Digital Signal Processing - Project

Digital Filter Design

EEE4114F

Brandon Ferriera
Lezerick Owies
FRRBRA002 & OWSLEZ001

Department of Electrical Engineering
Engineering and the Built Environment
University of Cape Town

Contents

List of Figures

List of Tables

I. Introduction

The following report deals with the design of a digital filter using an STM32F0Discovery board. The overview and design processed followed is covered through the contents of this report.

A. *Problem Statement*

For our project, we selected Task 1 of the DSP section. The task states that we must implement a digital filter in hardware using a system like an STM32. We need to implement an ADC and DAC so that experiments can be conducted on the system using lab equipment. Additionally, a project description can be selected to provide further directions for the project. We decided to implement a cross-over frequency filter set to filter out signals above 800 Hz. This would be for the woofer element of the speaker.

B. *Literature review*

This literature review focuses on the design and implementation of digital filters by making use of Digital Signal Processing (DSP) theory as a foundation. It will investigate the evolution of digital filters as well as the advantages and disadvantages of different filter structures within the realm of digital filtering processes. The two filter structures to be investigated are Finite Impulse Response (FIR) and Infinite Impulse Response Filters(IIR).

Brief History of DSP technology
Digital Signal processing as it is known in the modern day is still a relatively new field as it only really gained traction with the invention of the MOSFET and its adaption into integrated circuits (IC) back in the 1970's. However the first stand alone digital signal processor chip was only invented late in the 1970's by Texas Instruments facility, this chip was named TMS5100 [5]. From there, other developers such as American Microsystems (AMI) and Intel started to develop their own DSP ICs. This helped to propel the technology even further with faster ICs being developed and new features being added over time. One such feature was the inclusion of an internal ADC/DAC and a internal signal processor on their 2920 DSP chip in 1979 [6]. Fast forward 30 years with many more improvements leading up to the modern day digital processors which have greater performance as well as a range of features such as internal ADC/DAC, multiple external peripheral support and very high operating speeds.

Finite Impulse Response Filter - FIR
A finite impulse response filter can be seen as a filter who's impulse response becomes zero over a period of time. These filters are very commonly used in digital filters especially audio filtering as they are easy to implement and have a linear phase response which is desirable for audio processing. A FIR filter with order N can be described by the following linear constant coefficient different equation:

$$y[n] = \sum_{i=0}^{N} b_i x[n-i]$$

where:

- y[n]: Output of the system
- x[n]: Input to the system
- b_i: The value of the FIR at a particular instance.

Properties of FIR Filters:

- FIR filters are always stable. This means that a FIR filter will never tend to infinity for a finite input.
- Commonly have linear phase.
- The order [N] of the filter dictates the amount of memory needed in order to implement it digitally.
- FIR filters require no feedback thus making the implementation simpler.

FIR filter design can be done easily by making use of a filter design tool such as Matlab's filter design tool. They can also be designed by making use of standard FIR filter design methods like the Window Sinc method and the Least mean error method.

Infinite Impulse response filter - IIR Filter
Unlike the FIR filter the IIR filter has a infinite impulse response as suggested by the name. This means that irrespective of how much time passes the impulse response will never reach zero. However, the impulse response often tends to zero in the from of an decaying exponential. The IIR filter has a more complex difference equation implementation than the FIR filter as they never truly decay. The general form of a difference equation for the IIR filter can be described by the following equation:

$$y[n] = \frac{1}{a_0}(\sum_{i=0}^{P} b_i x[n-i] - \sum_{j=0}^{Q} a_j x[n-j])$$

where:

- y[n]: Output of the system
- x[n]: Input to the system
- b_i: Feedforward filter Coefficients
- a_j: Feedback filter Coefficients
- P: Feedforward filter order
- P: Feedback filer order

Properties of IIR Filters:

- IIR Filters have a infinite impulse response
- Not always stable. This means for a bounded input the output can grow infinitely big under certain circumstances.
- Often computationally less expensive than FIR filters to perform the same task.
- IIR filters have non-linear phase.

IIR filters can also be designed by making use of a filter design tool or by making use of standard IIR filter design techniques.

Comparison between FIR and IIR Filters

Criteria	FIR filter	IIR filter
Implementation Complexity	Low	Higher
Computational resources	Often higher	Low
Phase response	Linear	non-linear
Impulse response	finite	infinite
Stability	Always Stable	Can become unstable

TABLE I: FIR vs IIR

Sampling Theory
Sampling theory is a fundamental part of DSP as it outlines the properties of sampled continuous time signals. This is important as almost all real world signals are analogue signals and would thus need to be sampled into discrete time so that they can be analysed using DSP principles. To convert a signal from continuous time to discrete time the continuous to discrete converter can be used and vise versa (please note that these are not analogue-to-digital or digital-to-analogue converters but rather an idealised version of them). In the real-world, analog signals can be converted to digital through the use of a DAC or digital-to-analog converter.

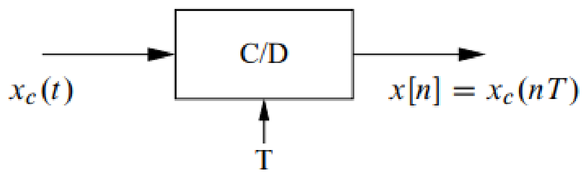


Fig. 1: Example of a Continuous to discrete converter [1]

Sampling of continuous signals The ideal expression for sampled continuous time signals can be seen below:

$$x[n] = x_c[nT], -\infty < T < \infty$$

where:

- T: Sampling period

One of the most important theories in sampling theory is the Nyquist sampling theory. It states that to sample a signal without causing aliasing, the sampling frequency must be at least two times the maximum frequency component in the signal you are trying to sample [7]. The Nyquist theory is especially relevant in this project as we have hardware speed limitations.

Audio cross over frequency
To reduce power usage and increase sound quality many audio systems make use of what is called a crossover. This essentially means that they make use of various different speaker that are better suited to handle a different range of frequencies [8]. The point at which the various frequency ranges intersect is called the cross-over frequency. The cross over frequency for any given speaker is specified by the manufacturer and is proportional to the size of the speaker. Bigger speakers handle low frequencies better but smaller speakers handle higher frequencies better [9].This is important as smaller speakers also need less power to produce higher frequencies which means by carefully choosing the speaker arrangement, better sound quality can be achieved at a lower power cost.

II. Methodology

The following section provides an overview of the theory and equipment used to design and test an appropriate filter. Additionally, the design processed used is shown as well as the code used to implement the filter.

A. Project Apparatus

The following section shows the main apparatus used during this project.

Oscilloscope
The oscilloscope was used to measure the output of our filter as well as the respective input signal from the signal generator.

Fig. 2: Tedtronix 2004B Oscilloscope [2]

WaveTek Signal Generator
The Signal generator was used to produce various input signals at different frequencies to test our filter response.

Fig. 3: WaveTek Signal Generator [3]

STM32F051 Development Board
The STM32F051 development board was used as the micro-controller for our project. This board was mainly selected due to its internal 12-bit ADC and DAC peripherals. The STM32 series also has extensive support and documentation for implementation of digital filters, thus making implementation relatively easy.

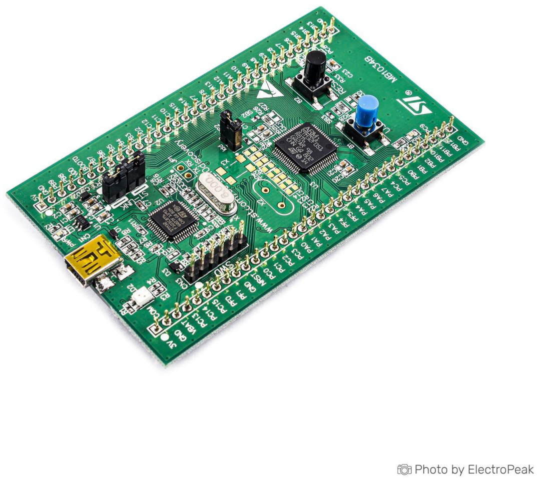


Fig. 4: STM32F051 Development board [4]

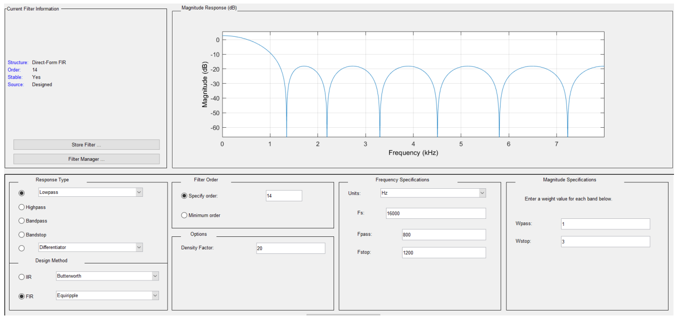
Other Relevant Apparatus

The following apparatus were also used during the project:

- Multi-meter: Used to take measurements and perform circuit debugging
- Matlab (FilterDesign Tool): Used to perform digital filter design for FIR filter

B. Design and Theory

All digital filters require coefficients to implement the difference equation required. The coefficients can be obtained through a design tool like Matlab's filterDesigner tool. The below image shows our filter design using Matlab's tools.


Current Filter Information
Structure: Direct-Form FIR
Order: 14
Stable: Yes
Source: Designed

Response Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Design Method: IIR (Butterworth) [✓ FIR (Equiripple)]

Filter Order: [✓ Specify order: 14] Minimum order

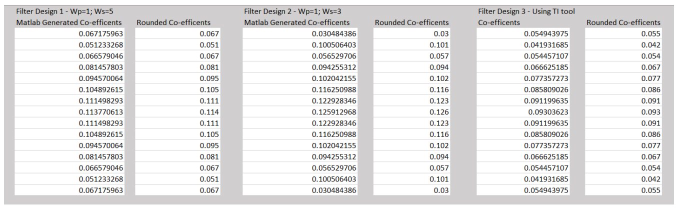
Options: Density Factor: 20

Frequency Specifications: Units: Hz; Fs: 16000; Fpass: 800; Fstop: 1200

Magnitude Specifications: Enter a weight value for each band below. Wpass: 1; Wstop: 3

Fig. 5: Matlab's filter design tool

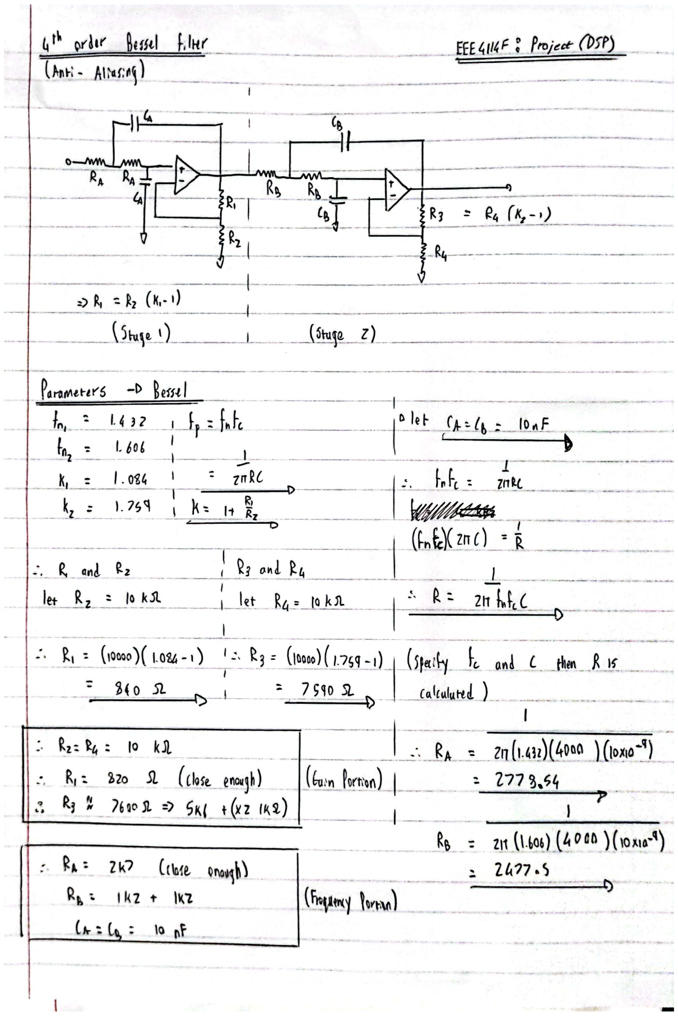
Two different designs were done using Matlab's filterDesigner tool and 1 design was done using TI's filter design tool. The purpose for using two design tools was to test the validity of the system operation. Below is the coefficients generated. The bode responses will be shown and discussed in the results section. For our filter design, the pass band was set to 800 Hz and the stop band was set to 1.2 kHz. This is so that the filter designed would perform the task of a cross-over frequency filter as defined in the problem statement.



Filter Design 1 - Wp=1; Ws=5		Filter Design 2 - Wp=1; Ws=3		Filter Design 3 - Using TI tool	
Matlab Generated Co-efficents	Rounded Co-efficents	Matlab Generated Co-efficents	Rounded Co-efficents	Co-efficents	Rounded Co-efficents
0.067175963	0.067	0.030484386	0.03	0.054943975	0.055
0.051233268	0.051	0.100506403	0.101	0.041931685	0.042
0.066579046	0.067	0.056529706	0.057	0.054457107	0.054
0.081457803	0.081	0.094255312	0.094	0.066625185	0.067
0.094570064	0.095	0.102042155	0.102	0.077357273	0.077
0.104892615	0.105	0.116250988	0.116	0.085809026	0.086
0.111498293	0.111	0.122928346	0.123	0.091199635	0.091
0.113770613	0.114	0.125912968	0.126	0.09303623	0.093
0.111498293	0.111	0.122928346	0.123	0.091199635	0.091
0.104892615	0.105	0.116250988	0.116	0.085809026	0.086
0.094570064	0.095	0.102042155	0.102	0.077357273	0.077
0.081457803	0.081	0.094255312	0.094	0.066625185	0.067
0.066579046	0.067	0.056529706	0.057	0.054457107	0.054
0.051233268	0.051	0.100506403	0.101	0.041931685	0.042
0.067175963	0.067	0.030484386	0.03	0.054943975	0.055

Fig. 6: Filter coefficients

To prevent high frequency components of our original signal from causing aliasing at the output we need to include a anti aliasing filter at the input. This filter is basically a low pass filter that will attenuated any high frequency components above $\frac{f_s}{4}$. The anti-aliasing filter was implemented using analog hardware and a fourth order Bessel Filter was chosen as the optimal design. The detailed filter design can be seen below.


4th order Bessel filter (Anti-Aliasing)

EEE4114F : Project (DSP)

$\Rightarrow R_1 = R_2(K_1 - 1)$

(Stage 1) (Stage 2)

Parameters → Bessel

$f_{n_1} = 1.432$

$f_{n_2} = 1.606$

$K_1 = 1.084$

$k_2 = 1.759$

$f_p = f_n f_c = \frac{1}{2\pi RC}$

$k = 1 + \frac{R_1}{R_2}$

∴ R_1 and R_2

let $R_2 = 10\,k\Omega$

∴ $R_1 = (10000)(1.084 - 1) = 840\ \Omega$

R_3 and R_4

let $R_4 = 10\,k\Omega$

∴ $R_3 = (10000)(1.759 - 1) = 7590\ \Omega$

$\boxed{\begin{array}{l} \therefore R_2 = R_4 = 10\ k\Omega \\ \therefore R_1 = 820\ \Omega \text{ (close enough)} \\ \therefore R_3 \approx 7600\,\Omega \Rightarrow 5k1 + (\times 2\ 1k2) \end{array}}$ (Gain Portion)

$\boxed{\begin{array}{l} \therefore R_A = 2k7 \text{ (close enough)} \\ R_B = 1k2 + 1k2 \\ C_A = C_B = 10\ nF \end{array}}$ (Frequency Portion)

let $C_A = C_B = 10\,nF$

∴ $f_n f_c = \frac{1}{2\pi RC}$

~~[illegible]~~

$(f_n f_c)(2\pi C) = \frac{1}{R}$

∴ $R = \frac{1}{2\pi f_n f_c C}$

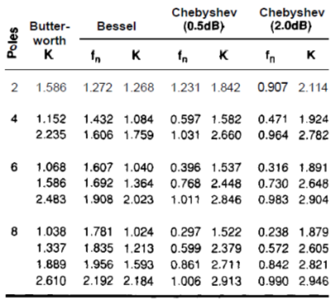
(Specify f_c and C then R is calculated)

∴ $R_A = \frac{1}{2\pi(1.432)(4000)(10\times10^{-9})} = 2778.54$

$R_B = \frac{1}{2\pi(1.606)(4000)(10\times10^{-9})} = 2477.5$

Fig. 7: Anti-aliasing filter design

The theory for the design comes from UCT's EEE3090F course which references the table shown below [10].



Poles	Butter-worth	Bessel		Chebyshev (0.5dB)		Chebyshev (2.0dB)	
	K	f_n	K	f_n	K	f_n	K
2	1.586	1.272	1.268	1.231	1.842	0.907	2.114
4	1.152	1.432	1.084	0.597	1.582	0.471	1.924
	2.235	1.606	1.759	1.031	2.660	0.964	2.782
6	1.068	1.607	1.040	0.396	1.537	0.316	1.891
	1.586	1.692	1.364	0.768	2.448	0.730	2.648
	2.483	1.908	2.023	1.011	2.846	0.983	2.904
8	1.038	1.781	1.024	0.297	1.522	0.238	1.879
	1.337	1.835	1.213	0.599	2.379	0.572	2.605
	1.889	1.956	1.593	0.861	2.711	0.842	2.821
	2.610	2.192	2.184	1.006	2.913	0.990	2.946

Fig. 8: Filter Design table

The values for a fourth order Bessel filter were used in the calculations shown above.

C. System Code

The main elements of our code include the actual FIR algorithm and coefficients array. The first few code snippets that are shown implement the actual FIR Filter. The FIR filter can be implemented in one of two ways, both use a difference equation. The difference comes in how the previous input data buffer is updated. The first implementation shown updates all the values in the data buffer after a calculation. The algorithm shifts all the data values over by 1. This can become computationally expensive for high order FIR filters as the amount of array updates equals the FIR filters order. The second implementation that is shown makes use of a circular buffer. Instead of updating all the values in the data array, you update only one, the oldest, and change the pointer array referencing where data elements are. This saves computational resources of the system as less array modifications have to be made. Experimentation shows that our implementation of a order 15 FIR filter has both methods producing acceptable filter responses. However, the circular buffer will be more efficient. The third code snippet shown initiates an ADC read which brings data into the system. Following this, the ADC data is sent to the FIR Filter code which then performs is filtering operation on the data. After the data has been filtered, the data is sent to the DAC so that it can be measured. The measured values can be seen in the results section.

```
// FIR Filter 1

uint16_t        FIR_Filter(uint16_t Fin, uint16_t Order)
{
        float FinF = (float)Fin;
        float Output = 0;

        for (int i=0;i<Order;i++)
        {
                Output +=
                ↪  (Buffer[Order-i-1]*Coefficients[Order-i-1]);
                if(i == Order-1){
                        Buffer[Order-i-1] = FinF;
                }
                else{
                        Buffer[Order-i-1] = Buffer[Order-i-2];
                }
        }
        return(uint16_t)Output;
}
```

```
// FIR Filter 2

uint16_t        FIR_Filter(uint16_t Fin, uint16_t Order)
{
        float FinF = (float)Fin;
        float Output = 0;

        for (int i=0;i<Order;i++)
        {
                Output += (Buffer[i]*Coefficients[Position[i]]);
                Position[i]++;
        }

        Buffer[PosCounter]=FinF;
        Position[PosCounter]=0;
        PosCounter++;

        if(PosCounter==Order)
        {
                PosCounter=0;
        }
        return(uint16_t)Output;
}
```

```
// Read ADC Value, Filter ADC value and send to DAC

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
        //adc_val = HAL_ADC_GetValue(&HADC1);
        adc_val = HAL_ADC_GetValue(hadc);
        //Toggle Blue Led
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_8);
        //add FIR call 8/5/2022
        dac_val=FIR_Filter(adc_val,15);
//        dac_val=adc_val;
        HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R,
        ↪  dac_val);
}
```

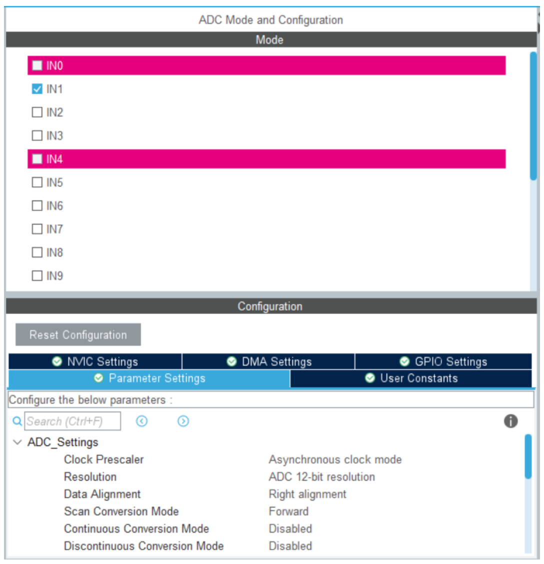
The next set of code is the setup of the coefficients data arrays. There are 15 coefficients in the array as this is an order 15 FIR filter. These coefficients will be multiplied with the various previous and current input data items in the FIR filter's difference equation. Three sets of coefficients were generated from two different design tools. The two design tools are Matlab's filterDesigner (2 sets) and TI's filter design tool (1 set). The reason 3 sets of coefficients were generated was to confirm that the filter hardware operated and produced similar results for data that should produce similar results.

```
//Added here 14/5/2022 rounded coefficients off to 3 places
//to allow the STM32 to keep up at 16kHZ sampling frequency

static float Coefficients[15]={
        0.055,
        0.042,
        0.054,
        0.067,
        0.077,
        0.086,
        0.091,
        0.093,
        0.091,
        0.086,
        0.077,
        0.067,
        0.054,
        0.042,
        0.055
};
```

The last bit of code shows function prototypes for the various functions used as well as some of the important global variables. These include configuration functions that can be generated when using STM32Cube IDE and the data buffer variables.

```
/* Private variables
↪  --------------------------------------------------------*/
ADC_HandleTypeDef hadc;

DAC_HandleTypeDef hdac1;

TIM_HandleTypeDef htim1;

/* USER CODE BEGIN PV */

uint16_t adc_val;                //added here 7/5/2022
uint16_t dac_val;                //added here 7/5/2022

/* USER CODE END PV */

/* Private function prototypes
↪  ---------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM1_Init(void);
/* USER CODE BEGIN PFP */

uint16_t FIR_Filter(uint16_t, uint16_t);                // FIR
↪  Filter Function

static float Buffer[15];                // Contains previous inputs
↪  for FIR filter
static int Position[15];                // References coefficients
↪  for a Buffer Position
static int PosCounter=0;
```

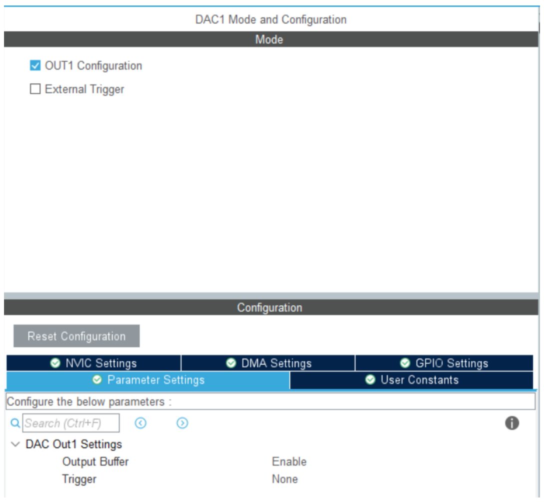
As mentioned previously, some of the code can be generated by STMCube IDE. The code includes port configuration functions for the ADC, DAC, and TIMER module. In order to generate the code, some settings must first be configured. This process is described below. In the IDE, modules can be configured in windows like those shown below. Once configured, the code can be generated and provides a template to base your project on.


ADC Mode and Configuration
Mode
IN0
[✓] IN1
IN2
IN3
IN4
IN5
IN6
IN7
IN8
IN9
Configuration
Reset Configuration
NVIC Settings | DMA Settings | GPIO Settings | Parameter Settings | User Constants
Configure the below parameters :
ADC_Settings

Clock Prescaler	Asynchronous clock mode
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Scan Conversion Mode	Forward
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled

Fig. 9: ADC configuration window

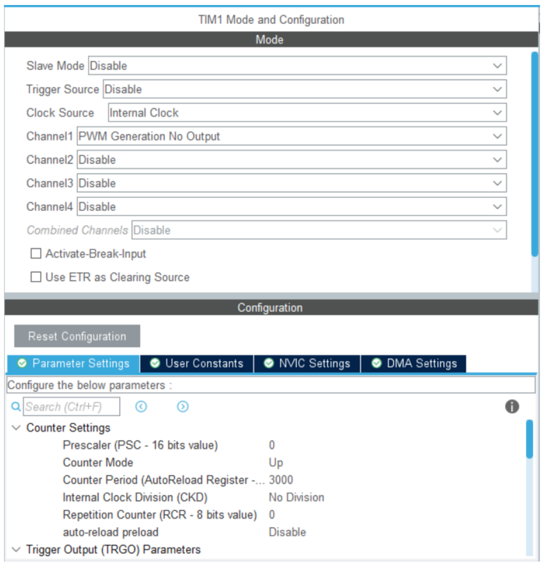
It is important to note that the ADC module is setup in interrupt mode. Every time the timer1 module reaches its maximum, it triggers an event. This triggers an ADC interrupt. An interrupt based approach was taken to produce a finite sampling time and frequency. Once the ADC has finished its conversion, the FIR Filter algorithm is called. Refer to the last code snippet shown.


DAC1 Mode and Configuration
Mode
[✓] OUT1 Configuration
External Trigger
Configuration
Reset Configuration
NVIC Settings | DMA Settings | GPIO Settings | Parameter Settings | User Constants
Configure the below parameters :
DAC Out1 Settings

Output Buffer	Enable
Trigger	None

Fig. 10: DAC configuration window

It is important to note that the counter period is set to a value of the clock frequency, set to 48 MHz by the internal oscillator multiplied by the PLL, divided the desired sample frequency. The clock period selected is roughly the smallest value that can be used to allow for interrupt latency to occur. Interrupt latency is the amount of time it takes for the interrupt to occur and be ready for the next trigger. A blue LED is toggled every time the interrupt service routine is entered. The toggling is done at half the sampling frequency and represents the Nyquist frequency. The timer module allows for discrete time interval sampling.


TIM1 Mode and Configuration
Mode

Slave Mode	Disable
Trigger Source	Disable
Clock Source	Internal Clock
Channel1	PWM Generation No Output
Channel2	Disable
Channel3	Disable
Channel4	Disable
Combined Channels	Disable

Activate-Break-Input
Use ETR as Clearing Source
Configuration
Reset Configuration
Parameter Settings | User Constants | NVIC Settings | DMA Settings
Configure the below parameters :
Counter Settings

Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register -...	3000
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - 8 bits value)	0
auto-reload preload	Disable

Trigger Output (TRGO) Parameters

Fig. 11: Timer configuration window

The below image shows the window that is used to set the various clock parameters

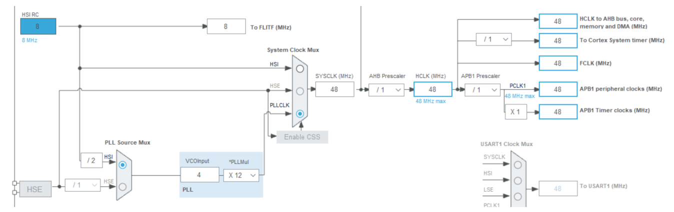


Fig. 12: Clock configuration window

D. Board Construction

Below shows the implemented design consisting of the STMDiscovery, anti-aliasing filter and connecting components soldered onto vero-board. This was done for neatness and presentation of design. Additionally, the connection points make it easier to connect signals and probes to the relevant pins for measurements and system operation. The smaller board is the anti-aliasing filter.

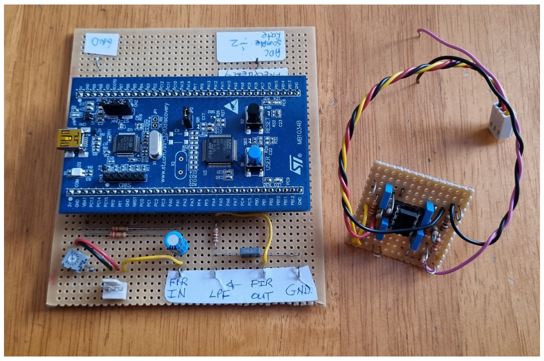


Fig. 13: Hardware Implementation

III. Results

A. Filter Operation

The below images show the filter response at the various frequencies. Orange shows the output and blue shows the input.

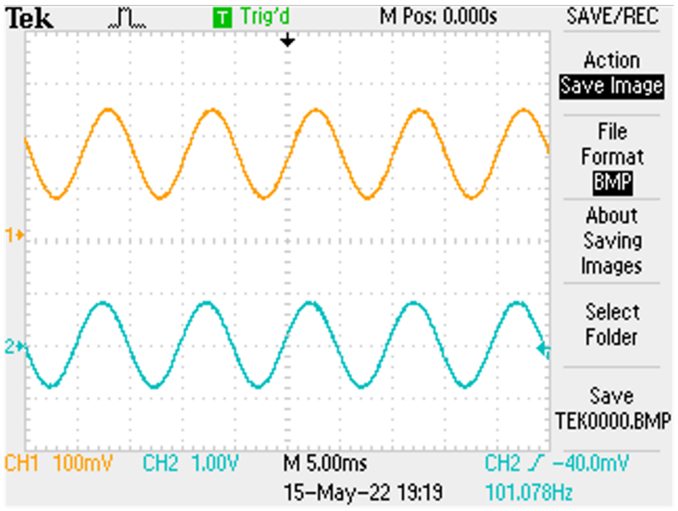


Fig. 14: Digital Filter Response at 100 Hz

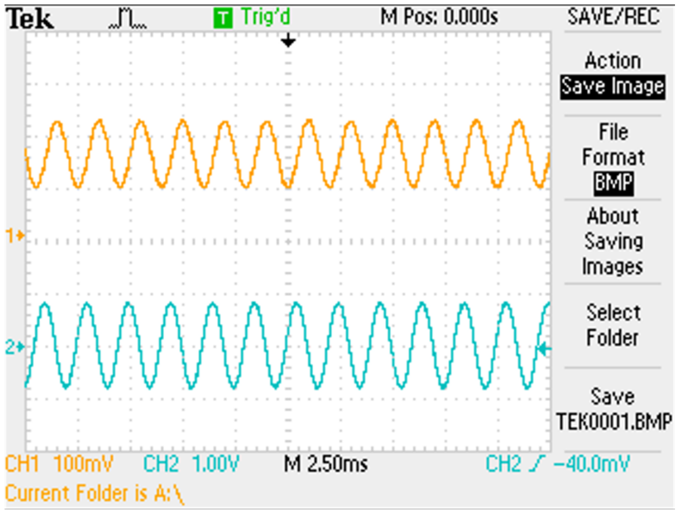


Fig. 15: Digital Filter Response at 500 Hz

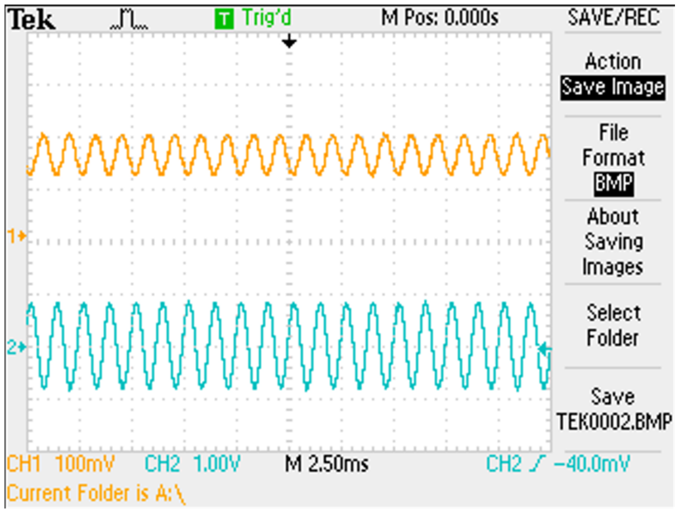


Fig. 16: Digital Filter Response at 800 Hz

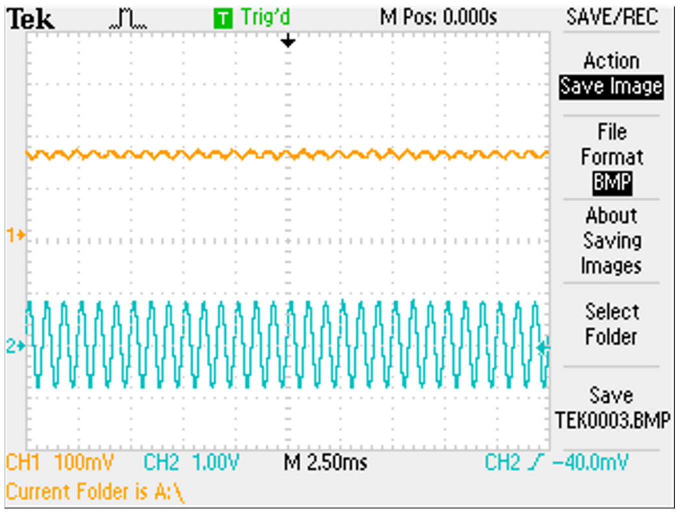


Fig. 17: Digital Filter Response at 1000 Hz

As can be seen from the images above, the frequency responses for 100 Hz and 500 Hz show no attenuation. The frequency response for 800 Hz, the cut-off frequency, shows some attenuation. The frequency response for 1 kHz shows heavy attenuation. The stop-band was set to 1.2 kHz in Matlab's filterDesigner tool. Therefore, this response is as expected. The levels of attenuation for 800 Hz is as expected from the design as the coefficient values were rounded and simulation does not follow real-world scenarios exactly. Additionally, the simulations seen in **??** show that the filter does attenuate the signal to some degree at 800 Hz.

B. Bode response for different co-efficient values with similar design specs

Referring to Figure 6,**??**, the below images are the bode magnitude responses for the different coefficient sets.

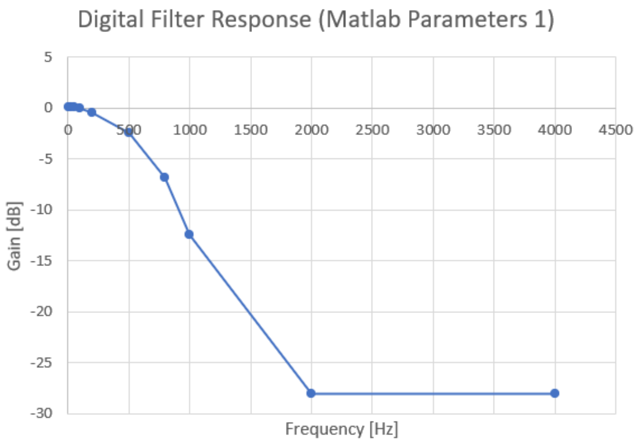


Fig. 18: FIR Filter 1 - Matlab Tool

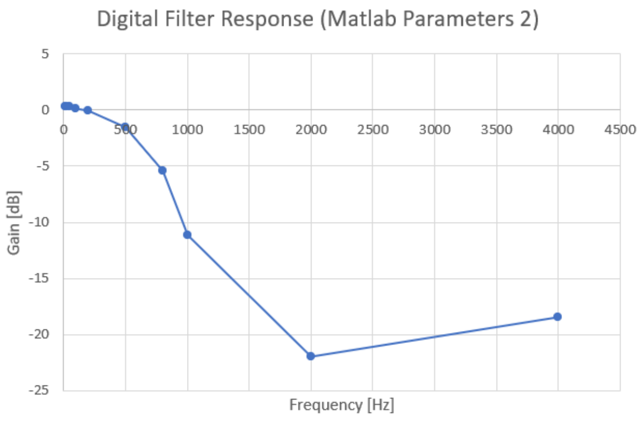


Fig. 19: FIR Filter 2 - Matlab Tool

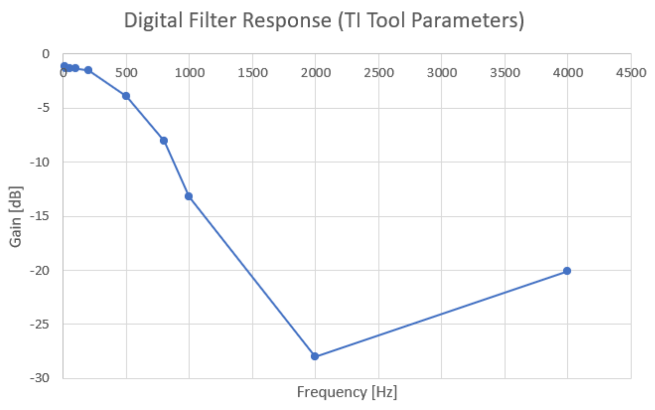


Fig. 20: FIR Filter 3 - TI Tool

As can be seen from all three frequency magnitude responses, the digital filter is operating as expected. The digital filter is filtering out signals that are above 800 Hz.

C. Anti-aliasing

A problem was discovered during experimentation with our final design. This problem is shown below.

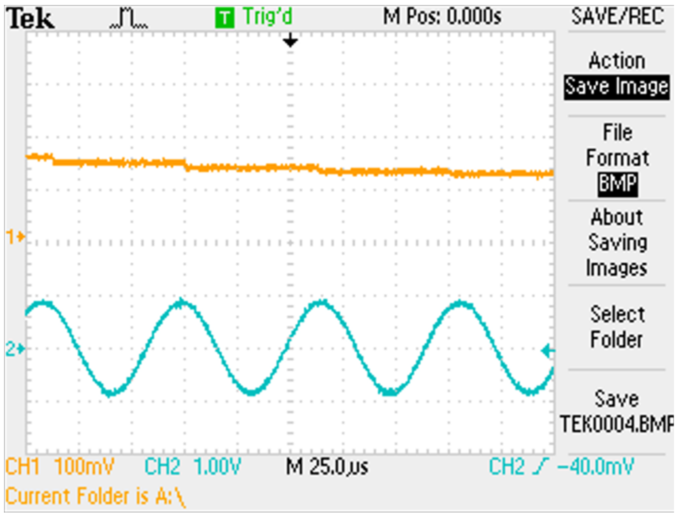


Fig. 21: Aliasing effect at sampling frequency

While the full waveform cannot be shown due to the limitations of the oscilloscope, what happened is that aliasing occurred. This problem occurred around the sampling frequency as well as successive octaves. The output wave-form became unaffected by the filter and reached an amplitude of about 100 mV. A potential solution to this problem is to use an anti-aliasing filter. The specific filter design used to test this theory was a fourth order low-pass Bessel filter designed around 4 kHz. A Bessel implementation was chosen because of its linear phase response, a desirable trait for audio filters. 4 kHz was chosen as it would demonstrated whether the solution work while not defeating the point of the digital filter. The digital filter has a cut-off frequency of 800 Hz therefore a cut-off of 4 kHz should have no effect on the validity of our digital filters results. Below is the resulting waveform once the anti-aliasing filter was added.

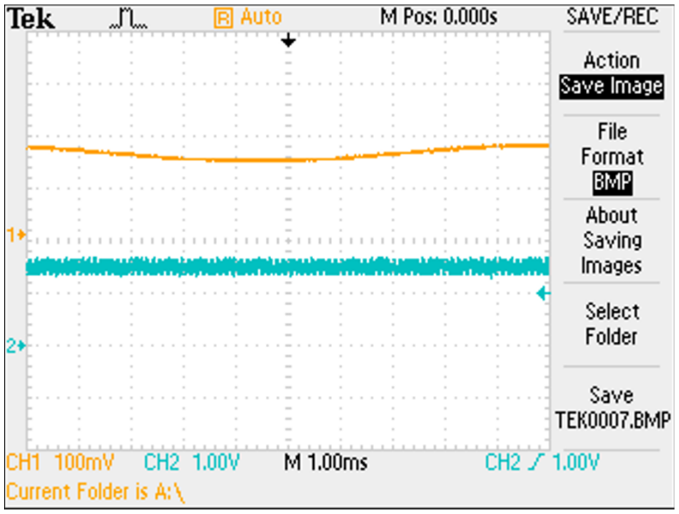


Fig. 22: Output at 15.38 kHz after anti-aliasing filter has been added

The result shown above has a clearer waveform. The Peak-to-Peak voltage of the output is about 20 mV. Further experiment to 32 kHz showed no output. Therefore, the anti-aliasing filter design will suppress the aliasing frequencies that broke through the digital filter implementation alone. The design of the anti-aliasing filter can be found in the methodology section.

IV. Conclusion

This project explored the implementation of an online digital filter with ADC and DAC capabilities for real time signal filtering. The STM32F051 development board was used for the filter implementation as it has built in DAC and ADC capabilities thus reducing the amount of external circuitry needed.

The filter design was primarily done in Matlab's filter design tool and verified using TI's filter design tool. The filter implemented was a FIR low-pass frequency with a cut of frequency of 800Hz. 800Hz was chosen due to project specifications. The sampling frequency of 16 kHz due to micro-controller speed and interrupt latency constraints. Therefore, this choice was to ensure smooth operation of the system and satisfaction of the Nyquist sampling criterion. Overall the project was a success as our filter attenuated values above 800 Hz and pass through values below. However, due to some leakage of high frequency components at the switching frequency of 16 kHz and its successive octaves, An anti-aliasing filter was included at the input of the filter to correct this. This was done as an analogue 4th order Bessel filter.

In future projects we would like to explore the possibility of having the anti-aliasing filter digitally as well. This would be interesting as we might need to use faster hardware to account for the extra calculation or develop a more computationally efficient filter that can operate faster on current hardware. Furthermore we would also look at implementing a multi-pass (band-pass) filter digitally for the purpose of filtering out very low frequency components such as 1 to 50 Hz. This would result in a cleaner audio signal for our application.

References

[1] F. Nichols, "Sampling of continuous-time signals," 2022.
[2] "Tds2004b tektronix digital oscilloscope." [Online]. Available: https://www.valuetronics.com/product/tds2004b-tektronix-digital-oscilloscope-used
[3] "188 wavetek." [Online]. Available: https://www.valuetronics.com/product/188-wavetek-function-generator-used
[4] "Stm32f0discovery - 32-bit arm cortex-m0 64kb flash stm32f0 discovery kit." [Online]. Available: https://www.evelta.com/stm32f0discovery-stm32f0-discovery-kit/
[5] S. Taranovich, "30 years of dsp: From a child's toy to 4g and beyond," Aug 2012. [Online]. Available: https://www.edn.com/30-years-of-dsp-from-a-childs-toy-to-4g-and-beyond/
[6] "The story of the intel® 4004." [Online]. Available: https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html
[7] [Online]. Available: https://www.tutorialspoint.com/digital_communication/digital_communication_sampling.htm
[8] Audiosolace, "How to set crossover frequency for speakers," Feb 2022. [Online]. Available: https://audiosolace.com/how-to-set-crossover-frequency-for-speakers/
[9] "Importance of crossover frequency," Mar 1964. [Online]. Available: https://electronics.stackexchange.com/questions/244519/importance-of-crossover-frequency
[10] M. Y. Abdul Gaffar, "Eee3090 electronic devices and circuits," 2021.