Internship Report

Smart Home Monitoring System

Lasaris Lab - Brno, Czech Republic

ZAOUALI Alaeddine

22/05/2019

ABSTRACT

During my stay in the lab, I was charged of building an IoT home monitoring system based on Arduino controllers. I divided the whole process into three steps. I start by conceiving and analyzing parts of the system I aim to achieve, then I start implementing the wiring and the code. Afterward, I make a security assessment and I try to understand the concerned technology on different layers.

Nearly 4 months spent in this project allowed me to visit most of the intended technologies and I used that period to implement a major part of the system. Though, I faced many technical difficulties, whether it is hardware or software problem. I made choices that took in consideration time and feasibility. I came up with a solution that works considering the limitations and I chose to limit the security analysis to specific technologies.

KEYWORDS

IoT - Arduino - Security - Home Automation - Wireless

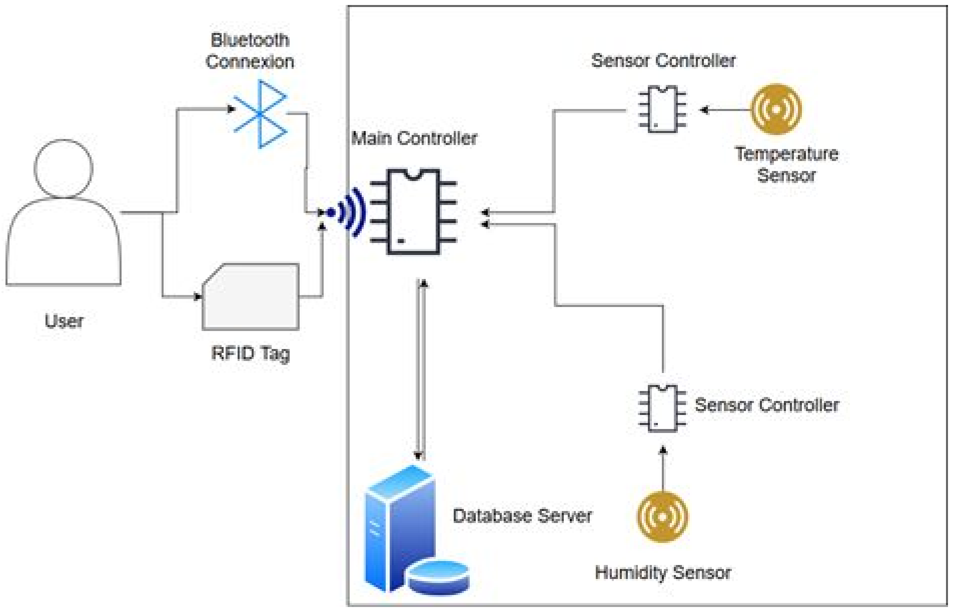
WORKPLACE

The lab is dedicated to research, development and teaching of topics related to various theoretical and practical problems related to the development of large software systems and employment of modern information technologies in practice. They address the issues and challenges related to the design and development of information systems, including process and data modeling, management of system development, and various technologies, incl. mobile technology. They are involved in research and development projects in the field of corporate and public information systems, complex event processing, and design of large-scale IT infrastructures, such as the smart energy networks (smart grids).

INTRODUCTION

Smart home systems come with the idea of associating different technologies in order to wirelessly control daily life devices. The emergence of the Internet of Things (IoT) has contributed in globalizing home automation systems by networking different devices and processing their respective sensors datas by a centralized server. This document presents the conception and the implementation of an arduino based system whose aim is to illustrate an easy and secure way to build a smart home monitoring system. The system makes use of different communication technologies like Bluetooth, ZigBee or RFID. We will parallelly assess and enhance the security of the some components and some communication layers. Using an arduino based circuit also makes this project accessible for beginners and support cheap components.

This internship allowed me to learn much about wireless technologies and introduced me to a more physical part of IT. It gave me the time to understand how these devices work in an information system. I also had the time to read research documents and work in a research environment. This document will start by introducing the different devices we have, then it will explain how to install everything by giving the different dependencies. I will finally connect the whole machinery together to get the intended system.

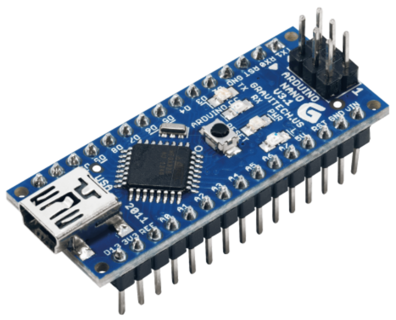
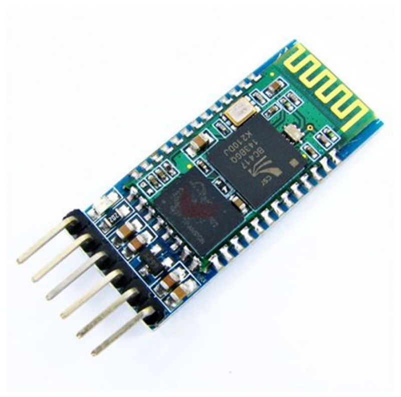
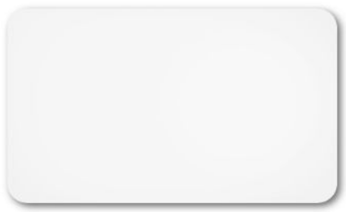
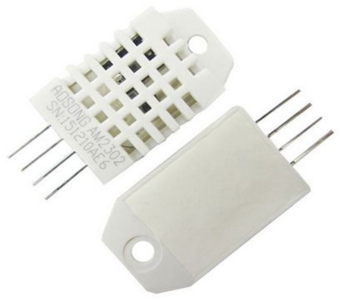


Smart Home Monitoring System

1- INSTALLATION

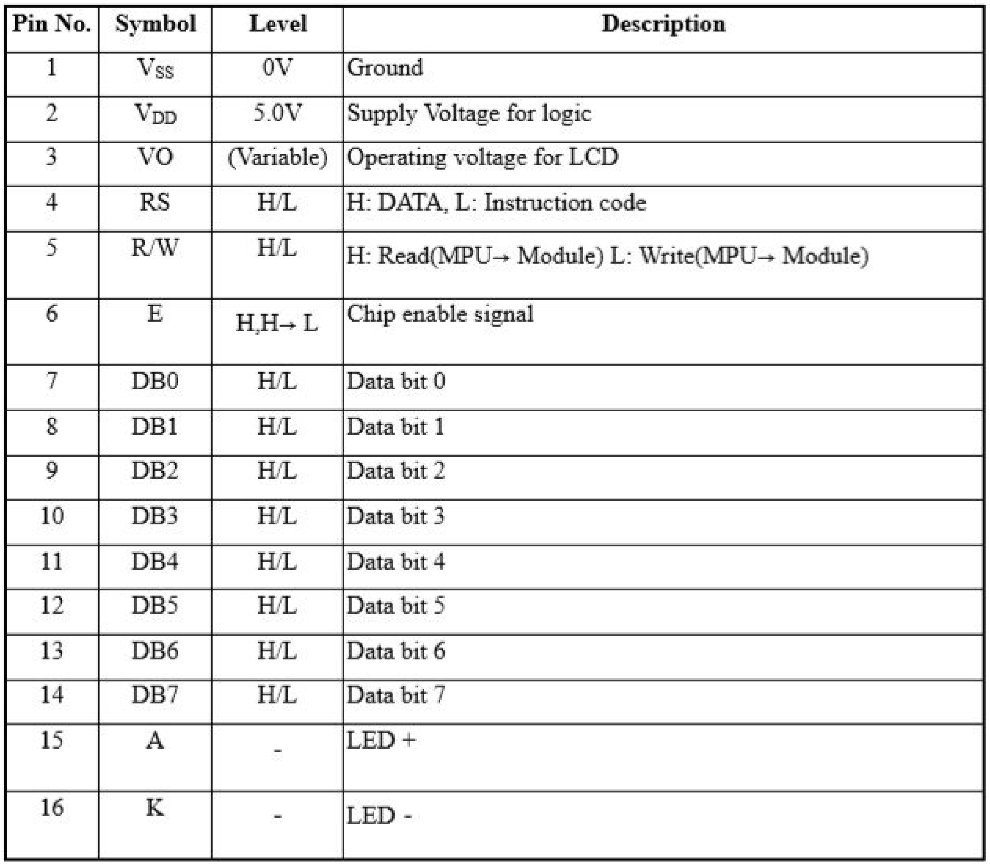
This part will only cover the individual installation of components, we will assemble them at the end. If you have enough knowledge about what you are doing, you can skip this part, and proceed to the final wiring.

1.1- COMPONENTS

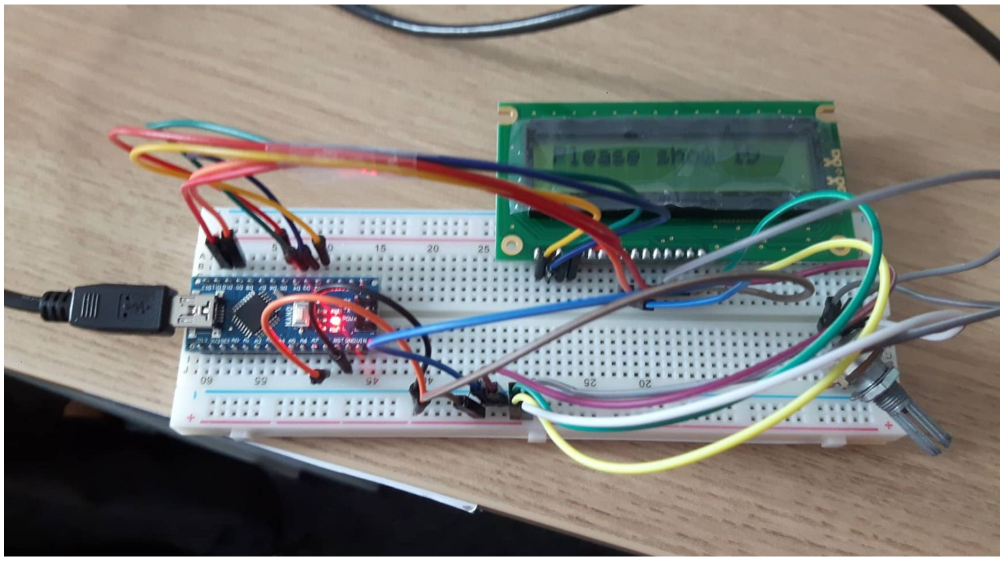
 Arduino Nano	 Bluetooth - hc05	ZigBee - Xbee S1
RFID Reader - rc522	 RFID Tag - Mifare Classic 1k	RFID Tag - Mifare Classic 1k
LCD - Winstar WH1602A	 Temperature & Humidity Sensor - ASAIR AM2302	Potentiometer

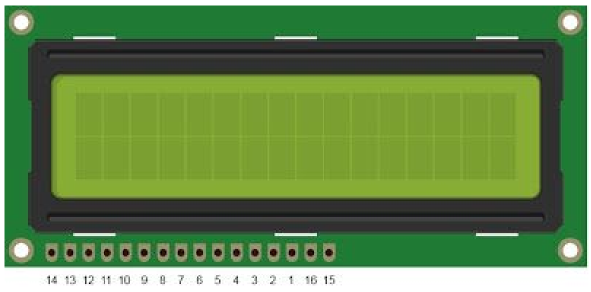
1.2- LCD Winstar WH1602A

We will use LCD screens (Liquid Crystal Display) to display sensors feedback and the system state. We can use it alongside with LEDs to inform users about successful and failed operations then display indications to help them. The LCD we're using is a Winstar WH1602A but most 16x2 LCDs should work the same. Check pins order on LCD before wiring.

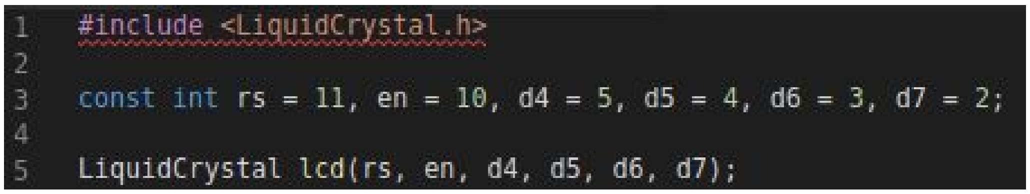


Pin No.	Symbol	Level	Description
1	V_{SS}	0V	Ground
2	V_{DD}	5.0V	Supply Voltage for logic
3	VO	(Variable)	Operating voltage for LCD
4	RS	H/L	H: DATA, L: Instruction code
5	R/W	H/L	H: Read(MPU→ Module) L: Write(MPU→ Module)
6	E	H,H→ L	Chip enable signal
7	DB0	H/L	Data bit 0
8	DB1	H/L	Data bit 1
9	DB2	H/L	Data bit 2
10	DB3	H/L	Data bit 3
11	DB4	H/L	Data bit 4
12	DB5	H/L	Data bit 5
13	DB6	H/L	Data bit 6
14	DB7	H/L	Data bit 7
15	A	-	LED +
16	K	-	LED -





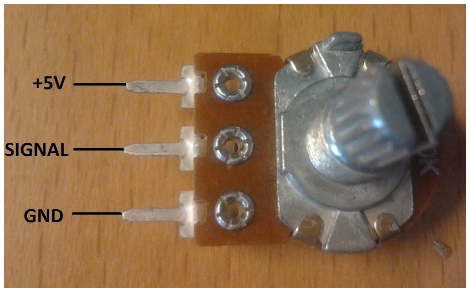
We need to import the **LiquidCrystal** library to use it.



```
#include <LiquidCrystal.h>

const int rs = 11, en = 10, d4 = 5, d5 = 4, d6 = 3, d7 = 2;

LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
```

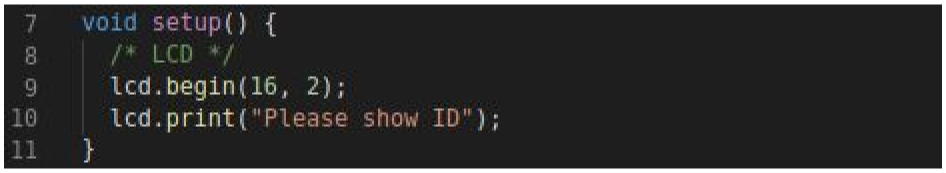
We will use the following wiring:

- **RS** : Decides whether LCD uses data register (High) or instruction register (Low).
  - Connected to **D11**
- **E** (en) - "Enabling" pin; when this pin is set to logical low, the LCD does not care what is happening with R/W, RS, and the data bus lines; when this pin is set to logical high, the - LCD is processing the incoming data.
  - Connected to **D10**
- **R/W** - Right to / Read from mode.
  - Connected to Ground (**GND**)
- **DB4 -- DB7** - 4 data bus lines, which perform read/write of data.
  - Connected to **D5 -- D2** (arbitrary)
- **VO** - Pin for LCD contrast (We will use a potentiometer to control it).
  - Connected to Potentiometer variable **signal pin**



- **Vss** / **Vdd** - Ground and Voltage supply respectively.
  - Connected to **GND** and **5V** respectively
- **A** / **K** - Used for the LCD backlight interface.
  - Connected to **5V** and **GND** respectively

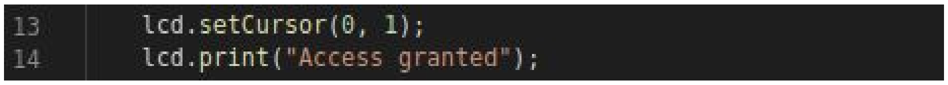
We defined previously our variable lcd, we can start by displaying a message when arduino is powered up.



```
void setup() {
  /* LCD */
  lcd.begin(16, 2);
  lcd.print("Please show ID");
}
```

There are 16x2 cells we can fill. First, we need to set the cursor with lcd.setCursor(),

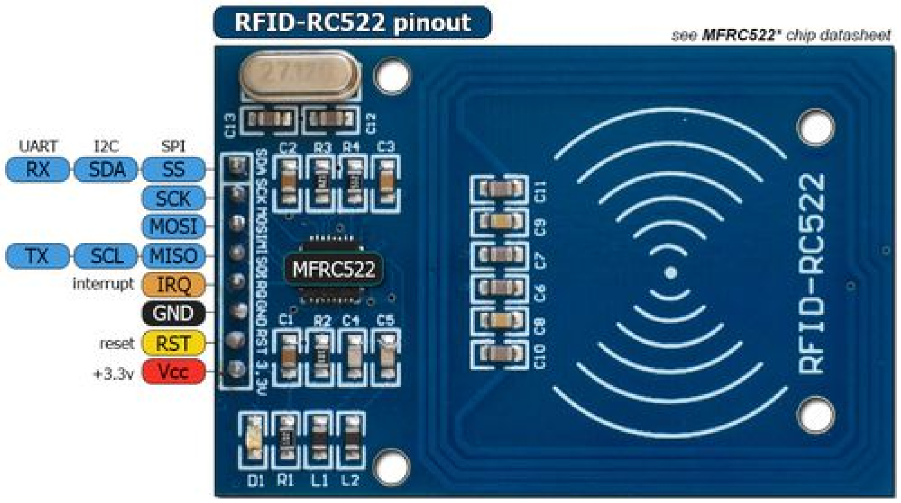
first argument is column index, second one is line index.



```
    lcd.setCursor(0, 1);
    lcd.print("Access granted");
```

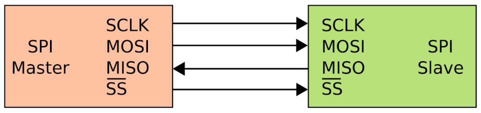
1.3- RFID Reader - RC522

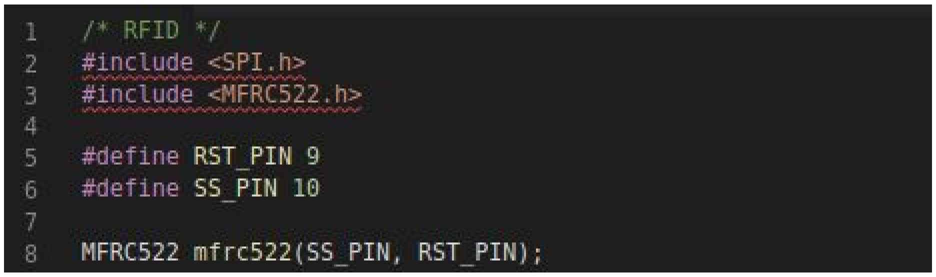
We will use the RFID reader RC522 and Mifare Classic cards for authentication. The RFID technology will be further explained on section 2. Reader has 8 pins:



We need to import the **MFRC522** library and include **SPI** library to use it.

(SPI = Serial Peripheral Interface) - synchronous serial communication, in this type of communication, SPI master controls communications with SPI slave.



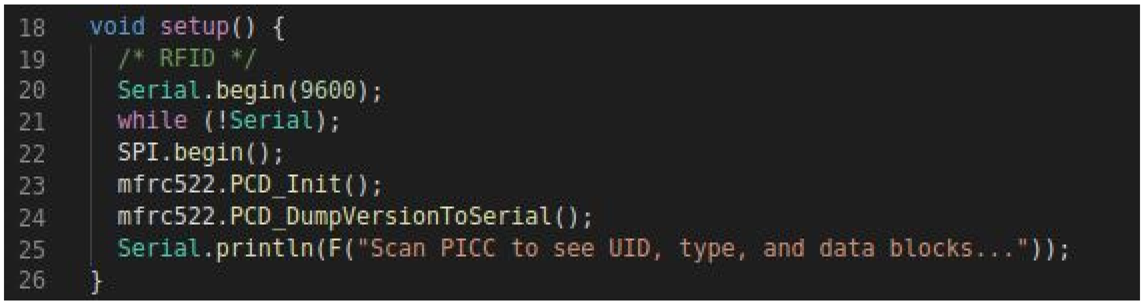


```
/* RFID */
#include <SPI.h>
#include <MFRC522.h>

#define RST_PIN 9
#define SS_PIN 10

MFRC522 mfrc522(SS_PIN, RST_PIN);
```

We will use 7 pins out of 8.

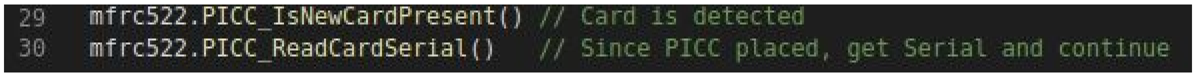
- **Vcc -** Power supply 3.3V.
  - Connected to **3.3V** pin or to **5V** with a logic shifter
- **RST -** Used for reset and to power-down reader.
  - Connected to **D9** (Arbitrary)
- **GND**
  - Connected to **GND** (Ground).
- **TX-SCL-MISO -** TX (Transmit) Acts as MISO (Master Input, Slave Output) since we're only concerned with SPI connection.
  - Connected to **D12** (Fixed)
- **MOSI -** Master Output, Slave Input
  - Connected to **D11** (Fixed)
- **SCK** (or SCLK) - Serial Clock, accepts arduino clock pulses.
  - Connected to **D13** (Fixed)
- **SS / SDA / RX -** RX (Receive) Acts as SS (Slave Select) since we're only concerned with SPI connection. This tells the slave that it should wake up and receive / send data.
  - Connected to **D10** (Arbitrary)

We can start by SPI connection and initializing reader.



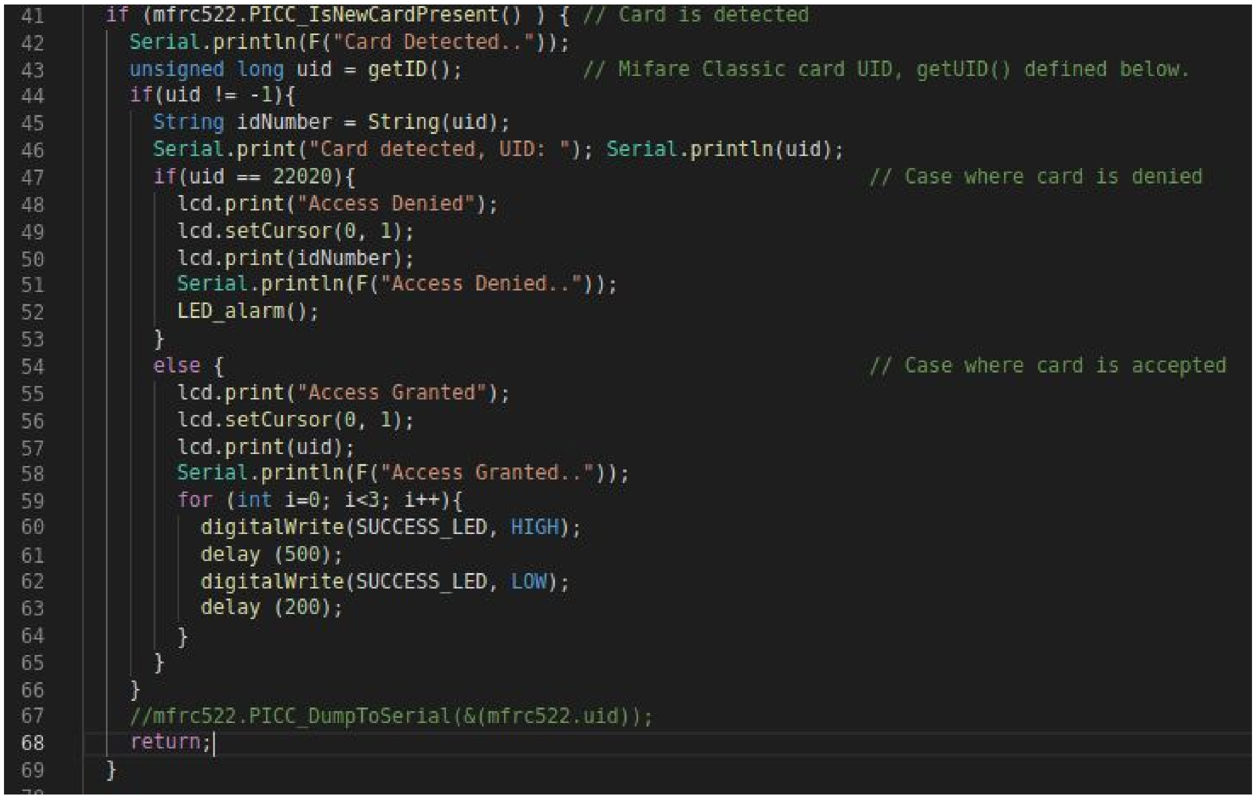
```
void setup() {
  /* RFID */
  Serial.begin(9600);
  while (!Serial);
  SPI.begin();
  mfrc522.PCD_Init();
  mfrc522.PCD_DumpVersionToSerial();
  Serial.println(F("Scan PICC to see UID, type, and data blocks..."));
}
```

We defined earlier mfrc522 variable, we can use PICC_IsNewCardPresent() to detect a new card and PICC_ReadCardSerial() to proceed serial communication.

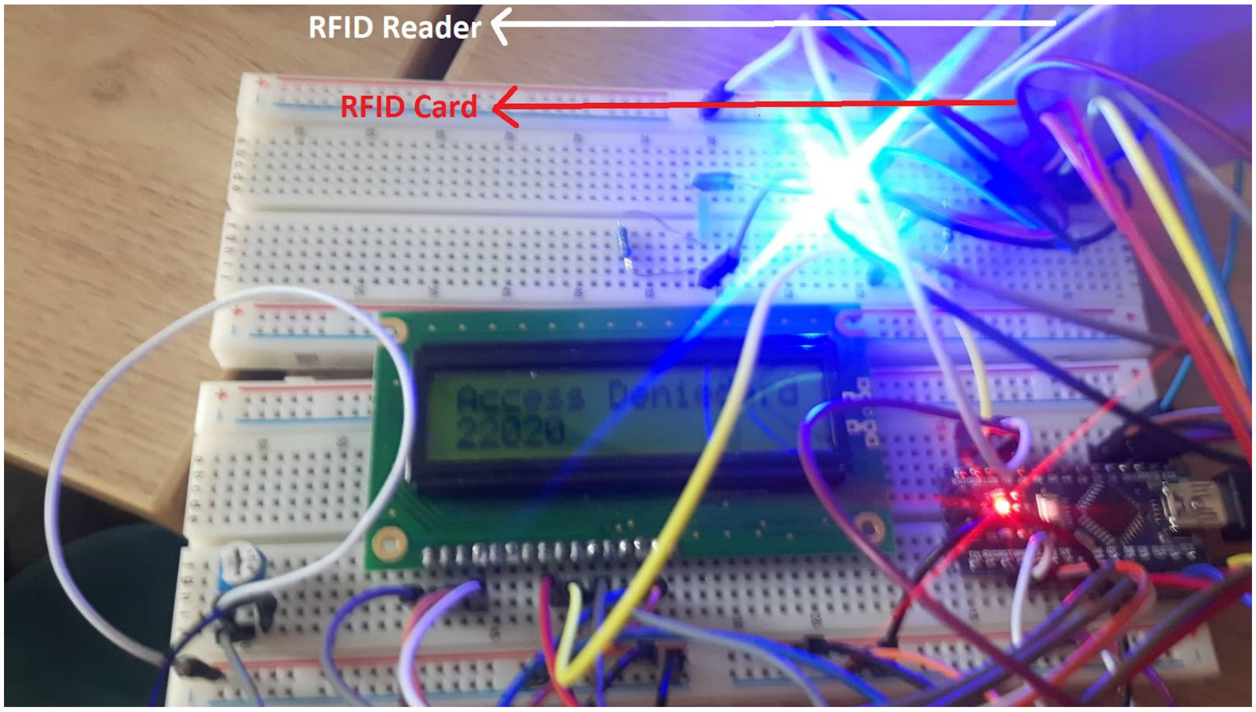


```
mfrc522.PICC_IsNewCardPresent() // Card is detected
mfrc522.PICC_ReadCardSerial()   // Since PICC placed, get Serial and continue
```

We can use it this way alongside with the LCD screen and 2 LEDs. (We moved LCD arbitrary pins since taken by reader). This code allows to identifiy a person by his card ID and to grant him access.

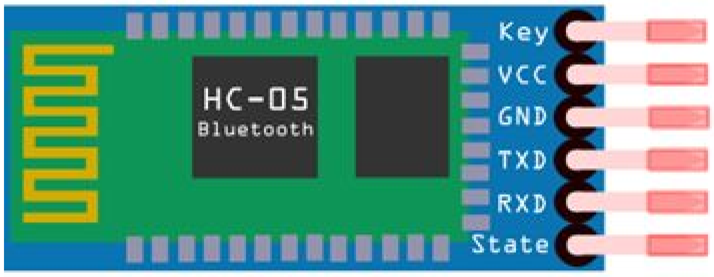


```
  if (mfrc522.PICC_IsNewCardPresent() ) { // Card is detected
    Serial.println(F("Card Detected.."));
    unsigned long uid = getID();          // Mifare Classic card UID, getUID() defined below.
    if(uid != -1){
      String idNumber = String(uid);
      Serial.print("Card detected, UID: "); Serial.println(uid);
      if(uid == 22020){                                         // Case where card is denied
        lcd.print("Access Denied");
        lcd.setCursor(0, 1);
        lcd.print(idNumber);
        Serial.println(F("Access Denied.."));
        LED_alarm();
      }
      else {                                                    // Case where card is accepted
        lcd.print("Access Granted");
        lcd.setCursor(0, 1);
        lcd.print(uid);
        Serial.println(F("Access Granted.."));
        for (int i=0; i<3; i++){
          digitalWrite(SUCCESS_LED, HIGH);
          delay (500);
          digitalWrite(SUCCESS_LED, LOW);
          delay (200);
        }
      }
    }
    //mfrc522.PICC_DumpToSerial(&(mfrc522.uid));
    return;
  }
```



1.4- Bluetooth - hc05

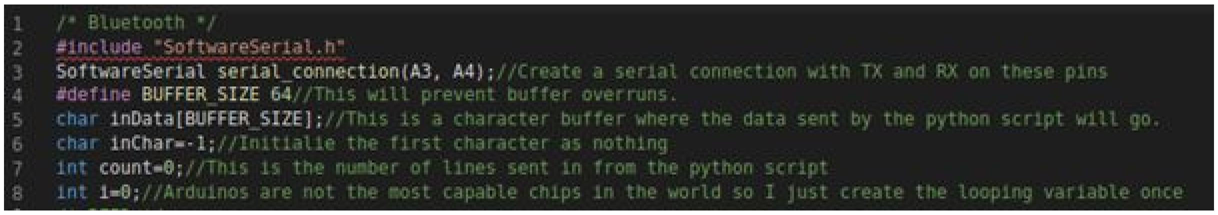
We will use bluetooth devices to make the communication bridge between the server and the coordinator controller, the communication is set an unique slave/master. For this task, I will be using a linux based server using a python script. You only have to connect the device through your bluetooth manager. We will be using the hc05 device.



We will use 5 pins out of 6:

- **Vcc** – Power supply **5V**.
    - Connected to 5V pin
- **GND**
    - Connected to **GND**
- **TX** – Transmission
    - Connected to TX pin or to an arbitrary pin used by the **SoftwareSerial** library.
- **RX** – Reception
    - Connected to RX pin or to an arbitrary pin used by the **SoftwareSerial** library.
- **State** – Allows you to use the AT mode (to configure the device)
    - Connected to an arbitrary pin defined in the code.

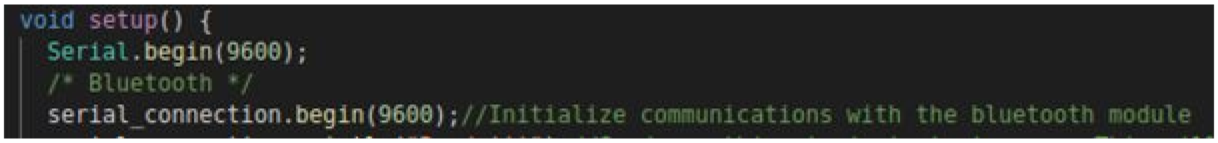
When using hc05 moduel with default parameters, we can detect it with any smartphone or computer. Once connected, we can affect a serial port to the communication. There's no need to import any library for that task. We just have to use the usual RX/TX communication or through the SoftwareSerial one.



```
/* Bluetooth */
#include "SoftwareSerial.h"
SoftwareSerial serial_connection(A3, A4);//Create a serial connection with TX and RX on these pins
#define BUFFER_SIZE 64//This will prevent buffer overruns.
char inData[BUFFER_SIZE];//This is a character buffer where the data sent by the python script will go.
char inChar=-1;//Initialie the first character as nothing
int count=0;//This is the number of lines sent in from the python script
int i=0;//Arduinos are not the most capable chips in the world so I just create the looping variable once
```

In this example, we're using the SoftwareSerial library that allows to simulate the TX/RX pins with normal ones. Then, we will define a buffer that will contain the bytes we receive from the server by bluetooth communication.

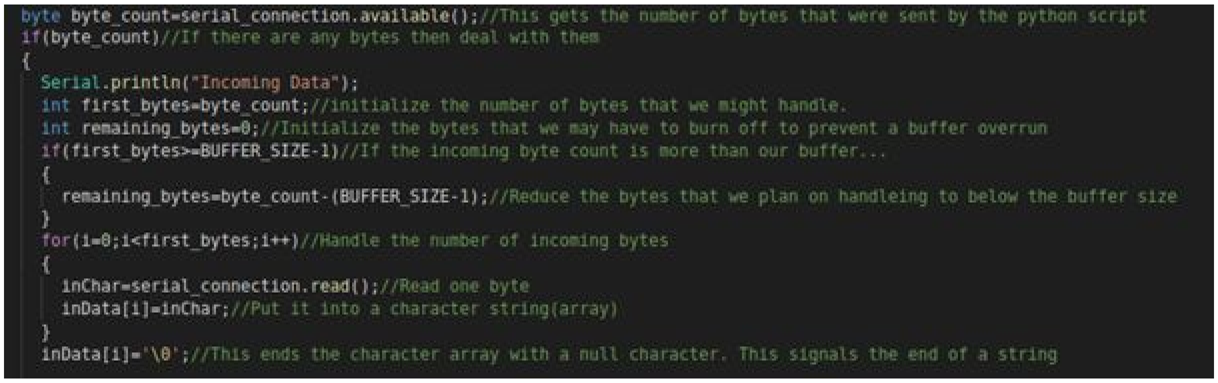
We will define our baud rate to 9600.



```
void setup() {
  Serial.begin(9600);
  /* Bluetooth */
  serial_connection.begin(9600);//Initialize communications with the bluetooth module
```

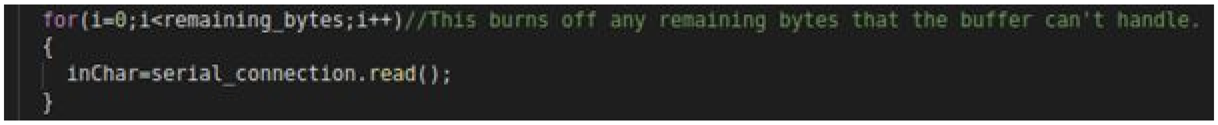
The bytes received are stored in a buffer, we can extract the buffer data by using serial_connection.read().

We start by checking if there's any available communication, then we will extract the message through in loop that goes over the number of bytes received for the message given by **serial_connection.available()**. For each iteration, we will use the **inChar** variable to get the byte then we add it to **inData[i]**.



```
byte byte_count=serial_connection.available();//This gets the number of bytes that were sent by the python script
if(byte_count)//If there are any bytes then deal with them
{
  Serial.println("Incoming Data");
  int first_bytes=byte_count;//initialize the number of bytes that we might handle.
  int remaining_bytes=0;//Initialize the bytes that we may have to burn off to prevent a buffer overrun
  if(first_bytes>=BUFFER_SIZE-1)//If the incoming byte count is more than our buffer...
  {
    remaining_bytes=byte_count-(BUFFER_SIZE-1);//Reduce the bytes that we plan on handleing to below the buffer size
  }
  for(i=0;i<first_bytes;i++)//Handle the number of incoming bytes
  {
    inChar=serial_connection.read();//Read one byte
    inData[i]=inChar;//Put it into a character string(array)
  }
  inData[i]='\0';//This ends the character array with a null character. This signals the end of a string
```

We delete the buffer when it gets more than **BUFFER_SIZE.**



```
  for(i=0;i<remaining_bytes;i++)//This burns off any remaining bytes that the buffer can't handle.
  {
    inChar=serial_connection.read();
  }
```

To send a message, we use **serial_communication.println()** or **serial_communication.write()**.

1.5- ZigBee - XBee S1

We will use zigbee devices to make the communication bridge between the coordinator and the end-device arduino controller, the communication is set on broadcast. I will be using the Xbee S1 device here. Two zigbee devices can communicate if they share the same channel and the same pan ID at least. In order to configure the zigbee devices, we need to install XCTU, which is a software brought by Digi, the constructor of Xbee S1. If needed, we must flash the Xbee S1 devices with the Xbee 802.15.4 option.

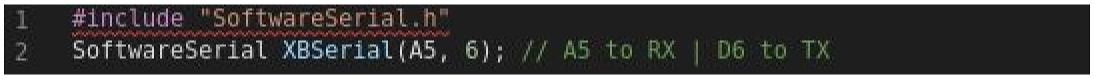
Zigbee devices are known for providing several options for the networking in IoT. it has a reach of ~100 meters and can be extended by using a **mesh** network that will forward the messages using its protocol. Though it is only possible with Xbee S2 devices. In our case, we can only use a **star** network Xbee S1 which only allows one **coordinator** for different **end-devices**.

Having encountered technical issues using a star network, I decided to communicate Xbee devices through **end-device** to **end-device** communication using a broadcast emission. This issue is proper to low level OSI layers. Thus, it doesn't affect the application level, it can be fixed without worrying about the code behind it.

For wiring, We will use 4 pins out of 5:

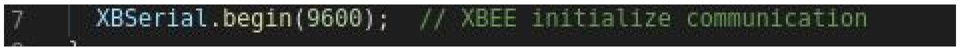
- **Vcc** – Power supply **5V**.
    - Connected to 5V pin
- **GND**
    - Connected to **GND**
- **TX** – Transmission
    - Connected to TX pin or to an arbitrary pin used by the **SoftwareSerial** library.
- **RX** – Réception
    - Connected to RX pin or to an arbitrary pin used by the **SoftwareSerial** library.

When using an Xbee S1 device, we only need to use the serial communication, pretty much like the bluetooth communication, we only need to import SoftwareSerial library.



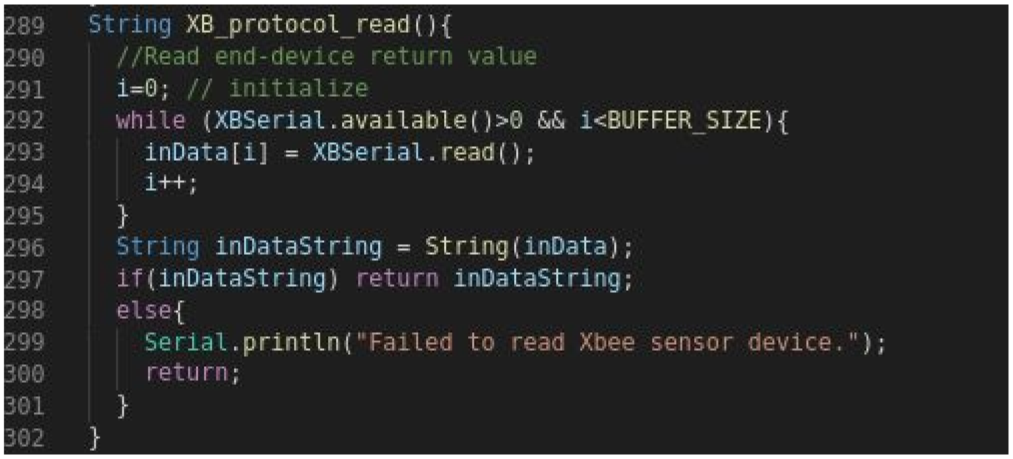
```
#include "SoftwareSerial.h"
SoftwareSerial XBSerial(A5, 6); // A5 to RX | D6 to TX
```

We start the communication in 9600 baud rate.



```
  XBSerial.begin(9600);  // XBEE initialize communication
```

As for bluetooth reading procedure, we can use it for zigbee using this function.

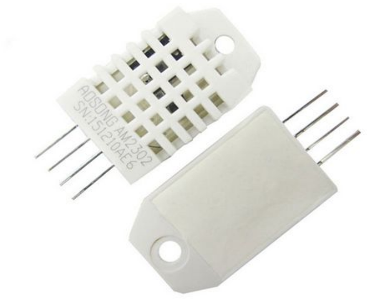


```
String XB_protocol_read(){
  //Read end-device return value
  i=0; // initialize
  while (XBSerial.available()>0 && i<BUFFER_SIZE){
    inData[i] = XBSerial.read();
    i++;
  }
  String inDataString = String(inData);
  if(inDataString) return inDataString;
  else{
    Serial.println("Failed to read Xbee sensor device.");
    return;
  }
}
```

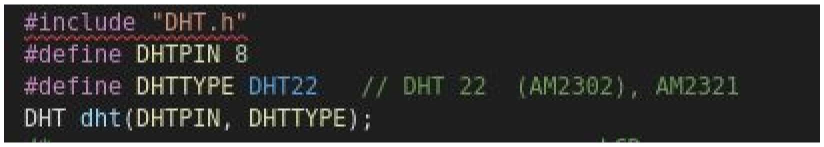
To send a message, we can use **XBSerial.write()**.

1.6- Sensor - ASAIR AM2302

We will use this sensor for humidity and temperature, we need to to import the DHT library and to use the following configuration, defining the DHTPIN you want.



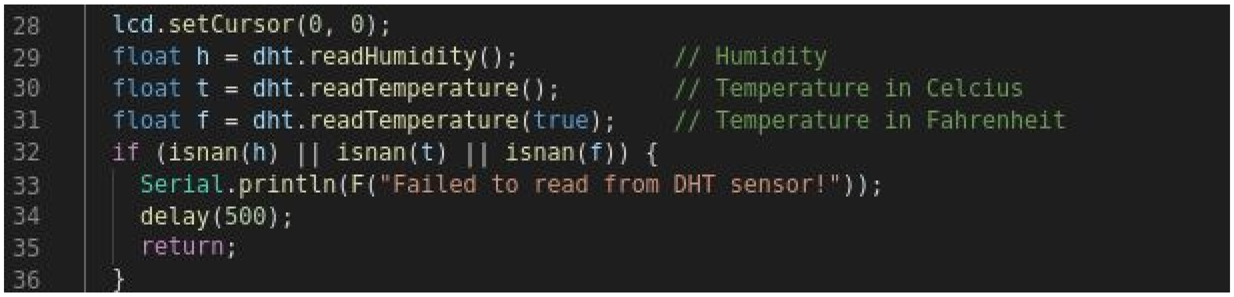
- **Vcc** – Power supply **5V**.
  - Connected to 5V pin
- **GND**
  - Connected to **GND**
- **Data** – Transmission
  - Connected to an arbitrary pin.



```
#include "DHT.h"
#define DHTPIN 8
#define DHTTYPE DHT22   // DHT 22  (AM2302), AM2321
DHT dht(DHTPIN, DHTTYPE);
```

We need to start the device by using **dht.begin().**

Then, we can take the measurements as following.



```
lcd.setCursor(0, 0);
float h = dht.readHumidity();           // Humidity
float t = dht.readTemperature();        // Temperature in Celcius
float f = dht.readTemperature(true);    // Temperature in Fahrenheit
if (isnan(h) || isnan(t) || isnan(f)) {
  Serial.println(F("Failed to read from DHT sensor!"));
  delay(500);
  return;
}
```

2- RFID Security

This section comes as the research part of my internship, I try to explain very specifically how security works in the RFID system I'm using, if you're only concerned by the functional part, please skip this part.

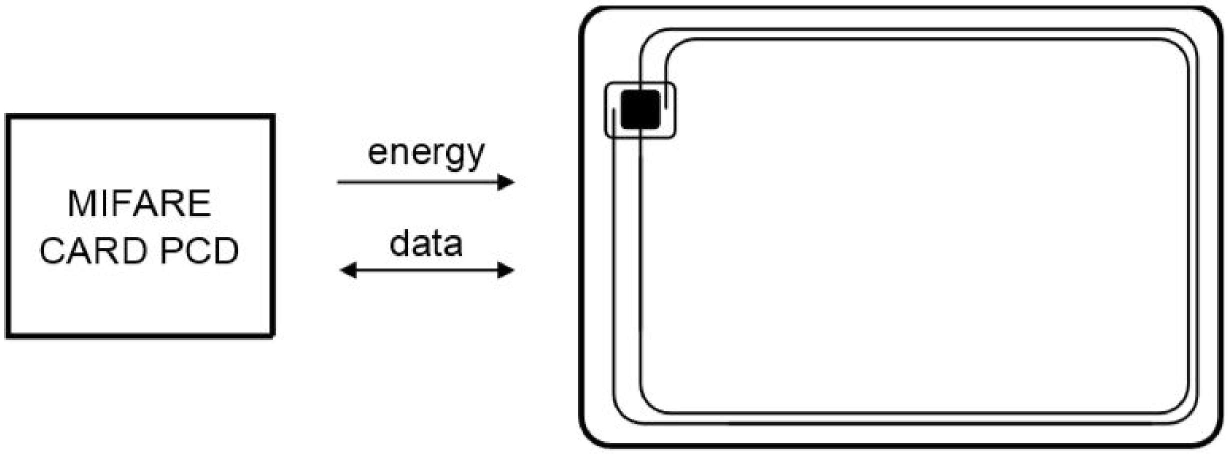
2.1- RFID Security

When it comes to smart buildings, authentication plays a major role in securing the whole system. Breaching the entry point can lead to severe consequences in a system where differents users have differents access rights to different devices. Authentication comes as a first security layer to our system. RFID (Radio Frequency Identification) is an efficient way to ensure a fast and secure authentication. In our case, we will be focusing on a specific smart card which is the Mifare Classic 1K using the RC522 RFID Reader over an arduino circuit. An arduino solution comes with the idea of a cheap way to make a smart home monitoring system and Mifare Classic cards (or tags) are widely known for their low prices. They rely on a proprietary cryptographic algorithm CRYPTO1 for authentication and encryption. The algorithm has been partially reverse engineered in 2007 [1] and completely reverse engineered in 2008 [2]. Section 2.2 describes the hardware structure of the card and covers its weaknesses. Section 2.3 shows the CRYPTO1 authentication protocol followed by the complete algorithm. Section 2.4 discusses the different weaknesses of the algorithms against exhaustive attacks and ciphertext-only cryptanalysis. Solutions over different layers are proposed in section 2.5. Finally, a summary and a conclusion are given in section 2.6.

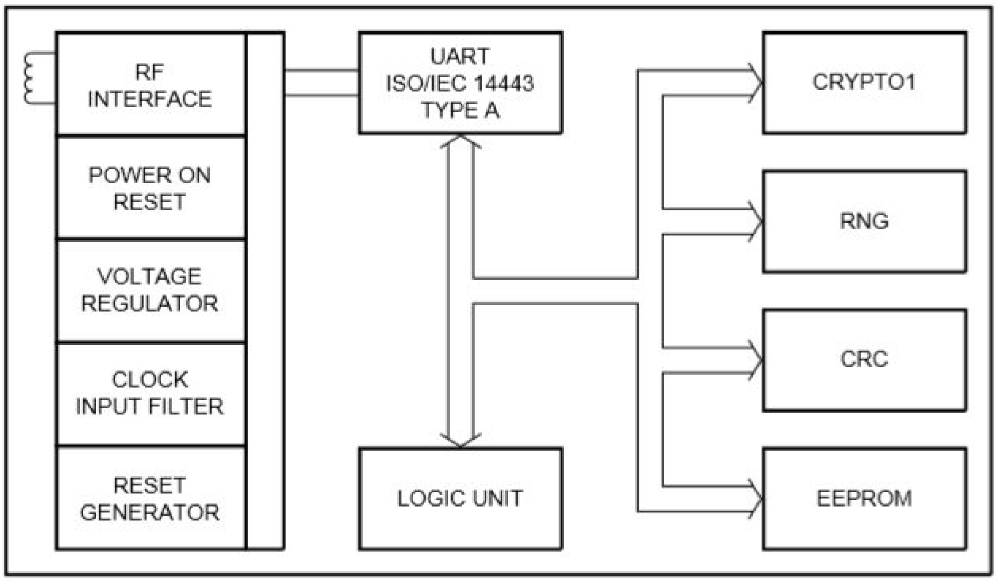
2.2- Mifare Classic

The Mifare Classic belongs to the family of passive RFID tags. Manufacturer delivers tags with random 7-bytes or 4-bytes UID. Operating distance goes up to 100 mm with a data transfer of 106 kbits/s on an operating frequency of 13.56 Mhz.

PCD (Proximity Coupling Device) delivers energy to the tag and tag responds with data transfer.



The tag hardware structure is composed of several layers that ensure an efficient communication. We will be focusing on the integrity, confidentiality and availability part.

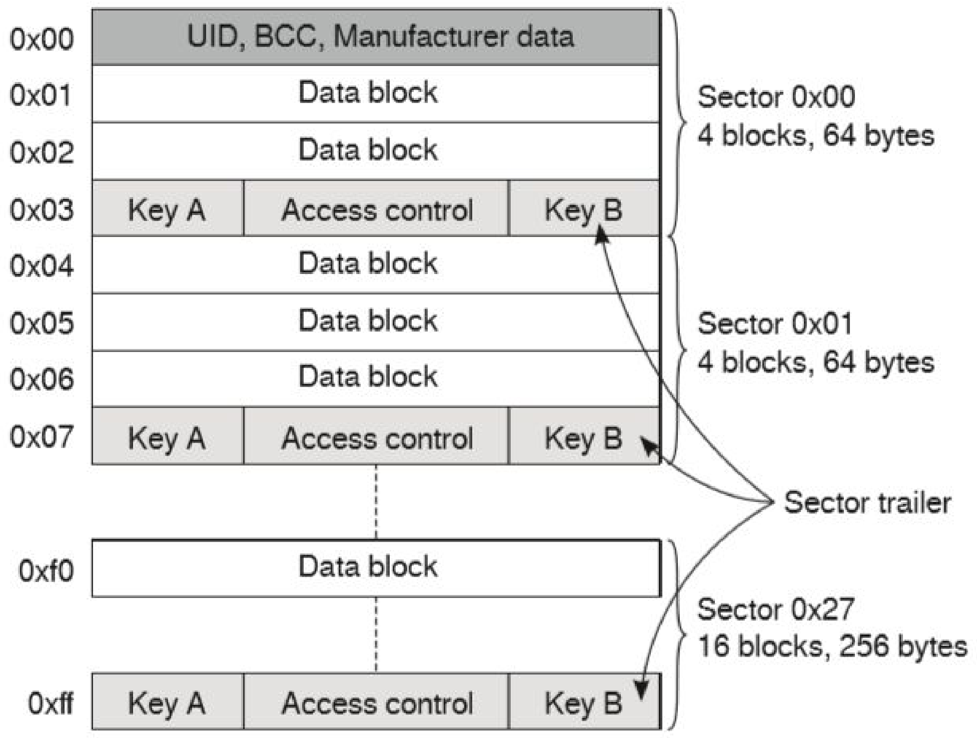


The above block diagram shows the different communication layers of the tag. We will only focus on the four blocks processed by the Logic Unit.

- **CRYPTO1** : Mifare Classic originally relied on a security-through-obscurity encryption with an algorithm called CRYPTO1. This part ensures the confidentiality of the communication. We will focus more on it in section 3 and 4.
- **RNG** : The random number generator block is used for the authentication part. We will also talk more about it in section 3 and 4.
- **CRC-16** : The cyclic redundancy check block ensures the integrity of datas, for each block of data it calculates a hash that will be compared with a freshly calculated hash on the reader side.
- **EEPROM** : This is where data is stored including the UID, the differents keys and the manufacturer data.

2.2.1 - Memory Structure

The Mifare Classic EEPROM memory is structured in different and independent sectors. Each sector has 4 blocks of 16 bytes each. In our case, there are 16 sectors.



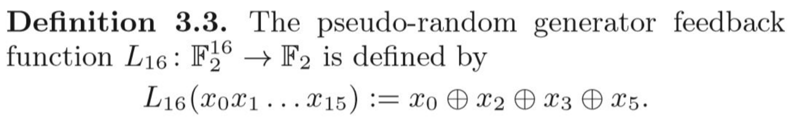
We have 16 sectors, 256 blocks and each block has 16 bytes of data and hence the card has 1024 kB of data.

The first block **0x00** of the first sector is only used to store the UID and the manufacturer informations. The last block of each sector is used to store authentication keys A and B, each one of the two keys has specific access control bytes stored in the same block.

2.2.2 - Hardware Weaknesses

We're only concerned about weaknesses of components directly linked to the CRYPTO1 block in figure 2. CRYPTO1 uses a stream cypher encryption for the authentication and communication part. The algorithm uses a 48-bit key, thus making an exhaustive attack easy already. Since the algorithm wasn't made public, there couldn't be an offline attack before it was reverse engineered. Every attempt would've taken 6 milliseconds [1], thus making an online exhaustive attack too long.

The RNG block also presents severe flaws. Random numbers are used to make 32-bit nonces used for the authentication phase where each bit is generated by a 16-bit LFSR (Linear Feedback Shift Register). As shown in [2]:



Definition 3.3. The pseudo-random generator feedback function $L_{16}\colon \mathbb{F}_2^{16} \to \mathbb{F}_2$ is defined by

$$L_{16}(x_0x_1 \dots x_{15}) := x_0 \oplus x_2 \oplus x_3 \oplus x_5.$$

The register also wraps every 0.6 seconds after generating 65,655 possible output values [1]. The register is also unnecessarily reset to a known state when tag is powered up. Hence, by controlling the timing of the protocol we can control the generated number (nonce).

CRYPTO1 is fully implemented in hardware and hasn't been peer-reviewed since it's been secret until 2008. The cryptography functions make up about 400 2-NAND (Ge) gates which is very light compared to a small implementation of an AES block cipher which requires 3400 Ges. It is very fast, it is very flawed though.

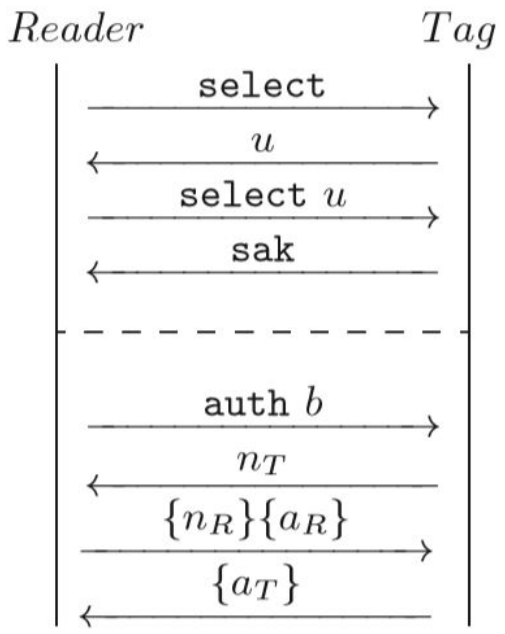
Mifare Classic cards passively answer requests from any reader, they are vulnerable to cloning and spoofing. Cloning basically means copying all datas from a certain tag and replicating it on a new card with a different ID. The copy card can still be easily detected since it has a different ID (IDs are generally read-only). Spoofing involves reading and recording data transmission from a tag and faking the ID when retransmitting the data, thus making it appear to be valid.

2.3- CRYPTO1 Algorithm

CRYPTO1 uses a 48-bit LFSR and a 48-bit key. The LFSR of CRYPTO1 can be seen as an engine (register) that is constantly shifted and fed with new inputs calculated from the previous state.

2.3.1- Authentication

First, we need to define the authentication protocol. When a tag is in contact with the reader, reader selects the tag and the tag answers with its UID.



In order to access each sector, reader has to successfully authenticate since each sector can have its own keys. For instance, reader wants to authenticate to sector b. The tag generates a random number **n_T** (nonce) and sends it to reader as a challenge.

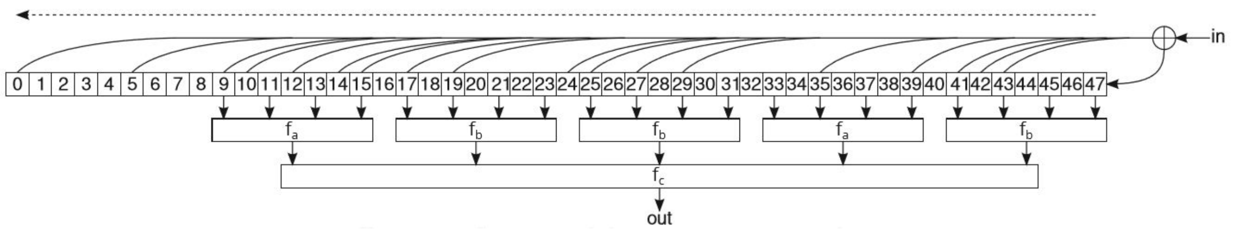
Starting from this point, encryption begins. Reader also generates a random number **n_R** and calculates an answer for the tag challenge **a_R = succ^{64}(n_T)** such as [2]:

Definition 3.4. The successor function suc: $\mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ is defined by

$$\mathrm{suc}(x_0 x_1 \dots x_{31}) := x_1 x_2 \dots x_{31} L_{16}(x_{16} x_{17} \dots x_{31})\,.$$

L_{16} is defined above in 2.2. We denote encryptions by {.} and define {**n_R**} and {**a_R**} ∈ GF(2). Reader then sends {**n_R**}{**a_R**} concatenated. Tag finally calculates the answer **a_T = succ^{64}(n_R)** and both are authenticated if **a_R** and **a_T** are mutually correct.

2.3.2- Encryption



When authentication starts, the register is fed with the 48-bit key. At each clock cycle, the stream cipher makes a left shift and adds a new bit according to the following definitions:

Definition 3.1. The cipher feedback function $L\colon \mathbb{F}_2^{48} \to \mathbb{F}_2$ is defined by $L(x_0x_1\ldots x_{47}) := x_0 \oplus x_5 \oplus x_9 \oplus x_{10} \oplus x_{12} \oplus x_{14} \oplus x_{15} \oplus x_{17} \oplus x_{19} \oplus x_{24} \oplus x_{25} \oplus x_{27} \oplus x_{29} \oplus x_{35} \oplus x_{39} \oplus x_{41} \oplus x_{42} \oplus x_{43}$.

Definition 3.5. Given a key $k = k_0k_1\ldots k_{47} \in \mathbb{F}_2^{48}$, a tag nonce $n_T = n_{T_0}n_{T_1}\ldots n_{T_{31}} \in \mathbb{F}_2^{32}$, a UID $u = u_0u_1\ldots u_{31} \in \mathbb{F}_2^{32}$, and a reader nonce $n_R = n_{R_0}n_{R_1}\ldots n_{R_{31}} \in \mathbb{F}_2^{32}$, the internal state of the cipher at time i is $\alpha_i := a_ia_{i+1}\ldots a_{i+47} \in \mathbb{F}_2^{48}$.

Here the $a_i \in \mathbb{F}_2$ are given by

$$\begin{aligned}
a_i &:= k_i & \forall i \in [0,47]\\
a_{48+i} &:= L(a_i,\ldots,a_{47+i}) \oplus n_{T_i} \oplus u_i & \forall i \in [0,31]\\
a_{80+i} &:= L(a_{32+i},\ldots,a_{79+i}) \oplus n_{R_i} & \forall i \in [0,31]\\
a_{112+i} &:= L(a_{64+i},\ldots,a_{111+i}) & \forall i \in \mathbb{N}.
\end{aligned}$$

Furthermore, we define the keystream bit $ks_i \in \mathbb{F}_2$ at time i by

$$ks_i := f(a_ia_{1+i}\ldots a_{47+i}) \qquad \forall i \in \mathbb{N}.$$

The 48-bit key is only used to to initialize the LSFR, the keystream bit is defined by the state of the LSFR register and during the authentication, $\mathbf{n_R}$ and $\mathbf{a_R}$ are XORed with keystream bits as below:

$$\begin{aligned}
\{n_{R_i}\} &:= n_{R_i} \oplus ks_{32+i} & \forall i \in [0,31]\\
\{a_{R_i}\} &:= a_{R_i} \oplus ks_{64+i} & \forall i \in [0,31]
\end{aligned}$$

For the first authentication, the tag challenge $\mathbf{n_T}$ is not encrypted. Though, if we authenticate to a second sector subsequently, $\mathbf{n_T}$ will be encrypted and literature [2] defines it as nested authentication.:

Definition 3.6. In the situation from Definition 3.5, we define $\{n_{T_i}\} \in \mathbb{F}_2$ by $\{n_{T_i}\} := n_{T_i} \oplus ks_i \forall i \in [0, 31]$

The keystream bits are defined by the filter function as shown in figure 5. It only depends on current state of the stream cipher and since the stream cipher is initialized with the 48-bit key, it also directly depends on the key.

Definition 3.4.2. The filter function $f\colon \mathbb{F}_2^{48} \to \mathbb{F}_2$ is defined by

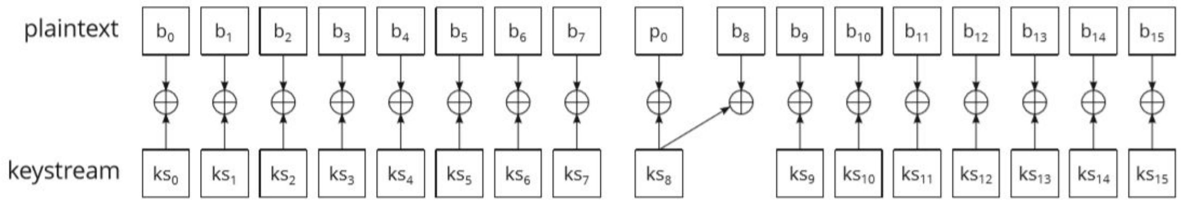
$$f(x_0x_1 \dots x_{47}) := f_c(f_a(x_9, x_{11}, x_{13}, x_{15}), f_b(x_{17}, x_{19}, x_{21}, x_{23}), f_b(x_{25}, x_{27}, x_{29}, x_{31}), f_a(x_{33}, x_{35}, x_{37}, x_{39}), f_b(x_{41}, x_{43}, x_{45}, x_{47})).$$

Here $f_a, f_b\colon \mathbb{F}_2^4 \to \mathbb{F}_2$ and $f_c\colon \mathbb{F}_2^5 \to \mathbb{F}_2$ are defined by $f_a(y_0, y_1, y_2, y_3) := ((y_0 \vee y_1) \oplus (y_0 \wedge y_3)) \oplus (y_2 \wedge ((y_0 \oplus y_1) \vee y_3))$, $f_b(y_0, y_1, y_2, y_3) := ((y_0 \wedge y_1) \vee y_2) \oplus ((y_0 \oplus y_1) \wedge (y_2 \vee y_3))$, and $f_c(y_0, y_1, y_2, y_3, y_4) := (y_0 \vee ((y_1 \vee y_4) \wedge (y_3 \oplus y_4))) \oplus ((y_0 \oplus (y_1 \wedge y_3)) \wedge ((y_2 \oplus y_3) \vee (y_1 \wedge y_4)))$.

Overall, CRYPTO1 uses the filter function to generate the keystream bits. Retrieving the keystream bits allows to retrieve the plaintext since encryption only consists in XORing plaintext with keystream bits.

2.4- CRYPTO1 Flaws

In order to ensure the integrity of the communication, the tag generates one parity bit for each byte that is communicated. Though, parity bits are computed over the plaintext and encrypted with the first keystream bit that is used to encrypt the next byte.


plaintext
b0 b1 b2 b3 b4 b5 b6 b7 p0 b8 b9 b10 b11 b12 b13 b14 b15
keystream
ks0 ks1 ks2 ks3 ks4 ks5 ks6 ks7 ks8 ks9 ks10 ks11 ks12 ks13 ks14 ks15

Definition 4.1. In the situation from Definition 3.5, we define the parity bits $p_j \in \mathbb{F}_2$ by

$$p_j := \bigoplus_{i=0}^{7} n_{T_{8j+i}} \oplus 1 \qquad \forall j \in [0,3]$$

and the encryptions $\{p_j\}$ of these parity bits by

$$\{p_j\} := p_j \oplus ks_{8j+8} \qquad \forall j \in [0,3]$$

Let $\{\mathbf{p_0}\}$ be the encrypted parity bit of the first byte $\{\mathbf{n_{T[0,7]}}\}$ and $\{\mathbf{n_{T8}}\}$ the first encrypted bit of the second byte {$\mathbf{n}_{T[8,15]}$}. We can deduce whether or not $\mathbf{n}_{T8}$ and $\mathbf{p}_0$ are equal since they are XORed with the same keystream bit.

CRYPTO1 was initially advertised with a 48-bit resistant encryption, which can be considerably reduced in order to retrieve the key. If we consider the nested authentication defined in 3.2 and the weak random number generation, we can retrieve 32 bits of the keystream generated by the 16-bit LSFR and used to encrypt $\mathbf{n}_T$. Since we can predict $\mathbf{n}_T$, we can XOR it with {$\mathbf{n}_T$} in order to find it.

The filter function used to generate the keystream bits only uses odd indexed bits from the main 48-bit LSFR as shown in figure 5 ie. $a_{9,}$ a_{11}, .., $a_{47.}$ This considerably reduces the computing power required for an exhaustive search from 2^{48} to 2^{39} since we start from a_9. This can even be further reduced to ~2^{20} if we split the cipher feedback into two parts, even and odd indexed bits. Combining them reduces the computing power to ($2^{20} + 2^{19}$) Proof. in [2].

Moreover, the authentication protocol leaks 4 keystream bits through errors. During the three pass authentication, if the answer a_R is wrong, tag sends a 4-bit error code 0x05 indicating a failed authentication. The error code is sent encrypted, thus leaking 4 keystream bits.

There are even more sophisticated ways to reduce even further the candidate keys for an exhaustive search in literature [2]. This also gives us enough informations about how security is initially dealt with Mifare Classic cards and gives us clues about where we should enhance security.

2.5- Solutions

In this section will discuss eventual solutions that could be applied in a smart building situation. Considering a backend server and a database storage, we will have to operate on different level in the arduino based system. We will consider enhancing security at the tag level, server side and reader wise. We will try to discuss solutions for different security issues ie. identification, authentication, integrity, authorization and accounting.

2.5.1- Card Security

There isn’t much we can do to enhance card security since everything is implemented in hardware and there isn’t much we can operate in apart from the EEPROM. UID is read-only in our case, which makes cloning the card a bit more difficult to make than usual, but with cheap hardware it is still possible to spoof the card. We cannot operate much in the CRYPTO1 protocol since the card executes the protocol with no condition. As such, we can only depend on CRYPTO1 to ensure authentication card-wise. However, we can operate the EEPROM data integrity. By adding an additional encryption layer, we could prevent a clone card to get information from the card. This can be done by adding an AES128 layer to encrypt stored datas. Initially, the Rjindael algorithm used for AES was designed for smart cards [3], and was also meant to be used as a hash method. It has been successfully implemented but AES128 isn’t strong enough for a hash by our standards today but it fits well with the 16-bytes blocks. We could apply AES256 by encrypting two sectors in a row which would enhance security. The attacker would still successfully authenticate though.

Each sector has a proper key for authentication. Manufacturer deliver the card with the same key for every key. Changing the key for each sector can improve security as it would take longer to clone.

Accounting requires a lot of datas, thus it doesn't fit in a 1kB EEPROM. Since CRYPTO1 protocol is very fast, we can possibly add an additional protocol where we put a new UID and process the communication on server-side.

2.5.2- Server Security

Since the whole system is connected to a server, authentication should be heavily enticed to it. Every connection should be normally logged. To prevent clone cards to operate efficiently, we can check the logs before granting access. For instance, we can check if the person is already inside, that would requires him to scan his card when he leaves too. We can check if the person is actually supposed to be in that very place, if he's on holiday or if he's not supposed to have access at that time. We can also decide whether we store encrypted datas on the card or we store it on a database or an in-between solution. Since Mifare Classic cards are vulnerable against cloning we cannot completely ensure a proper authentication. We can though enhance the security procedure by adding external parameters to reduce the probabilities of an intrusion. Another solution can be applied where we add a second factor authentication by adding a password. This can be done with a low price by adding a 3x3 keypad to the arduino controller. Though it would take much longer to authenticate compared to an RFID alone authentication.

We can store the hash of every tag on a database to check the integrity of the card. This requires to have different keys for each sectors in order to make it efficient.

2.5.3- Reader Security

If we consider adding a new UID in one of the available blocks, we can enhance security against cloning. For each authentication, we can change the UID on the card and update it on the server. The next authentication would require the new UID. For instance, let Alice be the original owner and Bob the intruder. If Bob clones Alice's card and Alice authenticates afterward before Bob does, Bob won't have access since the UID has changed. If Bob authenticates before Alice does, he would have access, but when Alice tries to authenticate afterward, she would know that someone cloned her card and would report to the administrators or get access with an alternative authentication method.

2.6- Conclusion

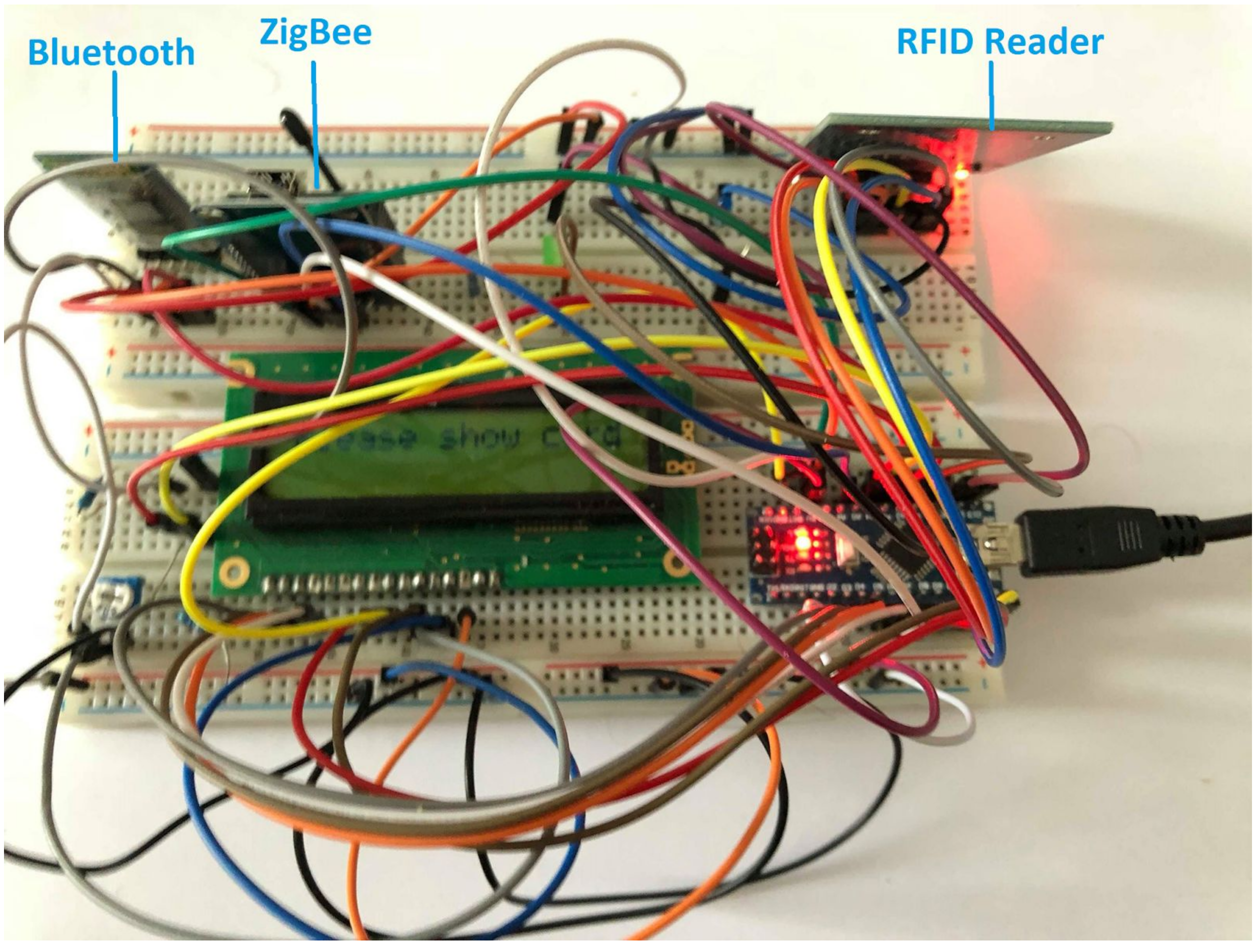
RFID is an efficient technology for storage and quick authentication. Today's RFID chips are more advanced and use advanced encryption such as AES and ECC. Though it comes at different prices, Mifare Classic are easy to get and come at a cheap price. It is though very poorly encrypted knowing that AES was already a standard in 1999 and Mifare Classics continued to be the main RFID product 10 years after, which support the idea that security-through-obscurity doesn't work since peer-reviewing is very important to assess the quality of an algorithm. Although, security operates at different independant layers and solution can be found. If we focus on an entry level Smart House system, we can find compromises to operate safely and work with an efficient technology at an entry level price.

There is a code, normally attached to this document, that puts AES128 encrypted data in RFID cards.

3- MONITORING SYSTEM

Due to technical limitations, I had to use the same machine for server and user.

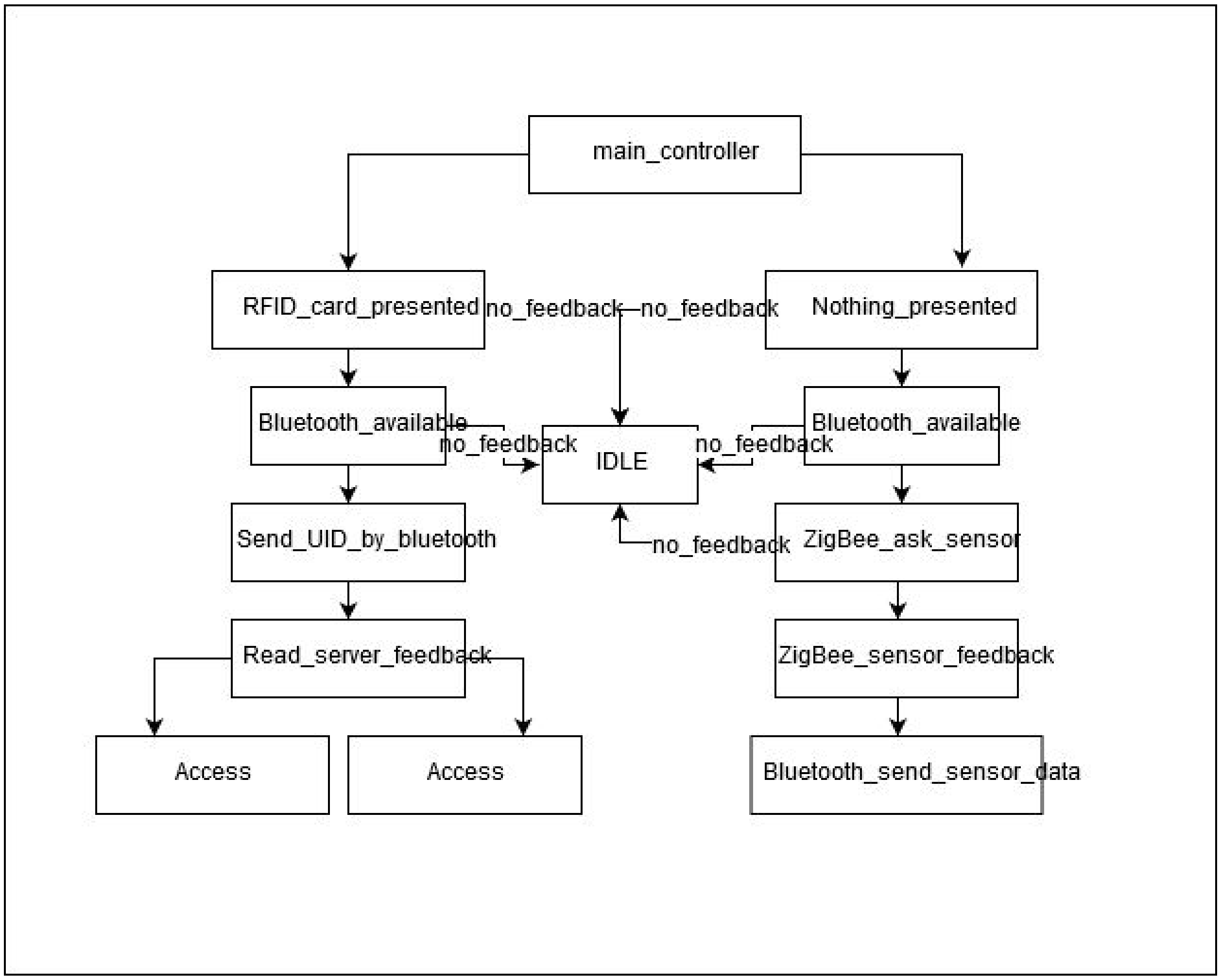
3.1- Coordinator

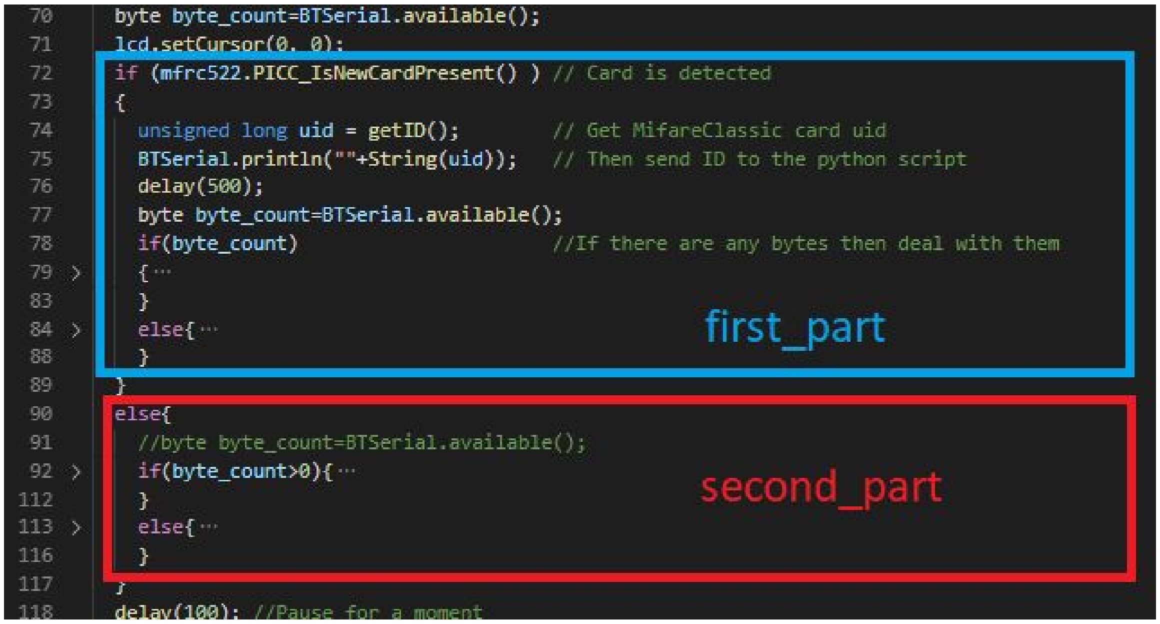


The coordinator is the main bridge between sensors and user/server. It will be loaded with different modules: Bluetooth, ZigBee, LCD, RFID and some LEDs.

RFID	Bluetooth	ZigBee	LCD	LEDs
RST_PIN 9 **SS_PIN 10**	**RX A3** **TX A4**	**RX A5** **TX 6**	**RS 8** **EN 7** **D4 5** **D5 4** **D6 3** **D7 2**	**SUCCESS_LED A1** **FAILURE_LED A0**

The following diagram shows how the system works independently.

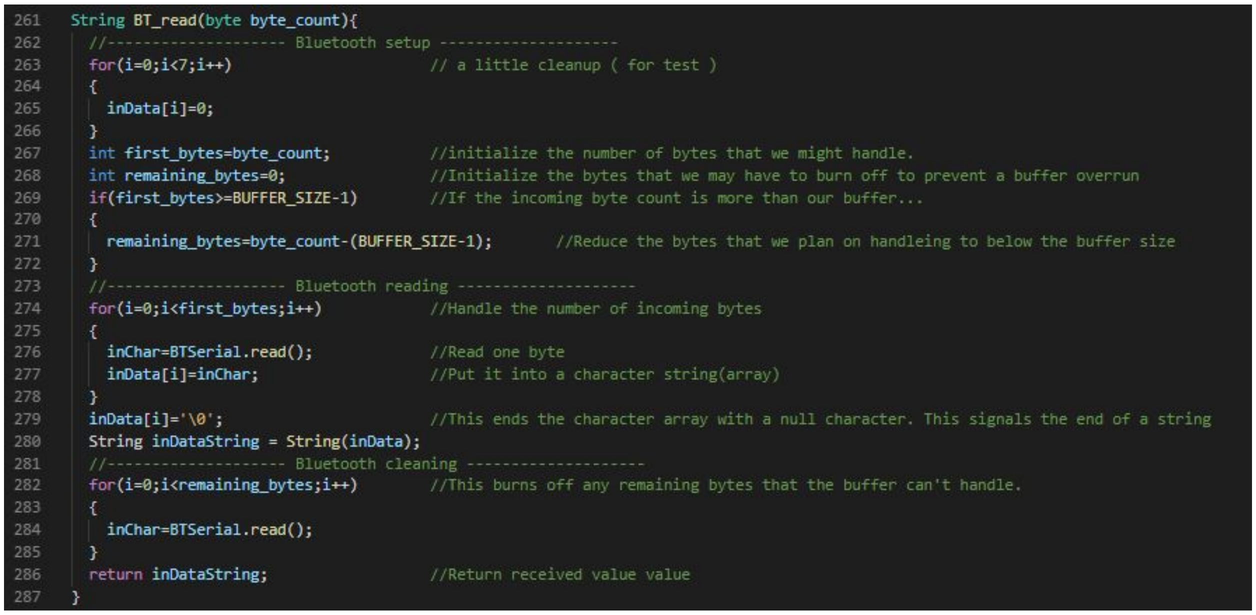




```
  byte byte_count=BTSerial.available();
  lcd.setCursor(0, 0);
  if (mfrc522.PICC_IsNewCardPresent() ) // Card is detected
  {
    unsigned long uid = getID();        // Get MifareClassic card uid
    BTSerial.println(""+String(uid));   // Then send ID to the python script
    delay(500);
    byte byte_count=BTSerial.available();
    if(byte_count)                      //If there are any bytes then deal with them
    {⋯
    }
    else{⋯
    }
  }
  else{
    //byte byte_count=BTSerial.available();
    if(byte_count>0){⋯
    }
    else{⋯
    }
  }
  delay(100); //Pause for a moment
```

Two options for controller, either it detects an RFID card or it just waits for instructions from Bluetooth communication. The RFID part can be used to grant access, the second part is meant to display the communication protocol in order to get it working.

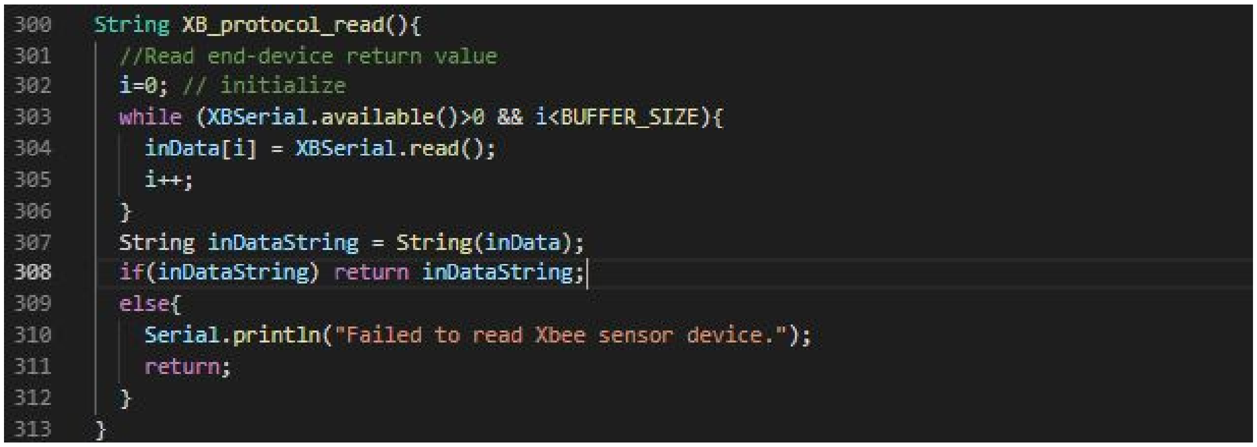
This is the method used for bluetooth detection:



```
String BT_read(byte byte_count){
  //-------------------- Bluetooth setup --------------------
  for(i=0;i<7;i++)                      // a little cleanup ( for test )
  {
    inData[i]=0;
  }
  int first_bytes=byte_count;           //initialize the number of bytes that we might handle.
  int remaining_bytes=0;                //Initialize the bytes that we may have to burn off to prevent a buffer overrun
  if(first_bytes>=BUFFER_SIZE-1)        //If the incoming byte count is more than our buffer...
  {
    remaining_bytes=byte_count-(BUFFER_SIZE-1);       //Reduce the bytes that we plan on handleing to below the buffer size
  }
  //-------------------- Bluetooth reading --------------------
  for(i=0;i<first_bytes;i++)            //Handle the number of incoming bytes
  {
    inChar=BTSerial.read();             //Read one byte
    inData[i]=inChar;                   //Put it into a character string(array)
  }
  inData[i]='\0';                       //This ends the character array with a null character. This signals the end of a string
  String inDataString = String(inData);
  //-------------------- Bluetooth cleaning --------------------
  for(i=0;i<remaining_bytes;i++)        //This burns off any remaining bytes that the buffer can't handle.
  {
    inChar=BTSerial.read();
  }
  return inDataString;                  //Return received value value
}
```

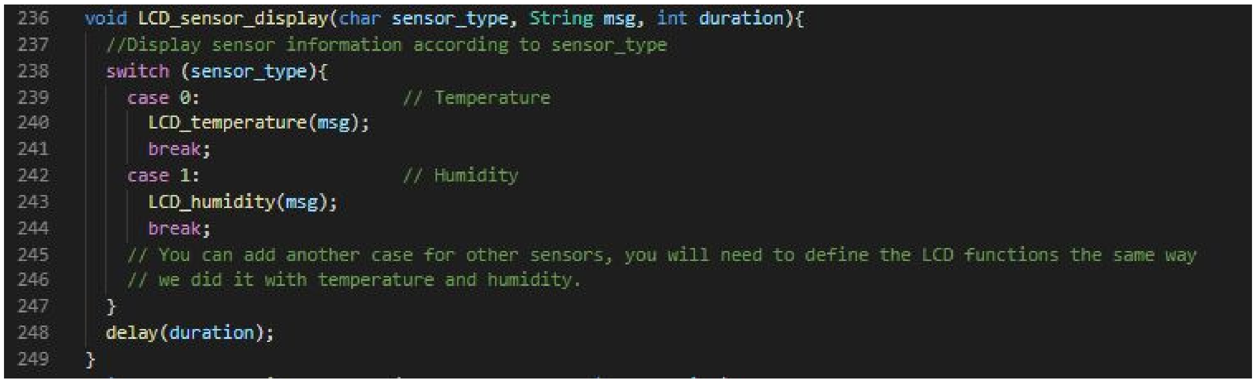
It will extract data from the arduino buffer byte by byte, then it will will process the information regarding the meant protocol.

It works the same for ZigBee:



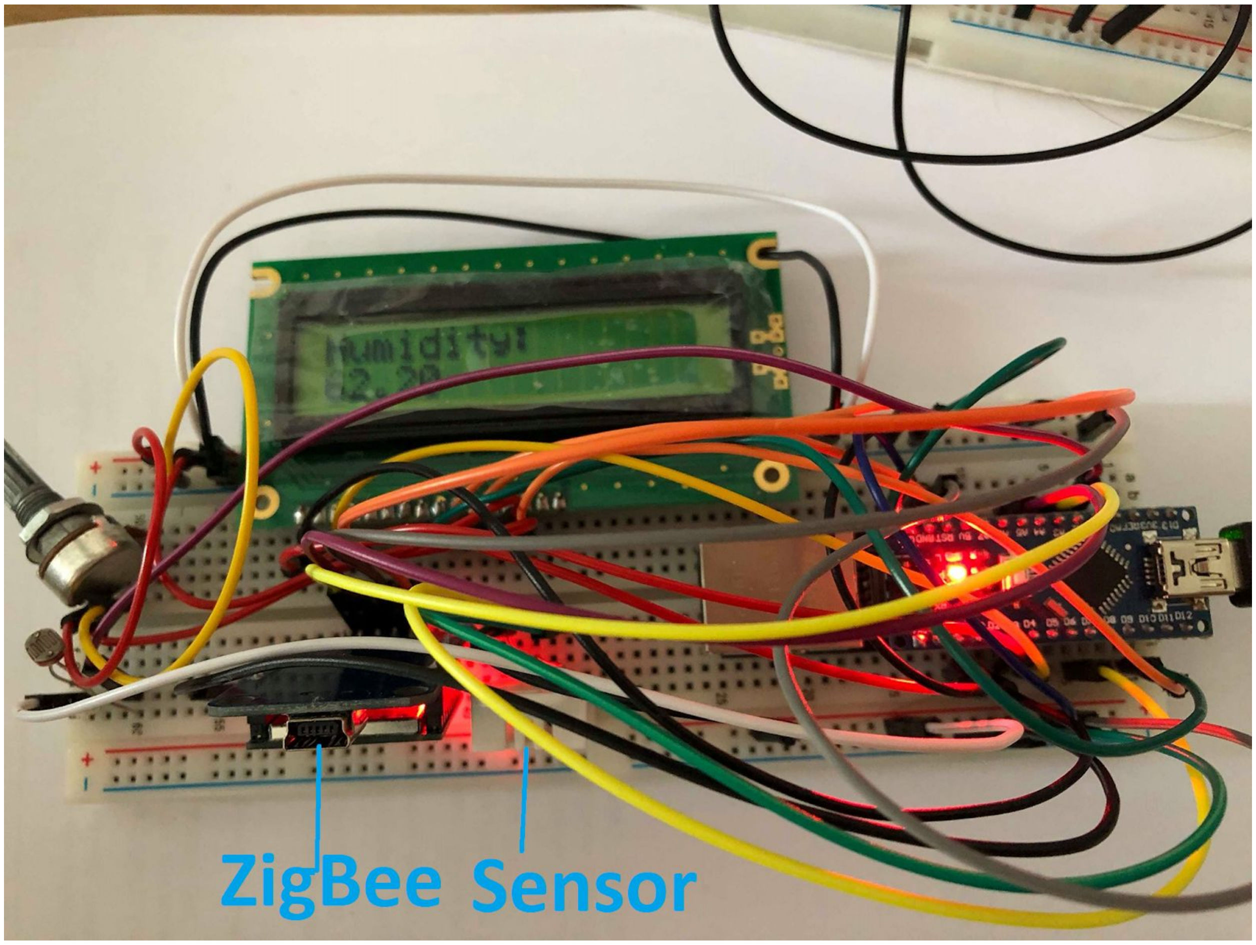
```
String XB_protocol_read(){
  //Read end-device return value
  i=0; // initialize
  while (XBSerial.available()>0 && i<BUFFER_SIZE){
    inData[i] = XBSerial.read();
    i++;
  }
  String inDataString = String(inData);
  if(inDataString) return inDataString;
  else{
    Serial.println("Failed to read Xbee sensor device.");
    return;
  }
}
```

We can also add other sensors by using this switch-case function and by defining the related functions under the loop() one.



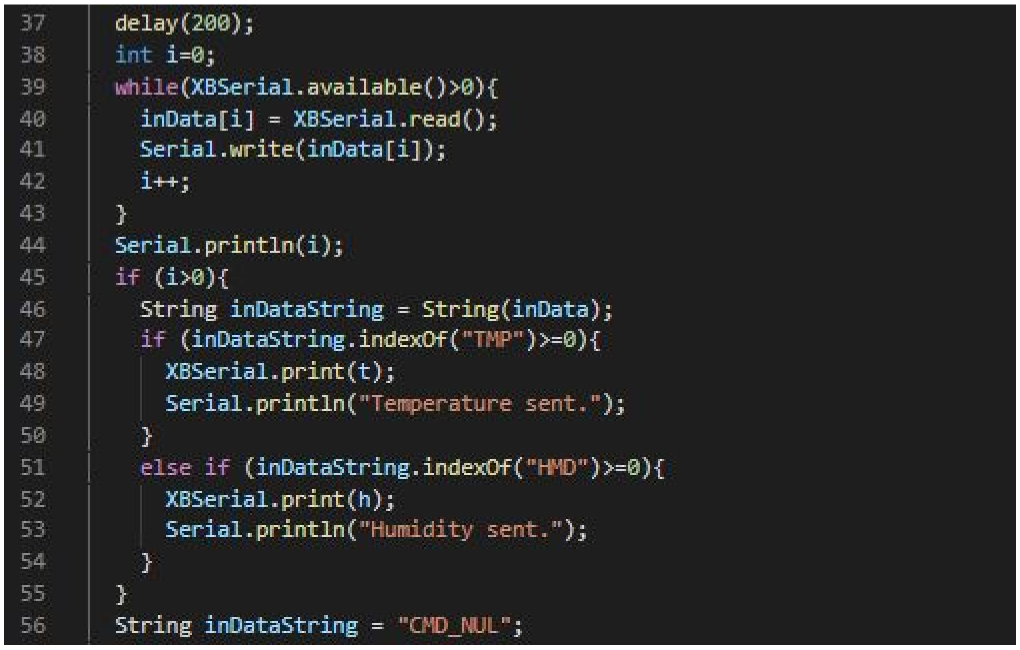
```
void LCD_sensor_display(char sensor_type, String msg, int duration){
  //Display sensor information according to sensor_type
  switch (sensor_type){
    case 0:                   // Temperature
      LCD_temperature(msg);
      break;
    case 1:                   // Humidity
      LCD_humidity(msg);
      break;
    // You can add another case for other sensors, you will need to define the LCD functions the same way
    // we did it with temperature and humidity.
  }
  delay(duration);
}
```

3.2- End-Device



End devices are the sensor controllers, they send feedback to coordinators informing them of sensor

ZigBee	LCD	Sensor
RX A3 TX A4	RS 12 EN 11 D4 5 D5 4 D6 3 D7 2	DHTPIN 8

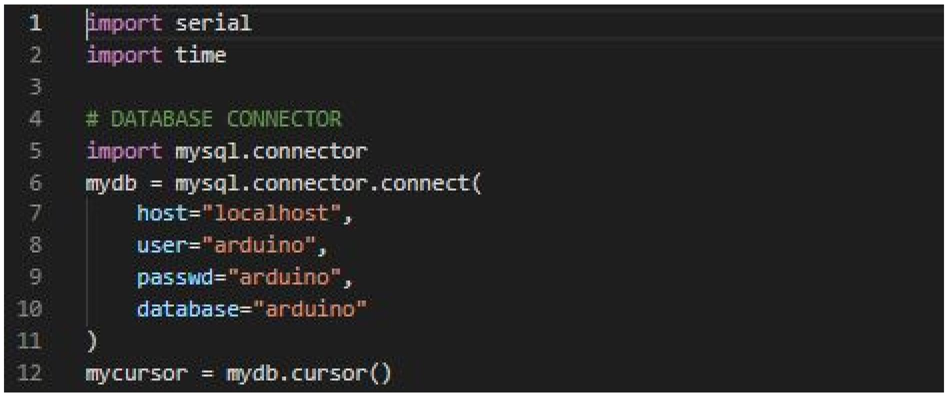


```
  delay(200);
  int i=0;
  while(XBSerial.available()>0){
    inData[i] = XBSerial.read();
    Serial.write(inData[i]);
    i++;
  }
  Serial.println(i);
  if (i>0){
    String inDataString = String(inData);
    if (inDataString.indexOf("TMP")>=0){
      XBSerial.print(t);
      Serial.println("Temperature sent.");
    }
    else if (inDataString.indexOf("HMD")>=0){
      XBSerial.print(h);
      Serial.println("Humidity sent.");
    }
  }
  String inDataString = "CMD_NUL";
```

This code allows to get the instruction from coordinator through ZigBee communication and to reply back with a message.

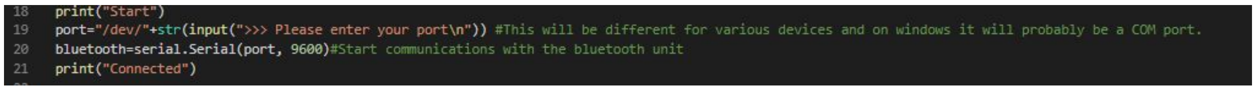
3.3- Server/User

For the server part, we will use a python script, that can be turned into a service in the future. We will need to import **mysql.connector** library to connect the python script to the mysql API. We also need to import **pySerial** to get the bluetooth communication in a specific port.



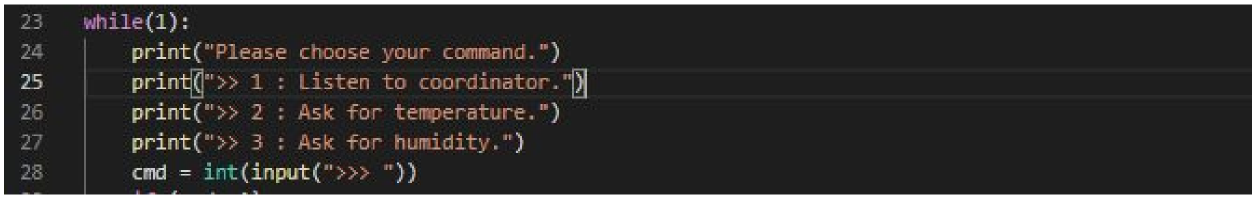
```
import serial
import time

# DATABASE CONNECTOR
import mysql.connector
mydb = mysql.connector.connect(
    host="localhost",
    user="arduino",
    passwd="arduino",
    database="arduino"
)
mycursor = mydb.cursor()
```

First, we define our mysql settings, then we define the port we chose previously for the bluetooth communication while connecting it.



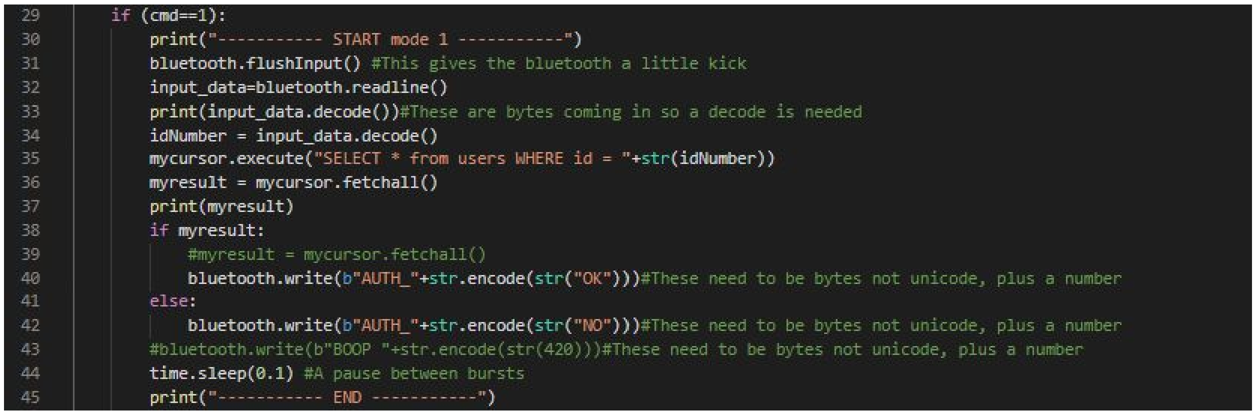
```python
print("Start")
port="/dev/"+str(input(">>> Please enter your port\n")) #This will be different for various devices and on windows it will probably be a COM port.
bluetooth=serial.Serial(port, 9600)#Start communications with the bluetooth unit
print("Connected")
```

The server part is not fully developed yet, so as a testing code, we can define it as below:



```python
while(1):
    print("Please choose your command.")
    print(">> 1 : Listen to coordinator.")
    print(">> 2 : Ask for temperature.")
    print(">> 3 : Ask for humidity.")
    cmd = int(input(">>> "))
```

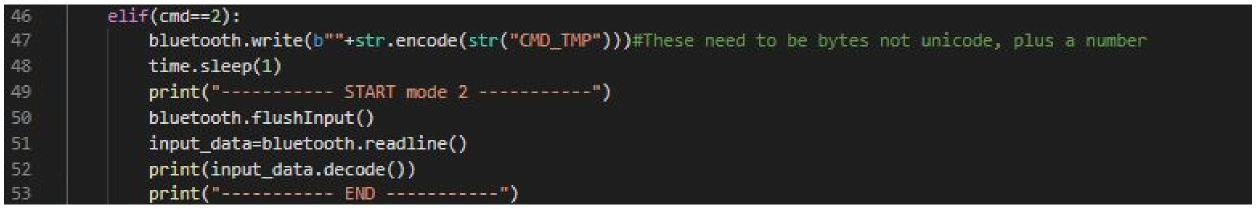
We start an infinite loop to get the command we want.



```python
    if (cmd==1):
        print("----------- START mode 1 -----------")
        bluetooth.flushInput() #This gives the bluetooth a little kick
        input_data=bluetooth.readline()
        print(input_data.decode())#These are bytes coming in so a decode is needed
        idNumber = input_data.decode()
        mycursor.execute("SELECT * from users WHERE id = "+str(idNumber))
        myresult = mycursor.fetchall()
        print(myresult)
        if myresult:
            #myresult = mycursor.fetchall()
            bluetooth.write(b"AUTH_"+str.encode(str("OK")))#These need to be bytes not unicode, plus a number
        else:
            bluetooth.write(b"AUTH_"+str.encode(str("NO")))#These need to be bytes not unicode, plus a number
        #bluetooth.write(b"BOOP "+str.encode(str(420)))#These need to be bytes not unicode, plus a number
        time.sleep(0.1) #A pause between bursts
        print("----------- END -----------")
```

For the authentication, we put the script on listening mode, and we wait until it gets an UID. Then, it will check the UID in the mysql database. The feedback comes after that.

For the sensor informations, we just need to send a bluetooth message.



```python
    elif(cmd==2):
        bluetooth.write(b""+str.encode(str("CMD_TMP")))#These need to be bytes not unicode, plus a number
        time.sleep(1)
        print("----------- START mode 2 -----------")
        bluetooth.flushInput()
        input_data=bluetooth.readline()
        print(input_data.decode())
        print("----------- END -----------")
```

And everything should be working just fine, software-wise.

4- CONCLUSION

As a project, the smart house monitoring system comes as a good way to explore the hardware issues and specifications in the IT domain. It introduced me to the physical part it, it had me think differently considering hardware constraints. This can be worked on by adding features for optimization and testing.

4.1- References

[1] Reverse-Engineering a Cryptographic RFID Tag, 2007

[2] Ciphertext-only Cryptanalysis on Hardened Mifare Classic Cards, 2008

[3] AES Proposal: Rijndael, Joan Daemen, Vincent Rijmen, 2000

[4] docs-05-3474-20-0csg-zigbee-specification