

**EQUAZLIER**

Group Name: Nothin' Yet

Members: Zac Idriss, Zach Williamson, Clayton Covalt

Contents

[Executive Summary 2](#_Toc481783013)

[IIR Filter - BiQuad 2](#_Toc481783014)

[Coefficient Computation 2](#_Toc481783015)

[Problem Statement 3](#_Toc481783016)

[Proposed Solution 3](#_Toc481783017)

[Implementation Details 3](#_Toc481783018)

[Initialization 3](#_Toc481783019)

[FLEXCOMM7 – BiQuad 4](#_Toc481783020)

[Audio Output to aux CODEX 5](#_Toc481783021)

[BiQuad Filter Function (Architecture) 6](#_Toc481783022)

[Coefficient Computation Function (Architecture) 6](#_Toc481783023)

[Main Function 7](#_Toc481783024)

[Coefficients and States Initialization 7](#_Toc481783025)

[Calculate RMS for band Display 7](#_Toc481783026)

[Display 8](#_Toc481783027)

[Band Display Architecture 8](#_Toc481783028)

[Conclusion/Results 9](#_Toc481783029)

[Appendices 10](#_Toc481783030)

[Appendix 1 – BiQuad Filter Diagram 10](#_Toc481783031)

[Appendix 2 – Initialize buffers for the storage of data 10](#_Toc481783032)

[Appendix 3 - Tuners 10](#_Toc481783033)

[Appendix 4 – Sliders 11](#_Toc481783034)

# Executive Summary

This project will be implemented using an LPCXpresso54608 board and is intended to be a real time digital signal processing (DSP) project. The goal is to create an Equalizer and input an analog signal, using the audio aux CODEC and output the filtered signal to a pair of either headphones or speakers. The signal will be parsed through parallel IIR filters, where each corresponding filter will be individually adjusted for intensity level using the touchscreen. Each band will be shown on the screen, with the level of intensity at which that frequency is found within the incoming signal. This will be the equalizer part of the project. The real time digital signal processing (DSP) project will need a high sampling rate of 44.1 kHz to process the incoming data efficiently. The clock rate used is 24.576MHz, yielding an over sample rate (OSR) of approximately 557.

## IIR Filter - BiQuad

The equalizer will ultimately dictate which tones are pronounced and in which manner. The international standard for the division of frequency bands are as follows: 32Hz, 64Hz, 128Hz, 250Hz, 500Hz, 1 kHz, 2 kHz, 4 kHz, 8 kHz, and 16 kHz. Five bands were created using the cut offs of 250Hz, 500Hz, 1 kHz, 2 kHz, and 4 kHz. The selection of five out of 10 bands was due to a code optimization, which cases significant lag of both the bars and the sliders. The bandwidth of each band, and so filter will be the difference between each adjacent cut off frequency mentioned above (between *f* [*n*] and *f* [*n* - 1]). The digital filter used to divide the signal into different bands is the BiQuad filter i.e. a second order recursive IIR filter. To create a 10-band equalizer, the BiQuad IR filters will need to run in parallel, where the incoming signal through *filter* [*n*] will be filtered through the first, and *f* [ *n*+1] through the second and continued till *filter* [*n* = 10] simultaneously and then summed.

## Coefficient Computation

The coefficient computation necessary to implement the different filters will be done in real time, but slightly slower to ensure smooth operation as the CPU may not handle full real time computation. The coefficient computation will be based on known BiQuad filter equations.

The screen contains each band as a kind of histogram with the level is displayed. Each band, and thus tone, is adjusted using a slider. A slider per band is used, which is directly correlated to the outgoing signal, and so changing the mid, lows, and highs in as close to real time as possible.

# Problem Statement

To create an equalizer that accepts an inputs signal through an audio aux CODEC port and outputs the same signal, manipulated based on user input received via tactile feedback of the touchscreen on the LPCXpresso54608 board**.** The equalizer was to originally have 10 bands each of which are to be centred around 32, 64, 128, 250,500, 1000, 2000, 4000, 8000, 16000 (Hz) where 16 kHz is thought to be close enough to the limit of hearing – 20 kHz, and so a sampling rate of 44.1 kHz was used. The goal of the project is to not only control and output each band but to display each band in real time on the screen**.** Half the screen is to be used for the bands, while the other half for the sliders. The user will be able to parse the outgoing signal, based on the toggles seen on the screen. Therefore the user can adjust the bands, so accentuate the lows, the mids, and the highs based on their need to suit the kind of audio incoming.

# Proposed Solution

The proposed solution will use an IIR BiQuad filter, where each band will use two filters, one for the left and one for the right channel. These will be implemented in parallel. Since the signal comes into the receiving interrupt sample-by-sample, the computation occurs sample-by-sample too. Therefore, a buffer will be used, and capped at 1024 samples to be employed by the main function for manipulation and display to the screen. The slider values, which will manipulate the equalizer intensity for each band, will be passed back to the receiving interrupt to tune the outgoing signal, then sent to the transmit interrupt, flexcomm 6. This program will rely on the fact that the BiQuad filtering occurs directly within the interrupt. The receive interrupt, within which the filtering occurs, received the signal directly as a 32 Bit number, where the upper and lower 16 bits and the left and right channel respectively. The filter for each left and right channel is computed, and then plotted side by side on the Expresso board. The solution will center each left and right BiQuad filter around the frequency of a desired band, and this is done in parallel for (in the final product) five bands. Each band is adjusted by a slider that corresponds to the RMS value over 1024 samples.

# Implementation Details

The following section will contain snippets of the code used to implement both the equalizer, and the graphic library. It will outline, through examples of code, each portion of the equalizer software, and how each bar and slider was drawn.

## Initialization

The initiation of the coefficients and the states is twofold within the code. Figure [1] below shows the instantiation of both, as type defined structures. Rather than create multiple variables, for all the coefficients and states need for all 10 BiQuad filter (5 x 2 for left and right channels), a variable of the types IIR\_Coeffecnets, and StateData was initialized and then combined to create a coherent variable type that could be passed, and then computed upon when necessary. These two structures are combined to create one, and then variables of the type were created.

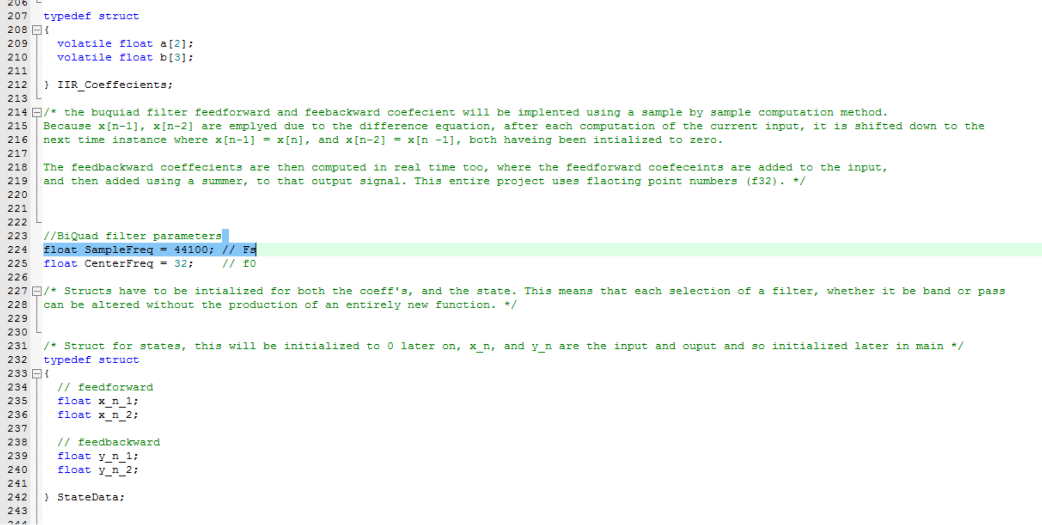
****

Figure : Coefficient and State Initialization

## FLEXCOMM7 – BiQuad

The receiving interrupt – through which the incoming signal is parsed – contains the bulk of the entire program. Within this interrupt, both the left and right channel are split and computed upon individually. As seen below there is a BiQuad filter function created within the program, which takes in three pointers, all of which are passed in the interrupt and of the type BiQuad parameters. BiQuad parameters is a structure created containing both the coefficient and state values. These are passed through as pointers, as their location resides outside the interrupt function. The respective channel is then passed as a pointer too, and is passed sample by sample. Therefore, the output of the BiQuad filter is sample by sample, and the interrupt then waits based on the value of a flag and fills the corresponding buffer to be called within the main loop of the program once full.

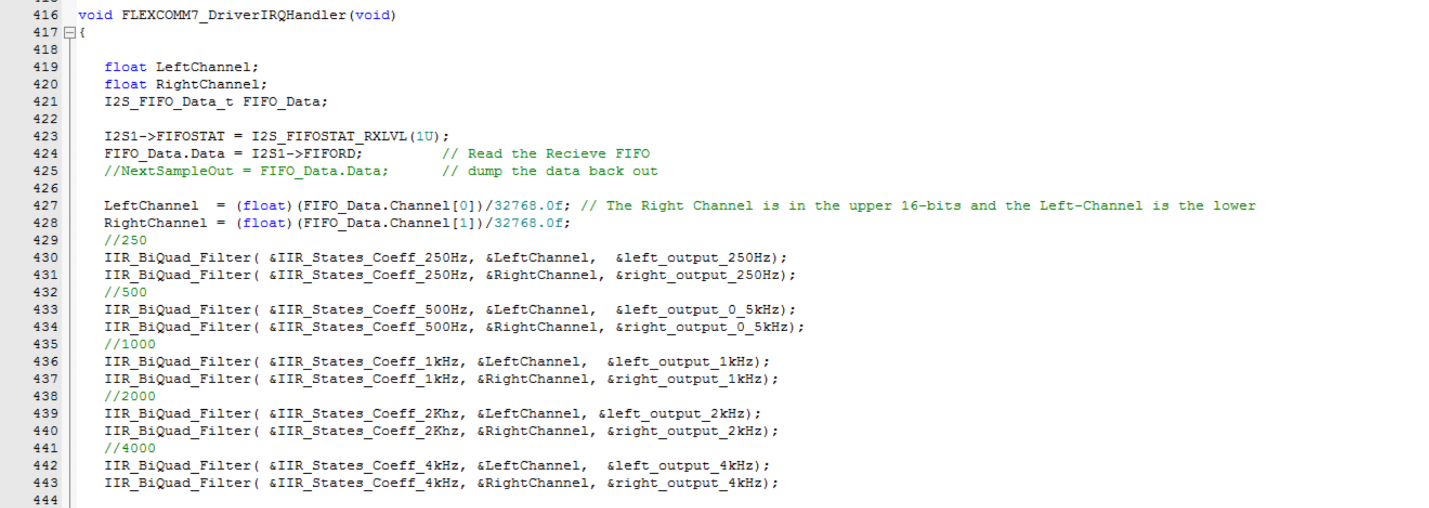
****

Figure : Data Receiving Interrupt in which the BiQuad Filter lies

The flag request\_new\_buffer is initially set to one, which corresponds to true and the main loop is requesting anew buffer to be filled. As Seen in figure 3, as the index remains under the value 1024, the flag remains one, and when the buffer reaches maximum capacity – where the index = 1024 – the flag is set to 0, where the main loop fires at a flag value of 0.

These parallel BiQuad filter’s essentially take the entire signal and parse it depending on their center frequency, so there is so splitting of the incoming audi0, but there is a summation of the output sample-by-sample values. This will be explained, along with their tuners alter on in the paper.

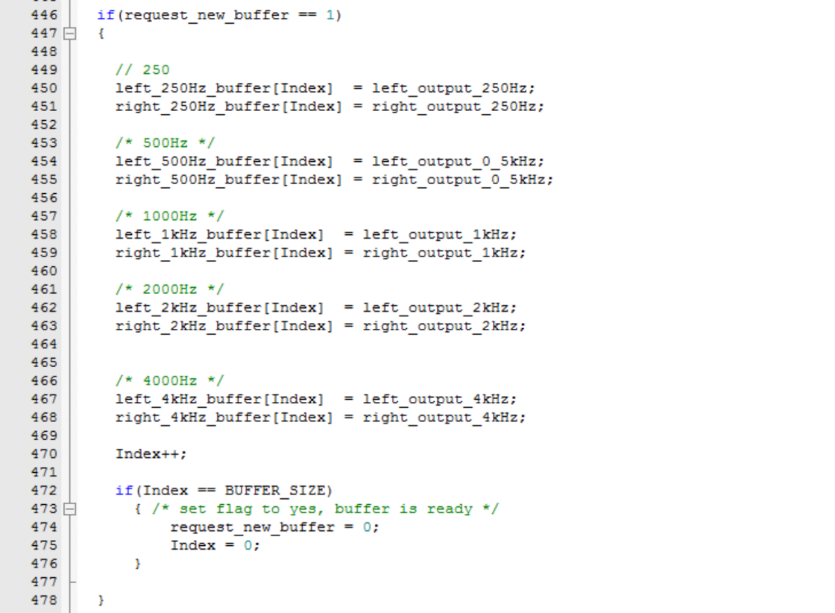


Figure : Next Buffer Ready storage and requesting loop for calling in the main loop

## Audio Output to aux CODEX

Below is the outgoing signal variable ‘NextSampleOut’ which outputs the signal sample-by-sample to the output interrupt, the flexcomm6 where ‘NextSampleOut’ is continuously called for every sample. As this happens almost instantly, the buffer is first initialized with two null values that are sent to the transmit (outgoing) interrupt. The outgoing variable is a summation of every each output of the 10 BiQuad filters. All the left and right channels are summed initially, where the left is bit shifted by 16 bits to create a 32-bit outgoing signal for both the left and right audio channels. As the channels have been normalized between zero and one, the multiplication by 32,768 returns the value to the original ratio.

Each channel is multiplied by what is called a tuner; derived directly from the normalized value of the slider position – a user input from the tactile screen. Each tuner, numbered 1 through 5 is globally defined and each multiplies both the left and right channel of its respective band, where 1 through 5 links to 250Hz, 500Hz, 1kHz, 2kHz, and 4kHz. The value of the slider has to be a float as the partial decimal values have to accounted for, the position of the slider then too had to be a float variable. As the initialization of the screen is set so that the lower positive of the slider is at the top, where the pixel number is lowest, the value had to be inverted so that the lower pixel number (at the top of the screen) corresponded to the maximum of the normalized slider position, then copied to the tuner. Many variables, the tuner ones incuded are copied and multiply defined throughout the program. This is an issue that would have to be addressed should the code get optimized. Optimizing the code would allow for a full 10 band equalizer.

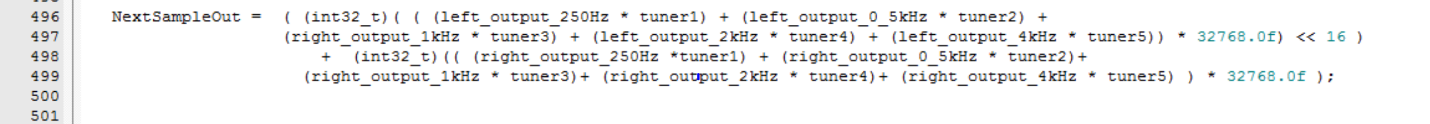


Figure : Outgoing signal, summed over all the filters and bit shifted

## BiQuad Filter Function (Architecture)

The BiQuad filter, as seen above in the interrupt is a function called 10 times for both the left and right channel of each chosen band for an input signal x[n]. The equation noted in Figure 5 below, on line 353 is the transfer function from which the variable stage is based upon. Based on the BiQuad structure (see Appendix 1 – BiQuad Filter Diagram) the coefficients b multiply the feedforward states, and the a coefficients a, negated to account for uncontrolled growth, multiply the feedbackward states where each sates, one, two, and three are summed to create the outgoing signal y[n]. The variable Local\_IIR\_set is of the variable type BiQuad\_Parameters, and contains the values for both the coefficients and the states, as seen below in the function.

As the BiQuad filter is a three level filter, the next stages are employed by setting the next stage equal to the previous one, ultimately outputting the final sample y[n] where the variable Output sends it back to the interrupt. As the filter is a function it can be called in parallel and used to compute each band simultaneously.

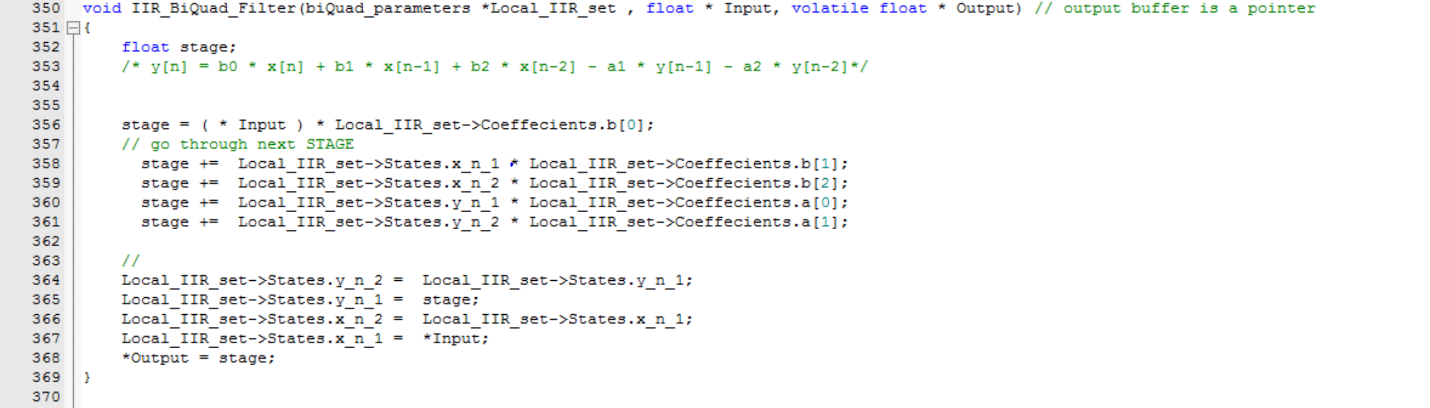


Figure : BiQuad Filter implementation

## Coefficient Computation Function (Architecture)

The coefficient computation is based upon a well-known set of transfer function that adhere to the nature of the BiQuad filter. As each set of coefficients, only need to be calculated once at startup based on the center and sample frequency, this function is called with the main loop and not in the interrupt. As seen below, the phase frequency ‘w0’, takes into account each sample and center frequency, which then affects all coefficients a[1:3], and b[1:3]. These coefficients are then normalized to a0 and each feedbackwards coefficient a[1:3] is negated to account for uncontrolled expansion.

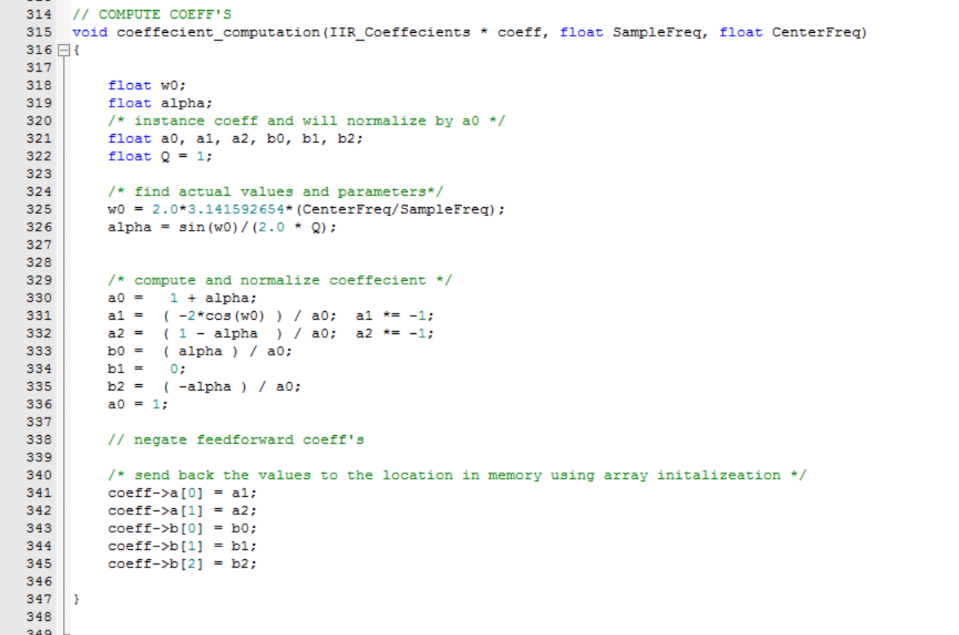


Figure : Coefficient Computation

## Main Function

### Coefficients and States Initialization

The following section of the paper utilize the functions and interrupts defined above. This is the main loop, and the functions along with their variable of type biquad\_parameters are called and computed upon. The following figure (figure 7) shows the computation of each coefficient structure, centered about each frequency input below.

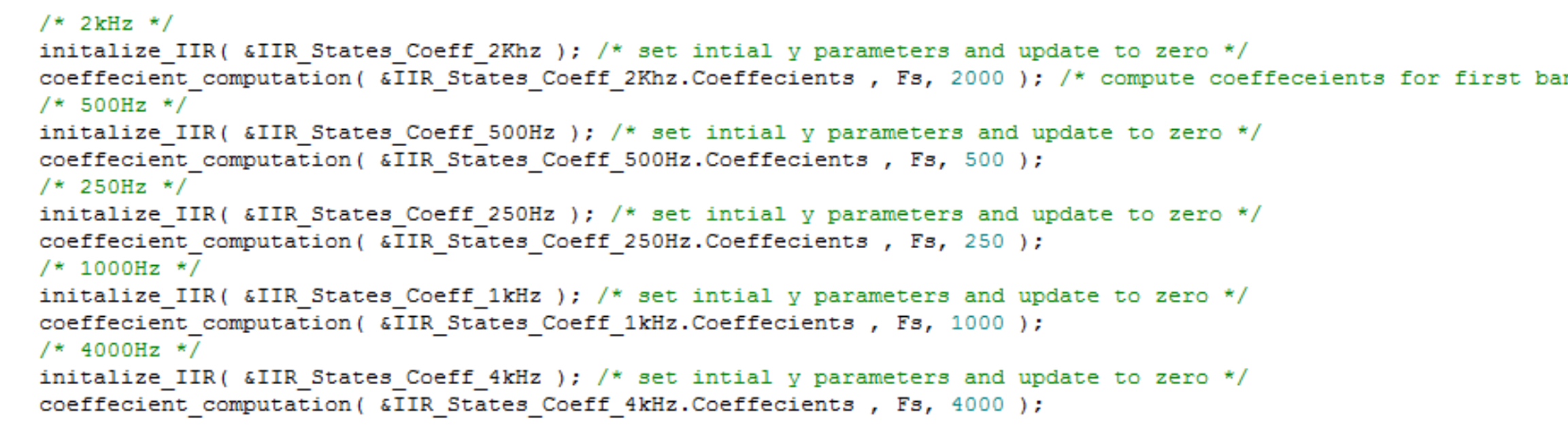


Figure : Calculate the coefficients for the bands, and initialize the IIR filter

### Calculate RMS for band Display

In order to display each band, the RMS value over the length of each buffer (1024 samples) is conitneouslty calculated as each buffer is dumped. The following arrays, summatin and rms\_value\_t are intialzied to accommodate for both the left and right channels of all five bands. The for-loop shown below in Figure 8 fires in real time, at the request of each new buffer that is dumped within the souroudning while loop. The summation is calcalued and the square root of the summation divided by the buffer size is stored in each corresponding rms\_value\_t index. The buffers upon which the summation occurs is still the normalized value, between 0 and 1, and so the RMS values carrry the characteristic (*side note:* The values increase with volume, so the de-normalizetion can not be so big that at a high volume the band would be too big for the screen).

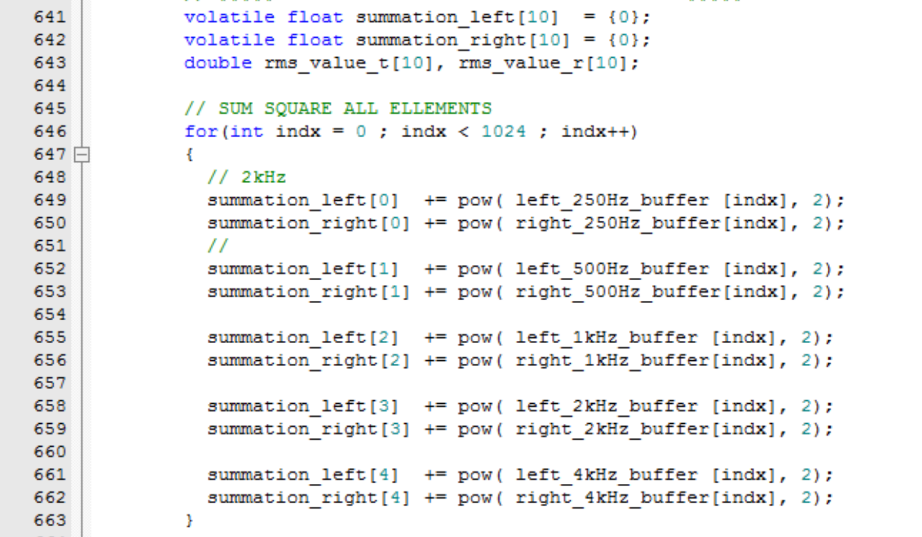


Figure : Summation over 1024 Sampels in the buffer for each BiQuad Filter

## Display

### Band Display Architecture

The display on the LCPXpresso board contains both the equalizer bars and tuners. The code generating both is housed within the main function. The architecture for creating the individual equalizer bar is shown below in Figure 9. The function eGFX\_DrawVline takes in two position parameters, both the x and y position. The for-loop plots the bar over the range desired – in this case 20 pixels for the left channel and 20 for the right. The RMS value is multiplied by the normalized and inverted slier position, and then multiplied by 1000 to ensure that the bar changes noticeably on the screen. It was seen during testing that some applications, with extreme volume do cause the bar to expand larger than the screen, but for normal audio listening multiplying by 1000 creates a nice visual. Two for-loops as shown below are individually run for each band, for a total of 10 output bands from the 10 BiQuad filters.

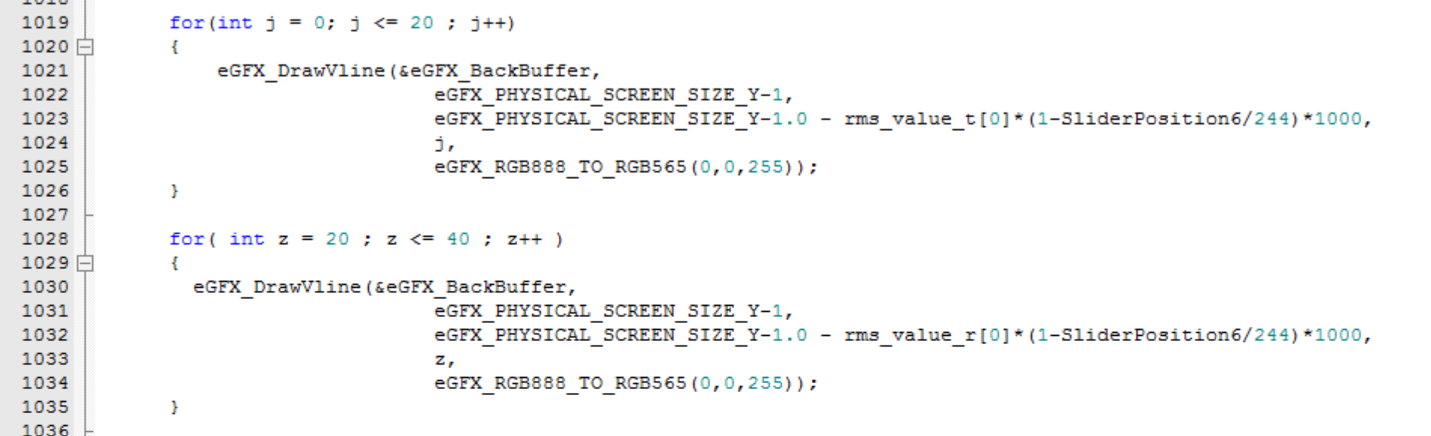


Figure 9: Plotting Equalizer Bars

# Conclusion/Results

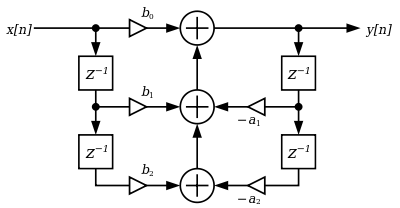
The equalizer was a success. The bands corresponding to each center frequency, when tested performed as expected. A sanity check, to ensure the correct computation of the bands was done using a tone generator. Centering the frequency of the tone generator to each band individually saw the accentuation of each respective band. Sweeping the frequency from a low value, to a high value, usually on the range of 100 Hz to 20 kHz, saw the bands each rise and drop in a moth continues motion as the sine wave frequency changed. The calculation of the coefficients was also checked, using the debugging features of the software Keil µVision. Comparing the values against the program ‘MonkeyJam’, in which the coefficients are known to be correct shows that the computation with the function for each frequency is accurate.

The outgoing signal should change depending on which band(s) is accentuated through the touchscreen value. When listening to the output of the audio CODEC the audio changed depending on the manipulation of the highs, the mids, and the lows.

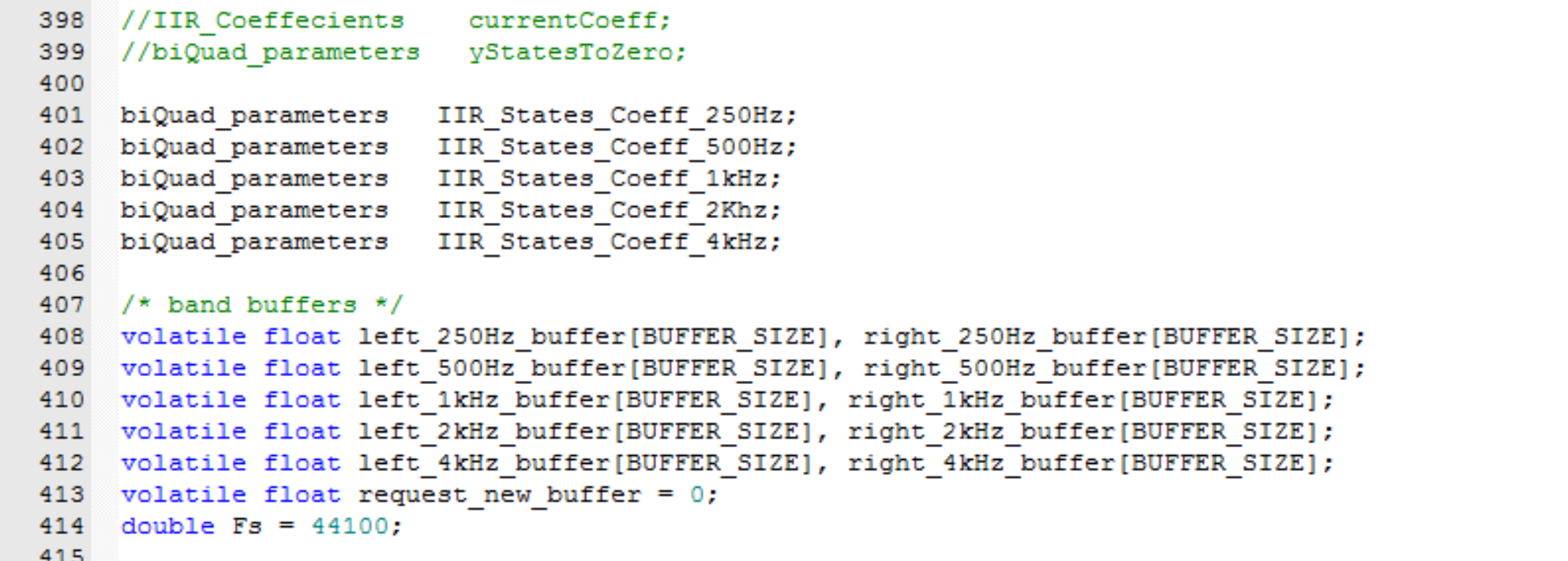
The tuners are the core of the project in my mind. Manipulating the bands, and displaying them to the screen based on the RMS value does not change the outgoing signal when the user listens to the output, but with the tuners, the values are passed back and change the outgoing signal. In this regard, this project was more than a graphical equalizer in that with the display of each band, the user can set and adjust each band to his or hers desired setting. Should we have had more time, we would have liked to have buttons with a set of preset equalizer settings, based on certain music types such as rock, classical, and pop.

# Appendices

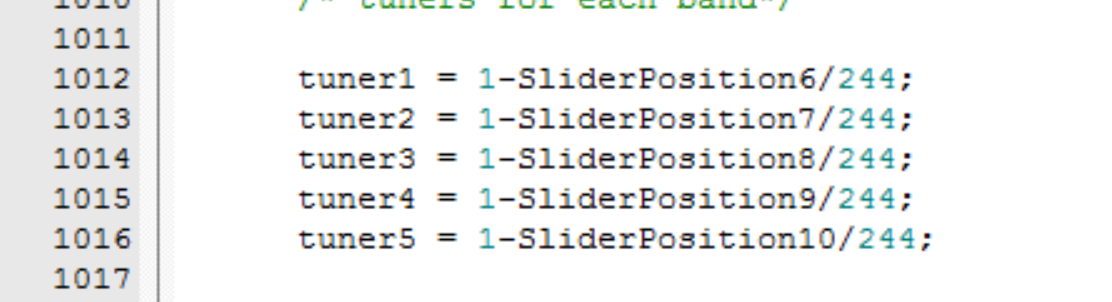
## Appendix 1 – BiQuad Filter Diagram



## Appendix 2 – Initialize buffers for the storage of data



## Appendix 3 - Tuners



## Appendix 4 – Sliders

