ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

LOGIC DESIGN PROJECT

EXPANSION SHIELD FOR STM32 NUCLEO INTERFACING WITH RAIN SENSOR AND LCD 1602 AND DHT20 SENSOR VIA I2C COMMUNICATION PROTOCOL COMMUNICATION

Giảng viên hướng dẫn:	T.S Lê Trọng Nhân	
	Nguyễn Thành Lộc	
Sinh viên thực hiện:	Ngô Quang Tùng	2213869
	Trương Phan Hoàng Vỹ	2214062

Thành phố Hồ Chí Minh, 12/2024

Contents

ACKNOWLEDGEMENT

First and foremost, we would express our most profound appreciation to our thesis supervisors, Dr. Lê Trọng Nhân. He has been there, providing his heartfelt support and guidance at all times. He has given us invaluable guidance, inspiration, and suggestions in our quests for knowledge during our university time. Without his assistance and dedicated involvement in every step throughout the process, this thesis would have never been accomplished.

We sincerely thank the teachers who occupy the Faculty of Computer Science and Engineering in particular and the Ho Chi Minh City University of Technology in general, who have constantly been imparting knowledge in the past four years. Their support, encouragement, and credible ideas have been great contributors to the completion of the thesis.

Last but not least, it would be inappropriate if we omit to thank our friends and family. Our late parents' unconditional love and blessings, the care of friends and acquaintances who never let things get dull, have all made a tremendous contribution in helping us reach this stage in our life. We thank them for putting up with us in difficult moments where we felt stumped and for goading us on to reach for our passions.

Finally, we would like to wish you good health and success in your noble life.

ABSTRACT

The project consists of eight chapters to explain our work clearly, and the last chapter is dedicated to the conclusion from our perspective. Our final result allows the system to measure the temperature and humidity of the environment where the system is placed. The information from this examination includes the temperature in degrees Celsius and the humidity in percentage. These values are then displayed on the LCD 1602. Additionally, we created the UART interface for the user to control the behavior of the system.

Keywords for our work: I^2C & UART connection protocol, LCD 1602 & DHT20 Sensor, STM Nucleo Development Board, STM32F103RBT6 MCU.

Chapter 1

INTRODUCTION

In this project, we aim to design and implement an extension shield for the STM Nucleo Development Board. The main functionality of this shield is to sequentially read the real-time temperature and humidity using the DHT-20 sensor. Meanwhile, the results will be displayed on a 16×2 LCD.

Initially, the design stage requires us to work with the Altium Designer application. With the assistance from the instructor, we were able to print out the circuit board and solder all necessary components such as the DHT-20 sensor, LCD 16×2, as well as resistors, capacitors, etc., to ensure the proper operation of the shield.

Our main components use the I2C communication protocol, and the STM32-F103RBT6 microcontroller on the STM Nucleo board supports this type of communication. Additionally, the UART protocol is needed to transmit control commands from the PC to the hardware.

Overall, the design step was relatively simple since we had support from the component library and the design patterns provided by the instructor. However, the programming work was much more complicated as we had to handle a new communication protocol and interact with different devices.

Chapter 2

THEORETICAL BASIS

This chapter will present knowledge that contributes to the implementation of our system, including some well-known communication protocols for interacting with peripheral devices, including the LCD and DHT-20 sensor, as well as controlling the operation of the system from the PC.

2.1 Universal Asynchronous Receiver - Transmitter (UART)

2.1.1 Introduction to UART Communication Protocol

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:

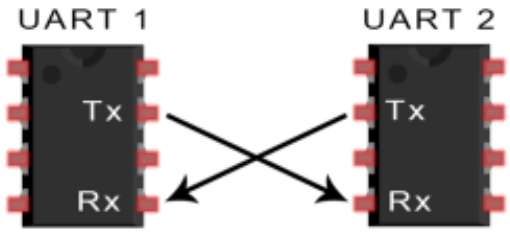

Figure 2.1: UART Communication

UARTs transmit data asynchronously, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the baud rate. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.
Both UARTs must also be configured to transmit and receive the same data packet structure.

Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous ?	Asynchronous
Serial or Parallel ?	Serial
Maximum number of Masters	1
Maximum number of Slaves	1

Table 2.1: UART Communication Properties

2.1.2 UART Protocol Working Principle

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller.

Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin.

The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end.

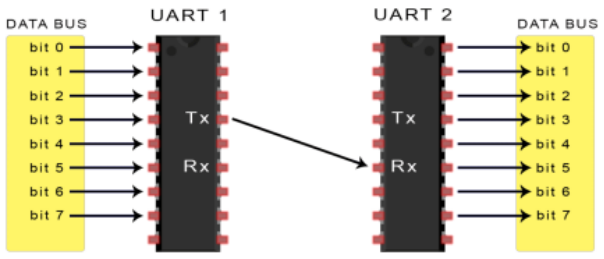

Figure 2.2: Working Principle

UART transmitted data is organized into packets. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional parity bit, and 1 or 2 stop bits

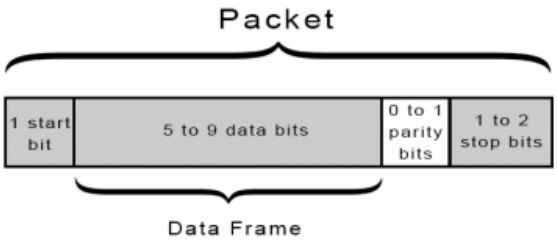

Figure 2.3: Structure of the transmitted data

Start Bit:
The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.
Data Frame:
The data frame contains the actual data being transferred. It can be 5 bits up to 8 bits long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long. In most cases, the data is sent with the least significant bit first.
Parity:
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers.
After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number.
When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.
Stop Bits:
To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

2.1.3 Steps of UART Data Transmission

1. The transmitting UART receives data in parallel from the data bus:

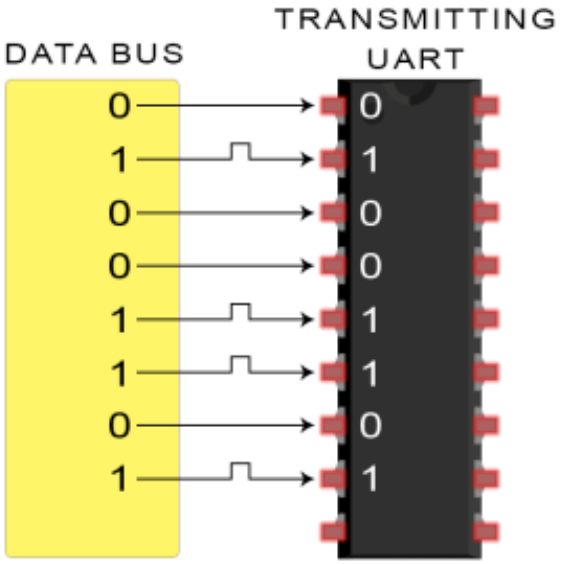

Figure 2.4: Step 1 - UART Data Transmission

2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:

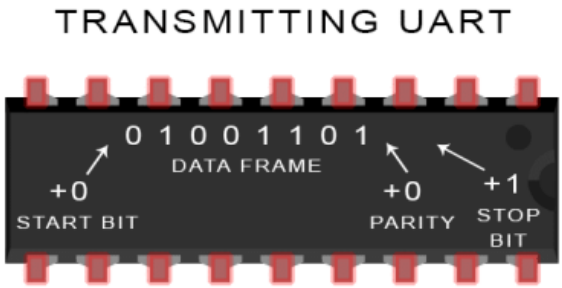

Figure 2.5: Step 2 - UART Data Transmission

3. The transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:

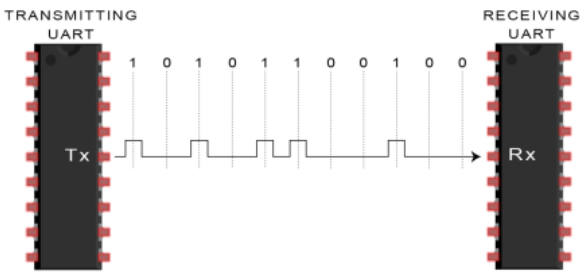

Figure 2.6: Step 3 - UART Data Transmission

4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:

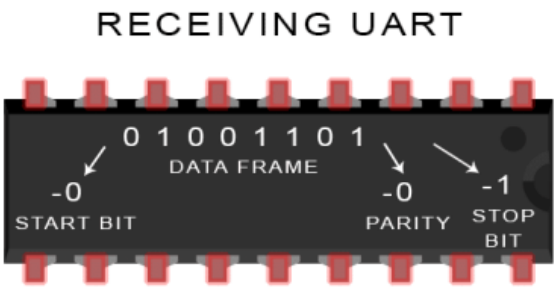

Figure 2.7: Step 4 - UART Data Transmission

5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:

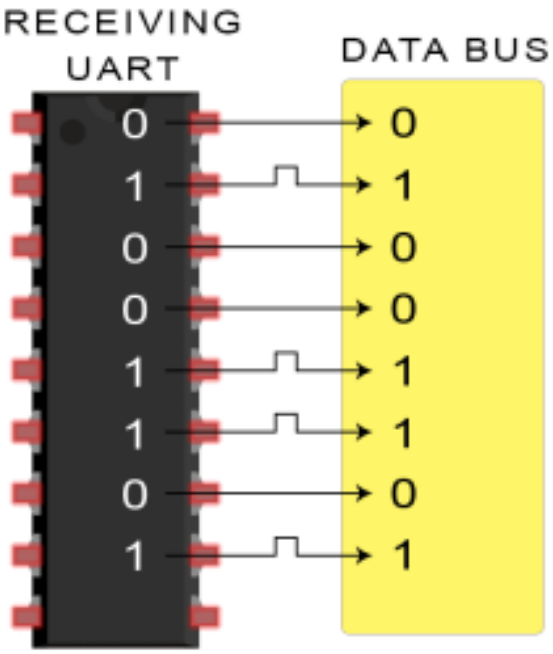

Figure 2.8: Step 5 - UART Data Transmission

2.1.4 Advantages and Disadvantages of UART

Advantages:

- Only uses two wires, no clock signal is required.
- A parity bit is available to allow for error checking.
- The structure of the data packet can be changed as long as both sides are set up for it.
- Well documented and widely used method.

Disadvantages:

- The size of the data frame is limited to a maximum of 9 bits.
- Doesn't support multiple slave or multiple master systems.
- The baud rates of each UART must be within 10% of each other.

2.2 I^2C - Inter Integrated Circuit Communication

2.2.1 Introduction of I^2C Communication Protocol

I^2C is termed as the abbreviated form of **Inter-Integrated Circuit**. This is a type of communication bus which is mainly designed and developed to establish inter-chip communication. This protocol is a bus interface connection that is embedded into multiple devices to set up serial connections.

I^2C is considered to be the most prominent chip-to-chip communication protocol as it holds the features of both UART and SPI. With I^2C, it is feasible to connect multiple slaves to a single master and we can have multiple masters controlling single or multiple slaves. This is really useful when we want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.

I^2C only uses two wires to transmit data between devices and the transmit is synchronous so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

Wires Used	2
Maximum Speed	Standard mode: 100 kbps Fast mode: 400 kbps High-speed mode: 3.4 Mbps Ultra-fast mode: 5Mbps
Synchronous or Asynchronous ?	Synchronous
Serial or Parallel ?	Serial
Maximum number of Masters	Unlimited
Maximum number of Slaves	1008

Table 2.2: I^2C Communication Properties

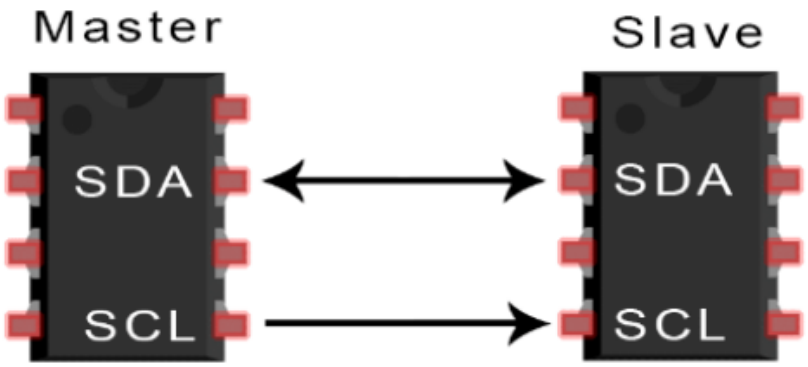

Figure 2.9: Master-Slave Communication in I^2C

2.2.2 I^2C Protocol Working Principle

The working of the I^2C communication protocol happens through open drain lines which are Serial Data (SDA) and Serial Clock (SCL). Initially, both the SDA and SCL lines are pulled high and the bus mainly functions in two modes: Master and Slave.

With I^2C, data is transferred in messages. Messages are broken up into frames of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame.

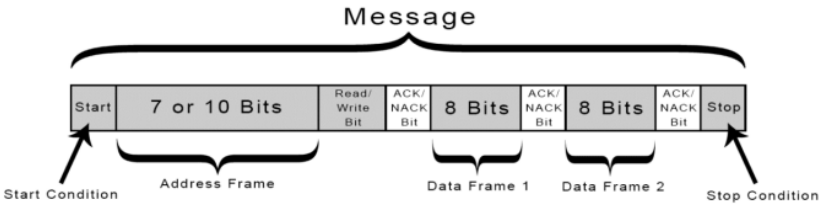

Figure 2.10: The data packet

Every bit which is transferred on the serial data line gets synchronized by a HIGH to LOW pulse for every clock signal that is received on the serial clock. The SDA line will not change when the SCL is in a HIGH state and it is changed only when SCL is in a LOW state. As discussed, SDA and SCL are open-drain lines, they require pull-up resistors to make them high because the devices on the bus are in the active LOW state.

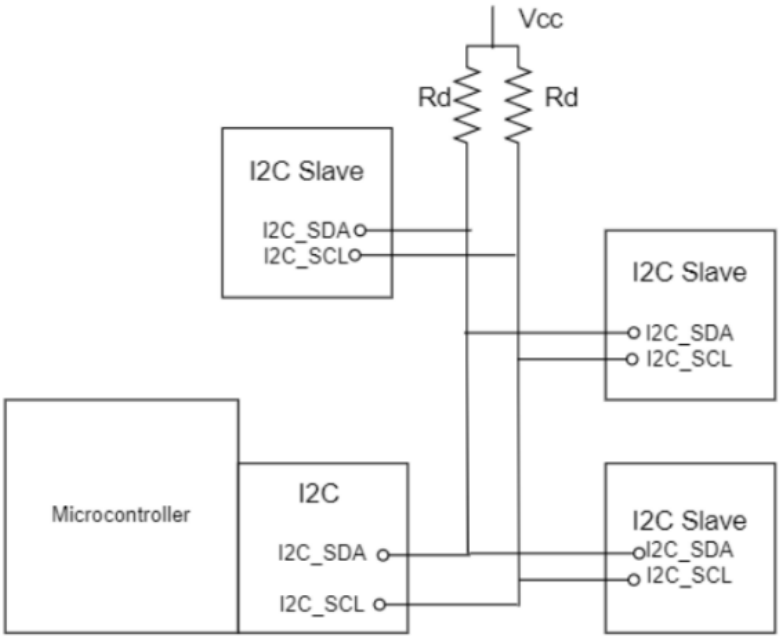

Figure 2.11: I^2C Protocol Architecture

Here, data transfer takes place in the form of packets and each packet consists of 9 bits. The sequence of bits is shown below:

Start & Stop Bits:

For the Start Condition, the SDA line switches from a high voltage level to a low voltage level before the SCL line switches from high to low. Meanwhile, the SDA line switches from a low voltage level to a high voltage level after the SCL line switches from low to high for the Stop Condition.

Address:

This is the immediate frame after the start bit. The master bit transfers the slave's address with which it has to communicate to each slave that was connected to it. Then the slave bit compares its own address with the address of the slave that was sent by the master. When both the addresses match, it transmits a low voltage ACK signal to the master. Whereas when both the addresses do not match, the slave remains idle and the serial data line stays at HIGH.

Read/Write Bit:

When the Read/Write indicates '1', then the master is transmitting data to the slave, whereas when Read/Write indicates '0', then the master is receiving data from the slave signal.

ACK/NACK:

The acknowledge/no-acknowledge bit is the subsequent bit of every frame in a message. When data/address was successfully transmitted, then the ACK signal gets back to the sender from the receiver device.

Data Frame:

When the ACK bit is detected by the master from the slave, it indicates that the first data frame can be transferred. This data frame is of 8-bit length. After the data frame, the ACK/NACK bit detects the successful transmission of the frame. Once the transmission was successful, the next data frame will be ready to transmit.

Once all the data frames are received, the master sends the STOP signal indicating

to stop the transmission. The below I2C protocol timing diagram gives a clear idea of the working of the protocol.

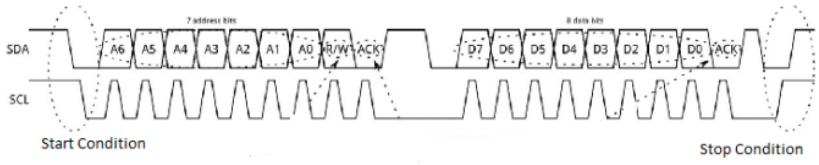

Figure 2.12: Timing Diagram

2.2.3 Steps of I²C Data Transmission

1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level before switching the SCL line from high to low:

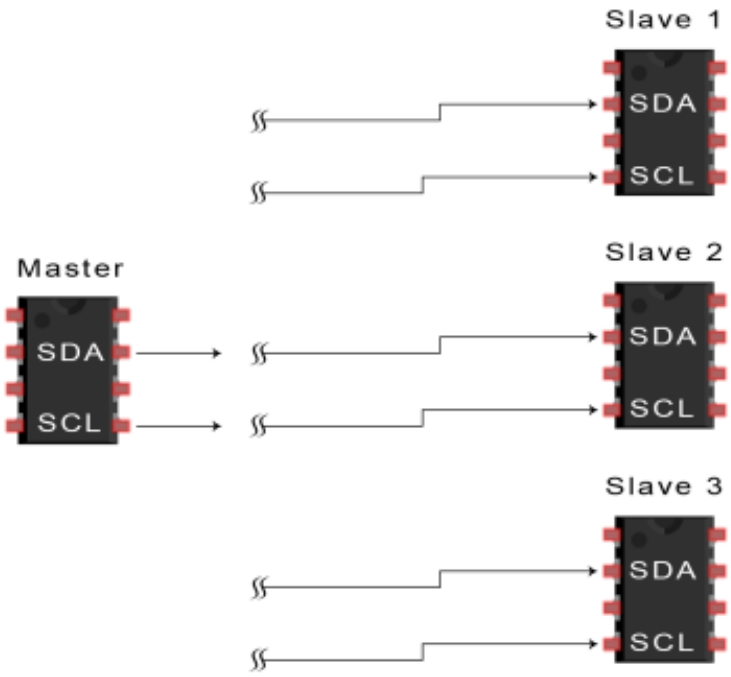

Figure 2.13: Step 1 - I²C Data Transmission

2. The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit:

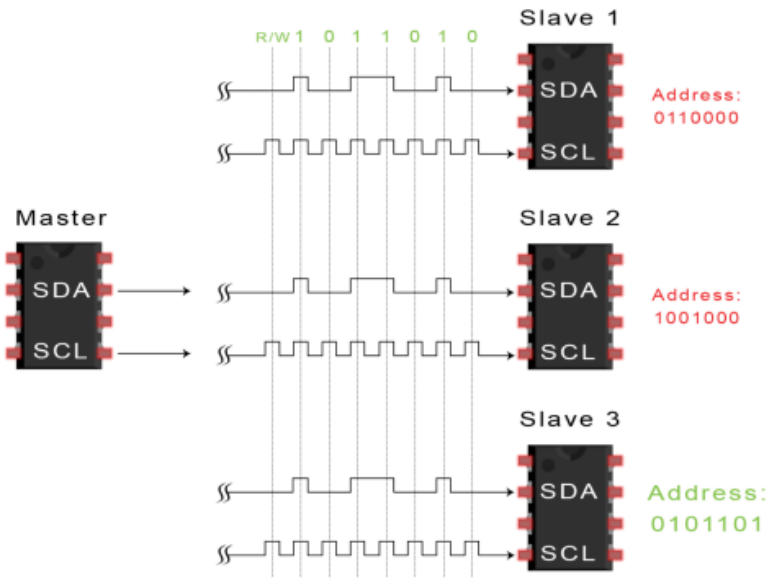

Figure 2.14: Step 2 - I^{2}C Data Transmission

3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.

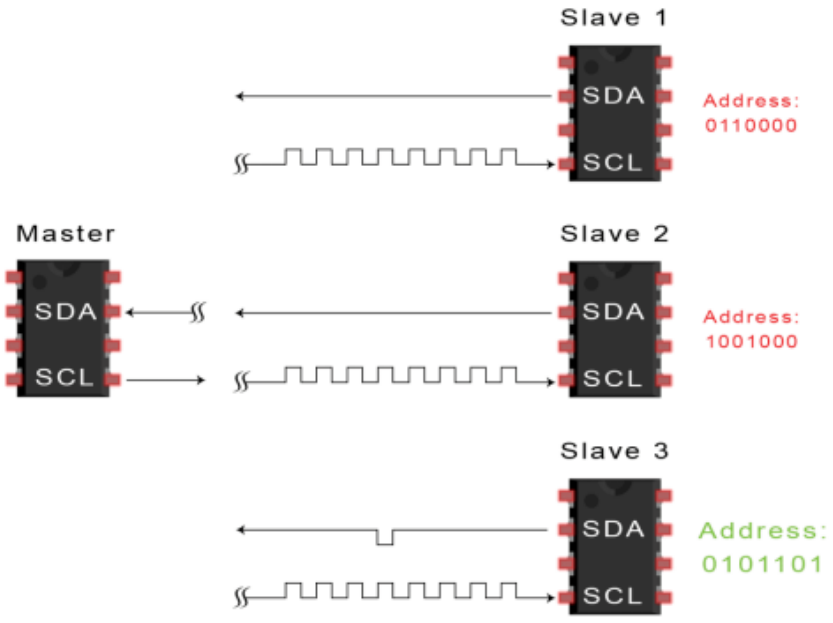

Figure 2.15: Step 3 - I^{2}C Data Transmission

4. The master sends or receives the data frame:

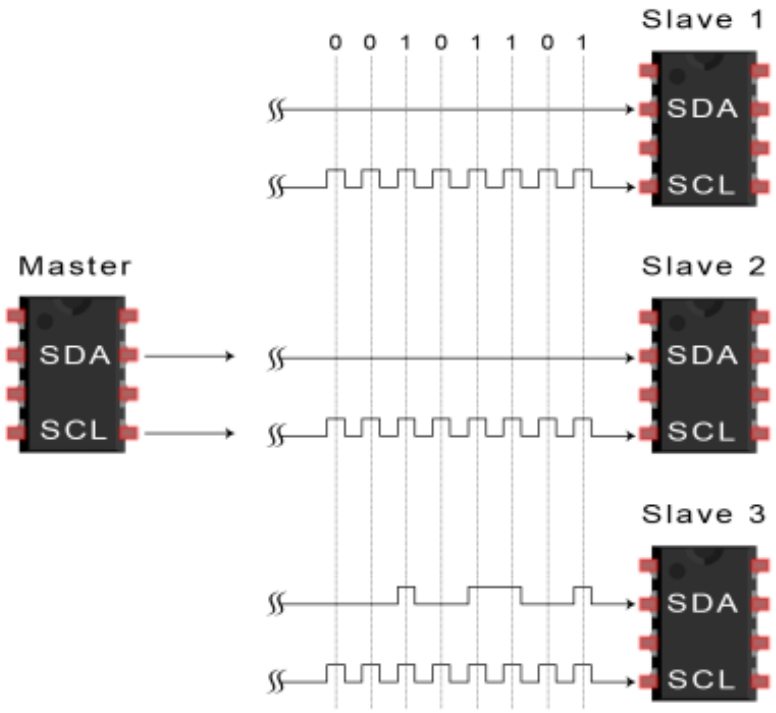

Figure 2.16: Step 4 - I^{2}C Data Transmission

5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:

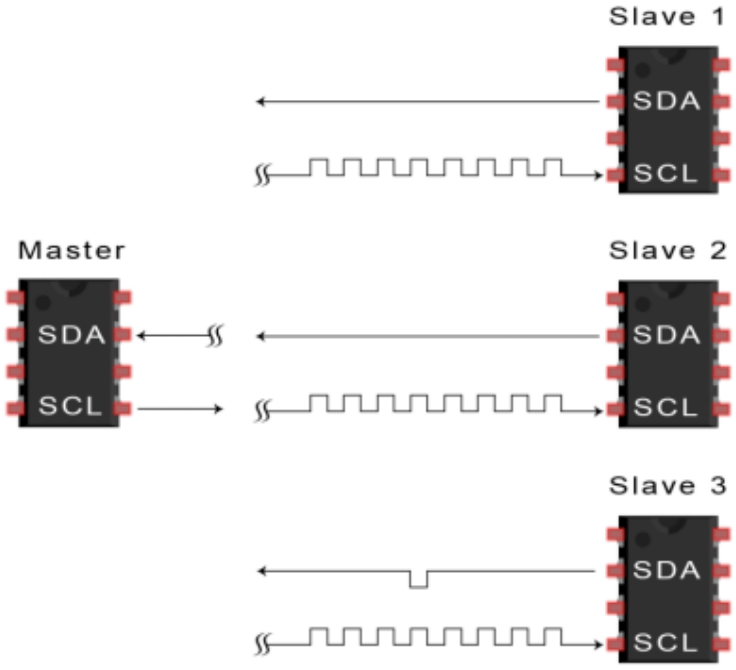

Figure 2.17: Step 5 - I^{2}C Data Transmission

6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:

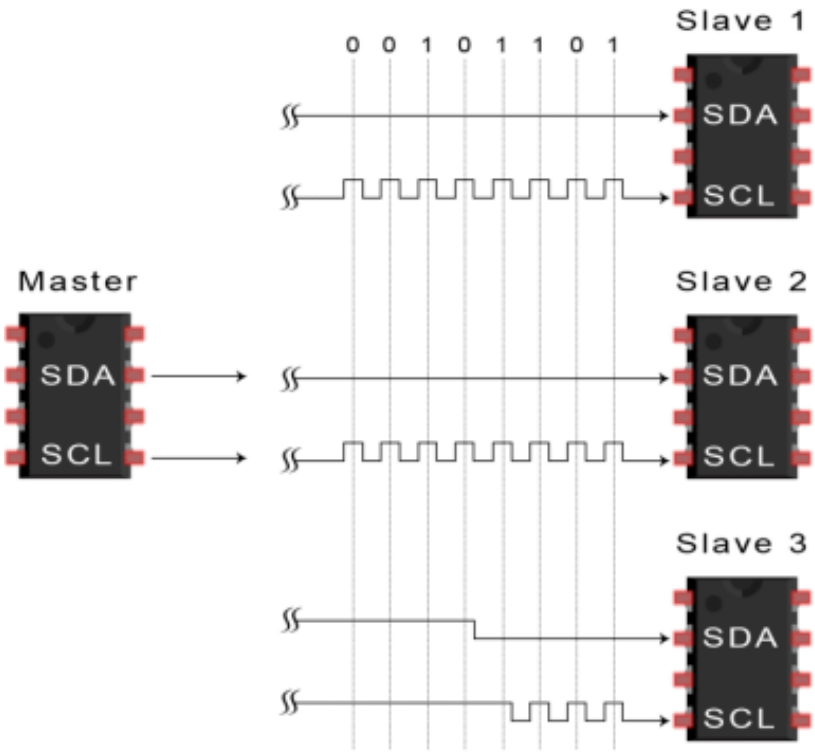

Figure 2.18: Step 6 - I^{2}C Data Transmission

2.2.4 Communication Schemes

Single Master with Multiple Slaves

Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7-bit address, 128 (2^7) unique addresses are available. Using 10-bit addresses is uncommon, but provides 1,024 (2^{10}) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to VCC:

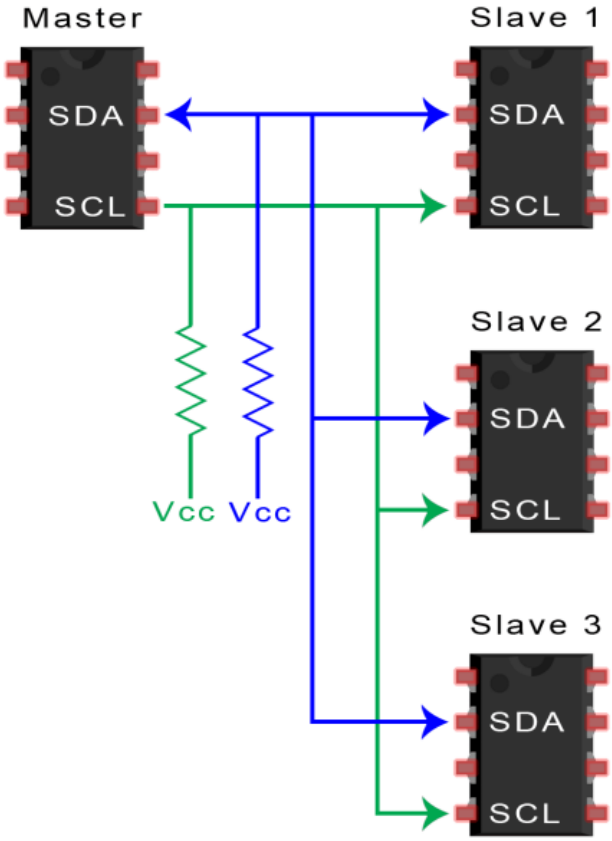

Figure 2.19: Single Master - Multiple Slaves

Multiple Masters with Multiple Slaves

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line.

To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it's safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to VCC:

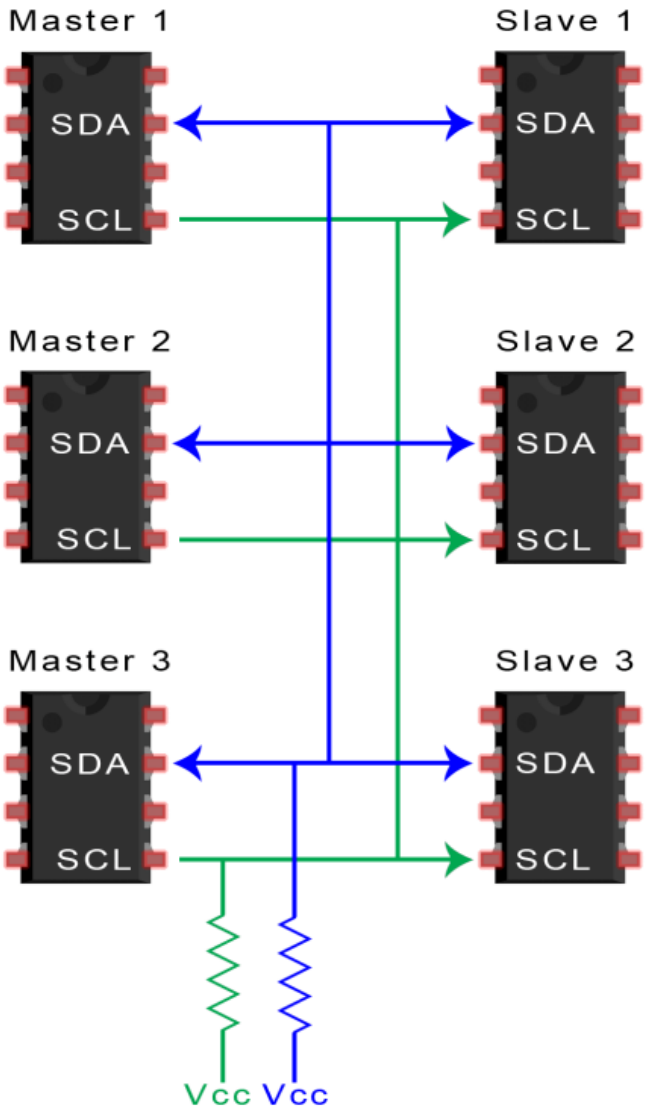

Figure 2.20: Multiple Masters - Multiple Slaves

2.2.5 Advantages and Disadvantages of I^2C

Advantages:

- Even though there are multiple devices on the bus, the signals required are less
- It has the ability to support many masters
- It has enhanced adaptability where the requirements of various slave devices can be achieved
- It has multiple-master and multiple-slave flexibility

Disadvantages:

- I^2C protocol is designed with an open-drain configuration where it limits the speed and also pull-up resistors are used which further lessens the communication speed
- The size of the data frame is limited to 8 bits

Chapter 3

DEVICES AND COMPONENTS

The main content of the chapter is to provide a brief overview of the STM32 Nucleo-64 development board. Beside, the working mechanism and functionality of each component used in the project are also presented.

3.1 Introduction of STM32 Nucleo-64 development board with STM32-F103RB MCU

The STM32 Nucleo-64 board provides an affordable and flexible way for users to try out new concepts and build prototypes by choosing from the various combinations of performance and power consumption features, provided by the STM32 microcontroller. For the compatible boards, the external SMPS significantly reduces power consumption in Run mode.

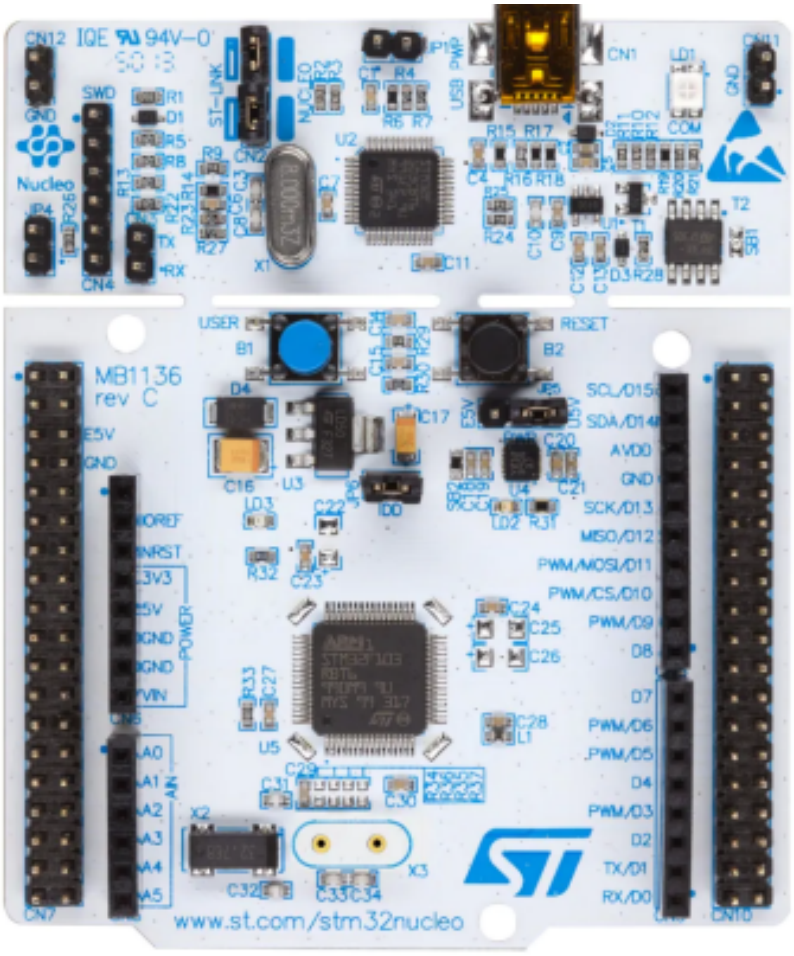

Figure 3.1: STM32 Necleo

The ARDUINO Uno V3 connectivity support and the ST morpho headers allow the easy expansion of the functionality of the STM32 Nucleo open development platform with a wide choice of specialized shields. The STM32 Nucleo-64 board does not require any separate probe as it integrates the ST-LINK debugger/programmer. The STM32 Nucleo-64 board comes with the STM32 comprehensive free software libraries and examples available with the STM32Cube MCU Package. Here are some common features of the board: **Here are some common features of the board:**

- STM32 microcontroller in LQFP64 or LQFP48 package
- 1 user LED shared with ARDUINO
- 1 user and 1 reset push-buttons

- 32.768 kHz crystal oscillator
- Board connectors: ARDUINO Uno V3 expansion connector and ST morpho extension pin headers for full access to all STM32 I/Os
- Flexible power-supply options: ST-LINK USB VBUS or external sources
- On-board ST-LINK debugger/programmer with USB re-enumeration capability: mass storage, Virtual COM port, and debug port
- Comprehensive free software libraries and examples available with the STM32Cube MCU Package
- Support of a wide choice of Integrated Development Environments (IDEs) including IAR Embedded Workbench, MDK-ARM, and STM32CubeIDE

Board-specific features:

- External SMPS to generate V_{core} logic supply
- 24MHz or 48MHz HSE
- Board connectors:

- External SMPS experimentation dedicated connector
- Micro-B or Mini-B USB connector for the ST-LINK
- MIPI debug connector

3.2 PCF8574/74A

3.2.1 Description

The PCF8574/74A provides general-purpose remote I/O expansion via the two-wire bidirectional I^2C-bus (serial clock (SCL), serial data (SDA)), with operating supply voltage 2.5 V to 6 V with non-overvoltage tolerant I/O held to V_{DD} with 100 μA current source.

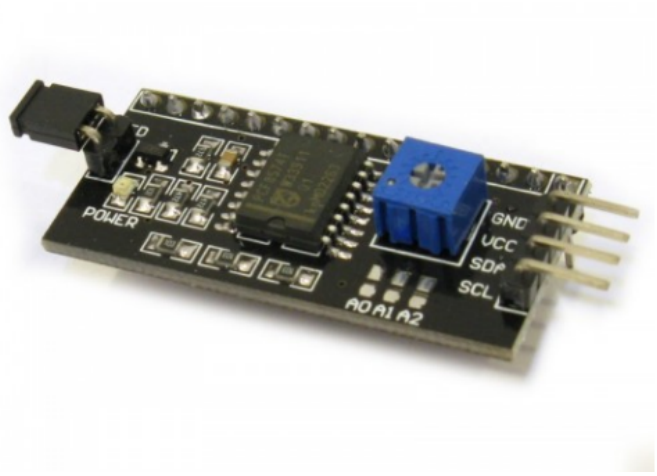

Figure 3.2: PCF8574/74A

The PCF8574 and PCF8574A are identical, except for the different fixed portion of the slave address. The three hardware address pins allow eight of each device to be on the same I^{2}C-bus, so there can be up to 16 of these I/O expanders PCF8574/74A together on the same I^{2}C-bus, supporting up to 128 I/Os (for example, 128 LEDs).

3.2.2 Block Diagram

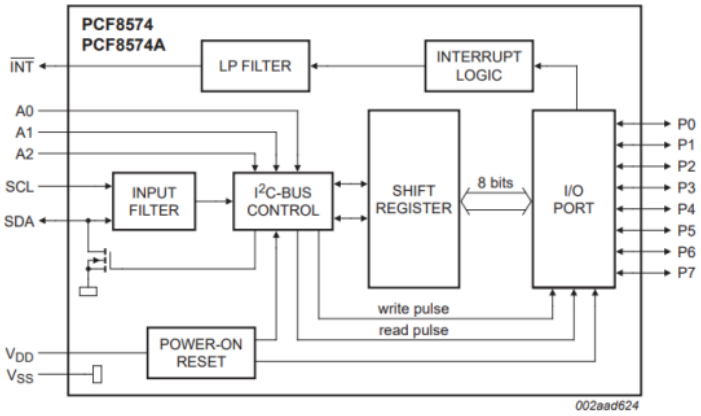

Figure 3.3: PCF8574/74A Block Diagram

3.2.3 Usage

Write to the port (Output Mode) The master (microcontroller) sends the START condition and slave address setting the last bit of the address byte to logic 0 for the write mode. The PCF8574/74A acknowledges and the master then sends the data byte for P7 to P0 to the port register.

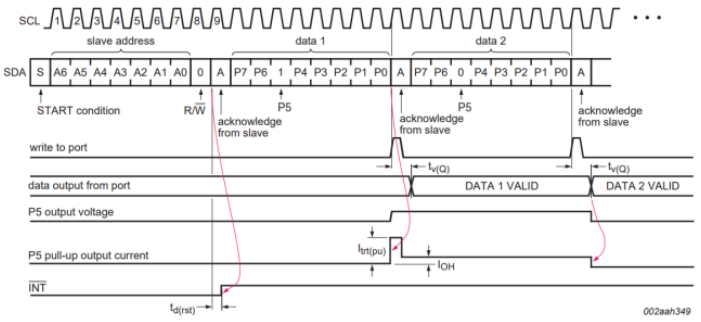

Figure 3.4: PCF8574/74A Write Mode

Simple code WRITE mode: `<S> <slave address + write> <ACK> <data out> <ACK> <data out> <ACK> ...<data out> <ACK> <P>` **Reading from a port (Input mode)**
The port must have been previously written to logic 1, which is the condition after power-on reset. To enter the Read mode, the master (microcontroller) addresses the slave device and sets the last bit of the address byte to logic 1 (address byte read). The slave will acknowledge and then send the data byte to the master. The master will NACK and then send the STOP condition or ACK and read the input register again.

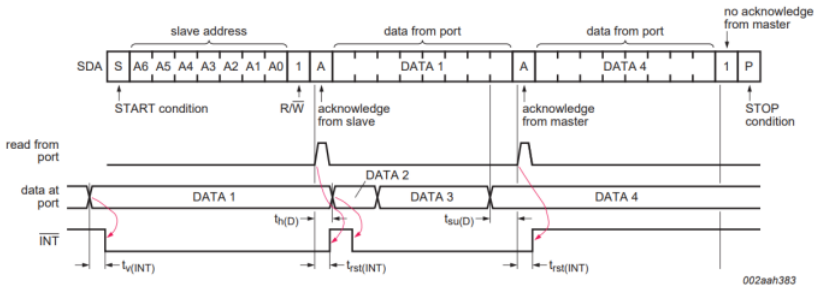

Figure 3.5: PCF8574/74A Read mode

Simple code for READ mode: `<S> <slave address + read> <ACK> <data in> <ACK> ... <data in> <ACK> <data in> <NACK> <P>`

I²C Address:

The PCF8574 and PCF8574A are functionally the same, but have a different fixed portion (A6 to A3) of the slave address. This allows 8 of the PCF8574 and 8 of the PCF8574A to be on the same I²C-bus without address conflict. Normally, we can interface with 16 LCDs at the same time. (We can also modify the fixed portion (A6 to A3) to extend communication up to 128 LCDs). The PCF8574 and PCF8574A are functionally the same, but have a different fixed portion (A6 to A3) of the slave address. This allows 8 of the PCF8574 and 8 of the PCF8574A to be on the same I²C-bus without address conflict. Normally, we can interface with 16 LCDs at the same time. (We can also modify the fixed portion (A6 to A3) to extend communication up to 128 LCDs).

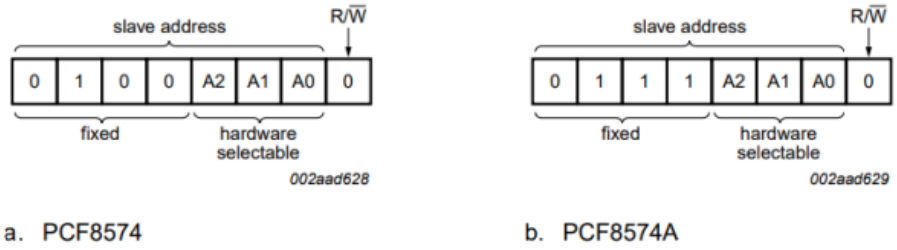

Figure 3.6: PCF8574/74A address

I²C Address Map

The I²C address range for PCF8574 is from 0x20 to 0x27, with the default value in the retail being 0x27 $[A2, A1, A0] = [1, 1, 1]$.

Because there is a R\W bit, we need to shift the address left by 1 when communicating.

Pin connectivity			Address of PCF8574								Address byte value		7-bit hexadecimal address_ without R/W
A2	A1	A0	A6	A5	A4	A3	A2	A1	A0	R/W	Write	Read	
V_{SS}	V_{SS}	V_{SS}	0	1	0	0	0	0	0	-	40h	41h	20h
V_{SS}	V_{SS}	V_{DD}	0	1	0	0	0	0	1	-	42h	43h	21h
V_{SS}	V_{DD}	V_{SS}	0	1	0	0	0	1	0	-	44h	45h	22h
V_{SS}	V_{DD}	V_{DD}	0	1	0	0	0	1	1	-	46h	47h	23h
V_{DD}	V_{SS}	V_{SS}	0	1	0	0	1	0	0	-	48h	49h	24h
V_{DD}	V_{SS}	V_{DD}	0	1	0	0	1	0	1	-	4Ah	4Bh	25h
V_{DD}	V_{DD}	V_{SS}	0	1	0	0	1	1	0	-	4Ch	4Dh	26h
V_{DD}	V_{DD}	V_{DD}	0	1	0	0	1	1	1	-	4Eh	4Fh	27h

Figure 3.7: PCF8574 Address Range

The I^2C address range for PCF8574A is from 0x38 to 0x3F, with the default value in retail being 0x3F $[A_2, A_1, A_0] = [1, 1, 1]$.

Pin connectivity			Address of PCF8574A								Address byte value		7-bit hexadecimal address_ without R/W
A2	A1	A0	A6	A5	A4	A3	A2	A1	A0	R/W	Write	Read	
V_{SS}	V_{SS}	V_{SS}	0	1	1	1	0	0	0	-	70h	71h	38h
V_{SS}	V_{SS}	V_{DD}	0	1	1	1	0	0	1	-	72h	73h	39h
V_{SS}	V_{DD}	V_{SS}	0	1	1	1	0	1	0	-	74h	75h	3Ah
V_{SS}	V_{DD}	V_{DD}	0	1	1	1	0	1	1	-	76h	77h	3Bh
V_{DD}	V_{SS}	V_{SS}	0	1	1	1	1	0	0	-	78h	79h	3Ch
V_{DD}	V_{SS}	V_{DD}	0	1	1	1	1	0	1	-	7Ah	7Bh	3Dh
V_{DD}	V_{DD}	V_{SS}	0	1	1	1	1	1	0	-	7Ch	7Dh	3Eh
V_{DD}	V_{DD}	V_{DD}	0	1	1	1	1	1	1	-	7Eh	7Fh	3Fh

Figure 3.8: PCF8574A Address Range

3.3 LCD 1602

3.3.1 Description

LCD1602 (or LCD16×2) is a device which can show up to 16 characters per row, and each character is built with a 5×8 pixel box. To operate the LCD, we need to use a voltage between 0.3V and 7.0V and a current of 1mA. There are two display modes: 4-bit and 8-bit. Additionally, the display can be obtained with a Blue or Green backlight, and it can also display customized characters.

3.3.2 Usage

PIN NO.	SYMBOL	DESCRIPTION	FUNCTION
1	VSS	GROUND	0V (GND)
2	VCC	POWER SUPPLY FOR LOGIC CIRCUIT	+5V
3	VEE	LCD CONTRAST ADJUSTMENT	
4	RS	INSTRUCTION/DATA REGISTER SELECTION	RS = 0 : INSTRUCTION REGISTER RS = 1 : DATA REGISTER
5	R/W	READ/WRITE SELECTION	R/W = 0 : REGISTER WRITE R/W = 1 : REGISTER READ
6	E	ENABLE SIGNAL	
7	DB0	DATA INPUT/OUTPUT LINES	8 BIT: DB0-DB7
8	DB1		
9	DB2		
10	DB3		
11	DB4		
12	DB5		
13	DB6		
14	DB7		
15	LED+	SUPPLY VOLTAGE FOR LED+	+5V
16	LED-	SUPPLY VOLTAGE FOR LED-	0V

Figure 3.9: Pin function of LCD 16×2

Clearer Specification of Each Pin

- Pin 1 (Ground/Source Pin): This is a GND pin of the display, used to connect the GND terminal of the microcontroller unit or power source.
- Pin 2 (VCC/Source Pin): This is the voltage supply pin of the display, used to connect the supply pin of the power source.
- Pin 3 (V0/VEE/Control Pin): This pin regulates the difference of the display, used to connect a changeable POT that can supply 0 to 5V.
- Pin 4 (Register Select/Control Pin): This pin toggles between command or data register, used to connect a microcontroller unit pin and obtains either 0 or 1 (0 = data mode, 1 = command mode).
- Pin 5 (Read/Write/Control Pin): This pin toggles the display between read or write operation, connected to a microcontroller unit pin to get either 0 or 1 (0 = Write Operation, 1 = Read Operation).
- Pin 6 (Enable/Control Pin): This pin should be held high to execute the Read-/Write process, and it is connected to the microcontroller unit and constantly held high.
- Pins 7-14 (Data Pins): These pins are used to send data to the display. These pins are connected in two-wire modes like 4-wire mode and 8-wire mode. In 4-wire mode, only four pins (0 to 3) are connected, whereas in 8-wire mode, 8 pins (0 to 7) are connected to the microcontroller unit.
- Pin 15 (LED+ Pin of the LED): This pin is connected to +5V.
- Pin 16 (LED- Pin of the LED): This pin is connected to GND.

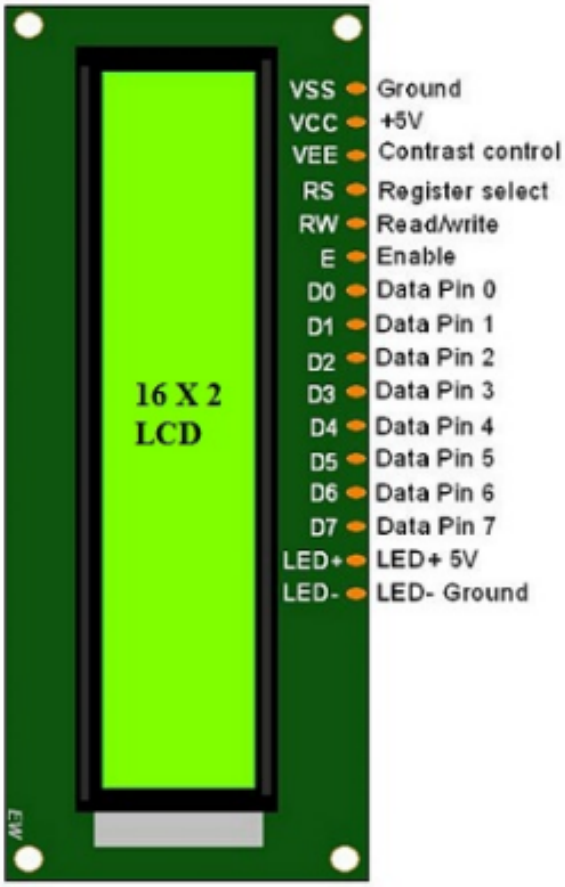

Figure 3.10: LCD-16×2 pins diagram

Instead of passing direct signal through the PINS of LCD 16×2 , we will now take the advantage of using PCF8574 to make LCD a I2C communication device

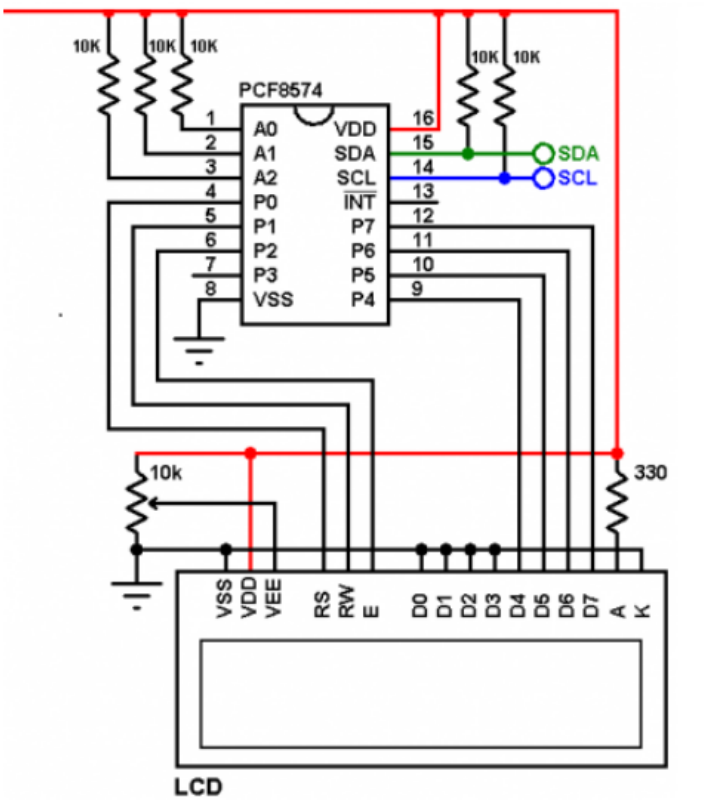

Figure 3.11: LCD-16×2 connecting to PCF8574

3.4 DHT20 SENSOR

3.4.1 Description

The DHT20 is a kind of sensor for humidity and temperature with digital I^2C output. It can be applied to HVAC, dehumidifiers, testing and inspection equipment, consumer products, automobiles, automatic control, data loggers, weather stations, home appliances, humidity control, medical, and other application fields that need to detect and control temperature and humidity.

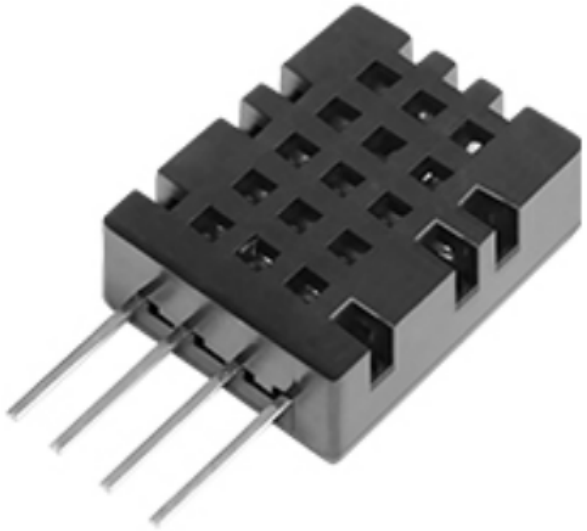

Figure 3.12: DHT20

3.4.2 Usage

Start the Sensor
The first step is to power up the sensor with the selected VDD supply voltage (range between 2.2V and 5.5V). After power-on, the sensor needs less than 100ms stabilization time (SCL is high at this time) to reach the idle state, and it is ready to receive commands sent by the host (MCU).

Start/Stop Sequence Each transmission sequence starts with the Start state and ends with the Stop state, as shown in Fig. **??** and Fig. **??**.

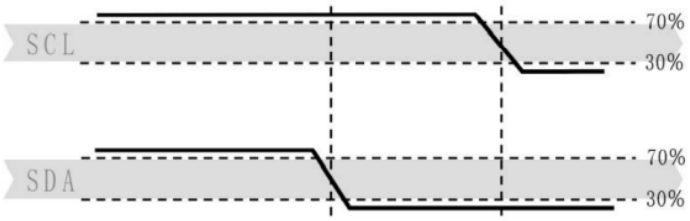

Figure 3.13: Start transmission status

When SCL is high, SDA is converted from high to low. The start state is a special bus state controlled by the master, indicating the start of the slave transfer (after Start, the BUS bus is generally considered to be in a busy state)

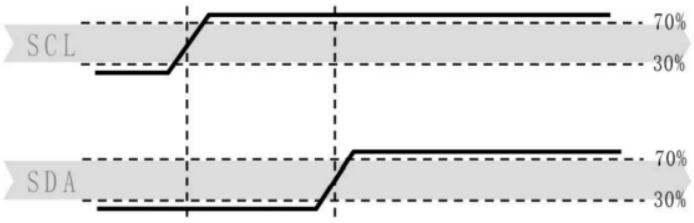

Figure 3.14: Stop transmission state

When SCL is high, the SDA line changes from low to high. The stop state is a special bus state controlled by the master, indicating the end of the slave transmission (after Stop, the BUS bus is generally considered to be in an idle state)

Send Command After the transmission is started, the first byte of I^2C that is subsequently transmitted includes the 7-bit I^2C device address 0x38 and a SDA direction bit x (read R: '1', write W: '0'). After the 8^{th} falling edge of the SCL clock, pull down the SDA pin (ACK bit) to indicate that the sensor data is received normally. After sending the measurement command 0xAC, the MCU must wait until the measurement is completed.

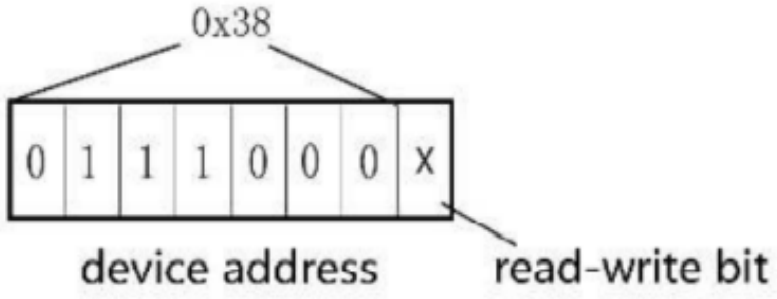

Figure 3.15: DHT20 address interface

Sensor Communication Process

1. After power-on, wait no less than 100ms. Before reading the temperature and humidity value, get a byte of the status word by sending 0x71. If the status word and 0x18 are not equal to 0x18, initialize the 0x1B, 0x1C, 0x1E registers by sending 3 bytes, `[Register, 0x00, 0x00]` to DHT20. Then, receive 3 bytes of reply from DHT20, and send the reply again but with (0xB0 or with "first reply byte").

2. Wait 10ms to send the 0xAC command (trigger measurement). This command parameter has two bytes, the first byte is 0x33, and the second byte is 0x00.

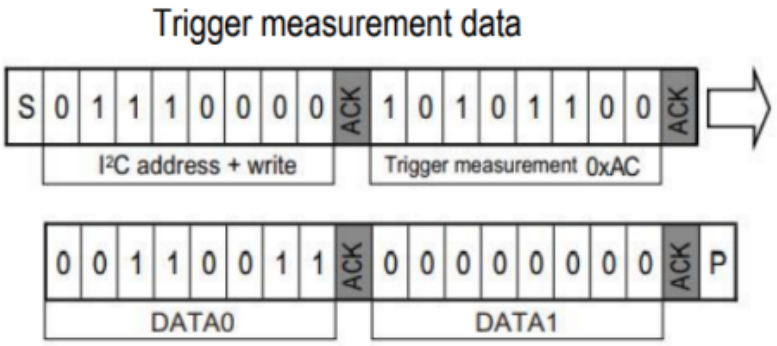

Figure 3.16: DHT20 trigger measurement interface

3. Wait 80ms for the measurement to be completed, if the read status word Bit [7] is 0, it means the measurement is completed, and then six bytes can be read continuously; otherwise, continue to wait.

Bits	Significance	Description
Bit[7]	Busy indication	1-Equipment is busy, in measurement mode 0- Equipment is idle, in hibernation state
Bit[6:5]	Retain	Retain
Bit[4]	Retain	Retain
Bit[3]	CAL Enable	1 - Calibrated 0 - Uncalibrated
Bit[2:0]	Retain	Retain

Figure 3.17: DHT20 status byte

4. First byte read indicate for the status of DHT20, the next 5 bytes regard to Humidity data and Temperature data, and the last byte use for errordetecting code CRC

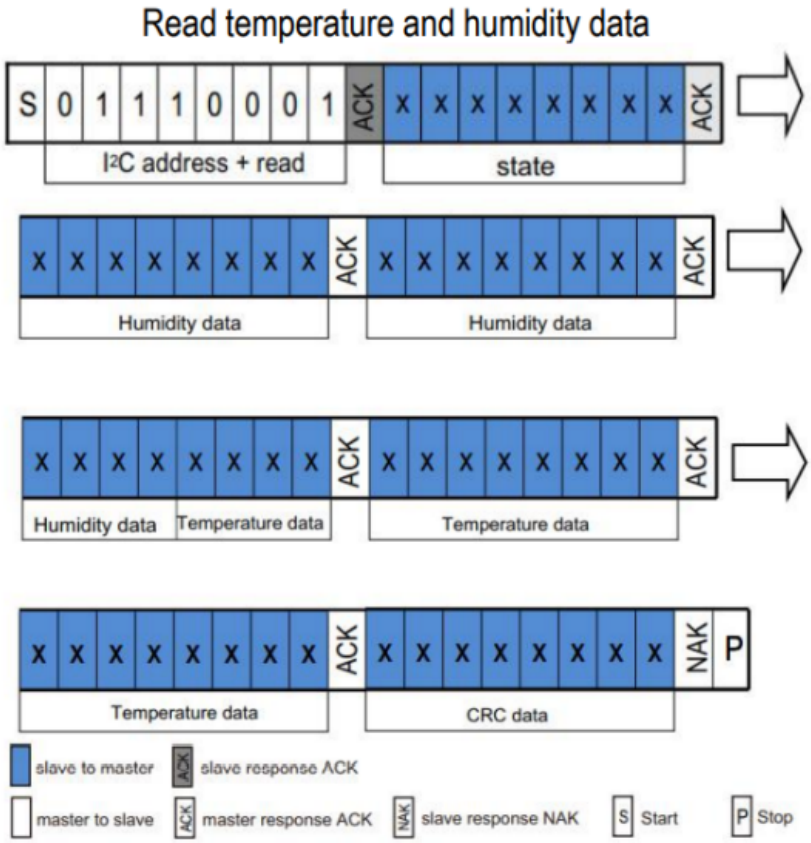

Figure 3.18: DHT20 reply value interface

5. After receiving six bytes, the next byte is the CRC (abbreviation for Cyclic Redundancy Check) check data. The user can read it out as needed. If the receiving end needs CRC check, an ACK will be sent after the sixth byte is received; otherwise, send NACK to end. The initial value of CRC is 0xFF, and the CRC8 check polynomial is:

$$\mathrm{CRC}[7{:}0] = 1 + X^4 + X^5 + X^8$$

Signal Conversion

Relative Humidity Conversion: The relative humidity (RH) can be calculated using the relative humidity signal S_{RH} output by SDA through the following formula (the result is expressed in %RH):

$$RH[\%] = \frac{S_{\mathrm{RH}}}{2^{20}} \times 100\%$$

Temperature Conversion: The temperature T can be calculated by substituting the temperature output signal S_{T} into the following formula (the result is expressed in degrees Celsius, °C):

$$T[^{\circ}\mathrm{C}] = (\frac{S_{\mathrm{T}}}{2^{20}} \times 200) - 50$$

3.5 RAIN SENSOR

3.5.1 Description

The rain sensor module is an easy tool for rain detection. It can be used as a switch when raindrop falls through the raining board and also for measuring rainfall intensity. The module features, a rain board and the control board that is separate for more convenience, power indicator LED and an adjustable sensitivity though a potentiometer.

Figure 3.19: RAIN SENSOR MODULE

3.5.2 Digital Output

The analog output is used in detection of drops in the amount of rainfall. Connected to 5V power supply, the LED will turn on when induction board has no rain drop, and DO output is high. When dropping a little amount water, DO output is low, the switch indicator will turn on. Brush off the water droplets, and when restored to the initial state, outputs high level.

3.5.3 Analog Output

The rain sensor module provides an analog signal that varies depending on the amount of rainfall detected. As rainwater falls on the rain sensor board, the resistance between the sensor's tracks changes, causing the analog output voltage to change. The more intense the rainfall, the lower the output voltage will be. This allows for more precise measurement of rainfall intensity compared to the digital output (DO) which only indicates the presence or absence of rain.

3.5.4 Usage

Name	Description
VCC	Connects supply voltage- 5V
GND	Connected to ground
D0	Digital pin to get digital output
A0	Analog pin to get analog output

Figure 3.20: Pin function of RAIN SENSOR

Clearer Specification of Each Pin

- VCC pin: Connects to VCC (3.3V to 5V).
- GND pin: Connects to GND (OV).
- DO pin: Digital output pin; it is HIGH when rain is not detected and Low when detected. The rain detection threshold is adjustable with a built-in potentiometer.
- AO pin: Analog output pin; the value decreases with increased water on the sensing pad and increases as water decreases.

1. Calculate the AO (Analog Output) Value from a Rain Sensor

The rain sensor provides an analog signal (AO) that reflects the intensity of rainfall. To calculate the AO value from this signal, an **Analog-to-Digital Converter** (ADC) is used to convert the analog signal into a digital value.

Assume that we read the value from the rain sensor through an ADC in a microcontroller. The value can be calculated as follows:

$$V_{AO} = \frac{ADC_{\text{value}}}{ADC_{\text{max}}} \times V_{\text{ref}}$$

Where:

- V_{AO} is the output voltage of the rain sensor.
- ADC_{value} is the digital value read from the ADC, ranging from 0 to ADC_{max}.
- ADC_{max} is the maximum value that the ADC can read (for example, with a 10-bit ADC, $ADC_{\text{max}} = 1023$).
- V_{ref} is the reference voltage of the system, typically 5V or 3.3V.

For example, if you are using a system with a reference voltage of $V_{\text{ref}} = 5V$ and a 10-bit ADC ($ADC_{\text{max}} = 1023$), the AO voltage can be calculated from the ADC value as follows:

$$V_{AO} = \frac{ADC_{\text{value}}}{1023} \times 5$$

Once the V_{AO} value is calculated, you can determine the rainfall intensity by calibrating the sensor such that the relationship between V_{AO} and the amount of rainfall is meaningful.

2. Calculating Rainfall (mm) from AO (Analog Output)
To calculate the rainfall in millimeters (mm) from the analog output (AO) value of the rain sensor, we need to use a calibration factor, denoted as k, which relates the voltage output of the sensor to the rainfall intensity. The formula for calculating the rainfall is:

$$\text{Rainfall}(mm) = k \times V_{AO}$$

Where:

- Rainfall is the amount of rainfall in millimeters (mm).
- k is the calibration constant, determined through sensor calibration, which reflects the relationship between the voltage output and the rainfall intensity.
- V_{AO} is the output voltage of the rain sensor, which is calculated using the ADC as shown earlier.

For example, if the output voltage V_{AO} is calculated as 2.5V, and the calibration constant k is found to be 0.2 mm/V, the amount of rainfall would be:

$$\text{Rainfall}(mm) = 0.2 \times 2.5 = 0.5\,\text{mm}$$

This formula can be applied to convert the sensor's analog output into a meaningful measurement of rainfall in millimeters.

3.6 FAN AND BLUZZER

In high-temperature conditions, both the fan and the buzzer can be utilized to ensure system safety and provide necessary alerts:

Fan Activation in High-Temperature Conditions The fan is activated when the system detects that the temperature has exceeded a predefined threshold. Its primary role is to dissipate heat and cool down the environment or device. By integrating the fan with a temperature sensor, the system can monitor real-time temperature changes and automatically adjust the fan speed using PWM (Pulse Width Modulation) control.

Buzzer Activation in High-Temperature Conditions The buzzer serves as an audible alert mechanism in case of critical temperature levels. When the temperature surpasses the safe operating range, the buzzer is triggered to notify users of the potential hazard. This sound alarm ensures that immediate action can be taken to prevent damage or system failure.

This combination of a fan for cooling and a buzzer for alerting ensures that the system remains both functional and safe in high-temperature conditions.

Integrating user notifications ensures that critical issues are addressed promptly, minimizing the risk .

3.7 BUTTON

A button is a simple and widely used input device in electronics. It is a mechanical switch that can make or break an electrical circuit when pressed or released. Buttons are often used to trigger specific actions, send commands, or interact with a system in various applications.

Types of Buttons
Buttons come in different forms, including:

- Momentary Buttons: These are active only while being pressed. Once released, they return to their default state. Examples include push buttons.
- Toggle Buttons: These maintain their state after being pressed, toggling between ON and OFF modes.
- Capacitive Touch Buttons: These are touch-sensitive and do not require physical pressing, commonly used in modern touch devices.

Applications in IoT and Electronics
Buttons play a critical role in IoT and electronics for:

- Device Control: Powering devices ON/OFF, switching modes, or starting/stopping operations.
- User Interaction: Providing an interface for users to interact with smart systems or devices.
- Input Commands: Sending specific instructions to microcontrollers, such as starting a timer or resetting a device.

Buttons are essential components in electronics and IoT, enabling intuitive user interactions and efficient system control.

3.8 ESP32 Microcontroller

3.8.1 Description

ESP32 is a microcontroller categorized as a low-power, cost-effective system-on-chip microcontroller. Most ESP32 variants integrate dual-mode Bluetooth and Wi-Fi, making it highly versatile, robust, and reliable for numerous applications.

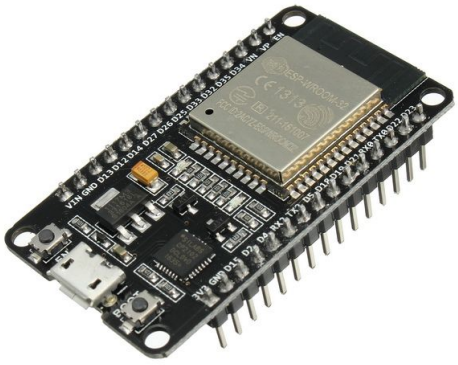

Figure 3.21: ESP32 microcontroller

It is the successor to the popular NodeMCU ESP8266 microcontroller and offers enhanced performance and features. The ESP32 microcontroller is manufactured by Espressif Systems and is widely used in various applications such as IoT, robotics, and automation.

The ESP32 is also designed for low power consumption, making it ideal for battery-powered applications. It includes a power management system that allows it to operate in sleep mode and wake up only when necessary, significantly extending battery life.

3.8.2 Usage

Clearer Specification of Each Pin

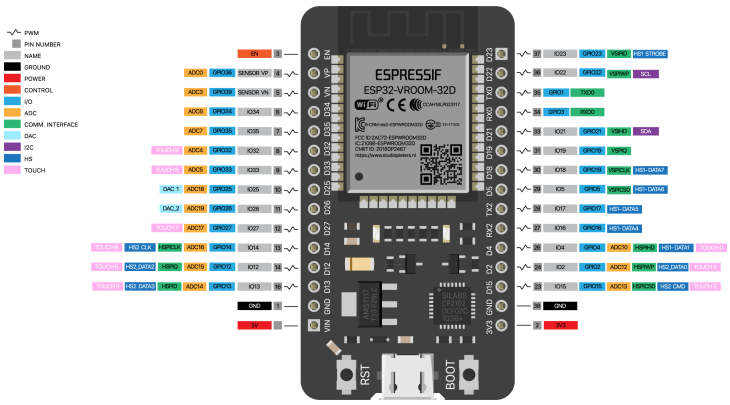

Figure 3.22: ESP32 Development Board Pinout

The ESP32 microcontroller offers a pin multiplexing feature that allows us to select which peripherals to connect from its 28 input/output pins. This feature provides flexibility to decide which pins to use for specific peripherals such as MISO, RX, SCLK, MOSI, TX, SCL, SDA, and several other options.

However, some pins dedicated to analog-to-digital converters (ADC) and digital-to-analog converters (DAC) are reserved for these peripherals and cannot be configured according to user requirements.

ESP32 Specifications

- Analog-to-Digital Converter (ADC) Channels: 18
- SPI Interfaces: 3
- UART Interfaces: 3
- I2C Interfaces: 2
- PWM Outputs: 16
- Digital-to-Analog Converters (DAC): 2
- I2S Interfaces: 2

On-Chip Sensors and Features

- Hall Sensor

- 32 kHz Crystal Oscillator
- Ultra-Low Noise Analog Amplifier
- 10x Capacitive Touch Interfaces

Powering the ESP32 Development Board

To power the ESP32 development board, we have the option to use either a USB port or a LiPo battery. When both power sources are connected, some development boards feature an integrated charging controller to charge the LiPo battery. Additionally, the board includes a 3.3 V voltage regulator capable of supplying up to 600 mA of current. During RF transmission, the board can consume up to 250 mA.

It is important to note that the general-purpose input/output (GPIO) pins cannot handle 5 V directly; therefore, level shifting is required to communicate with 5 V devices.

Programming the ESP32

The ESP32 board can be programmed using the Arduino IDE. However, certain drivers and libraries need to be installed to make it compatible with the IDE and ready for use. Below is a brief explanation of the necessary drivers and libraries:

Drivers: Ensure that the USB-to-serial converter driver for your ESP32 development board is installed (e.g., CP210x or CH340 drivers). Libraries: Install the ESP32 core libraries for Arduino IDE from the board manager or the official GitHub repository. Additional Tools: Configure the correct board, port, and upload settings in the Arduino IDE for seamless programming.

3.9 Sever Blynk

Blynk is an Internet of Things (IoT) platform that enables users to easily create remote control applications for smart devices. The standout feature of Blynk is its flexibility and ease of use. You can access the following link: `https://blynk.io/`

With Blynk, users can create IoT device control applications within minutes without requiring in-depth programming knowledge. Blynk provides a range of tools and an intuitive user interface that allows users to easily drag and drop, interact with control components, charts, sensors, and many other features.

Figure 3.23: Low-code IoT cloud platform

Specifically, users can create applications to control lights, fans, temperature sensors, humidity monitors, and even smart dishwashers. Blynk supports multiple platforms, ranging from mobile devices such as smartphones to embedded devices like Arduino, Raspberry Pi, ESP8266, and more.

Figure 3.24: Blynk mobile app

This makes Blynk a valuable tool not only for professional IoT developers but also for beginners who are interested in exploring the world of IoT. With its convenience and

flexibility, Blynk has been empowering people to bring their creative IoT ideas to life effectively.

Chapter 4

SYSTEM DESIGN

In this chapter, we propose the architecture of our system, introducing the hardware components used to implement it. Furthermore, we also provide an overview of the system with the block diagram. Finally, there will be a fully description for the connection protocol between each component in our system.

4.1 Proposal Architecture

In this project, we aim to build an extension shield to measure the real-time temperature and humidity of the surrounding environment with the participation of the DHT20 Sensor and LCD1602 for output display. Furthermore, the system also offers control of its operation by typing in commands through a simulation tool from the host computer.

In detail, the system consists of three modes which are Waiting, Reading (Every 3 seconds), and Capturing. However, Waiting is just a temporary intermediate state between system boot and Reading mode. When power is plugged in, it will print out a greeting message and then immediately change to Reading mode.

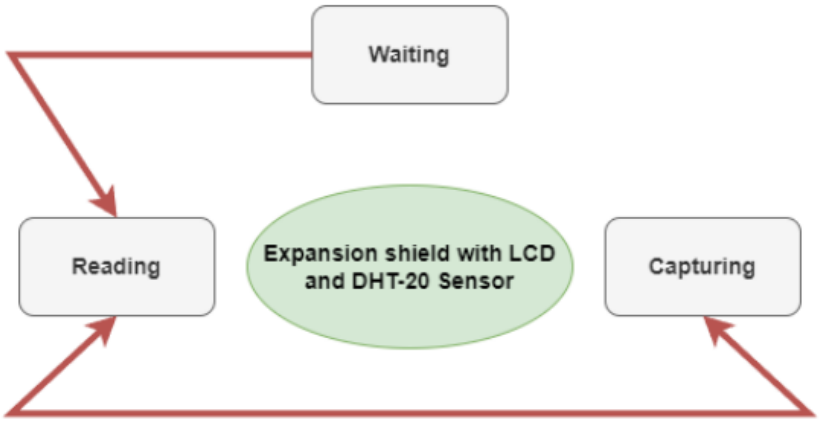

Figure 4.1: Proposal Architecture

In Reading mode, the LCD will display the value read from the DHT-20 sensor every 3 seconds. Whenever a UART command in the form `!C#` (stands for Capture) is received, the LCD will show the value at that time and stop fetching new data.

Similarly, if the command `!R#` (stands for Reset) is encountered, the system continues reading again. Generally, the system can change between Reading and Capturing modes frequently.

4.2 Hardware Usage

As we have introduced above, the system will exploit the STM Nucleo board as the main hub for our expansion shield, which attaches the DHT-20 sensor, the LCD 16×2, and some supported electrical components to ensure the correct operation.

4.3 Detailed Description

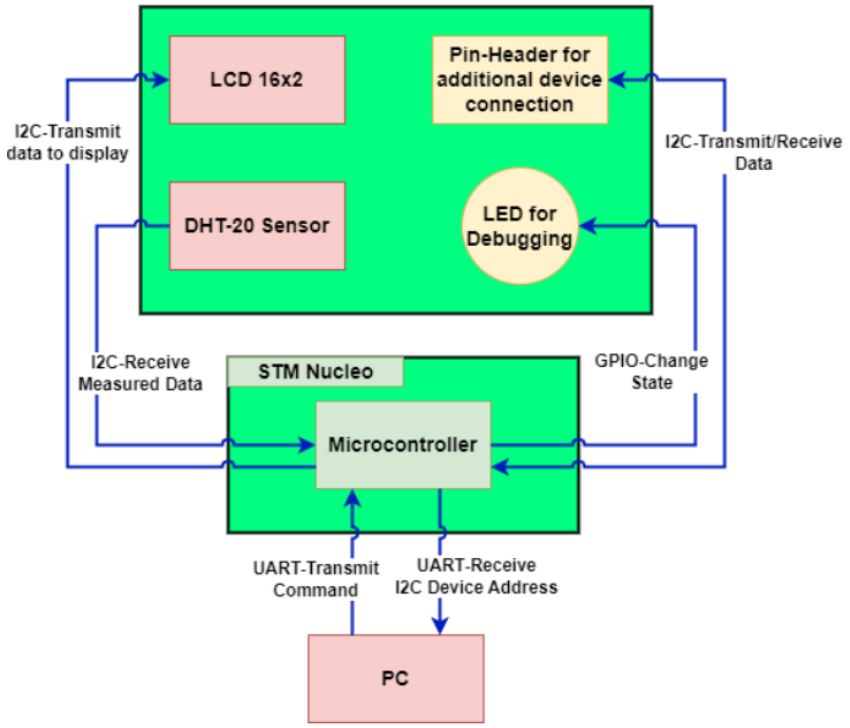

Figure 4.2: Block Diagram for the sytem

First of all, our two main components, which are the LCD 16x2 and DHT-20 sensor, will connect with the MCU through the I^2C protocol, but in different states: Transmit and Receive, respectively. Specifically, the DHT-20 acts as a slave device that will successively transmit the measured data back to the MCU. Then, the MCU will intercept the data, perform calculations to convert the data into a comprehensible form, which is the decimal value in degrees Celsius for temperature and percentage for humidity, respectively, and make those values ready for output.

The other additional components, such as the LED for debugging, use standard GPIO connections. Meanwhile, the Pin-Header links with the MCU via I2C and provides either Transmit or Receive mode. This part is the extension connector that can attach to different I2C devices.

The bridge between the PC and the MCU is constructed under the UART protocol with the assistance of Hercules or Hterm simulation tools. Then, we are able to receive the address of the I^2C device through the scan function, as well as send commands to the MCU.

Chapter 5

ALTIUM DESIGNER

Following up, we will demonstrate our work on design stage of the expansion shield in which we mainly interact with the Altium Designer application

5.1 Schematic Design

5.1.1 Schematic STM32F103RB_NUCLEO

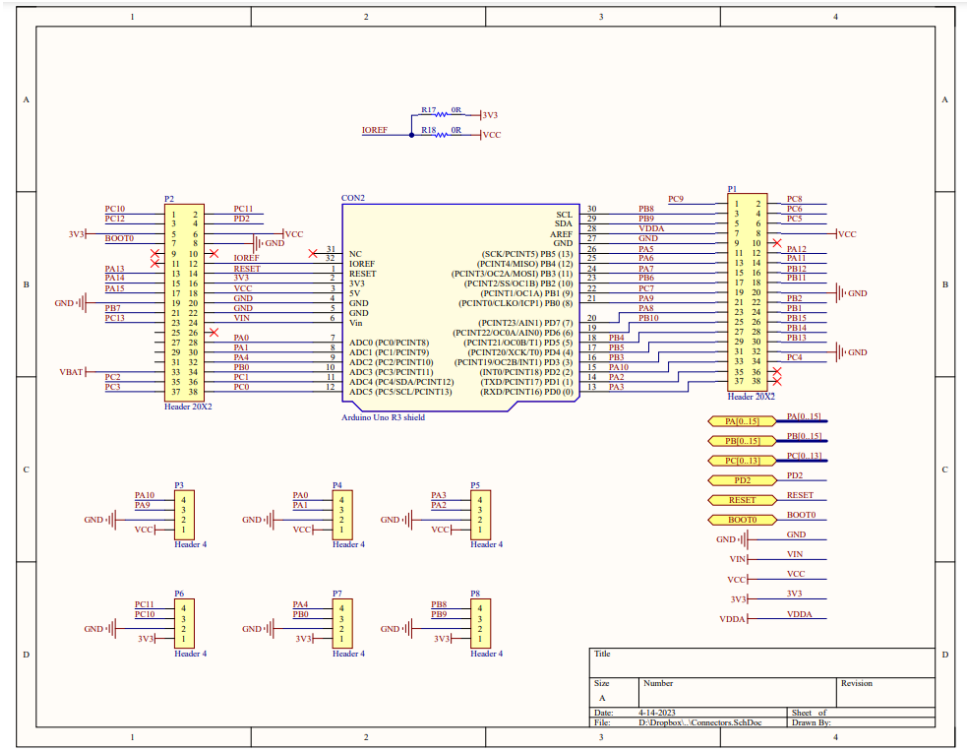

Figure 5.1: Schematic STM32F103RB_NUCLEO

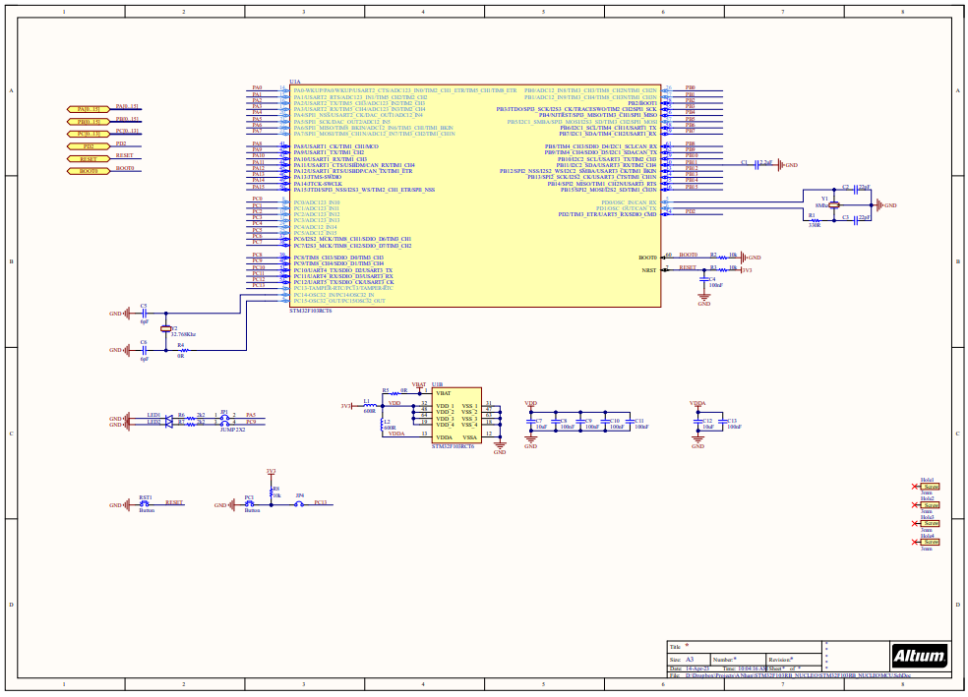

Figure 5.2: Schematic STM32F103RB_NUCLEO

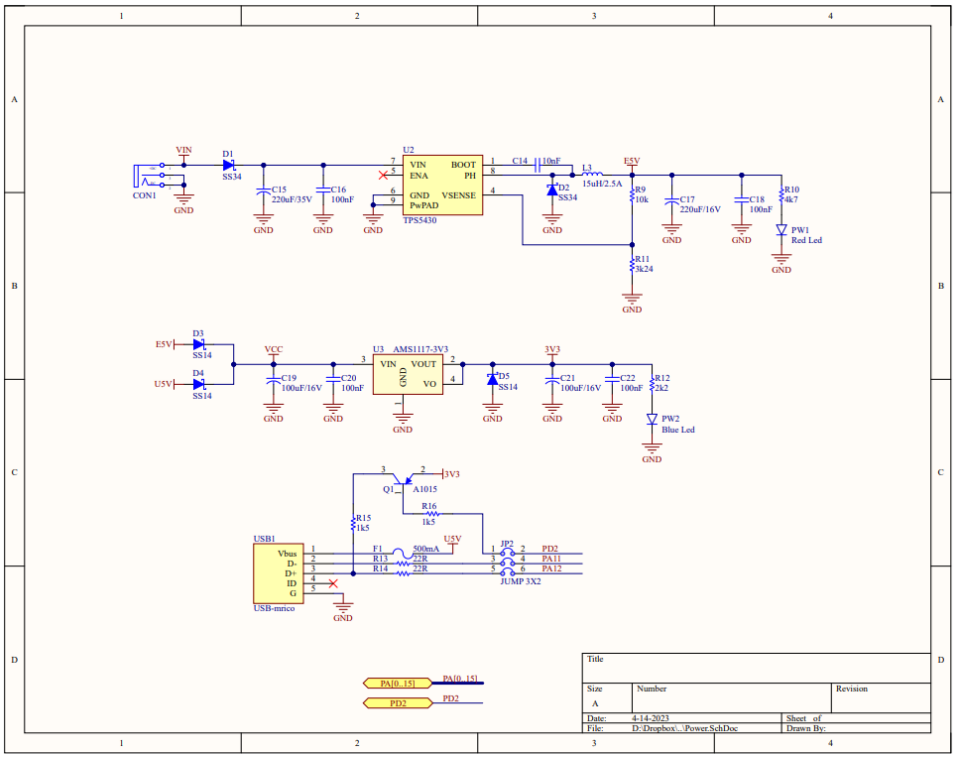

Figure 5.3: Schematic STM32F103RB_NUCLEO

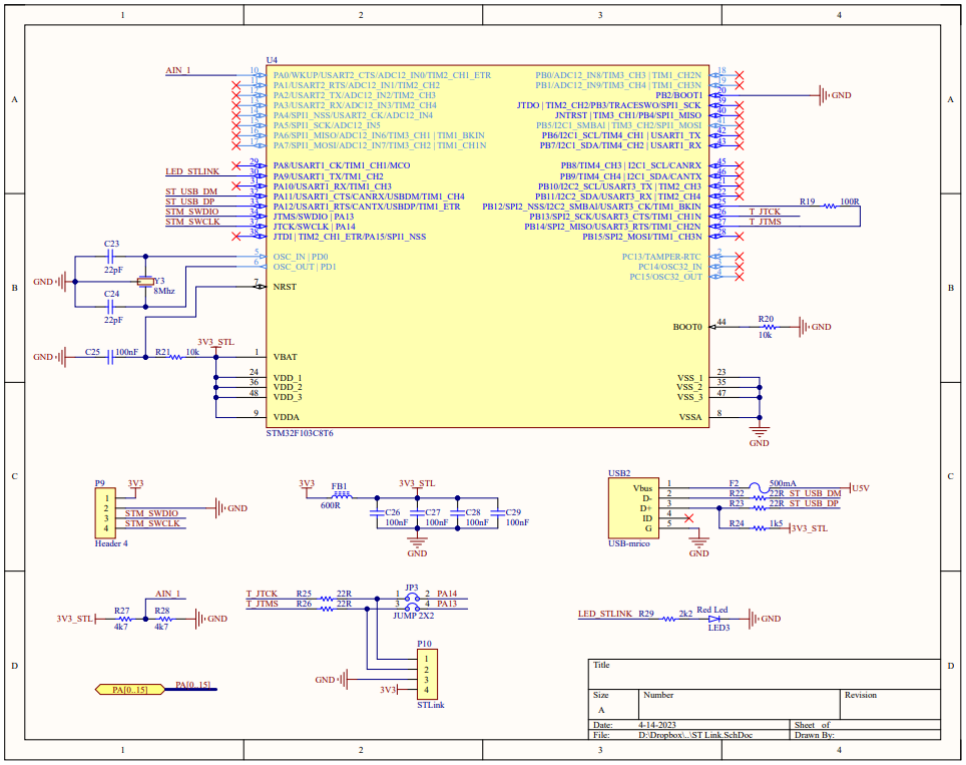

Figure 5.4: Schematic STM32F103RB_NUCLEO

5.1.2 NUCLEO_Schematic

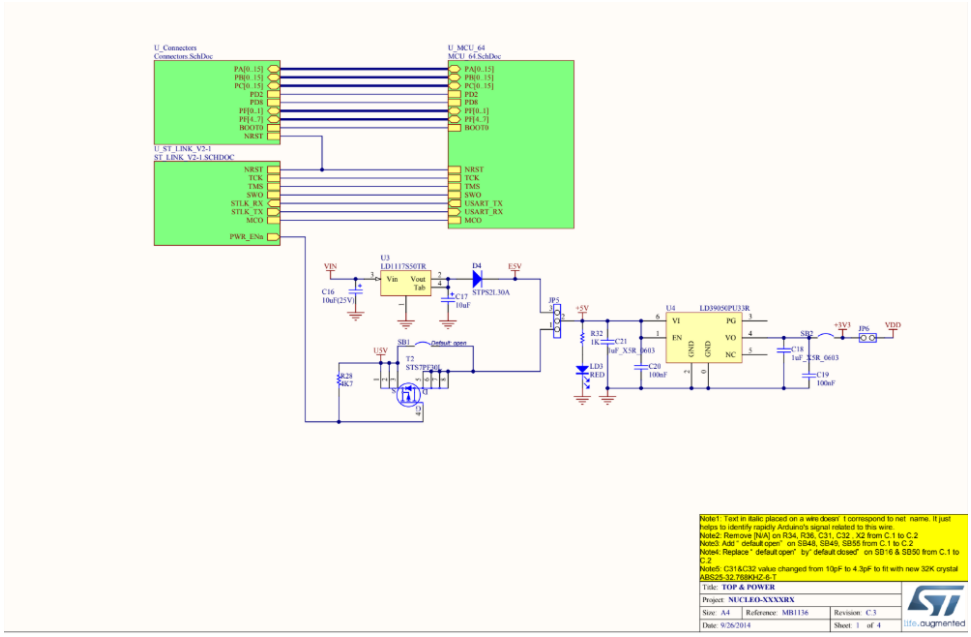

Figure 5.5: NUCLEO_Schematic

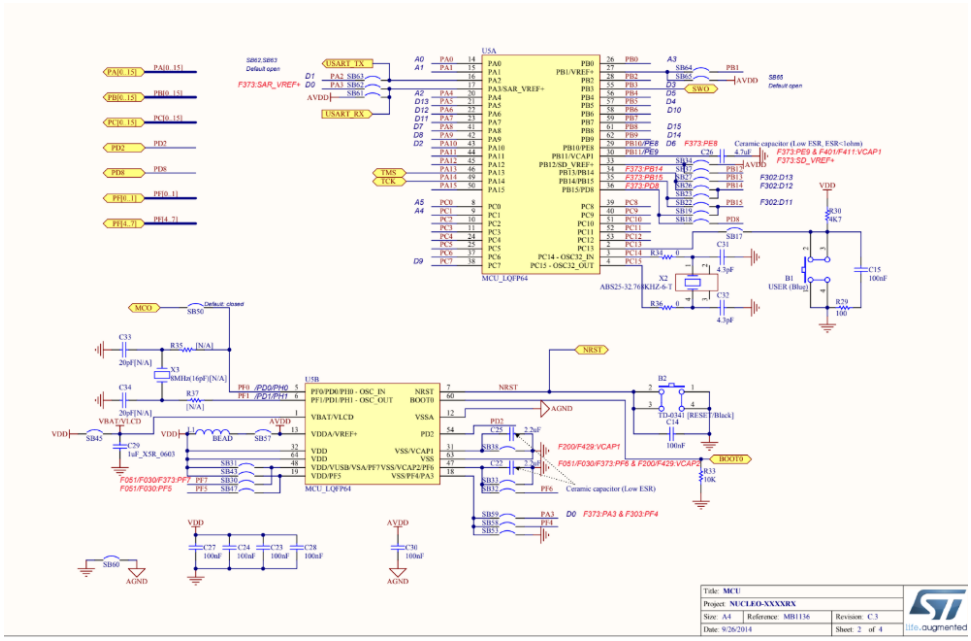

Figure 5.6: NUCLEO_Schematic

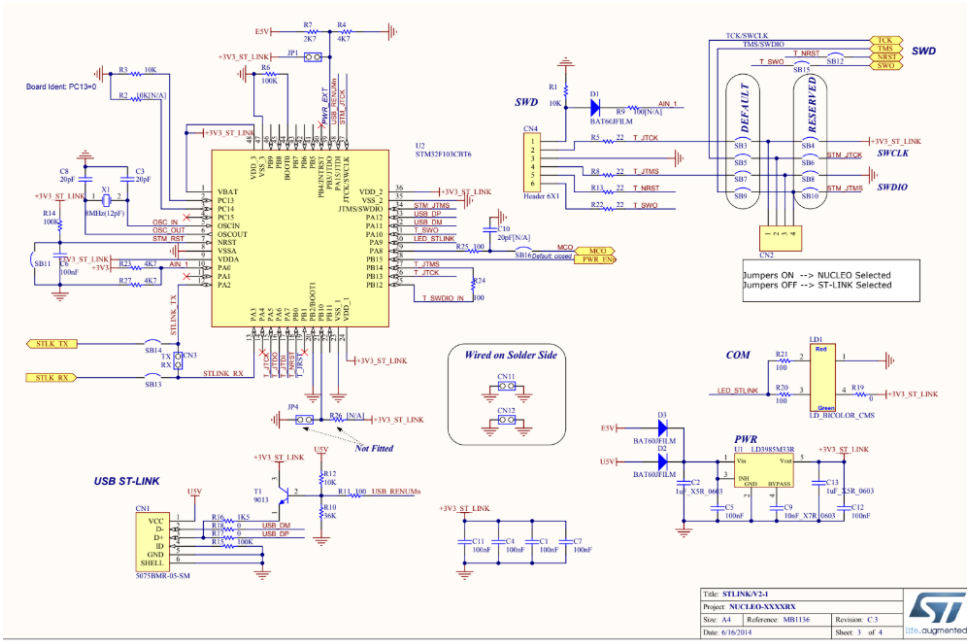

Figure 5.7: NUCLEO_Schematic

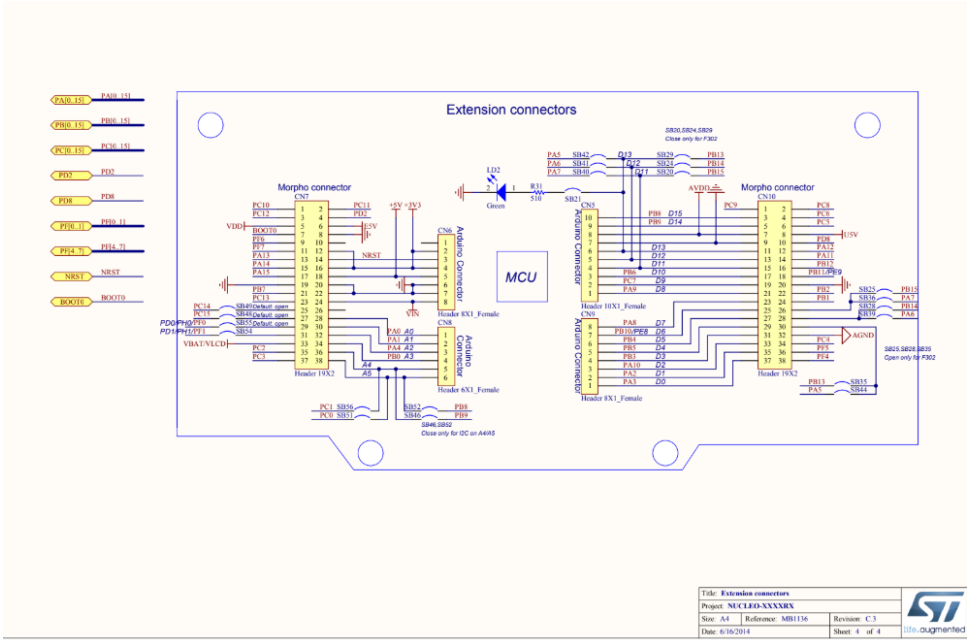

Figure 5.8: NUCLEO_Schematic

5.1.3 Schematic LCD 16x2

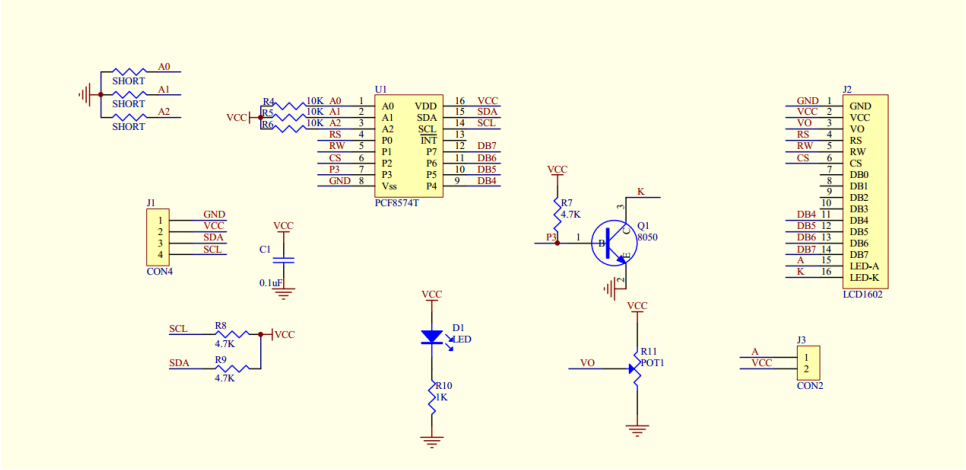

Figure 5.9: Schematic LCD 16x2

5.1.4 Schematic I²C

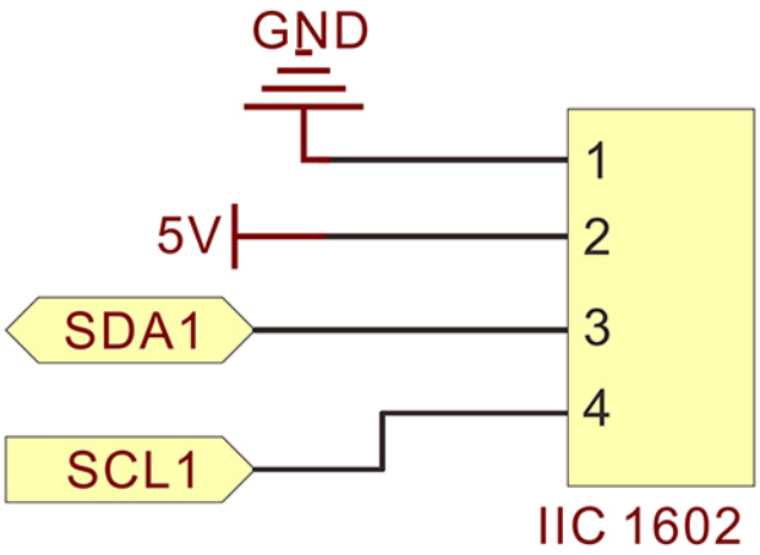

Figure 5.10: Schematic I²C

5.1.5 Schematic DHT20

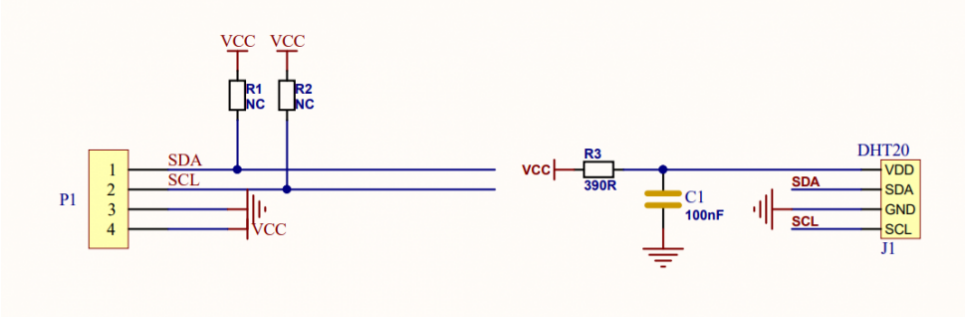

Figure 5.11: Schematic DHT20

5.1.6 Schematic UART

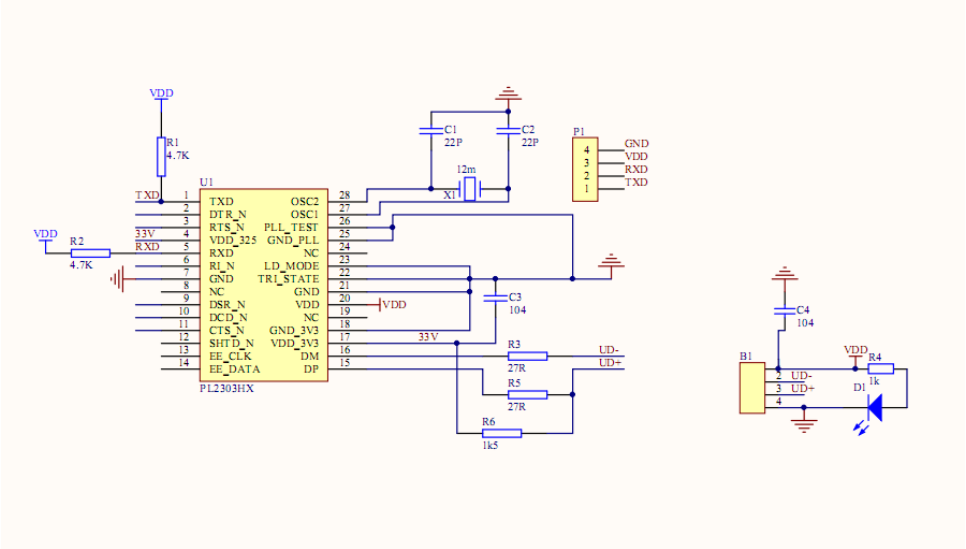

Figure 5.12: Schematic UART

5.1.7 Schematic RAIN SENSOR

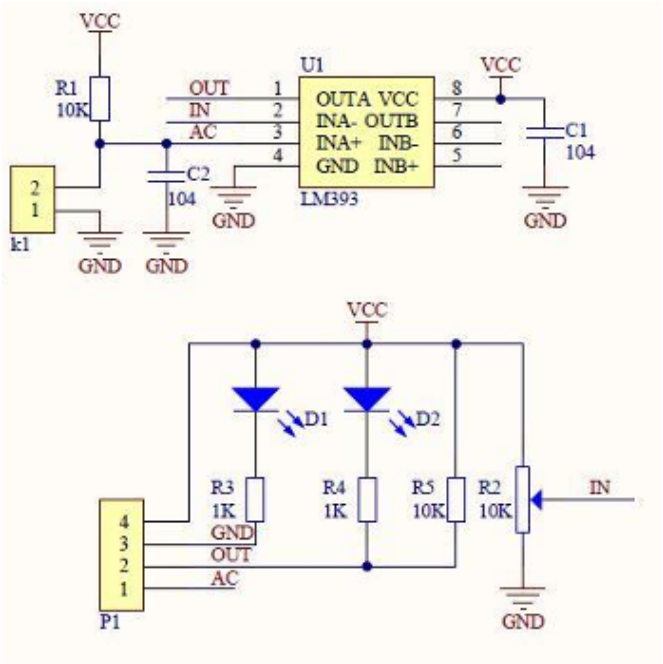

Figure 5.13: Schematic RAIN SENSOR

5.1.8 Schematic ESP32

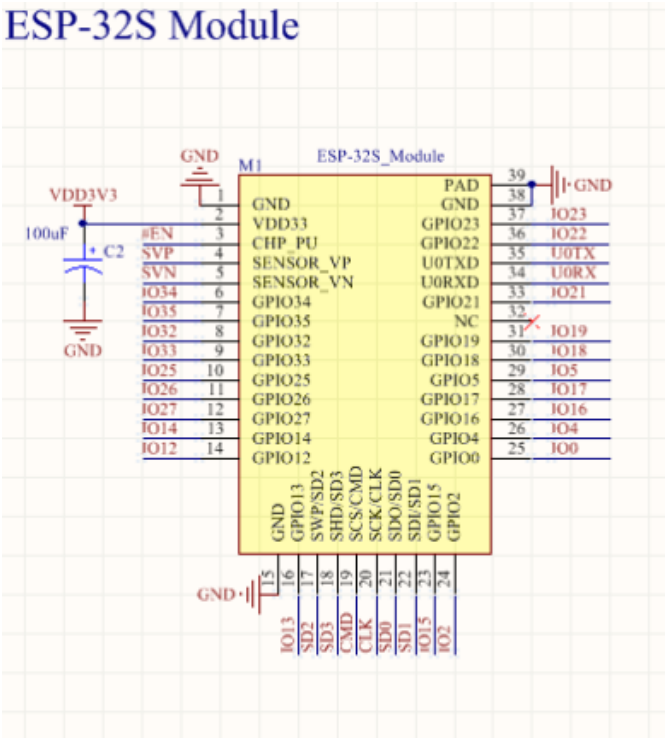

Figure 5.14: Schematic ESP32

Chapter 6

PROGRAM DESIGN

6.1 Program Layout

The program can be divided into three groups of functions, which are: Displaying information on the LCD, Scanning the device address & Interpreting the command, and lastly, Reading value from the DHT-20 sensor & Combining to a complete system.

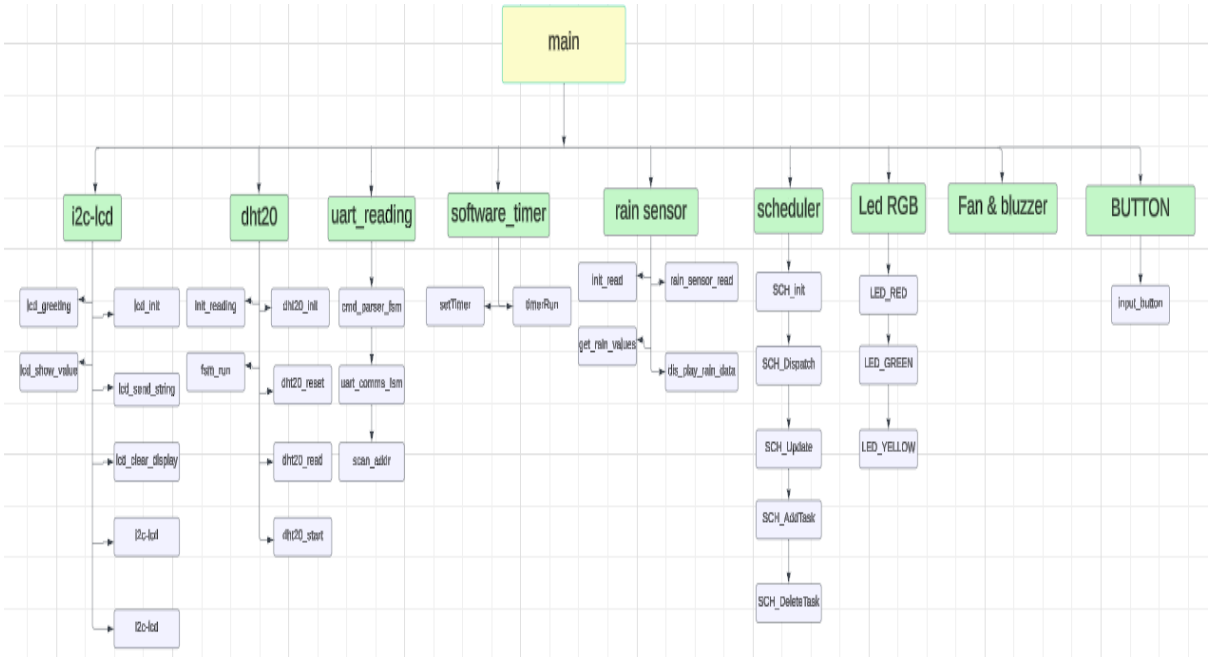

Figure 6.1: The program's layout tree

6.1.1 Display information on the LCD

The contents of this group can be found in the files `i2c-lcd.h` for prototype declaration and `i2c-lcd.c` for the implementation. These files include a set of functions that are capable of:

- Initializing the LCD to a 4-bit interface
- Sending data, strings, and commands to the LCD

- Clearing the LCD
- Moving the cursor to any location
- Sending a greeting for Waiting Mode
- Displaying measured values

6.1.2 Scanning the device address and interpreting the command

The files `uart_reading.h` and `uart_reading.c` are now in charge of this group of operations. Generally, they will perform the following tasks:

- Parse the input and determine the corresponding command (finite state machines are used here).
- Perform the associated action with each predefined command.
- Scan the address of I2C devices, including the DHT-20 sensor and LCD-1602A.

6.1.3 Reading value from the DHT-20 Sensor and Combining to a complete system

To service this group, `dht20.h` and `dht20.c` are the needed files. They will execute the following jobs:

- Initialize, reset, and start the reading operation of the DHT-20 sensor.
- Provide a finite state machine that allows the 3-second measuring routine.

6.1.4 Software Timer

This group of functions is used to provide a software timer for synchronizing the operation of our system. Since the interrupt is invoked every 10ms, calling the function `setTimer(100)` will give us a 1s timer interrupt.

6.1.5 Cooperative Scheduler

To service this group, scheduler.h and scheduler.c are the needed files.They will execute the following jobs.
1. Implement a task management scheduler.
2. Tasks called in the timer interrupt function must have O(1) complexity.

Functions to be Implemented

- `SSCH_Init`: Initialize the database for storing tasks.
- `SCH_Update`: Update the remaining waiting time of tasks in the queue.
- `SCH_Dispatch`: Execute tasks that are ready in the queue.
- `SCH_AddTask`: Add a task to the database.
- `SSCH_DeleteTask`: Remove a task from the database.

6.1.6 Button

When using buttons in embedded systems, mechanical noise or bouncing occurs as the button contacts stabilize upon pressing or releasing. This bouncing can cause erroneous multiple detections of a single press. Debouncing is the process of ensuring stable button state detection by filtering out these transient states.

Button State Processing

To solve the example problem, we define 3 states as follows:

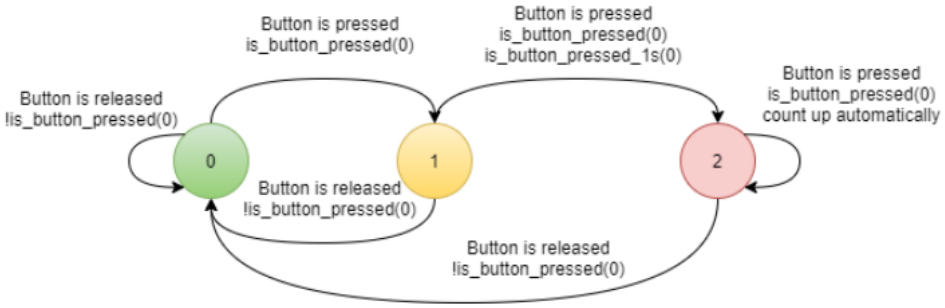

Figure 6.2: An FSM for processing a button

To service this group, input_button.h and input_button.c are the needed files. In addition, it also converts the printing status to the LCD screen.

6.1.7 Finite State Machine

For Command Parser

This FSM is built through the function `cmd_parser_fsm()` in the file `uart_reading.h` for prototype declaration and `uart_reading.c` implementation. The main feature of this FSM is to parse the input sequence from the user and determine which kind of command it is.

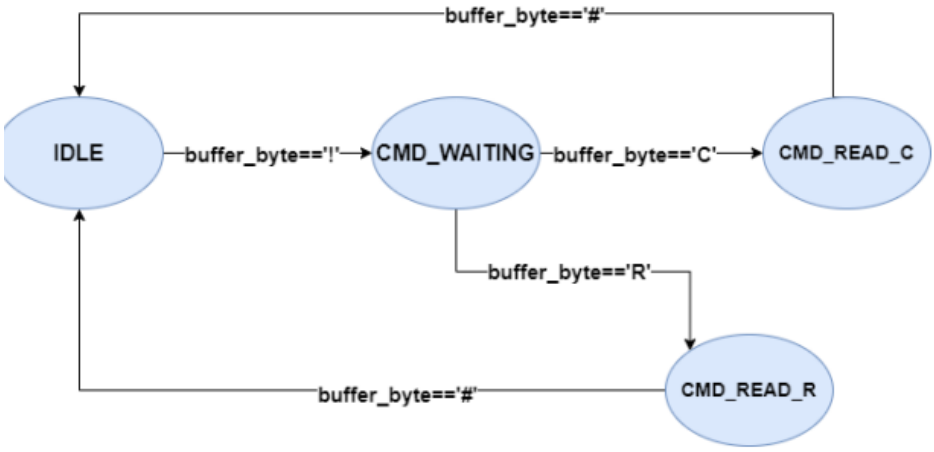

Figure 6.3: FSM For Command Parser

For Fully Operated System

This FSM is built through two functions: `uart_control_fsm()` and `reading_fsm_run()`. They are located in the sets `uart_reading` and `dht20`, respectively. The function `uart_control_fsm()` is in charge of intercepting the corresponding flag to change from Capture mode to Reset mode and vice versa. Meanwhile, the function `reading_fsm_run()` will successively fetch data from the DHT20 sensor every 3 seconds.

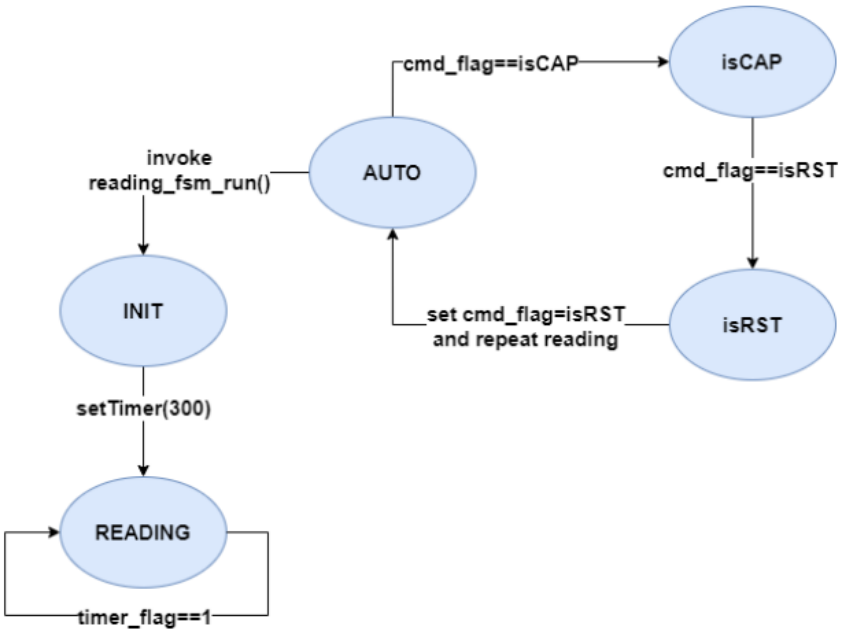

Figure 6.4: v

Usage of FSMs

The Command Parser FSM runs first. It will successively get the input and compare whether the receiving message matches any predefined commands. If so, the remaining task is to set the corresponding `cmd_flag`. This will act as a signal for the Full Operated System FSM to process.

At the beginning, if the user does not type anything, the Command Parser FSM stays in the IDLE state. However, since the `cmd_flag` is set to AUTO by default, the Full Operated System FSM will then call `reading_fsm_run()` to start the measurement.

6.1.8 Flow and Error Control in Communication

The target is to implement a UART communication between the STM32 and the ESP32. A data request is sent from the terminal to the STM32. Afterward, computations are performed at the STM32 before a data packet is sent to the terminal. The terminal is supposed to reply an ACK to confirm the communication successfully or not.

6.1.9 Program for ESP32

Upon receiving data from the STM32, the ESP32 program seamlessly relays this information to the server for monitoring and analysis. Simultaneously, it listens for commands

from the server, ensuring two-way communication by forwarding these commands back to the STM32. This robust interaction enables efficient data observation and control, forming a critical link in the system's overall functionality.

6.1.10 Implementation

For version control and synchronization, we use Git and GitHub to manage the project. In order to gain full access to the materials and code, please follow this link:
`https://github.com/tungngo2525`

Chapter 7

EXPERIMENTS

This chapter mainly includes images of the system in different operation mode. It is used to ensure the correctness of our design and implementation.

7.1 Measure ambient temperature

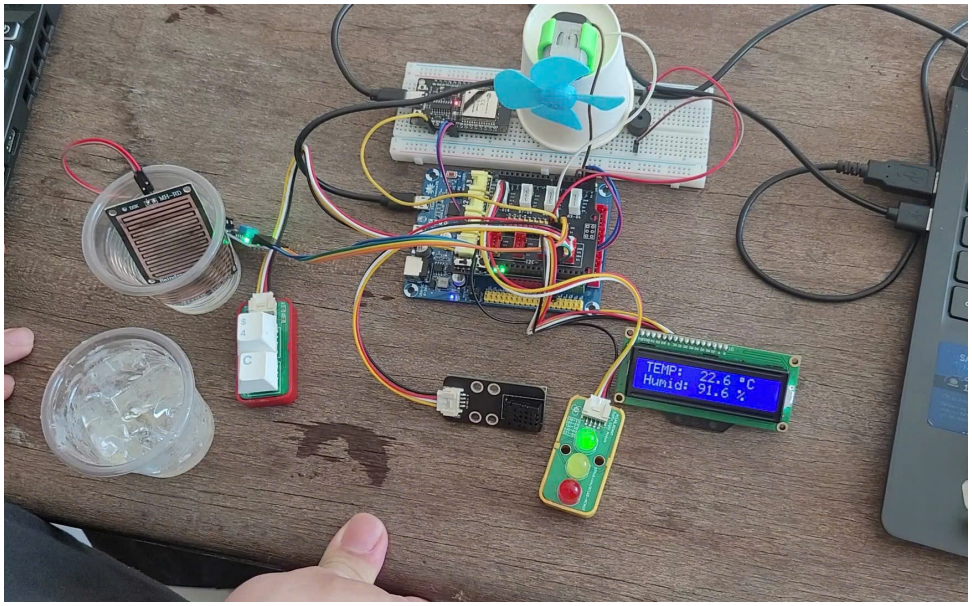

Figure 7.1: Normal ambient temperature

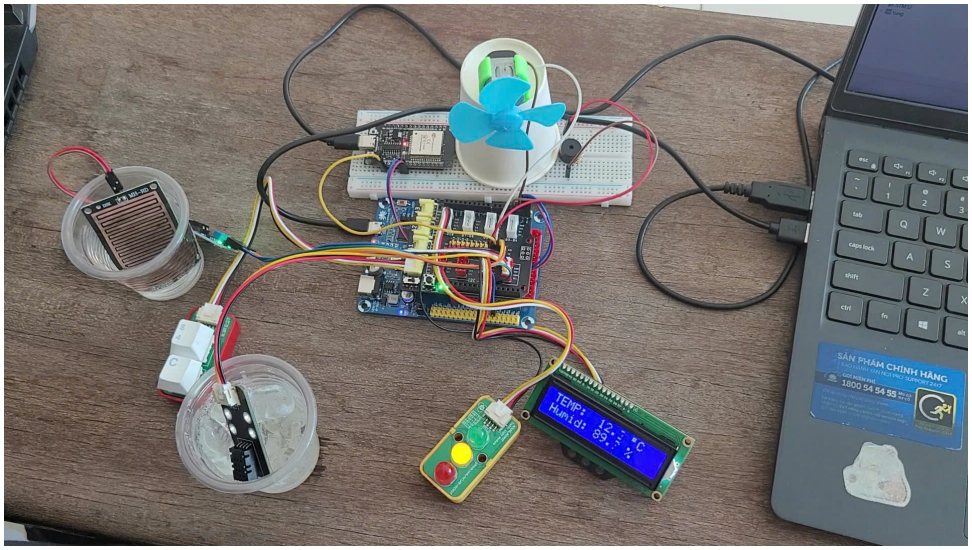

Figure 7.2: Ambient temperature is too cold

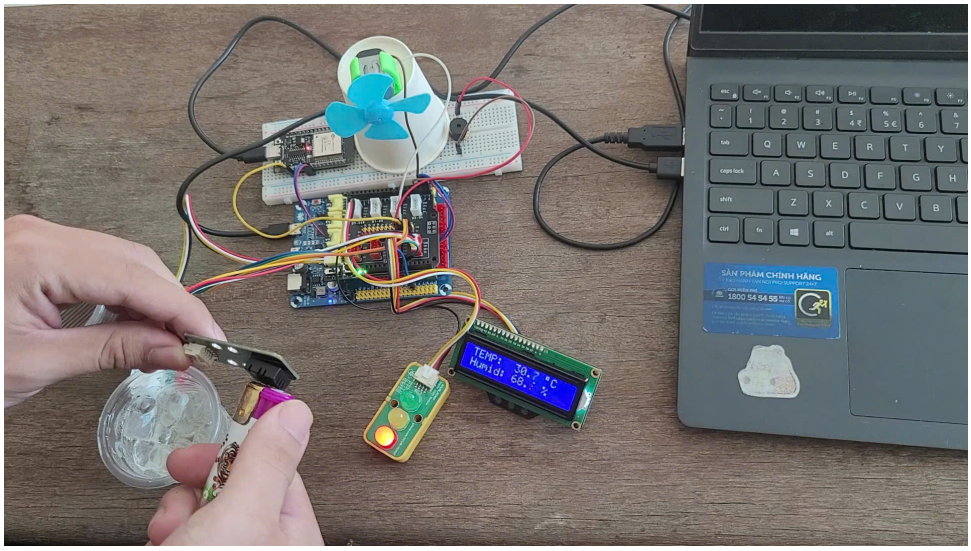

Figure 7.3: Ambient temperature is too hot

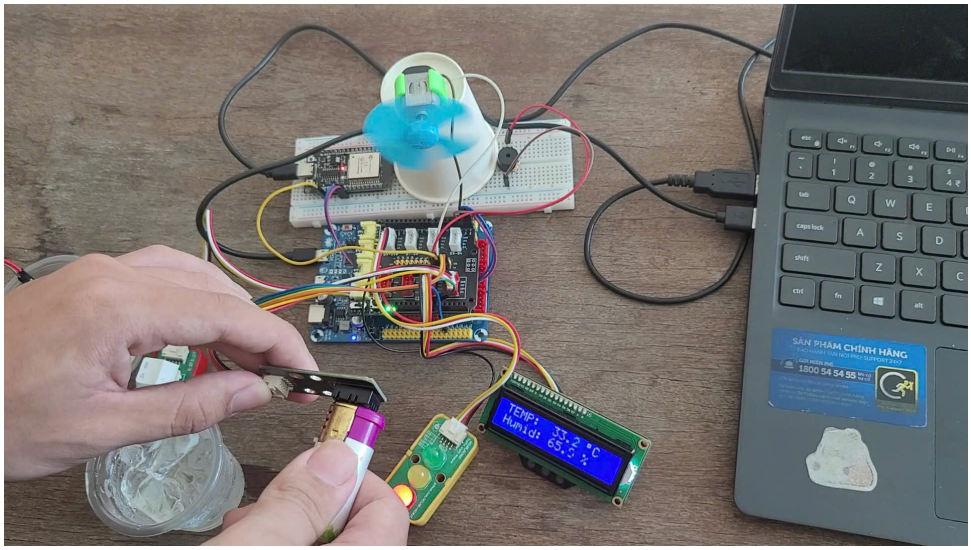

Figure 7.4: The ambient temperature exceeds the permissible level

7.2 Measure rain flow

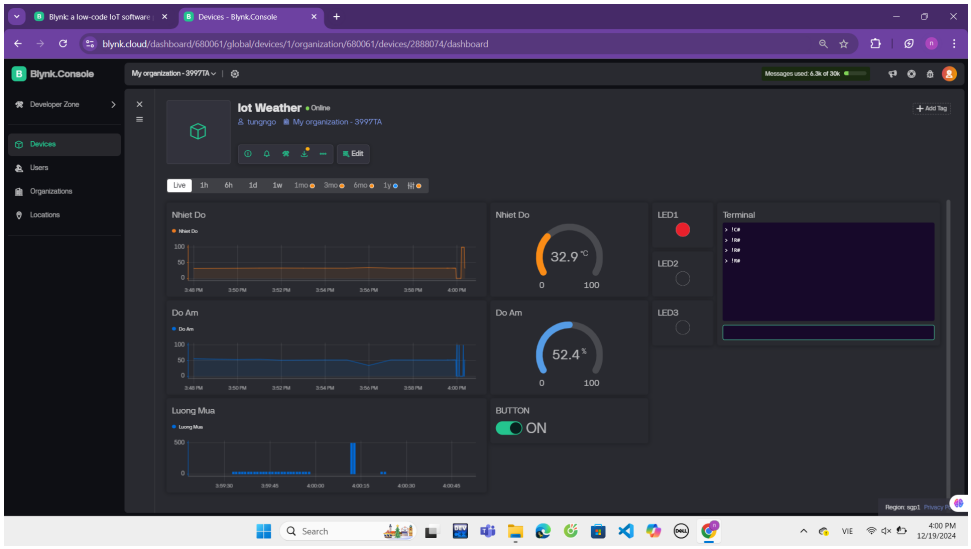

Figure 7.5: Send R# command to stm32

7.2.1 Monitored by phone app

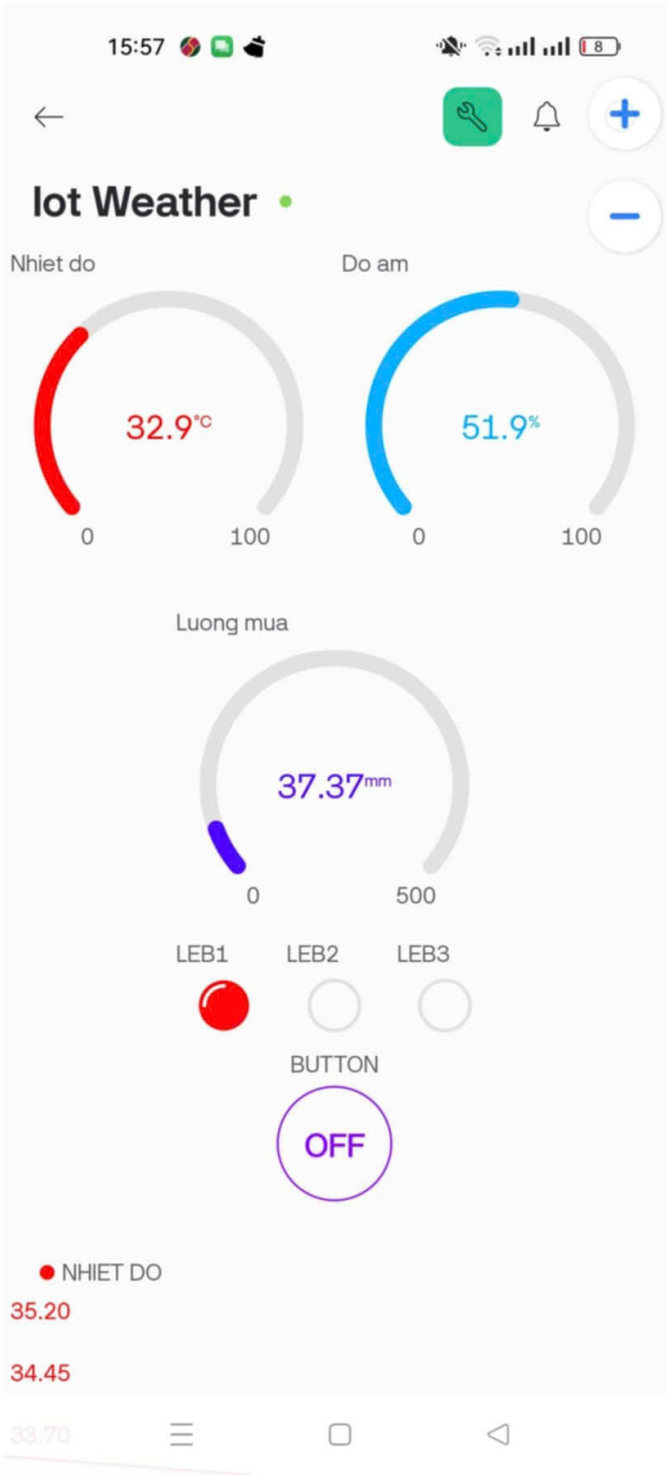

Figure 7.6: Monitored by phone app

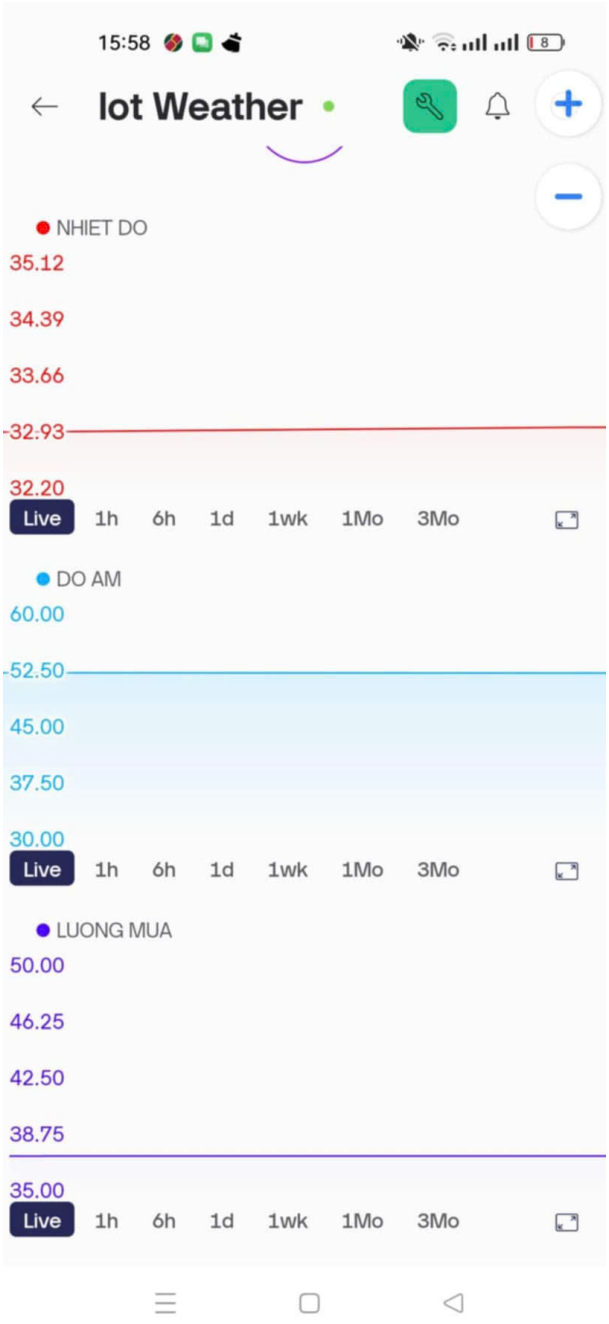

Figure 7.7: Monitored by phone app

Chapter 8

CONCLUSION & PERSPECTIVE

To sum up, we have fulfilled all goals that were set at the beginning of the project's research. With persistence, well-organization, and cooperation between team members, self-study capability, and the assistance from the instructors, we have overcome all problems that arose during the research process.

On the other side, there were many difficulties we encountered while doing the research. For instance, we spent weeks figuring out the address of the I2C devices since it is a core factor to ensure correct operation. As a result, all of the other tasks were delayed until the address was found. Furthermore, in order to complete the project, we needed to read and understand many user manuals, device datasheets, as well as academic references, to construct the system, program designs appropriately, and optimize the code for better performance.

In the near future, we would like to extend our work towards the IoT direction. Specifically, our module will act as a machine to collect information about the quality of the air, including the temperature, humidity, the presence of harmful gases (for instance, Ammonia - NH_3, Sulfur - S, Benzene - C_6H_6, Carbon dioxide - CO_2, etc.), using the MQ-135 sensor, and also the concentration of PM 2.5 dust particles with the Optical Dust Sensor PM 2.5 GP2Y1010AU0F.

All the relevant information collected from the sensor system will be stored on the AdaFruit Server and exploited by the mobile application. The app will then be in charge of showing the information and pushing notifications in case the air quality index exceeds the safety boundary.

Another feature that we have thought of is applying statistical and AI models in attempting to predict the quality of the air in the near future. Consequently, we will be able to give an early warning so that people have time to protect their health by wearing appropriate masks that can prevent fine dust or by turning on the air purifier.

Moreover, the app will also allow users to turn on or turn off the machine. This aspect will replace the old-fashioned UART communication protocol in sending commands to the system through a host computer. It would be much more convenient compared to the UART method, as not everyone is familiar with using simulation tools.