EXPERIMENT 1

1(a) AUTO AND POWER SPECTRUM

```
% Set seed for reproducibility
rng(42);

% Generate two random signals
signal_length = 1000;
signal_1 = randn(1, signal_length);
signal_2 = randn(1, signal_length);

% Compute autocorrelation
autocorr_result = xcorr(signal_1, signal_2);

% Plot autocorrelation
lags = -(signal_length - 1):(signal_length - 1);
figure;
plot(lags, autocorr_result);
title('Autocorrelation of Random Signals');
xlabel('Lag');
ylabel('Autocorrelation');
grid on;

% Compute and plot power spectral density
[psd, frequencies] = pwelch(signal_1, [], [], [], 1);
figure;
semilogy(frequencies, psd);
title('Power Spectral Density of Signal 1');
xlabel('Frequency (Hz)');
ylabel('Power/Frequency (dB/Hz)');
grid on;
[psd, frequencies] = pwelch(signal_2, [], [], [], 1);
figure;
semilogy(frequencies, psd);
title('Power Spectral Density of Signal 2');
xlabel('Frequency (Hz)');
ylabel('Power/Frequency (dB/Hz)');
grid on;
```

1(b) LOWPASS AND BANDPASS RANDOM PROCESS

```
% Parameters
Fs = 1000;            % Sampling frequency (Hz)
T = 1/Fs;             % Sampling period
t = 0:T:1;            % Time vector
```

```
L = length(t);          % Length of signal
f_cutoff_low = 50;      % Lowpass cutoff frequency (Hz)
f_center = 200;         % Center frequency for bandpass (Hz)
f_width = 50;           % Width of bandpass (Hz)

% Generate lowpass random process
x_lowpass = randn(1, L); % Gaussian white noise
[b, a] = butter(6, f_cutoff_low/(Fs/2)); % Design Butterworth filter
x_lowpass = filter(b, a, x_lowpass); % Apply lowpass filter

% Generate bandpass random process
x_bandpass = randn(1, L); % Gaussian white noise
f_low = f_center - f_width/2;
f_high = f_center + f_width/2;
[b, a] = butter(6, [f_low/(Fs/2), f_high/(Fs/2)], 'bandpass'); % Design Butterworth bandpass filter
x_bandpass = filter(b, a, x_bandpass); % Apply bandpass filter

% Plotting
figure;
subplot(2,1,1);
plot(t, x_lowpass);
title('Lowpass Random Process');
xlabel('Time (s)');
ylabel('Amplitude');
subplot(2,1,2);
plot(t, x_bandpass);
title('Bandpass Random Process');
xlabel('Time (s)');
ylabel('Amplitude');
```

<u>EXPERIMENT 2 CENTRAL LIMIT THEOREM</u>

```
% Experiment 2 (CLT)
% Parameters
population_size = 10000; num_samples = 1000;
sample_size = 30;
% Defining the population array
population = exprnd(2, 1, population_size);
sample_means = zeros(1, num_samples);
% For loop to iteratively compute the sample means and append to the array for i = 1 :
num_samples
for i=1:num_samples
sample = randsample(population, sample_size, true);
sample_means(i) = mean(sample);
```

```
end
% Plotting the population
figure(1);
subplot(2, 1, 1);
histogram(population, 'Normalization', 'pdf', 'EdgeColor','none'); title("Histogram of the
population");
xlabel("value");
ylabel("probability density");
subplot(2, 1, 2);
histogram(sample_means, 'Normalization', 'pdf', 'EdgeColor','none'); title("Sample means of
population");
xlabel("sample"); ylabel("Sample means");
% Deriving the PDF of the sample means and plotting them
mu_population = mean(population); sigma_population = std(population); expected_mean =
mean(sample_means);
expected_std = sigma_population/sqrt(num_samples);
x = linspace(min(sample_means), max(sample_means), 100);
y = normpdf(x, expected_mean, expected_std);
figure(2);
plot(x, y, 'r', 'LineWidth', 2); xlabel("sample mean");
ylabel("probability value"); title("PDF of sample means");
```

EXPERIMENT 3 LOWPASS SAMPLING THEOREM TIME DOMAIN

```
% Sampling theorem in time domain
tfinal = 0.01;
t = 0 : 0.0001 : 0.01;
xanalog = cos(2*pi*400*t) + cos(2*pi*700*t); subplot(4, 1, 1);
plot(t, xanalog, 'r-'); xlabel("time");
ylabel("amplitude"); title("analog signal");
% critical sampling (fs = 2*fm)
fs = 1400;
tsamp = 0 : 1/fs : tfinal;
xsampled = cos(2*pi*400*tsamp) + cos(2*pi*700*tsamp); subplot(4, 1, 2);
plot(tsamp, xsampled, 'b*-');
xlabel("time");
ylabel("amplitude");
title("Critical sampling");
% under sampling (fs < 2*fm)
fs = 1400;
tsamp = 0 : 1/fs : tfinal;
xsampled = cos(2*pi*400*tsamp) + cos(2*pi*700*tsamp); subplot(4, 1, 3);
plot(tsamp, xsampled, 'b*-');
xlabel("time");
ylabel("amplitude");
```

```
title("Under sampling");
% over sampling (fs > 2*fm)
fs = 2000;
tsamp = 0 : 1/fs : tfinal;
xsampled = cos(2*pi*400*tsamp) + cos(2*pi*700*tsamp); subplot(4, 1, 4);
plot(tsamp, xsampled, 'b*-');
xlabel("time");
ylabel("amplitude");
title("Over sampling");
```

FREQUENCY DOMAIN

```
% Sampling theorem in frequency domain
tfinal = 0.01;
t = 0 : 0.00001 : tfinal;
xanalog = cos(2*pi*400*t) + cos(2*pi*700*t);
%plotting the analog signal
figure;
subplot(4, 1, 1); plot(t, xanalog); xlabel("time"); ylabel("amplitude"); title("Analog signal");
% Critical sampling (fs = 2*fm)
fs = 1400;
tsamp = 0 : 1/fs : 13/fs;
xsampled = cos(2*pi*400*tsamp) + cos(2*pi*700*tsamp); xsampled_DFT = abs(fft(xsampled));
xsampled_length = 0 : (length(xsampled_DFT) - 1);
subplot(4, 1, 2);
stem(100 * xsampled_length, xsampled_DFT); xlabel("frequency");
ylabel("magnitude"); title("Critical sampling");
xreconstructed = ifft(fft(xsampled)); subplot(4, 1, 3);
plot(tsamp, xreconstructed, "b*-"); xlabel("time");
ylabel("amplitude");
title("Critical sampling");
% Under sampling (fs < 2*fm)
fs = 700;
tsamp = 0 : 1/fs : 6/fs;
xsampled = cos(2*pi*400*tsamp) + cos(2*pi*700*tsamp); xsampled_DFT = abs(fft(xsampled));
xsampled_length = 0 : (length(xsampled_DFT) - 1);
subplot(4, 1, 4);
stem(100 * xsampled_length, xsampled_DFT); xlabel("frequency");
ylabel("magnitude"); title("Under sampling");
xreconstructed = ifft(fft(xsampled));
figure;
subplot(4, 1, 1); plot(t, xanalog); xlabel("time"); ylabel("amplitude");
title("analog signal (400Hz + 700 Hz)");
subplot(4, 1, 2);
```

```
plot(tsamp, xreconstructed, "b*-"); xlabel("time");
ylabel("amplitude");
title("Under sampling (fs < 2*fm)");
% Over sampling (fs > 2*fm)
fs = 2000;
tsamp = 0 : 1/fs : 19/fs;
xsampled = cos(2*pi*400*tsamp) + cos(2*pi*700*tsamp); xsampled_DFT = abs(fft(xsampled));
xsampled_length = 0 : (length(xsampled_DFT) - 1);
subplot(4, 1, 3);
stem(100 * xsampled_length, xsampled_DFT); xlabel("frequency");
ylabel("magnitude"); title("Over sampling");
xreconstructed = ifft(fft(xsampled)); subplot(4, 1, 4);
plot(tsamp, xreconstructed, "b*-"); xlabel("Time");
ylabel("Amplitude");
title("Oversampling (fs > 2fm)");
```

EXPERIMENT 5 PCM FOR QUANTIZATION
UNIFORM

```
clear all;
clc;
t=[0:0.01:10];
a=sin(t);
[sqnr8,aquan8]=u_pcm(a,8);
[sqnr16,aquan16]=u_pcm(a,16);
display('sqnr8');
display('sqnr16');
plot(t,a,'-',t,aquan16,'-',t,zeros(1,length(t)));
legend('Original Signal','8 level quantized signal','16 level quantized signal');
function[sqnr,a_quan]=u_pcm(a,n)
amax=max(abs(a));
a_quan=a/amax;
d=2/n;
q=d.*[0:n-1];
q=q-((n-1)/2)*d;
for i=1:n
a_quan(find((q(i)-d/2<=a_quan)&(a_quan<=q(i)+d/2)))=q(i).*ones(1,length(find((q(i)-d/2<=a_qua
n)&(a_quan<=q(i)+d/2))));
end
a_quan=a_quan*amax;
nu=ceil(log2(n));
code=zeros(length(a),nu);
sqnr=20*log10(norm(a)/norm(a-a_quan));
```

```
end
```

NON UNIFORM

```
clear all;
clc;
t=[0:0.01:10];
a=sin(t);
[sqnr,aquan,code]=mula_pcm(a,16,255);
display('sqnr');
plot(t,a,'-',t,aquan,'-');
function[sqnr,a_quan,code]=u_pcm(a,n)
amax=max(abs(a));
a_quan=a/amax;
d=2/n;
q=d.*[0:n-1];
q=q-((n-1)/2)*d;
for i=1:n
a_quan(find((q(i)-d/2<=a_quan)&(a_quan<=q(i)+d/2)))=q(i).*ones(1,length(find((q(i)-d/2<=a_quan)&(a_quan<=q(i)+d/2))));
end
a_quan=a_quan*amax;
nu=ceil(log2(n));
code=zeros(length(a),nu);
sqnr=20*log10(norm(a)/norm(a-a_quan));
end
function[sqnr,a_quan,code]=mula_pcm(a,n,mu)
[y,maximum]=mulaw(a,mu);
[sqnr,y_q,code]=u_pcm(y,n);
a_quan=invmulaw(y_q,mu);
q_quan=maximum*a_quan;
sqnr=20*log10(norm(a)/norm(a-a_quan));
end
function[y,a]=mulaw(x,mu)
a=max(abs(x));
y=(log(1+mu*abs(x/a))./log(1+mu)).*sign(x);
end
function x=invmulaw(y,mu)
x=(((1+mu).^(abs(y))-1)./mu).*sign(y);
end
```

EXPERIMENT 6
DELTA MODULATION

```
clc;
clear all;
```

```
close all;
a=2;
t=0:2*pi/50:2*pi; % Signal Generation
x=a*sin(t);
l=length(x);
plot(x,'r');
delta=0.2;
%delta1=2*delta;%Apply delta modulation with doubling the step size
%delta2=3*delta;
hold on
xn=0;
for i=1:l;
if x(i)>xn(i)
d(i)=1;
xn(i+1)=xn(i)+delta;
else
d(i)=0; xn(i+1)=xn(i)-delta;
end
end
stairs(xn)
hold on
legend('Analog signal','DM with step size=0.2')
title('DELTA MODULATION')
```

ADAPTIVE MODULATION

```
close all
clear all
clc
td = 0.01;
ts = 0.02;
t = 0:td:5;
x = 8*sin(2*pi*t);
delta = 0.1;
figure(1)
plot(t,x);
ADMout = adeltamod(x,delta,td,ts);
figure(2)
plot(t,ADMout);
```

Function for a deltamod

```
%The working of the Advanced Delta Modulator is similar to the regular
% Delta Modulator. The only difference is that the amplitude step
% size is variable and it keeps getting doubled if the previous output/s
% don't seem to 'catch up' with the input signal. This problem is
% referred to as 'Slope overload' in textbooks.
function [ADMout] = adeltamod(sig_in, Delta, td, ts)
```

```
% Usage
% ADMout = adeltamod(sig_in, Delta, fs);
% Delta -- min. step size. This will be multiplied 2nX if required
% sig_in -- the signal input, should be a vector
% td -- the original sampling period of the input signal, sig_in
% ts -- the required sampling period for ADM output. Note that it
% should be an integral multiple of the input signal's period.
% If not, it will be rounded up to the nearest integer.
% Function output: ADMout
 if (round(ts/td) >= 2)
 Nfac = round(ts/td); %Nearest integer
 xsig = downsample(sig_in,Nfac);
 Lxsig = length(xsig);
 Lsig_in = length(sig_in);

 ADMout = zeros(Lsig_in); %Initialising output

 cnt1 = 0; %Counters for no. of previous consecutively increasing
 cnt2 = 0; %steps
 sum = 0;
 for i=1:Lxsig

 if (xsig(i) == sum)
 elseif (xsig(i) > sum)
 if (cnt1 < 2)
 sum = sum + Delta; %Step up by Delta, same as in DM
 elseif (cnt1 == 2)
 sum = sum + 2*Delta; %Double the step size after
 %first two increase
 elseif (cnt1 == 3)
 sum = sum + 4*Delta; %Double step size
 else
 sum = sum + 8*Delta; %Still double and then stop
 %doubling thereon
 end
 if (sum < xsig(i))
 cnt1 = cnt1 + 1;
 else
 cnt1 = 0;
 end
 else
 if (cnt2 < 2)
 sum = sum - Delta;
 elseif (cnt2 == 2)
```

```
 sum = sum - 2*Delta;
 elseif (cnt2 == 3)
 sum = sum - 4*Delta;
 else
 sum = sum - 8*Delta;
 end
 if (sum > xsig(i))
 cnt2 = cnt2 + 1;
 else
 cnt2 = 0;
 end
 end
 ADMout(((i-1)*Nfac + 1):(i*Nfac)) = sum;
 end
 end
end
```

EXPERIMENT 7 SIGMA DELTA

```
clc
clear all
close all
t = -5:0.01:5; %basic time axis
f =2;
w = 2*pi*f;
osr = 250; %can vary
fs1 = w/pi;
fs = fs1*osr;
%% sampling time
ts = -5:(1/fs):5; %sampling times are defined
y = @(t)sin(w.*t); %signal is defined
%% sigma delta quantisation
[u,q] = SDQ(y(ts),ts);
%% reconstruction algorithm
z = 0;
for k = 1:length(ts)
z = z + q(k).*sinc(w.*(t - ts(k)));
end
c = max(y(t))./max(z); %scaling is done as a consequence of oversampling
z = z.*c;
%% figures
figure(1)
subplot(3,1,1)
plot(t,y(t),'linewidth',2)
title('Original signal')
```

```
xlabel('Time')
ylabel('Amplitude')
subplot(3,1,2)
plot(ts,q)
title('SDQ signal');
xlabel('Time');
ylabel('Amplitude');
subplot(3,1,3)
plot(t,z,'linewidth',2);
title('Reconstructed signal');
xlabel('Time');
ylabel('Amplitude');
subtitle('ab')
figure(2);
plot(t,y(t),'linewidth',2)
hold on
plot(t,z,'linewidth',2);
title('Original vs Reconstructed');
subtitle('ab')

figure(3);
plot(abs(z - y(t)),'linewidth',2);
title('Error');
subtitle('ab')

figure(4);
subplot(3,1,1);
plot(abs(fftshift(fft(y(t)))));
xlabel('Frequency');
ylabel('Amplitude');
title('Spectrum of original signal');
subplot(3,1,2);
plot(abs(fftshift(fft(q))));
xlabel('Frequency');
ylabel('Amplitude');
title('Spectrum of SDQ');
subplot(3,1,3);
plot(abs(fftshift(fft(z))));
title('Spectrum of recovered signal');
xlabel('Frequency');
ylabel('Amplitude')
subtitle('ab')
%% mse computation
error = immse(z,y(t));
```

```
%% function

function [u,q] = SDQ(y,t)
%as per basic equations, models a sigma delta modulator
%% code logic
q = zeros(1,length(t));
u = zeros(1,length(t)); %quantizaton noise/state variable
u(1) = 0.9; %taken 0.9 as in between 0 and 1 for stability (non inclusive)
%recursive equations for SDQ
for k = 2:length(t)
q(k) = sign(u(k-1) + y(k));
u(k) = u(k-1) + y(k) - q(k);
end
end
```

EXPERIMENT 8

LINE CODES

```
%Input parameters
N= 10; % Number of input bits
a=floor(2*rand (1,N)) % generates random 1's and zero's and displays
A=5; % Pulse amplitude
Tb=1; %bit period
fs=100; % Number of samples (even number) taken in a bitperiod
%Unipolar NRZ
U=[];
for k=1:N;
U = [U A*a(k)*ones(1,fs)];
end
%Unipolar RZ
U_rz=[];
for k=1:N;
 c = ones(1,fs/2);
 b = zeros(1,fs/2);
 p = [c b];
 U_rz = [U_rz A*a(k)*p];
end
%Polar NRZ
P=[];
for k=1:N
P = [P ((-1)^(a(k) + 1))*A*ones(1,fs)];
end
%Polar RZ
```

```
P_rz=[];
for k = 1:N
 c = ones(1,fs/2);
 b = zeros(1,fs/2);
 p = [c b];
 P_rz = [P_rz ((-1)^(a(k)+1))*A*p];
end
%Bipolar NRZ
B=[];
count=-1;
for k= 1:N
 if a(k)==1
 if count==-1
 B = [B A*a(k)*ones(1,fs)];
 count=1;
 else
 B = [B -A*a(k)*ones(1,fs)];
 count=-1;
 end
 else
 B = [B A*a(k)*ones(1,fs)];
 end
end

%Bipolar RZ / AMI RZ
B_rz=[];
count=-1;
for k= 1:N
 if a(k)==1
 if count==-1
 B_rz = [B_rz A*a(k)*ones(1,fs/2) zeros(1,fs/2)];
 count=1;
 else
 B_rz = [B_rz -A*a(k)*ones(1,fs/2) zeros(1,fs/2)];
 count=-1;
 end
 else
 B_rz = [B_rz A*a(k)*ones(1,fs)];
 end
end

%Split-phase or Manchester code
M=[];
 for k = 1:N
```

```
 c = ones(1,fs/2);
 b = -1*ones(1,fs/2);
 p = [c b];
 M = [M ((-1)^(a(k)+1))*A*p];
 end

T = linspace(0,N*Tb, length(U));% Time vector % Lengths of all codes are same
figure(1)
subplot(4, 1, 1); plot(T,U,'LineWidth',2)
axis([0 N*Tb -6 6])
title('Unipolar NRZ')
grid on
subplot(4, 1, 2); plot(T,U_rz,'LineWidth',2)
axis([0 N*Tb -6 6])
title('Unipolar RZ')
grid on
subplot(4, 1, 3); plot(T,P,'LineWidth',2)
axis([0 N*Tb -6 6])
title('Polar NRZ')
grid on
subplot(4, 1, 4); plot(T,P_rz,'LineWidth',2)
axis([0 N*Tb -6 6])
title('Polar RZ')
grid on
figure(2)
subplot(3, 1, 1); plot(T,B,'LineWidth',2)
axis([0 N*Tb -6 6])
title('Bipolar NRZ')
grid on
subplot(3, 1, 2); plot(T,B_rz,'LineWidth',2)
axis([0 N*Tb -6 6])
title('Bipolar RZ / RZ-AMI')
grid on
subplot(3, 1, 3); plot(T,M,'LineWidth',2)
axis([0 N*Tb -6 6])
title('Split-phase or Manchester code')
grid on
```

PSD OF LINE CODES

```
v=1; % voltage level of a bit
R=1; % Bitrate
T=1/R; % Bit period
```

```
f=0:0.001*R:2*R; % frequency vector in terms of bit rate
f= f+1e-10; % Otherwise, sin(0)/0 is undefined
% PSD curves are plotted for Bitrate=1bps and Pulse amplitude=1V
%Unipolar NRZ
s=((v^2*T/4).*(sin(pi.*f*T)./(pi.*f*T)).^2);
s(1)=s(1)+(v^2/4);% corresponds to an impulse function of weight v^2/4 at f=0 added to s(f) at
f=0;
ff=0;
stem(ff,s(1),'*r','LineWidth',4)% sketching an impulse at f=0
hold on;
plot(f,s,'-r','LineWidth',2);
hold on;
%Manchester code
s=(v.^2.*T).*((sin(pi.*f*T/2)./(pi.*f*T/2)).^2).*(sin(pi.*f*T/2).^2);
plot(f,s,'--g','LineWidth',2);
hold on;
%Polar NRZ
s=((v^2*T).*(sin(pi.*f*T)./(pi.*f*T)).^2);
plot(f,s,'--b','LineWidth',2);
hold on;
%Bipolar RZ
s=(v.^2.*T/4).*((sin(pi.*f*T/2)./(pi.*f*T/2)).^2).*(sin(pi.*f*T).^2);
plot(f,s,'--k','LineWidth',2);
legend('Unipolar NRZ: impulse at at f=0','Unipolar NRZ', 'Manchestercode','PolarNRZ','Bipolar
RZ/ RZ-AMI');
xlabel('Normalized frequency)');
ylabel('Power spectral density');
```

PROBABILITY OF ERROR

```
%The probability of error, for
%equally likely data, with additive white Gaussian noise (AWGN) and matched filter
%Unipolar NRZ
E=[0:1:25]; % Eb/N0=SNR of the recieved signal
%Unipolar NRZ
P1=(1/2)*erfc(sqrt(E/2));
%polar NRZ and Manchester code has same Pe for equiprobable 1's and 0's
P2=(1/2)*erfc(sqrt(E));
%Bipolar RZ/ RZ-AMI
P3=(3/4)*erfc(sqrt(E/2));
E=10*log10(E); % SNR in dB
semilogy(E,P1,'-k',E,P2,'-r',E,P3,'-b','LineWidth',2)
```

```
legend('Unipolar NRZ','Polar NRZ and Manchester','Bipolar RZ/ RZ-AMI','Location','best');
xlabel('SNR per bit, Eb/No(dB)');
ylabel('Bit error probability Pe');
```

EXPERIMENT 9 FIR

```
#include "dsk6416_aic23.h"
Uint32 fs=DSK6416_AIC23_FREQ_8KHZ;
#define DSK6416_AIC23_INPUT_MIC 0x0015
#define DSK6416_AIC23_INPUT_LINE 0X0011
Uint16 inputsource=DSK6416_AIC23_INPUT_LINE;
#include<math.h>
static short in_buffer[100];
Uint32 sample_data;
short k=0;
float filter_coeff[]={ -0.0017 ,  -0.0020 ,  -0.0024  , -0.0027 ,  -0.0021  ,  0.0000  ,  0.0044  ,
0.0117 ,   0.0221  ,  0.0351 ,   0.0500  ,  0.0655,  0.0799 ,   0.0917   , 0.0994  ,  0.1021 ,   0.0994
,   0.0917 ,   0.0799  ,  0.0655  ,  0.0500  ,  0.0351  ,  0.0221,   0.0117, 0.0044  ,  0.0000  ,
-0.0021 ,  -0.0027  , -0.0024 ,  -0.0020 ,  -0.0017};

short l_input,r_input,l_output,r_output;
void comm_intr();
void output_left_sample(short);
short input_left_sample();
signed int FIR_FILTER(float *h,signed int);
interrupt void c_int11()
{
	l_input=input_left_sample();
	l_output=(Int16)FIR_FILTER(filter_coeff,l_input);
	output_left_sample(l_output);
	return;
}
signed int FIR_FILTER(float *h,signed int x)
{
	int i=0;
	signed long output=0;
	in_buffer[0]=x;
	for(i=31;i>0;i--)
		in_buffer[i]=in_buffer[i-1];
	for(i=0;i<31;i++)
```

```
                    output=output+h[i]*in_buffer[i];
          return(output);
}
void main()
{
          comm_intr();
          while(1);
}
```

EXPERIMENT 10 IIR

```
#include”DSK6713_AIC23”; //codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;//set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE;
#include”bs1800int.cof”;
short input_left_sample();
void output_left_sample(short);
void comm_intr();
short w[NUM_SECTIONS][2] = {0};
interrupt void c_int11() //interrupt service routine

{
short section; // index for section number
short input; // input to each section
int wn,yn; // intermediate and output values in each stage
input = input_left_sample();
for (section=0 ; section< NUM_SECTIONS ; section++)
{
wn = input - ((a[section][0]*w[section][0])&>>;15) - ((a[section][1]*w[section][1])&>>15);
yn = ((b[section][0]*wn)&>>15) + ((b[section][1]*w[section][0])&>>15) +
((b[section][2]*w[section][1])&>>15);
w[section][1] = w[section][0];
w[section][0] = wn;
input = yn; // output of current section will be input to next
}
output_left_sample((short)(yn)); // before writing to codec
return; //return from ISR
}
void main()
{
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```