Department of

CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master’s degree in Mechatronic Engineering 2022/2023

Electronic systems for sensor acquisition

Prof. Vacca

PROGRAMMING A MICROCONTROLLER FOR AUDIO DIRECTION RECOGNITION

Group 11:

Ardagna Federico

Bellagamba Alberto

Portella Guilherme

1. Objective:

The goal of this project is to design a system, composed of two microphones, which is able to recognize whether the sound generated by a finger snap comes from the left or the right of the board.

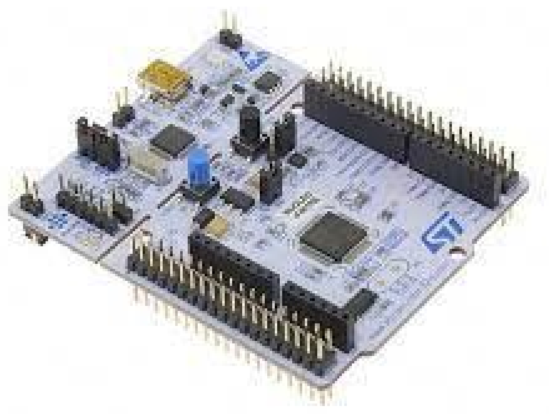
2. Material and Instruments:

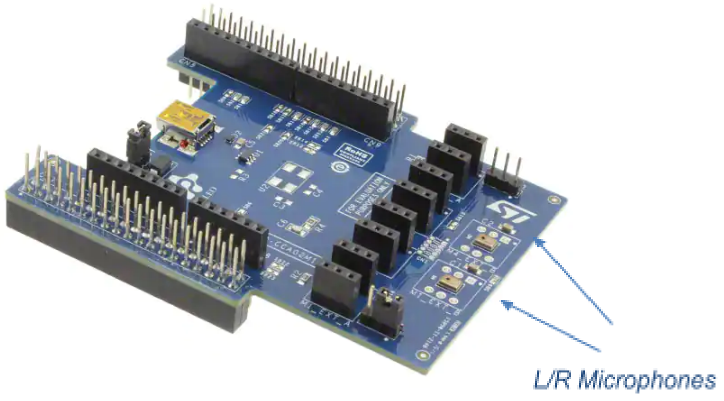
2.1 Hardware:

- STM Nucleo 32F401RE Board (Figure 1)
- STM X-NUCLEOCCA02M1 Extension Board (Figure 2)
- Mini USB cable
- Jumper wires (male to male)
- Oscilloscope (Rigol DS1054)

2.2 Software:

- STM32 Cube IDE (C code)
- Pycharm (Python code)
- Putty (data visualization)
- Matlab (data visualization)





3. Setup and Procedure:

The extension board has 2 microphones (figure 2) which are used to measure the sound produced by the finger snap. By analyzing the two measurements we are able to determine the direction from which the sound is coming from.

In order to acquire the sound measurement, it was important to correctly set up the connection between the Nucleo and the Extension Board: from a hardware standpoint the connection was simply obtained by plugging the Extension Board on top of the Nucleo while for the electrical connection it was critical to properly set up the SPI protocol (3.1).

Once the data from the microphones was acquired, we had to convert it, in order to, first of all, separate the data coming from the right microphone to the left one; then we had to perform a PDM to PCM conversion in order to obtain a digital value representing the sound value (3.2).

After the PCM conversion, there is still some signal processing to do: low-pass filtering to eliminate noise. In order to do this step, we chose a windowed-sinc filter, to have an acceptable frequency response (3.3).

Once we had a stream of digital values it was time to visualize it, in order to design the filter and to understand the fingersnaps sound characteristic. We did it by using firstly a short Python code, which plotted live data coming from one of the microphones and helped us understand the output in real time. Then we also used a brief Matlab script in order to offline capture the values coming from both microphones and with a better precision. (3.4)

To recognize the finger snap from the background noise we analyzed the output in different environments in order to discern the relevant sound and the useless ambient acoustic disturbances. (3.5)

In order to understand if the finger snap was coming from the left or the right we simply had to see which microphone recorded first a peak value (representing the snap) (3.6).

In the following sections a deeper explanation of the already mentioned passages will be provided.

3.1 Data acquisition:

As previously said, the mechanical connection was simply obtained by plugging the Extension board on top of the Nucleo, while the electrical connection was carried as follows:

looking at the Pin layout of the Nucleo (Figure 18) and the electric schematics of the Extension Board (Figure 2) we noticed how, in the Channel 10 of the microphone Board, pins 27 and 30 are connected to the same signal MIC_CLK_X2 and the corresponding pins on the Nucleo (i.e. the pins that get connected while the two boards are plugged) are the PB13 and PB4 so we configured PB4 to be the output of the timer generated by the Nucleo and PB13 to be the entrance of the SPI clock. Being the pins on the two boards directly connected and being the 2 pins short circuited the clock exit is physically connected to the SPI clock.

Figure 18. NUCLEO-F401RE

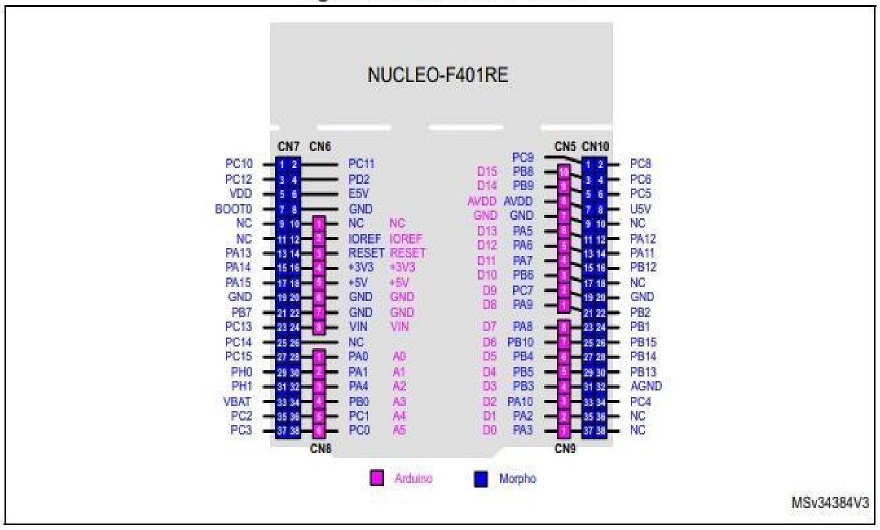
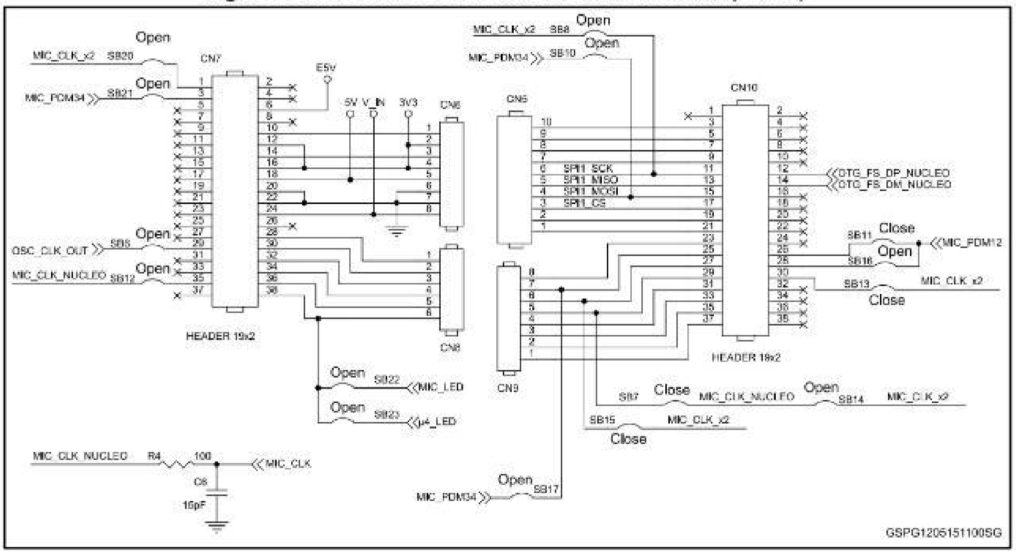


Figure 2: X-NUCLEO-CCA02M1 circuit schematic (Part 2)

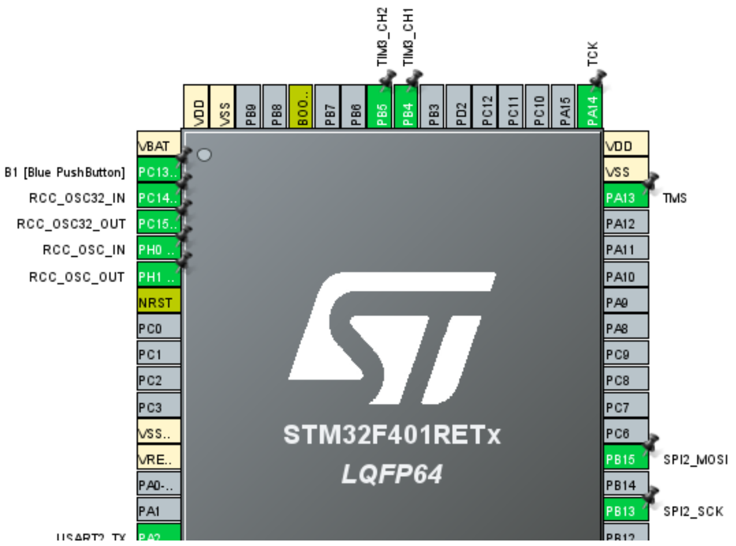


For the Microphone clock we saw from the electrical schematics that MIC_CLK was short circuited with MIC_CLK_NUCLEO and was connected to the pin 29 that in the Nucleo corresponds to PB5 so we set it up to be the clock sent to the microphone.

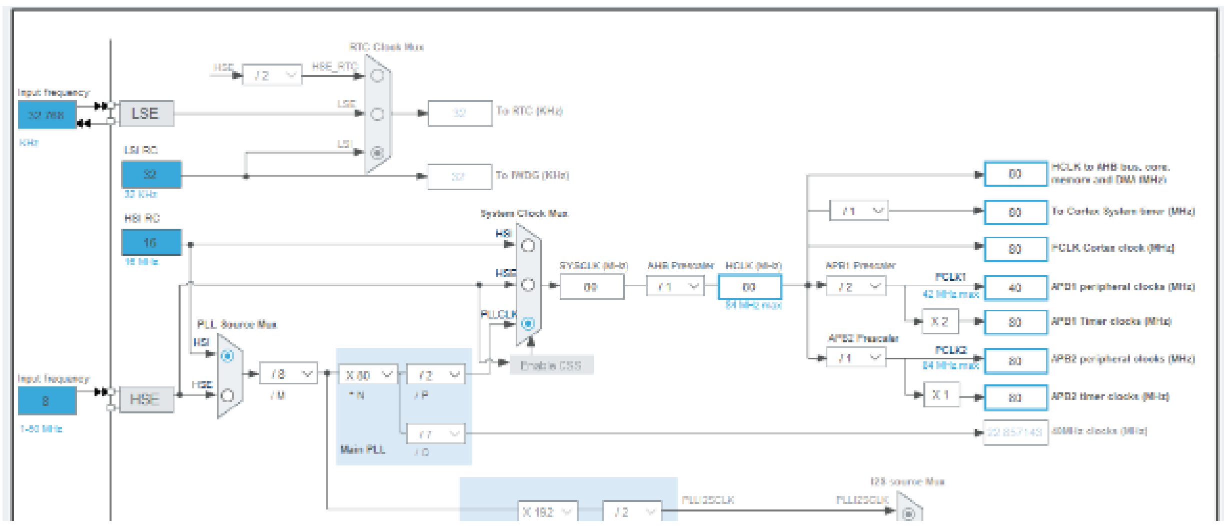
To take the output of the microphone we chose PB15 since it is connected to pin 26 that on the Extension Board corresponds to MIC_PDM12.

From the port PB4 we needed to generate a clock with a frequency that had to be double of the frequency of the microphone. This is because for every tick of the microphone clock, 2 bits are produced to be sent to the SPI, one for each microphone. As we read from the Extension Board datasheet, in order to work properly, the microphone must be fed with an input clock frequency of at least 1 MHz. This is why we chose a frequency of 2MHz for the signal SPI2_SCK; in order to produce this clock we used a PWM output timer, and set the proper values of Pulse and Auto Reload Value registers (see Figure).

As previously stated we used the PB5 port to generate the clock for the microphone, but if we generated two different clocks from two different timers they wouldn’t be synchronized. So we used the same timer that we used for the SPI clock (tim3), exploiting the channel 2, putting it in Output Compare Mode and setting it to Toggle on Match Output mode. Setting a proper compare value on the C code script we were able to obtain a clock of 1MHz synchronized with the SPI one. Also, in order to have calculus simplicity we set the clock configuration to have a 80MHz signal on the APB1 bus (Figure).



We report here some images,representing the complete pin layout and the internal clock configuration. All the other screenshots for the correct configuration can be found in the "img" folder.



Once the clocks were properly generated after some code on the STM IDE we were able to visualize the stream of data in the Putty software.

For the data acquisition we selected a BUFFER_SIZE equal to 32, which means that for each iteration we receive 32 bytes of data corresponding to 256 bits, 128 for each microphone; first of all we set up a flag to check if the data was correctly received. This flag was to be set in the file stm32f4xx_it.c, more precisely inside the DMA callback. In the case the data was correctly moved into the buffer by the SPI protocol, we convert it into a binary stream of data made by ‘0’ and ‘1’ according to the PDM coding. Now that we received the PDM values we have to separate the outputs of the two microphones, and we obtained this result by dividing the values in odd position to the ones in even position and putting them in two different vectors.

3.2 PDM to PCM Conversion (decimation and filtering)

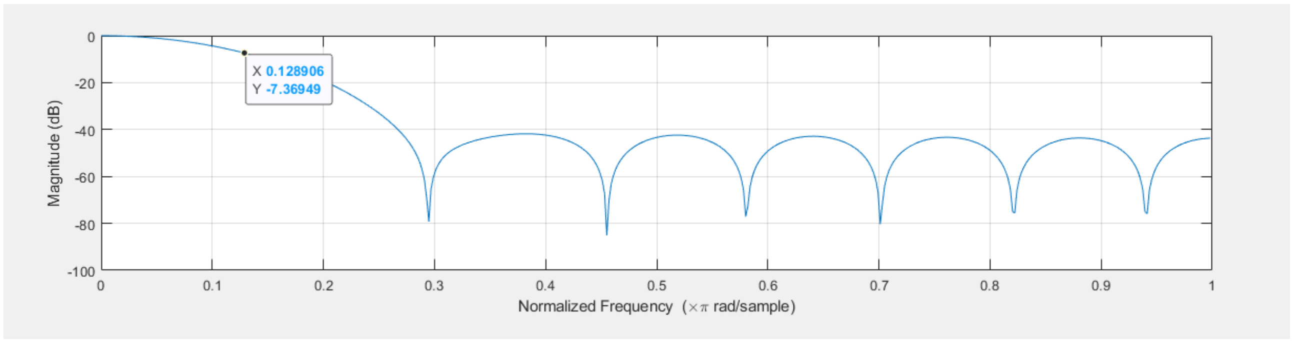
At this point we have the streams of data of the right and left microphone which were represented in PDM coding. This, as previously stated, is binary and hard to decipher; this is why we had to convert them in PCM, that is a digital value representing the sampled signal, more understandable and interpretable.

To obtain this result we simply counted the bits of each byte of the sampled data; since we received 16 bytes for each microphone we ended up with 16 PCM values for the left and 16 for the right. We initialized the destination of PCM values to 0 so that each value would range from 0 to 255; we did it this way because we had some errors with the UART transmission of negative values for debugging reasons.

3.3 Filtering

For the filtering part we used a windowed sinc-low pass filter with a cutoff frequency of 4kHz and 16 coefficients. This choice is due to the fact that the finger snap has usually a bandwidth that goes from 2500 and 3500 Hz, so we considered an additional margin of 0.5kHz. The number of coefficients is justified by the fact that in this way we could produce a filtered output value for each iteration, since we would have 16 inputs (PCM values) and 16 coefficients. Moreover, this choice should ensure us a transition bandwidth of about 0.25*fc (cfr slide 5, L29). The values of frequency are all relative to the sampling frequency in the code. So, if we consider a sampling frequency of f_PCM = 1MHz / 32(bytes) = 31.25kHz, we will have a relative value of frequency of f_c = 4kHz / 31.25kHz = 0.128.

Below an image representing the filter frequency response:

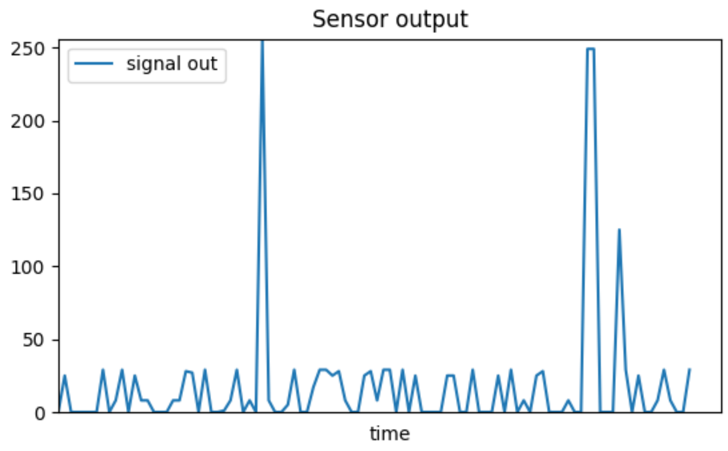


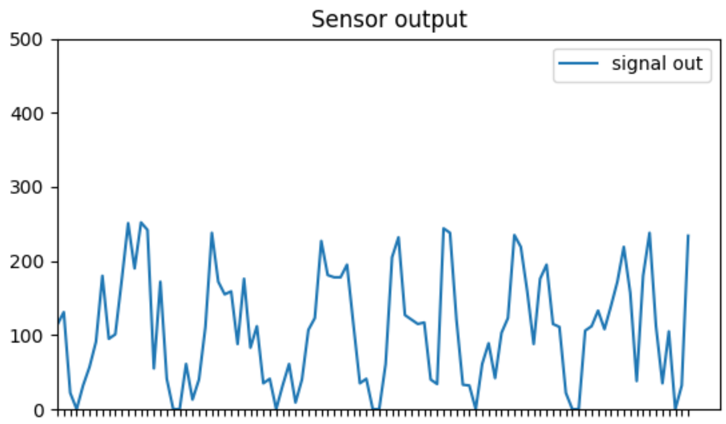
As we expected, the frequency response is quite good, and this is the reason why we used this kind of filter instead of a moving average one.

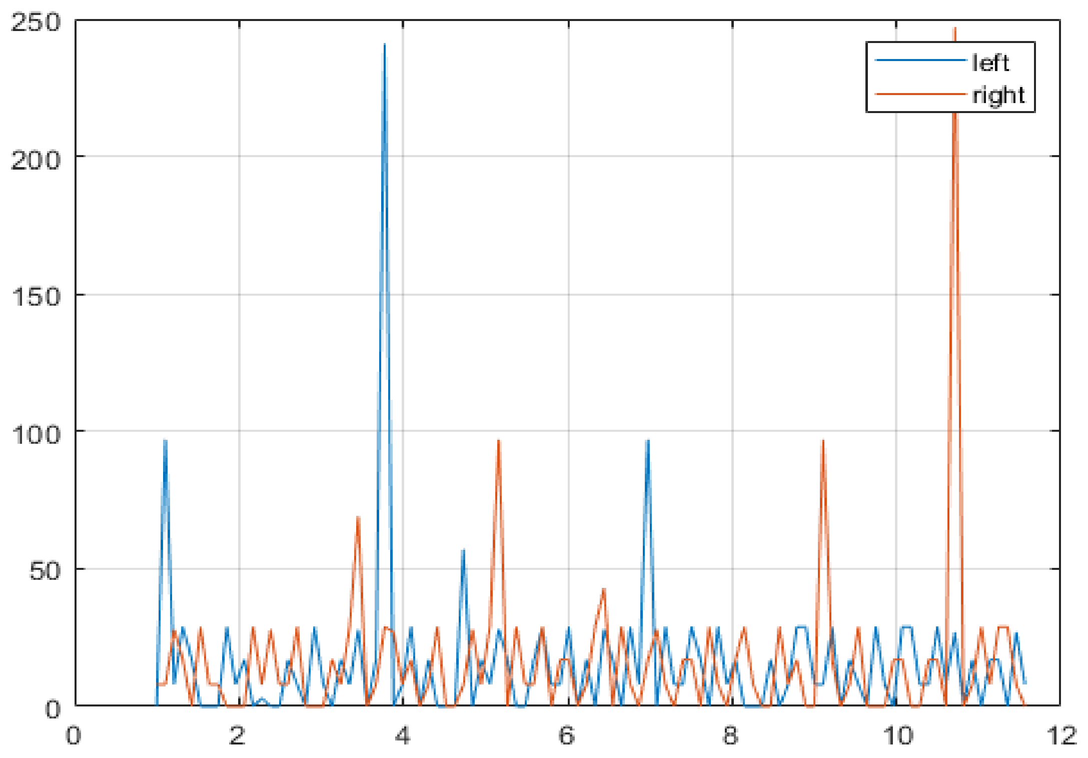
3.4 Data visualization

In order to visualize the data in real time we used the Python code live_plot.py and after setting up the serial communication and some python modules, we were able to plot the data of one microphone in real time. In the following pictures, the difference between the unfiltered output and the output after the filtering stage can be seen.

Moreover, in order to compare the signals from the right and left microphone we used a simple Matlab script (plot_output.m), that collects a bunch of samples from both mics and plots them.







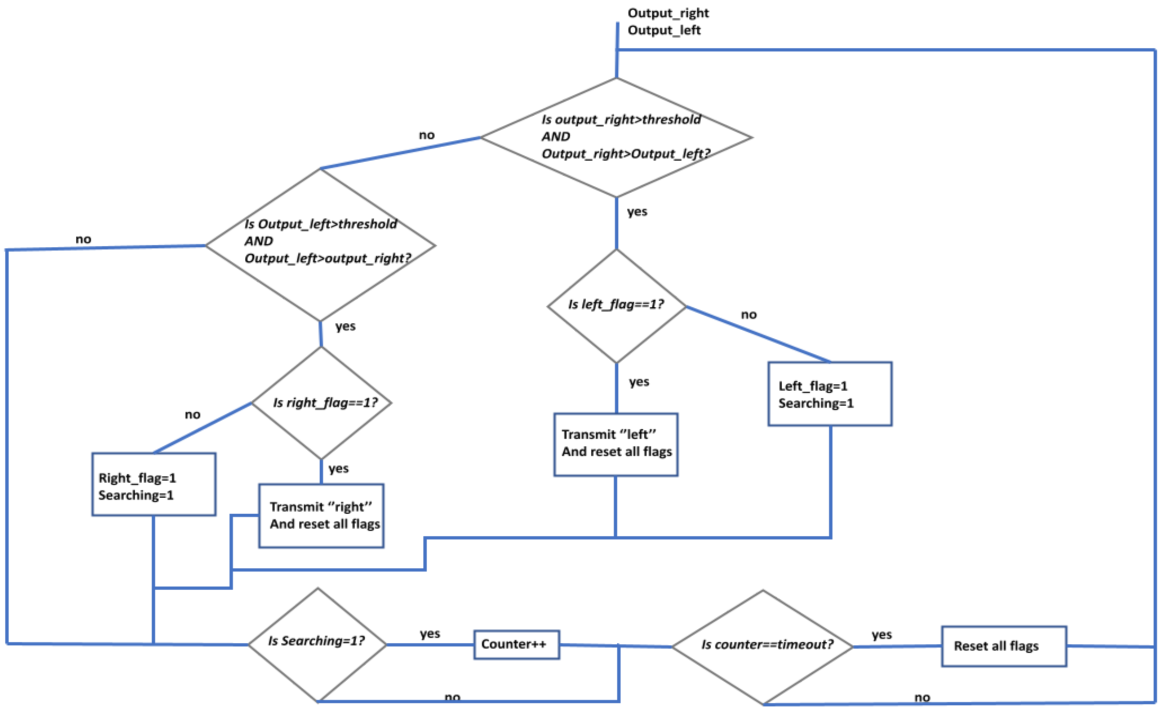
3.5 Finger snap recognition

From the previous plot it’s easy to distinguish from natural environment noise (lower peaks) and our fingersnaps (higher peaks). This was useful to set a threshold for the detection part of the code. In order to understand that there was a finger snap, we took in consideration only the peak values that reached a determined intensity (see the code for the exact value). From this point, the last part is to understand where the detected finger snap was coming from.

3.6 Position recognition

In order to output the direction of the sound, we choose to analyze the delay between the capture of the sound from the right microphone and from the left one. This can be still seen from the Matlab plot shown in the previous section, where we experience a delay of several samples from the detection by one microphone and the detection by the other one.

To do this we implemented the simple flow chart that can be found in the following figure. It compares the two outputs with a threshold and sets a flag if the threshold has been overcome by one microphone. If, within a certain timeout (expressed in number of iterations), the flag of the other microphone is not set, then maybe we are experimenting a noise, or in any case, a sound that does not have the characteristics of the fingersnap. If this is the situation, then our program won't print anything. Otherwise, if the first flag is activated, and then, after a certain time (that is less than the timeout), the flag of the other microphone is activated too, then the program will print a "Left" if the first flag was flag_left or "Right" if it was flag_right.



4. <u>Conclusions:</u>

As shown in the video attached to this report, our program can be considered working in a silent environment. We tested it also in more noisy rooms, and for some higher values of threshold, the program could still ignore the non-fingersnaps. Moreover, the timeout parameter for the direction recognition can be changed, but in that case a bigger value will lead to a slower program, because it will be waiting for a greater time for the other flag to be set. Unfortunately, we didn't have enough time to modify the code in order to detect the precise direction of the sound, but we guess the right way to do it could be by analyzing the difference in module and the delay between the two waves (right and left).

About the plots visualization, in the C code there are, in the form of comments, instructions to correctly visualize the signals using the scripts data_live.py and plot_output.m. This is due to the fact that the scripts are designed to work only if the inputs are of a determined type (only one microphone or both of them).

Together with this report, you should be able to find:

- screenshots for STM32IDE configuration
- data_live.py and plot_output.m
- main.c, main.h, stm32f4xx_it.c
- video test