IT Technology
Environmental Weather Station Unit system
Project Report

UNIVERSITY COLLEGE LILLEBAELT

Authors
Dainius Čeliauskas
Jacob Zobbe Mortensen
Jonathan Bar Rasmussen
Juan Mohamad

dain0084@edu.eal.dk
jaco973d@edu.eal.dk
jona9196@edu.eal.dk
juan0314@edu.eal.dk

Thursday, December 13, 2018

Abstract

This document describes a cheap, open source solution for measuring weather in different locations simultaneously using remote sensor systems.

Table of Contents

Table of Figures

I. Introduction

1. The Issue

Weather forecasting – arguably one of the most important applications of science and pattern-finding for any civilization. Its end uses are incredibly varied – forecasts based on temperature and precipitation are used in the agricultural and industrial sectors, as well as by individuals for deciding what to wear when going outside. Its history also spans millennia – the Babylonians predicted the weather from cloud patterns as far back as 650 BC.

Old weather forecasting methods usually relied on recognition of patterns – for example, if the sunset was particularly red, the next day often brought fair weather. Nowadays weather forecasts are done by collecting data about the current state of the atmosphere and using meteorology to predict any incoming changes to it. The data is gathered by measuring various properties of the atmosphere (temperature, humidity, pressure, etc.) using specialized equipment in a facility called a weather station.

Due to advancement of technology, electronic devices are decreasing in size more and more as time goes on – the same has applied to weather measurement equipment, leading to the invention of personal weather stations – automated instruments capable of measuring weather conditions using a vast array of sensors, while remaining relatively compact. Unfortunately, these devices are quite expensive to purchase and maintain, with little cheaper open-source alternatives available in the current market.

2. The Approach

The need for a customizable and cheap solution to weather measurement has sparked the creation and development for an alternative to current weather stations on the market. The end goal of this project is to develop and demonstrate a compact system that can measure temperature, humidity and CO2 level in the environment, all while being easy to set up and modify.

The system also combines weather measurement with the ability to remotely communicate with a central receiver unit using cutting-edge communication methods and lightweight messaging protocols. It makes it able to send measurements from multiple units simultaneously and periodically. Its design is flexible, allowing it to be configured and augmented even further.

3. Description and purpose

The project is a system consisting of multiple sensor boards capable of measuring temperature, humidity and CO2 level, controlled by ATmega328 microcontrollers, and a ATmega2560 receiver unit that remotely receives the data from the boards. The sensor units and the receiving units are communicating using LoRa, a digital wireless data communication technology developed in 2012.

The ATmega2560 can display the received data on an LCD display or through a wirelessly connected PC.

Some of the project’s features include:

- Periodic measurements of temperature, humidity and CO2 from multiple sets of sensors
- Sending periodic measurements of collected data remotely to an ATmega2560 receiver unit
- Displaying the data collected from the receiver unit or using it for further calculations
- Long range – The LoRa technology enables very-long-range transmissions (up to 15 km) while keeping power consumption low

The results of the project could have several potential applications, the most prominent one being a cost-effective alternative to currently available weather stations.

4. Aims and objectives

Primary objective: build a functioning set of sensor units capable of sending measured weather data remotely and periodically to a receiver unit.

Secondary objectives:

1: Build and program at least one system with temperature, humidity and CO2 sensors capable of periodically and accurately measuring relevant data.

2: Attach a LoRa module to a sensor system that can properly transmit the measured data.

3: Setup the receiver unit to be able to properly receive and process the sensor data.

5. Intended audience

People with at least basic knowledge of electronics and wireless communication.

6. Scope

This document includes short descriptions of the external hardware and software used in the project, as well as their relevancy. In addition, it provides the design choices for the project, problems encountered during development and the results of development. Finally, it discusses the development of the project and provides some recommendations regarding any potential future work for it.

7. Responsibilities

Name	**Responsibility**	**E-mail**
Dainius Čeliauskas	Software & Management	dain0084@edu.eal.dk
Jacob Zobbe Mortensen	Receiver unit setup	jaco973d@edu.eal.dk
Jonathan Bar Rasmussen	Communication setup	jona9196@edu.eal.dk
Juan Mohamad	Hardware & Part Supplying	juan0314@edu.eal.dk

8. Repository

This project has a GitHub repository, which contains the code used by the project and more: https://github.com/dainezasc/Environmental-Weather-Station-Unit-system

II. Background

This chapter describes the hardware components, software (scripts, libraries, etc.), protocols from sources outside the project and their purpose within it.

1. Hardware

1) AVR ATmega328

Figure 1: The ATmega328 microcontroller. Source from: https://www.microchip.com/wwwproducts/en/ATmega328

The ATmega328 is an 8-bit AVR RISC-based microcontroller containing 32KB flash memory with read-while-write capabilities, 1KB EEPROM, 2KB SRAM, 23 GPIO lines, 32 general purpose working registers, three flexible timer/counters with compare modes, internal and external interrupts, serial programmable UART, I2C, SPI serial port, 6-channel 10-bit A/D converter, programmable watchdog timer with internal oscillator, and five software selectable power saving modes.

This microcontroller is used in the Arduino Uno and Nano open-source development platforms. It is a simple, low-cost microcontroller that is commonly used in many other projects of a similar scope.

In the project the ATmega328 is the main microcontroller for the sensor boards. It is used to read measurements from the DHT22 and MQ135 sensors using its external interrupt and ADC pins, in addition to sending them through SPI to the SX1276 LoRa module for transmission.

2) AVR ATmega2560

Figure 2: The ATmega2560 microcontroller. Source from: https://www.microchip.com/wwwproducts/en/ATmega2560

The ATmega2560 is a huge performance but low power 8 bit AVR Microchip with a 256KB ISP flash memory 8KB SRAM, 4KB EEPROM, 86 general purpose I/O lines, 32 general purpose working registers, real time counter, six flexible timer/counters with compare modes, PWM, 4 USARTs, byte oriented 2-wire serial interface, 16-channel 10-bit A/D converter, and a JTAG interface for on-chip debugging. The device achieves a throughput of 16 MIPS at 16 MHz and operates between 4.5-5.5 volts.

This microcontroller is used in the Arduino Mega open-source development platform.

In the project the ATmega328 is the main microcontroller for the receiver unit. It is used to read the data received from the sensors using the LoRa module.

3) DHT22 (AM2302) temperature and humidity sensor

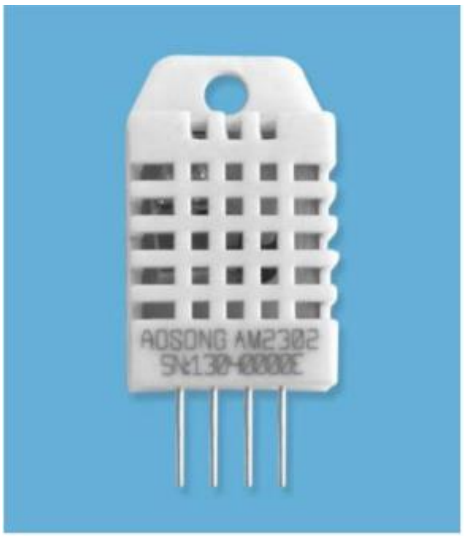


Figure 3: DHT22 sensor. Picture taken from datasheet: http://akizukidenshi.com/download/ds/aosong/AM2302.pdf

The DHT22 (also known as AM2302) is a module containing a capacitive humidity sensor and a high-precision temperature sensor. One of its most notable features is the inclusion of an 8-bit microcontroller responsible for calibrating and processing the collected measurements.

The sensor uses only 3 pins: VCC, GND and a digital data output pin used by the sensor’s single-bus interface. The interface is similar to I2C in design, featuring transmission of start/response signals and 40-bit chunks of data containing temperature, humidity and a parity bit. This design allows for incredibly easy and robust use with any device capable of processing the incoming data. In addition, its small size, low power consumption and signal transmission distance up to 20 meters makes it a very appealing choice for a wide range of applications.

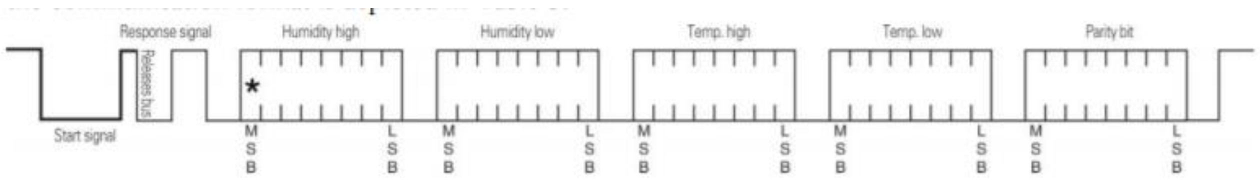


Figure 4: DHT22 single-bus interface. Picture taken from datasheet:
http://akizukidenshi.com/download/ds/aosong/AM2302.pdf

More information about the sensor and its single-bus interface can be found in its product manual (see References).

The DHT22 was used in the project to measure the ambient temperature and humidity periodically (every ~2 seconds), while sending its readings to the ATmega328 through the INT0 pin.

4) **MQ-135 gas sensor**

Figure 5: The MQ-135 gas sensor module that was used in the project. Sourced from:
https://www.trab.dk/da/sensorer/139-mq-135-luftkvalitets-sensor-co2.html

The MQ-135 gas sensors can be found in air quality control equipment and are suitable for detecting or measuring of CO2, NH3, smoke, alcohol, benzene, and other gases. The MQ-135 sensor module comes with a digital pin which makes this sensor able to operate without a microcontroller and is handy when only trying to detect one particular gas. If PPM measurement of gas is needed the analog pin can be used instead. The analog pin is TTL driven and works on 5V, therefore it can be used with most common microcontrollers. When reading the analog values (0-5V) using a microcontroller, the value will be directly proportional to the concentration of the gas to which the sensor detects.

The MQ-135 gas sensor works by using SnO2, which has higher resistance in clean air, as a gas-sensing material. When there is an increase in polluting gasses, the resistance of the sensor will decrease along with it. To measure PPM using MQ-135 sensor, it is needed to look into the (Rs/Ro) vs PPM graph taken from the MQ-135 datasheet.

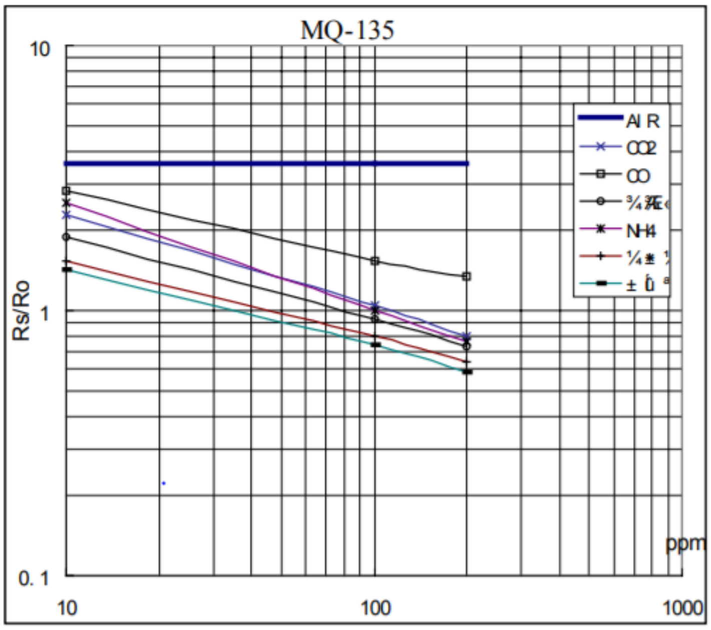


Figure 6: Sensitivity characteristics of the MQ-135. Sourced from the technical data manual: *https://www.olimex.com/Products/Components/Sensors/SNS-MQ135/resources/SNS-MQ135.pdf*

The above figure shows the typical sensitivity characteristics of the MQ-135 for several gasses while:

- Ambient temperature: 20 ℃,
- Ambient humidity: 65%,
- Ambient O2 concentration: 21%,
- Sensor load resistance (RL): 20kΩ,
- Ro: sensor resistance at 100ppm of NH3 in clean air,
- Rs: sensor resistance at various concentrations of gasses.

Calibration of the sensor is done by finding the values of Ro in fresh air, then using that value to find Rs, the value of resistance in gas concentration. Once Rs and Ro are found, the ratio between them can be found, then using the graph shown above the equivalent value of PPM for a particular gas can be calculated.

More information about the sensor and PPM measurement can be found in the datasheet (see References).

The MQ-135 was used in the project to detect CO2 PPM levels in the atmosphere, then passing the analog data to the ADC of the ATmega328.

5) **SX1276**

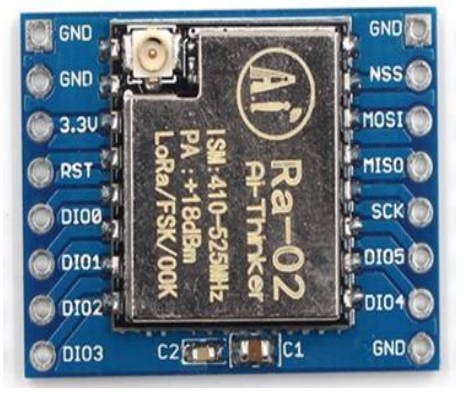


Figure 7: The SX1276 module used in the project. https://www.aliexpress.com/item/SX1278-LoRa-Module-433M-433MHz-10KM-Ra-02-Ai-Thinker-Wireless-Spread-Spectrum-Transmission-Socket-for/32860221098.html

This module was used for LoRa communication. It was cheap and known to work, since some of the other groups already had it up and running. The module should be powered with 3.3V but has a operating voltage between 1.8V - 3.7V.

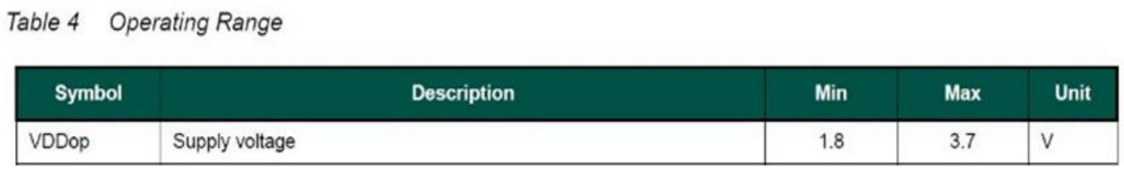


Table 4 Operating Range

Symbol	Description	Min	Max	Unit
VDDop	Supply voltage	1.8	3.7	V

Figure 8: Operating voltage of the module. Sourced from LoRa module datasheet

The module needs an antenna like the one below to function. Even though SX1276 actually has a built in antenna, it is not very useful as it has very short range.

Figure 9: Antenna used for the module.

6) **Other hardware components**

Other hardware components used in the project:

- Assortment of passive electronic components (capacitors, resistors, connectors, etc.)
- 5V power supply
- TS2937 3.3V voltage regulator

2. External software and protocols

1) SPI

SPI is a protocol used for electronic communication. It is very popular as it is easy to configure, it is a very fast way of communicating and it allows a master device to control multiple slave devices. It is used between the LoRa module and the ATmega328, both for configuring and for data transfer.

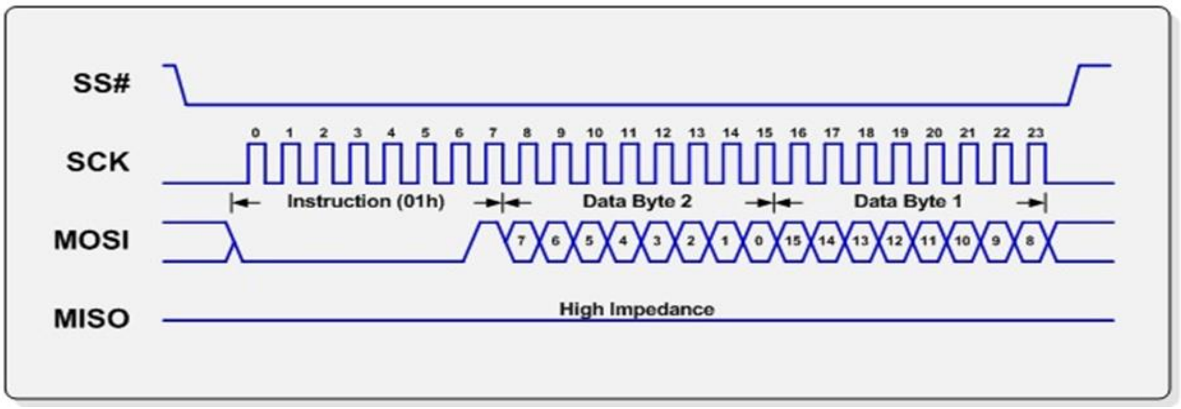


Figure 10: SPI transmission diagram. https://www.corelis.com/education/tutorials/spi-tutorial/

SS: Slave Select. It's active low, and the pin that initialize the transition.

SCK: Source clock. Usually the master outputs the SCK for the communication.

MOSI: Master Out, Slave In. Data pin.

MISO: Master In, Slave Out. Data pin. In this case MISO pin goes to a high impedance state, blocking all incoming signals.

2) DHT22 Interrupt driven library for AVR

This library can read the temperature and humidity values from the DHT22 sensor without using delay functions. By using the library, all the timing measurements of the sensor signal are done with a timer and a external interrupt pin. This way the processor of the microcontroller is not blocked when reading the sensor.

By default the library requires an AVR microcontroller that has:

- A pin with external interrupt (INT0, INT1 or other).
- A timer with Clear Timer on Compare Match mode (CTC).
- Timer prescaler that gives a timer frequency of 1MHz.

A timer in CTC is used to generate the host start condition. Pin is configured as output (this is done in the timer interrupt handler function). Then, the pin is switched to input with external interrupt. At each external interrupt the number of timer ticks (configured to occur at each microsecond) is counted.

The value of the counter (microseconds) is compared with a fixed value in a state machine, this way, the signal from DHT22 is interpreted. This is done at the external interrupt handler.

For more information about the library and a code example, see “DHT22 library” in References.

This library was used to read the temperature and humidity data from the DHT22 sensor, as a way to speed up project development.

3) MQ135 sensor library for AVR ATmega

This library can read the CO2 PPM levels recorded by the MQ135 sensor by utilizing the ADC of the ATmega328 and reading the sensor’s analog output.

As mentioned in the previous section on “MQ-135 gas sensor”, the sensor measures the PPM levels of certain gasses by adjusting its resistance (the Rs/Ro ratio). To read the sensor's resistance, measuring the voltage drop of the sensor is required. The method used to measure resistance is by using the ADC of the ATmega, and a pull-up (or pull-down) resistor.

For more information about the library, the calculations, code examples and more, see “MQ135 library” in References.

This library was used to read the PPM from the gas sensor, as a way to speed up project development.

4) LoRa

LoRa (LongRange) is a frequency shifting, modulation technique that allows long range communications. As the communication has a narrow bandwidth at a high frequency, it has a very low power consumption, but this also minimizes the amount of data the modem can send per time. This makes LoRa communication for suitable for application like sensor units, where the data rate is “low” and often, sensors are used in applications where a long battery lifetime is desired.

The modulation technique encrypts the data in a way that protects the signal from being interrupted by other signal and other types of interference.

Below are explanations of some different terms used when talking LoRa. It is important to understand these terms, in order to error troubleshoot and to make a proper setup.

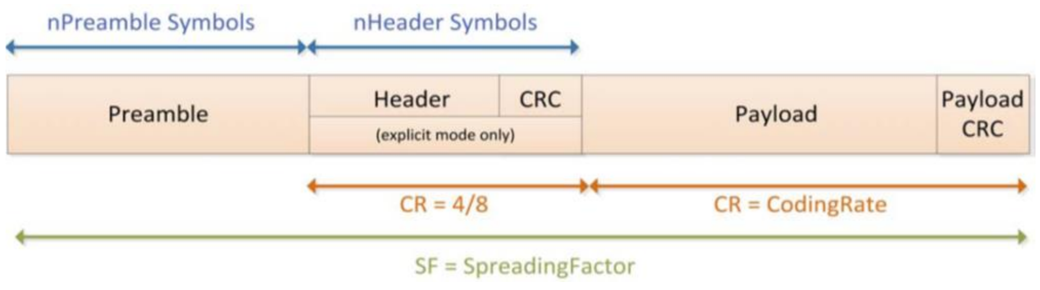


Figure 11: Example of a full length LoRa transmission. Sourced from LoRa module datasheet

Preamble is the first part of every LoRa transmission. The preamble is used for synchronizing the transmitting and receiving modules, kind of like the “handshake” for a transmission sequence. The **Header** is an optional part of the transmission. If the user chooses to include a header in the transmission, it will consist of information about the payload. Like the length of the payload and the error correction rate.

Payload: The actual data. Payload is transmitted after Preamble. **Chirp**: A signal that either rises or decay linear in frequency, Up-chirp when rising and Down-chirp when decaying. Signal doesn’t rise above the bandwidth threshold. **Bandwidth** is the frequency spectrum the signal can change within. A typical bandwidth setting is 125 KHz.

The **Spreading factor** represents the number of symbols sent per transmission.

The SF value is: 2√symbols. So, SF of 6 is equal to 2^6 which is 64 bits. SF must be set identical on both receiving and transmitting end, in order to work properly.

CRC is the payload error correction used for the communication.

CR stands for coding rate. The rate of data received except for the preamble.

Beneath is a figure of an actual LoRa transmission. On the X-axe we have time in ms and on the y-axe we have the frequency in KHz. From 0ms, there is 8 up-chirps, which is the preamble. At approx. 16,5ms, there is 2 down-chirps (which usually indicates the end of the preamble and the begin of payload) and then there is the payload, which in this case is 5 symbols.

“ETSI has defined 433 to 434 MHz frequency band for LoRa application. It uses 433.175 MHz, 433.375 MHz and 433.575 MHz frequency channels”.[1]
In this case it seems like the bandwidth is set to 125KHz and every time the chirp reaches the limit, the start frequency resets to 433MHz.

[1] Quoted from http://www.rfwireless-world.com/Tutorials/LoRa-frequency-bands.html

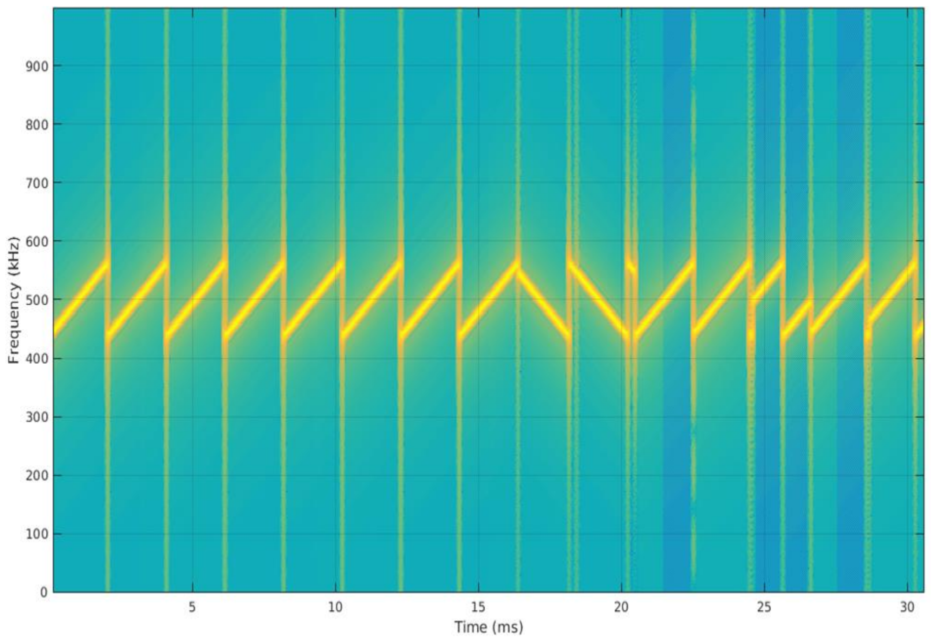


Figure 12: Example of an actual LoRa transmission. Sourced from LoRa module datasheet

5) **FIFO Buffer**

The SX1276 chip has a 256-byte RAM data buffer for incoming and outgoing data. The FIFO data buffer is where data are stored before later extracted through SPI to the ATmega328 chip.

The RegFIFOTXBaseAddr informs about the base address in the register for transmitting data. The same goes for the RegFIFORxBaseAddr, but instead of showing the base address for transmitting data, it shows it for the received data. The RegFIFORxCurrentAddr shows the location of the last package received by the FIFO. Before writing and reading to and from the FIFO buffer through the SPI, the FIFOAddrPtr needs to be set to the corresponding base address.

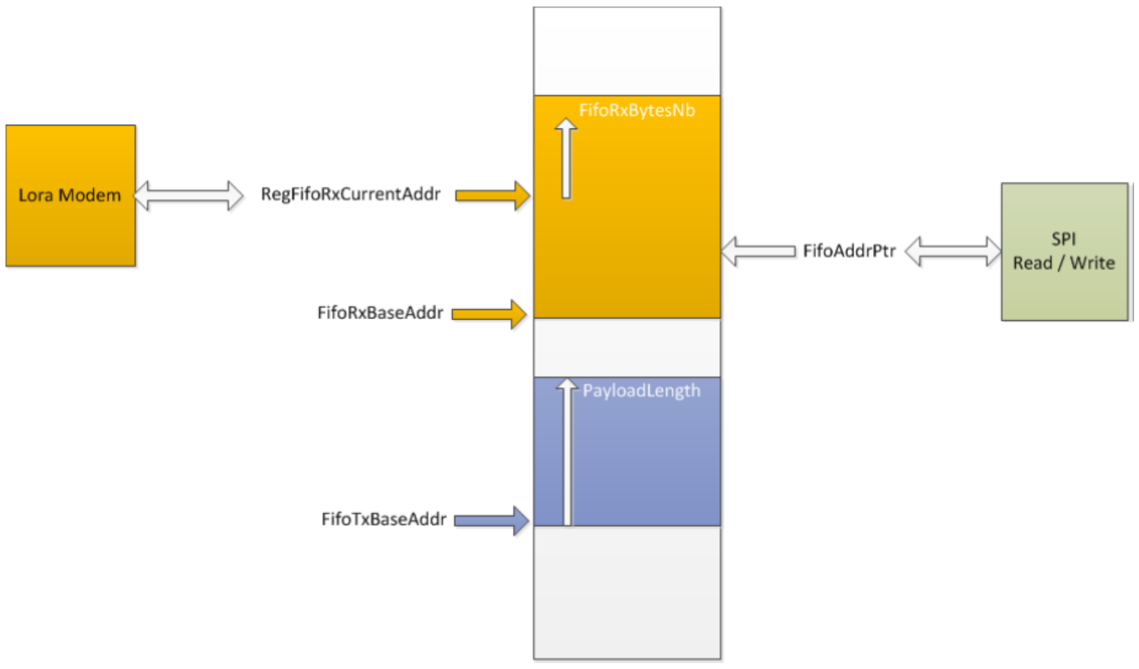


Figure 13: FIFO buffer register. Sourced from LoRa module datasheet

6) **UART**

The Universal asynchronous receiver-transmitter, or UART, is a hardware device for asynchronous serial communication where the data format and transmission speed can be configured. The UART is usually an integrated circuit used for serial communications over a computer or a peripheral device serial port (USB port). Multiple UARTS can be used on a single device which allows for more advanced set ups and connections.

The Universal asynchronous receiver-transmitter (UART) functions via a series of registers set on the ATmega2560. Below is the datasheet from the ATmega2560:

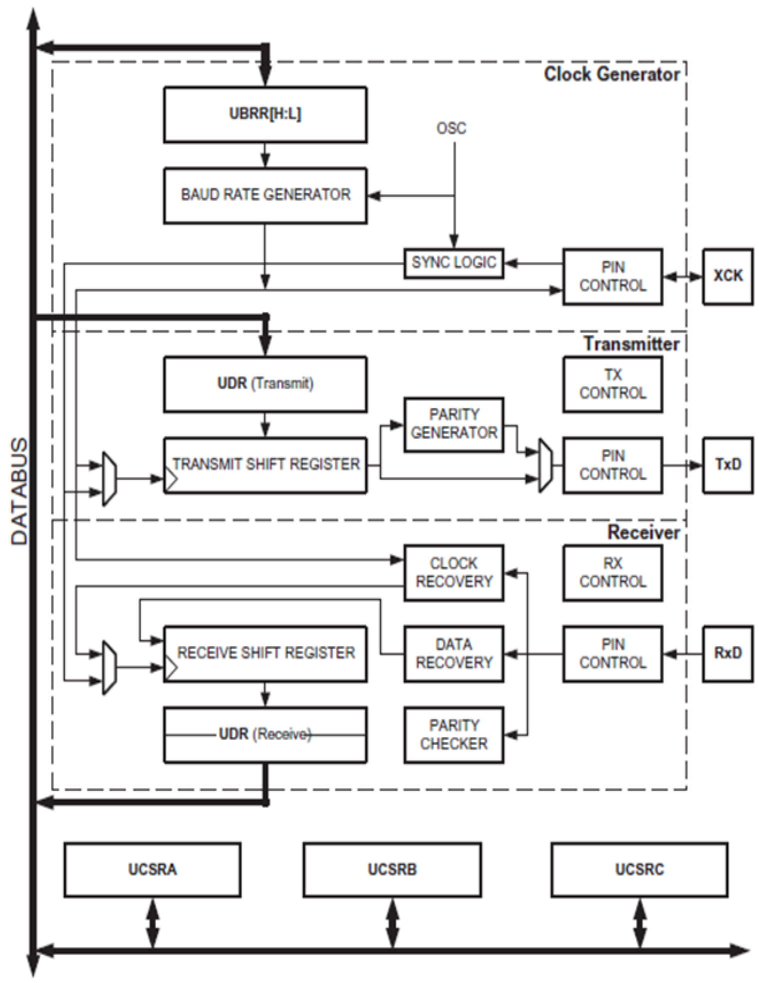
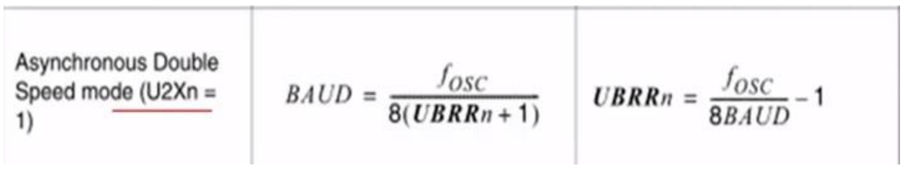


Figure 14: ATmega2560 UART diagram. Sourced from datasheet:
http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf

From the datasheet we can see that the following information is needed to set up the UART:

UBRR(H:L) is the register that determines the baud rate of the UART rate which determines the send speed of the communication. The UART baud rate is set in the register. The UART baud rate is usually set between 1200 to 190000 depending on whether speed or reliability is more important. As a rule of thumb, the higher the baud rate is, the faster the message is sent, but because of the small waves used to increase the send speed, the risk of one or more bits being lost will also increase.

To determine the needed baud rate, use this formula:



Asynchronous Double Speed mode (U2Xn = 1)	$BAUD = \frac{f_{OSC}}{8(\mathbf{UBRR}n + 1)}$	$\mathbf{UBRR}n = \frac{f_{OSC}}{8BAUD} - 1$

Figure 15: Baud rate calculation.

After the UBRR register is the UDR register. This is the transmit and receive register and acts as a gate for all data, i.e. all data must be received by the UDR before it will be sent from the UART.

Apart from this the UART also has three check registers: UCSRA, UCSRB and UCSRC (*User Control Setting Rest A, B and C*). The Check registers are an inbuilt security designed to check whether the message has been sent correctly. It is important to note that the check registers do not ensure that the message on the receiving end is identical as the one that was send. Consequently, it is important to set up an additional control, if sending important data where potential interference and loss of accuracy can prove to be a massive problem. In this case a system must be set up to ensure that there has been no loss or alteration of the data between the sender and the receiver. The most common way of doing this is by instigating a CTC *(Check to check)* check. In the CTC check the XOR function is used. This function calculates (sums up) all the bits of a selected string.

In the CTC check the function in applied to all bit in the send package and compared with the XOR-function in applied to all bit in the received package. If the XOR results are the same the message is likely to be uncorrupted and will be accepted without an error message. This method is not fool-proof as changes in different bits in the message may counterbalance each other. In this case the errors will not be detected, but the CTC check will decrease the likelihood of erroneous message going through the system.

7) **Other external software**

- **Atmel Studio 7.0 -** used in writing, compiling and programing the code for the ATmega328.
- **Putty** – used to view the sensor data that is sent using UART from the ATmega328.
- **Arduino Uno** – used as USB-to-TTL converter between PC and the ATmega328 for UART communication.

III. Design

This chapter covers the hardware and software design choices for the project along with the various issues found during its development.

1. Overview

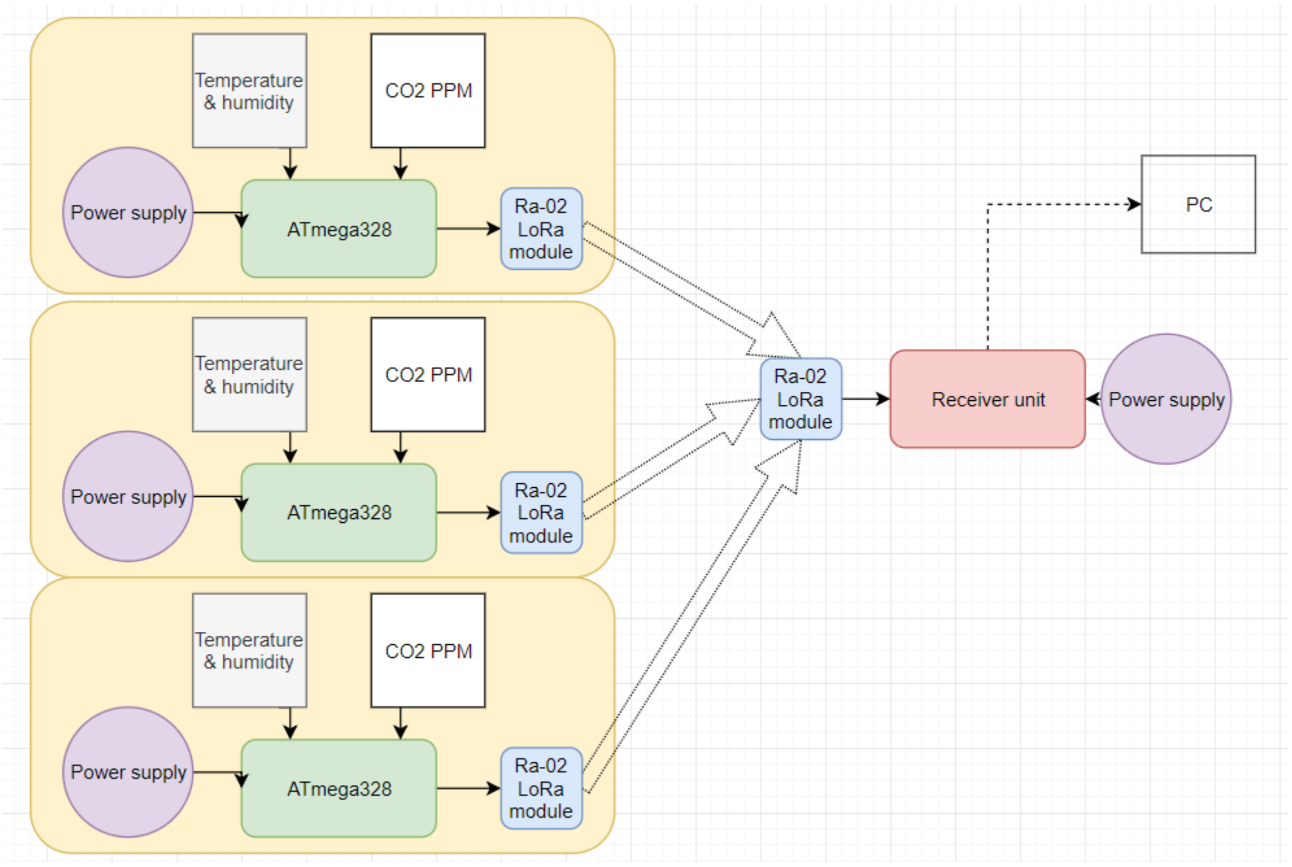


Figure 16: The overview diagram for the system of the project

The system is made of two parts: the sensor unit(s) and the receiver unit. The sensor unit(s) measure temperature, humidity and CO2 levels, then transmit the measurements to the receiver unit using the LoRa communication protocol.

Before the LoRa modules were tested and used, checking of sensor data was done using UART serial communication between the ATmega and a PC.

The receiver unit is responsible for retrieving the sensor data from the reciever LoRa module and storing it for future purposes, such as transmitting it to a PC.

2. Hardware

1) Sensor board

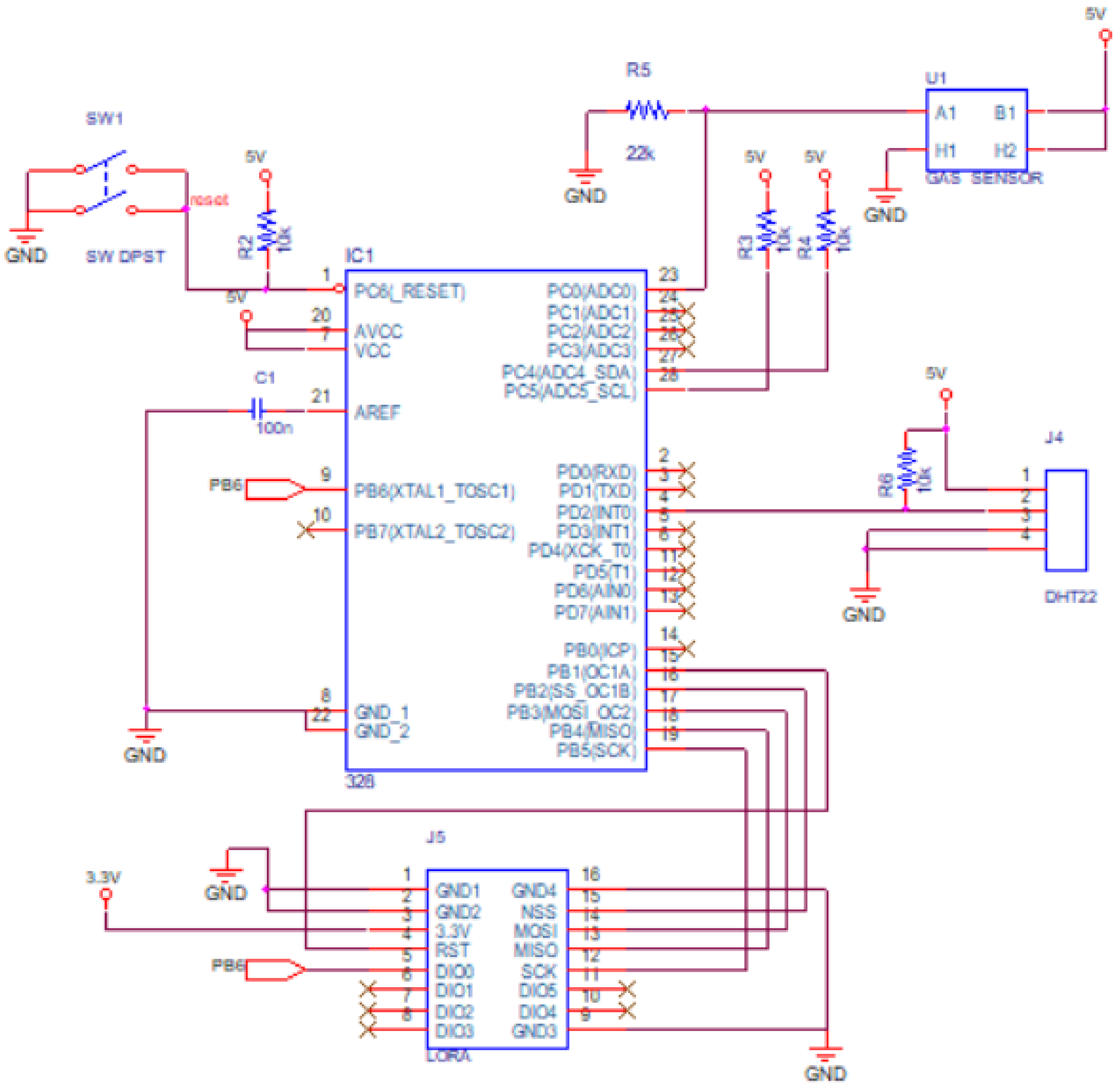


Figure 17: The schematic for the sensor board circuit

The main component of the sensor board circuit is the ATmega328, which is connected two connected sensors, the DHT22 and the MQ135. In addition, it is connected to the SX1276 LoRa module using SPI.

All components in the circuit are powered with 5V, except for the LoRa module, which has to be supplied with 3.3V. The other components have operating voltages of 3.3V – 5V, except for the MQ135, which needs 5V for its heating module.

A pull-up resistor of 10k ohms is put between the VCC and data pins of the DHT22 to keep the data line high.

The internal voltage reference AREF pin is decoupled by an external 100 nF capacitor to improve noise immunity.

The 22k ohm resistor connected to the MQ135 (R5) provides the resistance value used by the MQ135 library.

The following list describes the function of each connected pin of the ATmega328:

1. PC6 (RESET) – connected to reset switch. If pressed the ATmega is reset
2. PD0 (RXD) – used for UART communication; not intended for use in finished project
3. PD1 (TXD) – used for UART communication; not intended for use in finished project
4. PD2 (INT0) – external interrupt pin, connected to the DHT22 data pin. Used for reading measurements sent from sensor
7. 5V
8. Ground
9. PB6 – connected to DIO0 data pin of the SX1276 module, used to trigger a TX interrupt for data transmission (active low). Initially connected in series with a switch for manual control of the interrupt, which is not used in the finished project
15. PB1 – connected to RST pin of the SX1276 module
16. PB2 – connected to NSS (chip select) pin of the SX1276 module. Used in SPI communication
17. PB3 – connected to MOSI pin of the SX1276 module. Used in SPI communication
18. PB4 – connected to MISO pin of the SX1276 module. Used in SPI communication
19. PB5 – connected to SCK pin of the SX1276 module. Used in SPI communication
20. AVCC – connected to 5V
21. AREF – analog reference voltage pin. Connected by 100nF bypass capacitor to ground. Used by the ADC of the ATmega for reading analog output of the MQ135
22. Ground
23. PC0 (ADC0) – connected to analog output pin of the MQ135. Used to read data from the sensor
27. PC4 (SDA) – connected through pull-up to 5V, in case I2C support is needed in the future
28. PC5 (SCL) – connected through pull-up to 5V, in case I2C support is needed in the future

The 3.3V for the LoRa module was supplied through the use of a voltage regulator circuit:

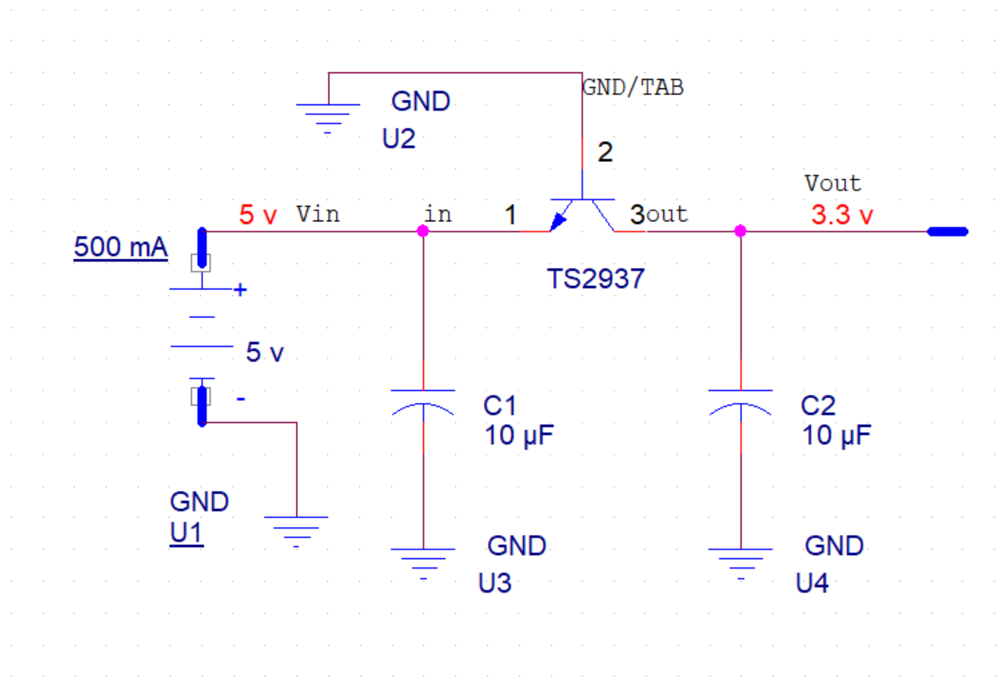


Figure 18: The schematic for the voltage regulator circuit. Based on: http://www.mouser.com/ds/2/395/TS2937_D13-522475.pdf

The primary component of the circuit is the TS2937 500mA ultra dropout voltage regulator and two 10µF capacitors to maintain stability and improve transient response.

3. Software

1) Implementing the sensor libraries

The library uses 3 main functions for initializing, enabling the sensor and checking data:

- `DHT22_Init();`
- `DHT22_StartReading();`
- `DHT22_CheckStatus(&sensor_data);`

At the start of the program, the `DHT22_init` function is called once, before the main loop. It is used to configure the sensor data pin as output (initially), then set the timer mode to CTC and enable compare match interrupts.

```
DHT22_DDR |= (1 << DHT22_PIN);

PIN_HIGH(DHT22_PORT,DHT22_PIN);
```

/* Timer config. */

`TIMER_SETUP_CTC` // Set timer to CTC

`TIMER_ENABLE_CTC_INTERRUPT` // Enable compare match interrupt

// Timer is started by the function DHT22_StartReading. For now

// it remains with prescaler = 0 (disable).

```
TIMER_STOP
```

To send a signal to the sensor to start reading and outputting the data, the `DHT22_StartReading` function is called. It returns a variable of type `DHT22_STATE_t` with the possible values:

- `DHT_BUSY`: The reading did not start because the state machine is doing something else, indicating that the previous reading did not finish.
- `DHT_STARTED`: The state machine has successfully started. The user can wait for and retrieve data using `DHT22_CheckStatus()` function.

```
		/* Check if the state machine is stopped. If so, start it. */
		if (state == DHT_STOPPED || state == DHT_DATA_READY || state == DHT_ERROR_CHECKSUM || state == DHT_ERROR_NOT_RESPOND){
		/* Configuring peripherals */
		//EIMSK &= ~(1 << INT0); // Disable external interrupt
		EXT_INTERRUPT_DISABLE
		DHT22_DDR |= (1 << DHT22_PIN); // Configuring sensor pin as output.
		PIN_LOW(DHT22_PORT,DHT22_PIN); // Write 0 to pin. Start condition sent to sensor.
		TIMER_OCR_REGISTER = 255; // Timer compare value equals to overflow (interrupt will fire at 255us).
		TIMER_COUNTER_REGISTER = 0; // Reset counter value.
		state = DHT_HOST_START; // Change state.
		TIMER_START // Start timer with prescaler such that 1 tick equals 1us (freq = 1MHz).
		return DHT_STARTED; // Return value indicating that the state machine started.
}
else{
		return DHT_BUSY; // If state machine is busy, return this value.
}
```

To start the measuring temperature and humidity, as well as sending the measurements to the microcontroller, the start signal needs to be sent first. However, to successfully initiate the sensor, timing is critical – since the function does not use delay functions, it has to keep track of time using the timer/counter.

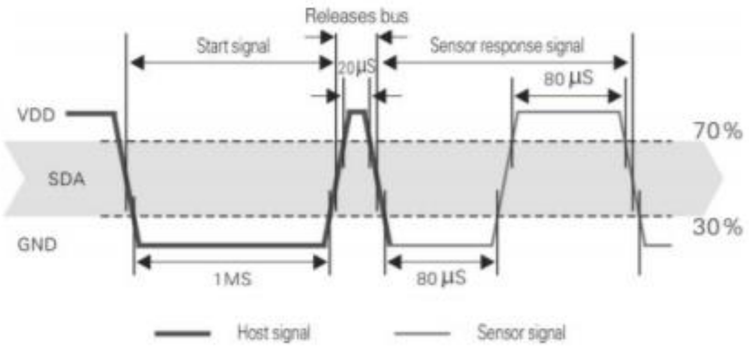


Figure 19: DHT22 initializing signal timing. Picture taken from datasheet: [*http://akizukidenshi.com/download/ds/aosong/AM2302.pdf*](http://akizukidenshi.com/download/ds/aosong/AM2302.pdf)

Control of the pin voltage and timing is done with the timer/counter, along with the external interrupt handler. For more information, refer to “DHT22 library” in References.

After the `DHT22_StartReading()` function, the `DHT22_CheckStatus(struct data)` is called in order to check if a transfer is complete. It is responsible for convering the data from the 40-bit format to integer values, and checking parity by matching the checksum with the data from the sensor.

It returns a `DHT22_STATE_t` variable with the state of the state machine. The returned values are:

- `DHT_DATA_READY`: Data is received and can be used by the program.
- `DHT_ERROR_CHECKSUM`: Error, the checksum does not match data.
- `333333333333333DHT_ERROR_NOT_RESPOND`: Sensor is not connected or not responding for some reason.

If the data is received, the `sensor_data` struct can be accessed, which contains the new (or updated) temperature and humidity values. Once the values are retrieved, they can be sent through UART or LoRa.

Because the CO2 PPM values from the MQ135 gas sensor are measured by the ratio of the Rs/Ro sensor resistances, they are affected by the ambient temperature and humidity:

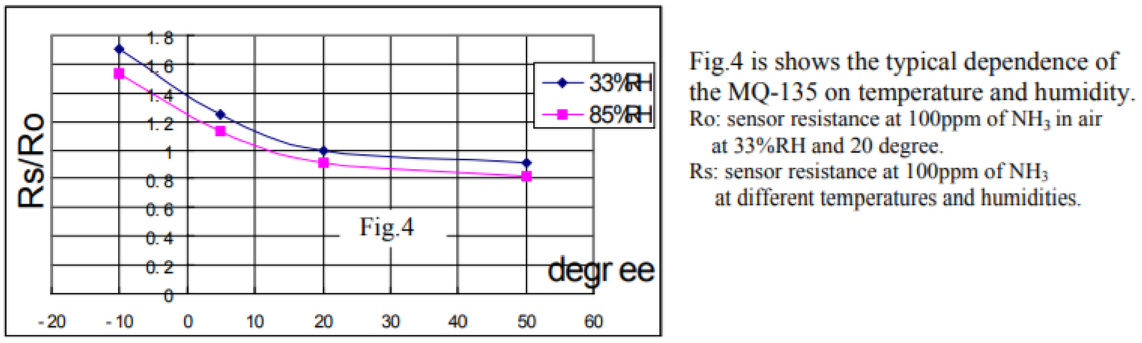


Fig.4 is shows the typical dependence of the MQ-135 on temperature and humidity.
Ro: sensor resistance at 100ppm of NH_3 in air at 33%RH and 20 degree.
Rs: sensor resistance at 100ppm of NH_3 at different temperatures and humidities.

Figure 20: Dependence of Rs/Ro resistances on different temperatures & humidities. Picture taken from datasheet: [*https://www.olimex.com/Products/Components/Sensors/SNS-MQ135/resources/SNS-MQ135.pdf*](https://www.olimex.com/Products/Components/Sensors/SNS-MQ135/resources/SNS-MQ135.pdf)

Using the humidity and temperature values from the DHT22 sensor, the recorded CO2 PPM value can be adjusted to be more accurate to the actual CO2 level in the atmosphere.

For more information on the MQ135 library, calculations and code examples, refer to “MQ135 library” in References.

2) Implementing LoRa

Flowchart on the right shows a LoRa transmission sequence. The module will be in stand-by mode waiting for the TX init/interrupt. If TX is initialized, data will be written to the FIFO buffer and the TX Mode Request will initialize the transmission. Module will then wait for the TxDone interrupt, and when it is received, the module returns to Stand-by.

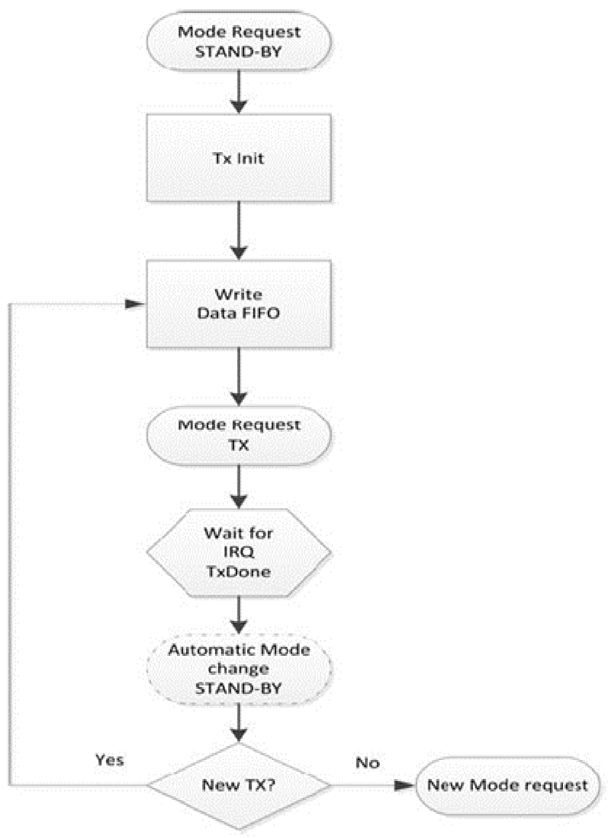


Figure 21: A typical LoRa transmission flowchart with the SX1276. Sourced from LoRa module datasheet

The flowchart below shows a receive sequence. When RX is initialized, there is 2 different scenarios depending on which receiving mode, the module has been configured to. In **single mode**, the module will search for a preamble in a giving amount of time. If not found at the end of the time given, the RxTimeout interrupt will occur, and the module will be set back into stand-by. If RxDone interrupt occurs, it means a payload has been successfully written to the FIFO buffer.

The PayloadCrcError interrupts will only occur if the error correction isn’t valid, but the data will be written to FIFO buffer no matter if the PayloadCrcError is set or not.

In **continuous mode**, the receiving LoRa module will be constantly searching for a preamble. If the preamble length doesn’t match what’s expected, the module will discard it, and search for a new preamble. RxTimeout won’t occur in continuous mode. When the RxDone interrupt occurs, the received data will be written to the FIFO buffer, and the module will search for a new incoming preamble.

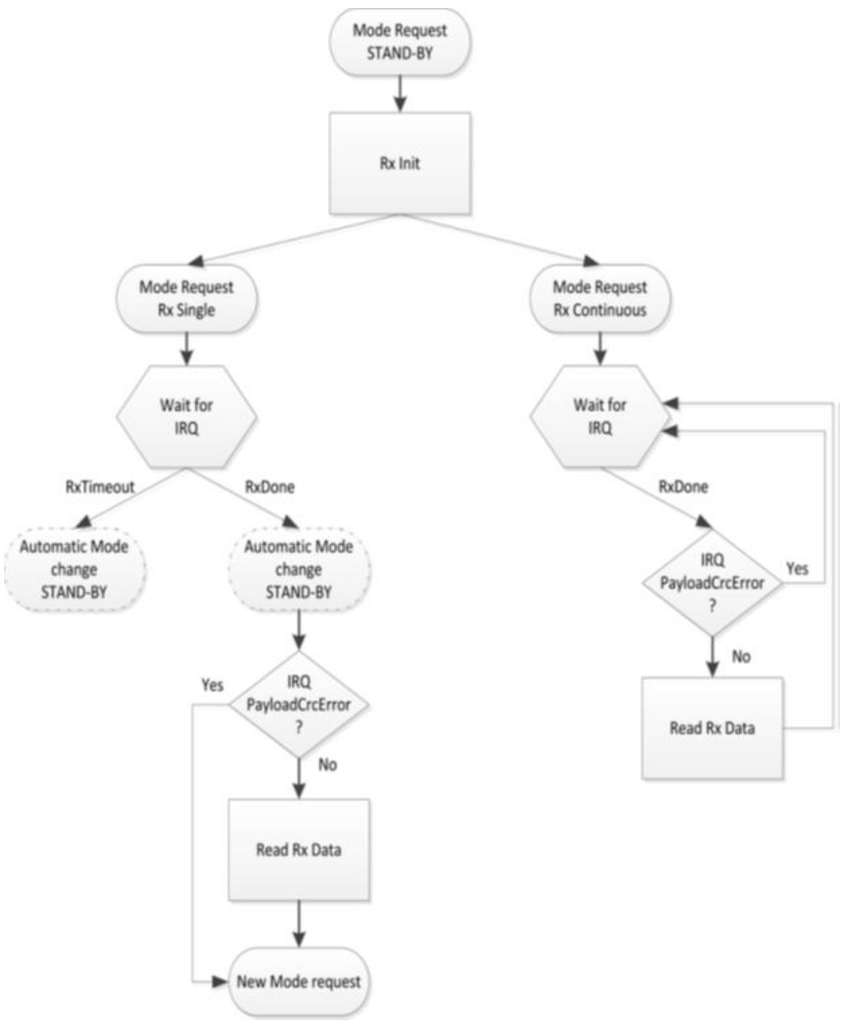


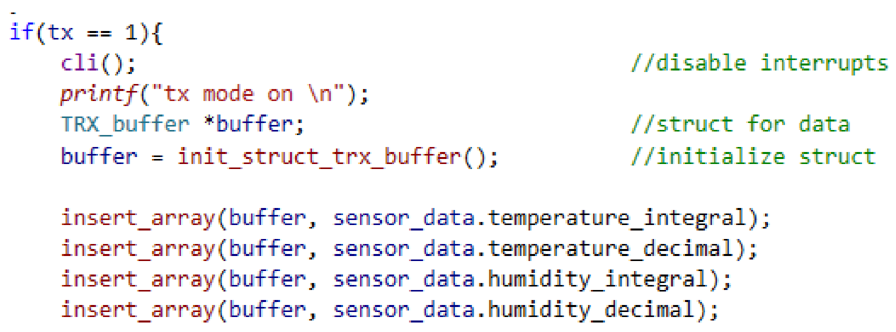
Figure 22: LoRa receive sequence with the SX1276. Sourced from LoRa module datasheet

Shown below is a code example of LoRa implementation on the sensor board:

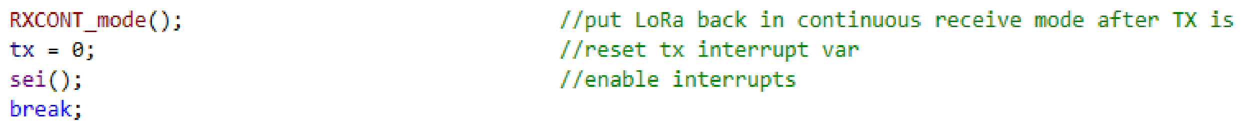
```
sei();

_delay_ms(1000);        //delay to make sure the lora module is up and running before communication is started

sx1278_LoRa_mode();     //setting the LoRa module to LoRa mode
setSignalBandwidth(7);  //BW 7 = 125KHz
setSpreadingFactor(9);  //Spreading factor of 9
setTxPower(2);          //dBm power settings. go from 2-17
enableCrc();
```

Getting LoRa module ready for communication. Setting appropriate configurations.

When an interrupt initialize TX mode, variable “tx” will be set to 1 in the Interrupt Service Routine. When tx is equal to 1, the if-statement in the while loop becomes true. Then, interrupts are disabled so it doesn’t interrupt while transmitting. Then point and initialize the trxbuffer struct and write the temperature and humidity sensors results into the struct. After this, the data is written to FIFO and LoRa is set in TX mode to transmit.



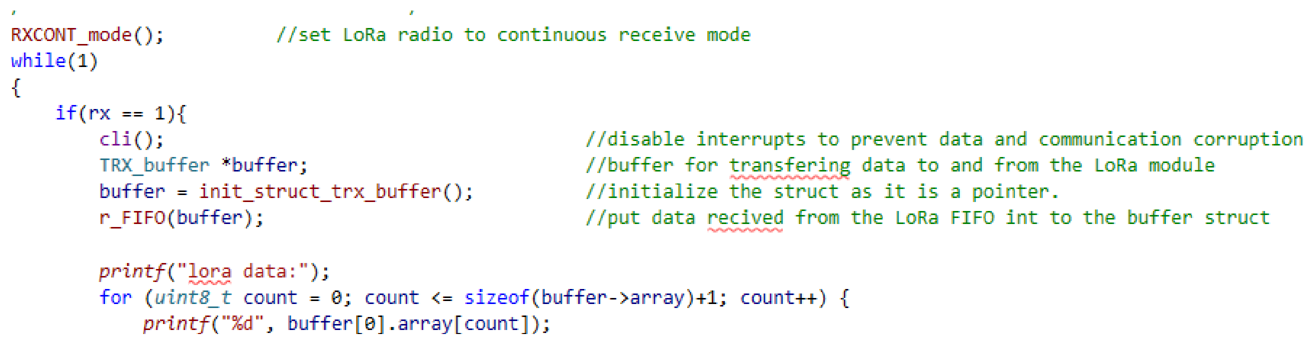
```
if(tx == 1){
    cli();                                  //disable interrupts
    printf("tx mode on \n");
    TRX_buffer *buffer;                     //struct for data
    buffer = init_struct_trx_buffer();      //initialize struct

    insert_array(buffer, sensor_data.temperature_integral);
    insert_array(buffer, sensor_data.temperature_decimal);
    insert_array(buffer, sensor_data.humidity_integral);
    insert_array(buffer, sensor_data.humidity_decimal);
```



```
RXCONT_mode();                              //put LoRa back in continuous receive mode after TX is
tx = 0;                                     //reset tx interrupt var
sei();                                      //enable interrupts
break;
```

After a transmission, the module is returned to RX continuous mode, TX is set to 0 and interrupts are enabled.

If rx is equal to 1, statement becomes true and the following will happen. Interrupts will be disabled to avoid interference; initialize pointer to the struct and then write received data to the buffer struct.



```
RXCONT_mode();          //set LoRa radio to continuous receive mode
while(1)
{
    if(rx == 1){
        cli();                              //disable interrupts to prevent data and communication corruption
        TRX_buffer *buffer;                 //buffer for transfering data to and from the LoRa module
        buffer = init_struct_trx_buffer();  //initialize the struct as it is a pointer.
        r_FIFO(buffer);                     //put data recived from the LoRa FIFO int to the buffer struct

        printf("lora data:");
        for (uint8_t count = 0; count <= sizeof(buffer->array)+1; count++) {
            printf("%d", buffer[0].array[count]);
```

3) UART implementation between receiver unit to PC

The goal is to use the UART to transfer data from the receiving AVR unit to the computer. The Uart0 on the ATmega2560 is used, because the Uart0 is attached to the USB-B port on the board. This makes it very simple to attach the AVR to the PC and simplifies the amount of cables necessary to run the unit. To make the UART work the following steps are taken:

1. Enable the receiver and transmit transmitter
2. Set the Baud rate: When setting the Baud, it needs to be ensured that it is low enough to ensure that the quality of message is adequate and on the same time high enough to ensure that it does not create a bottle neck.
3. Set the data bytes
4. Stop bits: Sets the end of the packet, this can be a check bit if running a CTC check
5. Check bit: This is an optional step and may be applied only if needed.

If this is done the result is evaluated and it is needed to decide whether to….

6. Declare a transmit and receive function

In practice, this means that the steps needed to take to get the UART working is enabling the receiver and transmitter then setting the baud rate to a speed fast enough that it is not bottlenecking, but not so fast that the packet is lost after being sent. Then it is needed to set the data bytes, stop bits and, if needed, the check bit and then decide whether to enable full-duplex to double the send & receive speed. After that a transmit and receive function is declared.

In the real world it is a piece of programming which looks like this:

```
/*initialiser uart0 til hastighed bestemt af baud, full duplex receive og
transmit polling)*/
void init_UART( int baud){
            UBRR0H =(unsigned char)(baud>>8);//s211 vi sætter baud-raten i to (8
bit) registre (UBRR0H og UBRR0L) for at få den vores 16 bit baud rate.
            UBRR0L =(unsigned char)baud;
            UCSR0A =(1<<U2X0);  //s223 full duplex
            UCSR0B|=(1<<RXEN0)|(1<<TXEN0);  //s. 224 i datablad  UART enabled
begge veje
            UCSR0C|=(1<<UCSZ00)|(1<<UCSZ01);  //8 databit og no parity se tabel
22-7

}


/*modtag en 8 bit char*/
char get_char(){
            while(!(UCSR0A&(1<<RXC0))); //vent på at der er en ny char modtaget
status bit.
            return UDR0;  //læs og returner karakteren
}


void put_char(char data){
            while(!(UCSR0A&(1<<UDRE0)));  //vent på at sidste karakter er sendt
status bit.
            UDR0=data;
}


/*funktion der kan modtage et array/string vha. get_char funktionen*/
void getsUSART0(char *ptr)      {
            char cx; // den char, der skal holde den modtagne char.
            while((cx=get_char())!='\r'){
                        *ptr=cx; // modtager en char og indsætter på den aktuelle
celle.
                        ptr++;     //inkrementere ptr (index-værdien i det givne
array)
            }
            *ptr=0;
}

/*funktion der kan sende et array/string vha. put_char funktionen*/
```

```
void putsUSART0(char *ptr)
{
            while(*ptr!=0){ //while løkke, der skal "køre", så længe der er
værdier i array'et
                        put_char(*ptr);//sender indholdet af ptr
                        ptr++; //inkrementere ptr indexet med én
            }
}
```

4. Problems and solutions

Here are some of the various problems related to the project that were found in its development, and how some of them were solved (or proposals on how to solve them).

Problem: No PPM values were recorded (always equal to 0) after the integration of the LoRa module, even though all other values associated with the ADC and the MQ135 were (mostly) correct.

Comment: this problem was never solved as there was not enough time to do so. The code was not changed from the time the sensor worked properly, therefore it was not a software issue. Initially it was suspected that the sensor did not work properly because the 5V rail was very unstable in terms of voltage, and the sensor could not reach its operating voltage. After the rail was stabilized, the problem persisted, even after testing multiple MQ135 sensors. The most likely cause of the problem is that since the circuit was changed, the sensor resistance read by the ADC and the code changed as well, causing it to behave unexpectedly.

Problem: The connections between the sensors and the microcontroller seem to be fluctuating or sometimes disappearing entirely.

Solution: This problem was caused by bad quality jumper cables. It was determined by measuring resistance between cable ends, and noticing that they frequenly slip off the headers because of their loose fitting. Replacing the cables with new, better quality jumpers solved the problem.

Problem: The LoRa module's operating voltage is 3.3V, while the gas sensor needs 5V for its heating module. How to give the correct voltage for each component in the same circuit, without using multiple power supplies? The voltage divider circuit using resistors did not work.

Solution: problem was solved by using a 3.3V voltage regulator circuit shown in the previous section.

Problem: The 5v rail is very unstable, presumably because of the 3.3V voltage regulator.

Solution: problem was solved by using another voltage regulator for the 5V rail. First, use a 9V battery for power, then get stable 5V from it using a 5V voltage regulator, then afterwards divert a 3.3V rail from the 5V using the previous voltage regulator circuit.

IV. Results

This chapter presents the results achieved at the end of the project's development, some of the tests that were taken, discussion of the project and its development and some recommendations for future work, should this project's development be continued.

1. Finished prototypes

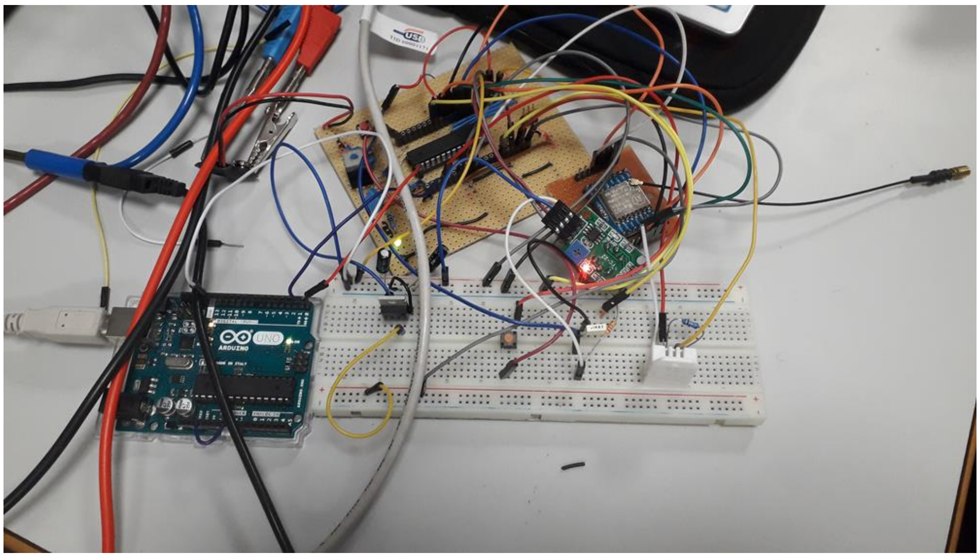


Figure 23: The sensor board circuit, connected through UART to a PC using Arduino Uno as USB-TTL converter

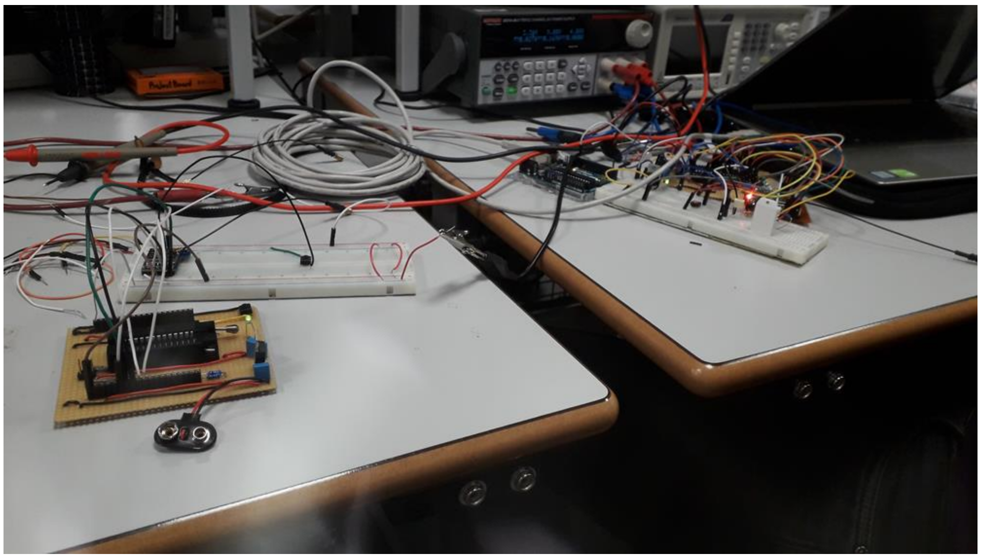


Figure 24: Testing LoRa communication between modules

2. Tests

The tests shown were chosen according to the objectives of the project:

Test number	Test Holder	Reason	Expected outcome	Actual outcome	Pass/ Fail
1	Dainius	Test if UART on the ATmega development board works	A message will be received from the ATmega and displayed on Putty	The message was received and displayed	Pass
2	Dainius & Juan	Check if both the DHT22 and MQ135 work together with the ATmega by reading sensor data[2] using UART	The temperature, humidity and CO2 PPM values would be displayed on Putty	The temperature, humidity and PPM values were displayed on Putty	Pass
3	Dainius & Jonathan	Check if the LoRa modules work by transmitting test data (sequence of numbers) from one module to another	After enabling TX on LoRa module, the terminal connected to the sensor board would display the number sequence	The number sequence was received and displayed	Pass
4	Dainius & Jonathan	Check if the sensor data can be transmitted from one module to another	The transmitted temperature, humidity and CO2 PPM would be received and displayed on Putty	The temperature and humidity were received and displayed. The MQ135 kept reading PPM as 0, even though it seemed to function properly[3]	Partial pass
5	Dainius & Jacob	Check if a Raspberry Pi receiver unit is feasible by connecting it to the LoRa module and waiting for data	After transmission from the sensor, the data would be displayed on the terminal	Nothing was displayed on the terminal	Fail

[2] The MQ135 was not properly calibrated since accuracy was not yet a concern, so wrong PPM was to be expected

[3] Every other recorded value associated with the gas sensor was as expected (non-zero)

3. Discussion and recommendations

Outside of obstacles related to project development, there are a number of reasons why the project could not be finished on time:

- Lack of motivation – during the semester, preparing for the international module and internships was a high priority for the group, which led to dedicating less time and attention to project development. This was further amplified by inconsistent scheduling for lectures and events, which made it difficult to plan meetings and work.
- Lack of group cohesion – each group member had experience working in smaller groups in the past, however this time the group was larger than ever before. In addition, each group member had different schedules, responsibilities and priorities. There have been many attempts made to streamline communication between members, such as Slack, Facebook, scheduled meetings and a freely available project plan, but it always remained difficult due to previously mentioned reasons.
- Outside interference – the constant influx of long-term side projects from lecturers, events and distractions diverted time and resources from the project development. Moreover, there was a lot of confusion regarding this semester's exam schedule and procedure, which put more stress and pressure on the group to wrap up development.

Despite these issues, a fair amount of work has been done on this project, as 2/3 of objectives initially set out by the group were met. In addition, the project has further development potential, allowing the group to continue working on it later if they wish.

If this project were to be continued, there is a lot of work that could be done in the future:

- Finishing work on the receiver unit
- Designing a PCB for the sensor boards
- Implementing Wi-Fi connection between the receiver unit and a PC to send the sensor data to
- Better quality sensors and connectors
- Battery and more types of sensors on the sensor board
- Modular sensor boards - additional sensors could be plugged and unplugged from the sensor board

V. Conclusion

1: A system of sensors capable of periodically measuring temperature, humidity and CO2 levels in the atmosphere was built.

2: A LoRa module was attached to the sensor system, and was capable of transmitting collected temperature and humidity data.

3: The receiver unit was designed, but it was never practically implemented and could not receive sensor data.

The project did not meet all of its goals, however enough significant milestones have been made to ensure its success if more time would be dedicated to its development in the future. As such, it is difficult to call this project failed since it still exhibits enormous potential.

VI. References

1. Bibliography

Brian Ray. (2018, June). What Is Lora? A Technical BreakDown. Retrieved from https://www.link-labs.com/blog/what-is-lora at December 11th, 2018.

Microchip Technology Inc. (2016, November). ATmega328/P DATASHEET COMPLETE. Retrieved from http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf at December 11th, 2018.

Microchip Technology Inc. (2014, February). Atmel ATmega640/V-1280/V-1281/V2560/V2561/V | 8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable Flash DATASHEET. Retrieved from http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf at December 11th, 2018.

Semtech Corporation. (2017, March). SX1272/73 - 860 MHz to 1020 MHz Low Power Long Range Transceiver. Retrieved from https://www.semtech.com/uploads/documents/sx1272.pdf at December 11th, 2018.

2. Useful links

MQ135 datasheet:
https://www.olimex.com/Products/Components/Sensors/SNS-MQ135/resources/SNS-MQ135.pdf

DHT22 product manual:
http://akizukidenshi.com/download/ds/aosong/AM2302.pdf

DHT22 Interrupt driven library:
https://moretosprojects.blogspot.com/2014/01/dht22-interrupt-driven-library-for-avr.html

MQ135 library and calculations:

https://davidegironi.blogspot.com/2014/01/cheap-co2-meter-using-mq135-sensor-with.html