BCSE312L Programming for IoT Boards

IoT Based Smart Door-Lock System

22BCT0116 Syed Furqaan Lateef
22BCT0052 Shreyas Sundar
22BCT0303 Arjun Sreedhar
22BCT0184 Nihal J
22BCT0244 Deerendra Saravanan
Under the Supervision of
Dr. Krishnamoorthy A
Associate Professor Grade 1
School of Computer Science and Engineering

B.Tech.

in

Computer Science and Engineering

School of Computer Science and Engineering

VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

October 2025
Google Drive Link:
https://drive.google.com/drive/folders/1w-Xryu4Bu6qUiA9PYi3iTZntR_1G1RsX?usp=sharing

TABLE OF CONTENTS

ABSTRACT

This project details the design and simulation of a flexible, multi-modal smart doorlock system within the Wokwi simulation environment. The main goal is to improve traditional door security by replacing old mechanical locks with strong, automated electronic systems. Two different implementations are developed and evaluated.

The first implementation uses an ESP32 microcontroller and its built-in WiFi capabilities for IoT integration. This model includes a keypad for entering a PIN, a PIR sensor for motion-triggered activation, and supports a One-Time Password (OTP) system for secure, time-sensitive access.

The second implementation targets ongoing, offline security using an Arduino Uno, an RFID-RC522 module for tag-based authentication, and onboard EEPROM for safely storing authorized user credentials, like RFID UIDs and PINs.

Both systems feature an LCD for user feedback and a servo motor to mimic the physical locking and unlocking action. The Wokwi simulation confirms the effectiveness of both designs, providing a side-by-side comparison of an IoT-connected system and a self-contained, persistent storage system.

1. INTRODUCTION

1.1 Aim

The primary aim of this project is to design, simulate, and analyze two distinct models of a smart doorlock system using the Wokwi online simulator. The project explores and contrasts the methodologies of an IoT-enabled, WiFi-connected system (using ESP32) with an offline, persistent-storage system (using Arduino, RFID, and EEPROM).

1.2 Objectives

To achieve the stated aim, the following objectives have been established:

To design a baseline smart lock circuit with common components, including a keypad for input, an LCD for display, and a servo motor as the lock actuator.

To implement an IoT-enabled version using an ESP32, integrating a PIR sensor for presence detection and leveraging WiFi for potential OTP-based authentication.

To implement a secure, offline version using an Arduino, integrating an RFID-RC522 module for card-based access.

To utilize the Arduino's internal EEPROM to demonstrate persistent storage of authorized RFID tags and security PINs, ensuring data retention after a power cycle.

To validate and test the functionality of both simulated systems on the Wokwi platform.

To prepare a comparative analysis of the two proposed architectures, highlighting their respective advantages, disadvantages, and ideal use cases.

1.3 Introduction

In an era of pervasive connection and smart gadgets, traditional home and workplace security solutions, particularly the mechanical lock and key, appear increasingly out of date. These traditional locks are vulnerable to physical attacks such as picking, banging, and the annoyance of misplaced or stolen keys. The Internet of Things (IoT) and low-cost microcontrollers have enabled "smart doorlocks," which provide increased security, convenience, and access control.

A smart doorlock is an electromechanical lock that can lock and unlock a door based on instructions from an authorised device. These systems frequently include various authentication methods, such as PIN codes, smartphone access, biometric data, or RFID tags.

This project focuses on the practical design of such a system, rather than physical prototyping, using Wokwi, a versatile online platform for electronics and IoT projects. Simulating the system allows us to quickly prototype, test, and iterate on various hardware configurations and software logic without incurring the expense and complexity of physical components.

This study describes two parallel smart doorlock solutions. The first model investigates the capabilities of the ESP32 as an IoT-native solution, including WiFi connectivity for advanced features such as OTP. The second model shows a more typical, self-contained system based on an Arduino, with an emphasis on security and reliability via RFID technology and an EEPROM for long-term data storage.

2. SYSTEM ANALYSIS

2.1 Related Work/Literature Survey

The topic of smart security systems is extensive, with numerous research and commercial products available. A survey of the current literature identifies three major design patterns:

Bluetooth Low Energy (BLE) Systems: Many commercial locks (such as the August Smart Lock) use BLE to interact with the user's smartphone. The unlock is triggered by either the phone's proximity or an app-based instruction. While convenient and power-efficient, they have a limited range and are dependent on the user's phone.

Biometric (fingerprint) Systems: Because they rely on unique biological characteristics, these systems are thought to be extremely safe. However, the sensors can be costly, and their dependability is prone to false rejection rates (authorised user refused) and false acceptance rates (unauthorised user accepted), particularly during "gummy finger" attacks.

Keypad-Only Systems: These are the simplest types of electronic locks. While successful, they are susceptible to "shoulder surfing" (observing someone enter the code) and code-sharing. The codes must also be manually updated.

RFID/NFC-Based Systems: These systems, which are widely utilised in corporate and hotel contexts, make use of radio frequency identification tags or cards. They are quick and convenient, but the tags themselves can be misplaced, stolen, or even duplicated.

Our suggested project expands on these ideas by developing a multimodal system. Layering security involves combining a keypad with either RFID or WiFi-based OTP. To get access, an attacker would need to compromise two independent factors (for example, take an RFID card and know the PIN), which addresses the flaws of a single-method system.

2.2 Existing System

The "existing system" in this project is the standard mechanical pin-tumbler lock, which is found in most residential and commercial properties.

Components:

A metallic key.

A lock cylinder with a succession of pins of different lengths.

Drawbacks:

No Access Control: There is no record of when or who opened the door.

Inflexible: Access cannot be granted or denied remotely or temporarily. Granting access necessitates physically duplicating a key.

Physical vulnerability: Keys are readily misplaced, stolen, or duplicated without authorisation. The locks themselves can be picked, drilled, or bumped.

Inconvenience: Users may be locked out if they forget or lose their key.

2.3 Proposed System

The proposed solution is a smart doorlock system simulated in Wokwi that outperforms the current mechanical lock. It offers automated multi-factor authentication to increase security. The system is presented in two different implementations.

2.3.1 System Architecture Overview

Both implementations have a similar set of components:

The microcontroller is the operation's "brain" (ESP32 or Arduino).

A keypad (4x4 matrix) is an input device that allows you to enter PINs or user commands.

LCD (16x2 I2C): A display that gives the user feedback (for example, "Enter PIN," "Access Granted," or "Access Denied").

A servo motor (SG90) is an actuator that simulates the physical lock mechanism by spinning 90 degrees to represent the "locked" and "unlocked" states.

2.3.2 Implementation A: ESP32-based IoT Lock (WiFi & OTP)

This approach focusses on connection and smart features, making advantage of the powerful ESP32 microcontroller.

Core component: ESP32-DevKitC.

Key features:

WiFi Connectivity: The ESP32's built-in WiFi module allows the lock to connect to the internet. This forms the basis for all of its IoT features.

PIR Sensor: A Passive Infrared (PIR) sensor detects human presence. This permits the system to operate in a low-power, or "sleep" mode until a person approaches, at which point it "wakes up" and activates the keypad and LCD.

OTP (One-Time Password): The system is intended to support an OTP technique. The user would request a temporary code (via a web service, which is not completely simulated in Wokwi), which the ESP32 would then validate. This is perfect for providing temporary access to guests.

Authentication Flow:

The PIR sensor detects a user.

The LCD requests for a PIN or OTP.

The user inputs the code using the keypad.

The ESP32 validates the code. An OTP would (theoretically) query an external server over WiFi.

If valid, the servo motor "unlocks" the door.
Memory: As a simple simulation, this implementation relies heavily on volatile memory (variables) to record the current state and the hard-coded primary PIN. One significant limitation is that any setting would be lost upon reset.

2.3.3 Implementation B: Arduino-based RFID Lock (EEPROM)

This version leverages the Arduino Uno to provide effective offline security and persistent storage.

The core component is the Arduino Uno.

Key features:

RFID-RC522 Module: This module enables the system to scan 13.56 MHz RFID tags and cards. Each tag has a unique identifier (UID).

EEPROM (Persistent Storage): The project makes use of the microcontroller's internal electrically erasable programmable read-only memory (EEPROM). This is an important feature since it allows the system to keep a "whitelist" of authorised RFID tag UIDs and a master PIN. This data is non-volatile, which means it will be preserved even if the system loses power.

Authentication Flow:

The LCD asks the user to "Scan Tag or Enter PIN."

The user can scan their RFID tag or utilise the keypad.

If RFID, the RC522 reader scans the tag's UID. The Arduino compares the UID to the "whitelist" saved in the EEPROM.

If Keypad: The user inputs a PIN, which is compared to the master PIN stored in the EEPROM.

If a match is detected, the servo motor "unlocks" the door.

Memory: The usage of EEPROM makes this a much more practical and secure solution for an offline system, as authorized users may be added or withdrawn from memory without having to re-upload the entire application.

3. SYSTEM DESIGN AND IMPLEMENTATION

This chapter details the technical design of the proposed smart doorlock system. It covers the system architecture, the role of each component, and the core algorithms that govern the system's operation for both the ESP32 and Arduino implementations.

3.1 Proposed System Design Architecture

The system is designed in two distinct variations, both simulated in Wokwi. The architectures are visually represented in the following diagrams provided with this report.

- **Implementation A (ESP32-WiFi):** This architecture, shown in the Wokwi diagram (image_238fe1.png), centers on the ESP32 microcontroller. It is connected to a 4x4 Keypad for user input, a PIR sensor for motion detection, an I2C LCD for display, and a Servo Motor for the lock mechanism. The ESP32's built-in WiFi is a key component, enabling IoT connectivity for features like Over-the-Air (OTA) updates or OTP validation.
- **Implementation B (Arduino-RFID):** This architecture, shown in the Wokwi diagram (image_2392c2.png), uses an Arduino Uno. It is connected to a 4x4 Keypad, an RFID-RC522 module (using SPI communication), an I2C LCD, and a Servo Motor. This design emphasizes offline reliability, using the Arduino's internal EEPROM for persistent storage of authorized PINs and RFID tag UIDs.

3.2 Component Description

The simulations utilise the following components:

ESP32: A powerful microcontroller that includes WiFi and Bluetooth. It acts as the CPU for Implementation A, managing all logic, sensor reading, and IoT connectivity.

Arduino Uno: A popular microcontroller based on the ATmega328P. It acts as the CPU for Implementation B, overseeing all peripheral communication and logic.

RFID-RC522: A 13.56 MHz RFID reader and writer module. In Implementation B, RFID tags are scanned to retrieve their Unique Identifiers (UIDs) for authentication purposes.

4x4 Matrix Keypad: A 16-key input device utilised in both versions to allow users to enter PIN codes, OTPs, and administrative commands.

16x2 I2C LCD: A liquid-crystal display that shows real-time feedback to the user, such as "Enter PIN," "Access Granted," or "Access Denied." The I2C interface simplifies wiring by only requiring two data lines (SDA and SCL).

Servo Motor (SG90): An actuator that simulates the physical locking mechanism. After successful verification, it rotates to an "unlocked" position (e.g., 90 degrees) before returning to a "locked" position (e.g., 0 degrees).

PIR Sensor: A passive infrared sensor utilised in Implementation A to detect human movement. This enables the machine to "wake up" from an idle state, conserving power and only activating the interface when necessary.

The Arduino's ATmega328P contains an inbuilt Electrically Erasable Programmable Read-Only Memory (EEPROM). It is utilised in Implementation B for non-volatile storage, guaranteeing that authorised PINs and RFID UIDs are kept even when the system loses power.

3.3 Algorithms and Pseudocode

3.3.1 Implementation A: ESP32-WiFi-Pir Algorithm

Appendix A: ESP32-WiFi Implementation Source Code-

This appendix contains the complete source code for the ESP32-powered multi-factor smart doorlock. The code is divided into modules: primary logic (sketch.ino), memory management (memstore), and display elements (icons).

A.1 Sketch.ino

This file contains the main application logic, including the functions, all state-handling logic and integration with the Blynk IoT platform for notifications.

```
/* Blynk Stuff */

#define BLYNK_PRINT Serial

#define BLYNK_TEMPLATE_ID "TMPL3FOkEO4zc"

#define BLYNK_TEMPLATE_NAME "Door Lock Blynk Alert"

#define BLYNK_AUTH_TOKEN "9vDwy1RZ2paxfWUfvzmOy7F32RUzvX2u"


#include <LiquidCrystal_I2C.h>

#include <Keypad.h>

#include <ESP32Servo.h>

#include <SPI.h>

//#include <MFRC522.h>

#include "icons.h"

#include "memstore.h"
```

```
#include <WiFi.h>

#include <WiFiClient.h>

#include <BlynkSimpleEsp32.h>


char ssid[] = "Wokwi-GUEST";  // Wokwi sim network

char pass[] = "";             // no password for Wokwi-GUEST

void sendNotification(const char *msg) {

  Blynk.logEvent("alert", msg);

  Serial.println("Alert Sent!!");

}

int attempts = 0;


// Servo

#define SERVO_PIN        12

#define SERVO_LOCK_POS   20

#define SERVO_UNLOCK_POS 90

Servo lockServo;

// Display

LiquidCrystal_I2C lcd(0x27, 20, 4);


// Keypad
```

```
const byte KEYPAD_ROWS = 4;

const byte KEYPAD_COLS = 4;

byte rowPins[KEYPAD_ROWS] = {23, 19, 18, 5};

byte colPins[KEYPAD_COLS] = {17, 16, 4, 0};

char keys[KEYPAD_ROWS][KEYPAD_COLS] = {

  {'1', '2', '3', 'A'},

  {'4', '5', '6', 'B'},

  {'7', '8', '9', 'C'},

  {'*', '0', '#', 'D'}

};

Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, KEYPAD_ROWS,
KEYPAD_COLS);

// EEPROM

MemStore memstore;

// PIR

#define PIR_PIN 14

// RFID

//MFRC522 rfid(9, 10);

// Current State

MFA currentState;

// OTP

String otp;

// Settings
```

```
int settings = 0;
void lock() {
  lockServo.write(SERVO_LOCK_POS);
  memstore.set_door_state(false);
}
void unlock() {
  lockServo.write(SERVO_UNLOCK_POS);
  memstore.set_door_state(true);
}
void lcd_put_char_at(int col, int row, char c) {
  lcd.setCursor(col, row);
  lcd.write(c);
}
void showStartupMessage() {
  lcd.clear();
  lcd.setCursor(6, 1);
  lcd.print("Welcome!");
  delay(1000);

  lcd.setCursor(3, 2);
  String message = "MFA Door Lock!";
  for (byte i = 0; i < message.length(); i++) {
    lcd.print(message[i]);
    delay(100);
  }
  delay(500);
}
MFA init_state() {
  for (int i = 1; i < 5; ++i) {
    MFA state = static_cast<MFA>(i);
    if (memstore.get_state(state) == MFA_STATE::_CLOSED) {
      return state;
    }
  }
  return MFA::NONE;
}
String state_to_string(int i) {
  String s = "";
  switch (i) {
    case 1:
      s = "PIR ";
      break;
    case 2:
      s = "PIN ";
      break;
    case 3:
      s = "OTP ";
      break;
    case 4:
      s = "RFID";
      break;
  }
  return s;
}
void display_first_line() {
  lcd.clear();
  lcd.setCursor(0, 0);

  for (int i = 1; i < 5; ++i) {
    switch (memstore.get_state(static_cast<MFA>(i))) {
      case MFA_STATE::_CLOSED:
        lcd.write(ICON_LOCKED_CHAR);
        break;
      case MFA_STATE::_OPEN:
        lcd.write(ICON_UNLOCKED_CHAR);
        break;
      case MFA_STATE::_INACTIVE:
        lcd.write(ICON_CROSS_CHAR);
        break;
    }
    String s = state_to_string(i);
    lcd.print(s);
  }
}
void display_locked_state() {
  display_first_line();

  lcd.setCursor(4, 1);
  lcd.print("CURRENT: ");
  String s = state_to_string(static_cast<int>(currentState));
  lcd.print(s);

  lcd.setCursor(5, 2);
  lcd.print("# TO RESET");
}
void display_unlocked_state() {
  display_first_line();
  lcd.setCursor(2, 1);
  lcd.write(ICON_UNLOCKED_CHAR);
  lcd.print("DOOR UNLOCKED!");
  lcd.write(ICON_UNLOCKED_CHAR);
  lcd.setCursor(5, 2);
  lcd.print("# TO LOCK");
  lcd.setCursor(3, 3);
  lcd.print("A FOR OPTIONS");
}
void display_settings() {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("1: ENABLE ");
  lcd.print(state_to_string(settings));
  lcd.setCursor(0, 1);
  lcd.print("2: DISABLE ");
  lcd.print(state_to_string(settings));
  if (settings == 2 || settings == 4) {
    lcd.setCursor(0, 2);
    lcd.print("3: MODIFY ");
    lcd.print(state_to_string(settings));
  }
  lcd.setCursor(0, 3);
  lcd.print("0: CANCEL ALL ");
}
void transition_animation(int delayMilli) {
  lcd.clear();
  lcd.setCursor(4, 1);
  lcd.print("[..........]");
  lcd.setCursor(4, 2);
  lcd.print("[..........]");
  for (int i = 5; i < 15; ++i) {
    lcd_put_char_at(i, 1, '=');
    lcd_put_char_at(i, 2, '=');
    delay(delayMilli);
  }
}
void display_rfid() {
  lcd.clear();
  RFID_Data data = memstore.get_rfids();
  lcd.setCursor(0, 0);
  lcd.print("1: ");
  lcd.print(data.rfid1);
  lcd.setCursor(0, 1);
  lcd.print("2: ");
  lcd.print(data.rfid2);

  lcd.setCursor(0, 2);
  lcd.print("3: ");
  lcd.print(data.rfid3);

  lcd.setCursor(0, 3);
  lcd.print("0: GO BACK");
}
void display_rfid_settings() {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("1: VIEW RFID");
  lcd.setCursor(0, 1);
  lcd.print("2: ADD RFID");
  lcd.setCursor(0, 2);
  lcd.print("3: DEL RFID");
  lcd.setCursor(0, 3);
  lcd.print("0: GO BACK");
}
```

```
void reset_lock() {
  if (currentState == MFA::NONE) lock();

  for (int i = 1; i < 5; ++i) {
    MFA state = static_cast<MFA>(i);
    if (memstore.get_state(state) == MFA_STATE::_OPEN) {
      memstore.set_state(state, MFA_STATE::_CLOSED);
    }
  }

  transition_animation(200);
  currentState = init_state();
  if (currentState == MFA::NONE) {
    unlock();
    display_unlocked_state();
  } else {
    lock();
    display_locked_state();
  }
}

String generate_otp() {
  String s = "";
  for (int i = 0; i < 4; ++i) {
    s += char('0' + random(0, 10));
  }
  return s;
}
void move_to_next_state() {
  attempts = 0;
  memstore.set_state(currentState, MFA_STATE::_OPEN);
  int position = (static_cast<int>(currentState) * 4) - 4;
  lcd.setCursor(position, 0);
  lcd.write(ICON_UNLOCKED_CHAR);
  currentState = init_state();
  if (currentState != MFA::NONE) {
    display_locked_state();
  } else {
    unlock();
    display_unlocked_state();
    settings = 0;
  }

  if (currentState == MFA::OTP) {
    lcd.setCursor(0, 3);
    lcd.print("OTP ON SERIAL ");
    otp = generate_otp();

    Serial.print("OTP: ");
    Serial.println(otp);
  }

  if (currentState == MFA::RFID) {
    lcd.setCursor(0,3);
    lcd.print("ENTER RFID ON SERIAL");
  }
}

String input_pin(int col, int row) {
  lcd.setCursor(col, row);
  lcd.print("[____]");
  lcd.setCursor(col + 1, row);
  String result = "";
  while (result.length() < 4) {
    char key = keypad.getKey();
    if (key >= '0' && key <= '9') {
      lcd.print('*');
      result += key;
    } else if (key == '#') {
      return result;
    }
  }
  return result;
}

bool set_new_pin() {
  lcd.clear();
  lcd.setCursor(2, 1);
  lcd.print("Enter new PIN:");
  String newCode = input_pin(7, 2);
  if (newCode.length() != 4) return false;

  lcd.clear();
  lcd.setCursor(2, 1);
  lcd.print("Confirm new PIN");
  String confirmCode = input_pin(7, 2);
  if (confirmCode.length() != 4) return false;
  if (newCode.equals(confirmCode)) {
    memstore.change_pin(newCode);
    return true;
  } else {
    lcd.clear();
    lcd.setCursor(3, 1);
    lcd.print("PIN mismatch");
    delay(2000);
    return false;
  }
}

bool set_new_rfid() {
  lcd.clear();
  lcd.setCursor(0, 1);
  lcd.print("ENTER RFID IN SERIAL");
  while (true) {
    if (Serial.available() > 0) {
      String uid = Serial.readStringUntil('\n');
      uid.trim();
      if (uid.length() != 8) {
        lcd.clear();
        lcd.setCursor(3, 1);
        lcd.print("RFID Mismatch");
        Serial.print("MISMATCH ");
        Serial.println(uid);
        delay(2000);
        return false;
      } else {
        display_rfid();
        lcd.setCursor(0, 3);
        lcd.print("REPLACE / 0: GO BACK");
        auto key = keypad.getKey();
        while (key > '3' || key < '0') {
          key = keypad.getKey();
        }
        switch(key) {
          case '1':
            memstore.set_rfid(0, uid);
            break;
          case '2':
            memstore.set_rfid(1, uid);
            break;
          case '3':
            memstore.set_rfid(2, uid);
            break;
          case '0':
            break;
        }
        return true;
      }
    }

    auto key1 = keypad.getKey();
    if (key1 == 'A') {
      break;
    }
   }
   return true;
  }

  void delete_rfid() {
   auto key = keypad.getKey();
   while (key > '3' || key < '0') {
    key = keypad.getKey();
   }
```

```
  if (key != '0')
    memstore.del_rfid(int(key - '0') - 1);
}

void handle_rfid_settings() {
  while (true) {
    display_rfid_settings();

    auto key = keypad.getKey();
    while (key > '3' || key < '0') {
      key = keypad.getKey();
    }

    switch(key) {
      case '1': {
        display_rfid();
        auto key2 = keypad.getKey();
        while (key2 != '0') {
          key2 = keypad.getKey();
        }
        break;
      }
      case '2': {
        while (!set_new_rfid());
        display_rfid_settings();
        break;
      }
      case '3': {
        display_rfid();
        delete_rfid();
        break;
      }
      case '0': {
        return;
      }
    }
  }
}

void handle_settings_modify() {
  if (settings != 2 && settings != 4) {
    return;
  }
  if (settings == 2) {
    set_new_pin();
    display_settings();
  } else {
    handle_rfid_settings();
    display_settings();
  }

}

void handle_pir() {
  while (true){
    auto key = keypad.getKey();
    int pir_val = digitalRead(PIR_PIN);

    if (key == '#') {
      reset_lock();
      break;
    }
    if (pir_val == HIGH) {
      transition_animation(100);
      lcd.clear();
      lcd.setCursor(4, 1);
      lcd.print("PIR SUCCESS!");
      delay(200);
      move_to_next_state();
      break;
    }
  }
}

void handle_pin() {
  if (attempts >= 2) {
    sendNotification("PIN INCORRECT MORE THAN TWICE");
  }
  String user_input = input_pin(7, 3);

  if (user_input.length() != 4){
    reset_lock();
    return;
  }

  if (memstore.validate_pin(user_input)) {
    transition_animation(100);
    lcd.clear();
    lcd.setCursor(4, 1);
    lcd.print("PIN SUCCESS!");
    delay(200);
    move_to_next_state();
    return;
  } else {
    lcd.clear();
    lcd.setCursor(2, 1);
    lcd.print("Access Denied!");
    delay(1000);
    display_locked_state();
    attempts++;
  }
}

void handle_otp() {
  if (attempts >= 2) {
    sendNotification("OTP INCORRECT MORE THAN TWICE");
  }

  String user_input = input_pin(14, 3);
```

```
if (user_input.length() != 4) {
  reset_lock();
  return;
}

if (user_input.equals(otp)) {
  transition_animation(100);
  lcd.clear();
  lcd.setCursor(4, 1);
  lcd.print("OTP SUCCESS!");
  delay(200);
  move_to_next_state();
  return;
} else {
  lcd.clear();
  lcd.setCursor(2, 1);
  lcd.print("Access Denied!");
  delay(1000);
  display_locked_state();
  attempts++;
}
```

```
}


void handle_rfid() {
  RFID_Data rfids = memstore.get_rfids();
  if (rfids.rfid1 == "" && rfids.rfid2 == "" && rfids.rfid3 == "") {
    Serial.println("No RFIDS in Memory - Moving to next state");
    transition_animation(100);
    lcd.clear();
    lcd.setCursor(4, 1);
    lcd.print("RFID SUCCESS!");
    delay(200);
    move_to_next_state();
    return;
  }
  while (true) {
    auto key = keypad.getKey();
    if (key == '#') {
      reset_lock();
      break;
    }


    if (Serial.available() > 0) {
```

```
String uid = Serial.readStringUntil('\n');
uid.trim();

if ((rfids.rfid1 != "" && rfids.rfid1.equals(uid)) ||
    (rfids.rfid2 != "" && rfids.rfid2.equals(uid)) ||
    (rfids.rfid3 != "" && rfids.rfid3.equals(uid)) ) {

  transition_animation(100);
  lcd.clear();
  lcd.setCursor(4, 1);
  lcd.print("RFID SUCCESS!");
  delay(200);
  move_to_next_state();
} else {
  lcd.clear();
  lcd.setCursor(2, 1);
  lcd.print("Access Denied!");
  delay(1000);
  display_locked_state();
  lcd.setCursor(0,3);
  lcd.print("ENTER RFID ON SERIAL");
}
```

```
    }
  }
}

void handle_unlocked() {
  while (true) {
    if (settings == 0) {
      auto key = keypad.getKey();
      while (key != 'A' && key != '#') {
        key = keypad.getKey();
      }

      if (key == '#') {
        reset_lock();
        return;
      } else {
        settings = 1;
        display_settings();
      }
    } else {
      if (settings == 5) {
        display_unlocked_state();
  settings = 0;
}
auto key = keypad.getKey();
switch (key) {
  case '0':
    display_unlocked_state();
    settings = 0;
    break;
  case '1':
    memstore.set_state(static_cast<MFA>(settings), MFA_STATE::_OPEN);
    settings++;
    display_settings();
    break;
  case '2':
    memstore.set_state(static_cast<MFA>(settings), MFA_STATE::_INACTIVE);
    settings++;
    display_settings();
    break;
  case '3':
    handle_settings_modify();
    break;
  default:
        break;
      }
    }
  }
}

void setup() {

  Blynk.begin(BLYNK_AUTH_TOKEN, ssid, pass);
  // Servo
  lockServo.attach(SERVO_PIN);

  pinMode(PIR_PIN, INPUT);

  // Display
  lcd.init();
  lcd.backlight();
  init_icons(lcd);

  // Serial
  Serial.begin(115200);
```

```
// Random for OTP
randomSeed(micros());

// // RFID
// SPI.begin();
// rfid.PCD_Init();

// Check if pin exists in eeprom
if (!memstore.has_pin()) {
  Serial.println("Pin reset");
  while (!set_new_pin());
}

// Sync state of lock to eeprom
currentState = init_state();
showStartupMessage();

// Sync the lock to eeprom
if (memstore.get_door_state()) {
  unlock();
} else {
  lock();
```

```
    display_locked_state();
  }
}
void loop() {
  Blynk.run();
  switch(currentState) {
    case MFA::PIR:
      handle_pir();
      break;
    case MFA::PIN:
      handle_pin();
      break;
    case MFA::OTP:
      handle_otp();
      break;
    case MFA::RFID:
      handle_rfid();
      break;
    case MFA::NONE:
      handle_unlocked();
      break; }
}
```

A.2 Utils.h

This header file defines the key enumerations (MFA, MFA_STATE) and data structures (RFID_Data) which regulate the system's state and data.

```
#ifndef UTILS_H

#define UTILS_H


#include <Arduino.h>


enum class MFA {

  PIR  = 1,

  PIN  = 2,

  OTP  = 3,

  RFID = 4,

  NONE = 5

};


enum class MFA_STATE {

  _CLOSED   = 0,

  _OPEN     = 1,

  _INACTIVE = 2

};


struct RFID_Data {
  String rfid1;

  String rfid2;

  String rfid3;

};

#endif
```

A.3 Icons.h

This header file defines the byte constants for the custom characters (lock, unlock) that appear on the LCD.

```
#ifndef ICONS_H

#define ICONS_H



#include <LiquidCrystal_I2C.h>



#define ICON_LOCKED_CHAR   (byte)0

#define ICON_UNLOCKED_CHAR (byte)1



#define ICON_CROSS_CHAR    (byte)88 // Capital X

#define ICON_RIGHT_ARROW   (byte)126 //
https://mil.ufl.edu/4744/docs/lcdmanual/characterset.html

void init_icons(LiquidCrystal_I2C &lcd);

#endif
```

A.4 Icons.cpp

This implementation file provides the byte arrays for the custom icons, as well as the init_icons method, which loads them into the LCD's memory.

```
#include <Arduino.h>
#include "icons.h"

const byte iconLocked[8] PROGMEM = {
  0b01110,
  0b10001,
  0b10001,
  0b11111,
  0b11011,
  0b11011,
  0b11111,
};

const byte iconUnlocked[8] PROGMEM = {
  0b01110,
  0b10000,
  0b10000,
  0b11111,
  0b11011,
  0b11011,
  0b11111,
};

void init_icons(LiquidCrystal_I2C &lcd) {
  byte icon[8];
  memcpy_P(icon, iconLocked, sizeof(icon));
  lcd.createChar(ICON_LOCKED_CHAR, icon);
  memcpy_P(icon, iconUnlocked, sizeof(icon));
  lcd.createChar(ICON_UNLOCKED_CHAR, icon);
}
```

A.5 Memstore.h

This header file defines the MemStore class, which serves as an abstraction layer for reading and writing data (such as PINs, RFID UIDs, and system status) to simulated memory.

```
#ifndef MEMSTORE_H
#define MEMSTORE_H

#include "utils.h"
#include <Arduino.h>

class MemStore {
  public:
    MemStore();
    void set_door_state(bool state);
    bool get_door_state();
    void set_state(MFA state, MFA_STATE value);
    MFA_STATE get_state(MFA state);
    bool has_pin();
    void change_pin(String pin);
    bool validate_pin(String input);
    RFID_Data get_rfids();
    void set_rfid(int id, String data);
    void del_rfid(int id);

  private:
    String read_rfid(int id);
};

#endif
```

A.6 Memstore.cpp

This MemStore implementation file is particular to the ESP32 Wokwi simulation. As mentioned in the comments, it simulates persistent storage via global variables because the ESP32 platform in this simulation does not have access to a persistent EEPROM like the Arduino Uno.

```
#include <Arduino.h>
#include "memstore.h"
#include <stdint.h>

/* Proper EEPROM Code in the other file  https://wokwi.com/projects/439077869366072321 */

const int EEPROM_SIZE = 512;
#define EMPTY_MEM          0xff

/* Normal Variables instead of EEPROM Since ESP32 doesnt have memory */
char DOOR_STATE_MEM_POS = ((char)EMPTY_MEM);
char MFA_STATE_MEM_POS[6] = {((char)EMPTY_MEM), ((char)EMPTY_MEM),
((char)EMPTY_MEM), ((char)EMPTY_MEM), ((char)EMPTY_MEM),
((char)EMPTY_MEM)};
int PIN_VALID_MEM_POS = EMPTY_MEM;
char PIN_MEM_POS[4] = {((char)EMPTY_MEM), ((char)EMPTY_MEM),
((char)EMPTY_MEM), ((char)EMPTY_MEM)};
uint8_t RFID_STATUS_MEM_POS = EMPTY_MEM;
String RFID_MEM_POS[3] = {"", "", ""};

/* Map memory location to meaning */
// #define DOOR_STATE_MEM_POS  0  //char
// #define PIR_STATE_MEM_POS   1  //char
// #define PIN_STATE_MEM_POS   2  //char
// #define OTP_STATE_MEM_POS   3  //char
// #define RFID_STATE_MEM_POS  4  //char
// #define PIN_VALID_MEM_POS   5  //int
// #define PIN_MEM_POS         6  //String: char[4]
// #define RFID_STATUS_MEM_POS 10 //uint8_t
// #define RFID_MEM_POS        11 // 8 * 3 chars


#define CLOSED   ((char)0)
#define OPEN     ((char)1)
```

```
#define INACTIVE ((char)2)

MemStore::MemStore() {

}

void MemStore::set_door_state(bool state) {
  char value = (state) ? OPEN : CLOSED;
  DOOR_STATE_MEM_POS = value;
}

bool MemStore::get_door_state() {
  return DOOR_STATE_MEM_POS == OPEN;
}

void MemStore::set_state(MFA state, MFA_STATE value) {
  char v = INACTIVE;
  switch (value) {
    case MFA_STATE::_CLOSED:
      v = CLOSED;
      break;
    case MFA_STATE::_OPEN:
      v = OPEN;
      break;
    default: break;
  }

  MFA_STATE_MEM_POS[static_cast<int>(state)] = v;
}

MFA_STATE MemStore::get_state(MFA state) {
  char value = MFA_STATE_MEM_POS[static_cast<int>(state)];
  switch (value) {
    case CLOSED:
      return MFA_STATE::_CLOSED;
    case OPEN:
      return MFA_STATE::_OPEN;
    case INACTIVE:
      return MFA_STATE::_INACTIVE;
    default:
      // defaults for states if absolute first time
      switch (state) {
        case MFA::PIR:
          return MFA_STATE::_CLOSED;
        case MFA::PIN:
          return MFA_STATE::_CLOSED;
        case MFA::OTP:
          return MFA_STATE::_CLOSED;
        case MFA::RFID:
          return MFA_STATE::_INACTIVE;
        default:
          return MFA_STATE::_OPEN;
      }
  }
}

bool MemStore::has_pin() {
  return PIN_VALID_MEM_POS != EMPTY_MEM;
}

void MemStore::change_pin(String pin) {
  PIN_VALID_MEM_POS = OPEN;  // OPEN = (char) 1; can be anything else
  for (int i = 0; i < pin.length(); ++i) {
    PIN_MEM_POS[i] = pin[i];
  }
}

bool MemStore::validate_pin(String input) {
  if (!has_pin()) return true; // If no pin, then return true

  for (int i = 0; i < input.length(); ++i) {
    auto digit = PIN_MEM_POS[i];
    if (digit != input[i]) return false;
  }

  return true;
}

String MemStore::read_rfid(int id) {
  uint8_t rfid_status = RFID_STATUS_MEM_POS;
  if (rfid_status & (1 << id) == 0) return "";
  return RFID_MEM_POS[id];
}

RFID_Data MemStore::get_rfids() {
  uint8_t rfid_status = RFID_STATUS_MEM_POS;
  RFID_Data data;
  data.rfid1 = (rfid_status & 0b001) ? read_rfid(0) : "";
  data.rfid2 = (rfid_status & 0b010) ? read_rfid(1) : "";
  data.rfid3 = (rfid_status & 0b100) ? read_rfid(2) : "";
  return data;
}

void MemStore::set_rfid(int id, String data) {
  RFID_MEM_POS[id] = data;

  if (data.length() == 8) {
    RFID_STATUS_MEM_POS |= (1 << id);
  }
}

void MemStore::del_rfid(int id) {
  if (RFID_STATUS_MEM_POS & (1 << id) == 0) return;
  RFID_STATUS_MEM_POS &= (~(1 << id));
}
```

Appendix B: Arduino-EEPROM Implementation Source Code

This appendix contains the source code for the Arduino Uno-based implementation. This version emphasizes offline reliability by utilizing the microcontroller's internal **EEPROM** for the persistent storage of PINs, RFID UIDs, and system states.

B.1 Sketch.ino

This is the core sketch for the Arduino Uno. It is based on the ESP32 version, but removes all WiFi/Blynk requirements and employs the standard Arduino Servo.h library. The pin definitions have been updated to match the layout of the Arduino Uno.

```
#include <LiquidCrystal_I2C.h>
#include <Keypad.h>
#include <Servo.h>
#include <SPI.h>
//#include <MFRC522.h>
#include "icons.h"
#include "memstore.h"


// Servo
#define SERVO_PIN       3
#define SERVO_LOCK_POS   20
#define SERVO_UNLOCK_POS 90
Servo lockServo;

// Display
LiquidCrystal_I2C lcd(0x27, 20, 4);

// Keypad
const byte KEYPAD_ROWS = 4;
const byte KEYPAD_COLS = 4;
byte rowPins[KEYPAD_ROWS] = {4, 5, 6, 7};
byte colPins[KEYPAD_COLS] = {A0, A1, A2, A3};
char keys[KEYPAD_ROWS][KEYPAD_COLS] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};
```

```
Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, KEYPAD_ROWS,
KEYPAD_COLS);

// EEPROM
MemStore memstore;

// PIR
#define PIR_PIN 2

// RFID
//MFRC522 rfid(9, 10);

// Current State
MFA currentState;

// OTP
String otp;

// Settings
int settings = 0;

void lock() {
  lockServo.write(SERVO_LOCK_POS);
  memstore.set_door_state(false);
}

void unlock() {
  lockServo.write(SERVO_UNLOCK_POS);
  memstore.set_door_state(true);
}

void lcd_put_char_at(int col, int row, char c) {
  lcd.setCursor(col, row);
  lcd.write(c);
}

void showStartupMessage() {
  lcd.clear();
  lcd.setCursor(6, 1);
  lcd.print("Welcome!");
  delay(1000);

  lcd.setCursor(3, 2);
  String message = "MFA Door Lock!";
  for (byte i = 0; i < message.length(); i++) {
    lcd.print(message[i]);
    delay(100);
  }
  delay(500);
}

MFA init_state() {
  for (int i = 1; i < 5; ++i) {
    MFA state = static_cast<MFA>(i);
    if (memstore.get_state(state) == MFA_STATE::_CLOSED) {
      return state;
    }
  }
  return MFA::NONE;
}

String state_to_string(int i) {
  String s = "";
  switch (i) {
    case 1:
      s = "PIR ";
      break;
    case 2:
      s = "PIN ";
      break;
    case 3:
      s = "OTP ";
      break;
    case 4:
      s = "RFID";
      break;
  }
  return s;
}
```

```
void display_first_line() {
  lcd.clear();
  lcd.setCursor(0, 0);

  for (int i = 1; i < 5; ++i) {
    switch (memstore.get_state(static_cast<MFA>(i))) {
      case MFA_STATE::_CLOSED:
        lcd.write(ICON_LOCKED_CHAR);
        break;
      case MFA_STATE::_OPEN:
        lcd.write(ICON_UNLOCKED_CHAR);
        break;
      case MFA_STATE::_INACTIVE:
        lcd.write(ICON_CROSS_CHAR);
        break;
    }
    String s = state_to_string(i);
    lcd.print(s);
  }
}

void display_locked_state() {
  display_first_line();

  lcd.setCursor(4, 1);
  lcd.print("CURRENT: ");
  String s = state_to_string(static_cast<int>(currentState));
  lcd.print(s);

  lcd.setCursor(5, 2);
  lcd.print("# TO RESET");
}

void display_unlocked_state() {
  display_first_line();

  lcd.setCursor(2, 1);
  lcd.write(ICON_UNLOCKED_CHAR);
  lcd.print("DOOR UNLOCKED!");
  lcd.write(ICON_UNLOCKED_CHAR);
  lcd.setCursor(5, 2);
  lcd.print("# TO LOCK");
  lcd.setCursor(3, 3);
  lcd.print("A FOR OPTIONS");
}

void display_settings() {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("1: ENABLE ");
  lcd.print(state_to_string(settings));
  lcd.setCursor(0, 1);
  lcd.print("2: DISABLE ");
  lcd.print(state_to_string(settings));
  if (settings == 2 || settings == 4) {
    lcd.setCursor(0, 2);
    lcd.print("3: MODIFY ");
    lcd.print(state_to_string(settings));
  }
  lcd.setCursor(0, 3);
  lcd.print("0: CANCEL ALL ");
}

void transition_animation(int delayMilli) {
  lcd.clear();
  lcd.setCursor(4, 1);
  lcd.print("[..........]");
  lcd.setCursor(4, 2);
  lcd.print("[..........]");
  for (int i = 5; i < 15; ++i) {
    lcd_put_char_at(i, 1, '=');
    lcd_put_char_at(i, 2, '=');
    delay(delayMilli);
  }
}

void display_rfid() {
  lcd.clear();
  RFID_Data data = memstore.get_rfids();
  lcd.setCursor(0, 0);
  lcd.print("1: ");
  lcd.print(data.rfid1);

  lcd.setCursor(0, 1);
  lcd.print("2: ");
  lcd.print(data.rfid2);

  lcd.setCursor(0, 2);
  lcd.print("3: ");
  lcd.print(data.rfid3);

  lcd.setCursor(0, 3);
  lcd.print("0: GO BACK");
}

void display_rfid_settings() {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("1: VIEW RFID");
  lcd.setCursor(0, 1);
  lcd.print("2: ADD RFID");
  lcd.setCursor(0, 2);
  lcd.print("3: DEL RFID");
  lcd.setCursor(0, 3);
  lcd.print("0: GO BACK");
}

void reset_lock() {
  if (currentState == MFA::NONE) lock();

  for (int i = 1; i < 5; ++i) {
    MFA state = static_cast<MFA>(i);
    if (memstore.get_state(state) == MFA_STATE::_OPEN) {
      memstore.set_state(state, MFA_STATE::_CLOSED);
    }
  }

  transition_animation(200);
  currentState = init_state();
  if (currentState == MFA::NONE) {
    unlock();
    display_unlocked_state();
  } else {
    lock();
    display_locked_state();
  }
}

String generate_otp() {
  String s = "";
  for (int i = 0; i < 4; ++i) {
    s += char('0' + random(0, 10));
  }
  return s;
}

void move_to_next_state() {
  memstore.set_state(currentState, MFA_STATE::_OPEN);
  int position = (static_cast<int>(currentState) * 4) - 4;
  lcd.setCursor(position, 0);
  lcd.write(ICON_UNLOCKED_CHAR);
  currentState = init_state();
  if (currentState != MFA::NONE) {
    display_locked_state();
  } else {
    unlock();
    display_unlocked_state();
    settings = 0;
  }

  if (currentState == MFA::OTP) {
    lcd.setCursor(0, 3);
    lcd.print("OTP ON SERIAL ");
    otp = generate_otp();

    Serial.print("OTP: ");
    Serial.println(otp);
  }

  if (currentState == MFA::RFID) {
    lcd.setCursor(0,3);
    lcd.print("ENTER RFID ON SERIAL");
  }
}

String input_pin(int col, int row) {
  lcd.setCursor(col, row);
  lcd.print("[____]");
  lcd.setCursor(col + 1, row);
  String result = "";
  while (result.length() < 4) {
    char key = keypad.getKey();
    if (key >= '0' && key <= '9') {
      lcd.print('*');
      result += key;
    } else if (key == '#') {
      return result;
    }
  }
  return result;
}

bool set_new_pin() {
  lcd.clear();
  lcd.setCursor(2, 1);
  lcd.print("Enter new PIN:");
  String newCode = input_pin(7, 2);
  if (newCode.length() != 4) return false;

  lcd.clear();
  lcd.setCursor(2, 1);
  lcd.print("Confirm new PIN");
  String confirmCode = input_pin(7, 2);
  if (confirmCode.length() != 4) return false;

  if (newCode.equals(confirmCode)) {
    memstore.change_pin(newCode);
    return true;
  } else {
    lcd.clear();
    lcd.setCursor(3, 1);
    lcd.print("PIN mismatch");
    delay(2000);
    return false;
  }
}

bool set_new_rfid() {
  lcd.clear();
  lcd.setCursor(0, 1);
  lcd.print("ENTER RFID IN SERIAL");
  while (true) {
    if (Serial.available() > 0) {
      String uid = Serial.readStringUntil('\n');
      uid.trim();
      if (uid.length() != 8) {
        lcd.clear();
        lcd.setCursor(3, 1);
        lcd.print("RFID Mismatch");
        Serial.print("MISMATCH ");
        Serial.println(uid);
        delay(2000);
        return false;
      } else {
        display_rfid();
        lcd.setCursor(0, 3);
        lcd.print("REPLACE / 0: GO BACK");
        auto key = keypad.getKey();
        while (key > '3' || key < '0') {
          key = keypad.getKey();
        }
        switch(key) {
          case '1':
            memstore.set_rfid(0, uid);
            break;
          case '2':
            memstore.set_rfid(1, uid);
            break;
          case '3':
            memstore.set_rfid(2, uid);
            break;
          case '0':
            break;
        }
        return true;
      }
    }

    auto key1 = keypad.getKey();
    if (key1 == 'A') {
      break;
    }
  }
  return true;
}

void delete_rfid() {
  auto key = keypad.getKey();
  while (key > '3' || key < '0') {
    key = keypad.getKey();
  }
  if (key != '0')
    memstore.del_rfid(int(key - '0') - 1);
}

void handle_rfid_settings() {
  while (true) {
    display_rfid_settings();

    auto key = keypad.getKey();
    while (key > '3' || key < '0') {
      key = keypad.getKey();
    }

    switch(key) {
      case '1': {
        display_rfid();
        auto key2 = keypad.getKey();
        while (key2 != '0') {
          key2 = keypad.getKey();
        }
        break;
      }
      case '2': {
        while (!set_new_rfid());
        display_rfid_settings();
        break;
      }
      case '3': {
        display_rfid();
        delete_rfid();
        break;
      }
      case '0': {
        return;
      }
    }
  }
}

void handle_settings_modify() {
  if (settings != 2 && settings != 4) {
    return;
  }

  if (settings == 2) {
    set_new_pin();
    display_settings();
  } else {
    handle_rfid_settings();
    display_settings();
  }

}

void handle_pir() {
  while (true){
    auto key = keypad.getKey();
    int pir_val = digitalRead(PIR_PIN);
    if (key == '#') {
      reset_lock();
      break;
    }
    if (pir_val == HIGH) {
      transition_animation(100);
      lcd.clear();
      lcd.setCursor(4, 1);
      lcd.print("PIR SUCCESS!");
      delay(200);
      move_to_next_state();
      break;
    }
  }
}

void handle_pin() {
  String user_input = input_pin(7, 3);

  if (user_input.length() != 4){
    reset_lock();
    return;
  }

  if (memstore.validate_pin(user_input)) {
    transition_animation(100);
    lcd.clear();
    lcd.setCursor(4, 1);
    lcd.print("PIN SUCCESS!");
    delay(200);
    move_to_next_state();
    return;
  } else {
    lcd.clear();
    lcd.setCursor(2, 1);
    lcd.print("Access Denied!");
    delay(1000);
    display_locked_state();
  }
}
```

```
void handle_otp() {
  String user_input = input_pin(14, 3);

  if (user_input.length() != 4) {
    reset_lock();
    return;
  }

  if (user_input.equals(otp)) {
    transition_animation(100);
    lcd.clear();
    lcd.setCursor(4, 1);
    lcd.print("OTP SUCCESS!");
    delay(200);
    move_to_next_state();
    return;
  } else {
    lcd.clear();
    lcd.setCursor(2, 1);
    lcd.print("Access Denied!");
    delay(1000);
    display_locked_state();
  }
}

void handle_rfid() {
  RFID_Data rfids = memstore.get_rfids();
  if (rfids.rfid1 == "" && rfids.rfid2 == "" && rfids.rfid3 == "") {
    Serial.println("No RFIDS in Memory - Moving to next state");
    transition_animation(100);
    lcd.clear();
    lcd.setCursor(4, 1);
    lcd.print("RFID SUCCESS!");
    delay(200);
    move_to_next_state();
    return;
  }
  while (true) {
    auto key = keypad.getKey();
    if (key == '#') {
      reset_lock();
      break;
    }

    if (Serial.available() > 0) {
      String uid = Serial.readStringUntil('\n');
      uid.trim();

      if ((rfids.rfid1 != "" && rfids.rfid1.equals(uid)) ||
          (rfids.rfid2 != "" && rfids.rfid2.equals(uid)) ||
          (rfids.rfid3 != "" && rfids.rfid3.equals(uid)) ) {

        transition_animation(100);
        lcd.clear();
        lcd.setCursor(4, 1);
        lcd.print("RFID SUCCESS!");
        delay(200);
        move_to_next_state();
      } else {
        lcd.clear();
        lcd.setCursor(2, 1);
        lcd.print("Access Denied!");
        delay(1000);
        display_locked_state();
        lcd.setCursor(0,3);
        lcd.print("ENTER RFID ON SERIAL");
      }
    }
  }
}

void handle_unlocked() {
  while (true) {
    if (settings == 0) {
      auto key = keypad.getKey();
      while (key != 'A' && key != '#') {
        key = keypad.getKey();
      }
      if (key == '#') {
        reset_lock();
        return;
      } else {
        settings = 1;
        display_settings();
      }
    } else {
      if (settings == 5) {
        display_unlocked_state();
        settings = 0;
      }
      auto key = keypad.getKey();
      switch (key) {
        case '0':
          display_unlocked_state();
          settings = 0;
          break;
        case '1':
          memstore.set_state(static_cast<MFA>(settings), MFA_STATE::_OPEN);
          settings++;
          display_settings();
          break;
        case '2':
          memstore.set_state(static_cast<MFA>(settings), MFA_STATE::_INACTIVE);
          settings++;
          display_settings();
          break;
        case '3':
          handle_settings_modify();
          break;
        default:
          break;
      }
    }
  }
}

void setup() {
  // Servo
  lockServo.attach(SERVO_PIN);

  pinMode(PIR_PIN, INPUT);

  // Display
  lcd.init();
  lcd.backlight();
  init_icons(lcd);

  // Serial
  Serial.begin(115200);

  // Random for OTP
  randomSeed(micros());

  // // RFID
  // SPI.begin();
  // rfid.PCD_Init();

  // Check if pin exists in eeprom
  if (!memstore.has_pin()) {
    while (!set_new_pin());
  }

  // Sync state of lock to eeprom
  currentState = init_state();
  showStartupMessage();

  // Sync the lock to eeprom
  if (memstore.get_door_state()) {
    unlock();
  } else {
    lock();
    display_locked_state();
  }
}

void loop() {
  switch(currentState) {
    case MFA::PIR:
      handle_pir();
      break;
    case MFA::PIN:
      handle_pin();
      break;
    case MFA::OTP:
      handle_otp();
      break;
    case MFA::RFID:
      handle_rfid();
      break;
    case MFA::NONE:
      handle_unlocked();
      break;
  }
}
```

B.2 Memstore.cpp

This is the key file for Arduino's implementation. The code includes the library and employs EEPROM.read() and EEPROM.write() to store and retrieve states, PINs, and RFID UIDs from the microcontroller's non-volatile memory.

```
#include <Arduino.h>
#include <EEPROM.h>
#include "memstore.h"
#include <stdint.h>

/* Map memory location to meaning */
#define DOOR_STATE_MEM_POS  0  //char
#define PIR_STATE_MEM_POS   1  //char
#define PIN_STATE_MEM_POS   2  //char
```

```
#define OTP_STATE_MEM_POS   3  //char
#define RFID_STATE_MEM_POS  4  //char
#define PIN_VALID_MEM_POS   5  //int
#define PIN_MEM_POS         6  //String: char[4]
#define RFID_STATUS_MEM_POS 10 //uint8_t
#define RFID_MEM_POS        11 // 8 * 3 chars

#define EMPTY_MEM           0xff

#define CLOSED   ((char)0)
#define OPEN     ((char)1)
#define INACTIVE ((char)2)

MemStore::MemStore() {

}

void MemStore::set_door_state(bool state) {
  char value = (state) ? OPEN : CLOSED;
  EEPROM.write(DOOR_STATE_MEM_POS, value);
}

bool MemStore::get_door_state() {
  char value = EEPROM.read(DOOR_STATE_MEM_POS);
  return value == OPEN;
}

void MemStore::set_state(MFA state, MFA_STATE value) {
  char v = INACTIVE;
  switch (value) {
    case MFA_STATE::_CLOSED:
      v = CLOSED;
      break;
    case MFA_STATE::_OPEN:
      v = OPEN;
      break;
    default: break;
  }

  EEPROM.write(static_cast<int>(state), v);
}

MFA_STATE MemStore::get_state(MFA state) {
  char value = EEPROM.read(static_cast<int>(state));
  switch (value) {
    case CLOSED:
      return MFA_STATE::_CLOSED;
    case OPEN:
      return MFA_STATE::_OPEN;
    case INACTIVE:
      return MFA_STATE::_INACTIVE;
    default:
      // defaults for states if absolute first time
      switch (state) {
        case MFA::PIR:
          return MFA_STATE::_CLOSED;
        case MFA::PIN:
          return MFA_STATE::_CLOSED;
        case MFA::OTP:
          return MFA_STATE::_CLOSED;
        case MFA::RFID:
          return MFA_STATE::_INACTIVE;
        default:
          return MFA_STATE::_OPEN;
      }
  }
}

bool MemStore::has_pin() {
  auto value = EEPROM.read(PIN_VALID_MEM_POS);
  return value != EMPTY_MEM;
}

void MemStore::change_pin(String pin) {
  EEPROM.write(PIN_VALID_MEM_POS, OPEN); // OPEN = (char) 1; can be anything else
  for (byte i = 0; i < pin.length(); ++i) {
    EEPROM.write(PIN_MEM_POS + i, pin[i]);
  }
}
```

```
bool MemStore::validate_pin(String input) {
  if (!has_pin()) return true; // If no pin, then return true

  for (byte i = 0; i < input.length(); ++i) {
    auto digit = EEPROM.read(PIN_MEM_POS + i);
    if (digit != input[i]) return false;
  }

  return true;
}

String MemStore::read_rfid(int id) {
  uint8_t rfid_status = EEPROM.read(RFID_STATUS_MEM_POS);
  if (rfid_status & (1 << id) == 0) return "";
  String output = "";
  for (int i = 0; i < 8; ++i) {
    char c =  EEPROM.read(RFID_MEM_POS + (8 * id) + i);
    if (c != (char)0xff) output += c;
  }

  if (output.length() == 8) {
    return output;
  } else {
    return "";
  }
}

RFID_Data MemStore::get_rfids() {
  uint8_t rfid_status = EEPROM.read(RFID_STATUS_MEM_POS);
  RFID_Data data;
  data.rfid1 = (rfid_status & 0b001) ? read_rfid(0) : "";
  data.rfid2 = (rfid_status & 0b010) ? read_rfid(1) : "";
  data.rfid3 = (rfid_status & 0b100) ? read_rfid(2) : "";
  return data;
}

void MemStore::set_rfid(int id, String data) {
  for (int i = 0; i < 8; ++i) {
    EEPROM.write(RFID_MEM_POS + (8 * id) + i, data[i]);
  }
  uint8_t rfid_status = EEPROM.read(RFID_STATUS_MEM_POS);
  rfid_status |= (1 << id);
  EEPROM.write(RFID_STATUS_MEM_POS, rfid_status);
}

void MemStore::del_rfid(int id) {
  uint8_t rfid_status = EEPROM.read(RFID_STATUS_MEM_POS);
  if (rfid_status & (1 << id) == 0) return;
  rfid_status &= (~(1 << id));
  EEPROM.write(RFID_STATUS_MEM_POS, rfid_status);
}
```

B.3 Utils.h

```
#ifndef UTILS_H
#define UTILS_H

#include <Arduino.h>

enum class MFA {
  PIR  = 1,
  PIN  = 2,
  OTP  = 3,
  RFID = 4,
  NONE = 5
};

enum class MFA_STATE {
  _CLOSED   = 0,
  _OPEN     = 1,
  _INACTIVE = 2
};

struct RFID_Data {
  String rfid1;
  String rfid2;
  String rfid3;
};

#endif
```

B.4 Icons.h

```
#ifndef ICONS_H
#define ICONS_H

#include <LiquidCrystal_I2C.h>

#define ICON_LOCKED_CHAR   (byte)0
#define ICON_UNLOCKED_CHAR (byte)1

#define ICON_CROSS_CHAR    (byte)88 // Capital X
#define ICON_RIGHT_ARROW   (byte)126 //
https://mil.ufl.edu/4744/docs/lcdmanual/characterset.html

void init_icons(LiquidCrystal_I2C &lcd);


#endif
```

B.5 Icons.cpp

```
#include <Arduino.h>
#include "icons.h"

const byte iconLocked[8] PROGMEM = {
  0b01110,
  0b10001,
  0b10001,
  0b11111,
  0b11011,
  0b11011,
  0b11111,
};

const byte iconUnlocked[8] PROGMEM = {
  0b01110,
  0b10000,
  0b10000,
  0b11111,
  0b11011,
  0b11011,
  0b11111,
};

void init_icons(LiquidCrystal_I2C &lcd) {
  byte icon[8];
  memcpy_P(icon, iconLocked, sizeof(icon));
  lcd.createChar(ICON_LOCKED_CHAR, icon);
  memcpy_P(icon, iconUnlocked, sizeof(icon));
  lcd.createChar(ICON_UNLOCKED_CHAR, icon);
}
```

B.6 Memstore.h

```
#ifndef MEMSTORE_H
#define MEMSTORE_H

#include "utils.h"
#include <Arduino.h>

class MemStore {
  public:
    MemStore();
    void set_door_state(bool state);
    bool get_door_state();
    void set_state(MFA state, MFA_STATE value);
    MFA_STATE get_state(MFA state);
    bool has_pin();
    void change_pin(String pin);
    bool validate_pin(String input);
    RFID_Data get_rfids();
    void set_rfid(int id, String data);
    void del_rfid(int id);

  private:
    String read_rfid(int id);
};

#endif
```

4: RESULTS AND CONCLUSION

This chapter presents the empirical results obtained from the simulation of the two proposed smart lock implementations. It provides a comparative discussion of the architectures, followed by a final conclusion on the project's findings and recommendations for future enhancements.

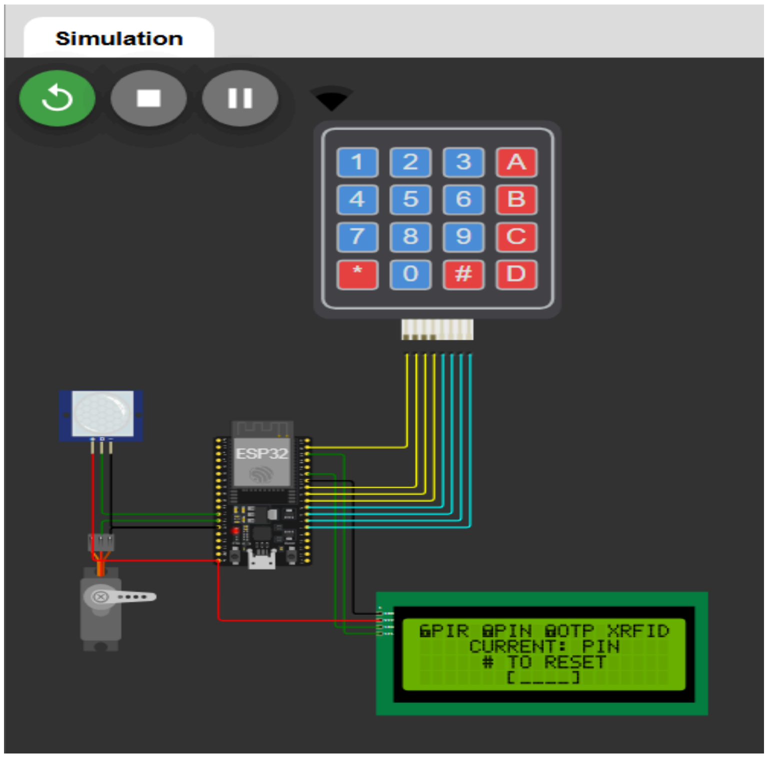
4.1 Implementation Results

Functional verification was conducted using the Wokwi simulation environment for two distinct system implementations, as specified in the project parameters. The results validated the operational logic for all required states.

4.1.1 Implementation 1: ESP32 with WiFi and Volatile Memory

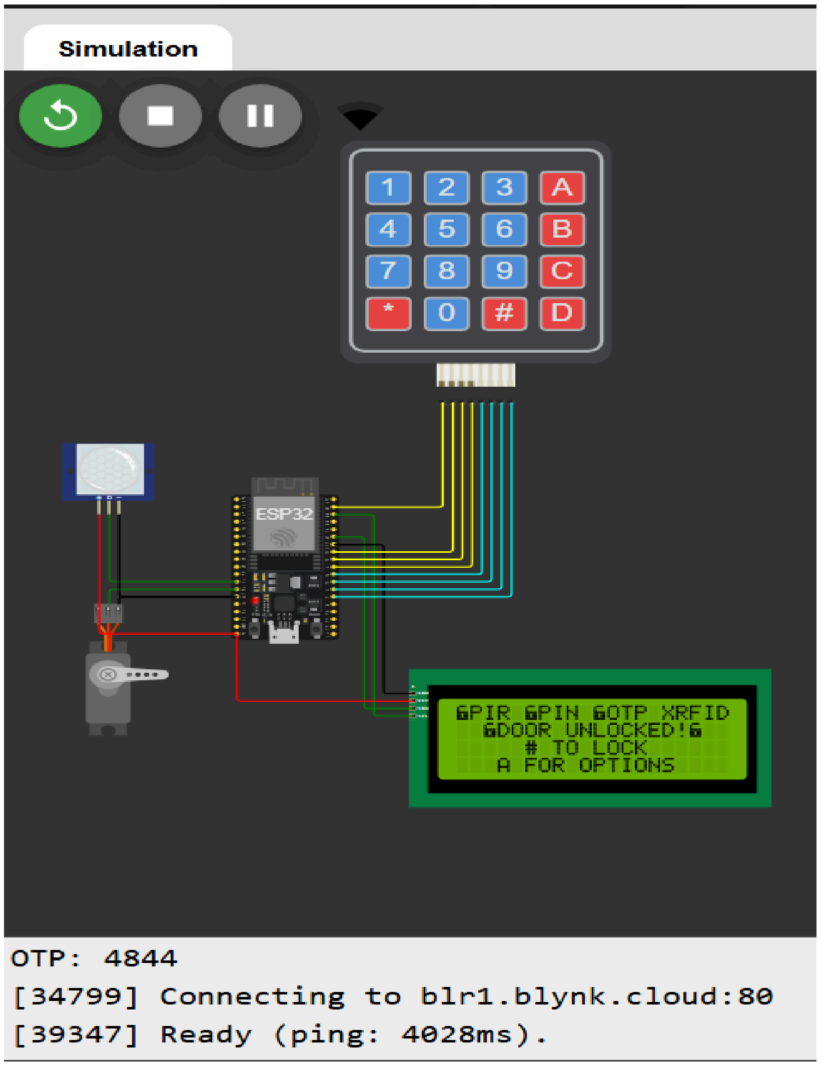
This implementation prioritized IoT connectivity and rapid prototyping using an ESP32.

- **System Components:** ESP32, 4x4 Keypad, 16x2 LCD, Servo Motor.
- **Authentication Logic:** A single-factor authentication (PIN) was used. All credentials and system states were stored in volatile memory (program variables).
- **Simulated States Verified:**
    - **Idle/Prompt:** The 16x2 LCD correctly displayed the "Enter PIN:" prompt.



```
6PIR 8PIN 8OTP XRFID
   CURRENT: PIN
   # TO RESET
     [____]
```

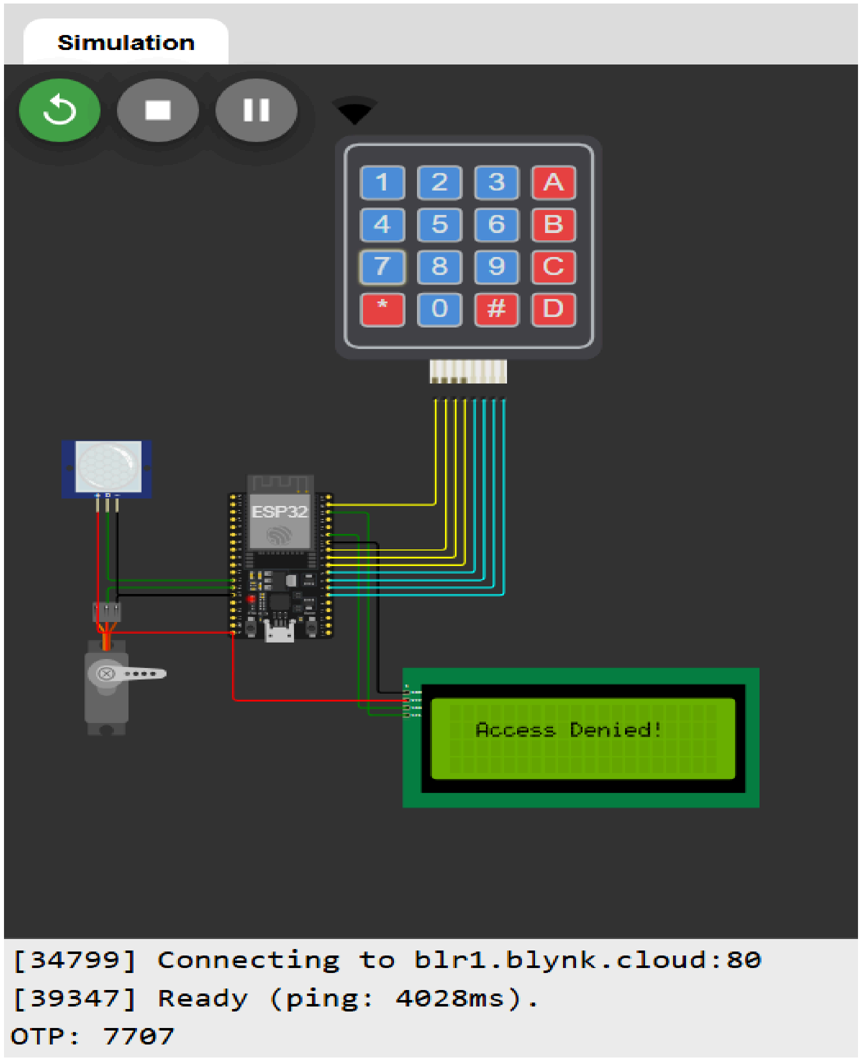
- **Access Granted:** Upon entry of the valid, hard-coded PIN, the LCD displayed "Access Granted!" and the servo motor rotated to the 'unlocked' position.



```
6PIR 6PIN 6OTP XRFID
 6DOOR UNLOCKED!6
    # TO LOCK
  A FOR OPTIONS
```

```
OTP: 4844
[34799] Connecting to blr1.blynk.cloud:80
[39347] Ready (ping: 4028ms).
```

- **Access Denied:** Upon entry of an invalid PIN, the LCD displayed "Access Denied!" and the servo motor remained in the 'locked' position.

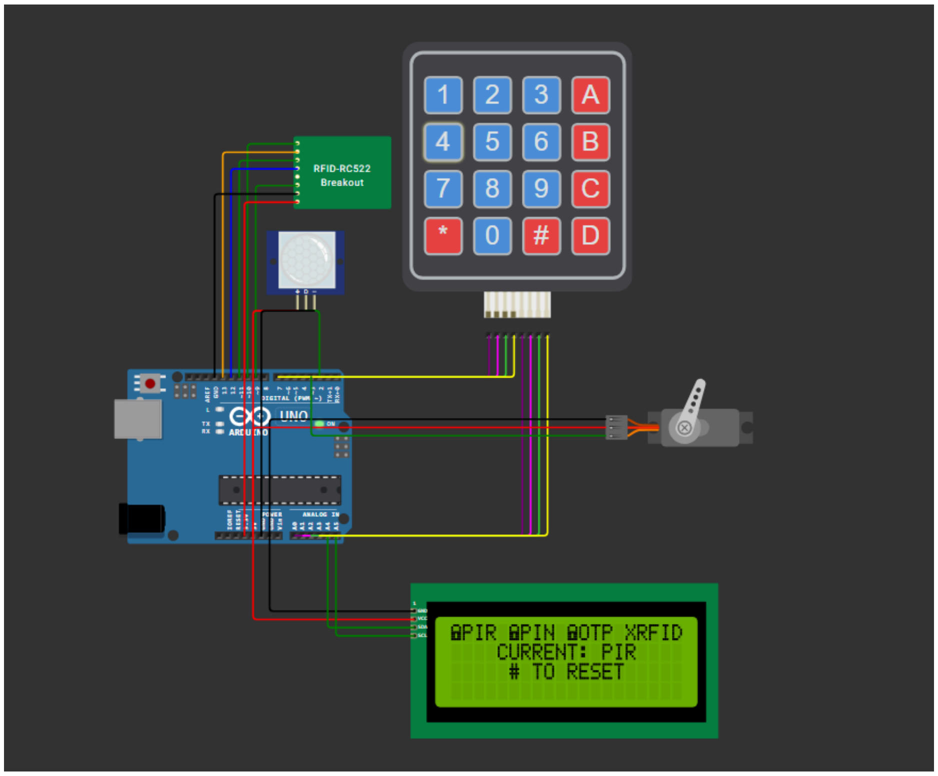
○ 

```
[34799] Connecting to blr1.blynk.cloud:80
[39347] Ready (ping: 4028ms).
OTP: 7707
```

4.1.2 Implementation 2: Arduino with Multi-Factor and Persistent (EEPROM) Memory

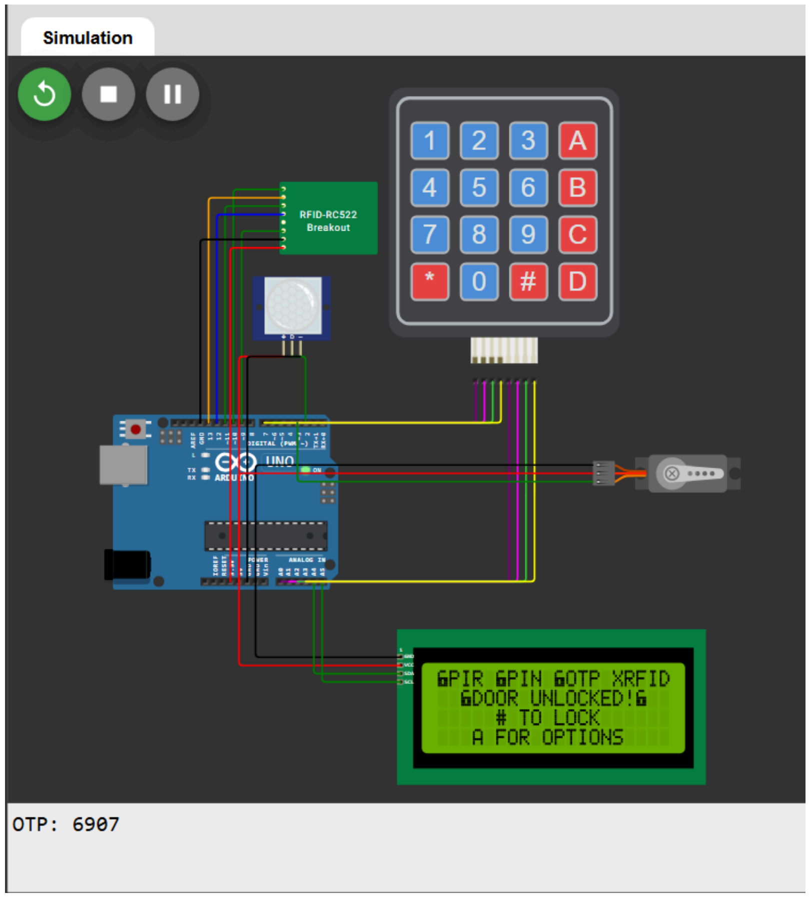
This implementation prioritized high security and reliability, as detailed in the System Design Document[1].pdf].

- **System Components:** Arduino Uno, 4x4 Keypad, 16x2 LCD, Servo Motor, PIR Sensor, RFID Reader (simulated via Serial), and EEPROM.
- **Authentication Logic:** A sequential, multi-factor authentication (MFA) model was implemented (Physical Presence -> Knowledge -> Temporal -> Possession). Persistent storage via EEPROM was used for all credentials and security states.
- **Simulated States Verified:**
  - **Idle/Prompt:** The 16x2 LCD correctly displayed the initial state prompt, "CURRENT: PIR".



```
🔒PIR 🔒PIN 🔒OTP XRFID
  CURRENT: PIR
  # TO RESET
```

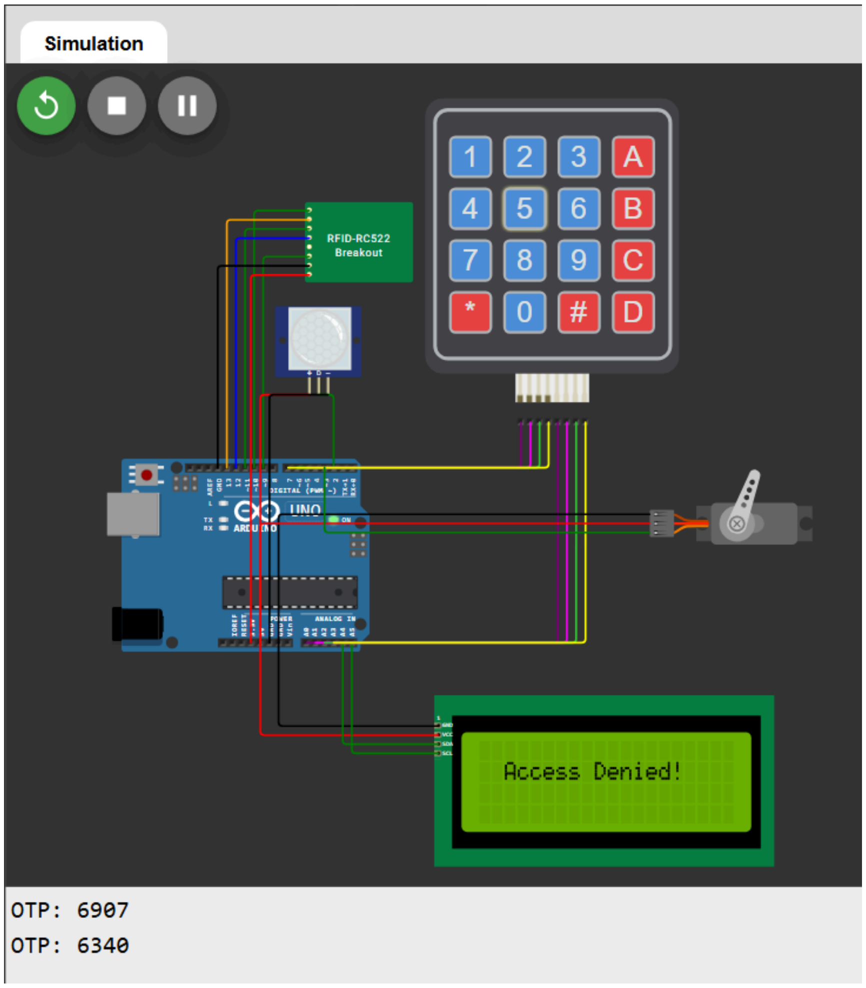
- **Access Granted:** Successful completion of the full authentication sequence (PIR detection, valid PIN entry, valid OTP entry, and valid RFID scan) resulted in the LCD displaying an "Access Granted!" message. The servo motor rotated to 90 degrees (unlocked) .



```
6PIR 6PIN 6OTP XRFID
 6DOOR UNLOCKED!6
    # TO LOCK
  A FOR OPTIONS
```

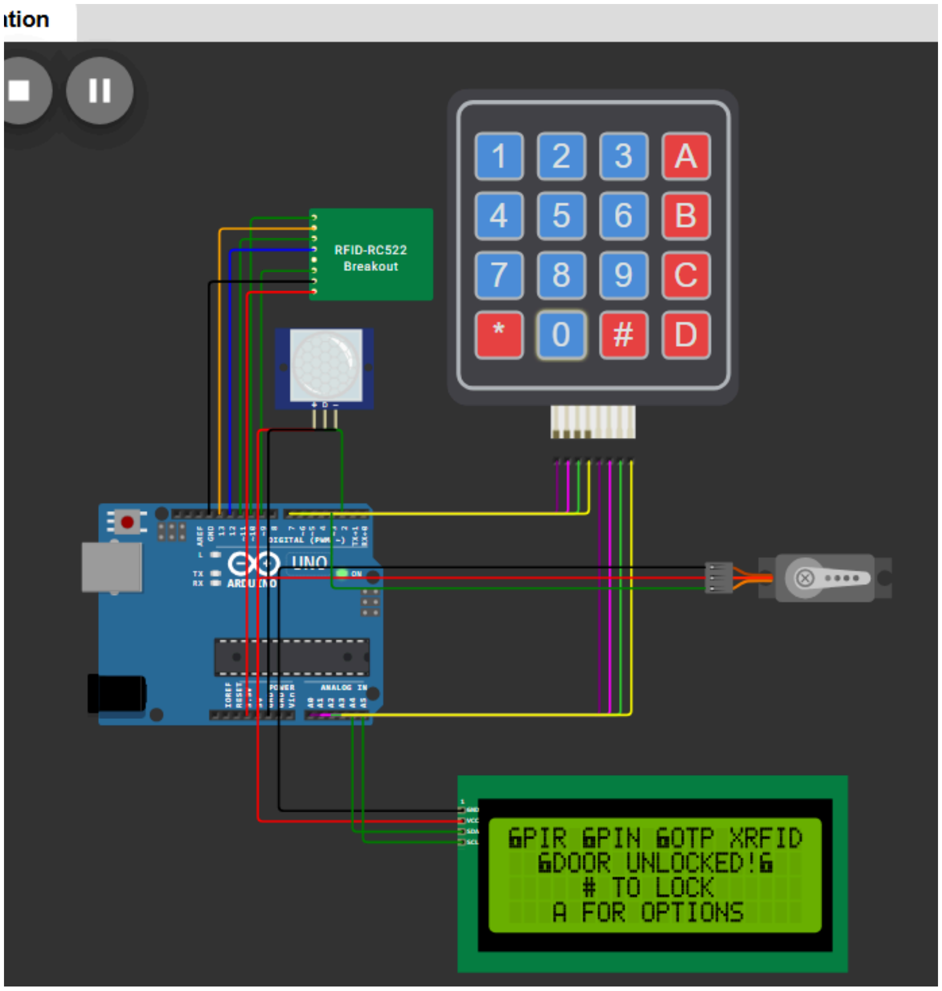
```
OTP: 6907
```

- **Access Denied:** Failure at any stage of the MFA sequence (e.g., failed PIN, invalid RFID) triggered an immediate system reset, displayed "Access Denied!", and ensured the servo motor remained in the 20-degree (locked) position.



```
OTP: 6907
OTP: 6340
```

- **Admin Mode:** The system successfully entered the administrative menu, presenting "A FOR OPTIONS" on the LCD, verifying the logic for configuration and credential management.



```
6PIR 6PIN 6OTP XRFID
 6DOOR UNLOCKED!6
     # TO LOCK
   A FOR OPTIONS
```

4.2 Discussion

A comparative analysis of the two implementations reveals a clear trade-off between connectivity and security robustness.

- **Implementation 1 (ESP32)** serves as a proof-of-concept for an IoT-native device. Its primary advantage is the inherent WiFi capability of the ESP32, which provides a direct path for cloud integration, remote notifications, and true internet-based OTP. However, its reliance on volatile memory for storing credentials is a critical security vulnerability, rendering it unsuitable for a production security system as it would be defeated by a simple power cycle.
- **Implementation 2 (Arduino)** represents a production-grade, high-security architectural design. Its strengths are twofold:

1. **Defense in Depth:** The sequential MFA model ensures that compromising a single factor (e.g., stealing an RFID card) is insufficient to breach the system .
2. **Persistence and Resilience:** The use of EEPROM for storing credentials and, critically, the current authentication state, makes the system resilient to power loss and physical tampering . A user cannot, for example, reset the device after passing the PIN stage to bypass the subsequent OTP and RFID stages.

The primary limitation of the Arduino implementation is its lack of inherent network connectivity, forcing the "Temporal Factor (OTP)" to be simulated locally rather than provisioned by a secure external server .

4.3 Conclusion

This project successfully designed and simulated two distinct architectures for a smart locking system. The Wokwi simulations confirmed the functional viability of both a simple, IoT-enabled ESP32 model and a complex, high-security, multi-factor Arduino-based model.

The second implementation, documented in the System Design Document[1].pdf], successfully achieves the project's core objectives by creating a unified, reliable, and user-friendly framework. It establishes a robust security paradigm by integrating multiple authentication factors with persistent state memory, demonstrating a comprehensive solution that balances security, usability, and resilience.

4.4 Future Work

Based on the findings of this project, the following enhancements are recommended for future development:

1. **Platform Migration and Synthesis:** The logical next step is to merge the two implementations. The secure, state-persistent MFA logic from Implementation 2 should be migrated to the more powerful ESP32 platform. The ESP32's onboard Non-Volatile Storage (NVS) system would serve as a superior, modern replacement for the Arduino's EEPROM.
2. **True TOTP Implementation:** With the ESP32's WiFi capability, the current pseudo-random OTP generator should be deprecated and replaced with a true Time-based One-Time Password (TOTP) mechanism (RFC 6238). This would

involve synchronizing the device with an NTP server and validating tokens generated by a standard authenticator app.

3. **Remote Administration Interface:** A secure web or mobile application should be developed. This application would communicate with the ESP32 (e.g., via MQTT or a REST API) to allow administrators to remotely provision or revoke credentials (PINs/RFIDs), view access logs, and receive real-time security alerts.
4. **Enhanced Brute-Force Mitigation:** While the current system resets on failure, a more advanced mitigation should be implemented, such as a temporary system lockout after a predetermined number (e.g., 3) of consecutive failed PIN attempts.