ADC IRQ - Method for Wireless Data Acquisition

A Project Report

Presented to

the Department of Electrical and Electronic Engineering

University of Rajshahi

In Partial Fulfillment of the

Requirements for the Degree of

B.Sc. in Electrical and Electronic Engineering

Submitted by

MD. AL-AMIN KABIR

Student ID: 1810674142

Registration No: 1810674142

B.Sc. Engineering, Session: 2017-18

Dedicated to my Parents

and

My Honorable Supervisor

Professor Dr. Md. Ariful Islam Nahid

Acknowledgments

I am very much delighted to express my heartfelt gratitude to my Project supervisor, Professor Dr. Md. Ariful Islam Nahid, Department of Electrical and Electronic Engineering, University of Rajshahi, Bangladesh. For his continuous guidance, valuable discussions and constructive suggestions in various phases of the project work. My sincere thanks are also due to all of my respected teachers in the Department of Electrical and Electronic Engineering for encouraging me during the period of the project work. I convey my gratefulness to all of my departmental staff for their kind-hearted support that inspired me to complete my project work.

I am very much glad and happy to express my profound gratitude to Almighty ALLAH for the blessings without which the work would not be possible to complete successfully.

Finally, I am taking an opportunity to convey my indebtedness and gratefulness to my parents whose inspiration, financial and psychological support have empowered me to concentrate my mind on full project work.

Date: 27th October, 2024

The author

Candidate's Declaration

I do hereby declare that the project work incorporated in this report entitled **ADC IRQ - Method for Wireless Data Acquisition** has been performed by me. To the best of my knowledge, the project report has not been submitted previously in any University or Institution for any other degree.

…….

Md. Al-Amin Kabir

Student ID: 1810674142

Registration No: 1810674142

B.Sc. Engineering Session: 2017-18

Department of Electrical and Electronic Engineering

Certificate of the Supervisor

Certified that the project work incorporated in this report entitled **ADC IRQ - Method for Wireless Data Acquisition** is the original work carried out by **Md. Al-Amin Kabir** under my supervision that has fulfilled partially the requirements for the Degree of B.Sc. in Electrical and Electronic Engineering and has been accepted as satisfactory.

…….

(**Dr. Md. Ariful Islam Nahid**)

Professor

Department of Electrical and Electronic Engineering

University of Rajshahi, Rajshahi, Bangladesh

ABSTRACT

The demand for efficient analog data acquisition and wireless data transmission is rising in industrial and research environments, where real-time monitoring is essential for performance optimization and safety. This project explores the implementation of an advanced data acquisition system using the STM32 microcontroller's Analog-to-Digital Converter (ADC) with Interrupt Request (IRQ) functionality, designed to efficiently capture analog data with minimal CPU intervention.

The core of the project involves the STM32 microcontroller for accurate data acquisition through the ADC, utilizing the IRQ method for real-time temperature monitoring. The temperature data is gathered from an LM35 sensor and displayed on an OLED screen via I2C communication. Furthermore, this data is transmitted wirelessly to a remote system using an ESP32 module via UART communication, ensuring real-time remote monitoring.

By integrating the STM32's ADC with IRQ and ESP32's wireless communication capability, this project demonstrates a scalable, low-power system suitable for real-time applications in industrial and IoT environments. This approach offers a reliable and efficient solution for analog data acquisition, significantly reducing CPU load and improving data transmission speed.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Introduction

In the world of embedded systems, understanding the ARM [1] architecture, especially through practical applications on STM32 [2] microcontrollers, is invaluable for developing robust, efficient systems. ARM-based STM32 microcontrollers are widely used in industrial, research, and IoT applications due to their processing power, low energy consumption, and versatile interfacing capabilities. By working with the STM32 microcontroller in this project, aimed to deepen knowledge of ARM architecture, focusing on register-level programming and understanding the internal workings of an embedded system.

This project centers on designing a real-time temperature monitoring system that leverages the capabilities of the STM32, paired with an LM35 temperature sensor, an OLED display, and ESP32 [3] for wireless data transmission [4]. The STM32's ADC [5] (Analog-to-Digital Converter) with interrupt (IRQ) [6] functionality is used for precise, timely data acquisition, which is essential in scenarios requiring reliable real-time monitoring.

Data visualization is achieved locally via an OLED display using the I2C [7] protocol, while the STM32 uses UART [8] to transmit data to the ESP32 module for wireless data relay to a remote server or user interface. Integrating these components and communication protocols illustrates the capabilities of ARM-based microcontrollers in practical applications, while providing a scalable framework suitable for larger IoT-based monitoring systems. This project serves as both a learning exercise in ARM architecture and as a foundational model for environmental monitoring, adaptable to industries, research labs, and smart city initiatives.

1.2 Motivation

The motivation behind this project was to deepen my understanding of ARM architecture microcontrollers and gain practical experience with register-level programming. As embedded systems evolve, the need for efficient and low-level control of hardware components has become essential, especially in applications that demand high performance and precision. ARM-based microcontrollers, like the STM32 series, offer a powerful platform for exploring these concepts due to their versatility, widespread use in industry, and efficient architecture.

The integration of wireless communication with embedded systems presents a promising solution to these challenges. Real-time environmental monitoring not only helps in maintaining optimal conditions but also provides crucial insights for predictive maintenance and anomaly detection. By utilizing low-cost, efficient components such as the STM32 microcontroller, LM35 temperature sensor, and ESP32 wireless module, it is possible to create a scalable and reliable monitoring system.

This project aims to address the need for an efficient and wireless solution by implementing ADC IRQ for real-time data acquisition and I2C and UART communication protocols for seamless data transmission. The combination of these technologies will enable remote data monitoring, ensuring that data can be accessed and analyzed in real-time, improving overall system efficiency and effectiveness.

1.3 Objective

The primary objective of this project is to explore embedded systems using STM32 microcontrollers and register-level programming by developing a real-time temperature monitoring system. The system is designed to accurately sense and wirelessly transmit temperature data for remote access and analysis. By leveraging the STM32 microcontroller and an LM35 temperature sensor, the project aims to deliver reliable and precise measurements, which are then transmitted via the ESP32's Wi-Fi module to a remote monitoring platform.

This project focuses on implementing efficient data acquisition through ADC interrupt request (IRQ), ensuring real-time response to temperature changes without waiting for other task completion. Additionally, it leverages I2C communication to interface with sensors and UART communication for seamless data transmission between components, allowing for robust and scalable monitoring in various industrial and environmental settings. Ultimately, the objective is to create a low-cost, energy-efficient solution that enhances decision-making through continuous and remote environmental data monitoring.

Chapter 2

System Description

2.1 ADC IRQ

An Analog-to-Digital Converter (ADC) is a crucial component in microcontrollers used to convert analog signals into digital values. Many sensors, such as temperature sensors, provide output in the form of analog voltages. The ADC reads this analog signal and converts it into a corresponding digital value that the microcontroller can process. For example, an ADC with 10-bit resolution can represent an analog input voltage in 1024 distinct digital levels, allowing the microcontroller to interpret varying sensor outputs accurately. In STM32 microcontrollers, ADCs often support multiple channels, allowing the conversion of signals from various analog sources.

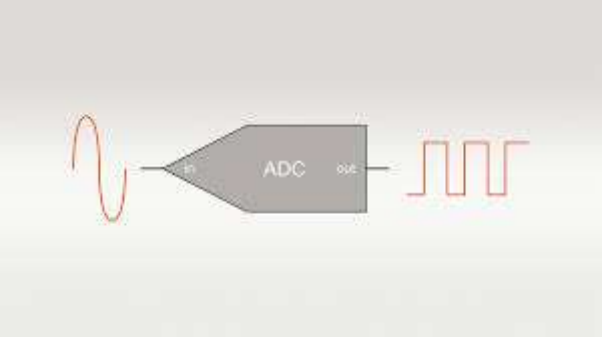


Fig 2.1 Analog to Digital Signal

An Interrupt Request (IRQ) is a signal that pauses the normal execution of code in a microcontroller to handle an urgent task. It allows the microcontroller to respond to specific events, such as a timer overflow or the arrival of new data. The microcontroller halts its current task and executes an interrupt service routine (ISR) to handle the event. Once the ISR is completed, the microcontroller resumes its normal operation. IRQs enhance system responsiveness and efficiency, as they eliminate the need for continuous polling by the CPU, allowing it to perform other tasks in the meantime.

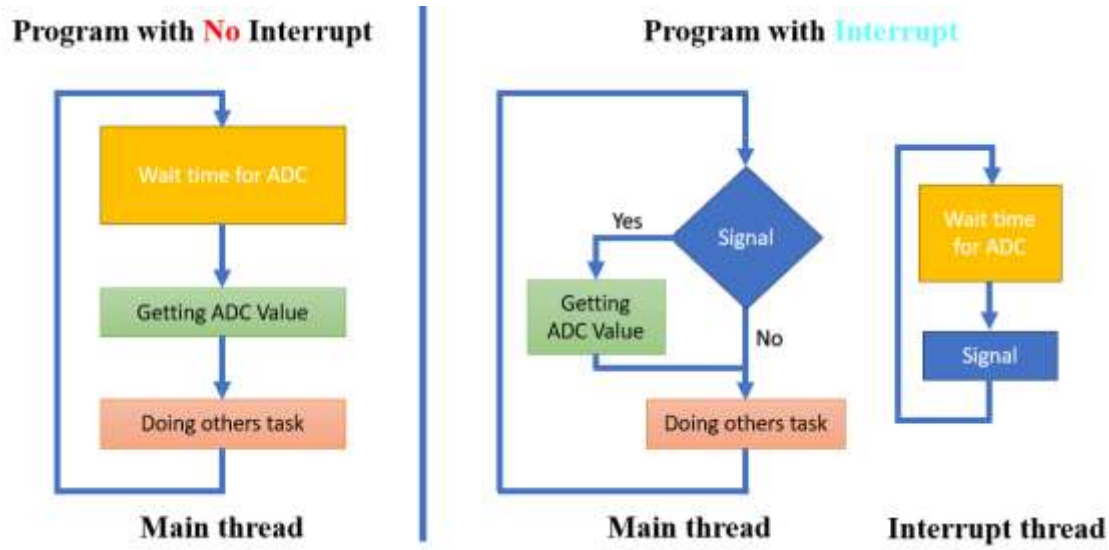


Fig 2.2 STM32F1 ADC Interrupt

In STM32 microcontrollers, the ADC can operate in conjunction with an IRQ to perform more efficient data acquisition. When an ADC conversion is completed, instead of polling the ADC status, an interrupt is triggered, notifying the microcontroller that the conversion is done and the result is ready to be processed. This allows the STM32 to handle multiple tasks simultaneously without wasting CPU cycles waiting for the ADC to complete its conversion. For instance, in a temperature monitoring system, the ADC IRQ can trigger an interrupt after converting the analog signal from the sensor, allowing the microcontroller to immediately handle and store the temperature data or display it on an OLED screen, while also managing other operations such as communication with an ESP32 for wireless data transmission. This setup improves system performance and responsiveness, particularly in real-time applications.

2.2 I2C Communication

I2C (Inter-Integrated Circuit) is a communication protocol used for connecting low-speed peripherals to a microcontroller. It uses two lines: **SCL** (Serial Clock Line) and **SDA** (Serial Data Line), allowing multiple devices to be connected via a shared bus. Each device on the bus has a unique address, enabling the master (typically the

microcontroller) to communicate with various peripherals like sensors, displays, and more using just two wires.

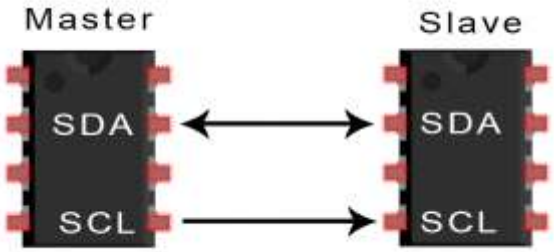


Fig 2.3 I2C Communication

In your project, I2C is used for communication between the STM32F103C6 microcontroller and the OLED display. Here, the STM32 serves as the master, sending and receiving data to and from the OLED (slave) via the SCL and SDA lines.

2.3 UART Communication

UART stands for universal asynchronous receiver / transmitter and defines a protocol, or set of rules, for exchanging serial data between two devices. UART is very simple and only uses two wires between transmitter and receiver to transmit and receive in both directions. Both ends also have a ground connection.

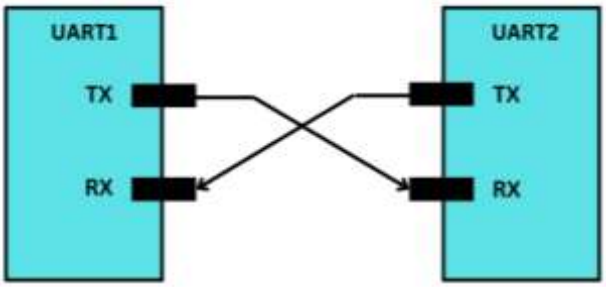


Fig 2.4 UART Communication

In your project, UART communication is likely used between the STM32F103C6 microcontroller and the ESP32 module. The STM32 uses its TX pin to send data to the ESP32's RX pin and vice versa. By using UART, the system ensures a simple, reliable,

point-to-point communication between the STM32 and ESP32 microcontroller, ideal for serial data transmission in embedded systems without the need for complex wiring.

2.4 Wireless Data Acquisition

Wireless Data Transmission in your project involves sending the temperature data from the STM32F103C6 microcontroller to the cloud via the ESP32 module using Wi-Fi. The ESP32 acts as a wireless bridge, enabling the STM32 to communicate with remote servers or cloud platforms.

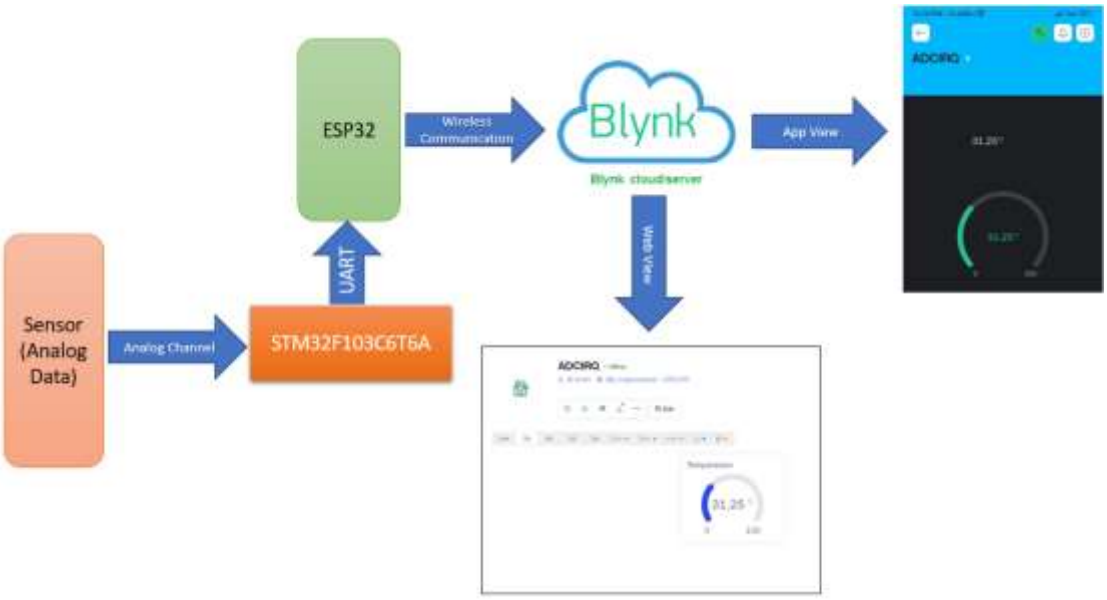


Fig 2.5 Wireless Data Transmission

In this project, the temperature data from the LM35 sensor, processed by the STM32, is sent to the ESP32 through UART communication. The ESP32, connected to a Wi-Fi network, uses wireless transmission to send this data to the Blynk IoT cloud platform and Google Cloude. Blynk allows you to monitor and control devices from anywhere using its cloud infrastructure, which is ideal for IoT applications, beside using Google cloud logging the data on google sheet.

Using the Blynk platform, the ESP32 sends the data to the cloud where it can be visualized in real-time through the Blynk app. This allows for remote monitoring of the temperature readings, providing an easy way to keep track of the sensor data from a smartphone or web interface. Wireless data transmission, therefore, plays a critical role in connecting your embedded system to the cloud, enabling remote data access and management.

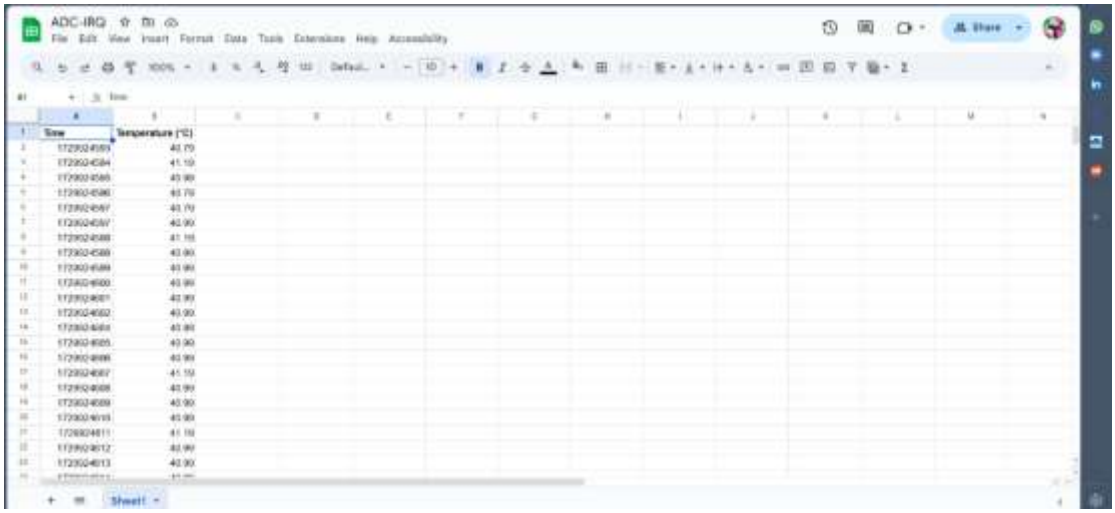


Fig 2.6 Logged Data on Google Sheet

Additionally, the ESP32 is configured to log data to Google Sheets. This is accomplished by setting up the ESP32 to communicate with Google Sheets through a web API, allowing temperature data to be logged and stored over time. This setup provides a reliable record of temperature readings, enabling data analysis and tracking directly in Google Sheets, which is ideal for long-term monitoring. This combination of real-time data with Blynk and historical logging in Google Sheets enhances the data acquisition process, making it versatile and accessible.

2.6 Schematic Diagram

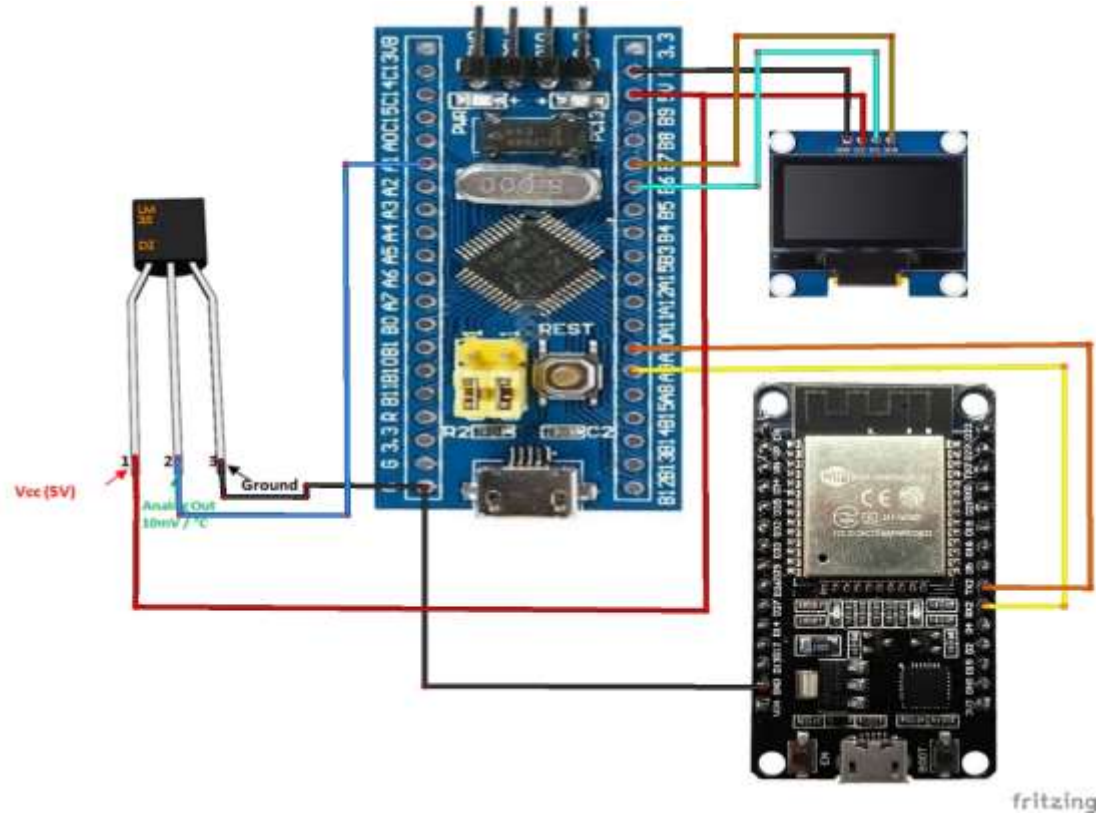


Fig 2.7 Schematic Diagram of Circuit

- ❖ STM 32 to OLED → I2C communication
- ❖ STM 32 to ESP32 → UART communication
- ❖ From LM35 to STM 32 take input as analog signal
- ❖ ESP 32 is connected to internet and sent data on cloud

Chapter3

Methodology

3.1 Overview of the Project

This project focuses on the design and implementation of a real-time temperature monitoring system using the STM32F103C6T6A microcontroller, an LM35 temperature sensor, a 0.96-inch I2C OLED display, and an ESP32 for wireless data transmission. The system utilizes the ADC (Analog-to-Digital Converter) with interrupt request (IRQ) data acquisition, displaying the temperature on the OLED display for local monitoring system and transmitting the data wirelessly to an ESP32, which sends it to a server or other client devices for remote monitoring system.

The core functionality of this project revolves around:

- **Temperature Sensing:** The LM35 sensor is connected to the STM32's ADC pin. The analog voltage from the sensor, which is proportional to the temperature, is converted to a digital value by the ADC.

- **ADC with Interrupt Handling (IRQ):** The ADC is configured to operate in interrupt mode, ensuring efficient and non-blocking temperature data acquisition. Instead of constantly polling the ADC, the STM32 only reacts when an interrupt is generated, improving overall system performance and CPU efficiency.

- **I2C Display:** The real-time temperature data is displayed on a 0.96-inch OLED screen using I2C communication. This provides a visual indication of the temperature measured by the LM35.

- **UART Communication with ESP32:** The STM32 transmits the temperature data to the ESP32 via UART communication. The ESP32 then handles the wireless data transmission, enabling remote monitoring of the temperature values.

- **Wireless Data Transmission:** The ESP32, equipped with Wi-Fi capabilities, sends the temperature data to a cloud server, local network, or any other system for remote access and monitoring.

This system combines the strengths of STM32's IRQ-based ADC for efficient data acquisition, I2C for display control, and UART communication with the ESP32 for wireless data handling. The design aims to provide a low-power, efficient, and real-time solution for temperature monitoring in applications where wireless data transmission and non-blocking sensor readings are required.

Chapter 4

Application and Future scope

4.1 Application

In the industrial environments to measure various types of analog sensor readings, such as temperature, pressure, or humidity, use the STM32 microcontroller for industrial automation. By processing the data from sensors, the microcontroller can monitor critical parameters in real-time and make decisions based on the readings. For example, if a temperature threshold is exceeded, the system can automatically trigger alarms, adjust equipment settings, or send alerts to operators, enhancing safety and efficiency in industrial operations.

In this project, ADC IRQ (Analog-to-Digital Converter Interrupt Request) is utilized to enhance the efficiency of temperature data acquisition from the LM35 sensor connected to the STM32 microcontroller. Instead of constantly polling the sensor, which consumes processor time and power, the ADC is configured to generate an interrupt whenever a new temperature reading is available. This interrupt-driven approach allows the STM32 to process the temperature data only when necessary, freeing up the processor to handle other tasks concurrently.

Key Benefits of **ADC IRQ** method instant **of Polling method** for Analog to Digital Conversation –

- **Efficient Resource Utilization:** By using ADC IRQ, the STM32 microcontroller only dedicates processing time to temperature readings when an interrupt occurs. This reduces the load on the processor, allowing it to perform other tasks, like managing I²C and UART communication, or processing data for display or transmission.

- **Real-Time Temperature Monitoring:** The ADC IRQ method enables real-time monitoring of temperature changes. This is critical in industrial applications, where immediate response to temperature fluctuations can prevent equipment damage and ensure safety.
- **Reduced Power Consumption:** The IRQ-based system enables power savings, as the microcontroller can enter low-power modes between interrupts. Polling methods require constant activity, which leads to higher power consumption, making ADC IRQ ideal for low-power, battery-operated industrial systems.

Overall, ADC IRQ provides a robust and efficient way to handle sensor data acquisition, making it ideal for continuous monitoring and responsive decision-making in industrial applications.

4.2 Future scope

This project, which demonstrates a temperature monitoring system using the STM32 microcontroller, LM35 sensor, I2C OLED display, and ESP32 for wireless data transmission, has the potential to be expanded into a more robust and versatile industrial solution. Below are several possible directions for future development:

1. **Integration of Multiple Sensors:** The system could be enhanced by integrating multiple types of sensors (e.g., humidity, pressure, gas sensors) to monitor various environmental parameters. This would allow the solution to cater to a broader range of industrial applications, such as HVAC systems, chemical plants, or food processing facilities.
2. **Advanced Wireless Communication Protocols:** This project currently uses ESP32 for wireless communication and Blynk IoT for remote monitor system, future iterations could explore advanced industrial wireless protocols such as LoRaWAN, Zigbee, or even 5G for long-range, high-bandwidth applications and custom server as IoT platform. This would be particularly useful in large industrial environments where multiple devices need to communicate over long distances or in areas where Wi-Fi signals are weak.

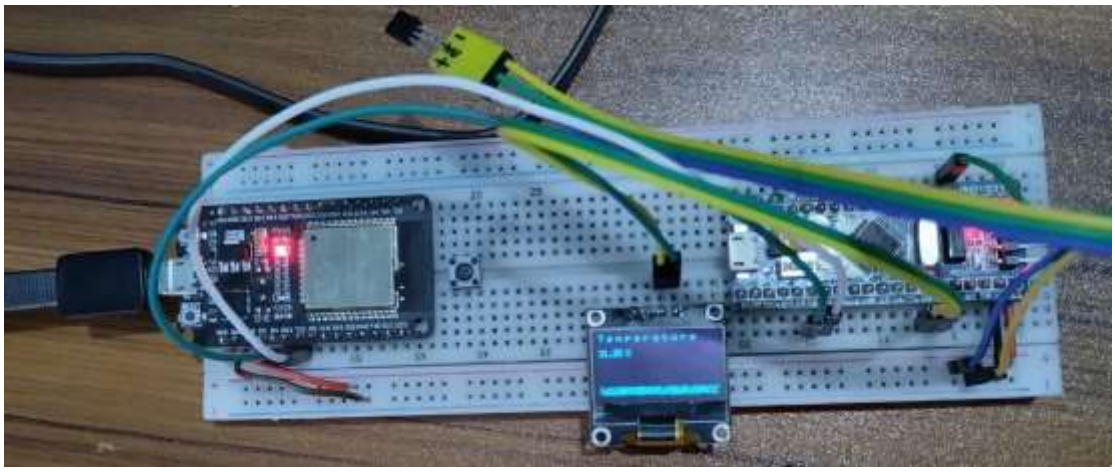
3. **Edge Computing and AI Integration:** With the growing trend of edge computing, future versions of the system could incorporate onboard processing of sensor data using machine learning algorithms deployed on the ESP32. By integrating TinyML models, the system could locally analyze the data for anomalies (e.g., detecting abnormal temperature fluctuations) without the need to rely on remote servers, improving response times and reducing network traffic.

Chapter 5

Result and Discussion

This project successfully demonstrates an efficient and reliable system for monitoring and transmitting temperature data from an LM35 sensor in an industrial setting. Using the ADC IRQ method on the STM32 microcontroller, the project effectively captures analog temperature readings, converts them to digital data, and processes them in real-time. This approach, based on interrupt handling, eliminates the need for constant polling, freeing up processing power and improving response time.

The collected temperature data is displayed on an OLED screen via I²C communication, confirming the system's ability to seamlessly integrate analog-to-digital conversion with display functionality. The use of I²C simplifies connections and maintains data integrity, providing a clear visual output for immediate monitoring of temperature changes.



Furthermore, the system successfully transmits temperature data wirelessly to the Blynk app and Google Sheets through the ESP32 module. This wireless data acquisition allows remote monitoring and logging, providing a comprehensive record of temperature variations over time. The data in Google Sheets enables long-term analysis, while real-time visualization in the Blynk app supports instant monitoring from any location with internet access.

The project demonstrates a versatile solution suitable for industrial applications, where timely and accurate temperature monitoring is critical. The integration of ADC IRQ, I²C, and UART with wireless data transmission makes this setup ideal for IoT-based monitoring systems, highlighting the effectiveness of embedded systems in real-world scenarios.

Appendix

Appendix-1: Programming code for the project

STM32F103C6T6A Resister Level Code on Keil IDE

main.c

```
#include "adc_drive.h"
#include "help_func.h"
#include "uart_drive.h"
#include "i2c_drive.h"
#include "oled_drive.h"
#include "string.h"

//char num[10];
//int analog_rx = 0;

#define VREF 3.3            // Reference voltage (V)
#define ADC_RESOLUTION 4096  // 12-bit ADC resolution

char temp_str[10];   // String buffer to store temperature as a string
char dummy_var[10];  // Dummy variable to store the converted value
int adc_value = 0;
float temperature = 0.0;

int analog_rx = 0;
int adc1_int = 0;
int adc2_int = 0;
int adc1_wd = 1;
int adc2_wd = 0;
char adc1_flag = 0;
char adc2_flag = 0;


int main(void) {
  // Initialize the ADC
  systick_init();
  UART_init(2, 115200);  // Initialize UART2 (PA2 TX, PA3 RX) at baud rate
115200
  UART_init(1, 9600);    // Initialize UART2 (PA9 TX, PA10 RX) at baud rate 9600


  adc_init(adc1, PA, 0);
  oled_init_64(1);
  oled_blank(1);

  __disable_irq();
  NVIC_EnableIRQ(ADC1_2_IRQn);
```

```
  __enable_irq();


  while (1) {
   if (adc_check(adc1, PA, 0)) {
    /*analog_rx = adc_rx(adc1, PA, 0) * 330;
    analog_rx = analog_rx / 100;
    UART_SEND(2,"ADC1: ");
    int2char(analog_rx,num);
    UART_SEND(2,num);
    str_empty(num);
    UART_TX(2,'\n');
    */

    adc_value = adc_rx(adc1, PA, 0);  // Read the ADC value from PA0

    // Calculate the temperature in Celsius
    // (adc_value / ADC_RESOLUTION) * VREF = Voltage from LM35
    float voltage = (adc_value * VREF);
    temperature = voltage / 10;  // LM35 gives 10mV/°C, so multiply by 100

    temperature = temperature * 0.615;

    float2char(temperature, temp_str, 2);  // Convert float temperature to string
    strcpy(dummy_var, temp_str);

    UART_SEND(2, "Temperature: ");
    UART_SEND(2, dummy_var);
    UART_SEND(2, "  deg C");

    str_empty(temp_str);
    UART_TX(2, '\n');



    oled_blank(1);
    oled_msg(1, 0, 0, "T e m p e r a t u r e ");
    oled_msg(1, 2, 0, dummy_var);
    oled_msg(1, 2, 25, " C");

    // Transmit the dummy variable (temperature or other data) to ESP32 via UART
    UART_SEND(1, dummy_var);  // Send dummy_var to UART2 (connected to
ESP32)
    UART_TX(1, '\n');

    DelayMs(1000);
   }
  }
}
```

```
void ADC1_2_IRQHandler() {
  if (adc1_int) {
    ADC1->CR1 &= ~0x20;
    analog_rx = ADC1->DR;
    adc1_flag = 1;
  }
  if (adc2_int) {
    ADC2->CR1 &= ~0x20;
    analog_rx = ADC2->DR;
    adc2_flag = 1;
  }
  if (adc1_wd) {
    ADC1->CR1 &= ~0x40;
    ADC1->SR &= ~0x01;
    analog_rx = ADC1->DR;
    adc1_flag = 1;
  }
  if (adc2_wd) {
    ADC2->CR1 &= ~0x40;
    ADC2->SR &= ~0x01;
    analog_rx = ADC2->DR;
    adc2_flag = 1;
  }
}
```

GPIO pin assigned driver

gp_drive.h

```
#define RCC_APBENR (*((volatile unsigned long *)0x40021018))

#define GPIO_A (*((volatile unsigned long *)0x40010800))
#define GPIO_B (*((volatile unsigned long *)0x40010C00))
#define GPIO_C (*((volatile unsigned long *)0x40011000))

#define PA 1
#define PB 2
#define PC 3

#define HIGH 1
#define LOW 0

#define IN 0
#define OUT10 1
#define OUT2 2
#define OUT50 3

#define I_AN 0
#define I_F 1
#define I_PP 2
```

```
#define O_GP_PP 0
#define O_GP_OD 1
#define O_AF_PP 2
#define O_AF_OD 3


void PINc(unsigned short pin, unsigned short STATUS);
void init_GP(unsigned short PORT, unsigned short PIN, unsigned short DIR,
unsigned short OPT);
int R_GP(unsigned short PORT, unsigned short pin);
void W_GP(unsigned short PORT, unsigned short pin, unsigned short STATUS);
void toggle_GP(unsigned short Port, unsigned short pin);
void BLED(unsigned short state);
void B_init(void);
void Digital_Input(unsigned short PORT, unsigned short PIN);
void Digital_Output(unsigned short PORT, unsigned short PIN);
```

gp_drive.c

```
//// Library used to setup GPIOs

#include "stm32f10x.h"
#include "gp_drive.h"


void init_GP(unsigned short PORT, unsigned short PIN, unsigned short DIR,
unsigned short OPT) {
  volatile unsigned long *CR;
  unsigned short tPin = PIN;
  unsigned short offset = 0x00;

  if (PIN > 7) {
    tPin -= 8;
    offset = 0x01;
  }


  if (PORT == 1) {
    RCC_APBENR |= 0x4;

    CR = (volatile unsigned long *)(&GPIO_A + offset);
  } else if (PORT == 2) {
    RCC_APBENR |= 0x8;
    CR = (volatile unsigned long *)(&GPIO_B + offset);
  } else if (PORT == 3) {
    RCC_APBENR |= 0x10;
    CR = (volatile unsigned long *)(&GPIO_C + offset);
  }

  *CR &= ~(0xf << (tPin)*4);
```

```
  *CR |= ((DIR << (tPin * 4)) | (OPT << (tPin * 4 + 2)));
}

int R_GP(unsigned short PORT, unsigned short pin) {
  volatile unsigned long *IDR;
  unsigned long offset = 0x02;
  int state;

  if (PORT == 1) {
    IDR = (volatile unsigned long *)(&GPIO_A + offset);
  } else if (PORT == 2) {
    IDR = (volatile unsigned long *)(&GPIO_B + offset);
  } else if (PORT == 3) {
    IDR = (volatile unsigned long *)(&GPIO_C + offset);
  }
  state = ((*IDR & (1 << pin)) >> pin);

  return state;
}

void W_GP(unsigned short PORT, unsigned short pin, unsigned short STATUS) {
  volatile unsigned long *ODR;
  unsigned long offset = 0x03;
  if (PORT == 1) {
    ODR = (volatile unsigned long *)(&GPIO_A + offset);
  } else if (PORT == 2) {
    ODR = (volatile unsigned long *)(&GPIO_B + offset);
  } else if (PORT == 3) {
    ODR = (volatile unsigned long *)(&GPIO_C + offset);
  }
  STATUS ? (*ODR |= (STATUS << pin)) : (*ODR &= ~(1 << pin));
}

void toggle_GP(unsigned short Port, unsigned short pin) {
  if (R_GP(Port, pin)) {
    W_GP(Port, pin, 0);
  } else {
    W_GP(Port, pin, 1);
  }
}


void PINc(unsigned short pin, unsigned short STATUS) {
  STATUS ? (GPIOC->ODR |= (STATUS << pin)) : (GPIOC->ODR &= ~(1 <<
pin));
}

void B_init(void) {
  init_GP(PC, 13, OUT50, O_GP_PP);
}
```

```
void BLED(unsigned short state) {
  W_GP(PC, 13, state);
}

void Digital_Input(unsigned short PORT, unsigned short PIN) {
  init_GP(PORT, PIN, IN, I_PP);
}

void Digital_Output(unsigned short PORT, unsigned short PIN) {
  init_GP(PORT, PIN, OUT50, O_GP_PP);
}
```

Driver for timer

systic_time.h

```
void systick_init(void);
void DelayMs(unsigned long t);
void systick_int(unsigned short uart_1_mgr[],unsigned short
uart_2_mgr[],unsigned short uart_3_mgr[]);
void systick_int_start(void);
void Delaymicro(void);
void DelayUs(unsigned long t);
```

systic_time.c

```
#include "stm32f10x.h"
#include "systick_time.h"

void systick_init(void) {
  SysTick->CTRL = 0;
  SysTick->LOAD = 0x00FFFFFF;
  SysTick->VAL = 0;
  SysTick->CTRL = 5;
}
void Delaymicro(void) {
  SysTick->LOAD = 72;
  SysTick->VAL = 0;
  while ((SysTick->CTRL & 0x00010000) == 0)
    ;
}

void DelayUs(unsigned long t) {
  for (; t > 0; t--) {
    Delaymicro();
  }
}


void DelayMillis(void) {
```

```
  SysTick->LOAD = 0x11940;
  SysTick->VAL = 0;
  while ((SysTick->CTRL & 0x00010000) == 0)
    ;
}

void DelayMs(unsigned long t) {
  for (; t > 0; t--) {
    DelayMillis();
  }
}

void systick_int_start(void) {
  __disable_irq();
  SysTick->CTRL = 0;
  SysTick->LOAD = 7200000;
  SysTick->VAL = 0;
  SysTick->CTRL |= 7;
  __enable_irq();
}

void systick_int(unsigned short uart_1_mgr[], unsigned short uart_2_mgr[],
unsigned short uart_3_mgr[]) {
  if (uart_1_mgr[0] != 0) {
    if (uart_1_mgr[6] == 0) {
      uart_1_mgr[0] = 0;
      uart_1_mgr[1] = 1;
      uart_1_mgr[5] = 0;
      systick_init();
    } else {
      uart_1_mgr[6]--;
    }
  } else if (uart_2_mgr[0] != 0) {
    if (uart_2_mgr[6] == 0) {
      uart_2_mgr[0] = 0;
      uart_2_mgr[1] = 1;
      uart_2_mgr[5] = 0;
      systick_init();
    } else {
      uart_2_mgr[6]--;
    }
  } else if (uart_3_mgr[0] != 0) {
    if (uart_3_mgr[6] == 0) {
      uart_3_mgr[0] = 0;
      uart_3_mgr[1] = 1;
      uart_3_mgr[5] = 0;
      systick_init();
    } else {
      uart_3_mgr[6]--;
    }
```

```
  }
}
```

ADC Driver

adc.h

```
#include "gp_drive.h"
#include "stm32f10x.h"
#include "systick_time.h"

#define adc1        1
#define adc2        2

char adc_init(char adc, short port, short pin);
char adc_check(char adc, short port, short pin);
int adc_rx(char adc, short port, short pin);
```

adc.c

```
#include "adc_drive.h"

/*
PA0 -> ADC12_IN0
PA1 -> ADC12_IN1
PA2 -> ADC12_IN2
PA3 -> ADC12_IN3
PA4 -> ADC12_IN4
PA5 -> ADC12_IN5
PA6 -> ADC12_IN6
PA7 -> ADC12_IN7
PB0 -> ADC12_IN8
PB1 -> ADC12_IN9

PC0 -> ADC12_IN10
PC1 -> ADC12_IN11
PC2 -> ADC12_IN12
PC3 -> ADC12_IN13
PC4 -> ADC12_IN14
PC5 -> ADC12_IN15

ADC12_IN16 input channel which is used to convert the sensor output voltage into
a digital value.


*/

// Initializing the ADC for the STM32F1
char adc_init(char adc, short port, short pin) {
  char channel;
  char result = 0;
```

```
  if (port == PA) {
    if (pin < 8) {
      result = 1;
      channel = pin;
    }
  } else if (port == PB) {
    if (pin < 2) {
      result = 1;
      channel = 8 + pin;
    }
  } else if (port == PC) {
    if (pin < 6) {
      result = 1;
      channel = 10 + pin;
    }
  }
  if (result) {
    init_GP(port, pin, IN, I_AN);
    if (adc == adc1) {
      RCC->APB2ENR |= 0x201;
      ADC1->CR2 = 0;
      ADC1->SQR3 = channel;
      ADC1->CR2 |= 1;
      DelayMs(100);
      ADC1->CR2 |= 1;
      ADC1->CR2 |= 2;
    } else if (adc == adc2) {
      RCC->APB2ENR |= 0x401;
      ADC2->CR2 = 0;
      ADC2->SQR3 = channel;
      ADC2->CR2 |= 1;
      DelayMs(100);
      ADC2->CR2 |= 1;
      ADC2->CR2 |= 2;
    }
  }
  return result;
}

// Reading the flag that says the data is ready
char adc_check(char adc, short port, short pin) {
  char check = 0;
  if (adc == adc1) {
    if (ADC1->SR & 2) {
      check = 1;
    }
  } else if (adc == adc2) {
    if (ADC2->SR & 2) {
      check = 1;
    }
```

```
  }


  return check;
}

// Reading the ADC value
int adc_rx(char adc, short port, short pin) {
  int result = 0;
  int data = 0;
  if (adc == adc1) {
    data = ADC1->DR;
  } else if (adc == adc2) {
    data = ADC2->DR;
  }

  result = (data * 1000) / 0xfff;


  return result;
}
```

UART Driver for UART Communication

uart.h

```
#include "stm32f10x.h"
#include "gp_drive.h"
#include "systick_time.h"


unsigned long USART_BRR(unsigned short usart, unsigned long BR);
void UART_init(unsigned short usart, unsigned long BR);
char UART_RX(unsigned short uart);
void UART_TX(unsigned short uart, char c);
void UART_ISR(unsigned short uart,unsigned short uart_mgr[], char str[]);
void UART_SEND(unsigned short uart, char str[]);
void UART_msg(unsigned short uart, char str[], unsigned short str_mgr[]);
```

uart.c

```
#include "uart_drive.h"
#include "systick_time.h"

void UART_init(unsigned short usart, unsigned long BR) {
  /// If using USART1 clock speed 72Mhz, else 36Mhz
  /// USART3 -> PB10 (Tx) and PB11(Rx)
  /// USART2 -> PA2 (Tx) and PA3(Rx)
  /// USART1 -> PA9 (Tx) and PA10(Rx)
```

```
  // Enable the Alternate Function for PINs
  unsigned long BRR_Cal;

  BRR_Cal = USART_BRR(usart, BR);

  RCC->APB2ENR |= 1;


  if (usart == 1) {
    __disable_irq();
    RCC->APB2ENR |= 0x4000;
    init_GP(PA, 9, OUT50, O_AF_PP);
    init_GP(PA, 10, IN, I_PP);
    // Setup the baude rate for 9600 bps
    USART1->BRR = BRR_Cal;
    USART1->CR1 |= 8;
    USART1->CR1 |= 4;
    USART1->CR1 |= 0x2000;
    USART1->CR1 |= 0x20;
    NVIC_EnableIRQ(USART1_IRQn);
    __enable_irq();
  } else if (usart == 2) {  //-----------Init UART ------------///
                        // Enable UART2
    __disable_irq();
    RCC->APB1ENR |= 0x20000;
    // Enable the related PINs
    init_GP(PA, 2, OUT50, O_AF_PP);
    init_GP(PA, 3, IN, I_PP);
    // Setup the baude rate for 9600 bps
    USART2->BRR = BRR_Cal;
    // Enable Uart Transmit
    USART2->CR1 |= 8;
    // Enable Uart Recive
    USART2->CR1 |= 4;
    // Enable Uart
    USART2->CR1 |= 0x2000;
    USART2->CR1 |= 0x20;
    NVIC_EnableIRQ(USART2_IRQn);
    __enable_irq();
  }
}

unsigned long USART_BRR(unsigned short usart, unsigned long BR) {
  unsigned long div = 36000000UL;
  unsigned long dec;
  unsigned long final;
  double frac = 36000000.00;
  double frac2 = 1.00;
```

```
  if (usart == 1) {
    div = 72000000UL;
    frac = 72000000.00;
  }
  div = div / (BR * 16);
  frac = 16 * ((frac / (BR * 16)) - div);
  dec = frac;
  frac2 = 100 * (frac - dec);
  if (frac2 > 50) {
    dec++;
    if (dec == 16) {
      dec = 0;
      div++;
    }
  }

  final = (div << 4);
  final += dec;

  return final;
}


char UART_RX(unsigned short uart) {
  char c;
  if (uart == 1) {
    while ((USART1->SR & 0x20) == 0x00) {};
    c = USART1->DR;
  } else if (uart == 2) {
    while ((USART2->SR & 0x20) == 0x00) {};
    c = USART2->DR;
  }
  if (uart == 3) {
    while ((USART3->SR & 0x20) == 0x00) {};
    c = USART3->DR;
  }
  return c;
}

void UART_TX(unsigned short uart, char c) {
  if (uart == 1) {
    while ((USART1->SR & (0x80)) == 0x00) {};
    USART1->DR = c;
  } else if (uart == 2) {
    while ((USART2->SR & (0x80)) == 0x00) {};
    USART2->DR = c;
  }
  if (uart == 3) {
    while ((USART3->SR & (0x80)) == 0x00) {};
    USART3->DR = c;
```

```
  }
}

/*
1- Define my uart
2- is it bridge or process or both
if process : Sting to fullfill, cnt , signal
*/

void UART_ISR(unsigned short uart, unsigned short uart_mgr[], char str[]) {
  if (uart_mgr[2] == 0) {
    str[uart_mgr[0]] = UART_RX(uart);
    if (uart_mgr[3]) {
      if (str[uart_mgr[0]] == uart_mgr[4]) {
        uart_mgr[0] = 0;
        uart_mgr[1] = 1;
      } else {
        uart_mgr[0]++;
      }
    } else {
      // Timer strategy
      uart_mgr[0]++;
      uart_mgr[6] = uart_mgr[5];
      systick_int_start();
    }
  } else {
    UART_TX(uart_mgr[2], UART_RX(uart));
  }
}

void UART_SEND(unsigned short uart, char str[]) {
  int i = 0;
  while (str[i] != '\0') {
    UART_TX(uart, str[i]);
    i++;
  }
}

void UART_msg(unsigned short uart, char str[], unsigned short str_mgr[]) {
  unsigned long timeOut = 720000000;
  UART_SEND(uart, str);
  while (str_mgr[1] == 0 & timeOut != 0) {
    timeOut--;
  }
  str_mgr[1] = 0;
}
```

I2C Driver for I2C Communication

i2c.h

```
#include "stm32f10x.h"
#include "gp_drive.h"

#define i2c_FM 0x2d
#define i2c_SM 0xB4

void i2c_init(char i2c,unsigned short speed_mode);
void i2c_write(char i2c, char address,char data[]);
void i2c_start(char i2c);
void i2c_add(char i2c, char address,char RW);
void i2c_data(char i2c,char data);
void i2c_stop(char i2c);
```

i2c.c

```
#include "i2c_drive.h"
/*
I2C2
PB10 -> SCL
PB11 -> SDA

I2C1
PB6 -> SCL
PB7 -> SDA

*/

// i2c_init()
void i2c_init(char i2c, unsigned short speed_mode) {
  RCC->APB2ENR |= 1;

  if (i2c == 1) {
    RCC->APB1ENR |= 0x200000;
    // Pin enable
    init_GP(PB, 6, OUT50, O_AF_OD);
    init_GP(PB, 7, OUT50, O_AF_OD);
    I2C1->CR1 |= 0x8000;
    I2C1->CR1 &= ~0x8000;
    I2C1->CR2 = 0x8;
    I2C1->CCR = speed_mode;
    I2C1->TRISE = 0x9;
    I2C1->CR1 |= 1;
  } else if (i2c == 2) {
    RCC->APB1ENR |= 0x400000;
    // Pin enable
    init_GP(PB, 10, OUT50, O_AF_OD);
    init_GP(PB, 11, OUT50, O_AF_OD);
```

```
    I2C2->CR1 |= 0x8000;
    I2C2->CR1 &= ~0x8000;
    I2C2->CR2 = 0x8;
    I2C2->CCR = speed_mode;
    I2C2->TRISE = 0x9;
    I2C2->CR1 |= 1;
  }
}

// Start step
void i2c_start(char i2c) {
  if (i2c == 1) {
    I2C1->CR1 |= 0x100;
    while (!(I2C1->SR1 & 1)) {};
  } else if (i2c == 2) {
    I2C2->CR1 |= 0x100;
    while (!(I2C2->SR1 & 1)) {};
  }
}
// Sending the address + R or Write
void i2c_add(char i2c, char address, char RW) {
  volatile int tmp;
  if (i2c == 1) {
    I2C1->DR = (address | RW);
    while ((I2C1->SR1 & 2) == 0) {};
    while ((I2C1->SR1 & 2)) {
      tmp = I2C1->SR1;
      tmp = I2C1->SR2;
      if ((I2C1->SR1 & 2) == 0) {
        break;
      }
    }
  } else if (i2c == 2) {
    I2C2->DR = (address | RW);
    while ((I2C2->SR1 & 2) == 0) {};
    while ((I2C2->SR1 & 2)) {
      tmp = I2C2->SR1;
      tmp = I2C2->SR2;
      if ((I2C2->SR1 & 2) == 0) {
        break;
      }
    }
  }
}
// Sending data step
void i2c_data(char i2c, char data) {
  if (i2c == 1) {
    while ((I2C1->SR1 & 0x80) == 0) {}
    I2C1->DR = data;
    while ((I2C1->SR1 & 0x80) == 0) {}
```

```
  } else if (i2c == 2) {
    while ((I2C2->SR1 & 0x80) == 0) {}
    I2C2->DR = data;
    while ((I2C2->SR1 & 0x80) == 0) {}
  }
}
// Stop step
void i2c_stop(char i2c) {
  volatile int tmp;
  if (i2c == 1) {
    tmp = I2C1->SR1;
    tmp = I2C1->SR2;
    I2C1->CR1 |= 0x200;
  } else if (i2c == 2) {
    tmp = I2C2->SR1;
    tmp = I2C2->SR2;
    I2C2->CR1 |= 0x200;
  }
}
// i2c_write()
void i2c_write(char i2c, char address, char data[]) {
  int i = 0;

  i2c_start(i2c);

  i2c_add(i2c, address, 0);

  while (data[i]) {
    i2c_data(i2c, data[i]);
    i++;
  }
  i2c_stop(i2c);
}
```

OLED Interface with STM32

oled.h

```
#include "i2c_drive.h"


void oled_init(char i2c, char screen_size);
void oled_init_64(char i2c);
void oled_init_32(char i2c);
void oled_blank(char i2c);
```

```
void oled_print(char i2c, char str[]);
void oled_msg(char i2c, char Ypos, char Xpos, char str[]);

static const char ASCII[][5] = {
  { 0x00, 0x00, 0x00, 0x00, 0x00 }  // 20
 ,
  { 0x00, 0x00, 0x5f, 0x00, 0x00 }  // 21 !
 ,
  { 0x00, 0x07, 0x00, 0x07, 0x00 }  // 22 "
 ,
  { 0x14, 0x7f, 0x14, 0x7f, 0x14 }  // 23 #
 ,
  { 0x24, 0x2a, 0x7f, 0x2a, 0x12 }  // 24 $
 ,
  { 0x23, 0x13, 0x08, 0x64, 0x62 }  // 25 %
 ,
  { 0x36, 0x49, 0x55, 0x22, 0x50 }  // 26 &
 ,
  { 0x00, 0x05, 0x03, 0x00, 0x00 }  // 27 '
 ,
  { 0x00, 0x1c, 0x22, 0x41, 0x00 }  // 28 (
 ,
  { 0x00, 0x41, 0x22, 0x1c, 0x00 }  // 29 )
 ,
  { 0x14, 0x08, 0x3e, 0x08, 0x14 }  // 2a *
 ,
  { 0x08, 0x08, 0x3e, 0x08, 0x08 }  // 2b +
 ,
  { 0x00, 0x50, 0x30, 0x00, 0x00 }  // 2c ,
 ,
  { 0x08, 0x08, 0x08, 0x08, 0x08 }  // 2d -
 ,
  { 0x00, 0x60, 0x60, 0x00, 0x00 }  // 2e .
 ,
  { 0x20, 0x10, 0x08, 0x04, 0x02 }  // 2f /
 ,
  { 0x3e, 0x51, 0x49, 0x45, 0x3e }  // 30 0
 ,
  { 0x00, 0x42, 0x7f, 0x40, 0x00 }  // 31 1
 ,
  { 0x42, 0x61, 0x51, 0x49, 0x46 }  // 32 2
 ,
  { 0x21, 0x41, 0x45, 0x4b, 0x31 }  // 33 3
 ,
  { 0x18, 0x14, 0x12, 0x7f, 0x10 }  // 34 4
 ,
  { 0x27, 0x45, 0x45, 0x45, 0x39 }  // 35 5
 ,
  { 0x3c, 0x4a, 0x49, 0x49, 0x30 }  // 36 6
 ,
```

```
{ 0x01, 0x71, 0x09, 0x05, 0x03 }  // 37 7
,
{ 0x36, 0x49, 0x49, 0x49, 0x36 }  // 38 8
,
{ 0x06, 0x49, 0x49, 0x29, 0x1e }  // 39 9
,
{ 0x00, 0x36, 0x36, 0x00, 0x00 }  // 3a :
,
{ 0x00, 0x56, 0x36, 0x00, 0x00 }  // 3b ;
,
{ 0x08, 0x14, 0x22, 0x41, 0x00 }  // 3c <
,
{ 0x14, 0x14, 0x14, 0x14, 0x14 }  // 3d =
,
{ 0x00, 0x41, 0x22, 0x14, 0x08 }  // 3e >
,
{ 0x02, 0x01, 0x51, 0x09, 0x06 }  // 3f ?
,
{ 0x32, 0x49, 0x79, 0x41, 0x3e }  // 40 @
,
{ 0x7e, 0x11, 0x11, 0x11, 0x7e }  // 41 A
,
{ 0x7f, 0x49, 0x49, 0x49, 0x36 }  // 42 B
,
{ 0x3e, 0x41, 0x41, 0x41, 0x22 }  // 43 C
,
{ 0x7f, 0x41, 0x41, 0x22, 0x1c }  // 44 D
,
{ 0x7f, 0x49, 0x49, 0x49, 0x41 }  // 45 E
,
{ 0x7f, 0x09, 0x09, 0x09, 0x01 }  // 46 F
,
{ 0x3e, 0x41, 0x49, 0x49, 0x7a }  // 47 G
,
{ 0x7f, 0x08, 0x08, 0x08, 0x7f }  // 48 H
,
{ 0x00, 0x41, 0x7f, 0x41, 0x00 }  // 49 I
,
{ 0x20, 0x40, 0x41, 0x3f, 0x01 }  // 4a J
,
{ 0x7f, 0x08, 0x14, 0x22, 0x41 }  // 4b K
,
{ 0x7f, 0x40, 0x40, 0x40, 0x40 }  // 4c L
,
{ 0x7f, 0x02, 0x0c, 0x02, 0x7f }  // 4d M
,
{ 0x7f, 0x04, 0x08, 0x10, 0x7f }  // 4e N
,
{ 0x3e, 0x41, 0x41, 0x41, 0x3e }  // 4f O
,
```

```
{ 0x7f, 0x09, 0x09, 0x09, 0x06 }  // 50 P
,
{ 0x3e, 0x41, 0x51, 0x21, 0x5e }  // 51 Q
,
{ 0x7f, 0x09, 0x19, 0x29, 0x46 }  // 52 R
,
{ 0x46, 0x49, 0x49, 0x49, 0x31 }  // 53 S
,
{ 0x01, 0x01, 0x7f, 0x01, 0x01 }  // 54 T
,
{ 0x3f, 0x40, 0x40, 0x40, 0x3f }  // 55 U
,
{ 0x1f, 0x20, 0x40, 0x20, 0x1f }  // 56 V
,
{ 0x3f, 0x40, 0x38, 0x40, 0x3f }  // 57 W
,
{ 0x63, 0x14, 0x08, 0x14, 0x63 }  // 58 X
,
{ 0x07, 0x08, 0x70, 0x08, 0x07 }  // 59 Y
,
{ 0x61, 0x51, 0x49, 0x45, 0x43 }  // 5a Z
,
{ 0x00, 0x7f, 0x41, 0x41, 0x00 }  // 5b [
,
{ 0x02, 0x04, 0x08, 0x10, 0x20 }  // 5c ?
,
{ 0x00, 0x41, 0x41, 0x7f, 0x00 }  // 5d ]
,
{ 0x04, 0x02, 0x01, 0x02, 0x04 }  // 5e ^
,
{ 0x40, 0x40, 0x40, 0x40, 0x40 }  // 5f _
,
{ 0x00, 0x01, 0x02, 0x04, 0x00 }  // 60 `
,
{ 0x20, 0x54, 0x54, 0x54, 0x78 }  // 61 a
,
{ 0x7f, 0x48, 0x44, 0x44, 0x38 }  // 62 b
,
{ 0x38, 0x44, 0x44, 0x44, 0x20 }  // 63 c
,
{ 0x38, 0x44, 0x44, 0x48, 0x7f }  // 64 d
,
{ 0x38, 0x54, 0x54, 0x54, 0x18 }  // 65 e
,
{ 0x08, 0x7e, 0x09, 0x01, 0x02 }  // 66 f
,
{ 0x0c, 0x52, 0x52, 0x52, 0x3e }  // 67 g
,
{ 0x7f, 0x08, 0x04, 0x04, 0x78 }  // 68 h
,
```

```
  { 0x00, 0x44, 0x7d, 0x40, 0x00 }  // 69 i
  ,
  { 0x20, 0x40, 0x44, 0x3d, 0x00 }  // 6a j
  ,
  { 0x7f, 0x10, 0x28, 0x44, 0x00 }  // 6b k
  ,
  { 0x00, 0x41, 0x7f, 0x40, 0x00 }  // 6c l
  ,
  { 0x7c, 0x04, 0x18, 0x04, 0x78 }  // 6d m
  ,
  { 0x7c, 0x08, 0x04, 0x04, 0x78 }  // 6e n
  ,
  { 0x38, 0x44, 0x44, 0x44, 0x38 }  // 6f o
  ,
  { 0x7c, 0x14, 0x14, 0x14, 0x08 }  // 70 p
  ,
  { 0x08, 0x14, 0x14, 0x18, 0x7c }  // 71 q
  ,
  { 0x7c, 0x08, 0x04, 0x04, 0x08 }  // 72 r
  ,
  { 0x48, 0x54, 0x54, 0x54, 0x20 }  // 73 s
  ,
  { 0x04, 0x3f, 0x44, 0x40, 0x20 }  // 74 t
  ,
  { 0x3c, 0x40, 0x40, 0x20, 0x7c }  // 75 u
  ,
  { 0x1c, 0x20, 0x40, 0x20, 0x1c }  // 76 v
  ,
  { 0x3c, 0x40, 0x30, 0x40, 0x3c }  // 77 w
  ,
  { 0x44, 0x28, 0x10, 0x28, 0x44 }  // 78 x
  ,
  { 0x0c, 0x50, 0x50, 0x50, 0x3c }  // 79 y
  ,
  { 0x44, 0x64, 0x54, 0x4c, 0x44 }  // 7a z
  ,
  { 0x00, 0x08, 0x36, 0x41, 0x00 }  // 7b {
  ,
  { 0x00, 0x00, 0x7f, 0x00, 0x00 }  // 7c |
  ,
  { 0x00, 0x41, 0x36, 0x08, 0x00 }  // 7d }
  ,
  { 0x10, 0x08, 0x08, 0x10, 0x08 }  // 7e ?
  ,
  { 0x78, 0x46, 0x41, 0x46, 0x78 }  // 7f ?
};
```

oled.c

```
#include "oled_drive.h"



//oled_cmd_1byte()
void oled_cmd_1byte(char i2c, char data) {

  i2c_start(i2c);

  i2c_add(i2c, 0x78, 0);  //0x78 is the primary address

  i2c_data(i2c, 0x00);  // Control function for a command
  i2c_data(i2c, data);

  i2c_stop(i2c);
}

//oled_cmd_2byte()
void oled_cmd_2byte(char i2c, char data[]) {
  int i = 0;

  i2c_start(i2c);

  i2c_add(i2c, 0x78, 0);  //0x78 is the primary address

  i2c_data(i2c, 0x00);  // Control function for a command
  for (i = 0; i < 2; i++) {
    i2c_data(i2c, data[i]);
  }

  i2c_stop(i2c);
}

//oled_init()
void oled_init(char i2c, char screen_size) {
  i2c_init(i2c, i2c_FM);
  char cmd[] = { 0xA8, 0x3F };
  oled_cmd_2byte(i2c, cmd);
  char cmd1[] = { 0xD3, 0x00 };
  oled_cmd_2byte(i2c, cmd1);

  oled_cmd_1byte(i2c, 0x40);
  oled_cmd_1byte(i2c, 0xA1);

  oled_cmd_1byte(i2c, 0xC8);

  char cmd2[] = { 0xDA, screen_size };
  oled_cmd_2byte(i2c, cmd2);
```

```
  char cmd3[] = { 0x81, 0x7F };
  oled_cmd_2byte(i2c, cmd3);

  oled_cmd_1byte(i2c, 0xA4);
  oled_cmd_1byte(i2c, 0xA6);

  char cmd4[] = { 0xD5, 0x80 };
  oled_cmd_2byte(i2c, cmd4);

  char cmd5[] = { 0x8D, 0x14 };
  oled_cmd_2byte(i2c, cmd5);

  oled_cmd_1byte(i2c, 0xAF);

  char cmd6[] = { 0x20, 0x10 };
  oled_cmd_2byte(i2c, cmd6);
}
void oled_init_64(char i2c) {
  oled_init(i2c, 0x12);
}

void oled_init_32(char i2c) {
  oled_init(i2c, 0x22);
}

//oled_data()
void oled_data(char i2c, char data) {

  i2c_start(i2c);

  i2c_add(i2c, 0x78, 0);  //0x78 is the primary address

  i2c_data(i2c, 0x40);  // Control function for a data
  i2c_data(i2c, data);

  i2c_stop(i2c);
}
// oled_pos()

void oled_pos(char i2c, char Ypos, char Xpos) {
  oled_cmd_1byte(i2c, 0x00 + (0x0F & Xpos));
  oled_cmd_1byte(i2c, 0x10 + (0x0F & (Xpos >> 4)));
  oled_cmd_1byte(i2c, 0xB0 + Ypos);
}

// oled_blank()
void oled_blank(char i2c) {
  int i, j;
  oled_pos(i2c, 0, 0);
  for (i = 0; i < 8; i++) {
```

```
   for (j = 0; j < 128; j++) {
     oled_data(i2c, 0x00);
    }
  }
  oled_pos(i2c, 0, 0);
}

//oled_print
void oled_print(char i2c, char str[]) {
  int i, j;
  i = 0;
  while (str[i]) {
    for (j = 0; j < 5; j++) {
      oled_data(i2c, ASCII[(str[i] - 32)][j]);
    }
    i++;
  }
}

//oled_msg
void oled_msg(char i2c, char Ypos, char Xpos, char str[]) {
  oled_pos(i2c, Ypos, Xpos);
  oled_print(i2c, str);
}
```

Typecasting from float to string

help.h

```
void float2char(float num, char str[], int precision);
```

help.c

```
void float2char(float num, char str[], int precision) {
    char lstr[30];          // Temporary string to hold intermediate result
    int int_part = (int)num;  // Extract integer part of the float
    float frac_part;         // Fractional part of the float
    int cnt = 0, j = 0, neg = 0;

    // Handle negative numbers
    if (num < 0) {
        num = -num;
        int_part = (int)num;
        neg = 1;
    }

    // Convert the integer part to string using a similar method to int2char
    while (int_part >= 10) {
        lstr[cnt] = int_part % 10 + '0';
```

```
        int_part /= 10;
        cnt++;
    }
    lstr[cnt] = int_part + '0';  // Store the last digit of the integer part

    // If the number is negative, add '-' to the string
    if (neg) {
        cnt++;
        lstr[cnt] = '-';
    }

    // Copy the integer part in reverse order
    for (j = cnt; j >= 0; j--) {
        str[cnt - j] = lstr[j];
    }
    str[cnt + 1] = '.';  // Add the decimal point
    cnt++;  // Update counter to track the current position

    // Extract the fractional part
    frac_part = num - (int)num;  // Fractional part is num - integer part

    // Convert the fractional part based on the specified precision
    for (j = 0; j < precision; j++) {
        frac_part *= 10;  // Shift fractional part by 1 decimal place
        int frac_digit = (int)frac_part;
        str[cnt + 1] = frac_digit + '0';  // Convert to character and store
        frac_part -= frac_digit;  // Remove the digit we just processed
        cnt++;
    }

    str[cnt + 1] = '\0';  // Null-terminate the string
}
```

ESP32 Embedded C++ coding on Arduino IDE

adc_irq.ino

```
#define BLYNK_TEMPLATE_ID "TMPL6WYmvnUOy"
#define BLYNK_TEMPLATE_NAME "ADCIRQ"
#define BLYNK_AUTH_TOKEN "ANBiCZgDUCRkSHMvSTxgNFmMq0wSIAbR"

#include <Arduino.h>
#include <WiFi.h>
#include <BlynkSimpleEsp32.h>
#include "time.h"
#include <ESP_Google_Sheet_Client.h>
// For SD/SD_MMC mounting helper
#include <GS_SDHelper.h>
```

```
// Your WiFi credentials
char ssid[] = "ESP_Network";
char pass[] = "AlAmin789";

// To store received data from STM32
char received_data[6];
int dataIndex = 0;

// Blynk virtual pin for displaying data
#define VIRTUAL_PIN V0

// Google Project ID
#define PROJECT_ID "adcirq"
#define CLIENT_EMAIL "adcirq@adcirq.iam.gserviceaccount.com"  // Service
Account's client email

// Service Account's private key

const char PRIVATE_KEY[] PROGMEM = "-----BEGIN PRIVATE KEY-----
\nMIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBAQDYuhlc5ba3
ZGMF\nC6UHUS+GUQOukony7e7cMXK4ljJ1dGAQgKQsGDVjBvmlTAAPpSFAf
wzwEwY4utJC\nh8vihl7HWL9dVTwzG59Z5nRmBf92j478ZJAPqosf04vHmsGkJ0j4
0IJIvN/FoHzi\nR3CC76WhBBjL/3RWLy3AdNHSDqk72o8Q92f0S1WMQ4TVjSDY
XVR2A4lKDNWxTdeP\nZUy0KzEUZ1fBk558CA9keNe2/kSrpk8C3cy7gN/ywXZQy
oac6/ILfwcQITBXGyhz\nODgcpolzoKxVbtNhlTHe+TjtuUyu9vcVIPuQoIzm4np+Z
0dPyebNXOYppAe0gIqj\n3RhaN/pVAgMBAAECggEADXGWn8axS8H8StruKaLvG
Q/CDZx4b/gzbQRCOThthXZE\nST6TMH48Vya4UKxP6qiHHgP9KCoKvkHuFGn
UPpQWoZrPNnmfKnQIF/o6j5bHdUuY\noCGXnOkq1SNFur2BgO6Q6u8TZ3JHFt
a90gZPCkg8VKpQ5KkCRrnv4BrYmYU55gT4\nHG+UZhOVSdUoYY8P08hu0pvX8
ckc/2BF7S9uwjmujjrVFY/5QGdh8XpNil8oWw3c\nRP9aHOIbnEMGxhd93RL7WsX
SF28k15JEJSRm4i/cHG6A63Y3xUyqtuENiywQgsU1\nhwSM0pKhYia/IVd7blZMyf
TFvENEBQvV/psqJgQ0xwKBgQD9dig2dR6vrRFguiKN\ngZlszSka0IHeDeE3y4Klx
3W2f1Cf0xk1op1KACHRQODaifKzYmj72uB7815mtrj6\n3U84BVpzg1pt47B1Wf1J
PT+FjtL2NRcft07G2BGbHbRuMMgm9mUtFGhizGXlkxjh\ngWytoxUrjKzW1135g5
AYdXp9KwKBgQDa5cJZ5wD1zrgwSod8onw618tw3A11fRHm\ncpJ8Zv3cK1jRvOC
2hTVzILr165U8ZpEVzsO5uG4iEV6TOYdDIaAP9vfbHTaHsNOY\n6HFe6cIxRZwF
9uLBJxOuQMJZI9plRQc6mp6sVyNeGb97+6WNeJAYwzXIUEp0GsMf\nvhLHTt6m
fwKBgQCLP2/mQ0ABmd5zOq+i+HF28dvETIscmJmEEr2LIGLOSXHc7Jfr\n/JPX
RO1PTgp6ZdE9tjIhM+Wik1joqzhDMnTEPUStM5vpxW57el4J76lHUekatJ4o\noYe
Y/Lnnc1FePFSp+zIoqw6SQOM2VjMUKQuw9Wi7DUUjnxyU6S+wEFKkEwKBgQ
C9\n8G4DGppPsI9Mf+8uq8NGP+esx5T6JIt6vmHzhl6zQ/2vCMioN9BYVMdGCqS
VI56z\nIJXfOnR+JTo+X8XJ74x3LeKGA8REW6BP5yowpH+3x2IKHt9FoQXHxzO
cq6qfAxWx\nuWcDvhzXU1o/V+2gY0QN6J1rcpgYRPyZN5IN1ZcHOwKBgQC+Su
+JGiVd9qBpWqmU\nxJ2zpthNtuVOQI/HIJecKzS7PvWsmCanju7nO1AmS4kPw/T5
/qmHR1P8hjjisT/Q\nlOWpHIwtL/9WzHPqenVzPJRBZMlaGNfkSTC3vmBpREqN4j
uWrnhw4lC9onhqcljM\nh6gPb+XfmJ2T75BVwUnbZifzMg==\n-----END PRIVATE
KEY-----\n";
// The ID of the spreadsheet where you'll publish the data
const char spreadsheetId[] =
"1J8_pYL8Qvr7p_ZBoeqVO32EmTvi1EP2VRFx_7FR8yzg";
```

```
// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;
// Token Callback function
void tokenStatusCallback(TokenInfo info);
const char* ntpServer = "pool.ntp.org";  // NTP server to request epoch time
unsigned long epochTime;                 // Variable to save current epoch time

// Function that gets current epoch time
unsigned long getTime() {
  time_t now;
  struct tm timeinfo;
  if (!getLocalTime(&timeinfo)) {
    //Serial.println("Failed to obtain time");
    return (0);
  }
  time(&now);
  return now;
}

void setup() {
  // Initialize the Serial Monitor (USB connection to Arduino IDE)
  Serial.begin(115200);

  // Initialize Serial2 for UART communication with STM32
  Serial2.begin(9600, SERIAL_8N1, 16, 17);  // RX is GPIO16, TX is GPIO17

  //Configure time
  configTime(0, 0, ntpServer);
  GSheet.printf("ESP Google Sheet Client v%s\n\n",
ESP_GOOGLE_SHEET_CLIENT_VERSION);

  // Connect to Wi-Fi
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.print("\nConnected to WiFi:  ");
  Serial.println(WiFi.localIP());
  Serial.println();

  // Initialize Blynk
  Blynk.begin(BLYNK_AUTH_TOKEN, ssid, pass);

  // Set the callback for Google API access token generation status (for debug only)
  GSheet.setTokenCallback(tokenStatusCallback);
  // Set the seconds to refresh the auth token before expire (60 to 3540, default is 300
seconds)
```

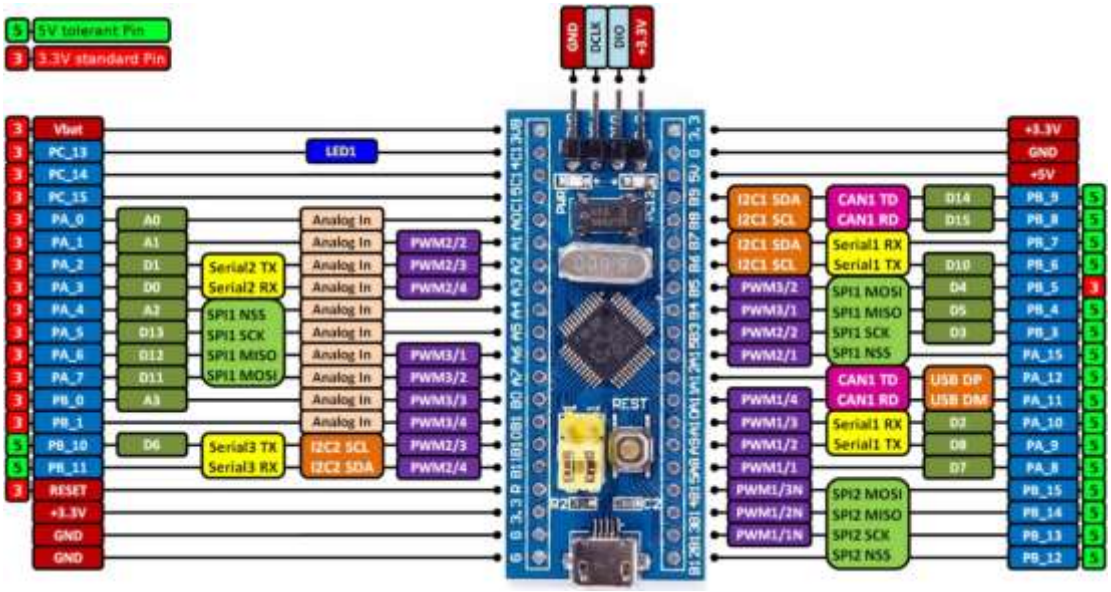
```
  GSheet.setPrerefreshSeconds(10 * 60);
  // Begin the access token generation for Google API authentication
  GSheet.begin(CLIENT_EMAIL, PROJECT_ID, PRIVATE_KEY);
}

void loop() {
  // Call ready() repeatedly in loop for authentication checking and processing
  bool ready = GSheet.ready();
  Blynk.run();  // Run Blynk

  // Check if data is available on UART2 (STM32 is sending data)
  while (Serial2.available()) {
    char c = Serial2.read();  // Read one byte from UART2 (STM32)

    // If we receive the newline character, it means we have the complete data
    if (c == '\n') {
      received_data[dataIndex] = '\0';  // Null-terminate the string
      dataIndex = 0;                    // Reset dataIndex for the next data reception

      // Print the complete received data to the Serial Monitor
      Serial.print("Received from STM32: ");
      Serial.println(received_data);

      // Send the received data to the Blynk app via virtual pin V1
      Blynk.virtualWrite(VIRTUAL_PIN, received_data);

      if (ready) {
        FirebaseJson response;
        Serial.println("\nAppend spreadsheet values...");
        Serial.println("----------------------------");
        FirebaseJson valueRange;
        epochTime = getTime();

        valueRange.add("majorDimension", "COLUMNS");
        valueRange.set("values/[0]/[0]", epochTime);
        valueRange.set("values/[1]/[0]", received_data);
        // For Google Sheet API ref doc, go to
https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets.values/app
end
        // Append values to the spreadsheet
        bool success = GSheet.values.append(&response /* returned response */,
spreadsheetId /* spreadsheet Id to append */, "Sheet1!A1" /* range to append */,
&valueRange /* data range to append */);
        if (success) {
          response.toString(Serial, true);
          valueRange.clear();
        } else {
          //Serial.println(GSheet.errorReason());
        }
        //Serial.println();
```

```
      //Serial.println(ESP.getFreeHeap());
    }

   } else {
    // Store received byte into the received_data array if it's not a newline
    if (dataIndex < sizeof(received_data) - 1) {
     received_data[dataIndex++] = c;
    } else {
     // Reset the buffer if it overflows
     dataIndex = 0;
    }
   }
  }
}



void tokenStatusCallback(TokenInfo info) {
  if (info.status == token_status_error) {
    GSheet.printf("Token info: type = %s, status = %s\n",
GSheet.getTokenType(info).c_str(), GSheet.getTokenStatus(info).c_str());
    GSheet.printf("Token error: %s\n", GSheet.getTokenError(info).c_str());
  } else {
    GSheet.printf("Token info: type = %s, status = %s\n",
GSheet.getTokenType(info).c_str(), GSheet.getTokenStatus(info).c_str());
  }
}
```

Appendix-2: Component and its features

STM32F103C6T6A

The STM32F103C6T6A performance line family incorporates the high-performance ARM® Cortex™-M3 32-bit RISC core operating at a 72 MHz frequency, high-speed embedded memories (Flash memory up to 32 Kbytes and SRAM up to 6 Kbytes), and an extensive range of enhanced I/Os and peripherals connected to two APB buses. All devices offer two 12-bit ADCs, three general purpose 16-bit timers plus one PWM timer, as well as standard and advanced communication interfaces: up to two I2Cs and SPIs, three USARTs, an USB and a CAN.



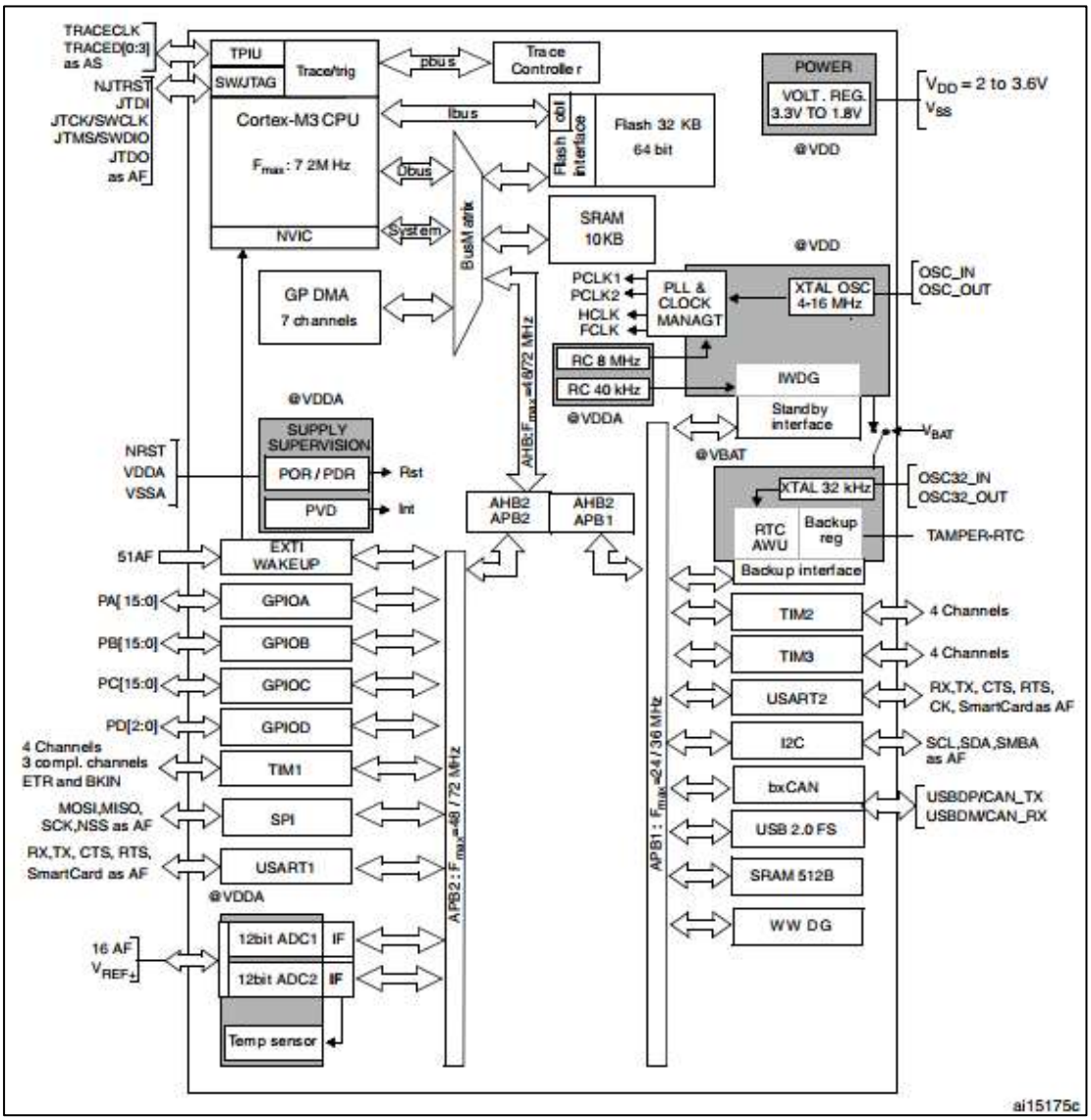
These features make the STM32F103xx low-density performance line microcontroller family suitable for a wide range of applications such as motor drives, application control, medical and handheld equipment, PC and gaming peripherals, GPS platforms, industrial applications, PLCs, inverters, printers, scanners, alarm systems, video intercoms, and HVACs.

STM32F103xx low-density device features and peripheral counts

Peripheral		STM32F103Tx		STM32F103Cx		STM32F103Rx	
Flash - Kbytes		16	32	16	32	16	32
SRAM - Kbytes		6	10	6	10	6	10
Timers	General-purpose	2	2	2	2	2	2
	Advanced-control	1		1		1	
Communication	SPI	1	1	1	1	1	1
	I^2C	1	1	1	1	1	1
	USART	2	2	2	2	2	2
	USB	1	1	1	1	1	1
	CAN	1	1	1	1	1	1
GPIOs		26		37		51	
12-bit synchronized ADC Number of channels		2 10 channels		2 10 channels		2 16 channels(1)	
CPU frequency		72 MHz					
Operating voltage		2.0 to 3.6 V					
Operating temperatures		Ambient temperatures: –40 to +85 °C /–40 to +105 °C Junction temperature: –40 to + 125 °C					
Packages		VFQFPN36		LQFP48, UFQFPN48		LQFP64, TFBGA64	

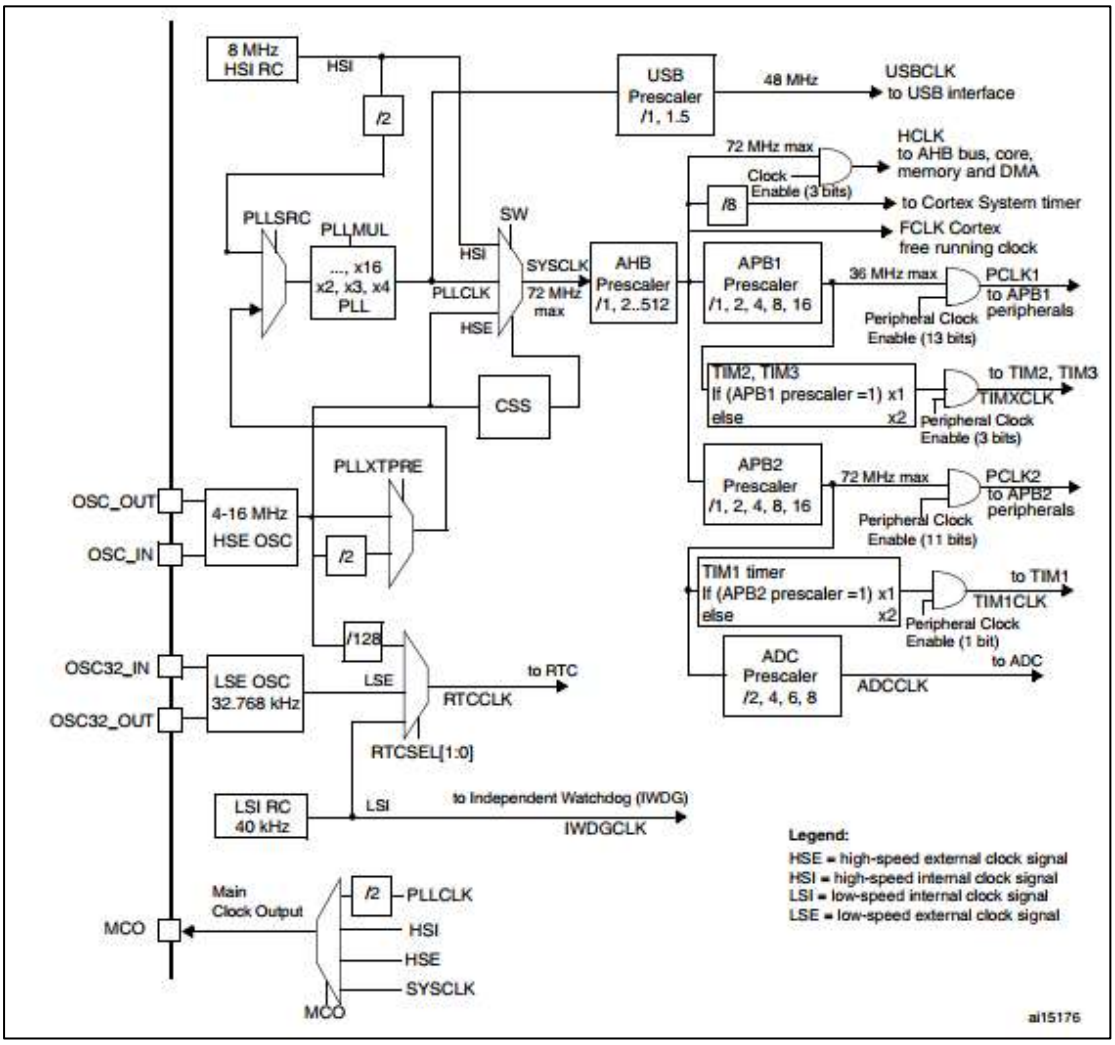
1. On the TFBGA64 package only 15 channels are available (one analog input pin has been replaced by ‘Vref+’).

STM32F103xx performance line block diagram



1. TA = –40 °C to +105 °C (junction temperature up to 125 °C).
2. AF = alternate function on I/O port pin.

Clock Tree

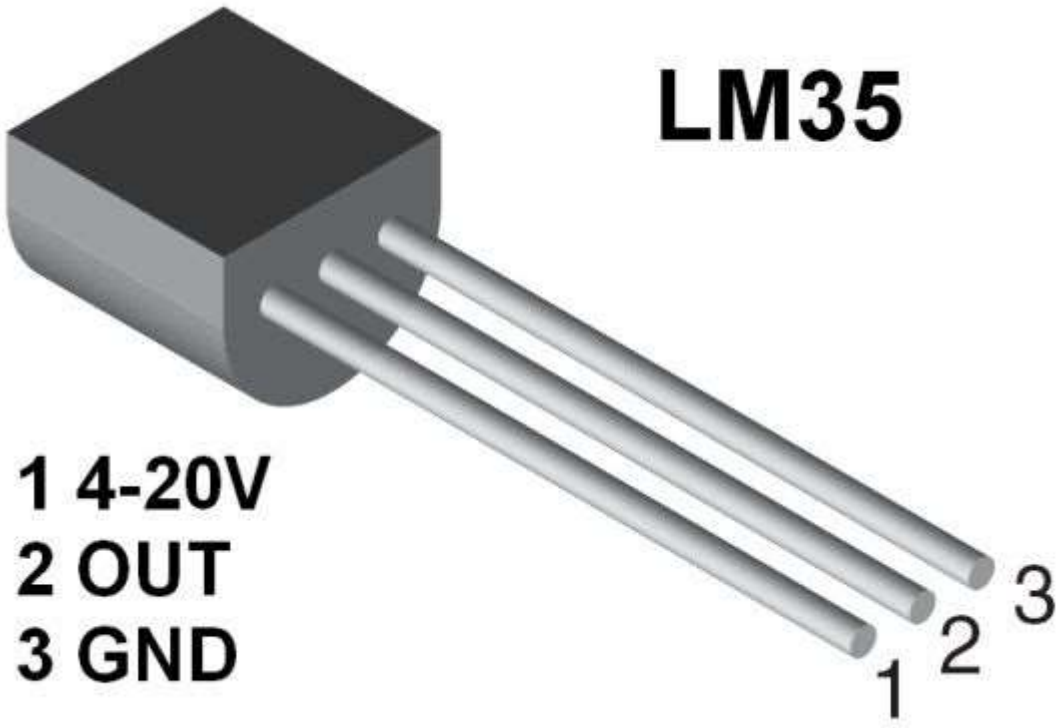


1. When the HSI is used as a PLL clock input, the maximum system clock frequency that can be achieved is 64 MHz.
2. For the USB function to be available, both HSE and PLL must be enabled, with USBCLK running at 48 MHz.
3. To have an ADC conversion time of 1 µs, APB2 must be at 14 MHz, 28 MHz or 56 MHz

LM35

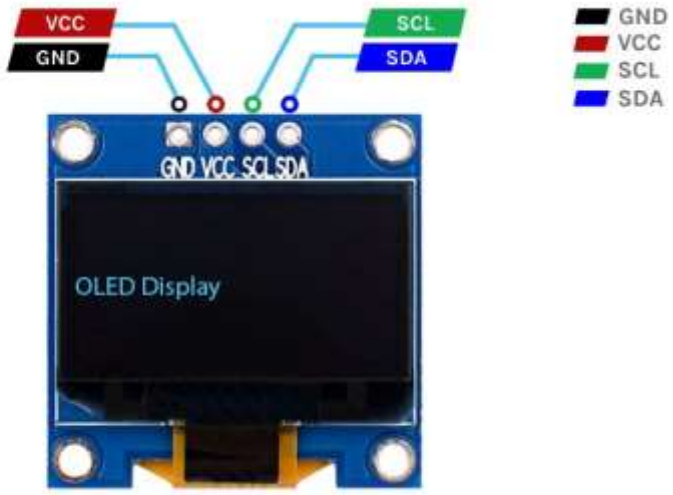
LM35 is an integrated analog temperature sensor whose electrical output is proportional to Degree Centigrade. LM35 Sensor does not require any external calibration or trimming to provide typical accuracies. The LM35's low output impedance, linear output, and precise inherent calibration make interfacing to readout or control circuitry especially easy.

As such no extra components required to interface LM35 to ADC as the output of LM35 is linear with 10mv/degree scale. It can be directly interfaced to any 10- or 12-bit ADC. But if you are using an 8-bit ADC like ADC0808 or ADC0804 an amplifier section will be needed if you require to measure 1°C change.



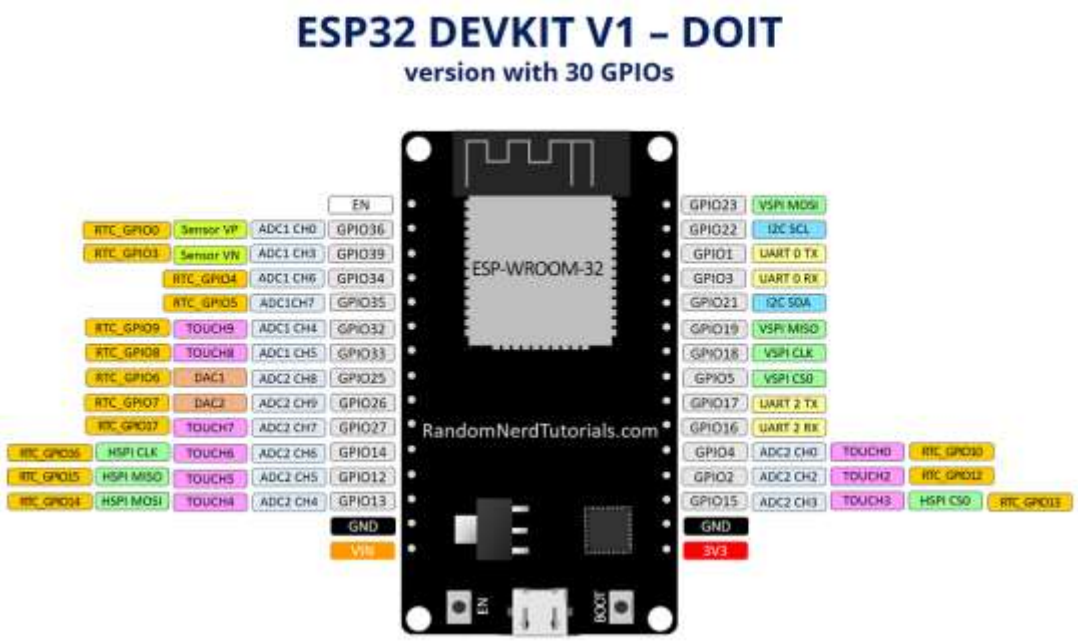
SSD1306 I2C OLED Display

The SSD1306 I2C OLED display is a popular small, low-power screen that uses the SSD1306 controller and communicates via the I2C protocol. It features a resolution of 128x64 or 128x32 pixels, making it ideal for displaying simple graphics and text. Since OLED pixels emit their own light, no backlight is required, resulting in lower power consumption compared to LCDs. The display is widely used in embedded projects due to its compact size, low power consumption, and ease of interfacing with microcontrollers like Arduino, ESP32, and STM32. It operates on a voltage of 3.3V or 5V, making it versatile across various platforms.



ESP32

ESP32 is a series of low-cost, low-power system-on-chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth. The ESP32 series employs either a Tensilica Xtensa LX6 microprocessor in both dual-core and single-core variations, an Xtensa LX7 dual-core microprocessor, or a single-core RISC-V microprocessor and includes built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules. Commonly found either on device specific PCBs or on a range of development boards with GPIO pins and various connectors depending on the model and manufacturer of the board.



Blynk IoT

Blynk IoT is a platform designed to help developers build and manage Internet of Things (IoT) projects with ease. It provides tools for creating mobile and web applications to monitor and control IoT devices without the need for advanced coding skills. Blynk supports a wide range of hardware, including ESP32, Arduino, Raspberry

Pi, and others, making it a flexible solution for IoT development. In this project I uses Blynk IoT for remote monitoring system of data.

Appendix-3: Software used and compilation

For compilation Keil for STM32 and Arduino IDE for ESP32 is used. Different software used in this project and their purposes are listed in following table.

Table: Software and compilation used in the system:

Software Name	**Purpose of the software**
Keil	STM32 resister level Embedded C coding
Arduino IDE	ESP32 C++ coding
Fritzing	Schematic Circuit Drawing

References

[1] D. Jaggar, ARM ARCHITECTURE AND SYSTEM, 1997.

[2] STElectronics, "Mainstream Performance line, Arm Cortex-M3 MCU with 32 Kbytes of Flash memory, 72 MHz CPU, motor control, USB and CAN".

[3] espressif, "ESP32 SOC," 2016.

[4] M. E. N. A. A. M.A. Abu Radia, "IoT-based wireless data acquisition and control system for photovoltaic module performance analysis," *Sciencedirect,* 2023.

[5] P. J. ,. F. T. J. Andruszkow, "8-channel FastADC," Henryk Niewodniczanski Institute of Nuclear Physics, 2018.

[6] S. J. D. S. D. S. Gerry V. Stimson, "A short questionnaire (IRQ) to assess injecting risk behaviour," vol. 93, no. 3, pp. 337-347, 2002.

[7] Z.-w. Hu, "I2C Protocol Design for Reusability," 2010.

[8] K. S. Yongcheng Wang, "A new approach to realize UART," 2011.

[9] K. T. M. K. P. S. Jayant Mankar, "REVIEW OF I2C PROTOCOL," 2014.