UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA

DIMES

COMPUTER ENGINEERING FOR THE INTERNET OF THINGS

Project Report

Smart Lock RFID Access Control with ATMega328P

for the exam

Low Level and Embedded System Programming

Professor

Prof. Angelo Furfaro

Prof. Carmelo Felicetti

Student

Mario Pingitore - 225638

November 2024

Academic Year 2023-2024

Summary

1. Introduction

The **Internet of Things (IOT)**, even if largely applied nowadays in a lot of scenarios, lacks a fixed definition and, in fact, we can try to describe this technology based on the different way the stakeholders approach it. If we think about **Things**, we of course focus our attention on devices and their "smart" behavior, so they can for example have sensors and actuators to perceive and react to the environment in which they are deployed, they can have process ability to make computation and can have some sort of tags (like RFID), to be identifiable in a larger infrastructure. If we think about the **Internet**, we focus instead on the network characteristics that connect these tiny, low power devices.
However, in more modern application of the technology, the concept of IoT is expanded, so that almost everything can be seen as "Things", for example people, physical and digital objects and representation of objects; the interconnection between these things has to be seen only as a way to create a framework in which everything is connected and works together to build a large ecosystem.
Having such large domain, the IoT sure has a lot of application scenarios, like Industry 4.0, Medical and Healthcare, Quality control, Logistic and supply chain optimization, House Automation, Smart Cities, Safety and Security.
Nowadays, **RFID Smart Locks** are widely used and seen, and have progressively replaced traditional key with RFID **tags** for access and identification, thus implementing a more advanced security system, integrated into the IoT ecosystem. In fact, smartphones and other devices can be linked to remotely operate and handle these locks, that can provide various functions, like activity logging temporary permissions granting and access control.

1.1 Purpose of the project

In this report a simple IoT implementation will be exploited, an Arduino UNO board powered by an ATMega328P microcontroller, connected to various sensors, to simulate a Smart Lock provided with an RFID access control.
The report is structured as follows:

- Chapter 2: the hardware components of architecture such as board, sensors and so on, are presented and discussed.
- Chapter 3: the software technologies and solutions utilized for the logic of the project.
- Chapter 4: the system is presented and discussed in its entirety, from the code used in the device and the implementation of the various sensors.
- Chapter 5: a photographic documentation of the prototype, also while running
- Chapter 6: a collection of the results obtained, and the problems encountered, but also ideas and suggestions for future improvements.

2. Hardware

In this chapter all the hardware components utilized in the building of the architecture will be presented. The main protagonist is obviously the Arduino UNO board, typically used for experimenting with IoT implementations. The board is equipped with an ATMega328P microcontroller, which will be described in further detail in this chapter, as well as an SG90 servo Motor, a 16x02 LCD display, an HC-SR04 ultrasonic sensor and, in the end, an RFID RC522 development kit.

2.1 Board

2.1.1 Arduino UNO R3 (ELEGOO)

The Board utilized is a dupe version of the Arduino UNO R3 and such as that, it is a microcontroller board based on the ATmega328P chip (that can be easily replaced, as it is not soldered on the board). It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header and a reset button. It contains everything needed to support the microcontroller; it just needs to be connected to a computer with a USB cable or power it with an AC-to-DC adapter or battery to be fully functional. It features 1kb of EEPROM, a memory which is not erased when powered off. The board can also be connected and integrated with a lot of sensors or modules, to design almost every kind of amateur, and maybe semi-professional, project.

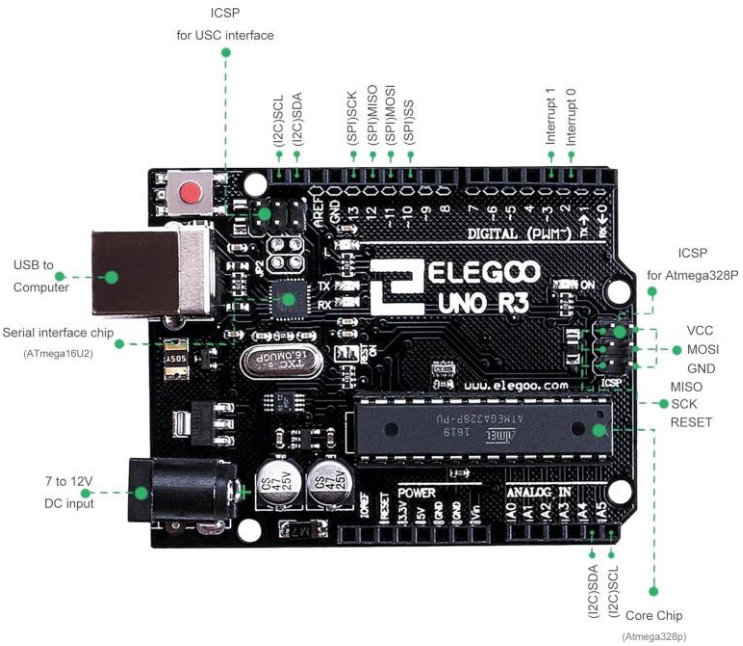


Figure 1:The Elegoo Uno R3

2.1.2 ATMega328P

The ATmega328P is an 8-bit AVR microcontroller based on a RISC architecture, that forms the core of the Arduino UNO board. It usually operates at 16 MHz and is equipped with 32 KB of flash memory for program storage, 2 KB of SRAM, and 1 KB of EEPROM. The microcontroller provides 23 general-purpose I/O lines and supports various communication protocols like UART, SPI, and I2C. Also, it includes three built-in timers (two 8-bit and one 16-bit), a 6-channel 10-bit ADC and PWM capabilities on 6 channels.

The ATmega328P can operate at voltages between 1.8V and 5.5V and its power consumption can be as low as 0.1 µA in power-down mode, making it suitable for battery-operated projects and for low power devices that are common in the IoT scenario. On the Arduino UNO, the ATmega328P comes pre-programmed with a bootloader, facilitating easy programming through the Arduino IDE.

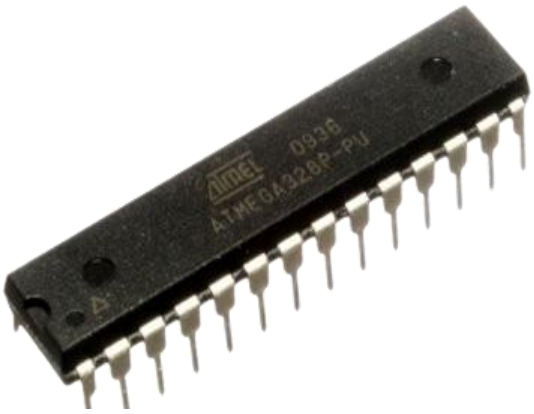


Figure 2: The ATMega328p microcontroller.

2.2 Sensors and modules

2.2.1 SG90 Micro Servo motor

The SG90 is a compact, low-cost digital servo motor widely used in hobbyist electronics and robotics projects. It uses a small DC motor, gearbox, and control circuit to provide precise position control. The servo can be easily controlled, as happen in this project, using Pulse Width Modulation (PWM) signals from most microcontrollers. Its main advantage is its simplicity and affordability, making it ideal for beginners and small-scale projects. The primary limitation is its relatively low torque and precision compared to more advanced servos: in fact, it is less powerful and less accurate, but it's significantly smaller and less expensive. Here are the technical details of the motor:

- Operating voltage: 4.8V to 6V (typically powered by 5V)
- Maximum current draw: Approximately 220mA during operation
- Torque: About 1.8 kg*cm at 4.8V
- Rotation speed: Approximately 0.1 sec/60° at no load
- Rotation range: About 180 degrees total (90 in each direction)
- Control signal: PWM, 50Hz (20ms period)
- Size: 22.2mm x 11.8mm x 31mm
- Weight: Approximately 9 grams

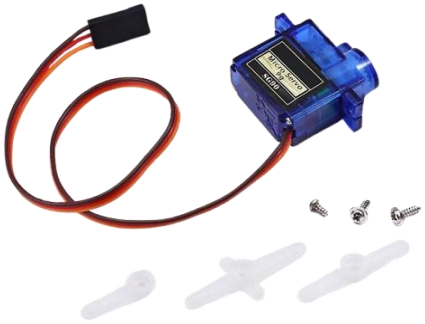


Figure 3: The SG90 Micro Servo motor.

The SG90 is compatible with most microcontroller platforms, including Arduino and Raspberry Pi. It can also be attached to an external power source in case the Arduino is not capable of supplying the needed voltage due to the large number of sensors connected.

2.2.2 LCD Display

The display utilized in this project is a 16x2 LCD display compatible with the Hitachi HD44780 driver. The LCDs have a parallel interface, meaning that the microcontroller has to manipulate several interface pins at once to control the display. The interface consists of the following pins:

- A **register select (RS)** pin that controls where, in the LCD's memory, data is written. It's possible to select either the data register, which handles what goes on the screen, or an instruction register, which is where the LCD's controller looks for instructions.
- A **Read/Write (R/W)** pin that selects reading mode or writing mode.
- An **Enable pin** that enables writing to the registers.
- **8 data pins (D0 -D7).** The states of these pins (high or low) are the bits that are written to a register when writing, or the values read when reading.

Figure 4: The 16x2 LCD display

There's also a display contrast pin (Vo), power supply pins (+5V and GND) and LED Backlight (Bklt+ and BKlt-) pins that we can use to power the LCD, control the display contrast, and turn on and off the LED backlight, respectively.
To display correctly the data on the display, the data must be put into the data registers, while the instructions in the instruction register.

2.2.3 HC-SR04 Ultrasonic Sensor

The HC-SR04 is a popular, low-cost ultrasonic ranging module that provides non-contact distance measurement functionality. It uses ultrasonic transmitters and receivers to measure the distance to an object by emitting high-frequency sound waves and calculating the time it takes for the echo to return.
The sensor offers a wide range of distance detection, from 2cm to 400cm (4m), and it can calculate a difference in space position with an accuracy of about 3mm. One of the main advantages of this sensor is its quick response time, allowing for multiple measurements per second, even if the accuracy of the sensor can be affected by the size and surface characteristics of the target object.
Here are the technical details of the HC-SR04 sensor:

- 5V DC power supply
- 15mA working current
- Measuring range: 2cm to 400cm (4m)
- Accuracy: up to 3mm
- Measuring angle: 15 degrees
- 40kHz operating frequency
- Maximum sampling rate: Recommended >60ms measurement cycle (about 16 Hz)
- Trigger input signal: 10µs TTL pulse
- Echo output signal: TTL level signal, proportional to the measured range
- Dimensions: 45mm x 20mm x 15mm

Figure 5: The HC-SR04 Ultrasonic sensor.

To use the HC-SR04, a short 10µs pulse is needed to be send to the **trigger** input to initiate ranging. The module then sends out an 8-cycle burst of ultrasound at 40 kHz and raises its **echo** pin. The width of the echo pulse is proportional to the distance of the detected object. Compared to other distance sensors, the HC-SR04 offers a good balance of range, accuracy, and cost.
To calculate the range, the following formula can be used:

$$distance = (echo\ pulse\ duration\ in\ \mu s) / 58\ (for\ centimeters, 148\ for\ inches)$$

2.2.4 RFID RC522 Development kit

The RC522 is a highly integrated reader/writer module for contactless communication based on NXP's MFRC522 IC. The module is designed for reading and writing RFID tags and cards that comply with ISO/IEC 14443A/MIFARE and NTAG standards. The development kit contains the RC522 module, a keychain tag and a card tag. As said before, the module is versatile, cost-effective, and can be used in various applications such as access control systems, like in this case, but also for attendance systems or/and payment systems.
The module's internal transmitter can drive a reader/writer antenna to communicate with ISO/IEC 14443A cards and transponders without additional active circuitry. Its receiver module provides robust and efficient implementation for demodulating and decoding signals from compatible cards and transponders. One of the key advantages of this module is its support for high-speed communication and multiple interface options.
Here are the technical details of the RC522 RFID module:

- Operating voltage: 2.5V to 3.3V DC
- Operating current: 13-26mA
- Standby current: 10-13mA
- Operating frequency: 13.56MHz
- Supported protocols: ISO/IEC 14443A, MIFARE, NTAG
- Communication speed: Up to 848 Kbps
- Typical operating distance: Up to 50mm (depending on antenna size and tuning)
- Interface options:
    - SPI (up to 10Mbit/s)
    - I2C (up to 400 kBd in Fast mode, up to 3400 kBd in High-speed mode)
    - UART (up to 1228.8 kBd, voltage levels dependent on pin voltage supply)

Figure 6: The RC522 RFID Development Kit.

As said before, the RC522 module operates with cards and tags in the 13.56MHz frequency range, so other RFID systems that, for example, work at 125KHz, are not compatible. The module's memory is divided into 16 sectors, each protected by two different keys (A and B), allowing for secure data storage and retrieval.

3. Software utilized for the implementation

3.1 Arduino IDE

The Arduino Integrated Development Environment - or Arduino Software (IDE) - contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus and was used to write the code for both the boards. Programs written using Arduino Software (IDE) are called sketches and saved with the file extension *.ino*. It allows, other than writing the program, to upload it on the board and to visualize the serial output through a serial monitor. It's also possible to utilize many kinds of **Libraries**, to provide extra functionality for use in sketches, e.g. working with hardware or manipulating data; to utilize them it is just necessary to insert one or more **#include** statement on top of the sketch, one for each library. In this specific project, I have used the legacy libraries of the sensors utilized such as the **LiquidCrystal Library**, to create a first implementation of the code, to assess the feasibility of the project and to make sure that all the sensors attached to the board could work well together.

3.2 Visual Studio Code and Platform.IO

Visual Studio Code (VS Code) is a free, open-source, and highly versatile code editor developed by Microsoft. First released in 2015, its lightweight yet powerful nature, combined with a vast ecosystem of extensions, makes it an ideal choice for both beginners and experienced developers. The integrated extension marketplace allows users to install, and manage extensions that add new functionality, language support, and integrations with other tools and services, like in this case with the PlatformIO extension. The latter is a comprehensive solution for embedded systems and Internet of Things (IoT) development. Created by Ivan Kravets in 2014, Platform IO has evolved into a powerful, professional-grade tool that simplifies the complexities often associated with embedded development. PlatformIO can also be installed as a standalone, multi-platform IDE, which provides support for over 1,000 development boards and more than 30 development frameworks, covering a wide range of microcontrollers and architectures. Of course, the extension has also access to all the libraries available for the different boards and architectures and provides the opportunity to work with a project-based model, with a built-in terminal, that also supports static code analysis, unit testing and debugging, leveraging VS Code's built-in debugging. Each PlatformIO project typically includes:

- *platformio.ini*: The main configuration file that defines project-specific settings, including the target board, framework, and build flags.
- *src/:* The directory containing the main source code files for the project.
- *lib/*: A directory for project-specific libraries.
- *test/*: Where unit tests for the project are stored.
- *.pio/*: A directory created by PlatformIO to store build artifacts, downloaded tools, and other temporary files.

PlatformIO abstract away many of the complexities involved in embedded development, by automatically managing tool chains and compilers for different architectures, without the need to manually install and configure these tools. This abstraction extends to the building process, where PlatformIO handles also compiling, building and uploading for different targets.

4. Implementation of the system

Until now, the report showed which were the hardware component of the project and the software applications and tools utilized to build the whole architecture. From now on, the practical realization of the project will be discussed in greater detail, starting by focusing on the aim of the project. Next, the functionalities implemented and the hardware architecture upon which the project is built will be discussed and, of course, which communication protocol were used to create the communication between the sensor modules and the microcontroller. In the end, the report will show an explanation about the code of the project.

4.1 Workflow of the system

Before starting with the workflow of the system, it's better to further clarify what the aim of the assignment is. In this project, an Arduino UNO board connected to an RFID module will scan different tags to regulate access control on a lock. In particular, the Ultrasonic sensor will scan for nearby person and will enable the RFID reading only on the presence of someone in proximity. Upon successful checking of the RFID tag, a servo motor will open a door and, after a timed interval, will close it. The system also discerns between two different kinds of tag:

- **Master tag**: is always the first scanned tag and, upon a successive scanning, will make the system enter in Master Mode.
- **Normal tag**: can only be used to access the smart lock.

The **Master Mode** will allow to add/remove a successive scanned tag from the list of ID that can access the lock: if the scanned tag is not already present in the list, will be added, otherwise will be removed. A normal tag can access the lock only if its ID is registered in the system, otherwise the access will be denied from the system.

The entire process is fully automated once the is activated and connected to a power source.

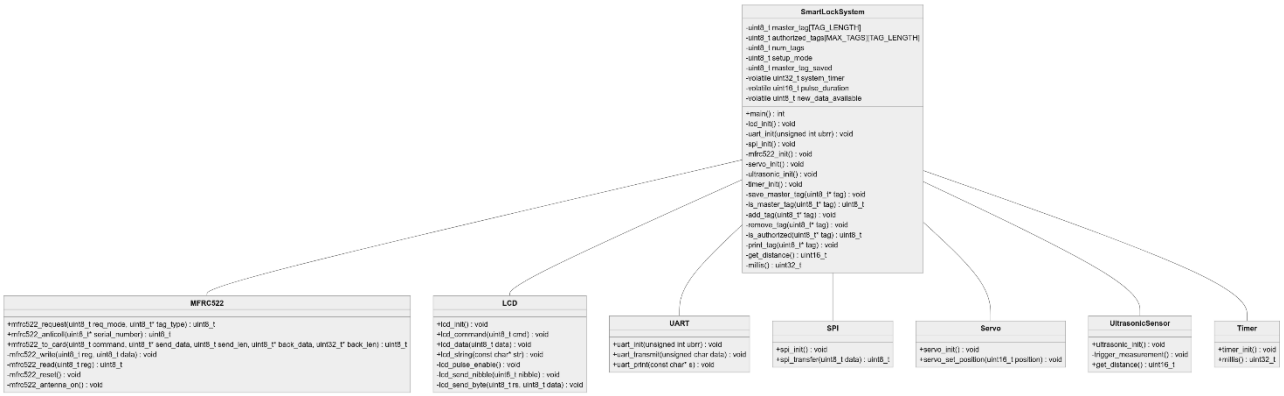


Figure 7: A UML diagram of the project.

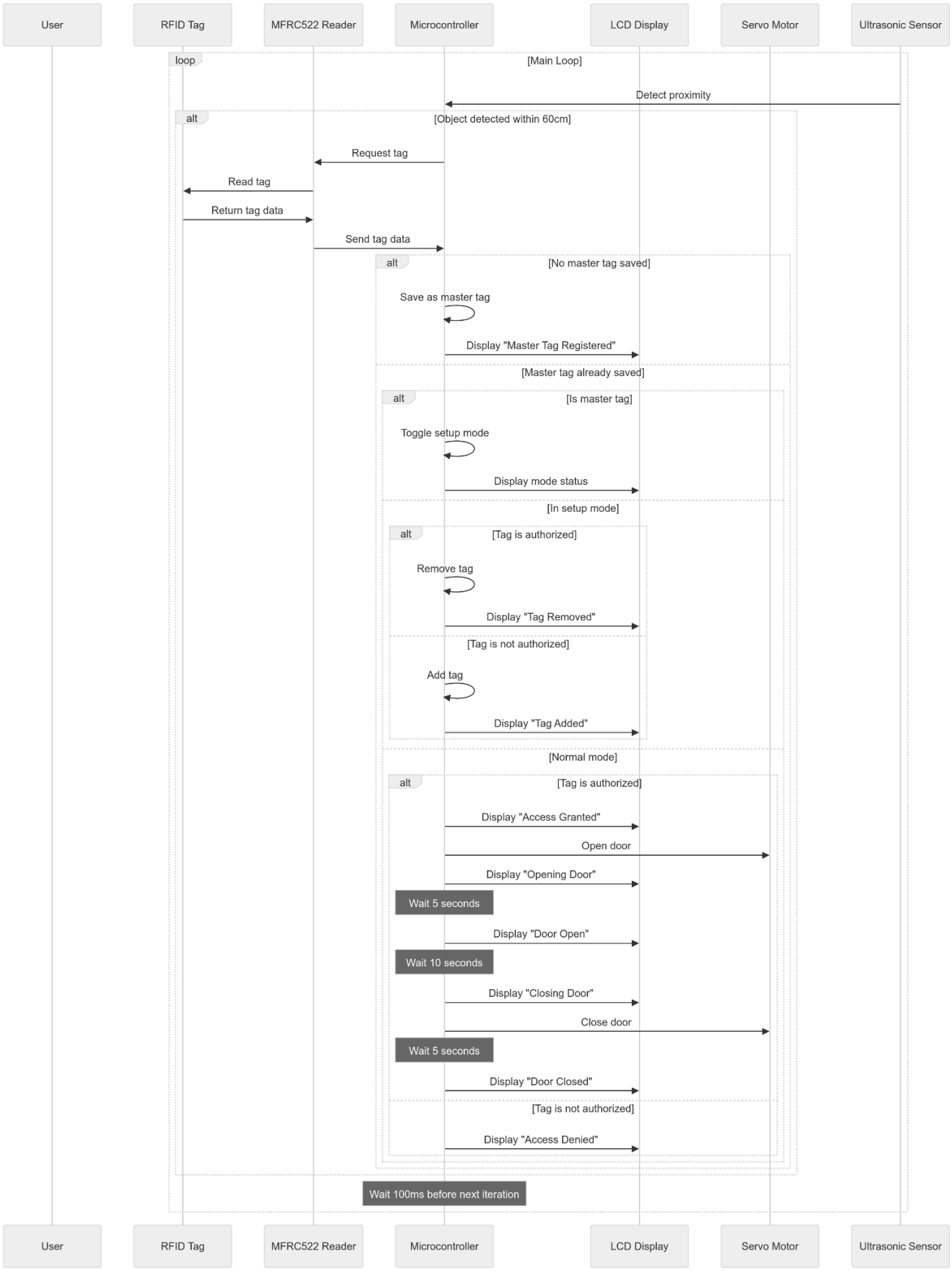


Figure 8: A Sequence diagram of the project.

4.2 Hardware architecture

Regarding the hardware architecture of the system, as mentioned before, we have an Arduino UNO R3 board equipped with an SG90 Servo motor, a 16x02 LCD display, an HC-SR04 Ultrasonic sensor and, in the end, an RFID RC522 module. Beneath, the schematics of the entire architecture.

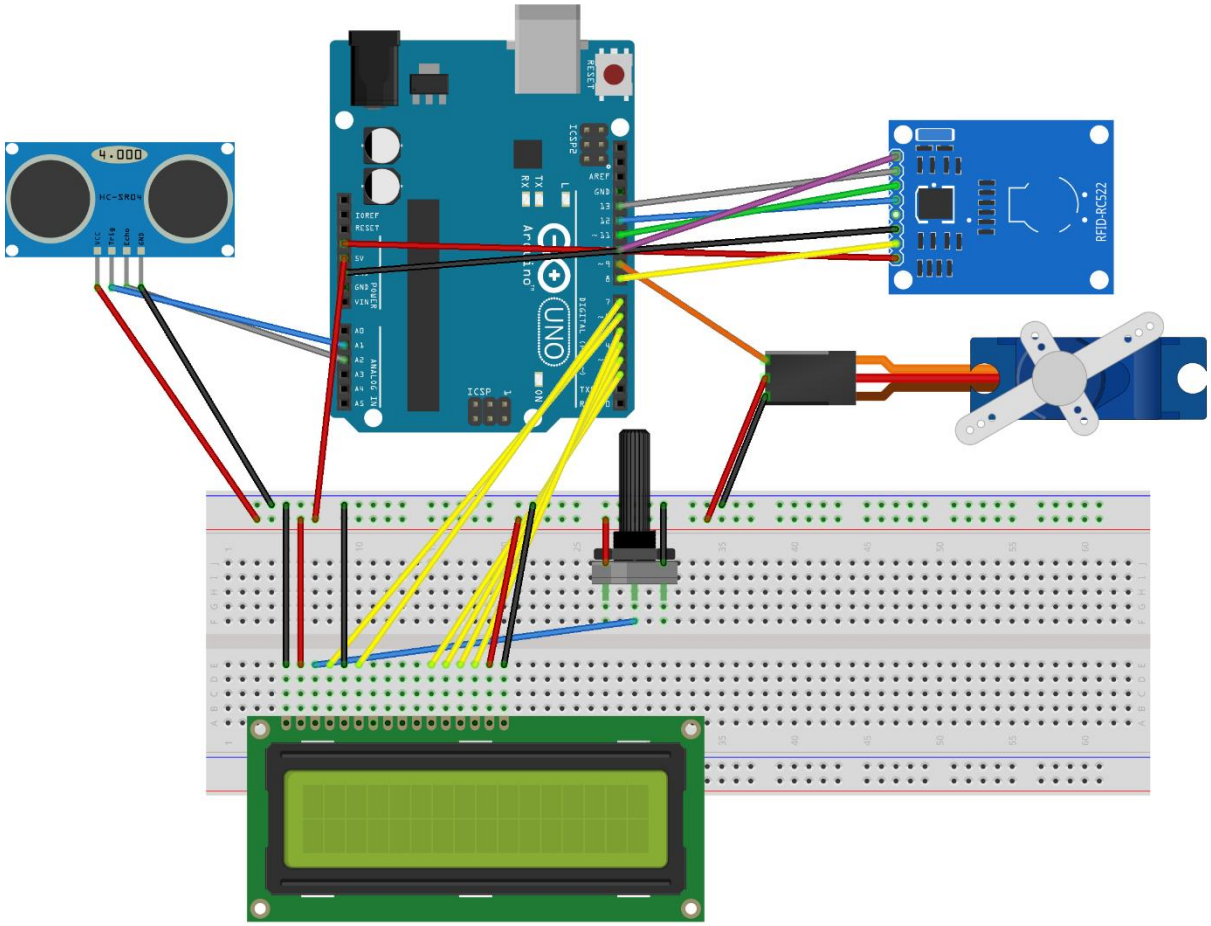


Figure 9: The schematics of the project.

4.3 Communication protocol

4.3.1 UART protocol

UART (Universal Asynchronous Receiver-Transmitter) is a hardware communication protocol that works is full-duplex mode, used for asynchronous serial communication. Unlike synchronous protocols, UART does not require a shared clock between the sender and the receiver but uses start and stop bits to frame the data. The data is transmitted bit by bit (serially) over a single communication line, and the baud rate (usually 9600 in serial communication) defines the speed of communication, measured in bits per second (bps). In UART communication:

- **TX (Transmit Pin)** sends data from the device to an external module or PC.
- **RX (Receive Pin)** receives incoming data. The baud rate must be configured identically for both devices involved in communication.

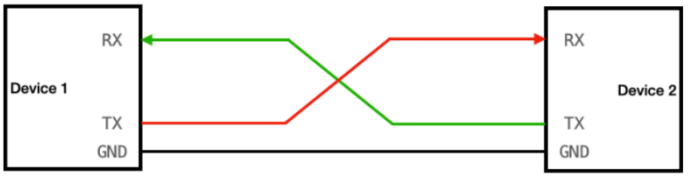


Figure 10: A UART communication representation.

A typical UART frame is composed of:

- **1 Start Bit:** When a UART transmission begins, the transmitter sends a start bit, which signals the receiver that data is about to follow. This bit is always a logical 0 (low voltage).
- **5 to 9 Data Bits:** The number is configurable (typically 8 bits in standard communication).
- **1 Parity Bit (Optional):** An optional parity bit, that can be even, odd or disabled and can be used to detect errors in transmission.
- **1 or 2 Stop Bit:** A stop bit (logical 1) is sent at the end of the data to signal the end of transmission. The stop bit duration can be 1 or 2 bits long, configurable via UART settings.

The UART protocol, in the project, was used only to display information on the Serial monitor, thus a protocol with synchronous capabilities, such as **USART**, wasn't needed.

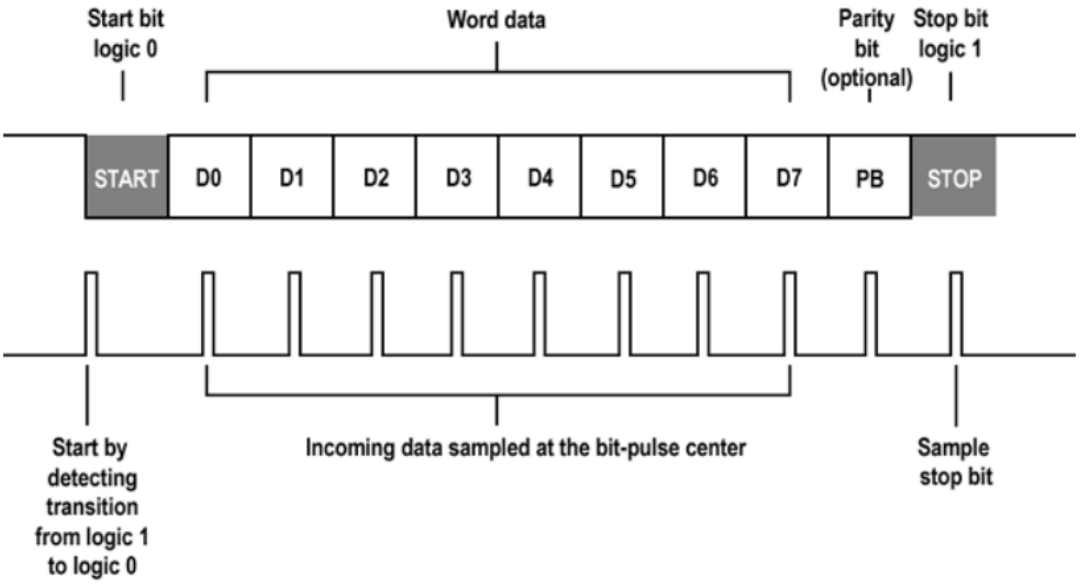


Figure 11: The UART frame structure.

4.3.2 SPI protocol

SPI (Serial Peripheral Interface) is a synchronous, full-duplex communication protocol used for high-speed data transmission between a master device (usually a microcontroller) and one or more slave devices, in this case the **MFRC522 RFID module**. It uses four primary signals:

- **MOSI** (Master Out Slave In): Data sent from the master to the slave.
- **MISO** (Master In Slave Out): Data sent from the slave to the master.
- **SCK** (Serial Clock): A clock signal generated by the master to synchronize data transmission.
- **SS** (Slave Select): Used by the master to select which slave to communicate with.

In the code, the **SS** pin and the **RST** pin (Reset) are defined to identify the slave, which is the RFID reader, and, if it's needed, to reset it.

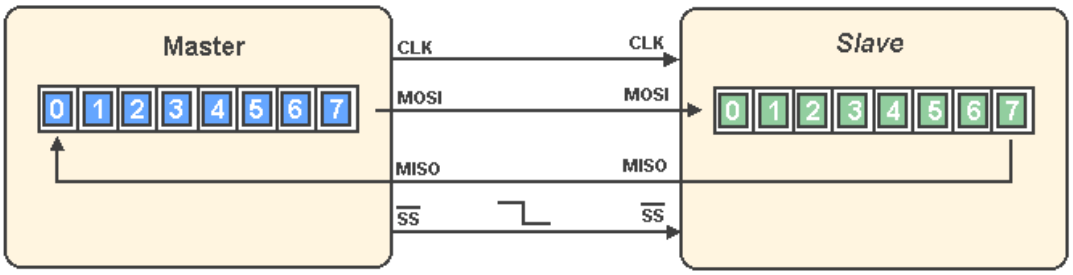
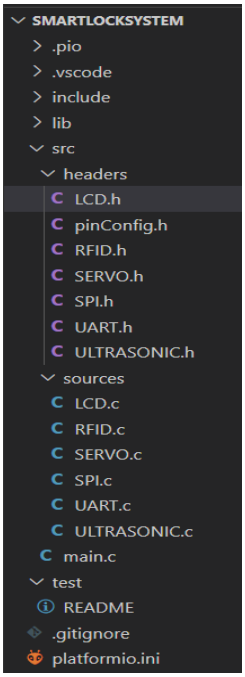


Figure 12: An SPI communication representation.

The various command registers allow the microcontroller to communicate to the module: the *PCD_TRANSMIT* command send data from the master to the RFID reader, while *PCD_RECEIVE* fetch data from the reader. Given the synchronous nature of the protocol, is assured that the data is transmitted and received at the same clock cycle, which is highly efficient for real-time applications. Given the full-duplex capabilities, the data can be transferred in both directions simultaneously: each device has a shift register that shifts out data bit by bit, synchronized with the clock signal. The master controls the clock, and for each clock cycle, one bit of data is transferred on the MOSI and MISO lines. Regarding the speed of communication, it is controlled by the master by adjusting the clock frequency.

4.4 Software code in the boards



In the following subchapters, the code flashed on the board will be analyzed in detail by isolating the behavior and thus the functions of all the modules connected to the microcontroller and, of course, also the *main* file that puts everything together.
As said before, the board has been programmed by using Arduino IDE in a first phase, then with Visual Studio Code. The code is organized in a project-based manner, like described in the previous chapter: as can be seen in the image, it's present an ***headers*** folder in which there are all the declarations of functions, data types, constants used in the project, like for example the *pinConfig.h* file, which contains the pinout of all the modules utilized in the project and also the register utilized, for example, for the RC522 RFID module. In the ***sources*** folder there is the logic of all modules, that are put together in the ***main*** file.
All the code snippets here shown may truncated comments for space reasons, but the latter are of course present in their entirety in the code attached to this report. For the same reason, the *pinConfig.h* file, given its length and its "trivial" content, is not reported.

Figure 13: Code structure of the project.

4.4.1 HC-SR04 Ultrasonic sensor

The sensor utilizes two PIN, defined in the *pinConfig.h* file, that are the **TRIGPIN** and the **ECHO** PIN. The first one is utilized to send a trigger to the sensor, the other one to receive the echo generated from the signal: the duration of this echo pulse is stored in the *pulse_duration* variable. In the *ultrasonic_init* function, the different pins are set as output and input, then **TRIGPIN** is set as LOW and, in the end, the interrupts are initialized with the *sei*() call. The system utilizes the interrupts to handle time: the *ISR* method is called automatically when there is a change on the **ECHO** pin and measure the timing between the rising edge (**ECHO** pin goes high) and the falling edge (**ECHO** pin goes down), utilizing the **TIMER1** of the microcontroller. The *trigger_measurement* function sends a 10us impulse to the trig pin to enact a measurement. Then, in the *get_distance* function, which is the one that will be called from the main, the results from the measurement are retrieved, if there are any, converted to millimeter (the calculation is based on the speed of sound and the round-trip time of the ultrasonic pulse) and visualized (through *UART* in the main).

```c
volatile uint16_t pulse_duration = 0; //memorize echo pulse duration
volatile uint8_t new_data_available = 0; //flag to check if there are new data

ISR(PCINT1_vect) {
    static uint16_t rising_edge, falling_edge;
    if (PINC & (1 << ECHOPIN)) {
        rising_edge = TCNT1;
    } else {
        falling_edge = TCNT1;
        pulse_duration = falling_edge - rising_edge;
        new_data_available = 1;
    }
}

void ultrasonic_init(void) {
    DDRC |= (1 << TRIGPIN); //set TRIGPIN as output
    DDRC &= ~(1 << ECHOPIN); //set ECHOPIN as input
    PORTC &= ~(1 << TRIGPIN); //Ensure TRIGPIN starts LOW

    PCICR |= (1 << PCIE1);
    PCMSK1 |= (1 << PCINT9);

    sei();
}
```

```c
void trigger_measurement(void) {
    PORTC |= (1 << TRIGPIN);
    _delay_us(10);
    PORTC &= ~(1 << TRIGPIN);
}

uint16_t get_distance(void) {
    trigger_measurement();
    _delay_ms(60); //wait for measurement to complete
    if (new_data_available) {
        new_data_available = 0;
        return pulse_duration / 58;  // Convert to mm
    }
    return 0;  // No echo received
}
```

Figure 14: Functions to handle the HC-S04 Ultrasonic sensor.

4.4.2 16x02 LCD Display

The pins utilized by the display are the **LCD_RS**, used to select the registers (0 for commands, 1 for data), the enabling pin **LCD_EN** and **LCD_D4-D7** for the data (in 4-bit mode like in this case). First, the display is initialized by the *lcd_init* function, which configures the necessary pin as output, configure the display for 2 rows, with 5x8 font, enables the display with the auto increment cursor and clear it.

The *lcd_send_nibble* method sets the 4 data pins based on the value of the nibble and then generates an enable impulse on the **LCD_EN** pin through the *lcd_pulse_enable* function, which lets the LCD know that the data on the pins are ready to be read. The *lcd_command* and the *lcd_data* both utilize the *lcd_send_byte* function by setting **LCD_RS**=0/1, which sends a complete byte (in two nibbles) to respectively send, as the name implies, commands and data to the LCD. It is important to note the use of delays: the shorter ones (1-50us) are used during basic operations, while the longer ones (1-5ms) are used during the initialization and after the commands.

```c
void lcd_pulse_enable(void) {
    PORTD |= (1 << LCD_EN);
    _delay_us(1);
    PORTD &= ~(1 << LCD_EN);
    _delay_us(50);
}

void lcd_send_nibble(uint8_t nibble) {
    PORTD &= ~((1 << LCD_D4) | (1 << LCD_D5)
    | (1 << LCD_D6) | (1 << LCD_D7));
    if (nibble & 0x01) PORTD |= (1 << LCD_D4);
    if (nibble & 0x02) PORTD |= (1 << LCD_D5);
    if (nibble & 0x04) PORTD |= (1 << LCD_D6);
    if (nibble & 0x08) PORTD |= (1 << LCD_D7);
    lcd_pulse_enable();
}

void lcd_send_byte(uint8_t rs, uint8_t data) {
    if (rs)
        PORTD |= (1 << LCD_RS);
    else
        PORTD &= ~(1 << LCD_RS);

    _delay_us(1);

    lcd_send_nibble(data >> 4);
    lcd_send_nibble(data & 0x0F);

    _delay_us(50);
}
```

```c
void lcd_init(void) {
    DDRD |= (1 << LCD_RS) | (1 << LCD_EN) | (1 << LCD_D4) |
    (1 << LCD_D5) | (1 << LCD_D6) | (1 << LCD_D7); //set pins as output

    _delay_ms(50); //wait for lcd initialization

    lcd_send_nibble(0x03);
    _delay_ms(5);
    lcd_send_nibble(0x03);
    _delay_ms(1);
    lcd_send_nibble(0x03);
    _delay_ms(1);
    lcd_send_nibble(0x02);
    _delay_ms(1);

    lcd_command(0x28);  // 4-bit mode, 2 lines, 5x8 font
    lcd_command(0x0C);  // Display on, cursor off, blink off
    lcd_command(0x06);  // Increment cursor, no shift
    lcd_command(0x01);  // Clear display
    _delay_ms(2);
}
```

```c
void lcd_command(uint8_t cmd) {
    lcd_send_byte(0, cmd);
    _delay_ms(2);
}

void lcd_data(uint8_t data) {
    lcd_send_byte(1, data);
}

void lcd_string(const char *str) {
    while (*str) {
        lcd_data(*str);
        str++;
        _delay_ms(1);
    }
}
```

Figure 15: Functions to handle the LCD display.

4.4.3 SG90 Micro Servo motor

The Servo motor is connected with the **PB1** (pin 9) of the board, because is the one that use the PWM signal. In the *servo_init* function, pin 9 is set as output and, while also setting the Fast PWM (mode 14), which is configured by setting **WGM11**, **WGM12** and **WGM13** to 1, with an 8 prescaler. The **ICR1** register is used to define the period of the PWM by using the following formula:

$$ICR1 = (16MHz \,/\, (8 * 50Hz)) \; - \; 1 \; = \; 39999$$
$$PWM\ period = ((39999 \; + \; 1) * \; 8) / \; 16MHz \; = \; 20ms$$

Where 50 Hz is the standard frequency of this kind of servo motor. The *servo_set_position* is the function called from the main, that will make the servo motor move based on the duty cycle set in the *OCR1A* register. The pulse width used to handle the movement is similar for this kind of motor, even if they can slightly be different based on the different manufacturers.

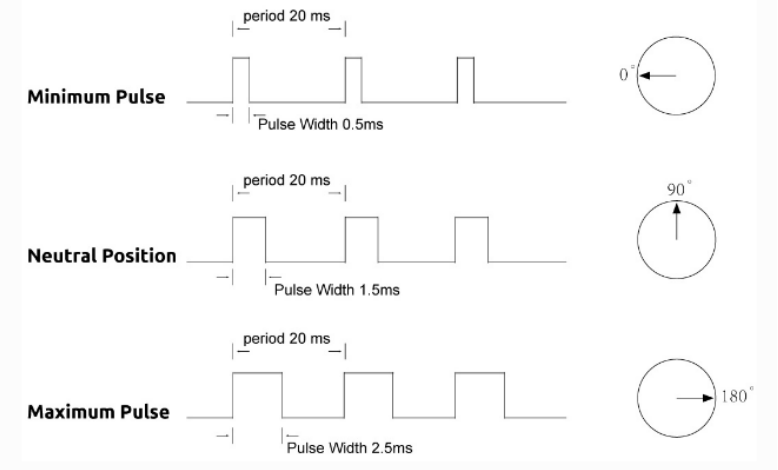


Figure 16: Pulse configuration to handle the servo motor.

Specifically, the values and formulas utilized for **OCR1A** are:

- For **180** degrees: $(pulse\ width \,/\, period) * ICR1 \; \rightarrow (2.4\ ms \,/\, 20\ ms) * 39999 \; = \; \mathbf{4799}$
- For **0** degrees: $(pulse\ width \,/\, period) * ICR1 \; \rightarrow (0.75\ ms \,/\, 20\ ms) * 39999 \; = \; \mathbf{1499}$

```
void servo_init(void) {
    DDRB |= 1 << SERVO_PIN;  // Set pin 9 as output
    TCCR1A |= (1 << WGM11) | (1 << COM1A1);
    TCCR1B |= (1 << WGM12) | (1 << WGM13) | (1 << CS11);
    ICR1 = 39999;  // 20ms period (50Hz)
}

void servo_set_position(uint16_t position) {
    OCR1A = position;
}
```

Figure 17: Functions to handle the SG90 Servo motor.

4.4.4 *UART* and *SPI* communication protocols

The UART communication protocol is utilized in all the code to enable the writing of information about the status of the project while it's running on the Serial Monitor, useful also for debug purposes. The communication is handled by only three functions: *init*, *transmit* and *print*. In *uart_init*, the UART registers are configured: **UBRROH** and **L** set the baud rate, **UCSR0B** enables the transmission or the response and **UCSR0C** configure the frame format (8 bit, with 1 stop bit). In *uart_transmit* a single byte is sent by writing it in the **UDR0** register, while waiting for the transmission buffer to be empty; in *uart_print*, instead, a string is sent one character at a time, by

calling, internally, uart_*transmit*.
The SPI communication protocol, instead, is used exclusively for the communication between the microcontroller and the RFID module. In *spi_init*, the SPI pins and registers are configured: it sets **MOSI**, **SCK** and **SS** as output and **MISO** as input and then it configures the microcontroller as SPI master, by also setting the SPI speed/frequency to the lowest possible, to achieve better stability. In *spi_transfer*, instead, a byte of data is sent by putting it in the **SPDR** register and, while waiting the finishing of the transmission by checking the **SPIF** bit of the **SPSR** register, the byte received at the same time is returned.

```c
void uart_init(uint16_t ubrr) {
    UBRR0H = (uint8_t)(ubrr >> 8);
    UBRR0L = (uint8_t)ubrr;
    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
    UCSR0C = (3 << UCSZ00);
}

void uart_transmit(uint8_t data) {
    while (!(UCSR0A & (1 << UDRE0)));
    UDR0 = data;
}

void uart_print(const char *s) {
    while (*s) {
        uart_transmit((uint8_t)*s);
        s++;
    }
}
```

```c
void spi_init(void) {
    DDRB |= (1<<PB3) | (1<<PB5) | (1<<SS_PIN);
    DDRB &= ~(1<<PB4);
    SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR1) | (1<<SPR0);
    SPSR &= ~(1<<SPI2X);
}

uint8_t spi_transfer(uint8_t data) {
    SPDR = data;
    while(!(SPSR & (1<<SPIF)));
    return SPDR;
}
```

Figure 18: Functions to handle the UART and SPI communication protocols.

4.4.5 RC522 RFID Module

The module is based on the MFRC522 Integrated Circuit: the pins that codify the specific functions of the RFID module are **RST_PIN**, used to reset the module, and the **SS_PIN** (Slave Select) used for the SPI communication: in fact, the system utilize the before mentioned UART and SPI communication protocols, the first to debug and the other to interact with the RFID module. The *mfrc522_read* and *write* are auxiliary functions, used internally, to read and write data on the registers of the module, while the *mfrc522_antenna_on* and *reset*, as the name implies, are used to enable the antenna of the module and to reset it by writing on the specific registers that regulate these behaviors. In the *mfrc522_init* function, the reset pin is configured, and the internal timer is initialized: **TModeReg** is set so that the timer is enabled and automatic, **TPrescalerReg** set the prescaler, in this case 64 and **TReloadRegL** and **H** set the charging value of the timer. The *mfr522_request* is the first step in the retrieving and identification process of the tags in the "working area" of the RFID reader. The *mfrc522_to_card* is the function that handles the low-level communication between the RFID module and the tags. First thing first, the variable for the communication and for the interrupts are initialized: specifically, the interrupts are configured based on the authentication or the transmission command (**PCD_AUTENTH** and **PCD_TRANSCEIVE**). The data are transmitted through a FIFO buffer, which is first cleared and then all the data are flushed in it. Then, the above-mentioned command is sent: if it's **PCD_TRANCEIVE**, then also the **BitFramingReg** is configured for the sending. At this point the communication is started: so, while waiting that it finishes or that a timeout elapse, the operation status is read from the interrupt register (**ComIrqReg**). At the end, if the communication was successful, the received data is read from the buffer and the operation status is returned (0 if successful), but the system can also handle other results, like in case of partial bits received. This function is called in the *mfrc522_anticoll*

function, which is also invoked in the main file and it is used to obtain a single UID when more tags could be present in the area covered by the reader. In the function, **BitFramingReg** is configured for the complete transmission of the byte and the command **ANTICALL** is performed with the parameter **0x20**, which implies that the system is looking for all UID. Then the *to_card* method is invoked and then, if the communication is successful and the length is correct (40 bit = 5 byte):

- Calculate the checksum by performing a XOR of the first four byte (UID).
- Compare the calculated checksum with the one retrieved, which is the fifth byte.
- If the two checksums are the same, the anti-collision has been successful and thus 0 is returned (2 in case of error).

```
uint8_t mfrc522_to_card(uint8_t command, uint8_t *send_data,
uint8_t send_len, uint8_t *back_data, uint32_t *back_len) {
    uint8_t status = 2;
    uint8_t irqEn = 0x00;
    uint8_t waitIRq = 0x00;
    uint8_t lastBits;
    uint8_t n;
    uint32_t i;

    switch (command) {
        case PCD_AUTHENT:
            irqEn = 0x12;
            waitIRq = 0x10;
            break;
        case PCD_TRANSCEIVE:
            irqEn = 0x77;
            waitIRq = 0x30;
            break;
        default:
            break;
    }

    n=mfrc522_read(ComIrqReg);
    mfrc522_write(ComIrqReg,n&(~0x80));//clear all interrupt bits
    n=mfrc522_read(FIFOLevelReg);
    mfrc522_write(FIFOLevelReg,n|0x80);//flush FIFO data

    for (i = 0; i < send_len; i++) {
        mfrc522_write(FIFODataReg, send_data[i]);
    }

    mfrc522_write(CommandReg, command);

    if (command == PCD_TRANSCEIVE) {
      n=mfrc522_read(BitFramingReg);
      mfrc522_write(BitFramingReg,n|0x80);
    }
```

```
    i = 2000;
    do {
        n = mfrc522_read(ComIrqReg);
        i--;
    } while ((i != 0) && !(n & 0x01) && !(n & waitIRq));

    mfrc522_write(BitFramingReg, mfrc522_read(BitFramingReg) & (~0x80));

    if (i != 0) {
        if (!(mfrc522_read(ErrorReg) & 0x1B)) {
            status = 0;
            if (n & irqEn & 0x01) {
                status = 1;
            }

            if (command == PCD_TRANSCEIVE) {
                n = mfrc522_read(FIFOLevelReg);
                lastBits = mfrc522_read(ControlReg) & 0x07;
                if (lastBits) {
                    *back_len = (uint32_t)(n-1)*8 + (uint32_t)lastBits;
                } else {
                    *back_len = (uint32_t)n*8;
                }

                if (n == 0) {
                    n = 1;
                }
                if (n > MAX_LEN) {
                    n = MAX_LEN;
                }

                for (i = 0; i < n; i++) {
                    back_data[i] = mfrc522_read(FIFODataReg);
                }
            }
        } else {
            status = 2;
        }
    }
```

Figure 19: The mfrc522_to_card function.

```c
void mfrc522_init(void) {
    DDRB |= (1<<RST_PIN);
    PORTB &= ~(1<<RST_PIN);
    _delay_ms(50);
    PORTB |= (1<<RST_PIN);
    _delay_ms(50);

    mfrc522_reset();

    mfrc522_write(TModeReg, 0x8D);
    mfrc522_write(TPrescalerReg, 0x3E);
    mfrc522_write(TReloadRegL, 0x30);
    mfrc522_write(TReloadRegH, 0);
    mfrc522_write(TxASKReg, 0x40);
    mfrc522_write(ModeReg, 0x3D);

    mfrc522_antenna_on();
}

uint8_t mfrc522_request(uint8_t req_mode, uint8_t *tag_type) {
    uint8_t status;
    uint32_t backBits;
    mfrc522_write(BitFramingReg, 0x07);
    tag_type[0] = req_mode;
    status = mfrc522_to_card(PCD_TRANSCEIVE, tag_type, 1,
    tag_type, &backBits);
    if ((status != 0) || (backBits != 0x10)) {
        status = 2;
    }
    return status;
}
```

```c
void mfrc522_write(uint8_t reg, uint8_t data) {
    PORTB &= ~(1<<SS_PIN);
    spi_transfer((reg << 1) & 0x7E);
    spi_transfer(data);
    PORTB |= (1<<SS_PIN);
}

uint8_t mfrc522_read(uint8_t reg) {
    uint8_t value;
    PORTB &= ~(1<<SS_PIN);
    spi_transfer(((reg << 1) & 0x7E) | 0x80);
    value = spi_transfer(0x00);
    PORTB |= (1<<SS_PIN);
    return value;
}

void mfrc522_reset(void) {
    mfrc522_write(CommandReg, PCD_RESETPHASE);
}

void mfrc522_antenna_on(void) {
    uint8_t temp = mfrc522_read(TxControlReg);
    if (!(temp & 0x03)) {
        mfrc522_write(TxControlReg, temp | 0x03);
    }
}
```

Figure 21: The mfrc522_init, request, write, read, reset and antenna_on functions.

```c
uint8_t mfrc522_anticoll(uint8_t *serial_number) {
    uint8_t status;
    uint8_t i;
    uint8_t serNumCheck = 0;
    uint32_t len;

    uart_print("Anticoll: Starting\r\n");

    mfrc522_write(BitFramingReg, 0x00);
    serial_number[0] = PICC_ANTICOLL;
    serial_number[1] = 0x20;

    uart_print("Anticoll: Before to_card\r\n");
    status = mfrc522_to_card(PCD_TRANSCEIVE, serial_number, 2, serial_number, &len);

    char buffer[3];
    uart_print("Anticoll: After to_card, status: ");
    byte_to_hex(status, buffer);
    uart_print(buffer);
    uart_print(", len: ");
    // Per unLen, possiamo usare una semplice conversione in decimale
    char len_buffer[6];  // Sufficiente per un uint16_t
    itoa(len, len_buffer, 10);
    uart_print(len_buffer);
    uart_print("\r\n");
```

```c
if (status == 0) {
    uart_print("Anticoll: to_card success\r\n");
    if (len == 40) {
        for (i = 0; i < 4; i++) {
            serNumCheck ^= serial_number[i];
            uart_print("Serial Number[");
            char index_buffer[2];
            itoa(i, index_buffer, 10);
            uart_print(index_buffer);
            uart_print("]: ");
            byte_to_hex(serial_number[i], buffer);
            uart_print(buffer);
            uart_print("\r\n");
        }
        uart_print("Calculated SerNumCheck: ");
        byte_to_hex(serNumCheck, buffer);
        uart_print(buffer);
        uart_print("\r\n");
        uart_print("Received Checksum: ");
        byte_to_hex(serial_number[4], buffer);
        uart_print(buffer);
        uart_print("\r\n");

        if (serNumCheck == serial_number[4]) {
            uart_print("Anticoll: Checksum correct\r\n");
            return 0;
        } else {
            uart_print("Anticoll: Checksum failed\r\n");
            return 2;
        }
    } else {
        uart_print("Anticoll: Incorrect data length\r\n");
        return 2;
    }
} else {
    uart_print("Anticoll: to_card failed\r\n");
}

return status;
```

Figure 20: The mfrc522_anticoll function.

4.4.6 *Main*

The main file is the core of the project and the one that puts the information and the logic of all the attached module together. Apart from the *main* function itself, it contains some auxiliary functions:

- *save_master_tag:* Save the tag received in the **master_tag** variable and set the **master_tag_saved** flag to 1.
- *is_master_tag*: Check if the read tag corresponds to the saved master tag.
- *add_tag*: Add the read tag to the **authorized_tags** array, while showing information about the process both on the LCD display and on the Serial Monitor.
- *remove_tag*: Check if the read tag is already saved in the **authorized_tags** array and, in this case, it removes it from the array.
- *is_authorized*: Check if the read tag is already registered in the **authorized_tags** array.
- *print_tag*: Print the hexadecimal representation of the byte by calling the *byte_to_hex* function.

```
void save_master_tag(uint8_t *tag) {
    memcpy(master_tag, tag, TAG_LENGTH);
    master_tag_saved = 1;  // set the flag when the master tag is saved
    uart_print("Master tag saved\r\n");
}

uint8_t is_master_tag(uint8_t *tag) {
    return memcmp(master_tag, tag, TAG_LENGTH) == 0;
}

void add_tag(uint8_t *tag) {
    if (num_tags < MAX_TAGS) {
        memcpy(authorized_tags[num_tags], tag, TAG_LENGTH);
        num_tags++;
        uart_print("Tag added\r\n");
        lcd_command(0x01);
        lcd_string("Tag Added");
        _delay_ms(2000);  // Show message for 2 seconds
        master_mode = 0;  // Exit from master mode
        uart_print("Exited master mode\r\n");
        lcd_command(0x01);
        lcd_string("Normal Mode");
    } else {
        uart_print("Tag memory full\r\n");
        lcd_command(0x01);
        lcd_string("Memory Full");
        _delay_ms(2000);
    }
}
```

```
void remove_tag(uint8_t *tag) {
    for (uint8_t i = 0; i < num_tags; i++) {
        if (memcmp(authorized_tags[i], tag, TAG_LENGTH) == 0) {
            memcpy(authorized_tags[i], authorized_tags[num_tags - 1],
            TAG_LENGTH);
            num_tags--;
            uart_print("Tag removed\r\n");
            lcd_command(0x01);
            lcd_string("Tag Removed");
            _delay_ms(2000);
            master_mode = 0;
            uart_print("Exited master mode\r\n");
            lcd_command(0x01);
            lcd_string("Normal Mode");
            return;
        }
    }
    uart_print("Tag not found\r\n");
    lcd_command(0x01);
    lcd_string("Tag Not Found");
    _delay_ms(2000);
}

uint8_t is_authorized(uint8_t *tag) {
    for (uint8_t i = 0; i < num_tags; i++) {
        if (memcmp(authorized_tags[i], tag, TAG_LENGTH) == 0) {
            return 1;
        }
    }
    return 0;
}



void print_tag(uint8_t *tag) {
    // Buffer to memorize the hexadecimal rappresentation of a byte
    char buffer[3];
    for (int i = 0; i < TAG_LENGTH; i++) {
        byte_to_hex(tag[i], buffer);
        uart_print(buffer);
        uart_print(" ");
    }
    uart_print("\r\n");
}
```

Figure 22: The "auxiliary" functions of the main file.

In the file, also the Timer usage is handled:

- *ISR* (Timer Interrupt Service Routine): Runs automatically in background to increment the **system_timer** variable, which is used through the code to manage various time-dependent operations.
- *millis*: Disable interrupts, retrieve the **system_timer** value and the re-enable them.

- *timer_init:* The **Timer0** is configured, first by setting the Waveform Generation Mode (**WGM**) of **TCCR0A** to Clear Timer on Compare Match mode (**CTC**), then by using a 64 prescaler (timer frequency become 16 MHz / 64 = 250 kHz) and thus the Output Compare Register **OCR0A** is set to 249. In the end, the interrupts on compare match are enabled.

Thanks to the 64 prescaler, the **ISR** function will be called every millisecond, which is the time needed to generate an interrupt. This allows the possibility to retrieve always the current time of the system without wasting CPU cycles on the main loop.

```
//timer
ISR(TIMER0_COMPA_vect) {
    system_timer++;
}

void timer_init() {
    TCCR0A = (1 << WGM01);  // CTC mode
    TCCR0B = (1 << CS01) | (1 << CS00);  // Prescaler 64
    OCR0A = 249;  // Per 1ms con clock a 16MHz
    TIMSK0 = (1 << OCIE0A);  // Enable interrupt pn compare match
}

uint32_t millis() {
    uint32_t m;
    cli();  // Disable  interrupts
    m = system_timer;
    sei();  // Re-enable  interrupts
    return m;
}
```

Figure 23: Functions to handling the timing in the main file.

In the end, in the *main* function, the logic of the entire process is performed. The various methods called will not be discussed in detail, because there has already been sufficient explanation in the previous subchapters.

1. When the board is first powered up, the system enters an "**idle**" state, waiting for a tag to be scanned.
2. The first tag that is scanned is registered as master tag and, after this, the system enters in "**normal mode**", a situation in which it waits for successive scanning, but the master tag is already saved. It is important to note that the scanning from the RFID reader is performed only if the Ultrasonic sensor perceives a proximity of 6cm, to avoid keeping the reader always active.
3. The successive scanning is a master tag: this will trigger the "**master mode**", that will grant or remove access authority to a normal tag. At this point, a normal tag must be scanned: if it's not present in the **authorized_tags**, it is added, if it's already present, is removed. It's good to notice that the **master mode** has also a timeout, at the end of which, if no tag has been scanned, it will make the system return to **normal mode**.
4. Scanning a non-authorized tag (not present in the **authorized_tags** array), will trigger an "access denied" alert, while by being authorized, the tag will be granted access.
5. Upon successful access, the door (servo motor) will open, will stay open for a given interval of time and then will close, returning the system in **normal mode**.

```c
int main(void) {
    uart_init(MYUBRR);
    spi_init();
    mfrc522_init();
    lcd_init();
    servo_init();
    ultrasonic_init();
    timer_init();
    sei();  // Enable global interrupts

    uart_print("RFID Access Control System Initialized\r\n");
    uart_print("Waiting for master tag...\r\n");

    uint8_t status;
    uint8_t tag_type[MAX_LEN];
    uint8_t serial_number[5];
    uint32_t last_action_time = 0;
    uint8_t last_tag[5] = {0};
    uint8_t door_state = 0;  // 0: close, 1: opening, 2: open, 3: closing
    uint32_t door_action_time = 0;
    uint32_t display_toggle_time = 0;
    uint8_t display_state = 0;  // 0: "Access Control", 1: "Waiting for master"

    while (1) {
        if (!master_tag_saved) {
            // Toggle display every 3 seconds until master tag is registered
            if (millis() - display_toggle_time > 3000) {
                lcd_command(0x01);  // Clear display
                if (display_state == 0) {
                    lcd_string("Access Control");
                    lcd_command(0xC0);  // Go to second line
                    lcd_string("System Ready");
                    display_state = 1;
                } else {
                    lcd_string("Waiting for");
                    lcd_command(0xC0);  // Go to second line
                    lcd_string("Master Tag");
                    display_state = 0;
                }
                display_toggle_time = millis();
            }
        }
```

```c
// Check for master mode - timeout
if (master_mode && (millis() - master_mode_start_time > 5000)) {
    master_mode = 0;
    lcd_command(0x01);  // Clear display
    lcd_string("Master Mode");
    lcd_command(0xC0);  // Go to second line
    lcd_string("Timeout");
    uart_print("Master mode timeout\r\n");
    _delay_ms(2000);  // Show timeout message for 2 seconds
    lcd_command(0x01);
    lcd_string("Access Control");
    lcd_command(0xC0);
    lcd_string("Scan Your Tag");
}
```

```
uint16_t distance = get_distance();
if (distance < 60) {  // If an object (person) is in a 6cm proximity
    status = mfrc522_request(PICC_REQIDL, tag_type);
    if (status == 0) {
        status = mfrc522_anticoll(serial_number);
        if (status == 0) {
            uart_print("Tag detected: ");
            print_tag(serial_number);
            // Check if is a new tag
            if (memcmp(serial_number, last_tag, 5) != 0) {
                memcpy(last_tag, serial_number, 5);
                if (!master_tag_saved) {    // If is a new card, is saved as Master tag
                    save_master_tag(serial_number);
                    lcd_command(0x01);  // Clear display
                    lcd_string("Master Tag");
                    lcd_command(0xC0);  // Go to second line
                    lcd_string("Registered");
                    uart_print("Master tag registered\r\n");
                    last_action_time = millis();
                } else if (is_master_tag(serial_number)) {
                    master_mode = !master_mode;
                    lcd_command(0x01);  // Clear display
                    lcd_string(master_mode ? "Master Mode" : "Normal Mode");
                    uart_print(master_mode ? "Entered master mode\r\n" : "Exited master mode\r\n");
                    if (master_mode) {
                        master_mode_start_time = millis();
                    }
                    last_action_time = millis();
                    } else if (master_mode) {
                        if (is_authorized(serial_number)) {
                            remove_tag(serial_number);
                        } else {
                            add_tag(serial_number);
                        }
                        master_mode_start_time = millis();  // Reset master mode timer
                        last_action_time = millis();
                    } else {
                        if (is_authorized(serial_number)) {
                            lcd_command(0x01);
                            lcd_string("Access Granted");
                            lcd_command(0xC0);
                            lcd_string("Opening Door");
                            servo_set_position(4799);  // Open door
                            door_state = 1;
                            door_action_time = millis();
                            last_action_time = millis();
                        } else {
                            lcd_command(0x01);
                            lcd_string("Access Denied");
                            last_action_time = millis();
                        }
                    }
                }
            }
        }
    } else {
        // If there is no tag, rest last_tag
        memset(last_tag, 0, 5);
    }
```

```
        // Handling of the door status
        switch (door_state) {
            case 1:  // Opening
                if (millis() - door_action_time > 3000) {
                    lcd_command(0x01);
                    lcd_string("Opening door");
                    door_state = 2;
                    door_action_time = millis();
                }
                break;
            case 2:  // Open
                if (millis() - door_action_time > 5000) {
                    lcd_command(0x01);
                    lcd_string("Closing door");
                    servo_set_position(1499);  // Closing the door
                    door_state = 3;
                    door_action_time = millis();
                }
                break;
            case 3:  // Closing
                if (millis() - door_action_time > 3000) {
                    lcd_command(0x01);
                    lcd_string("Door closed");
                    door_state = 0;
                    last_action_time = millis();
                }
                break;
        }

        // If more than 7 second are elapsed from the last action performed, return to the default state
        if (master_tag_saved && millis() - last_action_time > 7000 && door_state == 0 && !master_mode) {
        lcd_command(0x01);
        lcd_string("Access Control");
        lcd_command(0xC0);
        lcd_string("Scan Your Tag");
        last_action_time = millis();
        }

        _delay_ms(100);  // LLittle delay before the next iteration
    }

    return 0;
}
```

Figure 24: The main function.

5. Photographic documentation

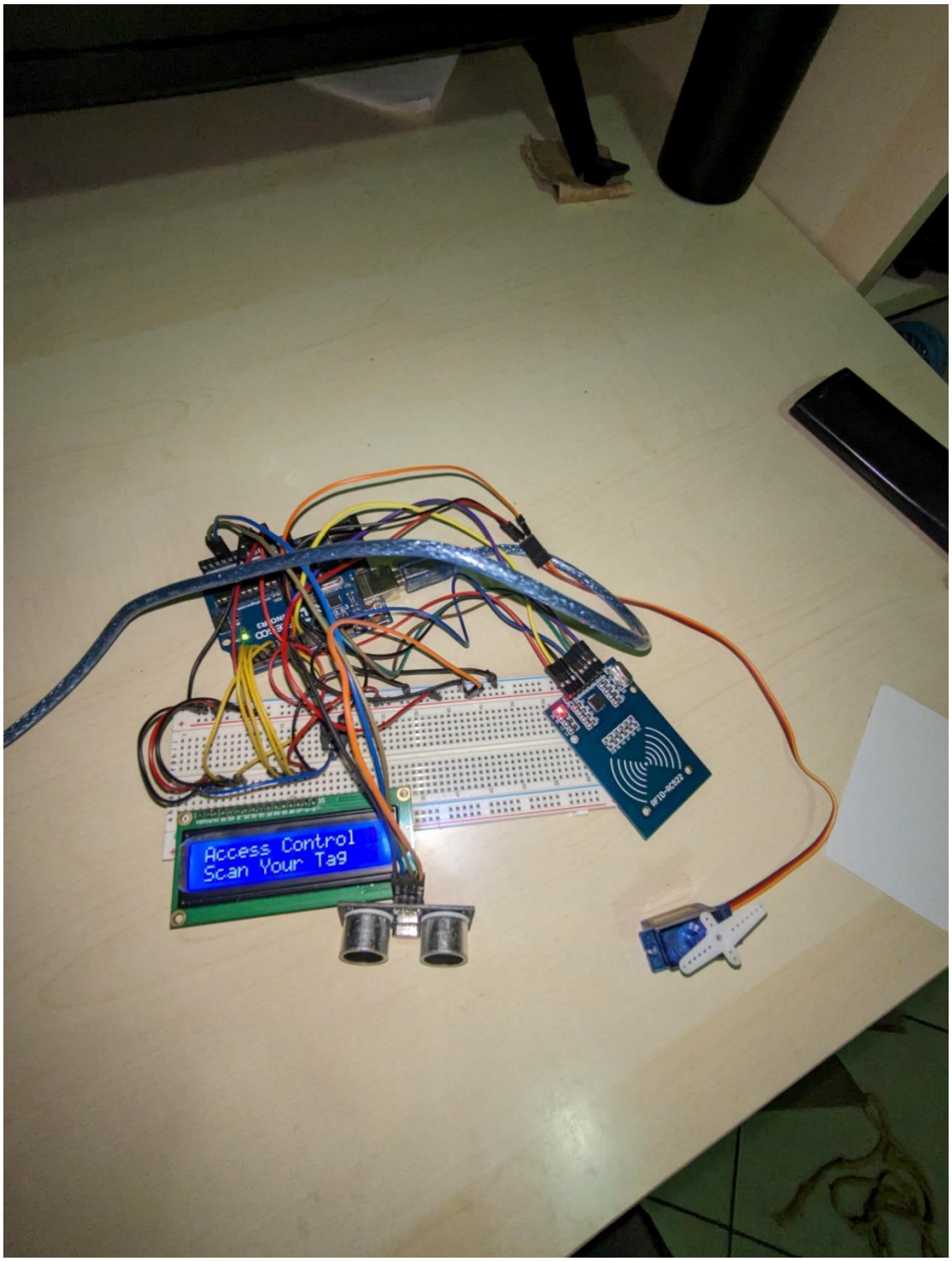


Figure 25: A photo of the realized prototype.

Along with this report, are also provided two videos, one showing the prototype during its execution, the other showing the serial monitor.

- **Serial Monitor video** --> https://drive.google.com/file/d/1-XMqe7CRjHYNNJOUZC5DVMY0ClDe44d1/view?usp=sharing
- **Prototype video** --> https://drive.google.com/file/d/1ToJc3LWXChYjXhk0mv_LgjAwCXINuBKA/view?usp=sharing

6. Conclusions and future improvements

The project proved to be effective by implementing an RFID-based control system which uses the Arduino platform and the ATMega328P microcontroller.
This work demonstrates the advantages of employing microcontrollers in the design of access control systems not only in terms of cost but also in customization. The system also manages multiple authorized users and provides real time feedback so that, ideally, can be used in many different applications, from residential to small business settings. However, while the developed Smart Lock System has been successfully introduced and serves its purpose, there are still several areas that may be improved in the system:

- **Encryption**: Deploy an RFID data encryption mechanism to address the prospect of data leakage and counterfeiting.
- **Logging System**: Develop a log that records all access attempts to the device and detects whether it is successful or not.
- **Wireless Connectivity**: Allow Wi-Fi to be available in the device, so that can be controlled and remotely, for example by connecting to the structure an ESP32 device.
- **Multiple verification step**: Enable various verification steps such as keying in a pin in an electronic keypad or inserting a fingerprint reader for a multi-factor identification.
- **Environmental adaptability**: Modify the design by using temperature and humidity sensors in the system so that it can limit its operation only under favorable environment conditions.
- **Integration with Smart Home Systems**: Enable integration with popular smart home platforms for seamless incorporation into broader home automation setups.
- **Quality of service**: Implement different timeout based on how long the tag is read by the RFID reader: for example, for the master tag, if it stays for 2 seconds, the system opens the door, if it stays for 5 seconds, it triggers master mode and if it stays for 10 seconds, it removes all the authorized tags already registered.
- **Persistent Storage for Registered Tags**: Implement a system to store registered tags in non-volatile memory, ensuring that the authorized tags are retained even after a board reset or power cycle, for example by utilizing the Arduino's built-in EEPROM to store the tag data.

If these aspects are worked on, the Smart Lock System could evolve from a simple study project into a more comprehensive solution in the access control scenario, by being not only safe but also user friendly and efficient and that can be adaptable to a wider range of applications and user needs.