VRIJE
UNIVERSITEIT
BRUSSEL

EMBEDDED DSP SYSTEMS

Bike Acoustic Danger Proximity Sensing Box

Ben Dupont, Daoud Uahabi, Ruben Parent, Sam Dilmaghanian, Seppe Goossens

June 21, 2023

Academic year 2022-2023

Industriële wetenschappen

Contents

1 Overview

Bike Acoustic Danger Proximity Sensing Box is a project focused on leveraging technological advancements to enhance the safety of cyclists during their rides.

The project introduces two battery-powered & wirelessly interconnected devices that can be easily mounted to a bicycle.

Device 1, known as the Sense-U, is positioned beneath the bike saddle. Equipped with a set of microphones and a powerful FPGA, the Sense-U utilizes beamforming algorithms to accurately determine the location of approaching cars. By analyzing sound waves and their direction, the sense-U can identify the source of the sound and provide the cyclist with information about nearby traffic. 2 PCB's and an FPGA development board make up the Sense-U. The first PCB is a microphone array. The 2nd is the Sense-U PCB itself. It functions as a motherboard to provide power & data-connectivity to the microphones and the FPGA development board. Device 2, called the Display-U, is mounted on the bike's handlebars. Its purpose is to visually display the location of cars detected by the Sense-U. This is achieved using 2 LEDs that illuminates either on the left or right side of the Display-U, indicating the relative direction of the approaching vehicle.

Embedded Bike Safety combines the sensing capabilities of the Sense-U and the informative display of the Display-U to give cyclists real-time information about their surroundings.
By providing them with awareness of nearby vehicles, cyclists can make more informed decisions while on the road, ultimately enhancing their safety.

Figure 1: Bike Acoustic Danger Proximity Sensing Box Logo (DALL-E)

2 Audio Source

Car noises contain a wide range of frequencies that overlap with other environmental sounds, such as wind and rain sounds, making it difficult to isolate and analyze in both the time and

frequency domain. The type of noise can also vary significantly based on factors like engine type, vehicle speed, road conditions, and surrounding traffic, further complicating accurate frequency analysis. This is why it is important to have an understanding about the audio that we are looking for. To get a feeling for the type of noises that we would encounter, we made some recordings [1] and looked at the different reoccurring frequencies.

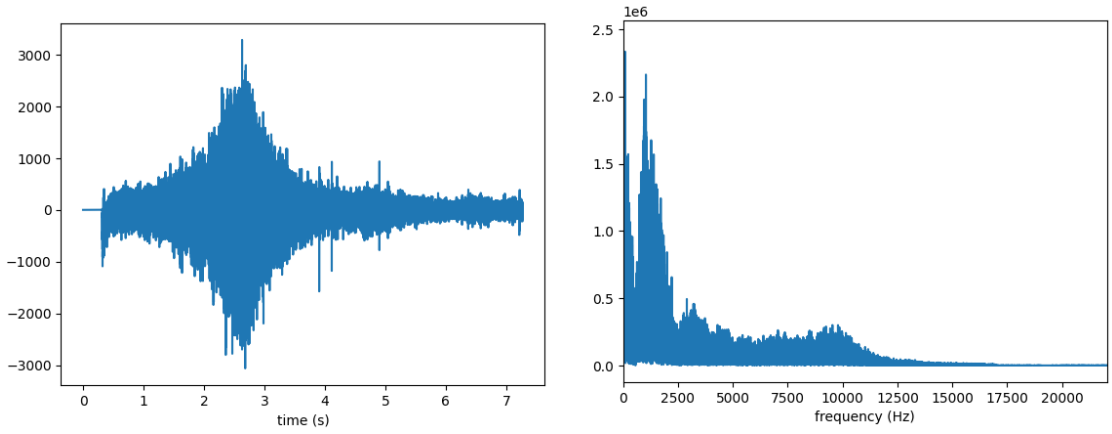


Figure 2: Recorded noise of an electric car at 50 km/h

Figure 1 shows an audio recording of an electric car driving at 50 km per hour. The sound that cars typically make is somewhere in between 50 Hz and 10 kHz, The peaks that are present in the audio spectrum of one recording are often not located at the same frequency in another audio recording. What can be used however is the energy of the signal in the already contained in the frequency spectrum. When using multiple microphones we do not necessarily need to search for the loudest noise but we can also detect based on whether the sound is coming from behind and whether the angle of the sound is increasing or not. The lack of signature means there is no need for a Fourier transform, but this also means that the noise of interest will easily be drowned out when larger sources of noise are present.

3 Localization Algorithm

The two most suitable algorithms for this project are Cross Correlation (CC) and Delay and Sum (DaS). In the following sections we will explain how these algorithms work and why we chose the latter as the localization algorithm.

3.1 Cross Correlation

Cross Correlation can be used to measure how closely related 2 signals are to each other as a function of displacement of one relative to the other. For this project, CC can be used to calculate the time difference between 2 (or more) signals. When the time difference of the 2 signals is obtained, we can use the geometric properties of the microphones to calculate the Angle of Arrival (AoA) of the sound source. To demonstrate this method we created a Matlab script that

[1] Example of a recording that we made, plotted in a Spectogram: YouTube

simulates 2 sound sources and the CC algorithm.

As testing data we used a recording that was made in the sound chamber. This sample has a sampling frequency of 32 kHz. The total length of the samples is 10s meaning we have a total of 320 000 samples. We can see that the we have a peek at $\approx 6s$, this is the sound source object we are looking for.

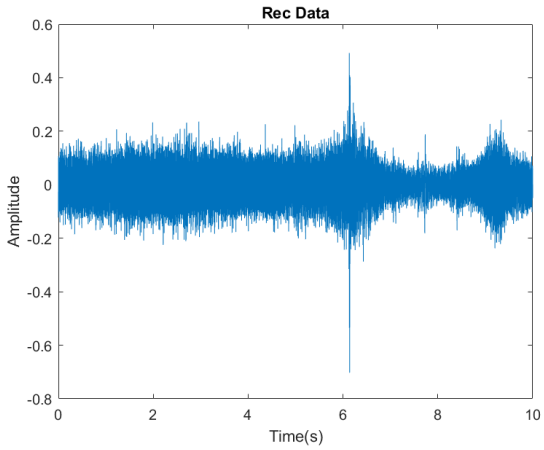


Figure 3: Cross Correlation Test Sample

Suppose we have 2 microphones. If the sound waves are not coming perpendicular to the 2 microphones, one microphone will receive the sound wave before the other. We can simulate this by duplicating the sound wave and shifting it in time:

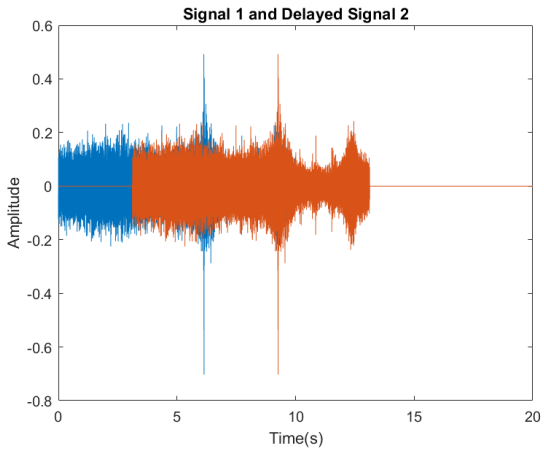


Figure 4: Cross Correlation Shifted Samples

Next we can perform the CC operation via the `xcorr` function in Matlab. We pass the 2 signals as arguments.

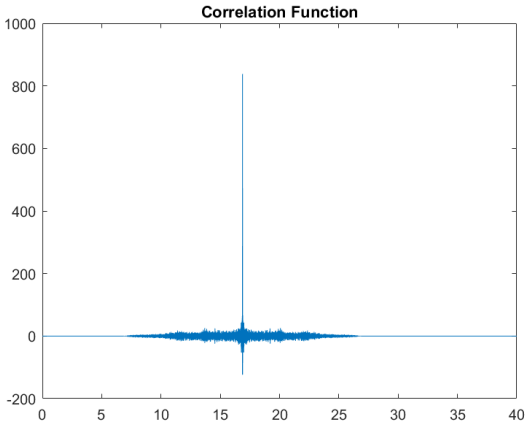


Figure 5: Cross Correlation Output

We can the corresponding delay by finding the x value of the peak value. This value in combination with the relative positions of the microphones is then used to calculate the source angle.

The downside of using CC is that the signals need to be very similar in shape to get an accurate result. For this project, considering we use multiple microphones that are located at the opposite sides of the bike and the amount of environmental noise there is on the street from all different angles makes using CC to get an accurate localization result rather difficult. Because of this we opted to use Beamforming, in particular the Delay and Sum algorithm.

3.2 Beamforming: Delay and Sum

The Delay and Sum algorithm allows to scan in a certain direction by creating a focused beam pattern. The beamformer does this by applying a delay to each microphone output and summing them. The delays are chosen to maximize the output result in a certain direction. By changing the delay values, the beamformer can "look" in multiple directions and calculate the sound strength per direction by summing the microphone signals. This implies that some signals are subject to constructive interference while others are subject to destructive interference. For this project we are using the Delay and Sum algorithm in the Time Domain.

3.2.1 Maximum Delay

A very important factor in the performance of the Delay and Sum algorithm is the geometry of the used microphones. To demonstrate this point let's take the following microphone setup:

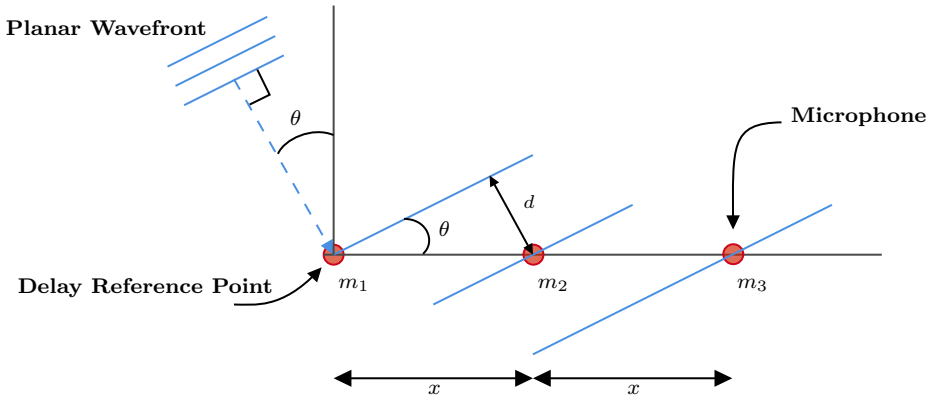


Figure 6: Microphone Setup Example

Suppose that we have a planar wavefront. By using some simple geometry we can calculate the distance between 2 wave fronts.

$$d = x\sin(\theta)$$

The maximum delay depends on the largest overall distance between the microphones. In this case microphone m_1 will receive the wavefront first so this is our delay reference point. All the other microphones that have a smaller distance will have a delay lying somewhere in between 0 (= delay of m_1) and the maximum delay (= delay of m_3). The largest distance is between microphone m_1 and m_3. The maximum delay in time can then be calculated:

$$delay_{max,time} = \frac{d_{m_1 m_3}}{v} = \frac{2x\sin(\theta)}{v_{sound}}$$

The ability of the embedded systems to distinguish two samples in time is directly correlated to the maximum delay and this delay is constrained by the largest distance between the microphones. Because of this it is very important that we chose an appropriate microphone geometry. See section 5 for the chosen microphone geometry.

Because we are working with discrete samples coming from the microphones we rather want to know what is the delay in **samples** and not in time. This can be calculated as follows:

$$delay_{max,samples} = \lfloor \frac{2x\sin(\theta) f_s}{v_{sound}} \rceil$$

3.2.2 Delay Table and Delay Buffers

Another important aspect is the delay table. This table contains the delays per microphone, per angle. This table can be calculated in advance and can easily be stored in a LUT on the embedded system. We first tried to make this table ourselves, which did work when all the microphones were located on one line, so 1 dimensional. But when we tried 2D microphone setup the delay table became rather complicated. We had some working code but it still contained some bugs. Because of this we seeked help from CABE. With CABE we just had to give it the coordinates of our microphone setup and choose the right DSP settings and it calculated all the rest. This also allowed us to make an informed decision about the right microphone setup. See section 5.

We created a simulation in Matlab to implement the DaS algorithm. To test the algorithm we used a 10kHz Sine that was sampled at 48kHz as sound source. See section 5 for the full microphone setup but in short, it contains 9 microphones in a half circle with a radius of 60mm. The used delay table was created by CABE and looks as follows:

17;16;14;12;8;5;2;1;1
17;16;14;12;9;5;3;1;1
17;16;15;12;9;5;3;1;1
17;16;15;12;9;6;3;1;1
17;16;15;12;9;6;3;1;1
17;16;15;12;9;6;3;1;1
17;16;15;12;9;6;3;1;1
17;16;15;12;9;6;3;1;1
17;16;15;13;9;6;3;1;1
17;16;15;13;10;6;3;1;1
17;16;15;13;10;6;3;1;1
16;16;15;13;10;7;4;1;1
16;16;15;13;10;7;4;1;1
16;16;15;13;10;7;4;1;1
16;16;15;13;10;7;4;1;1
16;16;15;13;10;7;4;1;1
16;16;15;13;10;7;4;1;1
16;16;15;13;10;7;4;2;1
16;16;15;13;11;7;4;2;1
16;16;15;13;11;7;4;2;1
16;16;15;14;11;8;4;2;1
16;16;15;14;11;8;4;2;1
16;16;16;14;11;8;5;2;1
15;16;16;14;11;8;5;2;1
15;16;15;14;11;8;5;2;1
...

Each row represent an angle. This table has 361 rows meaning we have a angle resolution of 1°. Each column represent a microphone. We can now easily find the delay for a given microphone depending on the angle.

The delay buffers contain the samples from the microphones. The length of the buffer is based on the maximum delay in the delay table. The maximum delay tells us what the furthest sample is we need to take, so there is no need to make the buffer size bigger than this value. To calculate the output, we iterate over all the angles and per angle we go over all the samples received from the microphone. The delay buffer is constantly being filled with new samples. When the delay buffer is full, we can calculate the output by consulting the delay table to find the corresponding delay index for that particular angle and microphone. This delay is then used as index in the delay buffer. The corresponding sample from the delay buffer is used to calculate the output by summing it up with the other samples from the other microphones. We keep doing this until we have iterated over all the samples and all the angles.

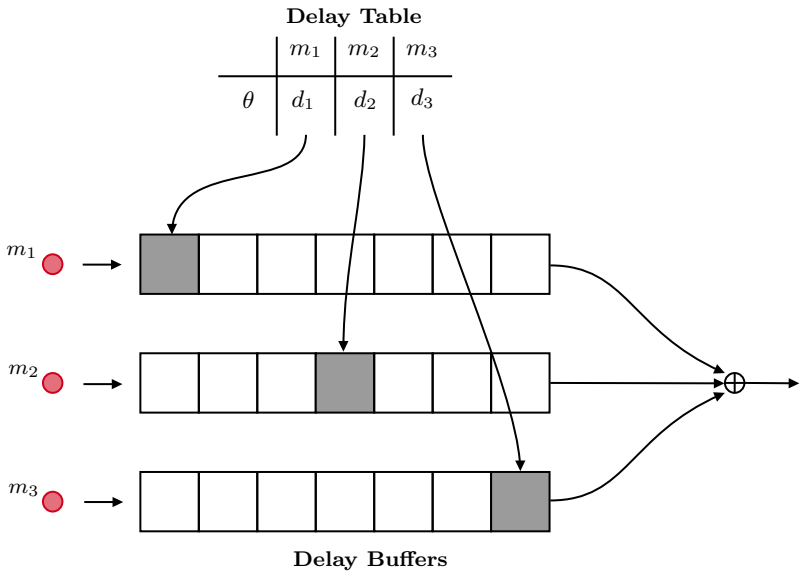


Figure 7: Delay and Sum algorithm using a delay table and delay buffers

To test the algorithm we first need to simulate the microphone signals. This is done by choosing a source angle from where the sound source is coming from. Let's take 60°for example. This corresponds to a certain delay in samples. We know that, depending on the microphone orientation, a source angle of 0°would correspond to a delay of 1 for example and a source angle of 180°would correspond to the maximum delay value. We can then map the source angle to this range. In our testing case, the max delay is 17 samples.

$$[0, 180] \longleftrightarrow [1, 17]$$

A source angle of 60°corresponds to a delay of 6 samples. The microphone signals can then be simulated by creating arrays, equal to the length of the original microphone samples + max delay in samples. These arrays are then filled with the samples but with the 6 sample offset. This creates the simulation of one microphone receiving the sample first, then the second and so on. When the output samples are calculated, we calculate the RMS values and normalize them. We can plot the results on a polar plot:

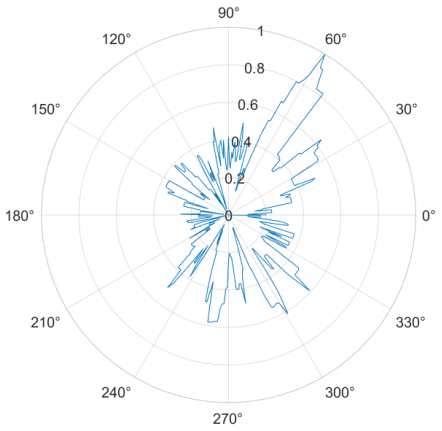


Figure 8: Delay and Sum Matlab Simulation Polarplot

We can indeed see that the highest value corresponds to an angle of 60°. This simulation in Matlab gives a good feeling for implementing the Delay and Sum algorithm on an embedded system. An advantage of the Delay and Sum algorithm is that unlike with CC the signals can be less similar in shape. This means that the microphone signals can endure more noise and this is especially important considering the application. Another advantage is that the DaS algorithm itself doesn't require any multiplications, only additions. The only resources needed are also LUTs for the delay tables and buffers for the delay buffers. The samples coming from the microphone can be implemented as a stream meaning we don't need a lot of memory.

3.2.3 Delay and Sum Implementation in VHDL

As discussed in section 6 we chose to use an FPGA as embedded platform. This means that the DaS algorithm has to be written in VHDL. The main VHDL file contains 2 processes [2]. One to read the delay table from a file and the other is the main DaS algorithm. The main algorithm follows the same principle as the Matlab version. The test bench contains a read and write process to read the samples from a file and write the output to a file. The input samples are stored in one large buffer, the read buffer. This is done so that we can simulate a stream of samples. The output samples are also written to one large array so that they can be written to a file in the corresponding process. The test bench process for the DaS algorithm does two main things. It first reads the large input array and sends samples to a second buffer, the sample buffer. This sample buffer has a arbitrary length of 6000 samples (can be changed after testing) and once filled, the DaS algorithm will start. Once the DaS algorithm is done, it will notify the test bench and the output samples will be written to the write buffer. While writing to the write buffer, a new set of samples is loaded into the sample buffer .This allows us to simulate a stream of samples. Doing this is important for our application because we want to process everything as fast as possible, in real-time.

[2]The implementation here is to test out the DaS algorithm. This is why we read and write to files. In the final VHDL design, the read and writes to the files would not be implemented.

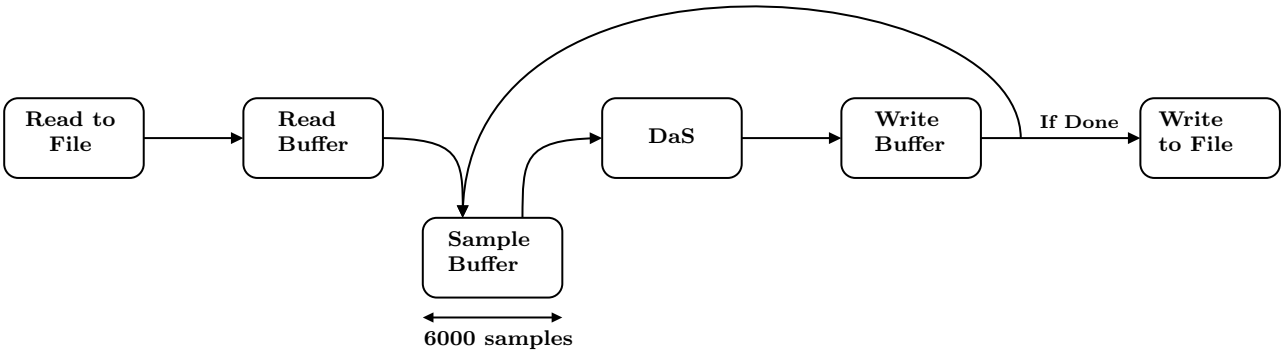


Figure 9: DaS VHDL Implementation Block Diagram

To check if the VHDL implementation returns the correct result, we can compare the test bench output with the output from the Matlab script. This is done by using a difference checker.

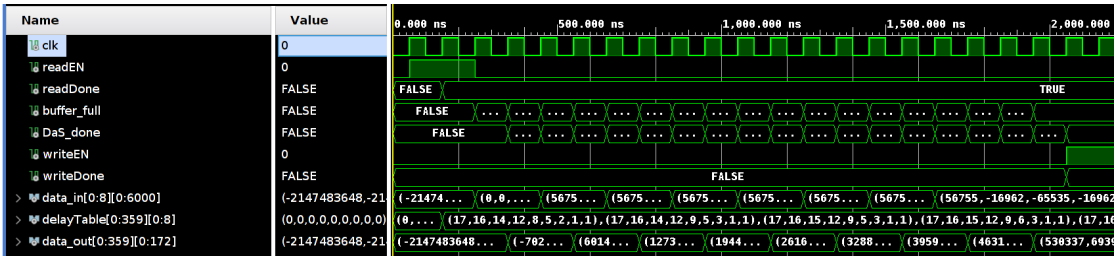


Figure 10: DaS VHDL Implementation Test Bench

The design however takes up, after adding the needed filtering, more than 200% of the available lookup tables on the FPGA board and so changes to the hardware design are needed to reduce the LUT consumption. This means in the first place to get rid of code that does not impact performance at all (lossless) but it involves also the removal of unnecessary code that only increases the quality of the design by a small margin (perceptually lossless).

152 of the 360 filter coefficients that are used for the DaS algorithm are an exact copy and the logic associated with these redundant angles could be completely removed, there would however need to be additional hardware to link the indexes to the right angle as the duplicates are not equally distributed along the coefficient table. A better option is to reduce the number of angles, it can be even more efficient by choosing a number of angles that is a power of two so computational hardware can be equally distributed. It turns out that 64 is the largest power of two for which there are no duplicate angle values. This gives an angle accuracy of 5.625˚instead of the initial 1˚. This also means that the angle output of the program needs to be interpreted this way, better for a computer but humans must take care.

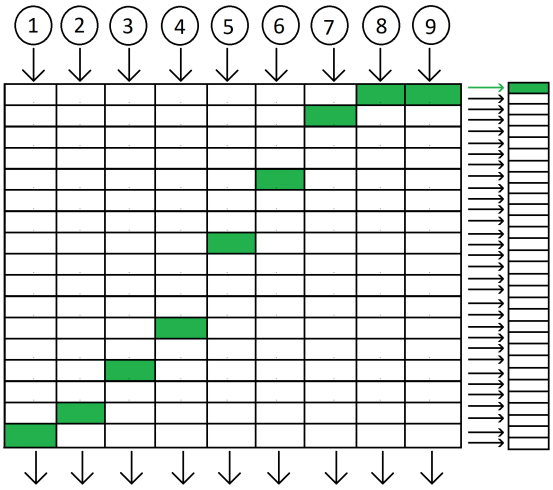


Figure 11: Delay and sum in Hardware

3.2.4 Required Calculations per Sample

We know that the delay and sum algorithm doesn't need any multiplications, only additions. The following calculation is based on 1 sample. Because the DaS would be implemented as a stream, we want to know how many additions we need per sample. When choosing a buffer size, we can simply multiply the amount of additions with the buffer size.

$$\begin{aligned} Additions &= Microphone\ Amount \times Angle\ Resolution \\ &= 9 \times 360 \\ &= 3240 \end{aligned}$$

So we need 3240 additions per sample.

4 DSP Setup

4.1 Data Representation

An important factor to take into account when implementing our application on an embedded platform is the choice of the data representation. We know that the PDM data coming from the microphone is 1 bit: either 0 or 1. Depending on the filter setup, this bit width of our data will change. We have to make a trade-off between precision and resource consumption. We still want a decent amount of precision but considering that these samples have to be run through the DaS algorithm we also want this to be as fast as possible. Because of this we chose to use integer notation. This will be the easiest to implement in VHDL and if we use a decent sized bit width we can still achieve optimal results. We chose for the following setup, each value represents the output bit width:

- PDM data - 1 bit
- CIC filter - 16 bit integer
- HBF filter - 16 bit integer

- FIR filter - 16 bit integer
- DaS algorithm - 16 bit integer
- RMS calculation - 16 bit integer
- Normalized output - 16 bit integer

The final output is a 16 bit integer. Considering the application, we think that this is a good trade-off between precision and resource consumption.

4.2 Filter Setup: PDM to PCM

In the labs we learned what the different type of filters are and how they work. In this section we will explain why we chose the filter setup that is used to convert the PDM data to PCM data.

4.2.1 General Design Constraints

Input and Output Sampling Frequency: From section 5.1 we know that the used microphones are configured with a clock frequency of 2.4 MHz. As output sampling frequency we chose 48 kHz. This is a common value that is used in audio formats. This however means that when using a input sampling frequency of 2.4 MHz that we have a decimation factor of x50. This value is not practical because it's only divisible by 2 once meaning that if we want to cascade decimation filters, we would not achieve the 48 kHz output sampling frequency. Because of this we configure the microphones with a clock frequency of 2.304 MHz. This means that we have a decimation factor of x48. This value is always divisible by 2 meaning that it will be easier to cascade decimation filters to reach the desired output sampling frequency.

Pass and Stopband Frequencies: From the labs we know that PDM microphones typically have a useful frequency response from somewhere in the range of 100 Hz to max 10 kHz. We can also see this in the datasheet of the chosen microphone:

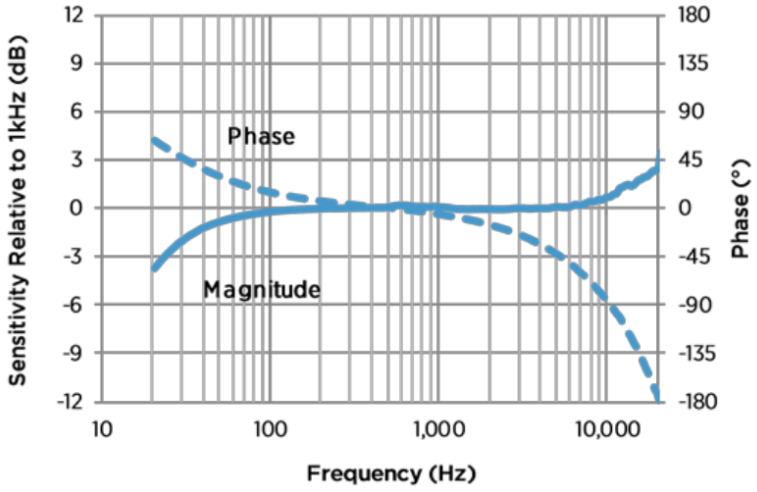


Figure 12: SPH0655LM4H-1 Typical Free Field Magnitude and Phase Response

We can see that between 100 Hz and 6 kHz the magnitude response is relatively flat. As discussed in section 2 we know that the audio sources we are interested in have a range of 50 Hz up to 10 kHz. This means that everything above 10 kHz can be discarded as noise. We see from figure 12 that at 10 kHz we have a sensitivity of 3dB which is reasonable considering our application. To filter out the noise above 10 kHz we need to design a filter setup that has a passband up to 10 kHz. For the start of the stopband we can take a look at the Ultrasonic Response of the microphone:

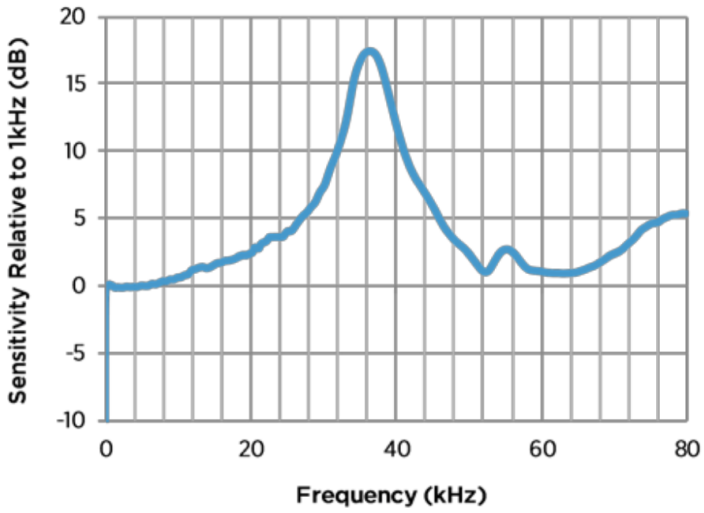


Figure 13: SPH0655LM4H-1 Typical Free Field Ultrasonic Response

We can see that around 15 kHz we have a sensitivity of around 3-4 dB which is also still acceptable. Starting the stopband at 15 kHz would be a good compromise to still achieve a passband that stops at 10 kHz. This means that we have a transition band of 5 kHz. These value also satisfy the Nyquist criterion considering we have an output sampling frequency of 48 kHz. A stopband starting from 15 kHz to 24 kHz is well within the bounds.

Dynamic Range and Maximum Ripple: The dynamic range of our microphone is based on 2 factors: AOP or the maximum sound pressure, the SNR measured with a 1 kHz sine wave at 94 dB SPL or 1 Pascal. We can get these values from the datasheet [3]:

[3]Datasheet of the SPH0655LM4H-1

Signal to Noise Ratio	SNR	94 dB SPL @ 1 kHz, A-weighted, Fclock = 1.2 MHz	-	64.5	-	dB(A)
		94 dB SPL @ 1 kHz, A-weighted, Fclock = 1.536 MHz	-	66	-	
		94 dB SPL @ 1 kHz, A-weighted, Fclock = 2.4 MHz	-	66	-	
		94 dB SPL @ 1 kHz, A-weighted, Fclock = 3.072 MHz	-	66	-	
		94 dB SPL @ 1 kHz, A-weighted, Fclock = 4.8 MHz	-	66	-	

(a) SNR

Acoustic Overload Point	AOP	10% THD @ 1 kHz, S = typ	-	132.5	-	dB SPL

(b) AOP

Figure 14: Signal to Noise Ratio and Acoustic Overload Point of the SPH0655LM4H-1

The dynamic range of the microphone is then calculated as follows:

$$DR = AOP - 94dBSPL(ref) + SNR$$
$$DR = 132.5dB - 94dB + 66dB = 104.5dB$$

Figure 15 illustrates the dynamic range of a microphone [4]. We can see that the dynamic range is based on the AOP value and the noise floor.

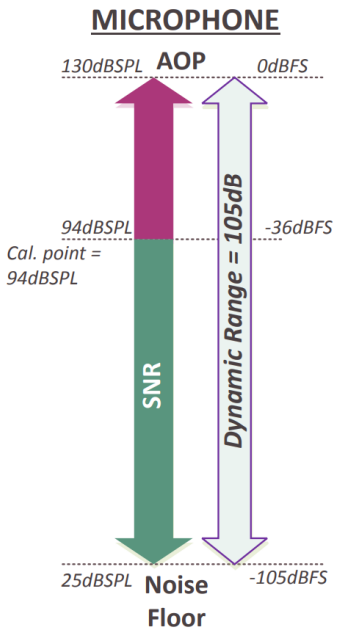


Figure 15: Dynamic Range of an IM69D130 Microphone [5]

We know that our microphone will have an output signal with a SNR of 104.5 dB. Because we will use decimation filters to convert the PDM to PCM, higher frequencies will be aliased into the lower frequency bands. Because of this noise, the SNR of the final signal will be reduced.

[4] infineon application note: Digital encoding requirements for high dynamic range microphones

This is an other important design constraint. How much do we want to deviate from our original SNR value. Considering we want to stay withing the 16 bit range to represent our audio signals, we need to check if the 16 bit data is enough to represent our SNR value. Using this table [6] we can check how much we need to lower the SNR to stay within the 16 bit range:

Signal-to-noise ratio and resolution of bit depths (unweighted)

# bits	SNR (Audio)	SNR (Video)	Minimum dB step difference (quantization rounding error)	No. of possible values (per sample)	Range (per sample) for signed representation
4	25.84 dB	34.31 dB	1.578 dB	16	−8 to +7
8	49.93 dB	58.92 dB	0.1948 dB	256	−128 to +127
11	67.99 dB	77.01 dB	0.0331 dB	2,048	−1,024 to +1,023
12	74.01 dB	83.04 dB	0.01806 dB	4,096	−2,048 to +2,047
16	98.09 dB	107.12 dB	0.00598 dB	65,536	−32,768 to +32,767
18	110.13 dB	-	0.000420 dB	262,144	−131,072 to +131,071
20	122.17 dB	-	0.000116 dB	1,048,576	−524,288 to +524,287
24	146.26 dB	-	0.00000871 dB	16,777,216	−8,388,608 to +8,388,607
32	194.42 dB	-	4.52669337E−8 dB	4,294,967,296	−2,147,483,648 to +2,147,483,647
48	290.75 dB	-	1.03295150E−12 dB	281,474,976,710,656	−140,737,488,355,328 to +140,737,488,355,327
64	387.08 dB	-	2.09836488E−17 dB	18,446,744,073,709,551,616	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Figure 16: Signal-to-noise ratio and resolution of bit depths

We see that to stay within the 16 bit range, the max SNR that is possible for digital audio is 98.09 dB. This value is still sufficient for our application.

We also have to decide how much passband ripple we want to allow on the output signal. For high quality audio, 0.1 dB is a typical requirement.

Summary:

- Decimation factor of x48:
  - PDM input sample frequency of 2.304 MHz
  - PCM output sample frequency of 48 kHz
- Passband from 0 Hz to 10 kHz
- Stopband that starts at 15 kHz.
- Output signal with a SNR of 98 dB
- Maximum passband ripple of 0.1 dB

These design constraints will determine how we design the filters and how we cascade them to achieve an optimal output result.

4.2.2 Multi Stage Filter Design

As discussed in the labs, using 1 large FIR filter to filter out all the noise is a waste of resources. Because of this we will cascade multiple filters to get a good balance between resource consumption and an accurate result. We used the following setup:

[6] Audio bit depth, wikipedia page

CIC $\longrightarrow$ HBF $\longrightarrow$ 2x Decimation $\longrightarrow$ HBF $\longrightarrow$ 2x Decimation $\longrightarrow$ FIR

We used a CIC filter because it doesn't require any multiplications and only 2 additions. The HBF filter before a decimation to reduces the amount of calculations by half compared to a generic FIR filter. As last a generic low pass FIR filter is used to filter out the remaining noise.

CIC Filter: We use a 4 stage CIC filter with a decimation factor of 12. This gives us a good trade-off between the passband and stopband attenuation's we want to achieve while not using to many delay registers. The decimation factor of 12 gives us an intermediate output sampling frequency of 192 kHz.

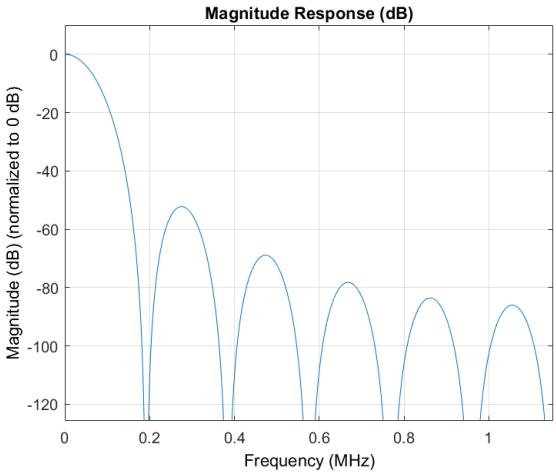


Figure 17: 12x Decimation 4 Stage CIC Filter

2x (HBF Filter + 2x Decimation): The CIC filter is followed up by 2 sets of a halfband filter combined with a 2x decimation. We use the halfband filter to reduce the amount of calculations needed by half (compared to a generic FIR filter). In this case the halfband filter acts as an anti-alias filter. The 2 sets, so 2 x 2x decimation results in an output sampling frequency of 48 kHz at this filter stage. Per set, 7 coefficients are used.

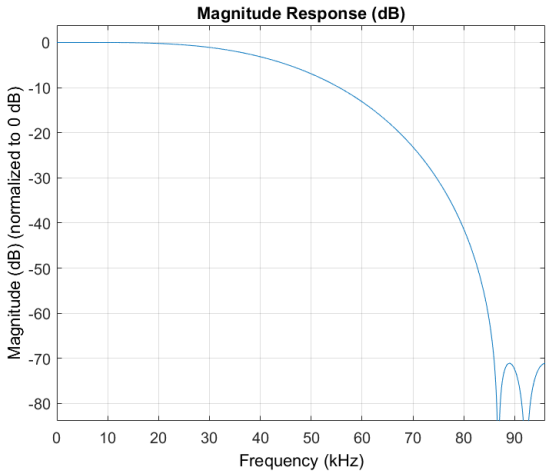


Figure 18: First Half Band Filter with 2x Decimation

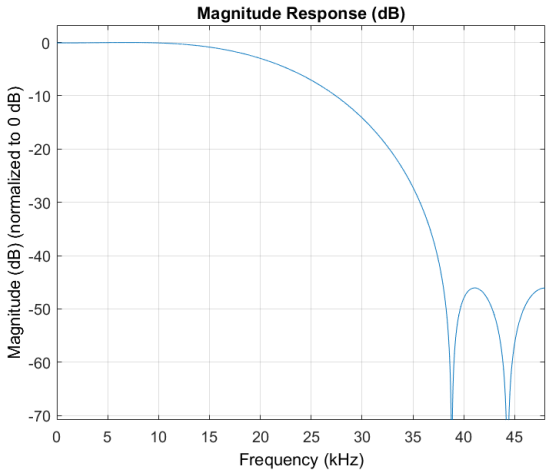


Figure 19: Second Half Band Filter with 2x Decimation

FIR Filter: The final FIR filters out the remaining noise. The passband of this filter goes up to 10 kHz and the stopband starts at 15 kHZ. We perform no decimation so the output sampling frequency is also 48 kHz. A total of 32 coefficients are used for this FIR filter.

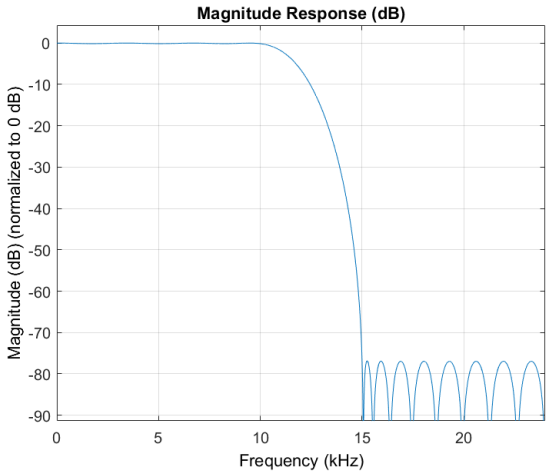


Figure 20: Final FIR filter

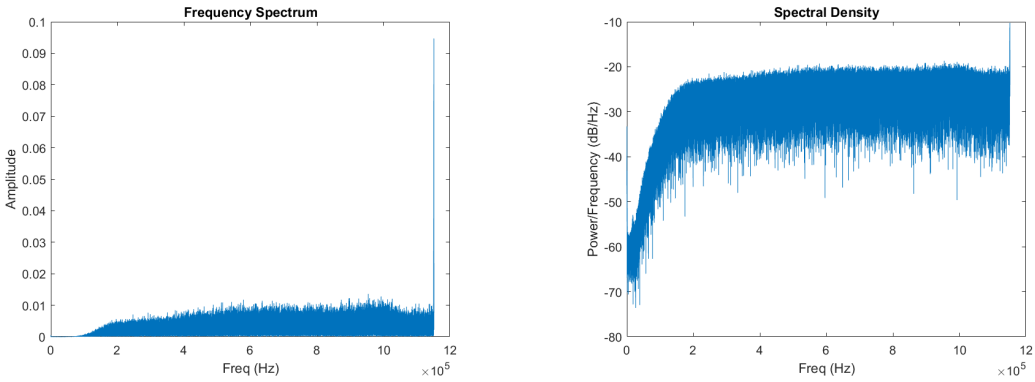
Required Multiplications/seconds: We can make a rough calculation of the required amount of mul/s for this multi stage filter design:

- CIC: 0
- HB1: 7 x 96k = 672k mul/s
- HB2: 7 x 48k = 336k mul/s
- FIR: 32 x 48k = 1536k mul/s
- Total: 2544k mul/s

If we had used a single FIR filter we would need a 197th order FIR filter to achieve the desired design constrains. This would result in an average of 9456k mul/s. This is almost a difference of x4 compared to the multi stage filter design.

4.2.3 Matlab Simulation

To convert the PDM data to PCM data we made a Matlab script that implements this multi stage filter design. As test signal we use a 200 Hz Sine wave in PDM format that is recorded in the sound chamber. The input PDM sampling frequency is 2.304 MHz and the output PCM sampling frequency 48 kHZ. The spectrum of the PDM signal looks as follows:



(a) Frequency Spectrum (b) Spectral Density

Figure 21: Spectrum PDM data

The PCM output signal looks as follows:

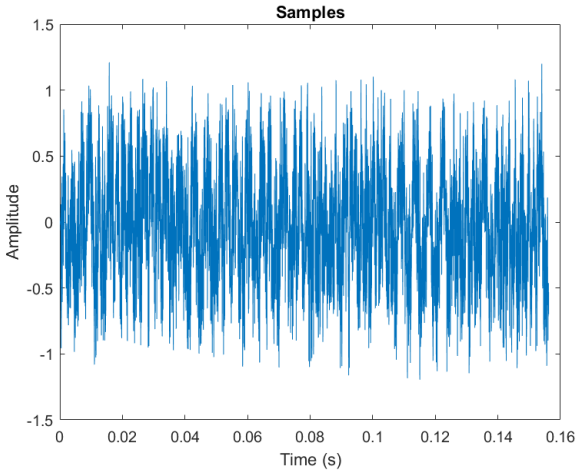
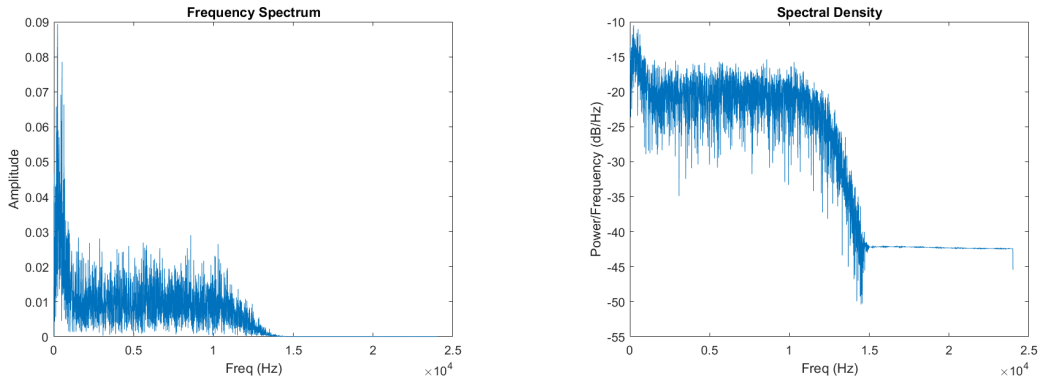


Figure 22: PCM Output Signal

And the spectrum:



(a) Frequency Spectrum

(b) Spectral Density

Figure 23: Spectrum PCM data

The largest peak in the spectrum has a frequency of 236.8 Hz according to Matlab. This is relatively close to the original 200 Hz Sine wave.

4.3 Total DSP Setup

4.3.1 Overview

The complete DSP setup is best represented by a block diagram:

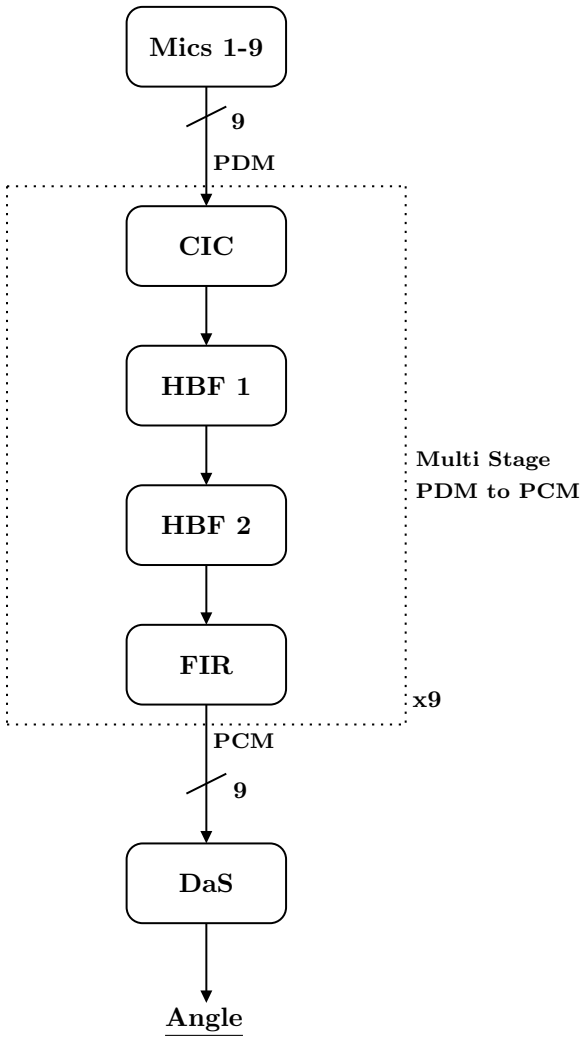


Figure 24: DSP Block Diagram

Note that we implement the filter setup 9 times, 1 for each microphone. Because we are using an FPGA, it is possible to perform all these operations in parallel an have a separate filter stages per microphone.

4.3.2 Total Needed Calculations

We know that the multi stage filter design requires a total of 2544k mul/s and the DaS algorithm requires 3240 additions per sample. Considering that we have an output sampling frequency of 48 kHz, we would need to perform 155 520 000 additions/s (see Section 3.2.4).

4.3.3 Memory Consumption

Considering that we want to implement everything in a stream, minimal storage is needed. The only place where storage would be needed is a buffer before DaS algorithm. This buffer would be chosen experimentally. Let's say we can buffer 5 second of sample captures. Our buffer size would need to be:

$$\begin{aligned} Buffer\ Size &= Microphone\ Amount \times Sample\ Amount \times Sample\ Size \\ &= 9 \times (5 \times 48000\ Samples) \times 16\ bits \\ &= 34.560.000\ bits = 4.32\ MB \end{aligned}$$

As we can see, the memory consumption is minimal. Because of this we can adjust the buffer size as needed.

4.4 VHDL Implementation

As test platform we used the MYIR Z-turn Board V2 containing the Zynq 7020 with the IOCAPE.

4.4.1 Reading out PDM Microphones

The first step was to read out the PDM microphones. We connected the microphone array to the IOCAPE using the PMOD connector. Because the IOCAPE has a different pin layout than the FPGA we had to cross-correlate the pins from the IOCAPE to the FPGA.

The Block Design for the VHDL code looks as follows:

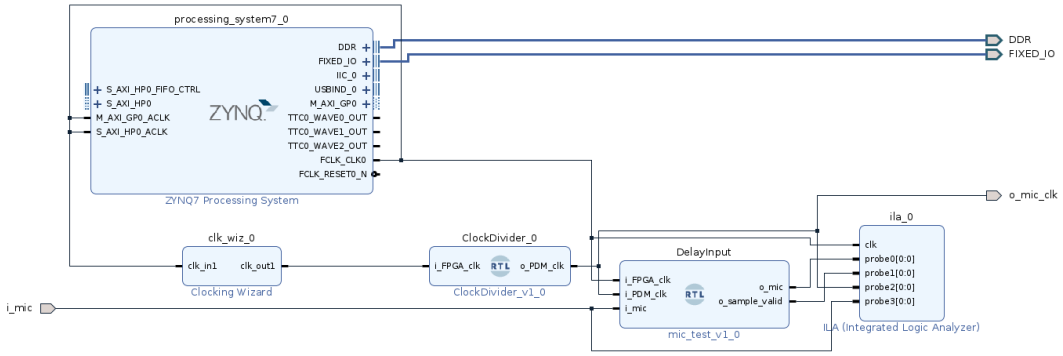


Figure 25: PDM Microphone Read Vivado Block Design

The Zynq 7 Processing System let's us control the PL Fabric of the FPGA. First we create our PDM clock by using an intermediate Clock Wizard to downgrade the 100 MHz FPGA clock. Next we created a clock divider to divide the intermediate clock to the PDM clock frequency of 2.304 MHz. Next we delayed the input samples by 70 nanoseconds. This value accounts for the delay in the silicon of the PCB and the delay in the cable between the microphone array and the FPGA.
To check our results we used the ILA IP Core from Xilinx. We've set a probe on the PDM output, clock and valid signal. This valid signal is used to indicate when to sample the PDM output for the ILA core. Note that we only use 1 microphone but this is easily expanded to

multiple microphones by changing the in and outputs to vectors

Programming the FPGA is done via JTAG. When programming the FPGA, the debug window pops up where we can interact with the ILA Core. In here, a buffer size can be chosen and the samples can be captured and saves as a .CSV file. These samples can than loaded into the Matlab script to test the result.

Because the ILA Core works at the FPGA clock frequency (100 MHz), the samples in the lower frequencies are over sampled and the necessary information is lost. We can see this when taking the FFT of the obtained samples:

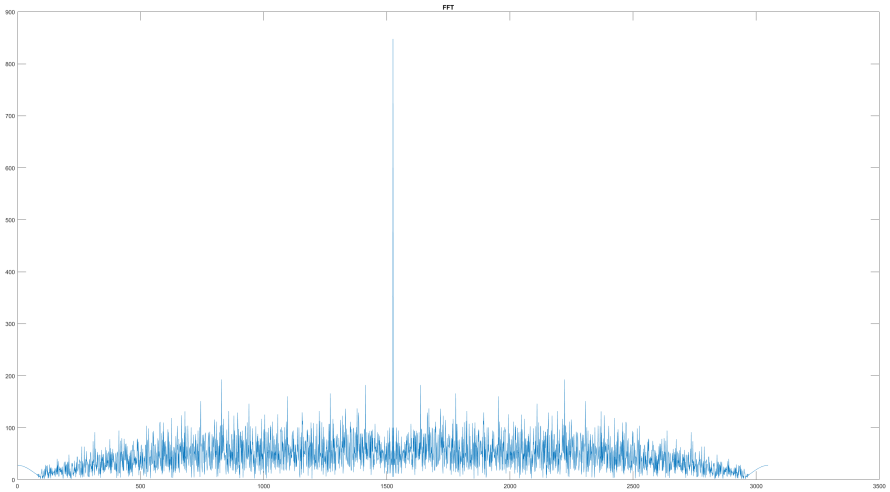


Figure 26: FFT of PDM signals

We can see that we get weird behavior in the lower frequencies.

Because of this we used the UART code and module from our professor to check the PDM results. When using this method, we achieved good results. See section 5.4

4.4.2 Multi Stage Filter Design and DaS

We know from the labs that there are IP-cores available for the filters in Vivado. Configuring them is straightforward. The coefficients can be calculated with the Filter Designer tool in Matlab. Remember that to keep the 16-bit integer notation, we also have to map the coefficients to this 16 bit integer range and set the ouput of the filters to 16 bit in the IP configurator. We can also select the amount of channels (in our case 9). The filters use an AXI-stream interface. This means that when cascading the filters, we can just connected to M and S -sides of the AXI interfaces to each other. When passing our PDM data to the first filters however we would need to convert our data to an AXI-stream interface. To do this we can use the available template in Vivado.

After these filters, the DaS block would follow. Also here we would have to make the conversion from the AXI interface to a 16 bit vector output. The ouput of the DaS is a single value that would be outputted to an IO pin.

Because we were short on time, we didn't implement this.

5 Microphone Array

5.1 Microphones

To be able to test the beam forming algorithm on real hardware it was necessary to design and manufacture a physical microphone array. Before designing any electrical drawings, we must first consider what type of microphone technology we would be using. In the field of embedded systems there are two kinds of microphone technologies that are frequently used, MEMS (Micro-Electro-Mechanical Systems) and ECM's (Electret condenser). In Table 1 we find a comparison of these two types of microphones.

Table 1: Comparison of MEMS and ECM microphone

feature	ECM Microphones	MEMS Microphones
Technology	Traditional condenser	Microfabrication based
Size	Larger	Smaller
Output	Analogue	digital
Power Consumption	Higher	Lower
Sensitivity	High	High
Frequency response	Wide	Wide
Signal-to-Noise ratio	High	High
Distortion	Low	Low
Cost	Lower	Higher
Packaging	Non smd	smd
Stability	Can drift over time	stable over time
Durability	Robust	Fragile
Sensing principle	Electrostatic	Piezoresistive or capacitive
Application	Consumer electronics	Mobile,wearables, IoT

After carefully investigating these different features of each type of microphone we can conclude that MEMS will be the best suited solution for our case. Firstly in chapter 5 we talk about using a circular microphone array holding 9 elements. This array is meant to be placed under a bicycle saddle resulting in quite some size restrictions. MEMS technology being much smaller then the classic ECM's therefore gets really interesting due to the size comparison. On top of that, MEMS are packaged as SMD components. Considering that it can be soldered on the surface of a PCB will also save quite some place in the overall structure of the array. Secondly a MEMS typically consumes less power than it's classic counterpart, in this small battery dependent system a low power microphone would be a great advantage. If we look at the typical application use of both technologies it becomes quite clear that using MEMS is a viable option for us as it is used in small embedded systems like noise cancellation units and mobile phones.

Now we need to figure out what type of MEMS microphone we will work with. Firstly we

have the choice between analogue and digital variants. Analogue MEMS have slight advantage in price & power consumption. These are important factors for the array but they don't outweigh the fact that there is still need a of an ADC and some extra pre processing that needs to be done before we could apply the beamforming algorithm on it. Digital MEMS come out of the box with an I2S or PDM interface. This makes it more interesting due to the ease of integration on microcontrollers or FPGA's. We can conclude that the cost of implementing an analogue microphone does not out way the possible advantages of an analogue microphone. In table 2 we find a comparison between analogue and digital MEMS

Now that we have established that we will be using digital MEMS microphones we have to think

Table 2: Comparison analogue & digital MEMS

feature	Analogue MEMS	digital MEMS
Output	Analog current or voltage	Digital data(I2S,PDM)
Cost	Lower	Higher
Noise	prone by external noise	Better noise performance
Power	Lower	varies depending on processing
DSP	Minimal to none	High DSP capabilities
Integration	moderate	straightforward (I2S,PDM)

about what type of output data we will want to work with. I2S has the most advantages in audio quality and interfacing, it is widely supported in audio systems and has higher bit depth than PDM. But PDM has a lower power consumption and less data bandwidth due to its single-bit stream. PDM output is a good option for power constrained applications, in our case where we use 9 MEMS continuously this seems as a logical choice to use digital PDM MEMS. In table 3 we find a comparison between the two different output data.

Table 3: Comparison I2S & PDM output

feature	I2S	PDM
Data Format	Serialized audio data	Single-bit stream
Output lines	Separate lines for data, clock and sync	Single data wire and clock
Compatibility	Widely supported in audio systems	Use for power constrained applications
Power	Higher	Lower
Audio quality	Better due to higher bit depth	Lower
Data bandwidth	Higher (multi bit)	Lower (single bit)

5.2 Design and Geometry

As explained in section 3.2.1, the geometry of the microphone array is very important factor in the overall performance of our application. There are several factors that have to be taken into account when designing the microphone array. One of them is the general shape. We used a semicircle. Not a full one because the microphone array will be mounted on the seatpost of the bike and the localization has to be done behind an around the bike. So a full circle is not necessary here. Using a circle keeps the math relatively simple.

Another important factor is the radius/diameter of the array. It determines the maximum delay possible as shown in section 3.2.1. Choosing this radius/diameter affects the ability to distinguish two successive signals or in other words, how many samples that can be captured

between the first microphone receiving the signal and the last microphone receiving the signal *(Assume that the largest distance between the microphones is equal to the diameter)*. Let's take a semicircle microphone array that has a radius of 20 mm and compare this to an array that has a radius of 200 mm and calculate the max delay using the formula from section 3.2.1:

$$R = 20mm : \frac{0.04m \cdot 48000\frac{samples}{s}}{343\frac{m}{s}} \approx 6samples$$

$$R = 200mm : \frac{0.4m \cdot 48000\frac{samples}{s}}{343\frac{m}{s}} \approx 60samples$$

We can see that using a large radius results in a larger amount of samples that can be captured between 2 successive signals.

The amount of microphones used, determines the precision of the microphone array. Let's say we would only use 2 microphones with a radius of 200 mm. This means we have a maximum delay of 60 samples. When using 2 microphones we can only evaluate at either the first microphone with a delay of 0 or the second microphone that has a delay of 60 samples. When in contrast we are going to use multiple microphones, we can evaluate the signals at more points thus increasing the accuracy. Let's say we use 5 microphones instead of 2:

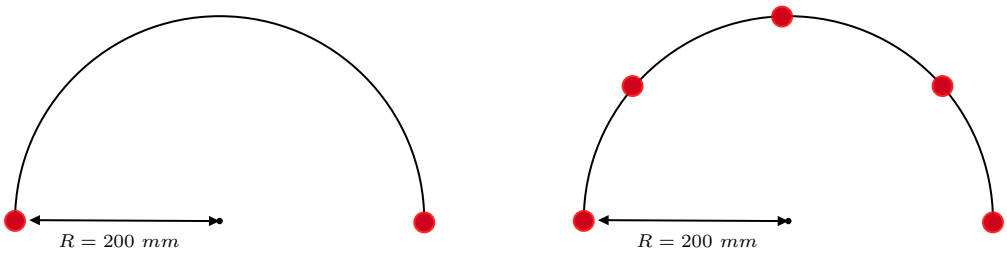


Figure 27: 2 vs 5 Microphone Array Setup

Instead of only evaluating the incoming signal at the first and last microphone, we can now also evaluate them at the intermediate microphones. If we assume that they are located at an angle of 45, 90 and 135 degrees than they will have the following delays:

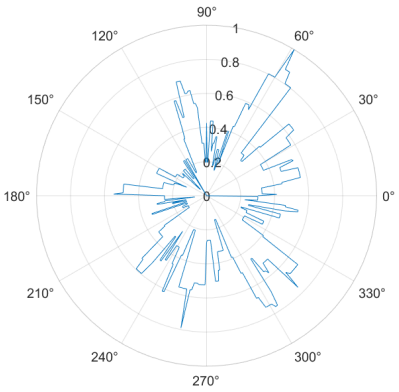
$$\text{2 mics: } 0, 60$$
$$\text{5 mics: } 0, 15, 30, 45, 60$$

We can see that we have now more delays and this means we can more accurately distinguish the incoming signal.

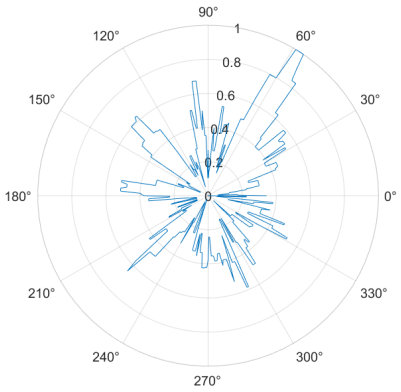
Using CABE we simulated different microphone arrays with varying radii and microphone amounts. This in combination with the Matlab simulation allowed us to choose the optimal microphone array geometry. We used the following setups:

- A constant radius of R = 60 mm and a variable microphone amount of: 5, 7 and 9.
- A constant amount of microphones of 9 and a variable radius of: R = 40 mm, R = 60 mm and R = 80 mm.

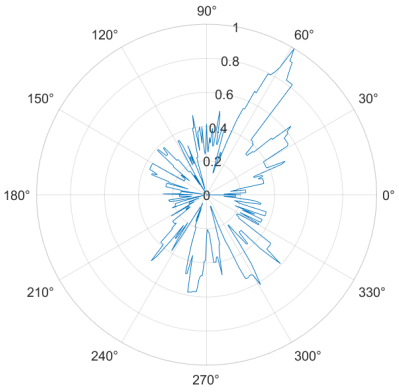
These are the results from the Matlab simulation:



(a) R = 60 mm, 5 mics

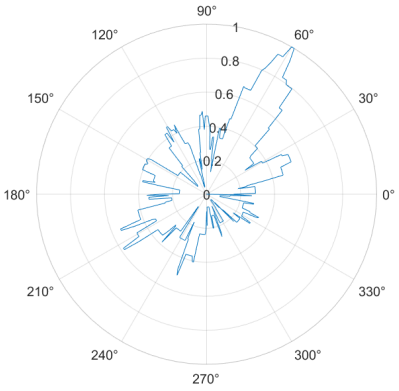


(b) R = 60 mm, 7 mics

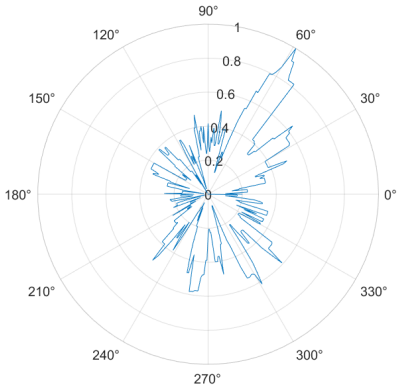


(c) R = 60 mm, 9 mics

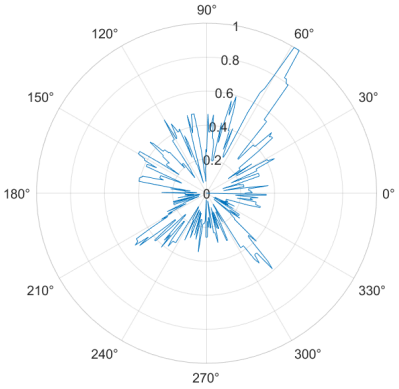
Figure 28: Constant radius and variable microphone amount



(a) R = 40 mm, 9 mics



(b) R = 60 mm, 9 mics



(c) R = 80 mm, 9 mics

Figure 29: Variable radius and constant microphone amount

We can see that using 9 microphones gives us the best result. On the other hand, increasing the radius from 60 mm to 80 mm didn't drastically change the overall accuracy. A radius of 60 mm is also a good size to fit on the bike.

5.3 PCB

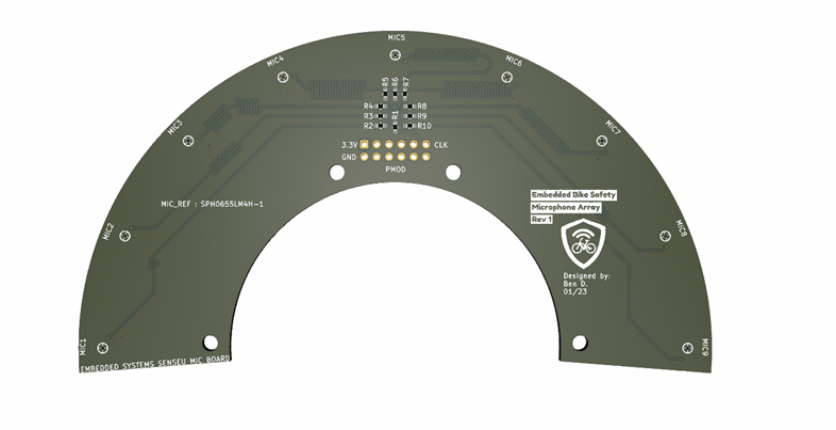


Figure 30: 3D Rendered Microphone Array PCB

The PCB has been designed completely in KiCad. All of the design rules were interpreted of the standard Eurocircuits rules for KiCad [7].

First it was necessary to accurately place all microphones on the circular PCB at an and equal distance from one another, also the angle of each microphone needed to be correctly set relatively to the position on the PCB. This was done by drawing a reference circle with the same diameter as the hole of our MEMS and placing one on the right position at 0°, 90° or 180°. Then using the "create from selection >>create circular array" option in KiCad it was possible to create a circular array with the correct positions for each microphone. Now placing all the MEMS on the reference circles and changing the angle's of each microphone relative to the PCB as shown in figure 31.

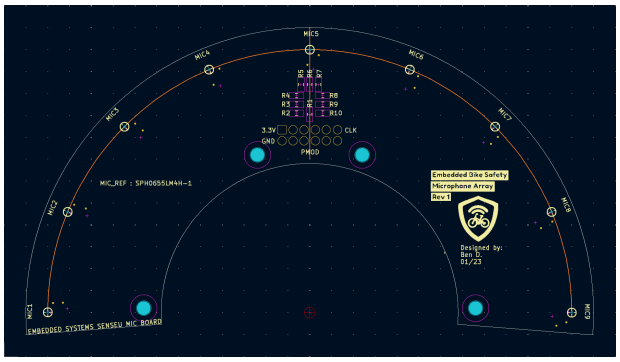
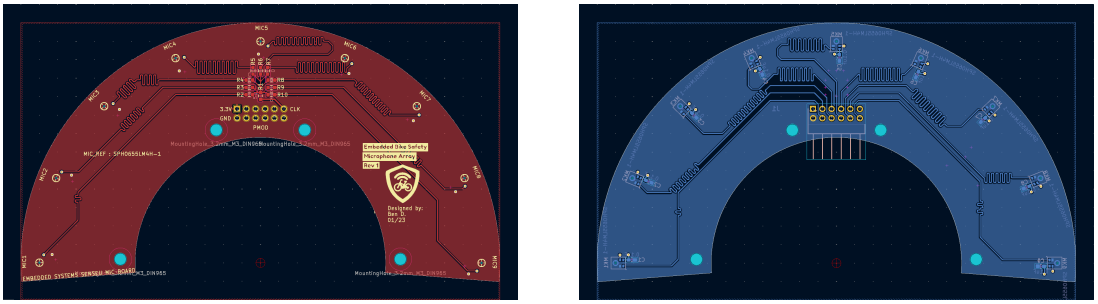


Figure 31: Microphone placement

The greatest challenge of designing the PCB were the constraints in routing area, due to the circular shape of the PCB it was quiet difficult to properly route all signals to the microphones.

[7] Eurocircuit rules for KiCad

Part of this issue was overcome by using a two layer PCB consisting in a ground plane (Bottom side) and a VDD plane (Top side) and connecting ground and VDD pads of the components directly (or through vias) to the copper planes. This resulted in reducing the number of routed tracks from 38 tot 22. Figure 32 show both copper planes.



(a) Top side

(b) Bottom side

Figure 32: Ground and VDD copper planes

Only the PDM and CLK signals still need routing at this point. These need to be routed using a specific tool named track tuning, this tool is used by routing all CLK signals and using the longest route in this specific net (CLK-net) as reference to tune the other routes to the same length as the reference length. This provides an easy way to make sure that all routes of the CLK signals going to each individual microphone have the same track length. This is an important factor because the microphones need to be synced at the same clock frequency to not create delays that would affect the beam forming algorithm. The same is applied for the PDM-net.

Specifically on the CLK-net it was necessary to first use 9 jumper resistors (0 Ω resistors). This is done to divide one CLK line over 9 separate CLK lines. By adding these jumpers it is possible to use length tuning for the separate CLK lines.

Connection wise we opted for the PMOD standard. This is a simple 2 by 6 row, right angle connector that we assigned some logic, CLK, VDD and ground pins to. This is a straightforward way of connecting a PCB to another module. The placement of our PMOD is a bit unfortunate because of the inner circle, this makes it hard to connect a module with PMOD straight to the PCB. Placement was restricted due to length tuning, so it would be advised to use a PMOD cable connector.

Lastly we used four M3 holes so the PCB could be mounted in some designated case. The final design of the PCB is shown in figure 33.

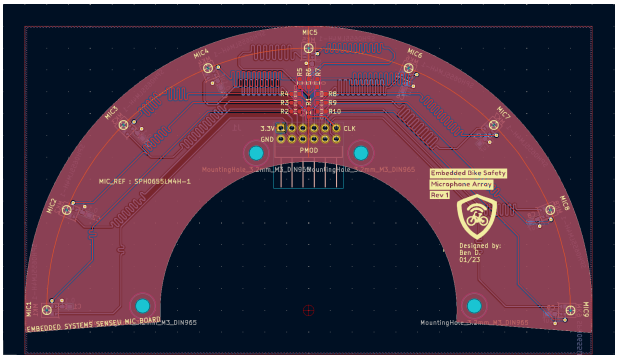


Figure 33: Final PCB design

5.3.1 PCB Manufacturing

The PCB boards were ordered on the website Aisler [**AISLER**]. We ordered three PCB's costing us a total of 105.71 EUR including stencils for assembling the PCB. Aisler was the choice of manufacturer because they make PCB's that are compatible with the eurocircuits solder paste stencil printer that we will be using for assembly of the PCB.

Once the boards arrived we started assembling the PCB, first we used the solder paste stencil printer to apply the solder paste on the PCB pads. Next we placed all the components on the PCB by using a precise manual pick and place machine, once this was properly done the components needed to be soldered to the PCB. This was done by using a re-flow solder oven. After this process we had to check if all the components were properly soldered on the PCB, this was simply done by checking all signals using a multimeter.

The first PCB that we assembled had a faulty microphone, this was probably not soldered properly on the board and was giving some short circuits, by removing the component and soldering a new one by hand on the board this was quickly fixed. In the process of fixing the first PCB we also assembled a second PCB that worked perfectly from the first time.

5.4 Testing the Microphone Array

The test method is already described in section 4.4.1. The photo below is a FFT of the obtained PDM signals from our microphone array. The test signal was a 2 kHz sine wave. We can see that the shape of the FFT is what we expect from a FFT of a PDM signal and that we can retrieve the 2 kHZ signal in the FFT.

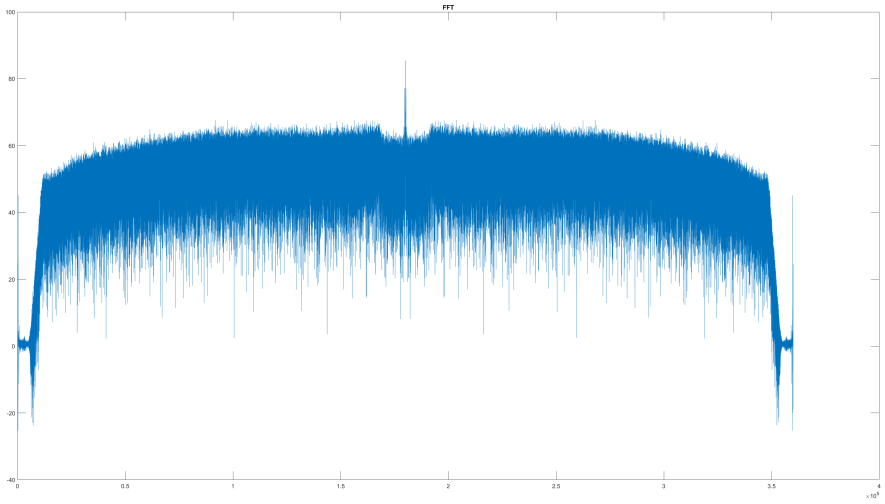


Figure 34: FFT of PDM signal

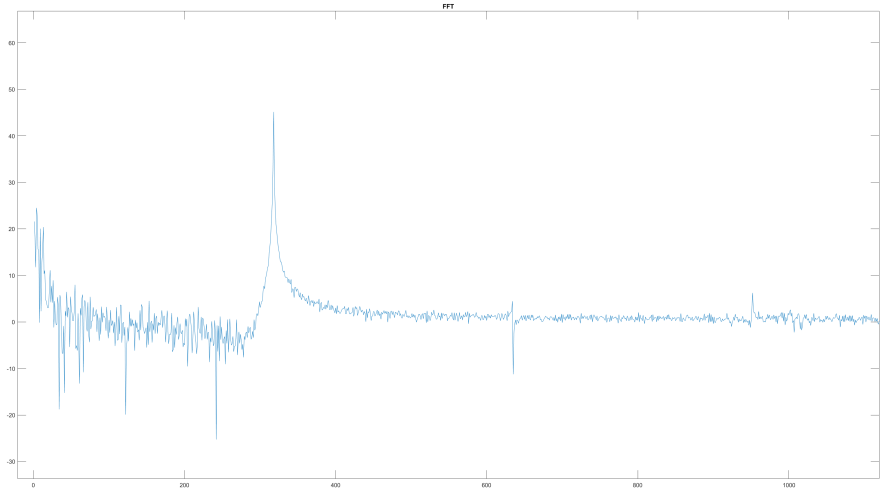


Figure 35: Captured 2kHz sine wave (x-axis not correct)

6 Sense-U

Sense-U is the part where all the processing is done in real time. The processing can be done on a microcontroller or an FPGA. This section presents a comparative analysis of these two hardware platforms, helping in the selection of the optimal option for implementation. Following the hardware platform comparison, the implementation will be developed in detail. The Sense-U

embarks also a wireless communication that will be further discussed in the Display-U section. To finish this off, a cost breakdown will be presented for the production of the Sense-u.

6.1 Real-Time Processing in Sense-U

In the field of digital audio signal processing, beamforming is a technique used to enhance the directional sensitivity of microphone arrays. The goal is to detect and isolate sound sources from a specific direction while suppressing noise and interference from other directions. When it comes to implementing beamforming algorithms in real-time applications, two hardware platforms commonly considered are FPGAs and MCUs. This section will compare these two platforms. By evaluating their capabilities and considering specific requirements, it is possible to determine which platform is better suited for this application.

6.2 FPGA vs. MCU

When choosing the appropriate platform it is recommended to analyze the necessary capabilities. To perform beamforming, the first step is to filter the input signals to eliminate the unwanted noise and enhance the target signals. Regarding the filtering, multiple questions can come to mind:

- What is the desired frequency range for detecting cars?
- What type of filters are needed (e.g., low-pass, high-pass, band-pass)?
- What is the required filter order or complexity?

Next step is applying the beamforming algorithm to the filtered signals to estimate the direction of the cars. Again, when applying the beamforming algorithm following questions must be answered:

- Which beamforming algorithm will be used ?
- What is the desired beamforming resolution and accuracy?
- Are the computations extensive ?

The hardware platform also needs to perform the computations in real time. A biker needs to get the information as soon as possible to be able to adapt his path or paying more attention on the road. If the information comes 10 seconds after the detection of a car, the device can be seen as useless. The real time processing also has some points to be considered:

- What is the required sampling rate for the microphone signals?
- What is the acceptable latency or delay in processing the signals?

This information is already discussed and further explained in previous sections. FPGAs, with their ability to parallelize computations and their low-latency characteristics, often excel in real-time applications. MCUs, on the other hand, might have limitations in terms of processing power and I/O capabilities, which could affect real-time performance.While MCUs might be suitable for simpler filtering and less computationally intensive beamforming algorithms, they may struggle with real-time performance and complex mathematical operations. On the other hand, FPGAs offer high-speed parallel processing, efficient filtering capabilities, and the ability to handle complex beamforming algorithms effectively. Therefore, considering the requirements of this specific application, an FPGA seems to be the more favorable choice for optimal performance in real-time car detection using beamforming.

6.3 Implementation

This design will use the FPGA from Xilinx, the Zynq 7020. Because routing a BGA chip is not yet part of our skillset we will use a programmable SoC module based on that chip. There are multiple possibilities considering the SoC modules like :

- The Trenz Electronic GmbH TE0720-03-64I63MA
- The Myrtech MYC-C7Z010/20-V2 module
- The enclustra Mars ZX3

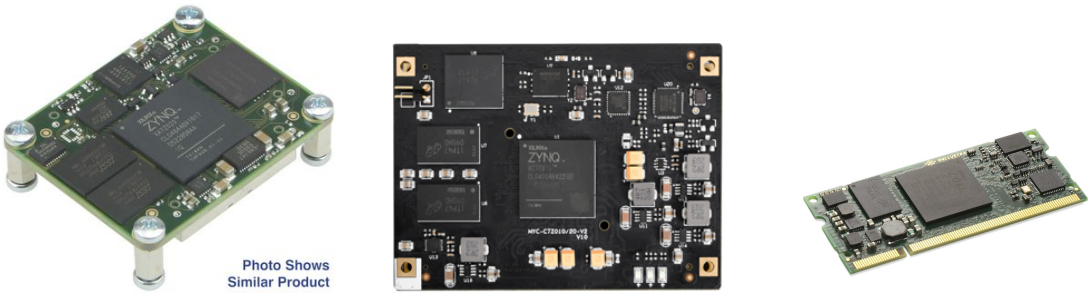


Figure 36: Trenz, Myrthech and Enclustra SoC

The main factor determining the choice of the SoC module for this project is its availability. While we could compare the different modules based on their features and benefits, it becomes clear that any of them would work well for this project. So, the main considerations are the price and whether the module is currently available. Out of all the options, the only one that is currently accessible in the market on digikey is the Enclustra Mars Zx3. For the convenience it is also easier to find SO-DIMM connectors on the market which will simplify the implementation. Therefore, we will focus on designing the project around this particular SoC module.

Some important features of the Mars ZX3:

- Xilinx Zynq-7020 AP SoC in the CLG484 package (XC7Z020)
  - ARM® dual-core Cortex™-A9 (32 bit, up to 766 MHz)
  - Xilinx Artix™-7 28nm FPGA fabric
- SO-DIMM form factor (67.6 x 30 mm, 200 pins)
- 108 user I/Os
  - 12 ARM peripheral I/Os (SPI, SDIO, CAN, I2C, UART) shared with FPGA I/Os
  - 96 FPGA I/Os (single-ended, differential or analog)
- Gigabit Ethernet and USB 2.0 OTG PHYs
- Up to 1 GB DDR3 SDRAM
- 512 MB NAND flash
- 64 MB quad SPI flash
- 3.3 V tolerant inputs
- Single 3.3 V supply voltage

Figure 37: MARS ZX3 Features

Considering the necessary requirements of Section 4.3.2 and 4.3.3 we can see that we have plenty of memory available to store the samples. In this case we would use the NAND Flash memory for the faster read and write speeds that are necessary for the DaS algorithm.
We also know that the Zynq-7020 can do single and double precision floating point up to 2 MFLOPS/MHz each [8] or 200TFLOPS/s. Considering we need to do 2544k mul/s and 115.52M additions/s, this will also be more than sufficient.

6.3.1 Implementation of the Enclustra Mars ZX3

The Mars ZX3 SoC Module is a powerful and complete embedded processing system that combines Xilinx's Zynq-7020 All Programmable SoC device with DDR3 SDRAM, NAND flash, quad SPI flash, a Gigabit Ethernet PHY, and an RTC. It comes in a compact SO-DIMM form factor, allowing for space-saving hardware designs and easy integration into the target application. The Mars ZX3 SoC Module is designed to reduce development effort, minimize redesign risks, and speed up time-to-market for embedded system.

As previously discussed the Mars ZX3 has an SO-DIMM DDR2 SDRAM form factor with 200 pins.

[8] Zynq-7000 SoC Data Sheet: Overview (DS190)

Figure 38: So-Dimm form factor

This makes it easier to route the FPGA on a PCB instead of routing a whole BGA FGPA on the board. On the Enclustra website a file can be found with the Pinout of the SoC. The most important pins used will be shown in the table below.

Pin number	**Pin Type**
1-3-5-7-9-11-197-199	3,3V
102-104-108-110-109-111-113-115-119	PDM signal
118	clk
143	UART NCTS
145	UART NRTS
153	UART RX
155	UART TX
159	USB D+
161	USB D-
176	SDA
178	SCL

Table 4: Pin usage MARS ZX3

6.3.2 Consumption of the Zynq 7020

We can roughly establish a table for the consumption of the Mars ZX3. The FPGA will always be in the ON state because of the real time processing it needs to perform while the biker is riding:

Symbol	Power Consumption	Unit
$I_{CCPINTQ}$	122	mA
$I_{CCPAUXQ}$	13	mA
I_{CCDDRQ}	4	mA
I_{CCINTQ}	78	mA
I_{CCAUXQ}	38	mA
I_{CCOQ}	3	mA
$I_{CCBRAMQ}$	6	mA

Table 5: Consumption Zynq 7020

6.3.3 UART communication

To facilitate the communication between the computer and the SoC module and to program the FPGA, the UART protocol will be used. To be able to use UART we implemented a USB-to-UART chip, the SparkFun Accessories USB to Serial IC - CH340E. It is an easy applicable chip which will convert the D+ and D- of the usb port to UART. The connection is then made with the UART pins showed in table 4. Her the block diagram of the CH340E:

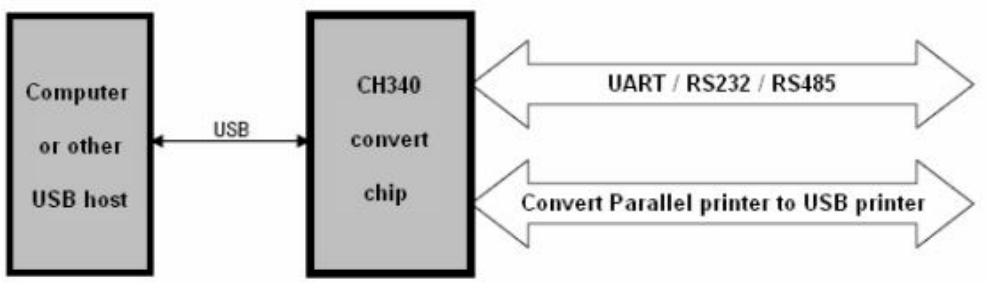


Figure 39: CH340E block diagram

The following table shows the average power consumption. Note that when the device is used, the CH340E is in low power mode.

Symbol	Power Consumption	Unit
I_{CC}	7	mA
I_{SLP}	0.1	mA

Table 6: Consumption CH340E

6.3.4 Peripherals

Aside from the previous important implementations. There is also a dedicated PMOD connection that has been added to be able to connect to the microphone array. For the wireless communication the Nordic nRF52820 is used and has been detailed in the Display-U section. A USB type C port has also been added.

6.3.5 Sense-U PCB

The PCB containing the SO-DIMM slot for the FPGA and the other necessary peripherals can be seen below:

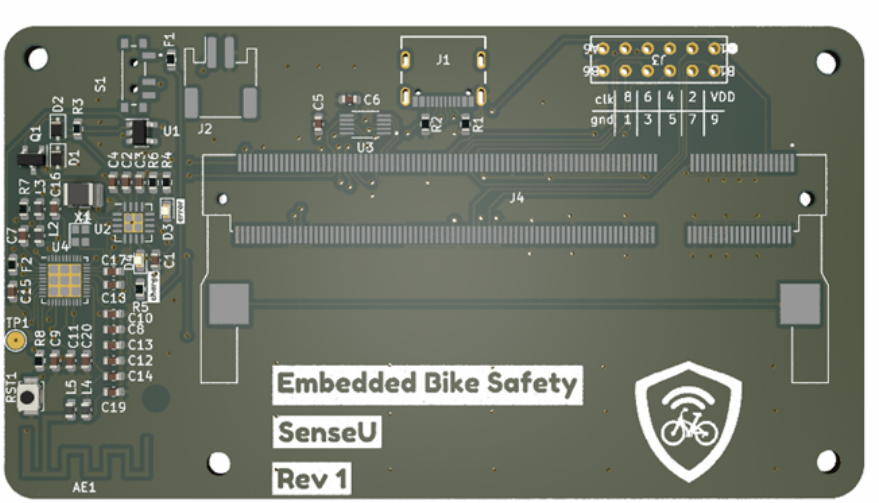


Figure 40: Sense-U PCB

6.4 Estimated Battery Life

The total power consumption is based on the average current draw of the MARS ZX3 and the CH340E IC. When the device is being used, the CH340E is in sleep mode.

$$\begin{aligned} I_{Total} &= I_{Total,Zynq7020} + I_{SLP,CH340E} \\ &= 264\ mA + 0.1\ mA \\ &= 264.1\ mA \end{aligned}$$

Considering that the Display-U has an average battery life of apprx. 16 hours, we need to make the battery big enough to also support this battery life for the Sense-U.

$$\begin{aligned} Battery\ Capacity\ &=\ Current\ draw\ \times\ Battery\ Life \\ &= 264.1\ mA\ \times\ 16\ hours \\ &= 4225.6\ mAh \end{aligned}$$

A battery with a capacity of 4300 mAh should be sufficient.

6.5 Cost Breakdown

Unit price for an order of 1000pcs, a combination of Digikey and Mouser was used to gather the prices. Batteries are bought from a Chinese wholesaler. These types of batteries are rarely stocked on the common electronics wholesalers. Prices for a 1s Lithium Polymer battery with a capacity of 300mAh range from 0,55 EUR to 2.7 EUR . This was rounded to 1 EUR to simplify calculations.

component	**Description**	**price**	**qty**	**total**
Microcontroller	nRF52820-CFAA-D-R7	2.04	1	2.04
Common P-ch	AO3421E	0.09326	1	0.09326
USB-C Port	USB4105-GF-A	0.0383	1	0.0383
LDO	AP2112K-3.3TRG1	0.10075	1	0.10075
Charger	LTC4001EUF#TRPBF	0.969155	1	0.969155
Battery Connection	S2B-PH-SM4-TB(LF)(SN)	0.27237	1	0.27237
Status LED	LTST-C191KGKT	0.04390	3	0.1317
0603 passives	Estimate	0.0004	48	0.0192
Battery	Chinese 3.7v 300mAh	1	1	1
USB to UART	CH340E	16.74	1	16.74
Enclustra SoC	Mars ZX3	415.23	1	415.23
PMOD	CGRID DR RA PN 675290MM TNA 12CK	0.97	1	0.97
PCB Assembly	JLC PCB	13.6667	1	13.6667
PCB Manufacturing	JLC PCB	0.2	1	0.2
Total				**460.5385**

Table 7: Price table Sense-U

Note that this is an unrealistic price for this device. This is because we used the Zynq 7020 FPGA which forced us to us a SoC which price lies on the higher side. The price for this PCB could be much lower if instead of using the Soc we used directly the chip itself. The problem here is not having enough experience in PCB design to implement such chip in our design. Here is a comparison with the chip used instead of the Soc Module. We see that the price is divided by more than 3. Subsequently there are cheaper FPGA chips that can do the job where the price lies lower than the zynq 7020 so the total cost can again be lowered.

component	**Description**	**price**	**qty**	**total**
Microcontroller	nRF52820-CFAA-D-R7	2.04	1	2.04
Common P-ch	AO3421E	0.09326	1	0.09326
USB-C Port	USB4105-GF-A	0.0383	1	0.0383
LDO	AP2112K-3.3TRG1	0.10075	1	0.10075
Charger	LTC4001EUF#TRPBF	0.969155	1	0.969155
Battery Connection	S2B-PH-SM4-TB(LF)(SN)	0.27237	1	0.27237
Status LED	LTST-C191KGKT	0.04390	3	0.1317
0603 passives	Estimate	0.0004	48	0.0192
Battery	Chinese 3.7v 300mAh	1	1	1
USB to UART	CH340E	16.74	1	16.74
AMD Zynq 7000 serie	XC7Z010-1CLG225I	79.66	1	79.66
PMOD	CGRID DR RA PN 675290MM TNA 12CK	0.97	1	0.97
PCB Assembly	JLC PCB	13.6667	1	13.6667
PCB Manufacturing	JLC PCB	0.2	1	0.2
Total				**124.9685**

Table 8: alternativce price table Sense-U

7 Display-U

Display-U is the display portion of our Embedded Bike Safety system. The device must display the data that is measured and processed by the Sense-U.

Design goals:

- Receive data wirelessly from the Sense-U
- Signal whether a car is coming from left or right (during bright sunlight)
- Signal the distance to the detected car
- Battery Powered & Rechargeable
- A minimum of 1 commuting week of battery life under normal use

7.1 Appearance, Size and Signaling

From the user's perspective Display-U appears as a low-profile bicycle computer. A thin enclosure is used to conceal the electronics while having see through sections for the signaling operation. Exterior design goals:

- Enough space between the 2 LED's
- Do not interfere with normal cycling operation

Dimensions: 85 x 35 x 17 mm
Material: ABS Injection moulded

This is a purely conceptual design. The mounting to the bike's handlebar & manufacturing were not taken into account. In an industry setting this would be handled by a different group of engineers.

Figure 41: Conceptual enclosure featuring 2 holes for signaling the cyclist

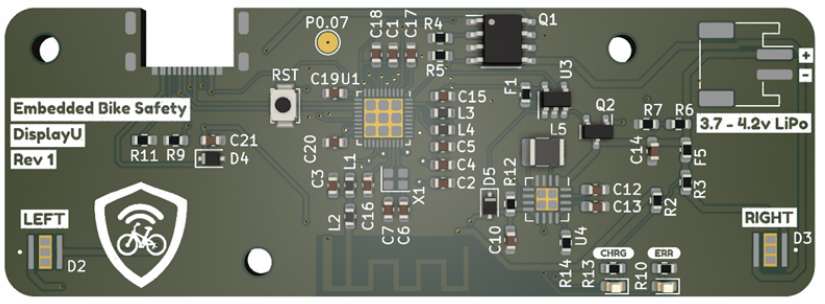


Figure 42: PCB Front View

The PCB only uses SMD components on the front layer. It is mounted to the enclosure using M3 fasteners.

7.2 Wireless Protocol

Since our system only needs to inform the bicycle rider when a car is nearing on him from behind, the Sense-Unit should only advertise to the Display-Unit when this action is happening. Especially due to the limited battery capacity it wouldn't be a good practice to continuously send data from one unit to another as this would consume a fair deal of power. Therefore, we are in need of a lightweight wireless protocol that only sends data in short intervals when necessary. This gives us two options of choice: BLE (Bluetooth Low Energy) and ANT+. Both these

options work with short bursts of data and are energy efficient designed protocols.

In terms of data transfer rate, both would be sufficient to transfer the amount of data that we would need to send from the Sense-Unit to the Display based on the fact that both protocols can achieve up to 1Mbps of data throughput if needed.

A strong recommendation for the wireless protocol is that it needs to have a fast and seamless connection between the two units. Both protocols should be a good choice for this as they are commonly used in applications where this also should be the case. BLE is used a lot in smart devices, IoT, wearables and fitness trackers, ANT+ is more specifically used in fitness and health applications like fitness sensors and cycling power meters etc...

Why ANT+?

Looking at the features of these protocols we could conclude that both are a viable option to use as a wireless solution for our system. But if we look at the application focus of the protocols it would be a logical choice to choose the ANT+ protocol, it is already widely used on bicycles specifically on cycling power meters, the concept of data communication is strongly the same with what we need to achieve. It's also a protocol that is supported by Garmin, a brand that is widely renowned for its embedded bicycle gear. From the stats there wouldn't be any reason why not to use the ANT+ protocol as it is made for this kind of data transfer.

Why BLE?

BLE protocol is an answer from classic Bluetooth protocol on the ANT+ protocol. They tried to come up with a standard that answers the need for low power small data simple wireless connectivity, and they have succeeded at it. The main reason why we would choose BLE over ANT+ is for the ease of implementation. As mentioned above, BLE is derived from the well know Bluetooth protocol, this means that there is tons of documentation and support on implementing this technology. Also, when looking at hardware to implement both protocols, BLE seems to be the most used and common form of low energy wireless data communication. ANT+ on the other hand has only one real source of documentation for their technology (ANT+ documentation). This means that BLE would have a much shorter time to market and is less prone to a vendor lock in as it is not maintained by a Tech manufacturer like ANT+ (Garmin) but instead is maintained by an well know organization that has been around for years and has been used in applications all sorts of domains all over the world.

Integration will depend on the possible component choices. There are typically 2 approaches. Either by using a wireless IC or a System on Chip (SoC). BLE enabled products are often built using a SoC. Having the control unit and networking unit inside 1 package can benefit integration, cost, and power efficiency. Because the requirements for a low power BLE/ANT+ device are quite common over the industry most SoC's can accommodate most designer's needs.
Another solution is to use a separate microcontroller and wireless chip. This increases the design flexibility of the project. Additionally, having the parts separate can aid in the upgradeability of a future iteration of the device.
For our purposes it was decided to go for the SoC approach.
For the rest of this document, we will be referring to our control unit as a RF-SoC.

7.3 System on Chip

Because the Display-U is basically a glorified wireless blink sketch, we did not focus on processing or program space. Most basic 32-bit (or even 8-bit) RF-SoC's (Radio Frequency System on Chips) would easily be up for the task. Instead, we decided focusing on these 3 pillars: Availability of documentation & Software Development tools, power consumption and price.

7.3.1 ESP32

A common option for connected embedded devices the ESP32. This is a BL(E) & WiFi-SoC released in 2016 by Espressif Systems [9]. This chip has many variations [10] and makes it a very flexible solution for many wireless systems.
We will look at 2 variations: The original (but updated) ESP32-D0WD-V3 and the ESP32-H2-MINI-1-N4

Parameter	ESP32-D0WD-V3	ESP32-H2-MINI-1-N4
Processor	Single 32-bit RISC-V 80-240 MHz	Single 32-bit RISC-V 96 MHz SC
SRAM	520 KB	320 KB SRAM
Flash	4 MB	4 MB
RF	Bluetooth 4.2 (LE) & 802.11b/g/n	Bluetooth 5 (LE) & 802.15.4
Price	1,69 EUR (6000 pcs+)	2,03 EUR (650 pcs+)

Table 9: Espressif Comparison Table

The original ESP32 has always been known to have a quite high power consumption.

ESP32-D0WD-V3 [11]	Power Consumption	Unit
Active - Transmit BT/BLE, POUT = 0 dBm	130	mA
Active - Receive BT/BLE	95 ∼100	mA
Modem Sleep (80 MHz)	20 ∼25	mA
Light Sleep	0.8	mA
Deep Sleep (RTC Timer + RTC Memory)	10	μA

Table 10: ESP32-D0WD Power Consumption Table

Newer lower clocked ESP32's improves on this:

ESP32-H2-MINI-1-N4 [12]	Power Consumption	Unit
Active - Transmit BT/BLE, POUT = 0 dBm	38	mA
Active - Receive BT/BLE	25	mA
Modem Sleep (80 MHz)	18 - 23	mA
Light Sleep	0.075	mA
Deep Sleep (RTC Timer + RTC Memory)	5	μA

Table 11: ESP32-H2 Power Consumption Table

It is important to note that ESP32 are heavily marketed for IoT applications. The Wi-Fi functionality they provide is of no interest to the Display-U.

[9] Espressif'Announces the Launch of ESP32 Cloud on Chip and Funding by Fosun Group, Espressif (2016)
[10] Espressif' 2023 ESP32 Product Lineup, Espressif (2023)

In 2023, Espressif's chips are well documented. Because they're widely used in the "makers-space" most documentation and examples are provided by actual users using the chips on the countless available ESP32-based development boards.
Programming can be done via USB. Typically, using the Arduino IDE. Other IDEs are also supported. PlatformIO is a popular alternative. MicroPython another option although much less widely used.

7.3.2 Nordic nRFxxxx

Nordic is a fabless chip producer specializing in wireless capable SoC's.

Their nRFxxx product line specializes in single core SoC with BLE connectivity. Additionally, they're often configurable to use ANT+. This means that these SoC's can do both.
Notably a Nordic nRF5xxx chip was introduced in the second iteration of the then 11-year-old Arduino nano in 2019. The Arduino Nano BLE 33, this pin equivalent iteration added an inertial measurement unit, gyroscope, and a magnetometer. It was marketed for use in robotics, digital compasses, and exercise trackers.

Figure 43: Arduino Nano BLE 33

Garmin bike products have been known to implement Nordic Chips [13] . More specifically Nordic's nRF8xxx & nRF24 Series are hardware RF controllers that are widely used to add RF functionality to microcontrollers.
These partnerships show the capabilities of these products. Making them a serious consideration for our project.

In the table below the full line-up of Nordics BLE-SoC's (nRF5xxxx) is shown.

[13] Teardown: The Garmin Forerunner 220 sport watch and heart monitor, Microcontrollertip.com (2017)

SoC	nRF5340	nRF52840	nRF52833	nRF52832	nRF52820	nRF52811	nRF52810	nRF52805
Bluetooth	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3
Thread	Yes	Yes	Yes		Yes	Yes		
Matter	Yes	Yes						
Zigbee	Yes	Yes	Yes		Yes			
Bluetooth Mesh	Yes	Yes	Yes	Yes	Yes			
Flash	1 MB + 256 KB	1 MB	512 KB	512/256 KB	256 KB	192 KB	192 KB	192 KB
RAM	512 KB + 64 KB	256 KB	128 KB	64/32 KB	32 KB	24 KB	24 KB	24 KB
CPU	128 MHz Arm Cortex-M33 + 64 MHz Arm Cortex-M33	64 MHz Arm Cortex-M4 with FPU	64 MHz Arm Cortex-M4 with FPU	64 MHz Arm Cortex-M4 with FPU	64 MHz Arm Cortex-M4	64 MHz Arm Cortex-M4	64 MHz Arm Cortex-M4	64 MHz Arm Cortex-M4
High-Speed SPI	Yes	Yes	Yes					
QSPI	Yes	Yes						
USB	Yes	Yes	Yes		Yes			
PWM, PDM, I2S	Yes	Yes	Yes	Yes		PWM, PDM	PWM, PDM	
ADC, Comp	Yes	Yes	Yes	Yes	COMP	ADC, COMP	ADC, COMP	ADC
Temp Range	-40 to 105 °C	-40 to 85 °C	-40 to 105 °C	-40 to 85 °C	-40 to 105 °C	-40 to 85 °C	-40 to 85 °C	-40 to 85 °C
Supply voltage range	1.7 to 5.5 V	1.7 to 5.5 V	1.7 to 5.5 V	1.7 to 3.6 V	1.7 to 5.5 V	1.7 to 3.6 V	1.7 to 3.6 V	1.7 to 3.6 V
Packages	aQFN94	aQFN73	aQFN73	QFN48	QFN40	QFN48	QFN48	WLCSP28
	WLCSP95	WLCSP94	QFN40	WLCSP50		QFN32	QFN32	
			WLCSP76			WLCSP33	WLCSP33	
GPIOs	48	48	18-42	32	18	15-32	15-32	10

Figure 44: nRF5xxxx Line Up

From this table the nRF52820 was chosen. It was the most affordable option from the line-up implementing an integrated USB controller usable for programming. We could've added a separate USB-controller combined with a cheaper Nordic BLE-SoC. However, this additional hardware controller would use space on the board and add to the total power draw.
The nRF5820 uses an easy to solder QFN footprint.

Figure 45: nRF5820 - QFN40

Nordic specialises in these RF-SoCs, and this is clearly viewable in its power draw:

Nordic nRF52820 [14]	Power Consumption	Unit
Active - Transmit BT/BLE, POUT = 0 dBm	14.1	mA
Active - Receive BT/BLE	7.2	mA
Active - CPU Only	2.1	mA
Deep Sleep (wake by reset)	3	μA

Table 12: Nordic nRF52820 Power Consumption Table

Compared to the ESP32 option we're drastically consuming less. Making it a better option for our battery powered Display-U.

Nordic provides a Software Development kit. This includes several tools like Nordic's own programmer, libraries and APIs, examples and templates, debugging ant testing tools, etc... This full suit of tools makes the Nordic's SoC's have a strong level of software development support. Using USB 2.0 on the Display-U we're programming the nRF52820 using its native USB hardware controller.

7.4 Power

The battery powering Display-U is a 300 mAh 3,7v Lithium Polymer battery. These can come in over 100 different dimensions [15]. The enclosure of the Display-U could easily implement any of them. This means that we could fit a larger cell if needed.

This is not a 100% fixed part. The capacity value we've picked gives us a starting point to calculate the autonomy of the device.
This battery cell is connected to the PCB using a common JST connecter typically used in drones or other small battery powered products. If the user feels comfortable swapping a battery, it would be doable.

To protect & charge this battery we're implementing the LTC4001 IC. It is specifically designed for a single cell battery thus it is pre-programmed (in hardware) with the corresponding voltages. Additionally, we've connected status LEDs for charge status and possible battery faults. The battery is charged using a USB-C port. This is limited to 3A. There is no implemented USB-Power-Deliver.
To power our SoC, we've chosen a common linear-dropout voltage regulator.

[15] 300mAh Battery specifications lipobattery.com (2023)

7.5 Estimated Battery Life

7.5.1 LDO Power Draw

$$
\begin{aligned}
Power\ Loss &= (V_{in} - V_{out}) \times I_{out} \\
V_{in} &= 4.2V\ (worst\ case) \\
V_{out} &= 3.3V \\
I_{out} &= 0.02A \\
&\Longrightarrow \\
Power\ Loss &= (4.2V - 3.3V) \times 0.02A \\
&= 0.9V \times 0.02A \\
&= 0.018W = 18mW
\end{aligned}
$$

Efficiency of 78.57 % and Quiescent Current of: $I_q = 0.055mA$

7.5.2 RF-SoC Power Draw

Symbol	Description	Min.	Typ.	Max.	Units
I_{S0}	CPU running CoreMark from flash, Radio transmitting @ 0 dBm output power, 1 Mbps *Bluetooth*® Low Energy (BLE) mode, Clock = HFXO, Regulator = DC/DC		7.3		mA
I_{S1}	CPU running CoreMark from flash, Radio receiving @ 1 Mbps BLE mode, Clock = HFXO, Regulator = DC/DC		7.2		mA
I_{S2}	CPU running CoreMark from flash, Radio transmitting @ 0 dBm output power, 1 Mbps BLE mode, Clock = HFXO		14.1		mA
I_{S3}	CPU running CoreMark from flash, Radio receiving @ 1 Mbps BLE mode, Clock = HFXO		13.7		mA

Figure 46: nRF52820 Compounded Power Consumption

Symbol	Description	Min.	Typ.	Max.	Units
$I_{ON\_RAMOFF\_EVENT}$	System ON, no RAM retention, wake on any event		0.4		µA
$I_{ON\_RAMON\_EVENT}$	System ON, full 32 kB RAM retention, wake on any event		0.6		µA
$I_{ON\_RAMON\_POF}$	System ON, full 32 kB RAM retention, wake on any event, power-fail comparator enabled		0.8		µA
$I_{ON\_RAMON\_GPIOTE}$	System ON, full 32 kB RAM retention, wake on GPIOTE input (event mode)		2.5		µA
$I_{ON\_RAMON\_GPIOTEPORT}$	System ON, full 32 kB RAM retention, wake on GPIOTE PORT event		0.6		µA
$I_{ON\_RAMOFF\_RTC}$	System ON, no RAM retention, wake on RTC (running from LFRC clock)		1.2		µA
$I_{ON\_RAMON\_RTC}$	System ON, full 32 kB RAM retention, wake on RTC (running from LFRC clock)		1.4		µA
$I_{OFF\_RAMOFF\_RESET}$	System OFF, no RAM retention, wake on reset		0.3		µA
$I_{OFF\_RAMON\_RESET}$	System OFF, full 32 kB RAM retention, wake on reset		0.5		µA

Figure 47: nRF52820 Sleep Modes

The RF-SoC is powered by a 78.57% efficient LDO. The effective currents must be adjusted:

Part	MPN	Sleep	Idle	Peak	Unit
SoC	nRF52820	0,003	7,2	7,2	mA
SoC (effective)	nRF52820	0,004	9,16	9,16	mA

Table 13: Adjusted Effective Currents

7.5.3 Display-U Modes

- Sleep: nRF built-in sleep mode (wake by reset)
- Idle: Continuously receiving data from SenseU
- Peak: Continuously receiving data while illuminating 1 LED

Part	MPN	Sleep	Idle	Peak	Unit
SoC	nRF52820	0,004	9,16	9,16	mA
Q1	FDS9926A	0,001	0,001	0	mA
LED	XPEBRD-L1-0000-00901	0	0	89,362	mA
Voltage Regulator	AP2112K-3.3TRG1	0,055	0,055	0,088	mA
Total		0,060	9,216	98,61	mA

Table 14: Current Consumption of Display-U Modes

The device enters sleep mode when by receiving a command sent by the Sense Unit. In this mode, the device barely uses any power:

$$Battery\ life\ (sleep) = 300\ mAh/0.06\ mA = 5000\ hours = 208.33\ days$$

The user can take breaks between usage while the device is put to sleep mode and sips power. In contrast with the Sense-U that must be manually turned off using a switch.
To exit the sleep mode the user must press a button on the device. This button resets the nRF, thus waking the device. This procedure was chosen as a "wake by reset" draws the least amount of current. Additionally, when the cyclist exit's his garage he is not interested in the state of the Display-U. The Display-U will boot up in a few seconds while the user starts his commute.

The real battery life is heavily depended on the total time the LEDs will be blinking. In use, the only 1 led will blink at a certain frequency while a car is approaching. This reduces the battery life while still sufficiently alerting the cyclist. Because of the conceptual nature of this product. We will have to assume that, during a cyclist's commute, 1 LED is continuously on for (on average) 10% of the total time.

$$\begin{aligned} Average\ Current\ Draw &= (90\%\ of\ 9.216\ mA) + (10\%\ of\ 98.61\ mA) \\ &= 8.294\ mA + 9.861\ mA \\ &= 8304.26\ mA \\ Battery\ life &= Battery\ capacity\ /\ Average\ current \\ &= 300\ mAh/18.155\ mA \\ &= 16.505\ hours \end{aligned}$$

Assume a 2 hour total daily commute resulting in a workweek with 10 hours commuting. This would be sufficient.
Several assumptions are being made for this calculation. The practical battery life will be significantly impacted by the number of cars approaching the cyclist during the commute., which directly impacts the LED's current draw. The battery is easily replaceable and could be swapped out for a larger one without any other design changes.

7.6 Cost Breakdown

Unit price for an order of 1000pcs, a combination of Digikey and Mouser was used to gather the prices. Batteries are bought from a Chinese wholesaler. These types of batteries are rarely stocked on the common electronics wholesalers. Prices for a 1s Lithium Polymer battery with a capacity of 300mAh range from 0,55 EUR to 2.7 EUR [16]. This was rounded to 1 EUR to simplify calculations.

[16] Battery price reference. dtpbattery (2023)

Microcontroller	nRF52820-CFAA-D-R7	2,04	1	2,04
LEDS	XPEBRD-L1-0000-00901	1,28	2	2,56
Power N-Ch	FDS9926A	0,291	1	0,291
High Power 47ohm R	CHP1206AFX-47R0ELF	0,11386	2	0,22772
USB C Port	USB4105-GF-A	0,383	1	0,383
LDO	AP2112K-3.3TRG1	0,10075	1	0,10075
Charger	LTC4001EUF#TRPBF	3,69155	1	3,69155
Common P-Ch	AO3421E	0,09326	1	0,09326
Battery Connection	S2B-PH-SM4-TB(LF)(SN)	0,27237	1	0,27237
Status LED	LTST-C191KGKT	0,0439	3	0,1317
0603 Passives	Estimate	0,0004	46	0,0184
Battery	Chinese 3.7v 300mAh	1	1	1
PCB Assembly	JLC PCB (Estimate)	9,9522	1	9,9522
PCB Manufacturing	JLC PCB (with shipping)	0,15	1	0,15
Total				20,91195

Table 15: Display-U Cost Breakdown

8 Annex

8.1 Github

Github Repository containing all the VHDL, Matlab and KiCad files.

8.2 Block Diagram

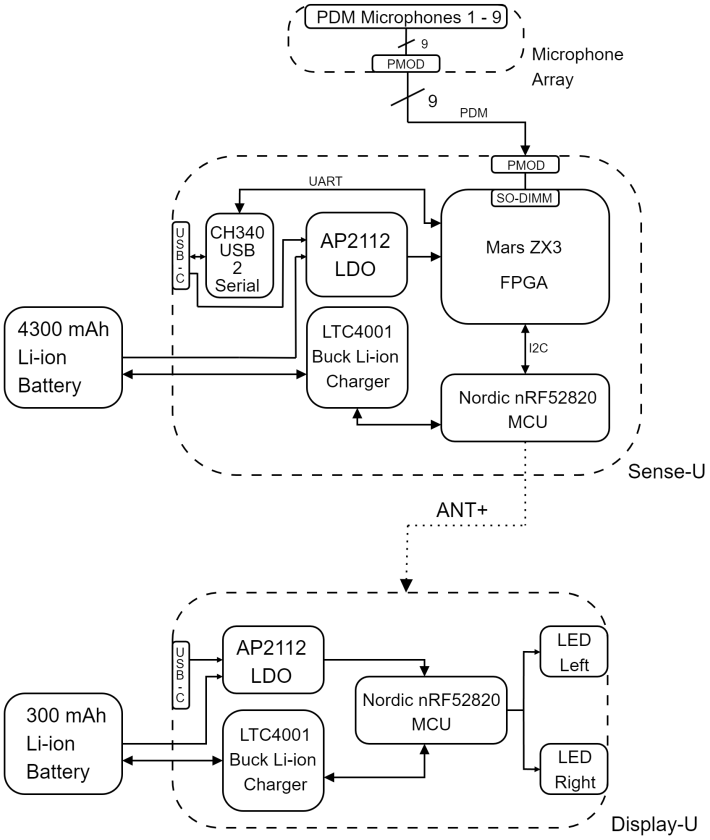


Figure 48: Block Diagram Bike Acoustic Danger Proximity Sensing Box