Sensor Interfacing for Lightning station

Submitted in partial fulfillment of the requirements for the

Summer Internship Program

at

Physical Research Laboratory(PRL) Ahmedabad

by

Siddhant Abhyankar

20BEE0198

Under the guidance of

Ms. Sonam Jitarwal

Interplanetary Dust Science laboratory

PRL Ahmedabad

भौतिक अनुसंधान प्रयोगशाला
Physical Research Laboratory

July,2024

DECLARATION

I here by declare that the report entitled "**Sensor Interfacing for Lightning station**" submitted by me, for the fulfillment of the *Summer Internship Programme at PRL* is a record of bonafide work carried out by me under the supervision of **Ms. Sonam Jitarwal**, Interplanetary Dust Science laboratory, PRL Ahmedabad.

Place: Ahmedabad

Date:05.07.2024 **Signature of the Candidate**

CERTIFICATE

This is to certify that the report entitled "**Sensor Interfacing for Lightning station**" submitted by **Siddhant Abhyankar(CP01584)**, Interplanetary Dust Science laboratory, for the award of the *Summer Internship certificate* , is a record of bonafide work carried out by him under my supervision during the period, 13.05.2024 to 5.07.2024, as per the PRL code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The thesis fulfills the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place: Ahmedabad

Date: 05/07/2024 **Signature of the Guide**

Acknowledgements

With immense pleasure and deep sense of gratitude, I wish to express my sincere thanks to my supervisor Ms. Sonam Jitarwal, without her motivation and continuous encouragement, this project work would not have been successfully completed.

I would like to acknowledge the support rendered by my colleagues in several ways throughout my project work. I wish to extend my profound sense of gratitude to my parents for all the sacrifices they made during my research and also providing me with moral support and encouragement whenever required.

Place: Ahmedabad

Date: 05/07/2024 **Student Name:** Siddhant Abhyankar

Executive Summary

Lightning is a giant spark of electricity that conducts between clouds, the air, or the ground. It's like a massive static shock in the atmosphere. Lightning is most commonly associated with thunderstorms, but it can also happen in volcanic eruptions, dust storms, and even snowstorms. There are different types of lightning too, depending on where the electrical discharge travels: between clouds (intra-cloud), between clouds and the air, or between clouds and the ground (cloud-to-ground). Lightning packs a punch in more ways than one. While it can be dangerous, it also plays a crucial role in our planet's health. Lightning breaks apart nitrogen molecules in the atmosphere. When this nitrogen combines with oxygen and hydrogen from rain, it creates nitrates - a natural fertilizer that helps plants grow. Without this process, plant life would struggle. Lightning's intense energy zaps oxygen molecules in the upper atmosphere, splitting them into single oxygen atoms. These free oxygen atoms can then bond with regular oxygen molecules (O_2) to form ozone (O_3). The ozone layer protects us from the harmful ultraviolet radiation from the sun. Self-cleaning Atmosphere: Lightning acts as a giant air purifier. The high temperatures from a strike can break down greenhouse gasses like methane, helping to regulate Earth's climate. Generally, Lightning is a microsecond's event, where electric currents conducts from one location to another. Therefore, it becomes crucial to monitor lightning using a proper station that can measure the values of humidity, temperature, pressure and potential difference between the antennas. Therefore, this report centers around the Development of Lightning station which will measure variations in humidity, temperature, pressure, and potential difference at the antenna at minimum sampling frequency of 2 MSPS (million samples per second).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Objective

Lightning packs a punch in more ways than one. While it can be dangerous, it also plays a crucial role in our planet's health: Air Fertilizer: Lightning breaks apart nitrogen molecules in the atmosphere. When this nitrogen combines with oxygen and hydrogen from rain, it creates nitrates - a natural fertilizer that helps plants grow. Without this process, plant life would struggle. Ozone Maker: Lightning's intense energy zaps oxygen molecules in the upper atmosphere, splitting them into single oxygen atoms. These free oxygen atoms can then bond with regular oxygen molecules (O_2) to form ozone (O_3). Self-cleaning Atmosphere: Lightning acts as a giant air purifier. The high temperatures from a strike can break down greenhouse gasses like methane, helping to regulate Earth's climate. Even though it can cause wildfires, lightning is a vital part of our planet's ecosystem. It helps sustain plant life, protects us from harmful radiation, and even cleans the air! The problem statement tackled here is "Can we develop a lightning system that can measure humidity, temperature, pressure and potential difference between the antennas at the time of lightning at a high speed of 2 Mega samples per second?"

1.2 Motivation

The main motivation to undertake this project was to record the lightning data at such a high sampling rate along with some other sensor data.

1.3 Background

High-resolution data on humidity, pressure, and temperature during lightning strikes will be crucial for understanding its environmental effects. While electromagnetic pulse (EMP) impacts on electronics are known, detailed data on these atmospheric parameters will pave the way for future planetary lightning studies.

CHAPTER 2

PROJECT DESCRIPTION AND GOALS

2.1 Review of Literature

Some of the literature reviews was done from scholarly articles like (1) Fanrun Meng et al, used MLX90614 and DS18B20 temperature sensor interfaced with stm32 and Arduino UNO microcontroller. The project was made to fastly detect the temperature of human beings during the coronavirus pandemic. (2) WU Xiao-b et al, developed their own novel method for measuring the temperature of the environment. The circuitry consisted of resistors, op-amps etc. The developed IC had a measurement range of -50 to 125 °C and a resolution of 10 mV/°C. (3) Nisha Kashyap et al, designed a low cost multi channel data acquisition system which consisted of temperature sensor like LM-35. (4) Kolapkar M.M et al, used SY-HS-220 humidity sensor and PT100 temperature sensor interfaced with 89E516RD. The authors also designed the signal conditioning circuit for both PT100 and SY-HS-220. (5) Jahedul Islam et al, used a DHT22 humidity sensor interfaced with Arduino UNO and displayed the output values on a LCD screen. (6) Mitar Simic, used SHT11 humidity sensor interfaced with AVR ATmega32 microcontroller. Later the collected data was stored on a SD card.

But most of the literature survey was made with the help of online technical blogs and YouTube videos like (7) deepblueembedded.com, used LM35 temperature sensor with STM32 MCU board to measure temperature in a closed environment. LM35 was used in a full range configuration which requires an external resistor. The output is given to the ADC of STM32 to convert the voltage readings into a digital binary form. Later with the help of some linear equation, the voltage readings can be used to find the corresponding temperature values. The temperature value was later displayed on a small LCD Board. (8) controllerstech.com, used DS18B20 temperature sensor and interfaced it with STM32 to get the temperature readings. The sensor has an operating range of -80 to 150 °C. (9) how2electronics.com et al, used DHT11 sensor for temperature readings, interfaced with STM32. Though DHT11 is known for humidity sensing but still gives very precise temperature values. But the rate of output is not very fast. (10) labprojectsbd.com, used DHT11 for temperature reading. It has a good temperature accuracy of ±0.3 °C and is I2C compatible. (11) instructables.com,

used Infrared sensors like MLX90614 for measuring temperature. These sensors are non contact temperature sensors which can give the temperature readings at a particular point. (12) eletronicswings.com, used a K type thermocouple based temperature sensor and interfaced it with ESP32, but the problem with thermocouple sensors is that they have very low resolution because of their wide range. (13) controllerstech.com, used BMP180 with stm32 to measure temperature, pressure and altitude.

Moreover, a detailed review was also made of all the different types of sensors available in the market with the help of mouser.com to get hands on the best available sensors as per project requirements. Table 2.1 gives a detailed review of temperature sensors. Table 2.2 gives a detailed review of pressure sensors. Table 2.3 gives a detailed review of humidity sensors and finally Table 2.4 gives a detailed review of all sensor equipment.

2.2 Conclusions from the literature review

The main conclusion that can be drawn from this literature review is that there will always be trade off between accuracy and resolution. Sensors with higher accuracy will have lower resolution and sensors with higher resolution will have lower accuracy. Sometimes even the operating range also influences accuracy and resolution.

Moreover, a point to note that out of all the pressure sensors choosen most of them are capacitive in nature because

- Capacitive sensors have heightened pressure sensitivity and measure both high and low pressures accurately.
- They are not affected by changes in temperature, and the temperature coefficient of sensitivity of a capacitive sensor is 10 times better than a piezoresistive pressure sensor.
- Most of them have low power consumption.
- Response time is also low.
- They exhibit less error, high accuracy and high reliability in measurement.
- Resistant to EMI (Electromagnetic Interference) and RFI (Radio Frequency Interference).

Refer (14) for more detailed information about different types of pressure sensor.

With that keeping in mind 3 different sensors were selected as per project requirements.

1. For temperature readings sensors like LM-35, DS18B20, SHT3X, ADT7420, TMP175, MLX90614, DHT22, PT100/PT1000 were shortlisted. Out of all only PT100 series of temperature sensors were selected because of its following salient features:

    - RTD type temperature sensor
    - Temperature range : -200 to 600 °C
    - Accuracy: ±0.1 °C
    - ADC compatible
    - Accuracy can be increased depending on the signal conditioning circuit being used to take the readings. Eg 2 wire, 3 wire, 4 wire configurations. Out of which 4 wire configuration is the most accurate one. and 2 wire configuration is the least accurate one.
    - Rate of sampling of data depends on the speed of the ADC.
    - Resolution also depends on the ADC.
    - Output is fairly linear
    - Cost: low

2. For pressure readings sensors like 2SMPP-02 , BMP280, BMP585, LPS22DF, MPX4115, MPL115A2S, MPX5050, MPXV6115V were shortlisted. Out of all, only the BMP series pressure sensors were selected because of the deep rich and experienced legacy of Bosch LTD in the pressure sensors market. Out of which BMP280 was finalized because of the following salient features:

    - Barometric type pressure sensor
    - Range: 300 to 1100 hPa.
    - Absolute Accuracy: ±100 Pa
    - I2C compatible
    - Resolution: 0.16 Pa/ bit
    - Operating temperature: -40 to 85 °C.
    - Can also measure temperature.

3. For humidity readings sensors like SY-HS-230, SCC30-DB, SHT-30, HDC302, HIH9000, BME-280, HTD2800, T9501 sensors were used. Out of all Sensirion series of humidity sensors were finalized and that too SHT-3X because of the following reasons:

    - Capacitive type sensor.

- Range: 0 to 100
- Accuracy: ±2
- I2C compatible
- Resolution: ± 1.5
- Can also measure temperature
- Operating temperature: -40 to 125 °C

2.2.1 Software

STM32CUBE IDE was used to burn or flash the embedded code on STM32F407VET6. MATLAB Embedded library can also be used for flashing code on the STM board. Arduino IDE was also used to flash code on Arduino UNO. Some driver files also needs to installed like USB to UART converter driver files and STlink-009.

2.2.2 Hardware

For the project STM32F407VET6 microcontroller board was used along with Adduino UNO. Sensors like SHT-30, BMP280, MAX31865 were also used. Later, a general purpose board was used to solder all the components at one place. Along with all these things resistors, jumper wires were also used. A long USB and LAN cables were also used to connect our device with the local PC server for the purpose of data storage.

2.3 Goals

- **Gathering the Lightning data at 2 MSPS**: With the help of NI DAQ we can gather lightning data at 2 MSPS, but not with STM32 MCU board because gathering data at such high MSPS and then gathering sensor data at such a high rate and then mapping them all together is next to impossible.
- **Gathering sensor data**: With the help of STM32 board we can easily get the sensor data with high precision and accuracy. Interfacing all the sensors on NI DAQ is not possible because NI DAQ doesn't have any Serial communication.
- **Sending data to the PC**: After collecting all the data, it becomes more crucial to send data at a very high speed to our local server. We can achieve this objective by using LAN and USB communication.

Name	Type	Range	Accuracy	Interfacing	Resolution	Advantages
LM-35 series	Active Semiconductor based sensor	-55 to 150°C	±0.75 °C	ADC	10 mV/°C	No external Calibration required low output impedance linear output
DS18B20	Active Semiconductor based sensor	-55 to 125 °C	±0.5 °C (from -10°C to 85°C)	1-wire Interfacing	9 to 12 bits	Programmable Resolution
SHT-Series	Active- Semiconductor based sensor	-40 °C to 120 °C	Highly variable	I2C	14 bit or 12 bit	Can calculate both Temperature and humidity at the same time
ADT7420	Active- Semiconductor based sensor	-40 to 150 °C	±0.25°C	I2C	13 bit or 16 bit	High Resolution
MAX6675	Active- Semiconductor based sensor used along with Thermocouple based sensor	0 °C to 1000°C	±19°C	SPI	12 bit	Generally used at temperatures above 546 °C
MAX6577	Active Semiconductor based sensor	-40 °C to 125°C	±4.5 °C	timer counter	Variable resolution	NA
TMP175	Active Semiconductor based sensor	(-40) °C to 125°C	± 2 °C	SMBus, 2-wire and I2C compatible	12 bit	Programmable Resolution 9 to 12 bits
MLX90614	Infrared sensor	-70°C to +380C	±0.5°C	SMBus	0.02 °C/ bit	Non Contact based sensor
MCP9700	Infrared sensor	-40°C to +125C	±6 °C	ADC compatible	8 or 12 bit	Non Contact based sensor
HEL-700 series	Active Semiconductor based sensor used along with RTD	-75 °C to 540 °C	±1 °F	ADC compatible	Depends on the ADC used	Higher range
PT-100	Platinum RTD	-200 to 600°C	±0.1°C	ADC compatible	depends on ADC used	Higher range
DHT22	Active Semiconductor based	-40 to 125 °C	±0.5°C	single wire communication	0.1 °C	Good for hobbyist projects

Table 2.1 Temperature sensors review

Name	Type	Range	Accuracy	Interfacing	Resolution
2SMPP02	Piezoresistance	0 to 37 kPa	0.8 % FS	ADC compatible	depends on the ADC
BMP280	Barometric	300 to 1100 hPa	±0.12 hPa (12 Pa)	I2C	16-20 bit
MPS20N0040D-S	Piezoresistance	0 to 40 kPa	0.25 % FS	ADC compatible	Depends on the ADC being used in the driver circuit
MPX4115	Barometric	15 to 115 k Pa	± 1.5 % FS	ADC compatible	46 mV/kPa
MPX4115A	Barometric	15 to 115 k Pa	± 1.5 % FS	ADC compatible	46 mV/kPa
MS5849-30BA	unknown	0.3 to 30 bar	± 0.1 bar	SPI, I2C	Missing from the datasheet
MPL115A2S	Gauge	50 to 115 kPa	± 1000 Pa	I2C	150 Pa/bit
MPX5050	Gauge	0 to 50 kPa	± 2.5 % FS	ADC compatible	90mV/kPa
MPXV6115V	Vacuum Gauge	(-115 kPa) to 0 kPa	± 1.5 % FS	ADC compatible	38.26 mV/kPa

Table 2.2 Pressure sensors review

Name	Type	Range	Accuracy	Interfacing	Resolution
SY-HS-220	Capacitive (GPT)	30 to 90 %RH	±5 %RH (at 25 °C)	ADC	0.032 %RH/mV
SCC30-DB	Capacitive	0 to 100 %RH	±3 %RH	I2C	16 bit
SY-HS-230	Capacitive(GPT)	10 to 90% RH	±5 %RH (at 25 °C)	ADC	0.03 %RH / mV
SHT-15	Capacitive	0 to 100 %RH	±2 %RH	I2C	8 bit or 12 bit
SHT41A	Capacitive (GPT)	0 to 100 %RH	±2 %RH	I2C	16 bit
HDC302 (TI)	Capacitive	0 to 100 %RH	±1.5 %RH	I2C	16 bit
HIH8000 (Honeywell)	Capacitive	0 to 100 %RH	±2 %RH (At 25 °C)	I2C and SPI	14 bit
BME-280	Capacitive	0 to 100 %RH	±3 %RH	I2C and SPI	16 bit
HIH9000 (Honeywell)	Capacitive	10 to 90% RH	±1.7 %RH	I2C and SPI	14 bit
HTD2800	Unknown	0 to 100 %RH , 1 to 250 kPa , -40 to 105 °C	± 3 %RH , ± 1 %FS ± 0.5 °C	CAN compatible	0.4% RH/bit , 0.5 kPa/bit , 0.03125°C/bit
T9501	Unknown	0 to 100 %RH	± 3.5 % RH	Digital Modbus RS485 communication	14 bit

Table 2.3 Humidity sensors review

Name	Type	Range	Accuracy	Interfacing	Resolution
HTD2800	Unknown	0 to 100 %RH , 1 to 250 kPa , -40 to 105 °C	± 3 %RH , ± 1 %FS ± 0.5 °C	CAN compatible	0.4% RH/bit , 0.5 kPa/bit , 0.03125°C/bit
T9501	Unknown	0 to 100 %RH	± 3.5 % RH	Digital Modbus RS485 communication	14 bit

Table 2.4 All in one sensors review

CHAPTER 3

TECHNICAL SPECIFICATION

3.1 Software Specification

The STM32CubeIDE is a comprehensive development toolchain from STMicroelectronics for STM32 microcontrollers. It combines the STM32CubeMX graphical configurator with an Eclipse-based IDE and the GNU toolchain for ARM Cortex-M processors. Refer (15) for detialed technical specification information of stm32cube ide. Here are the brief specifications and features:

Integrated Development Environment (IDE):

- Eclipse Framework: Based on the Eclipse platform, offering a familiar environment for many developers.
- Code Editor: Advanced code editor with syntax highlighting, code completion, and other typical IDE features.
- Project Management: Easy project creation and management with project templates for various STM32 devices.

STM32CubeMX Integration:

- Graphical Configurator: Allows easy configuration of STM32 microcontrollers and microprocessors, including pin assignments, clock tree, peripheral configurations, and middleware settings.
- Code Generation: Automatic generation of initialization C code based on configurations.

Debugger:

- Integrated Debugger: Supports ST-LINK, J-Link, and other debuggers.
- Debug Features: Real-time variable monitoring, memory, register views, peripheral register viewers, and graphical data plots.

- SWV Trace: Support for Serial Wire Viewer (SWV) for advanced debugging and tracing.

Compilation Tools:

- GNU Toolchain: GCC-based toolchain for ARM Cortex-M processors.
- Build System: Managed and makefile-based build options.

User Interface:

- GUI Customization: Customizable perspectives and views.
- Multi-tab Layout: Support for multi-tab and multi-window layout for efficient code and resource management.

Version Control:

- Integration with Git: Integrated support for Git version control system for source code management.
- Other VCS: Compatible with other version control systems via plugins.

Supported Languages:

- C/C++ Support: Primary support for C and C++ programming languages.

Middleware:

- HAL/LL Libraries: Includes Hardware Abstraction Layer (HAL) and Low-Layer (LL) libraries for STM32 peripherals.
- RTOS Support: FreeRTOS and other RTOS integrations.
- USB, TCP/IP, File System: Middleware stacks for USB, TCP/IP, FATFS, etc.

Compatibility and Platforms:

- Cross-Platform: Available for Windows, macOS, and Linux.
- STM32 Family Support: Supports all STM32 series, including STM32F, STM32L, STM32H, STM32G, and STM32WB.

Additional Tools and Features:

- Code Analysis: Static code analysis and code metrics tools.

- Profiler: Code profiler to analyze and optimize code performance.
- Resource Explorer: Manage project resources, including linker scripts, startup files, and other assets.

Documentation and Community:

- Extensive Documentation: Comprehensive documentation, user manuals, and application notes.
- Community Support: Access to STMicroelectronics community forums, tutorials, and example projects.

Licensing:

- Free to Use: STM32CubeIDE is free to download and use, with no licensing fees for STM32 microcontroller development.

Similarily, The Arduino Integrated Development Environment (IDE) is an open-source software platform used to write, compile, and upload code to Arduino boards. It is designed to be user-friendly, making it accessible to beginners while providing powerful features for advanced users. Refer (16) for detailed specifications of Arduino IDE. Below are the brief specifications and features of the Arduino IDE:

Integrated Development Environment (IDE):

- User-Friendly Interface: Simple and intuitive interface designed for ease of use.
- Code Editor: Basic text editor with syntax highlighting, auto-indentation, and brace matching.

Board Support:

- Wide Range of Arduino Boards: Supports a variety of Arduino boards, including Uno, Mega, Nano, Leonardo, Due, and more.
- Custom Board Definitions: Ability to add support for third-party boards using Board Manager.

Libraries:

- Built-in Libraries: Includes a wide range of built-in libraries for common functions (e.g., I2C, SPI, EEPROM, WiFi).
- Library Manager: Easy installation and management of additional libraries from the Arduino Library Manager.

Compilation and Upload:

- Compiler: Uses avr-gcc compiler for AVR-based boards, along with other toolchains for different architectures (e.g., ARM for SAMD boards).
- Upload Tool: Integrated tool for uploading compiled code to Arduino boards via USB or other communication interfaces.

Serial Monitor:

- Real-Time Serial Communication: Built-in serial monitor for debugging and communication with the Arduino board.
- Serial Plotter: Graphical tool for plotting serial data in real-time.

Supported Languages:

- Arduino Language: Based on C/C++ with additional simplifications and functions specific to Arduino.

Sketchbook:

- Project Management: Organizes code files into sketches, which are the basic units of code in Arduino.
- Examples: Includes a variety of example sketches for learning and quick prototyping.

Cross-Platform Compatibility:

- Supported Operating Systems: Available for Windows, macOS, and Linux.
- Portability: Sketches can be easily shared and run across different operating systems.

Plug-in and Extension Support:

- Third-Party Plug-ins: Supports additional tools and extensions through the use of plug-ins.
- PlatformIO: Can be integrated with PlatformIO for more advanced features and support for a wider range of hardware.

Community and Documentation:

- Extensive Documentation: Comprehensive documentation and tutorials available on the Arduino website.
- Active Community: Large and active user community with forums, blogs, and online resources for support and collaboration.

Educational Focus:

- Beginner-Friendly: Designed to be easy for beginners to learn programming and electronics.
- STEM Education: Widely used in educational settings for teaching science, technology, engineering, and math (STEM).

Open Source:

- Open-Source Software: Source code is freely available for modification and distribution.
- Hardware Schematics: Schematics and design files for Arduino boards are also open-source.

Licensing:

- Free to Use: The Arduino IDE is free to download and use for both personal and commercial projects.

3.2 Hardware Specification

The STM32F407VET6 is a high-performance microcontroller from the STMicroelectronics STM32F4 series, based on the ARM Cortex-M4 core with FPU (Floating Point Unit). It is designed for a wide range of applications, including industrial control, consumer electronics, and communication devices. Refer Table 3.1 for the hardware specification of STM32F407VET6.

Similarily, The Arduino Uno is one of the most popular microcontroller boards in the Arduino family. It is based on the ATmega328P microcontroller and is widely used for educational, prototyping, and hobby projects due to its simplicity and versatility. Refer Table 3.2 for the hardware specifications of the Arduino Uno.

Architecture	ARM Cortex-M4
Core Frequency	Up to 168 MHz
FPU	Single-precision Floating Point Unit (FPU)
Instruction Set	Thumb-2
Flash Memory	512 KB
SRAM	192 KB (128 KB + 64 KB)
Backup SRAM	4 KB
Internal Oscillator	16 MHz RC oscillator, 32 kHz RC oscillator
External Oscillator	4-26 MHz crystal oscillator, 32.768 kHz crystal oscillator for RTC.
PLL	1 main PLL, 2 PLLs dedicated to audio
Voltage Range	1.8 V to 3.6 V
Power Saving Modes	Sleep, Stop, and Standby modes
Advanced Clock System	Multiple clock sources and a clock tree structure for flexible configuration
USB	Full-speed and high-speed USB OTG (On-The-Go) with full-speed PHY
USART/UART	4 USARTs/2 UARTs
I2C	3 I2C interfaces
SPI	3 SPI interfaces (up to 42 MHz)
I2S	2 I2S interfaces
CAN	2 CAN interfaces
SDIO	1 SDIO interface for SD/SDIO/MMC cards
Ethernet	1 Ethernet MAC with dedicated DMA
General-purpose Timers	10 (16-bit and 32-bit)
Advanced-control Timers	2 (16-bit)
Basic Timers	2 (16-bit)
Real-Time Clock (RTC)	Yes
ADC	3 ADCs, 12-bit resolution, up to 24 channels
DAC	2 DACs, 12-bit resolution
Temperature Sensor	Integrated
Number of I/O Pins	82
Pin Functions	Multiple alternate functions per pin
Direct Memory Access	16-stream DMA controller with FIFOs and burst support.

Table 3.1 Hardware Specification of Stm32f407vet6

Architecture	AVR
Core Frequency	16 MHz
Flash Memory	32 KB (ATmega328P) of which 0.5 KB is used by the bootloader
SRAM	2 KB
EEPROM	1 KB
Pre-installed	Yes, occupies 0.5 KB of flash memory
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Power Supply Options	USB, external adapter (barrel jack), or external battery
Crystal Oscillator	16 MHz
Number of Digital I/O Pins	14 (of which 6 provide PWM output)
Pin Description	D0 to D13
Number of Analog Input Pins	A0 to A5
Analog Input Resolution	10-bit
USART	1 (Serial pins 0 (RX) and 1 (TX))
SPI	1 (pins 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK))
I2C	1 (A4 (SDA), A5 (SCL))
Number of PWM Outputs	6 (pins 3, 5, 6, 9, 10, 11)
Number of External Interrupts	2 (pins 2 and 3)
Typical Consumption	Varies depending on the application, typically around 50 mA
Low Power Modes	Not specifically optimized for low power, but can be put into sleep mode via software.

Table 3.2 Hardware Specification of Arduino UNO

CHAPTER 4

DESIGN APPROACH AND DETAILS

4.1 Design Approach

Sensors on the STM32 board will tell us about the humidity, temperature and pressure. NI DAQ alone will tell us about potential differences between the antenna during lightning at 2 MSPS. NI DAQ has been connected through high speed USB cable and STM32 has been connected to LAN. STM32 can send data at a max baud rate of 3 Mbps. STM32 has been interfaced with UART to USB and then USB to LAN converter for faster communication. Fig 4.1 depicts the block diagram of the project.

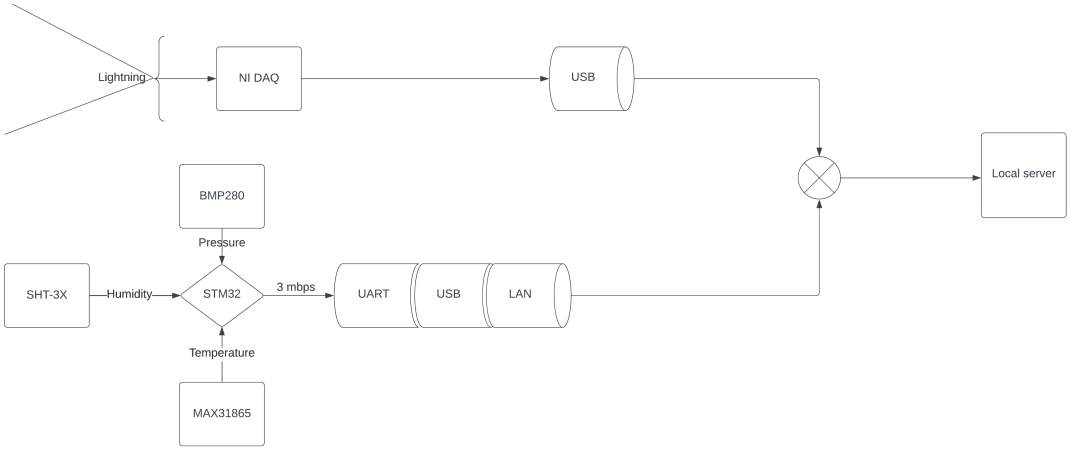

Fig. 4.1 Design approach

4.1.1 Proof of concept

At first, the Arduino UNO Microcontroller board was used to demonstrate the proof of concept of the Lightning station. Sensors like DHT22, BMP280, PT100 were used to measure humidity, pressure and temperature respectively. Surprisingly, both DHT22 and BMP280 can be used to measure temperature of the environment, but still PT-100 was preferred for temperature readings only because both DHT22 and BMP280 can't record temperatures more than 80 °C and 120 °C respectively. Whereas, PT100 being a thermocouple can measure upto 600 °C accurately at a tolerance value of 0.1 °C.

For measuring temperature a simple signal conditioning circuit was built which consisted of a wheatstone bridge, where three known resistances of 47k ohm were used along with PT-100 was used and a known source voltage of 3.3 V was used. For using PT100 in a least erroneous condition it is always advisable that the current going through PT100 should not exceed 1 mA, because if the current exceeds 1 mA then there is a phenomenon of self heating in PT100 which adds up in the final output value and thus makes our readings inaccurate.

Next, the BMP280 Pressure sensor was interfaced with Arduino UNO with the help of I2C. To interface BMP280 with a microcontroller:

- Include the Adafruit BMP280 library.
- Define a sensor object (bmp) and the I2C address of the BMP280 sensor.
- Then, Initialize serial communication for communication of BMP with PC.
- Use the begin() function of the bmp object to start communication with the sensor.
- Check if the sensor is detected and print a message accordingly.
- In a loop read the temperature and pressure
- Then apply calibration factors from the datasheet to get more accurate pressure readings.
- Add a delay to control the frequency of pressure readings

Later, a DHT22 sensor was used to measure the humidity. The DHT22 sensor uses a single-bus communication protocol to transmit data. This means it uses only one data pin to send temperature and humidity readings to the microcontroller. The communication involves a specific timing sequence where the microcontroller initiates the data transmission by sending a start signal, then waits for a response signal from the sensor, and finally reads the data bits transmitted by the sensor. The microcontroller initiates communication by sending a start signal, which is a low pulse for a specific duration. The DHT22 sensor then responds with its own start signal followed by the actual data transmission. The data transmission consists of 40 bits, including integer and decimal parts for both humidity and temperature readings, and a checksum for error checking. The microcontroller needs to time these data bits accurately to ensure correct data reception. Refer Fig 4.2 for understanding the single wire communication in DHT22. Fig 4.2 has been referred from its original company data sheet (17).

Refer Fig 4.3 for the whole circuit of our lightning system soldered on a general purpose board. This circuit is the proof of concept where Arduino has been used. But this Lightning system as some few disadvantages:

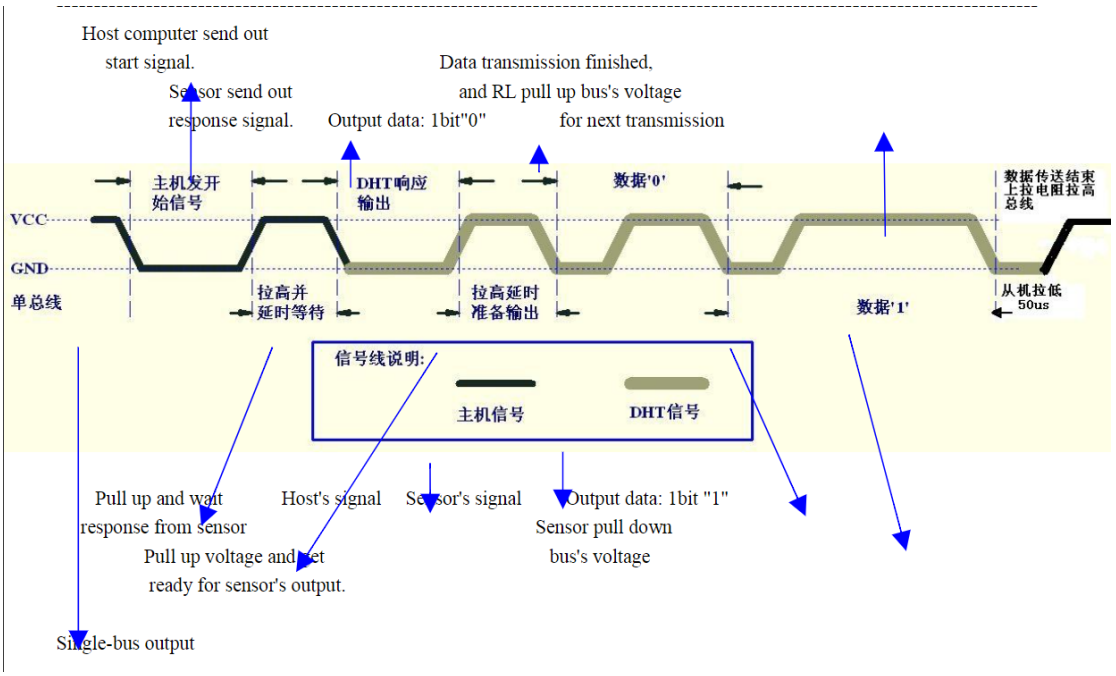

Fig. 4.2 DHT22 single wire communication procedure

- The Maximum frequency of the crystal oscillator of Arduino is 16 Mhz. That is way too less and it doesn't make Arduino UNO faster while taking data.
- DHT22 gives very accurate temperature and humidity readings but it is very slow in nature.
- PT100 is an ideal choice for taking temperature readings but using it inside a wheatstone bridge nullifies its advantages because the obtained values are very unstable in nature and consecutive data values can vary from ±10 °C.

Looking at all these disadvantages, it became important to find a faster MCU board, a faster Humidity sensor and a faster and reliable temperature sensor.

In Fig 4.3 an AC-DC adapter was used to give power supply to the Arduino. The adapter rating was 12V and 2 Amp supply voltage and current respectively, and it is called barrel jack wall adapter.

Refer Fig 4.4 for the whole arduino circuit diagram

4.1.2 Communication Channel for transferring data

At first, it was decided that Arduino will be used with some Wi-Fi modules or any other bluetooth or LoRa modules to send data wirelessly to the server.

But there were some other technical problems with this concept and that was the interference of lightning with the wireless communication systems.

Lightning often generates EMP pulses that causes interference with the communication channels and often corrupts the data packets. After looking at this aspect it was

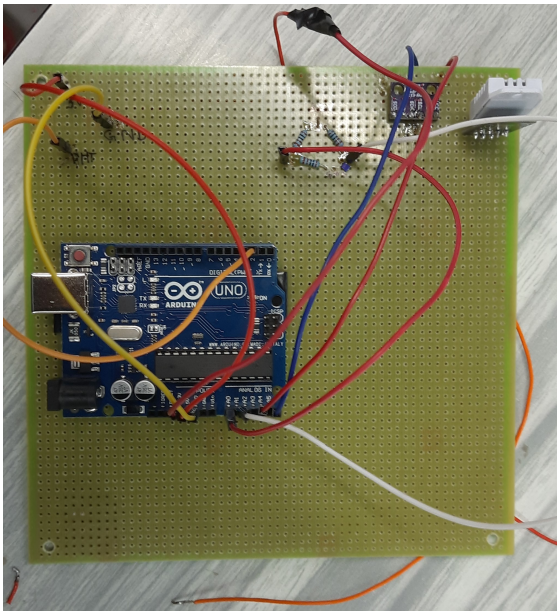

Fig. 4.3 Arduino on General purpose board

decided to use either LAN cables or USB cables to send data at faster rates without any interference of the lightning.

4.1.3 Upgrading to better modules

As mentioned earlier, DHT22 was replaced with SHT-3X and PT100 will be interfaced with MAX31865 and then with the microcontroller because then the obtained temperature values will be stable in nature. BMP280 is already a good choice thus, it won't be replaced at all.

Since, we know that Arduino's max frequency is 16 MHz. Thus, we need a MCU Unit with a higher frequency unit. Refer Table 4.1 for STM boards comparison. After a brief comparison, STM32F407 series of MCU's were selected and that too was STM32f407VET6 because of the following reasons:

Feature	STM32F469	STM32F407	STM32F2X7
Frequency	180 MHz	168 MHz	120 MHz
ADC	2.4 MSPS	2.4 MSPS	2 MSPS
UART	11.5 mbps	11.25 mbps	7.5 mbps
SPI	45 mbps	45 mbps	30 mbps
I2C	Available	Available	Available
SDIO	NA	Available	Available

Table 4.1 ST MCU Board comparison table

- Frequency = 168 MHz
- ADC = 2 MSPS

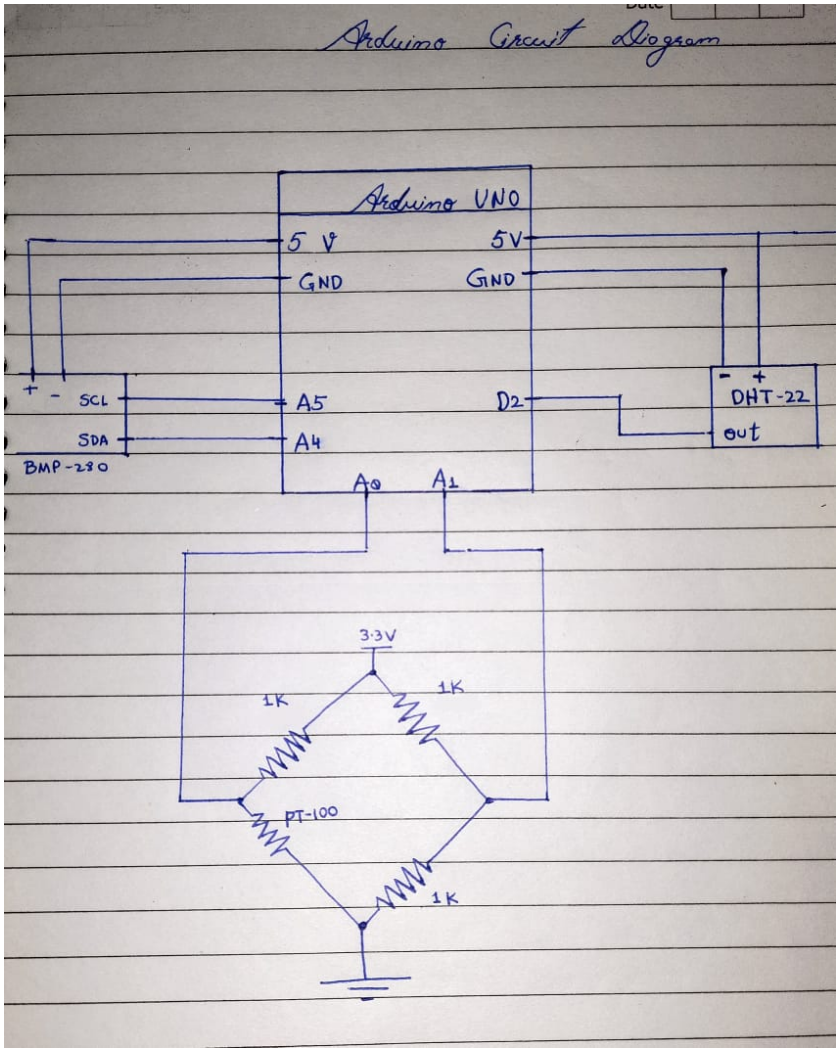

Fig. 4.4 Arduino Circuit diagram

- UART = 11.25 Mbps
- SPI = 45 Mbps
- I2C 100 kHz , 400 kHz

There was also a choice of using Microchip boards like XmegaA3BU, but the level of experience on these boards was very naive.

4.1.4 Interfacing all the upgraded sensors on STM32 board

In Fig 4.5 stm32f407vet6 has been interfaced with humidity, pressure and temperature sensors. Refer Fig 4.6 for the STM board circuit diagram.

Components used:

- STM32F407VET6- MCU Board.
- SHT3X - humidity sensor

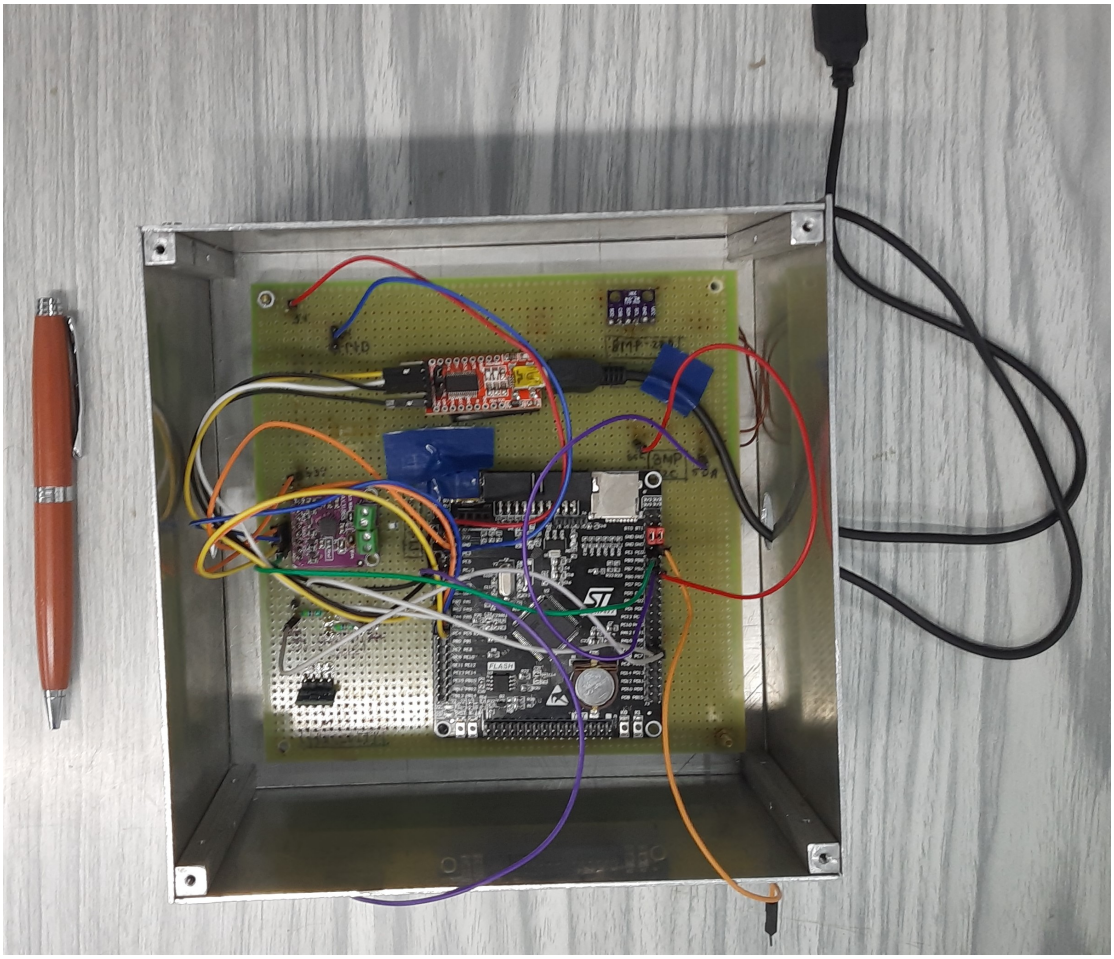

Fig. 4.5 STM32F407VET6 Interfaced with other sensors

- MAX31865 - temperature sensor
- BMP280 - pressure sensor
- FTDI Module - UART to USB Converter

4.1.5 Configurations of PT100 temperature sensor

We can't use PT100 directly with some microcontroller, here MAX31865 IC is required to measure the resistance changes of PT100 in different temperature environments.

Here's a breakdown of the MAX31865:

Function: It acts as a converter, taking the resistance measured by the RTD and turning it into a digital signal a microcontroller can understand.

Benefits of RTDs: RTDs, particularly PT100s (a type of RTD with a resistance of 100 ohms at 0°C), are known for their excellent accuracy and stability compared to other temperature sensors. This makes them ideal for applications requiring precise temperature measurement.

How it Works: The MAX31865 measures the resistance of the RTD compared to a

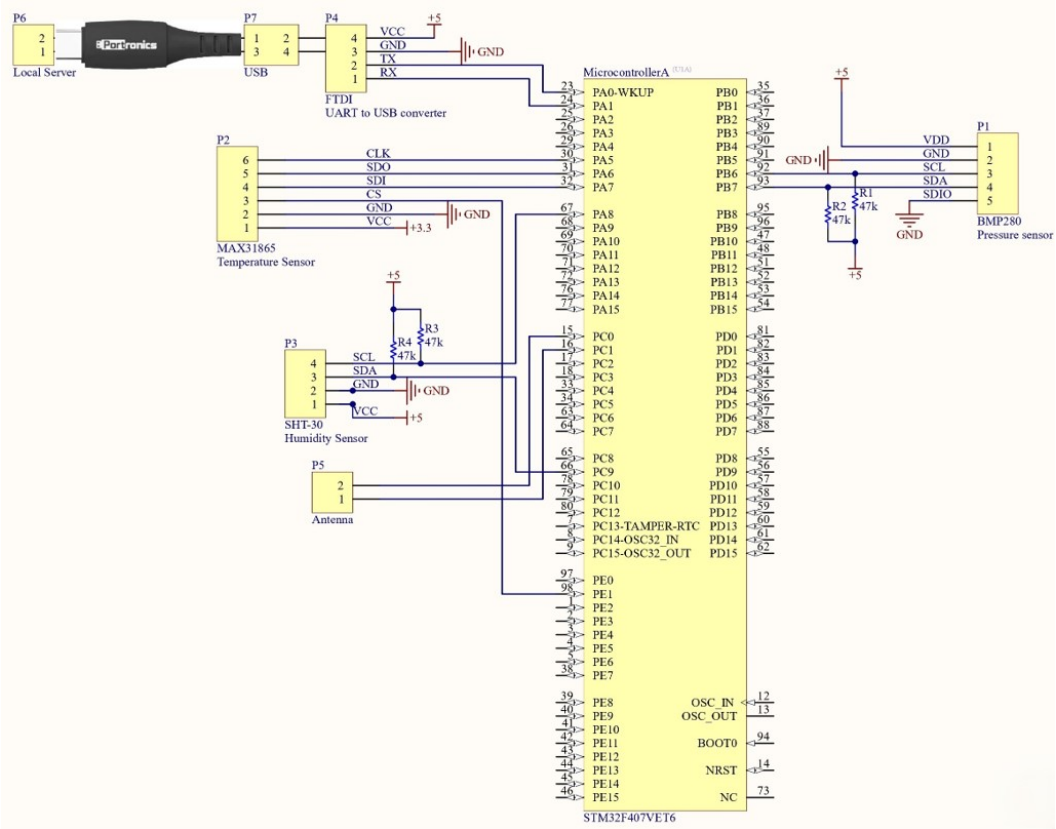

Fig. 4.6 STM32F407VET6 circuit diagram

reference resistor. Then, it uses a built-in analog-to-digital converter (ADC) to convert that ratio into a digital value which is transferred to an MCU using SPI communication protocol.

Platinum RTD can be interfaced with MAX31865 in 3 different configurations and they are as follows:

2 Wire Configuration: The most basic RTD measurement uses a two-wire RTD for temperature measurement. Shown below is a schematic and design for a two-wire RTD measurement with an ADC. A ratiometric measurement is created with the RTD as the input and a precision resistor as the reference input. Refer Fig 4.7 for 2 wire configuration of PT-100.

Pros:

- Simplest implementation of RTD temperature measurement
- Uses only two analog input pins for measurement and one IDAC current for sensor and reference resistor excitation
- Good for local measurements, where the lead length and resistance are small
- Ratiometric measurement, IDAC noise and drift are canceled

Cons:

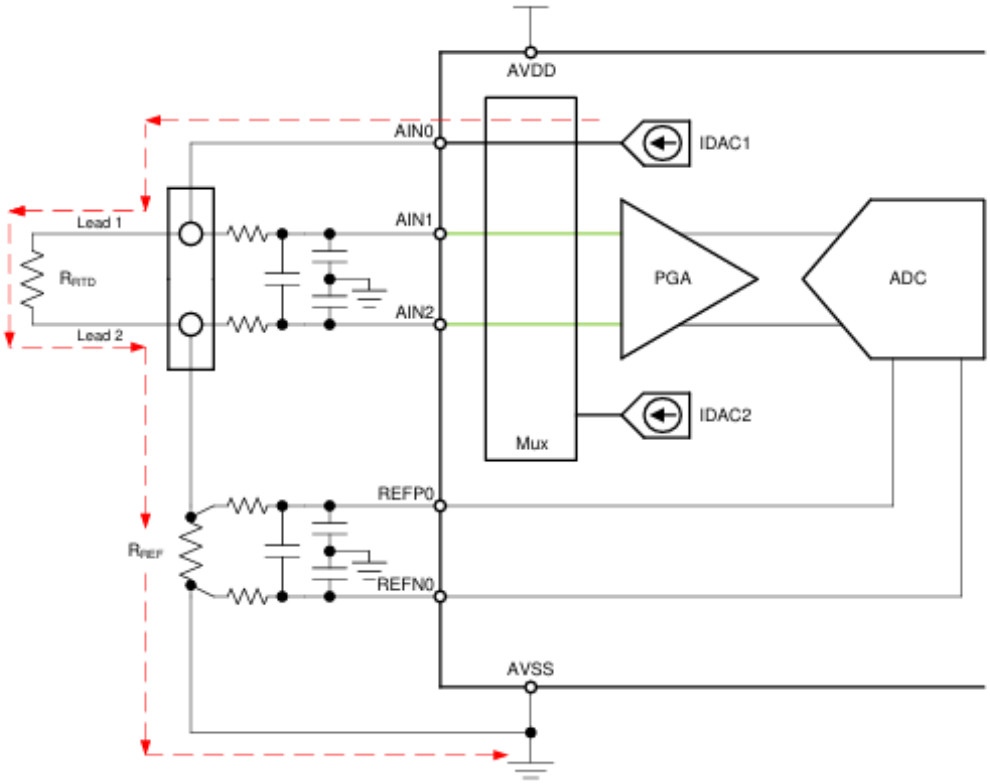

Fig. 4.7 2 Wire Configuration

- Least accurate measurement for RTDs
- No lead wire compensation; lead resistance affects measurement accuracy

3 Wire Configuration: For three-wire RTD design, two matched IDAC current sources are used to actively cancel the lead resistance errors. IDAC1 sources current through lead 1 of the RTD to both the RTD and the reference resistor, RREF. IDAC2 sources current through lead 2 of the RTD to the reference resistor. If IDAC1 and IDAC2 are identical and the lead resistances match, then the error from the lead resistances cancels in the measurement made from AIN1 and AIN2.

Pros:

- IDAC currents are used for sensor and reference resistor excitation
- Allows for lead wire compensation; errors from voltage drops across lead resistances are removed
- Ratiometric measurement, IDAC noise and drift are canceled

Cons:

- Requires two matched IDAC currents for both lead wire compensation and for RTD measurement

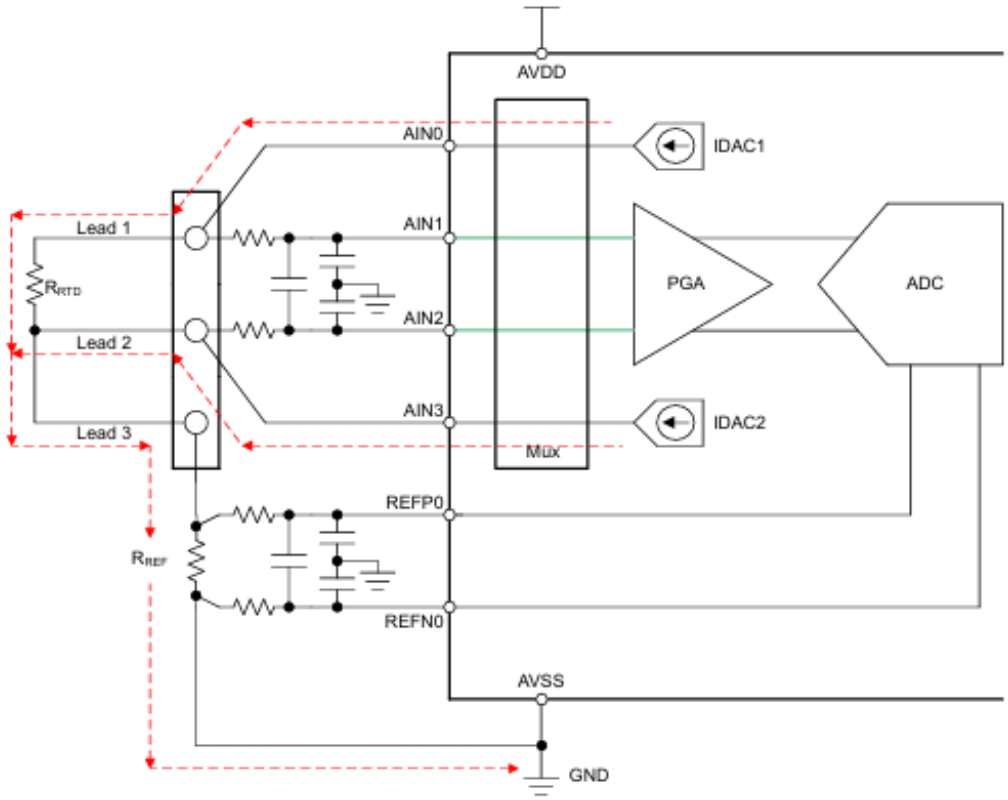

Fig. 4.8 3 Wire Configuration

4 Wire Configuration: In this configuration the IDAC current is being routed through an alternate input and measurements are taken from AIN1 and AIN2. However, the considerations in reference resistor size, IDAC current, reference voltage, and PGA input voltage are exactly the same as that of 2 wire configuration.

Pros:

- Single IDAC current is used for sensor and reference resistor excitation
- Most accurate RTD measurement, no need for lead compensation
- Ratiometric measurement, IDAC noise and drift are canceled

Cons:

- The four-wire RTD is the most expensive RTD configuration

Conclusion: For the sake of not complicating the task, PT100 has been used in 2 wire configuration. Refer (18) mannual for more detailed information on PT100 interfacing.

4.1.6 Interfacing MAX31865 sensor with STM32

MAX31865 is a RTD driver IC used for interfacing any choice of MCU with any given RTD's. The following stated information has been cited from (19).

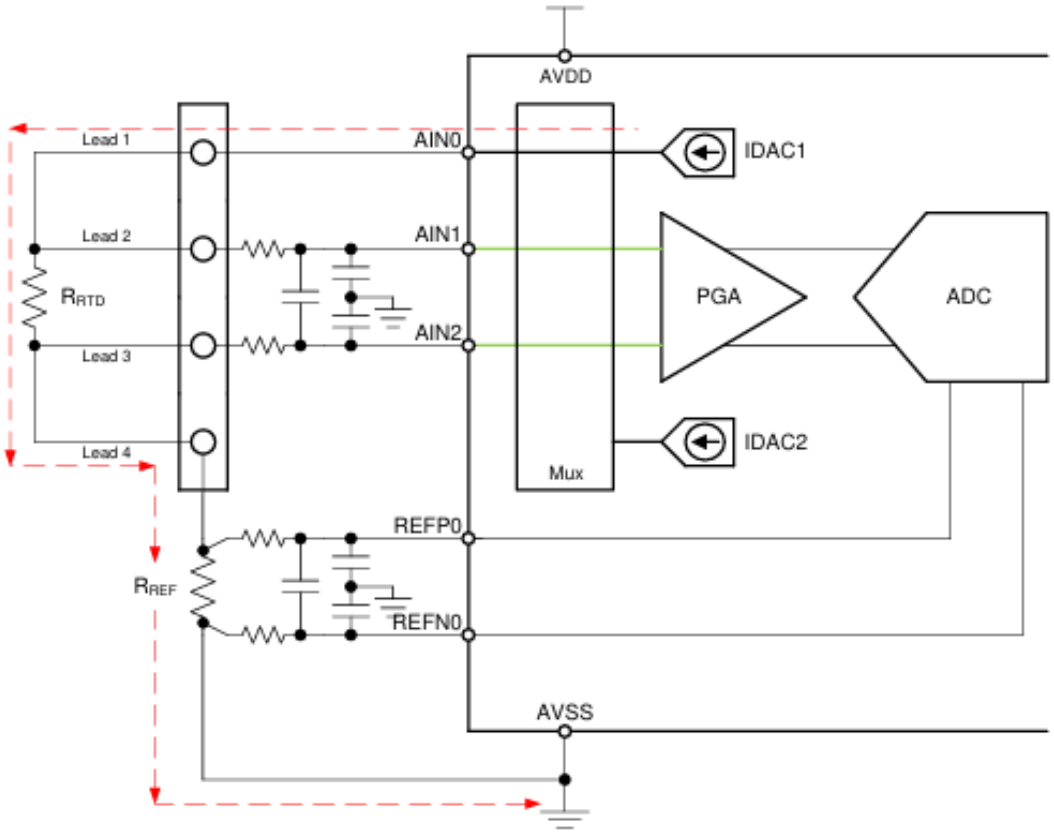

Fig. 4.9 4 Wire Configuration

Step 1: Hardware Setup

- Connect the MAX31865 to the STM32F407VET6:
- Connect the VCC pin of the MAX31865 to the 3.3V power supply of the STM32.
- Connect the GND pin of the MAX31865 to the ground (GND) of the STM32.
- Connect the SDO pin of the MAX31865 to an SPI MISO pin (e.g., PA6) on the STM32.
- Connect the SDI pin of the MAX31865 to an SPI MOSI pin (e.g., PA7) on the STM32.
- Connect the SCK pin of the MAX31865 to an SPI clock pin (e.g., PA5) on the STM32.
- Connect the CS (chip select) pin of the MAX31865 to a GPIO pin (e.g., PA4) on the STM32.
- Connect the RTD (PT100) to the MAX31865 according to the datasheet, usually involving four connections: RTDIN+, RTDIN-, RTDREF+, and RTDREF-.

Step 2: Initialize the SPI Interface

- Configure the SPI Peripheral.
- Initialize the SPI peripheral on the STM32F407 to communicate with the MAX31865.
- Set the SPI clock speed, mode (typically mode 1 or 3), and data frame format.
- Enable the SPI peripheral.

Step 3: Configure the MAX31865

- Initialize the MAX31865.
- Configure the MAX31865 by writing to its configuration registers. This includes setting parameters like the RTD type (e.g., PT100), the number of wires (2, 3, or 4), and the conversion mode (continuous or single-shot).

Step 4: Read Data from the Sensor

- Send Configuration Command: Use SPI communication to send the necessary configuration commands to the MAX31865. This usually involves writing to a control register to set up the sensor.
- Start a Measurement: Initiate a temperature measurement by writing to the appropriate register on the MAX31865.
- Wait for Measurement Completion: Wait for the measurement to complete. This may involve polling a status register or waiting for a predefined time based on the conversion mode.
- Read Measurement Data: Once the measurement is complete, read the temperature data from the MAX31865 by performing an SPI read operation. This involves reading multiple bytes from the appropriate data registers.

Step 5: Process the Sensor Data

- Process Raw Data: The data read from the MAX31865 is usually in a raw format. Convert this raw data to meaningful temperature values using the formulas provided in the MAX31865 datasheet or by using the Callendar-Van Dusen equation for PT100 sensors.
- Handle Errors: Implement error handling for SPI communication errors or sensor-specific error statuses, such as open or shorted RTD connections.

Summary of SPI Communication Steps:

- Chip Select Low: Select the MAX31865 by pulling the CS pin low.
- Send Device Command: Send the command byte(s) to the MAX31865, specifying whether you are writing or reading.
- Send/Receive Data: Write configuration data or read measurement data from the MAX31865 as required.
- Chip Select High: Deselect the MAX31865 by pulling the CS pin high to end the SPI transaction.
- Process and Use Data: Process the received data and use it in your application.

4.1.7 Interfacing SHT-3X sensor with STM32

The following stated information has been cited from (20).

Step 1: Hardware Setup

- Connect the SHT-3X Sensor to the STM32F407VET6:
- Connect the VCC pin of the SHT-3X to the 3.3V power supply of the STM32.
- Connect the GND pin of the SHT-3X to the ground (GND) of the STM32.
- Connect the SDA (data line) of the SHT-3X to one of the I2C data pins (e.g., PB9) on the STM32.
- Connect the SCL (clock line) of the SHT-3X to one of the I2C clock pins (e.g., PB8) on the STM32.
- Optionally, add pull-up resistors (typically 47k ohms) to the SDA and SCL lines if they are not already on the sensor module.

Step 2: Initialize the I2C Interface

- Configure the I2C Peripheral:
- Initialize the I2C peripheral on the STM32F407 to communicate with the SHT-3X.
- Set the I2C clock speed (typically 100kHz or 400kHz).
- Enable the I2C peripheral.

Step 3: Configure the Sensor

- Wake up and Initialize the SHT-3X.

- The SHT-3X sensor requires no special wake-up commands. It is ready to communicate once powered up.
- Check the sensor's datasheet for any specific initialization sequences if necessary.

Step 4: Read Data from the Sensor

- Send Read Command: Use the I2C protocol to send a measurement command to the SHT-3X. The sensor's datasheet specifies various commands for different measurement modes.
- Typically, a measurement command consists of a write operation where you send the command byte(s) to the sensor's address.
- Wait for Measurement Completion:The sensor requires some time to complete the measurement. Refer to the datasheet for the appropriate delay.
- Read Measurement Data:Once the measurement is complete, read the data from the sensor. This involves an I2C read operation where you request a certain number of bytes from the sensor.

Step 5: Process the Sensor Data

- Process Raw Data: The data read from the sensor is usually in a raw format. Convert this raw data to meaningful values (e.g., temperature and humidity) using the formulas provided in the sensor's data sheet.
- Handle Errors: Implement error handling for I2C communication errors or sensor-specific error statuses.

Summary of I2C Communication Steps:

- Start Condition: Initiate communication by generating a start condition on the I2C bus.
- Send Device Address: Send the 7-bit I2C address of the SHT-3X followed by the read/write bit (write for sending commands, read for receiving data).
- Send Command/Data: Write the command or data bytes to the sensor.
- Repeated Start Condition: If needed, generate a repeated start condition to switch from write to read mode.
- Read Data: Read the specified number of bytes from the sensor.
- Stop Condition: Terminate communication by generating a stop condition on the I2C bus.

4.1.8 Interfacing BMP280 sensor with STM32

The following stated information has been cited from (21)

Step 1: Hardware Setup

- Connect the BMP280 to the STM32F407VET6:
- Connect the VCC pin of the BMP280 to the 3.3V power supply of the STM32.
- Connect the GND pin of the BMP280 to the ground (GND) of the STM32.
- Connect the SDA (data line) of the BMP280 to one of the I2C data pins (e.g., PB9) on the STM32.
- Connect the SCL (clock line) of the BMP280 to one of the I2C clock pins (e.g., PB8) on the STM32.
- Add pull-up resistors (typically 4.7k ohms) to the SDA and SCL lines if they are not already on the sensor module.

Step 2: Initialize the I2C Interface

- Configure the I2C Peripheral.
- Initialize the I2C peripheral on the STM32F407 to communicate with the BMP280.
- Set the I2C clock speed (typically 100kHz or 400kHz).
- Enable the I2C peripheral.

Step 3: Configure the BMP280

- Initialize the BMP280: Reset and configure the BMP280 by writing to its configuration registers. This includes setting parameters like oversampling rates for temperature and pressure, the power mode (normal, forced, or sleep), and the IIR filter coefficient.

Step 4: Read Data from the Sensor

- Send Configuration Command: Use the I2C protocol to send the necessary configuration commands to the BMP280. This usually involves writing to control and configuration registers.
- Start a Measurement: Initiate a temperature and pressure measurement by writing to the appropriate register on the BMP280.

- Wait for Measurement Completion: Wait for the measurement to complete. This may involve polling a status register or waiting for a predefined time based on the oversampling settings.
- Read Measurement Data: Once the measurement is complete, read the temperature and pressure data from the BMP280 by performing I2C read operations. This involves reading multiple bytes from the appropriate data registers.

Step 4: Process the Sensor Data

- Process Raw Data: The data read from the BMP280 is usually in a raw format. Convert this raw data to meaningful temperature and pressure values using the compensation formulas provided in the BMP280 data sheet.
- Handle Errors: Implement error handling for I2C communication errors or sensor-specific error statuses.

Summary of I2C Communication Steps:

- Start Condition: Initiate communication by generating a start condition on the I2C bus.
- Send Device Address: Send the 7-bit I2C address of the BMP280 followed by the read/write bit (write for sending commands, read for receiving data).
- Send Command/Data: Write the configuration data or read the measurement data from the BMP280 as required.
- Repeated Start Condition: If needed, generate a repeated start condition to switch from write to read mode.
- Read Data: Read the specified number of bytes from the BMP280.
- Stop Condition: Terminate communication by generating a stop condition on the I2C bus.
- Process and Use Data: Process the received data and use it in your application.

4.1.9 Laboratory Experiment

A Laboratory experiment for the simulation of lightning was conducted to test how lightning is observed on an oscilloscope. Here the Vandegraff generator was used to simulate lightning strikes. A Pre-amplifier was used to amplify the lightning signal in case the signal was too small to detect. Observations have been made in 2 modes i.e. differential mode (where signal is being taken from 2 Antennas in the vicinity) and non-differential mode (where signal is being taken from only 1 antenna and the other antenna

has been grounded).Ref Fig 4.10 for the insights of the laboratory experiment. Refer Fig 4.11 for the graph of non-differential antenna setup, in this setup one of the antennas was grounded and the other one was connected to the oscilloscope. The obtained graph on oscilloscope clearly shows how lightning can be seen graphically when data acquisition is done at 2 MSPS. Refer Fig 4.12 for the graph of differential antenna setup, in this setup both the antennas were connected to one channel of the oscilloscope. Refer Fig 4.13 for the antenna signal with pre-amplifier. Refer Fig 4.14 for differential antenna circuit setup. Refer Fig 4.15 for antenna setup with pre-amplifier.

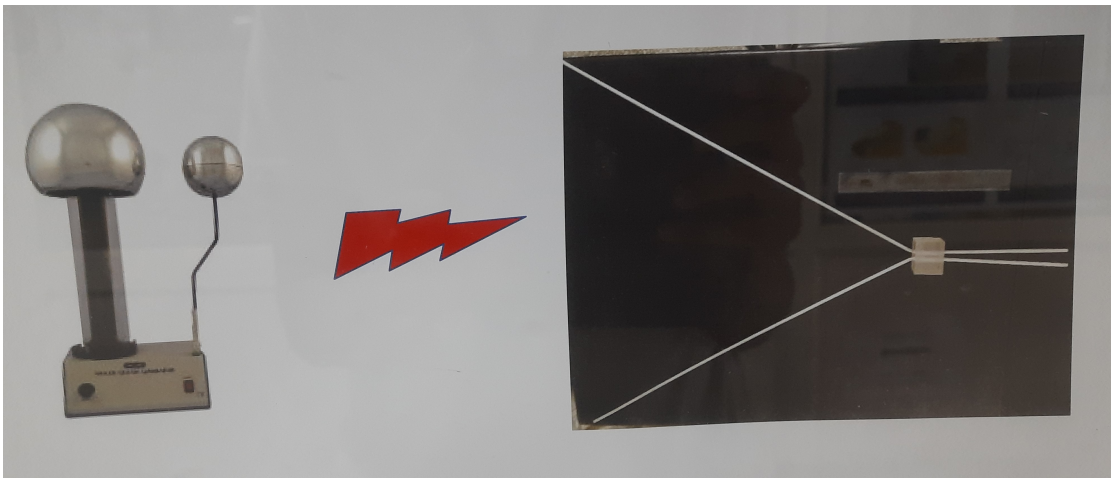

Fig. 4.10 Vandegraff setup

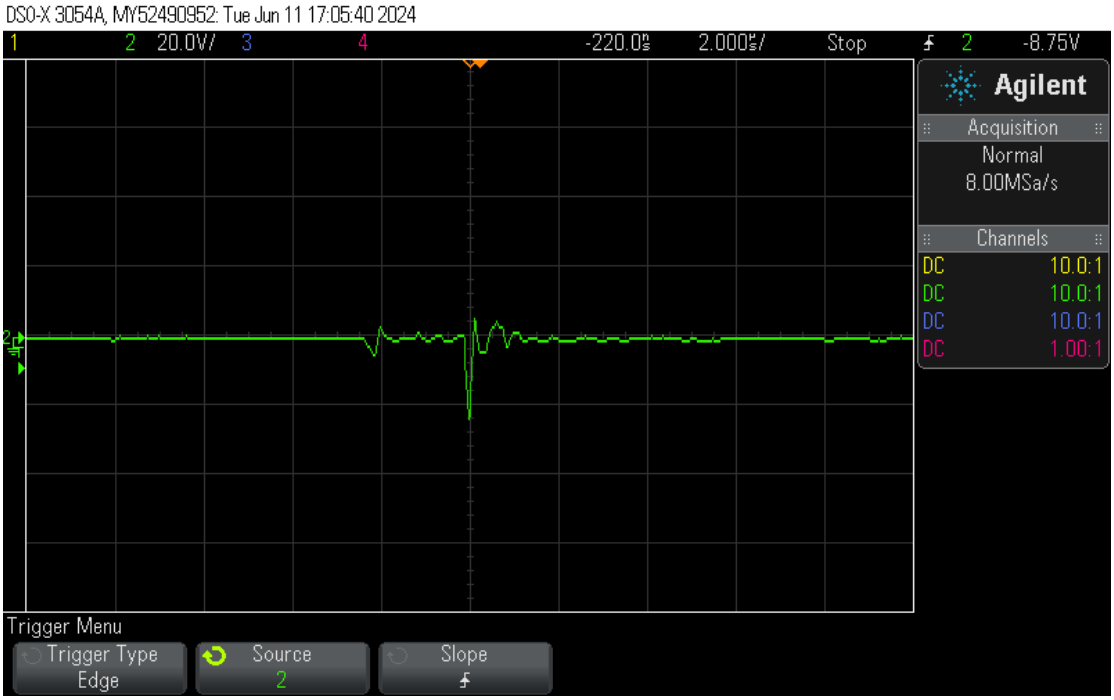

Fig. 4.11 Antenna non differential signal

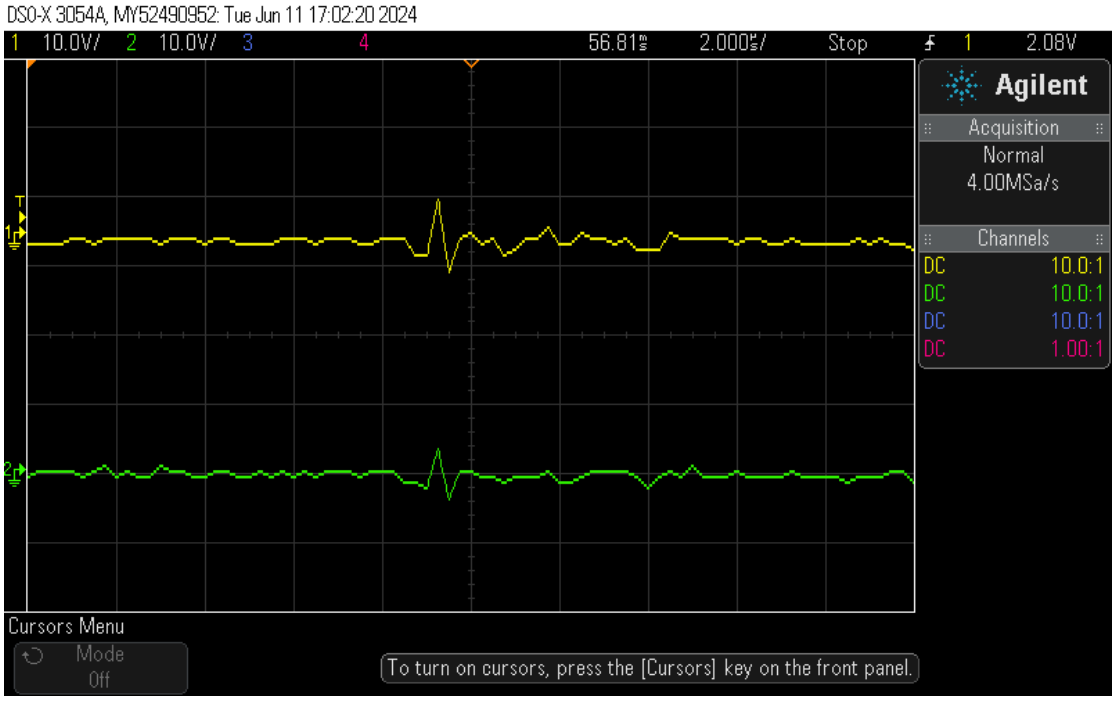

Fig. 4.12 Antenna differential signal

Later, it was concluded that non-differential antenna setup yields better readings for visualization and no more time needs to be spent on computation rather than the differential one.

Moreover, inclusion of pre-amplifier is also necessary because it amplifies very small lightning signals at a decent scale which later makes the observations and analysis simpler.

4.1.10 Terrace Experiment

The same laboratory experiment was later again executed on the terrace using NI DAQ. Lightning was again simulated using the Vandegraff generator.

4.1.11 Assembling the Final Setup

After successful testing of all the components, a final assembly was carried out and a final test was done to check the credibility of the setup. Fig 4.16 shows the operation of MCU board after final assembling. Fig 4.17 depicts the setup of instruments for lightning data acquisition.

4.2 Constraints, Alternatives and Trade offs

Speed of sending data from STM32 and NI DAQ might be very different because DAQ sends only lightning data, but STM32 has to gather 3 different types of data one by one

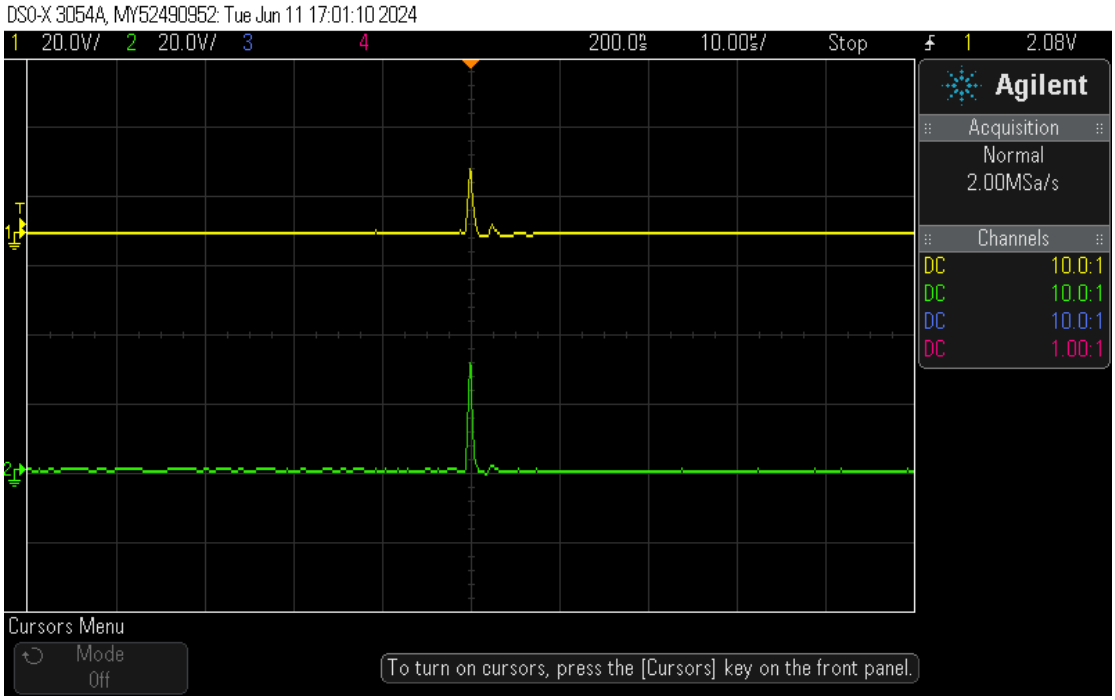

Fig. 4.13 Antenna signal with pre-amplifier
Yellow Signal: Input Signal
Green Signal: Output of the pre-amplifier

using both I2C and SPI communication. Thus, there is a possibility that after gathering data there might be a problem of mapping them. The possible alternative to this constraint would have been to use FPGA's for all sorts of data collection and then using SPI communication to send it local server.

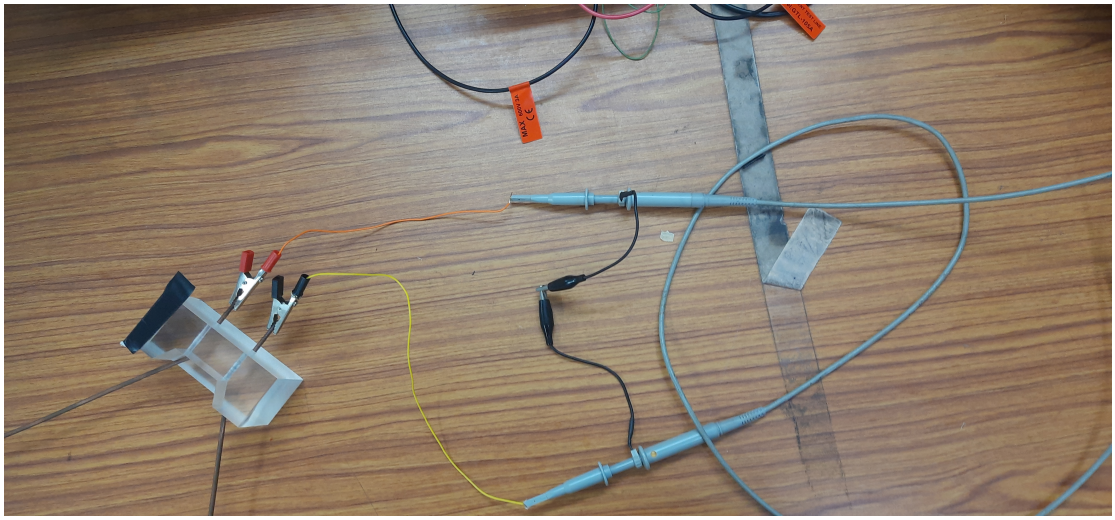

Fig. 4.14 Antenna setup for differential reading

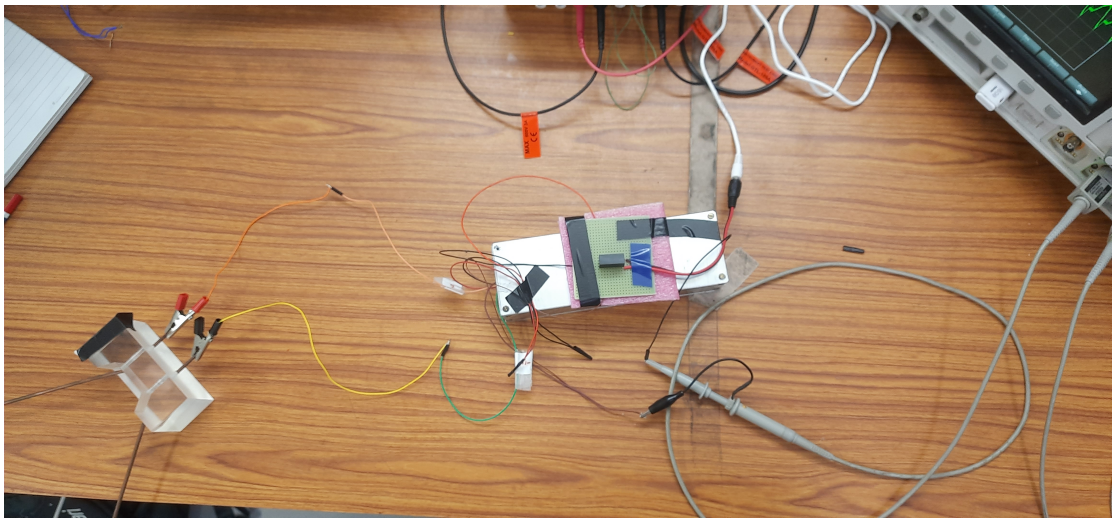

Fig. 4.15 Antenna setup with pre-amplifier

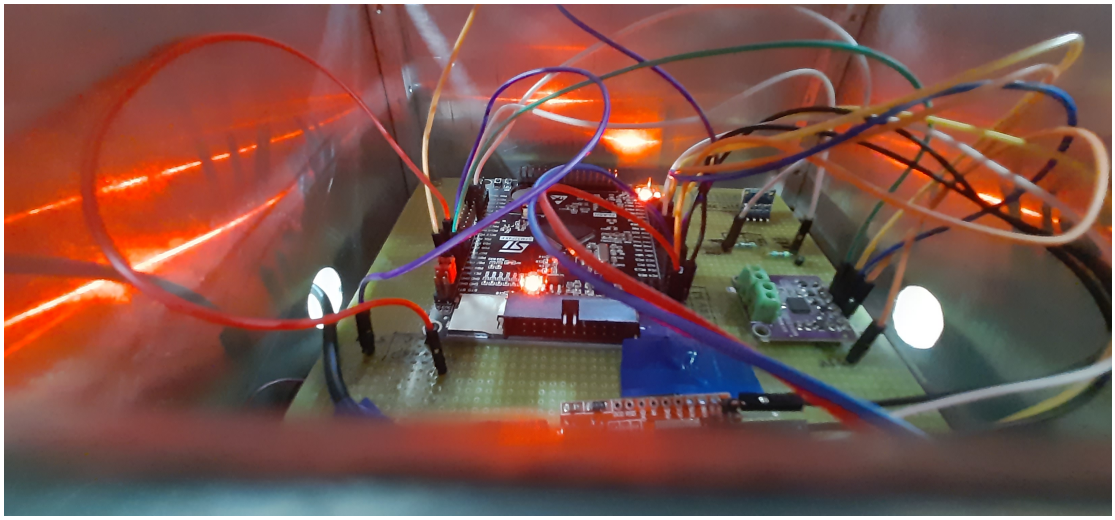

Fig. 4.16 MCU inside the box

Fig. 4.17 Terrace Setup

CHAPTER 5

MATHEMATICAL EQUATIONS

5.1 Math Equation in Thesis

5.1.1 ADC Sampling frequency calculation

Given that:

$$APB2_{clk} = 70\,MHz$$
$$prescaler = 2$$

Thus, our conversion time will be:

$$ADC_{clk} = \frac{APB2_{clk}}{prescaler} \tag{5.1}$$

$$T_{conv} = \frac{Sampling_cycles + Resolution}{ADC_{clk}} = \frac{12+3}{\frac{70*10^6}{2}} = 0.428\,\mu sec \tag{5.2}$$

Refer (15) for the ADC formulas

Thus, our sampling frequency can be written as:

$$f_{sampling} = \frac{1}{T_{conv}} = 2.33\,MSPS \tag{5.3}$$

which is the maximum ADC sampling frequency our STM32F407VET6 board can operate on.

5.1.2 UART Baud Rate Calculation

Given that:

$$APB1_{clk} = 70\,MHz \tag{5.4}$$

$$USARTDIV = [1, 2^{13} - 1] \tag{5.5}$$

Refer (15) reference mannual for more details regarding the working of UART in STM32

Since, the maximum Baud Rate limit of FTL232RL is 3 Mbps thus, the maximum baud rate of our STM board should not exceed 3 Mbps mark.

$$BaudRate_{max} = 3\,Mbps \tag{5.6}$$

Refer (22) for more details about FTDI module

$$BaudRate = \frac{APB1_{clk}}{8 * USARTDIV} \tag{5.7}$$

If USARTDIV = 2 then baud rate will be as follows:

$$BaudRate = \frac{70 * 10^6}{8 * 2} = 4.375\,Mbps \tag{5.8}$$

which exceeds the 3 Mbps limit of our FTDI module. Thus, USARTDIV cannot be equal to 2.

If USARTDIV = 3 then baud rate will be as follows:

$$BaudRate = \frac{70 * 10^6}{8 * 3} = 2.916666\,Mbps \tag{5.9}$$

which doesn't exceeds the 3 Mbps limit of our FTDI module. Thus, USARTDIV can be equal to 3.

5.1.3 Wheatstone bridge Pt-100

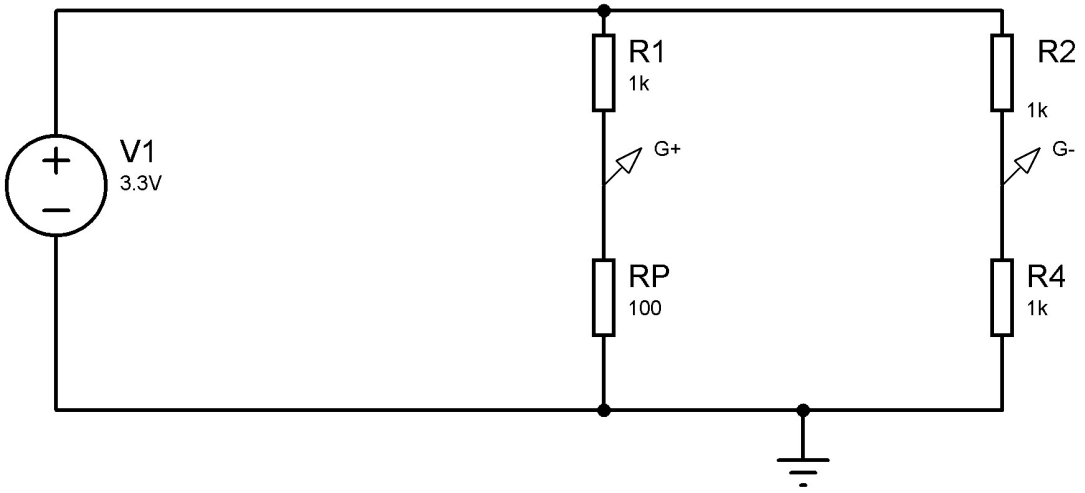

Fig. 5.1 Wheatstone Bridge

Using KVL and KCL in the circuit diagram of Fig 5.1 we get:

$$3.3 - I_1 * (R + R_p) = 0 \tag{5.10}$$

$$3.3 - I_2 * (R + R) = 0 \tag{5.11}$$

We know that

$$I_1 + I_2 = I \tag{5.12}$$

Using equation 5.10 and 5.11 we can write as:

$$I_1 = \frac{3.3}{R + R_p} \tag{5.13}$$

$$I_2 = \frac{3.3}{2 * R} \tag{5.14}$$

Again using KVL inside the wheatstone bridge, we can get:

$$G - R_p * I_1 + R * I_2 = 0 \tag{5.15}$$

We can write equation 5.15 as:

$$G - R_p * \frac{3.3}{R + R_p} + R * \frac{3.3}{2 * R} = 0 \tag{5.16}$$

$$G = \frac{3.3 * R_p}{R + R_p} - \frac{3.3}{2} \tag{5.17}$$

Now, rewriting the whole equation 5.17 with respect to Rp we get:

$$R_p = R * \frac{3.3 + 2 * G}{3.3 - 2 * G} \tag{5.18}$$

Finally. to convert the resistance value into its equivalent temperature value we can use Callendar-Van Dusen Equation for temperatures greater than 0 °C. Refer (18) for more information about Callendar-Van Dusen equation.

$$R_{RTD}(T) = R_0[1 + (AT) + (BT^2)] \tag{5.19}$$

There is also a simplified version of this equation and that is:

$$R_{RTD}(T) = R_0[1 + a * T] \tag{5.20}$$

where a = 0.00385 $^{\circ}C^{-1}$

Refer (23) for the modified Callendar-Van Dusen equation.

CHAPTER 6

PROJECT DEMONSTRATION

The following is the brief demonstration of the project where real time lightning data was recorded with the help of NI DAQ. Moreover STM32 data was also collected for this brief moment. Fig 6.1 are the real time lightning pulses recorded through NI DAQ. Fig 6.2 shows the real time data acquisition of STM32 on labview.

6.1 Figure

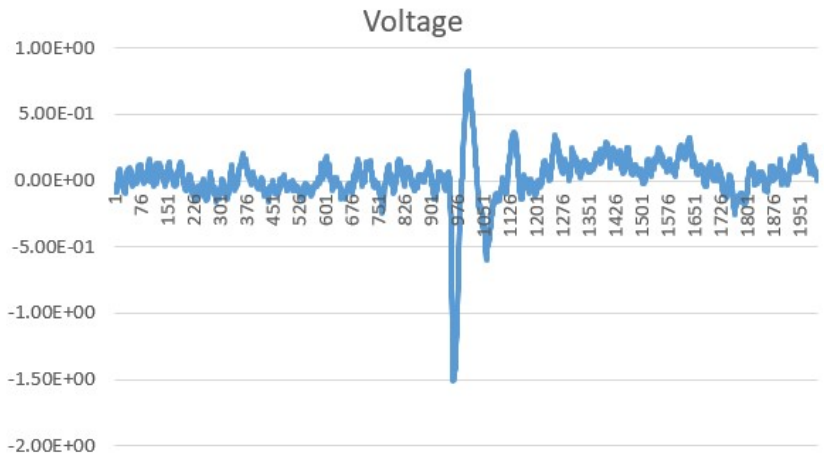

Fig. 6.1 Lightning Pulse

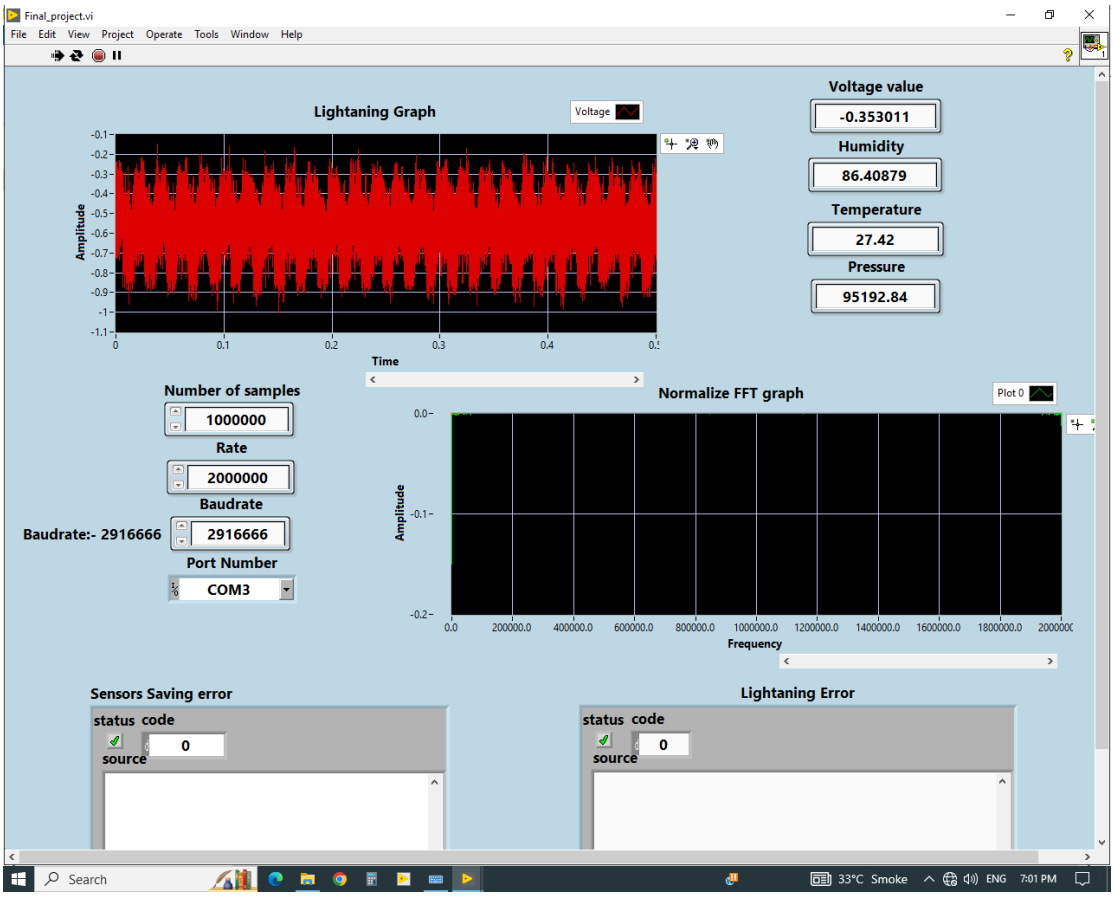

Fig. 6.2 STM data acquisition

CHAPTER 7

RESULTS AND DISCUSSION

7.1 Results and Discussion

The above results have revealed that using NI DAQ at 2 MSPS and STM32F407VET6 at System clock of 168 MHZ can still create the mapping problem. The rate of sending data for NI DAQ is way too high than STM32F407VET6 can compete with.

Moreover, It has also been observed that at times of lightning, humidity will cross the 70 % RH mark.

If in future we plan of using STM32 as standalone device for this project then following things should be kept in mind:

- **ADC Problem:** When we are using HAL Coding. If there is a conflict with any channel of the ADC the whole ADC fails to function properly. This issue can only be solved when the Code inserted in the stm32 is pure Embedded C code and not the HAL Code. Moreover, if Sampling cycle is reduced to 3 cycles then sampling will be high but accuracy will be lower. On top of that, the ADC on STM32 is unipolar in nature, but we might require a bipolar ADC for lightning detection. Thus, it would be better to use an external ADC with high sampling rate, high accuracy and with faster communication protocol.

- **Interrupt Problem:** In HAL coding it has been observed that when we are using Timer Interrupt and ADC Interrupt simultaneously then the lower priority interrupt stops responding. This problem can only be resolved only when we use proper Embedded C programming or Assembly programming for the project.

- **FTDI Problem:** FTDI FT232RL Module used in the project for UART to USB conversion has a max Baud rate of 3 Mbps. If baud rate is increased to 10 Mbps then NI DAQ can be replaced with STM32 for Lightning data acquisition.

- **Floating point operations problem:** STM32 takes a very long time for floating point operations, thus sending raw data through STM32 and then doing data processing at the end node would help us to achieve the 2 MSPS mark on stm32 alone.

- **Other Considerations:** Using RTOS on stm32 would have helped to perform some sort of parallel processing and thus would have increased the rate of sending data. Using Sensors with only SPI communication protocol will increase the rate of taking data from sensors to STM32 board. Thus, increasing the rate of sending data.

Looking at all these shortcomings of embedded systems. It will be wiser to use the FPGA Board specifically for Lightning(ADC) and communication. Another solution of the problem is to have a multi core Embedded system board, where 1 core will gather sensor data and the other core will read ADC at 2 MSPS and the final core will gather data from all the cores and send it to PC through USB communication.

CHAPTER 8

SUMMARY

8.1 Summary

Generally, Lightning is a microsecond's event, where electric currents conducts from one location to another. Therefore, it becomes crucial to monitor lightning using a proper station that can measure the values of humidity, temperature, pressure and potential difference between the antennas. Therefore, this report centers around the Development of Lightning station which will measure variations in humidity, temperature, pressure, and potential difference at the antenna at minimum sampling frequency of 2 MSPS (million samples per second).

Initially, we have expected that both STM32 and NI DAQ will be sending data at the same rate. But, unfortunately due to lack of experience regarding such real life projects that target was not achieved. Use of FPGA for lightning data acquisition would have been a game changer move. Use of RTOS, SPI based sensors on STM32 would have increased the rate of data acquisition. Use of a better FTDI module would have increased the rate of data transmission through UART.

References

[1] F. Meng, Z. Zhu, and L. Zhou, "Non-contact infrared temperature measuring device based on stm32," *Frontiers in Computing and Intelligent Systems*, vol. 5, pp. 54–57, 11 2023.

[2] W. Xiao-bo, Z. Meng-lian, F. Zhi-gang, and Y. Xiao-lang, "Novel voltage output integrated circuit temperature sensor," *Journal of Zhejinag University: Science*, vol. 3, pp. 553–558, 01 2002.

[3] N. Kashyap and U. C. Pati, "Multi channel data acqusition and data logging system for meteorology application," 05 2015, pp. 220–225.

[4] M. Kolapkar, K. P.W., and S. Sayyad, "Design and development of embedded system for measurement of humidity, soil moisture and temperature in polyhouse using 89e516rd microcontroller," *International Journal of Advanced Agricultural Science and Technology*, vol. 5, pp. 96–110, 10 2016.

[5] J. Islam, U. Habiba, H. Kabir, K. Martuza, F. Akter, F. Hafiz, M. Abu, S. Haque, M. Hoq, M. Abdul, and A. N. Chowdhury, "Design and development of microcontroller based wireless humidity monitor," 11 2021.

[6] M. Simic, "Microcontroller based system for measuring and data acquisition of air relative humidity and temperature," 09 2013.

[7] deepblueembedded.com, "Stm32 lm35 temperature sensor example – lm35 with stm32 adc," *https://deepbluembedded.com/stm32-lm35-temperature-sensor-example-lm35-with-stm32-adc/*, 2015.

[8] controllerstech.com, "Ds18b20 and stm32," *https://controllerstech.com/ds18b20-and-stm32/*, 2016.

[9] how2electronics.com, "Humidity/temperature monitor with dht11 stm32 microcontroller," *https://how2electronics.com/humidity-temperature-monitor-dht11-stm32-microcontroller/*, 2014.

[10] labprojectsbd.com, "How to interface sht31 with stm32," *https://labprojectsbd.com/2023/03/21/how-to-interface-sht31-with-stm32/*, 2019.

[11] M. Choma, "Infrared temperature sensor - mlx90614," *https://www.instructables.com/Infrared-Temperature-Sensor-MLX90614/*, 2018.

[12] electronicwings.com, "Thermocouple sensor interfacing with esp32," *https://www.instructables.com/Infrared-Temperature-Sensor-MLX90614/*, 2018.

[13] controllerstech.com, "Interface bmp180 with stm32," *https://controllerstech.com/interface-bmp180-with-stm32/*, 2017.

[14] pressuresensorfinder.com, "Piezoresistive pressure sensors vs. capacitive pressure sensors – advantages disadvantages," *https://www.pressuresensorfinder.com/piezoresistive-pressure-sensors-vs-capacitive-pressure*, 2018.

[15] *Reference manual*, ST Microelectronics, 2 2024, rev. 20.

[16] *Arduino UNO IDE datasheet*, Arduino, 6 2021.

[17] *Digital-output relative humidity temperature sensor/module*, Aosong Electronics, 4 2010.

[18] *A Basic Guide to RTD Measurements*, Texas Instruments, 3 2023, sBAA275.

[19] *MAX31865 RTD-to-Digital Converter*, Analog Devices Ltd, 7 2015.

[20] *Datasheet SHT3x-DIS*, Sensirion, 12 2019.

[21] *BMP280 Digital Pressure sensor*, Bosch Sensortec Ltd, 5 2015.

[22] *FT232R USB UART I.C*, Future Technology Devices International Ltd, 1 2006, rev. 1.04.

[23] Roboro, "Measuring temperature from pt100 using arduino," *https://www.instructables.com/Reading-Temperature-From-PT100-Using-Arduino/*, 2015.

Appendix A

Arduino UNO Code

```
#include <Wire.h>
#include "DHT.h"
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#define DHTTYPE DHT22
Adafruit_SSD1306 display = Adafruit_SSD1306(128, 64, &Wire, -1);
unsigned long delayTime;
uint8_t DHTPin = 2;
DHT dht(DHTPin, DHTTYPE);
float Temperature;
float Humidity;
float Temp_Fahrenheit;
#include <SPI.h>
#include <Adafruit_BMP280.h>
#define BMP280_ADDRESS 0x76
Adafruit_BMP280 bmp; // I2C
void setup() {
  Serial.begin(9600);
  pinMode(DHTPin, INPUT);
  dht.begin();
  Serial.begin(9600);
  while ( !Serial ) delay(100);   // wait for native usb
  unsigned status;
  status = bmp.begin(BMP280_ADDRESS);
  if (!status) {
    Serial.println(F("Could not find a valid BMP280 sensor"));
    Serial.print("SensorID was: 0x");
    Serial.println(bmp.sensorID(),16);
    Serial.print("ID of 0xFF probably means a bad address\n");
    Serial.print("ID of 0x56-0x58 represents a BMP 280,\n");
```

```
    Serial.print("ID of 0x60 represents a BME 280.\n");
    Serial.print("ID of 0x61 represents a BME 680.\n");
    while (1) delay(10);
}


bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,
                Adafruit_BMP280::SAMPLING_X2,
                Adafruit_BMP280::SAMPLING_X16,
                Adafruit_BMP280::FILTER_X16,
                Adafruit_BMP280::STANDBY_MS_500);
}


void loop() {
  // put your main code here, to run repeatedly:


  Humidity = dht.readHumidity();
  // Read temperature as Celsius (the default)
  Temperature = dht.readTemperature();

  // Check if any reads failed and exit early (to try again).
  if (isnan(Humidity) || isnan(Temperature) ||
  isnan(Temp_Fahrenheit)) {
    Serial.println(F("Failed to read from DHT sensor!"));
    return;
  }
  Serial.println("DHT_11");
  Serial.print(F("Humidity: "));
  Serial.print(Humidity);
  Serial.print("%  ");
  Serial.print(F("Temperature: "));
  Serial.print(Temperature);
  Serial.print(F("°C "));
  Serial.println();
  Serial.println();
  Serial.println("BMP_280");
  Serial.print(F("Temperature = "));
```

```
  Serial.print(bmp.readTemperature());
  Serial.println(" °C");
  Serial.print(F("Pressure = "));
  Serial.print(bmp.readPressure());
  Serial.println(" Pa");
  Serial.print(F("Approx altitude = "));
  Serial.println(" m");
  Serial.println();
    }
}
```

Appendix B

STM32 Code

```
#include "main.h"
#include "stm32f4xx_hal.h"
#include "stdio.h"
#include "stdbool.h"
#include "math.h"
#include "string.h"

#define MAX31856_CONFIG_REG             0x00
#define MAX31856_CONFIG_BIAS            0x80
#define MAX31856_CONFIG_MODEAUTO        0x40
#define MAX31856_CONFIG_MODEOFF         0x00
#define MAX31856_CONFIG_1SHOT           0x20
#define MAX31856_CONFIG_3WIRE           0x10
#define MAX31856_CONFIG_24WIRE          0x00
#define MAX31856_CONFIG_FAULTSTAT       0x02
#define MAX31856_CONFIG_FILT50HZ        0x01
#define MAX31856_CONFIG_FILT60HZ        0x00

#define MAX31856_RTDMSB_REG             0x01
#define MAX31856_RTDLSB_REG             0x02
#define MAX31856_HFAULTMSB_REG          0x03
#define MAX31856_HFAULTLSB_REG          0x04
#define MAX31856_LFAULTMSB_REG          0x05
#define MAX31856_LFAULTLSB_REG          0x06
#define MAX31856_FAULTSTAT_REG          0x07


#define MAX31865_FAULT_HIGHTHRESH       0x80
#define MAX31865_FAULT_LOWTHRESH        0x40
#define MAX31865_FAULT_REFINLOW         0x20
```

```
#define MAX31865_FAULT_REFINHIGH        0x10
#define MAX31865_FAULT_RTDINLOW         0x08
#define MAX31865_FAULT_OVUV             0x04

#define _MAX31865_RREF      430.0f
#define _MAX31865_RNOMINAL  100.0f

float readTemp = 0.0f;

#define RTD_A 3.9083e-3
#define RTD_B -5.775e-7


/* Private variables ---------------------------------------------
ADC_HandleTypeDef hadc1;

I2C_HandleTypeDef hi2c1;
I2C_HandleTypeDef hi2c3;

SPI_HandleTypeDef hspi1;

UART_HandleTypeDef huart4;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_I2C1_Init(void);
static void MX_I2C3_Init(void);
static void MX_SPI1_Init(void);
static void MX_UART4_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */
```

```
/* Private user code ---------------------------------------------
/* USER CODE BEGIN 0 */
#define BMP280_ADDR 0x76 << 1 // SDO pin is grounded

void BMP280_Init(void);
void BMP280_Read_Calibration(void);
void BMP280_Read_Data(void);
void BMP280_Compensate_Temperature(int32_t adc_T);
float BMP280_Compensate_Pressure(int32_t adc_P);

uint32_t adc1_value = 0;

          /* Calculate the equivalent voltage values */
float voltage1 = 0.0f;

float BMP_pressure = 0.0f;
uint8_t check_bit = 0;

bool isOk;
uint8_t data_buffer[16];
char char_buffer[30];
char char_buffer_ADC[13];

uint16_t dig_T1;
int16_t dig_T2, dig_T3;
uint16_t dig_P1;
int16_t dig_P2, dig_P3, dig_P4, dig_P5, dig_P6, dig_P7, dig_P8
, dig_P9;
int32_t t_fine;


void ConvertFloatsToString(float var2, int var3, float var4
, char *buffer, size_t bufferSize) {
    // Create a temporary buffer to hold the formatted string
    snprintf(buffer, bufferSize, ",%.2f,%d,%.2f\n"
    , var2, var3, var4);
}
```

```
void BMP280_Init(void) // It initializes the BMP280 sensor
{
    uint8_t config[2];

    config[0] = 0xF4; // Control measurement register
    //config[1] = 0x27; // Normal mode, temp and pressure
    oversampling rate = 1
    config[1] = 0x57; // Normal mode, Ultra high resolution
    , temp pressure oversampling rate = 2,16
    HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, config, 2
    , HAL_MAX_DELAY);

    config[0] = 0xF5; // Config register
    //config[1] = 0xA0; // Standby time 1000ms, filter off
    config[1] = 0x00; // Standby time 0.5ms, filter off
    HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, config, 2
    , HAL_MAX_DELAY);
}

void BMP280_Read_Calibration(void) // This function helps to store
the coefficient values that will later be used for
data conversion to human readable form
{
    uint8_t calib[24];
    uint8_t reg = 0x88;
    HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, &reg, 1
    , HAL_MAX_DELAY);
    HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, calib, 24
    , HAL_MAX_DELAY);

    dig_T1 = (calib[1] << 8) | calib[0];
    dig_T2 = (calib[3] << 8) | calib[2];
    dig_T3 = (calib[5] << 8) | calib[4];

    dig_P1 = (calib[7] << 8) | calib[6];
    dig_P2 = (calib[9] << 8) | calib[8];
    dig_P3 = (calib[11] << 8) | calib[10];
    dig_P4 = (calib[13] << 8) | calib[12];
    dig_P5 = (calib[15] << 8) | calib[14];
```

```
    dig_P6 = (calib[17] << 8) | calib[16];
    dig_P7 = (calib[19] << 8) | calib[18];
    dig_P8 = (calib[21] << 8) | calib[20];
    dig_P9 = (calib[23] << 8) | calib[22];
}

void BMP280_Read_Data(void) // Reads data from BMP
{
    uint8_t reg = 0xF7; // Data registers
    uint8_t data[6];
    HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, &reg, 1
    , HAL_MAX_DELAY);
    HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, data, 6
    , HAL_MAX_DELAY);

    int32_t adc_P = (data[0] << 12) | (data[1] << 4)
    | (data[2] >> 4);
    int32_t adc_T = (data[3] << 12) | (data[4] << 4)
    | (data[5] >> 4);

    BMP280_Compensate_Temperature(adc_T);
    BMP_pressure = BMP280_Compensate_Pressure(adc_P);
}

void BMP280_Compensate_Temperature(int32_t adc_T)
{
    int32_t var1, var2;
    var1 = ((((adc_T >> 3) - ((int32_t)dig_T1 << 1)))
    * ((int32_t)dig_T2)) >> 11;
    var2 = (((((adc_T >> 4) - ((int32_t)dig_T1)) * ((adc_T >> 4)
    - ((int32_t)dig_T1))) >> 12) * ((int32_t)dig_T3)) >> 14;
    t_fine = var1 + var2;
}


float BMP280_Compensate_Pressure(int32_t adc_P)
{
    int64_t var1, var2, p;
    var1 = ((int64_t)t_fine) - 128000;
```

```
    var2 = var1 * var1 * (int64_t)dig_P6;
    var2 = var2 + ((var1 * (int64_t)dig_P5) << 17);
    var2 = var2 + (((int64_t)dig_P4) << 35);
    var1 = ((var1 * var1 * (int64_t)dig_P3) >> 8) + ((var1
    * (int64_t)dig_P2) << 12);
    var1 = (((((int64_t)1) << 47) + var1)
    * ((int64_t)dig_P1)) >> 33;
    if (var1 == 0)
    {
        return 0; // avoid exception caused by division by zero
    }
    p = 1048576 - adc_P;
    p = (((p << 31) - var2) * 3125) / var1;
    var1 = (((int64_t)dig_P9) * (p >> 13) * (p >> 13)) >> 25;
    var2 = (((int64_t)dig_P8) * p) >> 19;
    p = ((p + var1 + var2) >> 8) + (((int64_t)dig_P7) << 4);
    return (float)p / 256.0f; // Convert from int to float
}

void Max31865_readRegisterN(uint8_t addr, uint8_t *buffer
, uint8_t n)
{
  //uint8_t tmp = 0xFF;
 addr &= 0x7F; // Ensure the MSB is 0 for reading
 HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1
 , GPIO_PIN_RESET); // Select the MAX31865
 HAL_SPI_Transmit(&hspi1,&addr, 1, 100); // Send the
 address
 while (n--)
 {
    //HAL_SPI_TransmitReceive(&hspi1, &tmp, buffer, 1
    , 100); // Read n bytes
    HAL_SPI_Receive(&hspi1, buffer, 1, 100); // Read n bytes
  buffer++; // Incrementing the pointer addresses
 }
 HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_SET)
 ; // Deselect the MAX31865
}
```

```
uint8_t Max31865_readRegister8(uint8_t addr)
{
  uint8_t ret = 0;
 Max31865_readRegisterN( addr, &ret, 1); // Read a single byte
 return ret;
}

uint16_t Max31865_readRegister16(uint8_t addr)
{
 uint8_t buffer[2] = {0, 0};
 Max31865_readRegisterN(addr, buffer, 2); // Read two bytes
 uint16_t ret = buffer[0];
 ret <<= 8;
 ret |=  buffer[1];
 return ret;
}

void Max31865_writeRegister8(uint8_t addr, uint8_t data)
{
 HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_RESET)
 ; // Select the MAX31865
  addr |= 0x80; // Ensure the MSB is 1 for writing
 HAL_SPI_Transmit(&hspi1,&addr, 1, 100); // Send the address
 HAL_SPI_Transmit(&hspi1,&data, 1, 100); // Send the data
 HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_SET)
 ; // Deselect the MAX31865
}

uint8_t Max31865_readFault()
{
  return Max31865_readRegister8(MAX31856_FAULTSTAT_REG);
}



void Max31865_clearFault()
{ // Clears all the previous faults
 uint8_t t = Max31865_readRegister8( MAX31856_CONFIG_REG)
 ;// Read the config register
 t &= ~0x2C;  // Clear fault bits
```

```
 t |= MAX31856_CONFIG_FAULTSTAT; // Set fault status bit
 Max31865_writeRegister8(MAX31856_CONFIG_REG, t)
 ; // Write back to config register
}

void Max31865_enableBias(uint8_t enable)
{ // Enables the VBIAS
 uint8_t t = Max31865_readRegister8( MAX31856_CONFIG_REG)
 ; // Read the config register
 if (enable)
  t |= MAX31856_CONFIG_BIAS; // Enable bias
 else
  t &= ~MAX31856_CONFIG_BIAS;  // Disable bias
 Max31865_writeRegister8( MAX31856_CONFIG_REG, t)
 ; // Write back to config register
}

uint16_t Max31865_readRTD ()
{
 Max31865_clearFault(); // Clear any existing faults
 Max31865_enableBias(1); // Enable bias
 HAL_Delay(10); // Wait for bias to stabilize
 uint8_t t = Max31865_readRegister8(MAX31856_CONFIG_REG)
 ; // Read config register
 t |= MAX31856_CONFIG_1SHOT; // Set one-shot mode
 Max31865_writeRegister8( MAX31856_CONFIG_REG, t)
 ; // Write back to config register
 HAL_Delay(65); // Wait for conversion to complete
 uint16_t rtd = Max31865_readRegister16( MAX31856_RTDMSB_REG)
 ; // Read RTD value
 rtd >>= 1
 ; // Discard the least significant bit because it is fault bit
 return rtd;
}

void Max31865_setWires(uint8_t numWires)
{
 uint8_t t = Max31865_readRegister8( MAX31856_CONFIG_REG);
 // Read config register
```

```
 if (numWires == 3)
  t |= MAX31856_CONFIG_3WIRE;
 else
  t &= ~MAX31856_CONFIG_3WIRE; // Set 2-wire
 Max31865_writeRegister8(MAX31856_CONFIG_REG, t);
 // Write back to config register
}

void Max31865_setFilter( uint8_t filterHz)
{
 uint8_t t = Max31865_readRegister8( MAX31856_CONFIG_REG);
 // Read config register
 if (filterHz == 50)
  t |= MAX31856_CONFIG_FILT50HZ;
 else
  t &= ~MAX31856_CONFIG_FILT50HZ;
 Max31865_writeRegister8( MAX31856_CONFIG_REG, t);
 // Write back to config register
}

void Max31865_autoConvert( uint8_t enable)
{
 uint8_t t = Max31865_readRegister8( MAX31856_CONFIG_REG);
 // Read config register
 if (enable)
  t |= MAX31856_CONFIG_MODEAUTO;  // Enable auto conversion
 else
  t &= ~MAX31856_CONFIG_MODEAUTO; // Disable auto conversion
 Max31865_writeRegister8( MAX31856_CONFIG_REG, t);
 // Write back to config register
}

void  Max31865_init(SPI_HandleTypeDef *spi
, GPIO_TypeDef  *cs_gpio , uint16_t cs_pin , uint8_t  numwires)
{

  HAL_Delay(1); // Small delay
  HAL_GPIO_WritePin(GPIOE , GPIO_PIN_1,GPIO_PIN_SET);
  // Deselect MAX31865
```

```
  HAL_Delay(1); // Small delay
  Max31865_setWires( numwires); // Set wire configuration
  Max31865_enableBias(0); // Disable bias
  Max31865_autoConvert(0); // Disable auto conversion
  Max31865_clearFault(); // Clear any faults
  //Max31865_setFilter( filterHz);
}

void Max31865_readTempC(float *readTemp)
{
  isOk = false;
  float Z1, Z2, Z3, Z4, Rt, temp;
 Rt = Max31865_readRTD(); // Read RTD value
 Rt /= 32768;
 Rt *= _MAX31865_RREF; // Calculate resistance
 Z1 = -RTD_A;
 Z2 = RTD_A * RTD_A - (4 * RTD_B);
 Z3 = (4 * RTD_B) / _MAX31865_RNOMINAL;
 Z4 = 2 * RTD_B;
 temp = Z2 + (Z3 * Rt);
 temp = (sqrtf(temp) + Z1) / Z4; // Calculate temperature

 if (temp >= 0)
  {
    *readTemp = temp;
    if(Max31865_readFault() == 0)
      isOk = true;

  }
 Rt /= _MAX31865_RNOMINAL;
 Rt *= 100;
 float rpoly = Rt;
 temp = -242.02;
 temp += 2.2228 * rpoly;
 rpoly *= Rt;  // square
 temp += 2.5859e-3 * rpoly;
 rpoly *= Rt;  // ^3
 temp -= 4.8260e-6 * rpoly;
 rpoly *= Rt;  // ^4
```

```
 temp -= 2.8183e-8 * rpoly;
 rpoly *= Rt;  // ^5
 temp += 1.5243e-10 * rpoly;

  *readTemp = temp;
  if(Max31865_readFault() == 0)
    isOk = true;

}

bool is_spi_busy(SPI_HandleTypeDef *hspi)
{
    // Check if the SPI busy flag is set
    if(__HAL_SPI_GET_FLAG(hspi, SPI_FLAG_BSY))
    {
        // SPI is busy
        return true;
    }
    else
    {
        // SPI is not busy
        return false;
    }
}

/* Private user code ---------------------------------------------
/* USER CODE BEGIN 0 */

uint8_t buffer[6];
uint16_t humidity_raw;
float humidity;


uint16_t ADC_buffer;
uint32_t ADC_counter = 0;

void humidity_init(){
 while(HAL_I2C_GetState(&hi2c3) != HAL_I2C_STATE_READY);
```

```
    // Send measurement command (0x2C, 0x06 for high repeatability
    , clock stretching enabled)
    uint8_t command[2] = {0x2C, 0x06};
    if (HAL_I2C_Master_Transmit(&hi2c3, 0x44 << 1, command, 2
    , 1000) != HAL_OK)
    {
        Error_Handler();
    }
}

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_ADC1_Init();
  MX_I2C1_Init();
  MX_I2C3_Init();
  MX_SPI1_Init();
  MX_UART4_Init();

  BMP280_Init();
  BMP280_Read_Calibration();
  Max31865_init(&hspi1 , GPIOE,GPIO_PIN_1, 2);
  //HAL_ADC_Start(&hadc1);
  humidity_init();


  while (1)
  {

   BMP280_Read_Data();
```

```
   while(is_spi_busy(&hspi1));
   Max31865_readTempC(&readTemp);


      // Wait for measurement to complete (minimum 15ms)
      //HAL_Delay(20);
      while(HAL_I2C_GetState(&hi2c3) != HAL_I2C_STATE_READY);

      // Read 6 bytes of data
      if (HAL_I2C_Master_Receive(&hi2c3, 0x44 << 1, buffer, 6
      , 1000) != HAL_OK)
      {
          Error_Handler();
      }


      humidity_raw = (buffer[3] << 8) | buffer[4];


      ConvertFloatsToString(readTemp, humidity_raw, BMP_pressure
      , char_buffer, sizeof(char_buffer));

      // Transmit the buffer content through UART
      HAL_UART_Transmit(&huart4, (uint8_t*)char_buffer
      , strlen(char_buffer), HAL_MAX_DELAY);

      check_bit^=0x01;

  }
  /* USER CODE END 3 */
}
/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
```

```
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

/** Configure the main internal regulator output voltage
*/
__HAL_RCC_PWR_CLK_ENABLE();
__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

/** Initializes the RCC Oscillators according
to the specified parameters
* in the RCC_OscInitTypeDef structure.
*/
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue
= RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 8;
RCC_OscInitStruct.PLL.PLLN = 140;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
RCC_OscInitStruct.PLL.PLLQ = 4;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
  Error_Handler();
}

/** Initializes the CPU, AHB and APB buses clocks
*/
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK
|RCC_CLOCKTYPE_SYSCLK  |RCC_CLOCKTYPE_PCLK1
|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct
, FLASH_LATENCY_4) != HAL_OK)
{
```

```
    Error_Handler();
  }
}

/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

  /** Configure the global features of the ADC (Clock,
  Resolution, Data Alignment and number of conversion)
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
```

```
    Error_Handler();
  }

  /** Configure for the selected ADC regular channel its
  corresponding rank in the sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_10;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief I2C1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_I2C1_Init(void)
{

  /* USER CODE BEGIN I2C1_Init 0 */

  /* USER CODE END I2C1_Init 0 */

  /* USER CODE BEGIN I2C1_Init 1 */

  /* USER CODE END I2C1_Init 1 */
  hi2c1.Instance = I2C1;
  hi2c1.Init.ClockSpeed = 100000;
  hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
  hi2c1.Init.OwnAddress1 = 0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
```

```
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(&hi2c1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN I2C1_Init 2 */

  /* USER CODE END I2C1_Init 2 */

}

/**
  * @brief I2C3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_I2C3_Init(void)
{

  /* USER CODE BEGIN I2C3_Init 0 */

  /* USER CODE END I2C3_Init 0 */

  /* USER CODE BEGIN I2C3_Init 1 */

  /* USER CODE END I2C3_Init 1 */
  hi2c3.Instance = I2C3;
  hi2c3.Init.ClockSpeed = 100000;
  hi2c3.Init.DutyCycle = I2C_DUTYCYCLE_2;
  hi2c3.Init.OwnAddress1 = 0;
  hi2c3.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c3.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c3.Init.OwnAddress2 = 0;
  hi2c3.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c3.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(&hi2c3) != HAL_OK)
```

```
  {
    Error_Handler();
  }
  /* USER CODE BEGIN I2C3_Init 2 */

  /* USER CODE END I2C3_Init 2 */

}

/**
  * @brief SPI1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_SPI1_Init(void)
{

  /* USER CODE BEGIN SPI1_Init 0 */

  /* USER CODE END SPI1_Init 0 */

  /* USER CODE BEGIN SPI1_Init 1 */

  /* USER CODE END SPI1_Init 1 */
  /* SPI1 parameter configuration*/
  hspi1.Instance = SPI1;
  hspi1.Init.Mode = SPI_MODE_MASTER;
  hspi1.Init.Direction = SPI_DIRECTION_2LINES;
  hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
  hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
  hspi1.Init.CLKPhase = SPI_PHASE_2EDGE;
  hspi1.Init.NSS = SPI_NSS_SOFT;
  hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_32;
  hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
  hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
  hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
  hspi1.Init.CRCPolynomial = 10;
  if (HAL_SPI_Init(&hspi1) != HAL_OK)
  {
```

```
    Error_Handler();
  }
  /* USER CODE BEGIN SPI1_Init 2 */

  /* USER CODE END SPI1_Init 2 */

}

/**
  * @brief UART4 Initialization Function
  * @param None
  * @retval None
  */
static void MX_UART4_Init(void)
{

  /* USER CODE BEGIN UART4_Init 0 */

  /* USER CODE END UART4_Init 0 */

  /* USER CODE BEGIN UART4_Init 1 */

  /* USER CODE END UART4_Init 1 */
  huart4.Instance = UART4;
  huart4.Init.BaudRate = 2916666;
  huart4.Init.WordLength = UART_WORDLENGTH_8B;
  huart4.Init.StopBits = UART_STOPBITS_1;
  huart4.Init.Parity = UART_PARITY_NONE;
  huart4.Init.Mode = UART_MODE_TX;
  huart4.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart4.Init.OverSampling = UART_OVERSAMPLING_8;
  if (HAL_UART_Init(&huart4) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN UART4_Init 2 */

  /* USER CODE END UART4_Init 2 */
```

```
}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  __HAL_RCC_GPIOE_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_RESET);

  /*Configure GPIO pin : PE1 */
  GPIO_InitStruct.Pin = GPIO_PIN_1;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
```

```
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the
  HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source
  line number
  * where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file
  name and line number,
     ex: printf("Wrong parameters value:
     file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```