University of East London

EMBEDDED SYSTEMS MAJOR TASK

Submitted to:

Dr Sherif Hammad

Eng Ahmed Hassabou

Eng Hesham Salah

Submitted by:

Kareem Ahmed 23P0225

Ferass Ahmed 23P0304

Omar Osama 2300090

Youssef Maged 2301081

Mohammed Kamal 2301144

Ahmed Elian 23p0446

Ahmed Hesham 23P0167

Youssef Elnaggar 23P0341

Paula Hanna 23P0440

Ahmed Elmlahe 23P0035

Abdullah Ali 23P0158

Ahmed Sadek 23P0140

University of East London

Contents

1. Project Overview

This project implements a secure Door Locker system using two TM4C123-based ECUs:

- HMI_ECU for user interaction (LCD, keypad, potentiometer via ADC, RGB LED)
- Control_ECU for decision logic, EEPROM persistence, motor actuation, alarm/buzzer, and security lockout

The ECUs communicate over UART. The system supports: initial password setup, door open/close with auto-lock timeout, password change, incorrect-attempt lockout with alarm, and persistent storage for password and timeout.

2. System Overview

2.1 HMI_ECU (User side):

- Displays menus/messages via 16×2 LCD
- Gets user input via 4×4 Keypad
- Adjusts timeout using Potentiometer → ADC
- Indicates state using RGB LED
- Sends commands/data to Control_ECU via UART

2.2 Control_ECU (Control side):

- Validates passwords and enforces security rules
- Stores password + timeout in EEPROM
- Controls DC motor to lock/unlock
- Activates buzzer alarm for security failures
- Uses timers / SysTick for delays and sequences
- Receives commands via UART and responds with status

3. Hardware & Peripherals Implemented

3.1 HMI_ECU Drivers Delivered

- **LCD driver (4-bit mode):** lcd.c/h
- **Keypad scanning:** keypad.c/h
- **ADC for potentiometer:** adc.c/h
- **RGB LED control:** led.c/h
- **Digital IO abstraction:** dio.c/h
- **Timing:** systick.c/h

3.2 Control_ECU Drivers Delivered

- **UART2 communication:** uart.c/h
- **EEPROM persistence:** eeprom.c/h
- **Motor control + timing:** motor.c/h
- **Buzzer alarm:** buzzer.c/h
- **Application logic:** main.c (labeled Control_ECU)

4. Software Architecture (Layered Design)

4.1 MCAL (Microcontroller Abstraction Layer)

Low-level and driverlib-based peripherals:

- UART2 init and TX/RX: uart.c
- ADC acquisition: adc.c
- Timer base delays (SysTick): systick.c
- Digital IO services: dio.c

4.2 HAL (Hardware Abstraction Layer)

Hardware components built on MCAL:

- LCD module: lcd.c
- Keypad module: keypad.c
- LED module: led.c
- Motor module: motor.c
- Buzzer module: buzzer.c
- EEPROM module: eeprom.c

4.3 Application Layer

System workflow and command interpretation:

- Control_ECU logic: main.c (password logic, command parsing, sequence control)

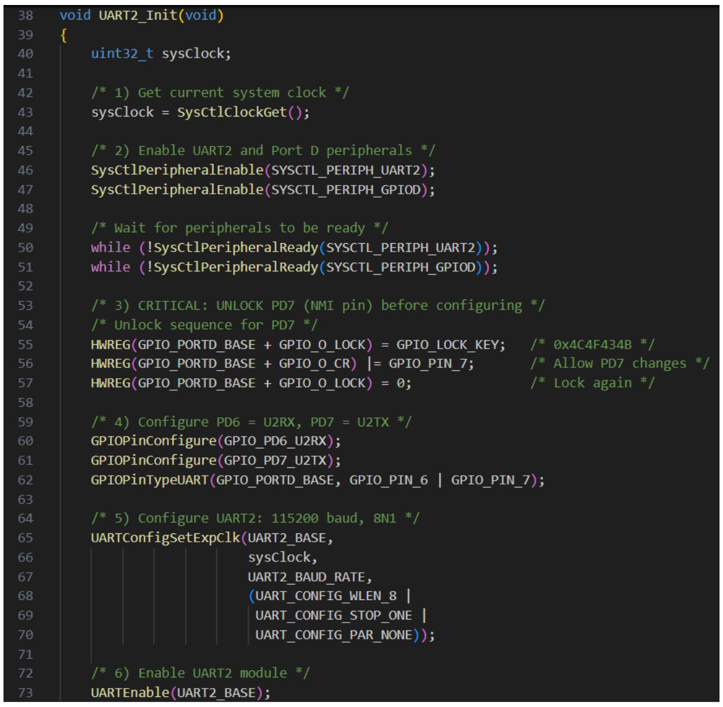
5. Inter-ECU Communication (UART Protocol Implementation)

The system uses UART2 for communication. The UART driver is implemented in uart.c/h.

5.1 UART2 Initialization

The UART code properly:

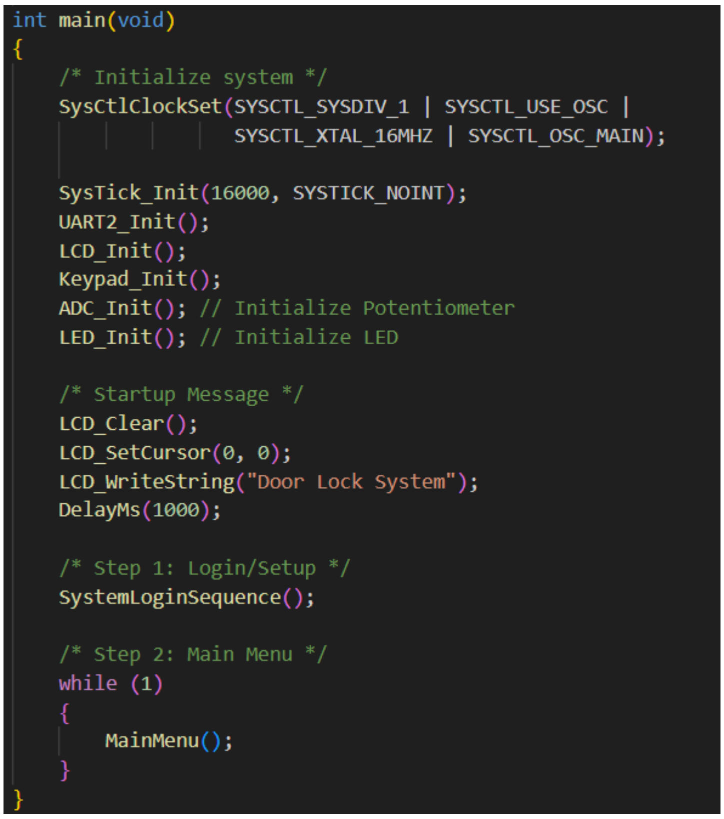
- Enables UART2 and Port D
- Unlocks PD7 (NMI pin) before use
- Configures PD6/PD7 for UART RX/TX
- Sets 115200 baud, 8N1



```c
void UART2_Init(void)
{
    uint32_t sysClock;

    /* 1) Get current system clock */
    sysClock = SysCtlClockGet();

    /* 2) Enable UART2 and Port D peripherals */
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART2);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);

    /* Wait for peripherals to be ready */
    while (!SysCtlPeripheralReady(SYSCTL_PERIPH_UART2));
    while (!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOD));

    /* 3) CRITICAL: UNLOCK PD7 (NMI pin) before configuring */
    /* Unlock sequence for PD7 */
    HWREG(GPIO_PORTD_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;   /* 0x4C4F434B */
    HWREG(GPIO_PORTD_BASE + GPIO_O_CR) |= GPIO_PIN_7;       /* Allow PD7 changes */
    HWREG(GPIO_PORTD_BASE + GPIO_O_LOCK) = 0;               /* Lock again */

    /* 4) Configure PD6 = U2RX, PD7 = U2TX */
    GPIOPinConfigure(GPIO_PD6_U2RX);
    GPIOPinConfigure(GPIO_PD7_U2TX);
    GPIOPinTypeUART(GPIO_PORTD_BASE, GPIO_PIN_6 | GPIO_PIN_7);

    /* 5) Configure UART2: 115200 baud, 8N1 */
    UARTConfigSetExpClk(UART2_BASE,
                        sysClock,
                        UART2_BAUD_RATE,
                        (UART_CONFIG_WLEN_8 |
                         UART_CONFIG_STOP_ONE |
                         UART_CONFIG_PAR_NONE));

    /* 6) Enable UART2 module */
    UARTEnable(UART2_BASE);
```

5.2 Control_ECU UART Command Handling

The Control_ECU continuously reads UART characters into a buffer and processes complete commands (terminated by a delimiter pattern in the application).


```c
int main(void)
{
    /* Initialize system */
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
                   SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);

    SysTick_Init(16000, SYSTICK_NOINT);
    UART2_Init();
    LCD_Init();
    Keypad_Init();
    ADC_Init(); // Initialize Potentiometer
    LED_Init(); // Initialize LED

    /* Startup Message */
    LCD_Clear();
    LCD_SetCursor(0, 0);
    LCD_WriteString("Door Lock System");
    DelayMs(1000);

    /* Step 1: Login/Setup */
    SystemLoginSequence();

    /* Step 2: Main Menu */
    while (1)
    {
        MainMenu();
    }
}
```

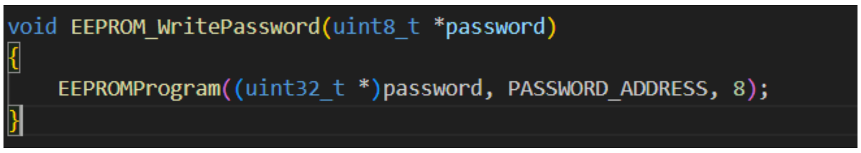
This structure makes it easy to log, test, and validate communication sequences.

6. EEPROM Persistence (Password + Timeout)

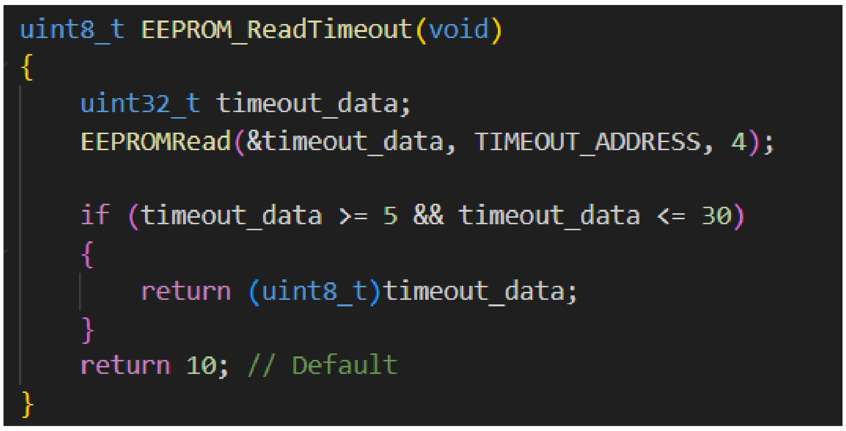
EEPROM persistence is implemented in eeprom.c/h:

- Write/read password
- Write/read timeout
- Password “set” flag mechanism (for first-time setup logic)

6.1 Password Storage


```c
void EEPROM_WritePassword(uint8_t *password)
{
    EEPROMProgram((uint32_t *)password, PASSWORD_ADDRESS, 8);
}
```

6.2 Timeout Storage with Validation (5–30 seconds)

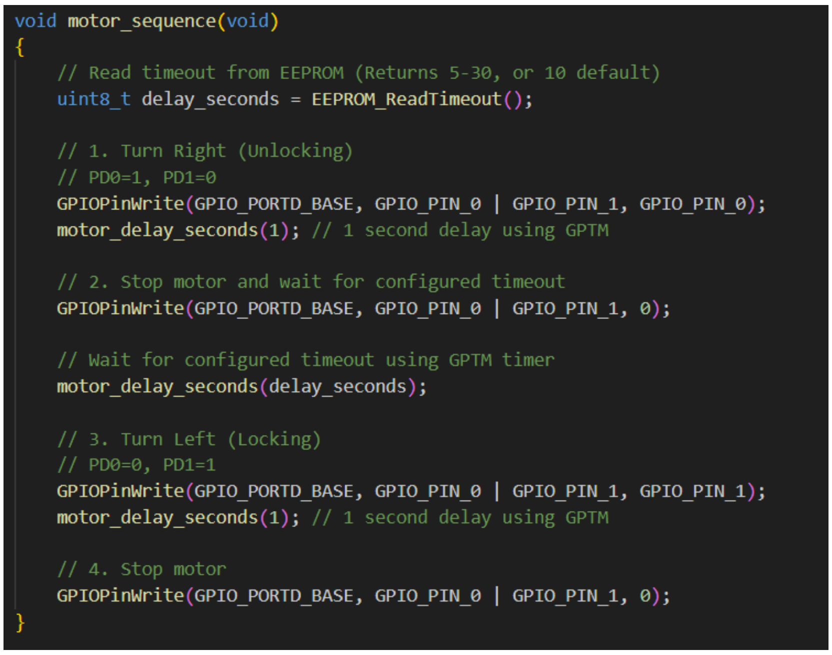

```c
uint8_t EEPROM_ReadTimeout(void)
{
    uint32_t timeout_data;
    EEPROMRead(&timeout_data, TIMEOUT_ADDRESS, 4);

    if (timeout_data >= 5 && timeout_data <= 30)
    {
        return (uint8_t)timeout_data;
    }
    return 10; // Default
}
```

This directly supports persistent configuration storage and validated timeout range.

7. Door Actuation (Motor Control + Auto-Lock Timeout)

The motor driver (motor.c/h) enforces the door open/close sequence using EEPROM timeout and timed motor activation.

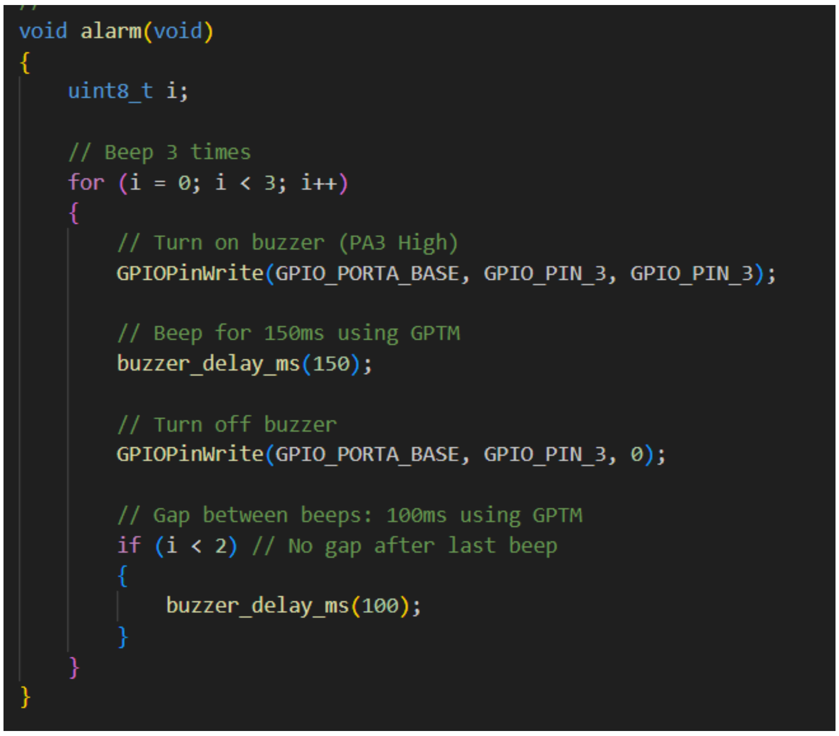
7.1 Motor Sequence (Unlock → Wait → Lock)


```c
void motor_sequence(void)
{
    // Read timeout from EEPROM (Returns 5-30, or 10 default)
    uint8_t delay_seconds = EEPROM_ReadTimeout();

    // 1. Turn Right (Unlocking)
    // PD0=1, PD1=0
    GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_0 | GPIO_PIN_1, GPIO_PIN_0);
    motor_delay_seconds(1); // 1 second delay using GPTM

    // 2. Stop motor and wait for configured timeout
    GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_0 | GPIO_PIN_1, 0);

    // Wait for configured timeout using GPTM timer
    motor_delay_seconds(delay_seconds);

    // 3. Turn Left (Locking)
    // PD0=0, PD1=1
    GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_0 | GPIO_PIN_1, GPIO_PIN_1);
    motor_delay_seconds(1); // 1 second delay using GPTM

    // 4. Stop motor
    GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_0 | GPIO_PIN_1, 0);
}
```

- Timeout logic is configuration-driven
- The door actuation sequence is encapsulated
- Motor is fully controlled from a single high-level function (easy to test)

8. Alarm & Lockout Signaling (Buzzer)

The buzzer module (buzzer.c/h) provides audible alarm behavior. The alarm() function implements a clear “beep pattern” using timer-based delays.



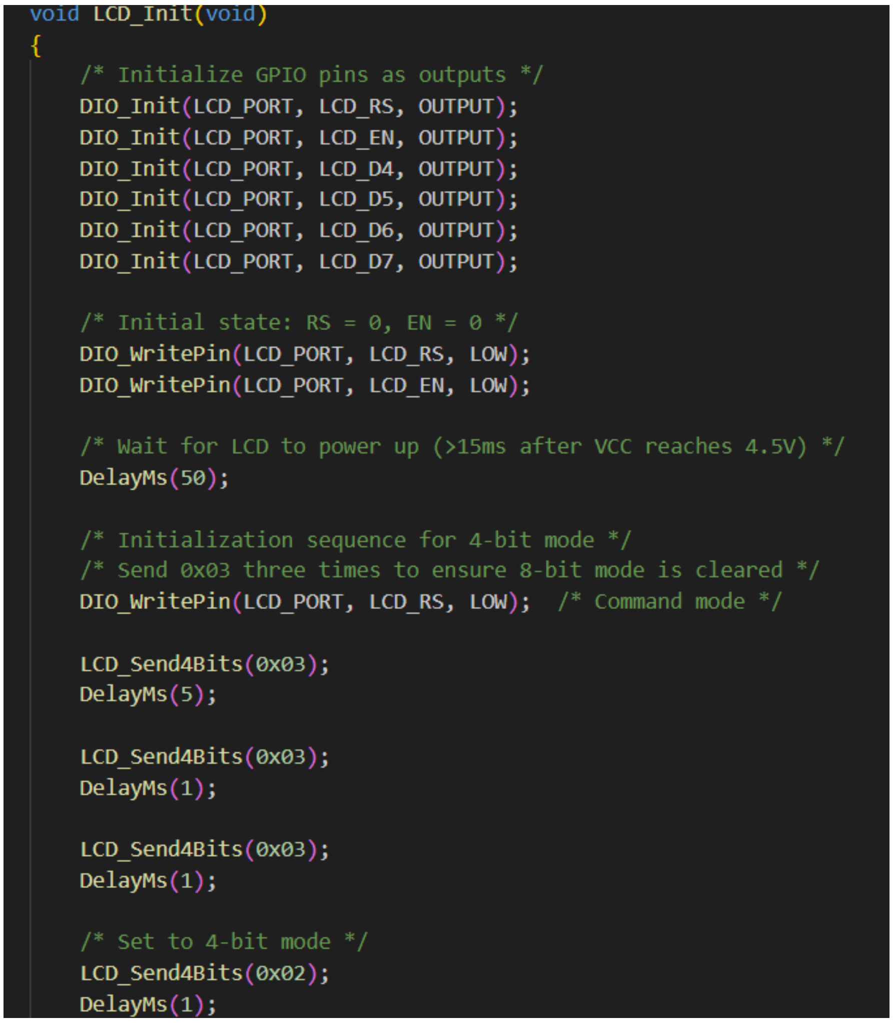
```c
void alarm(void)
{
    uint8_t i;

    // Beep 3 times
    for (i = 0; i < 3; i++)
    {
        // Turn on buzzer (PA3 High)
        GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, GPIO_PIN_3);

        // Beep for 150ms using GPTM
        buzzer_delay_ms(150);

        // Turn off buzzer
        GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, 0);

        // Gap between beeps: 100ms using GPTM
        if (i < 2) // No gap after last beep
        {
            buzzer_delay_ms(100);
        }
    }
}
```

This matches the rubric requirement for alarm activation after repeated failures (system lockout behavior is enforced at the application layer).

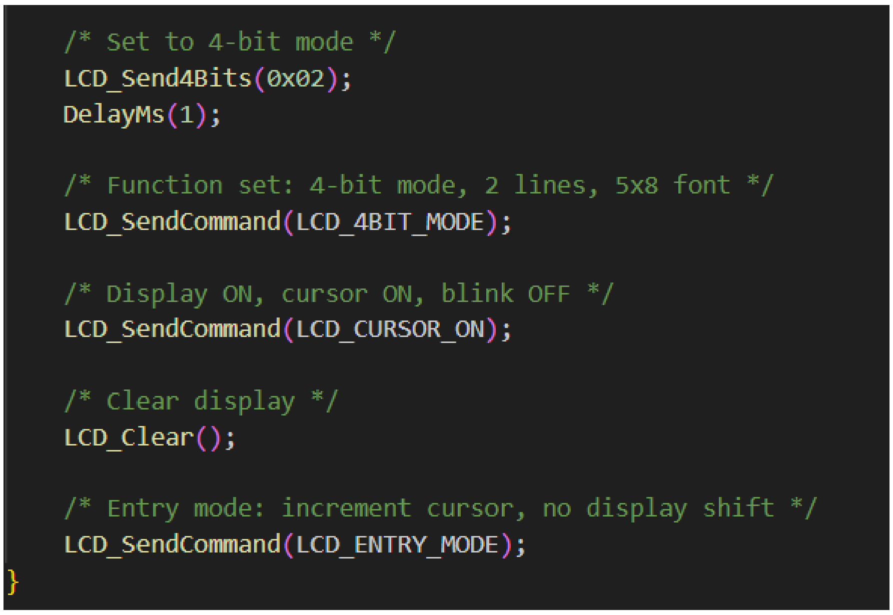
9. HMI_ECU Implementation Evidence (LCD + Keypad + ADC + LED)

9.1 LCD Driver (4-bit mode)

The LCD driver initializes and controls a 16×2 LCD with well-defined pin mapping and a correct 4-bit initialization sequence.



```c
void LCD_Init(void)
{
    /* Initialize GPIO pins as outputs */
    DIO_Init(LCD_PORT, LCD_RS, OUTPUT);
    DIO_Init(LCD_PORT, LCD_EN, OUTPUT);
    DIO_Init(LCD_PORT, LCD_D4, OUTPUT);
    DIO_Init(LCD_PORT, LCD_D5, OUTPUT);
    DIO_Init(LCD_PORT, LCD_D6, OUTPUT);
    DIO_Init(LCD_PORT, LCD_D7, OUTPUT);

    /* Initial state: RS = 0, EN = 0 */
    DIO_WritePin(LCD_PORT, LCD_RS, LOW);
    DIO_WritePin(LCD_PORT, LCD_EN, LOW);

    /* Wait for LCD to power up (>15ms after VCC reaches 4.5V) */
    DelayMs(50);

    /* Initialization sequence for 4-bit mode */
    /* Send 0x03 three times to ensure 8-bit mode is cleared */
    DIO_WritePin(LCD_PORT, LCD_RS, LOW);  /* Command mode */

    LCD_Send4Bits(0x03);
    DelayMs(5);

    LCD_Send4Bits(0x03);
    DelayMs(1);

    LCD_Send4Bits(0x03);
    DelayMs(1);

    /* Set to 4-bit mode */
    LCD_Send4Bits(0x02);
    DelayMs(1);
```



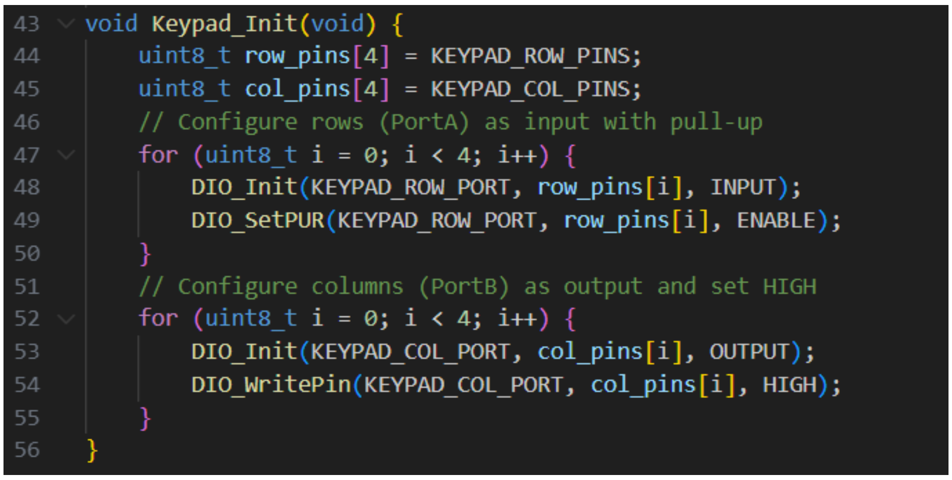
```c
    /* Set to 4-bit mode */
    LCD_Send4Bits(0x02);
    DelayMs(1);

    /* Function set: 4-bit mode, 2 lines, 5x8 font */
    LCD_SendCommand(LCD_4BIT_MODE);

    /* Display ON, cursor ON, blink OFF */
    LCD_SendCommand(LCD_CURSOR_ON);

    /* Clear display */
    LCD_Clear();

    /* Entry mode: increment cursor, no display shift */
    LCD_SendCommand(LCD_ENTRY_MODE);
}
```

9.2 Keypad Scanning (4×4)

The keypad driver uses a standard scanning approach:

- Rows = inputs with pull-ups

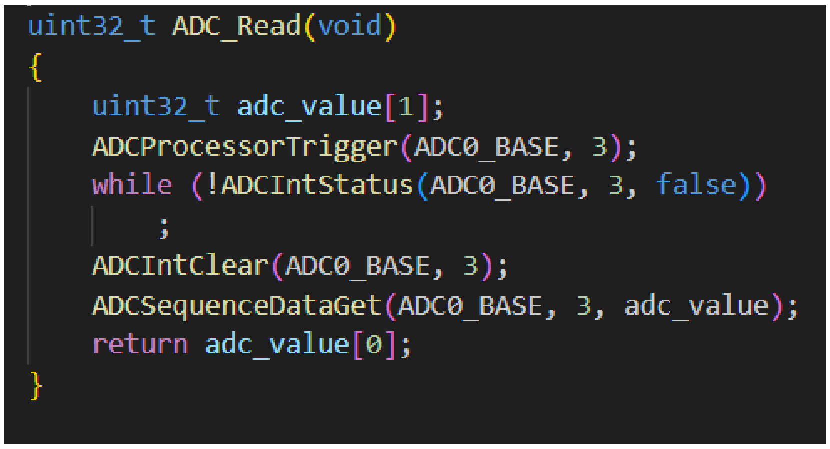
- Columns = outputs driven low one at a time



```c
void Keypad_Init(void) {
    uint8_t row_pins[4] = KEYPAD_ROW_PINS;
    uint8_t col_pins[4] = KEYPAD_COL_PINS;
    // Configure rows (PortA) as input with pull-up
    for (uint8_t i = 0; i < 4; i++) {
        DIO_Init(KEYPAD_ROW_PORT, row_pins[i], INPUT);
        DIO_SetPUR(KEYPAD_ROW_PORT, row_pins[i], ENABLE);
    }
    // Configure columns (PortB) as output and set HIGH
    for (uint8_t i = 0; i < 4; i++) {
        DIO_Init(KEYPAD_COL_PORT, col_pins[i], OUTPUT);
        DIO_WritePin(KEYPAD_COL_PORT, col_pins[i], HIGH);
    }
}
```

9.3 ADC Reading (Potentiometer)

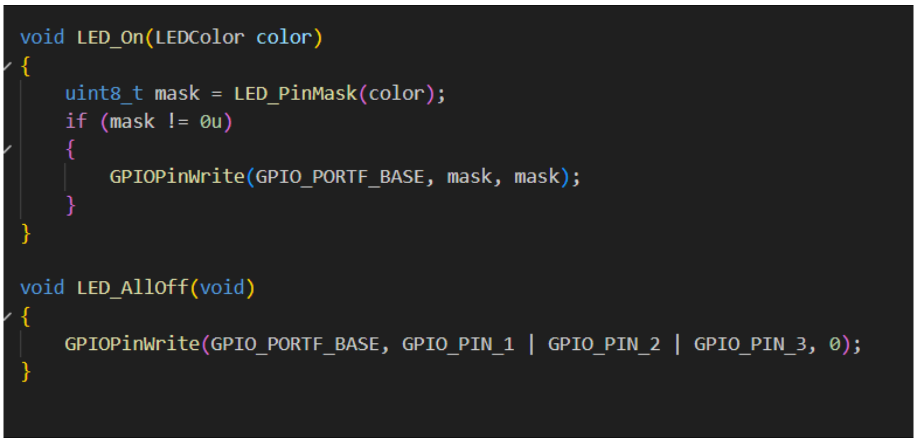
ADC sampling is implemented cleanly:



```c
uint32_t ADC_Read(void)
{
    uint32_t adc_value[1];
    ADCProcessorTrigger(ADC0_BASE, 3);
    while (!ADCIntStatus(ADC0_BASE, 3, false))
        ;
    ADCIntClear(ADC0_BASE, 3);
    ADCSequenceDataGet(ADC0_BASE, 3, adc_value);
    return adc_value[0];
}
```

9.4 RGB LED Control

LED control uses a color-to-pin mapping and a clean on/off interface:



```c
void LED_On(LEDColor color)
{
    uint8_t mask = LED_PinMask(color);
    if (mask != 0u)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, mask, mask);
    }
}

void LED_AllOff(void)
{
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0);
}
```

10. Development Environment & Professional Tooling

To strengthen the “professional practices” portion, Workbench settings and debug configuration artifacts are included such as:

- embProj.ewp, embProj.eww, embProj.ewd, embProj.ewt, embProj.dep
- Debug/run configurations: embProj.crun, embProj.dbgdt
- C-SPY launch scripts: embProj.Debug.cspy.bat, embProj.Debug.cspy.ps1
- Debug configuration files: embProj.Debug.driver.xcl, embProj.Debug.general.xcl
- Workspace debug: project.wsdt

11. Testing Evidence and Verification

Testing was performed at three levels: unit testing, integration testing, and full system testing.

11.1 Unit Testing

Each driver module was tested independently:

- UART: loopback and send/receive verification
- EEPROM: write/read consistency tests
- Motor: GPIO direction and timing validation
- Buzzer: alarm pattern timing verification
- LCD: text display and cursor control tests
- Keypad: key mapping accuracy
- ADC: potentiometer sweep and monotonic response
- LED: color activation verification

11.2 Integration Testing

- HMI_ECU command transmission verified via UART
- EEPROM-stored timeout correctly applied during motor operation
- Password verification logic tested across both ECUs

11.3 System Testing

End-to-end tests included:

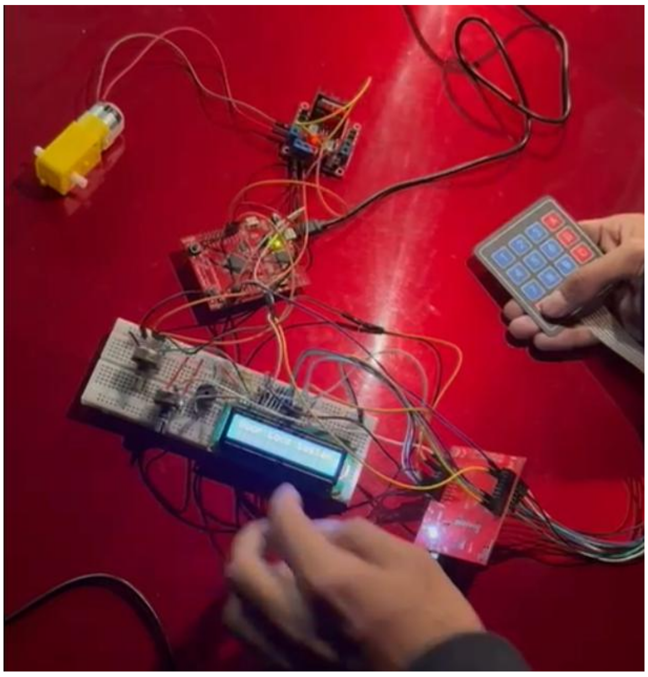
- First-time password setup
- Door unlock → wait → relock sequence
- Wrong password attempts triggering alarm and lockout
- Password change procedure

- Timeout configuration persistence after reset

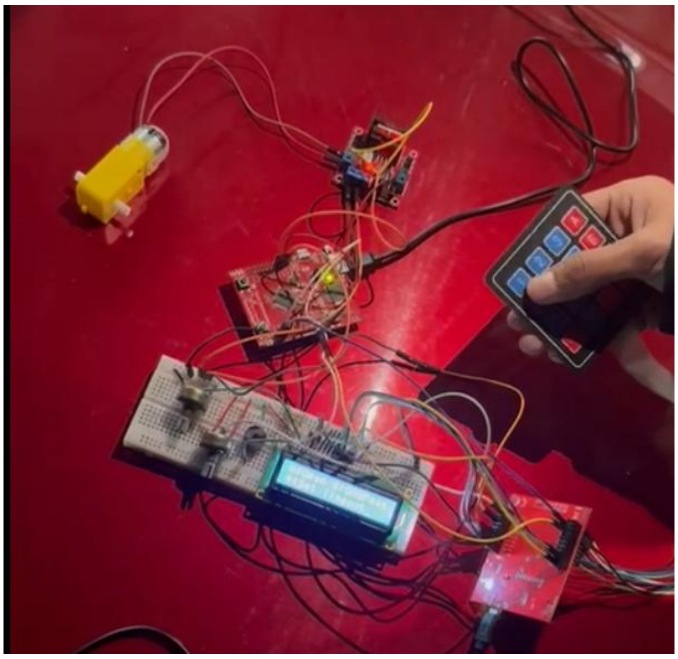
All tested features produced the expected system behavior.

11.4 Tests from Video

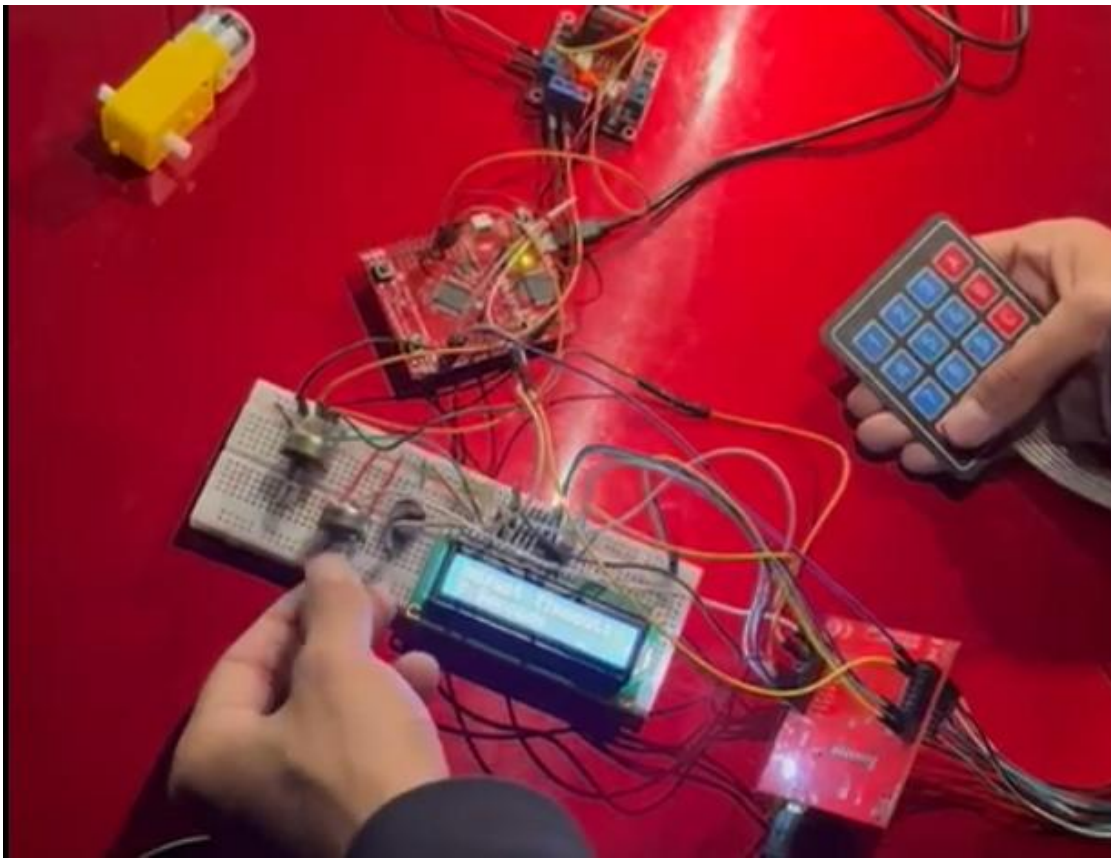
This is the initial setup when it says to enter password

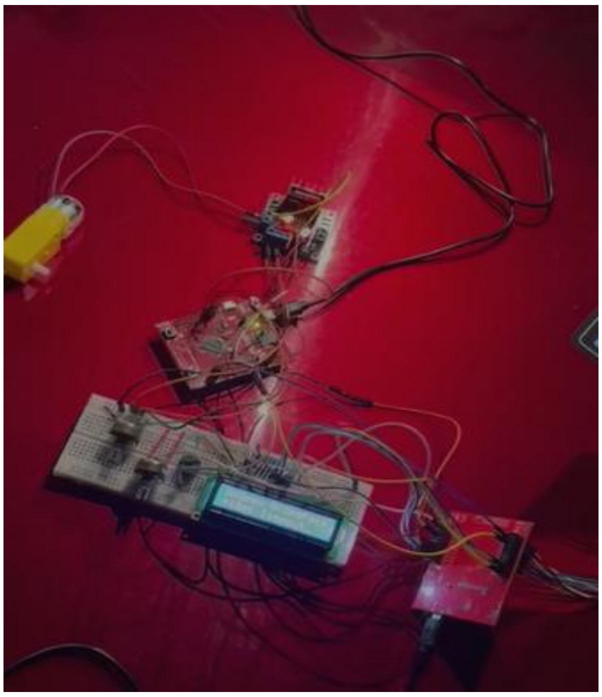


This is the green light when you enter the correct password

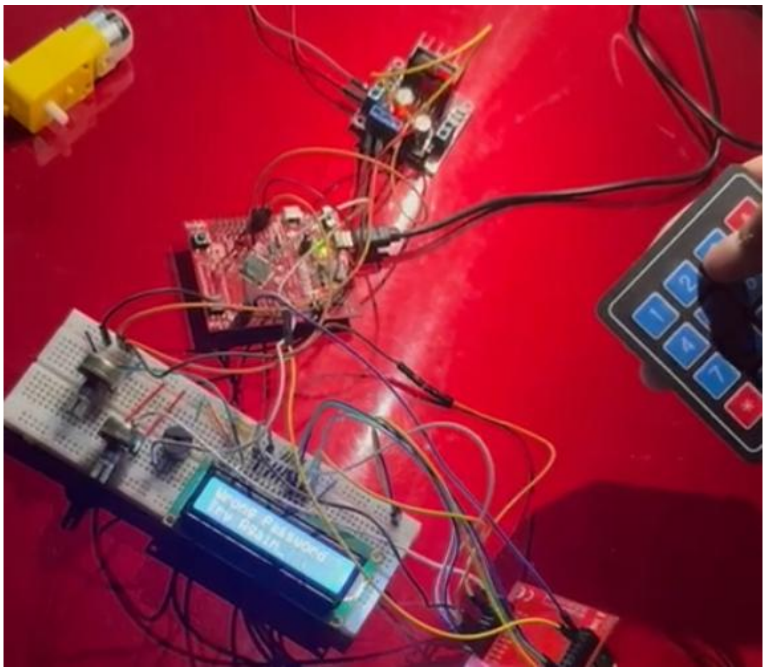


This is for adjusting timeout





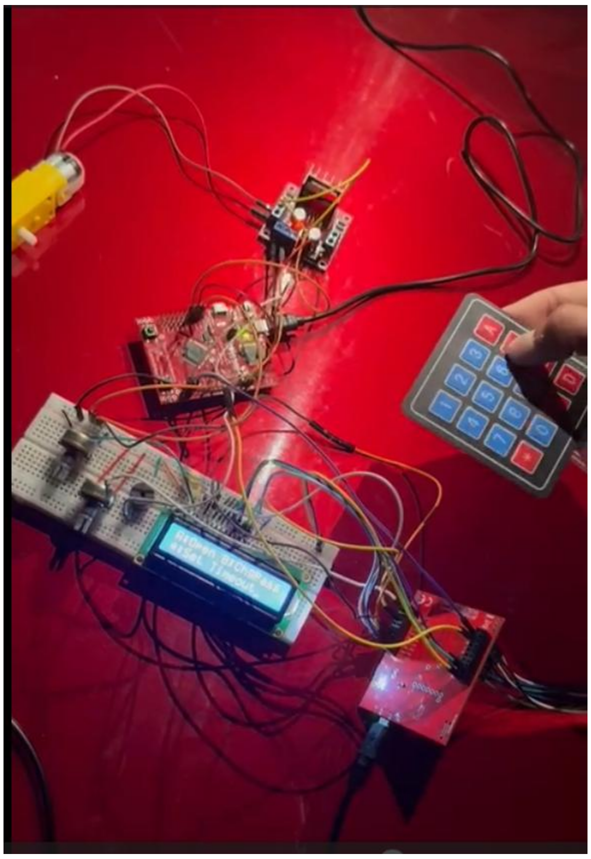
After timeout, you must wait for 8 seconds until you enter the password again



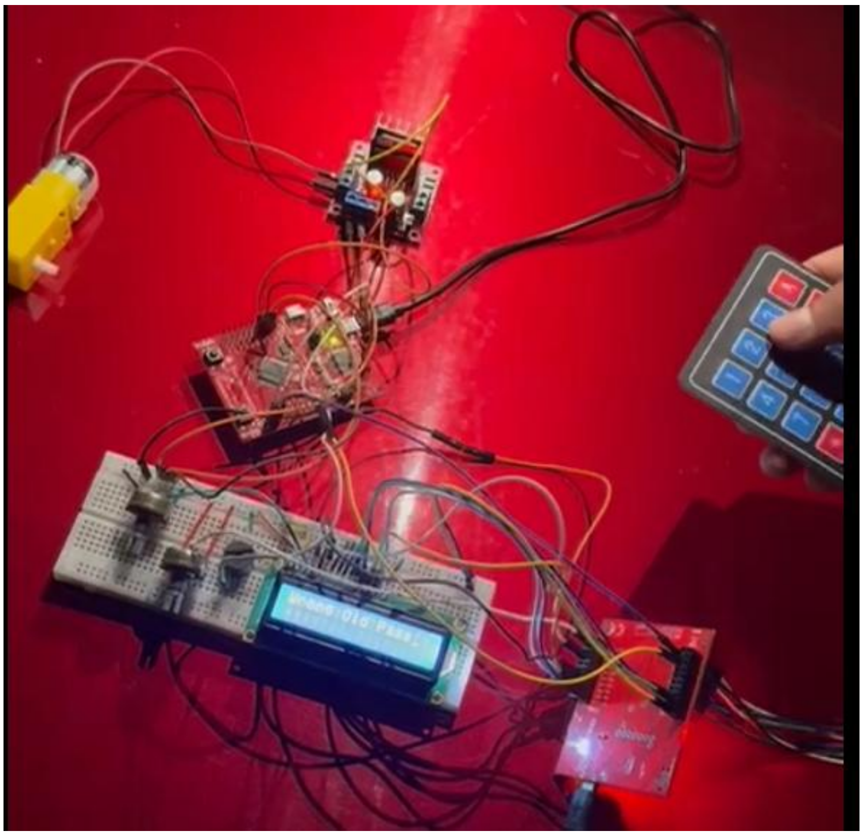
This is what happens after entering wrong password

After entering the password wrong for 3 times, the buzzer activates and makes a sound, and the system locks out for 20 seconds until you can try again.

Now we will try changing the password after pressing B



It asks you first to enter the old password to verify, then enter the new password and confirm to enter the new password



Verification of Old password

12. Challenges Faced and Technical Issues

During development, several practical challenges were encountered and resolved:

12.1 EEPROM Password and Timeout Storage

Initially, saving and retrieving password and timeout values from EEPROM caused inconsistent behavior after reset.

Solution:
EEPROM access was isolated into a dedicated driver, and validation logic was added to ensure only valid values were used by the system.

12.2 LCD Wiring and Initialization Issues

Early LCD connections resulted in unstable or corrupted output.

Solution:
Pin connections were rechecked, and the LCD initialization sequence was corrected to follow the HD44780 4-bit mode specification with proper startup delays.

12.3 Initial Lack of Layered Architecture

The project initially did not follow a layered design, leading to tightly coupled code.

Solution:
The codebase was reorganized into MCAL, HAL, and Application layers, improving readability, maintainability, and testability.

12.4 UART PD7 (NMI) Configuration Issue

UART communication failed initially due to PD7 being configured as an NMI pin by default.

Solution:
PD7 was explicitly unlocked before UART configuration, enabling correct UART communication.

13. Memory Usage Analysis

ROM & RAM Analysis

HMI_ECU (User-Side Tiva-C)

Microcontroller: TM4C123GH6PM
Flash (ROM): 256 KB
SRAM: 32 KB

13.1. ROM (Flash Memory) Analysis – HMI_ECU

Sources of ROM Consumption

ROM is used to store:

- Application code (Application Layer)
- Driver code (MCAL / HAL)
- Constant data (strings, lookup tables)

ROM-Consuming Modules

Module	Description	ROM Impact
main.c	Menus, password logic, UART protocol, state machine	High
lcd.c	LCD commands and initialization sequences	Medium
keypad.c	Key scanning logic and key mapping table	Medium
uart.c	UART2 initialization and transmit/receive functions	Medium
adc.c	ADC initialization and read logic	Low
dio.c	GPIO abstraction layer	Medium
systick.c	Delay functions and SysTick configuration	Low

Module	Description	ROM Impact
led.c	RGB LED control	Very Low

The primary source of ROM usage is main.c, due to:

- sprintf usage
- strcmp operations
- Menu and UI strings
- Password handling logic
- Timeout and timing calculations

Constant Data (.rodata Section)

The code includes multiple string literals such as:

- "Enter Password:"
- "Door Lock System"
- "Wrong Password"
- "Timeout Saved!"

In addition to:

const char keypad_codes[4][4];

All constant strings and lookup tables are stored in **Flash memory (.rodata)**, not in RAM.

ROM Sections Utilized

Section	Contents
.text	All functions and interrupt service routines (ISRs)
.rodata	LCD messages, keypad mapping, constant data

The system does **not** include:

- Dynamic code loading
- Large lookup tables
- External software libraries

ROM Usage Summary (HMI_ECU)

Item	Estimation
Application code	High
Drivers	Medium
Constant strings	Medium

Item	Estimation
Total ROM Used	**Low–Moderate (< 10% of 256 KB)**

Flash usage remains well within safe operating limits.

13.2. RAM (SRAM) Analysis – HMI_ECU

RAM Sections

The TM4C123 SRAM is divided into:

- .data → Initialized global variables
- .bss → Uninitialized global variables
- Stack
- Heap

Global and Static Variables

From drivers:

volatile uint32_t msTicks;

static uint8_t interruptMode;

From main.c:

bool access_granted;

bool confirmed;

uint8_t attempts;

uint32_t adc_val;

Global RAM usage is **very small**.

Local Buffers (Stack Usage)

The largest RAM usage occurs from local arrays inside functions:

Variable	Size
char pass1[6]	6 bytes
char pass2[6]	6 bytes
char password[6]	6 bytes
char new_pass1[6]	6 bytes
char new_pass2[6]	6 bytes
char str_buffer[16]	16 bytes

Worst-case stack usage:
< 500–700 bytes

Heap Usage

No dynamic memory allocation is used.

- No malloc
- No free

Heap size = 0 bytes

RAM Usage Summary (HMI_ECU)

RAM Component	**Approximate Size**
.data	< 32 bytes
.bss	< 64 bytes
Stack (worst case)	~700 bytes
Heap	0 bytes
Total RAM Used	**< 1 KB**
Available SRAM	**32 KB**

More than 96% of RAM remains free.

13.3. Comparison Insight: HMI_ECU vs Control_ECU

Aspect	**HMI_ECU**	**Control_ECU**
ROM Usage	Higher (UI + strings)	Medium
RAM Usage	Slightly higher	Lower
Stack Usage	Higher (menus & buffers)	Lower
EEPROM Usage	No	Yes

Reasoning:

HMI_ECU responsibilities:

- LCD display handling
- Keypad input
- User interface strings
- User interaction logic

Control_ECU responsibilities:

- Decision and control logic
- EEPROM storage
- Motor and buzzer control

13.4. Final Engineering Conclusion

The HMI_ECU software exhibits efficient memory utilization. RAM consumption is minimal because of the exclusive use of fixed-size buffers and absence of dynamic memory allocation.

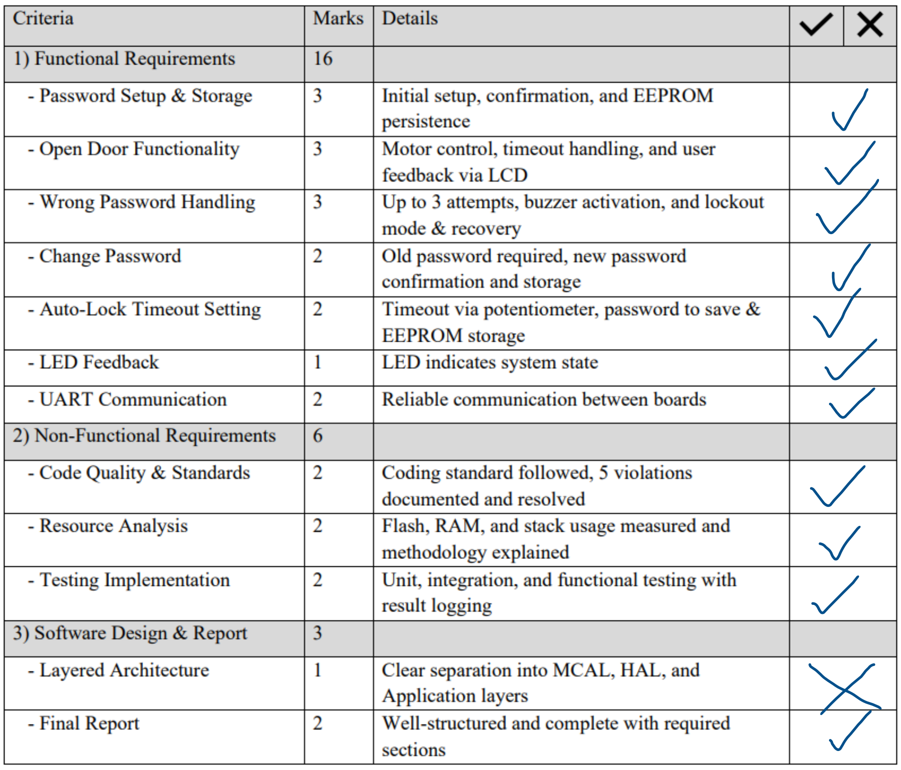
Flash memory usage is primarily dominated by application logic and user interface strings yet remains within the 256 KB available limit.

The system maintains a substantial safety margin in both RAM and ROM, ensuring high reliability and future scalability.

14. Team Contribution Table

Team Member	Tasks Contributed
Kareem Ahmed Talat	System timing (Systick) and Documentation
Youssef Maged	LCD, Keypad
Omar Osama	LCD, Keypad
Ferass Ahmed	Motor, Buzzer, ADC
Mohammed Kamal	Memory Usage Analysis, Motor and Documentation
Ahmed Elian	LCD, Keypad
Ahmed Elmlahe	EEPROM
Youssef Elnaggar	EEPROM
Ahmed Hesham	DIO
Abdullah Ali	Motor, Buzzer, ADC
Ahmed Sadek	UART
Paula Hanna	UART

15. Grading Rubric



Criteria	Marks	Details	✓	✗
1) Functional Requirements	16			
- Password Setup & Storage	3	Initial setup, confirmation, and EEPROM persistence	✓	
- Open Door Functionality	3	Motor control, timeout handling, and user feedback via LCD	✓	
- Wrong Password Handling	3	Up to 3 attempts, buzzer activation, and lockout mode & recovery	✓	
- Change Password	2	Old password required, new password confirmation and storage	✓	
- Auto-Lock Timeout Setting	2	Timeout via potentiometer, password to save & EEPROM storage	✓	
- LED Feedback	1	LED indicates system state	✓	
- UART Communication	2	Reliable communication between boards	✓	
2) Non-Functional Requirements	6			
- Code Quality & Standards	2	Coding standard followed, 5 violations documented and resolved	✓	
- Resource Analysis	2	Flash, RAM, and stack usage measured and methodology explained	✓	
- Testing Implementation	2	Unit, integration, and functional testing with result logging	✓	
3) Software Design & Report	3			
- Layered Architecture	1	Clear separation into MCAL, HAL, and Application layers		✗
- Final Report	2	Well-structured and complete with required sections	✓	

16. GitHub Link

https://github.com/ferasahmed285/CSE322-Door-Lock-System

17. Google Drive Video Link

https://drive.google.com/file/d/1_zwOZ4obHhpJO3RRtCwUy-yFGeYOmu6J/view?usp=sharing