VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Quynh Luong (e1601128)

Hoang Nguyen (e1601111)

Thanh Van (e1601139)

CARBON MONOXIDE DETECTOR

Embedded System Design

School of Technology
2019

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
School of Technology

ABSTRACT

Author:	Quynh Luong, Hoang Nguyen and Thanh Van
Title:	Carbon monoxide Detector
Year:	2019
Language:	English
Pages:	51 + 3 Appendices
Name of Supervisor:	Jani Ahvonen

The thesis was made for learning and developing purposes. It can be used as a reference for what need to be considered when implementing any Carbon monoxide Detector or any Air Quality Control similar system. The thesis also covered a step by step instruction of how to retrieve data from Air Quality Control 5 Click sensor as well as how to use Xbee to transmit data. A custom printed circuit board was designed and built to connect to microcontrollers, sensors, resistors with its first priority is that the system should be as small and convenience as possible.

The study was carried out to solve air quality management problems inside industries. The system will read sensor measurement to calculate and evaluate the level of pollutants on the air. The system communicates to server using XBee as a data transmission device. One element of the thesis was to find out the working process of a typical air detector system.

The current implementation of the system is basically completed. The hardware connection, software and communication are working properly. However, for a long-term improvement, the equation for sensor should be enhanced so that the system can bring out not only the data it read but also the exact evaluation and the correct impact level of polluted air on people's health.

Keywords Learning, stimulation, teaching methods

CONTENTS

APPENDICES

LIST OF FIGURES AND TABLES

LIST OF APPENDICES

1 TERMS AND ABBREVIATIONS

PCB	Printed Circuit Board
CO	Carbon monoxide
I2C	Inter-Integrated Circuit
ADC	Analog to Digital Conversion
V	Voltage
R	Resistor
Ppm	Parts per million
MOS	Metal Oxide Semiconductor
IoT	Internet of Things
Rs	The resistance measured by sensor in a current environment
Ro	The resistance measured by sensor in condition environment
UART	Universal Asynchronous Receiver-Transmitter
RF	Radio Frequency
DI	Data Input
DO	Data Output
CTS	Clear To Send
RTS	Request To Send
MSB	Most Significant Bit
LSB	Least Significant Bit
API	Application Programming Interface
PGA	Programmable Gain Amplifier
SPS	Samples Per Second
MUX	multiplexer

ESD	Electrostatic Discharge
TX	Transmit

2 INTRODUCTION

The scope of this thesis was to design and implement a CO detector system, using Air Quality Control 5 Click sensor with STM32L432KC microcontroller, and use XBee to establish communication between system and server.

STM32 was chosen since it provides several advantages for embedded developers. The STM32 Nucleo-32 boards provide an affordable and flexible way for users to try out new concepts and build prototypes with the STM32 microcontrollers. With Adruino Nano Connectivity, it is easy to expand the functionality of the STM32 Nucleo open development platform with a choice of specialized shields.

XBee is chosen as the communication device because it is a complete ecosystem of wireless modules, gateways, adapters, and software. One socket allows user to connect to IoT networks around the globe.

Air Quality 5 Click Sensor can detect number of different gases. It provides a wide range of evaluation with the errors being minimized. This click board uses a compact of MOS sensor with three fully independent sensing elements in one packet. Each of these sensors reacts with the specific type of gases as well as providing the corresponding gas readings.

We perform this thesis in order to record our work process during the whole course and to understand more about the concept of electronics component and how to use them to build an IoT system. This report resumes our total 120h work load during the course.

3 REQUIREMENTS SPECIFICATION

3.1 Device description

The product has to manage the property monitoring and evaluating. The product will be used inside small industry or middle size industry. The designed device can also be used in personal properties such as car, truck or any kind of transportations to calculate the amount of CO has been relieved also. The marketing area for product is world-wide. The product contains a CO measurement system that can be used to measure at the range from 1 to 1000 ppm.

3.2 Block diagram of device

Base on requirement description, we came up with the system component structure as can be seen under block diagram from figure 1 below.

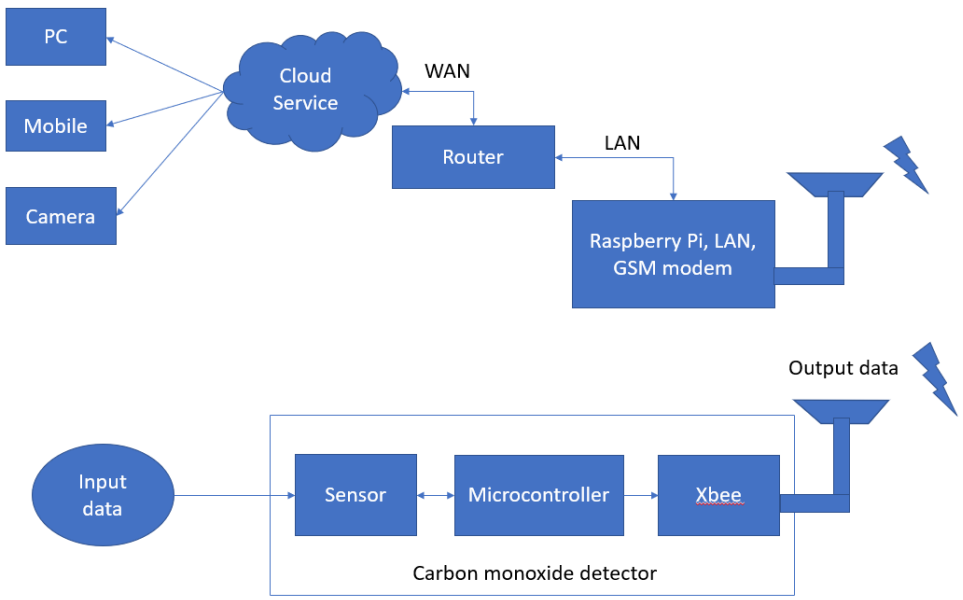


Figure 1. Block diagram describe structure of device

The main components were defined for both transmit side and receive side inside the whole system. At first, the input data will be read by sensor. There are three different sensing elements which reacts with eight different gases inside the click board. However, in this project, we will only use the sensing element that reacts

and provides data reading in CO. The data which were retrieved from sensor was under ADC format.

Then, sensor will send data to microcontroller (in this case is STM32) and microcontroller will convert this data from ADC to ppm format through equations implemented inside its software. After this process, data is converted in Xbee data frame format and automatically being sent to the server.

On the other side, after receiving data successfully, server will display data on screen using third-party devices. To enhance the convenience of experience, user can monitor corresponding information through these devices.

3.3 Hardware requirement

All of the hardware components need to be compatible and suitable with each other. The microcontroller needs to be able to communicate and retrieve data from sensor as well as transmitting data to Xbee. Besides sensor, microcontroller and Xbee as three main components, we also need to pay attention while choosing battery. The battery needs to have enough power supply for all electrical components and the whole system. A final circuit board needs to be small, effective and convenient. The whole hardware system should be able to work properly and fits inside a box.

3.4 Software requirement

The system will be implemented on STM32. It should be able to retrieve data from sensor and communicate to third party devices. The displayed data will be updated automatically and able for user to configure it. The software collects measured data and counts minimum amount of CO, maximum amount and average amount has been leaked into air.

XBEE CONFIGURATION

3.5 XBee characteristics

The Xbee RF modules utilize ZigBee protocols among a few other. Base on the description provided on Digi website, it is suitable for “applications requiring low latency and predictable communication timing”.

Figure 2. XBee Module Transceiver with Antenna

Compared to Wifi, it is definitely the less popular. However, I believe each of them has their own purposes. Practically the XBee module can be viewed as a modem because it mainly uses a UART (serial interface) to communicate with the main board. The advantage is simplicity and the possibility to re-use many serial tools. The main advantages of Xbee can be listed as lower power consumption, easier to install, reliable, flexible network structure and support a large number of nodes.

3.6 XBee in communication

The Xbee interfaces host device through a logic-level asynchronous serial port. Through this serial port, module can communicate with compatible UART. The module can also use a level translator to any serial device instead.

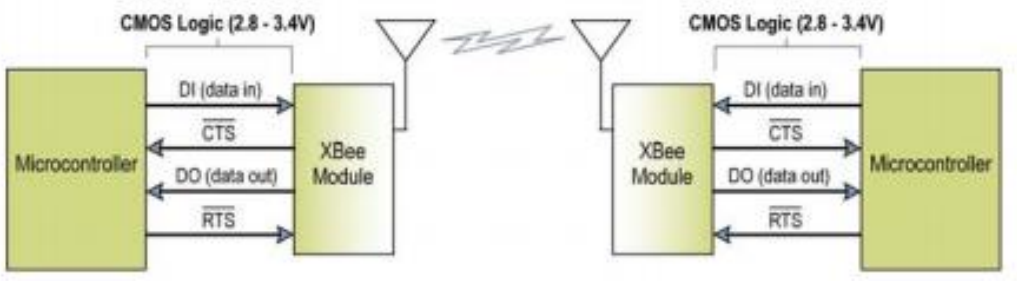


Figure 3. System Data Flow Diagram in a UART environment

Devices that includes UART interface can connect directly to the pins of RF module as shown in figure 3 above.

3.6.1 Flow of Data In (DI)

When serial data enters RF module through Data In (DI) pin, the data will be stored in DI Buffer until it is processed. The corresponding hardware flow control in this case is CTS (Clear To Send). When the DI buffer is 17 bytes away from being full, the module de-asserts CTS (high) to signal to the host device to stop sending data by default. CTS is re-asserted after the DI Buffer has 34 bytes of memory available.

In case DI buffer become full and possibly overflow, if the module is receiving a continuous stream of RF data, any serial data that arrives on the DI pin is placed in the DI Buffer. The data in the DI buffer will be transmitted over-the-air when the module is no longer receiving RF data in the network.

3.6.2 Flow of Data Out (DO)

When RF data is received, the data enters the Data Out (DO) buffer and is sent out the serial port to a host device. Once the DO Buffer reaches capacity, any additional incoming RF data is lost. The corresponding hardware flow control in this case is RTS (Request To Send). RTS is enabled for flow control data will not be sent out the DO Buffer as long as RTS (pin 16) is de-asserted.

However, we should notice that there are two cases in which the DO buffer may become full and possibly overflow

1. If the RF data rate is set higher than the interface data rate of the module, the module will receive data from the transmitting module faster than it can send the data to the host.
2. If the host does not allow the module to transmit data out from the DO buffer because of being held off by hardware or software flow control.

3.6.3 Xbee Serial Data

Data enters the module UART through the DI pin (pin 3) as an asynchronous serial signal. The signal should idle high when no data is being transmitted. Each data byte includes a start bit (low), 8 data bits (least significant bit first) and a stop bit (high). The data format can be seen under figure 4 below. In the example below, data transmitted is 0x1F (or 32 in decimal format)

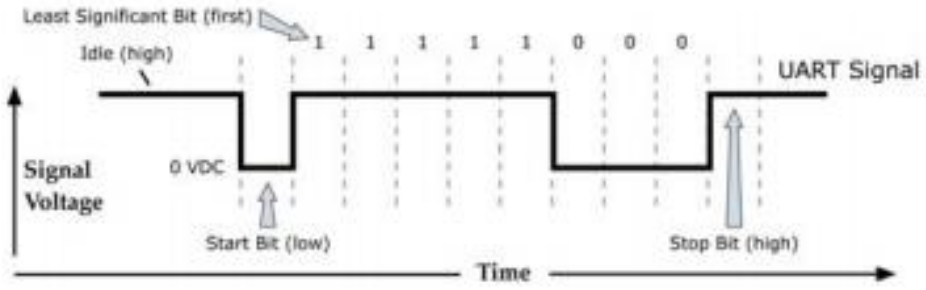


Figure 4. UART data transmitted through RF module

3.7 Xbee API data frame structure

By default, all UART data received through the DI pin is queued up for RF transmission. When the module receives an RF packet, the data is sent out the DO pin with no additional information. The API (Application Programming Interface) specifies how command responses and module status messages are sent and received from the module using a UART Data Frame.

The most typical form of UART data frame can be shown as in the figure 5 below.

Start delimiter	Length		Frame data: Frame type	Frame data: Data							Checksum
1	2	3	4	5	6	7	8	9	...	n	n+1
0x7E	MSB	LSB	API frame type	Frame-type-specific data							Single byte

Figure 5. UART Data Frame Structure

Any data received through the serial interface prior to the start delimiter is silently discarded by the XBee. If the frame is not received correctly, or if the checksum fails, the data is also discarded and the module indicates the nature of the failure by replying with another frame. So, in order to transmit data successfully, we need to convert the data that needed to be transmitted into correct data frame.

3.7.1 API data frame elements

There are four main data frame elements that should be include inside an API data frame in sequence which are start delimiter, data length, frame data and checksum.

Start delimiter: The start delimiter is the first byte of a frame consisting of a special sequence of bits that indicate the beginning of a data frame. Its value is always 0x7E. This allows for easy detection of a new incoming frame.

Length: The length field specifies the total number of bytes included in the frame data field. Its two-byte value excludes the start delimiter, the length, and the checksum.

Frame data: This field contains the information received or to be transmitted. Frame data is structured based on the purpose of the API frame. There are two elements inside frame data which are frame type and data. Frame type is the API frame type identifier. It determines the type of API frame and indicates how the information is organized in the Data field. The second part, data, contains the data itself. The information is included here and its order depends on the type of frame defined in the Frame type field

Checksum: Checksum is the last byte of the frame and helps test data integrity. It is calculated by taking the hash sum of all the API frame bytes that came before it, excluding the first three bytes (start delimiter and length).

3.7.2 Checksum calculation

To calculate the checksum of API frame, we need to follow these three main steps described below

Step 1: Add all bytes of the packet, excluding the start delimiter 0x7E and the length (the second and third bytes)

Step 2: From the given result, keep only the lowest 8 bits (two last bytes).

Step 3: Subtract this quantity from 0xFF

3.7.3 Xbee coding explain

According to theoretical steps explained onwards, we executed the coding as below. We need the module to send RF data as an RF packet, hence, we used TX Request message frame.

At first, we defined the unchanged variables inside an API data frame. The start delimiter is always 0x07E as default. We limited the number of bytes in the data frame as 16 bytes, so the length will be defined as

$$16 - 4 \ = 12$$

Convert to hexadecimal format, 12 is equal to C. Since we also applied TX Request case, the API identifier value is 0x01. The API frame ID is also defined as 0x01. Optional byte is not needed also, so it will be set as 0x00. The destination address is server's address destination.

```
unsigned char start_delimeter = 0x7E;
unsigned char length_MSB = 0x00;
unsigned char length_LSB = 0x0C;
unsigned char frame_type = 0x01;                //Transmit Request Frame, API identifier
unsigned char frame_id = 0x01;          //Identifies data frame to enable respond frame, API Frame ID
unsigned char option = 0x00;
unsigned char destination_add_MSB = 0xAB;
unsigned char destination_add_LSB = 0x01;
```

Figure 6. Data frame elements definition in code

Then, the next step is to calculate checksum and bind all data elements inside a data frame. To calculate checksum, we first add all bytes of the packet, excluding the start delimiter 0x7E and the length (the second and third bytes) in a variable called sum. Then, we take two bytes from given result in a variable called two_last_digit and subtract the quantity form 0xFF. The final value is checksum.

```
int sum2 = 0x00;
int sum1 = frame_type + frame_id + destination_add_MSB + destination_add_LSB + option;
for (int i = 0; i < strlen(dataHexa); i++) {
        sum2 += dataHexa[i];
}
int sum = 0;
sum = sum1 + sum2;
unsigned char two_last_digit = sum & 0xFF;
unsigned char checksum = 255 - two_last_digit;
```

Figure 7. Checksum calculation

The checksum calculation can be seen in figure 7 above. After calculating check-sum value, we will bind it into data frame as under figure 8 below

```
unsigned char message[16] = { start_delimeter, length_MSB, length_LSB,frame_type, frame_id, destination_add_MSB,
                              destination_add_LSB,option, 0, 0, 0, 0, 0, 0, 0, checksum };
for (int i = 0; i < 7; i++) {
        message[8 + i] = dataHexa[i];
}
HAL_UART_Transmit(&huart1, message, 16, 100);
```

Figure 8. Binding checksum into data frame

The whole function for this sending data process can be seen and sum up under figure 3.9 below

```
void send_to_xbee(char dataHexa[8]){
        int sum2 = 0x00;
        int sum1 = frame_type + frame_id + destination_add_MSB + destination_add_LSB + option;
        for (int i = 0; i < strlen(dataHexa); i++) {
                sum2 += dataHexa[i];
        }
        int sum = 0;
        sum = sum1 + sum2;
        unsigned char two_last_digit = sum & 0xFF;
        unsigned char checksum = 255 - two_last_digit;
        unsigned char message[16] = { start_delimeter, length_MSB, length_LSB,frame_type, frame_id, destination_add_MSB,
                                      destination_add_LSB,option, 0, 0, 0, 0, 0, 0, 0, checksum };
        for (int i = 0; i < 7; i++) {
                message[8 + i] = dataHexa[i];
        }
        HAL_UART_Transmit(&huart1, message, 16, 100);
        //HAL_UART_Transmit(&huart2, message, 16, 100);
}
```

Figure 9. Function sending data from XBee to server

The binding data in message needed to be transmitted includes all elements in an API data frame. This data frame will be fully transmitted to sever.

4 SENSOR CONFIGURATION:

The Click board contains the MiCS-6814 and ADS1015. ADS1015 is a low-power 12bit ADC with Internal reference, and programmable comparator. The MiCS-6814 sensor consists of three independent metal oxide sensors, heated by three separate heater structures.

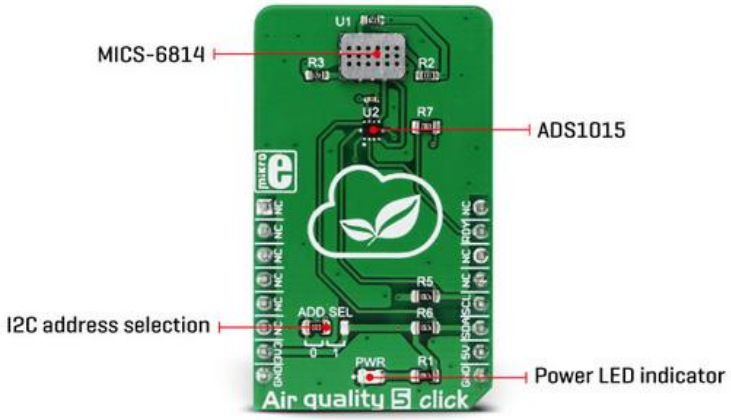


Figure 10. Air quality 5 click board

There are three sensors on the same die. Each of them reacts with a different type of gas. There is a RED sensor which reacts with the reducing gas agents, an OX sensor which reacts with the oxidizing gas agents, and lastly, a sensor which reacts with NH3.

Every heating structure is powered from the 5V power rail via the resistor, recommended by the manufacturer. This ensures the maximum life cycle of the device to be achieved since current ratings above the recommended would damage the sensors and heaters. It is recommended to pre-heat the sensors for at least 30 seconds before valid readings can be made. The longer the pre-heat period is, the more accurate the measurement becomes.

The changes of the sensor resistance are measured and sampled by the onboard ADC. The ADS1015 ADC has four multiplexed inputs, of which three are connected to each of the sensors. The ADC has an internal reference, it is very simple

to operate, it offers inputs that can handle voltages across the sensors and require a low number of external components. These attributes make it perfectly suitable for this Click board™. In addition, it is possible to change the slave I2C address of the device. This is done by using the SMD jumper, labeled as ADD SEL. This jumper allows selection of the I2C LSB bit state (0 or 1), allowing more than one Click board™ on the same I2C bus.

Both 5V and 3.3V rails from the are used. The ADC is powered by 3.3V rail, but the sensor requires 5V rail to be used as well. Therefore, the Click board™ requires both 3.3V and 5V pins to be supplied with the power.

Click board schematic:

The Air quality 5 click board consists of 2 main components: MiCS-6814 MOS sensor and ADS1015 12-bit ADC:

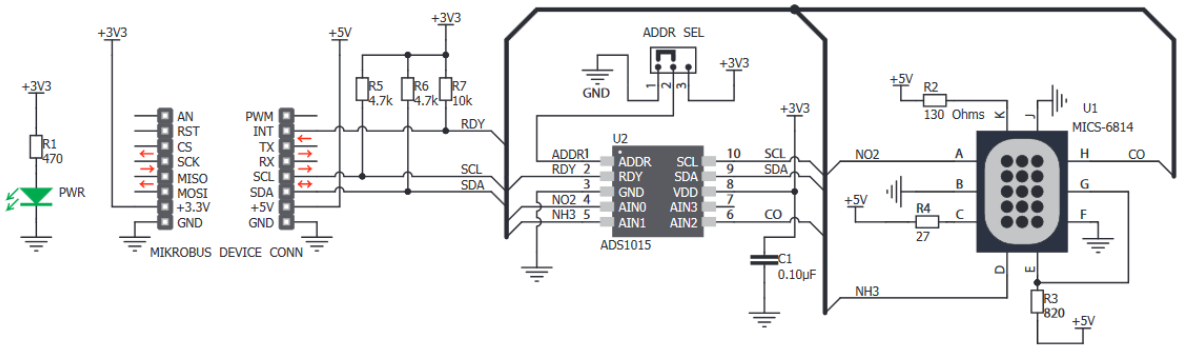


Figure 11. Air quality 5 click schematic v100.

4.1 MiCS-6814 MOS sensor:

The MiCS-6814 is a robust MEMS sensor for the detection of pollution from auto-mobile exhausts and for agricultural/industrial outdoors.

4.1.1 Features:

Smallest footprint for compact designs (5x7x1.55mm).

High-volume manufacturing for low-cost applications.

Short lead-times.

4.1.2 Detectable gases:

Carbon monoxide	CO	1-1000ppm
Nitrogen dioxide	NO_2	0.05-10ppm
Ethanol	C_2H_5OH	10-500ppm
Hydrogen	H_2	1-1000ppm
Ammonia	NH_3	1-500ppm
Methane	CH_4	>1000ppm
Propane	C_3H_8	>1000ppm
Iso-butane	C_4H_{10}	>1000ppm

Table 1. Detectable gas

MiCS-6814 can detect quite many gases with different range. For some gases, the range is sensitive enough to detect the gases in harsh environment.

Performance RED sensor:

Characteristic RED sensor	Symbol	Typ	Min	Max	Unit
Sensing resistance in air	R_o	-	100	1500	kΩ
Typical CO detection range	FS		1	1000	ppm
Sensing factor	S_{60}	-	1.2	50	-

Table 2. RED sensor's characteristics

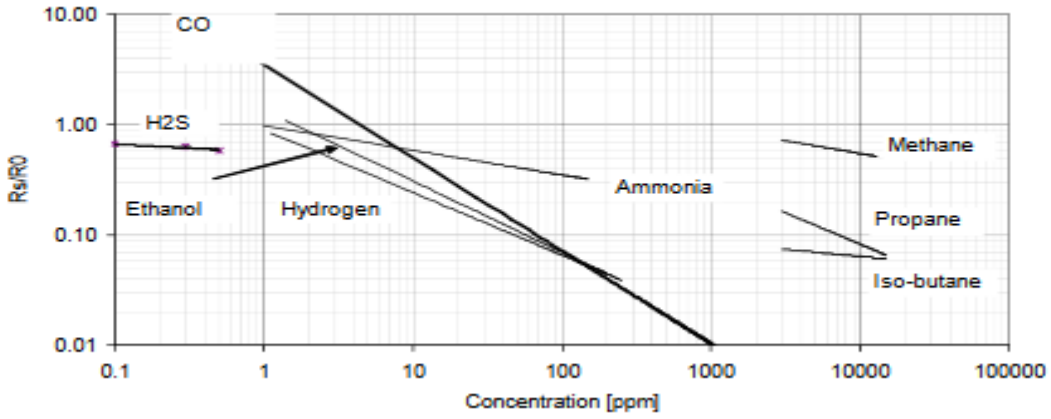


Figure 12. RED sensor, continuous power on, 25oC, 50% RH

The RED is used to detect Carbon Monoxide. For this type of sensor, the sensing resistance in air going from 100 to 1500 kΩ. The detection range is from 1 to 1000 ppm.

Performance OX sensor:

Characteristic OX sensor	Symbol	Typ	Min	Max	Unit
Sensing resistance in air	R_o	-	0.8	20	kΩ
Typical NO_2 detection range	FS		0.05	10	ppm
Sensing factor	S_R	-	2	-	-

Table 3. OX sensor's characteristics

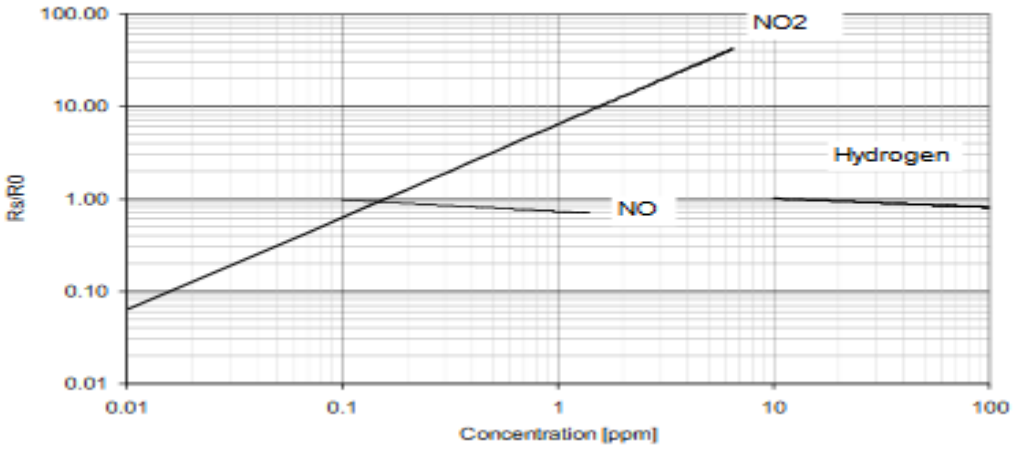


Figure 13. OX sensor, continuous power on, 25oC, 50% RH

The OX sensor is used to detect Nitrogen dioxide. For this type of sensor, the sensing resistance in air going from 0.8 to 20 kΩ. The detection range is from 0.05 to 10 ppm.

Performance NH3 sensor:

Characteristic NH3 sensor	Symbol	Typ	Min	Max	Unit
Sensing resistance in air	R_o	-	10	1500	kΩ
Typical NH3 detection range	FS		1	300	ppm
Sensing factor	S_R	-	1.5	15	-

Table 4. NH3 sensor's characteristics

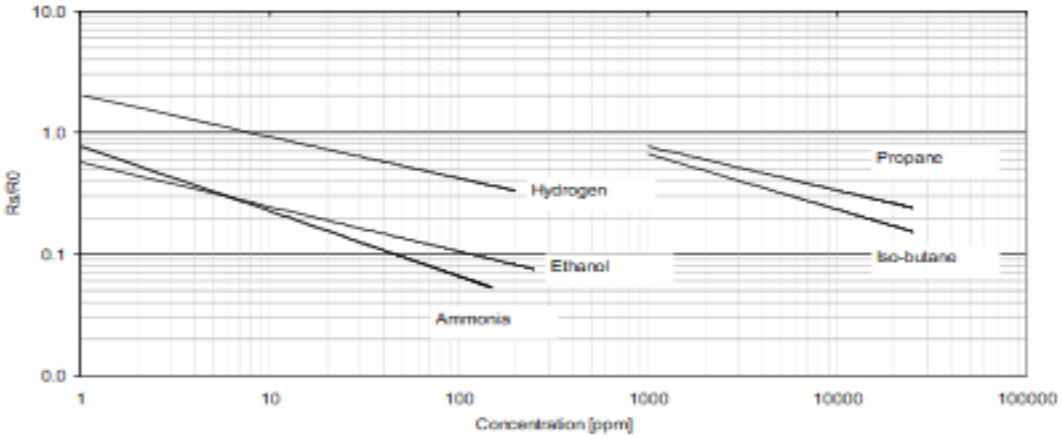


Figure 14. NH3 sensor, continuous power on, 25oC, 50% RH

The NH3 sensor is used to detect Ammonia. For this type of sensor, the sensing resistance in air going from 10 to 1500 kΩ. The detection range is from 1 to 300 ppm.

4.1.3 Heating parameter:

To get the most accurate measurement, the sensors need to be heated before reading. The table below displayed the heating power, heating voltage, heating current and heating resistance that needed for heating process.

Parameter RED sensor/OX sensor/NH3 sensor	Symbol	Typ	Min	Max	Unit
Heating power	P_H	76/43/66	71/30/60	81/50/73	mW
Heating voltage	V_H	2.4/1.7/2.2	-	-	V
Heating current	I_H	32/26/30	-	-	mA
Heating resistance at normal power	R_H	74/66/72	66/59/64	82/73/80	Ω

Table 5. Heating parameter of all sensors

In this project, we used **RED sensor** to measure the amount of **Carbon monoxide (CO)** in air.

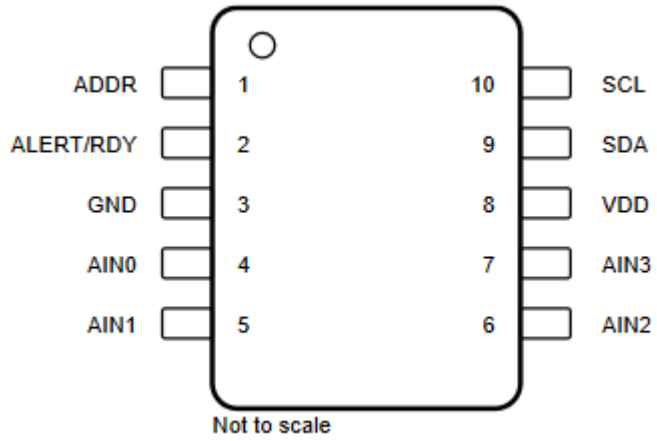
4.2 ADS1015 12-bit ADC:

ADS1015 is precision, low-power, 12-bit, I2C compatible, analog to digital converters offered in an ultra-small and leadless. The ADS1015 device incorporate a low-drift voltage reference and an oscillator. The ADS1015 also incorporate a programmable gain amplifier (PGA) and a digital comparator.

The ADS1015 performs conversions at data rates up to 3300 samples per second (SPS). The PGA offers input ranges from ±256mV to ±6.144V, allowing precise large and small signal measurements. The ADS1015 features and input multiplexer (MUX) that allows two differential or four single-ended input measurements. Use the digital comparator in the ADS1015 for under and over voltage detection

The ADS1015 operate in either continuous conversion mode or single shot mode. The device is automatically powered down after one conversion in single shot mode; therefore, power consumption is significantly reduced during idle periods

4.2.1 Pin Configuration and Functions:



PIN(1)				TYPE	DESCRIPTION
NAME	ADS1013	ADS1014	ADS1015		
ADDR	1	1	1	Digital input	I²C slave address select
AIN0	4	4	4	Analog input	Analog input 0
AIN1	5	5	5	Analog input	Analog input 1
AIN2	—	—	6	Analog input	Analog input 2 (ADS1015 only)
AIN3	—	—	7	Analog input	Analog input 3 (ADS1015 only)
ALERT/RDY	—	2	2	Digital output	Comparator output or conversion ready (ADS1014 and ADS1015 only)
GND	3	3	3	Analog	Ground
NC	2, 6, 7	6, 7	—	—	Not connected
SCL	10	10	10	Digital input	Serial clock input. Clocks data on SDA
SDA	9	9	9	Digital I/O	Serial data. Transmits and receives data
VDD	8	8	8	Analog	Power supply. Connect a 0.1-μF, power-supply decoupling capacitor to GND.

Figure 15. Pin configuration and functions.

Pins from AIN0 to AIN3 are the analog input which are connected to different pins on the sensors. ADDR pin used to slave the address of the I2C. SCL is the clock input of the I2C and SDA is used to transmit and receive data. VDD is power supply pins and in our case it is 3.3V.

4.2.2 Specifications:

4.2.2.1 Absolute Maximum Ratings:

Over operating free air temperature range:

		MIN	MAX	UNIT
Power-supply voltage	VDD to GND	–0.3	7	V
Analog input voltage	AIN0, AIN1, AIN2, AIN3	GND – 0.3	VDD + 0.3	V
Digital input voltage	SDA, SCL, ADDR, ALERT/RDY	GND – 0.3	5.5	V
Input current, continuous	Any pin except power supply pins	–10	10	mA
Temperature	Operating ambient, T_A	–40	125	°C
	Junction, T_J	–40	150	
	Storage, T_{stg}	–60	150	

Figure 16. Absolute Maximum Ratings

Figure 16 displays the power supply voltage and input voltage of the ADS1015. As being mentioned before, the supply voltage is 3.3V. The analog voltage will be from GND-0.3 to VDD+0.3. For the digital input voltage, it will go from GND-0.3 to 5.5V. The input current of all pins is similar (-10 to 10 mA) except for the power supply pins.

4.2.2.2 Recommended Operating Conditions:

The whole system will give the best performance under the following conditions:

		MIN	NOM	MAX	UNIT
POWER SUPPLY					
	Power supply (VDD to GND)	2		5.5	V
ANALOG INPUTS(1)					
FSR	Full-scale input voltage range(2) ($V_{IN} = V_{(AINP)} – V_{(AINN)}$)	±0.256		±6.144	V
$V_{(AINx)}$	Absolute input voltage	GND		VDD	V
DIGITAL INPUTS					
V_{DIG}	Digital input voltage	GND		5.5	V
TEMPERATURE					
T_A	Operating ambient temperature	–40		125	°C

Figure 17. Recommended Operating Conditions

(1) AINP and AINN denote the selected positive and negative inputs. AINx denotes one of the four available analog inputs.

(2) This parameter expresses the full-scale range of the ADC scaling. No more VDD+0.3V must be applied to the analog inputs of the device.

4.2.2.3 Timing Requirements I2C:

Over operating ambient temperature range and VDD=2V to 5.5V

		FAST MODE		HIGH-SPEED MODE		
		MIN	MAX	MIN	MAX	UNIT
f_{SCL}	SCL Clock Frequency	0.01	0.4	0.01	3.4	MHz
t_{BUF}	Bus free time between START and STOP condition	600		160		ns
t_{HDSTA}	Hold time after repeated START condition. After this period, the first clock is generated.	600		160		ns
t_{SUSTA}	Setup time for a repeated START condition	600		160		ns
t_{SUSTO}	Setup time for STOP condition	600		160		ns
t_{HDDAT}	Data hold time	0		0		ns
t_{SUDAT}	Data setup time	100		10		ns
t_{LOW}	Low period of the SCL clock pin	1300		160		ns
t_{HIGH}	High period for the SCL clock pin	600		60		ns
t_F	Rise time for both SDA and SCL signals(1)		300		160	ns
t_R	Fall time for both SDA and SCL signals(1)		300		160	ns

Figure 18. Timing Requirements

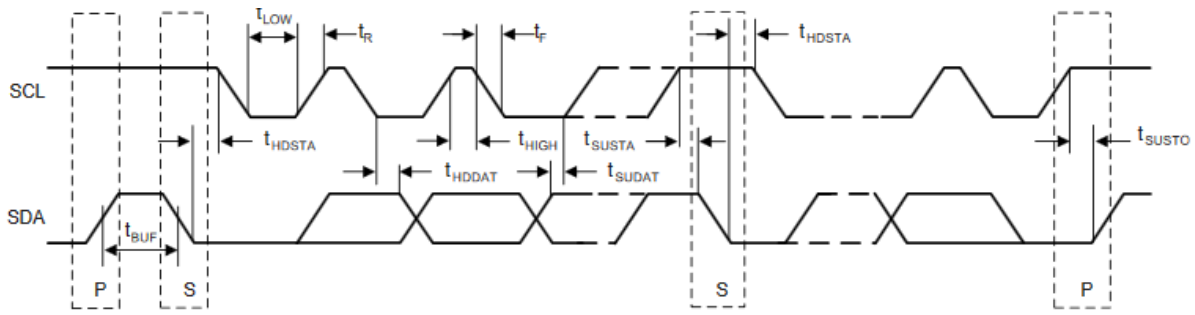


Figure 19. I2C interface timing

I2C has two different modes: fast mode and high-speed mode. These two modes have different clock frequency and time.

(1) For high-speed mode maximum values, the capacitive load on the bus line must not exceed 400 pF

4.2.3 Feature Descriptions:

4.2.3.1 Multiplexer:

The ADS1015 contains an input multiplexer (MUX), as shown in **Figure 20**. Either four single ended or two differential signals can be measured. Additionally, AIN0 and AIN1 may be measured differential to AIN3. When single ended signals are measured, the negative input of the ADC is internally connected to GND by a switch within the multiplexer.

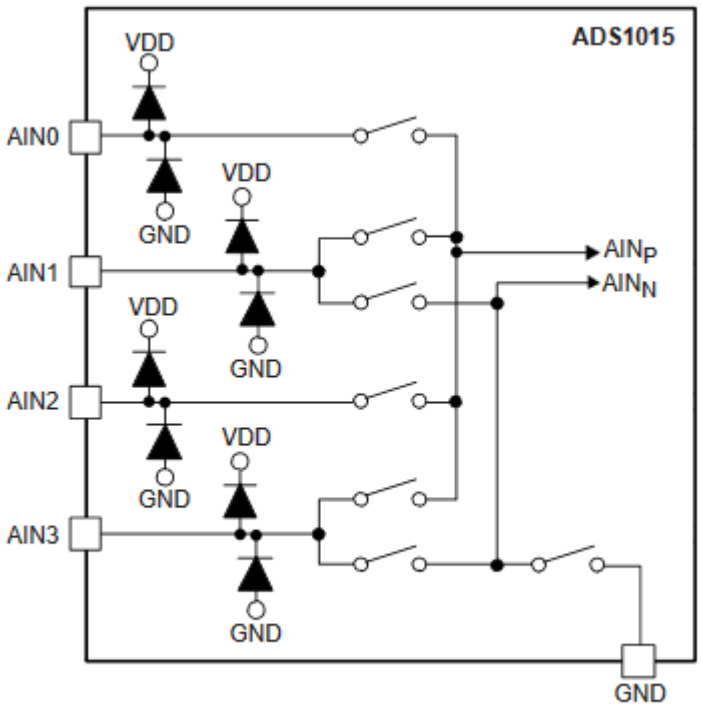


Figure 20. Input Multiplexer

Electrostatic discharge (ESD) diodes connected to VDD and GND protect the ADS1015 analog inputs. Keep the absolute voltage of any input within the range shown in the equation below to prevent the ESD diodes from turning on

$$GND - 0.3\ V < V_{(AINX)} < VDD + 0.3\ V$$

4.2.3.2 Full Scale Range (FSR) and LSB size:

A programmable gain amplifier (PGA) is implemented before the ΔΣ ADC of the ADS1015. The full-scale range is configured by bits PGA[2:0] in the Config register and can be set to ±6.144V, ±4.096V,±2.048V, ±1.024V, ±0.512V, ±0.256V. Equation below shows how to calculate the LSB size from the selected full-scale range.

FSR	LSB SIZE
±6.144 V(1)	3 mV
±4.096 V(1)	2 mV
±2.048 V	1 mV
±1.024 V	0.5 mV
±0.512 V	0.25 mV
±0.256 V	0.125 mV

(1) This parameter expresses the full-scale range of the ADC scaling. Do not apply more than VDD + 0.3 V to the analog inputs of the device.

Figure 21. Full scale range and corresponding LSB size

Analog input voltages must never exceed the analog input voltage limits given in the **Absolute Maximum Ratings.** If a VDD supply voltage greater than 4V is used, the ±6.144V full-scale range allows input voltages to extend up to the supply.

4.2.3.3 Oscillator:

The ADS1015 have an integrated oscillator running at 1 MHz. No external clock can be applied to operate these devices. The internal oscillator drifts over temperature and time. The output data rate scales proportionally with the oscillator frequency.

4.2.3.4 Output Data Rate and Conversion Time:

The ADS1015 offer programmable output data rates. Use the DR[2:0]bits in the **Config register** to select output data rates of 128 SPS, 250 SPS, 490 SPS, 920 SPS, 1600SPS, 2400SPS, or 3300SPS. Conversions in the ADS1015 settle within a single cycle; thus, the conversion time is equal to 1 / DR.

4.2.4 Operating Modes:

In this project, we used the **Single Shot Mode** as a solution to **reduce the power consumption** of the whole system.

Single Shot mode: When the MODE bit in the Config register is set to 1, the ADS1015 enter a power-down state, and operate in single-shot mode. This power-

down state is the default state for the ADS1015 when power is first applied. Although powered down, the devices still respond to commands. The ADS1015 remain in this power-down state until a 1 is written to the operational status (OS)bit in the Config register. When the OS bit is asserted, the device powers up in approximately 25μs, resets the OS bit to 0, and starts a single conversion. When conversion data are ready for retrieval, the device powers down again. Writing a 1 to the OS bit while a conversion is ongoing has no effect. To switch to continuous-conversion mode, write a 0 to the MODE-bit in the Config register.

4.2.5 I2C interface:

4.2.5.1 I2C Address Selection:

The ADS1015 have one address pin, ADDR, that configures the I2C address of the device. This pin can be connected to GND, VDD, SDA, or SCL, allowing for four different addresses to be selected with one pin, as shown in **Figure 22**.

ADDR PIN CONNECTION	SLAVE ADDRESS
GND	1001000
VDD	1001001
SDA	1001010
SCL	1001011

Figure 22. ADDR Pin connection and corresponding Slave Address

In this project, the ADDR pin is connected to the ground so the Slave address is **1001000.**

4.2.5.2 Writing to and Reading from the Registers:

To access a specific register from the ADS1015, the master must first write an appropriate value to register address pointer bits P[1:0] in the **Address Pointer register**. The Address Pointer register is written to directly after the slave address byte, low R/W bit, and a successful slave acknowledgement. After the Address Pointer Register is written, the slave acknowledges, and the master issues a STOP or a repeated START condition.

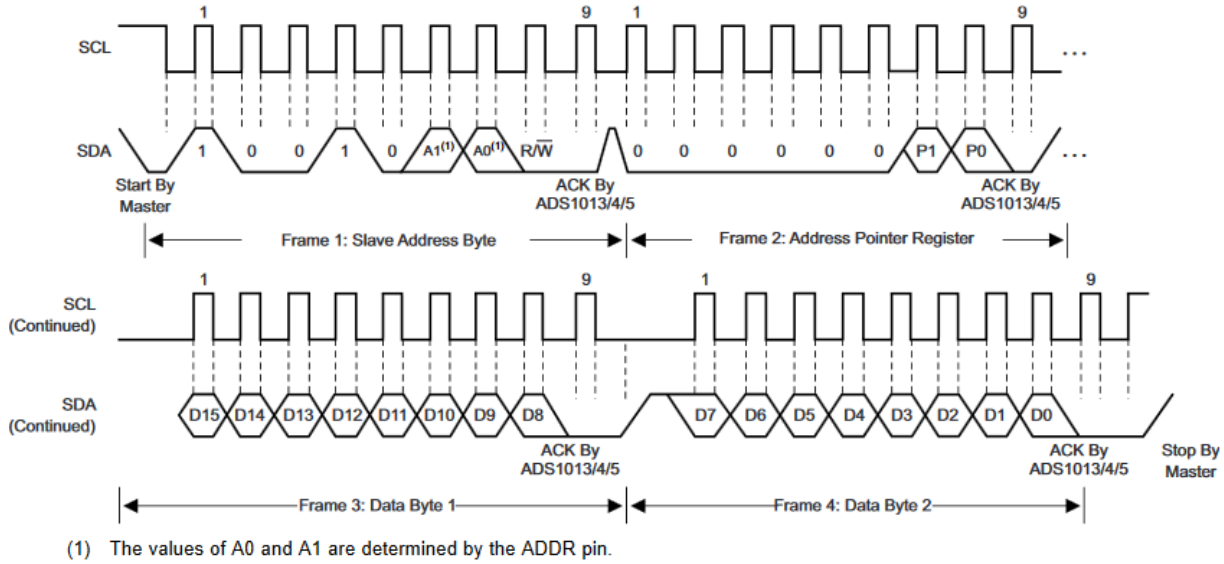
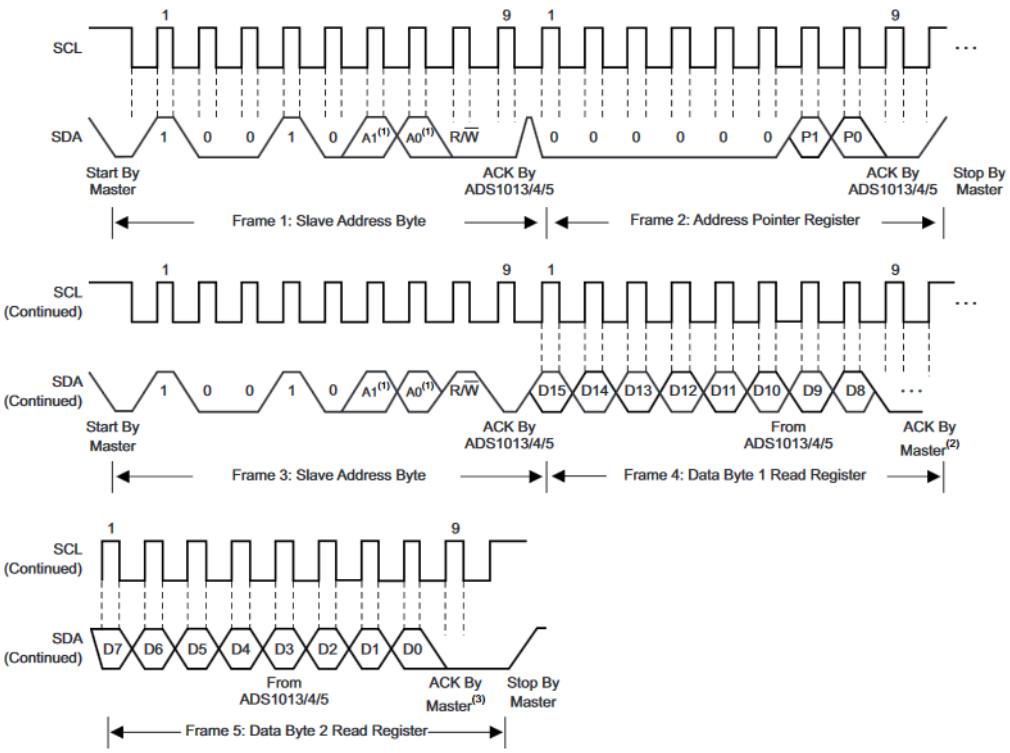


Figure 23. Timing Diagram for Writing to ADS1015

When reading from the ADS1015, the previous value written to bits P[1:0] determines the register that is read. To change which register is read, a new value must be written to P[1:0]. TO write a new value to P[1:0], the master issues a slave address byte with the R/W bit low, followed by the Address Pointer Register. If no additional data has to be transmitted, a STOP condition can be issued by the master. The master can issue a START condition and send the slave address byte with the R/W bit high to begin the read.



(1) The values of A0 and A1 are determined by the ADDR pin.
(2) Master can leave SDA high to terminate a single-byte read operation.
(3) Master can leave SDA high to terminate a two-byte read operation.

Figure 24. Timing Diagram for Reading from ADS1015

4.2.6 Register Map

4.2.6.1 Address Pointer Register

All four registers are accessed by writing to the Address Pointer register

Bit	Field	Type	Reset	Description
7:2	Reserved	W	0h	Always write 0h
1:0	P[1:0]	W	0h	**Register address pointer** 00 : Conversion register 01 : Config register 10 : Lo_thresh register 11 : Hi_thresh register

Figure 25. Address Pointer Register Field Descriptions

We used **Conversion register (0x00)** and **Config register (0x01)**

```
180                 ADSwrite[0]=0x01;
181                 HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 3, 100);
182                 ADSwrite[0]=0x00;
183                 HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 1, 100);
```

Figure 26. Transmit address pointer code

4.2.6.2 Config Register

Bit	Field	Type	Reset	Description
15	OS	R/W	1h	**Operational status or single-shot conversion start** This bit determines the operational status of the device. OS can only be written when in power-down state and has no effect when a conversion is ongoing. When writing: 0 : No effect 1 : Start a single conversion (when in power-down state) When reading: 0 : Device is currently performing a conversion 1 : Device is not currently performing a conversion
14:12	MUX[2:0]	R/W	0h	**Input multiplexer configuration (ADS1015 only)** These bits configure the input multiplexer. These bits serve no function on the ADS1013 and ADS1014. 000 : AIN_P = AIN0 and AIN_N = AIN1 (default) 001 : AIN_P = AIN0 and AIN_N = AIN3 010 : AIN_P = AIN1 and AIN_N = AIN3 011 : AIN_P = AIN2 and AIN_N = AIN3 100 : AIN_P = AIN0 and AIN_N = GND 101 : AIN_P = AIN1 and AIN_N = GND 110 : AIN_P = AIN2 and AIN_N = GND 111 : AIN_P = AIN3 and AIN_N = GND
11:9	PGA[2:0]	R/W	2h	**Programmable gain amplifier configuration** These bits set the FSR of the programmable gain amplifier. These bits serve no function on the ADS1013. 000 : FSR = ±6.144 V(1) 001 : FSR = ±4.096 V(1) 010 : FSR = ±2.048 V (default) 011 : FSR = ±1.024 V 100 : FSR = ±0.512 V 101 : FSR = ±0.256 V 110 : FSR = ±0.256 V 111 : FSR = ±0.256 V
8	MODE	R/W	1h	**Device operating mode** This bit controls the operating mode. 0 : Continuous-conversion mode 1 : Single-shot mode or power-down state (default)
7:5	DR[2:0]	R/W	4h	**Data rate** These bits control the data rate setting. 000 : 128 SPS 001 : 250 SPS 010 : 490 SPS 011 : 920 SPS 100 : 1600 SPS (default) 101 : 2400 SPS 110 : 3300 SPS 111 : 3300 SPS

Bit	Field	Type	Reset	Description
4	COMP_MODE	R/W	0h	**Comparator mode (ADS1014 and ADS1015 only)** This bit configures the comparator operating mode. This bit serves no function on the ADS1013. 0 : Traditional comparator (default) 1 : Window comparator
3	COMP_POL	R/W	0h	**Comparator polarity (ADS1014 and ADS1015 only)** This bit controls the polarity of the ALERT/RDY pin. This bit serves no function on the ADS1013. 0 : Active low (default) 1 : Active high
2	COMP_LAT	R/W	0h	**Latching comparator (ADS1014 and ADS1015 only)** This bit controls whether the ALERT/RDY pin latches after being asserted or clears after conversions are within the margin of the upper and lower threshold values. This bit serves no function on the ADS1013. 0 : Nonlatching comparator . The ALERT/RDY pin does not latch when asserted (default). 1 : Latching comparator. The asserted ALERT/RDY pin remains latched until conversion data are read by the master or an appropriate SMBus alert response is sent by the master. The device responds with its address, and it is the lowest address currently asserting the ALERT/RDY bus line.
1:0	COMP_QUE[1:0]	R/W	3h	**Comparator queue and disable (ADS1014 and ADS1015 only)** These bits perform two functions. When set to 11, the comparator is disabled and the ALERT/RDY pin is set to a high-impedance state. When set to any other value, the ALERT/RDY pin and the comparator function are enabled, and the set value determines the number of successive conversions exceeding the upper or lower threshold required before asserting the ALERT/RDY pin. These bits serve no function on the ADS1013. 00 : Assert after one conversion 01 : Assert after two conversions 10 : Assert after four conversions 11 : Disable comparator and set ALERT/RDY pin to high-impedance (default)

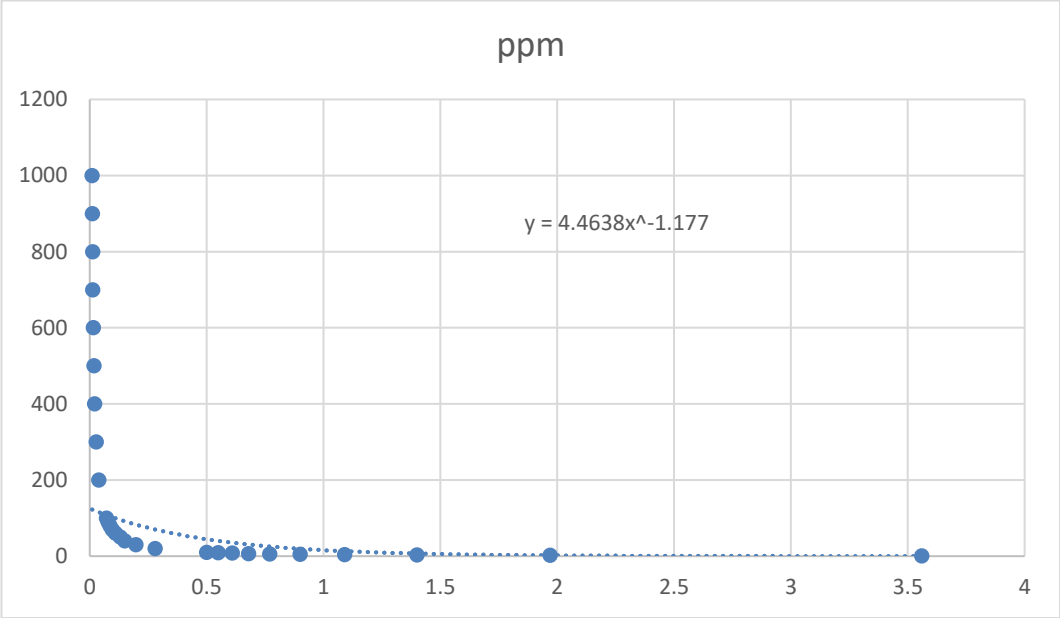
Figure 27. Config Register Field Descriptions

For the first 8 bits, we used the following register: **1 110 010 1 (0xE5);** which means Start a single conversion, the input multiplexers are AIN2 and GND, the FSR= ±2.048V, and run in single shot mode.

For the last 8 bits, we used the following register: **100 0 0 0 11 (0x83);** which means we used the default settings of the ADC.

4.3 Calculations

To exchange the ratio of Rs/Ro to PPM, we used Microsoft Excel. For each line on the **Figure 12**, we took data points manually from the diagram and put them in the spreadsheet. X axis represents for the ratio and Y axis represents for PPM value.



To determine the value of Ro, we calculate the average value of the ADC output after 30 seconds (reading 1) and used the following formulas:

$$V_{Ro} = 5 * \text{reading 1} * \frac{2.048}{2048}$$

$$I_{Ro} = \frac{V_{Ro}}{56}$$

$$Ro = \frac{\text{reading 1} * \frac{2.048}{2048}}{I_{Ro}}$$

To determine the value of Rs, we took the output of the ADC (reading 2) every second and used the previous formulas for reading 2 instead of reading 1.

After getting the ratio between Rs and Ro, we applied the ratio to x in the formula of the graph to find the value in ppm.

4.4 Code explanation

According to all the theory parts mentioned before, we started to write the code as below:

```
54  unsigned char ADS1015_ADDRESS=0x48;                    //1001000
55  unsigned char ADS1015_ADDRESS_write=0x90;       //10010000
56  unsigned char ADS1015_ADDRESS_read=0x91;        //10010001
57  unsigned char ADSwrite[6];
58  unsigned char ADSwrite2[6];
59  unsigned char received_data[6];
60  unsigned char received_data2[6];
61  int16_t reading;
62  int16_t reading2;
63  int16_t reading1=0;
64  int CO;
65  const float voltageConv=2.048/2048;
66  float VRo, VRs, IRo, IRs, Ro, Rs, ratio;
```

Figure 28. Declaration of variables

First, we declared the variables as in Figure 4.17. ADS1015_ADDRESS is the I2C slave address, ADS1015_ADDRESS_write is the address for the I2C to write to and ADS1015_ADDRESS_read is the address for the I2C to read from. ADSwrite and ADSwrite2 are arrays of the config register. For the output of the click board, we choose reading, reading1 and reading2 for different purposes. CO is the final value in ppm, voltageConv is the voltage conversion of the ADC and the rest are all the needed variables for calculation.

```
83  static void MX_USART2_UART_Init(void);
84  static void MX_USART1_UART_Init(void);
85  static void MX_I2C1_Init(void);


141   MX_USART2_UART_Init();
142   MX_USART1_UART_Init();
143   MX_I2C1_Init();
```

Figure 29. Functions declaration and init

Because our project used UART1 to send data to Xbee, I2C to read data from the sensor and UART2 to debug so here is where declare and init all the needed functions for communication protocols.

```
146     ADSwrite[0]=0x01;
147     ADSwrite[1]=0xE5;     //11100101
148     ADSwrite[2]=0x83;     //10000011
149     /* USER CODE END 2 */
150
151     for(int i=0; i<30; i++){
152             HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 3, 100);
153             ADSwrite[0]=0x00;
154             HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 1, 100);
155             //HAL_Delay(500);
156             HAL_I2C_Master_Receive(&hi2c1, ADS1015_ADDRESS_read, received_data, 2, 100);
157             reading = ((received_data[0] << 8) | received_data[1]) >> 4;
158             reading1+=reading;
159     }
160     reading1=reading1/30;
161     VRo=5-reading1*voltageConv;
162     IRo=VRo/56;
163     Ro=(reading1*voltageConv)/IRo;
```

Figure 30. Code for getting the value of Ro

As in figure 4.19, here is the part we get the ADC value to calculate Ro. First, we declare the value for ADSwrite. Secondly, we use HAL_I2C_Master_Transmit to write the config register to the writing address of the ADS1015. After that, we give ADSwrite[0] a new value and send it to the writing address again to inform the ADC that we are going to use conversion register. Then, the function HAL_I2C_Master_Receive is used to get the data from reading address of the ADC and store it into received_data array. Finally, we used the formula given by the manufacturer to get the right result; calculate the average value after 30 times and applied that to the formula in section 4.3 to calculate Ro.

```
166     while (1)
167     {
168     /* USER CODE END WHILE */
169                     char data[8]="";
170                     ADSwrite[0]=0x01;
171                     HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 3, 100);
172                     ADSwrite[0]=0x00;
173                     HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 1, 100);
174                     HAL_Delay(2000);
175                     HAL_I2C_Master_Receive(&hi2c1, ADS1015_ADDRESS_read, received_data2, 2, 100);
176                     reading2 = ((received_data2[0] << 8) | received_data2[1]) >> 4;
177                     VRs=5-reading2*voltageConv;
178                     IRs=VRs/56;
179                     Rs=(reading2*voltageConv)/IRs;
180                     ratio=Rs/Ro;
181                     CO=4.4638*pow(ratio, -1.177);
182                     if(CO < 1)
183                             CO=1;
184                     sprintf(data,"1+%d", CO);
185                     send_to_xbee(data);
186     /* USER CODE BEGIN 3 */
187
188     }
```

Figure 31. Code to get the value of Rs, CO and send final value to xbee

In this part, first we have to initialize the value of ADSwrite[0] again to 0x01 to write the config register to the writing address of the ADS1015. After that, the process is almost the same as for getting value of Ro. The data from reading address of the ADC is now stored in received_data2 array. Because this part of code is inside an infinite loop so we did not calculate the average of the value but applied reading2 into the formula given before to calculate Rs. According to that, we calculated the ratio between Rs and Ro and applied that to the formula of the graph (found in section 4.3 Calculations) to calculate the final result in ppm and send it to xbee using send_to_xbee function.

5 CIRCUIT BOARD BUILDING

5.1 Determining diagram of component inside black box

By using the AutoCAD, we can calculate the position of each components inside the black box, then using this diagram and dimension for making PCB purpose. The **Figure 32** below show how component take its place inside the box.

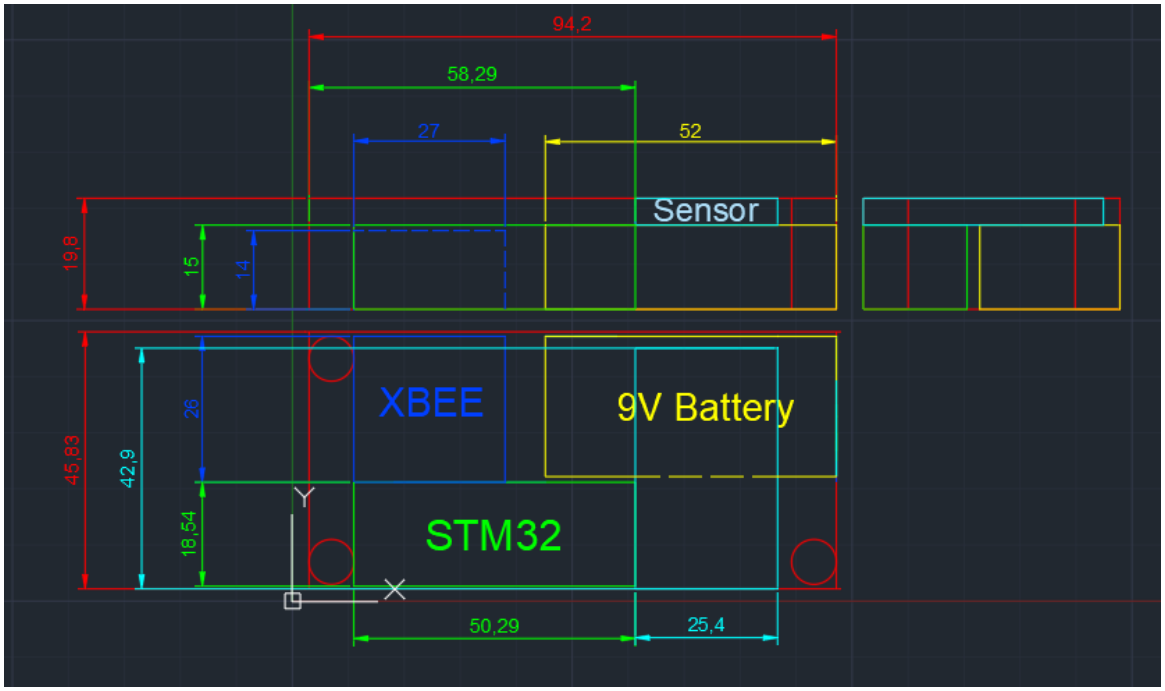


Figure 32. AutoCAD version of components diagram

As shown in picture, the CO sensor take place above the 9V battery, so that the block of component should be 2 layers. In the top layer, there is a PCB that connect CO sensor with the upper side of STM32 circuit. The second layer include another PCB that connect XBEE with the lower side of STM32 circuit. In additional, because lack of space inside the box so we have made decision of cutting 1 drill hole connection, which was taken its own place at 9V battery position.

5.2 Making wire connection with PADs logic

PADs logic program has been used for creating PCB purpose, which include the original connection between STM32, CO sensor and XBEE as shown in **Figure 33.**

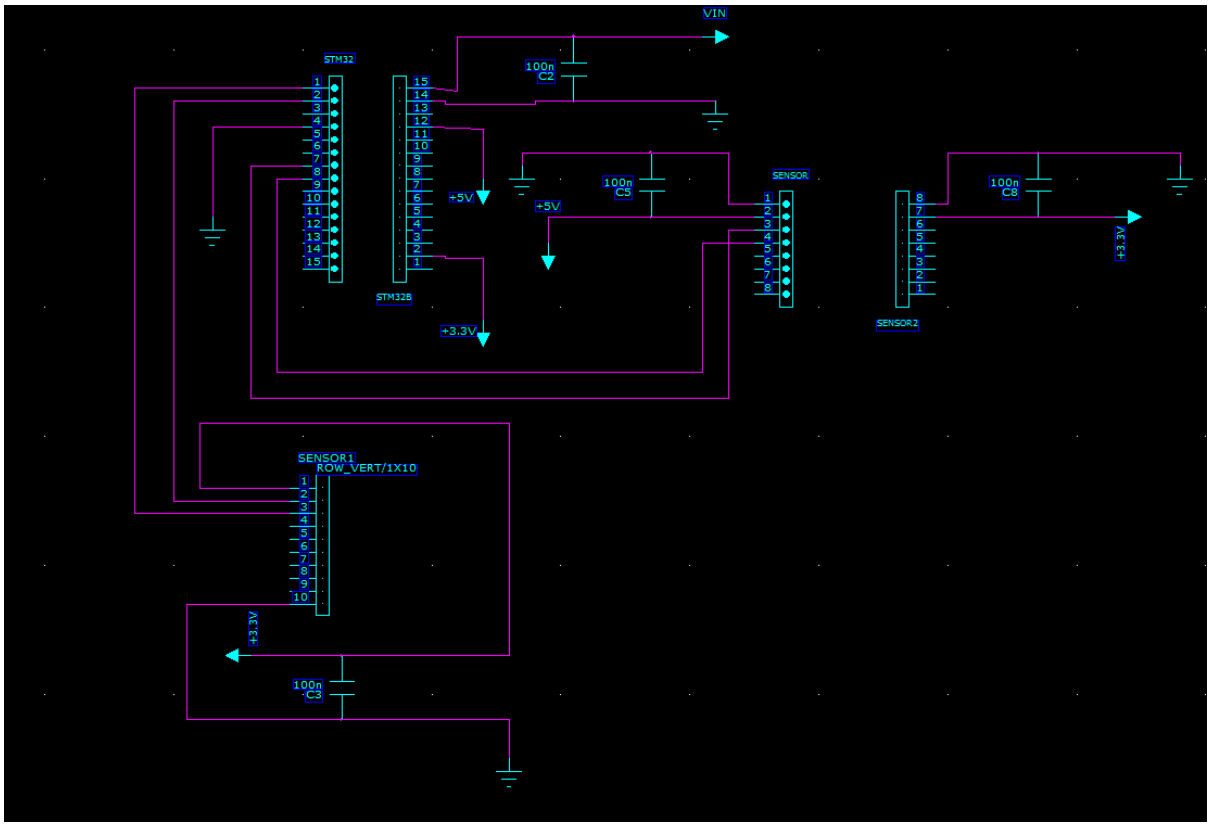


Figure 33. The original connection between 3 components

As shown in the picture above, we add one more capacitor for power source and for each component to prevent noise. In practice, we divide the original connection into 2 difference connection to create 2 difference PCBs.

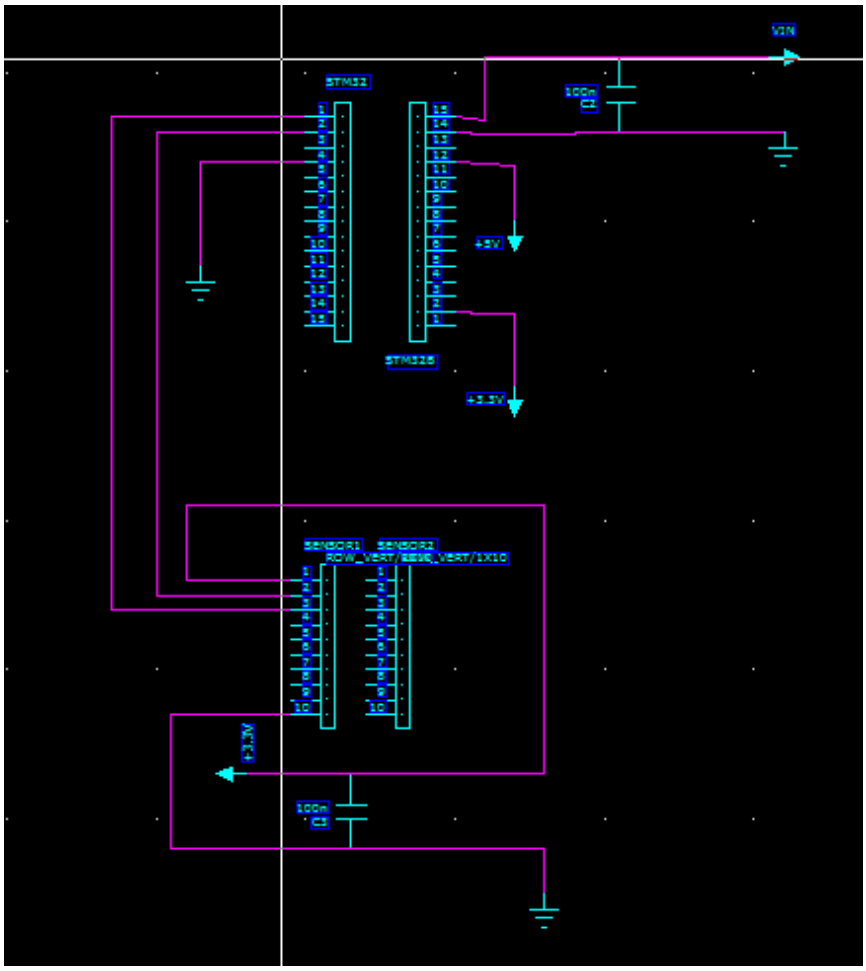


Figure 34. Connection between STM32 and XBEE

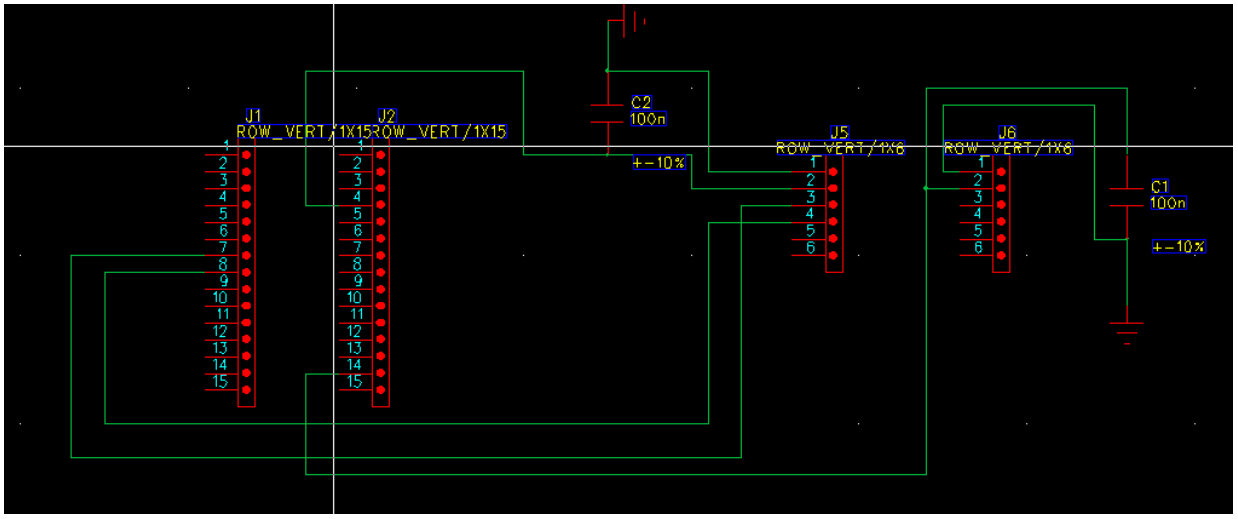


Figure 35. Connection between STM32 and CO sensor

5.3 Creating PCB in PADs layout

After making wire connection, the PCB is ready to build, the only problem is that there is no component sample for STM32, XBEE or CO sensor so we must customize new component in PADs layout with exactly dimension, size of drill hole and space between them, which can be found in user manual of each one.

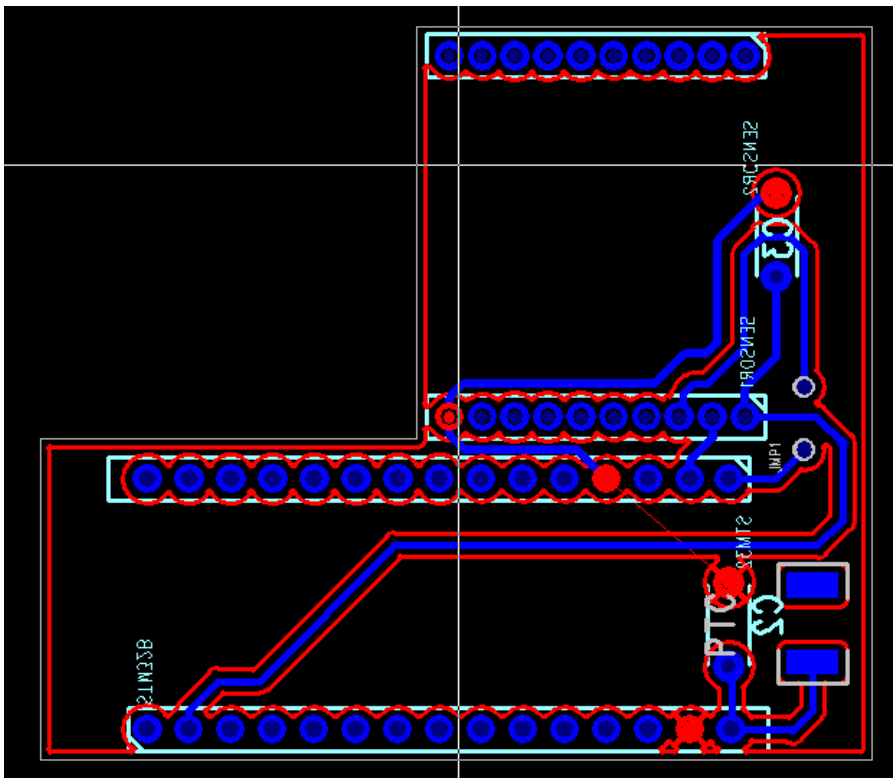


Figure 36. PCB connect STM32 with Xbee

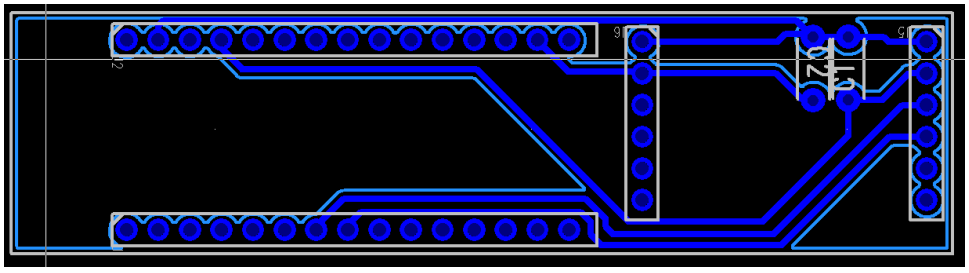


Figure 37. PCB connect STM32 with CO sensor

As shown in **Figure 36** and **Figure 37**, there are 2 PCBs with cooper pour in top layer. The connector position of STM32 in has been flipped in **Figure 36** because it connected to the lower pin of STM32.

5.4 Printed Circuit Board in reality

These figures below are our PCB in reality, it totally fit inside the box. The figures below will show our step how to connect each electronic component to the PCB and how to make it fit inside the box.

First, we need to place electronic devices to their correct place on board.

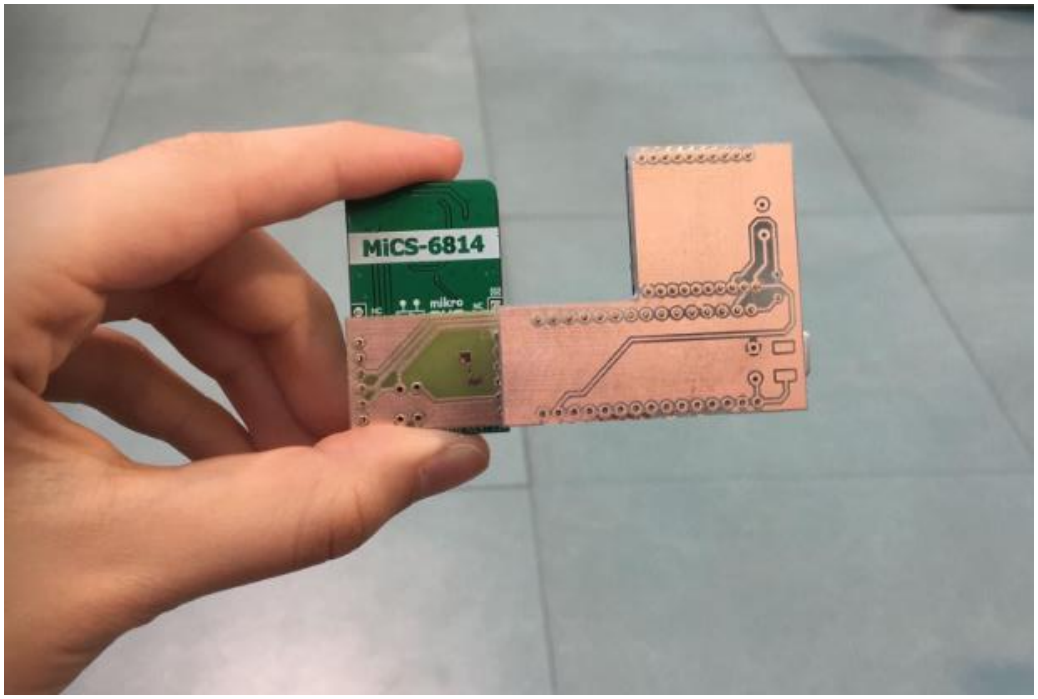


Figure 38. Implement Air Quality 5 Click Sensor on PCB

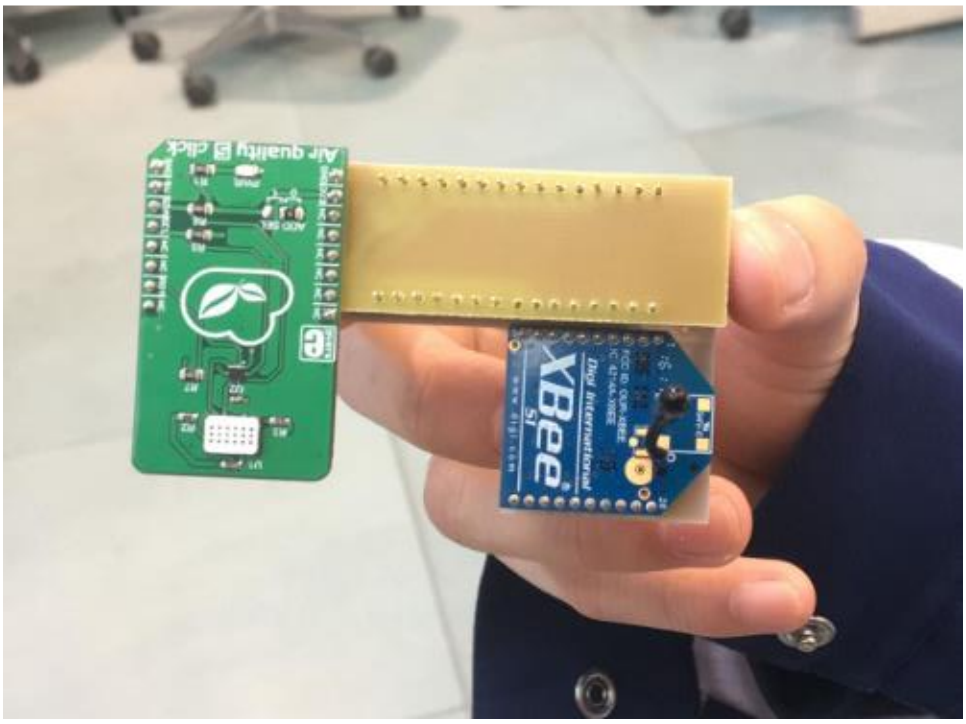


Figure 39. Implement XBee on PCB

The STM32 microcontroller is implemented as a conjunction device to connect to two PCB boards and merge the whole system

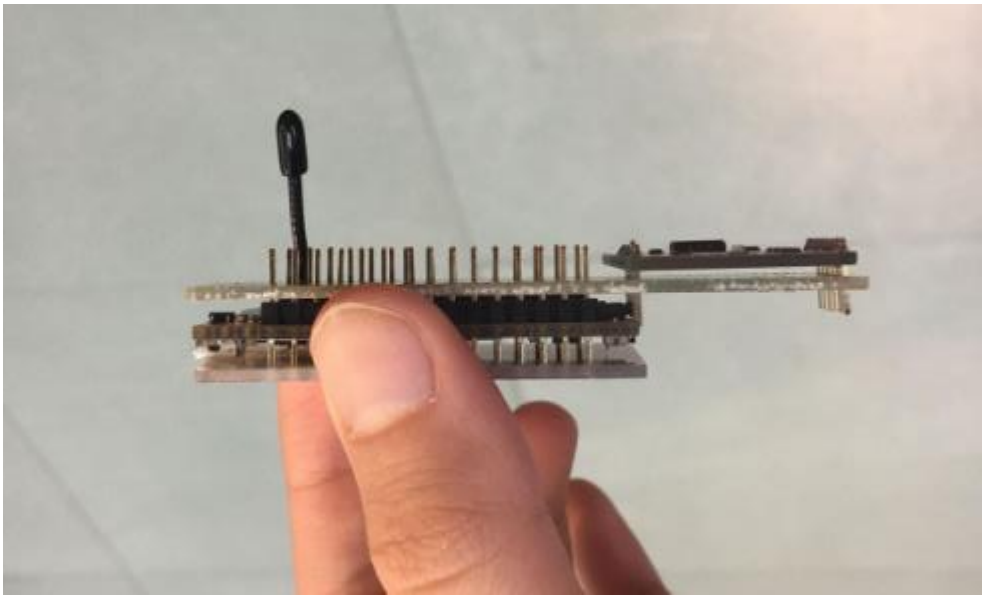


Figure 40. Implement STM32 on PCBs

Then, the battery will also be put in the box as a part of the system so that it can supply power to make system works independently

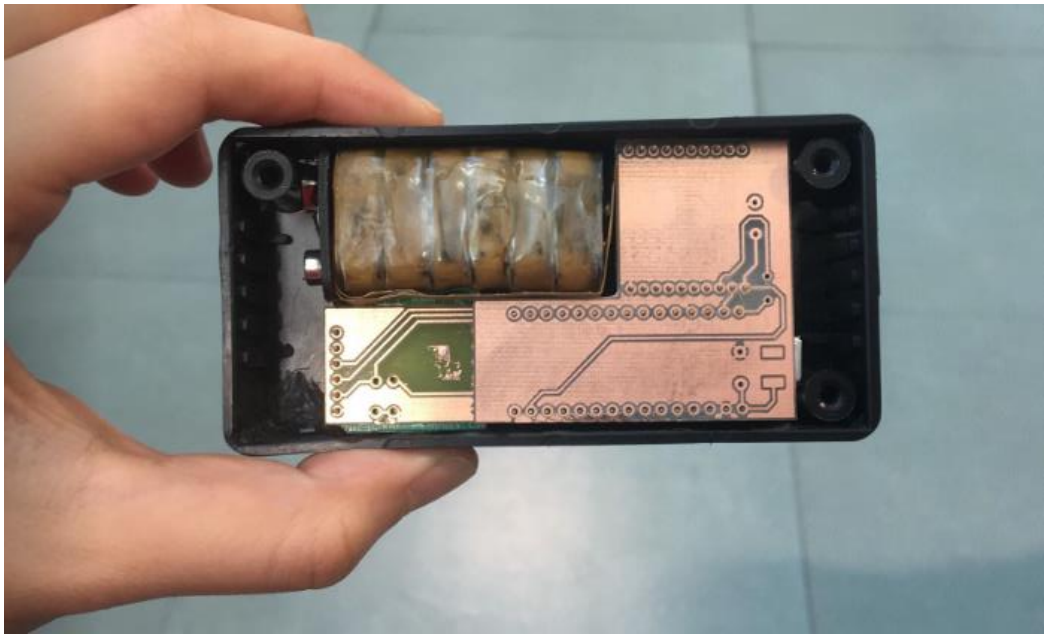


Figure 41. System from top view

Figure 42. System from bottom view

6 TEST ENVIRONMENT AND TEST RESULT

To test the system, we used a bowl and a candle to create an environment that lack of oxygen. When we burned the candle, the fire will slowly burn all the oxygen inside the bowl and left with carbon dioxide and carbon monoxide.

Figure 43. Test environment

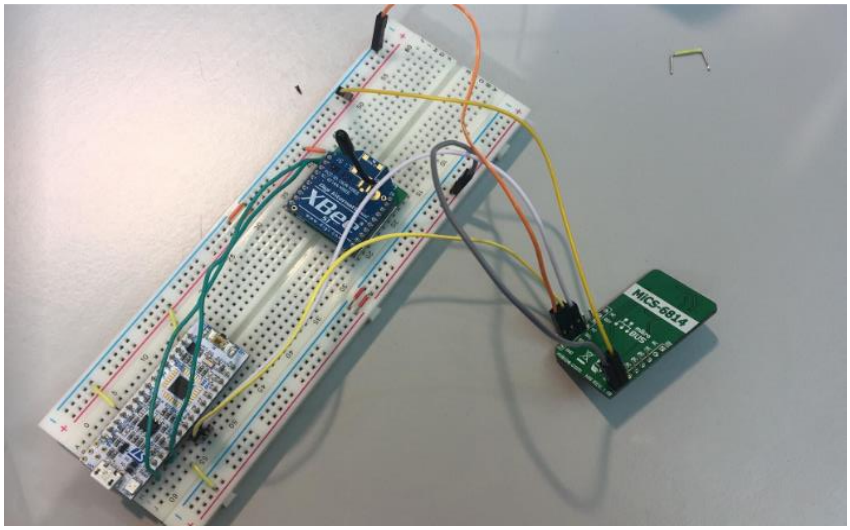


Figure 44. A closer look to the system implemented on breadboard

In figure 6.3, the sensor sent the result to the server. As you can see, in normal condition, the amount of ppm in air is about 3ppm

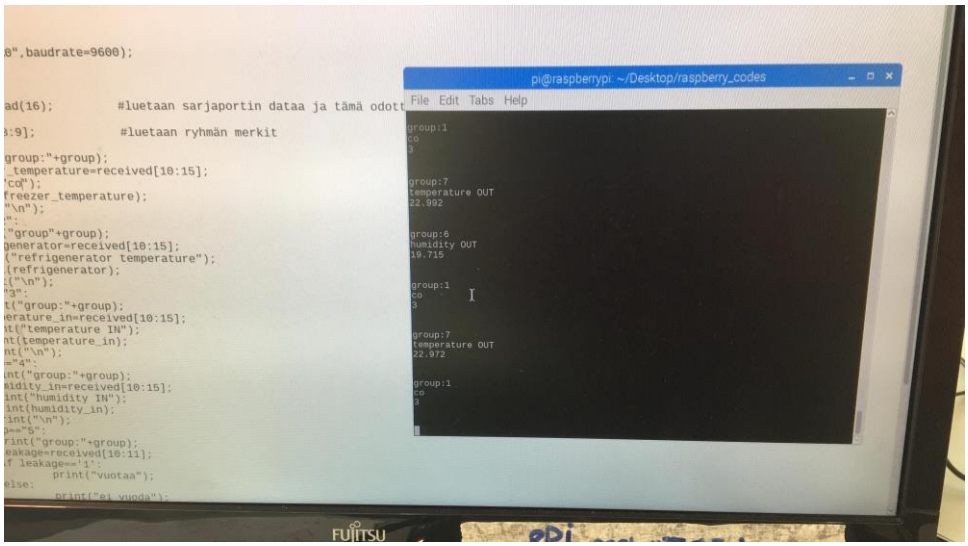


Figure 45. Test in normal environment

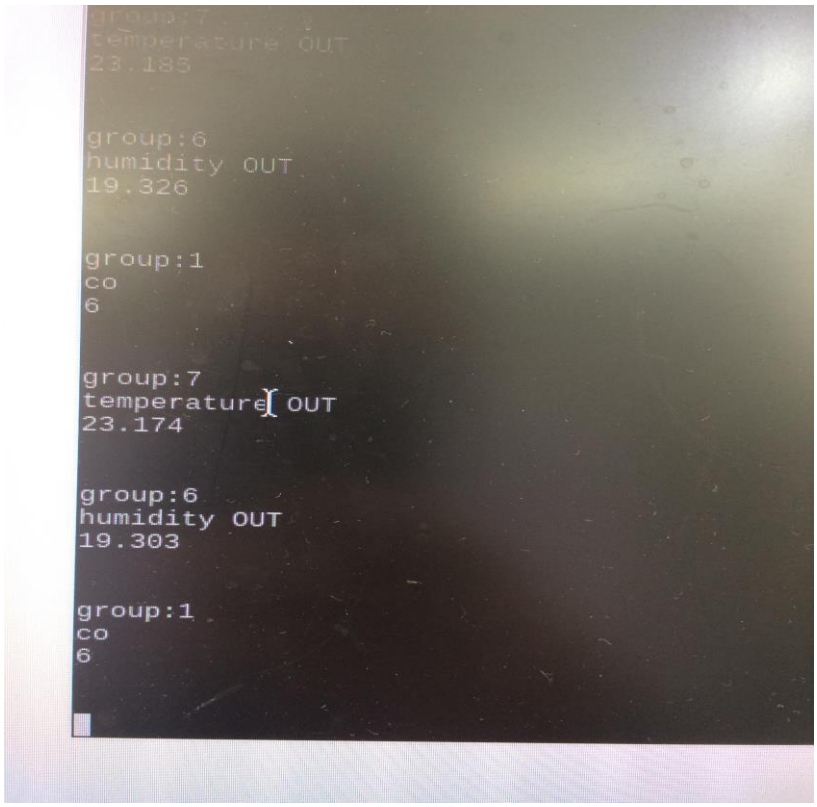


Figure 46. Test when the candle is slowly burn out

When the amount of oxygen in the bowl is low, the amount of carbon monoxide slowly increases from 3ppm to 6ppm.

The full software program will be attached on appendix as reference for this thesis.

7 CONCLUSION

The goal of this thesis was to design, implement and building software for CO detector and to understand the characteristics of this system inside an industry. This system is a simple format of Internet of Things application. It requires a endure power supply with low power consumption.

The Air Quality Control 5 Click Sensor provides a wide range to detect various kind of gases on air. It is not as popular as MQ-family but it is good enough to detect and evaluate in small and middle size industry. It can work independently and provide end to end process from detecting gases to sending data to user.

During the project, we learned a lot of detailed information about the process of building and designing a system from scratch, from choosing corresponding components to managing our working process. We also had a chance to enhance knowledge about embedded system programming and hardware implementation.

8 REFERENCES

- IITS2017-1 Embedded Systems Design
- https://www.st.com/en/evaluation-tools/nucleo-l432kc.html
- https://www.mikroe.com/air-quality-5-click
- https://download.mikroe.com/documents/datasheets/ads1013.pdf
- https://download.mikroe.com/documents/datasheets/103-21-394-008-EH-0915.pdf
- https://www.digi.com/resources/documentation/Digidocs/90001496/concepts/c_api_frame_structure.htm?TocPath=XBee%20API%20mode%7C_____2

9 APPENDICES

```
43
44  /* USER CODE BEGIN Includes */
45  unsigned char start_delimeter = 0x7E;
46  unsigned char length_MSB = 0x00;
47  unsigned char length_LSB = 0x0C;
48  unsigned char frame_type = 0x01;                //Transmit Request Frame, API identifier
49  unsigned char frame_id = 0x01;          //Identifies data frame to enable respond frame, API Frame ID
50  unsigned char option = 0x00;
51  unsigned char destination_add_MSB = 0xAB;
52  unsigned char destination_add_LSB = 0x01;
53
54  unsigned char ADS1015_ADDRESS=0x48;                     //1001000
55  unsigned char ADS1015_ADDRESS_write=0x90;       //10010000
56  unsigned char ADS1015_ADDRESS_read=0x91;        //10010001
57  unsigned char ADSwrite[6];
58  unsigned char ADSwrite2[6];
59  unsigned char received_data[6];
60  unsigned char received_data2[6];
61  int16_t reading;
62  int16_t reading2;
63  int16_t reading1=0;
64  int CO;
65  const float voltageConv=2.048/2048;
66  float VRo, VRs, IRo, IRs, Ro, Rs, ratio;
67  /* USER CODE END Includes */
68
69  /* Private variables ---------------------------------------------------------*/
70  I2C_HandleTypeDef hi2c1;
71
72  UART_HandleTypeDef huart1;
73  UART_HandleTypeDef huart2;
74
75  /* USER CODE BEGIN PV */
76  /* Private variables ---------------------------------------------------------*/
77
78  /* USER CODE END PV */
79
80  /* Private function prototypes -----------------------------------------------*/
81  void SystemClock_Config(void);
82  static void MX_GPIO_Init(void);
83  static void MX_USART2_UART_Init(void);
84  static void MX_USART1_UART_Init(void);
85  static void MX_I2C1_Init(void);
86
```

Appendix A. Variable and function declarations

```
void send_to_xbee(char dataHexa[8]){
int sum2 = 0x00;
int sum1 = frame_type + frame_id + destination_add_MSB + destination_add_LSB + option;
for (int i = 0; i < strlen(dataHexa); i++) {
sum2 += dataHexa[i];
}
int sum = 0;
sum = sum1 + sum2;
unsigned char two_last_digit = sum & 0xFF;
unsigned char checksum = 255 - two_last_digit;
unsigned char message[16] = { start_delimeter, length_MSB, length_LSB,frame_type, frame_id, destination_add_MSB,
destination_add_LSB,option, 0, 0, 0, 0, 0, 0, 0, checksum };
for (int i = 0; i < 7; i++) {
message[8 + i] = dataHexa[i];
}
HAL_UART_Transmit(&huart1, message, 16, 100);
//HAL_UART_Transmit(&huart2, message, 16, 100);
}
```

Appendix B. Code for Xbee to send data

```
/* USER CODE BEGIN 2 */
ADSwrite[0]=0x01;
ADSwrite[1]=0xE5;     //11100101
ADSwrite[2]=0x83;     //10000011
/* USER CODE END 2 */

for(int i=0; i<30; i++){
        HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 3, 100);
        ADSwrite[0]=0x00;
        HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 1, 100);
        //HAL_Delay(500);
        HAL_I2C_Master_Receive(&hi2c1, ADS1015_ADDRESS_read, received_data, 2, 100);
        reading = ((received_data[0] << 8) | received_data[1]) >> 4;
        reading1+=reading;
}
reading1=reading1/30;
VRo=5-reading1*voltageConv;
IRo=VRo/56;
Ro=(reading1*voltageConv)/IRo;
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */
                char data[8]="";
                ADSwrite[0]=0x01;
                HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 3, 100);
                ADSwrite[0]=0x00;
                HAL_I2C_Master_Transmit(&hi2c1, ADS1015_ADDRESS_write, ADSwrite, 1, 100);
                HAL_Delay(2000);
                HAL_I2C_Master_Receive(&hi2c1, ADS1015_ADDRESS_read, received_data2, 2, 100);
                reading2 = ((received_data2[0] << 8) | received_data2[1]) >> 4;
                VRs=5-reading2*voltageConv;
                IRs=VRs/56;
                Rs=(reading2*voltageConv)/IRs;
                ratio=Rs/Ro;
                CO=4.4638*pow(ratio, -1.177);
                if(CO < 1)
                        CO=1;
                sprintf(data,"1+%d", CO);
                send_to_xbee(data);
/* USER CODE BEGIN 3 */
```

Appendix C. Code for sensor to send data