Roskilde University

Specialization Project in Computer Science F2025

MSc in Computer Science - 2nd Semester

COURSE

PHYSICAL COMPUTING

Members:
MAREK LASLO 82428

Source Code Repository:

Git Repository

https://github.com/RUC-MSc-CS-2sem-PC-2025/Specialisation-Project

Handed in:

2nd June of 2025

Contents

1 Introduction

This project focuses on the design and implementation of a digital musical instrument (DMI), specifically a microcontroller-based digital synthesiser that employs digital signal processing (DSP) techniques for sound generation and modification.

The report begins by establishing the central research question and outlining the project's practical constraints and intended context of use (Section 2). This foundation informs the development of technical requirements, supported by a review of relevant theory in acoustics, sensor technology, and signal processing (Section 3).

Design decisions are explored in Sections 4 and 5, covering hardware selection and the development of a responsive, low-latency software architecture. These sections also address key design challenges and trade-offs.

Section 6 details the implementation process, including build instructions and the application of DSP techniques. The system's performance is evaluated against the specified requirements in Section 7, followed by a short discussion and future work in Section 8 and conclusion in Section 9.

1.1 Motivation

The motivation behind this project is to contribute to the advancement of digital musical instruments by examining the design process and addressing common challenges in microcontroller-based systems. It aims to support both educational and artistic communities by offering insights for beginners and developing a synthesiser suitable for performative and creative applications.

2 Problem Definition

This section introduces the central research question that informs the development of the digital synthesiser. It builds upon the broader project context and constraints outlined in Section 2.2, and serves as the foundation for the technical and design requirements presented in Section 3. By clearly defining the problem space, the scope and direction for the subsequent design (Sections 4 and 5) and implementation efforts (Section 6) are established.

2.1 Research Question

To explore the intersection of physical computing, digital audio synthesis, and digital synthesiser design, this project poses the following research question:

How can a microcontroller-based synthesizer be developed to generate audio using digital signal processing and to be responsive to human and environmental stimuli?

This research question explores how a microcontroller, using digital signal processing (DSP) algorithms, can interact with physical stimuli to generate audio output. To address this question, the project will investigate several key areas:

- understanding the fundamental principles of sound,
- selecting appropriate hardware components, such as microcontrollers and sensors,
- implementing real-time DSP techniques
- ensuring the system is responsive to both environmental and human stimuli.

2.2 Project Context and Limitations

The synthesiser is intended to operate as a component within a larger collaborative installation, which is scheduled for exhibition at Roskilde Festival in 2025. This project specifically examines the development of a digital musical instrument, with particular emphasis on hardware selection and the implementation of real-time audio digital signal processing software.

The physical encapsulation of the instrument and the protection of its sensors are not addressed within the scope of this report, as these considerations will be incorporated during the integration of the instrument into the final collaborative installation for Roskilde Festival 2025.

2.2.1 Environment and Monitoring Specification

Given that the project is required to respond to environmental stimuli, and considering the wide range of possible environmental factors alongside the constraints of time and resources available at RUC FabLab, the device has been designed to focus exclusively on monitoring climatic factors, as defined in Section 3.3.1.

2.2.2 Use of Instrument

The instrument's controls should be intuitive, meaning that operating the device should not require prior musical knowledge or technical skill.

2.2.3 Music Specification

The audio produced is intended to be experimental and, therefore, should remain unconstrained in its sound design. Given that listening is a subjective experience, no judgments or definitions of what constitutes "good" or "bad" sound should be imposed. Additionally, the output audio must be clean, free from artifacts caused by improper application of digital signal processing algorithms or issues related to real-time processing and CPU performance.

2.2.4 Availability of Components

The project should be constructed from the components that are available at RUC Fablab. And total amount spent on the project should not exceed 50€.

2.2.5 Open-Source Commitment

As this is a university project focused on skill development and the promotion of open knowledge, all components, including hardware designs, software code, and documentation, will be made publicly available. The project exclusively relies on open-source tools and libraries, with the exception of spectral analysis tools, which are free to use but not open-source.

2.2.6 Software Utilised

The following software tools were employed throughout the development of the project:

- **Integrated Development Environment (IDE):** Visual Studio Code (Link) was used for writing and editing source code.
- **Programming Language:** C++ (Link) served as the primary programming language for firmware development.
- **Graph Plotting:** Python (Link), in conjunction with Jupyter Notebooks (Link), was utilised for data visualisation and analysis.
- **Monitoring Tools:** Voxengo Span (Link) (not open-source) was used to monitor audio signals.
- **Electronic Schematics:** KiCad (Link) was employed for designing circuit schematics.

3 Requirements

In this section, the background theory necessary for deriving the technical requirements of the system is presented. Core concepts from acoustics are first introduced to identify which sound attributes can be meaningfully influenced (Section 3.1). This is followed by an examination of the principles of digital audio (Section 3.2) to establish the capabilities and limitations of microcontrollers in real-time digital signal processing applications. Relevant aspects of sensor technology are then explored (Section 3.3) to determine how environmental climatic factors and human stimuli can be detected. Digital signal processing techniques for sound synthesis and modification are discussed in Section 3.4. Based on this foundation, a set of functional requirements is derived (Section 3.5) to guide the design analyses presented in Sections 4 and 5.

3.1 Attributes of Sound

Understanding the attributes of sound is essential for designing a digital musical instrument capable of meaningful audio synthesis. This section explores both the physical and perceptual characteristics of sound, which inform the initial technical requirements of the project.

- Nature of Sound (Section 3.1.1)
- Perceived Attributes of Sound (Section 3.1.2)

3.1.1 Nature of Sound

To facilitate a clearer understanding of sound, a simple sine wave is used as an illustrative example, as shown in Figure 1.

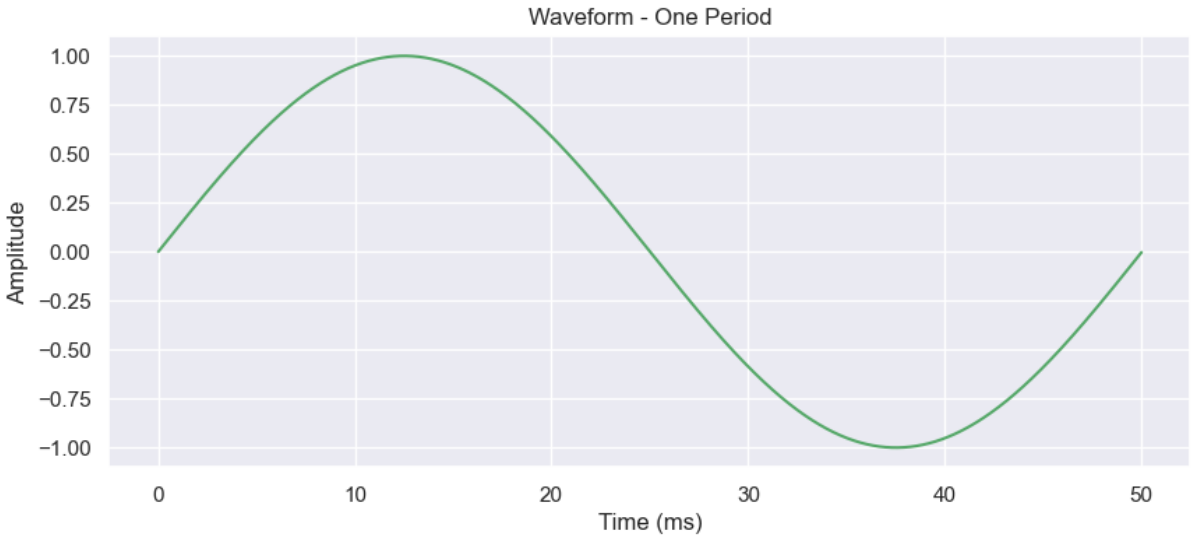

Figure 1: *A plot of one cycle of a sine wave at a frequency of* 20*Hz*

Figure 1 visualises a sine wave, which has been plotted using the Equation 1 (Park, 2009):

$$y(t) = A \cdot \sin(2 \cdot \pi \cdot f \cdot t + \varphi) \tag{1}$$

The variables in Equation 1 correspond to fundamental physical attributes of sound:

- **Duration** (t) – The lifespan of a tone, expressed in seconds.
- **Frequency** (f) – Frequency is a perceptual attribute of sound associated with its periodic, or more commonly, quasi-periodic characteristics. It represents the number of complete oscillation cycles that occur per second and is typically measured in Hertz (Hz).
- **Amplitude** (A) – Refers to the strength or intensity of the signal. Amplitude is usually normalised to the range of ± 1.0 and can include all real values within this range.
- **Initial Phase** (φ) – Indicates the point at which the oscillation begins, measured in radians.

From the explanation above, it can be concluded that the fundamental physical attributes of sound include duration (t), amplitude (A), pitch or frequency (f), and phase

(φ). Although this example uses a simple sine wave, the more complex waveforms also share these properties, albeit with more intricate structures see Figure 2.

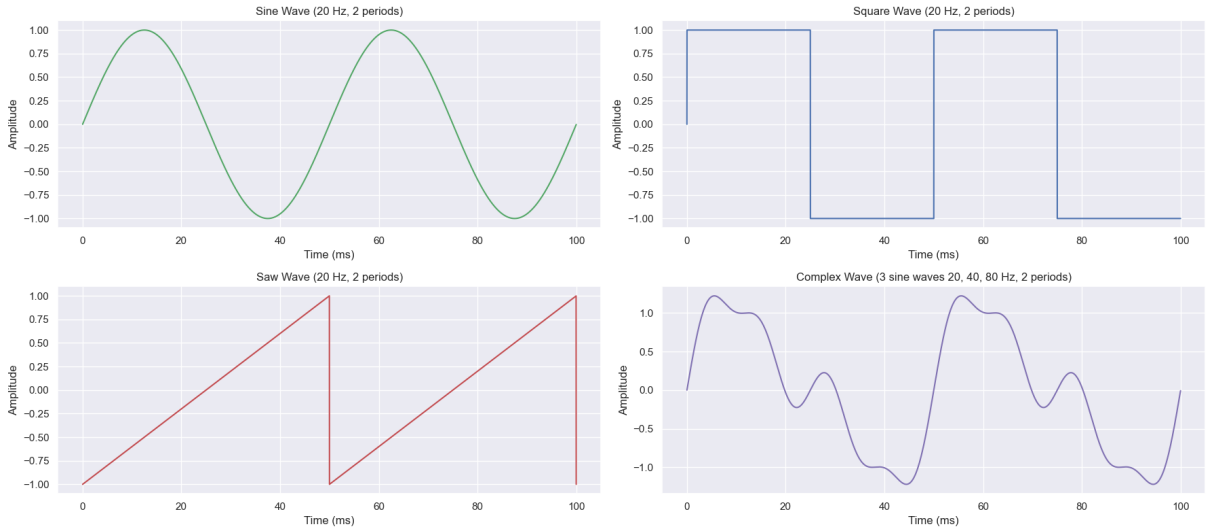

Figure 2: *Representation of different audio waveforms: Sine, Square, Saw, and Complex waveforms*

Figure 2 presents a representation of waveforms in the time domain. An alternative method of representing waveforms is in the frequency domain, as illustrated in Figure 3. In this representation, a waveform is decomposed into multiple sine waves. The most dominant component is referred to as the root (or fundamental) frequency, while the remaining components are identified as harmonics.

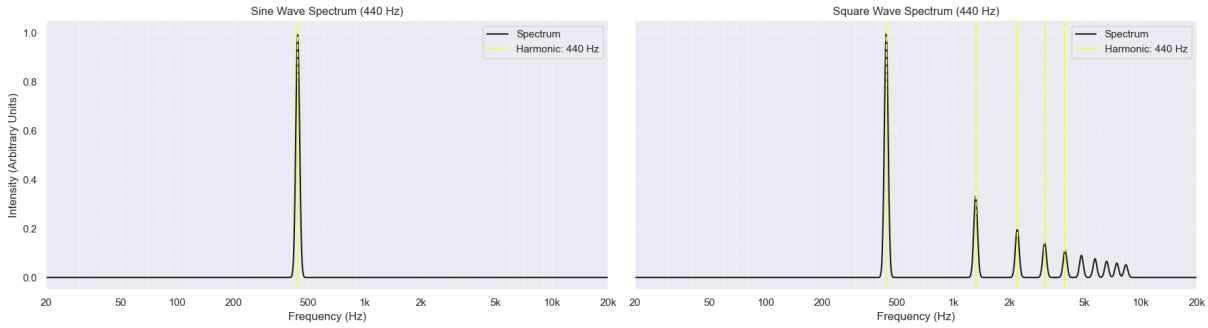

Figure 3: *Representation of sine wave and square wave in frequency spectrum.*

Section 3.1.2 explores how different waveforms and alterations in physical properties are perceived by the human brain.

3.1.2 Perceived Attributes of Sound

Having established the fundamental physical attributes of sound, it is important to examine how the brain perceives these sounds.

Sound consists of vibrations of air molecules, which are detected by the eardrums. These vibrate at the same frequency as the air vibrations. It is well established that human hearing can recognise frequencies ranging from approximately 20 Hz to 22,000 Hz (Levitin, 2019).

Perceived sound can be characterised by the following attributes (Levitin, 2019):

- **Tone:** The sound that is heard.
- **Pitch:** The psychological perception of how high or low a tone sounds, which is related to its frequency and position within a scale. Higher frequencies correspond to higher perceived pitches.
- **Timbre:** The unique tonal quality that enables the distinction between different instruments; it is associated with the shape of the waveform, or more specifically, the harmonic content it comprises.
- **Loudness:** The perceived volume of a tone, related to its amplitude.
- **Spatial Location:** The perceived direction or position from which the sound originates, often considered in stereo sound.
- **Reverberation:** The perception of space or distance in sound, influenced by echoes and room size, which contributes to emotional depth and atmosphere.

These attributes are generally separable and can be varied independently without significantly affecting one another.

3.1.3 Conclusion

In conclusion, the physical properties of sound are perceived by the brain as distinct sound attributes. Consequently, altering the physical properties of sound results in changes to these perceived attributes.

- The device should be capable of altering at least one of the sound attributes described in Section 3.1.2.

3.2 Microcontroller Requirements for High-Quality Audio

This section provides a detailed examination of real-time audio processing challenges and formulates requirements that will assist in selecting an appropriate microcontroller, as discussed later in Section 4.

This section examines the following four areas:

- Audio Signal Processing (Section 3.2.1)
- Sampling, Sample Rate, and Aliasing (Section 3.2.2)
- Quantisation and Bit Depth Effects (Section 3.2.3)

3.2.1 Audio Signal Processing

Based on the definition by Juan Louder found online, the DSP can be defined as the manipulation and modification of signals using digital processor. (Louder, 2023)

Mathematical algorithms transform, filter, and enhance audio signals. These processes involve sampling the continuous waveform and presenting it as a sequence of numbers. An analogue signal must first be converted into digital data to process audio signals, and after processing, the digital signal is converted back to the analogue signal. (Louder, 2023)

- ADC - Is a component of DSP which converts analogue signals to digital signals
- DAC - Is a component of DSP which converts digital signals to analogue signals

You can see the visual representation of both conversion processes in Figure 4:

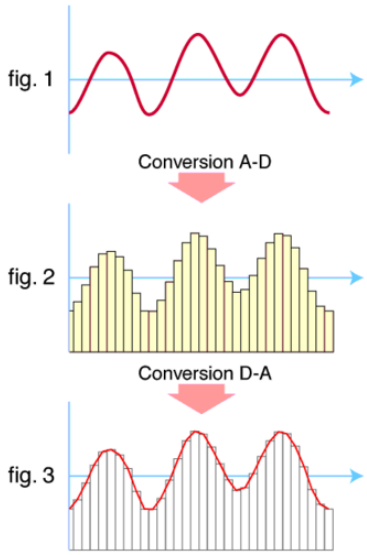

Figure 4: *Conversion from analogue to digital and back to analogue. (Megodenas, 2025)*

In summary, analogue audio is converted into digital audio through analogue-to-digital conversion (ADC), while digital audio is converted back to analogue via digital-to-analogue conversion (DAC). These processes involve mathematical algorithms that transform voltage signals into numerical samples and vice versa, where the numbers represent a digital encoding of sound. Section 3.2.2 will examine the concepts of sampling and sample rate, and will also introduce the issue of aliasing.

3.2.2 Sample, Sample Rate, and Aliasing

As introduced in Section 3.2.1, sampling is the process of converting an analogue audio signal into a digital format by measuring its amplitude at regular intervals (Zölzer, 2011). This is performed by an analogue-to-digital converter (ADC), which produces discrete-time samples from a continuous signal.

The time between samples is called the sampling period (T), and the number of samples taken per second is the sampling frequency or sample rate (f_s), measured in Hertz.

To accurately represent an analogue signal digitally, the sampling frequency must satisfy the Nyquist theorem:

$$f_s \geq 2f_{\max} \tag{2}$$

Equation 2 states that the sample rate must be at least twice the highest frequency component ($f_{\max}$) in the signal to avoid aliasing - an effect that causes higher frequencies to be misrepresented as lower ones (Zölzer, 2011). Figure 5 illustrates this phenomenon, showing how undersampling can lead to incorrect frequency representation.

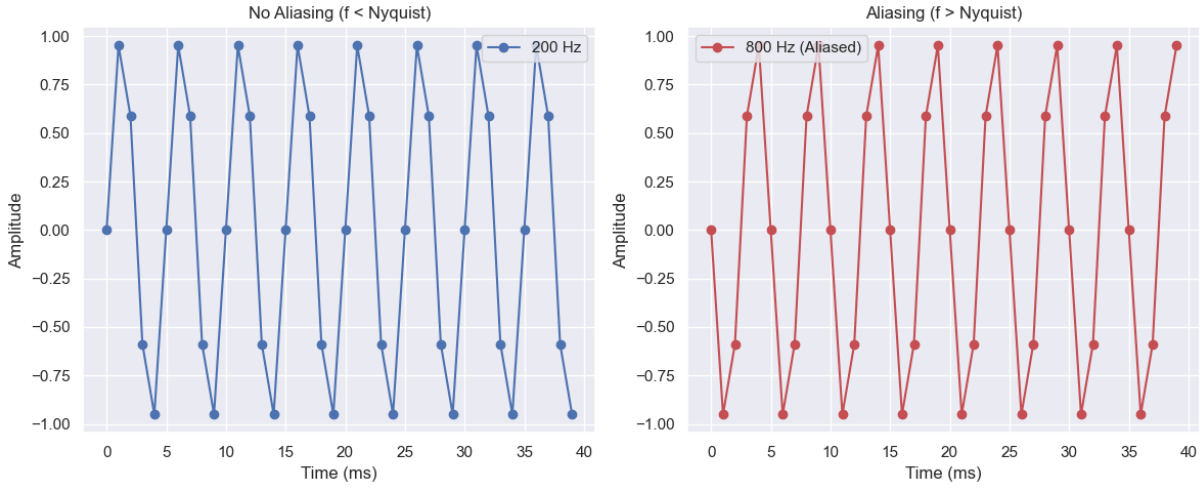

Figure 5: *Left: A 200 Hz sine wave sampled at 1 kHz (below the Nyquist frequency) is accurately reconstructed. Right: An 800 Hz sine wave sampled at 1 kHz (above the Nyquist frequency) is misrepresented as a 200 Hz wave due to aliasing, demonstrating how higher frequencies can fold back into the audible range when sampled below twice their frequency.*

As noted in Section 3.1.2, the upper limit of human hearing is approximately 22,000 Hz. Applying Equation 2, the minimum required sample rate is 44,000 Hz. This aligns with the Compact Disc (CD) standard of 44,100 Hz, which ensures accurate reproduction of the full audible spectrum.

Figure 6 illustrates how a continuous signal is sampled at discrete intervals:

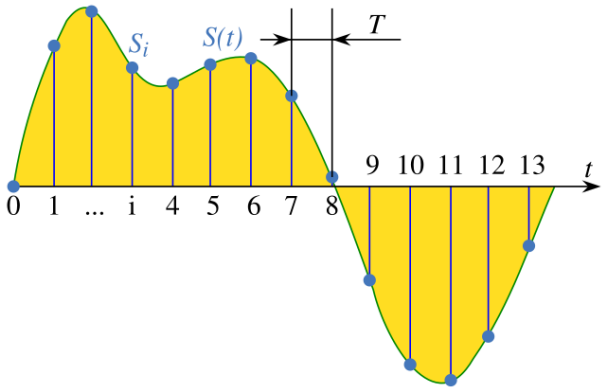

Figure 6: *Signal sampling representation. The continuous signal $S(t)$ is shown in green, while the discrete samples are indicated by blue vertical lines (Email4mobile & Ilyin, 2025).*

In summary, to prevent audible frequency aliasing and ensure high-fidelity audio reproduction, the microcontroller must support a minimum sample rate of 44,000 Hz - preferably 44,100 Hz in accordance with the CD-DA standard (Wikipedia contributors, 2024).

3.2.3 Quantisation and Bit Depth Effect

Once an analogue signal has been sampled in time, each amplitude value must be approximated to the nearest available digital level - a process known as quantisation (Park, 2009). This step introduces a degree of error, as the continuous amplitude values are mapped to a finite set of discrete levels.

In digital audio, the precision of this mapping is determined by the bit depth - the number of bits used to represent each sample's amplitude. A higher bit depth allows for more discrete levels: for instance, 8-bit audio provides $2^8 = 256$ levels, while 16-bit audio offers $2^{16} = 65,536$ levels (Park, 2009).

The discrepancy between the original analogue amplitude and its quantised digital value is known as quantisation error. This error introduces quantisation noise, which becomes more pronounced at lower bit depths. Figure 7 illustrates this concept, where the orange lines represent the deviation between the analogue waveform and its digital approximation.

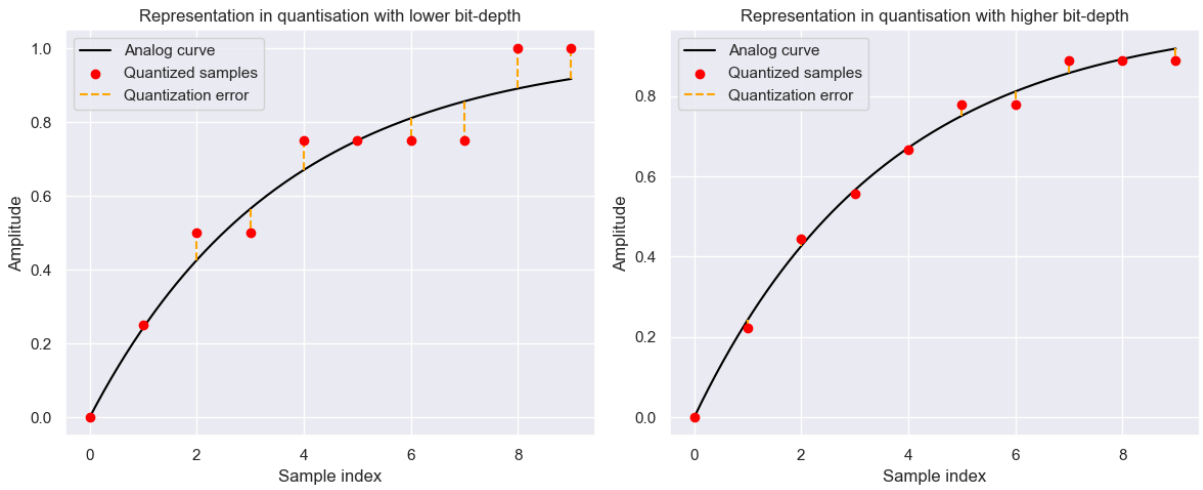

Figure 7: *Analogue and digital representations of a waveform with and without quantisation.*

Higher bit depths reduce quantisation error, resulting in a more accurate representation of the original waveform and improved audio fidelity. This is particularly important in high-quality audio applications, where preserving subtle details is essential (Park, 2009).

Figure 8 compares the spectral representation of a 440 Hz sine wave at two different bit depths. The 16-bit version (left) maintains a clean signal, while the 8-bit version (right) exhibits noticeable noise artifacts due to increased quantisation error.

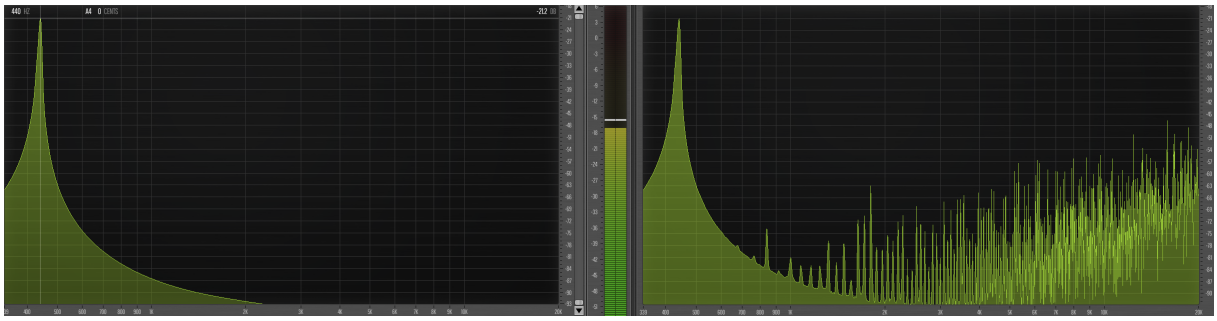

Figure 8: *Comparison of a 440 Hz sine wave at 16-bit (left) and 8-bit (right) quantisation levels. Noise artifacts are evident in the lower bit-depth version. The X-axis represents frequency (Hz), and the Y-axis represents loudness.*

In summary, increased bit depth enhances the resolution of digital audio and reduces quantisation noise. To meet the requirements of high-fidelity sound reproduction, the

microcontroller must support a minimum bit depth of 16 bits, consistent with the Compact Disc Digital Audio (CD-DA) standard (Wikipedia contributors, 2024).

3.2.4 Conclusion

In conclusion, supported sample-rate and high resolution DAC conversion has an effect on microcontroller selection. Because of that, these requirements have been drafted.

- The microcontroller must support an audio processing sample rate of at least 44,100 Hz
- The output bit depth must be a minimum of 16-bit

3.3 Sensor Technologies for Human and Environmental Interaction

This section explores how sensor technologies can be used to detect both environmental and human inputs. It begins by identifying relevant environmental factors and their measurable physical properties (Section 3.3.1), followed by an overview of sensor types commonly used for human interaction in digital musical instruments (Section 3.3.2). The section concludes with a summary of sensor-related requirements (Section 3.3.3), which inform the hardware design discussed in Section 4.

3.3.1 Environmental Sensor Overview

As stated in Subsection 2.2.1, this project focuses on climatic factors. According to Posudin, these include (Posudin, 2014):

- **Temperature**
- **Light**
- **Humidity**
- **Wind**

- **Pressure**
- **Solar Radiation**

These factors can be monitored using a variety of sensors. Table 1 summarises suitable sensor types for each environmental factor, based on descriptions by Fraden and Posudin (Fraden, 2016; Posudin, 2014).

Environmental Factor	**Example Sensors**
Light	Light Dependent Resistor, Photodiode, Phototransistor, UV detector
Pressure	Digital barometric pressure sensor
Wind	Cup anemometer, Windmill anemometer
Humidity	Capacitive humidity sensors, Resistive humidity sensors, Thermal conductivity sensor
Temperature	Silicon PTC temperature sensors, Ceramic thermistors
Radiation	Solid-State Detectors

Table 1: *Sensors for Measuring Environmental Factors (Fraden, 2016; Posudin, 2014)*

In summary, the project will focus on sensors capable of measuring light, pressure, wind, temperature, humidity, and radiation. These sensor types will be considered for integration into the hardware design in Section 4.

3.3.2 Sensor Types in Existing DMIs

This subsection identifies sensor types suitable for detecting human interaction, based on their prevalence in Digital Musical Instruments (DMIs). Medeiros et al. provide a comprehensive list of commonly used sensors in DMI design (Medeiros & Wanderley, 2014):

- Accelerometers

- FSR^{TM} (Force Sensing Resistors^{TM})
- Buttons and potentiometers
- Gyroscopes
- Video/image sensors
- Infrared (IR) sensors
- Magnetometers
- Capacitive sensors
- Biosensors
- Piezoelectric discs
- Non-definable sensors
- Microphones
- Textile-based sensors
- Photo/light sensors
- Bend sensors
- Hall effect sensors
- Ultrasonic sensors
- Pressure/flow sensors
- Fibre optic sensors

These sensors are widely adopted in DMI applications for capturing various forms of human input. Sensor selection for this project will be limited to these categories and further evaluated in Section 4.

3.3.3 Conclusion

This section researched the possibilities of measuring the environment and human stimuli. In addition to helping with the sensor selection, the section also researched commonly used sensors within digital musical instruments.

From the information collected, these sensor requirements can be constructed:

- When monitoring the environment, the sensor used must be one of those listed in Table 1.
- When monitoring human stimuli, the sensor used must be selected from those listed in Section 3.3.2.

3.4 Digital Signal Processing requirements

This section discusses the audio synthesis and alteration techniques that can be implemented using digital signal processing (DSP), along with an overview of latency issues. Section 3.4.1 examines various synthesis techniques, while Section 3.4.2 explores methods for sound alteration. Section 3.4.3 introduces the concept of audio latency and outlines the expected maximum acceptable values, and a concept of buffer size. The section concludes with a summary of DSP-related requirements (Section 3.4.4), which will be considered and evaluated in the software analysis presented in Section 5.

3.4.1 Sound Synthesis Techniques

Sound synthesis refers to the process of generating digital audio signals. Broadly, synthesis methods fall into two categories: algorithmic generation using mathematical models, and playback of pre-calculated waveforms or audio samples (O'Sullivan, 2012). In programming, this is typically implemented either by computing amplitude values in real time or by looping and modifying stored arrays of samples. While both approaches can yield similar auditory results, the choice of method depends on the desired sonic characteristics and computational constraints (Park, 2009).

Several synthesis techniques are commonly used in digital signal processing, each offering distinct levels of control, complexity, and timbral quality (O'Sullivan, 2012; Park, 2009; Zölzer, 2011):

Subtractive Synthesis – Begins with a harmonically rich waveform (e.g., sawtooth or square wave), which is shaped using filters to remove specific frequency components. Filtering techniques are discussed in Section 3.4.2. A sawtooth wave, for example, can be generated using the following equations:

The waveform's amplitude at each sample index n is defined by (MathWorks, 2025):

$$y[n] = A \cdot \left(2 \cdot \left(\frac{\phi[n]}{2\pi}\right) - 1\right) \tag{3}$$

The phase increment per sample is given by:

$$\phi[n+1] = \phi[n] + \frac{2\pi f}{f_s} \tag{4}$$

To ensure the phase remains within a valid range, it is wrapped using:

$$\phi[n] = \phi[n] \bmod 2\pi \tag{5}$$

Equations 3–5 define the amplitude, phase progression, and phase wrapping necessary to generate a continuous sawtooth waveform in discrete time.

Figure 9 illustrates the subtractive synthesis process, where a band-pass filter is applied to a 440 Hz sawtooth wave to remove selected frequencies and shape the final sound.

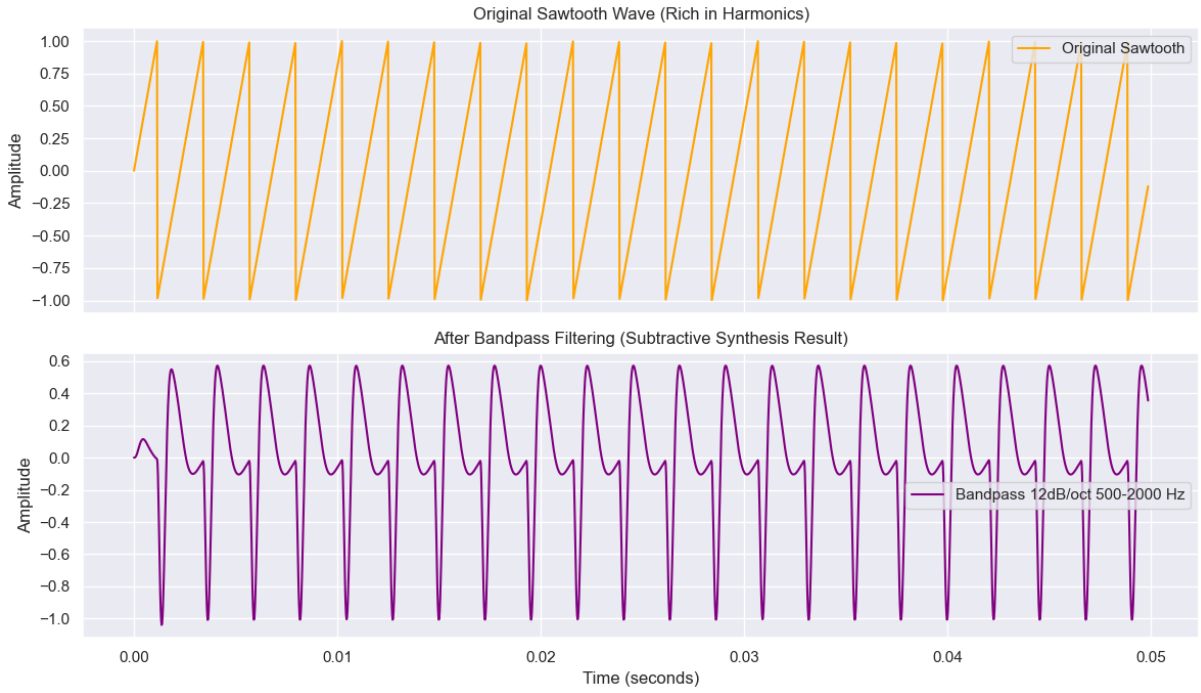

Figure 9: *Subtractive synthesis: a band-pass filter is applied to a 440 Hz sawtooth wave, removing selected frequencies.*

Additive Synthesis – Constructs complex waveforms by summing multiple sine waves, each representing a harmonic component. This is mathematically expressed in Equation 6, where each term contributes a harmonic of the fundamental frequency:

$$y[n] = \frac{2}{\pi} \cdot \sum_{k=1}^{K} \sin\left(\frac{\pi k}{2}\right) \cdot \left(\frac{\sin(2\pi f k n)}{k}\right) \tag{6}$$

Figure 10 demonstrates an additive synthesis with 3 sine waves.

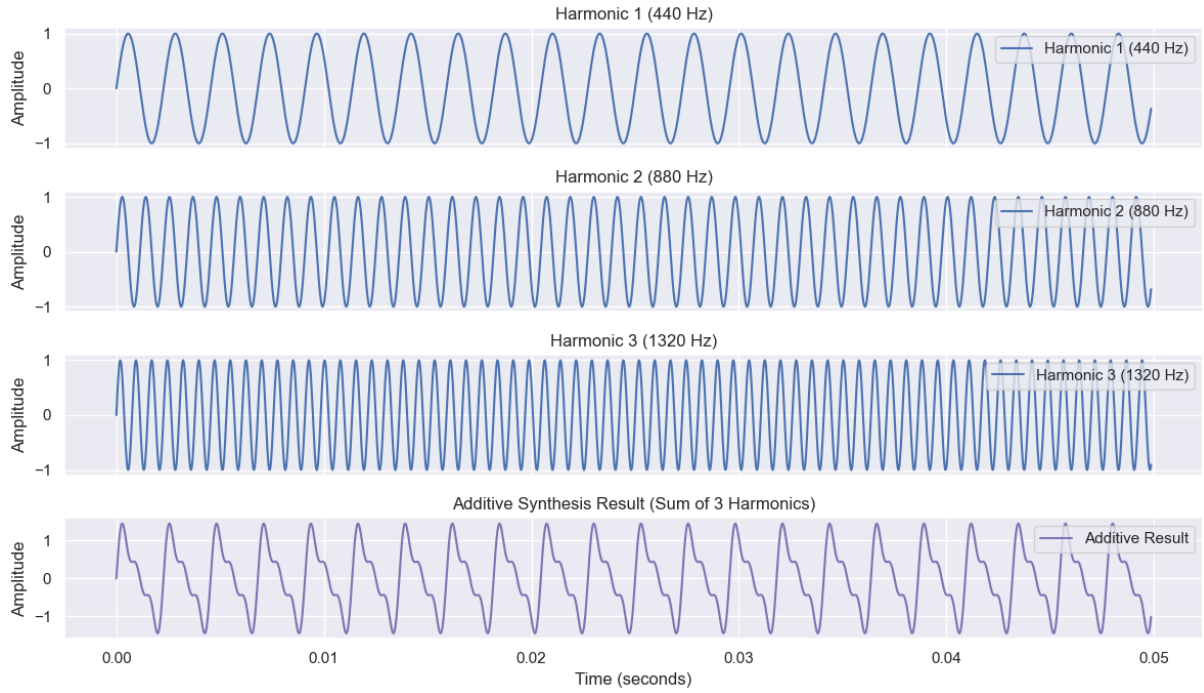

Figure 10: *Additive synthesis: construction of a waveform by summing the first three harmonics of a 440 Hz fundamental.*

Amplitude Modulation (AM) and Frequency Modulation (FM) – Both techniques involve a carrier and a modulator waveform. In FM synthesis, the modulator alters the carrier's frequency, as shown in Equations 7 and 8. In AM synthesis, the modulator affects the carrier's amplitude, as described in Equation 9.

$$y(n) = A_{carrier} \cdot \sin\left(2\pi f_{carrier} \cdot \frac{n}{f_s} + g(n)\right) \tag{7}$$

$$g(n) = A_{mod} \cdot \sin\left(2\pi f_{mod} \cdot \frac{n}{f_s}\right) \tag{8}$$

$$y[n] = \cos\left(2\pi f_{carrier} \cdot \frac{n}{f_s}\right) \cdot \left(A_{mod} \cos\left(2\pi f_{mod} \cdot \frac{n}{f_s}\right) + A_{carrier}\right) \tag{9}$$

In Figure 11, it is possible to see a difference in results when using AM and FM.

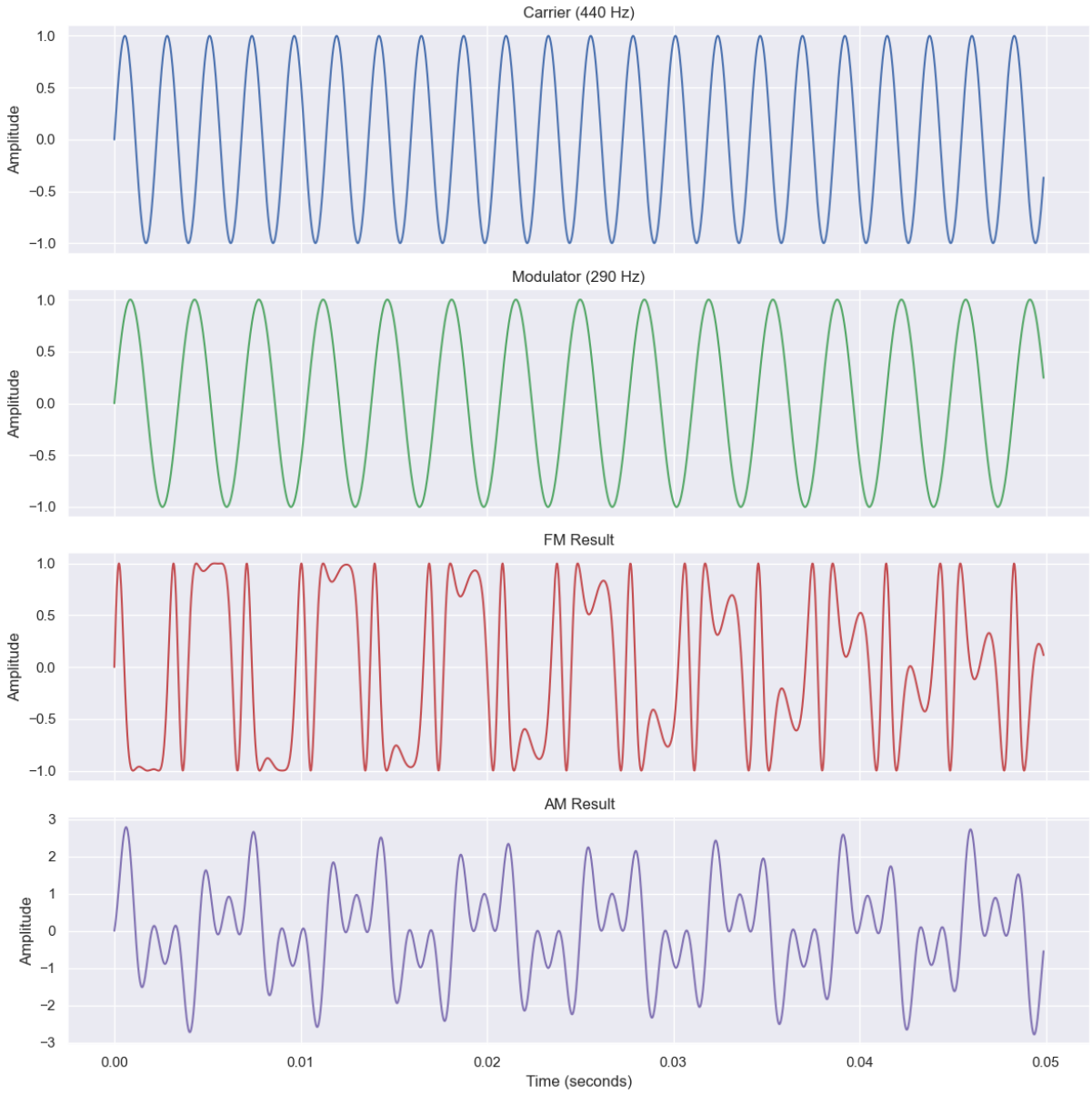

Figure 11: *AM and FM synthesis: comparison of carrier, modulator, and resulting waveforms for a 440 Hz carrier and 290 Hz modulator.*

Granular Synthesis – Constructs sound from small audio fragments called grains. These grains are arranged along the time axis and modulated in amplitude or density to form complex textures. Figure 12 demonstrates one of the possible granular synthesis scenarios.

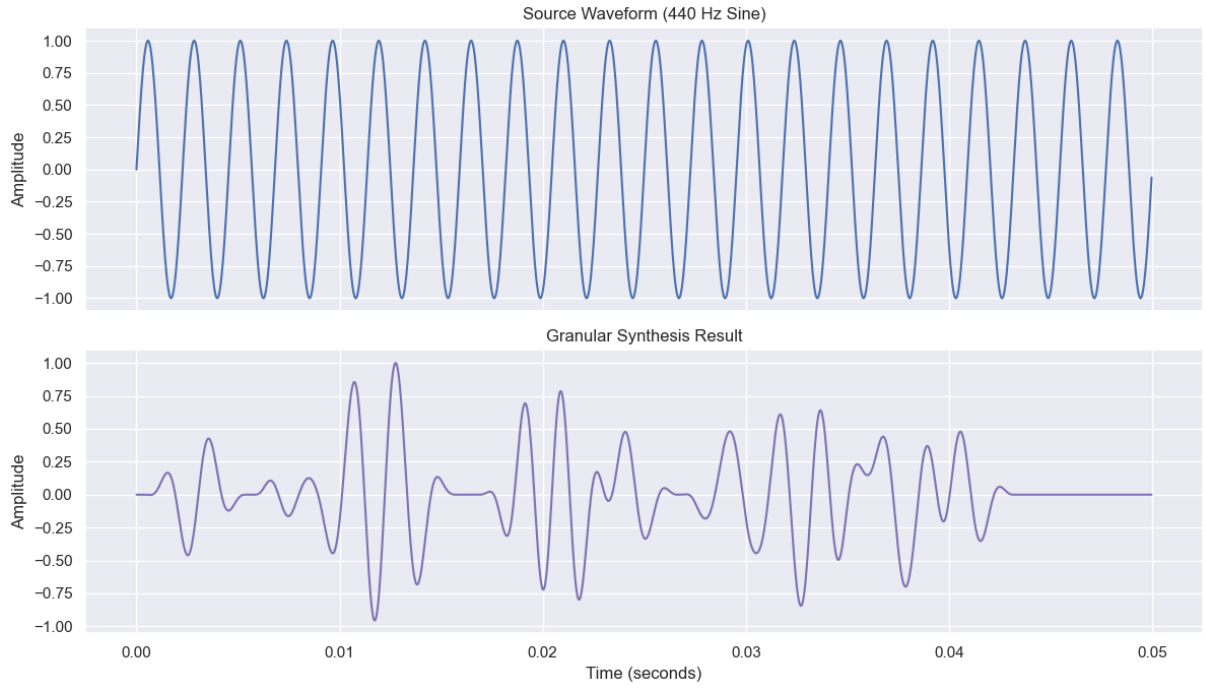

Figure 12: *Granular synthesis: recombination of grains from a 440 Hz sine wave to form a new waveform.*

Wavetable Synthesis – Uses precomputed waveforms stored in a table. During playback, these waveforms are read and modified in real time, offering efficient synthesis with low computational cost.

Physical Modelling – Simulates the physical properties of real-world instruments using mathematical models. This technique enables expressive and realistic sound generation by modelling elements such as string tension, air flow, and material stiffness.

Sampling – Involves recording and storing audio as discrete samples, which can then be manipulated or replayed. While not a synthesis method in the strictest sense, it is often used in hybrid systems.

In summary, these synthesis techniques offer a range of possibilities for generating digital audio. The choice of method will depend on the desired sonic characteristics,

computational efficiency, and responsiveness. The final selection will be discussed in the software design section (Section 5).

3.4.2 Sound Alteration Techniques

Once a sound has been synthesised, it can be further shaped using digital signal processing (DSP) techniques applied in either the frequency or time domain. While a comprehensive overview of all DSP methods is beyond the scope of this project, this section highlights several fundamental techniques that form the basis for more advanced processing (Park, 2009).

Frequency-Based Modifiers

- **Filters** – Filters modify the amplitude and phase of a signal without changing its frequency content. For example, given a sinusoidal signal with amplitude A, frequency f, and phase ϕ, a filter may alter A and ϕ, but not f. Common filter types include low-pass, high-pass, and band-pass filters, each designed to attenuate or preserve specific frequency ranges relative to a cutoff frequency.

  A key characteristic of many filters is *resonance*, which amplifies frequencies near the cutoff point. The steepness of the filter's frequency response - often described in decibels per octave - determines how sharply frequencies outside the desired range are attenuated. Figure 13 demonstrates how different filter settings affect a 440 Hz sawtooth wave.

  In addition to shaping timbre, filters are also essential for mitigating aliasing artifacts introduced during synthesis, particularly when generating harmonically rich waveforms.

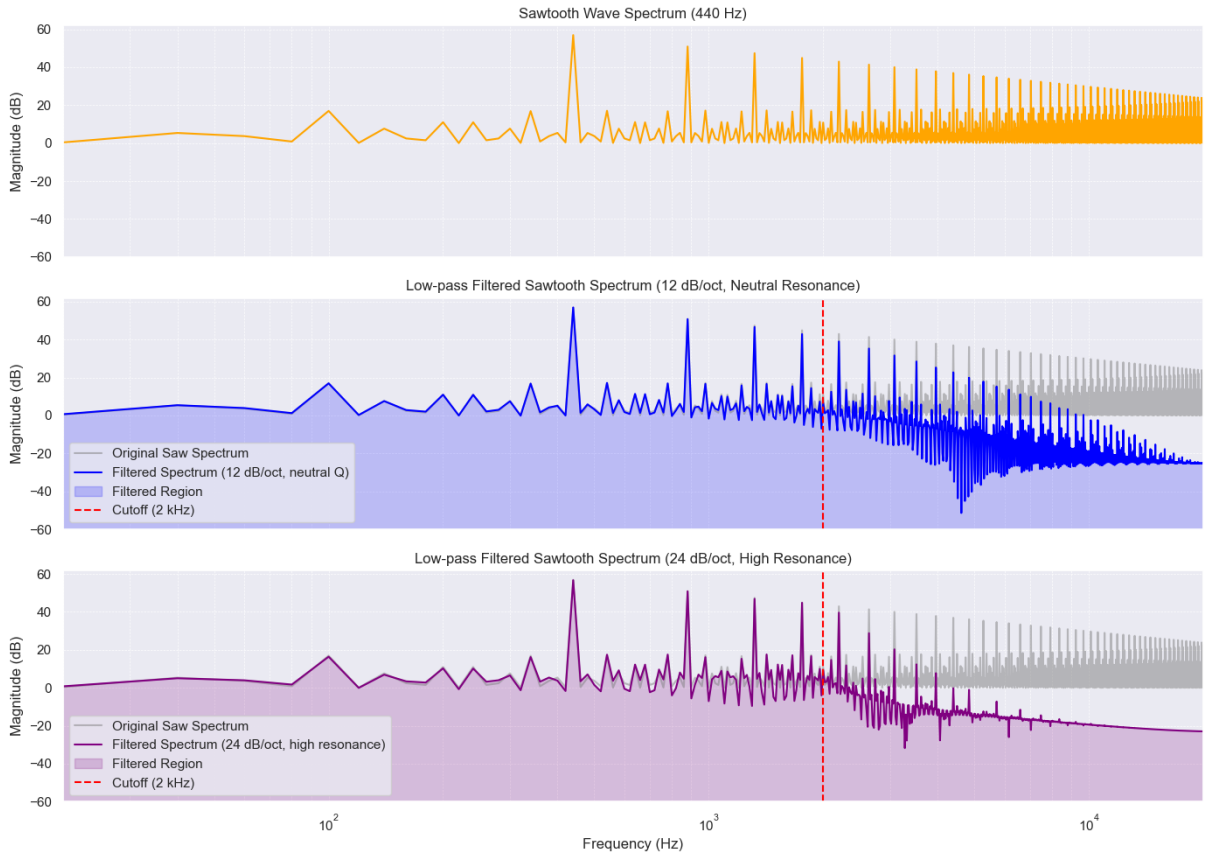

Figure 13: *Subtractive filtering of a 440 Hz sawtooth wave. Top: original waveform. Middle: low-pass filter with low resonance and 12 dB slope. Bottom: low-pass filter with high resonance and 24 dB slope.*

- **Ring Modulation** – This technique multiplies two signals (a carrier and a modulator), producing a signal that contains only the sum and difference frequencies, excluding the original carrier. This differs from amplitude modulation (AM), which retains the carrier component. The ring modulation process is described in Equation 10, where the absence of the constant term (as seen in Equation 9) results in a more metallic or bell-like sound (Park, 2009).

$$y[n] = A_{mod} \cos\left(2\pi f_{mod} \frac{n}{f_s}\right) \cdot \cos\left(2\pi f_{carrier} \frac{n}{f_s}\right) \tag{10}$$

- **Distortion and Clipping** – These occur when a signal's amplitude exceeds the representational limits of the digital system. For instance, in a 16-bit system, values beyond the range $[-1.0, 1.0]$ are clipped to the nearest boundary. This

introduces harmonic distortion and alters the waveform's shape. Figure 14 compares hard and soft clipping. Soft clipping applies a gradual attenuation near the threshold, resulting in less harsh distortion than hard clipping (Park, 2009).

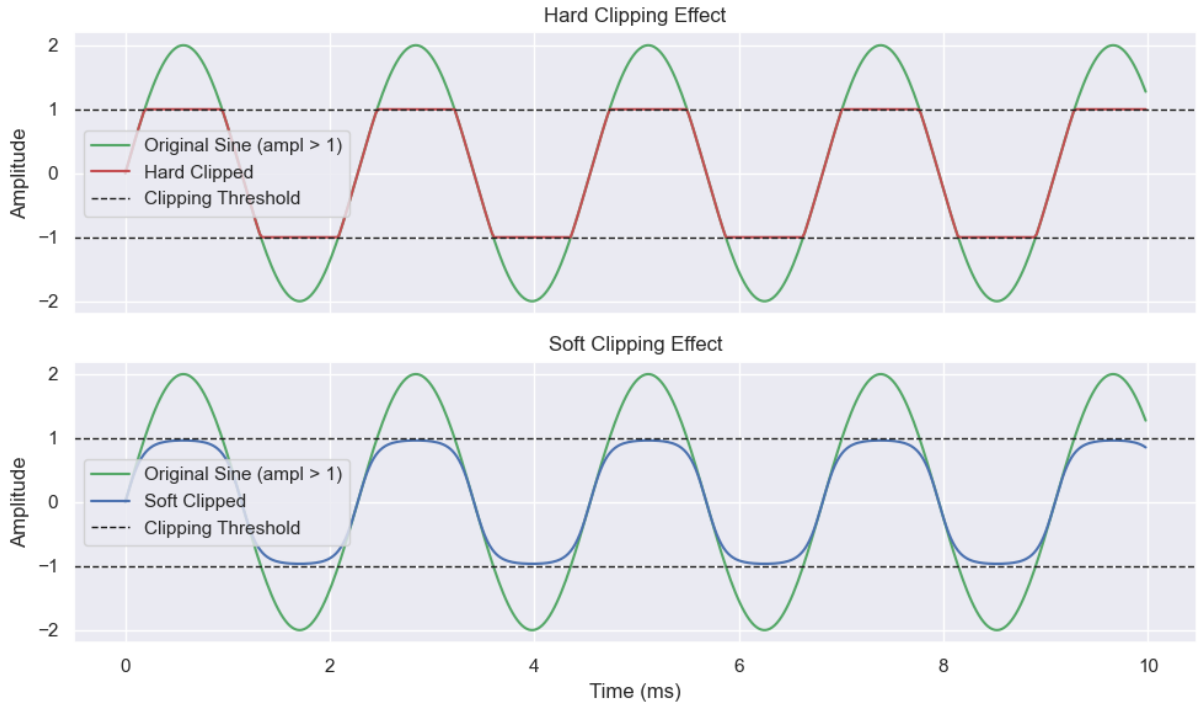

Figure 14: *Hard vs. soft clipping applied to a sine wave exceeding the range* $[-1.0, 1.0]$.

Time-Based Modifiers

- **Low-Frequency Oscillator (LFO)** – An LFO generates a low-frequency waveform (e.g., sine, square, or sawtooth) that is used to modulate parameters such as pitch, amplitude, or filter cutoff. Although not audible itself, the LFO introduces periodic variation, adding movement and expressiveness to the sound (Park, 2009).
- **Delay** – Delay effects introduce a time offset to the original signal and mix it with the unaltered version. This creates echoes, enhances spatial perception, or adds rhythmic complexity. A basic echo effect is described in Equation 11, where N is the delay in samples and b_N is a scaling factor that attenuates the delayed signal (Park, 2009).

$$y[n] = x[n] + b_N \cdot x[n - N] \tag{11}$$

In summary, the techniques described in this section - filtering, modulation, distortion, and delay - are foundational tools in digital audio processing. They will inform the system requirements and be implemented as part of the audio engine described in Section 6.

3.4.3 Audio Latency and Buffer Size

Latency refers to the time delay between initiating an action, such as pressing a key on a digital musical instrument, and hearing the resulting sound. In digital audio systems, this delay arises from several stages: input detection, control signal processing (e.g., MIDI), real-time synthesis and effects computation, and digital-to-analog conversion (Russ, 2009).

A key factor influencing latency is the *buffer size*, which defines the number of audio samples processed at once by the system (Focusrite Support, 2025). In setups involving a computer or digital audio workstation (DAW), buffer size directly affects system responsiveness. Larger buffers reduce CPU load but increase latency, while smaller buffers improve responsiveness at the cost of higher computational demand. Buffer configuration strategies are discussed further in Section 5 (Russ, 2009).

Although modern systems can minimise latency through efficient hardware and software, zero latency is physically unattainable due to inherent processing requirements. Achieving low-latency performance suitable for live interaction requires optimised buffer settings, real-time operating systems or drivers, and streamlined signal paths.

Research suggests that an acceptable upper limit for audio latency in real-time applications is typically between 20 and 30 milliseconds. Latency exceeding this threshold can negatively impact user experience and performance accuracy (Lago & Kon, 2004).

This section establishes a critical system requirement: the total audio processing latency must remain below 20 milliseconds to ensure responsiveness appropriate for live performance and interactive use.

3.4.4 Conclusion

This section has examined various methods for generating and modifying audio signals using digital signal processing (DSP) techniques. It also highlighted the significance of maintaining low audio latency and identified the primary factors contributing to latency in digital audio systems.

Based on the analysis presented, the following functional requirements have been established to guide the development of the DSP component of this project:

- The software must not introduce more than approximately 20 milliseconds of audio latency.
- The system must implement at least one sound synthesis technique, as discussed in Section 3.4.1.
- The system must incorporate at least one sound alteration technique, as outlined in Section 3.4.2.

3.5 Requirements Summary

This section outlines the key functional requirements that were defined to inform and constrain the problem definition.

1. The device should be capable of altering at least one of the sound attributes described in Section 3.1.2.
2. The microcontroller must support an audio processing sample rate of at least 44,100 *Hz*.
3. The output bit depth must be a minimum of 16 bits.
4. When monitoring the environment, the sensor used must be one of those listed in Table 1.
5. When monitoring human stimuli, the sensor used must be selected from those listed in Section 3.3.2.

6. The software must not introduce more than approximately 20 milliseconds of audio latency.

7. The system must implement at least one sound synthesis technique, as discussed in Section 3.4.1.

8. The system must incorporate at least one sound alteration technique, as outlined in Section 3.4.2.

In summary, the outlined requirements define both the functional and non-functional expectations for the system. These specifications form the foundation for the subsequent design and analysis phases discussed in Sections 4 and 5, where hardware and software solutions are developed and evaluated in alignment with the defined objectives. The following sections demonstrate how these requirements have been interpreted and addressed through specific design decisions, while Section 7 evaluates the extent to which the functional requirements have been fulfilled.

4 Hardware Design and Analysis

This section provides a detailed analysis of three key design decisions related to the hardware component of the project. Section 4.1 examines the rationale behind the selection of a suitable microcontroller. Section 4.2 analyses the considerations involved in choosing an appropriate environmental sensor. Section 4.3 discusses the selection process for a human stimulus sensor. These design decisions will subsequently be implemented in the implementation phase, as outlined in Section 6.

4.1 Microcontroller Selection

One of the most critical design decisions in this project involves the selection of a suitable microcontroller. This component is essential, as it will be responsible for executing all core logic and handling the digital signal processing tasks required by the system.

A wide range of microcontrollers is available on the market; however, given the project's open-source philosophy and its emphasis on digital audio capabilities, the selection is constrained to devices that inherently support these requirements. Furthermore, as this is an educational project with limited financial resources, cost-effectiveness is a critical consideration - the selected microcontroller must be affordable while still fulfilling the necessary technical specifications.

As established in Section 3.2, the microcontroller must support an audio output with a minimum sample rate of 44.1 kHz and a bit depth of at least 16 bits.

Based on these criteria and a review of available open-source microcontrollers, three viable candidates were identified:

- **Daisy Seed** (Electrosmith, 2025)
- **Teensy 4.1** (PJRC, 2025)
- **ESP32** (Espressif Systems, 2025)

4.1.1 Technical Specifications and Comparison

The comparison of these microcontrollers can be found in the Table 2:

Feature / Criterion	Daisy Seed	Teensy 4.1	ESP32
CPU	ARM Cortex-M7 @ 480 MHz	ARM Cortex-M7 @ 600 MHz	Dual-core Xtensa @ 240 MHz
RAM	64MB SDRAM	1MB RAM	~512KB RAM
ADC channels	x12 ADC inputs (16-bit)	x18 ADC inputs (10-bit)	x18 ADC inputs (12-bit)
GPIO pins	31	55	34
Output Voltage	$3.3V$	$3.3V$	$3.3V$
Input Voltage	$3.3V$, GPIO pins support $5V$	$3.3V$	$3.3V$
Audio Codec	PCM3060	Requires external shield	I2S only; external codec required
Audio Fidelity	Professional-grade (24-bit/96kHz)	Good with shield	Basic (8-bit DAC or I2S only)
Audio Library	DaisySP	Teensy Audio Library	community based libraries
Hardware Design	Open-source	Closed bootloader	Open-source
Development Tools	Arduino, C++, Max/MSP, Pure Data	Arduino, closed loader	Arduino, ESP-IDF
Approximate Price in Euros	21	28 (+13 shield)	8

Table 2: *Combined Comparison of Microcontroller Hardware, Development Tools, and Cost (Electrosmith, 2025; Espressif Systems, 2025; PJRC, 2025)*

When comparing the information compiled in Table 2, the following advantages and disadvantages can be identified.

Daisy Seed

Advantages

- **High RAM Capacity:** With 64MB of SDRAM, Daisy Seed offers significantly more memory than the others, making it ideal for audio and signal processing

tasks.

- **Professional Audio Fidelity:** It supports 24-bit audio at 96kHz, offering professional-grade sound, aided by the onboard PCM3060 codec. Comes with the DaisySP audio library, which is optimised for the device
- **16-bit ADC:** Supports high-resolution ADC pins, which can be beneficial in more precise control
- **Flexible Development Tools:** Supports a broad range including Arduino, C++, Max/MSP, and Pure Data.

Disadvantages

- **Lower CPU Clock Speed:** Though powerful, its 480 MHz CPU is slower than the Teensy 4.1's 600 MHz.
- **Higher Cost:** At approximately €21, it's more expensive than the ESP32.
- **Limited ADC Channels:** Offers only 12 ADC inputs compared to 18 on the other two boards.

Teensy 4.1

Advantages

- **Fastest CPU:** Powered by an ARM Cortex-M7 at 600 MHz, it has the highest processing speed among the three.
- **Wide ADC Coverage:** Provides 18 ADC inputs, useful for sensor-rich projects.
- **Extensive I/O Options:** Offers 55 I/O pins, the highest among the three.
- **Good Audio Support (with Shield):** Can achieve good audio performance using the Teensy Audio Library and an external shield.
- **Versatile Development:** Uses Arduino IDE and the popular Teensy Loader for programming.

Disadvantages

- **Closed Bootloader:** Unlike the Daisy and ESP32, Teensy does not have an open bootloader, which limits low-level customizations.
- **No Onboard Codec:** Requires an external shield for audio input/output, adding to complexity and cost.
- **Highest Total Cost:** Priced at €28 with the shield, it's the most expensive option when full audio capabilities are needed.
- **10-bit ADC:** Supports only 10-bit resolution for analogue input reading.

ESP32

Advantages

- **Low Cost:** At approximately €8, ESP32 is the most budget-friendly of the three.
- **Dual-Core Processor:** Offers efficient multitasking with two Xtensa cores at 240 MHz.
- **Decent ADC and GPIO Coverage:** Provides 18 ADC channels and up to 34 GPIOs, making it versatile for general-purpose applications.
- **Open-source & Flexible Tools:** Fully open-source and supports Arduino and ESP-IDF for both beginner and advanced users.

Disadvantages

- **Limited RAM:** Only $\sim$512KB, significantly less than the other options, limiting complex or memory-heavy applications.
- **Basic Audio Capabilities:** Requires external I2S codec for audio and only supports 8-bit DAC or I2S, resulting in lower fidelity.
- **Lower Audio Performance:** Not suitable for professional audio applications without additional hardware.

In the context of this audio project, the Daisy Seed and Teensy 4.1 emerge as more suitable candidates due to their superior audio capabilities and availability of dedicated libraries. Benchmark studies by Thomas Lorensen indicate that ARM Cortex-M7 processors, such as those used in these microcontrollers, lead in real-time audio processing performance (Arm Ltd., 2025; Lorenser, 2016). Conversely, the dual-core Xtensa architecture found in the ESP32 offers efficient multitasking capabilities. Regarding analogue-to-digital converter (ADC) pins, the Daisy Seed supports fewer inputs but provides 16-bit resolution, allowing for more precise audio control. From a budgetary perspective, as outlined in Section 2, the ESP32 is a cost-effective option, leaving more funds available for additional hardware components. In contrast, selecting the Teensy 4.1 would reduce the remaining budget to approximately 9 euros for other parts.

4.1.2 Partial Conclusion

After evaluating the candidates, the decision was made to proceed with the **Daisy Seed** by Electrosmith. This microcontroller offers the strongest support for audio processing, which is the central focus of this project. It provides superior analogue resolution and features an ARM Cortex-M7 CPU with built-in support for 24-bit/96 kHz audio output. Additionally, it is fully open-source and is not the most expensive option among the candidates. Although multitasking capabilities are limited due to its single-core architecture, and its clock speed is lower than that of the Teensy 4.1, the Daisy Seed remains a suitable choice. However, it offers the fewest ADC pins, restricting the number of analogue inputs to twelve.

The implementation and utilization of the Daisy Seed will be detailed in Section 6.

4.2 Environmental Sensor Selection

Another key design decision involved selecting suitable sensors for measuring the environmental climatic factors. As outlined in Section 3.3, six distinct environmental parameters can be measured, and the section also presents various sensor options for capturing these parameters. To satisfy the project requirements, at least one of these sensors must be utilised. This section discusses the rationale and considerations behind the final sensor selection.

After reviewing the available components at the RUC Fablab, the following candidates were selected for environmental sensing. These sensors enable the monitoring of light, temperature, and humidity:

- Light Dependent Resistor (LDR)
- Ceramic Thermistor
- Resistive Humidity Sensor

4.2.1 Comparison of Sensors

After acquiring the candidate sensors in physical form, it proved challenging to identify the exact model numbers and locate the corresponding data sheets. As a result, conducting a detailed technical comparison was not possible. Instead, a general evaluation of advantages and disadvantages was conducted based on available resources, discussing sensor types rather than specific models.

Light Dependent Resistor

Advantages (World, 2023)

- **High Sensitivity to Light:** Accurately detects small changes in light, ideal for precise measurement applications.
- **Simplicity and Compact Design:** Easy to use and integrate due to straightforward, compact construction.
- **Cost-Effectiveness:** Inexpensive and widely available, suitable for both hobby and professional use.
- **High Light-Dark Resistance Ratio:** Provides a clear distinction between light and dark, useful for reliable detection.
- **Ease of Integration:** Simple to add to circuits and compatible with common microcontrollers.
- **Good Linearity:** Resistance changes proportionally with light, beneficial for audio and control applications.

Disadvantages (World, 2023)

- **Limited Spectral Response:** Only detects a narrow range of light wavelengths, reducing versatility.
- **Temperature Stability Issues:** Performance is affected by temperature changes, causing inaccurate readings.
- **Slow Response Time:** Reacts slowly to changes in light, unsuitable for fast detection needs.
- **Environmental Susceptibility:** Sensitive to temperature and humidity, leading to less accurate detection.
- **Poor Linearity Under Strong Illumination:** Becomes non-linear in bright light, reducing measurement accuracy.
- **Limited Sensing Capabilities:** Mainly detects light intensity, so accuracy drops with changing ambient light.
- **Hysteresis Effect:** Sensor response depends on previous light conditions, causing inconsistent readings.

Ceramic Thermistor

Advantages (Thorat, 2023)

- **More accurate output:** Delivers precise measurement results.
- **Suitable for remote locations:** Can be used effectively in distant or inaccessible areas.
- **Flexible manufacturing:** Can be made in various shapes and sizes.
- **High accuracy:** Ensures a high degree of measurement precision.
- **Good stability and repeatability:** Provides consistent and reliable performance over time.
- **Withstands stresses:** Resistant to mechanical and electrical stresses.

Disadvantages (Thorat, 2023)

- **Highly non-linear behavior:** Output does not change proportionally with input, complicating measurement.
- **Limited measuring range:** Can only accurately measure within a specific range.
- **Self-heating:** May heat up during operation, affecting accuracy.
- **Fragile:** Easily damaged due to delicate construction.

Resistive Humidity Sensor

Advantages (Electricity and Magnetism, 2025)

- **Cost-effectiveness:** Affordable.
- **Simplicity:** Simple design and operation, easy to integrate into systems.
- **Versatility:** Available in various configurations for different applications.

Disadvantages (Electricity and Magnetism, 2025)

- **Temperature dependence:** Performance can vary with temperature, requiring compensation for accuracy.
- **Long-term drift:** May need periodic recalibration due to gradual changes over time.
- **Sensitivity to contaminants:** Dust, chemicals, or oils can affect sensor reliability and performance.

In the context of an outdoor music installation, selecting an appropriate environmental sensor involves balancing responsiveness, robustness, ease of integration, and expressive potential. Among the available options—such as thermistors for temperature or resistive sensors for humidity—the Light Dependent Resistor (LDR) presents a particularly compelling candidate.

LDRs offer simplicity, low cost, and straightforward integration with analog microcontroller inputs. Their sensitivity to ambient light makes them ideal for environments with dynamic lighting—such as transitions between day and night or variations caused by audience interaction and stage lighting. These changes can be effectively mapped to modulation parameters in sound synthesis, adding an expressive, interactive layer to the installation.

However, LDRs also have notable drawbacks. Their relatively slow response time limits their use in fast-changing scenarios, and they are affected by temperature and humidity, which may compromise accuracy.

In contrast, thermistors and humidity sensors provide more stable and quantifiable data but tend to change slowly and predictably—making them less suitable for real-time sound interaction.

4.2.2 Partial Conclusion

Considering the trade-offs, the **Light Dependent Resistor (LDR)** was selected for this project. While not the most accurate or stable sensor, its responsiveness to ambient light and ease of integration align well with the artistic goals of the installation. Its limitations are acceptable given the focus on gradual, perceptible variation rather than precision, and its capacity to drive expressive modulation makes it well-suited for outdoor interactive sound applications.

The implementation and integration of the LDR will be detailed in Section 6.

4.3 Human Sensor Selection

Another key design decision involves selecting a suitable sensor for human interaction detection. As defined in Section 3.3, the range of possible sensors was narrowed to those commonly used in Digital Musical Instrument (DMI) designs. Furthermore, the availability of components at the RUC Fablab was considered to ensure practical feasibility.

Taking these constraints into account, the following candidates were selected for human interaction sensing:

- Potentiometer
- Switch button
- Hall Effect Sensor
- Infrared Sensor

4.3.1 Comparison of Candidates

Similar to the process described for environmental sensors in Section 4.2, it was difficult to obtain datasheets for all candidate human sensors. As a result, the comparison presented here is based on general characteristics and commonly documented performance traits, rather than specific technical specifications.

Potentiometer

Advantages(Fraden, 2016; Saini et al., 2018)

- **Simple and cost-effective:** Easy to use and affordable for most applications.
- **High resolution:** Provides fine control over position or resistance.
- **Analogue output:** Delivers a continuous range of values, useful for precise adjustments.

Disadvantages(Fraden, 2016; Saini et al., 2018)

- **Mechanical wear:** Moving parts are subject to wear and tear over time.
- **Requires physical contact:** Needs direct interaction, which can limit durability.
- **Noise filtering required:** Analogue signals may require additional filtering to reduce noise.

Switch Button

Advantages(Fraden, 2016; Saini et al., 2018)

- **Simple operation:** Easy to use

- **Reliable control:** Provides clear and direct user control over a device.
- **Low cost:** Inexpensive and widely available in various forms.
- **Long lifespan:** Mechanical switches can last for thousands of cycles if used properly.

Disadvantages(Fraden, 2016; Saini et al., 2018)

- **Mechanical wear:** Moving parts can degrade over time, leading to failure.
- **Contact bounce:** May produce multiple signals when toggled, requiring debouncing in digital circuits.
- **Binary output:** Only provides on/off states, unsuitable for applications needing variable or analogue control.

Hall Effect Sensor

Advantages(Fraden, 2016; Saini et al., 2018)

- **Contactless:** Measures position or speed without physical contact, reducing wear.
- **Durable:** Robust design suitable for long-term use.
- **Magnetic field-based:** Can detect magnetic fields, enabling unique sensing applications.

Disadvantages(Fraden, 2016; Saini et al., 2018)

- **Requires magnets:** Needs a magnetic source for operation.
- **Limited range:** Effective only within a certain distance from the magnet.
- **Magnetic interference:** Susceptible to errors from external magnetic fields.

Infra-Red Sensor

Advantages(Fraden, 2016; Saini et al., 2018)

- **Contactless:** Detects objects or gestures without physical contact.

- **Gesture recognition:** Capable of identifying hand or object movements.
- **Low-power consumption:** Operates efficiently, suitable for battery-powered devices.

Disadvantages(Fraden, 2016; Saini et al., 2018)

- **Affected by ambient light:** Performance can degrade in bright environments.
- **Limited resolution:** May not detect fine details or small movements accurately.
- **Line-of-sight required:** Needs a clear path between sensor and target for reliable operation.

For the purpose of human detection in an interactive outdoor music installation, the selected sensor must prioritize high resolution, adequate range, and resilience to environmental conditions. Some mechanical wear is acceptable, provided the components are inexpensive and easily replaceable.

Among the options considered - potentiometers, switch buttons, Hall effect sensors, and infrared (IR) sensors - each presents distinct advantages and limitations. Potentiometers offer excellent resolution and analog output, making them ideal for expressive control. However, their reliance on direct physical contact and susceptibility to mechanical wear can be drawbacks in public outdoor settings where durability is essential.

Switch buttons are robust and reliable for simple triggering tasks, but their binary nature (on/off) limits their expressive potential, making them less suitable for dynamic audio modulation.

Hall effect sensors provide contactless operation and good durability, offering a balance between precision and wear resistance. However, they require a magnet for operation, which complicates integration, and their limited range may restrict user interaction in open environments.

Infrared sensors are attractive for their contactless gesture recognition and low power consumption. Yet, they are highly sensitive to ambient light and require a clear line of sight, both of which can compromise reliability in outdoor festival conditions.

4.3.2 Partial Conclusion

For this project, **potentiometers** were ultimately chosen for human interaction detection due to their high resolution, analog output, and intuitive physical interface. While mechanical wear and the need for physical contact are valid concerns, these are mitigated by the low cost and easy replaceability of the components. In a temporary festival setting, occasional wear is acceptable, especially when weighed against the precision and tactile feedback that potentiometers provide.

Compared to Hall effect sensors, which offer contactless operation and better long-term durability, potentiometers were preferred for their simpler integration. Hall sensors require precise magnet alignment and are susceptible to magnetic interference, which could be problematic in a crowded, unpredictable outdoor environment. Infrared sensors, while promising in theory, are too vulnerable to ambient light and line-of-sight issues for reliable use in this context.

Therefore, despite some limitations, potentiometers offer the best balance of expressivity, control, and practical integration for this musical installation.

The implementation and utilization of the potentiometers will be detailed in Section 6.

4.4 Conclusion

In conclusion, the hardware component of the device will consist of the following selected elements:

- **Daisy Seed** – chosen for its superior audio processing capabilities.
- **Light Dependent Resistor (LDR)** – selected for measuring ambient light levels.
- **Potentiometer** – used as a human-interaction sensor for real-time control input.

These hardware decisions directly support functional requirements 2, 3, 4, and 5 outlined in Section 3.5, ensuring alignment between system design and project objectives. Section 6 will then show how these decisions were implemented.

5 Software Design and Analysis

This section provides a detailed analysis of three key software design decisions undertaken in this project. Section 5.1 examines the rationale behind the mapping strategies used to translate sensor readings into sound attributes. Section 5.2 analyses the considerations involved in choosing between sample-by-sample processing and block processing. Section 5.3 discusses the selection of sound synthesis methods appropriate for the project. These design decisions will be implemented and evaluated during the implementation phase, as described in Section 6.

5.1 Mapping Strategy Selection

A critical design decision involved selecting the appropriate mapping strategy for translating sensor data into sound attributes. According to the classifications proposed by Dobrian and Koppelman, 2006, several mapping strategies are commonly employed in digital musical instruments to relate incoming sensor inputs to outgoing musical parameters.

The following mapping strategies were identified for consideration in this project:

Simple mapping entails a direct one-to-one correspondence between a control input and a single sound parameter. This means that each action by the performer affects one specific aspect of the sound in a straightforward manner (Dobrian & Koppelman, 2006). Figure 15 illustrates an example where a potentiometer controls the pitch of the sound.

Figure 15: *Simple Mapping Example: Potentiometer controlling pitch*

Complex mapping involves more intricate relationships between input and output, where a single sensor can influence multiple sound parameters simultaneously (Dobrian & Koppelman, 2006). Figure 16 demonstrates an example in which a potentiometer affects both the pitch and volume of the generated sound.

Figure 16: *Complex Mapping Example of potentiometer controlling both pitch and volume of sound*

Gesture mapping (or gesture recognition) involves interpreting patterns or characteristics of a performer's movements to infer their intent and translate this into sound parameters. For instance, a specific movement of the potentiometer could be recognised as a predefined gesture, triggering an effect such as temporarily multiplying the pitch (Dobrian & Koppelman, 2006). Figure 17 provides a visual representation of a gesture recognition mapping.

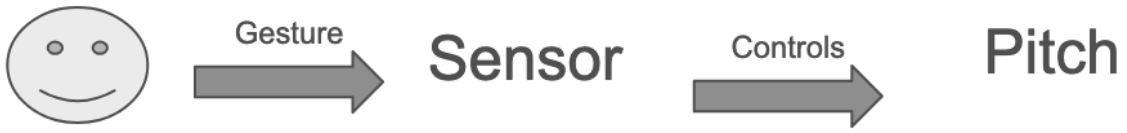

Figure 17: *Gesture Recognition Mapping*

5.1.1 Comparison of Mappings

Simple mapping

Advantages

- **Intuitive:** Easy for users to understand and operate, making it accessible for beginners (Dobrian & Koppelman, 2006).
- **Steep learning curve:** Users can quickly become proficient due to the straightforward nature of the mapping (Dobrian & Koppelman, 2006).

Disadvantages

- **Limited expressive depth:** May not offer enough flexibility or nuance for advanced expression (Dobrian & Koppelman, 2006).

Complex mapping

Advantages

- **Rich expressive potential:** Enables more dynamic and nuanced interactions, allowing for greater depth in performance (Dobrian & Koppelman, 2006).

Disadvantages

- **Less intuitive:** Can be harder to learn and operate, especially for new users (Dobrian & Koppelman, 2006).
- **Increased system complexity:** More challenging to implement and maintain due to the intricate mapping structure.

Gesture mapping

Advantages

- **Highly intuitive and responsive:** Gesture recognition makes the interface natural and enhances expressive control (Dobrian & Koppelman, 2006).

Disadvantages

- **Requires calibration:** Needs setup and adjustment to function accurately for different users.
- **Complex implementation:** Developing reliable gesture recognition can be technically demanding.

5.1.2 Partial Conclusion

In the context of designing a digital synthesiser intended for public interaction at a music festival (Section 2.1), intuitiveness and accessibility for beginners were identified as critical design priorities. Given the project's objective to engage a broad audience, including individuals with little or no experience in music technology, a **simple**

mapping strategy was selected. This approach establishes a direct, one-to-one relationship between sensor inputs and sound parameters, promoting a shallow learning curve that enables users to quickly understand and enjoy the instrument with minimal instruction.

Although complex mappings offer richer expressive potential by allowing a single input to control multiple sound characteristics, they are inherently less intuitive. This increased complexity could lead to user confusion or disengagement in a fast-paced, unsupervised festival setting. Gesture mapping, while highly expressive and responsive, often requires calibration and more sophisticated implementation. These additional requirements introduce potential technical instability and variability between users, which may compromise the reliability of the experience in a public installation.

Therefore, the simple mapping strategy was chosen not only for its ease of use but also for its robustness, clarity, and suitability in interactive public contexts. It ensures that even first-time users can achieve meaningful and rewarding sonic interactions confidently and immediately. The specific implementation of sensor-to-parameter mappings will be discussed in greater detail in Section 6.

5.2 Sample-By-Sample or Block Processing?

Section 3.4.3 briefly introduced the concept of buffer size and its impact on audio latency and CPU load. Building upon that foundation, a key software design decision in this project involves selecting between sample-by-sample processing and block-based processing for handling digital audio signals. This choice determines whether the system processes each audio sample individually in real-time or in groups (blocks) of a defined size.

In block-based processing, one buffer of audio samples is played back while a second buffer is simultaneously filled with new data, ready to be played next. This double-buffering technique ensures continuous audio playback but introduces a trade-off between latency and computational efficiency (Kuo et al., 2013). Each approach carries implications for latency, processing performance, and implementation complexity, which are examined in this section.

In digital signal processing, buffer size is typically defined as a power of two, starting

from $2^5 = 32$ samples and increasing incrementally (e.g., 64, 128, 256) (Sweetwater Sound, 2022). According to the official documentation, the Daisy Seed microcontroller supports buffer sizes up to 256 samples (Electro-Smith, 2025b).

Based on these constraints, the following buffer configurations were considered:

- Sample-by-sample processing (buffer size of 1 sample)
- Block processing with buffer sizes of 32, 64, 128, and 256 samples

5.2.1 Comparison of Options

Equation 12 defines the latency incurred by each buffer size, based on the sample rate (Calculator Academy Team, 2024). The resulting latencies are illustrated in Figure 18.

$$\text{Latency} = \frac{\text{Buffer Size}}{\text{Sample Rate}} \tag{12}$$

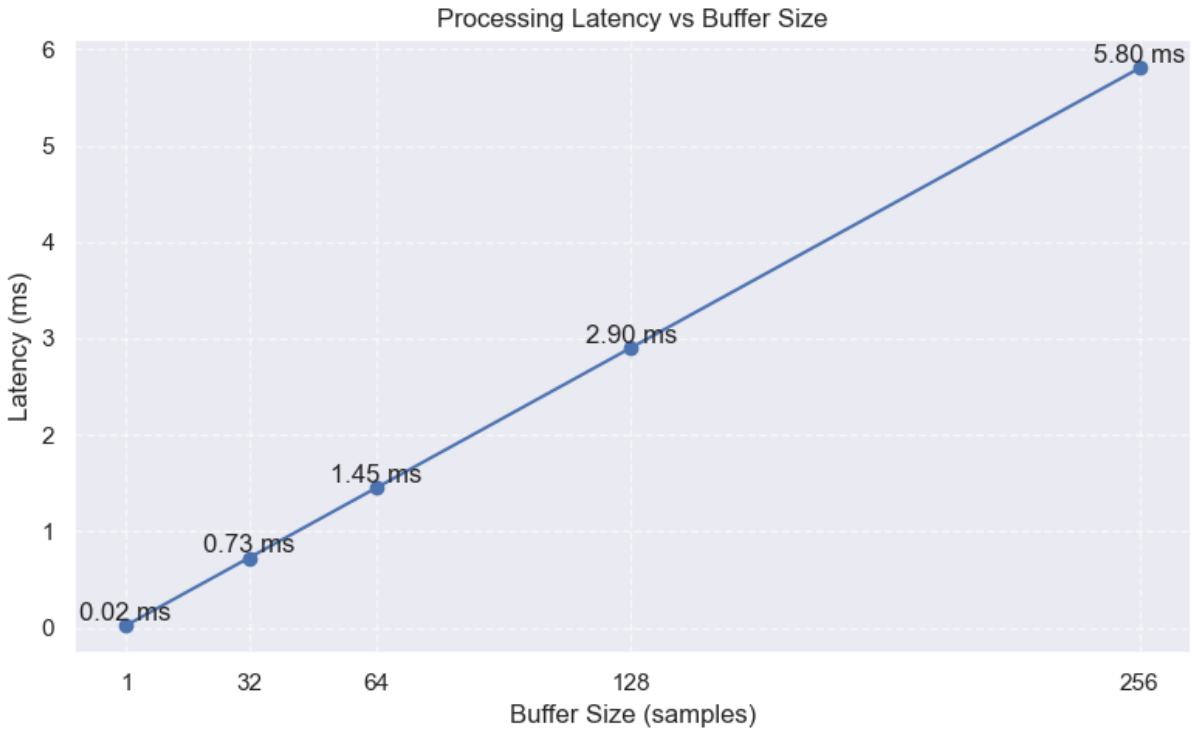

Figure 18: *Calculated latency for various buffer sizes at a sample rate of 44,100 Hz*

As shown in Figure 18, latency increases linearly with buffer size. Referring to the specifications in Section 4, the Daisy Seed's CPU runs at 480 MHz. To estimate the CPU load, it is assumed that each sample requires approximately 1,000 CPU cycles for processing.

Additionally, a fixed overhead of 5,000 cycles per buffer is included, accounting for tasks such as memory management, buffer setup, and interrupt handling. This overhead has a more significant impact on smaller buffer sizes since it is incurred more frequently.

These estimates are based on typical embedded DSP workloads and are consistent with benchmarks from the Daisy developer community using ARM Cortex-M7 microcontrollers, such as the STM32H750 (Electro-Smith, 2025a; Electro-Smith Community, 2023).

To estimate CPU usage, the following formula is used, adapted from real-time DSP literature (Kuo et al., 2013):

$$\text{CPU Load} = \frac{C \cdot N + C_{\text{overhead}}}{T_{\text{buffer}} \cdot f_{\text{CPU}}} \times 100 \tag{13}$$

Where:

- C is the number of CPU cycles required per sample (assumed to be 1000),
- C_{overhead} is the fixed overhead per buffer in cycles (assumed to be 5,000),
- N is the buffer size in samples,
- $T_{\text{buffer}} = \frac{N}{f_s}$ is the buffer duration in seconds (identical to Equation 12),
- f_s is the sample rate (44,100 Hz),
- $f_{\text{CPU}} = 480 \times 10^6$ is the CPU clock frequency in Hz.

The calculated CPU loads are presented in Figure 19.

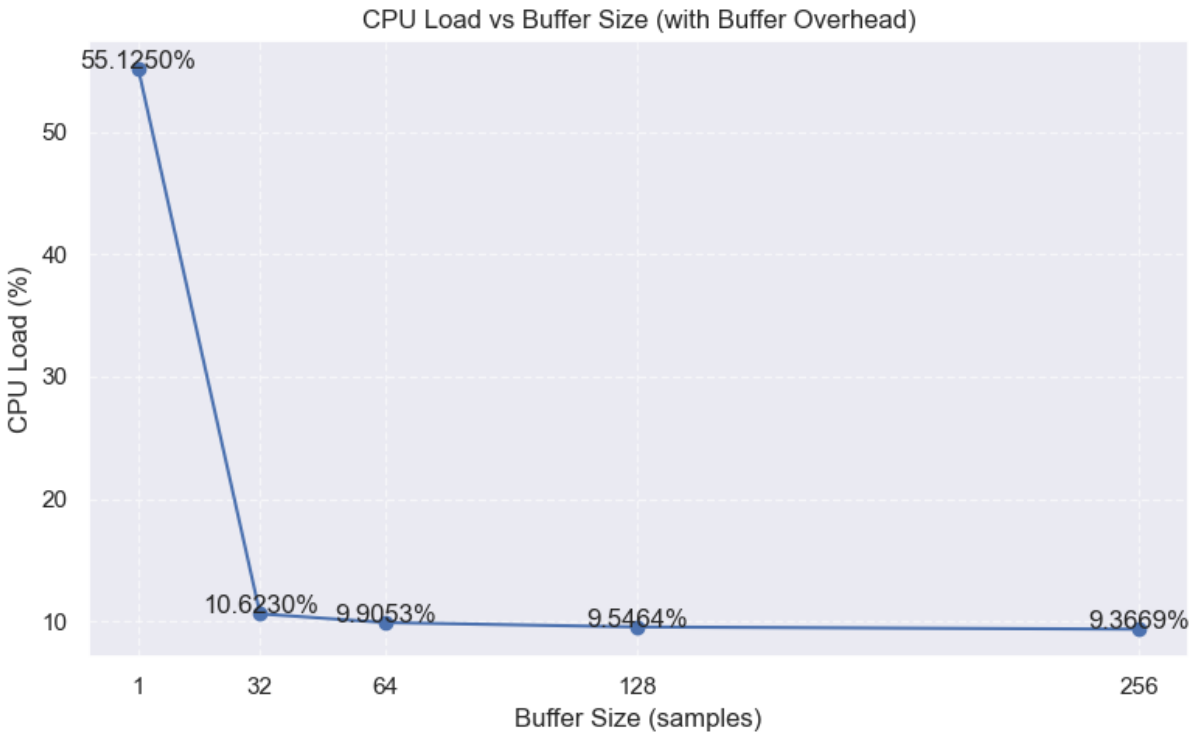

Figure 19: *Estimated CPU load for various buffer sizes*

As shown in Figure 19, CPU load decreases with increasing buffer size, thereby providing more computational headroom for additional processing. However, this improvement comes at the cost of higher latency, as depicted in Figure 18.

5.2.2 Partial Conclusion

Section 3.5 defines an acceptable latency threshold of 20 ms. As observed in Figure 18, even the maximum buffer size considered (256 samples) results in latency well below this threshold. Thus, higher buffer sizes can be used without negatively affecting responsiveness.

Given these findings, block-based processing was selected for this project, with an initial buffer size of 32 samples. This configuration provides a balanced compromise between achieving low audio latency and maintaining a manageable CPU load. The implementation of the buffer size will be discussed in Section 6, where adjustments

may be made to increase the buffer size if additional computational headroom is required during development.

5.3 Synthesis Technique Selection

Selecting an appropriate synthesis method was a critical software design decision that directly influenced both the expressive potential and computational feasibility of the instrument. Given the project's context - an interactive, stimulus-responsive installation at a public music festival (see Section 2.1 and 2.2) - the chosen technique needed to strike a balance between sonic richness, real-time responsiveness, and ease of implementation on embedded hardware. Each synthesis method outlined in Section 3.4.1 was evaluated with these criteria in mind.

Subtractive synthesis was identified as highly suitable due to its moderate computational requirements and sonic flexibility. By shaping harmonically rich waveforms using filters, subtractive synthesis enables a wide range of timbres with relatively simple operations (Zölzer, 2011). This method aligns well with real-time control mapping and benefits from low processing overhead, particularly when combined with block-based processing strategies (Section 5.2) (Park, 2009).

Additive synthesis, while conceptually straightforward, involves summing multiple sine waves to recreate complex tones. Although it allows precise control over harmonic content, this approach can be computationally expensive, especially on embedded systems such as the Daisy Seed. A convincing output may require a large number of oscillators, increasing CPU usage and limiting the system's ability to process multiple voices or effects concurrently (Zölzer, 2011).

Amplitude Modulation (AM) and **Frequency Modulation (FM) synthesis** offer rich timbral possibilities using fewer oscillators than additive synthesis. FM synthesis, in particular, can generate complex and evolving tones with only two oscillators (Chowning, 1973). However, both techniques require careful parameter tuning, and the resulting sonic behaviours may be less intuitive for inexperienced users to understand or control. Effective integration in this context would require more sophisticated mapping or user interface design (Misra & Cook, 2009).

Granular synthesis was deemed less appropriate for this application due to its high

memory usage and greater implementation complexity. While capable of producing highly textured and evolving sounds, granular synthesis requires real-time manipulation of numerous audio grains, which places significant demands on both CPU and memory, potentially exceeding the capabilities of the Daisy Seed or reducing system stability in a live context (Truax, 1988).

Wavetable synthesis presents an efficient alternative by using precomputed waveform tables, substantially reducing real-time computational load (Roads, 2004). However, its timbral range is inherently limited to the content of the wavetable unless techniques like interpolation or morphing are implemented. While potentially useful for stable and predictable sounds, this limitation could reduce expressive flexibility (Zölzer, 2011).

Physical modelling synthesis was ultimately excluded due to its significant computational demands and implementation complexity. Although it offers highly expressive and realistic instrument emulations, it exceeds the real-time processing capabilities of the project's hardware, particularly when constrained by the block processing strategy and CPU load limitations identified in Section 5.2 (Smith, 1992).

Sampling, while not strictly a synthesis technique, was also evaluated. It allows for the quick playback of high-quality pre-recorded audio but lacks the dynamic modulation capabilities required for interactive control. Furthermore, its storage and memory requirements make it less suitable for embedded platforms unless applied in a highly constrained or static manner (O'Sullivan, 2012).

5.3.1 Partial Conclusion

Subtractive synthesis was chosen as the primary synthesis method for this project due to its optimal balance of expressiveness, computational efficiency, and intuitive control mapping (Zölzer, 2011). It supports real-time modulation via filters and control inputs, complements the simple mapping strategy adopted (Section 5.1), and remains within the processing constraints of the Daisy Seed platform (Section 5.2).

Other methods, such as additive and FM synthesis, were retained as potential secondary features. These may be selectively implemented in future iterations or employed for specific sound types if sufficient computational resources are available.

Granular synthesis, physical modelling, and sampling were excluded from the current scope due to their demanding requirements and limited suitability for the project's public, fast-paced environment.

5.4 Conclusion

Based on the evaluations and justifications presented in this section, the following software design decisions have been established for the project:

- **Mapping strategy:** Simple mapping, prioritising intuitive and responsive control for public interaction.
- **Audio processing logic:** Block-based processing, starting with a buffer size of 32 samples and adjustable based on performance requirements.
- **Sound synthesis method:** Subtractive synthesis, offering a balance between sonic flexibility and computational efficiency.

These decisions directly support project requirements 1, 6, 7, and 8 as outlined in Section 3.5, ensuring that the system remains responsive, robust, expressive, and suitable for real-time performance in an embedded context. The following Section 6 will detail how these design choices were implemented in practice.

6 Implementation

This section presents the practical implementation of the design decisions outlined in the hardware and software analyses (Sections 4 and 5). It details the setup of the development environment, configuration of the Daisy Seed microcontroller, hardware schematics, and integration of sensor inputs and digital signal processing components.

6.1 Setting up the project

This section describes how to set up the development environment and project structure, including how to integrate required libraries, set up the Makefile, and configure the development workspace in Visual Studio Code.

A thorough, step-by-step tutorial for configuring the development environment and starting a new project can be found in the official Daisy Seed documentation: https://daisy.audio/tutorials/cpp-dev-env/. The initial configuration in this project followed the steps outlined in that guide.

The LibDaisy and DaisySP libraries were then added to the project directory. DaisySP provides a collection of digital signal processing algorithms for audio synthesis, while LibDaisy offers low-level hardware abstraction for the Daisy platform.

The following commands were used to incorporate these libraries into the project as Git submodules:

```
git submodule add git@github.com:electro-smith/DaisySP.git
git submodule add git@github.com:electro-smith/libDaisy.git
```

Listing 1: *Include libraries to the git repository*

As illustrated in Figure 21, both DaisySP and LibDaisy have been added to a custom `libraries` folder located at the root of the project directory.

```
  # Project Name
TARGET = sensorsynth

# Sources
CPP_SOURCES = main.cpp $(shell find ./SynthLib -name "*.cpp") $(shell find ./Hardware
    -name "*.cpp")
```

```
6  CXXFLAGS += -I. -I./SynthLib -I$(shell find ./SynthLib -type d)
7  CXXFLAGS += -I./Hardware -I$(shell find ./Hardware -type d)
8  CFLAGS += -I../libraries/libDaisy/src
9
10
11 CPP_SOURCES = main.cpp \
12             $(shell find ./SynthLib -name "*.cpp") \
13             $(shell find ./Hardware -name "*.cpp") \
14
15 # Library Locations
16 LIBDAISY_DIR = ../libraries/libDaisy/
17 DAISYSP_DIR = ../libraries/DaisySP/
18
19 # Core location, and generic Makefile.
20 SYSTEM_FILES_DIR = $(LIBDAISY_DIR)/core
21 include $(SYSTEM_FILES_DIR)/Makefile
22
23 # Float support
24 # Uncomment the following line to enable float support in printf
25 # LDFLAGS += -u _printf_float
26
27
28 C_INCLUDES += -I../libraries/libDaisy/Drivers/STM32_USB_Device_Library/Class/CDC/Inc
29 C_INCLUDES += -I../libraries/libDaisy/Drivers/STM32_USB_Device_Library/Core/Inc
```

Listing 2: *Makefile Config root/SensorSynth/Makefile*

As shown in Listing 2, a Makefile was configured to automate the build process. The target executable was designated as `sensorsynth`, and the source files were aggregated from multiple directories using shell commands (see lines 4–5). Specifically, all `.cpp` files located within the `SynthLib` and `Hardware` directories were recursively identified and included. Include paths were defined using the `-I` flag to ensure that the compiler could locate the necessary header files, including those within subdirectories (lines 6–8). The locations of the `libDaisy` and `DaisySP` libraries were specified (lines 11–13), and USB support headers were added to `C_INCLUDES` (lines 28–29). Finally, the core system files and a generic Makefile were included (lines 20 -21).

The final step in setting up the project and development environment was to configure the Visual Studio Code workspace by adding the relevant library paths to the `c_cpp_-`

`properties.json` file. This configuration ensures that IntelliSense and the build system can correctly resolve header files from both the local project files and the external libraries. As shown in Listing 3, the `includePath` field was updated to include paths to newly created folders within the project (`SynthLib` and `Hardware`), which will be implemented later in this section, as well as the external `LibDaisy` and `DaisySP` libraries.

```
"includePath": [
"${workspaceFolder}/",
"${workspaceFolder}/SensorSynth/SynthLib/",
"${workspaceFolder}/SensorSynth/Hardware/",
"${workspaceFolder}/../libraries/libDaisy/",
"${workspaceFolder}/../libraries/DaisySP/**"
],
```

Listing 3: *.vscode/c_cpp_properties.json settings*

Once this was completed, the setup could be tested by running the commands shown in Listing 4. If the build was successful, a message would confirm that the `sensorsynth.bin` file had been created, as shown in Figure 20.

```
cd libraries/libDaisy && make clean && make &&
cd ../DaisySP && make clean && make &&
cd ../../SensorSynth && make clean && make
```

Listing 4: *Build commands*

```
arm-none-eabi-objcopy -O ihex build/sensorsynth.elf build/sensorsynth.hex
arm-none-eabi-objcopy -O binary -S build/sensorsynth.elf build/sensorsynth.bin
```

Figure 20: *Build success message confirming creation of sensorsynth.bin*

Figure 21 again shows the project folder structure, highlighting the inclusion of the `LibDaisy` and `DaisySP` libraries within the `libraries` directory.

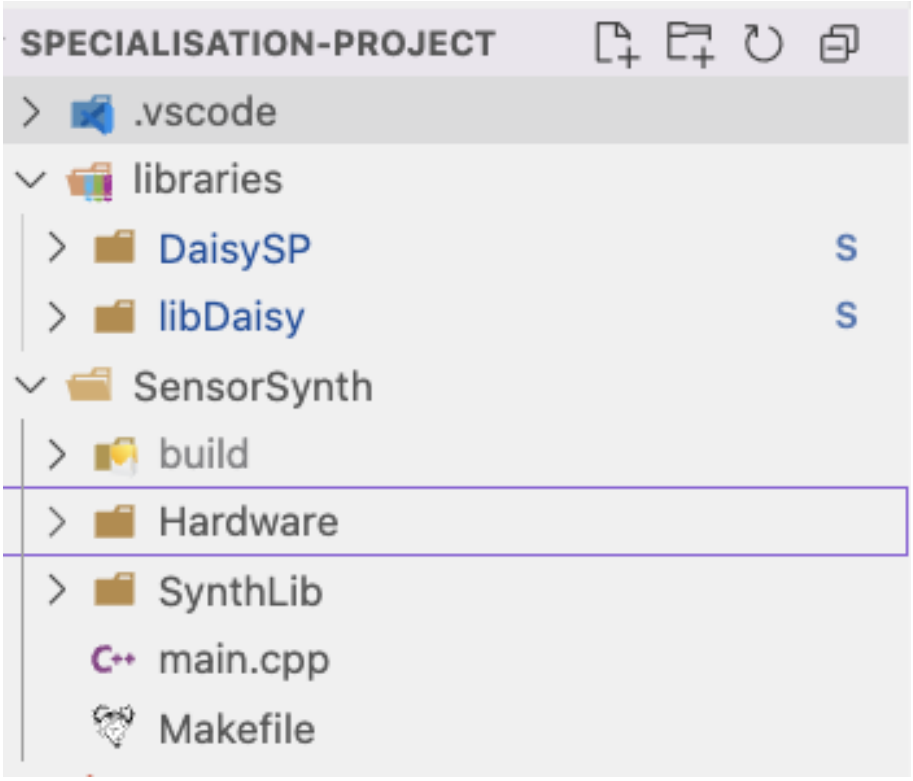

Figure 21: *Project folder structure showing included libraries*

Once the development environment was fully set up and successfully tested, the next step was to configure the Daisy Seed for the project, as detailed in Section 6.2.

6.2 Daisy Seed Configuration

This section is going to have a closer look to how Daisy seed was configured in software design. Using `main.cpp`, `hardware.h` and `controls.h`.

6.2.1 `Hardware` class

To manage hardware initialization and configuration, a Hardware class was implemented in `hardware.h` within the `sensorsynth` namespace (Listing 5). This class encapsulates setup procedures for the Daisy Seed microcontroller, leveraging the `DaisySeed` class from the LibDaisy library. Its responsibilities include configuring the audio system and initializing analogue-to-digital conversion (ADC) channels.

The `Init` method performs the primary hardware setup. It sets the desired audio block size mentioned in Section 5.2 and calls private helper methods to configure necessary pins and ADC channels. ADC input pins are defined using a `std::vector` of `daisy::Pin` objects, populated with predefined analogue inputs. The `ConfigureADC` method dynamically creates an array of `AdcChannelConfig` objects based on the number of sensors, associating each channel with the corresponding pin and initializing them accordingly. Once configured, the ADC system is started.

Audio processing is initiated using the `StartAudio` method, which accepts a callback function that is passed to the Daisy Seed's internal audio system. Callback function is described in Section 6.4.1

```
    #include "daisysp.h"
#include "daisy_seed.h"
#include <memory>

#pragma once
#ifndef HRDWR_H
#define HRDWR_H

#ifdef __cplusplus

using namespace daisysp;
using namespace daisy;
namespace sensorsynth
{
    class Hardware
    {
    public:
        daisy::DaisySeed hw_;

        Hardware() {};
        ~Hardware() {};

        float Init(size_t blocksize)
        {
            hw_.Init();
            hw_.SetAudioBlockSize(blocksize);
```

```
27
28          ConfigurePins();
29          sensor_count_ = adc_pins.size();
30
31          ConfigureADC();
32
33          return hw_.AudioSampleRate();
34      };
35
36      void StartAudio(daisy::AudioHandle::AudioCallback cb) { hw_.StartAudio(cb); };
37
38  private:
39      void ConfigureADC()
40      {
41          AdcChannelConfig adc_cfg[sensor_count_];
42
43          for (u_int8_t i = 0; i < sensor_count_; i += 1)
44          {
45              adc_cfg[i].InitSingle(adc_pins[i]);
46          }
47
48          hw_.adc.Init(adc_cfg, sensor_count_);
49          hw_.adc.Start();
50      }
51      void ConfigurePins()
52      {
53          adc_pins.push_back(daisy::seed::A0);
54          adc_pins.push_back(daisy::seed::A1);
55          adc_pins.push_back(daisy::seed::A2);
56          adc_pins.push_back(daisy::seed::A3);
57          adc_pins.push_back(daisy::seed::A4);
58      }
59      std::vector<daisy::Pin> adc_pins;
60
61      u_int8_t sensor_count_;
62  };
63  }
64
65  #endif
```

```
#endif
```

Listing 5: *hardware.h*

Once the hardware is configured, the next step is to set up analogue input processing as shown in Section 6.2.2.

6.2.2 `Control` Class

To manage analogue input processing, a `Controls` class was developed, as shown in Listing 6. This class initialises and updates the state of five analogue inputs - four potentiometers and one light dependent resistor. Each input is represented by an instance of the `daisy::AnalogControl` class provided by the LibDaisy library. The `AnalogControl` class inherently filters input noise and normalises values to a range between 0.0 and 1.0.

The `Init` method assigns each control to a specific ADC channel by referencing pointers obtained from the `DaisySeed` hardware abstraction layer. This method also configures the sample rate for each control to enable accurate real-time signal tracking. During runtime, the `Process` method is invoked to update the values of the controls continuously. This method refreshes the internal state of each analogue input, allowing the system to respond dynamically to user interactions.

```
#pragma once
#include "daisy_seed.h"

#ifndef CTRLS_H
#define CTRLS_H
#ifdef __cplusplus

namespace sensorsynth
{
    class Controls
    {
    public:
        daisy::AnalogControl pot1, pot2, pot3, pot4, ldr1;

```

```
15        void Init(const daisy::DaisySeed &hw, float sample_rate)
16        {
17            pot1.Init(hw.adc.GetPtr(0), sample_rate);
18            pot2.Init(hw.adc.GetPtr(1), sample_rate);
19            pot3.Init(hw.adc.GetPtr(2), sample_rate);
20            pot4.Init(hw.adc.GetPtr(3), sample_rate);
21            ldr1.Init(hw.adc.GetPtr(4), sample_rate);
22        }
23
24        void Process()
25        {
26            pot1.Process();
27            pot2.Process();
28            pot3.Process();
29            pot4.Process();
30            ldr1.Process();
31        }
32    };
33 }
34 #endif
35 #endif
```

Listing 6: *controls.h file*

Once this setup was complete, it was time to initialize and configure these classes in the `main.cpp` file, as outlined in Section 6.2.3.

6.2.3 `main.cpp`

The `main.cpp` file (see Figure 7) contains the `Main` function and the `AudioCallback` function, which is passed to `StartAudio(daisy::AudioHandle::AudioCallback cb)` as discussed in Section 6.2. This section focuses on the `Main` function, while the `AudioCallback` function will be examined in detail in Section 6.4.

The `Main` function is responsible for starting the audio processing loop, enabling control interfaces, and initializing hardware components. To guarantee their persistence throughout the program's lifetime, the `Hardware` and `Controls` classes are instantiated as static objects. The `Init` method of the `Hardware` class is called at startup with a spec-

ified audio block size specified in Section 5.2. This method returns the audio sample-rate and bit depth of the system, which is 24-bit / 48,000 Hz by default (Electro-Smith, 2025a). By the calculations mentioned in Section 5.2, new estimated audio latency is ≈ 0.66 ms. This was not changed, because these settings guarantee a better quality then the minimum required in Section 3.5 for requirements 2 and 3 .

Audio processing is started by passing a callback function, `AudioCallback`, to the `StartAudio` method. Within the infinite loop, the `Process` method of the `Controls` class is called repeatedly to update the state of the analogue controls in real time. This ensures continuous sensor data reading for audio signal processing.

```
#include "daisy_seed.h"
#include "./Hardware/hardware.h"
#include "./Hardware/controls.h"

using namespace daisy;
using namespace sensorsynth;

static sensorsynth::Hardware hw;
static sensorsynth::Controls controls;

const size_t bufferSize = 32;


static void AudioCallback(AudioHandle::InputBuffer in,
                          AudioHandle::OutputBuffer out,
                          size_t size)
{

}

int main(void)
{
    float sample_rate = hw.Init(256);

    controls.Init(hw.hw_, sample_rate);

    hw.StartAudio(AudioCallback);

```

```
    while (1)
    {
        controls.Process();
    }
}
```

Listing 7: *main.cpp*

Once the Daisy Seed was configured, it was possible to proceed with connecting the hardware components as described in Section 6.3.

6.3 Connecting Hardware

This section provides a list of the components used, along with the hardware schematics and a prototype photograph.

6.3.1 Components List

The following components were used for the project:

1. 1x Daisy Seed
2. Jumper wires and connecting cables
3. 2x Prototyping breadboards
4. 4x $10k\Omega$ linear potentiometers
5. 1x Light dependent resistor (LDR)
6. 1x $1k\Omega$ resistor
7. 3.5mm female audio jack connector
8. Headphones or speakers

6.3.2 Hardware Schematics

Figure 22 shows the designed hardware schematics. This image is also available in the Docs folder of the project's Git repository: https://github.com/RUC-MSc-CS-2sem-PC-2025/Specialisation-Project.

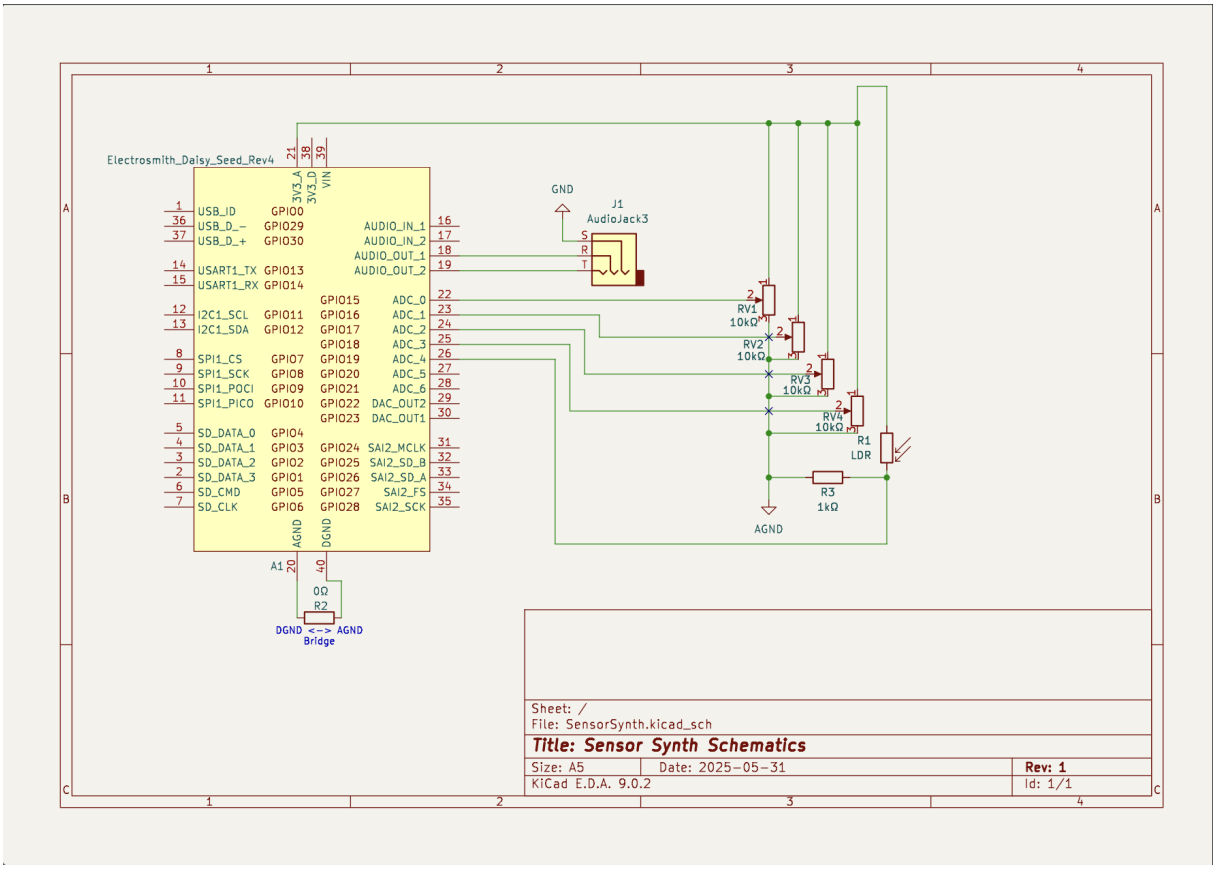

Figure 22: *Sensor Synth Schematics Rev 1*

6.3.3 Prototype

This section presents images of the assembled prototype from two perspectives: the top view (Figure 23) and the front view (Figure 24).

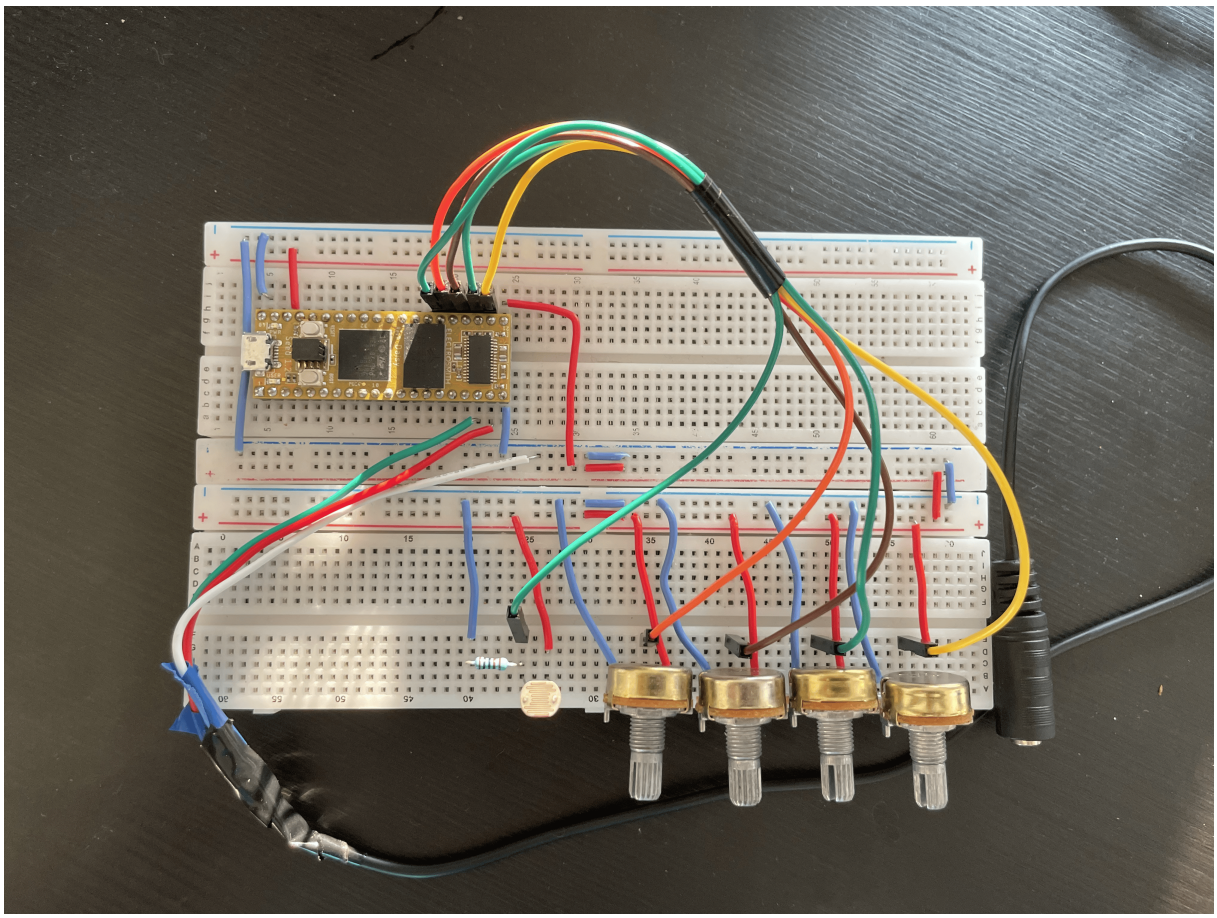

Figure 23: *Prototype: Top View*

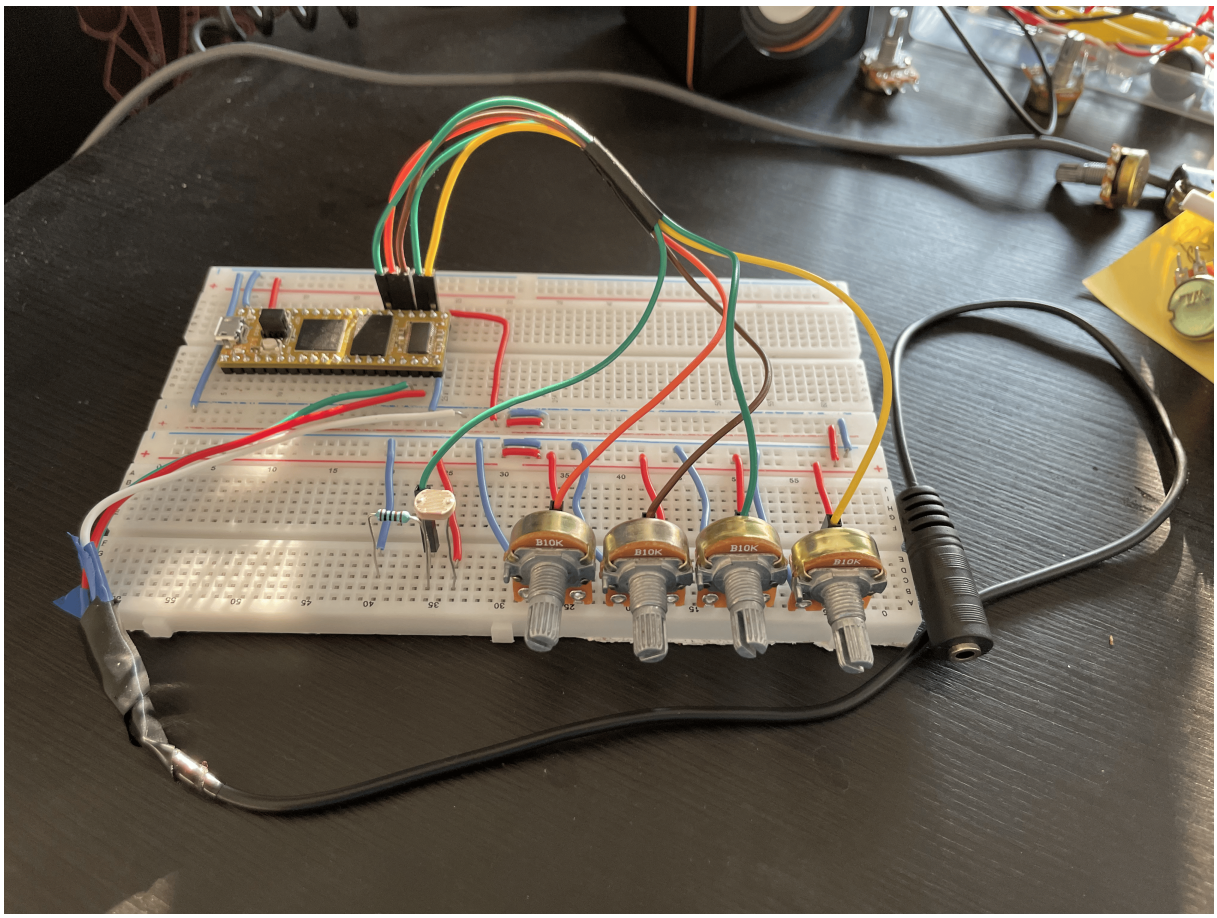

Figure 24: *Prototype: Front View*

Once the prototype was assembled, the project proceeded to the next phase - describing the sound design process in the following Section 6.4.

6.4 Sound design

This section details the `AudioCallback` function and describes the implementation of the digital signal processing algorithms used in the project. The focus will be on the subtractive synthesis engine (Section 6.4.1) and the audio effects applied, including filtering, ring modulation (RM), delay, and soft clipping (Section 6.4.2).

6.4.1 Subtractive Synthesis

The `SubtractiveSynth` class (Listing 8) implements a subtractive synthesis engine capable of generating a continuous sawtooth waveform. The class provides methods for initialization, parameter adjustment, and audio block processing. During initialization, the sample rate is stored, and default values are assigned to the oscillator's frequency, amplitude, and phase.

The `SetFrequency` and `SetAmplitude` methods allow external control over the oscillator's pitch and volume, respectively. The `ProcessBlock` method generates a sawtooth waveform by computing each sample based on the current phase and amplitude, following the formula described in Section 5.3. The generated discrete samples are written into the provided buffer.

This implementation encapsulates the core functionality required for a simple subtractive synthesizer, forming the foundation for further extension with filtering or modulation components.

```
#pragma once

#ifndef SUBTRACTIVE_H
#define SUBTRACTIVE_H

#include <array>
#include <cmath>
#include <algorithm>

namespace sensorsynth
{
    class SubtractiveSynth
    {
    public:
        SubtractiveSynth() {};
        ~SubtractiveSynth() {};

        void Init(float sampleRate)
        {
            sampleRate_ = sampleRate;
```

```
            frequency_ = 440.0f;
            amplitude_ = 1.0f;
            phase_ = 0.0f;
        }

        void SetFrequency(float freq) { frequency_ = freq; }
        void SetAmplitude(float amp) { amplitude_ = amp; }

        void ProcessBlock(float *buf, size_t size)
        {
            for (size_t i = 0; i < size; ++i)
            {
                float sample = amplitude_ * (2.0f * (phase_ / (2.0f * M_PI)) - 1.0f);
                phase_ += 2.0f * M_PI * frequency_ / sampleRate_;
                phase_ = fmodf(phase_, 2.0f * M_PI);
                buf[i] = sample;
            }
        }

    private:
        float sampleRate_;
        float frequency_;
        float amplitude_;
        float phase_;
    };
}

#endif // SUBTRACTIVE_H
```

Listing 8: *subtractive.h file*

After implementing the subtractive synthesis engine, it was integrated into `main.cpp`, where initialization occurs and control inputs are mapped to synthesis parameters.

The `AudioCallback` function handles real-time audio processing by updating the synthesizer's amplitude and frequency using sensor readings. These are passed to the synthesizer via:

```
subtractive.SetAmplitude(amplitude);
subtractive.SetFrequency(pitch);
```

A temporary buffer with a buffer size 32 samples is allocated using `std::array`, and the `ProcessBlock` method fills it with the generated sawtooth waveform:

```
std::array<float, bufferSize> block{};
subtractive.ProcessBlock(block.data(), size);
```

The resulting mono signal is duplicated across both output channels in output buffer:

```
for (size_t i = 0; i < size; i++)
{
    out[0][i] = block[i];
    out[1][i] = block[i];
}
```

In `main`, the synthesizer is initialized with the system sample rate, and default values are set:

```
subtractive.Init(sample_rate);
subtractive.SetAmplitude(0.5f);
subtractive.SetFrequency(440.0f);
```

Control values from two potentiometers are processed in the main loop. One controls amplitude directly, while the other maps pitch to a musically relevant range between 33 Hz to 1,000 Hz:

```
amplitude = controls.pot1.Value();

pitch = controls.pot2.Value();
pitch = 33.0f + pitch * (1000.0f - 33.0f);
```

This structure enables responsive, real-time synthesis driven by physical sensor input. The complete implementation is shown in Listing 10.

Once the subtractive synthesizer was implemented, the project was rebuilt and flashed to the Daisy Seed via USB using the following command:

```
make clean; make; make program-dfu
```

Listing 9: *Build and flash*

Audio and video recordings of the subtractive synthesis engine can be found at:

- (Video Link) (https://video.ruc.dk/media/t/0_jil5swxf)
- (Audio Link) (https://video.ruc.dk/media/t/0_7b2tywyu)

6.4.2 Effects Implementation

During initial listening tests of the device, it became evident that additional movement and complexity in the harmonic content would improve the sonic texture. Consequently, several audio effects were implemented to enrich the synthesized sound. These include ring modulation, dynamic filtering, delay lines, low-pass filtering, and soft clipping.

Ring Modulator: The ring modulation effect was implemented using the `daisysp::Oscillator` class to generate a square wave that amplitude modulates the output of the subtractive synthesizer (see Section 6.4.1). This modulation introduces effects, enhancing the harmonic motion.

A low-frequency oscillator (LFO) was initialized with the system's sample rate and configured to output a square waveform using:

```
lfo.Init(sample_rate);
lfo.SetAmp(0.3f);
lfo.SetWaveform(daisysp::Oscillator::WAVE_SQUARE);
```

Within the `AudioCallback` function, the LFO frequency is updated in real time using data from a light-dependent resistor (LDR), enabling sensor-driven control:

```
lfo.SetFreq(lfo_freq);
for (size_t i = 0; i < size; i++)
{
    block[i] *= lfo.Process() * 0.75;
```

```
}
```

This multiplies each sample of the audio block by the current LFO output, introducing binary ring modulation. The modulated audio is output to both stereo channels.

In the `main` loop, the LFO frequency is mapped to a range of 20 Hz to 500 Hz based on the LDR input:

```
lfo_freq = controls.ldr1.Value();
lfo_freq = 20.0f + lfo_freq * (500.0f - 20.0f);
```

This mapping allows dynamic, real-time interaction based on ambient light conditions. The complete implementation can be found in Listing 10.

Audio and video recordings of the resulting sound can be found at:

- Video Link (https://video.ruc.dk/media/t/0_6pyscz60)
- Audio Link (https://video.ruc.dk/media/t/0_urf06s2o)

Filter: To further shape the timbre of the synthesized output, a resonant band-pass filter was introduced using the `daisysp::LadderFilter` class. The filter operates in 12 dB band-pass mode and is updated in real time with control data from two potentiometers: one for cutoff frequency and one for resonance.

After the ring modulated signal is generated, it is passed through the filter using `filter.ProcessBlock(block.data(), size)`, enabling real-time sculpting of the spectral content. The cutoff frequency is mapped between 20 Hz and 15,000 Hz:

```
filter_cutoff = controls.pot3.Value();
filter_cutoff = 20.0f + filter_cutoff * (15000.0f - 20.0f);
filter.SetFreq(filter_cutoff);
```

Resonance is normalized and capped to ensure system stability:

```
resonance = controls.pot4.Value();
resonance = std::clamp(resonance, 0.0f, 1.0f) * 0.8f;
filter.SetRes(resonance);
```

The filter is initialized in the `main` function with:

```
filter.Init(sample_rate);
filter.SetFilterMode(daisysp::LadderFilter::FilterMode::BP12);
```

This setup allows for expressive, real-time control over timber of the sound. The complete implementation is shown in Listing 10.

Audio demonstrations of the filtered output are available:

- Video Link (https://video.ruc.dk/media/t/0_aposx6rg)
- Audio Link (https://video.ruc.dk/media/t/0_hlexd5dj)

Delay, Low-Pass Filter, and Soft Clipping: To add echo and control output dynamics, the final version of the signal processing chain incorporates a delay line, a low-pass filter, and soft clipping. The `daisysp::DelayLine` class is used to instantiate a short delay (14400 samples = 300 ms in sample-rate of 48,000 Hz). Delay includes feedback to create sustained echo effects.

Following modulation and band-pass filtering, the audio signal is sent through the delay line. The resulting dry and wet signals are mixed, and the combined output is soft-clipped using the `daisysp::SoftClip` function to prevent digital distortion and maintain a consistent output level.

Finally, a 12 dB low-pass filter is applied to remove any high-frequency artifacts introduced by earlier processing stages. This completes the synthesis chain and ensures a smoother output.

The complete implementation is shown in Listing 10.

```
#include "daisysp.h"
#include "daisy_seed.h"
#include "./SynthLib/synthlib.h"
#include "./Hardware/hardware.h"
#include "./Hardware/controls.h"

#include <cmath>
```

```
8
9  using namespace daisysp;
10 using namespace daisy;
11 using namespace sensorsynth;
12
13 static sensorsynth::Hardware hw;
14 static sensorsynth::Controls controls;
15 static sensorsynth::SubtractiveSynth subtractive;
16 static daisysp::LadderFilter filter, filterLP;
17 static daisysp::DelayLine<float, 48000> delayS;
18 static daisysp::Oscillator lfo;
19
20 const size_t bufferSize = 32;
21
22 float amplitude = 0.5f;
23 float pitch = 440;
24 float lfo_freq = 100.0f;
25 float filter_cutoff = 500.0f;
26 float resonance = 0;
27
28 static void AudioCallback(AudioHandle::InputBuffer in,
29                           AudioHandle::OutputBuffer out,
30                           size_t size)
31 {
32     subtractive.SetAmplitude(amplitude);
33     subtractive.SetFrequency(pitch);
34     lfo.SetFreq(lfo_freq);
35
36     filter.SetFreq(filter_cutoff);
37     filter.SetRes(resonance);
38
39     std::array<float, bufferSize> block{};
40
41     subtractive.ProcessBlock(block.data(), size);
42
43     for (size_t i = 0; i < size; i++)
44     {
45         block[i] *= lfo.Process() * 0.75;
46     }
47
```

```
48      filter.ProcessBlock(block.data(), size);
49
50      for (size_t i = 0; i < size; ++i)
51      {
52          float delayed = delayS.Read();
53          float input_with_feedback = block[i] + delayed * 0.7;
54          delayS.Write(input_with_feedback);
55          float wet = 0.5f * delayed;
56          float dry = 0.5f * block[i];
57
58          float drymix = dry + wet;
59
60          float out_sample = daisysp::SoftClip(drymix);
61
62          block[i] = out_sample;
63      }
64
65      filterLP.ProcessBlock(block.data(), size);
66
67      for (size_t i = 0; i < size; i++)
68      {
69          out[0][i] = block[i];
70          out[1][i] = block[i];
71      }
72  }
73
74  int main(void)
75  {
76      float sample_rate = hw.Init(bufferSize);
77
78      controls.Init(hw.hw_, sample_rate);
79
80      subtractive.Init(sample_rate);
81      subtractive.SetAmplitude(0.5f);
82      subtractive.SetFrequency(440.0f);
83
84      lfo.Init(sample_rate);
85      lfo.SetAmp(0.3f);
86      lfo.SetWaveform(daisysp::Oscillator::WAVE_SQUARE);
87
```

```
    filter.Init(sample_rate);
    filter.SetFilterMode(daisysp::LadderFilter::FilterMode::BP12);

    delayS.Init();
    delayS.SetDelay(14400.f);

    filterLP.Init(sample_rate);
    filterLP.SetFilterMode(daisysp::LadderFilter::FilterMode::LP12);
    filterLP.SetRes(0);

    hw.StartAudio(AudioCallback);

    while (1)
    {
        controls.Process();

        amplitude = controls.pot1.Value();

        pitch = controls.pot2.Value();
        pitch = 33.0f + pitch * (1000.0f - 33.0f);

        filter_cutoff = controls.pot3.Value();
        filter_cutoff = 20.0f + filter_cutoff * (15000.0f - 20.0f);

        resonance = controls.pot4.Value();
        if (resonance < 0.0f)
            resonance = 0.0f;
        else if (resonance > 1.0f)
            resonance = 1.0f;
        resonance *= 0.8f;

        lfo_freq = controls.ldr1.Value();
        lfo_freq = 20.0f + lfo_freq * (500.0f - 20.0f);
    }
}
```

Listing 10: *Final version of main.cpp*

Audio demonstrations of the final output are available:

- Video Link (https://video.ruc.dk/media/t/0_twwsc9ai)
- Audio Link (https://video.ruc.dk/media/t/0_8j032jw6)

Flow diagrams of `main` function and `AudioCallback` can be found in Section A Appendix Figures 28 and 29.

7 Evaluation

Table 3 maps functional requirement from Section 3.5 to its corresponding implementation in Section 6.

Req. #	Met?	Implementation Reference (Section 6)
1	Yes	Real-time control of amplitude and pitch via potentiometers in subtractive synthesis (Sections 6.4.1, 6.4.2)
2	Yes	Daisy Seed initialized with 48,000 Hz sample rate (Section 6.2.3)
3	Yes	24-bit output via onboard codec on Daisy Seed (Section 6.2.3)
4	Yes	LDR used for LFO modulation (Sections 6.2.2, 6.4.2)
5	Yes	Potentiometers used for amplitude, pitch, filter cutoff, and resonance (Sections 6.3, 6.2.2, 6.4.2)
6	Partial	Buffer size of 32 samples ensures low latency; real latency not formally measured (Section 6.2.3)
7	Yes	Subtractive synthesis implemented via `SubtractiveSynth` class (Section 6.4.1)
8	Yes	Ring modulation, band-pass filtering, delay, and soft clipping (Section 6.4.2)

Table 3: *Mapping of functional requirements to implementation components*

8 Discussion and Future Work

When listening to the final recording Audio Link (https://video.ruc.dk/media/t/0_8j032jw6), a noticeable noise floor around -70 dB is evident, as illustrated in Figure 25.

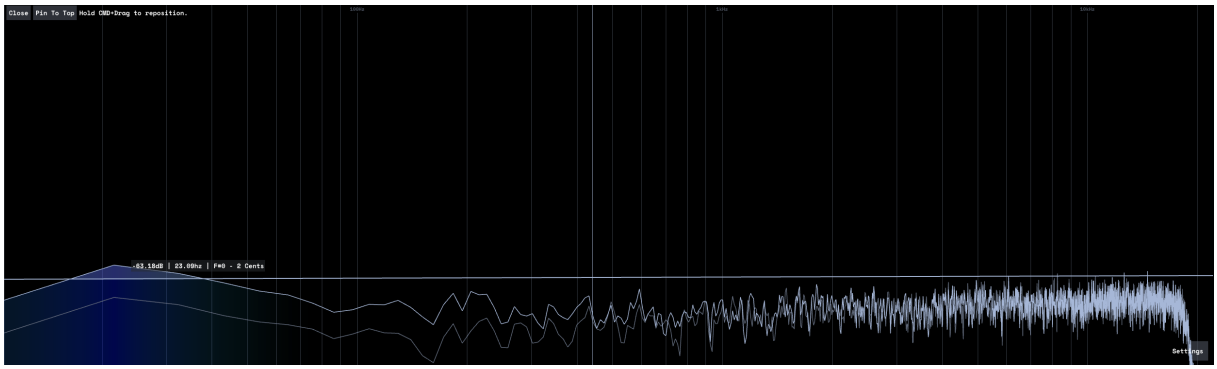

Figure 25: *Observed noise in the output signal*

This phenomenon warrants further investigation, as it may be attributed to incorrect hardware connections, recording inaccuracies, or errors within the digital signal processing (DSP) implementation. But most probably it is affected by how the recording environment was set up, because, the noise was not audible prior recording.

While the system was designed with low-latency performance in mind—using a buffer size of 32 samples at a 48,000 Hz sample rate, the total system latency (including ADC, DSP, DAC, and output stages) was not measured. Future work should include precise latency profiling.

Furthermore, the implementation of delay line led to a significant increase in SDRAM usage, approximately 35%—as illustrated in Figures 26 and 27. A more efficient delay line should be implemented to save memory and enable the integration of additional audio effects.

Memory Region	Used Size	Region Size	% Used
FLASH	81864 B	128 KB	62.46%
DTCMRAM	0 GB	128 KB	0.00%
SRAM	14700 B	512 KB	2.80%
RAM_D2	16704 B	288 KB	5.66%
RAM_D3	0 GB	64 KB	0.00%
BACKUP_SRAM	12 B	4 KB	0.29%
ITCMRAM	0 GB	64 KB	0.00%
SDRAM	0 GB	64 MB	0.00%
QSPIFLASH	0 GB	8 MB	0.00%

Figure 26: *Memory usage prior to delay line implementation*

Memory Region	Used Size	Region Size	% Used
FLASH	82176 B	128 KB	62.70%
DTCMRAM	0 GB	128 KB	0.00%
SRAM	206716 B	512 KB	39.43%
RAM_D2	16704 B	288 KB	5.66%
RAM_D3	0 GB	64 KB	0.00%
BACKUP_SRAM	12 B	4 KB	0.29%
ITCMRAM	0 GB	64 KB	0.00%
SDRAM	0 GB	64 MB	0.00%
QSPIFLASH	0 GB	8 MB	0.00%

Figure 27: *Memory usage following delay line implementation*

9 Conclusion

This project explored how a microcontroller-based synthesiser could be developed to generate audio using digital signal processing and respond to both human and environmental stimuli. The project successfully met most of the requirements, except Requirement 6, because latency was not formally measured. Overall, the device is capable of generating sound and being controlled by both human and environmental stimuli.

9.1 AI Declaration

Generative AI was used to assist with language editing and restructuring in specific paragraphs. No content was generated or written entirely by AI, and all ideas and final wording are the author's own.

A Appendix

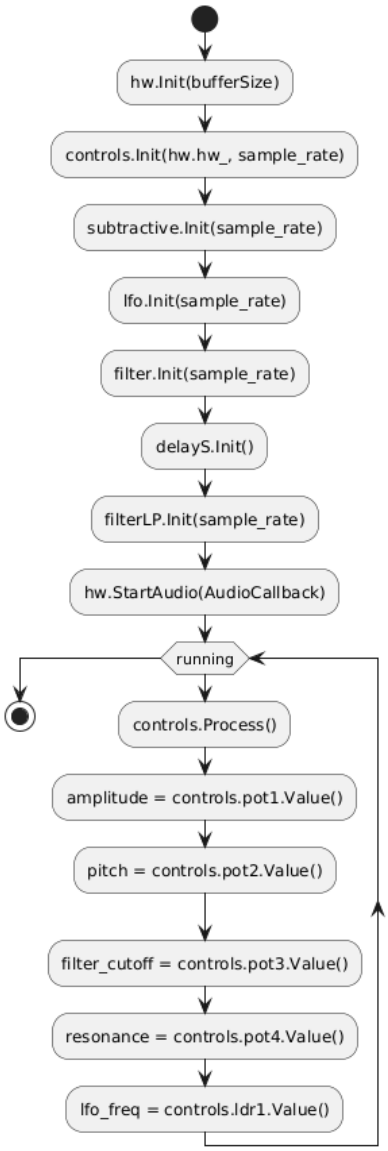

Figure 28: *Main program initialization and control loop: This diagram illustrates the setup of hardware and DSP components, followed by the continuous loop where sensor values are read and mapped to synthesis parameters in real time.*

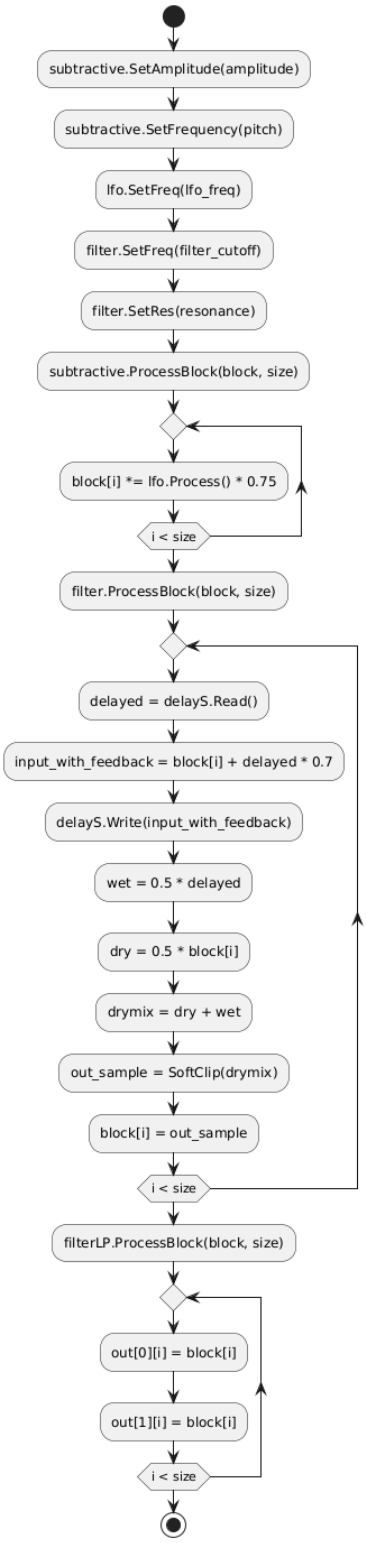

Figure 29: *DSP signal flow in the AudioCallback: The diagram shows the sequential processing steps for each audio block, including subtractive synthesis, LFO modulation, bandpass filtering, delay with feedback and soft clipping, lowpass filtering, and stereo output assignment.*

References

Arm Ltd. (2025). Cortex-M7 — High-Performance DSP for Automotive & IoT [Accessed: 2025-05-19]. https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m7

Calculator Academy Team. (2024). Audio latency calculator [Accessed: 2025-05-19]. https://calculator.academy/audio-latency-calculator/

Chowning, J. M. (1973). The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, *21*(7), 526–534.

Dobrian, C., & Koppelman, D. (2006). The e in nime: Musical expression with new computer interfaces. *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME06)*, 277–282.

Electricity and Magnetism. (2025). Resistive humidity sensor [Accessed: 2025-05-30]. https://www.electricity-magnetism.org/resistive-humidity-sensor/

Electro-Smith. (2025a). Getting started - audio [Accessed: 2025-05-20]. https://electro-smith.github.io/libDaisy/md_doc_2md_2__a3___getting-_started-_audio.html

Electro-Smith. (2025b). Libdaisy: Getting started - audio [Accessed: 2025-05-30]. https://electro-smith.github.io/libDaisy/md_doc_2md_2__a3___getting-_started-_audio.html

Electrosmith. (2025). *Daisy seed* [Accessed: 2025-05-03]. https://daisy.nyc3.cdn.digitaloceanspaces.com/products/seed/Daisy_Seed_datasheet_v1-1-5.pdf

Electro-Smith Community. (2023). Daisy seed cpu cycle benchmarks [Accessed: 2025-05-30]. https://forum.electro-smith.com/t/solved-how-to-do-mcu-utilization-measurements/1236

Email4mobile & Ilyin, D. (2025). Signal sampling [Vectorization by D. Ilyin. Original work by Email4mobile. CC0 license. Accessed April 30, 2025].

Espressif Systems. (2025). *Esp32 series socs* [Accessed: 2025-05-03]. https://www.esp32.dk/esp32_datasheet_en.pdf

Focusrite Support. (2025). Sample rate, bit depth and buffer size explained [Accessed: 2025-05-19]. https://support.focusrite.com/hc/en-gb/articles/115004120965-Sample-Rate-Bit-Depth-Buffer-Size-Explained

Fraden, J. (2016). *Handbook of modern sensors: Physics, designs, and applications*. Springer New York. https://doi.org/10.1007/b97321

Kuo, S. M., Lee, B. H., & Tian, W. (2013). *Real-time digital signal processing: Fundamentals, implementations and applications*. John Wiley & Sons. https://books.google.com/books/about/Real_Time_Digital_Signal_Processing.html?id=1RpwAAAAQBAJ

Lago, N. P., & Kon, F. (2004). The quest for low latency. *Proceedings of the International Computer Music Conference (ICMC)*. http://www-ccrma.stanford.edu/groups/soundwire/performdelay.pdf

Levitin, D. J. (2019). *This is your brain on music: Understanding a human obsession*. Penguin Books.

Lorenser, T. (2016, November). *The dsp capabilities of arm® cortex®-m4 and cortex-m7 processors: Dsp feature set and benchmarks* (tech. rep.) (Whitepaper). ARM Ltd.

Louder, J. (2023, April). What is audio digital signal processing (dsp)? a concise overview. https://soundstudiomagic.com/what-is-audio-digital-signal-processing-dsp/#digitaltoanalog-converter-dac

MathWorks. (2025). Sawtooth - sawtooth or triangle wave [[Online; accessed 31-May-2025]].

Medeiros, C. B., & Wanderley, M. M. (2014). A comprehensive review of sensors and instrumentation methods in devices for musical expression [Special Issue: State-of-the-Art Sensors in Canada 2014, edited by Prof. Dr. M. Jamal Deen]. *Sensors*, *14*(8), 13556–13591. https://doi.org/10.3390/s140813556

Megodenas. (2025). Waveform of a pure tone [Own work, Public Domain. Accessed April 30, 2025].

Misra, A., & Cook, P. R. (2009). Toward synthesized environments: A survey of analysis and synthesis methods for sound designers and composers. *Proceedings of the International Computer Music Conference (ICMC)*. http://soundlab.cs.princeton.edu/listen/synthesis_example/

O'Sullivan, S. (2012). Understanding the basics of sound synthesis. *Audiofanzine*. https://en.audiofanzine.com/sound-synthesis/editorial/articles/understanding-the-basics-of-sound-synthesis.html

Park, T. H. (2009). *Introduction to digital signal processing: Computer musically speaking*. World Scientific Publishing. https://doi.org/10.1142/6705

PJRC. (2025). *Teensy 4.1 development board* [Accessed: 2025-05-03]. https://www.pjrc.com/store/teensy41.html

Posudin, Y. (2014). *Methods of measuring environmental parameters* [Accessed: 2025-05-03]. John Wiley & Sons, Inc. https://doi.org/10.1002/9781118914236

Roads, C. (2004). *The computer music tutorial*. MIT Press.

Russ, M. (2009). *Sound synthesis and sampling* (3rd). Focal Press, Routledge. https://www.routledge.com/Sound-Synthesis-and-Sampling/Russ/p/book/9780240521053

Saini, I., Arora, A., Aggarwal, A., & Singh, M. (2018). Human motion detection and tracking for video surveillance: A cognitive science approach. In *Proceedings of the international conference on computing, communication and automation (iccca)* (pp. 689–696). Springer. https://doi.org/10.1007/978-981-13-1501-5_69

Smith, J. O. (1992). Physical modeling using digital waveguides. *Computer Music Journal*, *16*(4), 74–91.

Sweetwater Sound. (2022). *Which buffer size setting should i use in my daw?* [Accessed: 2025-05-30]. https://www.sweetwater.com/sweetcare/articles/which-buffer-size-setting-should-i-use-in-my-daw/

Thorat, S. (2023). Hermistor — types, diagram, working, advantages, application. *Metrology and Instrumentation*. https://learnmech.com/what-is-thermistor-types-of-thermistor-advantages-and-disadvantages/

Truax, B. (1988). Real-time granular synthesis with a digital signal processor. *Computer Music Journal*, *12*(2), 14–26.

Wikipedia contributors. (2024). Compact disc digital audio — wikipedia, the free encyclopedia [Accessed: 2025-05-28].

World, R. W. (2023). Photoresistor advantages and disadvantages. *RF Wireless World*. https://www.rfwireless-world.com/terminology/rf-components/photoresistor-advantages-and-disadvantages

Zölzer, U. (Ed.). (2011). *Dafx: Digital audio effects* (2nd). John Wiley & Sons. https://doi.org/10.1002/9781119991298