UNIVERSITY OF SUSSEX

868H1: Digital Signal Processing Laboratory

Course Report – Spring 22/23

Student Name: Muhammed Emre DUMAN

Candidate Number: 268272

Table of Contents

1. Introduction

The ability to accurately recognize and distinguish spoken words is a vital aspect of modern voice-activated systems and human-computer interaction. In this report, we present an approach for discerning between the spoken words "yes" and "no" using the FM4-S6E2CC-ETH device. This work has significant implications in the development of efficient and reliable speech recognition systems with real-world applications. Our approach involves utilizing the FM4-S6E2CC-ETH device to analyse the provided sound files of the words "yes" and "no" and subsequently plotting their mean power spectra. By examining the frequency domain characteristics of the two words, we can identify unique features that allow us to differentiate between them. In particular, we design low-pass and high-pass filters with appropriate cut-off frequencies based on the mean power spectra. Once the filters are established, the input voice signal is processed through both low-pass and high-pass filters. The resulting filtered signal's ratio is then used to determine whether the spoken word is "yes" or "no". To provide immediate visual feedback, the blue LED on the FM4-S6E2CC-ETH board is turned on when the word "yes" is detected, while the green LED is activated for the word "no”. This report provides a comprehensive overview of the methodology employed, the design of the filters, and the performance evaluation of our approach.

2. ARM architecture and development board

ARM Architecture

ARM (Advanced RISC Machines) architecture is a family of Reduced Instruction Set Computer (RISC) architectures designed for efficient and low-power consumption processors. This architecture has become the industry standard for embedded systems, smartphones, and various other electronic devices. RISC processors, like ARM, use a simplified set of instructions, enabling them to execute operations more rapidly and consume less power compared to Complex Instruction Set Computing (CISC) architectures.

Cypress FM4-S6E2CC-ETH ARM Cortex-M4 Starter Kit

The FM4-S6E2CC-ETH board is built around the Cypress S6E2CC series microcontroller, which features a 32-bit ARM Cortex-M4F processor core. This powerful core operates at up to 160 MHz and integrates a Floating-Point Unit (FPU) and a Memory Protection Unit (MPU). In addition, the board offers 512 KB of flash memory, 64 KB of SRAM, and a rich set of peripherals, including Ethernet, USB, and multiple serial communication interfaces. The board also includes features such as:

- ARM Cortex-M4 core with DSP and FPU (Floating Point Unit) support, running at up to 160 MHz.
- Up to 2 MB of flash memory and 256 KB of SRAM for code and data storage.
- Integrated Ethernet MAC (Media Access Control) for high-speed network connectivity.
- Comprehensive set of peripherals, including GPIO, I2C, SPI, UART, and ADC.
- Support for various communication protocols, such as CAN, LIN, and USB.
- On-board debug and programming interface.
- Expansion headers for easy interfacing with external hardware or daughter boards.

The ARM Cortex-M4F processor is particularly suitable for Digital Signal Processing (DSP) applications, thanks to its specific architectural features and enhanced instruction set. Some key aspects that contribute to its suitability for DSP tasks include:

- Single-cycle Multiply-accumulate (MAC) operation: The Cortex-M4F core supports single-cycle MAC operations, which are fundamental for DSP algorithms such as filtering, convolution, and Fast Fourier Transforms (FFT). This capability allows for efficient multiplication and accumulation of data, leading to improved performance in DSP applications.

- Floating-Point Unit (FPU): The FPU integrated into the Cortex-M4F core enables high-precision arithmetic operations, critical for accurate signal processing. The FPU supports single-precision floating-point operations, providing extended dynamic range and allowing complex calculations without the risk of overflow or underflow.
- SIMD (Single Instruction Multiple Data) instructions: ARM Cortex-M4F includes SIMD instructions, enabling the processor to perform multiple operations simultaneously. This feature accelerates vector and matrix operations commonly used in DSP applications, such as audio processing and image manipulation.
- DSP instruction set: The Cortex-M4F core offers a specialized DSP instruction set, including Saturated Arithmetic (SAT) and Bit-Reverse (RBIT) instructions. These instructions help prevent data overflow and simplify bit manipulation, enhancing the execution of DSP algorithms.
- Nested Vectored Interrupt Controller (NVIC): The NVIC in the Cortex-M4F core supports low-latency interrupt handling, enabling real-time responsiveness in DSP applications. This feature allows the processor to efficiently manage and prioritize multiple interrupt sources, ensuring timely processing of time-sensitive signals.
- Low-power consumption: ARM Cortex-M4F processors are designed for low-power operation, making them ideal for battery-powered or energy-efficient devices. This feature is particularly beneficial for DSP applications in portable and remote devices, where energy efficiency is crucial.

In summary, the ARM Cortex-M4F processor's architecture and the features of the FM4-S6E2CC-ETH board make them well-suited for DSP applications. The combination of single-cycle MAC operations, an integrated FPU, SIMD instructions, a specialized DSP instruction set, NVIC, and low-power consumption ensures that the ARM Cortex-M4F processor can efficiently execute complex signal processing tasks, making it an excellent choice for a wide range of applications.

All the details are obtained from https://developer.arm.com/Processors/Cortex-M44

3. Project Design

Project Description

The proposed project entails the development of a digital signal processing (DSP) based system, specifically designed to discern between the spoken words "Yes" and "No." It is essential to note that no other words will be taken into consideration. This system will be implemented on the aforementioned ARM Cortex-M4 based development board, leveraging the DSP algorithms discussed in prior lectures.

A diverse set of audio test files have been supplied to evaluate the system's performance. These files will be fed through the line input of the development board and the headphone output of a computer. The test files exhibit varying pronunciation, tonality, and pitch, necessitating that the devised system possess the capability to accurately categorize the audio data, despite these disparities.

The project's development process will be bifurcated into two distinct categories: model-based design and hardware deployment. This approach ensures a comprehensive and structured methodology to achieve the project's objectives.

Designing the model

To ascertain the appropriate DSP technique to employ, it is imperative to analyse the signal's attributes. It has been established that the phoneme "S" in the word "Yes" exhibits a high-frequency band component, while simultaneously retaining the low-frequency band originating from the same syllable. In contrast, the word "No" predominantly encompasses a low-frequency band component. This information is crucial for selecting the most suitable DSP approach to effectively distinguish between the two words.

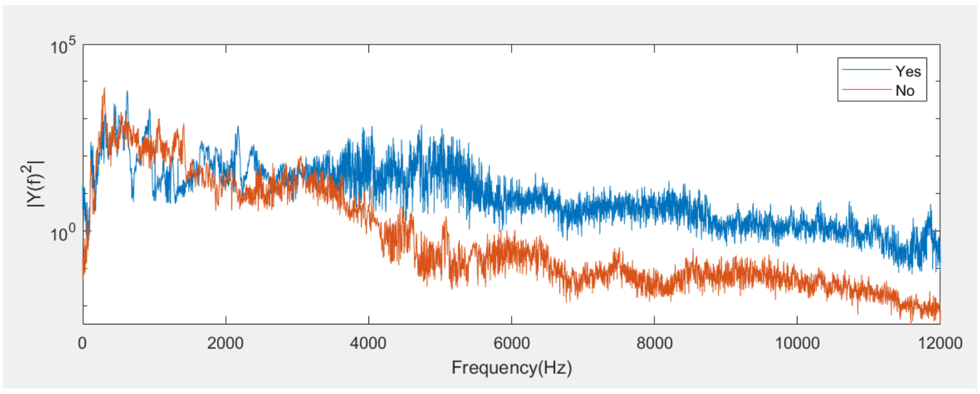
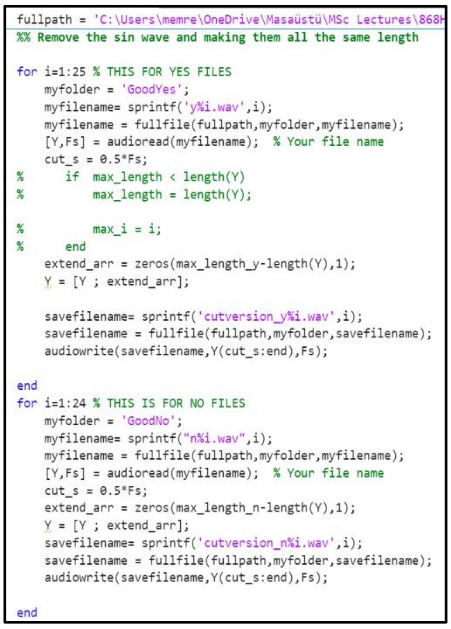
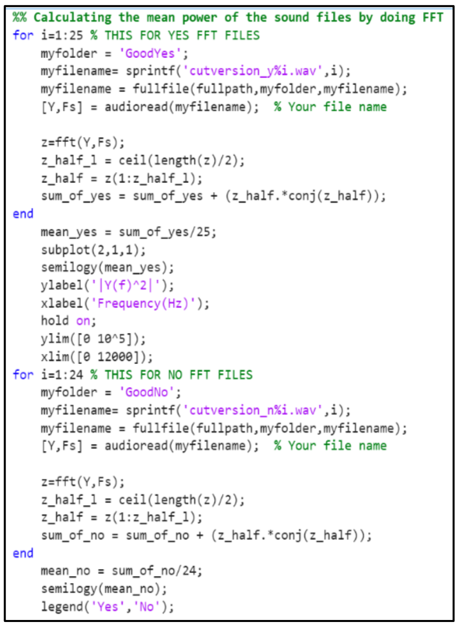


Figure 1 - Mean power spectrum graph

Figure 1 represents the mean power spectrum of provided “Yes” and “No” files.



```
fullpath = 'C:\Users\memre\OneDrive\Masaüstü\MSc Lectures\868H
%% Remove the sin wave and making them all the same length

for i=1:25 % THIS FOR YES FILES
    myfolder = 'GoodYes';
    myfilename= sprintf('y%i.wav',i);
    myfilename = fullfile(fullpath,myfolder,myfilename);
    [Y,Fs] = audioread(myfilename);  % Your file name
    cut_s = 0.5*Fs;
%      if  max_length < length(Y)
%          max_length = length(Y);

%          max_i = i;
%      end
    extend_arr = zeros(max_length_y-length(Y),1);
    Y = [Y ; extend_arr];

    savefilename= sprintf('cutversion_y%i.wav',i);
    savefilename = fullfile(fullpath,myfolder,savefilename);
    audiowrite(savefilename,Y(cut_s:end),Fs);

end
for i=1:24 % THIS IS FOR NO FILES
    myfolder = 'GoodNo';
    myfilename= sprintf("n%i.wav",i);
    myfilename = fullfile(fullpath,myfolder,myfilename);
    [Y,Fs] = audioread(myfilename);  % Your file name
    cut_s = 0.5*Fs;
    extend_arr = zeros(max_length_n-length(Y),1);
    Y = [Y ; extend_arr];
    savefilename= sprintf('cutversion_n%i.wav',i);
    savefilename = fullfile(fullpath,myfolder,savefilename);
    audiowrite(savefilename,Y(cut_s:end),Fs);

end
```

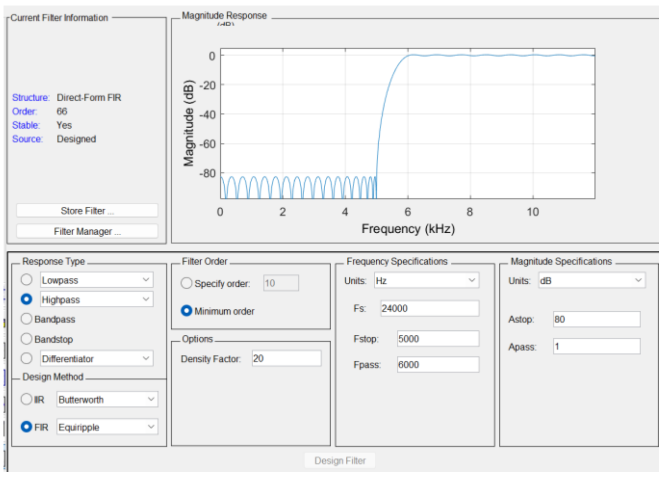


```
%% Calculating the mean power of the sound files by doing FFT
for i=1:25 % THIS FOR YES FFT FILES
    myfolder = 'GoodYes';
    myfilename= sprintf('cutversion_y%i.wav',i);
    myfilename = fullfile(fullpath,myfolder,myfilename);
    [Y,Fs] = audioread(myfilename);  % Your file name

    z=fft(Y,Fs);
    z_half_l = ceil(length(z)/2);
    z_half = z(1:z_half_l);
    sum_of_yes = sum_of_yes + (z_half.*conj(z_half));
end
    mean_yes = sum_of_yes/25;
    subplot(2,1,1);
    semilogy(mean_yes);
    ylabel('|Y(f)^2|');
    xlabel('Frequency(Hz)');
    hold on;
    ylim([0 10^5]);
    xlim([0 12000]);
for i=1:24 % THIS FOR NO FFT FILES
    myfolder = 'GoodNo';
    myfilename= sprintf('cutversion_n%i.wav',i);
    myfilename = fullfile(fullpath,myfolder,myfilename);
    [Y,Fs] = audioread(myfilename);  % Your file name

    z=fft(Y,Fs);
    z_half_l = ceil(length(z)/2);
    z_half = z(1:z_half_l);
    sum_of_no = sum_of_no + (z_half.*conj(z_half));
end
    mean_no = sum_of_no/24;
    semilogy(mean_no);
    legend('Yes','No');
```

Figure 2 - Removing sin wave and making all the files same length.

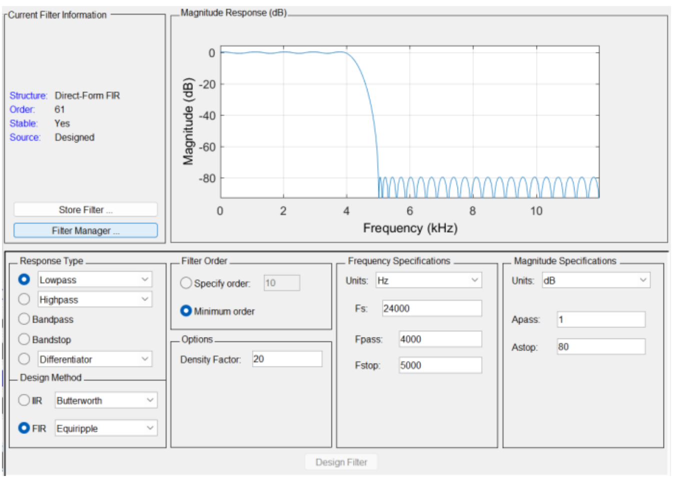
First, the sin wave noise at the beginning of each sound files is removed from the files. Afterwards, the sound file which has maximum length is found (commented part in the left-hand side figure) and all files saved to have the same length. At the right-hand side, each “Yes” and “No” files are read and the means power of them are calculated and plotted as Figure 1.

As a result, the low-pass and high-pass filters can be design according to mean power graph in Figure 1 so that the spoken word can be distinguished. Therefore, the low-pass filter’s cut off frequency is selected as 4000 Hz to 5000 Hz (stop band) while the high-pass filter’s cut off frequency is selected as 5000 Hz to 6000 Hz (stop band).



Current Filter Information
Structure: Direct-Form FIR
Order: 66
Stable: Yes
Source: Designed

Response Type: Lowpass [✓ Highpass] Bandpass Bandstop Differentiator
Design Method: IIR Butterworth [✓ FIR Equiripple]
Filter Order: Specify order: 10 [✓ Minimum order]
Options: Density Factor: 20
Frequency Specifications: Units: Hz; Fs: 24000; Fstop: 5000; Fpass: 6000
Magnitude Specifications: Units: dB; Astop: 80; Apass: 1



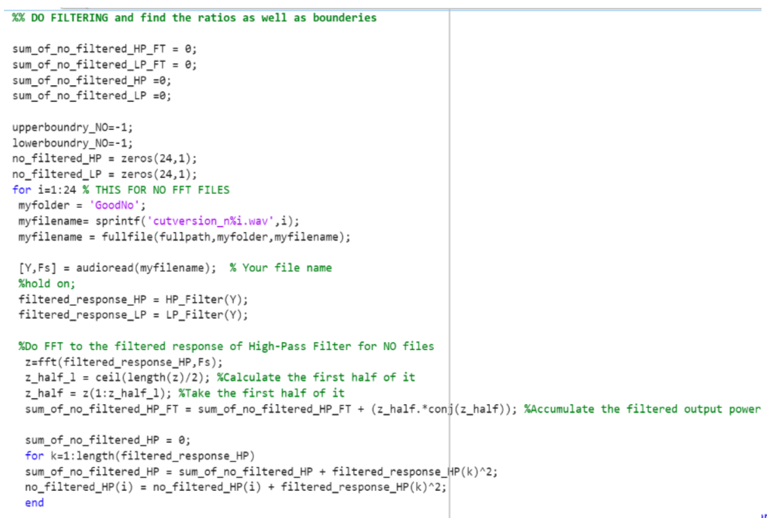
Current Filter Information
Structure: Direct-Form FIR
Order: 61
Stable: Yes
Source: Designed

Response Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator
Design Method: IIR Butterworth [✓ FIR Equiripple]
Filter Order: Specify order: 10 [✓ Minimum order]
Options: Density Factor: 20
Frequency Specifications: Units: Hz; Fs: 24000; Fpass: 4000; Fstop: 5000
Magnitude Specifications: Units: dB; Apass: 1; Astop: 80

Figure 3 - FIR high-pass and low-pass filter design on MATLAB

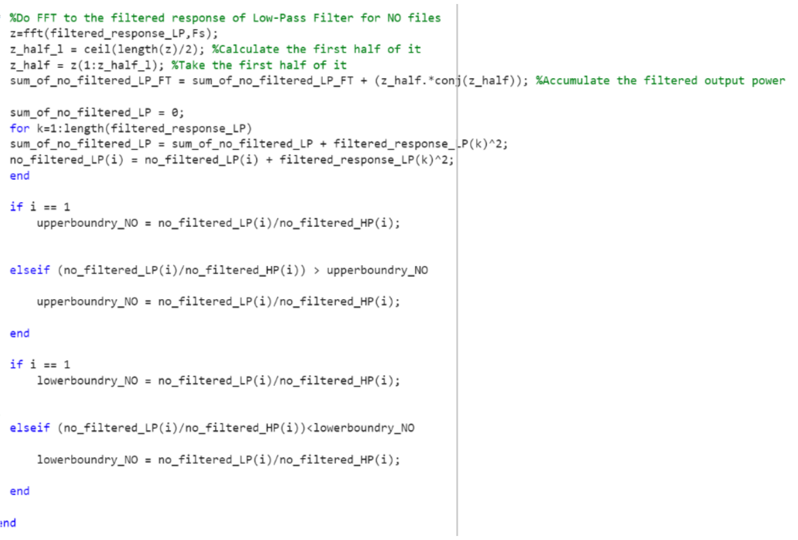
The filters are designed in Filter Design app in MATLAB as shown in Figure 3, so the specified pass and stop frequencies are entered and sampling frequency is selected as 24 kHz because the given audio files are 24kHz. Since the application is not required high sample rate, the higher frequencies were not preferred. The number of taps (order number) is selected as minimum order on the app, so the app can generate it automatically. The gap between pass band and stop band is 1 kHz, so that the numbers of orders are computational for the development board.

After designing the filters, the filters are applied to the sound files and upper and lower boundaries are found for both “Yes” and “No” files. The boundaries are going to use to detect yes or no voice. Hence, if the input voice is in the upper and lower boundary, so the input voice can be categorized as yes or no depending on which boundary it fits. Ultimately, the upper and lower boundaries are determined as maximum/minimum ratio between sum of low-pass filter and sum of high-pass filter, respectively.



```
%% DO FILTERING and find the ratios as well as bounderies

sum_of_no_filtered_HP_FT = 0;
sum_of_no_filtered_LP_FT = 0;
sum_of_no_filtered_HP =0;
sum_of_no_filtered_LP =0;

upperboundry_NO=-1;
lowerboundry_NO=-1;
no_filtered_HP = zeros(24,1);
no_filtered_LP = zeros(24,1);
for i=1:24 % THIS FOR NO FFT FILES
 myfolder = 'GoodNo';
 myfilename= sprintf('cutversion_n%i.wav',i);
 myfilename = fullfile(fullpath,myfolder,myfilename);

 [Y,Fs] = audioread(myfilename);  % Your file name
 %hold on;
 filtered_response_HP = HP_Filter(Y);
 filtered_response_LP = LP_Filter(Y);

 %Do FFT to the filtered response of High-Pass Filter for NO files
  z=fft(filtered_response_HP,Fs);
  z_half_l = ceil(length(z)/2); %Calculate the first half of it
  z_half = z(1:z_half_l); %Take the first half of it
  sum_of_no_filtered_HP_FT = sum_of_no_filtered_HP_FT + (z_half.*conj(z_half)); %Accumulate the filtered output power

  sum_of_no_filtered_HP = 0;
  for k=1:length(filtered_response_HP)
  sum_of_no_filtered_HP = sum_of_no_filtered_HP + filtered_response_HP(k)^2;
  no_filtered_HP(i) = no_filtered_HP(i) + filtered_response_HP(k)^2;
  end
```

1



```
%Do FFT to the filtered response of Low-Pass Filter for NO files
z=fft(filtered_response_LP,Fs);
z_half_l = ceil(length(z)/2); %Calculate the first half of it
z_half = z(1:z_half_l); %Take the first half of it
sum_of_no_filtered_LP_FT = sum_of_no_filtered_LP_FT + (z_half.*conj(z_half)); %Accumulate the filtered output power

sum_of_no_filtered_LP = 0;
for k=1:length(filtered_response_LP)
sum_of_no_filtered_LP = sum_of_no_filtered_LP + filtered_response_LP(k)^2;
no_filtered_LP(i) = no_filtered_LP(i) + filtered_response_LP(k)^2;
end

if i == 1
    upperboundry_NO = no_filtered_LP(i)/no_filtered_HP(i);


elseif (no_filtered_LP(i)/no_filtered_HP(i)) > upperboundry_NO

    upperboundry_NO = no_filtered_LP(i)/no_filtered_HP(i);

end

if i == 1
    lowerboundry_NO = no_filtered_LP(i)/no_filtered_HP(i);


elseif (no_filtered_LP(i)/no_filtered_HP(i))<lowerboundry_NO

    lowerboundry_NO = no_filtered_LP(i)/no_filtered_HP(i);

end

end
```

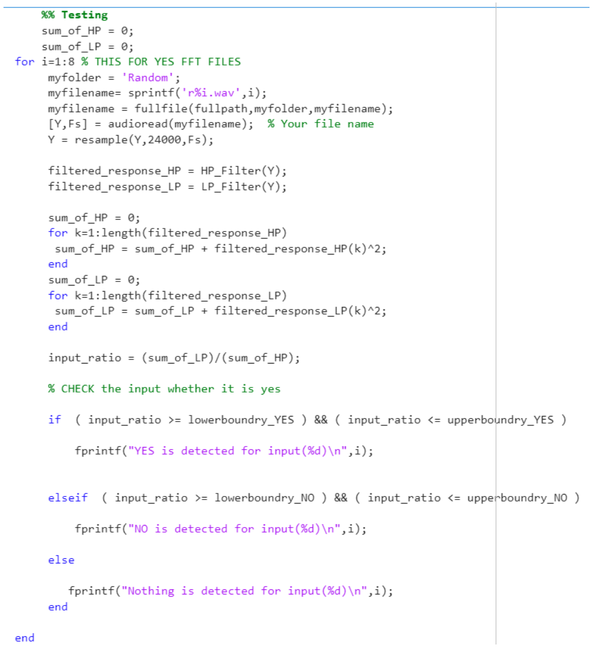
2

Figure 4 - Finding upper and lower boundary for "No" files.

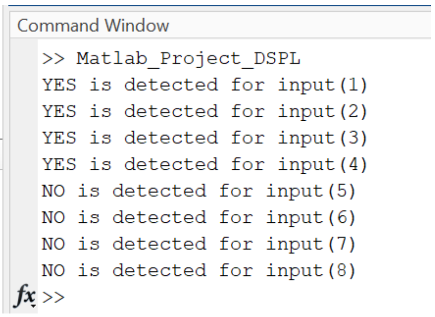
Figure 4 shows MATLAB code that applies designed low-pass and high-pass filter to each “No” files and finding maximum and minimum ratio between low-pass filter and high-pass filter. Same algorithm is applied to find boundaries of “Yes” files as well. First, the filters are applied then the response of the filter is converted into FFT domain and first half of the FFT domain is used to accumulate power. Basically, the half of the FFT response is multiped with its conjugate in order to calculated power.

Simulation of the design

After finding the boundaries four yes and four no sound files are recorded by different people to test the design. The above MATLAB code is developed to test the design. The sound files are randomly given to the design and the output of the design is given the above right hand-side. Therefore, the design detected the sound files correctly.



```
    %% Testing
    sum_of_HP = 0;
    sum_of_LP = 0;
for i=1:8 % THIS FOR YES FFT FILES
     myfolder = 'Random';
     myfilename= sprintf('r%i.wav',i);
     myfilename = fullfile(fullpath,myfolder,myfilename);
     [Y,Fs] = audioread(myfilename);  % Your file name
     Y = resample(Y,24000,Fs);

     filtered_response_HP = HP_Filter(Y);
     filtered_response_LP = LP_Filter(Y);

     sum_of_HP = 0;
     for k=1:length(filtered_response_HP)
      sum_of_HP = sum_of_HP + filtered_response_HP(k)^2;
     end
     sum_of_LP = 0;
     for k=1:length(filtered_response_LP)
      sum_of_LP = sum_of_LP + filtered_response_LP(k)^2;
     end

     input_ratio = (sum_of_LP)/(sum_of_HP);

     % CHECK the input whether it is yes

     if  ( input_ratio >= lowerboundry_YES ) && ( input_ratio <= upperboundry_YES )

         fprintf("YES is detected for input(%d)\n",i);


     elseif  ( input_ratio >= lowerboundry_NO ) && ( input_ratio <= upperboundry_NO )

         fprintf("NO is detected for input(%d)\n",i);

     else

        fprintf("Nothing is detected for input(%d)\n",i);
     end

end
```



```
>> Matlab_Project_DSPL
YES is detected for input(1)
YES is detected for input(2)
YES is detected for input(3)
YES is detected for input(4)
NO is detected for input(5)
NO is detected for input(6)
NO is detected for input(7)
NO is detected for input(8)
>>
```

Please check the appendices for recorded sound files.

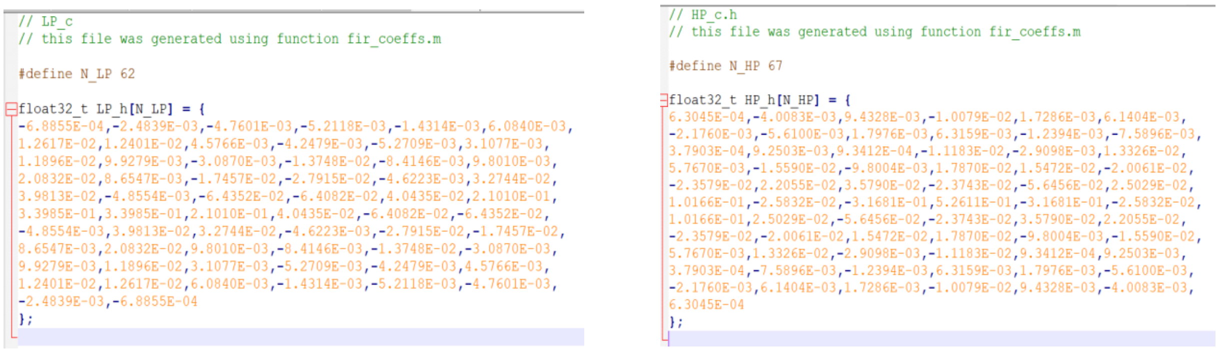
The reason why FIR filter is selected for this application
FIR filters are often used in applications where a linear phase response is required, such as in audio and image processing. The key characteristic of FIR filters is that they are always stable, and their frequency response can be controlled precisely by the number of filter coefficients or taps.

On the other hand, IIR filter stands for "Infinite Impulse Response" filter. It is a type of digital filter that has an impulse response that extends to infinity. The output of an IIR filter depends on past input samples as well as past output samples, which introduces feedback in the filter structure. IIR filters are commonly used in applications where a more complex frequency response is required, such as in control systems and communication systems. The key characteristic of IIR filters is that they are more computationally efficient than FIR filters, but they can be unstable if their coefficients are not chosen carefully.

Since the number of taps the designed FIR filter is computationally small enough to run on DSP board, FIR filter is chosen for this application to avoid instability.

Implementation of the design on the board

The designed filters' sampling frequencies are changed to 32 kHz because the development board supports 8 kHz, 32 kHz, 48 kHz, and 96 kHz sampling frequencies. Both of the filters' coefficients is converted into C header files to be used in main program by using the provided MATLAB file "fir_coeffs.m".

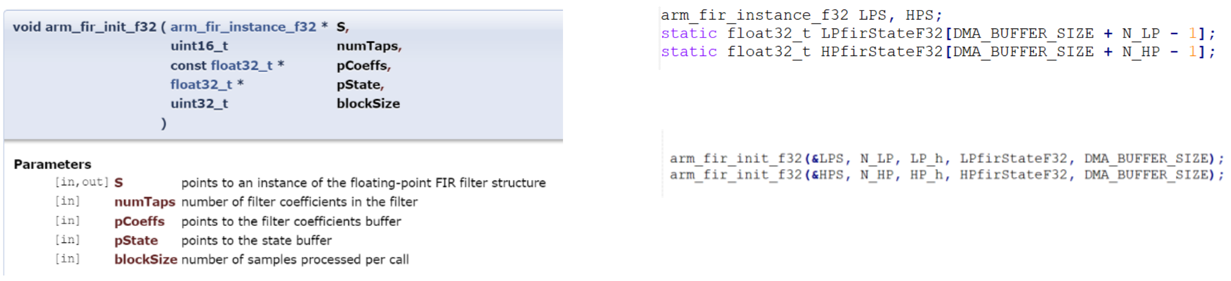


```
// LP_c
// this file was generated using function fir_coeffs.m

#define N_LP 62

float32_t LP_h[N_LP] = {
-6.8855E-04,-2.4839E-03,-4.7601E-03,-5.2118E-03,-1.4314E-03,6.0840E-03,
1.2617E-02,1.2401E-02,4.5766E-03,-4.2479E-03,-5.2709E-03,3.1077E-03,
1.1896E-02,9.9279E-03,-3.0870E-03,-1.3748E-02,-8.4146E-03,9.8010E-03,
2.0832E-02,8.6547E-03,-1.7457E-02,-2.7915E-02,-4.6223E-03,3.2744E-02,
3.9813E-02,-4.8554E-03,-6.4352E-02,-6.4082E-02,4.0435E-02,2.1010E-01,
3.3985E-01,3.3985E-01,2.1010E-01,4.0435E-02,-6.4082E-02,-6.4352E-02,
-4.8554E-03,3.9813E-02,3.2744E-02,-4.6223E-03,-2.7915E-02,-1.7457E-02,
8.6547E-03,2.0832E-02,9.8010E-03,-8.4146E-03,-1.3748E-02,-3.0870E-03,
9.9279E-03,1.1896E-02,3.1077E-03,-5.2709E-03,-4.2479E-03,4.5766E-03,
1.2401E-02,1.2617E-02,6.0840E-03,-1.4314E-03,-5.2118E-03,-4.7601E-03,
-2.4839E-03,-6.8855E-04
};
```

```
// HP_c.h
// this file was generated using function fir_coeffs.m

#define N_HP 67

float32_t HP_h[N_HP] = {
6.3045E-04,-4.0083E-03,9.4328E-03,-1.0079E-02,1.7286E-03,6.1404E-03,
-2.1760E-03,-5.6100E-03,1.7976E-03,6.3159E-03,-1.2394E-03,-7.5896E-03,
3.7903E-04,9.2503E-03,9.3412E-04,-1.1183E-02,-2.9098E-03,1.3326E-02,
5.7670E-03,-1.5590E-02,-9.8004E-03,1.7870E-02,1.5472E-02,-2.0061E-02,
-2.3579E-02,2.2055E-02,3.5790E-02,-2.3743E-02,-5.6456E-02,2.5029E-02,
1.0166E-01,-2.5832E-02,-3.1681E-01,5.2611E-01,-3.1681E-01,-2.5832E-02,
1.0166E-01,2.5029E-02,-5.6456E-02,-2.3743E-02,3.5790E-02,2.2055E-02,
-2.3579E-02,-2.0061E-02,1.5472E-02,1.7870E-02,-9.8004E-03,-1.5590E-02,
5.7670E-03,1.3326E-02,-2.9098E-03,-1.1183E-02,9.3412E-04,9.2503E-03,
3.7903E-04,-7.5896E-03,-1.2394E-03,6.3159E-03,1.7976E-03,-5.6100E-03,
-2.1760E-03,6.1404E-03,1.7286E-03,-1.0079E-02,9.4328E-03,-4.0083E-03,
6.3045E-04
};
```

Figure 5 - Header file of low-pass and high-pass filters

Both converted header files are shown in Figure 5. In order to use the filters efficiently on the board CMSIS-DSP Library is used.



```
void arm_fir_init_f32 ( arm_fir_instance_f32 *  S,
                        uint16_t                numTaps,
                        const float32_t *       pCoeffs,
                        float32_t *             pState,
                        uint32_t                blockSize
                      )
```

Parameters

[in,out]	S	points to an instance of the floating-point FIR filter structure
[in]	numTaps	number of filter coefficients in the filter
[in]	pCoeffs	points to the filter coefficients buffer
[in]	pState	points to the state buffer
[in]	blockSize	number of samples processed per call

```
arm_fir_instance_f32 LPS, HPS;
static float32_t LPfirStateF32[DMA_BUFFER_SIZE + N_LP - 1];
static float32_t HPfirStateF32[DMA_BUFFER_SIZE + N_HP - 1];
```

```
arm_fir_init_f32(&LPS, N_LP, LP_h, LPfirStateF32, DMA_BUFFER_SIZE);
arm_fir_init_f32(&HPS, N_HP, HP_h, HPfirStateF32, DMA_BUFFER_SIZE);
```

Figure 6 - Declaration of the filters

On the left-hand side, the initialization of the filter is shown in the CMSIS-DSP library, so at the right hand-side both filters instance is declared and state buffers for both filters are defined as well. The length of the pState buffer is suggested as numTaps+blockSize-1 in the manual of the library. After defining the required variables, the FIR filters are initialized in the main program as shown on the above bottom right-hand side.

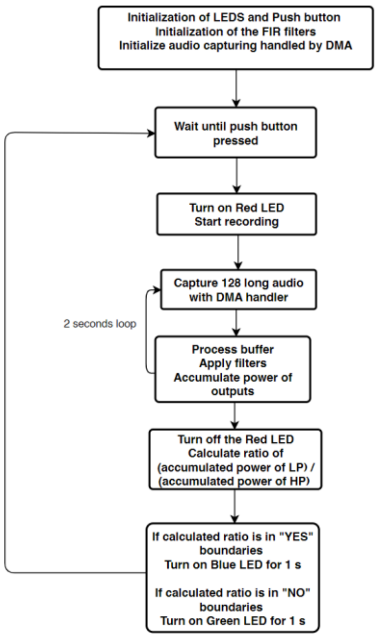

Initialization of LEDS and Push button
Initialization of the FIR filters
Initialize audio capturing handled by DMA
Wait until push button pressed
Turn on Red LED
Start recording
Capture 128 long audio with DMA handler
2 seconds loop
Process buffer
Apply filters
Accumulate power of outputs
Turn off the Red LED
Calculate ratio of (accumulated power of LP) / (accumulated power of HP)
If calculated ratio is in "YES" boundaries
Turn on Blue LED for 1 s
If calculated ratio is in "NO" boundaries
Turn on Green LED for 1 s

Figure 7 - Flow chart of the main program

Figure 7 shows the flow of the main program running on the board. After initialization of the necessary peripherals, the program waits until the push button is pressed. After button is pressed, the recorded buffer is processes while red LED is turned on for 2 seconds. When the 2 seconds is over, the program calculated the ratio of accumulated outputs' power of filters. If the ratio is in "Yes" boundaries, blue LED is turned on for 1 second, else if the ratio is in "No" boundaries, green led is turned on for 1 second while none of LEDs is turned on if the ratio is neither in "Yes" nor "No" boundaries.

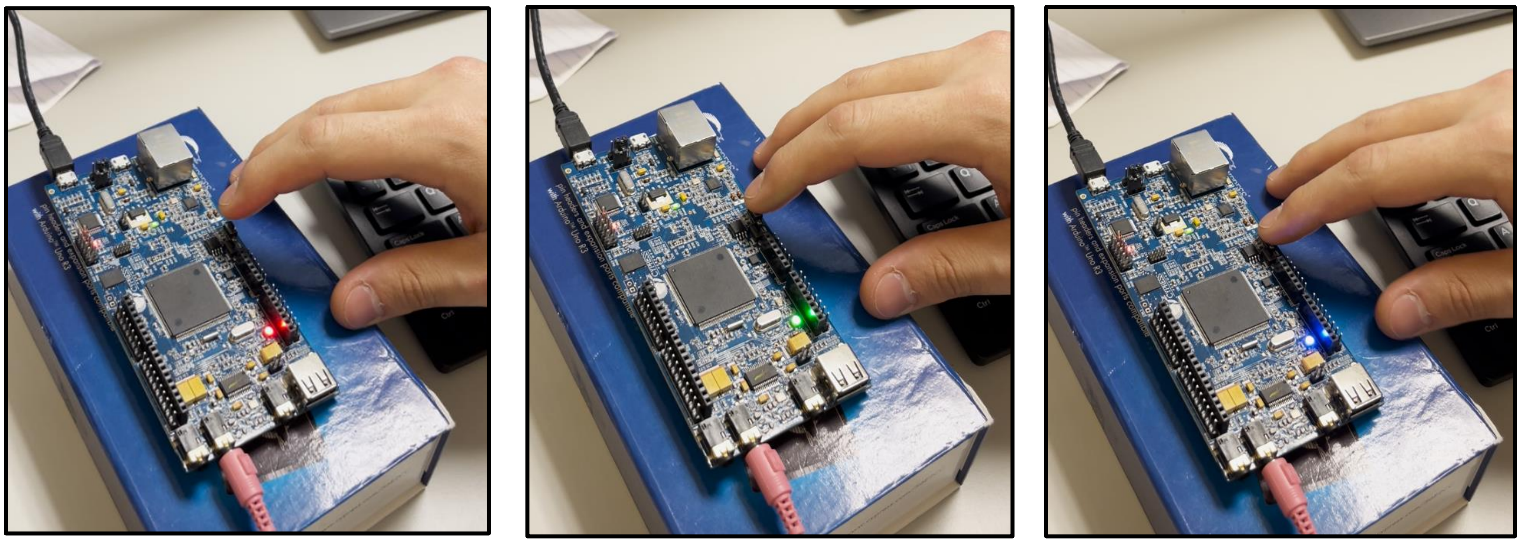


Figure 8 - Demonstration of the program

Figure 8 includes some snapshots from the demonstration video of the program, the first snapshot shows the recording stage while the other snapshots represent the evaluation of the capture audio. Please have a look at the appendices for the demonstration video that shows both "Yes" or "No" evaluation successfully.

Result of hundred recording that has fifty “Yes” and “No” spoken words.		**Actual values**	
		YES	**NO**
Predicted Values	**YES**	38	31
	NO	12	19

Table 1 - Confusion matrix

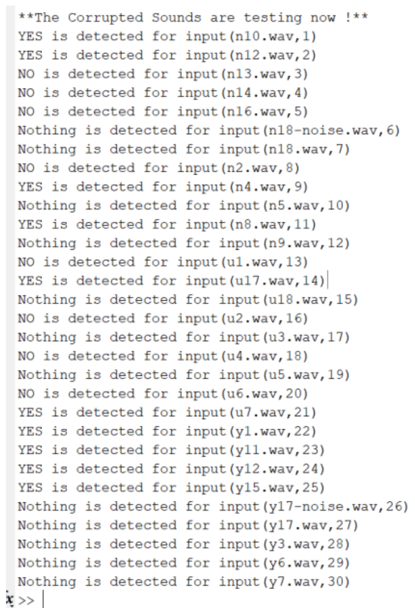
Table 1 shows the confusion matrix of the implemented program on the board. Therefore, the test contains hundred spoken words which are fifty of “Yes” and fifty of “No” words. The program shows almost %70 accuracy. It is good to point out that the programs have lower accuracy to detect “No” words.

Test on corrupted sound files and suggested solutions

The below table shows MATLAB simulation is done on the corrupted signal by the developed program.

Filename	Method	Detected as	Notes
n10.wav	Additive Gaussian White Noise added	Yes ×	Completely corrupted by noise
n12.wav	Audio volume is wrong	Yes ✓	This file is actually yes file
n13.wav	Audio volume is wrong	No ✓	
n14.wav	Audio volume is wrong	No ✓	
n16.wav	Audio volume is wrong	No ✓	
n18-noise.wav		Nothing is detected ×	Completely corrupted by noise
n18.wav		Nothing is detected ×	Completely corrupted by noise
n2.wav	Wrong tone	No ✓	
n4.wav	Wrong tone	Yes ×	
n5.wav	Sine wave noise added to signal	Nothing is detected ×	Heavily noised but could be detected
n8.wav	Additive Gaussian White Noise added	Yes ×	Heavily noised but could be detected
n9.wav	Additive Gaussian White Noise added	Nothing is detected ×	Heavily noised but could be detected
u1.wav	Tone + unknown sound	No ×	Unknown word is detected as “No”
u17.wav	Tone + unknown sound	Yes ×	Completely corrupted by noise
u18.wav	Tone + unknown sound	Nothing is detected ✓	
u2.wav	Tone + unknown sound	No ×	Unknown word is detected as “No”

u3.wav	Tone + unknown sound	Nothing is detected ✓	
u4.wav	Tone + unknown sound	No ×	Unknown word is detected as “No”
u5.wav	Tone + unknown sound	Nothing is detected ✓	
u6.wav	Tone + unknown sound	No ×	Unknown word is detected as “No”
u7.wav	Tone + unknown sound	Yes ×	Unknown word is detected as “Yes”
y1.wav	Wrong tone	Yes ✓	
y11.wav	Additive Gaussian White Noise added	Yes ✓	
y12.wav	Audio volume is wrong	Yes ✓	
y15.wav	Audio volume is wrong	Yes ✓	
y17-noise.wav	Tone + signal corrupted by chirp	Nothing is detected ×	Completely signal corrupted by chirp
y17.wav	Tone + signal corrupted by chirp	Nothing is detected ×	Completely signal corrupted by chirp
y3.wav	Wrong tone	Nothing is detected ×	
y6.wav	Sine wave noise added to signal	Nothing is detected ×	Heavily noised
y7.wav	Sine wave noise added to signal	Nothing is detected ×	Heavily noised



```
**The Corrupted Sounds are testing now !**
YES is detected for input(n10.wav,1)
YES is detected for input(n12.wav,2)
NO is detected for input(n13.wav,3)
NO is detected for input(n14.wav,4)
NO is detected for input(n16.wav,5)
Nothing is detected for input(n18-noise.wav,6)
Nothing is detected for input(n18.wav,7)
NO is detected for input(n2.wav,8)
YES is detected for input(n4.wav,9)
Nothing is detected for input(n5.wav,10)
YES is detected for input(n8.wav,11)
Nothing is detected for input(n9.wav,12)
NO is detected for input(u1.wav,13)
YES is detected for input(u17.wav,14)
Nothing is detected for input(u18.wav,15)
NO is detected for input(u2.wav,16)
Nothing is detected for input(u3.wav,17)
NO is detected for input(u4.wav,18)
Nothing is detected for input(u5.wav,19)
NO is detected for input(u6.wav,20)
YES is detected for input(u7.wav,21)
YES is detected for input(y1.wav,22)
YES is detected for input(y11.wav,23)
YES is detected for input(y12.wav,24)
YES is detected for input(y15.wav,25)
Nothing is detected for input(y17-noise.wav,26)
Nothing is detected for input(y17.wav,27)
Nothing is detected for input(y3.wav,28)
Nothing is detected for input(y6.wav,29)
Nothing is detected for input(y7.wav,30)
>> 
```

The output of the simulation is shown above figure under the table. For better understanding, the table is created according to results of the simulation. It can be seen that the developed program can do false detection when the inputs are noised, heavily noised, and completely corrupted. In addition, the application sometimes does false detection on wrong tone as well.

For corrupted signals that are hard to solve, you might want to consider using a combination of some methods. For example, you can use an FFT-based method to analyse the frequency content of the signal and identify the dominant noise components. Then, you can design an adaptive filter (e.g., an LMS adaptive filter) to target these noise components and remove them from the signal. In some cases, a combination of FIR and IIR filters may be used to achieve the desired filtering performance while maintaining computational efficiency.

Summary

In this academic report, we present a speech recognition system using the FM4-S6E2CC-ETH device to distinguish between the spoken words "yes" and "no". Sound files for both words are provided to plot their mean power spectra using MATLAB, which then informs the design of low-pass and high-pass filters with appropriate cut-off frequencies. The performance of the system is evaluated using 8 pre-recorded sound files, with the results detailed in the report. The input voice detection relies on the ratio of low-pass and high-pass filtered signals to determine the spoken word as either "yes" or "no". Visual feedback is given through the activation of blue and green LEDs on the board for "yes" and "no" respectively. This work contributes to the development of efficient and reliable speech recognition systems with potential real-world applications.

Appendices

Keil studio files and MATLAB files

Published_Project

Demonstration video

Demo.MOV