MSc Semester Project

# Development of a Stand-Alone Controller Solution for Electroacoustic Resonators

Laboratory of Wave Engineering (LWE)


**Professor:**
Hervé Lissek

**Student:**
Rodrigue de Guerre


**21th January 2026**

**Autumn 2025 - Spring 2026**

**Abstract**

This project investigates the feasibility of implementing an impedance synthesis control loop for electroacoustic resonators on a stand-alone embedded platform. The objective is to provide an alternative to Speedgoat Real-Time Target Machines prototyping systems with a compact and cost-effective microcontroller-based solution. The proposed controller is implemented on an STM32F767ZI developement board and executes a second-order control law derived from the physical parameters of a loudspeaker. The continuous-time control law is discretised using the bilinear transform and realized in close to real-time, using optimized digital signal processing routines. A User Interface has also been developed to automate the parameters computation, discrete filter conversion, firmware generation, and programming of the target device. Experimental validation has been performed through hardware-in-the-loop measurements and comparison with a Speedgoat reference system. The results demonstrate that, although some discrepancies remain, the embedded implementation reproduces the expected behavior of the impedance synthesis controller, confirming the viability of the proposed architecture, while highlighting remaining limitations related to gain calibration and output conditioning.

# Contents

# 1 Introduction

## 1.1 Context: The Challenge of Low-Frequency Absorption

Low-frequency noise control remains a significant challenge in room acoustics. Traditional passive treatments, such as porous absorbers or Helmholtz resonators, require substantial volume and depth to be effective at long wavelengths, often making them impractical for standard living spaces. To address this physical limitation, **Electroacoustic Absorbers (EAs)** have emerged as a compact and tunable alternative [1]. An electroacoustic absorber consists of a loudspeaker driver enclosed in a sealed cabinet, where the membrane's dynamics are actively modified by a control loop. By adjusting the acoustic impedance at the diaphragm, the system can mimic the behavior of a perfect absorber over a targeted frequency bandwidth, effectively "synthesizing" a resonator with optimal absorption properties.

## 1.2 Motivation: Moving Beyond Rapid Prototyping

The core of this active control relies on implementing a specific control loop, typically a second-order transfer function, that modifies the target impedance of the speaker based on the acoustic pressure at the membrane.

Currently, the validation of this control loop is performed using high-performance rapid prototyping systems, such as Speedgoat real-time target machines running MATLAB/Simulink models. While these systems offer exceptional flexibility and computational power for research and development, they represent a significant barrier to widespread adoption [2]. For a dedicated application like a single-channel electroacoustic absorber, a Speedgoat system is:

- **Over-dimensioned:** It provides far more processing power and I/O versatility than required for a fixed second-order filter.
- **Too expensive:** The hardware cost is incompatible with consumer electronics markets.
- **Not portable:** The reliance on bulky external racks limits testing and integration.

## 1.3 Hardware Selection: MCU vs FPGA

To transition from a lab prototype to a stand-alone solution, a choice had to be made between a Microcontroller Unit (MCU) and a Field-Programmable Gate Array (FPGA).

While FPGAs offer superior parallel processing and ultra-low latency suitable for high-frequency audio (MHz range), they generally come with significantly higher hardware costs and require development in HDL. Consequently, an MCU-based approach was selected for this project.

Since the active control of electroacoustic absorbers operates primarily in the low-frequency range (typically $< 1$ kHz), the STM32 architecture (with a Cortex-M7 core running at 216 MHz) provides a good balance between performance and cost.

## 1.4 Project Objectives

The primary objective of this semester project is to bridge the gap between laboratory validation and industrial prototyping. Specifically, in the context of a collaboration with **LG Electronics**, the goal is to migrate the control architecture to a stand-alone STM32 MCU. This involves:

1. Discretising the theoretical continuous-time control law.
2. Implementing the control loop on the STM32F767ZI development board.
3. Validating the embedded solution against the established Speedgoat reference.

## 2 Theory: Impedance Synthesis

To understand the control logic implemented on the MCU, we must first define the physical behavior of the loudspeaker and the target impedance we wish to achieve.

To implement an effective stand-alone controller, we must first translate the physical behavior of the electroacoustic resonator into a digital control law. This section details the modeling of the loudspeaker, the derivation of the impedance synthesis transfer function, and its discretisation for the STM32 microcontroller.

### 2.1 Loudspeaker Physical Model

The dynamics of a moving-coil loudspeaker in a sealed enclosure are governed by its mechanical impedance $Z_{ms}(s)$, which relates the driving force $F$ to the diaphragm velocity $v$. In the Laplace domain, the specific acoustic impedance $z_{ms}$ is given by:

$$z_{ms}(s) = \frac{p(s)}{v(s)} = \frac{M_{ms}s^2 + R_{ms}s + \frac{1}{C_{mc}}}{sS_d} \tag{1}$$

Where the physical parameters are defined as:

- $M_{ms}$: Mechanical mass of the moving assembly ($kg$).
- $R_{ms}$: Mechanical resistance ($N \cdot s/m$).
- $C_{mc}$: Mechanical compliance of the suspension and enclosure air ($m/N$).
- $S_d$: Effective surface area of the diaphragm ($m^2$).

### 2.2 Target Impedance Synthesis

The goal of the active control is to modify the speaker's natural impedence to match a target impedance $Z_{mt}(s)$. This target impedance allows us to tune the resonance frequency and damping ratio of the absorber to match specific room modes. The target impedance is defined using scaling factors ($\mu$) relative to the natural parameters:

$$Z_{mt} = S_d Z_{st} = j\omega\mu_M M_{ms}s + \mu_R R_{ms} + \frac{\mu_C}{j\omega C_{mc}} \tag{2}$$

By adjusting $\mu_M$, $\mu_R$, and $\mu_C$, we can soften the suspension or lighten the apparent mass, thereby shifting the absorption frequency.

### 2.3 Control Law Derivation

To achieve this target impedance, we introduce a feedback current $i(s)$ into the voice coil. The relationship between the measured pressure $p(s)$ and the required control current $i(s)$ is given by the transfer function $\Theta(s)$:

$$\Theta(s) = \frac{i(s)}{p(s)} = \frac{S_d}{Bl} \cdot \frac{Z_{ms}(s) - Z_{mt}(s)}{Z_{mt}(s)} \tag{3}$$

Substituting the expressions for $Z_{ms}$ and $Z_{mt}$, we obtain the specific second-order transfer function that must be implemented in the digital controller:

$$\Theta(s) = \frac{S_d}{Bl} \cdot \frac{s^2(\mu_M - 1)M_{ms}C_{mc} + s(\mu_R - 1)R_{ms}C_{mc} + (\mu_C - 1)}{s^2(\mu_M)M_{ms}C_{mc} + s(\mu_R)R_{ms}C_{mc} + \mu_C} \tag{4}$$

This rational function takes the standard biquadratic form:

$$H(s) = \frac{b_2 s^2 + b_1 s + b_0}{a_2 s^2 + a_1 s + a_0} \tag{5}$$

This continuous-time transfer function represents the exact synthesis logic that was previously running on the Speedgoat system. The challenge of this project is to discretise this equation, using the Bilinear Transform, and compute it efficiently on the STM32 MCU within a reasonable delay ($< 50\mu s$) to avoid instability.

### 2.4 Discrete-Time Implementation

To execute this control law on the MCU, the continuous transfer function $H(s)$ must be discretised into a difference equation that can be computed code.

We use the Tustin (Bilinear) Transformation to map the $s$-domain to the $z$-domain. The substitution is given by:

$$s \approx \frac{2}{T_s} \frac{1 - z^{-1}}{1 + z^{-1}}$$

where $T_s$ is the sampling period. Substituting this into the continuous transfer function yields a discrete transfer function in the $z$-domain:

$$H(z) = \frac{I(z)}{P(z)} = \frac{B_0 + B_1 z^{-1} + B_2 z^{-2}}{1 + A_1 z^{-1} + A_2 z^{-2}} \tag{6}$$

By applying the inverse Z-transform, we obtain the difference equation used to calculate the control output current $i[n]$ at discrete time step $n$:

$$i[n] = B_0 p[n] + B_1 p[n-1] + B_2 p[n-2] - A_1 i[n-1] - A_2 i[n-2] \tag{7}$$

Where:

- $p[n]$ is the current pressure measurement (ADC input).
- $p[n-1], p[n-2]$ are the previous pressure measurements.
- $i[n-1], i[n-2]$ are the previous calculated output currents.
- $B_0, B_1, B_2$ and $A_1, A_2$ are the discretised coefficients derived from the physical parameters and sampling time $T_s$.

This calculation will be performed using single-precision floating-point arithmetic. On the Cortex-M7 core, we can accelerate these operations by using the hardware Floating Point Unit (FPU), ensuring the loop executes within the timing constraints and without CPU stalling. We will see later how the CMSIS-DSP library enables us to do this.

## 3 Hardware and Development Environment

### 3.1 MCU Selection Criteria

To transition from a lab prototype to a stand-alone solution, the choice of the embedded platform was critical. Several candidates were evaluated, including the NXP i.MX RT1050 (600 MHz), the PJRC Teensy 4.1, and the STM32 Nucleo series.

Ultimately, the **STM32 Nucleo-F767ZI** was selected as the optimal platform. The key deciding factors were:

- **Performance:** The Arm Cortex-M7 core running at 216 MHz provides enough headroom for audio processing at low frequencies ($< 1$ kHz) [3].
- **Memory:** With 512kB of RAM and 2MB of Flash, the MCU can easily handle the buffering required for the control loop without external memory [3].
- **Integrated Peripherals:** The board features 3x 12-bit ADCs (2.4 MSPS) and 2x 12-bit DACs, removing the need for an external ADC & DAC. [4].

### 3.2 Hardware Setup

The hardware implementation is centered around the Nucleo board, serving as the bridge between the acoustic domain and the digital control logic.

- **Input:** An electret microphone (sensitivity $\approx 20$ mV/Pa) is connected to the ADC1 input (Pin PA_3) to measure the front pressure ($p_f$) at the speaker membrane.
- **Output:** The calculated control signal $i(t)$ is generated via the on-board DAC (Pin PA_5). This signal then goes through an Howland current pump [2] to drive the loudspeaker.

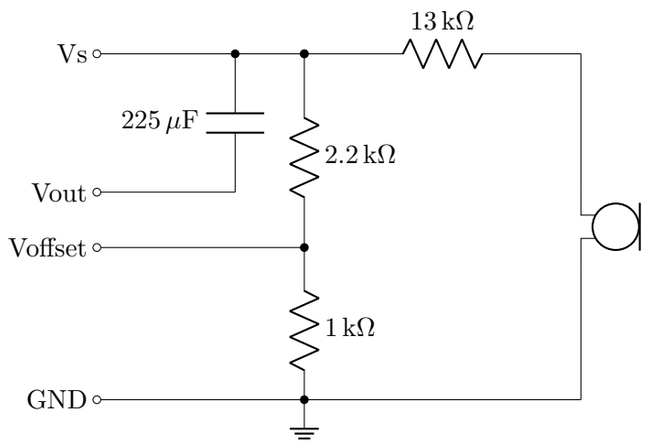



Figure 1: Microphone integration schematic

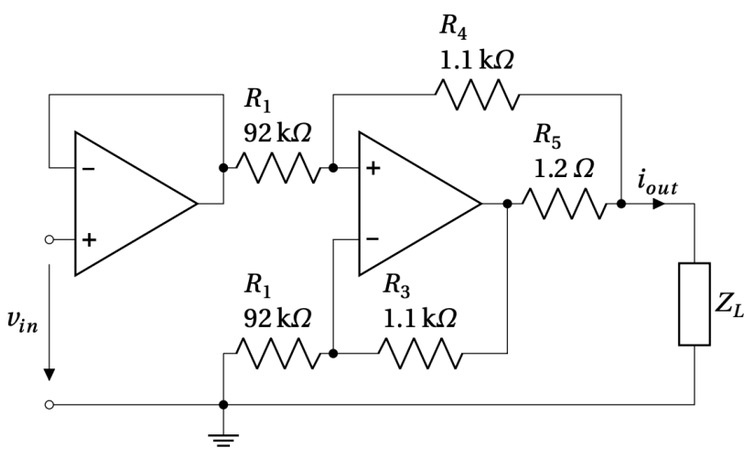



Figure 2: Howland Current Pump schematic [2]

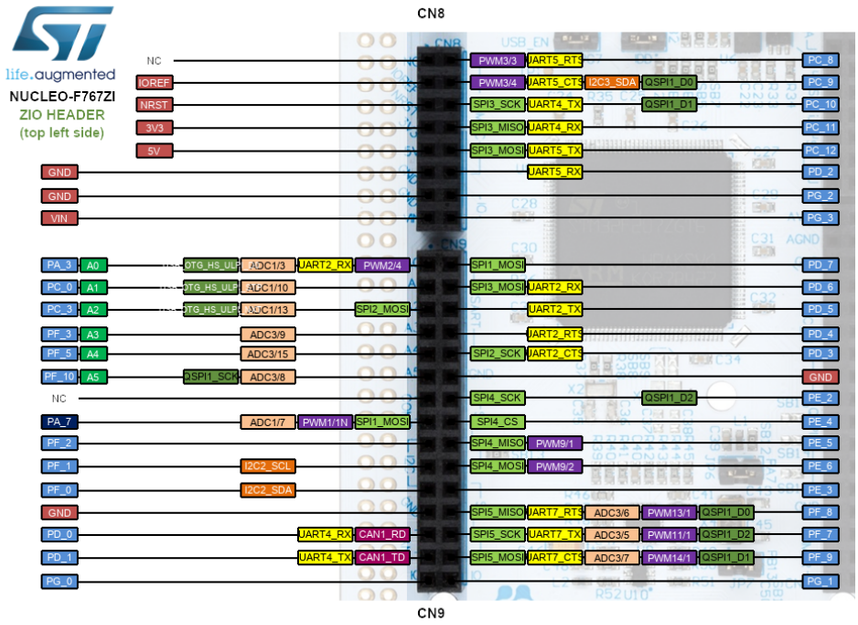



Figure 3: Nucleo F767ZI I/O PIN Config (left side of the board).

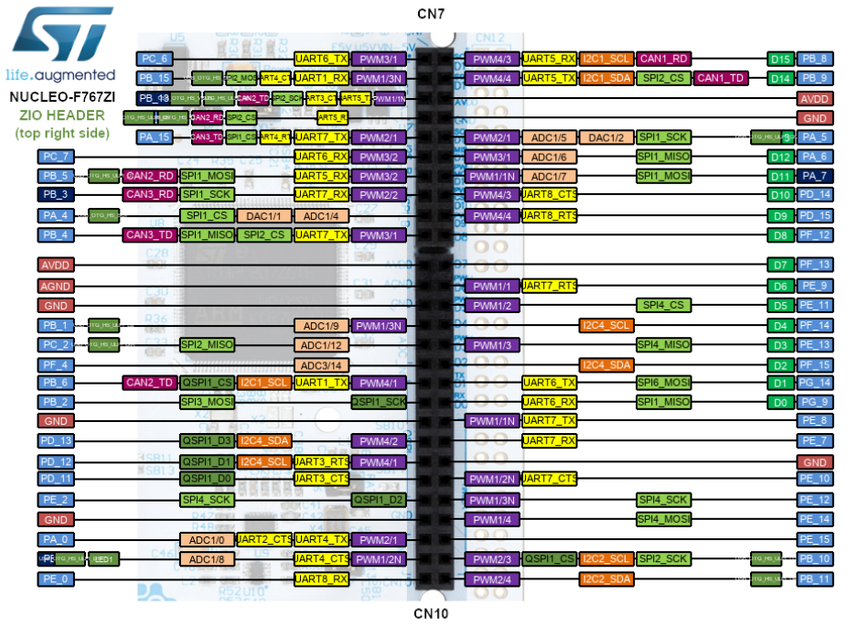



Figure 4: Nucleo F767ZI I/O PIN Config (right side of the board).

An important thing to note, is that the I/O pins for soldering are labbeled coreectly (at the back of the board), but the black I/O pins aren't (ie: Pin A0 is actually PA3).

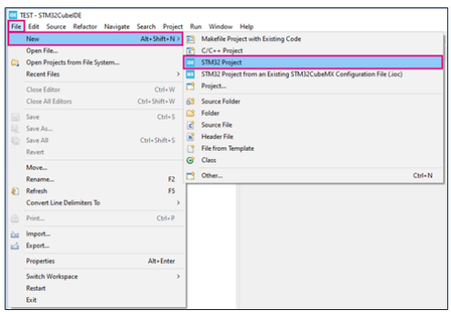
## 3.3 Development Environment: STM32CubeIDE

The firmware development was done using the STM32CubeIDE, an all-in-one (Eclipse-based) tool provided by STMicroelectronics. This section will show step by step how one could setup the same project from start using this IDE.

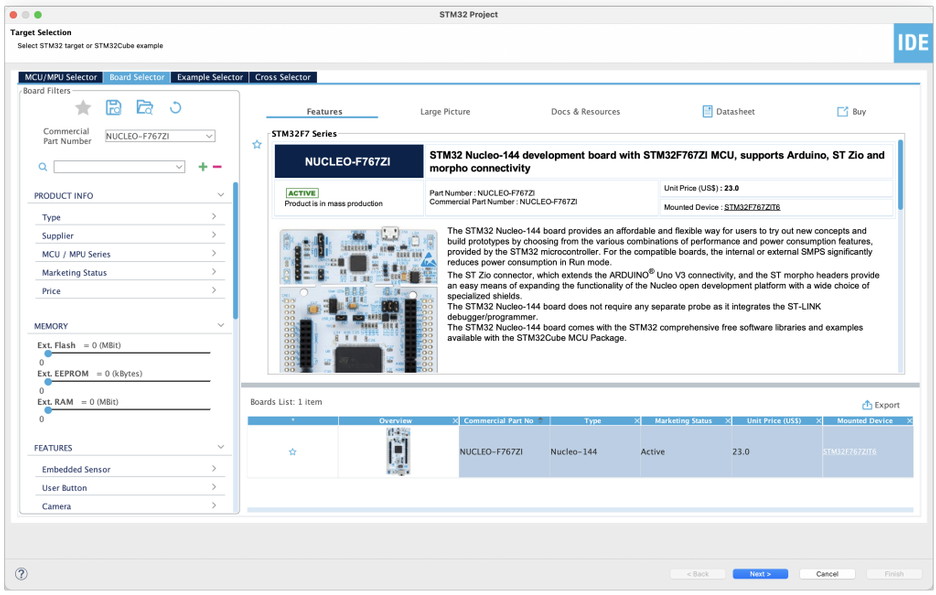
### 3.3.1 Step 1: Project Initialization

The first step is to create a new project targeting the specific hardware platform.

1. Launch STM32CubeIDE and navigate to **File** > **New** > **STM32 Project**.

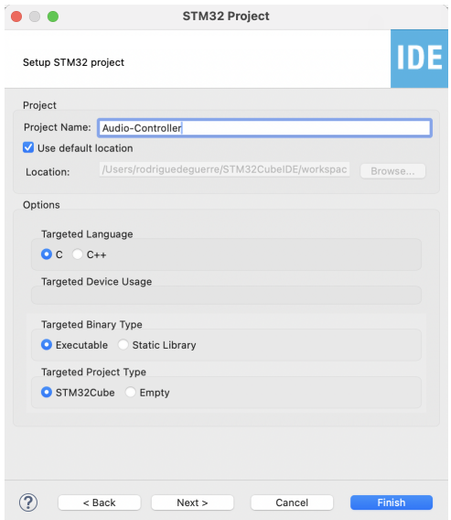



2. The "Target Selection" window will appear. In the "Board Selector" tab, search for the part number `NUCLEO-F767ZI`.




3. Select the board from the list and click **Next**.

4. Name the project (e.g., `Acoustic_Controller`) and click **Finish**. When asked to "Initialize all peripherals with their default Mode?", click **Yes**.



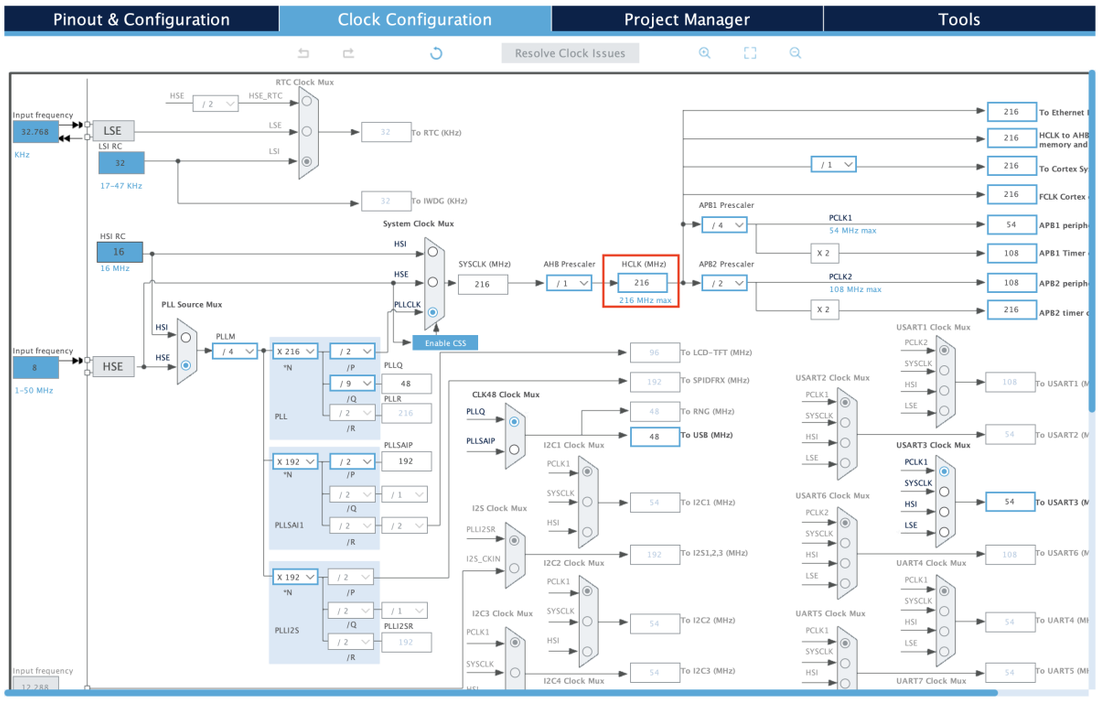

### 3.3.2 Step 2: Clock Configuration

To achieve the lowest possible latency between input and output, the microcontroller's core clock must be maximized.

1. Open the `.ioc` configuration file to enter the device configuration interface.

2. Navigate to the **Clock Configuration** tab.

3. Locate the **HCLK (MHz)** field at the far right of the clock tree.




4. Enter the maximum value of **216** MHz and press Enter [3]. The software will automatically calculate the necessary PLL (Phase Locked Loop) multipliers and dividers to reach this speed from the input source.

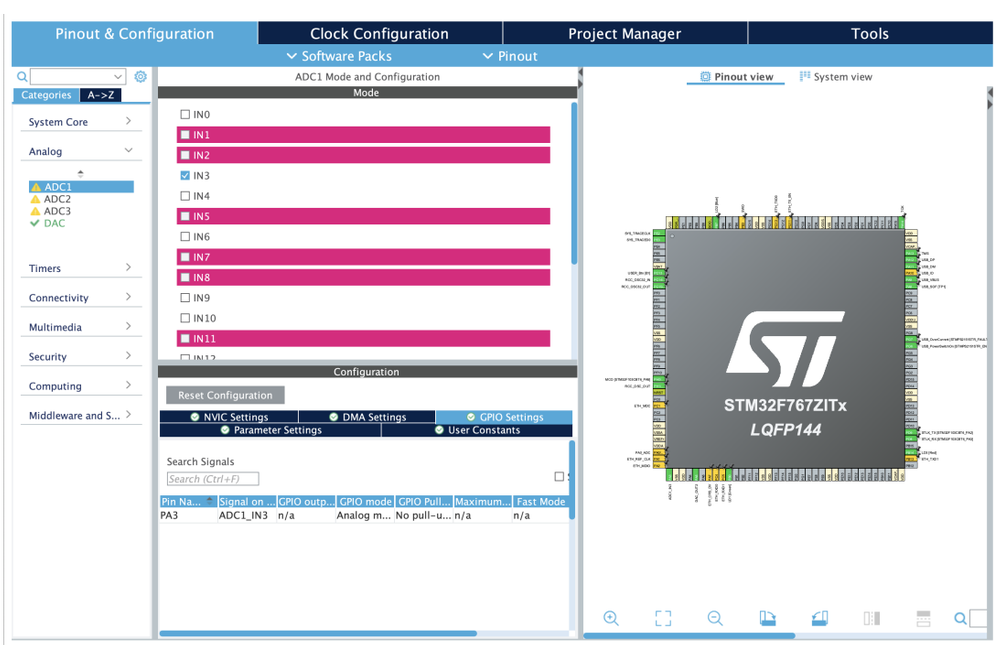
### 3.3.3 Step 3: Peripheral Configuration

The active control loop requires continuous sampling of the microphone and immediate output to the speaker.

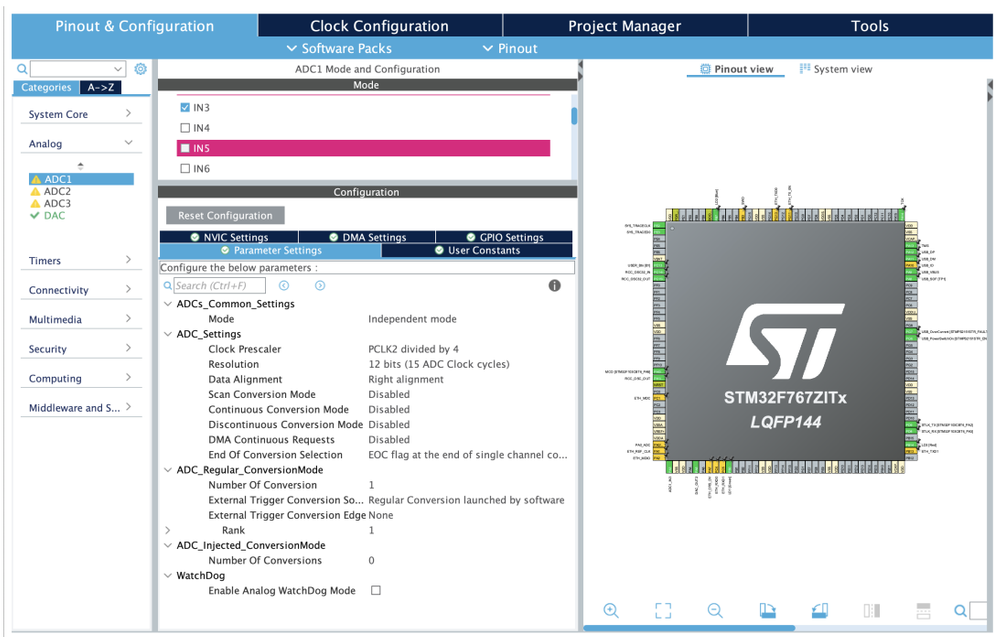
#### ADC Setup

1. In the **Pinout & Configuration** tab, expand **Analog** and select **ADC1**.

2. Set **IN3** to "IN3 Single-ended" and make sure that Pin PA_3 is selected.



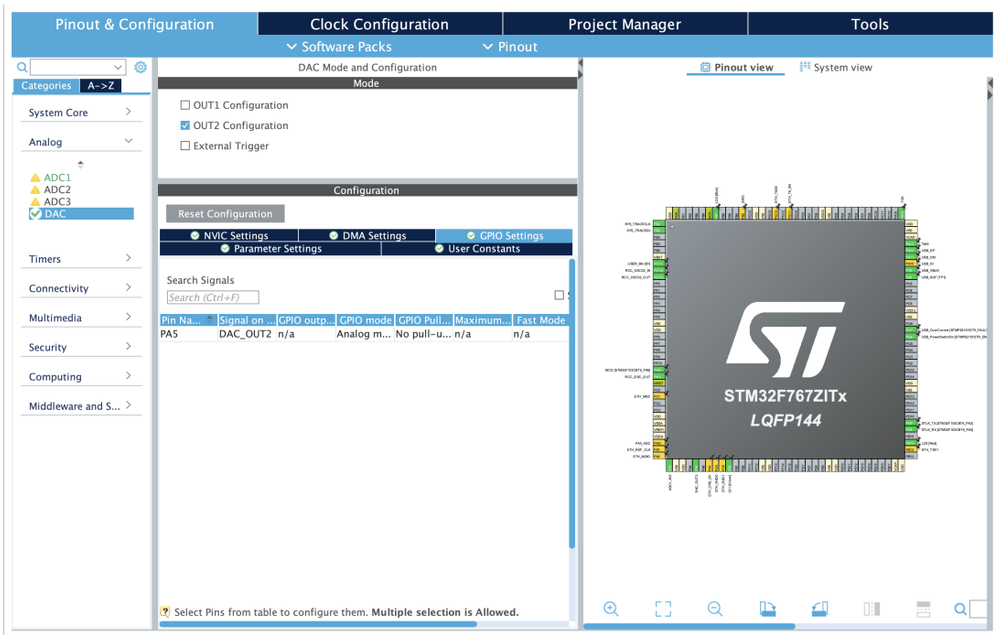
3. In the **Configuration** window below, apply the following settings:



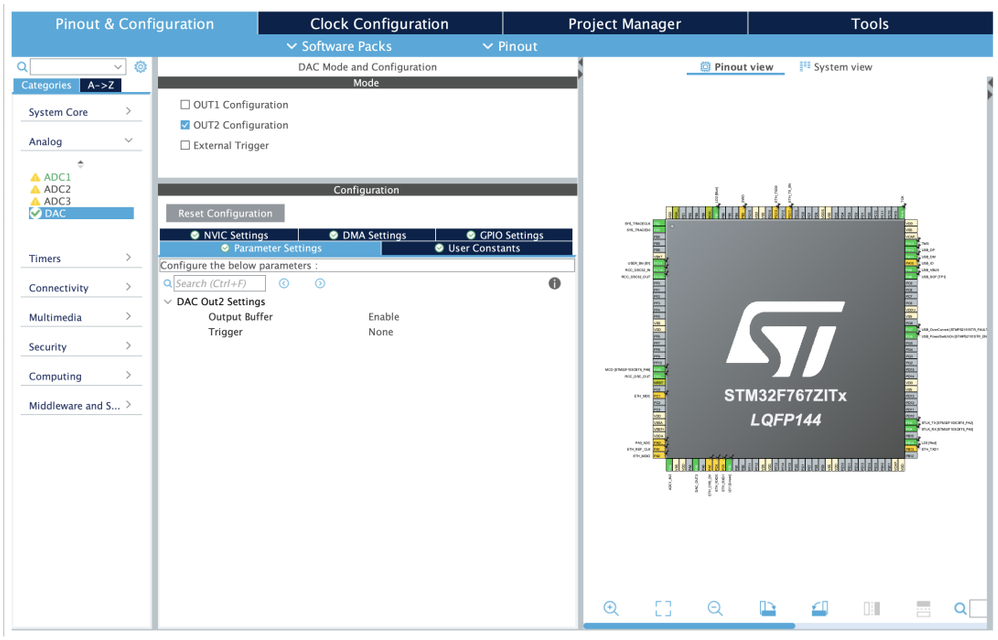

- **Resolution:** 12-bit (range 0–4095).
- **Continuous Conversion Mode: Enabled**. This is vital for ensuring the control loop never waits for a trigger; the ADC should always be sampling the acoustic pressure.
- **Sampling Time:** Select **15 Cycles** or a similar low value to minimize the acquisition time.

### DAC Setup

1. Under **Analog**, select **DAC**.
2. Check the box for **OUT2 Configuration** to enable the output channel and make sure that Pin PA_5 is selected.




3. In the configuration, ensure the **Output Buffer** is **Enabled**. This allows the DAC to drive the external amplifier circuit with lower impedance, improving signal integrity.




### Timer Setup

The digital control loop must run at a precise and deterministic sampling frequency ($f_s$). We utilize a hardware timer (TIM6) to trigger the control loop interrupt.

The update frequency of the timer determines our sampling frequency $f_s$. It is derived from the internal timer clock ($f_{timer}$) according to this formula:

$$f_s = \frac{f_{timer}}{(PSC + 1) \cdot (ARR + 1)} \tag{8}$$

On the STM32F767ZI, the APB1 timer clock ($f_{timer}$) typically runs at 108 MHz (half the HCLK frequency) [3]. Hence, to achieve a sampling frequency of 20 kHz, we have:

$$ARR = \frac{108,000,000}{20,000 \cdot (0 + 1)} - 1 = 5,399 \tag{9}$$

Note that it is recommanded to keep the Prescaler (PSC) as low as possible (usually 0). It is also important to note that the sampling frequency $f_s$ calculated here must be the same as the one used to calculate the discretised coefficients of the control.

1. Under **Timers**, select **TIM6**.

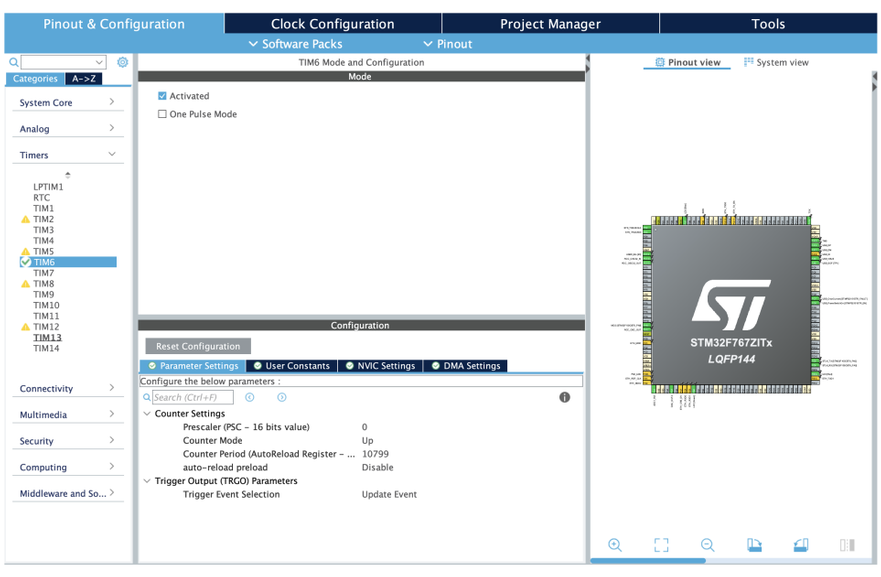



Figure 5: TIM6 configuration for fixed sampling frequency.

2. Check the box **Activated** to enable Timer 6.

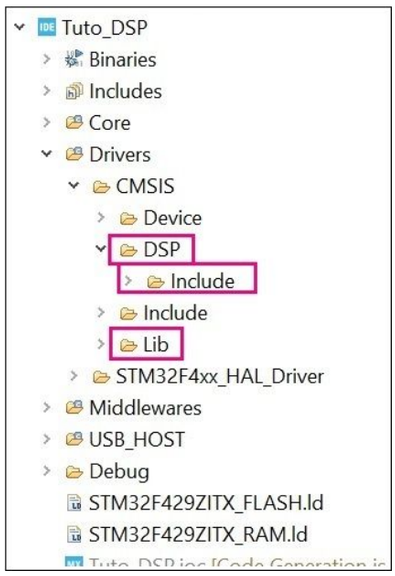
3. In the configuration, set the **Prescaler (PSC)** to `0` and the **Counter Period (ARR)** to `5399` (or `10799` if you want to lower $f_s$ to 10 kHz).

#### 3.3.4 Step 4: DSP Library Integration

To implement the second-order transfer function efficiently, the hardware Floating Point Unit (FPU) must be utilized via optimized libraries. We will now show how to integrate the CMSIS-DSP library in order to utilize this said FPU.

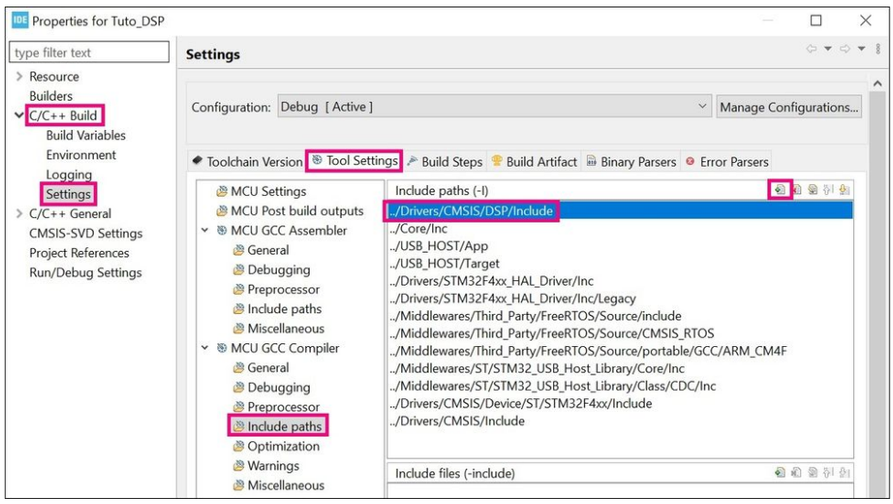
1. **File Integration:**

   - Navigate to the project directory and create a folder named `DSP` inside `Drivers/CMSIS`.
   - Copy the `Include` folder from the STM32Cube firmware repository `STMicroelectronics/STM32CubeF7/Drivers/CMSIS/DSP/Include` and paste it in the created folder.
   - Copy the `Lib` folder from the STM32Cube firmware repository `STMicroelectronics/STM32CubeF7/Drivers/CMSIS/Lib` and paste it under `../Drivers/CMSIS`.



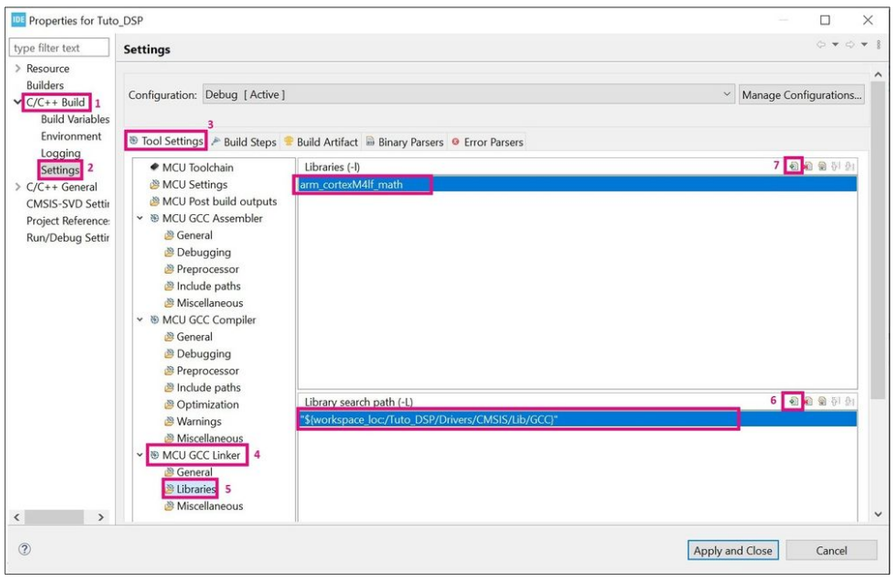

2. **Configure Include Paths:**

   - In STM32CubeIDE, right-click the project and select **Properties**.
   - Navigate to **C/C++ Build** > **Settings** > **MCU GCC Compiler** > **Include paths**.
   - Add the path to the new DSP headers: `../Drivers/CMSIS/DSP/Include`.



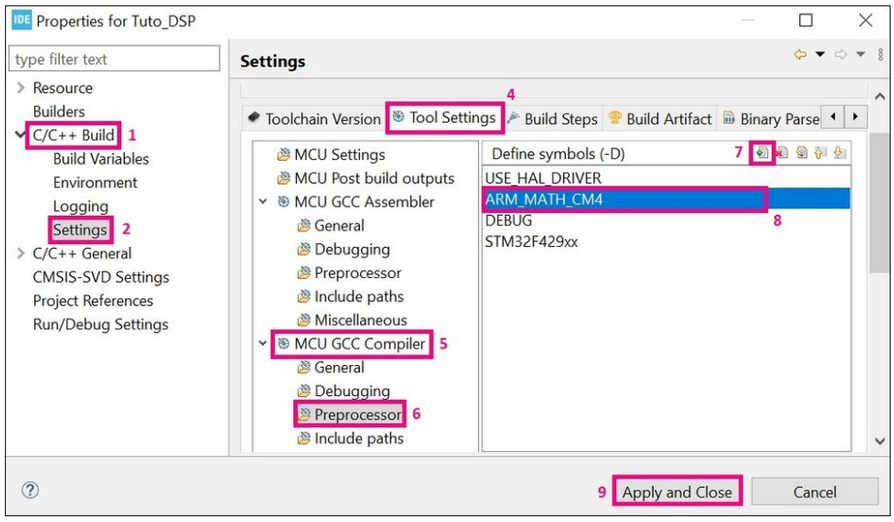
3. **Link the Library:**

   - In the same Settings window, navigate to **MCU GCC Linker > Libraries**.
   - Add the library search path: `../Drivers/CMSIS/DSP/Lib/GCC`.
   - Add the specific library for the Cortex-M7 core: `arm_cortexM7l_math` (ensure the `lib` prefix and `.a` extension are omitted in this field).




4. **Define Symbols:**

   - Navigate to **MCU GCC Compiler > Preprocessor**.
   - Add the symbol `ARM_MATH_CM7` to enable the FPU-specific instructions for the STM32F7.




### 3.3.5 Step 5: Code Generation

Finaly, once the configuration is complete:

1. Close the **Device Configuration Tool**. When asked to "Save the configuration and generate code", click Yes.
2. The IDE will generate the `main.c` file with all hardware initialization code ('HAL_Init', 'SystemClock_Config', 'MX_ADC1_Init', etc.) pre-written.
3. **Important:** All custom control logic must be written between the

```
/* USER CODE BEGIN */

/* USER CODE END */
```

comments to prevent it from being overwritten during future re-generations. The `main.c` file will always prevail, so it is a good practice to refer to it once the configuration is done for any specific question. One can also directly modify the `main.c`, skipping the configuration wizard altogether, but it requires carefull attention.

# 4 Firmware Testing and Validation

Now that the IDE has been properly set up, we can move on to the details of the firmware's implementation and overview the tests that were effectuected. This step required a lot of trial and error and ultimately, testing against the existing Speedgoat solution.

## 4.1 Implementation of the Real-Time Control Loop

The core of the firmware is the `control_step` function, which executes the following steps:

- **Acquisition and Normalization:** The 12-bit (0-4095) ADC value is converted to a floating-point voltage. The raw signal is normalized based on the reference voltage ($V_{REF}$).
- **DC Removal:** Because the microphone signal is biased at mid-rail, a recursive DC estimator tracks the bias:

$$dc\_bias[n] = 0.999 \cdot dc\_bias[n-1] + 0.001 \cdot V_{raw}[n]$$

  This bias is subtracted to isolate the AC pressure component ($V_{in\_ac}$).
- **DSP Filtering:** The centered voltage is scaled by the microphone sensitivity (`sens_p`) and processed by the CMSIS-DSP Biquad function:

```
arm_biquad_cascade_df2T_f32(&S, &input_f32, &output_f32, 1);
```

  This implements the second-order impedance synthesis derived in Section 2.
- **Signal Conditioning:** To ensure stability and hardware safety, a first-order low-pass filter (LPF) is applied to the output. Furthermore, a hard limit of 90% of the dynamic range is enforced to prevent rail-clipping.
- **DAC Formatting:** Since the DAC is single-ended, the bipolar command signal is re-centered by adding $V_{REF}/2$ before being converted back to a 12-bit integer (0-4095) for the hardware output.

## 4.2 Initial Functional Tests

Initial "open-air" tests were conducted to verify the basic operation of the microphone acquisition and speaker actuation. This setup confirmed that the ADC and DAC peripherals were correctly initialized and that the software could process acoustic signals in real-time.

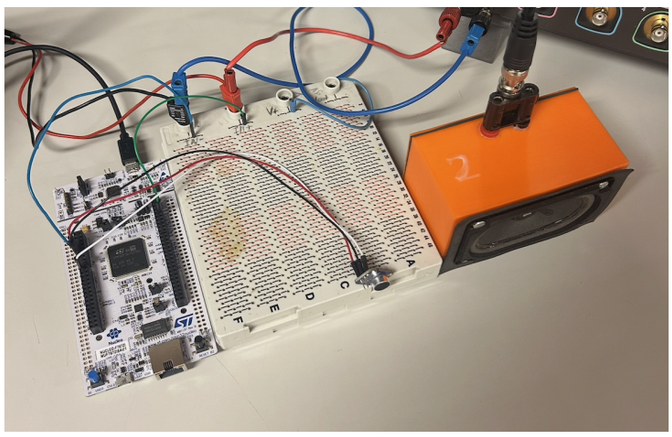


Figure 6: Experimental setup for open-air functional verification.

Following these preliminary tests, the system was installed in a tube to simulate controlled acoustic conditions. However, initial results indicated discrepancies between the simulated and measured acoustic responses, necessitating a deeper analysis of the internal transfer function.

### 4.3 Transfer Function Validation with Brüel & Kjær PULSE

To isolate the digital filter's behavior, the system was then tested using the Brüel & Kjær PULSE analyzer to measure the Input/Output transfer function. These measurement led to the discovery of a critical flow: the filter was using continous coeficients instead of the discretised version.

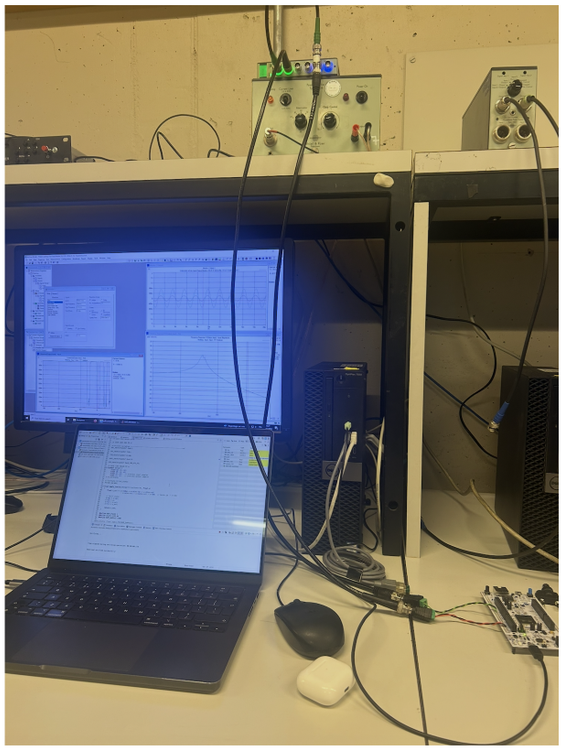


Figure 7: Hardware-in-the-loop testing using the Brüel & Kjær PULSE system.

Some other design flaws were later discovered, including:

- **Dynamic Range and Output Clipping:** It was observed that the DAC output was clipping, which distorted the control signal.
- **DAC DC Offset:** Because the integrated DAC operates on a single-ended supply (0–3.3V) and cannot produce negative voltages, a DC offset was implemented in the software. This allowed for a full sinusoidal centered within the DAC's dynamic range, preventing lower-rail clipping.

- **Gain Discrepancies:** Significant issues were identified regarding the input and output gain staging.

### 4.4 Discretisation of the filter coeficients

In order to get the same results as the existing laboratory Speedgoat reference, the filter coefficients where first discretised using the Tustin (bilinear) transform. The discretisation was initially modeled in Matlab using the following logic:

```
% Continuous transfer function
Phi_c = tf(num_c, den_c);

% Tustin transform
Phi_d = c2d(Phi_c, Ts, 'tustin');

% Discrete coefficients
[num_d, den_d] = tfdata(Phi_d, 'v');
```

Once the discrete coefficients ($b_0, b_1, b_2$ and $a_1, a_2$) are obtained, they could be integrated into the firmware code. As mentioned earlier, the filter in itself is performed in real-time using the CMSIS-DSP library to leverage the hardware Floating Point Unit (FPU) of the Cortex-M7:

```
arm_biquad_cascade_df2T_f32(&S, &input_f32, &output_f32, 1);
```

We will not cover the specific implementation of this function, as it goes beyond the scope of this report; the curious reader can however, read the documentation, or the fucntion's source code, for more informations.

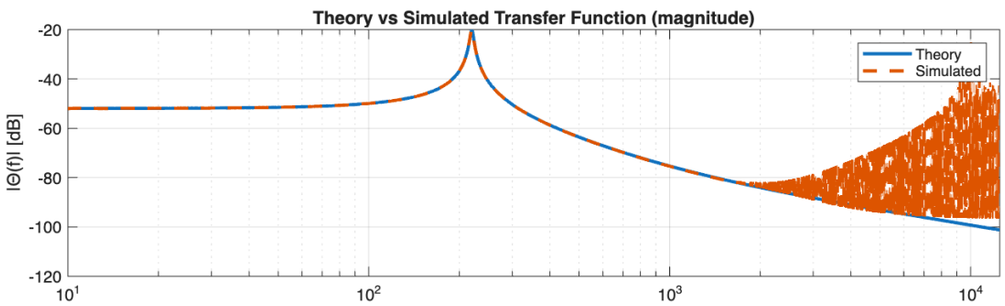



Figure 8: Verifying the discretised coefficients against the theoretical target.

Review Explanation In order to verify the hardware implementation, the control law was validated through numerical simulations in Matlab in order to compare against it. Using discretised parameters, which provided the expected simulation results. This confirmed the theoretical derivation of the second-order filter before it was ported to the embedded environment.

### 4.5 Timer-Driven Control Loop

A critical realization during the firmware development was that executing the control law inside the main program loop leads to non-deterministic sampling intervals. In a typical embedded application, the `while(1)` loop executes as fast as possible and may be interrupted by background tasks, peripheral servicing, or blocking instructions. As a result, calling the control function in this loop does not guarantee a fixed sampling period.

The initial implementation followed this approach:

```
while (1)
{
    control_step();
}
```

However, this structure caused jitter in the effective sample time, which led the previous firmware implementation to a mismatch in sampling frequency, between the one used to discretised the filter coefficients and the actual sampling frequency of the control system (which was therefore not fixed). This of course directly impacted the stability and frequency response of the control.

To ensure a strictly deterministic sampling frequency, the control loop was instead executed inside a hardware timer interrupt. Timer 6 (TIM6) was configured to generate periodic interrupts at the desired sampling frequency $f_s$, and the control step was executed exclusively within the corresponding interrupt service routine (ISR).

In the main program initialization, the interrupt is configured as follows:

```
HAL_NVIC_SetPriority(TIM6_DAC_IRQn, 0, 0); // Highest priority
HAL_NVIC_EnableIRQ(TIM6_DAC_IRQn);         // Enable interrupt
```

The timer callback function provided by the HAL library is then used to execute the control algorithm at each timer update event:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM6)
    {
        control_step();
    }
}
```

Finally, the corresponding interrupt handler is enabled in `stm32f7xx_it.c`, ensuring proper propagation of the hardware interrupt to the HAL layer:

```
/**
  * @brief TIM6 global interrupt handler.
  */
void TIM6_DAC_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim6);
}
```

This interrupt-driven architecture guarantees a constant and precisely controlled sampling period, which is essential to preserve the theoretical behavior of the discretised filter.

### 4.6 Comparative Analysis: MCU vs. Speedgoat

The final stage of testing involved a side-by-side comparison with the existing Speedgoat reference. This led to the final realization, that the filter coeficients where actually in the wrong order.

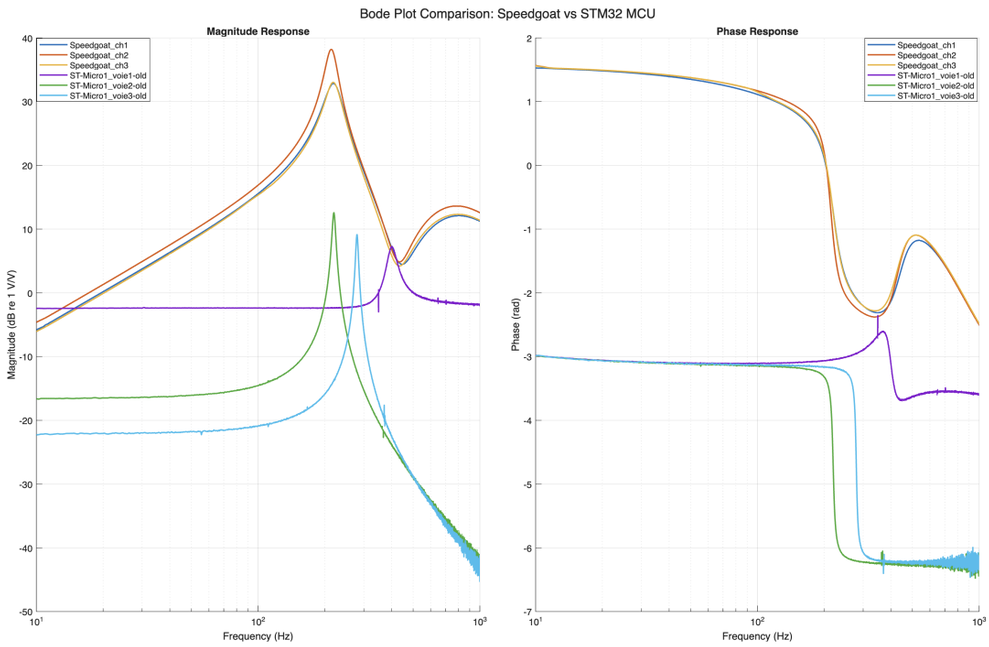



Figure 9: Initial broken implementation against the Speedgoat reference.

```
# In the parameters computation
def get_physics_params(values):
    p = Params()
    ...
    # Denominator
    p.b2 = p.muM * p.Mms * p.Cmc
    p.b1 = p.muR * p.Rms * p.Cmc
    p.b0 = p.muC

    # Numerator
    p.a2 = (p.muM - 1.0) * p.Mms * p.Cmc
    p.a1 = (p.muR - 1.0) * p.Rms * p.Cmc
    p.a0 = (p.muC - 1.0)

    # Instead of the appropriate:
    p.b0 = p.muM * p.Mms * p.Cmc
    p.b1 = p.muR * p.Rms * p.Cmc
    p.b2 = p.muC

    p.a0 = (p.muM - 1.0) * p.Mms * p.Cmc
    p.a1 = (p.muR - 1.0) * p.Rms * p.Cmc
    p.a2 = (p.muC - 1.0)

    return p

# And in the filter's coefficients computation
def compute_filter_coeffs(params):
    num_c = [
        scaling * params.a2,
        scaling * params.a1,
        scaling * params.a0
    ]
    den_c = [params.b2, params.b1, params.b0]
```

```
# Instead of the expected order:
num_c = [
    scaling * params.a0,
    scaling * params.a1,
    scaling * params.a2
]
den_c = [params.b0, params.b1, params.b2]
```

With the new, adjusted filter settings, we got the following resulsts which are much more promising:

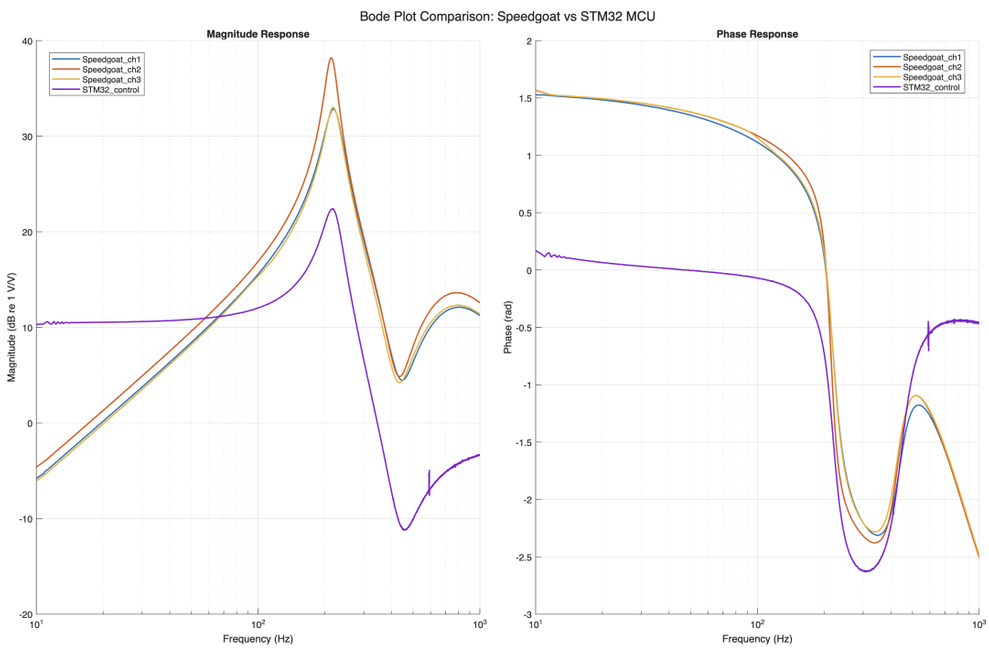



Figure 10: Fixed filter implementation against the Speedgoat reference.

The results indicated that the MCU-based controller is capable of partially replicating the Speedgoat's output signal. While the current firmware demonstrates a successful proof-of-concept, further adjustments to the input/output gain and signal conditioning are required to achieve the same results as the Speedgoat benchmark. Ongoing work focuses on fine-tuning the I/O gain and offset needed.

Figure 11: Experimental setup for live tests in a reverberant room.

# 5 Custom GUI Solution

## 5.1 Motivation and Architecture

Early firmware iterations relied on hard-coded control parameters directly embedded in the C source code. This approach was impractical, as every change to physical or control parameters required manual code modifications after discretising the parameters in Matlab.

To address this limitation, a custom Python Graphical User Interface (GUI) was developed. The GUI acts as an abstraction layer between the acoustic modeling domain and the embedded build system, allowing physical and control parameters to be modified without directly interacting with the firmware source code or the STM32CubeIDE environment.

Architecturally, the GUI orchestrates parameter definition, automated coefficient computation, firmware generation, and board programming, while the underlying embedded software structure remains unchanged.

## 5.2 Installation and Usage

The application runs inside a dedicated Python virtual environment (`.venv`) and is compiled into standalone executables for macOS and Windows through a GitHub Actions workflow. This process bundles the GUI, firmware, and STM32 toolchain together into platform-specific installers (`.dmg` and `.exe`). Upon download, the user can follow the installation wizard and subsequently launch the application (the initial startup may take some time).

The GUI provides structured input fields for all relevant physical and control parameters, grouped according to their role in the system:

- **Transducer Parameters:** Effective diaphragm surface $S_d$, force factor $Bl$, mechanical resistance $R_{ms}$, moving mass $M_{ms}$, and compliance $C_{mc}$.
- **Control Targets:** Target resonance frequency $f_{\text{target}}$ and tuning factors $\mu_M$ and $\mu_R$.
- **Calibration Parameters:** Microphone sensitivity (`sens_p`) and output gain (`i2u`).

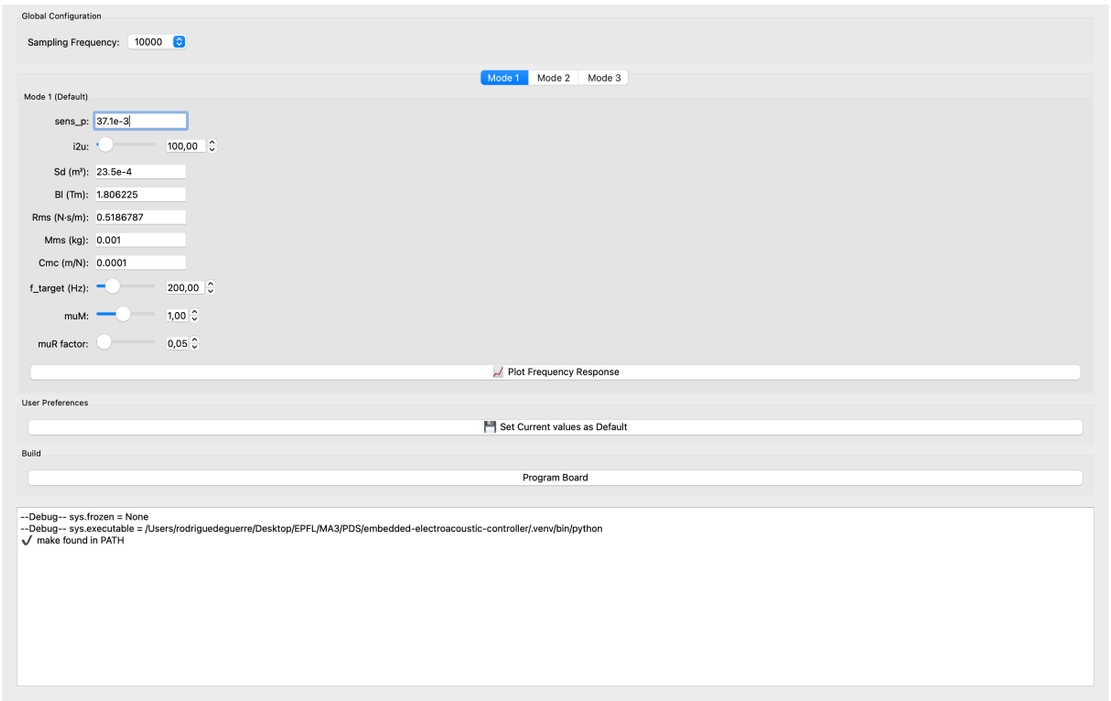



Figure 12: Custom Python-based GUI used for parameter configuration and board programming.

To build and flash the firmware onto the board, the user can simply click on the **Program Board** button. This action saves all currently defined parameters, builds the firmware, and flashes it onto the target board.

If more granular control is required, the build process can be split into distinct steps by selecting **Show Build Actions** from the **Settings** menu:

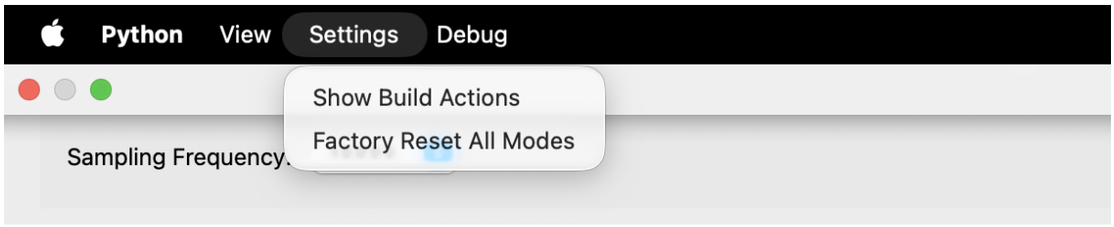



Figure 13: Settings menu providing access to build actions and the factory reset utility.

From this menu, the user can also reset all modes to their default factory values.
When using the advanced build actions, the following workflow applies:

- Pressing **Save all parameters** updates the generated header file `generated_params.h`.
- Pressing **Build Firmware** triggers the firmware compilation using the embedded toolchain.
- Pressing **Flash Firmware** uploads the compiled binary to the target board.
- Pressing **Clean Build Folder** deletes all previously generated build artifacts and resets the build directory.

*Note that it is important to ensure that the board is properly connected using a micro-USB data cable before flashing the firmware.*

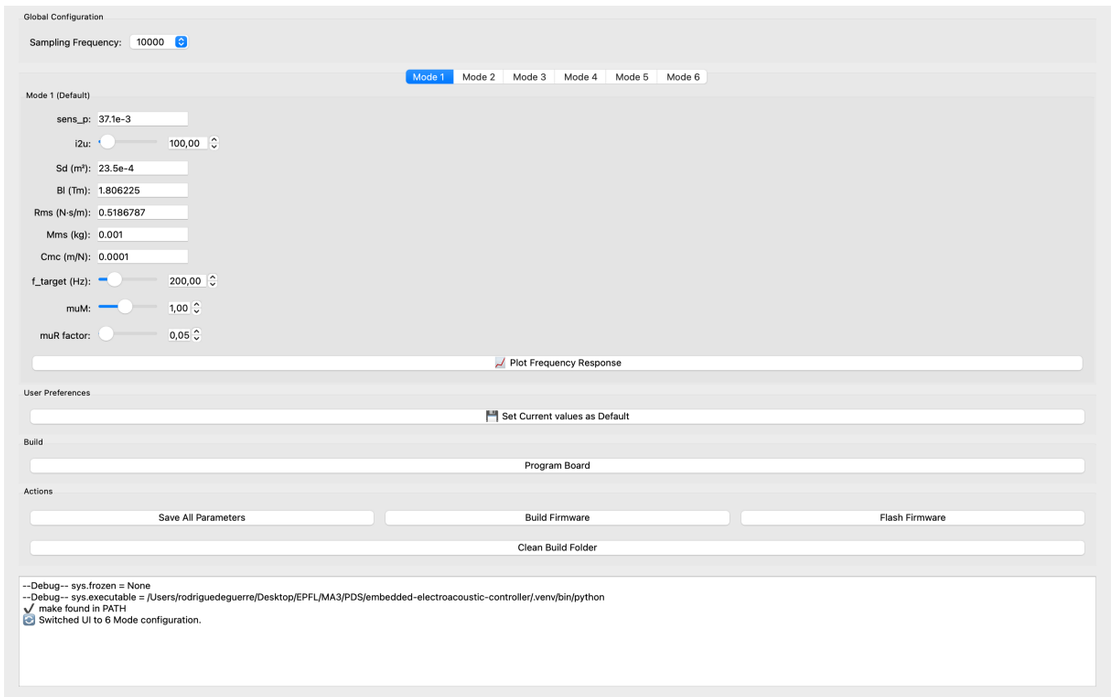



Figure 14: Extended GUI showing integrated user preferences and build action panels.

As shown in Fig. 14, the GUI also allows the solution to be built using either three or six operating modes (in addition to an always-included off mode). The desired configuration can be selected by enabling the `6 Modes` option from the `View` menu.

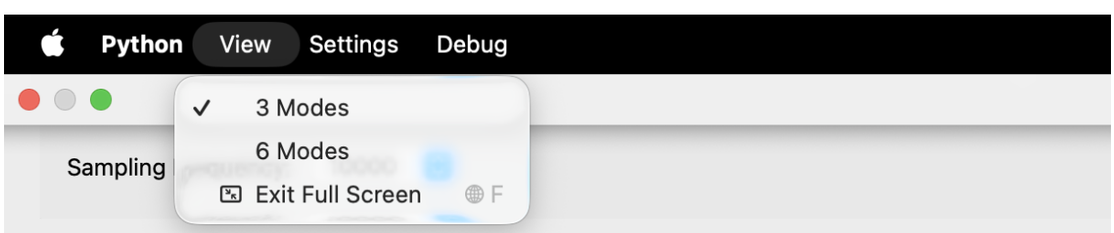



Figure 15: Dynamic switching between three- and six-mode configurations.

## 5.3 Python-Based Automated Parameter Workflow

With the graphical interface, a Python workflow was developed to automate the computation of digital control parameters. The script loads all user-defined parameters from the GUI, computes the continuous-time filter coefficients, discretises them following the same methodology previously applied in Matlab, and finally exports the results into a generated header file within the firmware project.

To support validation, the tool also provides the ability to plot the expected discrete-time transfer function prior to deployment. This enables direct comparison between the theoretical target impedance and the expected digital implementation.

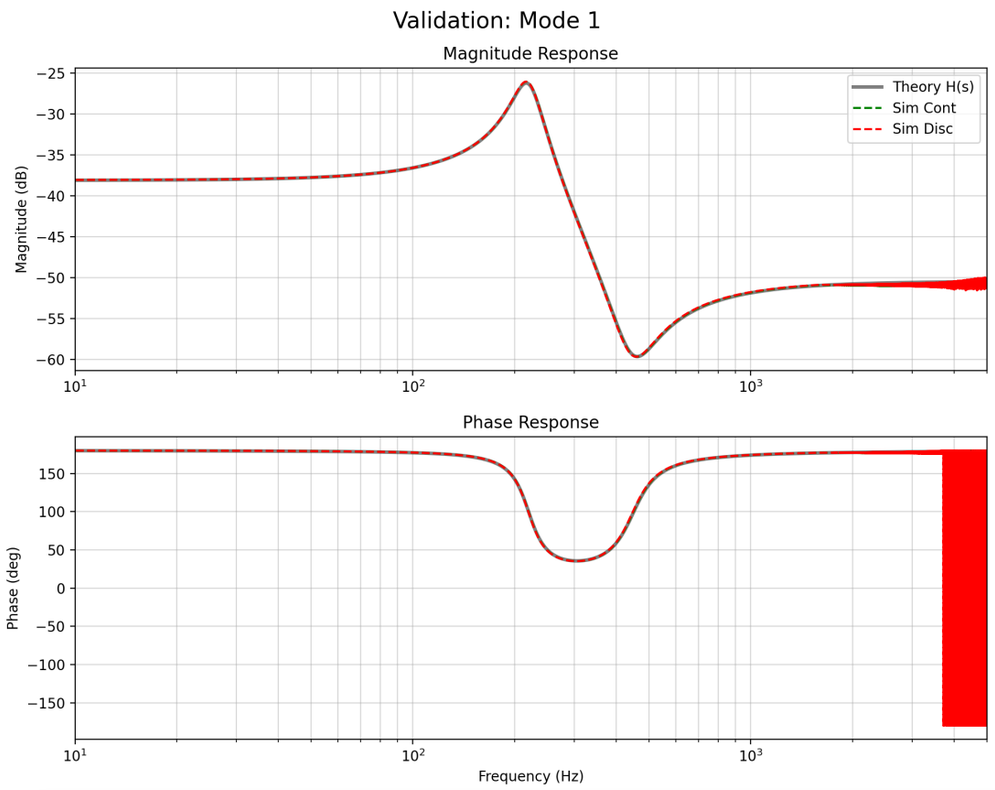



Figure 16: Python-based validation of the discretised transfer function against the theoretical target.

This integrated workflow significantly improves reproducibility, traceability, and iteration speed during experimentation.

## 5.4 Application Build and Distribution Workflow (CI)

To ensure reproducibility and simplify application distribution, an automated build workflow has been implemented using GitHub Actions. This workflow is responsible for building and packaging the GUI application for both Windows and macOS platforms, producing standalone `.exe` and `.dmg` installers.

The workflow is triggered when a new version tag is released in the repository, thereby avoiding unnecessary rebuilds on every commit. During this process, the application is compiled and packaged into platform-specific installers, which are automatically attached to the corresponding GitHub release. This ensures that distributed binaries are directly traceable to a specific, tagged version of the source code.

Due to certificate availability constraints, the generated binaries are not signed using commercially trusted code-signing certificates. As a result, the operating system may display a security warning when the application is launched for the first time. This limitation does not affect the functionality of the software but requires explicit user confirmation before execution.

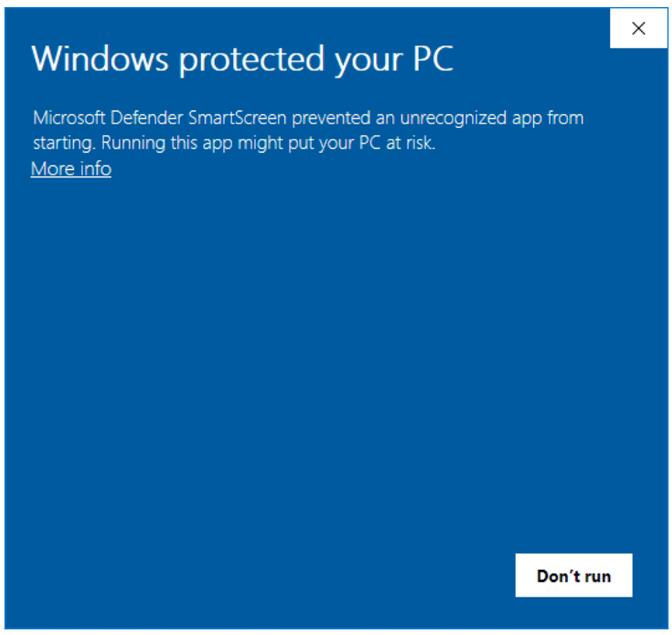



Figure 17: Windows security warning displayed for unsigned application binaries.

To proceed past the warning shown in Fig. 17, the user must select *More info* and then *Run anyway*.

As the application is intended primarily for academic use and has no commercial objective, obtaining a commercially trusted code-signing certificate would not be economically justified, as such certificates typically incur annual costs of several hundred CHF when issued by recognized certificate authorities [5]. As a possible improvement, the use of a self-signed certificate could be considered. However, self-signed certificates are not trusted by default by operating systems [6], and reputation-based mechanisms such as Microsoft SmartScreen rely on applications gradually accumulating trust through repeated downloads and usage, meaning that security warnings may still be displayed for newly distributed binaries [7].

# 6 Conclusion

This semester project explored the feasibility of implementing an impedance synthesis control law for electroacoustic resonators on a stand-alone microcontroller platform.

Starting from a physical model of the loudspeaker, a second-order control law was derived, discretised using the bilinear transform, and implemented in real time on an STM32F767ZI microcontroller. A timer-driven interrupt architecture and optimized DSP routines were employed to ensure deterministic sampling and computational efficiency. In parallel, a Python GUI was developed to streamline parameter configuration, discretisation, and firmware deployment.

Experimental validation, including hardware-in-the-loop measurements and comparison with a Speedgoat Real-Time Target Machine, confirmed that, although some discrepancies in gain scaling and signal conditioning remain, the embedded controller has the potential to reproduce the expected qualitative behavior of the reference implementation.

Overall, the results demonstrate that a microcontroller-based solution is a viable alternative to laboratory rapid-prototyping systems, such as the Speedgoat Real-Time Target Machines, for electroacoustic impedance synthesis at low frequencies. The developed firmware architecture and software tools provide a foundation for future work, which should focus on improved analog interfacing, calibration procedures, and extended performance evaluation under controlled acoustic conditions.

## References


[1] E. T. J.-L. RIVET, "Room modal equalisation with electroacoustic absorbers," Master Thesis, EPFL, 2016.

[2] M. VOLERY, "Robust tunable acoustic impedance control on electroacoustic resonators for aircraft noise reduction," Master Thesis, EPFL, 2023.

[3] STMicroelectronics, *Stm32 datasheet*, UM1727, STMicroelectronics, 2016. [Online]. Available: `https://www.st.com/en/evaluation-tools/nucleo-f767zi.html#documentation`.

[4] STMicroelectronics, *Getting started with stm32 nucleo board software development tools*, UM1727, STMicroelectronics, 2016.

[5] DigiCert, Inc., *Code signing certificates – instilling trust in your software*, 2024. [Online]. Available: `https://www.digicert.com/code-signing`.

[6] Wikipedia =, *Self-signed certificate*, 2025. [Online]. Available: `https://en.wikipedia.org/wiki/Self-signed_certificate`.

[7] Wikipedia, *Microsoft smartscreen*, 2025. [Online]. Available: `https://en.wikipedia.org/wiki/Microsoft_SmartScreen`.

[8] M. F. PADLEWSKI, "Active acoustic metamaterials: A new way to understand nonlinear and topological phenomena," Master Thesis, EPFL, 2025.

[9] STMicroelectronics. "Getting started with dac," Accessed: Jan. 2, 2025. [Online]. Available: `https://wiki.st.com/stm32mcu/wiki/Getting_started_with_DAC`.

[10] STMicroelectronics. "Getting started with adc," Accessed: Jan. 2, 2025. [Online]. Available: `https://wiki.st.com/stm32mcu/wiki/Getting_started_with_ADC`.

[11] ARM Ltd. "Cmsis-dsp software library," Accessed: Jan. 2, 2025. [Online]. Available: `https://github.com/ARM-software/CMSIS-DSP`.

[12] STMicroelectronics. "Stm32cubef7 firmware package," Accessed: Jan. 2, 2025. [Online]. Available: `https://github.com/STMicroelectronics/STM32CubeF7`.

[13] STMicroelectronics Community. "How to integrate cmsis-dsp libraries on an stm32 project," Accessed: Jan. 2, 2025. [Online]. Available: `https://community.st.com/t5/stm32-mcus/how-to-integrate-cmsis-dsp-libraries-on-a-stm32-project/ta-p/666790`.

[14] STMicroelectronics Community. "Configuring dsp libraries on stm32cubeide," Accessed: Jan. 2, 2025. [Online]. Available: `https://community.st.com/t5/stm32-mcus/configuring-dsp-libraries-on-stm32cubeide/ta-p/49637`.

[15] sourceforge. "Make for windows," Accessed: Jan. 1, 2026. [Online]. Available: `https://gnuwin32.sourceforge.net/packages/make.htm`.