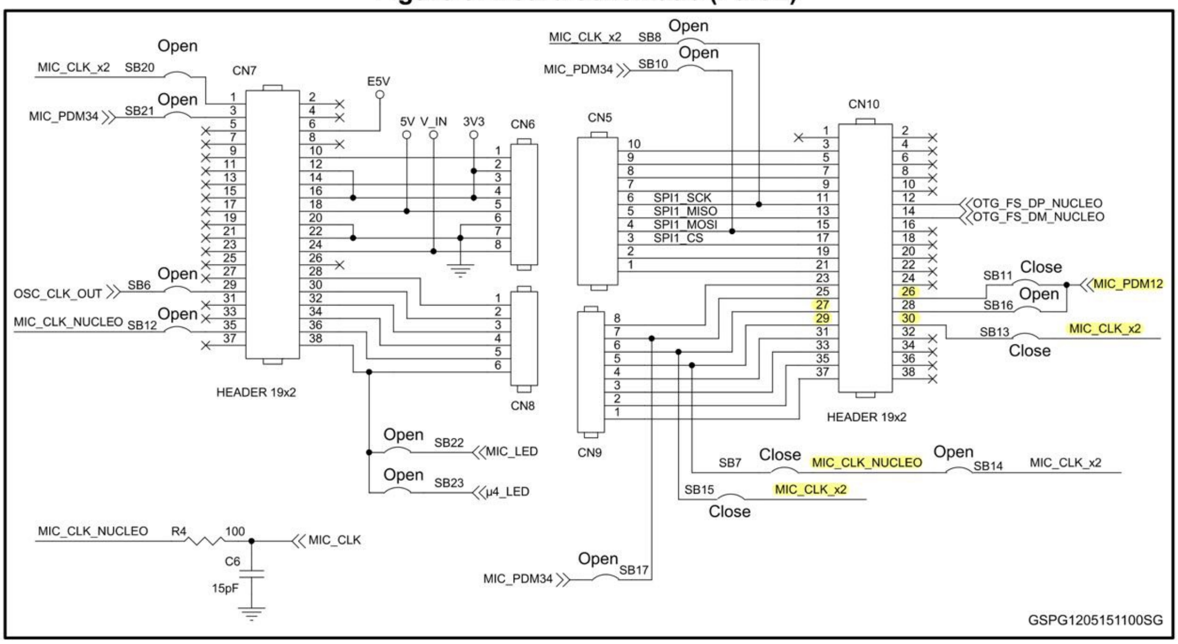
🔊

Sound Localization STM32

Connection between CCA02M1 and STM32

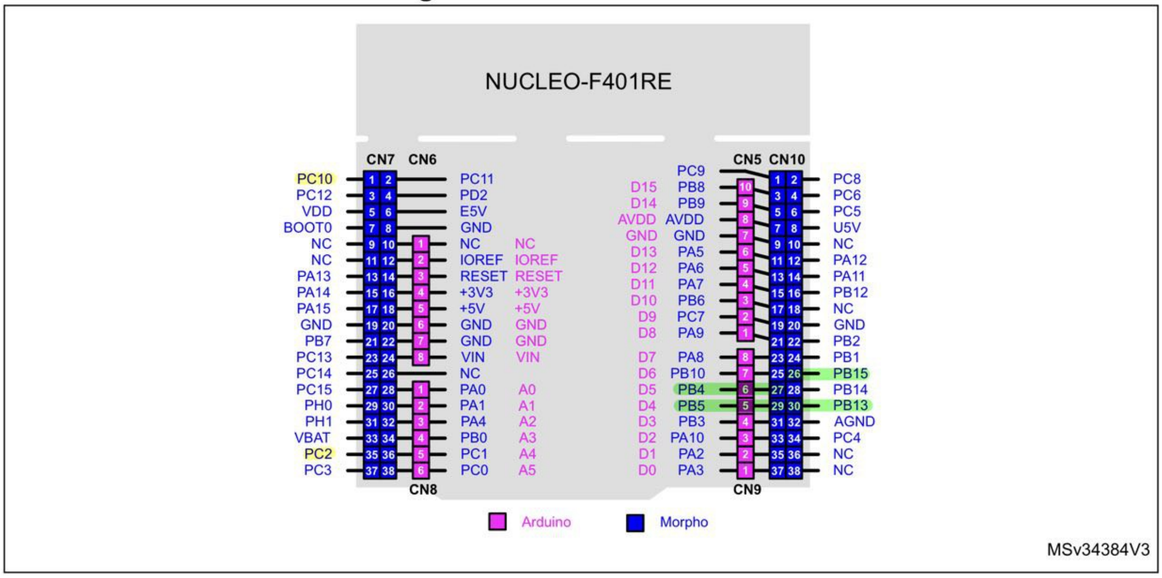
As part of our project, a NUCLEO-F401RE microcontroller was used with the X-NUCLEO-CCA02M1 extension board. This board, produced by STMicroelectronics, is equipped with two MEMS microphones. The physical connection between the two devices was established using morpho pin connectors. To better understand the communication between the X-NUCLEO-CCA02M1 and the STM32, the datasheets for both devices were consulted. These documents provided information on the communication protocols and interfaces used by the devices, as well as other relevant details.

Figure 6: Board schematic (Part 2)



CCA02M1 schematic, pin used are highlight

Figure 18. NUCLEO-F401RE



NUCLEO_F401RE schematic, pin used are highlight

To allow communication between the X-NUCLEO-CCA02M1 extension board and the STM32 microcontroller, we decided to use the Serial Peripheral Interface (SPI) protocol

Communication protocol

SPI is a synchronous serial communication protocol that is commonly used in embedded systems and other applications where high-speed data transfer is required. It allows multiple devices to be connected to a single bus, with each device able to send and receive data over the same set of wires.

One of the main advantages of SPI is its high data transfer rates, which can be as fast as several megabits per second. This makes it suitable for applications that require fast communication between devices. Another advantage of SPI is that it requires only a few wires to establish a connection, making it a simple and cost-effective solution for device communication.

In our project, we chose to use SPI because it offered the combination of high data transfer rates and simplicity that was necessary to meet the requirements of the project. By implementing the SPI protocol, we were able to establish reliable communication between the X-NUCLEO-CCA02M1 and the STM32, enabling the transfer of data between the two devices.

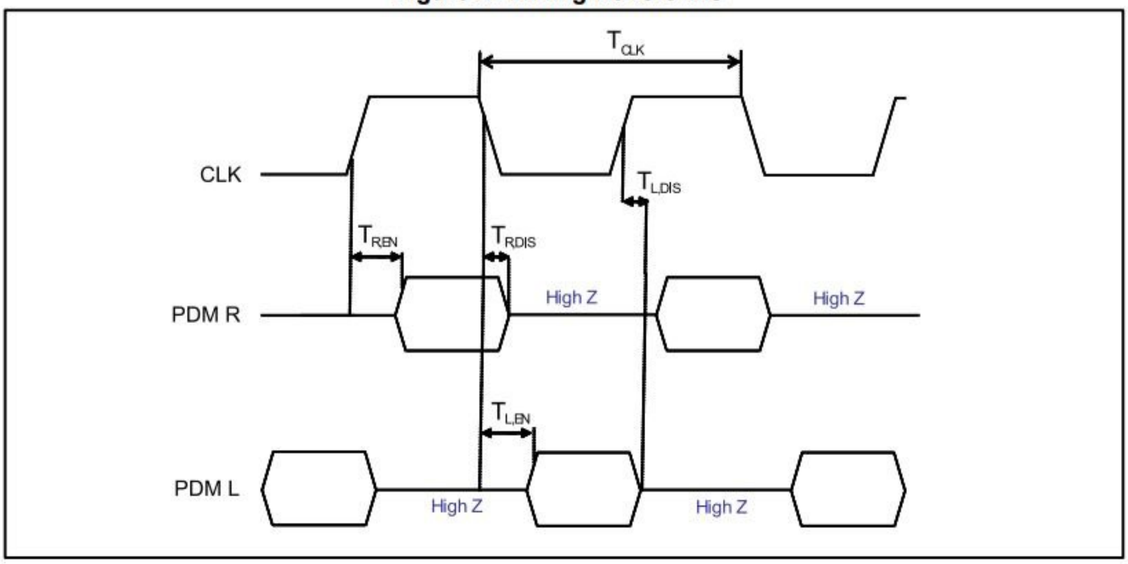
Pin configuration

To establish communication between the X-NUCLEO-CCA02M1 extension board and the STM32 microcontroller, we used the Serial Peripheral Interface (SPI) protocol. By examining the schematics of the X-NUCLEO-CCA02M1, we were able to identify the pins that were compatible with the SPI2 protocol and used these to establish a connection between the two devices.

One of the signals that were identified in the schematics was **MIC_CLK_NUCLEO**, which represents the **sampling frequency** of both microphones. This clock is generated by the NUCLE_F401RE as a pulse-width modulated (PWM) signal and is used to synchronize the data transfer between the microphones and the microcontroller.

Another signal identified in the schematics was **MIC_CLK_x2**, which is a PWM signal that represents the sampling frequency of the SPI. This sampling frequency needs to be twice the sampling frequency of the microphones since the data coming from the two microphones are interleaved, as shown in the following figure.

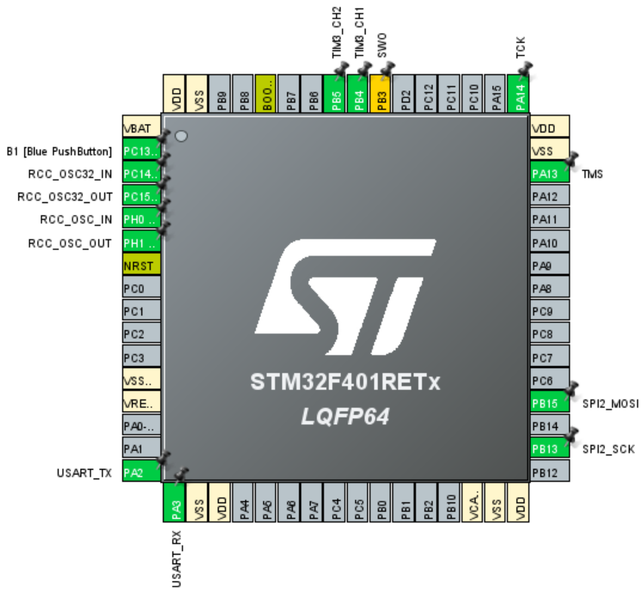
Figure 2: Timing waveforms



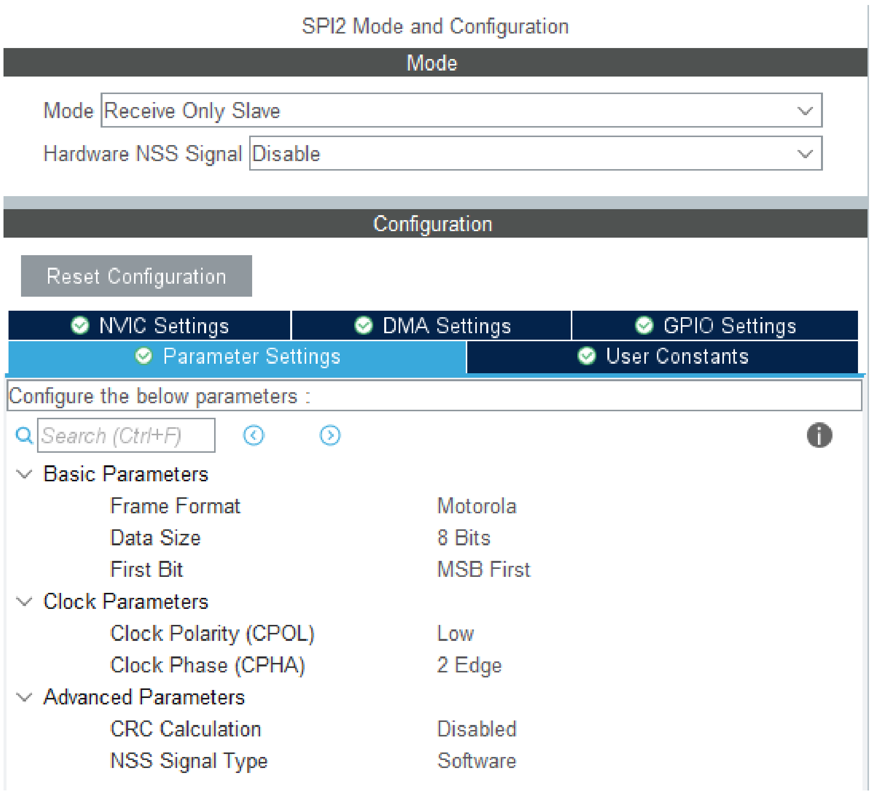
Output PDM signal

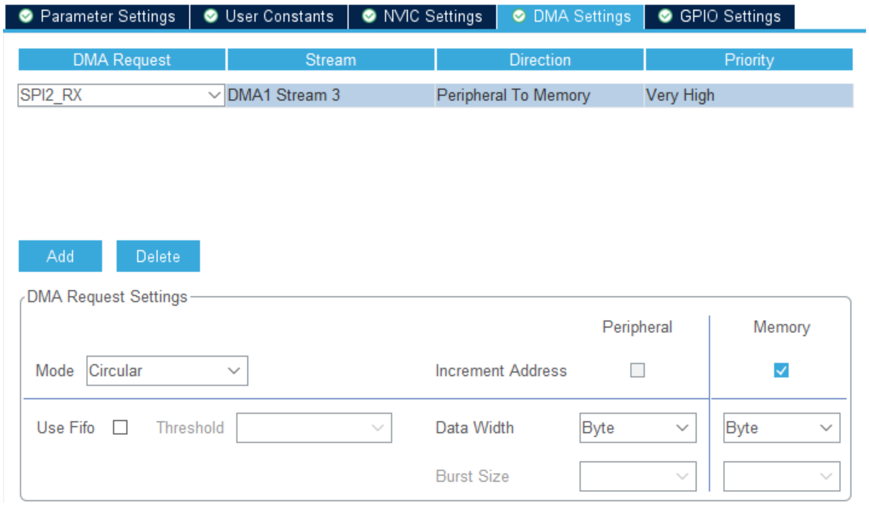
Finally, the **MIC_PDM12** signal represents the output **pulse density modulation** (PDM) signal that is read by the SPI. This signal contains the audio that has been captured by the microphones.

All the configurations were set using STM32CubeMX:


TIM3_CH2
TIM3_CH1
SWO
TCK
B1 [Blue PushButton]
RCC_OSC32_IN
RCC_OSC32_OUT
RCC_OSC_IN
RCC_OSC_OUT
USART_TX
USART_RX
TMS
SPI2_MOSI
SPI2_SCK
STM32F401RETx
LQFP64

PIN setting


SPI2 Mode and Configuration
Mode
Mode Receive Only Slave
Hardware NSS Signal Disable
Configuration
Reset Configuration
NVIC Settings
DMA Settings
GPIO Settings
Parameter Settings
User Constants
Configure the below parameters :
Search (Ctrl+F)
Basic Parameters
Frame Format Motorola
Data Size 8 Bits
First Bit MSB First
Clock Parameters
Clock Polarity (CPOL) Low
Clock Phase (CPHA) 2 Edge
Advanced Parameters
CRC Calculation Disabled
NSS Signal Type Software

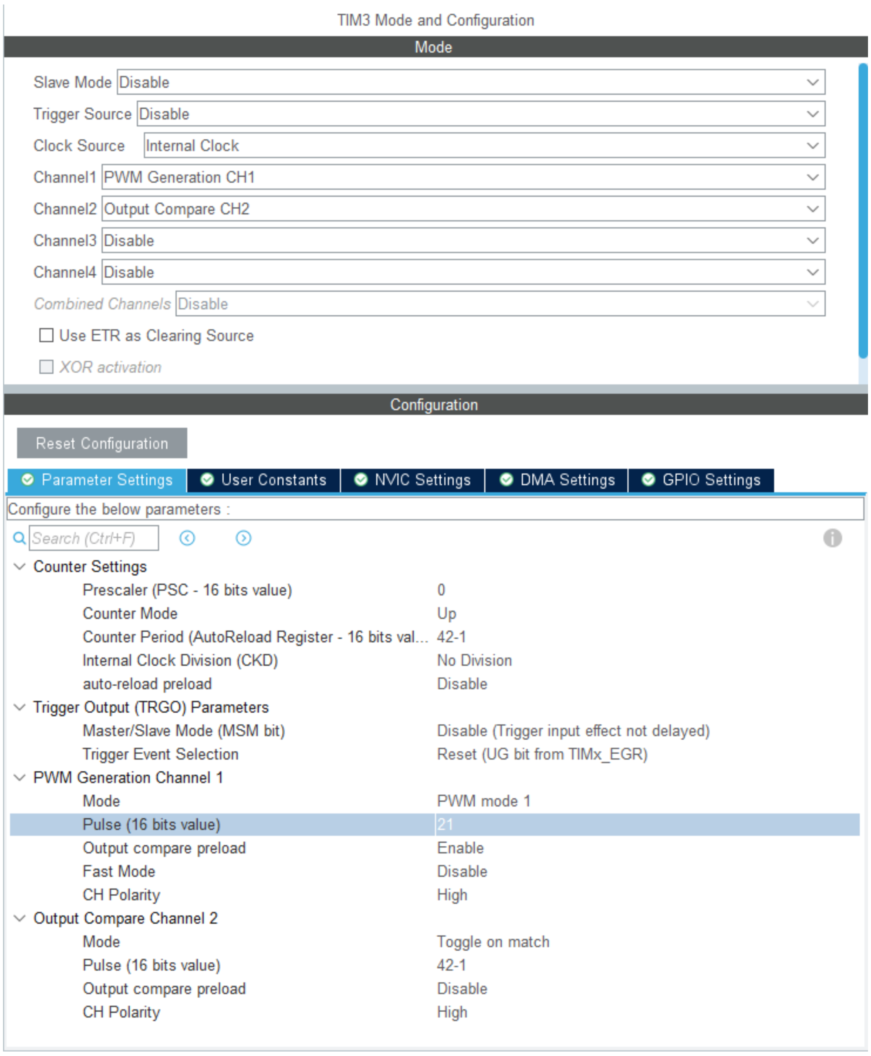


SPI2 configuration

The configuration of the SPI is depicted in the figure. To efficiently fill the memory, we have chosen to utilize **DMA** in a **circular mode**. This allows for a continuous flow of data to be sent to the STM32.

The SPI is set to operate in **receive-only slave mode**, and the TIM3 is utilized to generate the signal that controls the sampling frequency at which the SPI is capturing the signal. This signal is received using CCA02M1 PINs.

Another important parameter to choose was the size of the **SPI buffer**, longer the buffer more time is needed to fill the buffer, but if it is too long too much delay is added to the response of the system. A good trade-off that we reached during multiple tests was **128**.



TIM 3 Configuration

To generate the **TIM3** signals that were needed for our project, we used an 84 MHz as an internal clock reference.

To obtain a signal around 2 MHz , we used the auto-reload register in TIM3 and configured channel 2 in **output compare mode**. The signal is **toggled on each match**, which allowed us to create the 2 MHz signal that we needed.

To obtain a 1 MHz signal, we used channel 1 in **pulse-width modulated** (PWM) mode with a **50% duty cycle**. We were able to create the correct signal setting pulse value.

$$F_{SPI,sampling} = \frac{84MHz}{42} = 2MHz$$

$$F_{mic} = \frac{2MHz}{2} = 1MHz$$

Implementing interrupt for SPI

To implement an interrupt when the SPI has filled the buffer we write the following code in "stm32f4xx_it.c".

```
/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN PFP */
__weak void SPI2_DMA_RX_Complete_Callback(void);
/* USER CODE END PFP */

/**
  * @brief This function handles DMA1 stream3 global interrupt.
  */
void DMA1_Stream3_IRQHandler(void)
{
  /* USER CODE BEGIN DMA1_Stream3_IRQn 0 */

  // Check if the DMA transfer is complete
  if(__HAL_DMA_GET_IT_SOURCE(&hdma_spi2_rx, DMA_IT_TC) != RESET){
    SPI2_DMA_RX_Complete_Callback();
  }

  /* USER CODE END DMA1_Stream3_IRQn 0 */
  HAL_DMA_IRQHandler(&hdma_spi2_rx);
  /* USER CODE BEGIN DMA1_Stream3_IRQn 1 */

  /* USER CODE END DMA1_Stream3_IRQn 1 */
}

/* USER CODE BEGIN 1 */
/**
  * @brief  Rx complete callback in DMA mode
  * @param  None
  * @retval None
  */
__weak void SPI2_DMA_RX_Complete_Callback()
{
  /* NOTE : This function Should not be modified, when the callback is needed,
            the SPI2_DMA_RX_Complete_Callback could be implemented in the user file
   */
}
/* USER CODE END 1 */
```

From SPI buffer to PDM signal

The SPI convert 8 bit in an uint8 and then fill the buffer. To obtain the originals signals a re-conversion to 8-bit from uint8 is needed. Is important to remember that the PDM signals are interleaved as shown in the figure.

Bit:	15	14	13	12	11	...	0
Content:	$M1\_b_N$	$M2\_b_N$	$M1\_b_{N+1}$	$M2\_b_{N+1}$	$M1\_b_{N+2}$	...	$M2\_b_{N+7}$

PDM data arrangement

To obtain two signals a de-interleave stage is added after the conversion from uint8 to bit. Now is possible to process both PDM signals.

Decimation and filtering stage

Decimation

Decimation refers to the process of reducing the sample rate of a signal **by a factor of N**, where N is an integer. This process is often used in digital signal processing to reduce the amount of data that needs to be processed or stored, while still preserving the essential characteristics of the original signal.

In our case, the decimation was needed to reduce to elevate the number of data to be processed. The only parameter to be chosen is the decimation factor N. To choose this parameter we analyze the physical phenomen. We measure the distance between the 2 microphones to be around 1.5 cm, and then we use the standard value for the speed of the sound Vs = 340 m/s.

$$t = \frac{1.5}{340} = 4.41 * 10^{-5}s$$

Where t is the minimum time delay between the two microphones. Since we want to have more samples during this time we need to choose a sampling frequency able to provide at least 5 measurements between the delay. Since we are using a 1MHz sampling rate N is chosen to be 8.

$$f_{equivalent} = \frac{1MHz}{N} = 125kHz$$

$$T = \frac{1}{125kHz} = 8 * 10^{-6}s$$

$$samples = \frac{T}{t} = 5.5$$

Filtering

Low-pass filtering of audio signals is often used to remove high-frequency noise or unwanted high-frequency components from the signal. Since our signal is sampled at a high frequency we need to remove the components that are not used.

During our research, we find that a finger snap frequency is between 1500 Hz to 3500 Hz. This is the only constraint that we had in choosing the passband frequency. One other important consideration is that we don't have problems with the reconstruction of the signal since the Nyquist frequency is much greater than our wanted frequency.

An important choice was which filter is better for our implementation. In the end, we decide to use an FIR filter, since they can be executed together with the decimation, and even if FIR filters have higher cost implementation, this provides us with an easier way to decimate and filter the data together compare with the IIR filter. Another relevant factor was that an IIR filter has an unequal delay at different frequencies, while an FIR filter has a consistent delay at every frequency.

The FIR filter is obtained as a vector of coefficients, this can be implemented as int or float (in the code both vectors are present). We tried both implementations, measuring the time needed to process the signals using TIM4. As we think the float implementation was more time expensive but gives better results, the PCM signals look smoother. Another important step was to try to reduce as possible to the number of coefficients needed for the FIR, at the need we decide to use these parameters:

$$PassBand : 0Hz \text{ to } 8000Hz$$

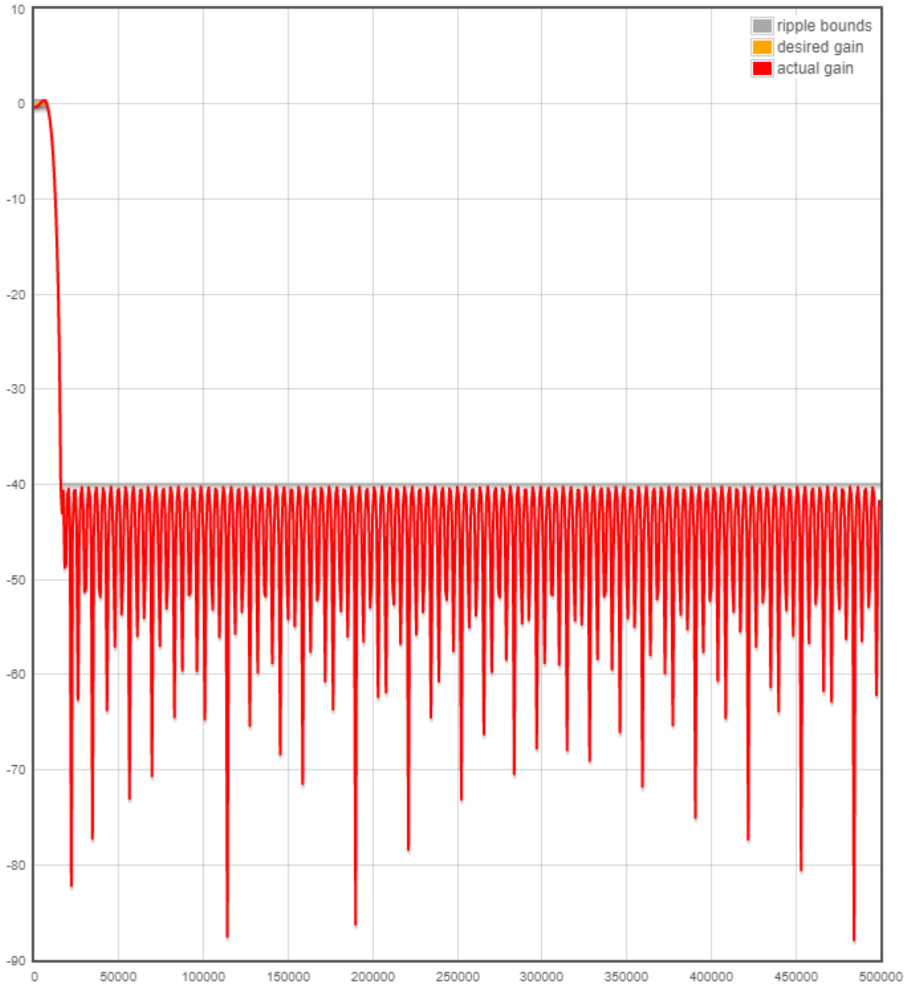
$$RolloffBand : 8000Hz \text{ to } 15625Hz$$

$$StopBand : 15625Hz \text{ to } F_{nyquist,MIC}$$

Where:

$$F_{nyquist} = \frac{F_{MIC}}{2} = \frac{1MHz}{2} = 500kHz$$

Is important to notice that the filtering is executed before decimation. This mean that the sampling frequency is still 1MHz and not the equivalent one 1MHz/N.



Bode diagram of FIR filter designed

FIR filter and decimation implementation

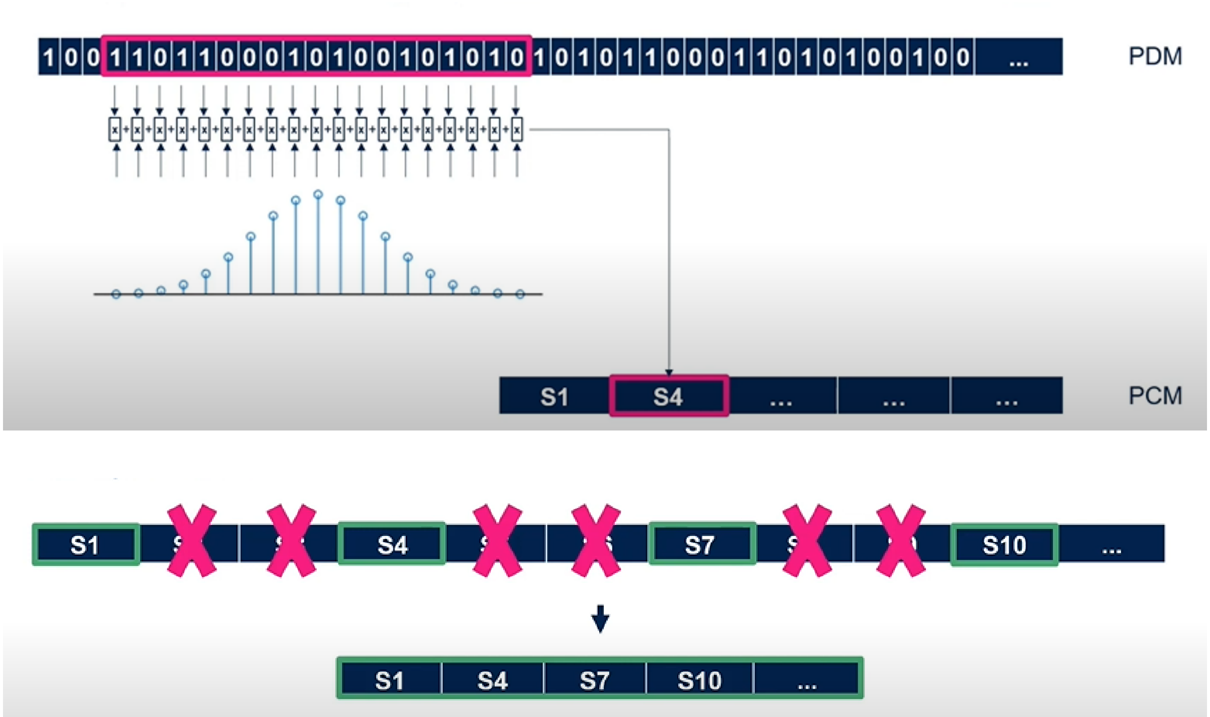
Both FIR filter and decimation were implemented with the same function. Is important to execute first the filtering, and after the decimation, this is done to avoid high-frequency signals to produce noise in the final signal. Then is important to have a stop frequency lower than the **equivalent Nyquist frequency** in order to avoid aliasing.

$$F_{equivalent} = \frac{F_{MIC,sampling}}{N} = 125000Hz$$

$$F_{nyquist,equivalent} = \frac{F_{MIC,sampling}}{N * 2} = 62500Hz$$

Note that to avoid aliasing:

$$StopFrequency = 15625Hz < F_{nyquist,equivalent}$$



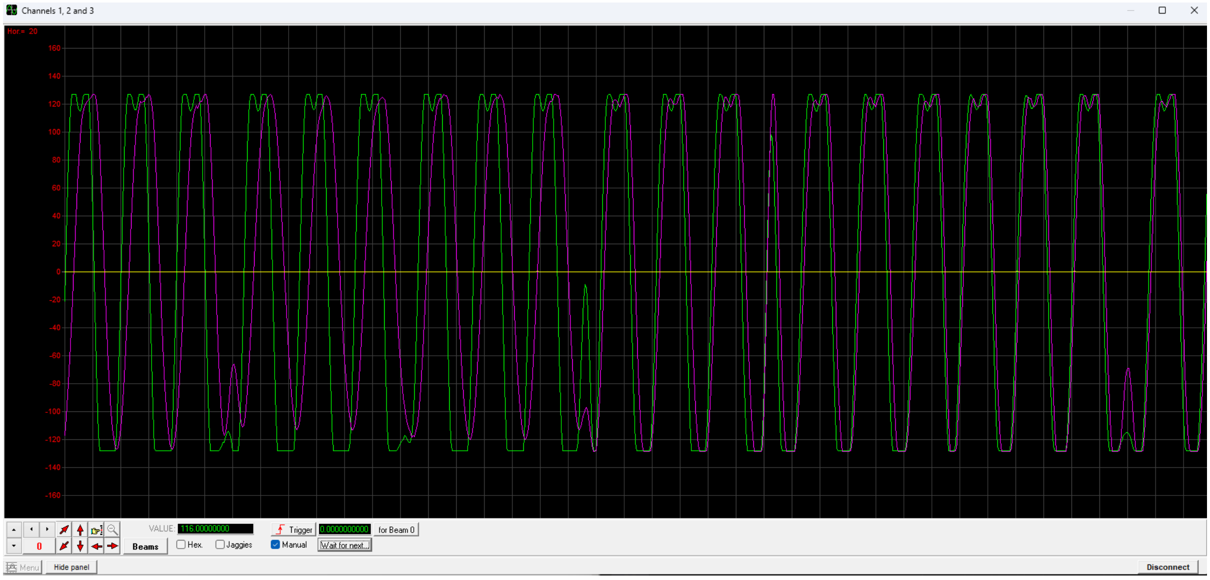
Filtering and decimation phase implemented together in case of N = 3

As shown in the figure, since we have implemented decimation and filtering together only samples that will keep are computed.

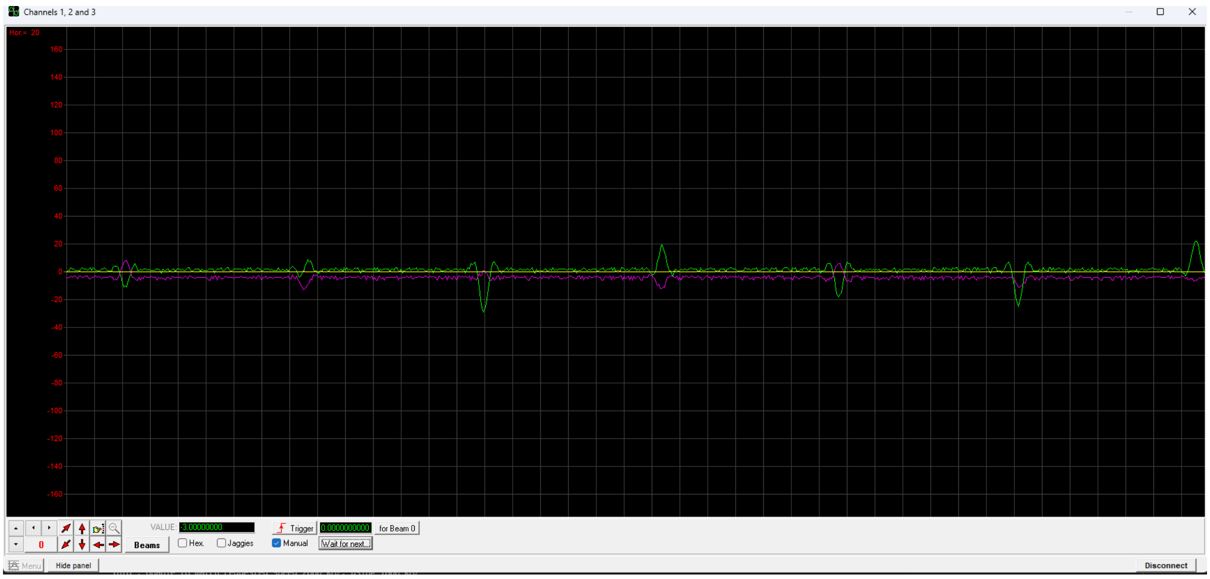
After the filtering and decimation stage, the output signal is scale between -127 to 128.

Testing the output signal

To understand if the output signals were correct we used "**SerialOscilloscope**" to plot the data. One other tool used is "**Frequency Sound Generator**", an android app able to generate soundwaves at different frequencies. Using the app we feed a signal at 3000 Hz (near the snap frequency) and a signal at a frequency bigger than the stop frequency. These are the results:



3000 Hz soundwave

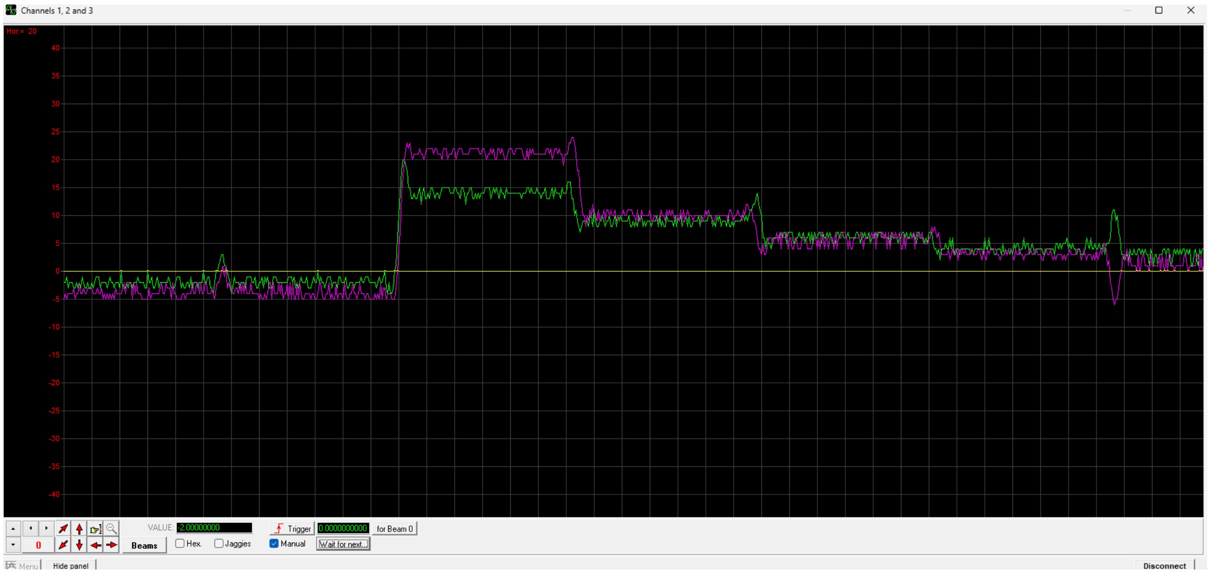


16000 Hz soundwave

The results were correct. In the first case the signal was correcly represented and the amplitude is between -128 and 127. In the second case signal was attenuate since the frequency is higher than the stop frequency of the filter.

Recognition of the finger snap

In order to recognize a finger snap we observe the output signal in case of finger snaps. This is an example of the typical output signal:



finger snap signal

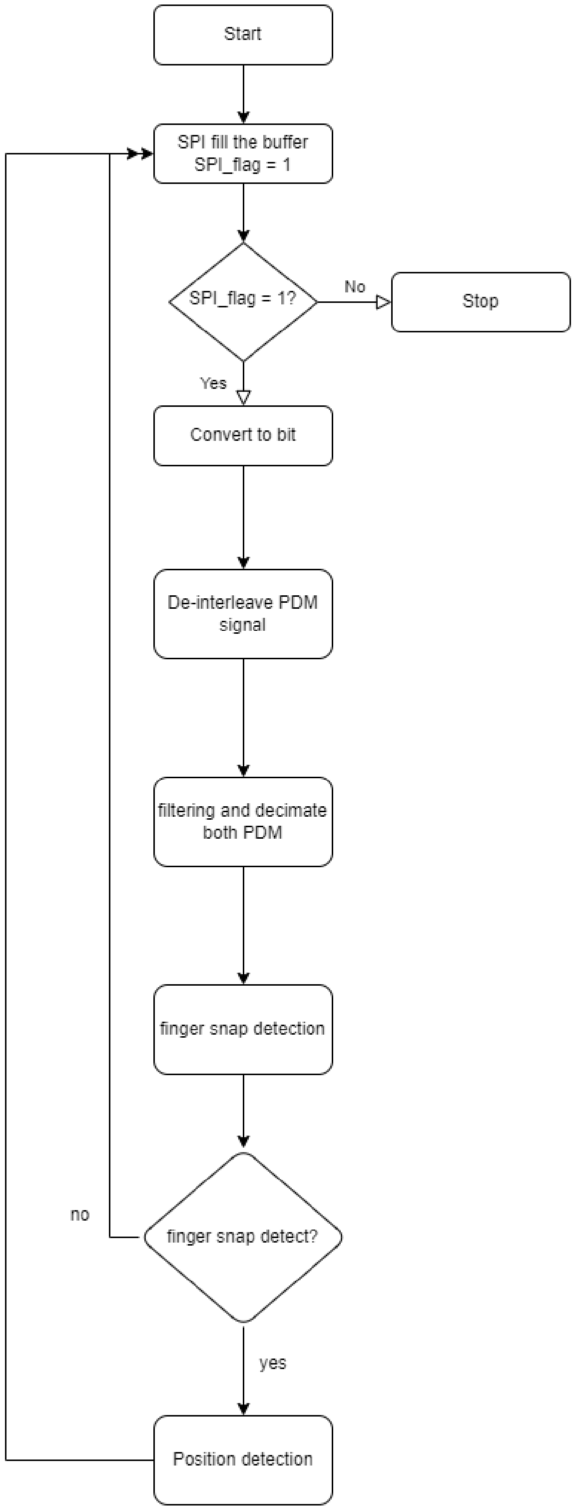
Doing different tests we notice that a finger snap has typically an intensity higher than 10. This value was used as the threshold to catch a finger snap.

In the end we were able to correctly detect a finger snap. To avoid problems of multiple detections we decide to detect only 1 snap, after a correct recognition of the snap the button is press to restart the detection.

Position recognition

We have attempted to develop a program that utilizes the cross-correlation technique to determine the direction of a sound source (even others more simpler techniques were used). However, despite our efforts, the implementation has not been successful for most of the time. We incorporate those techniques in our code, even if we haven't been able to achieve the recognition of the direction of the sound.

Flow Chart



Flow chart