CHEMSAFE TECHNOLOGY

**INDUSTRIAL AIR QUALITY MONITORING SYSTEM FOR CHEMICAL LEAK DETECTION**

SAKTHIVEL BALAKUMAR

**ABSTRACT**

Gas leakage is one factor in a broader range of issues contributing to economic instability, including climate change, the energy crisis, and environmental degradation. Findings from the recent investigation indicate pipeline leakage occurs every 40 hours on average. From 2010 to 2021, nearly 2,600 significant leaks were reported, resulting in almost 600 injuries and over 125 fatalities.

Recently, on September 20, 2024, a severe gas leak occurred at the chemical factory in Ambarnath city in Maharastra, which triggered a wave of health concern. Even now, local residents are reporting multiple health issues, including breathing difficulties, eye irritation, and throat discomfort. This incident highlights the need for obligatory safety measures and emergency preparedness in industrial settings.

This project focuses on developing a comprehensive gas leakage detection system designed to detect and alert to chemical leaks in real-time. The system's wireless communication capabilities enable remote monitoring and automatic reporting to concerned parties. This predictive approach enhances preventive maintenance strategies, reducing operational costs and minimizing environmental impact.

**V CYCLE**

The V-Cycle is a model used in product development, especially in engineering and systems development. It is particularly popular in fields like software engineering, systems engineering, and project management. The V-Cycle is often depicted in a V-shaped diagram, illustrating the relationship between development phases and corresponding verification and validation activities.

The key stages of the V-Cycle are as follows:

1. **Requirements Analysis**

The left side of the V represents the stages of specifying requirements for the product. This phase involves gathering user needs, functional requirements, and constraints.

* Functional Requirements:
* Measure ambient temperature, sound levels, and CO2 concentration.
* Transmit data via Bluetooth to a smartphone.
* Non-Functional Requirements:
* Low power consumption.
* Minimal carbon footprint.
* User-friendly smartphone interface.
* Low development cost

2. **System Design**

Creating a high-level design based on the requirements. This includes defining system architecture and components.

* Architecture:
* Identification of components
* Tiva™TM4C123GH6PM Microcontroller with Bluetooth capability is used.
* Select appropriate sensors (e.g., DHT22 for temperature, sound sensor module for sound levels, MH-Z19 for CO2).
* Power Management:
* Communication Protocol:
* Define a Bluetooth Low Energy (BLE) protocol for data transfer.

3. **Detailed Design**

* Sensor Integration:
* Firmware Development:
  + Tri Colour LED blinking with delay
  + Write firmware for data acquisition, processing, and Bluetooth communication
* Mobile Application:

4. **Implementation**

* Hardware Development:
* Preparation of Electronic board [Soldering]
* Assemble the prototype with chosen sensors and microcontroller.
* Software Development:
* LED Illumination Response to Voltage Variations and displaying in PuTTY console
* LED Illumination Response to Voltage Variations and use the Bluetooth link to communicate with a smartphone
* Displaying the LOGO
* Implement firmware on the microcontroller.
* Develop the mobile application for data visualization.

5. **Testing and Validation**

* Unit Testing:
* Test individual components for functionality.
* Integration Testing:
* Performance Testing:
* The current consumed by individual LED light (RED – GREEN -BLUE - YELLOW) is measured.
* Maximum current provided by a digital port of the microcontroller at level 1 and at level 0.
* User Acceptance Testing:

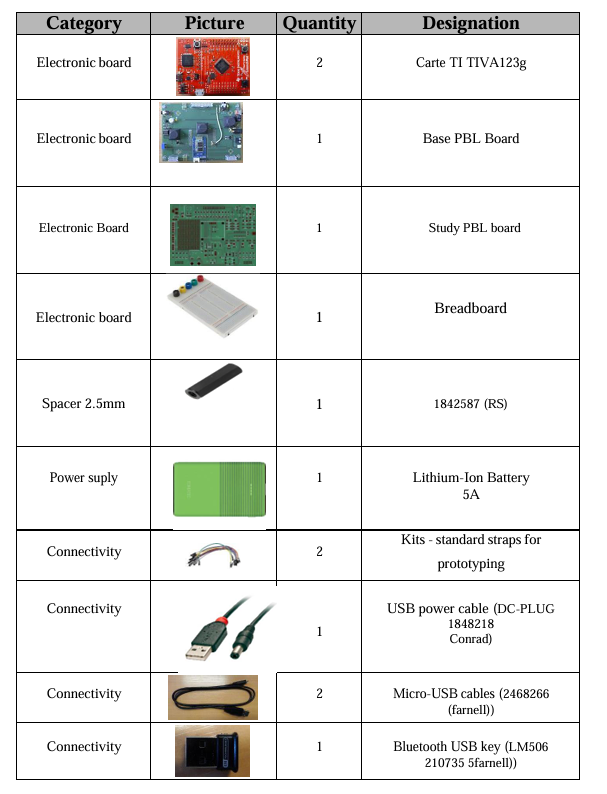
6. **Evaluation**

7. **Deployment**

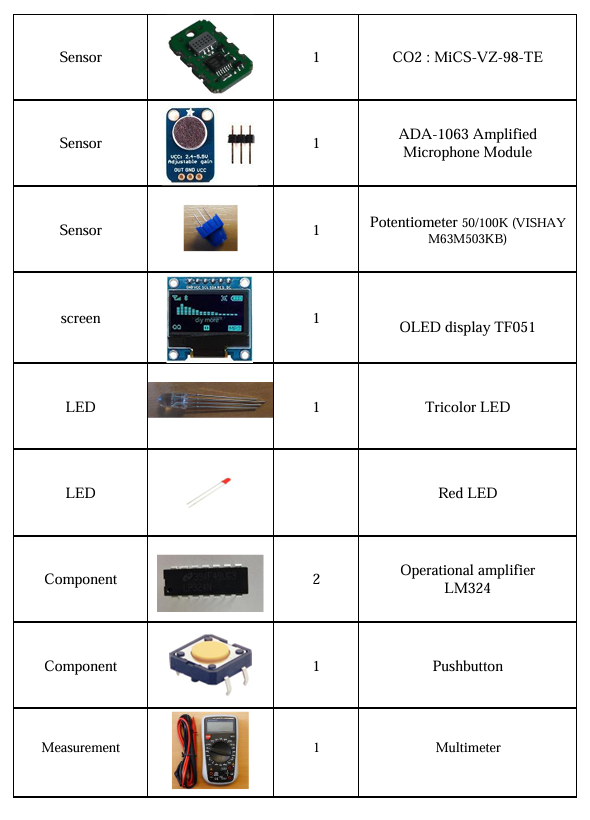
8. **Presentation**

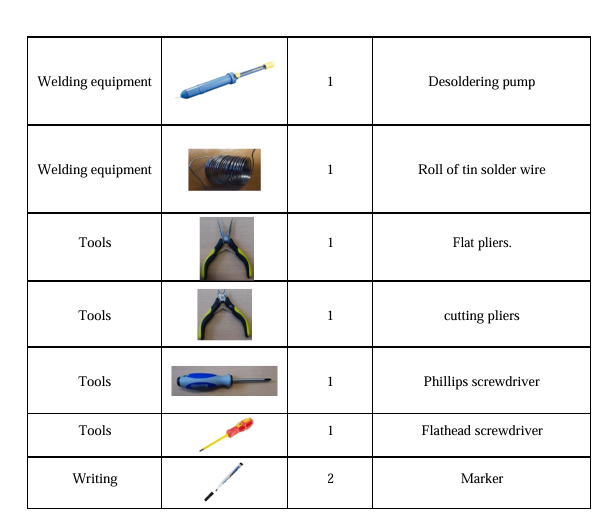
* Prepare a comprehensive presentation of the project:
* Overview of the system and its purpose.

**IDENTIFICATION OF COMPONENTS**









**STUDY OF TIVA ™TM4C123GH6P MICROCONTROLLER PARAMETERS**

**Size of the Memories**

The TM4C123GH6PM microcontroller features multiple types of memory, each with a distinct purpose:

1. **Flash Memory:**
   * **Size:** 256 KB (Kilobytes)
   * **Purpose:** Flash memory is non-volatile, meaning it retains its content even when power is lost. It is used to store the program code and constants. Developers can write data to Flash memory when needed, although it is slower to write to than other memory types like RAM.
2. **SRAM (Static RAM):**
   * **Size:** 32 KB
   * **Purpose:** SRAM is volatile memory, meaning it loses its data when the power is off. It is used to store data temporarily during program execution (e.g., stack, variables, buffers). It offers faster access than Flash memory, making it ideal for tasks requiring quick data access.
3. **ROM (Read-Only Memory):**
   * **Size:** 2 KB
   * **Purpose:** The ROM contains a bootloader and basic routines that assist in loading and running software. It is used to initialize the system and contains routines that help interface with external hardware.
4. **EEPROM (Electrically Erasable Programmable Read-Only Memory):**
   * **Size:** 2 KB
   * **Purpose:** EEPROM is non-volatile memory used for storing configuration settings, calibration data, or other values that need to be retained after power-off but may change occasionally. It is slower to access than SRAM but allows more write cycles compared to Flash.

**Clock Frequency**

The clock frequency represents the speed at which the microcontroller executes instructions. In the case of the TM4C123GH6PM, the system clock can run up to **80 MHz**, meaning the microcontroller can execute up to 80 million instructions per second, depending on the specific operation. This frequency is adjustable using internal clock modules (like the PLL or Precision Internal Oscillator), and in many cases, the actual clock speed can be scaled down to reduce power consumption.

The clock frequency is critical for determining the performance and response time of the microcontroller. It affects how fast peripheral devices are accessed, how quickly calculations are made, and how real-time constraints are handled.

**Utility of a Floating-Point Unit (FPU)**

The TM4C123GH6PM microcontroller includes a **single-precision floating-point unit (FPU)**. The FPU accelerates calculations that involve floating-point numbers (i.e., numbers with decimals, such as 3.14 or 2.718). Without an FPU, floating-point operations would have to be emulated using integer arithmetic, which can be slower and more complex.

**Key benefits of an FPU:**

1. **Increased Performance:** The FPU performs floating-point arithmetic (addition, subtraction, multiplication, division) much faster than integer emulation.
2. **Precision:** Floating-point calculations allow for more precision in calculations, which is important for applications like control systems, signal processing, and scientific computations.
3. **Reduced Code Complexity:** Without an FPU, implementing floating-point operations would require additional software routines, increasing both the code size and complexity.

In applications like motor control, robotics, or digital signal processing (DSP), where complex calculations (such as trigonometric functions) are frequently performed, the FPU is particularly valuable.

**Embedded Peripherals**

The TM4C123GH6PM microcontroller is rich in peripherals, which are embedded hardware modules that enhance the functionality of the microcontroller by interfacing with the external world. Some key embedded peripherals include:

1. **Timers (General-Purpose and PWM Timers):**
   * **Utility:** Used for generating time delays, creating precise time intervals, or generating periodic events. PWM (Pulse Width Modulation) timers are crucial for controlling motors or adjusting the brightness of LEDs by varying the duty cycle of the output signal.
2. **Analog-to-Digital Converter (ADC):**
   * **Resolution:** 12-bit
   * **Utility:** The ADC converts analog signals (e.g., from sensors like temperature or light sensors) into digital values that the microcontroller can process. The 12-bit resolution means that it can represent input signals as one of 4096 discrete digital values.
3. **Universal Asynchronous Receiver-Transmitter (UART):**
   * **Utility:** Used for serial communication between the microcontroller and other devices such as PCs, GPS modules, or other microcontrollers. It enables asynchronous communication over two lines (transmit and receive).
4. **Inter-Integrated Circuit (I2C):**
   * **Utility:** I2C is a multi-master, multi-slave serial bus used to connect low-speed peripherals (e.g., sensors, memory devices) to the microcontroller. It is widely used due to its simplicity and ability to connect multiple devices using just two wires (SCL and SDA).
5. **Serial Peripheral Interface (SPI):**
   * **Utility:** SPI is a faster, synchronous serial communication protocol that supports higher data rates than I2C. It is used to interface with high-speed peripherals like flash memory, display controllers, or sensors requiring high-speed communication.
6. **GPIO (General-Purpose Input/Output):**
   * **Pins:** 43 GPIO pins
   * **Utility:** GPIO pins are used to interface with external devices like buttons, LEDs, sensors, and other digital devices. They can be configured as either inputs or outputs and are highly flexible in operation.
7. **USB 2.0 Full-Speed Device/Host/OTG:**
   * **Utility:** The microcontroller supports USB communication, making it capable of acting as a USB host (to control other USB devices), a USB device (to be controlled by a PC or other device), or using OTG (On-The-Go) functionality, allowing it to switch between host and device modes.
8. **CAN (Controller Area Network):**
   * **Utility:** CAN is used in automotive and industrial applications for robust communication between multiple devices or microcontrollers in harsh environments. It is a fault-tolerant, high-speed communication protocol.

**SETTING UP OF ENERGIA CONSOLE**

When using the Tiva™ TM4C123GH6PM microcontroller with Energia IDE, we need to select the correct board and port for uploading code to the microcontroller.

**Steps for Port and Board Selection in Energia**

1. Install Energia IDE

2. Connect the Tiva C LaunchPad to Your Computer

* Connect the Tiva™ TM4C123GH6PM microcontroller to your computer via a USB cable.
* Ensure the drivers for the Tiva LaunchPad are installed. These are often installed automatically, but if needed, you can download them manually.

3. Select the Correct Board in Energia

* In Energia, go to Tools > Board.
* From the list, select "LaunchPad (Tiva C) w/ TM4C123 (80MHz)".

4. Select the Correct Port in Energia

* Go to Tools > Port.
* Select the COM port that corresponds to the Tiva™ LaunchPad.

5. Upload Code

* After selecting the correct board and port, write your code in the Energia IDE.
* Click the Upload button (the right arrow) to compile and upload the code to the Tiva™ TM4C123GH6PM microcontroller.

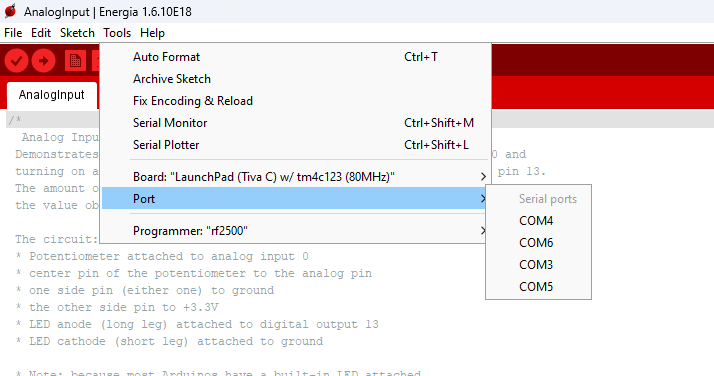


Figure 2 : Port and Board selection

**SETTING UP PuTTY CONSOLE**

1. Install Putty

2. Identify the Serial Port (COM Port)

* + Right-click the Start menu and select Device Manager and expand the Ports (COM & LPT) section.
  + When the Tiva LaunchPad is connected, device will be listed as "USB Serial Device (COMx)", where x is the COM port number (e.g., COM3, COM4).

3. Configure PuTTY for Serial Communication

* + Launch PuTTY
  + Set the Connection Type to Serial
  + In the Serial line field, enter the COM port
  + Set the Speed (baud rate) to 9600

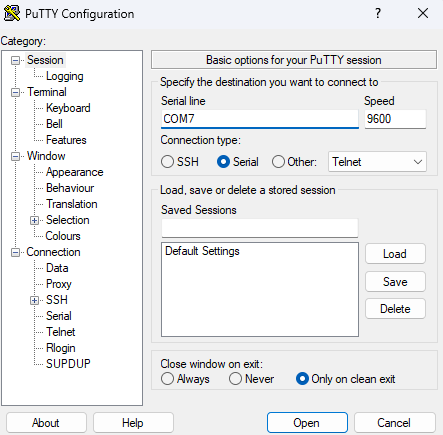


Figure 3 : PuTTY configuration

1. Click Open to start the serial connection.
2. Upload Code to the Tiva Microcontroller and monitor the Output in PuTTY

**SETTING UP OF BLUETOOTH MODULE**

1. Connect the Bluetooth module to the TM4C123GH6PM microcontroller.
2. Serial1 Configuration:

* The Serial1 port is used instead of Serial for Bluetooth communication on microcontrollers like the Tiva™ TM4C123GH6PM because each of these ports corresponds to a different UART (Universal Asynchronous Receiver-Transmitter) interface, and they serve different purposes in communication setups.
* Serial typically refers to the default UART interface on a microcontroller. On many development boards, like the TM4C123GH6PM, the Serial port is often used for communication between the microcontroller and a computer via USB or other standard serial communication channels.
* Serial1 refers to a secondary UART interface, which corresponds to another set of TX (transmit) and RX (receive) pins on the microcontroller. This port is separate from the default Serial interface and can be used for other communication purposes, such as connecting external modules like Bluetooth devices.

1. Bluetooth Pairing:

* Power on the electronic board
* Turn on Bluetooth on your smartphone and make your phone discoverable if needed.
* Use a terminal emulator on your phone to communicate with the microcontroller and verify that data is being sent and received properly.
* Scan for available devices and select the G07b Bluetooth module when it appears.
* Enter the default passcode (usually 1234 or 0000) when prompted.

1. Terminal Emulator Setup:
   * Open the Serial Bluetooth Terminal app on your smartphone.
   * In the app, tap the "Connect" or "Devices" button (usually in the top right corner of the app). This will bring up a list of paired Bluetooth devices.
   * In the list of devices, find and select your G07b Bluetooth module.

**LM35 TEMPERATURE SENSOR**

The LM35 series are precision integrated-circuit temperature devices with an output voltage linearly-proportional to the Centigrade temperature. The LM35 device has an advantage over linear temperature sensors calibrated in Kelvin, as the user is not required to subtract a large constant voltage from the output to obtain convenient Centigrade scaling. The LM35 device does not require any external calibration or trimming to provide typical accuracies of ±¼°C at room temperature and ±¾°C over a full −55°C to 150°C temperature range. Lower cost is assured by trimming and calibration at the wafer level. The low-output impedance, linear output, and precise inherent calibration of the LM35 device makes interfacing to readout or control circuitry especially easy. The device is used with single power supplies, or with plus and minus supplies. As the LM35 device draws only 60 µA from the supply, it has very low self-heating of less than 0.1°C in still air. The LM35 device is rated to operate over a −55°C to 150°C temperature range.

**PARAMETERS**

|  |  |
| --- | --- |
| **PARAMETERS** | **RANGE** |
| Local sensor | 0.5, 1 |
| Operating temperature range (°C) | -55 to 150 |
| Supply voltage (min) (V) | 4 |
| Supply voltage (max) (V) | 30 |
| Supply current (max) (µA) | 114 |
| Interface type | Analog output |
| Sensor gain (mV/°C) | 10 |

**MiCS-VZ-89TE – CO2 SENSOR**

The MiCS-VZ-89TE is a semiconductor-based gas sensor that is designed to detect air contaminants such as volatile organic compounds (VOCs) and provide a CO₂ equivalent (CO₂eq) measurement. It is commonly used for indoor air quality monitoring applications due to its sensitivity to a wide range of gases, including harmful volatile organic compounds and carbon dioxide equivalents. The sensor can detect both total VOCs (tVOCs) and CO₂ equivalents. It can measure 400 to 2000 ppm equivalent CO₂, providing a measure of indoor air quality and 0 to 1000 ppb (parts per billion) of isobutylene equivalent, representing the total volatile organic compounds in the air.

**PARAMETERS**

|  |  |
| --- | --- |
| **PARAMETER** | **DETAILS** |
| |  |  | | --- | --- | | Detection Method |  | | Semiconductor gas sensor, detecting a wide range of VOCs |
| Monitoring Range | |  |  | | --- | --- | |  | 400-2000 ppm equivalent CO₂  0-1000 ppb isobutylene equivalent tVOCs | |
| Output Types | |  |  | | --- | --- | |  | PWM Output  I²C Output | |
| PWM Output Specifications | TTL output  30 Hz ± 1% frequency  Range: 5–95% duty cycle, 3.3V |
| Pull-up Resistor for PWM (Pin 1) | 10 kΩ (for 3.3V operation), between Pin 1 and Pin 6 |
| Pull-up Resistor for I²C (Pins 2, 4) | 4.7 kΩ on master SDA and SCL |
| Response Time (tVOC) | < 5 seconds |
| Refresh Output Frequency | 1 Hz (data updated every second) |
| Supply Voltage | 3.3V DC regulated ±5% |
| Operating Power | 125 mW |
| Warm-up Time | |  | | --- | |  |  |  | | --- | | 15 minutes | |
| Operating Temperature | 0°C to 50°C |
| Operating Humidity | 0%RH to 95%RH (non-condensing) |
| Storage Temperature | -40°C to 80°C |
| Storage Humidity | 0%RH to 95%RH (non-condensing) |

**MICRO ELECTRET (AMB-707-RC)**

The **AMB-707-RC** is a type of **microphone** known as an **electret condenser microphone**. It is widely used in audio capture applications like voice recording, sound detection, and consumer electronics such as phones, laptops, and other communication devices. The electret microphone (such as the AMB-707-RC) operates similarly to a condenser microphone but with a key difference in how the electrical field is generated. At its core, it features a diaphragm—typically a thin, metal-coated plastic membrane—that vibrates when exposed to sound pressure. Behind the diaphragm is a special electret material, which has a quasi-permanent electric charge. Unlike traditional condenser microphones that require an external polarizing voltage, the electret's charge serves this purpose. The diaphragm and a backplate behind it form a capacitor. When sound waves cause the diaphragm to move, the distance between the diaphragm and the backplate changes, altering the capacitance. This change in capacitance, coupled with the electret's permanent charge, generates a small voltage that corresponds to the sound signal. This signal, however, is of high impedance, so the microphone includes a built-in Field Effect Transistor (FET) to convert it into a low-impedance signal, making it suitable for amplification and further processing.

**PARAMETERS**

|  |  |
| --- | --- |
| **PARAMETER** | **DETAILS** |
| Type | |  | | --- | |  |  |  | | --- | | Electret Condenser | |
| Polar Pattern | Omni-directional |
| Sensitivity | |  | | --- | |  |  |  | | --- | | -42 dB ±3 dB | |
| Frequency Range | 20 Hz – 16 kHz |
| Impedance | 2.2 kΩ |
| Operating Voltage | 1.5V to 10V (typically 2V) |
| Current Consumption | 0.5 mA |
| Signal-to-Noise Ratio | >60 dB |
| Diameter | 9.7 mm |
| Height | |  | | --- | |  |  |  | | --- | | 4.5 mm | |
| Operating Temperature | -20°C to +70°C |
| Storage Temperature | -40°C to +85°C |
| Output Type | Analog |

**BandPass Filter and Nyquist Criteria**

A band-pass filter is a critical tool in signal processing that selectively allows signals within a specific frequency range to pass while attenuating those outside the range. This functionality makes band-pass filters invaluable in various applications, from communications and audio processing to biomedical engineering and instrumentation. In general, these filters are used to isolate a desired frequency band, remove noise, and enhance signal clarity.

In the context of communication systems, band-pass filters are essential for modulating and demodulating signals. For instance, in radio frequency (RF) transmission, each channel is assigned a specific frequency band. A band-pass filter isolates this band, ensuring that only the desired signal is processed while rejecting interference from adjacent channels. This capability is particularly crucial in environments with high spectral congestion, where multiple signals coexist. Similarly, in audio processing, band-pass filters can isolate specific instruments or voices from a mix, enabling focused analysis or enhancement.

Another significant application of band-pass filters is in biomedical signal processing. For example, electrocardiogram (ECG) signals often contain noise from muscle movements or electrical interference. By applying a band-pass filter tuned to the frequency range of cardiac signals, typically 0.5 Hz to 50 Hz, it is possible to remove unwanted components while preserving critical diagnostic information. This selective filtering improves the accuracy of automated analysis algorithms and enhances the interpretability of the data for clinicians.

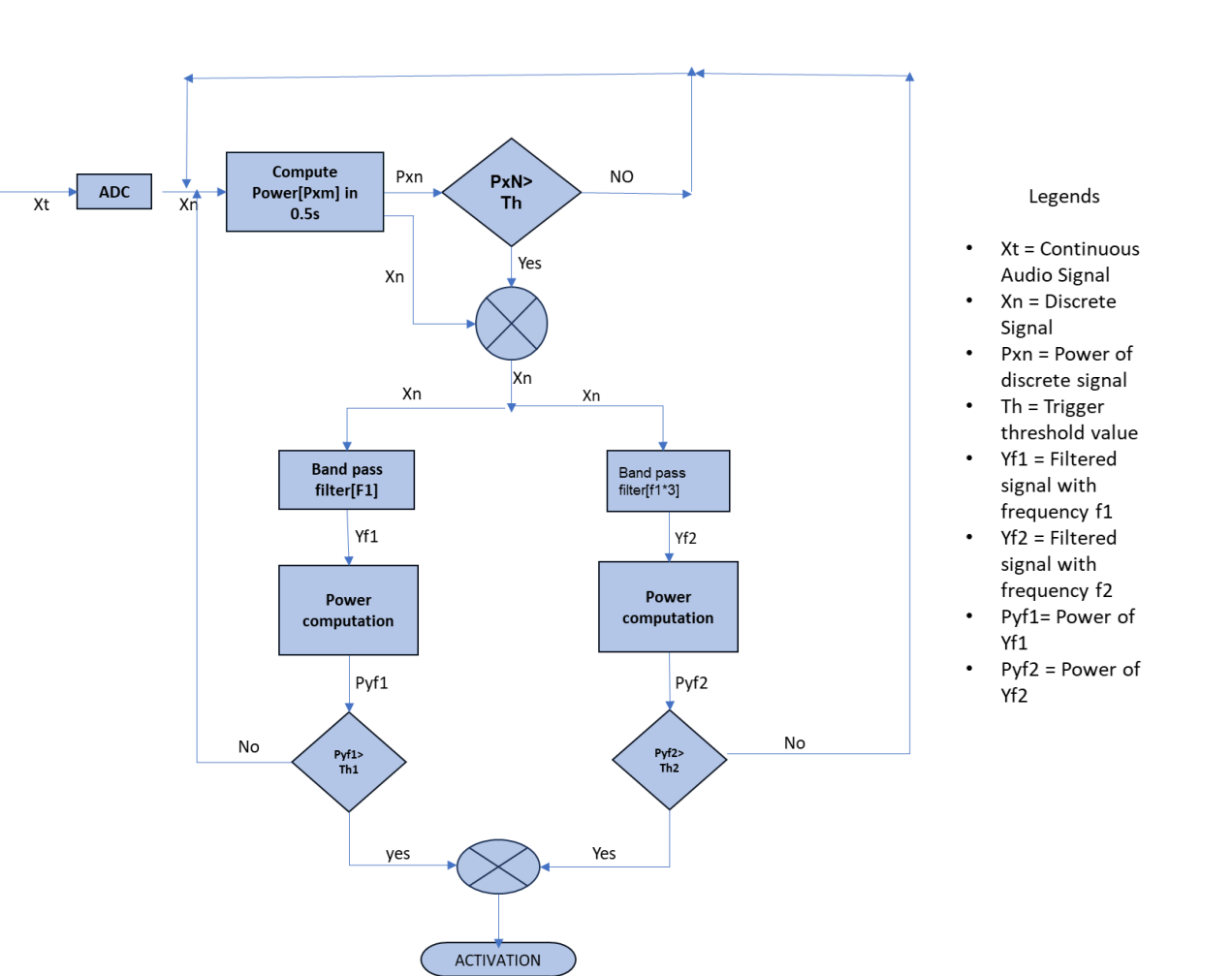
From a theoretical perspective, the design and application of band-pass filters are guided by the Nyquist-Shannon sampling theorem. This fundamental principle states that to accurately reconstruct a signal, it must be sampled at a rate at least twice its highest frequency component. In practical terms, this criterion ensures that signals of interest within the passband are preserved without aliasing. When designing a band-pass filter, it is essential to ensure that the sampling rate of the system satisfies the Nyquist criterion for the upper cutoff frequency of the filter. Failure to do so results in aliasing, where higher frequency components are folded back into the passband, causing distortion and reducing signal integrity.

In digital signal processing (DSP), band-pass filters are implemented using discrete algorithms, such as finite impulse response (FIR) or infinite impulse response (IIR) techniques. These implementations benefit from the Nyquist criterion by ensuring that digital representations of the signal retain their fidelity. For instance, in applications like speech recognition, band-pass filters can isolate the range of human speech, typically 300 Hz to 3400 Hz, while eliminating low-frequency noise and high-frequency hiss. By preserving the essential components of the speech signal, the filter enhances the performance of recognition algorithms.

The versatility of band-pass filters extends to instrumentation and control systems, where they are used to isolate signals from sensors or transducers. For example, in vibration analysis, a band-pass filter can focus on the frequency range associated with specific mechanical components, enabling precise fault diagnosis. Similarly, in radar and sonar systems, band-pass filters help isolate echoes from targets within a specific range, improving detection accuracy.

In summary, band-pass filters play a foundational role in signal processing by enabling the selective isolation and enhancement of signals within a defined frequency range. Their applications span communication, audio processing, biomedical engineering, and beyond, addressing challenges like noise reduction, signal isolation, and feature extraction. Guided by principles like the Nyquist criterion, these filters ensure accurate signal representation and processing, making them indispensable in modern technology. By understanding and leveraging the capabilities of band-pass filters, engineers and scientists can develop systems that are both efficient and robust, meeting the demands of diverse real-world scenarios.

**System Activation Flow Chart**



The flowchart illustrates a signal processing system designed to analyze and filter audio signals for activation based on specific power thresholds. The process begins with a continuous audio signal (“Xt”) being converted into a discrete signal (“Xn”) using an Analog-to-Digital Converter (ADC). This conversion ensures that the analog signal is transformed into a digital form suitable for further processing. Once digitized, the power of the discrete signal (“Pxn”) is computed over a 0.5-second interval. This computation helps determine the signal's energy content, which is critical for deciding whether the signal warrants further analysis.

The computed power (“Pxn”) is then compared to a predefined threshold value (“Th”). If the power is below the threshold, the system deems the signal insignificant and returns to the beginning, continuously monitoring the incoming audio. However, if the power exceeds the threshold, the system progresses to the next stage, indicating that the signal contains potentially useful information requiring further processing.

At this stage, the discrete signal (“Xn”) is split into two parallel paths for frequency-based filtering. The first path processes the signal through a band-pass filter tuned to frequency range “f1”, producing a filtered signal (“Yf1”). Similarly, the second path applies a band-pass filter designed for a different frequency range, specifically “f1\*3”, resulting in another filtered signal (“Yf2”). These band-pass filters are critical for isolating specific frequency components of interest from the input signal, allowing for a focused analysis on targeted frequency bands.

The filtered signals (“Yf1” and “Yf2”) are then subjected to power computation, resulting in “Pyf1” and “Pyf2”, respectively. These power values represent the energy contained within the respective frequency bands. Each power value is subsequently compared to its corresponding threshold (“Th1” for “Pyf1” and “Th2” for “Pyf2”). If the power of either filtered signal exceeds its respective threshold, the system identifies the signal as significant and activates the desired functionality.

The activation stage signifies that the input audio signal has met the criteria for both power and frequency-specific thresholds, making it relevant for further processing or action. The system ensures that only signals with sufficient energy in specific frequency bands trigger the activation, thereby minimizing false positives and improving efficiency.

In summary, the flowchart represents a robust signal processing system that employs power computation, thresholding, and band-pass filtering to isolate and analyze audio signals. This approach is particularly useful in applications requiring selective signal activation, such as voice recognition, biomedical signal monitoring, or noise-reduction systems. By leveraging the power of band-pass filters and precise thresholding mechanisms, the system ensures accurate and reliable signal detection and activation.

**RESULTS**

1. **Soldering of components**

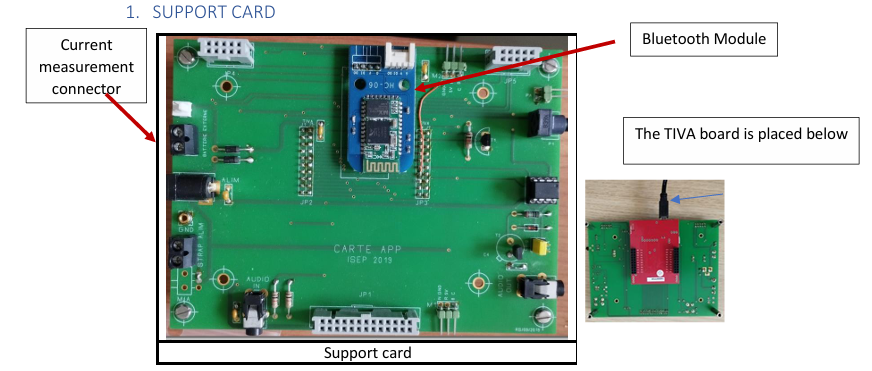


Figure 4: support card

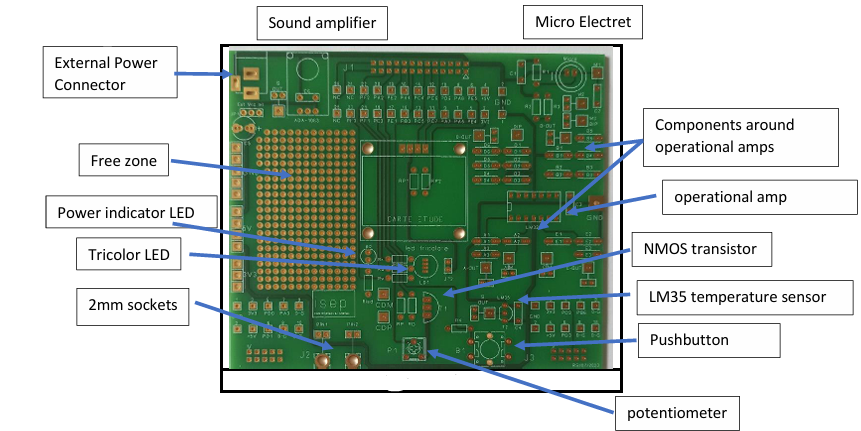
****

Figure 5: Team card

1. **Measuring the current and voltage used by LED**

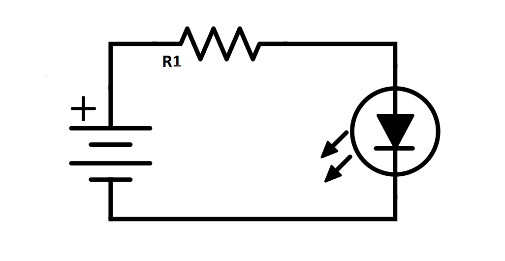


Figure 3. Basic Schematic diagram

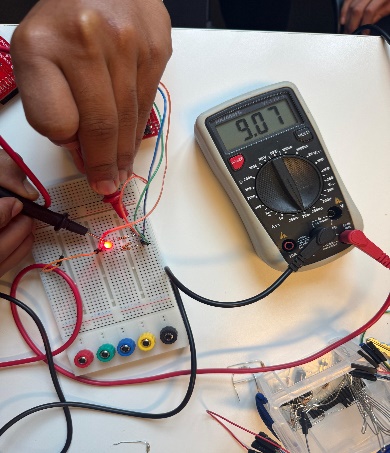
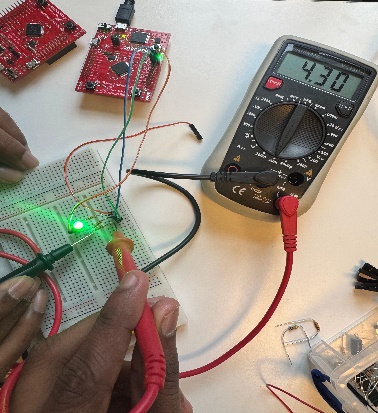
Using ohm’s law,

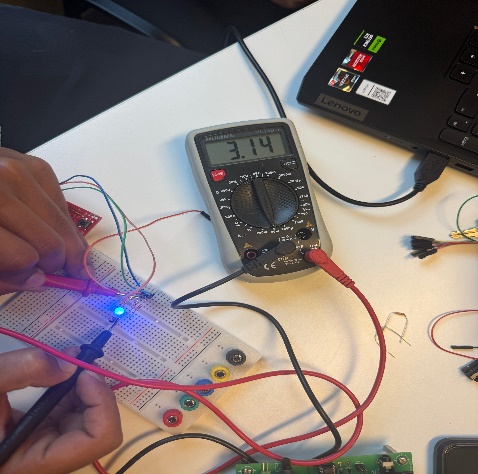
V=IR

R=100 Ω

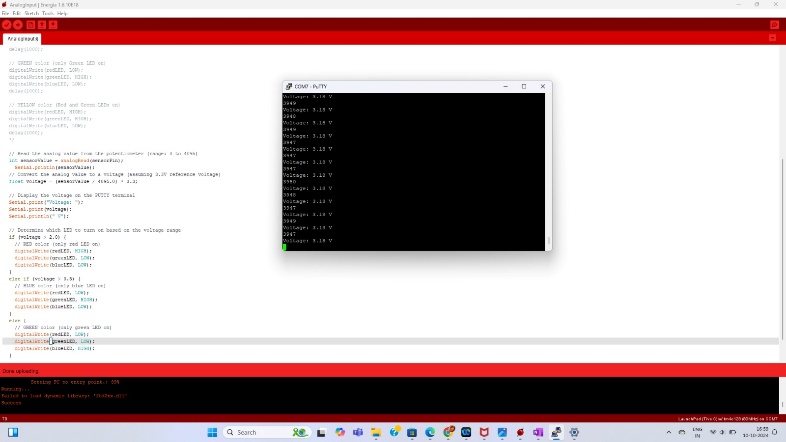
Supply voltage = 3.3 V

|  |  |  |
| --- | --- | --- |
| COLORS | Practical Values | |
| V(Volt) | I(mA) |
| Red | 3.12 | 9.07 |
| Blue | 3.04 | 3.14 |
| Green | 3.2 | 4.28 |
| Yellow | 3.06 | 6.30 |

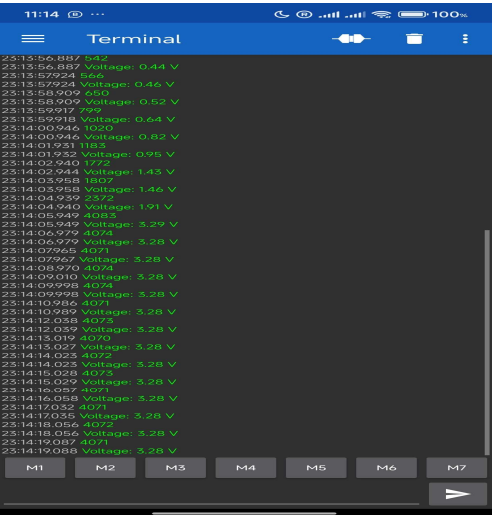
 



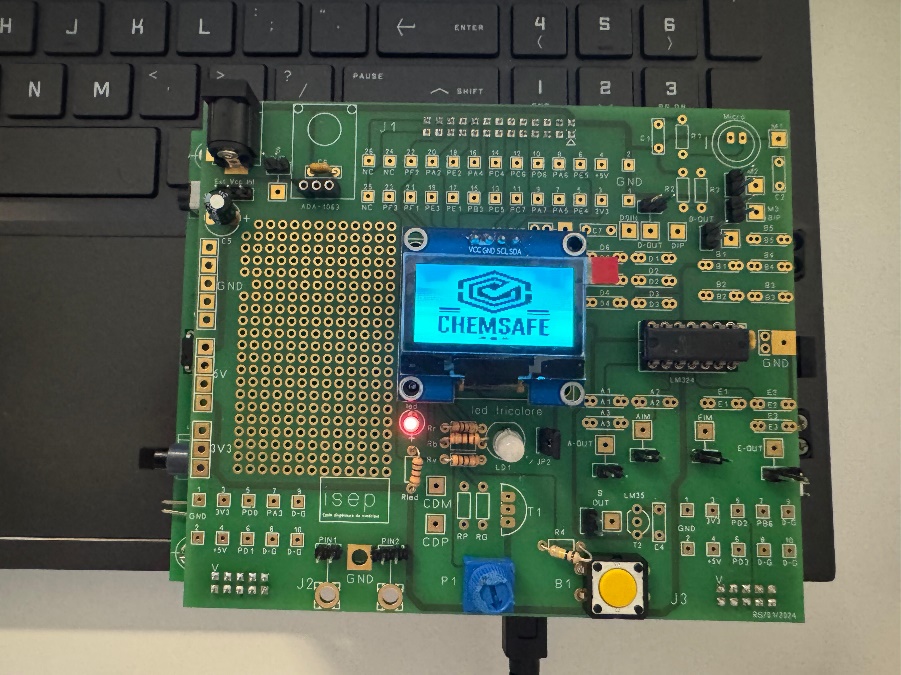
1. **LED Illumination Response to Voltage Variations and displaying in PuTTY console**



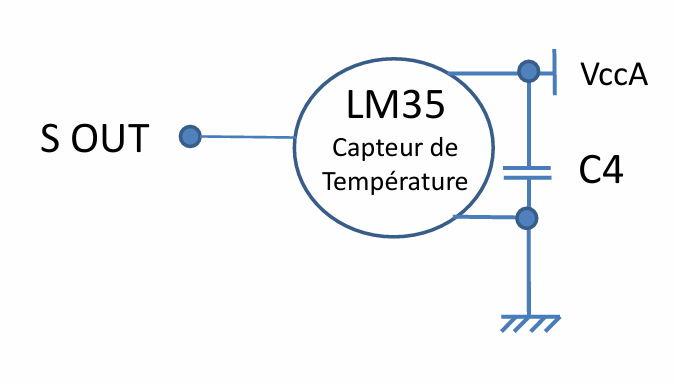
1. **LED Illumination Response to Voltage Variations and use the Bluetooth link to communicate with a smartphone**

****

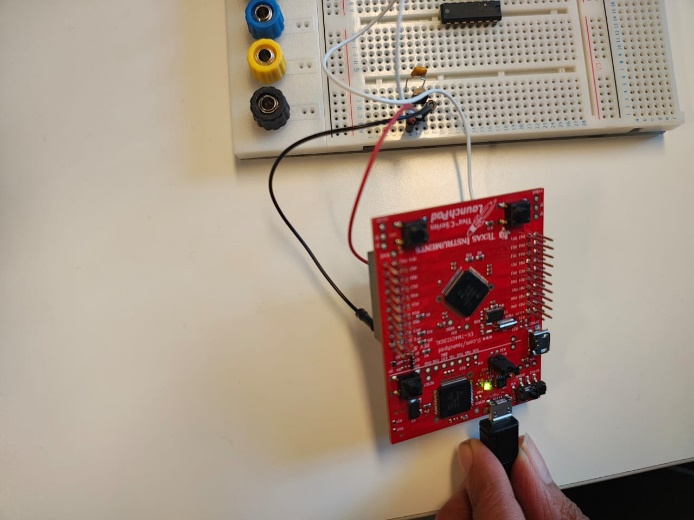
1. **Displaying LOGO in OLED**

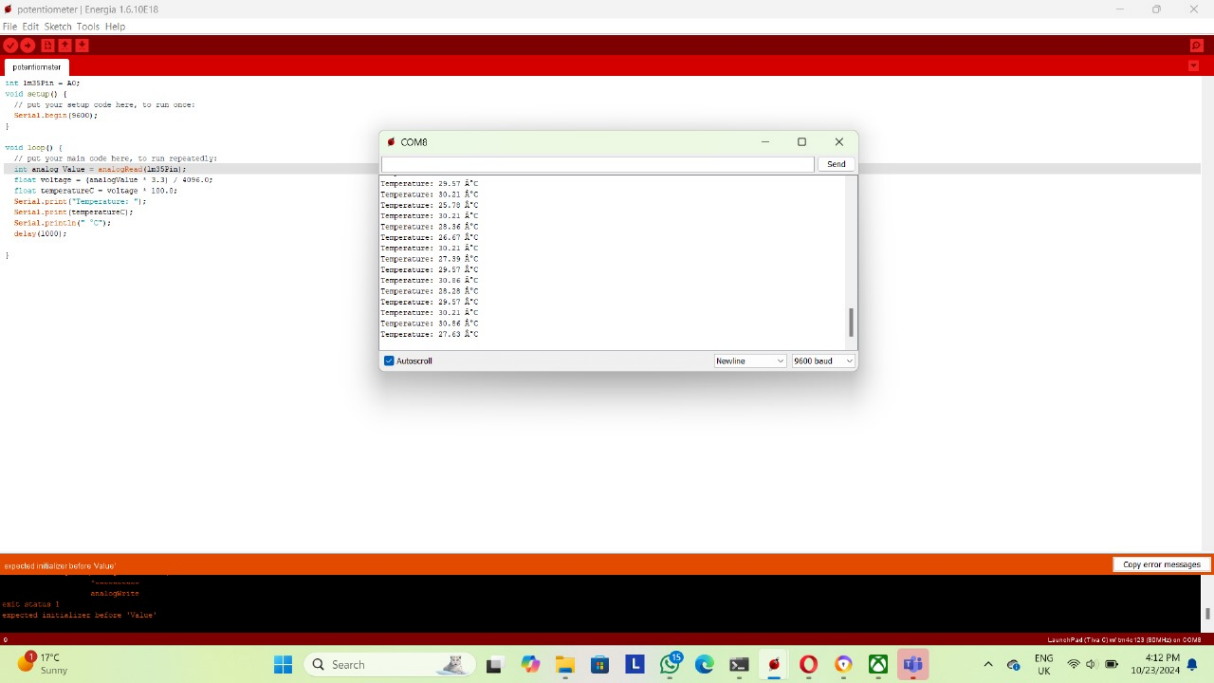


**Temperature Measurement**



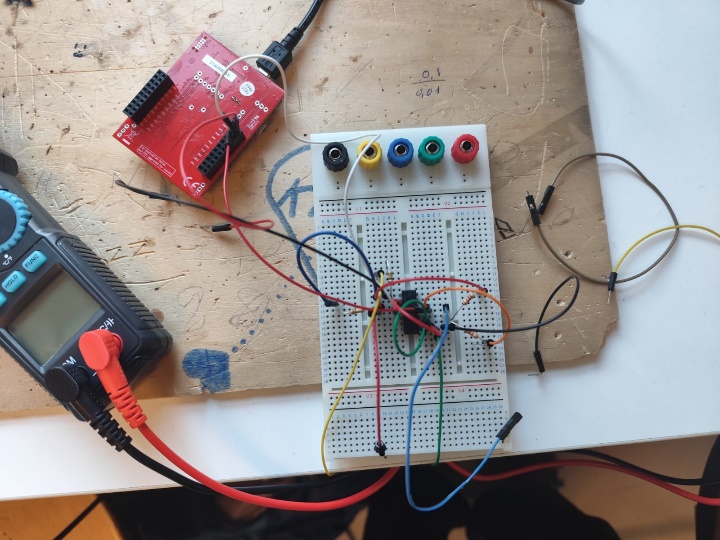
The **LM35** is a temperature sensor with its **VccA** pin is connected to a positive voltage source, while the **Ground** pin is connected to the circuit's ground. The **S OUT** pin provides an analog voltage output that corresponds to the measured temperature, where the output increases by 10mV for every 1°C rise in temperature. A capacitor, labeled **C4**, is placed between **VccA** and ground. This capacitor acts as a **bypass capacitor**, filtering out any noise or voltage fluctuations in the power supply to ensure the sensor operates stably and provides a consistent, accurate output signal. This setup allows for smooth and reliable temperature measurement in the circuit.

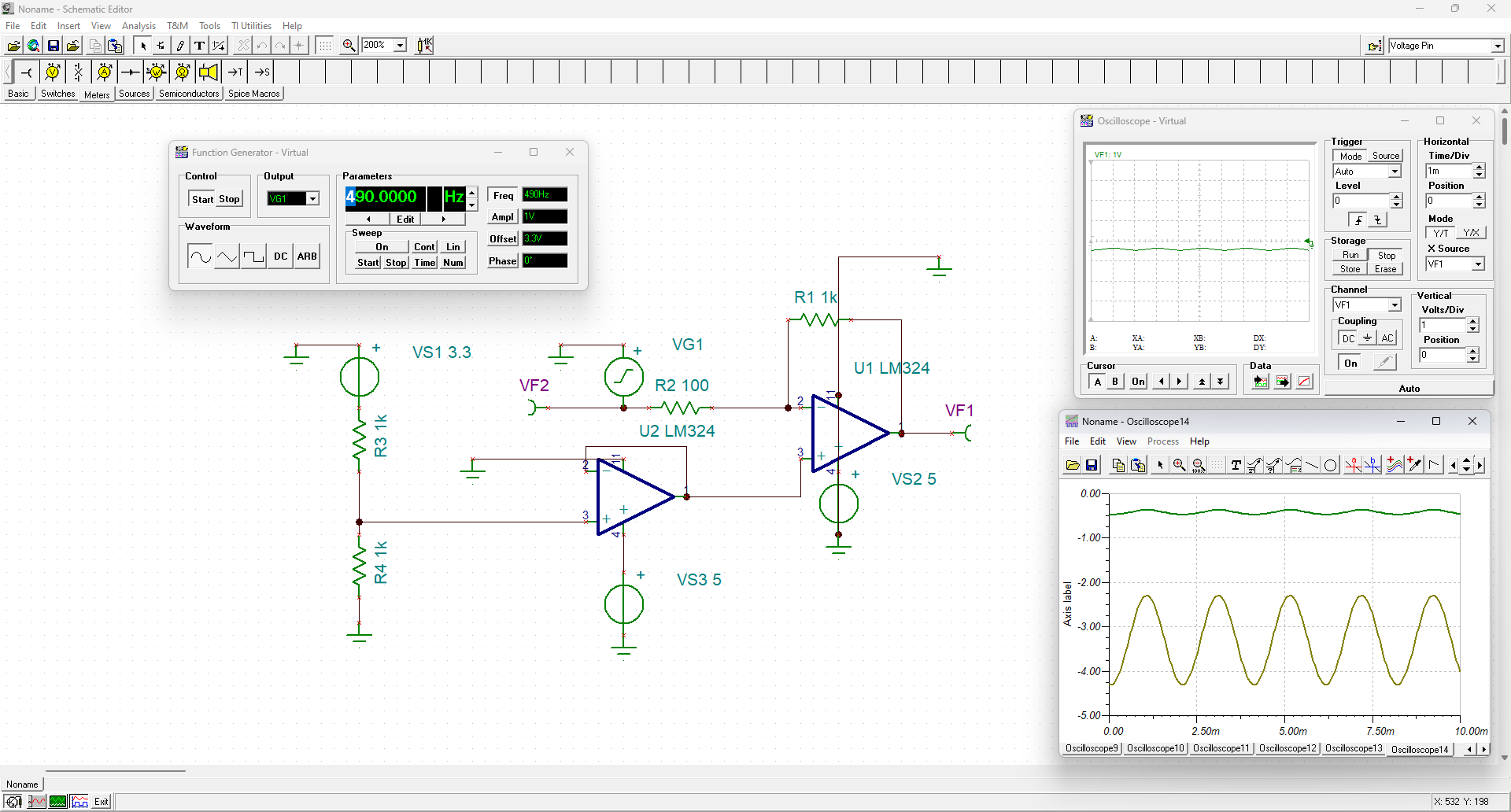




The accuracy of the LM35 temperature sensor is influenced by several factors. Intrinsic accuracy is typically ±0.5°C at room temperature (25°C), but this can vary to around ±1°C to ±2°C over the sensor's full operating range. The operating temperature range of the LM35 is from -55°C to +150°C, though its accuracy is most reliable near room temperature. At extreme temperatures, environmental factors may degrade performance. Additionally, power supply noise or fluctuations can impact accuracy by introducing errors in the analog output. The capacitor (C4) in the circuit helps mitigate this by filtering out noise and ensuring stable power. While the LM35 comes pre-calibrated from the manufacturer, small variations between individual sensors or environmental factors might require additional calibration in high-precision applications. Lastly, self-heating can occur if excessive current is supplied to the sensor, causing a slight increase in temperature that may lead to minor errors in the reading.

1. **Amplification of sound**





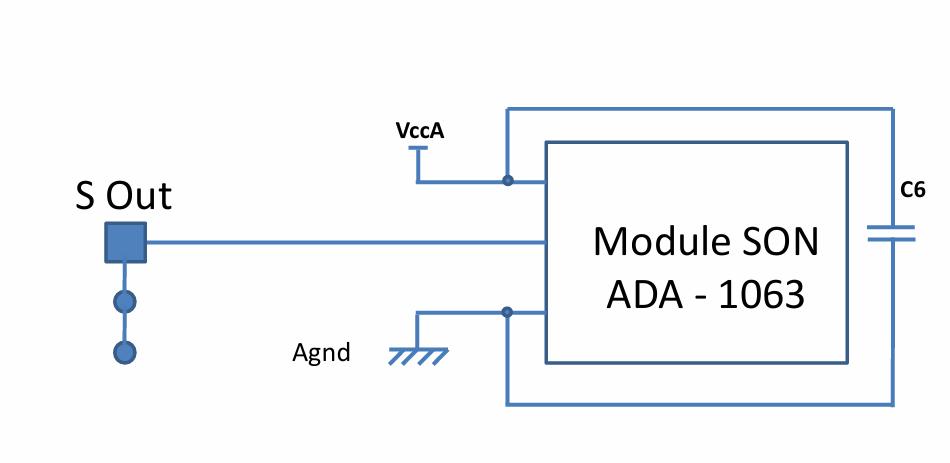
This image displays a circuit simulation environment with an operational amplifier (op-amp) circuit, set up alongside virtual measurement tools like a function generator and oscilloscopes. The circuit is powered by two DC voltage sources labeled \*\*VS1\*\* (3.3V) and \*\*VS2\*\* (5V), providing the necessary operating voltages. Various resistors—specifically \*\*R1\*\*, \*\*R2\*\*, \*\*R3\*\*, and \*\*R4\*\* with values of 1kΩ, 100Ω, 1kΩ, and 1kΩ respectively—are included in the circuit. These resistors are likely part of the biasing and feedback network essential for controlling the gain and stability of the op-amps.

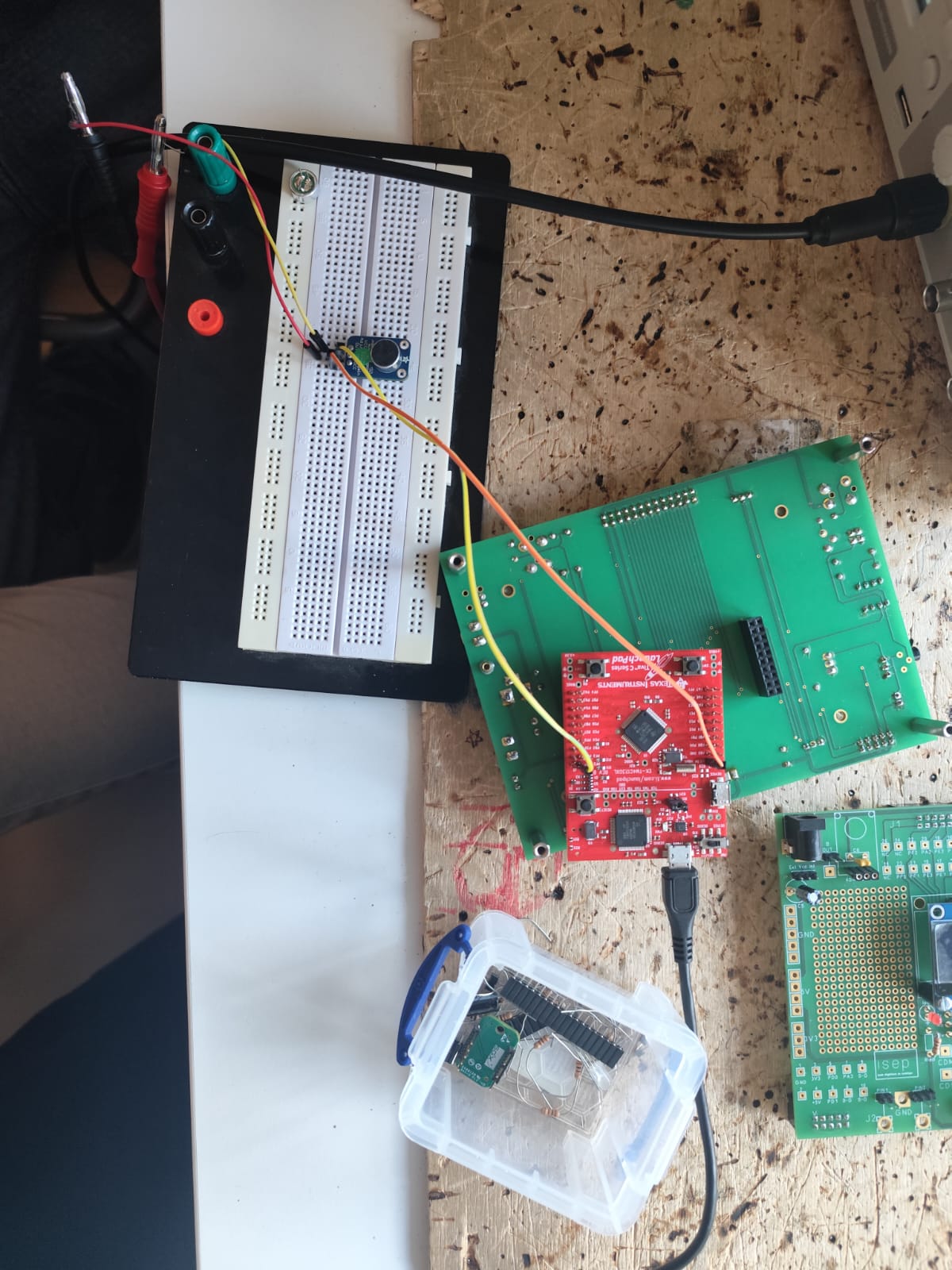
The circuit includes two \*\*LM324\*\* operational amplifiers, labeled \*\*U1\*\* and \*\*U2\*\*, which are commonly used for general-purpose amplification tasks. \*\*U1\*\* appears to be in an inverting amplifier configuration with a feedback resistor (R1 = 1kΩ) connecting its output to its inverting input. This setup indicates that \*\*U1\*\* is amplifying or inverting the signal in some way. \*\*U2\*\*, the second op-amp, is connected to a signal source (\*\*VG1\*\*) through a resistor (R2 = 100Ω). This configuration suggests that \*\*U2\*\* could be used to buffer or further amplify the input signal.The function generator window displays parameters set for a sinusoidal waveform at a \*\*frequency of 490 Hz\*\*, \*\*amplitude of 1V\*\*, and \*\*offset of 3.3V\*\*, with a phase of \*\*0°\*\*. This signal is injected into the circuit through \*\*VG1\*\* to test how the circuit responds to an AC input. This sinusoidal input oscillates between 2.3V and 4.3V, given the 3.3V offset and 1V amplitude.

Two virtual oscilloscopes are set up to monitor the signal at various points in the circuit. The first oscilloscope (top right) shows a low-amplitude, nearly flat waveform, indicating it may be observing a DC component or a steady-state signal with minimal fluctuations. The second oscilloscope (bottom center), however, displays a more pronounced sinusoidal waveform, suggesting that this point in the circuit has an active AC signal with observable amplitude peaks and troughs. This difference in the signals between the two oscilloscope views suggests they are connected to different parts of the circuit, with the first potentially capturing a less dynamic part, and the second capturing an amplified or processed AC response.

In summary, this circuit setup appears to be testing the op-amps' handling of an AC signal. The function generator provides an AC input, while the op-amps and resistors work to amplify or condition the signal. The oscilloscopes provide insights into the signal's transformation through the circuit, allowing for the analysis of amplification levels, phase shifts, and any potential signal distortion. This simulation offers a comprehensive way to evaluate the performance of analog circuits in amplifying and shaping AC signals.

1. **Power Consumed by the ADA-1063 Amplified Microphone Module**

****

****

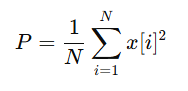
This pin diagram illustrates the connections for a module labeled \*\*SON ADA-1063\*\*, showing how to interface its power, ground, and signal output. The \*\*S Out\*\* pin serves as the signal output, allowing the module to send processed or generated signals to external circuits. The \*\*Agnd\*\* pin is designated as the analog ground, providing a stable ground reference for the module’s analog components, which helps reduce noise and ensure reliable performance. \*\*VccA\*\* is the analog power supply input, supplying the necessary voltage for the module's analog circuitry. Additionally, a capacitor labeled \*\*C6\*\* is connected between VccA and Agnd; this capacitor likely serves as a decoupling component, helping to stabilize the power supply and filter out noise. Together, these connections support the stable operation of the \*\*SON ADA-1063\*\* module, particularly in environments where power stability is crucial for accurate analog signal processing.

**CONNECTION DETAILS**

|  |  |
| --- | --- |
| Record Length | 2500.00 |
| Sample Interval | 0.01 |
| Trigger Point | 0.00 |
| Source | CH1 |
| Vertical Units | V |
| Vertical Scale | 0.50 |
| Vertical Offset | -0.04 |
| Horizontal Units | s |
| Horizontal Scale | 2.50 |
| Pt Fmt | Y |
| Yzero | 0.00 |
| Probe Atten | 1.00 |
| Model Number | TBS1102B-EDU |
| Serial Number | C030427 |
| Firmware Version | FV:v4.06 |

**Compute Signal Power from Audio Samples**

The power of an audio signal can be computed from its samples by taking the mean square value of the sample amplitudes. Assuming the samples are given as a discrete-time sequence, you can calculate the power P as:



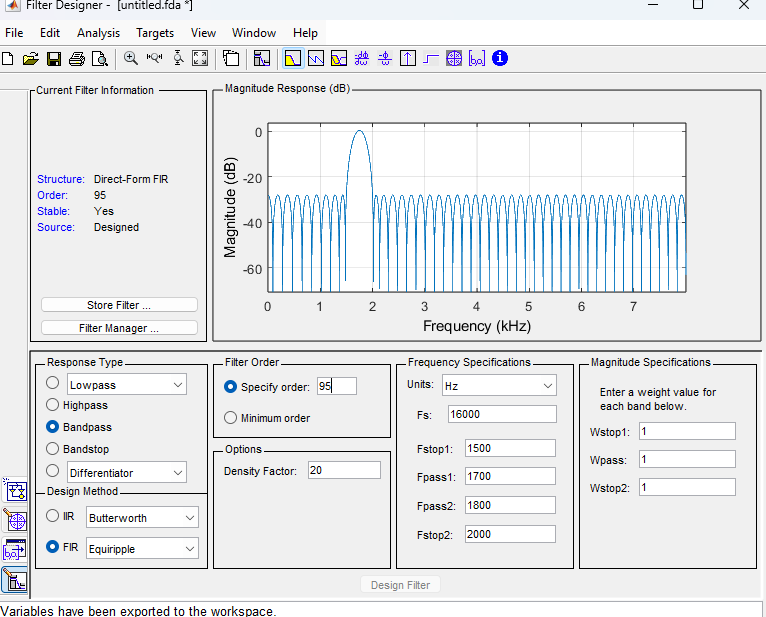
where:

* x[i] is the i-th sample value,
* N is the total number of samples = 2500

Power consumed = 2055.9548/2500

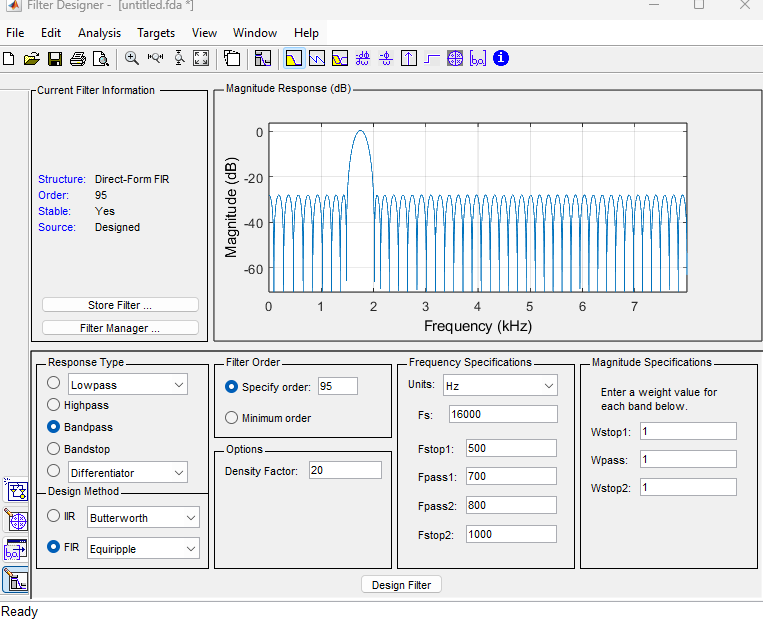
~0.822 w

1. **FIR Band Pass Filter**



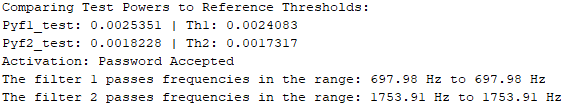
The design of a Finite Impulse Response (FIR) bandpass filter was carried out using MATLAB's Filter Designer tool, employing the Equiripple method. The Equiripple design technique is well-suited for FIR filters, as it minimizes the maximum error between the desired and actual frequency responses by distributing the error uniformly across the passband and stopband. For this filter, the order was specified as 95, ensuring a high degree of control over the filter's frequency response and sharpness of transitions.

The sampling frequency (FsF\_sFs​) was set to 16,000 Hz, with the passband frequencies defined between 1,700 Hz and 1,800 Hz (Fpass1F\_{pass1}Fpass1​ and Fpass2F\_{pass2}Fpass2​, respectively). The stopband frequencies were set at 1,500 Hz (Fstop1F\_{stop1}Fstop1​) and 2,000 Hz (Fstop2F\_{stop2}Fstop2​). Uniform weighting factors of 1 were assigned to both the stopband and passband, indicating equal importance in minimizing the ripple magnitudes in these regions.

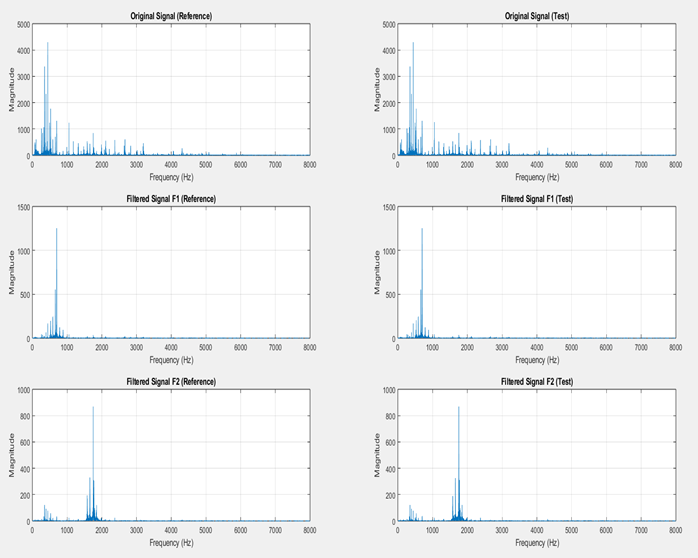


Design of a Finite Impulse Response (FIR) bandpass filter using MATLAB's Filter Designer tool with the Equiripple design method. This method is chosen for its capability to minimize the maximum error between the desired and actual frequency responses by evenly distributing the ripple across the passband and stopband. The filter order is specified as 95 to ensure a precise design with well-defined frequency characteristics. The sampling frequency (FsF\_sFs​) is set to 16,000 Hz, with a passband ranging from 700 Hz to 800 Hz (Fpass1F\_{pass1}Fpass1​ and Fpass2F\_{pass2}Fpass2​, respectively), allowing frequencies within this range to pass. The stopband frequencies are set at 500 Hz (Fstop1F\_{stop1}Fstop1​) and 1,000 Hz (Fstop2F\_{stop2}Fstop2​), ensuring attenuation of frequencies outside the passband. The magnitude specifications include equal weights for the stopband and passband ripples, both set to 1, to provide balanced importance to minimizing deviations in these regions. The magnitude response plot shown in the interface confirms the filter's characteristics, with ripples evident in the passband and stopband, a hallmark of the Equiripple design. The transitions between the passband and stopband are sharp, demonstrating the filter’s effectiveness in meeting the specified design constraints. This FIR bandpass filter is well-suited for applications requiring precise frequency selection, and its parameters and coefficients are exported for further implementation and analysis. This design process highlights the capability of MATLAB's Filter Designer tool in creating high-performance digital filters for signal processing tasks.

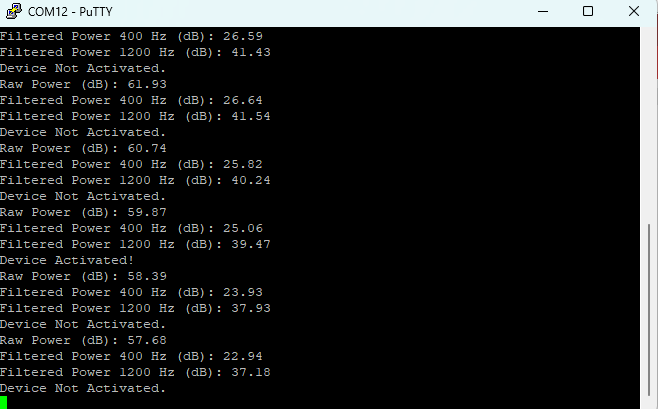
**OUTPUT**



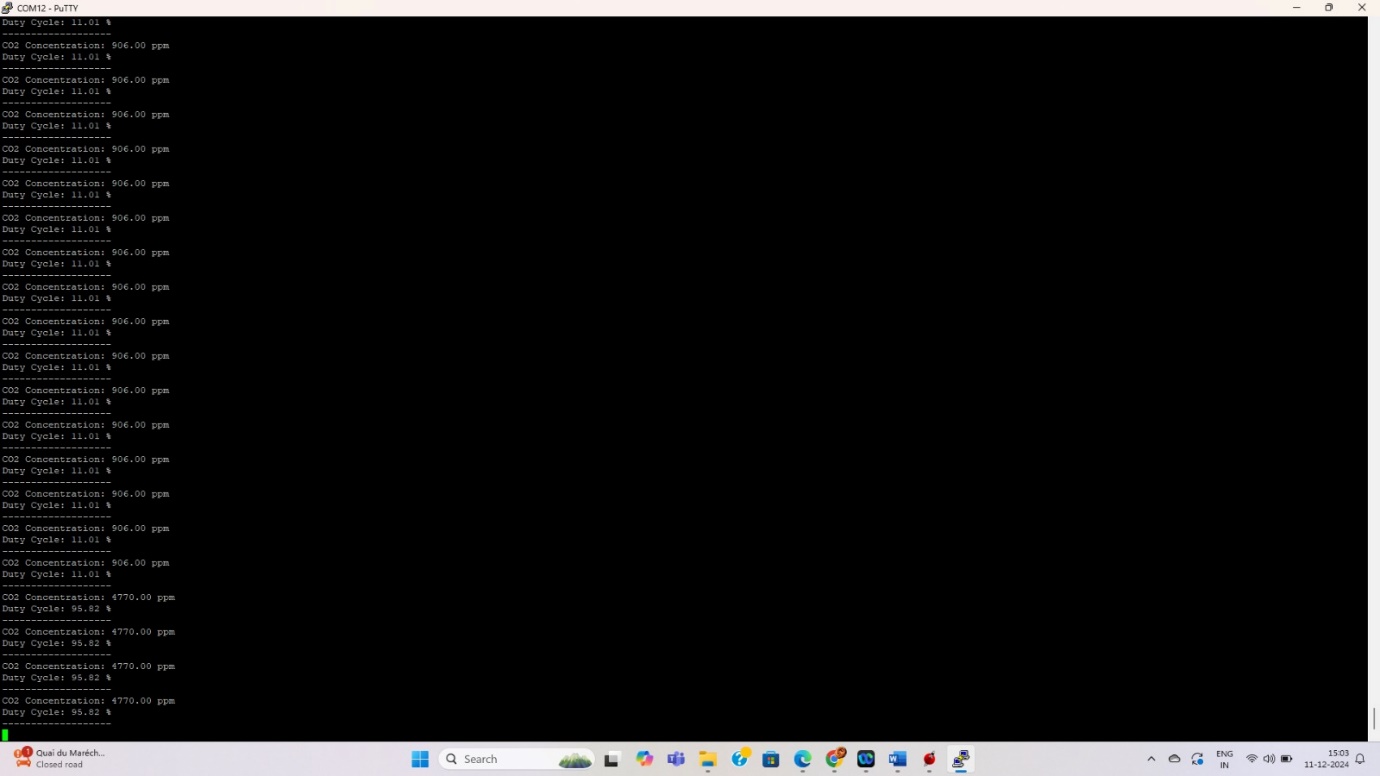
The first filter allows frequencies in the range of 697.98 Hz, while the second filter permits frequencies in the range of 1753.91 Hz. The filters were likely designed using MATLAB's Filter Designer tool with parameters tailored to meet the requirements of the application. The Equiripple method, which minimizes the maximum error between the desired and actual frequency responses, appears to have been used. This ensures precise frequency selection with ripples uniformly distributed across the passband and stopband. The design effectively isolates the desired frequency ranges, meeting the activation criteria for system functionality.



1. **Device Activation using password signal**



1. **CO2 Detection**



**CODES**

1. **Blinking of LED and generate the following colors**

**BLUE – RED – GREEN – YELLOW**

const int RED\_PIN = 30; // Pin for Red LED

const int GREEN\_PIN = 39; // Pin for Green LED

const int BLUE\_PIN = 40; // Pin for Blue LED

void setup() {

pinMode(RED\_PIN, OUTPUT);

pinMode(GREEN\_PIN, OUTPUT);

pinMode(BLUE\_PIN, OUTPUT);

}

void loop() {

// Turn Red LED on

digitalWrite(RED\_PIN, HIGH);

delay(1000);

digitalWrite(RED\_PIN, LOW);

delay(3000);

// Turn Green LED on

digitalWrite(GREEN\_PIN, HIGH);

delay(1000);

digitalWrite(GREEN\_PIN, LOW);

delay(3000);

// Turn Blue LED on

digitalWrite(BLUE\_PIN, HIGH);

delay(1000);

digitalWrite(BLUE\_PIN, LOW);

delay(3000);

// Turn yellow LED on

digitalWrite(GREEN\_PIN, HIGH);

digitalWrite(RED\_PIN, HIGH);

delay(3000);

digitalWrite(GREEN\_PIN, LOW);

digitalWrite(RED\_PIN, LOW);

// Turn off all LEDs

delay(1000);

}

1. **Code to blink LED in accordance with voltage and display on putty console**

* **Red LED** will glow when the voltage is greater than **2.0V**.
* **Green LED** will glow when the voltage is greater than **0.5V** but less than or equal to **2.0V**.
* **Blue LED** will glow when the voltage is less than or equal to **0.5V**.

const int sensorPin = A0; // select the input pin for the potentiometer

int sensorValue = 0; // variable to store the value coming from the sensor

int redLED = 30; // Connect Red pin of LED to pin 30

int greenLED = 39; // Connect Green pin of LED to pin 39

int blueLED = 40; // Connect Blue pin of LED to pin 40

void setup() {

// put your setup code here, to run once:

Serial.begin(9600);

pinMode(redLED, OUTPUT);

pinMode(greenLED, OUTPUT);

pinMode(blueLED, OUTPUT);

pinMode(sensorPin,INPUT);

}

void loop() {

// Read the analog value from the potentiometer (range: 0 to 4095)

int sensorValue = analogRead(sensorPin);

Serial.println(sensorValue);

// Convert the analog value to a voltage (assuming 3.3V reference voltage)

float voltage = (sensorValue / 4095.0) \* 3.3;

// Display the voltage on the PUTTY terminal

Serial.print("Voltage: ");

Serial.print(voltage);

Serial.println(" V");

// Determine which LED to turn on based on the voltage range

if (voltage > 2.0) {

// RED color (only red LED on)

digitalWrite(redLED, HIGH);

digitalWrite(greenLED, LOW);

digitalWrite(blueLED, LOW);

}

else if (voltage > 0.5) {

// BLUE color (only blue LED on)

digitalWrite(redLED, LOW);

digitalWrite(greenLED, HIGH);

digitalWrite(blueLED, LOW);

}

else {

// GREEN color (only green LED on)

digitalWrite(redLED, LOW);

digitalWrite(greenLED, LOW);

digitalWrite(blueLED, HIGH);

}

delay(1000); // Wait for 1 second before the next reading

}

1. **Code for Bluetooth Communications**

const int sensorPin = A0; // select the input pin for the potentiometer

int sensorValue = 0; // variable to store the value coming from the sensor

int redLED = 30; // Connect Red pin of LED to pin 30

int greenLED = 39; // Connect Green pin of LED to pin 39

int blueLED = 40; // Connect Blue pin of LED to pin 40

void setup() {

// put your setup code here, to run once:

**Serial1**.begin(9600);

pinMode(redLED, OUTPUT);

pinMode(greenLED, OUTPUT);

pinMode(blueLED, OUTPUT);

pinMode(sensorPin,INPUT);

}

void loop() {

// Read the analog value from the potentiometer (range: 0 to 4095)

int sensorValue = analogRead(sensorPin);

**Serial1**.println(sensorValue);

// Convert the analog value to a voltage (assuming 3.3V reference voltage)

float voltage = (sensorValue / 4095.0) \* 3.3;

// Display the voltage on the PUTTY terminal

**Serial1**.print("Voltage: ");

**Serial1**.print(voltage);

**Serial1**.println(" V");

// Determine which LED to turn on based on the voltage range

if (voltage > 2.0) {

// RED color (only red LED on)

digitalWrite(redLED, HIGH);

digitalWrite(greenLED, LOW);

digitalWrite(blueLED, LOW);

}

else if (voltage > 0.5) {

// BLUE color (only blue LED on)

digitalWrite(redLED, LOW);

digitalWrite(greenLED, HIGH);

digitalWrite(blueLED, LOW);

}

else {

// GREEN color (only green LED on)

digitalWrite(redLED, LOW);

digitalWrite(greenLED, LOW);

digitalWrite(blueLED, HIGH);

}

delay(1000); // Wait for 1 second before the next reading

1. **Code for displaying LOGO on OLED**

#include "iseplogo128.h"

void setup() {

// put your setup code here, to run once:

InitI2C();

InitScreen();

Display(motif); // affichage de l'image décrite dans le tabelau de donnée motif.h

**iseplogo128.h**

unsigned char motif[128\*8] ={

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x7f, 0x7f, 0x7f, 0x3f, 0x3f, 0x3f, 0x1f, 0x1f, 0x1f, 0x8f, 0x8f, 0x8f, 0x87,

0xc7, 0x8f, 0x8f, 0x8f, 0x1f, 0x1f, 0x1f, 0x3f, 0x3f, 0x3f, 0x7f, 0x7f, 0x7f, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0x1f, 0x0f, 0x0f, 0x0f, 0x07, 0xc7, 0xc7, 0xc3, 0xe3, 0x63, 0x63, 0x71, 0x31, 0x31,

0x38, 0x18, 0x18, 0x9c, 0x8c, 0x8c, 0xce, 0xc6, 0xc6, 0x67, 0x63, 0x63, 0x33, 0x31, 0x31, 0x11,

0x19, 0x31, 0x31, 0x33, 0x63, 0x23, 0x07, 0x06, 0x86, 0x8e, 0xcc, 0xcc, 0xec, 0xf8, 0xf8, 0xf8,

0xf1, 0xf1, 0xf1, 0x61, 0x63, 0xe3, 0xc3, 0xc7, 0xc7, 0x07, 0x0f, 0x0f, 0x0f, 0x1f, 0xbf, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0x30, 0x30, 0x30, 0x30, 0x30, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0xfe,

0xff, 0xff, 0x03, 0x01, 0x01, 0x01, 0x10, 0x38, 0x38, 0x7c, 0x7c, 0xfc, 0xf8, 0xf8, 0xf0, 0xf0,

0xf0, 0xf0, 0xf8, 0xfc, 0x7c, 0x7e, 0x3e, 0x3f, 0x1f, 0x1f, 0x0f, 0x0f, 0x07, 0x03, 0xe3, 0xf1,

0xf1, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0x30, 0x30, 0x30, 0x30, 0x30, 0x37, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xc0, 0xc0, 0x80, 0x80, 0x80, 0x8f, 0x1f, 0x1f, 0x1f, 0x38, 0x30, 0x30, 0x70, 0x73,

0x63, 0x63, 0xe6, 0xc6, 0xc4, 0xcc, 0x8c, 0x8c, 0x88, 0x18, 0x18, 0x10, 0x30, 0x31, 0x21, 0x61,

0x61, 0x21, 0x30, 0x30, 0x30, 0x18, 0x18, 0x98, 0x8c, 0x8c, 0xcc, 0xc4, 0xc6, 0xe6, 0x63, 0x63,

0x63, 0x70, 0x30, 0x30, 0x38, 0x18, 0x1f, 0x1f, 0x0f, 0x80, 0x80, 0x80, 0xc0, 0xc0, 0xe8, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7,

0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe, 0xfe, 0xfe, 0xfc, 0xfc,

0xfc, 0xf8, 0xf8, 0xf8, 0xf0, 0xf1, 0xf1, 0xe1, 0xe3, 0xe3, 0xc3, 0xc7, 0xc6, 0x86, 0x8e, 0x8e,

0x8c, 0x8e, 0x86, 0xc6, 0xc7, 0xc3, 0xe3, 0xe3, 0xe1, 0xf1, 0xf1, 0xf0, 0xf8, 0xf8, 0xf8, 0xfc,

0xfc, 0xfc, 0xfe, 0xfe, 0xfe, 0xfe, 0xff, 0xff, 0xff, 0xff, 0xff, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7,

0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xa7, 0xbf, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x03, 0x03, 0x03, 0xf3, 0xfb, 0xf3, 0xc3, 0xc3, 0xc3, 0xff, 0xff, 0x03, 0x03,

0x03, 0x03, 0xff, 0xff, 0x7f, 0x03, 0x03, 0x03, 0xfb, 0xff, 0xab, 0x03, 0x03, 0x03, 0xf3, 0xf3,

0xf3, 0xf3, 0xfb, 0xff, 0x03, 0x03, 0x03, 0x03, 0x03, 0x1f, 0xff, 0xff, 0x3f, 0x07, 0x03, 0x03,

0x03, 0x03, 0xff, 0xff, 0x87, 0x03, 0x03, 0x03, 0x73, 0xf3, 0xc3, 0xc3, 0xc3, 0xc7, 0xff, 0xff,

0xff, 0x0f, 0x03, 0x03, 0x83, 0x03, 0x03, 0x1f, 0xff, 0xff, 0xff, 0x03, 0x03, 0x03, 0x03, 0xf3,

0xf3, 0xf3, 0xf3, 0xfb, 0xff, 0x03, 0x03, 0x03, 0x03, 0xf3, 0xf3, 0xf3, 0xf3, 0xfb, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0x80, 0x80, 0x00, 0x3f, 0x3f, 0x3f, 0x07, 0x87, 0x87, 0xff, 0xff, 0x00, 0x00,

0x00, 0x00, 0xfe, 0xfe, 0xfc, 0x00, 0x00, 0x00, 0x7f, 0xff, 0x03, 0x00, 0x00, 0x00, 0x3c, 0x3c,

0x3c, 0x3e, 0x3f, 0xff, 0x00, 0x00, 0x00, 0x1e, 0xf8, 0xc0, 0x80, 0x83, 0x80, 0xf0, 0xfc, 0x00,

0x00, 0x00, 0x7f, 0xff, 0x87, 0x87, 0x06, 0x06, 0x3e, 0x3c, 0x00, 0x80, 0x81, 0xc3, 0x7f, 0x0f,

0x00, 0x00, 0xe0, 0xe6, 0xe7, 0xe0, 0x00, 0x00, 0x01, 0x3f, 0xff, 0x00, 0x00, 0x00, 0x00, 0xfc,

0xfc, 0xfc, 0xfc, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x3e, 0x3c, 0x3c, 0x3e, 0xbf, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xf7, 0xf7, 0xf3, 0xf3, 0xf3, 0xf3, 0xff, 0xff, 0xff, 0xf7, 0xe3, 0xe3,

0xe3, 0xe3, 0xe3, 0xff, 0xff, 0xff, 0xff, 0xf7, 0xf3, 0xf7, 0xf3, 0xf7, 0xf7, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff

};

ACCESS AUTHORIZED

1. **Script for password verification using FIR bandPass Filters**

% Load the reference (password) audio signal

[audio\_signal\_ref, fs\_ref] = audioread('Chord2.wav'); % Replace with the actual password signal file

% Load the test signal

[audio\_signal\_test, fs\_test] = audioread('Chord2.wav'); % Replace with the test signal file

% Check if sampling rates match

if fs\_ref ~= fs\_test

error('Sampling rates do not match!');

end

% Normalize both the reference and test signals

audio\_signal\_ref = audio\_signal\_ref / max(abs(audio\_signal\_ref)); % Normalize reference

audio\_signal\_test = audio\_signal\_test / max(abs(audio\_signal\_test)); % Normalize test signal

% --- Manually Define the Filter Coefficients for Two Filters ---

% Filter 1 (b1, a1)

b1 = [0.0116671452699185,-0.0116726018834218,-0.00297837974462980,0.00292681028148567,0.00631196657771458,0.00769967330378614,0.00762119267707962,0.00649072705999318,0.00458495869159855,0.00208769555478214,-0.000866263684301049,-0.00412751906357623,-0.00748950673773928,-0.0106905369797799,-0.0134178928642583,-0.0153353686472723,-0.0161349061592552,-0.0155871527557103,-0.0135700799975076,-0.0101010565049414,-0.00536138269958242,0.000318210780389534,0.00647826184160335,0.0125580881767080,0.0179479859223657,0.0220663485471834,0.0244122358139987,0.0246174748052294,0.0225175647626171,0.0181592062762769,0.0117979321877662,0.00394913813709299,-0.00476863629406524,-0.0135113943034798,-0.0215291656883664,-0.0279744964141303,-0.0322080082901047,-0.0337534484031634,-0.0323399325756422,-0.0280149975846610,-0.0211084187416335,-0.0121784270040432,-0.00199504878918228,0.00852060328732265,0.0184087174135660,0.0267667171543739,0.0328149692798007,0.0359853584849739,0.0359853584849739,0.0328149692798007,0.0267667171543739,0.0184087174135660,0.00852060328732265,-0.00199504878918228,-0.0121784270040432,-0.0211084187416335,-0.0280149975846610,-0.0323399325756422,-0.0337534484031634,-0.0322080082901047,-0.0279744964141303,-0.0215291656883664,-0.0135113943034798,-0.00476863629406524,0.00394913813709299,0.0117979321877662,0.0181592062762769,0.0225175647626171,0.0246174748052294 0.0244122358139987,0.0220663485471834,0.0179479859223657,0.0125580881767080,0.00647826184160335,0.000318210780389534,-0.00536138269958242,-0.0101010565049414,-0.0135700799975076,-0.0155871527557103,-0.0161349061592552,-0.0153353686472723,-0.0134178928642583,-0.0106905369797799,-0.00748950673773928,-0.00412751906357623,-0.000866263684301049,0.00208769555478214,0.00458495869159855,0.00649072705999318,0.00762119267707962,0.00769967330378614,0.00631196657771458,0.00292681028148567,-0.00297837974462980,-0.0116726018834218,0.0116671452699185];

% Example numerator coefficients (replace with actual values)

a1 = [1]; % Example denominator coefficients (replace with actual values)

% Filter 2 (b2, a2)

b2 = [0.00608889843030509,-0.013381474731563,0.00390572964064437,0.00465411898267589,-0.00140657840700819,-0.00778658974755154,-0.0110264682515848,-0.00991067122475507,-0.00490894036016601,0.00214363792250272,0.00847134141951499,0.0112651103099765,0.0088893596456841,0.00186314185851542,-0.00700597527727073,-0.0136341631485089,-0.0144816219207682,-0.00838240111426295,0.00253783665247588,0.0135006125911595,0.0191570301016647,0.0161826142407083,0.00512943919819457,-0.00945012159761774,-0.0208497158737086,-0.0232983413403139,-0.014867344569561,0.00126796588964945,0.0179824731957233,0.0273665899421053,0.0244730742819113,0.00989415799337704,-0.0101565739518016,-0.0265092398464763,-0.0312865877888191,-0.0217028768046953,-0.00161530722132674,0.0199933760810102,0.0330806186843893,0.0312791771995344,0.0149902706013513,-0.00861261627438423,-0.0287467338817879,-0.0360302668452552,-0.0269117722653193,-0.00534916743638492,0.0188177975957225,0.0345211235191627,0.0345211235191627,0.0188177975957225,-0.00534916743638492,-0.0269117722653193,-0.0360302668452552,-0.0287467338817879,-0.00861261627438423,0.0149902706013513,0.0312791771995344,0.0330806186843893,0.0199933760810102,-0.00161530722132674,-0.0217028768046953,-0.0312865877888191,-0.0265092398464763,-0.0101565739518016,0.00989415799337704,0.0244730742819113,0.0273665899421053,0.0179824731957233,0.00126796588964945,-0.014867344569561,-0.0232983413403139,-0.0208497158737086,-0.00945012159761774,0.00512943919819457,0.0161826142407083,0.0191570301016647,0.0135006125911595,0.00253783665247588,-0.00838240111426295,-0.0144816219207682,-0.0136341631485089,-0.00700597527727073,0.00186314185851542,0.0088893596456841,0.0112651103099765,0.00847134141951499,0.00214363792250272,-0.00490894036016601,-0.00991067122475507,-0.0110264682515848,-0.00778658974755154,-0.00140657840700819,0.00465411898267589,0.00390572964064437,-0.013381474731563,0.00608889843030509];

% Example numerator coefficients (replace with actual values)

a2 = [1]; % Example denominator coefficients (replace with actual values)

% --- Apply First Bandpass Filter (F1) to Reference and Test Signals ---

Yf1\_ref = filter(b1, a1, audio\_signal\_ref); % Apply the first bandpass filter to the reference signal

Yf1\_test = filter(b1, a1, audio\_signal\_test); % Apply the first bandpass filter to the test signal

% --- Apply Second Bandpass Filter (F2) to Reference and Test Signals ---

Yf2\_ref = filter(b2, a2, audio\_signal\_ref); % Apply the second bandpass filter to the reference signal

Yf2\_test = filter(b2, a2, audio\_signal\_test); % Apply the second bandpass filter to the test signal

% --- Plot the Frequency Spectrum for Filtered Signals ---

% Function to plot the frequency spectrum of the signal

% --- Plot Original and Filtered Signals ---

figure; % Create a new figure for all plots

% Plot Original Reference Signal

subplot(3, 2, 1);

plot\_frequency\_spectrum(audio\_signal\_ref, fs\_ref, 'Original Signal (Reference)');

title('Original Signal (Reference)');

% Plot Original Test Signal

subplot(3, 2, 2);

plot\_frequency\_spectrum(audio\_signal\_test, fs\_ref, 'Original Signal (Test)');

title('Original Signal (Test)');

% Plot Filtered Signal F1 (Reference)

subplot(3, 2, 3);

plot\_frequency\_spectrum(Yf1\_ref, fs\_ref, 'Filtered Signal F1 (Reference)');

title('Filtered Signal F1 (Reference)');

% Plot Filtered Signal F1 (Test)

subplot(3, 2, 4);

plot\_frequency\_spectrum(Yf1\_test, fs\_ref, 'Filtered Signal F1 (Test)');

title('Filtered Signal F1 (Test)');

% Plot Filtered Signal F2 (Reference)

subplot(3, 2, 5);

plot\_frequency\_spectrum(Yf2\_ref, fs\_ref, 'Filtered Signal F2 (Reference)');

title('Filtered Signal F2 (Reference)');

% Plot Filtered Signal F2 (Test)

subplot(3, 2, 6);

plot\_frequency\_spectrum(Yf2\_test, fs\_ref, 'Filtered Signal F2 (Test)');

title('Filtered Signal F2 (Test)');

% --- Calculate the Power of the Filtered Signals ---

Pyf1\_ref = mean(Yf1\_ref.^2); % Power of the filtered signal F1 for reference

Pyf2\_ref = mean(Yf2\_ref.^2); % Power of the filtered signal F2 for reference

% --- Calculate Thresholds from the reference signal (password) ---

Th1 = 0.95 \* Pyf1\_ref; % Threshold for filtered signal F1

Th2 = 0.95 \* Pyf2\_ref; % Threshold for filtered signal F2

% --- Calculate Powers of the Filtered Test Signals ---

Pyf1\_test = mean(Yf1\_test.^2); % Power of the filtered signal F1 for test

Pyf2\_test = mean(Yf2\_test.^2); % Power of the filtered signal F2 for test

% --- Compare Test and Reference Signal Powers ---

disp('Comparing Test Powers to Reference Thresholds:');

disp(['Pyf1\_test: ', num2str(Pyf1\_test), ' | Th1: ', num2str(Th1)]);

disp(['Pyf2\_test: ', num2str(Pyf2\_test), ' | Th2: ', num2str(Th2)]);

if Pyf1\_test > Th1 && Pyf2\_test > Th2

disp('Activation: Password Accepted');

else

disp('Activation: Password Rejected');

end

% Step 1: Compute the Fourier Transform of the filtered signal

N = length(Yf1\_ref); % Length of the filtered signal

Yf1\_fft = fft(Yf1\_ref, N); % Compute FFT of the filtered signal

f = (0:N-1) \* (fs\_ref / N); % Frequency vector

% Step 2: Analyze the magnitude spectrum

Yf1\_magnitude = abs(Yf1\_fft(1:N/2+1)); % Magnitude of positive frequencies

f\_positive = f(1:N/2+1); % Positive frequency vector

% Step 3: Normalize the magnitude and convert to dB

Yf1\_dB = 20 \* log10(Yf1\_magnitude / max(Yf1\_magnitude)); % Convert to dB

% Step 4: Identify significant frequencies (above -3 dB)

threshold = -3; % -3 dB threshold

significant\_frequencies = f\_positive(Yf1\_dB >= threshold);

% Step 5: Determine the frequency range

if ~isempty(significant\_frequencies)

freq\_range = [min(significant\_frequencies), max(significant\_frequencies)];

fprintf('The filter 1 passes frequencies in the range: %.2f Hz to %.2f Hz\n', freq\_range(1), freq\_range(2));

else

fprintf('No frequencies passed above the -3 dB threshold.\n');

end

% Step 1: Compute the Fourier Transform of the filtered signal

N = length(Yf2\_ref); % Length of the filtered signal

Yf2\_fft = fft(Yf2\_ref, N); % Compute FFT of the filtered signal

f = (0:N-1) \* (fs\_ref / N); % Frequency vector

% Step 2: Analyze the magnitude spectrum

Yf2\_magnitude = abs(Yf2\_fft(1:N/2+1)); % Magnitude of positive frequencies

f\_positive = f(1:N/2+1); % Positive frequency vector

% Step 3: Normalize the magnitude and convert to dB

Yf2\_dB = 20 \* log10(Yf2\_magnitude / max(Yf2\_magnitude)); % Convert to dB

% Step 4: Identify significant frequencies (above -3 dB)

threshold = -3; % -3 dB threshold

significant\_frequencies = f\_positive(Yf2\_dB >= threshold);

% Step 5: Determine the frequency range

if ~isempty(significant\_frequencies)

freq\_range = [min(significant\_frequencies), max(significant\_frequencies)];

fprintf('The filter 2 passes frequencies in the range: %.2f Hz to %.2f Hz\n', freq\_range(1), freq\_range(2));

else

fprintf('No frequencies passed above the -3 dB threshold.\n');

end

This MATLAB code implements a password-based audio signal activation system using two manually defined FIR bandpass filters. The reference signal (Chord2.wav) is treated as the "password," and the test signal (also Chord2.wav in this case) is compared against it. First, the signals are normalized to ensure consistent amplitude scaling. Two bandpass filters with predefined coefficients are applied to both the reference and test signals to isolate specific frequency ranges. The frequency spectrum of the original and filtered signals is plotted to visualize the frequency content, demonstrating the filters' ability to isolate desired frequency bands. The code calculates the power of the filtered reference signals and defines activation thresholds based on 95% of these values. It then computes the power of the filtered test signals and compares them to the thresholds. If the test signals' powers exceed the thresholds for both filters, the system activates, indicating that the password is accepted. Otherwise, the password is rejected.

Additionally, the code analyzes the frequency ranges passed by each filter. Using the Fast Fourier Transform (FFT), it calculates the magnitude spectrum of the filtered signals and identifies the significant frequencies above the -3 dB threshold, determining the passband range for each filter. This provides insight into the filters' frequency characteristics and ensures that they meet the design specifications. The results are displayed as frequency ranges for the two filters, validating their performance. Overall, this code demonstrates a robust method for implementing frequency-based password verification, suitable for audio signal processing applications requiring precise bandpass filtering and threshold-based comparison.

1. **Device Activation using sound signal**

#define SAMPLE\_FREQ 10000 // Sampling frequency (10 kHz)

#define SAMPLE\_COUNT 1000 // Number of samples

#define FILTER\_ORDER 51 // Order of FIR filter

#define ADC\_PIN A1 // Analog pin connected to microphone OUT

int soundData[SAMPLE\_COUNT]; // Array to store sound samples

float filteredData\_400[SAMPLE\_COUNT]; // Array to store 400 Hz filtered samples

float filteredData\_1200[SAMPLE\_COUNT]; // Array to store 1200 Hz filtered samples

float fir\_coeff\_400[FILTER\_ORDER]; // FIR filter coefficients for 400 Hz

float fir\_coeff\_1200[FILTER\_ORDER]; // FIR filter coefficients for 1200 Hz

unsigned long sampleInterval = 1000000 / SAMPLE\_FREQ; // Interval in microseconds

unsigned long lastSampleTime = 0; // Time for the last sample

int sampleIndex = 0; // Index for sampling

bool samplingComplete = false; // Flag for completed sampling

void setup() {

Serial.begin(9600); // Initialize serial communication

analogReadResolution(12); // Set ADC resolution to 12 bits (Tiva C)

// Calculate FIR filter coefficients for 400 Hz band-pass (e.g., 380–420 Hz)

calculate\_fir\_coefficients(fir\_coeff\_400, 380, 420, SAMPLE\_FREQ);

// Calculate FIR filter coefficients for 1200 Hz band-pass (e.g., 1180–1220 Hz)

calculate\_fir\_coefficients(fir\_coeff\_1200, 1180, 1220, SAMPLE\_FREQ);

}

void loop() {

// Sampling loop to get 1000 samples

if (!samplingComplete) {

unsigned long currentTime = micros();

if (currentTime - lastSampleTime >= sampleInterval) {

lastSampleTime = currentTime;

// Read the microphone signal

soundData[sampleIndex++] = analogRead(ADC\_PIN);

// Check if we have collected all samples

if (sampleIndex >= SAMPLE\_COUNT) {

sampleIndex = 0;

samplingComplete = true;

}

}

}

// Process samples and compute power if ready

if (samplingComplete) {

samplingComplete = false;

// Apply FIR filters to the sampled data

for (int i = 0; i < SAMPLE\_COUNT; i++) {

filteredData\_400[i] = apply\_fir\_filter(soundData, fir\_coeff\_400, SAMPLE\_COUNT, i);

filteredData\_1200[i] = apply\_fir\_filter(soundData, fir\_coeff\_1200, SAMPLE\_COUNT, i);

}

// Calculate raw power of the pre-filtered signal

float rawPower = 0;

for (int i = 0; i < SAMPLE\_COUNT; i++) {

rawPower += (soundData[i] - 2048) \* (soundData[i] - 2048); // Centering and squaring

}

rawPower /= SAMPLE\_COUNT;

// Calculate power of the 400 Hz filtered signal

float filteredPower\_400 = 0;

for (int i = 0; i < SAMPLE\_COUNT; i++) {

filteredPower\_400 += filteredData\_400[i] \* filteredData\_400[i]; // Squaring filtered signal

}

filteredPower\_400 /= SAMPLE\_COUNT;

// Calculate power of the 1200 Hz filtered signal

float filteredPower\_1200 = 0;

for (int i = 0; i < SAMPLE\_COUNT; i++) {

filteredPower\_1200 += filteredData\_1200[i] \* filteredData\_1200[i]; // Squaring filtered signal

}

filteredPower\_1200 /= SAMPLE\_COUNT;

// Convert power to decibels (dB)

float rawPowerDB = 10 \* log10(rawPower);

float filteredPowerDB\_400 = 10 \* log10(filteredPower\_400);

float filteredPowerDB\_1200 = 10 \* log10(filteredPower\_1200);

// Check activation conditions

bool isActivated = (filteredPowerDB\_400 >= 25 && filteredPowerDB\_400 <= 29) &&

(filteredPowerDB\_1200 >= 37 && filteredPowerDB\_1200 <= 40);

// Print the results

Serial.print("Raw Power (dB): ");

Serial.println(rawPowerDB);

Serial.print("Filtered Power 400 Hz (dB): ");

Serial.println(filteredPowerDB\_400);

Serial.print("Filtered Power 1200 Hz (dB): ");

Serial.println(filteredPowerDB\_1200);

if (isActivated) {

Serial.println("Device Activated!");

} else {

Serial.println("Device Not Activated.");

}

delay(1000); // Add a delay for readability

}

}

// Function to calculate FIR filter coefficients

void calculate\_fir\_coefficients(float \*coeffs, float low\_cutoff, float high\_cutoff, float fs) {

int mid = FILTER\_ORDER / 2;

float fc1 = low\_cutoff / fs; // Normalize cutoff frequencies

float fc2 = high\_cutoff / fs;

for (int n = 0; n < FILTER\_ORDER; n++) {

if (n == mid) {

coeffs[n] = 2 \* (fc2 - fc1);

} else {

float t = n - mid;

coeffs[n] = (sin(2 \* PI \* fc2 \* t) - sin(2 \* PI \* fc1 \* t)) / (PI \* t);

}

coeffs[n] \*= 0.5 \* (1 - cos(2 \* PI \* n / (FILTER\_ORDER - 1))); // Apply Hamming window

}

}

// Function to apply FIR filter

float apply\_fir\_filter(int \*signal, float \*coeffs, int bufferSize, int currentIndex) {

float result = 0;

for (int n = 0; n < FILTER\_ORDER; n++) {

int index = (currentIndex - n + bufferSize) % bufferSize; // Handle circular buffer

result += signal[index] \* coeffs[n];

}

return result;

}

This program implements a real-time audio signal processing system that samples sound from a microphone, filters the signal using FIR (Finite Impulse Response) filters, and analyzes the power levels within specific frequency bands to determine activation conditions. The system operates at a sampling frequency of 10 kHz and collects 1000 samples in each cycle. The sampled data is processed through two band-pass FIR filters, one targeting the 400 Hz range (380–420 Hz) and the other targeting the 1200 Hz range (1180–1220 Hz). The FIR filters are implemented using pre-calculated coefficients based on the sinc function, further refined with a Hamming window to reduce spectral leakage.

Once the samples are collected, the program applies the FIR filters to isolate the desired frequency components and computes the power of the raw and filtered signals. The raw power and filtered power for each frequency band are calculated as the average squared magnitude of the respective signals and then converted to decibels (dB). The system checks whether the filtered power levels fall within specified thresholds for activation (25–29 dB for 400 Hz and 37–40 dB for 1200 Hz). If the conditions are met, the device is marked as activated, and the results are displayed on the serial monitor.

This implementation showcases real-time signal acquisition and frequency analysis, allowing targeted frequency band evaluation and decision-making based on audio signal properties. Such a system has applications in signal detection, audio filtering, and sound pattern recognition, demonstrating a practical approach to handling real-world audio data

1. **Detection of CO2**

const int pwmPin = 7; // Input pin for PWM signal (connected to sensor's PWM\_CO2 output)

volatile unsigned long pulseHigh = 0; // Duration of HIGH pulse

volatile unsigned long pulseLow = 0; // Duration of LOW pulse

const float expectedFrequency = 30.0; // Expected frequency of PWM signal in Hz

const float expectedPeriod = 1000000.0 / expectedFrequency; // Period in microseconds (33.33 ms)

void setup() {

pinMode(pwmPin, INPUT); // Set PWM pin as input

Serial.begin(9600); // Start serial communication at 9600 baud

attachInterrupt(pwmPin, measurePulse, CHANGE); // Interrupt on pin state change

}

void loop() {

// Calculate total time

unsigned long totalTime = pulseHigh + pulseLow; // Total PWM period in microseconds

if (totalTime > 0) {

// Calculate duty cycle (percentage of HIGH time in the PWM signal)

float dutyCycle = (float)pulseHigh / totalTime \* 100.0; // Duty cycle in percentage

// Display raw duty cycle value

Serial.print("Duty Cycle: ");

Serial.print(dutyCycle);

Serial.println(" %");

// Validate PWM frequency (period should be around 33.33 ms for 30 Hz signal)

if (abs(totalTime - expectedPeriod) < (expectedPeriod \* 0.1)) { // 10% tolerance

Serial.print("PWM Frequency: ");

Serial.print(1000000.0 / totalTime); // Calculate and print actual frequency

Serial.println(" Hz");

// Map duty cycle to CO2 concentration (ppm) if in expected range

if (dutyCycle >= 55.0 && dutyCycle <= 95.0) {

float co2Concentration = map(dutyCycle, 55, 95, 400, 2000); // Map duty cycle to CO2 ppm range

// Display CO2 concentration

Serial.print("CO2 Concentration: ");

Serial.print(co2Concentration);

Serial.println(" ppm");

} else {

Serial.println("Duty cycle out of expected range.");

}

} else {

Serial.println("Invalid PWM frequency detected.");

}

Serial.println("-------------------");

} else {

Serial.println("Waiting for valid PWM signal...");

}

delay(1000); // Update every second

}

// Interrupt service routine to measure pulse duration

void measurePulse() {

static unsigned long lastTime = 0;

unsigned long currentTime = micros(); // Current time in microseconds

if (digitalRead(pwmPin) == HIGH) {

pulseLow = currentTime - lastTime; // Calculate LOW pulse duration

} else {

pulseHigh = currentTime - lastTime; // Calculate HIGH pulse duration

}

lastTime = currentTime; // Update last time

}

This program is designed to measure and interpret a PWM (Pulse Width Modulation) signal, typically generated by a CO2 sensor, to calculate the duty cycle and map it to CO2 concentration levels. The PWM signal's HIGH and LOW pulse durations are measured using an interrupt service routine (ISR) triggered on every signal state change (rising or falling edge). The ISR calculates the duration of the HIGH and LOW pulses, which are then used to determine the signal's duty cycle, total period, and frequency. These measurements are analyzed in the main loop, where the duty cycle is expressed as a percentage, and the frequency is validated against the expected value of 30 Hz with a 10% tolerance.

If the signal frequency is valid and the duty cycle falls within the expected range (55%–95%), the duty cycle is mapped to a CO2 concentration range (400–2000 ppm) using a linear mapping function. The program outputs the duty cycle, signal frequency, and CO2 concentration to the serial monitor for real-time monitoring. If the PWM frequency deviates significantly or the duty cycle is out of range, appropriate error messages are displayed. This implementation demonstrates a practical method for decoding PWM signals and translating them into meaningful data, such as CO2 concentration, making it suitable for sensor integration in environmental monitoring systems.