ECE444: HW7

Thomas Smallarz

December 7, 2020

Contents

Introduction

The goal of this assignment is to design an audio equalizer and implement it onto our NXP FRDM-K22F development board. While the actual frequency response is up to the student, there are some requirements laid out for this design.

Design Requirements

- Frequency Response must be smooth (continuous) over the full range of (non-aliased) frequencies
- Phase Response must be linear (h[n] is finite in duration and symmetric or asymmetric about midpoint)
- FIR filter using Frequency Weighted Least Squares method
- Minimum filter order of $K = 20$

Implementation Requirements

- Filter should be designed so there is no overflow in the output
- Utilize one of the buttons to switch the filter on/off
- Represent coefficients and input/output data using 16 bits
- Fixed sampling rate of 20kHz

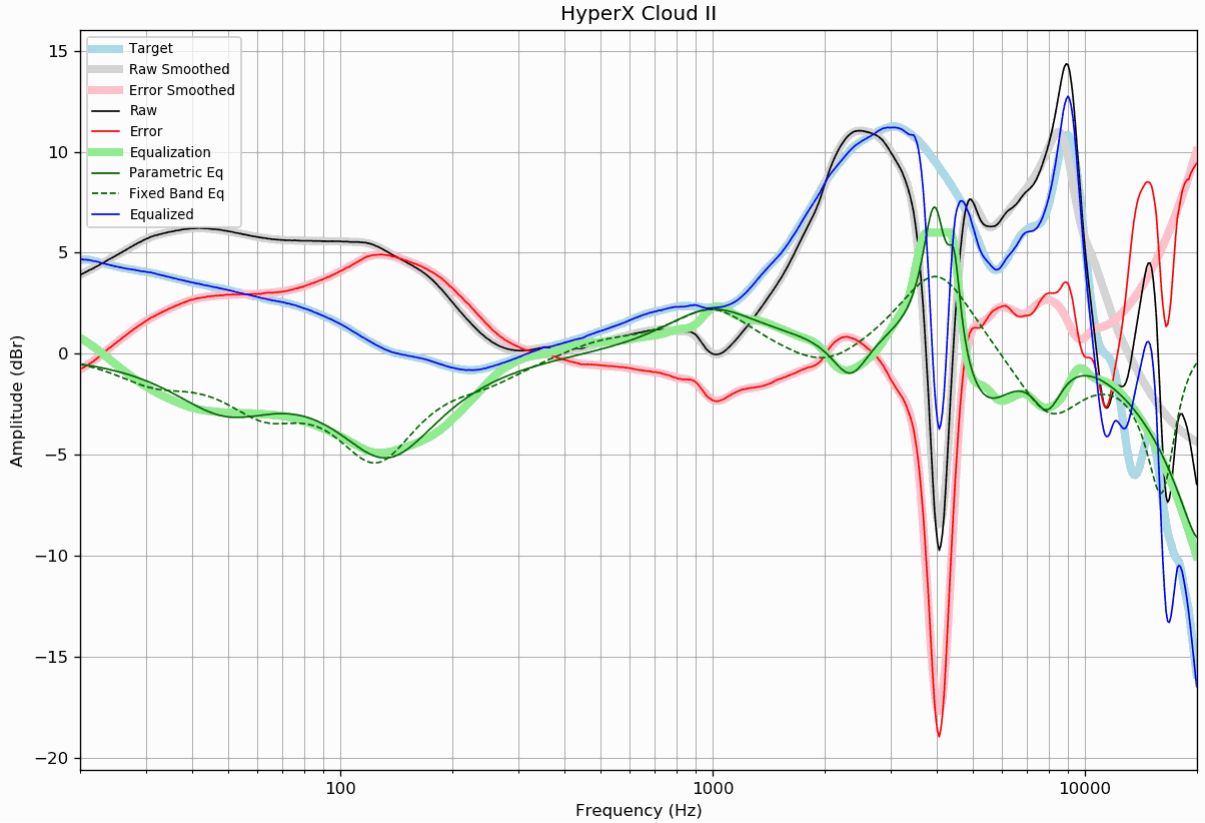
Design

Desired Frequency Response

While researching for typical frequency equalizer values used, I came across GitHub project called “AutoEq” started by Jaakko Pasanen. The idea of the project is to apply an equalizer to audio based on the pair of headphones you are using. The equalizer will “normalize” your headphones frequency response so that you are listening to what the audio producer intended when making the song.

This project has instructions for how to measure your headphones to determine how to normalize the audio. They currently have over 2,500 different audio devices in the “results”, and I happen to own one of the pairs that has been measured – the HyperX Cloud II.

Below is an image from the GitHub project for my headphones. The data for the “Equalization” values at each frequency is stored in an .csv file attached to project.



MATLAB

The first step is to import our data. Using MATLAB’s functions for importing .xls files we can import this data into two variables: frequency and equalization.

```
clear variables; close all;

Fs = 20e3; T = 1 / Fs;
cutoff_freq = 10e3; % 10kHz cutoff due to 20kHz fixed sampling rate

% ***********************************************
% ***** Importing and manipulating data.xls *****
% ***********************************************
% import data from .xls file
opts = detectImportOptions("data.xls");
labels = opts.VariableNames;
data = readmatrix("data.xls");
```

```
clear opts;

% cutoff any data that is above sampling frequency
[minValue,closestIndex] = min(abs(data(:,1) - cutoff_freq));
if(data(closestIndex,1) > cutoff_freq)
    closestIndex = closestIndex - 1;
end
data_cutoff = data([1:closestIndex],:);

clear closestIndex minValue;

% creating variables with necessary data (except for frequency...)
eval(labels(1) + " = data_cutoff(:,1);");
data_needed = [4 5 6];
for i = data_needed
    eval(labels(i) + " = db2mag(data_cutoff(:,i));");
end

clear data data_cutoff data_needed labels;
```

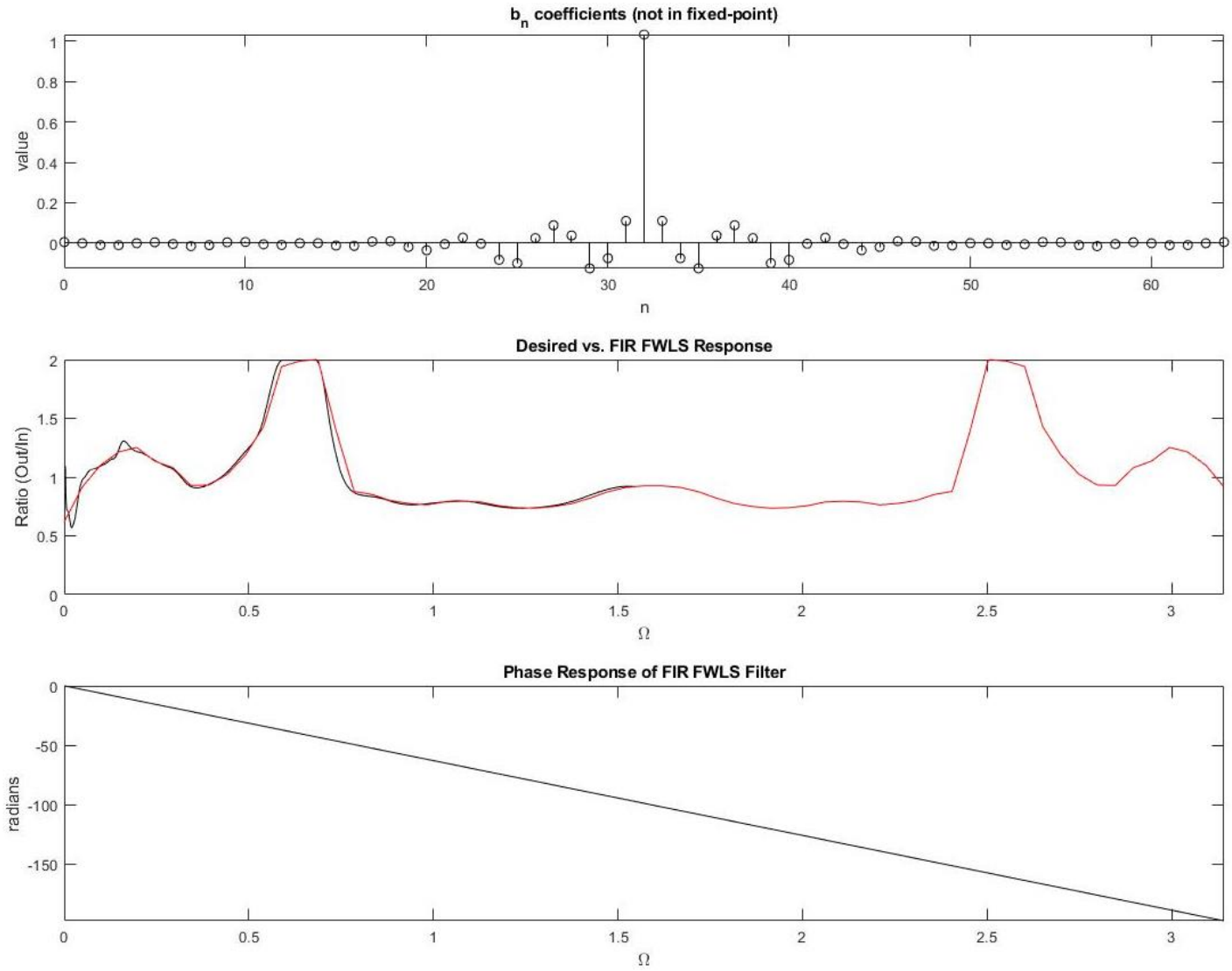
Now that we have all the frequency samples, we can use the built-in firls( ) function in MATLAB to generate b_0 coefficients for an FIR filter using the least-squares method. The desired order, frequency values, and amplitude of desired frequency at those frequencies are inputted, and the impulse response (b_0) is outputted.

```
% ************************************
% ***** Generating FIR FWLS filter *****
% ************************************
om=frequency'*2*T;
eq = equalization;
K = 64;
om(end) = []; eq(end) = [];
h = firls(K,om,eq);
```

Now plot frequency / phase response of coefficients generated to validate that the firls( ) function worked correctly.

```
% ********************
% ***** Plotting *****
% ********************
figure();
subplot(311); stem(0:length(h)-1,h,'k'); axis tight;
title("b_n coefficients (not in fixed-point)"); xlabel("n"); ylabel("value");
subplot(312); plot(om*pi/2,abs(eq),'k');
hold on; plot(linspace(0,pi,length(h)),abs(fft(h)),'r'); axis([0 pi 0 2]);
title("Desired vs. FIR FWLS Response"); xlabel("\Omega"); ylabel("Ratio (Out/In)");
subplot(313); plot(linspace(0,pi,length(h)),unwrap(angle(fft(h))),'k'); axis tight;
title("Phase Response of FIR FWLS Filter"); xlabel("\Omega"); ylabel("radians");
```

The top subplot is a the b_n coefficients. The middle subplot is the desired (black) vs. the actual (red) response of our outputted filter. The bottom subplot is the phase response of our actual filter. As we can see, our b_n coefficients are symmetrical and odd in number, and the phase response of our filter is linear.



Now we need to output our coefficients to a header file. I attempted to implement fixed-point arithmetic, but failed, so instead I used 32-bit floats.

```
% ****************************************
% ***** Generating coefficient header *****
% ****************************************
GenerateHeaderFloat(h);
```

```
function [] = GenerateHeaderFloat(B)
    fid = fopen('coef_f.h','w');

    fprintf(fid,'#define Lb %d // Length of B coef\n',length(B));
    fprintf(fid,'#define Lbuf %d // Length of buffer\n',length(B)-1);

    % ******************************
    % ***** CREATE B Coef Array *****
    % ******************************
    fprintf(fid,'float B[Lb] = {\n');
        for i = 1:length(B)
            if i == length(B) % last number to print
                fprintf(fid,'\t%.6f',B(i));
            elseif mod(i,5)==0 % last number in row
```

```
                fprintf(fid,'\t%.6f,\n',B(i));
            else % otherwise
                fprintf(fid,'\t%.6f,',B(i));
            end
        end
    fprintf(fid,'\n};\n\n');

    % ************************
    % ***** CREATE BUFFER *****
    % ************************
    fprintf(fid,'float buf[Lbuf] = { \n');
        for i = 1:length(B)-1
            if mod(i,10)==0
                fprintf(fid,'\t0,\n');
            elseif i == length(B)-1
                fprintf(fid,'\t0 ');
            else
                fprintf(fid,'\t0, ');
            end
        end
    fprintf(fid,'\n};\n');

    fclose(fid);
end
```

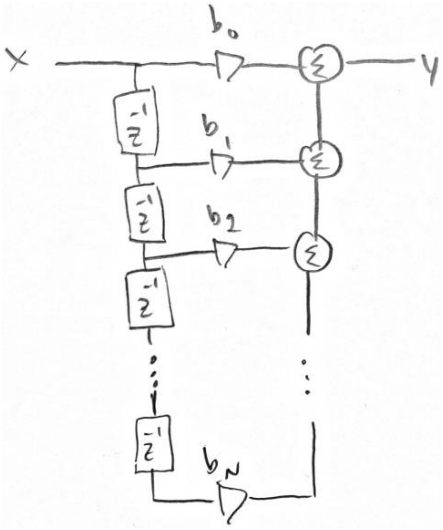
This gives us a coef_f.h file that looks like:

```
#define Lb 65 // Length of B coef
#define Lbuf 64 // Length of buffer
float B[Lb] = {
    0.005212,   -0.000393,  -0.009628,  -0.010105,  -0.000388,
    0.004326,   -0.004472,  -0.014884,  -0.009731,  0.004178,
    0.005796,   -0.005574,  -0.009016,  -0.000190,  0.000009,
    -0.011826,  -0.013338,  0.007920,   0.009983,   -0.018864,
    -0.035255,  -0.004906,  0.026978,   -0.002814,  -0.082539,
    -0.099353,  0.024649,   0.088491,   0.038021,   -0.124911,
    -0.074722,  0.110323,   1.032836,   0.110323,   -0.074722,
    -0.124911,  0.038021,   0.088491,   0.024649,   -0.099353,
    -0.082539,  -0.002814,  0.026978,   -0.004906,  -0.035255,
    -0.018864,  0.009983,   0.007920,   -0.013338,  -0.011826,
    0.000009,   -0.000190,  -0.009016,  -0.005574,  0.005796,
    0.004178,   -0.009731,  -0.014884,  -0.004472,  0.004326,
    -0.000388,  -0.010105,  -0.009628,  -0.000393,  0.005212
};

float buf[Lbuf] = {
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
    0,  0,  0,  0
};
```

Implementation

Realization Type

Out of the four types of realizations (DFI, DFII, TDFI, TDFII) we can immediately leave out DFI and TDFI as they are non-canonical. This leaves DFII and TDFII, which I chose DFII. This will look like:



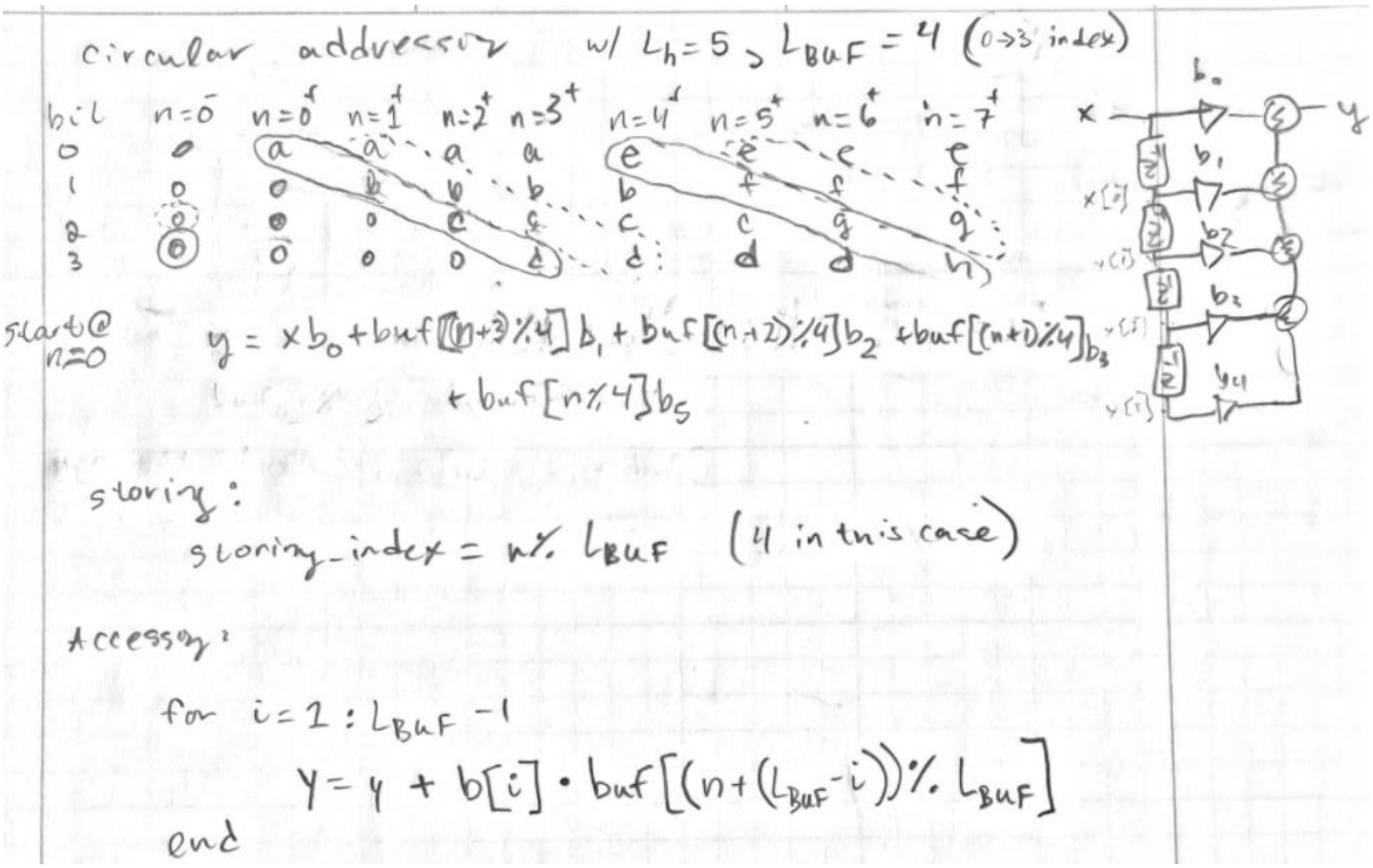
In pseudo code, the structure will look like:

```
X = ADC
Y = X * B[0]
for i = 1:N-1
   Y += Buf[i-1]*B[i];
end
```

Circular Addressing

In our pseudo code above, we would need to shift N-2 memory locations to add a new value to our buffer. Instead we want to use circular addressing to only replace one value, but instead change which locations are being indexed during the for-loop.

Below are the notes I took to realize a function for a rotating buffer:



From this we can see that to access during for-loop:

$$buf\left[\left(n + \left(L_{buf} - i\right)\right)\%L_{buf}\right]$$

And to store values in the buffer:

$$buf\left[n\%L_{buf}\right]$$

Programming Methods

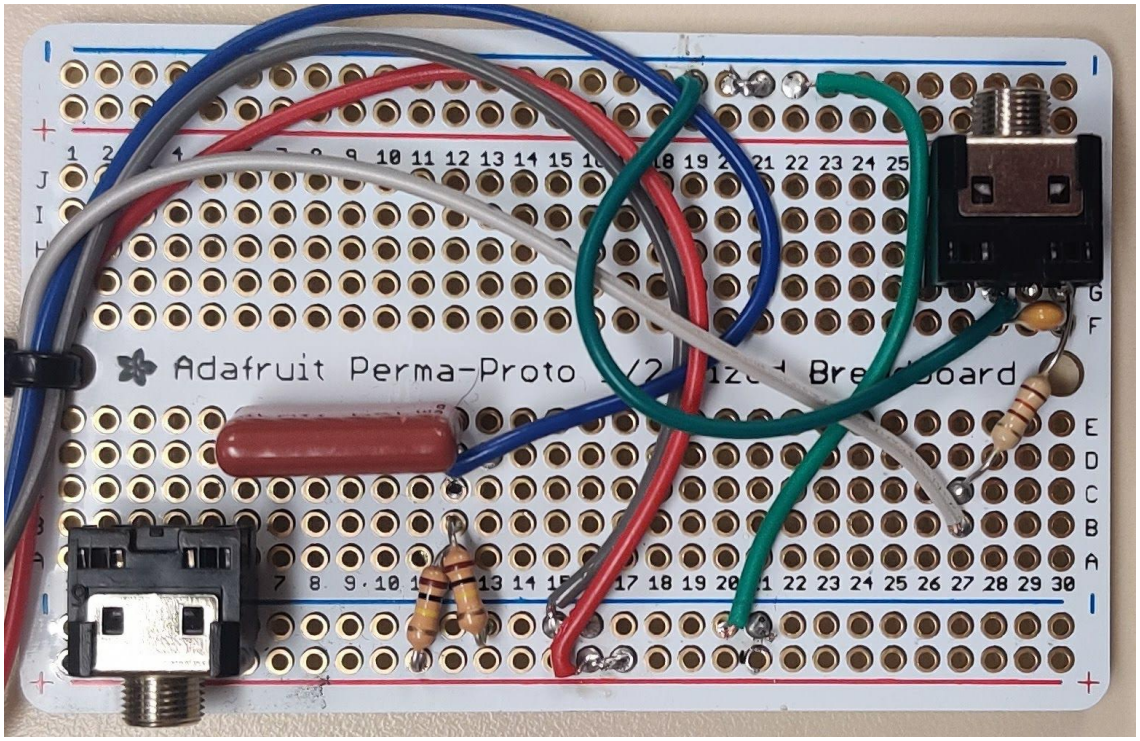
Below is the main.c code from Keil uVision5. There is nothing special about this. Inside the "PIT0_IRQHandler" is where the main interrupt is.

```
#include "MK22F51212.h"    //Device header
#include "MCG.h"                                    //Clock header
#include "TimerInt.h"                             //Timer Interrupt Header
#include "ADC.h"                                    //ADC Header

#include "DAC.h"                                    //DAC Header
#include "BUTTONS.h"                              //BUTTONS Header
#include "RGBLED.h"                               //RGB LED Header
#include "coef_f.h"                                 //Coefficients Header
#include <stdbool.h>                              //Boolean data types

uint16_t adc_meas = 0;
float x = 0;
float y = 0;
uint8_t i = 0;
uint8_t n = 0;
bool fs = false; // filter select: 0->digital wire, 1->FIR Filter

void PIT0_IRQHandler(void){       //This function is called when the timer interrupt expires
```

```
	GPIOA->PSOR |= GPIO_PSOR_PTSO(0x1u << 1); // Turn on Red LED
	ADC0->SC1[0] &= 0xE0;		//Start conversion of channel 1
	adc_meas = ADC0->R[0];  //Read channel 1 after conversion

	if(fs){ //if filter select is not set to digital wire
		// first stage
		x = (((float)adc_meas)-((float)2048));
		y = B[0]*x;

		// rest of stages
		for(i=1;i<Lbuf;i++){
			y += B[i]*(buf[(n+(Lbuf-i))%Lbuf]);
		}

		buf[n%(Lbuf)] = x; // store new measurement into circular buffer

		adc_meas = (uint16_t)(y+2048);
	}

	// may need to move this into if statement
	n = (n+1)&(Lbuf-1); // masking index

	DAC0->DAT[0].DATL = DAC_DATL_DATA0(adc_meas & 0xFFu); //Set Lower 8 bits of Output
	DAC0->DAT[0].DATH = DAC_DATH_DATA1((adc_meas >> 0x8)&0xFFu); //Set Upper 8 bits of Output

	NVIC_ClearPendingIRQ(PIT0_IRQn); //Clears interrupt flag in NVIC Register
	PIT->CHANNEL[0].TFLG	= PIT_TFLG_TIF_MASK; //Clears interrupt flag in PIT Register


	GPIOA->PCOR |= GPIO_PCOR_PTCO(0x1u << 1); // Red LED = 0
}

// K++ BUTTON
void PORTB_IRQHandler(void){ //This function might be called when the SW3 is pushed
	if(fs){ //if filter select is equal to 0 (digital wire)
		fs = false; //set equal to zero (loop back around to lowest starting passband freq)
	}else{fs=true;}

	// clear buffer just to be safe?
	for(i=0;i<Lbuf;i++){buf[i]=0;}

	NVIC_ClearPendingIRQ(PORTB_IRQn);	//CMSIS Function to clear pending interrupts on
PORTB
	PORTB->ISFR = (0x1u << 17);					//clears PORTB ISFR flag
}

int main(void){
	RGBLED_Init();
	BUTTONS_Init();
	MCG_Clock120_Init();
	ADC_Init();
	DAC_Init();
	TimerInt_Init();
	while(1){

	}
}
```

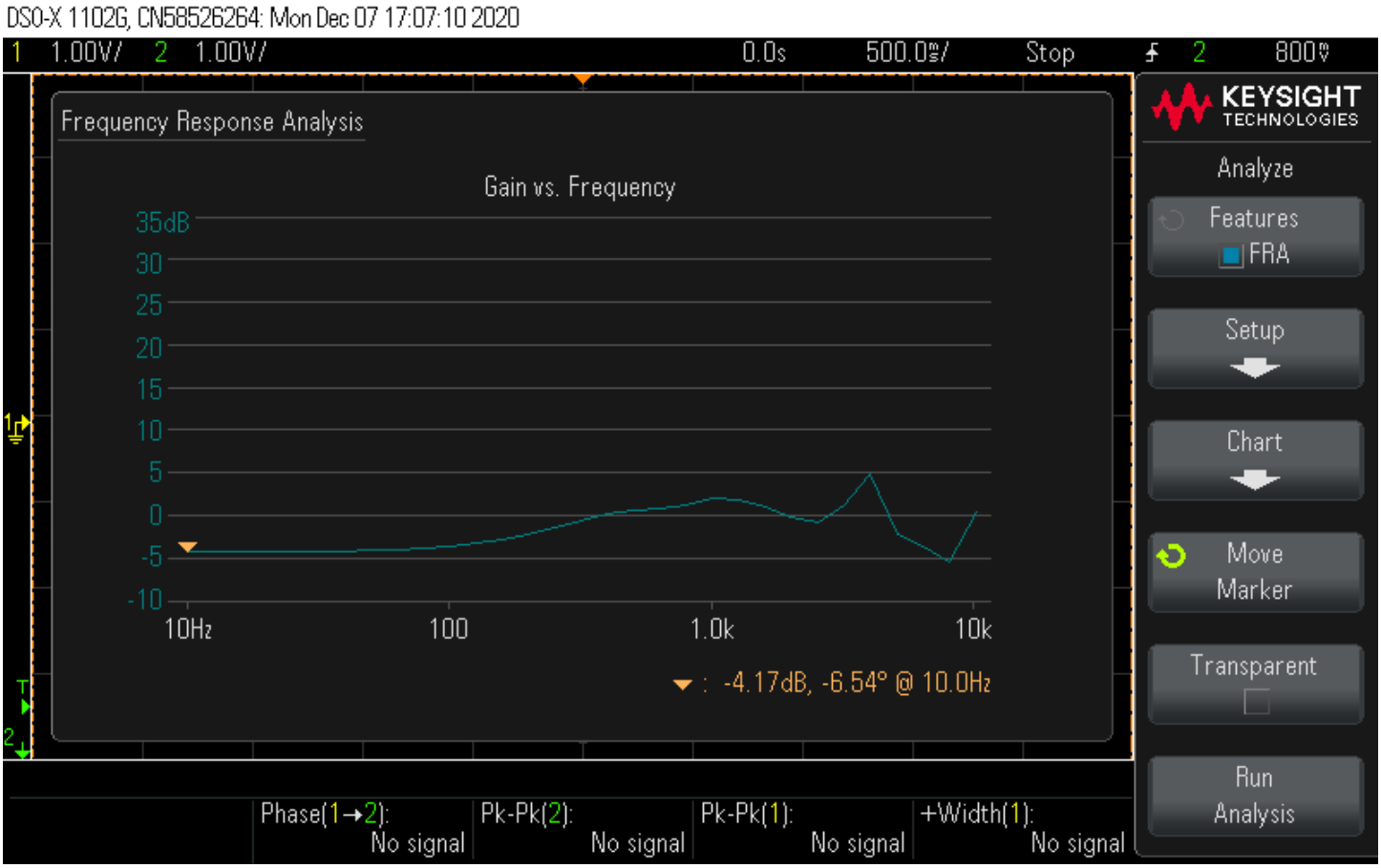
Hardware

The same hardware used in HW4 was used in this. Below is an image: The input is on the lower-left, the output on the upper-right. The DAC directly drives the headphones with a 120Ω resistor in series to avoid sourcing more current from DAC pin than it is capable of.



Testing

When playing music, it is slightly noticeable when changing from “digital wire” to FIR FWLS filter. There is some noise on the output either way, so maybe that distracts from the difference. Performing a Frequency Response Analysis on the oscilloscope I got the following frequency response (note that it is plotted on a log-based scale whereas our frequency response earlier was plotted linearly).



This is shaped very similarly to our desired frequency response, so I consider that a win.

Checking timing on our red LED that is set up to turn on at the beginning of the interrupt and turn off at the end we can see that we are making timing by just under $\sim 10\mu s$.