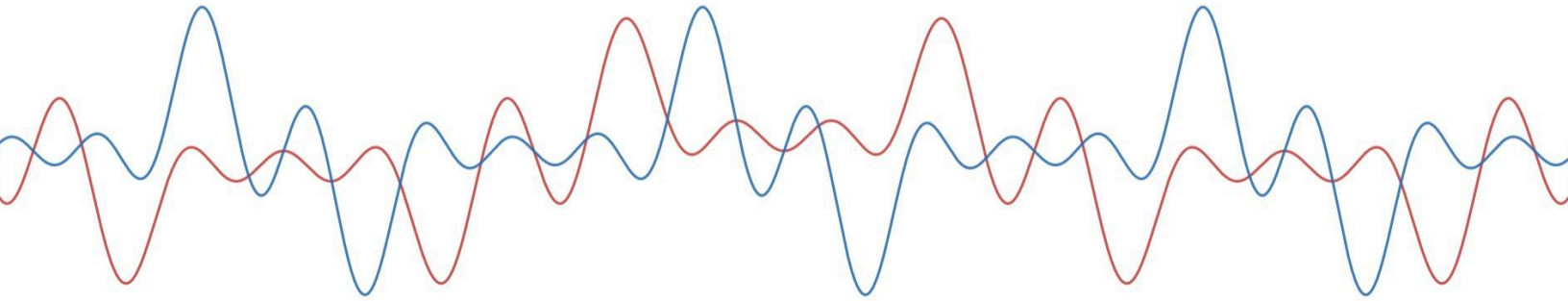
RFID Lockbox

Microcontroller Systems - CMPE 3815 Final Project

Report by Alexa Witkin & Nate Seibold

Project GitHub Repository



Abstract

This project presents the design and implementation of a compact RFID-controlled lockbox intended to provide secure personal storage for an end user sharing a living space with a roommate. The goal was to create a reliable, stand-alone access-control system while gaining hands-on experience working with the ATmega328P as a discrete microcontroller rather than relying on a full Arduino development board. The system integrates an RC522 RFID reader operating at 13.56 MHz, a custom-designed PCB incorporating power regulation, logic-level shifting, and microcontroller support circuitry, and a servo-based mechanical locking mechanism housed within a laser-cut wooden enclosure. Throughout development, particular attention was given to power constraints, signal-integrity challenges between 5V and 3.3V domains, and practical issues such as servo initialization and UID formatting. The final design successfully demonstrates a functional, durable lockbox that reliably authenticates RFID tags and actuates a locking mechanism, achieving the project's objectives in both usability and technical learning.

RFID Overview

Radio Frequency Identification (RFID) is a contactless communication technology that uses radio waves to identify and exchange data with electronic tags. An RFID system consists of two main components: a reader and a tag. The reader emits an electromagnetic field, and when a passive tag enters this field, it becomes powered by induction and sends data back to the reader using a technique called load modulation or backscatter [1]. Because the tag does not contain its own power source, the communication range is typically short; roughly a few centimeters for the RC522 system.

RC522 RFID Module

The RC522 is a widely used RFID reader/writer module built around the MFRC522 integrated circuit. It operates at the 13.56 MHz high-frequency band, which is commonly used for short-range applications such as access control, contactless cards, and inventory systems [1][2]. The module supports various ISO/IEC standards, most notably ISO/IEC 14443A, and is compatible with common tag types including MIFARE and NTAG cards and key fobs [2].

The RFID module is designed to be easily integrated with microcontrollers (such as the Arduino). It offers three communication interfaces: SPI, I²C, and UART. SPI is the most commonly used due to its speed and simplicity [1][2]. In typical Arduino applications, the module is connected via SPI using pins such as MOSI, MISO, SCK, and SDA/SS.

Electrical and Operating Characteristics

The RC522 operates at 2.5-3.3 V, and although the chip itself is designed for 3.3V logic, many breakout boards are built with circuitry that remains tolerant of 5V Arduino logic on communication pins, making them straightforward to connect to the Arduino Uno [1][2]. The module typically draws 13-26 mA during operation and around 10-13 mA in standby mode [2].

In practical use, the module achieves a read distance of about 5 cm, although the exact range depends on the tag type, antenna size, and module orientation [1][2].

RFID Tags

The RC522 communicates with passive 13.56 MHz tags, which require no battery to operate. These tags store data in memory blocks that can be read or written by the reader, depending on tag type and security permissions [1]. The tags provided in typical RC522 starter kits use an authentication scheme that must be completed before reading from or writing to protected memory sectors.

Using the RC522 With an Arduino

To use the RC522 module with an Arduino, the most common approach is to connect the module via SPI, install the widely used MFRC522 library, and upload example code to read tag UIDs (Unique Identifiers) [1]. The UID is a series of bytes uniquely associated with each tag and is often used in projects such as attendance systems, door locks, or inventory tracking.

Design Considerations

To begin the design process, several elements were first evaluated. Firstly, the project's permanent installation played a key role in the move towards a custom PCB design being used. The main advantage of using a PCB in this case would be to provide all of the circuitry in a compact form factor, while also being more permanent than a breadboard project. This would be ideal as the circuit would be expected to remain idle for long periods of time, while also retaining its reliability. In order to keep the PCB as compact as possible, it was also decided to use a standalone microcontroller IC, as the inclusion of even a small Arduino (Arduino Nano, or equivalent) would increase the size of the board, and be less "exciting". Due to its familiarity and low-cost, the ATMega328P was chosen for this purpose, as it is the same chip found in the original Arduino UNO. With this in mind, the microcontroller can also be programmed in a familiar manner using the Arduino IDE.

During the early stages of project planning, it was realized that the RFID module selected for the design ran on 3.3 V instead of 5 V. Research conducted showed that the module could still safely receive 0-5 V logic inputs, provided that the chip itself was powered by 3.3 V. Early design implementations used this to their advantage, offering a significantly simplified circuit layout, however, testing quickly revealed that the modules did not perform this way. This led to the realization that 3 of the 4 (SDA, SCK, MOSI) signals passed between the microcontroller and RFID module needed to be stepped down from 5 V to 3.3 V. Since the ATMega328P's logic high threshold is approximately 3 V, it was determined that for added simplicity, the MISO signal did not need to be amplified from 3.3 V to 5 V. As the signals were unidirectional, a logic level converter IC was added to the circuit to perform the level conversion more efficiently than a typical resistive voltage divider. The specific IC chosen was the Texas Instruments CD40109.

Lastly, the servo for the project had to be selected. At first, a 3 kg servo was chosen with the thought of its additional strength being needed to keep the box safe from potential intruders. With that in mind, additional research was performed on a specific motor before it was discovered that the motor was able to draw over 3 A under load. To reduce the risk of a PCB trace blowing out, it was decided to use a smaller servo motor, with a different latch design that reduced the strain on the servo motor itself. This design change allowed for the standard GS-90 servo to be used instead. The much lower power was considered an acceptable tradeoff to reduce the risk of PCB failure, or even fire. This was of the utmost importance as the final PCB would be placed inside a wooden box, in the bedroom of the end-user.

PCB Design

With the initial design outlined, work on the PCB could now begin, of course starting with a schematic. The first section of the schematic design was the power regulation. This section contained two voltage regulators, one for the 5 V system, and one for the 3.3 V system. To simplify layout, a family of voltage regulators with both of these options available was chosen, in this case the Diodes Incorporated AZ1117CH2. Both regulators were paired with the datasheet-specified decoupling capacitors. Additionally, a Schottky diode was added in series with the main power input trace to prevent reverse polarity, and also a power LED was added. All surface mount components utilized standard 1206 footprints, mainly to allow for easy assembly. Below, Fig. 1 shows the schematic for this section.

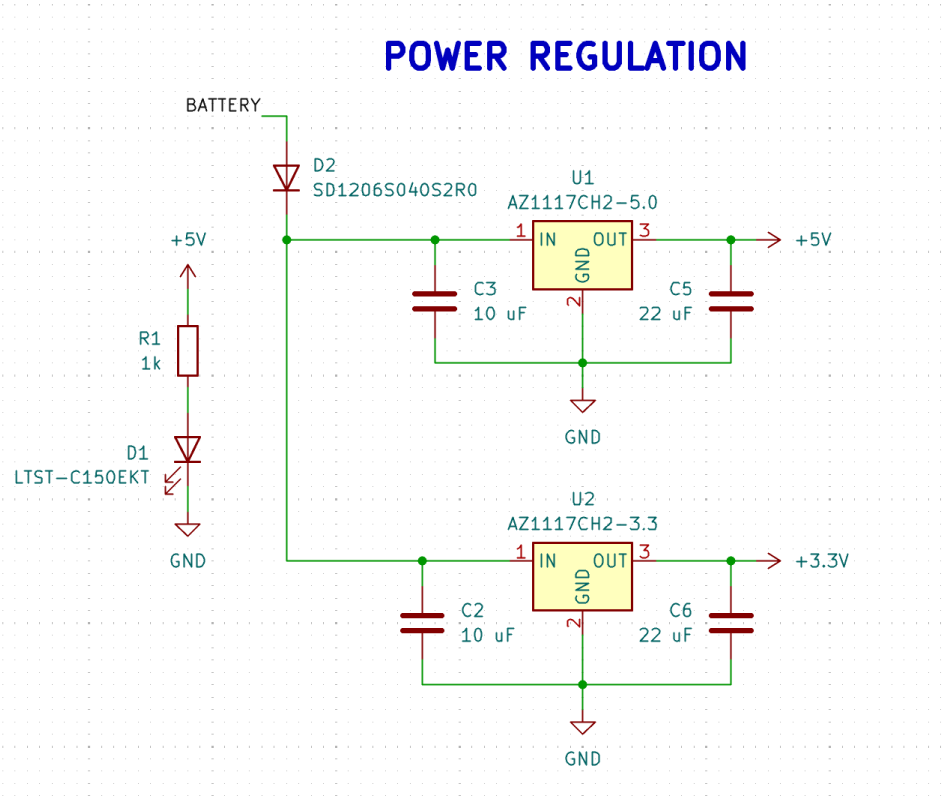


Fig. 1 - KiCad schematic of power regulation circuitry.

With the power supply of the PCB designed, the microcontroller could then be incorporated. The microcontroller was connected to 5 V with a decoupling capacitor, while also ensuring to connect the inbuilt ADC reference voltage to the same power supply. After incorporating a 16 MHz crystal, a basic reset switch circuit was added. The only remaining connections at that point were for I/O. The specific pins were chosen to mirror those used in Arduino UNO-based prototypes. The completed schematic of the ATMega can be seen below in Fig. 2.

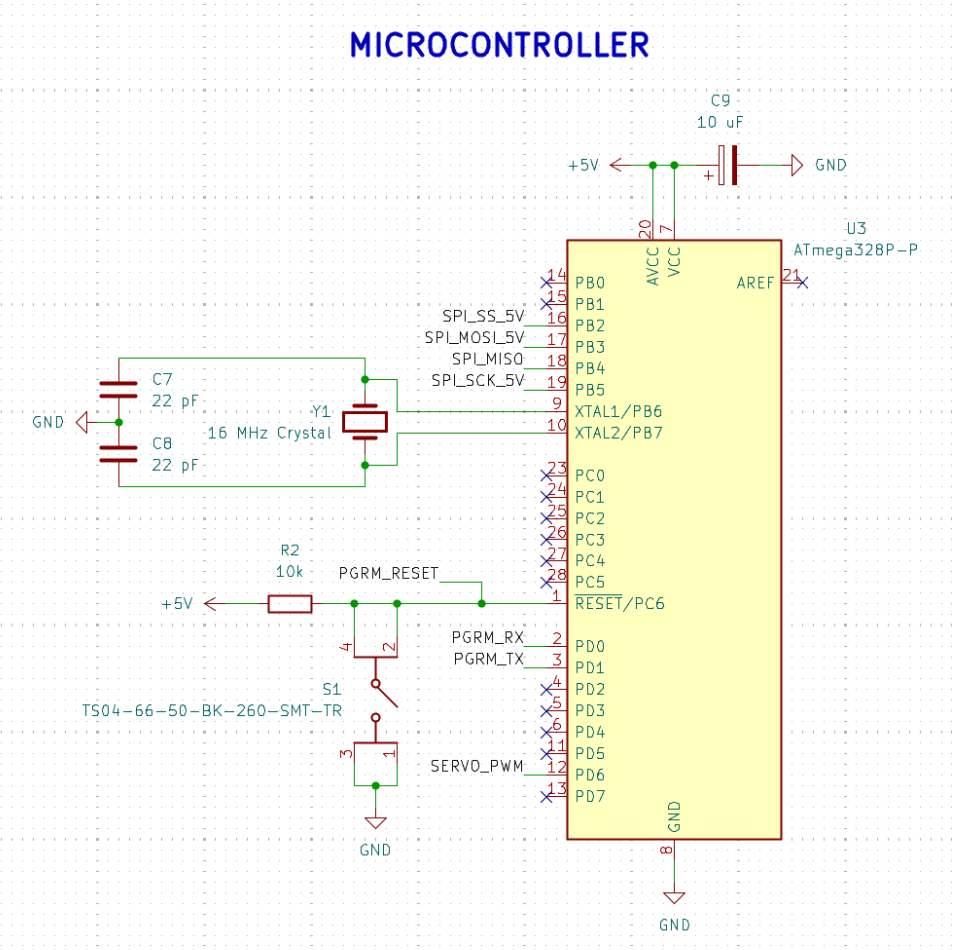


Fig. 2 - KiCad schematic of ATMega328P integration.

As previously discussed, 3 of the 4 connections to the RFID would require level shifting. And therefore the next aspect to be incorporated was the logic level shifting IC. Since this IC had never been used before, the datasheet was consulted, and the recommendations for integration were followed. The final design included decoupling capacitors, and the required signal traces, as seen below in Fig. 3.

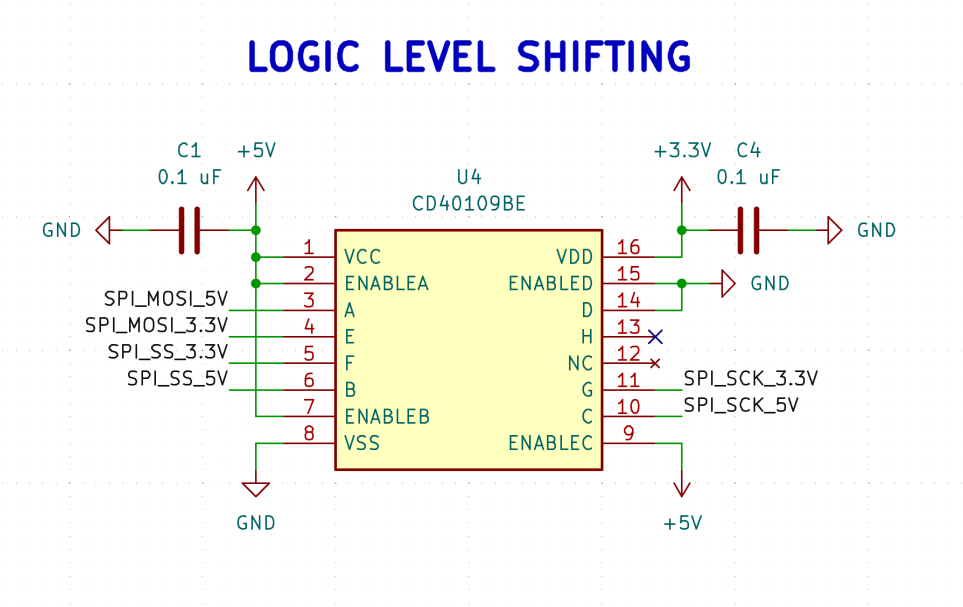


Fig. 3 - KiCad schematic of TI CD40109 integration.

Lastly, the I/O was added to the circuit. Although bulkier, screw terminals were used to prevent the connections from coming loose over time. Three terminals were used in total, one for power, one for the servo, and one for the RFID. The corresponding pinouts can be seen below in Fig. 4.

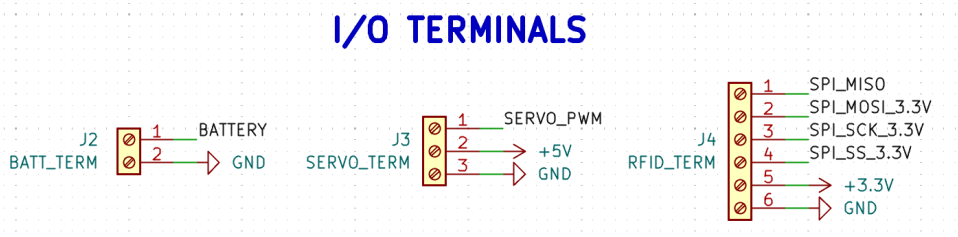


Fig. 4 - KiCad schematic of PCB I/O.

Additionally, a header was added including all the necessary pins for bootloader burning, and also programming using a separate Arduino UNO board, since no USB-Serial converter was included. The header pinout can be found below in Fig. 5.

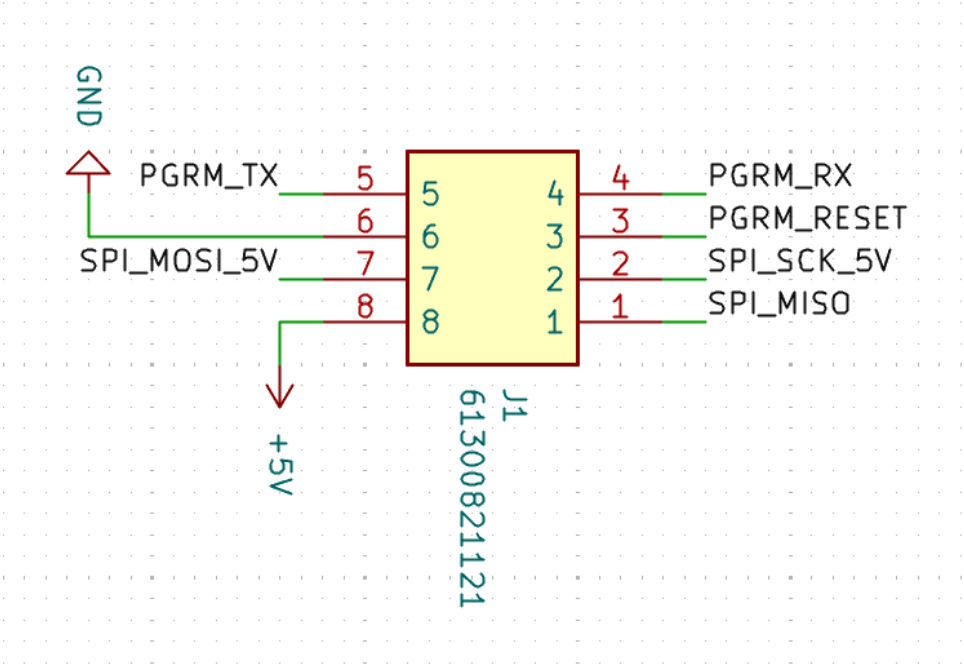


Fig. 5 - KiCad schematic of programming headers.

With the schematic determined, a design for the PCB could be started. The goal of the design was to be compact, but also practical to assemble with basic soldering skills. With that in mind, the footprints of the individual components were placed. Attempts were made to provide clean routes for traces, although due to the conflicting pinouts of some components, several non-ideal traces remained. Below, Fig. 6 shows the footprints outlined on the board.

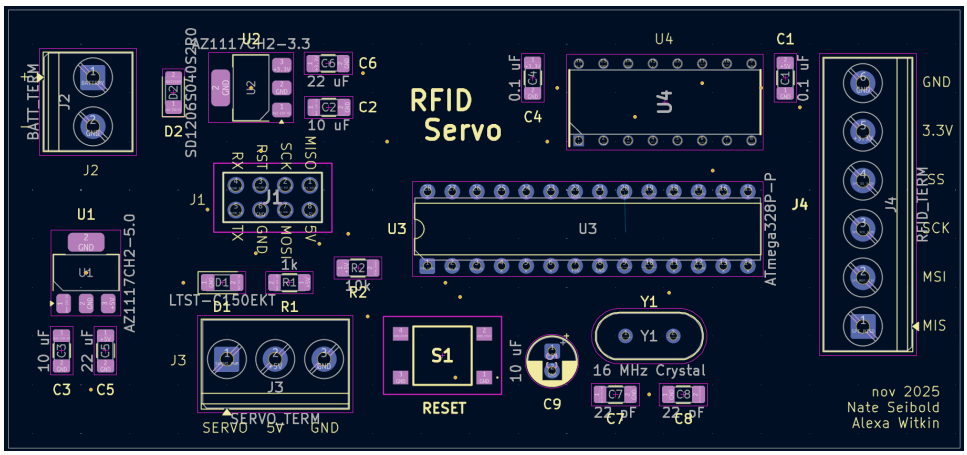


Fig. 6 - KiCad layout of component footprints.

Following standard practice, power and ground planes were used to produce a stable and easily manufactured board. Signal routing was split between the top and

bottom planes fairly evenly, with care taken to widen traces associated with the power regulation circuitry. One notable error noticed after production was the “island” of 5 V power between the ATMega and level shifter. This is likely attributed to severe sleep deprivation during the design phase, and was fixed after the fact using a jumper wire. Top and bottom copper plane layouts can be found below in Fig. 7.

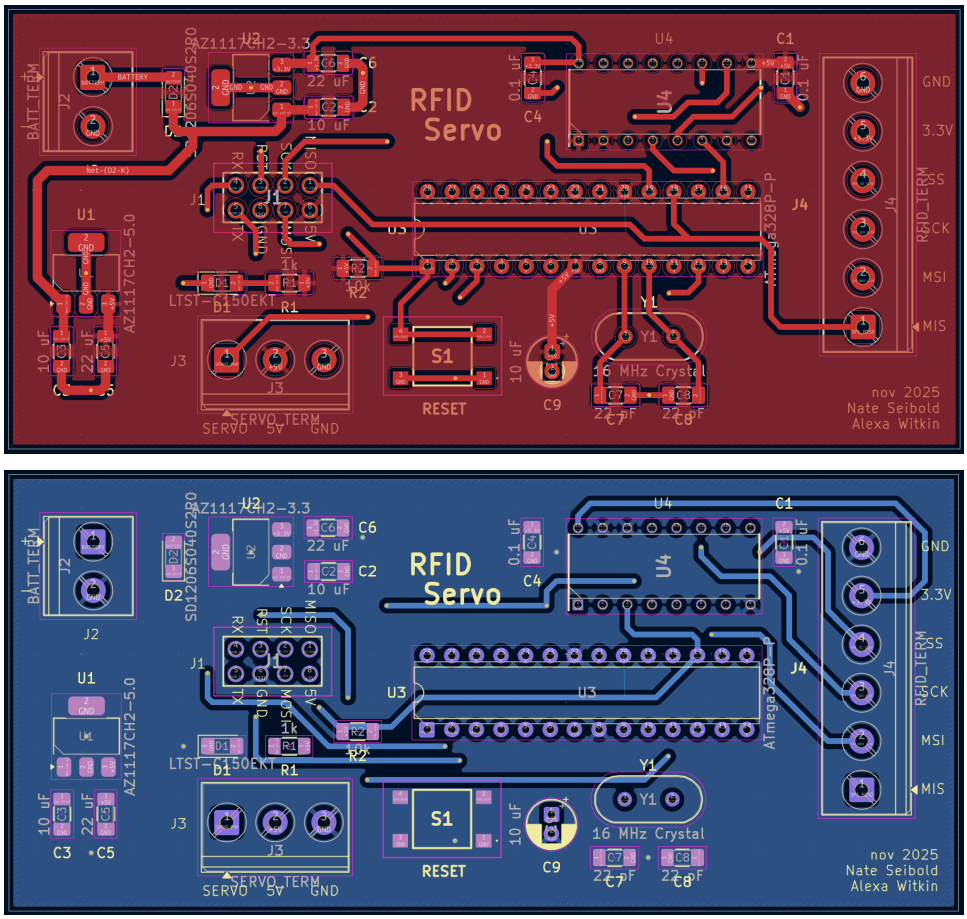


Fig. 7 - KiCad layout of top (red) and bottom (blue) copper planes.

Following their design, the PCBs were manufactured using JLCPCB. The final product can be seen below in Fig. 8.

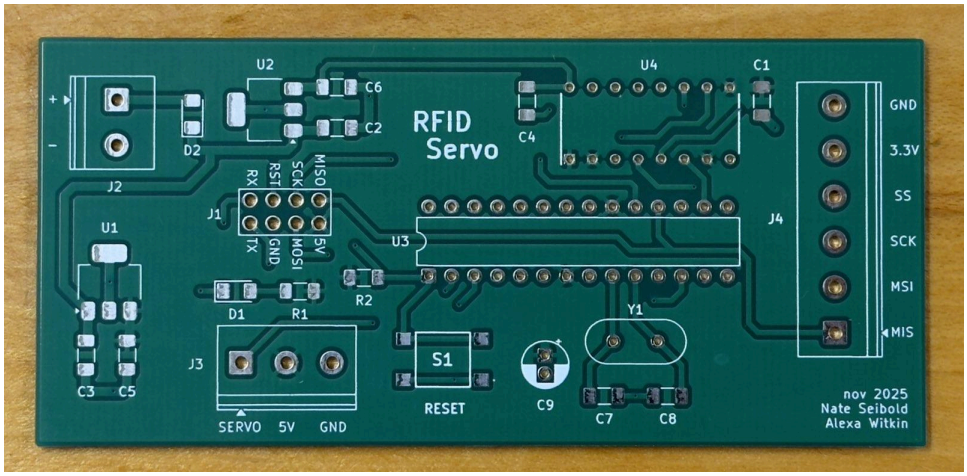


Fig. 8 - Manufactured PCB.

PCB Implementation

With the PCBs manufactured, the assembly process could begin. The option of having the PCBs assembled by the manufacturer was considered, however the use of several more unique through-hole components made this option not cost-effective. Instead, the through-hole components were hand soldered, and solder paste with a heat gun was used for the surface-mount components. A completed PCB can be seen below in Fig. 9.

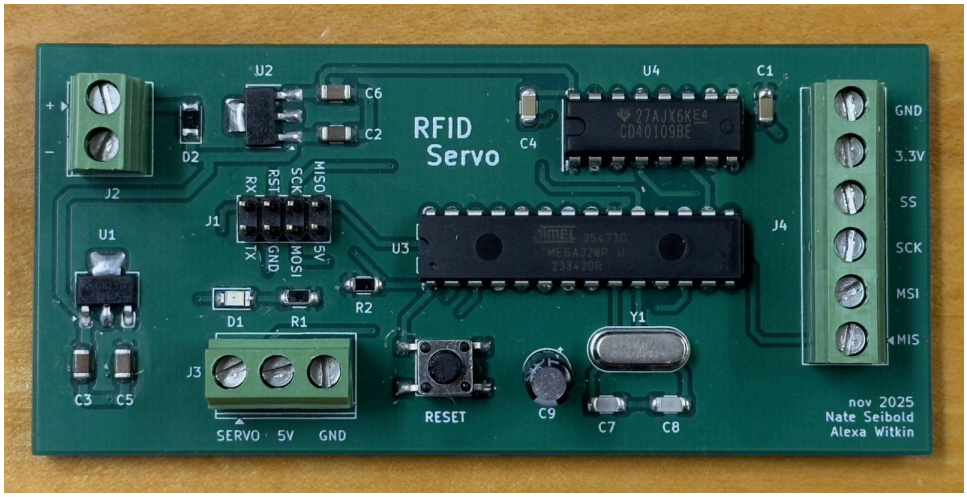


Fig. 9 - Manufactured PCB after component assembly.

After installing the components, the first step was to burn a bootloader onto the ATMega328P to allow for it to be easily programmed later on. A bootloader is a small but essential section of code that is permanently burned into the fuses of the microcontroller. This code is what instructs the microcontroller how to behave upon

startup. In this case, the bootloader checks for new code and also instructs the microcontroller to expect an external clock signal from the 16 MHz crystal. The Arduino IDE contains all the necessary programs to use an existing Arduino UNO to burn the bootloader into a new ATMega, however several difficulties were experienced when this was attempted. The burning procedure repeatedly failed after not being able to identify the new microcontroller. After multiple attempts, an ATMega328P on a breadboard was able to be bootloaded by bypassing the external clock (instead utilizing the internal 8 MHz oscillator) and by providing a temporary 1 MHz jump-start signal using a waveform generator. This setup can be viewed below in Fig. 10.

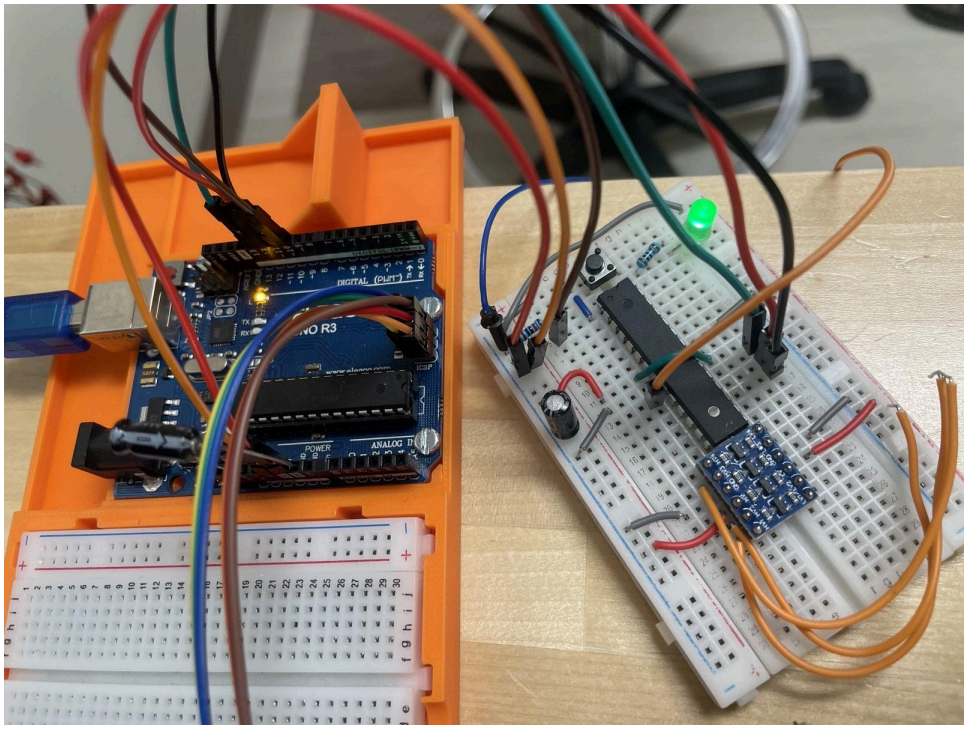


Fig. 10 - Arduino UNO burning bootloader into ATMega328P using ArduinoISP.

It is still unclear why the ATMega was unable to be bootloaded using the external 16 MHz crystal, as the internal oscillator is not the typical operational state of the chip. Several crystals were used to verify their operation, and they were not found to be the problem. One possible explanation would be locked fuses inside the ATMega, however the only solution to this problem is the use of a high-voltage programmer.

Once bootloaded, code can be uploaded to the ATMega using a disassembled Arduino Uno. By removing the ATMega from a (previously) operational Arduino UNO, the USB-Serial converter can be connected to the new ATMega using jumper cables. In this setup, the new board can be programmed normally using the Arduino IDE. Below, Fig. 11 shows this configuration being used on a breadboard.

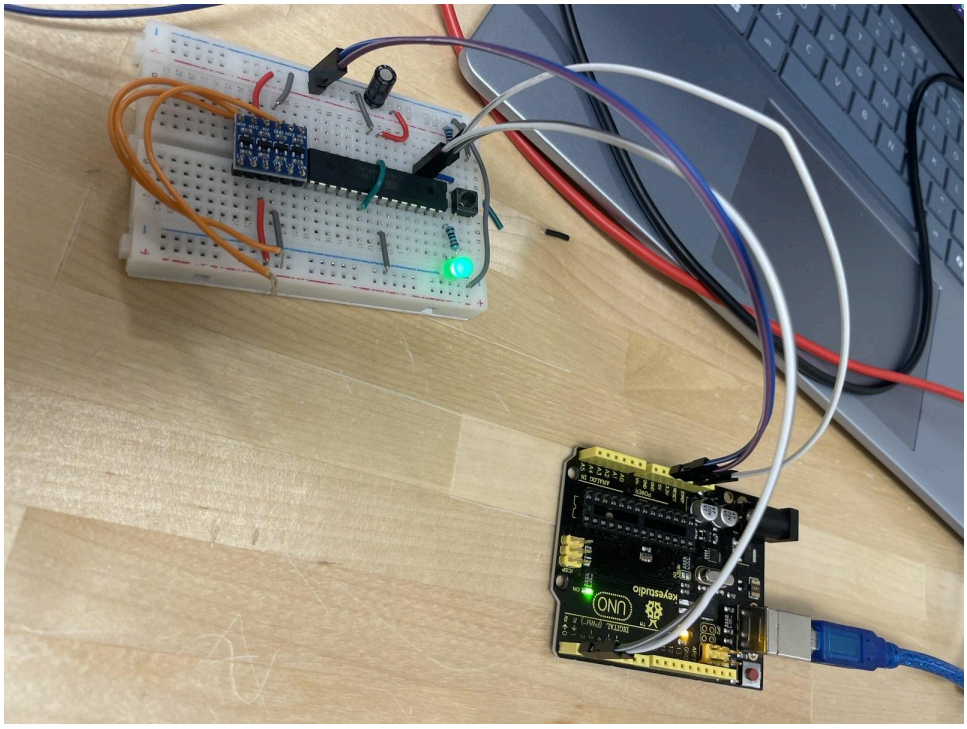


Fig. 11 - Disassembled Arduino UNO being used to program ATMega328P.

By this stage in the design process, the code had been verified on an Arduino UNO, and therefore, when the code was uploaded to the PCB ATMega and did not work, it was quickly narrowed down to be hardware based. Considering the servo motor still actuated properly, it was determined that the microcontroller was functioning properly. The logic level shifter was then explored, however it was difficult to trouble shoot the system as there was no stable method with which to probe the logic signals. Instead of continuing to test the level converter IC, the component was removed from the PCB, and a breakout of jumper wires was added to allow for an external level shifting board to be added. The board contained BSS138 MOSFETs in a common configuration for level shifting. Although not necessarily needed in this design, this circuitry allows for bi-directional level shifting of signals. This design was originally going to be included on the PCB, however the extremely small component size led to the use of an IC instead.

After making the switch, the circuit behaved as expected. In future iterations, a MOSFET-based design would likely be considered instead of the IC, or at a minimum, breadboard testing of the IC would need to take place. Another note is the importance of good signal continuity for the RFID module. The data transmission is extremely susceptible to noise and/or disconnects, and to ensure optimal performance, 22 AWG wire was soldered onto the terminals directly, with heat shrinking for protection. With the long list of issues encountered during the troubleshooting phase, ample caution was taken during the final assembly to ensure smooth operation.

Box Design

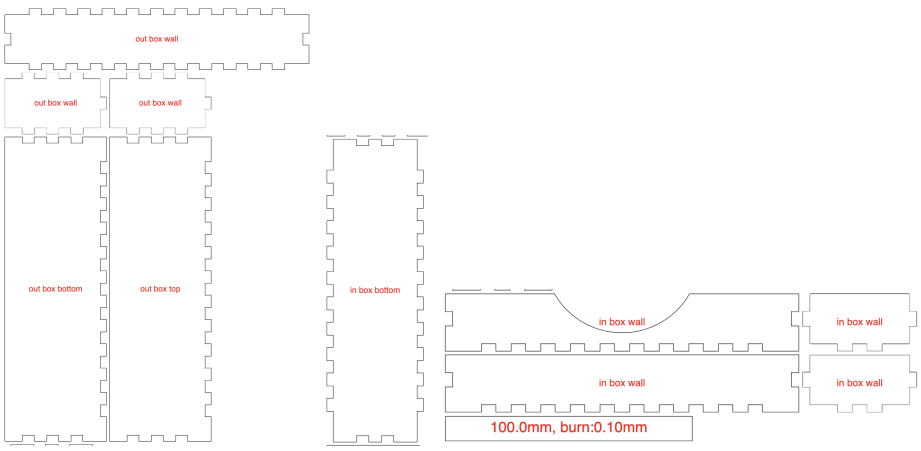


Fig. 12 - SVG file containing Sliding Drawer.

The Sliding Drawer was created using an interface on Boxes.py to specify the dimensions of the pre-made, generic drawer file [3]. This made the process very simple. Once the dimensions were measured to match the cardboard prototype (made for and checked by the end user), the .svg file (Fig. 12) was downloaded and sent to the UVM FabLab. The laser cutting machine takes the svg file and follows the print given on a sheet of wood (3mm thickness).

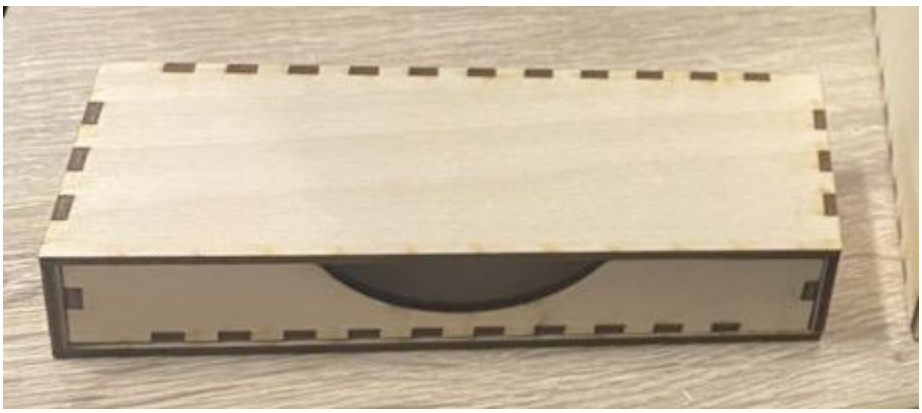


Fig. 13 - Assembled Sliding Drawer.

This drawer (Fig. 13) was created to serve as the housing for the batteries used by the circuit; allowing the user to access the batteries even when the box is locked. To ensure the battery pack wiring would be the only component passing through, a small hole was drilled in the top left corner of the outer portion of the drawer.

The sliding drawer will sit within the larger box, and the outer portion of the drawer will be glued flush to the inside of the box; again ensuring that the only things

able to pass through are the wires to the battery pack. To account for the additional 3mm wall, a piece of wood of the same thickness was glued to the back of the inner drawer component.

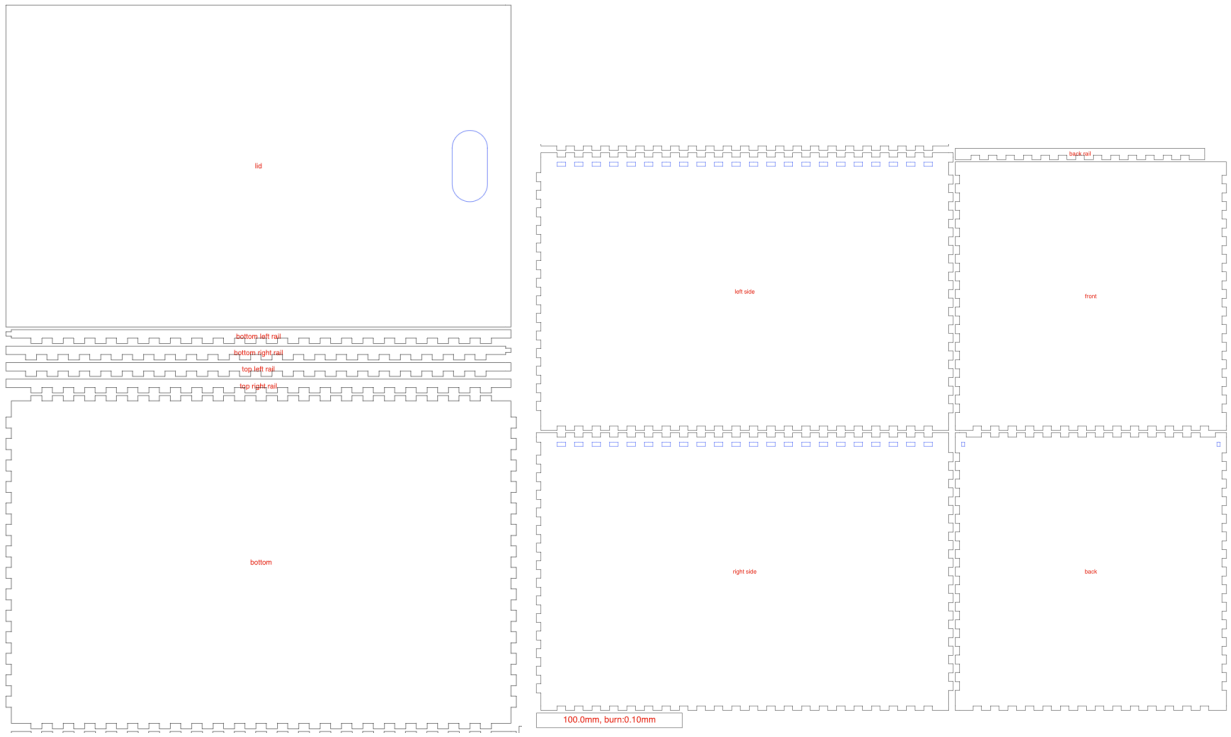


Fig. 14 - SVG file containing Sliding Lid Box.

The Sliding Lid Box was also created using the interface on Boxes.py to specify the dimensions of the pre-made, generic box with a sliding lid [4]. Again the dimensions were measured to match the cardboard prototype (made for and checked by the end user). The svg file (Fig. 14) was downloaded and sent to the UVM FabLab. The laser cutting machine takes the svg file and follows the print given on a sheet of wood (3mm thickness).

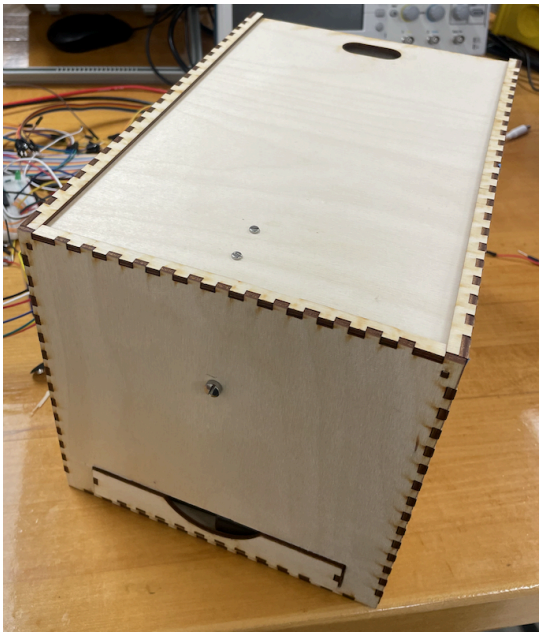


Fig. 15 - Assembled Sliding Lid Box.

This box (Fig. 15) serves as the housing for the Sliding Drawer, PCB, wired components, locking mechanisms, and the user's belongings. A rectangle was cut out of the back wall of the box for the Sliding Drawer, and several holes were drilled to add the locking mechanism; bolts for rigidity and screws to secure wire that would act as the hook for the lock. The user requested roughly 6inx6inx6in of interior space for their belongings, so all of the hardware is on the rear side of the box.

Code

To create a basic access-control mechanism, the final system brings together three major components: RFID scanning, UID verification, and Servo actuation. The code was organized into three files: main.ino, RFIDfunction.ino, and Servofunction.ino. Separating the functionality made the program easier to troubleshoot and modify as individual problems came up during development.

Main Program Flow (main.ino)

The central logic of the system is inside main.ino, which handles initialization and runs a loop that continuously checks for an RFID card. Initialization includes setting up serial communication, starting the SPI bus, and initializing the MFRC522 RFID module.

```
void setup() {
```

```
 Serial.begin(19200); // Initialize serial communications with the PC
 SPI.begin();      // Init SPI bus
 rfid.PCD_Init();    // Init MFRC522
 delay(4); // Optional delay
 Serial.println(F("This code scan the MIFARE Classic NUID."));

 lock.attach(6); // lock is in pin D10
}
```

One practical issue that came up early was servo initialization. At first, we expected the servo to start at position 0°, but in reality, servos often power up in whatever position they were last physically in. This meant the first motion could be unpredictable or even jam the locking mechanism. To address this, the program reads the servo's current position dynamically rather than assuming a fixed starting angle. That correction happens later inside servoFunction().

The loop that drives the system is simple and relies on the helper function getUID():

```
void loop() {
 if (getUID() == validID) {
   Serial.println("Servo running");
   servoFunction();
 }
}
```

During early testing, the system was unable to trigger the servo even when the correct RFID tag was present. To check what UID values were being read by the getUID() function, there were placeholder values added to show when no card was detected, or when the wrong UID was received.

Reading the RFID Tag (RFIDfunction.ino)

```
FVersion: 0x18 = (unknown)
Scan PICC to see UID, SAK, type, and data blocks...
Firmware Version: 0x18 = (unknown)
Scan PICC to see UID, SAK, type, and data blocks...
Card UID: C9 53 01 83
Card SAK: 08
PICC type: MIFARE 1KB
Sector Block   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15  AccessBits
  15     63  MIFARE_Read() failed: A MIFARE PICC responded with NAK.
  15     62  PCD_Authenticate() failed: Timeout in communication.
  14     59  PCD_Authenticate() failed: Timeout in communication.
  13     55  PCD_Authenticate() failed: Timeout in communication.
  12     51  PCD_Authenticate() failed: Timeout in communication.
  11     47  PCD_Authenticate() failed: Timeout in communication.
  10     43  PCD_Authenticate() failed: Timeout in communication.
   9     39  PCD_Authenticate() failed: Timeout in communication.
   8     35  PCD_Authenticate() failed: Timeout in communication.
   7     31  PCD_Authenticate() failed: Timeout in communication.
   6     27  PCD_Authenticate() failed: Timeout in communication.
   5     23  PCD_Authenticate() failed: Timeout in communication.
   4     19  PCD_Authenticate() failed: Timeout in communication.
   3     15  PCD_Authenticate() failed: Timeout in communication.
   2     11  PCD_Authenticate() failed: Timeout in communication.
   1      7  PCD_Authenticate() failed: Timeout in communication.
   0      3  PCD_Authenticate() failed: Timeout in communication.
```

Fig. 16 - Serial Monitor Output of DumpInfo from the MFRC522 Library.

The function responsible for scanning a card and extracting its UID (as done in Fig. 16) is shown below:

```
String getUID() {
 uid = "";
 delay(1000);
 // Reset the loop if no new card present on the sensor/reader. This
saves the entire process when idle.
 if ( ! rfid.PICC_IsNewCardPresent())
   return "123456789";
 // Verify if the NUID has been read
 if ( ! rfid.PICC_ReadCardSerial()) {
   return "123456789";
 }
 // Serial.print(F("PICC type: "));
 MFRC522::PICC_Type piccType = rfid.PICC_GetType(rfid.uid.sak);

 // Check is the PICC of Classic MIFARE type
 if (piccType != MFRC522::PICC_TYPE_MIFARE_MINI &&
```

```
    piccType != MFRC522::PICC_TYPE_MIFARE_1K &&
    piccType != MFRC522::PICC_TYPE_MIFARE_4K) {
    Serial.println(F("Your tag is not of type MIFARE Classic."));
    return "000000000";
  }
  // Store NUID into string
  for (byte i = 0; i < 4; i++) {
    uid += rfid.uid.uidByte[i];
  }

  // Halt PICC
  rfid.PICC_HaltA();

  // Stop encryption on PCD
  rfid.PCD_StopCrypto1();

  return uid;
}
```

The main challenge during coding was ensuring the UID string matched the stored validID. UID formatting varies depending on whether you print it as decimal, hexadecimal, or with spacing. The MFRC522 library examples often output UIDs in HEX, but our initial validID was stored as a decimal string. For a while, every card scan appeared “incorrect,” not because the tag was wrong, but because the format did not match our stored value.

The fix was converting each byte of the UID to a decimal string and appending it directly:

```
uid += rfid.uid.uidByte[i];
```

Although simple, this approach ensured consistent formatting and made string comparisons reliable.

The improvement of returning unique placeholder values (123456789 vs. 000000000) was incredibly helpful in deciphering whether the reader failed due to no card present, failed read, or wrong tag type. These changes made debugging much clearer while testing.

Controlling the Servo (Servofunction.ino)

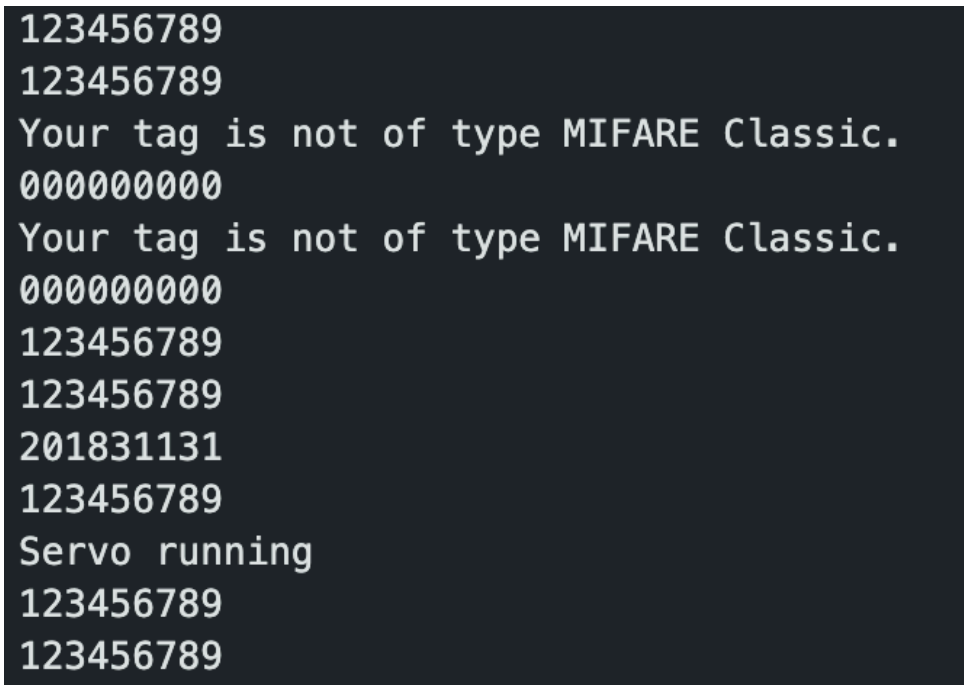


Fig. 17 - Serial Monitor Output of Correct UID Causing Servo to Move vs Placeholders.

As seen in Fig. 17, when a valid card is detected, the servo moves the lock mechanism. The servo control logic is shown below:

```
#include <Servo.h>

void servoFunction() {
 int startPos = lock.read();  // read current position (0-180)
 int endPos = startPos + 90;

 // Limit to max 180 degrees
 if (endPos > 180) endPos = 180;

 // Move to unlocked position
 for (int pos = startPos; pos <= endPos; pos++) {
   lock.write(pos);
   delay(15);
 }

 // Wait 10 seconds
```

```
  delay(10000);

  // Move back to original position
  for (int pos = endPos; pos >= startPos; pos--) {
    lock.write(pos);
    delay(15);
  }
}
```

One of the earliest issues encountered was that the servo didn't always start where the program assumed it would. Originally we hard-coded something like:

```
int startPos = 0;
```

But this caused problems because if the servo physically wasn't at 0°, the first movement would jerk violently or stall. Reading the servo's actual position using:

```
int startPos = lock.read();
```

made the motion smooth and predictable. This was one of the key practical insights gained during testing.

The motion duration and the waiting period also went through several iterations. A full one-minute delay (delay(60000)) was too long for testing and debugging, so it was temporarily shortened to 10 seconds. This will likely continue to be the length of the period as the user can always use their RFID tag again if they require more time.

Conclusions

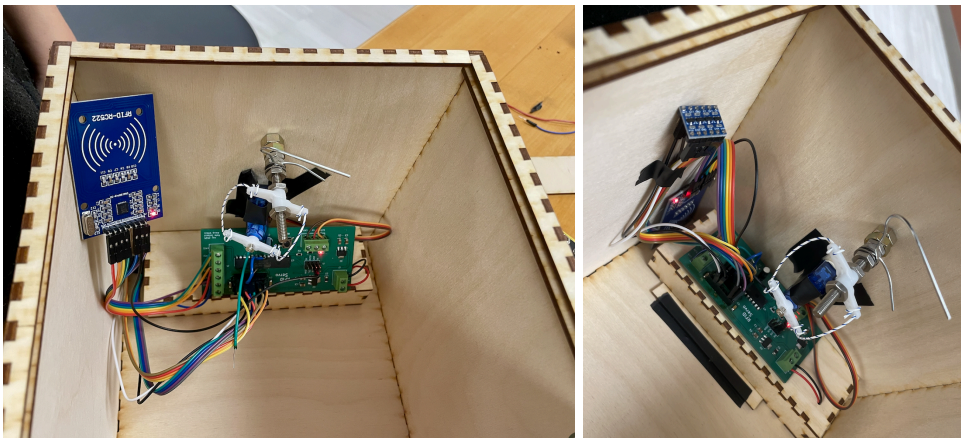


Fig. 18 - The Interior of the Lock Box
(RFID More Visible on Left and Logic Converter More Visible on Right).

The development of the RFID lockbox successfully fulfilled both the functional requirements of the end user and the educational goals of the project. By moving away from a prefabricated Arduino board and instead implementing the ATmega328P at the IC level, we gained deeper insight into microcontroller bootloading, external clock configuration, power regulation, and low-level hardware integration. The design process also highlighted the importance of proper logic-level shifting, signal stability, and thorough validation of component assumptions; particularly regarding the RC522 module and servo behavior.

Although several challenges arose during the PCB creation, including difficulties bootloading the microcontroller and issues with the initial level-shifting IC, the iterative debugging process resulted in a more robust and better-understood final circuit. The custom PCB, combined with the laser-cut enclosure and revised locking mechanism, produced a compact and reliable system that met the end user's storage and security needs. Overall, the project not only achieved a fully functional RFID-controlled lockbox, but also served as a meaningful exploration of embedded hardware design, PCB development, and practical engineering tradeoffs.

Sources

[1] LastMinuteEngineers, *"What is RFID? How It Works? Interface RC522 RFID Module with Arduino,"* Last Minute Engineers. [Online]. Available: https://lastminuteengineers.com/how-rfid-works-rc522-arduino-tutorial/

[2] Handson Technology, *RC522 RFID Development Kit Data Specs (based on NXP MFRC522),* HandsonTec.com. [Online]. Available: https://www.handsontec.com/dataspecs/RC522.pdf

[3] *SlidingDrawer* — Boxes.py generator, Hackerspace Bamberg. [Online]. Available: https://boxes.hackerspace-bamberg.de/SlidingDrawer?language=en

[4] *SlidingLidBox* — Boxes.py generator, Hackerspace Bamberg. [Online]. Available: https://boxes.hackerspace-bamberg.de/SlidingLidBox?language=en