft(x3));

  if (i==1)
      y(1:l3)=x4(1:l3);
  else
      y(i:i+l3-1)=y(i:i+l3-1)+x4(1:l3);
  end
end
disp('The output sequence y(n)=');
disp(y(1:N));
subplot(3,1,1);stem(x);xlabel('Time ');ylabel('Amplitude ');
title('First sequence ');
subplot(3,1,2);stem(h);xlabel('Time ');ylabel('Amplitude ');
title('Second sequence ');
```

```
subplot(3,1,3);stem((y(1:N)));xlabel('Time ');ylabel('Amplitude ');
title('Overlap Add Method ');
```

Output:

Enter the sequence x(n) = [1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1]

Enter the sequence h(n) = [1 2 3]

output using inbuilt function

1 4 10 16 22 28 34 40 46 50 50 44 38 32 26 20 14 8 3

The output sequence y(n)=

1 4 10 16 22 28 34 40 46 50 50 44 38 32 26 20 14 8 3