Table of Contents

1 Problem description

1.1 Overall Problem Statement

The main objective of this project is to achieve precise synchronization among multiple audio recording devices, such as AudioMoth[1] receivers, placed in a defined area. Synchronization is important to ensure audio data from different devices can be aligned accurately during later analysis and processing. Accurate time synchronization across multiple audio recording devices is essential for applications such as environmental monitoring, wildlife tracking, and distributed acoustic sensing. Traditional synchronization methods often rely on GPS or network-based protocols, which may not be feasible in remote or infrastructure-limited environments.

To solve this problem, we propose creating a specialized synchronization signal that acts as a timing reference. Ideally, this signal would be sent via radio waves to all audio recorders in the network. Once received, the synchronization signal would be mixed with the main audio captured by each receiver, embedding a clear timing reference into the recordings. During post-processing, this embedded synchronization signal makes it possible to align audio recordings from multiple devices and potentially extract additional metadata for better analysis.

At this stage of the project, our primary focus has been on researching and implementing analog watermarking techniques to embed synchronization information directly into audio signals, eliminating the need for modifications to existing recording devices. This involved generating a specialized synchronization signal, integrating it with the primary audio, and developing methods for its extraction during post-processing. To streamline the current development phase, we have deliberately deferred the implementation of radio-wave transmission. Achieving synchronization via radio transmission remains a significant objective for future advancements of the project.

1.2 Specifications

- Every watermark should be easily discernible and have a unique sequence.
- The embedded metadata and synchronization signals must be easily extractable during post-processing for precise alignment of audio data from different devices.
- Synchronization should be accurate to the nearest 1 ms.
- The system must be fully standalone, operating independently without reliance on external sources or networks.
- The system should not require any modification to the recorder it is connected to.
- The system should include the necessary components to mix the synchronization signal with the primary audio signal, controlling their relative amplitude and the full-scale amplitude of the mix.
- The watermarking process should introduce minimal distortion to the primary audio signal, ensuring high fidelity of the recorded data

2 Solutions

2.1 Concept and Approach

Initially, research began with a thorough review of existing audio watermarking solutions, including an examination of resources available on IEEE websites and studying the textbook "Audio Watermark - A Comprehensive Foundation Using MATLAB"[2]. Most current watermarking techniques rely heavily on digital processing methods that require complete access to the audio recordings. However, such approaches are unsuitable for the application, as synchronization sequences must be embedded into audio recordings in real-time during device operation.

Using the textbook methods as a starting point, several approaches were developed tailored specifically to the needs. The first approach considered was embedding synchronization signals using frequency chirps at imperceptible ranges, either below 20Hz or above 20kHz. However, lower-frequency signals tend to become obscured by background noise, while higher-frequency signals require significantly higher sampling rates, thus increasing data storage requirements for recorders typically designed to operate continuously over extended periods (days to weeks).

Ultimately, a pseudo-random noise burst/chirp sequence was selected as the final approach. This method effectively addresses the sampling rate variability issue because noise spreads across a broad frequency spectrum. Additionally, noise bursts introduce minimal distortion, preserving the fidelity of the original audio recordings. Through comparative research on various noise types, including white, pink, and brown noise, brown noise turned out to be the optimal choice. It exhibits a frequency spectrum that declines by approximately 20dB per decade[3], further reducing sampling rate sensitivity and minimizing perceptibility to the human ear, as lower frequencies are generally less noticeable than higher frequencies.

Synchronization is achieved by embedding predefined noise bursts into each audio recording. In post-processing, a correlation analysis is performed between the recorded audio signals mixed with different noise sequences. This correlation process is repeated with multiple noise sequences until a distinctive correlation peak emerges consistently across different recordings, confirming synchronization alignment.

To achieve all this a noise burst generator, mixing circuit and postprocessing software was developed.

2.2 Design

In the design, the heart of the synchronization system is an STM32G431KB Nucleo board[4] running a custom noise burst generator. This microcontroller implements a Brownian (pseudo-random) noise algorithm, drawing on the LFSR-based generator that feeds the resulting waveform out of its DAC (Digital to analog converter). From the STM32, the analog sync signal enters a shared channel (initially a cable, but ultimately intended to be radio transmission) that distributes it to each recording node. At each node, the sync line and the device's primary audio source (either an external player or a microphone) are brought into a small analog mixing circuit. Here, relative amplitudes of the sync burst and the audio signal are adjusted with trimmers to embed the timing watermark without perceptibly degrading audio quality. The mixed signal then passes directly into a standard audio recorder (for example, AudioMoth devices) with no modifications to the recorder itself.

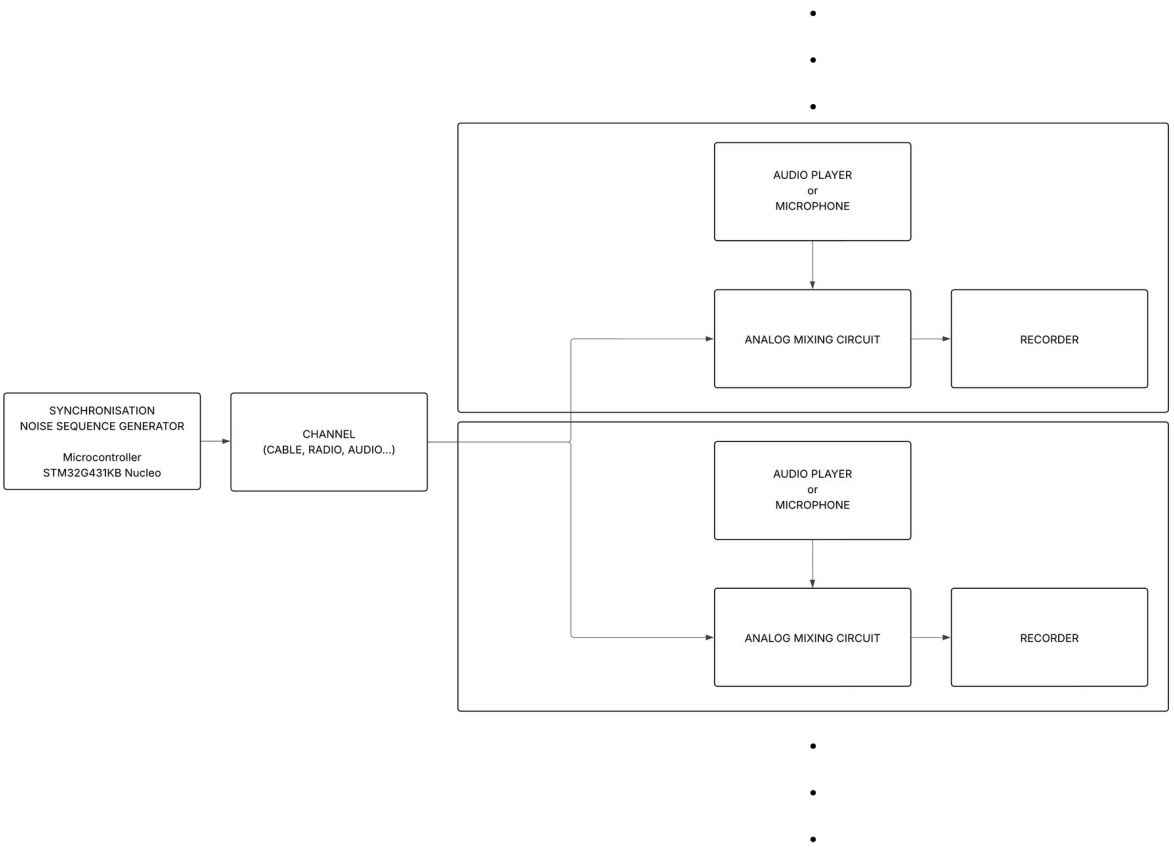


Figure 1: Overview of the whole system.

2.2.1 Synchronization Signal Generation

2.2.1.1 Synchronisation Signal Generation - Code

The synchronization signal embedded into the audio is based on a pseudo-random Brownian noise sequence. To generate this sequence, a custom algorithm was implemented using an LFSR (Linear Feedback Shift Register) and integrated noise shaping logic. The purpose of this approach is to produce a low-frequency, smooth, and non-repetitive waveform that is robust in the analog domain and well-suited for correlation-based detection during post-processing.

The generation process begins with the `white_noise_sample()` function, which creates a 12-bit pseudo-random number using XOR-shift operations on a 16-bit Galois LFSR. This produces a sequence of white noise values. The output is masked to retain only the lower 12 bits, ensuring compatibility with the 12-bit DAC used in the STM32 microcontroller.

To avoid generating repeated sequences, the function `generate_new_seed()` is used to update the LFSR seed after each full sequence. It applies a fixed XOR mask (0xB400) to the current seed and ensures that the result never becomes zero, since a zeroed LFSR would lock the output into a static state.

The core of the synchronization waveform generation lies in the function `GenerateSeq(uint16_t *SeqArray)`, which synthesizes a 12-bit Brownian noise signal from white noise samples. Brownian noise, also known as red noise or integrated white noise, can be modeled as the cumulative sum, or discrete integration, of a white noise sequence. In this implementation, each sample of Brownian noise is built step-by-step by integrating small perturbations derived from white noise. A new white noise value is generated using the `white_noise_sample()`. From this white noise value, a “step” is calculated using the expression:

```
int32_t step = (white_noise & 0x7F) – 64;
```

This maps the lower 7 bits of the white noise sample to a small signed step in the range of –64 to +63.

To stabilize the waveform around the midpoint of the DAC range (2048), a weak restoring force is applied using a proportional term:

```
int32_t correction = (2048 - brownian_state) / 256;
```

This term nudges the signal gently back toward center whenever it begins to drift too far in either direction. The correction is small enough to preserve randomness while preventing long-term saturation or clipping.

The total step (noise-based deviation plus correction) is added to the running state variable `brownian_state`:

```
brownian_state += step;
```

This accumulation is equivalent to the integration of white noise, each new value depends on the previous one, with small random deviations. The state is then clipped to the 0–4095 range to ensure compatibility with the 12-bit DAC, and the result is stored in the output array. After generating the entire sequence of 4096 samples, the state is reset to the midpoint (2048), and the LFSR seed is updated to ensure that subsequent sequences are independent.

After the generation of the sequence the array is ready to be transmitted by the DMA(Direct memory access) to the DAC buffer, which happens after TIM2 overflows. After outputting the whole sequence the interrupt routine triggered by the TIM8_UPDATE triggers new generation of the noise sequence with the new seed.

2.2.1.2 Synchronisation Signal Generation - Hardware

Synchronization signals are generated on the STM32G431KB Nucleo board, chosen for its integrated 12-bit DAC and rich peripheral set, all easily configured via STM32CubeIDE [5]. The goal was to output a 4 096-sample noise sequence at 10 kHz, repeating once per second, with precise timing and minimal CPU load.

First the board and all the required peripherals, such as timers, clocks, pins, DMA and DAC were configured. To achieve the correct timing and output frequency of the DAC multiple interconnected timers that trigger the generation of the output were used. This was achieved by following the STM32G431KB reference manual[6] and datasheet [7], and the Nucleo board manual[8] and datasheet[9] .

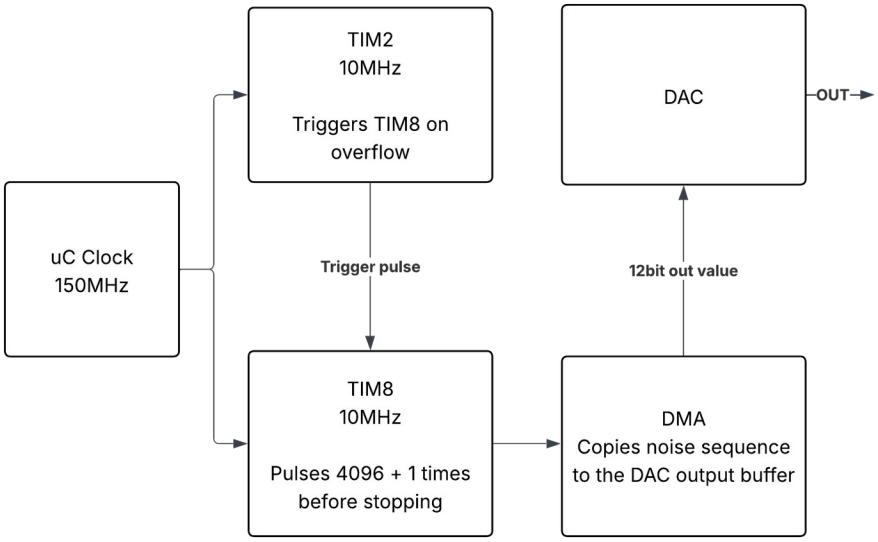


Figure 2: Overview of STM32 peripherals connection

The combination of timers and DMA enables the control and repeatability in generating the noise sequence. TIM2 is 32 bit timer that controls the frequency of the generation of the bursts with triggering the TIM8 at a defined interval. TIM8 is a 16bit timer configured in triggered oneshot capture compare mode with the repetition counter set to the number of samples in the noise sequence. This means that the timer TIM8 is stopped until it receives a trigger pulse from TIM2 (TIM2_UPDATE) which happens every time TIM2 reaches the value of ARR (Automatic reload register) and overflows to the value 0 where it continues counting. After receiving the trigger, TIM8 starts counting to the ARR and CCR (capture compare register), both are set to the same value. On overflow the repetition count is decreased and a TIM8_CR1 (capture compare event) event/trigger is generated. When the repetition down-counter reaches zero, an update event is generated, timer stops – armed for another trigger, and the repetition count is reset to the initial value (4096). This approach creates a train of evenly spaced pulses every time TIM2 reaches overflow value, the frequency of the noise sequence generation can be set. Last 4097^{th} pulse is used to move the value 2048 to the DAC output to have a constant voltage in between bursts.

The TIM8_CR1 events are triggering the DMA module which for each event transfers a 12bit value of the data from the pregenerated noise sequence array in the memory to the output buffer of the DAC, which in turn outputs that value on the GPIO (general purpose input-output) pin.

$$V_o = \frac{3.3V}{4096} * RegVal$$

2.2.2 Signal Mixing

Before designing the mixing circuit, two texts on op-amp applications were consulted, Bruce Carter and Thomas R. Brown's *Handbook of Operational Amplifier Applications[10]* and Walter G. Jung's *Audio IC Op-Amp Applications[11]*, where was discovered that the summing-amplifier topology perfectly matches the requirements. Its fundamental relationship, demonstrates that the output is the inverted, scaled sum of all input voltages. The overall gain of the mixer is set with the feedback resistor, and each input's mixing level is determined by its input resistor.

$$V_o = -R_f\left(\frac{V_1}{R_{in1}} + \frac{V_2}{R_{in2}} + \frac{V_3}{R_{in3}} + \dots\right)$$

In the next figure is the schematic of the mixing circuit the heart of the circuit is MCP6001[12] amplifier used as a buffer, summing amplifier and unity gain inverter. Schematic was drawn in software KiCAD[13] .

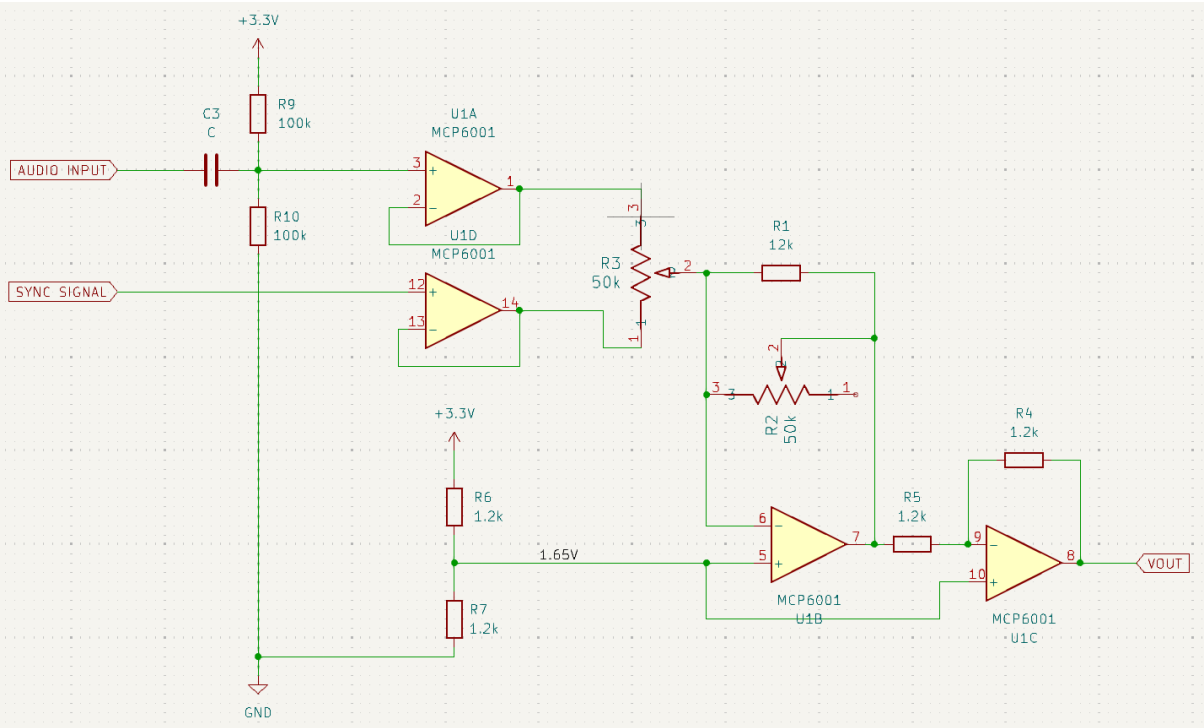


Figure 3: Mixing circuit schematic

The mixing circuit is powered from a single 3.3 V rail and ground, which in this stage of the project is provided by the STM32, but can later be replaced by a battery or the recorder. A 1.65 V virtual-ground node is created by R6/R7 serving as the reference for all op-amp stages. Running entirely from a single 3.3 V rail, the circuit uses a 1.65 V "virtual ground" so that every stage can swing its AC

waveform ±1.65 V around mid-rail without ever hitting 0 V or 3.3 V. This biasing ensures the inverting amplifiers have enough headroom to handle the full dynamic range without clipping. The external audio input is first AC-coupled through C3, then recentered about 1.65 V by the R9/R10 bias network, and finally buffered by U1A (MCP6001) to provide low output impedance and isolate the input stage. In parallel, the STM32G431KB's DAC output which provides the pseudo-random synchronization waveform, is biased internally to the same mid-rail level and buffered by U1D. Next, each buffer output feeds a 50 kΩ potentiometer wired as a two-input voltage divider, one input for audio signal, other for sync signal, so you can precisely set their relative amplitudes by changing the R_{in1} and R_{in2} from the equation with only one potentiometer. The wiper output of the potentiometer is feed into the U1B, which is the summing amplifier with the feedback resistor $R_{f,}$ being another potentiometer, used to set the output gain. Because U1B inverts the combined signal, a final unity-gain inverting buffer (U1C) restores the original polarity. The resulting VOUT, still sitting at 1.65 V DC, can be delivered to the recorder, whose own AC-coupling removes the offset and yields a clean, mixed audio-plus-sync signal. This topology requires only a single 3.3 V supply and ground, provides ±1.65 V of headroom around mid-rail, and imposes no modifications on the recorder itself.

2.2.3 Postprocessing

Once multiple devices have captured their composite audio, our post-processing software upsamples the known noise sequence to the recorder's sample rate, removes DC bias, and performs cross-correlation to detect the embedded bursts in each track and matches the sharp correlation peaks between recordings.

For detection, the function `PulseDetect(filename, NoisePulse)` reads a WAV file, converts both the recorded audio and the upsampled and normalised Brownian sequence to zero-mean floating arrays, and performs a full cross-correlation. The `detect_high_peaks()` then filters the absolute correlation signal, identifies peaks spaced at least 2000 samples apart, and selects only those above 90 % of the maximum height, if more than one are detected then we discard the sequence/alignment because of the overly noisy recordings and ambiguity of the detected noise sequence burst peak.

To align two recordings, `AllignSignals(sig1, sig2, peaks1, peaks2)` computes the sample offset between their detected peaks and circularly shifts one signal accordingly, padding the lead or lag with zeros. As soon as cross-correlation detects matching peaks in both channels, the script aligns the audio and correlation traces, plots the raw noise burst, both audio streams after alignment, and their correlation envelopes.

3 Verification and tests

This section shows the real-world trials of the whole system. It opens with oscilloscope and spectrum captures that verify the STM32-generated Brownian burst's timing, amplitude range, and spectral roll-off. Then it presents how the single-supply summing amplifier blends the burst with audio. The section finishes with multi-recorder synchronisation experiments, first on a dual-channel scope, then on two independent Zoom H5s, where cross-correlation of the embedded burst aligns tracks to sub-millisecond accuracy.

The system was implemented on the breadboard for the ease of debugging, prototyping and modifying. In the next figure the whole system with two mixers and one sequence generator can be seen.

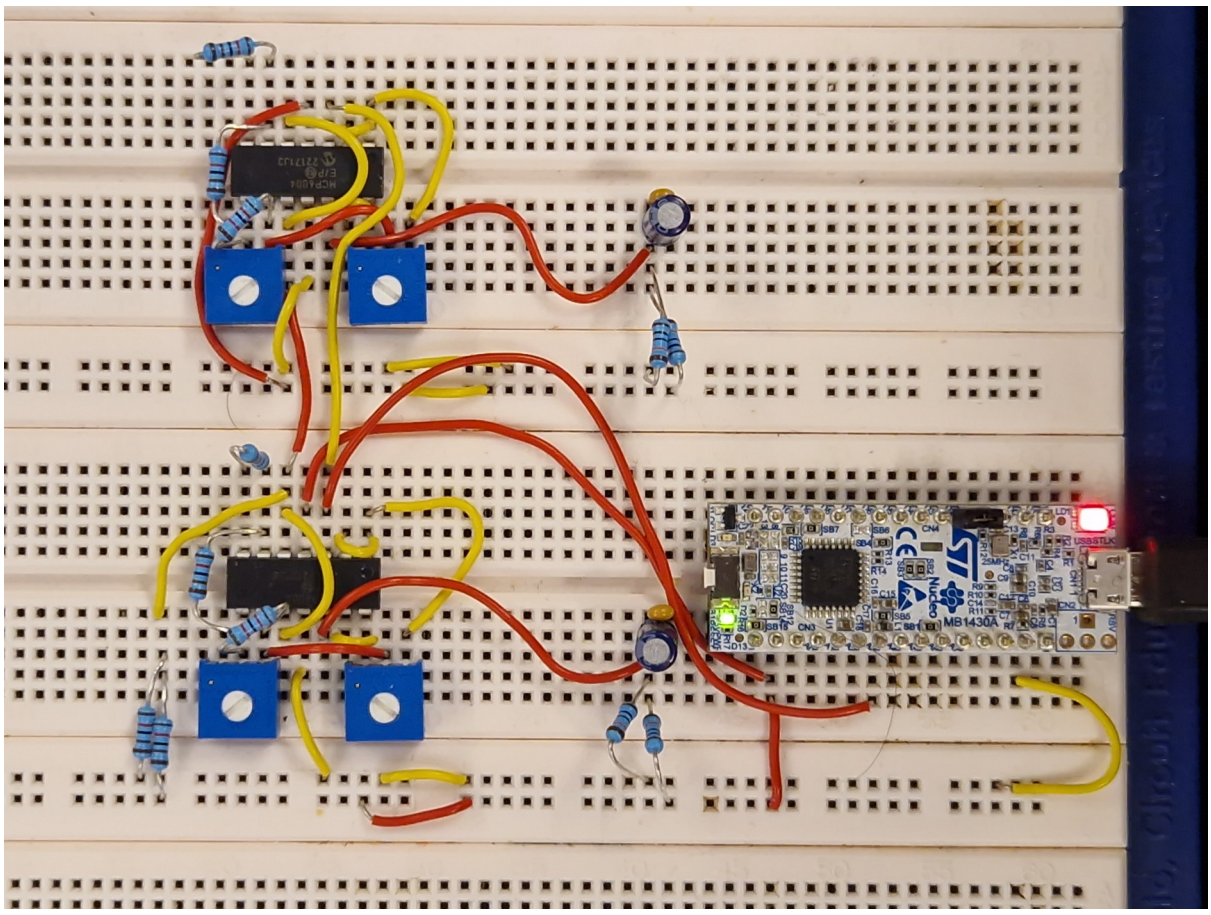


Figure 4: Implemented circuit on the breadboard

3.1 Synchronization Signal Generation

The noise-sequence generator was assembled on a solderless breadboard and powered directly from the Nucleo's 3.3 V rail. Its DAC output pin was routed into a Digilent Analog Discovery 2 oscilloscope[14] . From the moment power is applied, the STM32 begins producing the 4 096-sample noise burst every second.

Figure 5 and Figure 6 summarize the captured output. Figure 5 presents a single noise-sequence burst Figure 6 overlays several back-to-back bursts at a 1 Hz repetition rate, showing the bursts riding on the mid-rail bias and the burst's spectrogram.

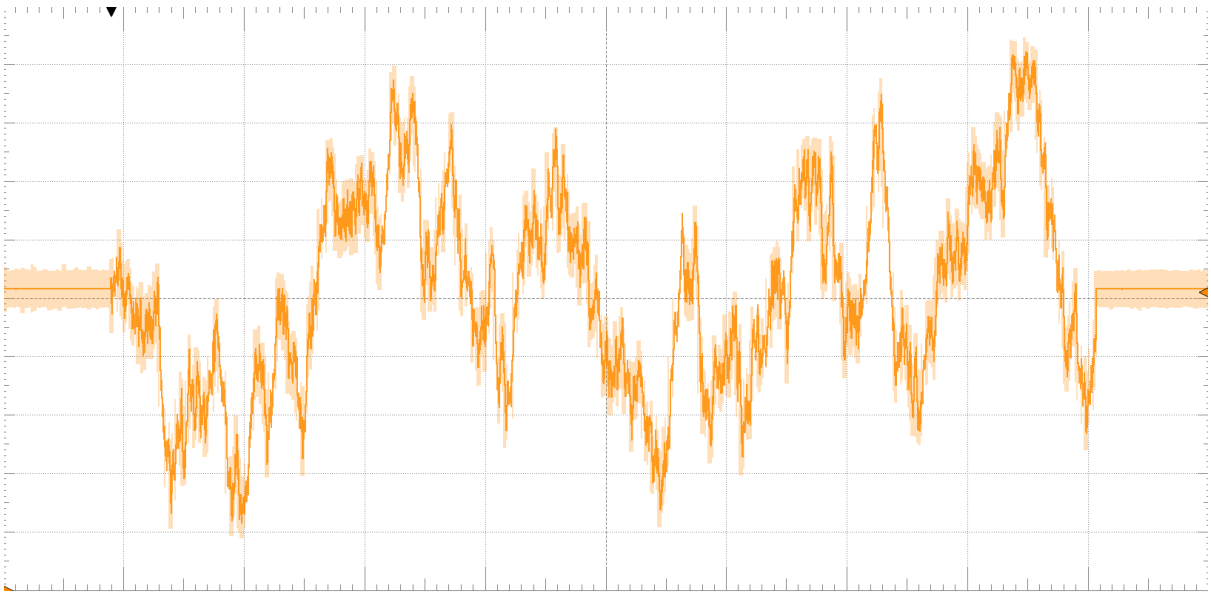


Figure 5: Single measured noise sequence burst

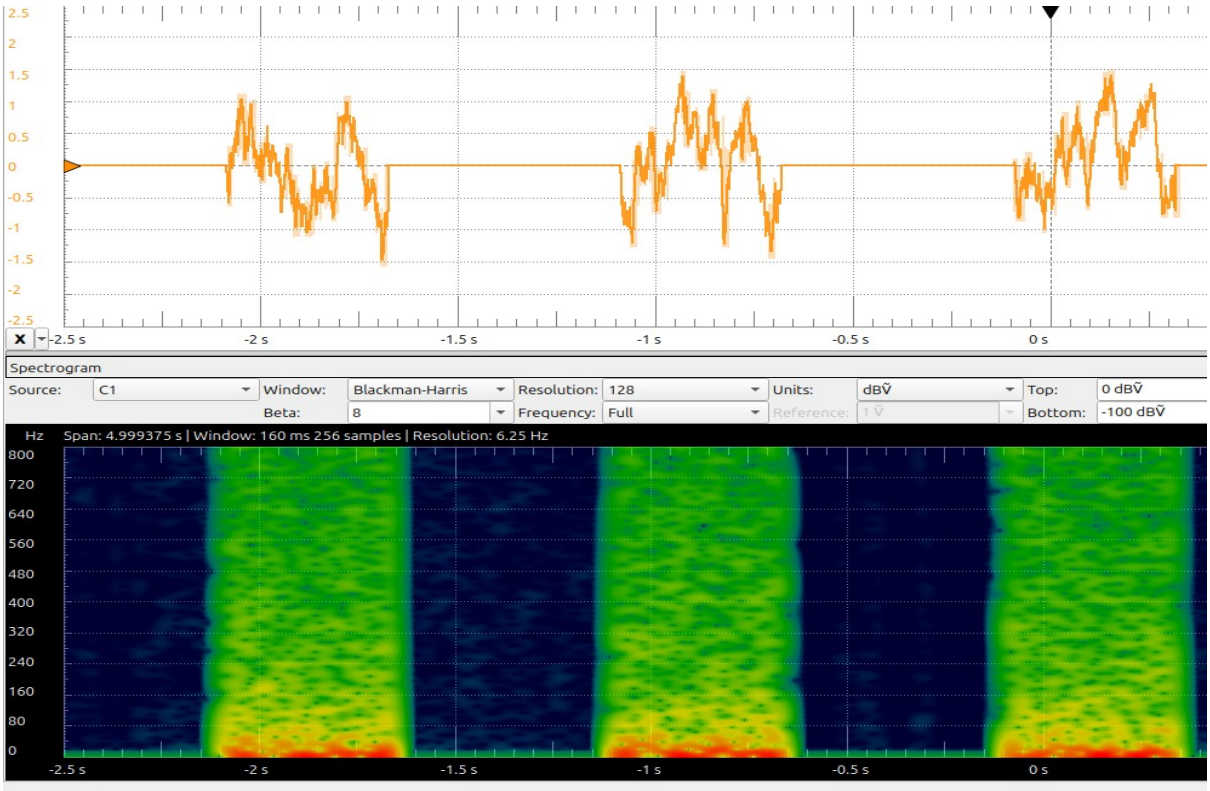


Figure 6: Train of measured noise sequence bursts and spectrogram

This measurements verified the workings of the noise sequence generator.

3.2 Signal mixing

Following the schematics the mixing circuit was built on the breadboard. It was powered by the previously installed STM32 board. One of the inputs was the output of the STM32 board providing the noise sequence every second. The second input was connected to the signal generator of the Digilent analog discovery 2, using this device the measurements, signal generation and in realtime observations were done using the Waveforms software[15].

Figure 7 and Figure 8 shows the output of the mixer when a pure sine tone was combined with the generated Brownian noise sequence. In the time-domain trace (top), the regular sinusoidal oscillation is modulated by the random fluctuations of the noise sequence. In the spectrogram, the Brownian noise sequence appears as a continuous broadband “haze” whose intensity steadily diminishes at higher frequencies.

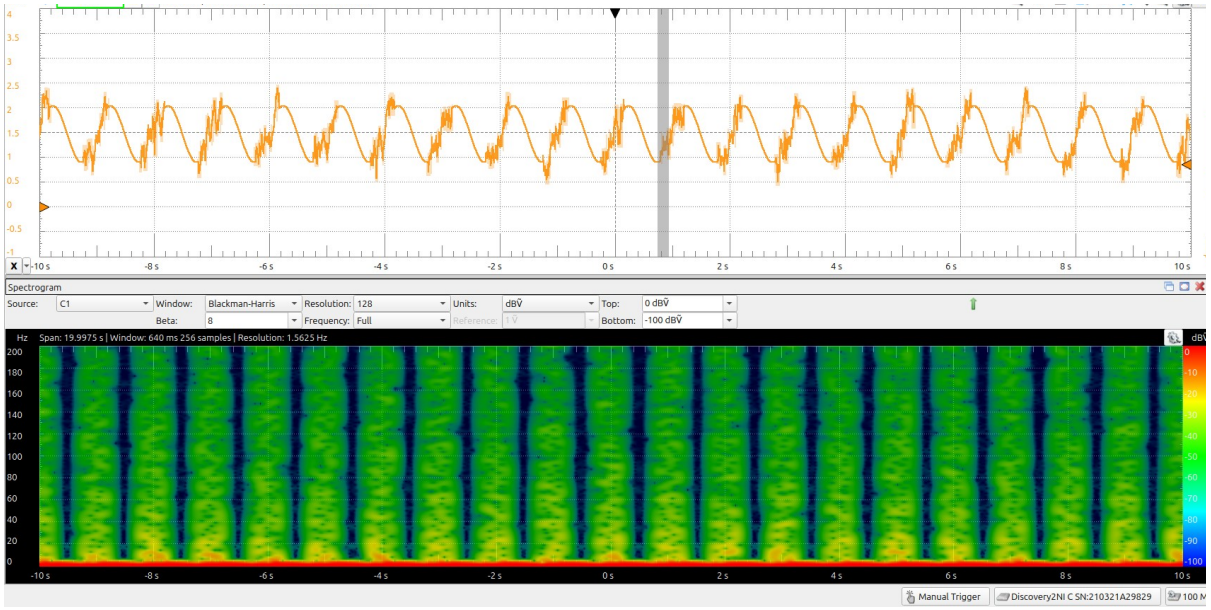


Figure 7: Waveform and spectrogram of the mixed signal, strong noise

After that another test was created, here the noise sequence was deeply decreased/hidden in the signal. The noise burst can be hardly seen with just looking at the waveform but can be clearly seen in the spectrogram.

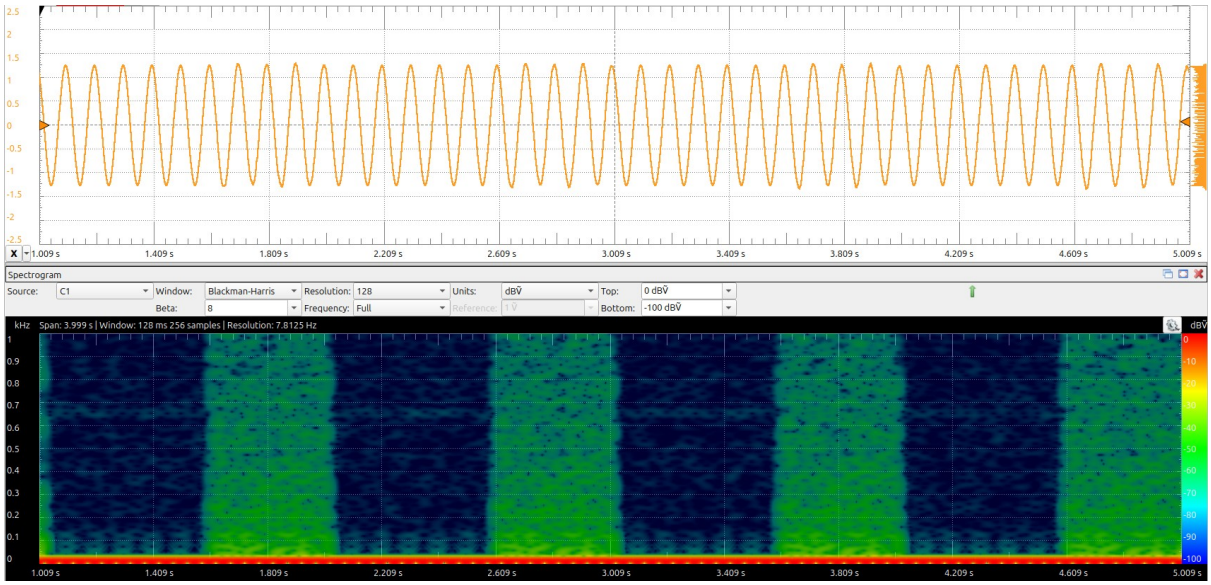


Figure 8: Waveform and spectrogram of a decreased noise signal and a sinusoid

This confirms both that the two signals have been successfully summed.

3.3 Postprocessing and Synchronisation

To validate the post-processing software, two mixer circuits were created and its outputs were recorded in parallel. In early tests, each output fed one of the Analog Discovery 2's two analog inputs; later a standard, unmodified recorder for each channel was used. The resulting WAV files were imported into the post-processing software, which performed correlation-based analysis, computed alignment offsets, and generated synchronized waveform plots for review.

The synchronisation routine works by computing the full cross-correlation between the two sweep waveforms, which effectively slides one signal across the other and measures their similarity at each integer-sample offset, then finding the lag, in samples, that gives the highest correlation. That lag tells you how many samples one sweep must be shifted to best align with the other, and dividing it by the sample rate converts it into a time delay, in seconds. By subtracting each signal's mean beforehand, the alignment depends only on the sweep content rather than any DC offset.

For initial debugging, a static (non-varying) noise burst was embedded to confirm that detection and alignment operated correctly under controlled conditions. In the following experiment, two frequency-sweep tones, generated by a synchronized dual-channel signal source, were each mixed with the Brownian noise sequence and recorded. Frequency sweeps were used to best show the effects of synchronisation. The post-processing software then aligned the two sweeps by one of the detected peaks, and the next measurements demonstrate the achieved synchronization accuracy down to sub-millisecond precision.

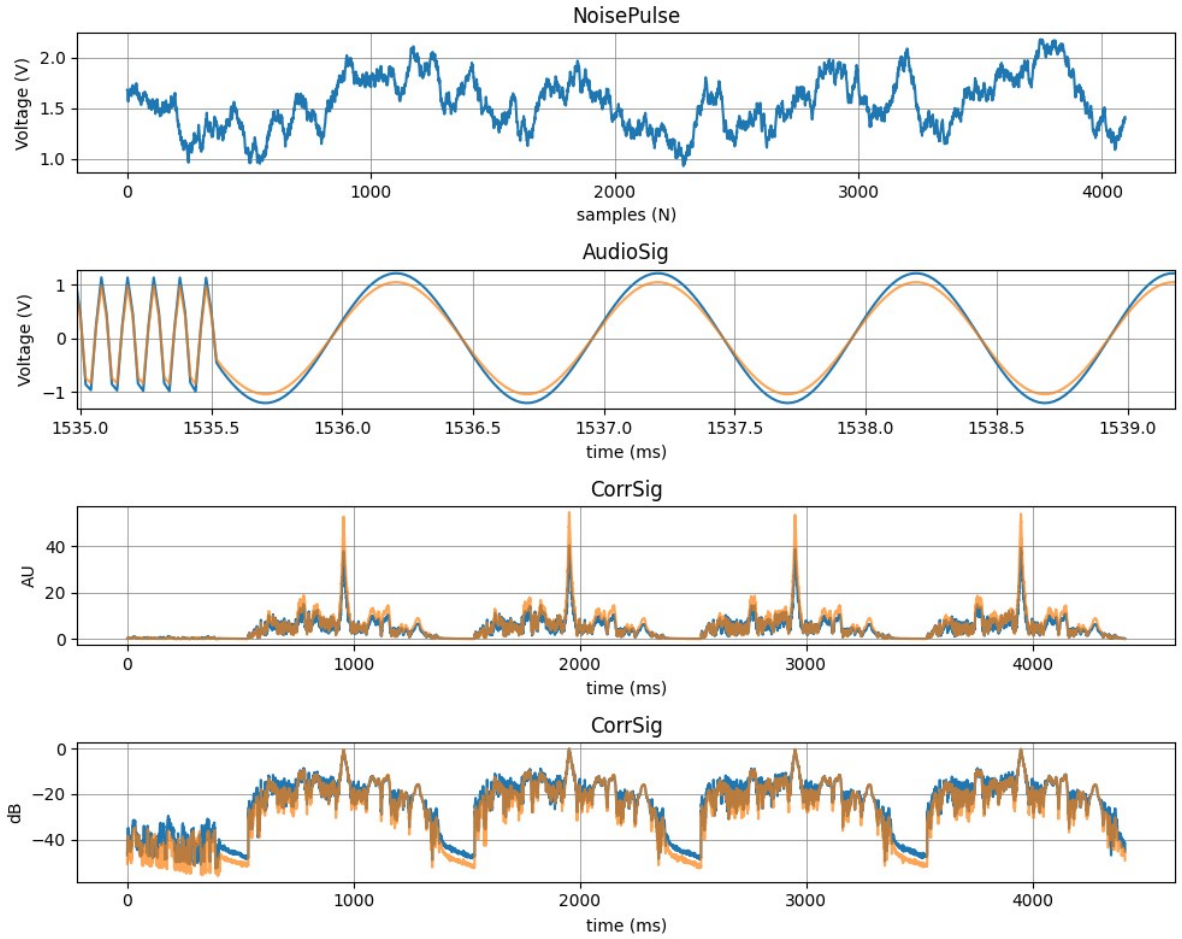


Figure 9: Multiple waveforms of the measurment and processing. Allignment of signals can be clearly seen. AudioSig plot is zoomed into where the sweep restarts, best showing the effects of the allignment.

```
Trying to find the signal N-try:0
Location of peaks from signal 1: [ 47788  97638 147560 197464]
Location of peaks from signal 2: [ 47788  97648 147560 197464]
Delay(ms): 0.0 Lag(samples): 0
Found the allignment in N-tries: 0
```

Figure 10: Output of the processing software. Showing the locations of peaks, delay and lag.

Here the synchronisation was so good that the peaks and signals aligned without any delay/lag. This great efficiency of the system can be attributed to using a single dual channel recorder with synchronised function generator channels, therefore it is an ideal example.

Later a similar measurement and processing has been done, except here the input signals were sweeps with a 180 degree phase shift. A small synchronisation error appeared and it can be seen as a small error in phase, but the signals are still synchronised within 1ms. Correlation peaks are really prominent here, this reflects the real life – while listening to the recordings the noise sequence bursts can be heard.

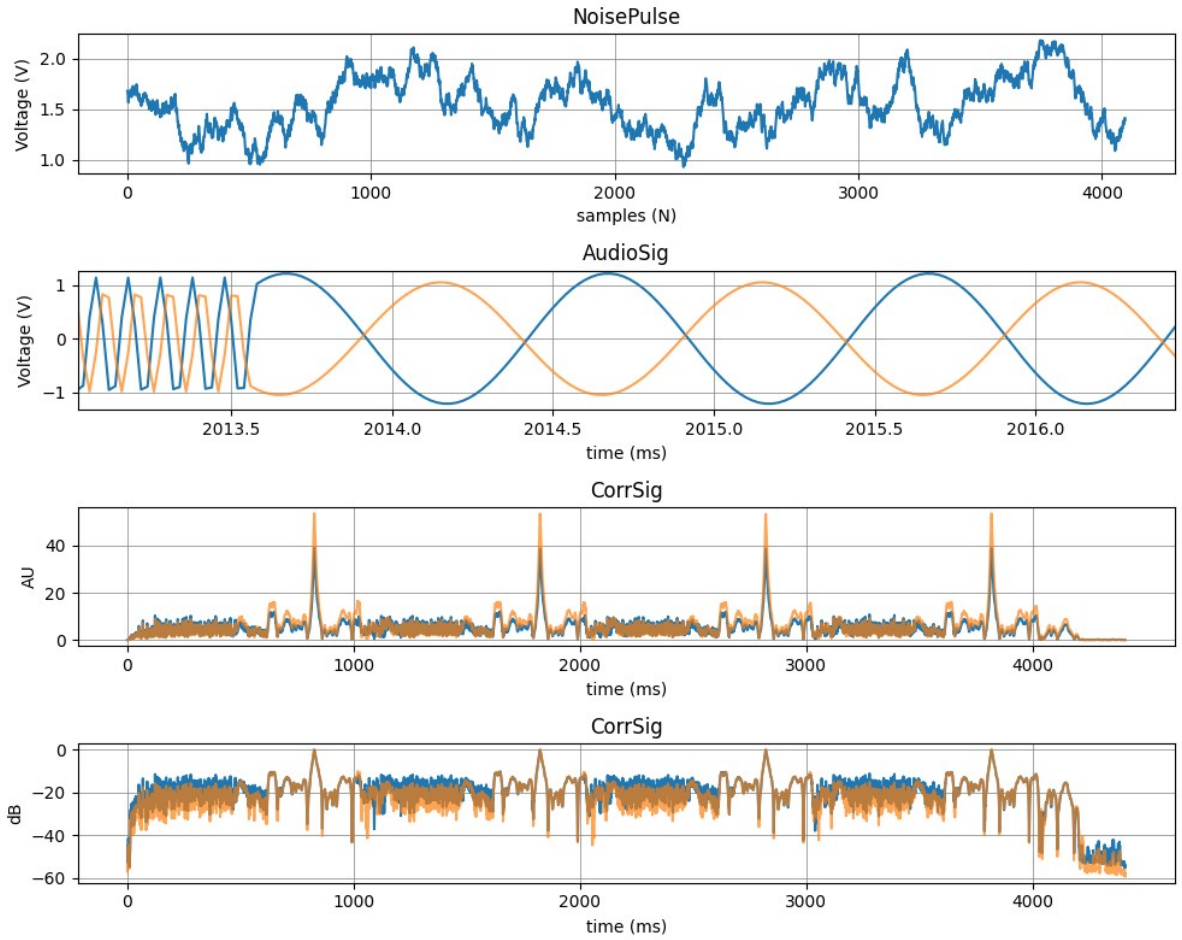


Figure 11: Multiple waveforms of the measurment of a signal with 180 phase shift and processing. AudioSig plot is zoomed into where the sweep restarts, best showing the effects of the allignment. Slight misallignment can be observed in the zero crossings.

Continuing the verification a test of alignment with changing noise sequence bursts was done. Here only one main correlation peak was expected. Correlation peaks of the noise sequence bursts can be barely seen, especially in the blue signal. This reflects real life as the bursts were unperceptible to the ear in this measurements.

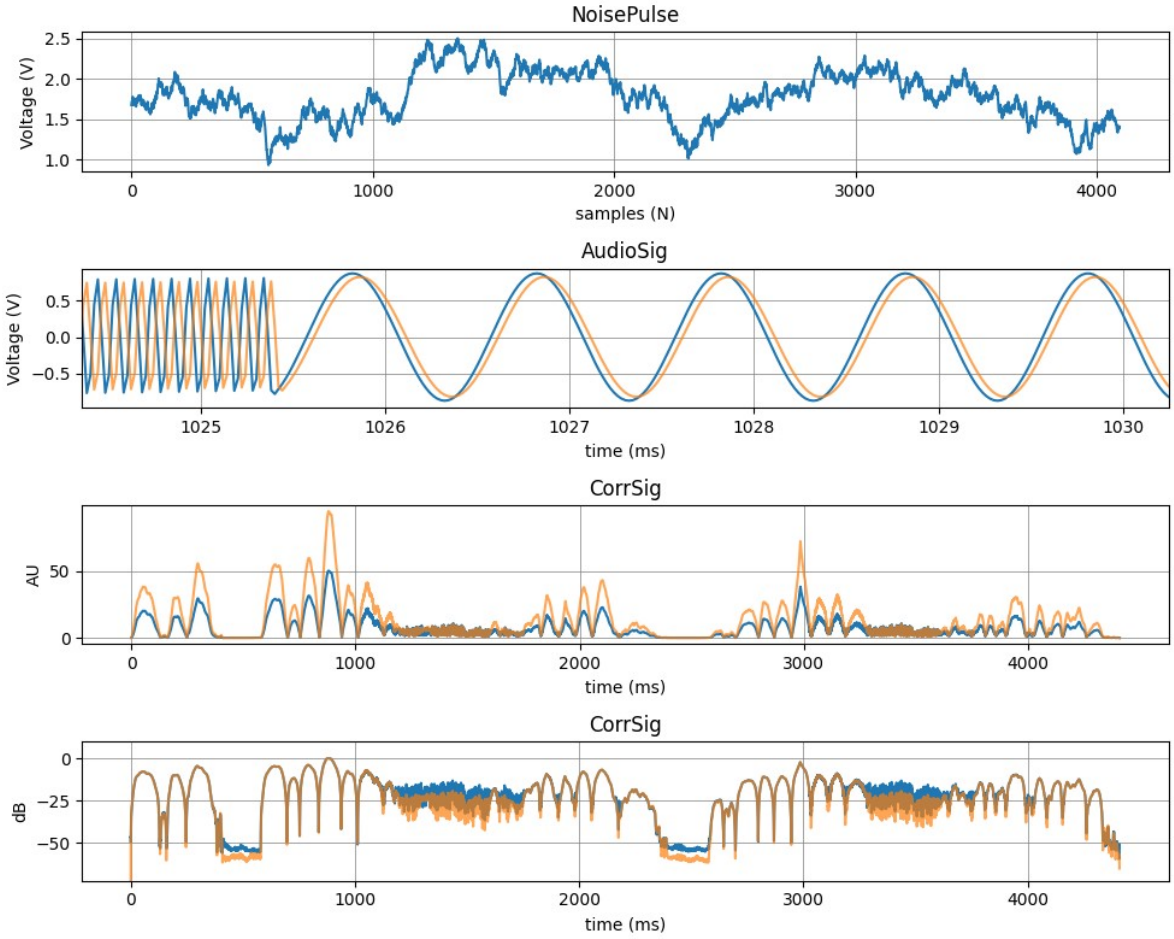


Figure 12: Multiple waveforms of the measurment of a signal with 180 phase shift and processing. AudioSig plot is zoomed into where the sweep restarts, best showing the effects of the allignment. Slight misallignment can be observed.

```
Trying to find the signal N-try:3
Location of peaks from signal 1: [43971]
Location of peaks from signal 2: [43969]
Delay(ms): 0.04 Lag(samples): 2
Found the allignment in N-tries: 3
```

Figure 13: Output of the processing software. Showing the locations the peaks, delay and lag.

Next came the test of the whole system using two separate recorders from company ZOOM model H5 Handy Recorder[16]. In the next Figure 14 you can see the measurement setup. The sweeps were generated by the Digilent discovery 2 and were fed into the mixing circuit. Each output from the circuit was then fed into the recorders.

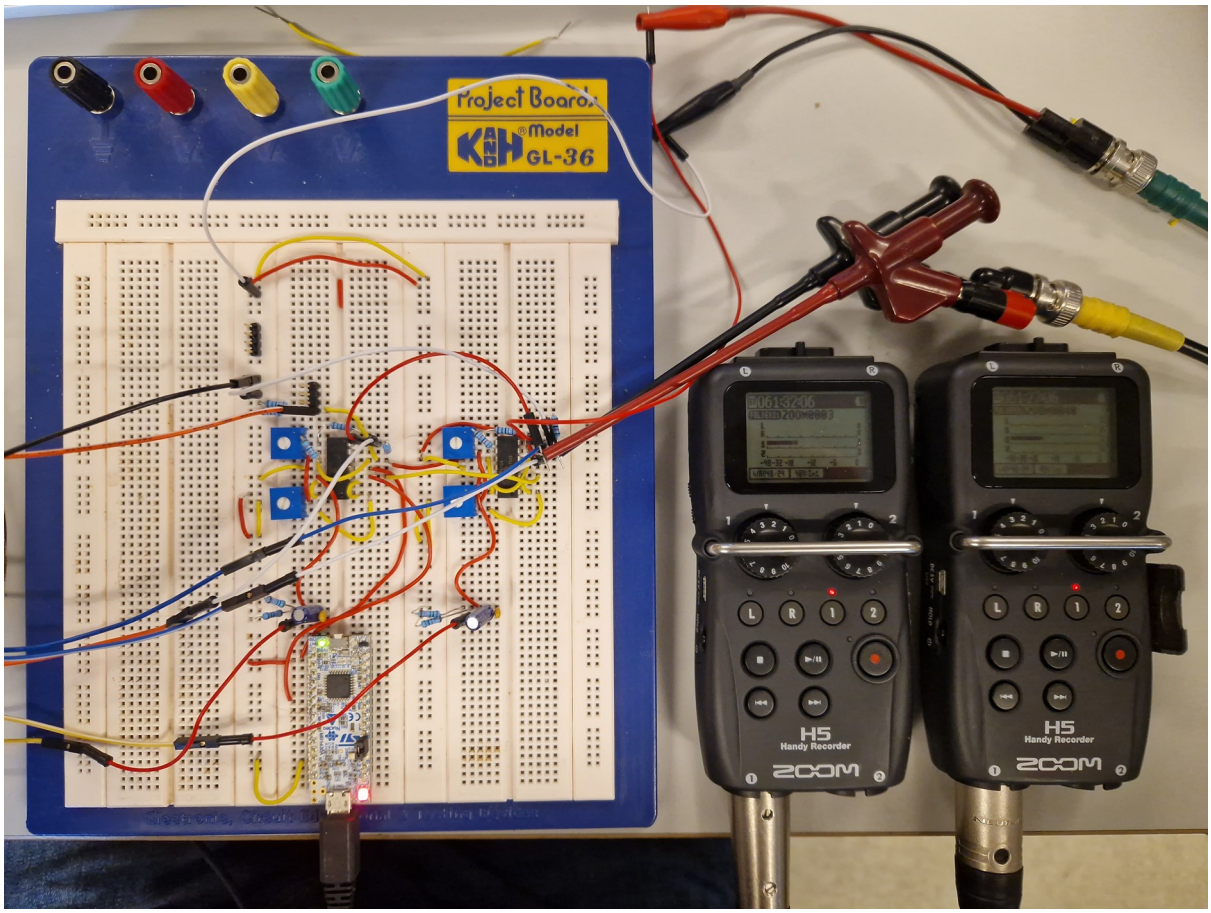


Figure 14: Whole system with recorders configuration

With this setup multiple recordings were done and then processed in the software. In the next figures the results are presented. Synchronisation had a varying accuracy but all of the signals were synchronised to less than 1ms relative to each other.

The first measurement with the recorders best shows the alignment of two differently long measurements. The two audio signals are aligned to sub 1ms.

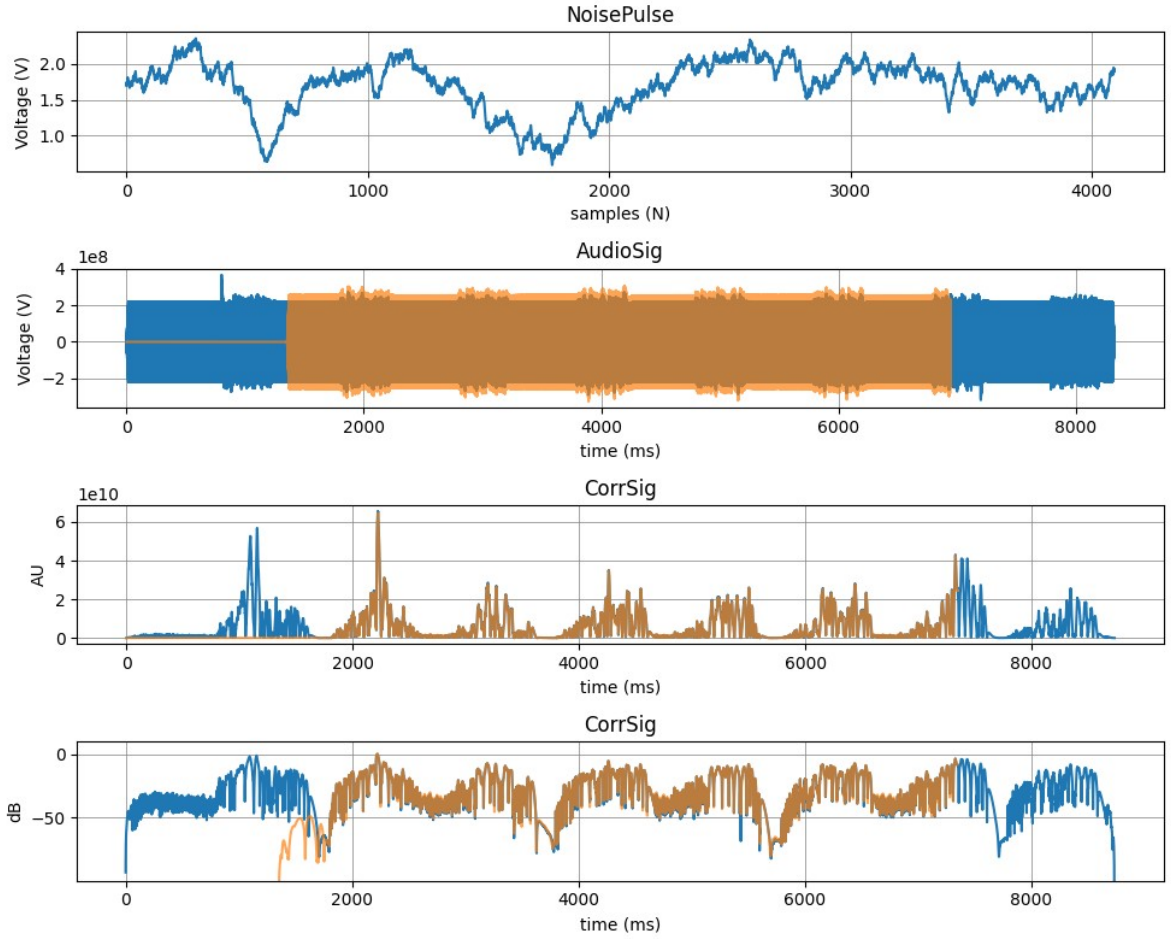


Figure 15: Multiple waveforms of the measurment and allignment of two signals. Differently long signals are cleary alligned.

In the Figure 16 small misalignment can be observed as a phase shift, but it is still below 1ms.

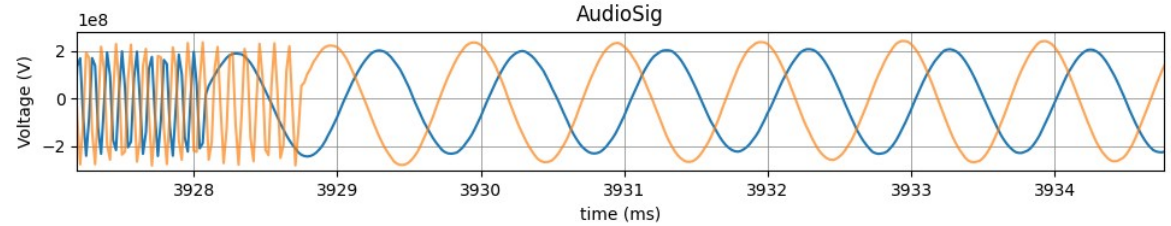


Figure 16: A misallignment can be observed here, but still below 1ms.

The output from the postprocessing confirms the delay being below 1ms and it corresponds to the observation.

```
Trying to find the signal N-try:2
Location of peaks from signal 1: [106745]
Location of peaks from signal 2: [41745]
Delay(ms): 0.64 Lag(samples): 32
Found the allignment in N-tries: 2
```

Figure 17: Output of the postprocessing software, a sub 1ms delay was measured.

Second measurement achieved much better alignment, 0.06 ms or 3 samples.

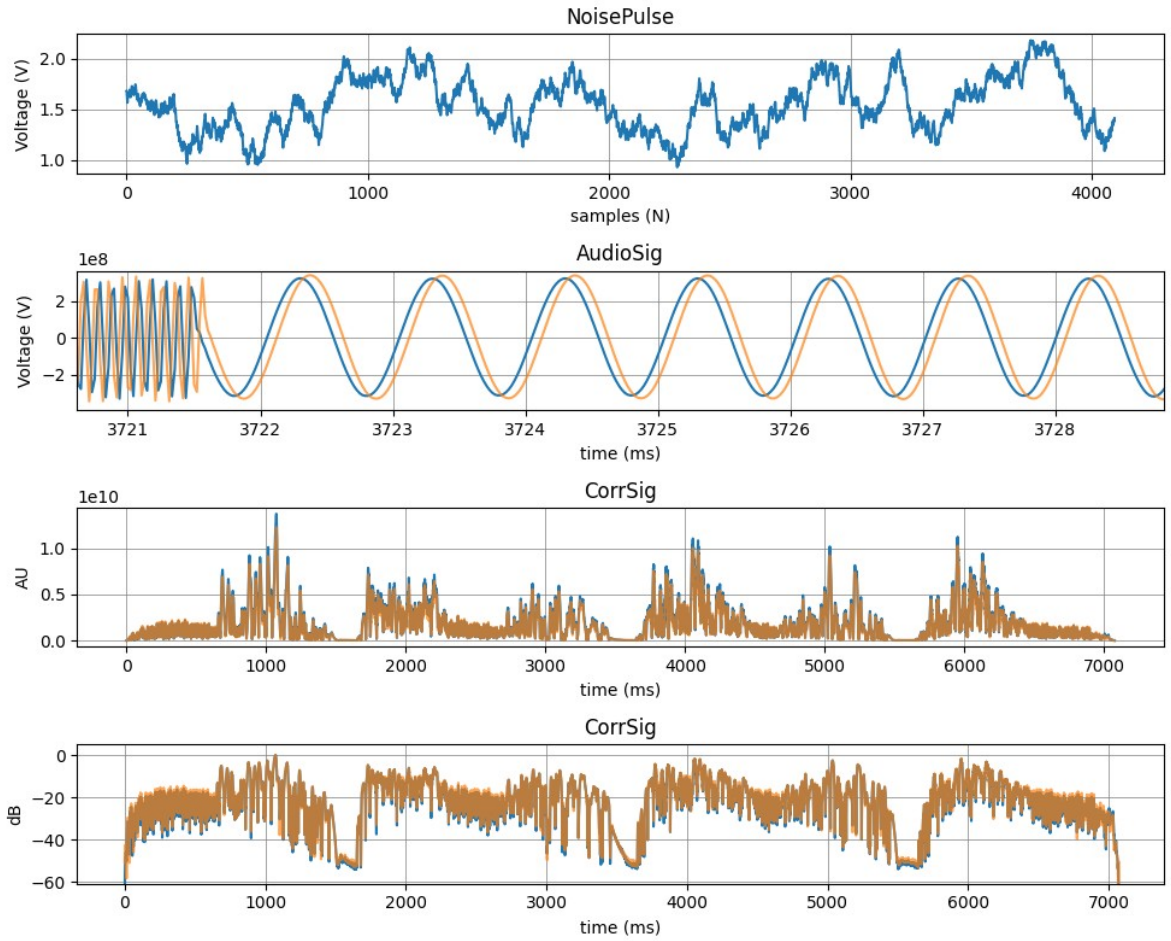


Figure 18: Multiple waveforms of the measurment and allignment of two signals. Small misallignment.

```
##############################################
Trying to find the signal N-try:0
Location of peaks from signal 1: [51508]
Location of peaks from signal 2: [51660]
Delay(ms): 0.060000000000000005 Lag(samples): 3
Found the allignment in N-tries: 0
```

Figure 19: Output of the postprocessing software, a sub 1ms delay was measured.

The third measurement shows one of the best alignments, only 0.02ms delay or one sample of a difference.

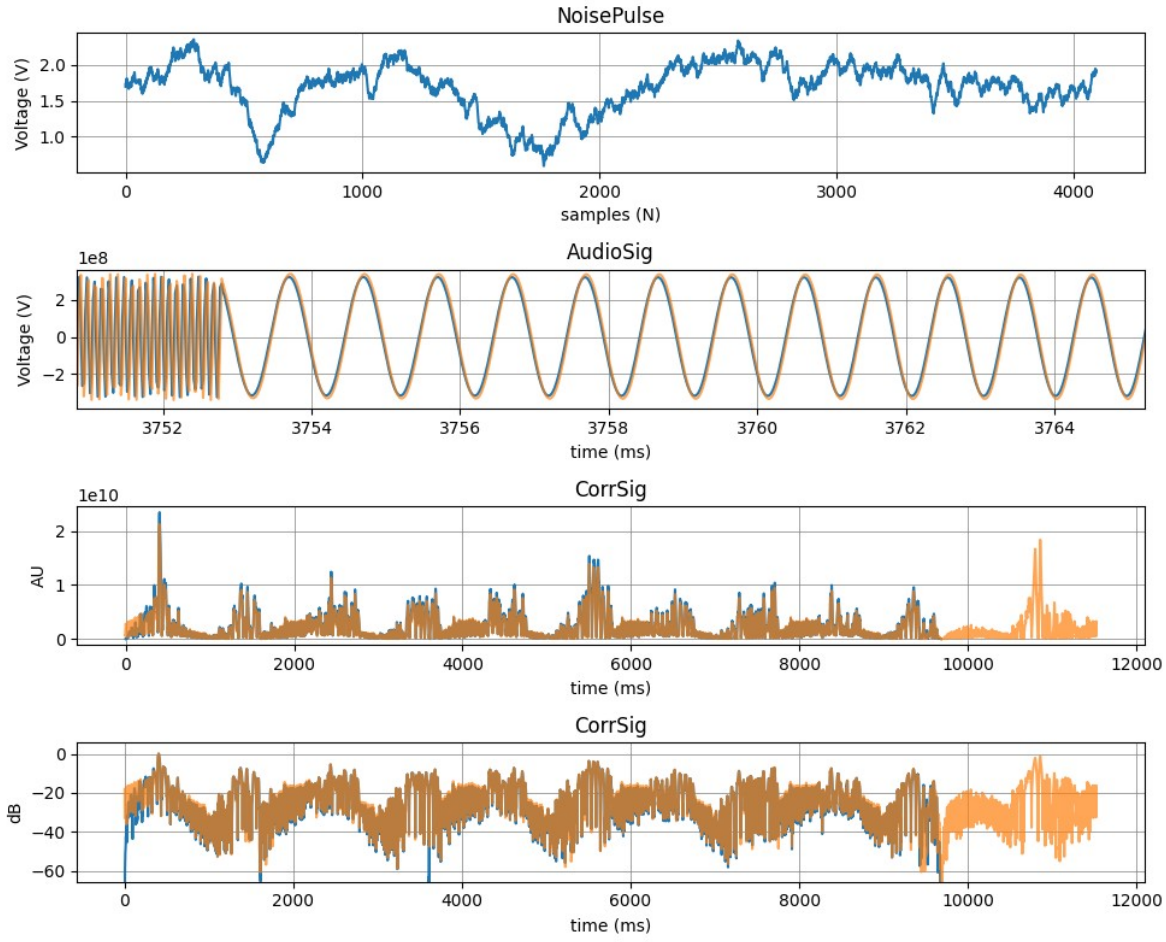


Figure 20: Multiple waveforms of the measurment and allignment of two signals. Barely visible misallignment.

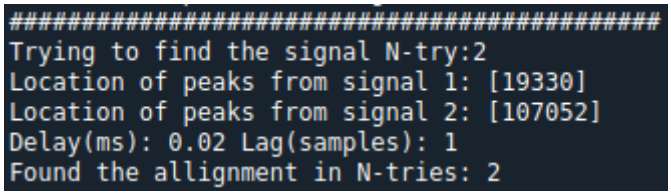


Figure 21: Output of the postprocessing software, a sub 1ms delay was measured.

Throughout more than a dozen measurement campaigns, including idealised dual-channel trials and fully independent Zoom H5 recordings, the Brownian-noise bursts were consistently recovered, and the post-processing pipeline aligned all channels to within 1 ms, with the best case reaching 0.02 ms (one sample at 48 kHz). Taken together, these results demonstrate that the prototype meets every functional requirement: the watermark is robust yet unobtrusive, the hardware operates autonomously from a single 3.3 V rail, and the correlation-based decoder delivers sub-millisecond synchronisation across heterogeneous recorders. Hence, the system is ready for integration on a dedicated PCB and for the forthcoming wireless distribution phase.

4 Conclusion

In this project, we have designed, built, and validated a fully standalone analog watermark generator that embeds unique Brownian noise synchronization bursts into audio signals, without any modification to off the shelf recorders. By implementing a pseudo random noise burst generator on an STM32G431KB, a precision summing amplifier mixing circuit, and a robust cross correlation post processing pipeline, we achieved reliable extraction of synchronization markers and demonstrated sub millisecond alignment across both idealized dual channel tests and real world ZOOM H5 recordings. These results confirm that our approach can deliver high fidelity analog audio watermarking and precise time alignment. We are enthusiastic about refining the system through streamlining scale control, migrating to a custom PCB, exploring wireless distribution and source localization, and further reducing perceptibility through adaptive filtering, and we look forward to continuing this work toward a versatile, field deployable synchronization solution.

4.1 Future work

- Simplify amplitude and scale setting.
  - Integrate digitally controlled potentiometers or add microcontroller-driven gain control to streamline repeatable scale adjustments.
- Develop a dedicated PCB.
  - Migrate from breadboard to a custom PCB for the mixer and sequence generator to improve robustness, reduce noise coupling, and ease deployment.
- Implement wireless synchronization.
  - Attach RF modules and appropriate antennas to distribute the noise sequence over radio, eliminating the need for physical sync cables.
- Leverage the synchronisation with microphone inputs for source localization.
  - Incorporate microphone arrays at each node and explore direction-of-arrival algorithms to localize audio sources using the synchronised audio recordings.
- Suppress noise bursts at signal peaks.

- Knowing the noise bursts and their location they can be subtracted from the recordings, increasing the fidelity of the audio

- Apply smoothing window functions.
  - Shape the noise-burst envelope (e.g., using Hanning or Gaussian windows) to reduce spectral leakage and further minimize perceptibility in primary audio.

4.2 Specification overview

- Every watermark should be easily discernible and have a unique sequence.
  - By generating 4 096-sample Brownian noise bursts via an LFSR-based algorithm on the STM32G431KB, each watermark exhibits a pseudo-random pattern with a distinct correlation signature. In post-processing, these bursts produce sharp peaks in the cross-correlation, ensuring reliable detection across multiple recordings.
- The embedded metadata and synchronization signals must be easily extractable during post-processing for precise alignment of audio data from different devices.
  - The combination of DC-bias removal, upsampling to match the recorder's sample rate, and a median-filter plus Gaussian smoothing pipeline enables robust extraction of the embedded noise bursts. Tests runs with both idealized (dual-channel) and real-world (separate ZOOM H5 recorders) setups confirmed consistent peak detection and alignment.
- Synchronization should be accurate to the nearest 1 ms.
  - Cross-correlation of two recorded tracks, each mixed with the same Brownian sequence, yielded relative delays consistently below 1 ms—and in the best case down to 0.02 ms (one sample).
- The system must be fully standalone, operating independently without reliance on external sources or networks.
  - All synchronization signal generation and mixing run independently of external networks or GPS, relying solely on on-board STM32 peripherals and analog circuitry.
- The system should not require any modification to the recorder it is connected to.

    - The mixed audio + sync output, easily interfaces with standard recorders (ZOOM H5) without firmware or hardware changes.

- The system should include the necessary components to mix the synchronization signal with the primary audio signal, controlling their relative amplitude and the full-scale amplitude of the mix. + The watermarking process should introduce minimal distortion to the primary audio signal, ensuring high fidelity of the recorded data
    - A single-supply summing-amplifier topology using MCP6001 op-amps provides precise amplitude control via potentiometers, preserving audio fidelity. Spectrogram analyses of mixed signals confirm the expected low-frequency roll-off of Brownian noise and imperceptibility to the human ear.

5 References

[1] Open Acoustic Devices. **AudioMoth – Low-cost, full-spectrum acoustic logger.** https://www.openacousticdevices.info/audiomoth

[2] Lin, Y. & Abdulla, W. ***Audio Watermark: A Comprehensive Foundation Using MATLAB.*** Springer, 2015. https://link.springer.com/book/10.1007/978-3-319-07974-5

[3] Wikipedia. **Brownian noise.** https://en.wikipedia.org/wiki/Brownian_noise

[4] STMicroelectronics. **NUCLEO-G431KB – STM32 Nucleo-32 development board.** https://www.st.com/en/evaluation-tools/nucleo-g431kb.html

[5] STMicroelectronics. ***STM32CubeIDE.*** *https://www.st.com/en/development-tools/stm32cubeide.html*

[6] STMicroelectronics. ***RM0440 – STM32G4 Series Reference Manual.*** Rev 6, Nov 2024. https://www.st.com/resource/en/reference_manual/rm0440-stm32g4-series-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[7] STMicroelectronics. **STM32G431x6/x8/xB Device Data Sheet – 32-bit Arm Cortex-M4 MCU.** https://www.st.com/resource/en/datasheet/stm32g431c6.pdf

[8] STMicroelectronics. ***UM2397 – STM32G4 Nucleo-32 board (MB1430) User Manual.*** Rev 4, Aug 2023. https://www.st.com/resource/en/user_manual/um2397-stm32g4-nucleo32-board-mb1430-stmicroelectronics.pdf

[9] STMicroelectronics. **UM1727 Getting started with STM32 Nucleo board software development tools.** https://www.st.com/resource/en/user_manual/um1727-getting-started-with-stm32-nucleo-board-software-development-tools-stmicroelectronics.pdf

[10] Carter, B. & Brown, T. ***Handbook of Operational Amplifier Applications.*** Texas Instruments, 2016. https://www.ti.com/lit/an/sboa092b/sboa092b.pdf

[11] Jung, W. G. ***Audio IC Op-Amp Applications.*** 3rd ed., Howard W. Sams, 1987.

[12] Microchip Technology Inc. ***MCP6001/2/4 – 1 MHz Low-Power Operational Amplifier Data Sheet.*** https://ww1.microchip.com/downloads/en/DeviceDoc/MCP6001-1R-1U-2-4-1-MHz-Low-Power-Op-Amp-DS20001733L.pdf

[13] **KiCad EDA – Schematic Capture & PCB Design Software.**
https://www.kicad.org/

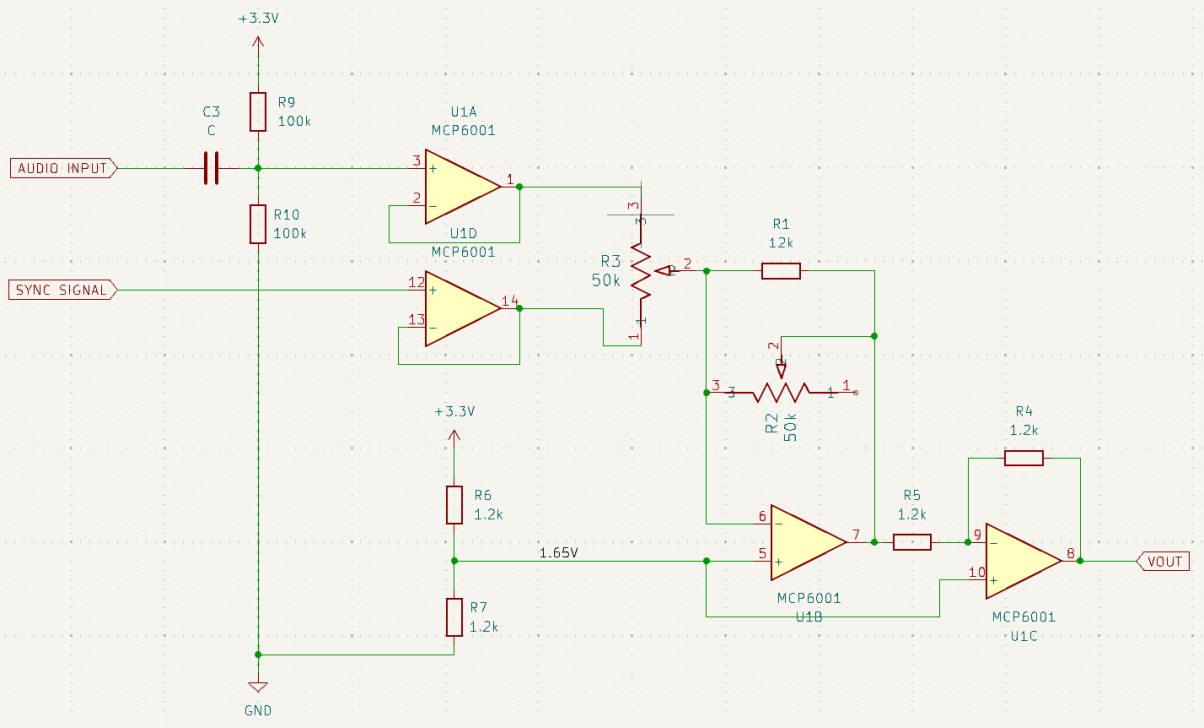
[14] **Analog Discovery 2 – Reference Manual.** Digilent.
https://digilent.com/reference/test-and-measurement/analog-discovery-2/reference-manual

[15] **WaveForms 3 – Virtual Instrument Suite.** Digilent.
https://digilent.com/reference/software/waveforms/waveforms-3/start

[16] **H5 Handy Recorder.** Zoom Corporation product page.
https://zoomcorp.com/en/jp/handy-recorders/handheld-recorders/h5/

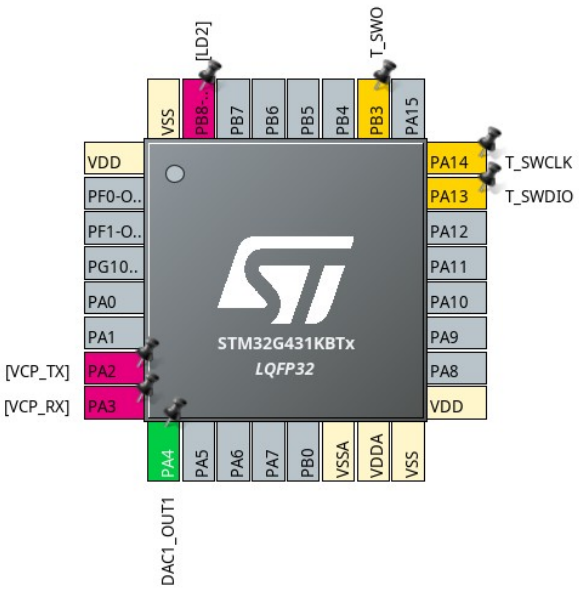
Appendix

A. Mixing circuit schematics

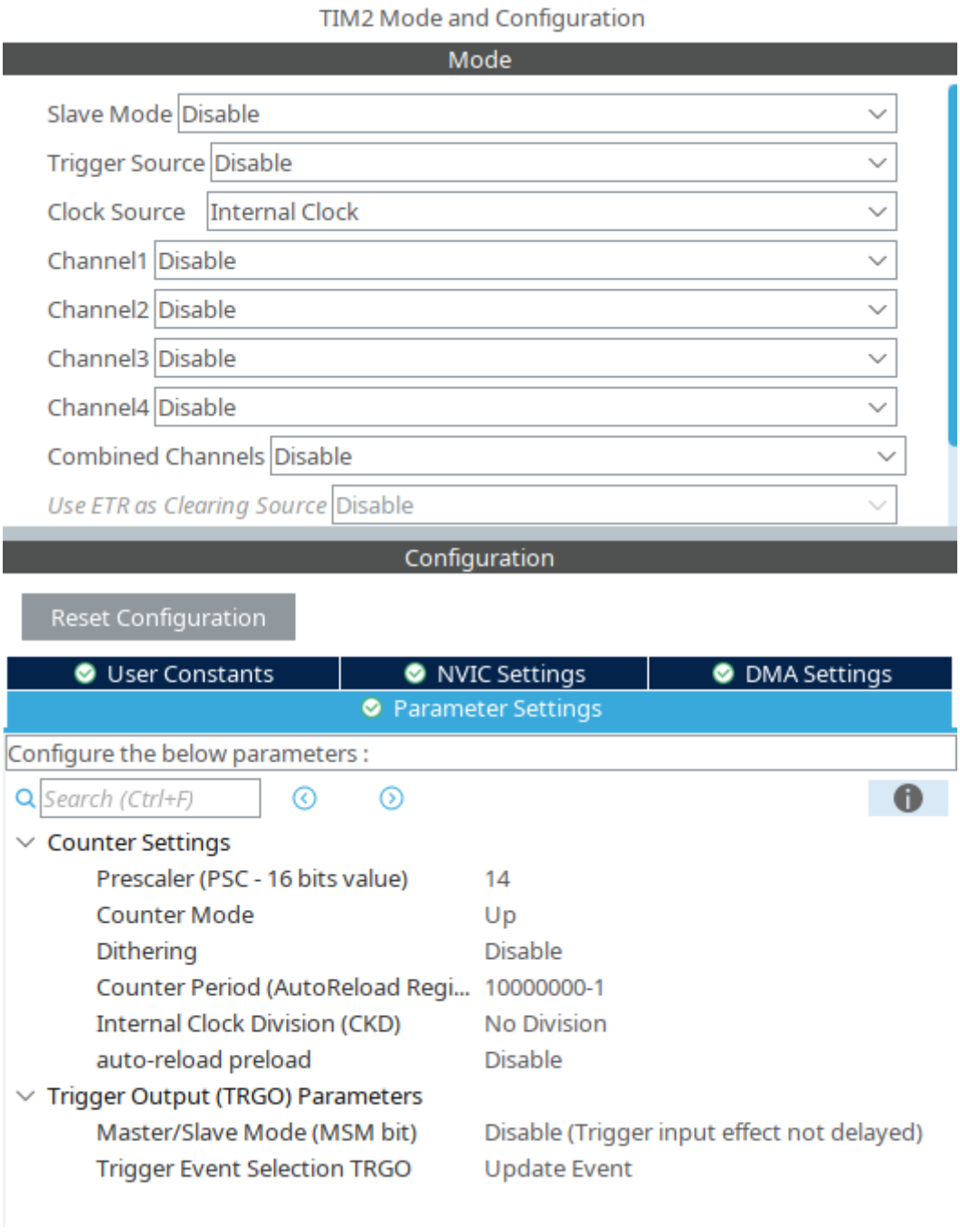


B. Synchronisation signal generation code – STM32

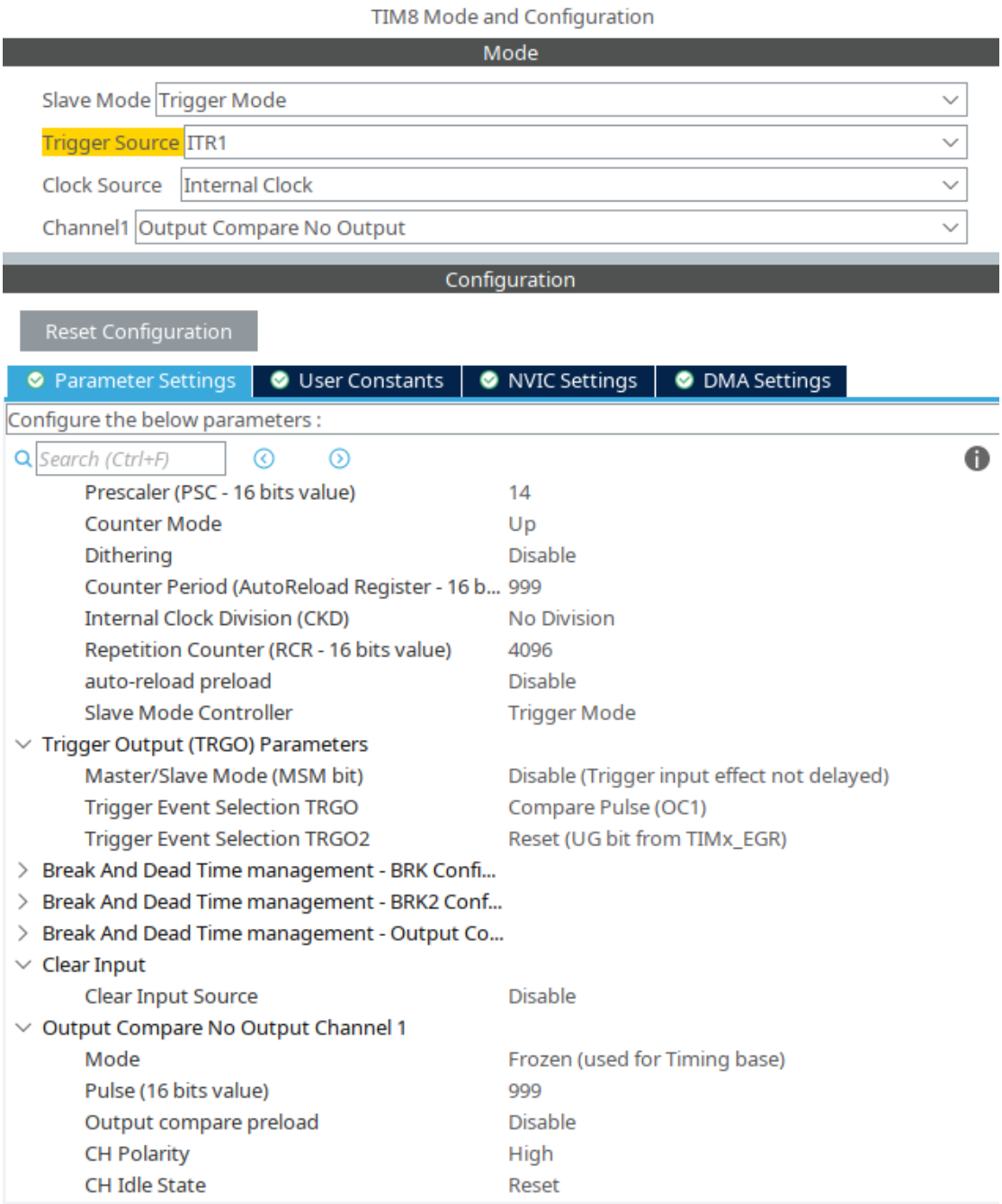
a. STM32G4 pinout schematic



b. STM32 TIM2 setup in CUBE



c. STM32 TIM8 setup in CUBE – OneShot is also ticked under the Mode menu



d. STM32 IDE Code – main() - included is only user code

```
  /* USER CODE BEGIN WHILE */
  //First generation
  GenerateSeq(out_seq);
  out_seq[4096] = 2048; //Last element should be middle rail

  uint32_t *dac_address = (uint32_t *)&(DAC1->DHR12R1);

  HAL_DMA_Init(&hdma_tim8_ch1);
  HAL_DMA_Start(&hdma_tim8_ch1, out_seq, dac_address, 4097); //One more
than 4096 because I want the zero to be the last value
  HAL_DAC_Start(&hdac1, DAC1_CHANNEL_1);

  TIM8->DIER |= TIM_DIER_CC1DE; //Autogen misses some registers...
  TIM8->DIER |= TIM_DIER_UIE;
  HAL_TIM_OC_Start_IT(&htim8, TIM_CHANNEL_1);

  TIM2->DIER |= TIM_DIER_UDE;
  HAL_TIM_Base_Start(&htim2);

  while (1)
  {
      //For monitoring/debug in STM32 Monitor – No functional value
        DAC1_mon = DAC1->DHR12R1;
        Seed_mon = lfsr;
        DMA_CNT_mon = DMA1_Channel1->CNDTR;
        TIM8_CNT_mon = TIM8->CNT;
        TIM2_CNT_mon = TIM2->CNT;

    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
  }
```

e. STM32 IDE Code – Burst generation

```
void TIM8_UP_IRQHandler(void)
{ // Every time TIM8 Finishes with repetition and counting – stops
genertate new sequence to be used the next time.
      GenerateSeq(out_seq);
      HAL_TIM_IRQHandler(&htim8);
}


uint16_t white_noise_sample(){
    // 16-bit Galois LFSR (XOR-Shift)
    lfsr ^= lfsr >> 7;
    lfsr ^= lfsr << 9;
    lfsr ^= lfsr >> 13;
    // Mask to 12 bits (0-4095)
    return lfsr & 0x0FFF;
}


uint16_t generate_new_seed(uint16_t current_seed) {
      current_seed = current_seed ^ 0xB400;
      if (current_seed == 0) {current_seed = 1;}
    return current_seed; // Prime-derived XOR mask
}


//Generates a brownian sequence, gets ready for the next time it will be
needed
void GenerateSeq(uint16_t *SeqArray){


      for (int i = 0; i < 4096; i++) { //This generates a whole next
sequence to be transmitted
            uint16_t white_noise = white_noise_sample();
            int32_t step = (white_noise & 0x7F) - 64; // Small step (-64 to
+63)
            if (white_noise & 1) {step += 1;} // Ensure symmetry
            int32_t correction = (2048 - brownian_state) / 256; // Small
force toward center
            step += correction;
```

```
            brownian_state += step; // Integrate step


            if (brownian_state < 0) {brownian_state = 0;}
            if (brownian_state > 4095) {brownian_state = 4095;}


            SeqArray[i] = (uint16_t)brownian_state;
      }
      brownian_state = 2048;
      lfsr = generate_new_seed(lfsr); //Create new seed for the next
generation
}
```

f. STM32 Private variables

```
static uint16_t lfsr = 0x3701; // First seed
static int32_t brownian_state = 2048; // Start in the middle of 12-bit
range
uint16_t out_seq[4097] = {0};


uint32_t DAC1_mon = 0;
uint32_t DMA_CNT_mon = 0;
uint32_t TIM8_CNT_mon = 0;
uint32_t TIM2_CNT_mon = 0;
uint16_t Seed_mon = 0;
```

C. Python code

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator, AutoMinorLocator
from scipy.io import wavfile
from scipy.signal import correlate
from scipy import signal
from scipy.signal import resample, find_peaks, medfilt
from scipy.ndimage import gaussian_filter1d



TimeConv = 1



class BrownianNoiseGenerator:
    def __init__(self, seed=0x3701):
        self.lfsr = seed if seed != 0 else 1  # Avoid zero state
        self.brownian_state = 2048  # Start in the middle of the 12-bit
range

    def white_noise(self):
        """Generate a 12-bit pseudo-random number using LFSR."""
        self.lfsr ^= (self.lfsr >> 7) & 0xFFFF
        self.lfsr ^= (self.lfsr << 9) & 0xFFFF
        self.lfsr ^= (self.lfsr >> 13) & 0xFFFF
        return self.lfsr & 0x0FFF  # Return 12-bit value (0-4095)

    def generate_new_seed(self):
        """Generate a new LFSR seed."""
        self.lfsr ^= 0xB400
        if self.lfsr == 0:
            self.lfsr = 1

    def generate_sequence(self):
        """Generate a sequence of 12-bit Brownian noise."""
        seq_array = []
```

```
        for _ in range(4096):
            white_noise = self.white_noise()
            step = (white_noise & 0x7F) - 64  # Small step (-64 to +63)

            if white_noise & 1:
                step += 1  # Ensure symmetry

            correction = (2048 - self.brownian_state) // 256  # Small force
toward center
            step += correction

            self.brownian_state += step  # Integrate step

            self.brownian_state = max(0, min(4095, self.brownian_state))
            seq_array.append(self.brownian_state)

        self.brownian_state = 2048
        self.generate_new_seed()  # Create new seed for the next generation

        return np.array(seq_array, dtype=np.float64)

    def set_seed(self, seed):
        """Set a new LFSR seed (avoiding zero)."""
        self.lfsr = seed if seed != 0 else 1

    def upsample(self, noise_sequence, input_rate=10000,
output_rate=50000):
        """Upsample the noise sequence using linear interpolation."""
        factor = output_rate / input_rate
        upsampled_length = int(len(noise_sequence) * factor)
        return resample(noise_sequence, upsampled_length)
```

```
def detect_high_peaks(signal, height_offset=0.9):



    signal = abs(signal)
    signal = medfilt(signal, 27)
    signal = gaussian_filter1d(signal, 13)


    peaks, properties = find_peaks(signal, distance=2000)


    if len(peaks) == 0:
        return [], {}


    peak_heights = properties['peak_heights'] if 'peak_heights' in
properties else signal[peaks]
    mean_height = np.mean(signal)
    std_height = np.std(signal)
    max_height = max(peak_heights)
    threshold = 0.9*max_height




    high_peaks = peaks[peak_heights > threshold]
    if(len(high_peaks) > 1): #If there were more peaks found then it must
be noisy and we need to discard
        return signal, [], properties
    return signal, high_peaks, properties
```

```
def PulseDetect(FileName, NoisePulse):
    global TimeConv

    # Read audio file
    Fs, AudioSig = wavfile.read(FileName)
    AudioSig = AudioSig.astype(np.float64)
    AudioSig = AudioSig



    NoisePulse = generator.upsample(NoisePulse, input_rate=10000,
output_rate=Fs) #Upsample to frequency of the measured signal
    NoisePulse = (NoisePulse/4096)*3.3 # Normalize

    TimeConv = (1/Fs)*1000

    NoisePulse = NoisePulse - np.mean(NoisePulse) #remove DC Bias
    AudioSig = AudioSig - np.mean(AudioSig) #remove DC Bias

    # Perform cross-correlation
    CorrSig = correlate(AudioSig, NoisePulse, mode='full', method='direct')

    CorrSig, high_peaks, _ = detect_high_peaks(CorrSig, 0.9)

    return CorrSig, high_peaks, AudioSig



def ShiftSignals(signal, shift_val):
    shifted_signal = np.roll(signal, shift_val)

    if shift_val > 0: #Signal shifted to the right
        shifted_signal[:shift_val] = 0
    elif shift_val < 0: #signal shifted to the left
        shifted_signal[len(shifted_signal)-shift_val:] = 0
    else:
        pass

    return shifted_signal
```

```
def AllignSignals(sig1, sig2, peaks1, peaks2): #TODO make it so that you
can allign multiple signals


    shift = peaks1[-1] - peaks2[-1]  # Compute shift based on the last
detected peak


    sig2 = ShiftSignals(sig2, shift)


    return sig1, sig2


def sweep_delay(sig1, sig2, fs, refine=True):


    # remove any DC offset
    x1 = sig1 - np.mean(sig1)
    x2 = sig2 - np.mean(sig2)

    # full cross-correlation
    corr = correlate(x2, x1, mode='full')

    # lag axis: from -(N-1) to +(N-1)
    n = len(x1)
    lags = np.arange(-n+1, n)

    # index of max correlation
    i_peak = np.argmax(corr)
    lag = lags[i_peak]

    delay_s = lag / fs
    return delay_s, lag
```

```
if __name__ == "__main__":
    # Instantiate the generator with a specific seed
    generator = BrownianNoiseGenerator(seed=0x3701)  # Set the same seed as
in C


    MainSig = []
    AllignSig = []
    CorrSig1 = []
    CorrSig2 = []


    for i in range(0,20): # go over 10 possible noises sequences, if it
exists align and quit
        NoisePulse = generator.generate_sequence()# Normalize



        CorrSig1, high_peaks1, AudioSig1 =
PulseDetect("./RecorderRecordings/Left/record_left07.WAV", NoisePulse)
        CorrSig2, high_peaks2, AudioSig2 =
PulseDetect("./RecorderRecordings/Right/record_right07.WAV", NoisePulse)


        print("################################################")
        print("Trying to find the signal N-try:" + str(i))
        print("Location of peaks from signal 1: " + str(high_peaks1))
        print("Location of peaks from signal 2: " + str(high_peaks2))
        try:
            if len(high_peaks1) != 0 and len(high_peaks2) != 0:
                if len(high_peaks1) < len(high_peaks2):
                    high_peaks1 = np.pad(high_peaks1, (len(high_peaks2) -
len(high_peaks1), 0))


                elif len(high_peaks2) < len(high_peaks1):
                    high_peaks2 = np.pad(high_peaks2, (len(high_peaks1) -
len(high_peaks2), 0))


        except:
            print("Delta of the signals: " + "No deltas found")


        if len(high_peaks1) != 0 and len(high_peaks2) != 0:
```

```
            MainSig, AllignSig = AllignSignals(AudioSig1, AudioSig2,
high_peaks1, high_peaks2)
            CorrSig1, CorrSig2 = AllignSignals(CorrSig1, CorrSig2,
high_peaks1, high_peaks2)


            delay_s, lag = sweep_delay(MainSig, AllignSig, 50000,
refine=True)
            print("Delay(ms): " + str(delay_s*1000) + " Lag(samples): " +
str(lag))


            print("Found the allignment in N-tries: " + str(i))


            break
            pass
        else:
            MainSig = []
            AllignSig = []
            CorrSig1 = []
            CorrSig2 = []
        pass

    # Plotting

    t_main = np.arange(len(MainSig)) * TimeConv
    t_alling = np.arange(len(AllignSig)) * TimeConv
    t_corr1 = np.arange(len(CorrSig1)) * TimeConv
    t_corr2 = np.arange(len(CorrSig2)) * TimeConv

    fig, axs = plt.subplots(4, 1, figsize=(10, 8))

    major_spacing = 1000  # major ticks every 100 ms
    for ax in axs.flatten():
        ax.grid(which='major', linestyle='-', linewidth=0.5, color='gray')



    CorrSig1DB = CorrSig1 / np.max(CorrSig1)
    CorrSig1DB   = 20 * np.log10(CorrSig1DB + 1e-12)
```

```
    CorrSig2DB = CorrSig2 / np.max(CorrSig2)
    CorrSig2DB   = 20 * np.log10(CorrSig2DB + 1e-12)

    axs[0].plot((NoisePulse/4098) * 3.3)
    axs[0].set_title("NoisePulse")
    axs[0].set_xlabel("samples (N)")
    axs[0].set_ylabel("Voltage (V)")

    axs[1].plot(t_main, MainSig)
    axs[1].plot(t_alling, AllignSig, alpha=0.7)
    axs[1].set_title("AudioSig")
    axs[1].set_xlabel("time (ms)")
    axs[1].set_ylabel("Voltage (V)")

    axs[2].plot(t_corr1, CorrSig1)
    axs[2].plot(t_corr2, CorrSig2, alpha=0.7)
    axs[2].set_title("CorrSig")
    axs[2].set_xlabel("time (ms)")
    axs[2].set_ylabel("AU")

    axs[3].plot(t_corr1, CorrSig1DB)
    axs[3].plot(t_corr2, CorrSig2DB, alpha=0.7)
    axs[3].set_title("CorrSig")
    axs[3].set_xlabel("time (ms)")
    axs[3].set_ylabel("dB")



    plt.tight_layout()
    plt.show()



#
==========================================================================
==
#     # Generate the spectrogram
#     frequencies, times, spectrogram =
signal.spectrogram(AudioSig[0:300000], Fs)
#     # Plot the spectrogram
```

```
#     plt.figure(figsize=(10, 4))
#     plt.pcolormesh(times, frequencies, 10 * np.log10(spectrogram),
shading='gouraud')
#     plt.ylabel('Frequency [Hz]')
#     plt.xlabel('Time [sec]')
#     plt.title('Spectrogram')
#     plt.colorbar(label='Intensity [dB]')
#     plt.show()
#
================================================================================
==


    # Playing the audio (requires simpleaudio or sounddevice)


    # import sounddevice as sd
    # sd.play(AudioSig / np.max(np.abs(AudioSig)), samplerate=Fs)
```