Graphic Equalizer

Shubhang Bhatnagar Mat No-N1903718K

18 November, 2019

1 Introduction

The goal of this project is to design a graphic equalizer system using the TMS320C5515 eZDSP kit. An equalizer is a system which can specifically apply a gain/attenuation different frequency bands of the input. A graphical equalizer is an equalizer which can be controlled through a graphical interface. They are widely used in music recording studios and Hi-Fi systems to remove the effect of the rooms acoustics on the signal. They are also used in forensics and other applications where only certain frequency bands might need to be amplified and others suppressed.
The project aims to design a graphic equalizer using a DSP by digitally filtering and applying gains and attenuation to different bands using the digital filters. Such a digitally implemented equalizer system is usually much more versatile and cheaper than an analog one. The challenge in designing such a system is that the filtering should be done in real time and the quantization noise and reconstruction error due to the digitisation should be well within limits.

2 Design and construction

Humans have a range of hearing from around 20 Hz- 20 kHz. My equalizer system divides this into 3 different bands whose gains we can independently vary.Also, sampling frequency of 48 kHz, s specified in the assignment, is chosen as this would satisfy the Nyquist rate for all the frequencies in the human hearing range comfortably and would require lesser computation (and real time issues) than when choosing a higher sampling frequency.
We divide the hearing range into 3 bands whose gain can be controlled independently. The ranges of the chosen bands and the filters used to isolate them are-

- Band 1- Low frequency band- 0-5 kHz -low pass filter
- Band 2- Middle band- 5-12 kHz Band pass filter
- Band 3- High frequency band- 12 kHz-24 kHz($= \frac{Sampling frequency}{2}$) High pass filter

2.1 Filter Design

The required digital filters for isolating the bands are designed using MATLAB's fdatool.FIR filters are chosen in place of IIR for our system, as they do not distort the input signals phase. The specifications for each filter are-

- Max passband ripple= 1dB

- Minimum stopband attenuation= 60dB
- Order of filter=255

The stop band and the transition band width is adjusted to get a filter meeting these specifications.

2.2 Interface design

We have 2 switches to enable the user to interact with the system. Switch 1 is used to select the Bypass mode, where no filtering is applied and simply the input signal = output signal. Switch 2 selects the Equaliser mode and enables a prompt to the user for entering the respective gain and attenuation. 25 values of gain, varying from -12 dB to +12 dB (step size of 1 dB) can be chosen for each band by the user. Also, an LED lghts up indicating the band for which you should enter the gain.
A blue LED light indicates the Bypass mode being enabled, while the 3 red, green and yellow LEDs being on together indicate that the 3 bands are being processed. The interaction with the switch is enabled through the SAR (successive approximation register), which is initialized when the system starts. Also, appropriate code is written to enable the correct behaviour as stated above on pressing of the switches. The LED's are controlled using the memory mapped GPIO registers. The memory addresses to which the GPIO registers are mapped to are summarized in table1-

Register Name	Memory mapping	Function
GPIO Data out Register 1	0x1C0A	Control Blue and Yellow LED
GPIO Data out Register 2	0x1C0B	Control Red and Green LED
GPIO Direction Register 1	0x1C06	Set Blue and Yellow LED as output
GPIO Direction Register 2	0x1C07	Set Red and Green LED as output

Table 1: MMIO mappings and functions

The LEDs are appropriately controlled and initialised by setting the corresponding bits in thee registers correctly.

2.3 Applying gain and merging the filters

We now have 3 filters which need to be applied to the signal separately. Then their outputs need to be scaled and combined back again to get the complete output (having all 3 bands scaled correctly). This would be very inefficient in terms of the amount of computation it requires. The FIR assembly function would need to be called 3 times on the same input signal for applying these filters. Also, the outputs would then again need to be multiplied by the appropriate scaling factors.
Instead, we exploit the linearity of the FIR filters (in z domain) and create one "Effective" filter from the 3 FIR filters and their respective gains chosen. The scaling of the output of one filter is equivalent to scaling the filters coefficients by the same amount (due to linearity of FIR).So we calculate the effective filter coefficients and then filter the input only once, with a filter having the desired response characteristics.

$$Effectivecoef = gain\ LPF \times coef\ LPF + gain\ BPF \times coef\ BPF + gain\ HPF \times coef\ HPF$$

Also, before such an multiplication is done, the gains are appropriately converted from dB scale to a linear scale using a look up table.
Additionally, while combining and scaling filters, there is a risk of overflow. So we scale the down coefficients appropriately to avoid overflow. We later scale up the output by the same amount to compensate for the earlier scale down.

2.4 The actual filtering

The filtering is done by the library assembly function FIR given by TI. It takes as an input the new sample/s (1 at a time or a block depending on it's arguments) and the coefficients and gives the filtered signals the output. It uses the processors internal MAC to do the filtering and is highly optimised.

3 Results

The magnitude and phase responses of the filters designed are as follows-

3.1 Low pass Filter

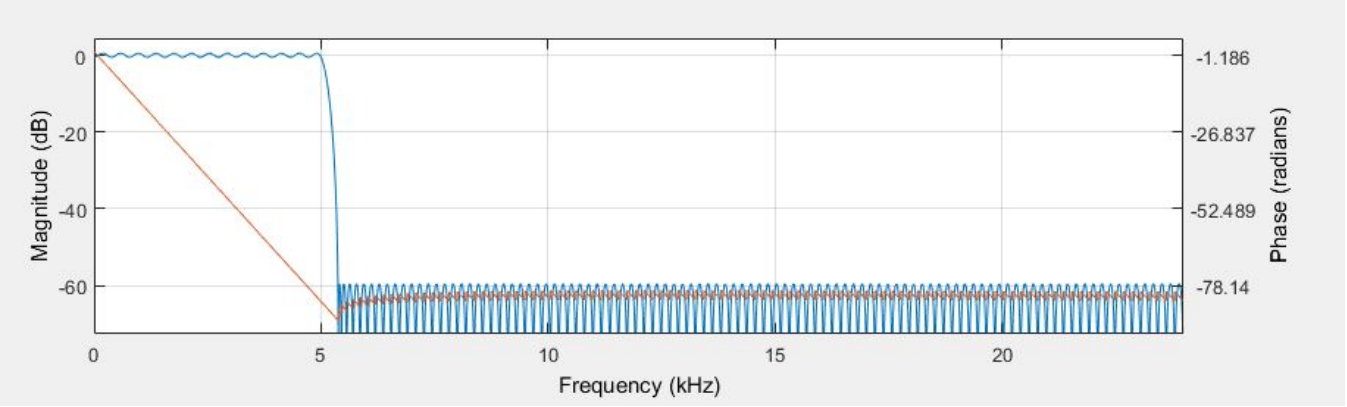

Figure 1: Magnitude and phase response of the LPF

3.2 Band pass filter

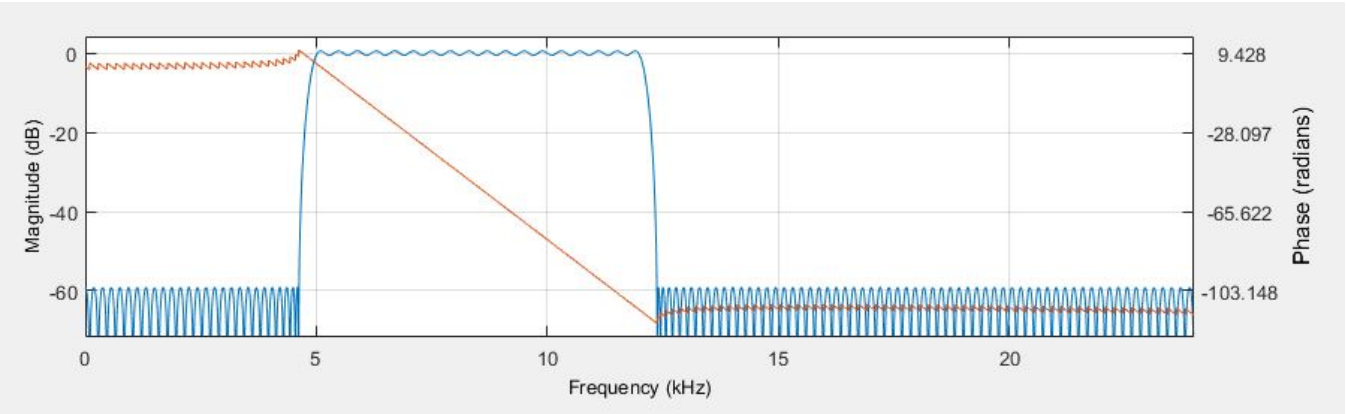

Figure 2: Magnitude and phase response of the LPF

3.3 High Pass filter

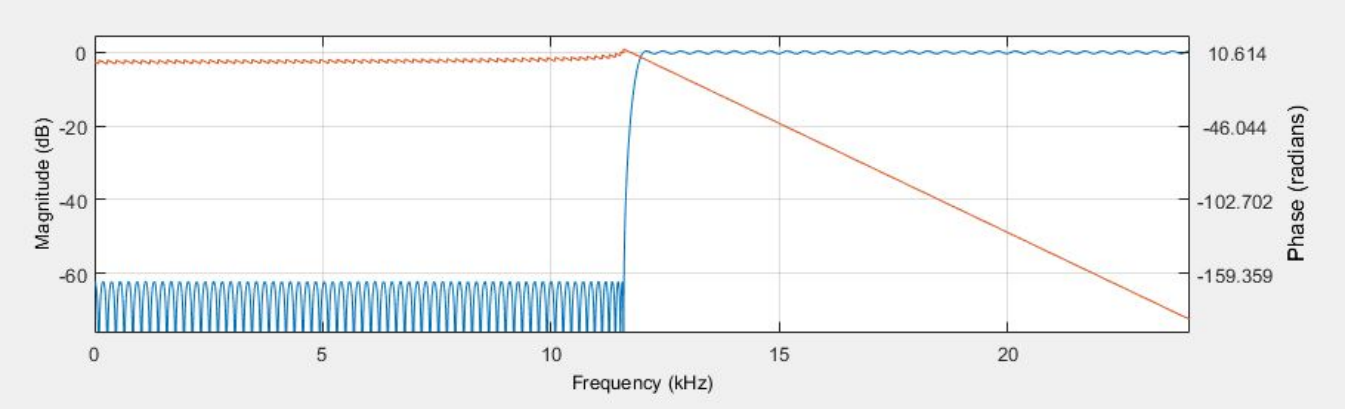

Figure 3: Magnitude and phase response of the LPF

3.4 MIPS computation

The code for the system is also profiled to check if it is running correctly in real time. It is done using the built in timer in the system. The delta time value we got from the profiling=312
Number of cycles=delta time $\times$ 2=624 cycles
Clock speed= 100 MHz Sampling frequency=48 kHz We assume 1 instruction is executed in 1 cycle.So, the required number of instructions for getting 1 output sample=624 instructions/sample. The maximum time in which 1 sample can be calculated= sampling period=$\frac{1}{F_s}$ where the sampling frequency F_s=48 kHz. So , the MIPS required=

$$MIPS = \frac{624 \times 48 \times 10^3}{10^6} = 29.95 MIPS$$

This is well below the rated value of 240 MIPS for the TMS320C5515 processor. So, we can believe that our algorithm works in real time.

4 Conclusion

The graphic equalizer hence designed works in real time and is capable of removing or amplifying the bass or treble from different inputs.Several important techniques were used to ensure that it can be done in real time, the most important one of them being the technique used to combine multiple filters into one. Also, while combining the filters and applying the gain, we need to take into consideration the limited 16 bit size of the data. We must scale the coefficients properly to ensure there is no overflow while calculating the effective coefficients. The SAR and LEDs of the DSP also needed to be configured correctly to provide a good user interface.

5 Appendix

The C code for the equalizer implementation is appended for reference-

```
/*
 * Assignment 2 Shubhang N1903718K
 */

#include <usbstk5515.h>
#include <usbstk5515_i2c.h>
#include <AIC_func.h>
#include <stdio.h>
#include "firc.h"
#include <Dsplib.h>
#include <tms320.h>

#include<sar.h>
//#include "sar.h"




//Addresses of the MMIO for the GPIO out registers: 1,2
#define LED_OUT1 *((ioport volatile Uint16*) 0x1C0A  )
#define LED_OUT2 *((ioport volatile Uint16*) 0x1C0B )
//Addresses of the MMIO for the GPIO direction registers: 1,2
#define LED_DIR1 *((ioport volatile Uint16*) 0x1C06 )
#define LED_DIR2 *((ioport volatile Uint16*) 0x1C07 )

//LED_DIR- IODIR registers control whether GPIO pin is used as input or output.
//LED_OUT1- IODATAOUT registers control if output is 1 or 0 if GPIO set in output mod
//pay attention to numbering direction and convention in manual
//Toggles LED specified by index Range 0 to 3
// see C5515 GPIO guide especially section 1 and 3


Int16 coefBand1[TAPS1] = {
#include "band1.dat"
```

```
};
Int16 coefBand2[TAPS1] = {
#include "band2.dat"
};
Int16 coefBand3[TAPS1] = {
#include "band3.dat"
};

Int16 gainToLin[25]={0x080A,0x0904,0x0A1E,0x0B5A,0xCBD3,0x0E4A,0x1009,0x11FE,0x1460,0
// linear gains from dB -12 to 12-{0.2511,0.2818,0.3162,0.3548,0.3981,0.4466,0.5011,0
//1.1220,1.2589,1.4125,1.5848,1.7782,1.9952,2.2387,2.5118,2.8183,3.1622,3.5481,3.9810
// here in hex, everything is multiplied by 2^13 for scaling
int i1;
Int16 gain1,gain2,gain3;
Uint16 delta_time;
Uint16 start_time;
Uint16 end_time;
Uint16 bypass;
Uint16 val2;
Int16 g1,g2,g3;
void My_LED_init()
{
Uint16 temp=0x00;
Uint16 temp2=0x01;
temp |=(1<<14);
temp |=(Uint16)(1<<15);
LED_DIR1 |= temp; // set Yellow, Blue (14,15) as OUTPUT
temp2 |=(1<<1);
LED_DIR2 |= temp2; // set Red, Green (0,1) as OUTPUT

LED_OUT1 |= temp;  //Set LEDs 0, 1 to off
LED_OUT2 |= temp2;  //Set LEDs 2, 3 to off
}

void takeAction(int index)
{

//1022 no pressed
//only SW 1 pressed - 680
//only SW2 pressed- 508
//both pressed- 405
if(index==680)
{
bypass=1;
//Blue LED used to indicate bypass on 0 means on
LED_OUT1 = LED_OUT1 & (Uint16)(0xBFFF);//(1<<(bitGpio14));
```

```
//filtering LED's turned off
//yellow band 1
LED_OUT1 = LED_OUT1 | (Uint16)(0x8000);//(1<<(bitGpio15));

//Red band 2
LED_OUT2 = LED_OUT2 | (Uint16)(0x0001);//(1<<(bitGpio16));

//Green band 3
LED_OUT2 = LED_OUT2 | (Uint16)(0x0002);//(1<<(bitGpio17));

}
else if(index==508)
{

bypass=0;
//Blue LED for bypass turned off
LED_OUT1 = LED_OUT1 | (Uint16)(1<<14);//(1<<(bitGpio14));
printf("Enter the gain values (in dB) for the 3 bands\n");

//other band LEDs turned on
//yellow band 1
LED_OUT1 = LED_OUT1 & (Uint16)(0x7FFF);//(1<<(bitGpio15));
scanf("%d",&gain1);
gain1=gainToLin[gain1+12];

//Red band 2
LED_OUT2 = LED_OUT2 & (Uint16)(0xFFFE<<0);//(1<<(bitGpio16));
scanf("%d",&gain2);
gain2=gainToLin[gain2+12];

//Green band 3
LED_OUT2 = LED_OUT2 & (Uint16)(0xFFFD);//(1<<(bitGpio17));
scanf("%d",&gain3);
gain3=gainToLin[gain3+12];



//make new combined filter
for(i1=0;i1<256;i1++)
{
//all gains are in Q13 format. All coefs in Q15 format. We want no overflow after mul
// gain needs 13 coef needs 15
g1=(gain1>>10)*(coefBand1[i1]>>4);
g1=g1>>2;
g2=(gain2>>10)*(coefBand2[i1]>>4);
g2=g2>>2;
g3=(gain3>>10)*(coefBand3[i1]>>4);
g3=g3>>2;
```

```
coef[i1]=g1+g2+g3;

}

}

}

void main(void)
{



//Initialize variables and processor
USBSTK5515_init();  //Initializing the Processor
AIC_init();  //Initializing the Audio Codec

Uint16 value=0;
Uint16 i=0;
DATA right, left; //AIC inputs DATA is short
DATA out_right, out_left; //AIC output
DATA dbuffer_left;

TCR0 = TIME_STOP;
TCR0 = TIME_START;//Resets the time register

dbuffer_left=TAPS1+2;

gain1=0;
gain2=0;
gain3=0;

//SAR and LED initialized
Init_SAR();
My_LED_init();

//set default as bypass
bypass=1;
LED_OUT1 = LED_OUT1 & (Uint16)(0xBFFF);//(1<<(bitGpio14));
while(1)
{

value=(Uint16)Get_Sar_Key();

//check if witch presses and takeAction accordingly
takeAction(value);
```

```
//read input data
AIC_read2(&right, &left);

start_time=TIMCNT1_0;
if(bypass==1)
{
out_left=left;
out_right=right;
}
else
{
//call assembly function for filtering
fir(&right,//input
coef,//coef
&out_right,//output
&dbuffer_left,
1,
TAPS1//length of filter
);
fir(&left,//input
coef,//coef
&out_left,//output
&dbuffer_left,
1,
TAPS1//length of filter
);
//scale back output (compensate for coeffecient
// scaling in filter)
out_left=out_left<<3;
out_right=out_right<<3;
}
end_time=TIMCNT1_0;
delta_time=start_time-end_time;//error here in orignal... end time is less
AIC_write2(out_right,out_left);
}
}
```