MINISTRY OF EDUCATION AND RESEARCH
OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software and Automation Engineering

VREMERE ADRIAN, FAF-232

Report

Laboratory Work n.1.2

User Interaction: LCD + KeyPad

on Embedded Systems

Checked by:
Martîniuc Alexei, *univ. asist.*
FCIM, UTM

Chişinău – 2026

1. Domain Analysis

1.1. Objective of the Laboratory Work

The objective of this laboratory work is to design and implement an interactive embedded system that uses a 4x4 matrix keypad for user input and a 16x2 LCD display (connected via I2C) for visual feedback. The system implements an electronic lock with a menu-driven interface controlled entirely through the keypad, where the asterisk character (*) serves as a delimiter between command operands and the hash character (#) finalizes and executes commands.

The key learning objectives include understanding I2C serial communication for display interfacing, implementing matrix keypad scanning with software debouncing, designing a finite state machine (FSM) to manage complex user interaction flows, and building a modular software architecture that cleanly separates hardware drivers from application logic. This laboratory extends the concepts from Lab 1.1 (STDIO serial interface) by introducing physical user interface components (keypad and LCD) that interact with the user directly, rather than through a serial terminal.

1.2. Problem Definition

The task requires the development of an advanced menu-based interface for an electronic lock system with the following functional requirements:

1. Use the asterisk (*) character as a delimiter between operands in command sequences.
2. Use the hash (#) character to finalize and execute the current command.
3. Implement four commands accessible through the keypad:
   - `*0#` — Unconditional lock activation.
   - `*1*1234#` — Unlock with password verification (where `1234` is an example valid code).
   - `*2*1234*5678#` — Change password (from `1234` to `5678`).
   - `*3#` — Display the current lock state.
4. At each press of the * character, the LCD must display the available options for the current command context.
5. Upon pressing #, the system must execute the command and display a confirmation message.
6. The interface must provide clear visual feedback through LEDs (red for locked, green for unlocked) and descriptive messages on the LCD.
7. The application must be structured modularly, with the FSM logic separated from hardware drivers.

8. Software debouncing must be implemented for the keypad to prevent false transitions.
9. The system must respond to input changes with latency below 100ms.

1.3. Used Technologies

I2C Serial Communication Protocol

I2C (Inter-Integrated Circuit) is a synchronous, multi-master, multi-slave serial communication bus widely used in embedded systems for connecting low-speed peripherals to microcontrollers. Developed by Philips Semiconductor (now NXP), I2C uses only two bidirectional lines: SDA (Serial Data) for data transfer and SCL (Serial Clock) for clock synchronization.

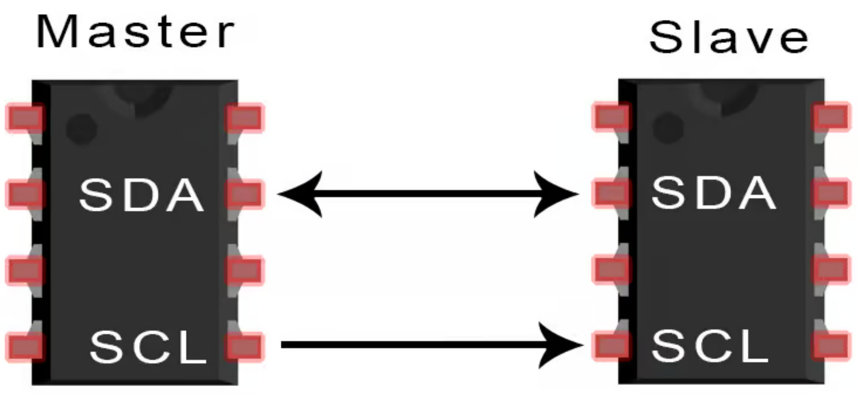


Figure 1.1 *– I2C serial communication protocol timing and frame structure*

The protocol operates in a master-slave configuration where the master device (Arduino Mega 2560 in this lab) initiates all communication by generating the clock signal and addressing specific slave devices. Each slave on the bus has a unique 7-bit address; the LCD module used in this lab responds to address `0x27`. Communication begins with a START condition (SDA transitions low while SCL is high), followed by the 7-bit slave address, a read/write bit, and then data bytes, each acknowledged by the receiver. The transaction ends with a STOP condition (SDA transitions high while SCL is high).

Key advantages of I2C for this application include minimal wiring (only two signal lines plus power and ground), built-in addressing (enabling multiple devices on the same bus), and widespread library support in the Arduino ecosystem through the Wire library.

Finite State Machines in Embedded Systems

A Finite State Machine (FSM) is a computational model consisting of a finite number of states, transitions between those states triggered by events (inputs), and actions performed during transitions or within states. FSMs are fundamental to embedded systems design because they provide a structured, deterministic approach to managing complex interactive behavior.

In this laboratory, the FSM manages the electronic lock's operational logic. The machine processes one keypad event at a time, transitioning between states that represent different phases of user interaction (idle, menu display, password entry, result display). Each state defines which keys are valid, what actions they trigger, and how the display content is updated. This approach ensures predictable behavior, simplifies testing (each state transition can be verified independently), and produces maintainable code.

LiquidCrystal_I2C Library

The LiquidCrystal_I2C library provides a high-level API for controlling HD44780-compatible LCD displays through an I2C backpack module. It abstracts the low-level I2C communication and LCD controller commands behind intuitive methods such as `init()`, `setCursor()`, `print()`, `clear()`, `backlight()`, and `noBacklight()`. The library depends on the Arduino Wire library for I2C bus communication.

Keypad Library

The Keypad library by Mark Stanley and Alexander Brevig provides a complete solution for reading matrix keypads on Arduino platforms. It handles the scanning algorithm (sequentially testing row-column combinations), software debouncing (using a configurable debounce time), and key state tracking (IDLE, PRESSED, HOLD, RELEASED). The `getKey()` method performs a non-blocking scan and returns the character of the pressed key, or null (`0`) if no key is currently active.

PlatformIO Build System

PlatformIO is a professional open-source ecosystem for embedded development. It provides project management, library dependency resolution, multi-board support, and integrated build/upload/monitor tools. In this lab, PlatformIO is used within VS Code to compile, upload, and monitor the application on the Arduino Mega 2560.

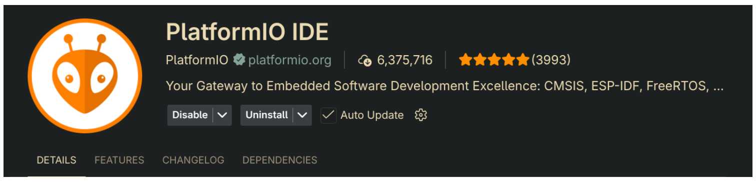

PlatformIO IDE
PlatformIO platformio.org | 6,375,716 | ★★★★★ (3993)
Your Gateway to Embedded Software Development Excellence: CMSIS, ESP-IDF, FreeRTOS, ...

***Figure 1.2** – PlatformIO development environment integrated with VS Code*

Wokwi Simulator

Wokwi is an online and VS Code-integrated simulator for embedded systems. It supports Arduino, ESP32, and other popular platforms. Wokwi allows testing of the application in a virtual environment before deploying to physical hardware.

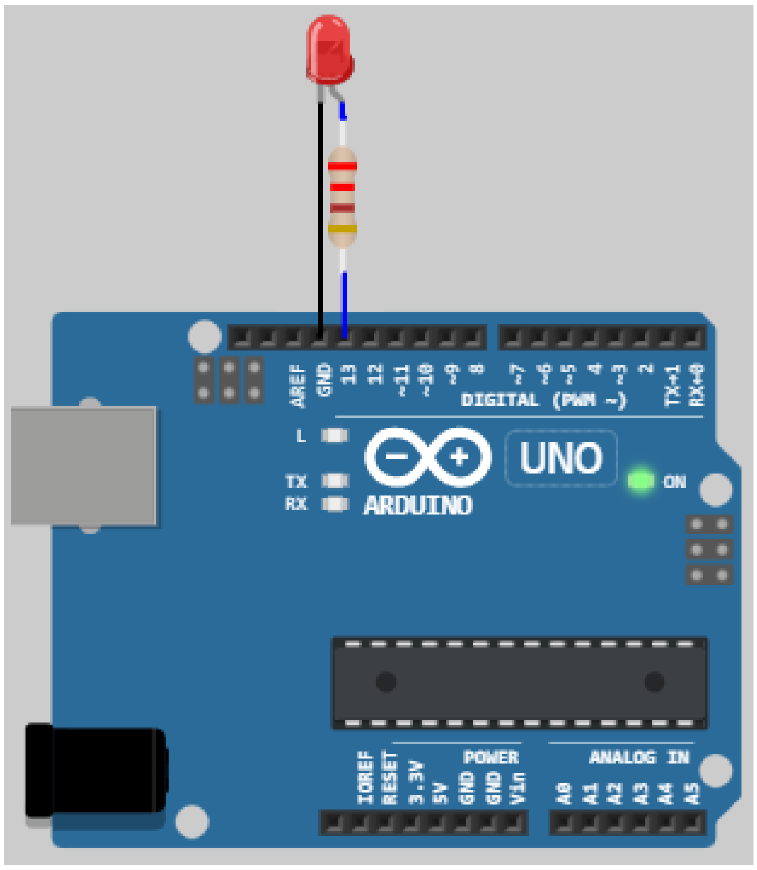


Figure 1.3 – Wokwi simulator for testing embedded systems and circuits

In this laboratory, Wokwi is integrated with PlatformIO through a configuration file (`wokwi.toml`), allowing the compiled firmware to be tested in the simulator without physical hardware.

1.4. Hardware Components

Arduino Mega 2560

The Arduino Mega 2560, based on the ATmega2560 AVR microcontroller, serves as the central processing unit. Its extensive pin count (54 digital I/O) is essential for this lab: 8 pins for the keypad matrix (4 rows + 4 columns), 2 pins for the I2C LCD (SDA + SCL), and 2 pins for the LED indicators, totaling 12 GPIO pins. The Mega's dedicated I2C hardware (on pins 20/SDA and 21/SCL) with internal pull-up resistors simplifies LCD integration.

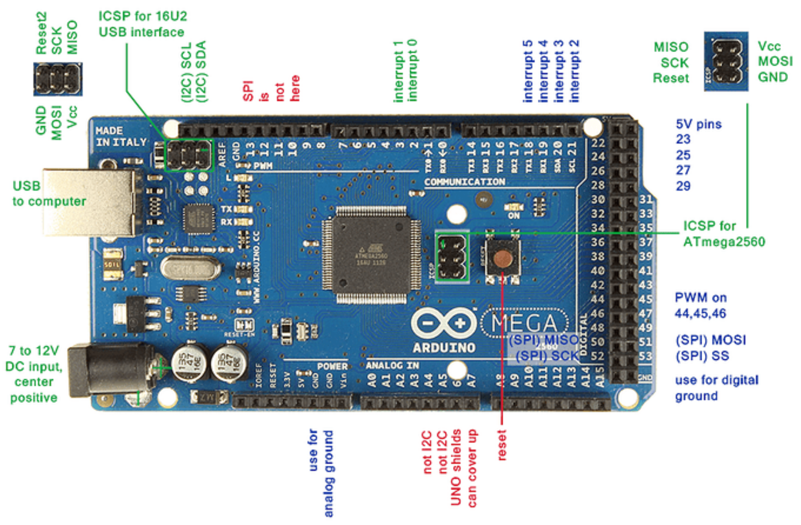


***Figure 1.4** – Arduino Mega 2560 development board with pin assignments*

Key specifications:

- **Microcontroller:** ATmega2560 (8-bit AVR, 16 MHz)
- **Flash memory:** 256 KB (8 KB used by bootloader)
- **SRAM:** 8 KB
- **Digital I/O pins:** 54 (12 used in this lab)
- **I2C:** Hardware TWI on pins 20 (SDA) and 21 (SCL)
- **Operating voltage:** 5V

LCD 1602 with I2C Backpack

The LCD 1602 is a character display module with 2 rows of 16 characters each, based on the HD44780 controller. The I2C backpack (typically using a PCF8574 I/O expander) reduces the parallel interface from 16 pins to just 4 (VCC, GND, SDA, SCL). The display supports ASCII characters and a set of custom characters, with an integrated backlight controlled via software.

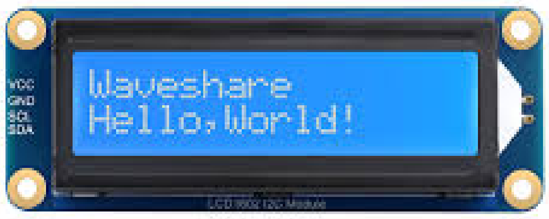


***Figure 1.5** – LCD 1602 display module with I2C backpack (PCF8574)*

Display specifications:

- **Display type:** 16 characters $\times$ 2 rows
- **Controller:** HD44780-compatible
- **Interface:** I2C via PCF8574 (address `0x27`)
- **Operating voltage:** 5V
- **Backlight:** LED, software-controllable

4x4 Membrane Keypad

The 4x4 membrane keypad provides 16 tactile buttons arranged in a 4-row by 4-column matrix. The standard layout includes digits 0–9, the asterisk (*), hash (#), and letters A–D. For this lock system, * and # serve as command delimiters and execution triggers respectively, while digits 0–9 are used for command selection and password entry.

The keypad operates using matrix scanning: the scanning algorithm sequentially drives each row LOW while reading the column pins. When a button is pressed, it creates an electrical connection between its row and column, pulling the corresponding column pin LOW and identifying the pressed key's position. This design requires only 8 GPIO pins (4 rows + 4 columns) instead of 16 individual pins.

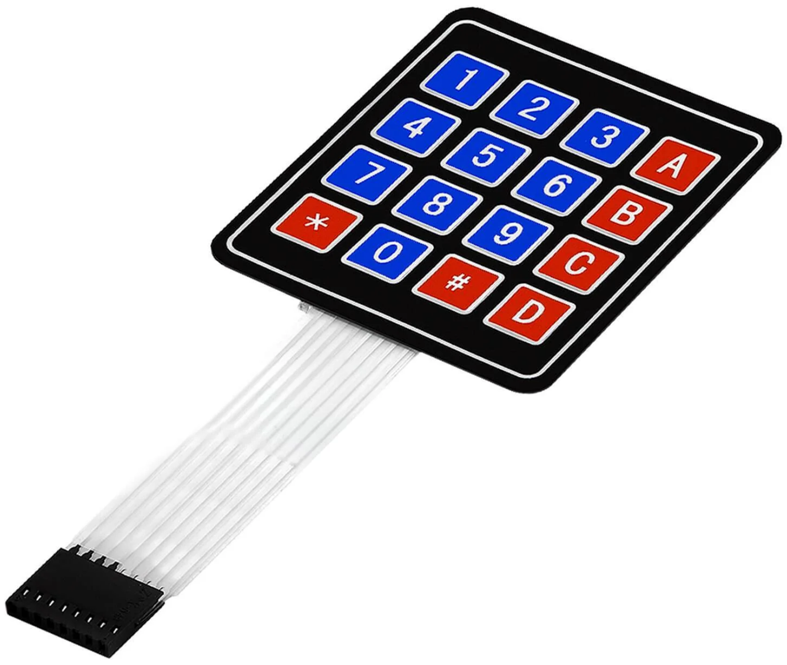


***Figure 1.6** – 4x4 membrane keypad matrix layout and scanning principle*

Software debouncing is essential for reliable keypad operation. Mechanical switch contacts bounce (make and break connection rapidly) for several milliseconds after being pressed or released. The Keypad library implements a state machine that tracks key press and release transitions, with a configurable debounce interval (set to 20ms in this lab) during which rapid state changes are filtered out.

Keypad specifications:

- **Keys:** 16 (digits 0–9, *, #, A–D)
- **Interface:** 8 pins (4 rows + 4 columns)
- **Technology:** Membrane (non-mechanical)
- **Bounce time:** approximately 5–20ms

LEDs and Current-Limiting Resistors

Two LEDs provide visual indication of the lock state: a red LED (connected to pin 7) indicates the LOCKED state, and a green LED (connected to pin 6) indicates the UNLOCKED state. Each LED is connected in series with a 220 Ω resistor to limit current to approximately 13.6 mA, calculated as $I = (5V - 2V)/220\Omega \approx 13.6mA$ for a typical LED forward voltage of 2V.

1.5. Software Components

Visual Studio Code with PlatformIO

Visual Studio Code with the PlatformIO IDE extension serves as the primary development environment. PlatformIO manages the build configuration through `platformio.ini`, where separate environments (`env:lab1_1`, `env:lab1_2`) allow independent compilation of each lab's code. PlatformIO also handles automatic installation of external library dependencies (LiquidCrystal_I2C, Keypad) specified in `lib_deps`.

Wokwi VS Code Extension

The Wokwi simulator provides a virtual Arduino Mega 2560 with connected LCD, keypad, and LEDs. It reads the circuit definition from `diagram.json` and the compiled firmware path from `wokwi.toml`. The simulation supports I2C communication, keypad matrix scanning, and LED visualization, enabling full system testing without physical hardware.

1.6. System Architecture and Justification

The system follows a layered architecture that separates concerns across five levels. The Application Layer (APP) contains `lab1_2_main`, which orchestrates peripheral initialization and the main loop — reading keypad input, feeding it to the FSM, updating the LCD, and controlling LEDs. The Service Layer (SRV) contains `LockFSM`, which implements all command processing logic, password management, and state transitions as a self-contained finite state machine that is independent of hardware specifics. The ECU Abstraction Layer (ECAL) contains three modules: `LcdDisplay` (wrapping the LiquidCrystal_I2C library for line-based LCD operations), `KeypadInput` (wrapping the Keypad library for debounced key reading), and `StdioSerial` (redirecting C standard I/O to UART for debug logging). The Microcontroller Abstraction Layer (MCAL) contains the `Led` driver, reused from Lab 1.1, providing GPIO-based LED control. Finally, the Hardware (HW) layer represents the physical ATmega2560 MCU with its I2C, GPIO, and UART peripherals.

This architecture was chosen because the FSM is the most complex component and benefits from complete isolation from hardware details. The FSM receives key characters as input and produces display content as output, making it testable independently of any peripheral. The LCD and keypad drivers are thin wrappers that adapt external libraries to the project's coding conventions, while the existing Led and StdioSerial modules are reused without modification from Lab 1.1.

1.7. Case Study: Keypad-Based Access Control Systems

Keypad-based access control is one of the most pervasive applications of embedded systems in everyday life. Residential door locks, such as Yale and Schlage smart locks, use membrane keypads combined with LCD or LED feedback to allow homeowners to set and enter PIN codes for entry. Commercial buildings employ keypad access panels (often from manufacturers like Honeywell or Bosch) at restricted entry points, where employees enter personal identification numbers to unlock doors, with the system logging each access event. Vehicle immobilizer systems in automotive security use keypad-based PIN entry as a secondary authentication method, while parking garage entry systems use keypads for ticket validation and automatic barrier control.

The FSM-driven command interface demonstrated in this lab mirrors the operational patterns of these commercial systems. The use of a delimiter-based command protocol (* for field separation, # for execution) is directly analogous to the interaction model used in alarm system keypads (e.g., DSC PowerSeries panels), where * accesses function menus and # confirms operations. The password verification and change functionality implemented here represents the core security operations found in every access control system, demonstrating fundamental concepts that scale from simple lab prototypes to production security infrastructure.

2. Design

2.1. System Architecture Diagrams

Component-Level Architecture

The system consists of four main component groups: a host PC (for serial debug monitoring), the Arduino Mega 2560 microcontroller, the input peripherals (4x4 keypad), and the output peripherals (LCD display and LEDs). The keypad provides user input through matrix scanning on GPIO pins 22–29. The MCU processes input through the lock FSM and drives the LCD over the I2C bus (pins 20/21) and the LEDs through GPIO pins 6 and 7. Debug information is transmitted to the serial terminal over USB-UART.

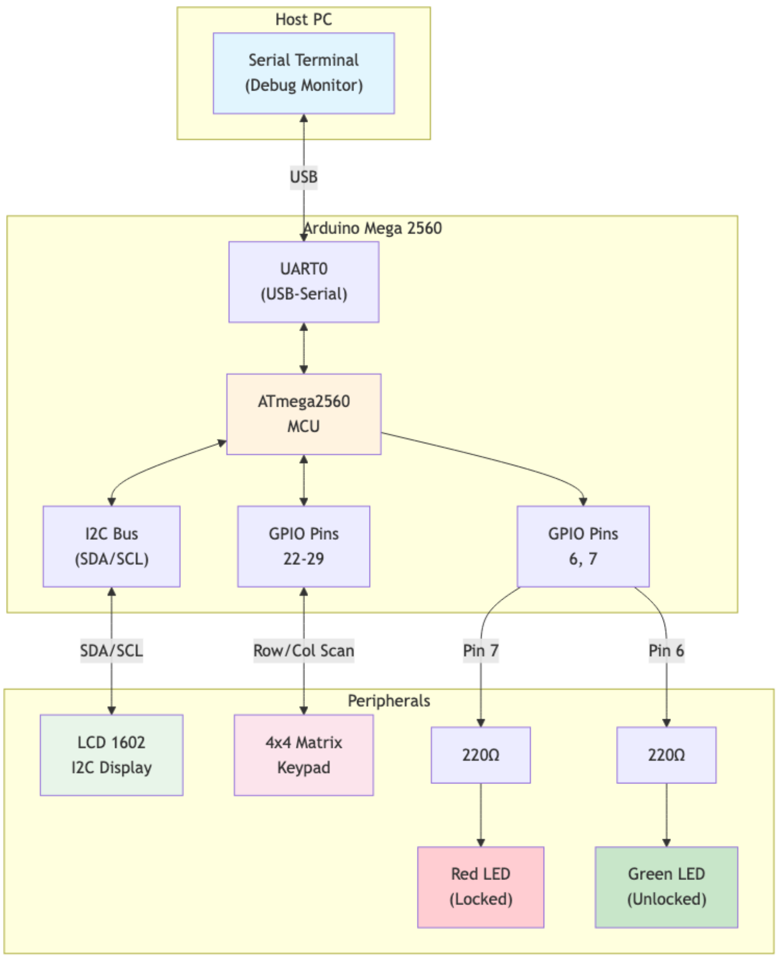


Figure 2.1 *– System structural diagram showing PC, MCU, keypad, LCD, and LED circuits*

The data flow operates as follows: the user presses a key on the 4x4 membrane keypad, which is detected by the MCU through periodic matrix scanning on GPIO pins 22–29. The key character is fed into the Lock FSM, which processes it according to the current state and generates updated display content. The application layer writes the new content to the LCD over I2C and updates the LED indicators to reflect the lock state. Simultaneously, debug information (key presses, state transitions, command results) is transmitted via `printf()` over UART to the serial terminal.

Layered System Architecture

The software follows a five-layer architecture that cleanly separates application logic from hardware interaction:

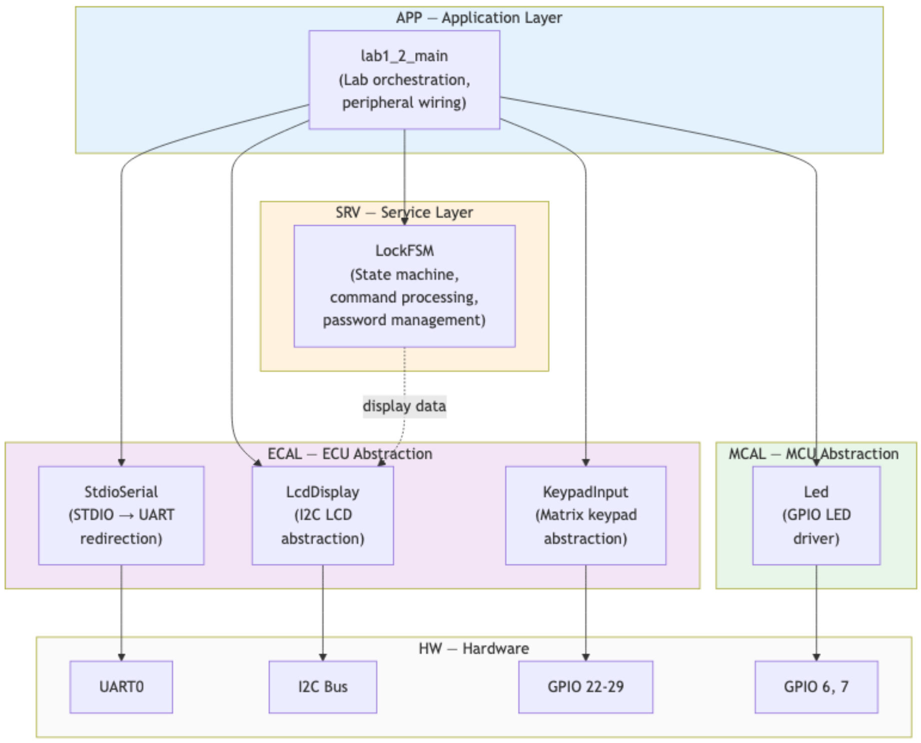

APP – Application Layer
lab1_2_main (Lab orchestration, peripheral wiring)
SRV – Service Layer
LockFSM (State machine, command processing, password management)
display data
ECAL – ECU Abstraction
StdioSerial (STDIO → UART redirection)
LcdDisplay (I2C LCD abstraction)
KeypadInput (Matrix keypad abstraction)
MCAL – MCU Abstraction
Led (GPIO LED driver)
HW – Hardware
UART0
I2C Bus
GPIO 22-29
GPIO 6, 7

Figure 2.2 *– Layered software architecture of the lock system*

The Application Layer (APP) contains `lab1_2_main`, which serves as the lab's entry point. It initializes all peripherals, runs the main loop (keypad read → FSM process → LCD update → LED update), and coordinates communication between the FSM service and hardware drivers. The Service Layer (SRV) contains `LockFSM`, which encapsulates all lock logic: state management across 10 FSM states, password storage and verification, command parsing, and display content generation. The FSM is hardware-independent — it accepts character inputs and produces display data, enabling independent testing. The ECU Abstraction Layer (ECAL) contains `LcdDisplay` (I2C LCD operations with line padding), `KeypadInput` (debounced 4x4 keypad scanning), and `StdioSerial` (STDIO-to-UART redirection from Lab 1.1). The Microcontroller Abstraction Layer (MCAL) contains the `Led` driver (GPIO-based LED control, reused from Lab 1.1). The Hardware (HW) layer represents the physical ATmega2560 with UART0, I2C bus, and GPIO pins.

2.2. Block Diagrams

Application Flowchart

The application operates in two phases: initialization and the main event loop. During initialization, all peripherals are configured in a specific order (serial first to enable debug output, then LCD, keypad, LEDs, and finally the FSM). The main loop follows a non-blocking event-driven pattern: it checks for keypad input, processes it through the FSM, and updates outputs only when changes occur.

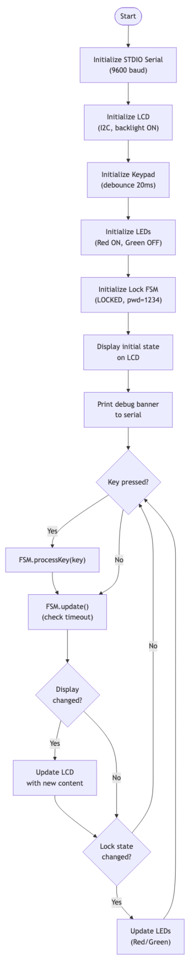

Start
Initialize STDIO Serial (9600 baud)
Initialize LCD (I2C, backlight ON)
Initialize Keypad (debounce 20ms)
Initialize LEDs (Red ON, Green OFF)
Initialize Lock FSM (LOCKED, pwd=1234)
Display initial state on LCD
Print debug banner to serial
Key pressed?
Yes
No
FSM.processKey(key)
FSM.update() (check timeout)
No
Display changed?
Yes
No
Update LCD with new content
Lock state changed?
Yes
Update LEDs (Red/Green)

***Figure 2.3** – Flowchart of the application initialization and main loop*

Unlike the blocking I/O approach used in Lab 1.1 (where `fgets()` halted the main loop), this lab uses a non-blocking architecture. The `keypad.getKey()` method returns immediately with either a key character or null (0), allowing the loop to service other tasks (FSM timeout checks, display updates, LED management) without delay. This design ensures the system responds to input within one loop iteration, well under the 100ms latency requirement.

FSM State Diagram

The lock system's behavior is governed by a finite state machine with 10 states. The FSM starts in the IDLE state and progresses through command selection, parameter entry, execution, and result display before returning to IDLE.

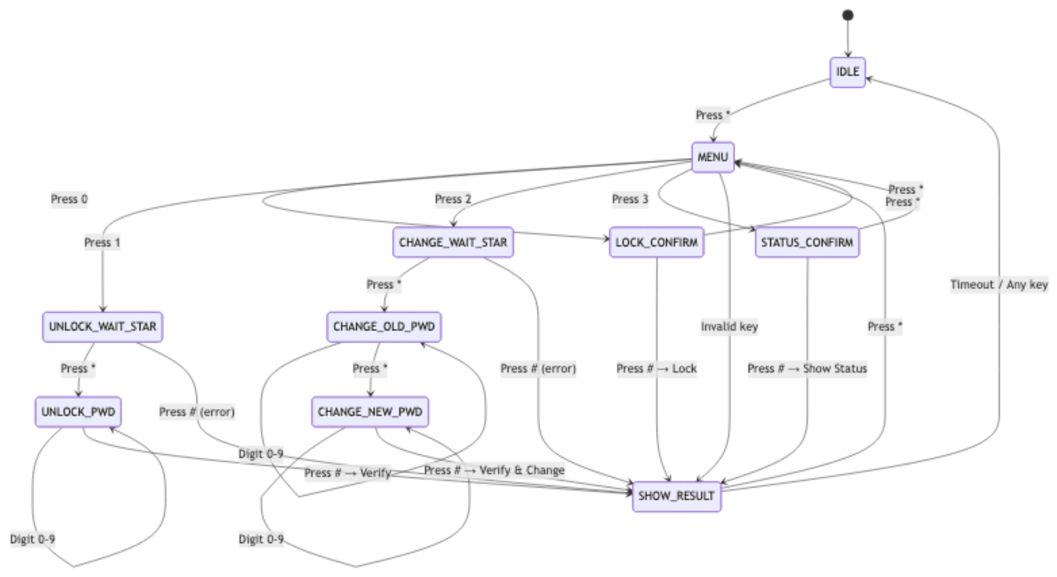


__Figure 2.4__ – State diagram of the Lock FSM

The states and their roles are:

- **IDLE**: Welcome screen. Waits for * to begin a command sequence.
- **MENU**: Displays available commands (0–3). Transitions based on the digit pressed.
- **LOCK_CONFIRM**: Command 0 selected. Waits for # to execute lock.
- **UNLOCK_WAIT_STAR**: Command 1 selected. Waits for * to begin password entry.
- **UNLOCK_PWD**: Password entry for unlock. Accepts digits, executes on #.
- **CHANGE_WAIT_STAR**: Command 2 selected. Waits for * to begin old password entry.
- **CHANGE_OLD_PWD**: Old password entry. Accepts digits, delimited by *.
- **CHANGE_NEW_PWD**: New password entry. Accepts digits, executes on #.
- **STATUS_CONFIRM**: Command 3 selected. Waits for # to display status.
- **SHOW_RESULT**: Displays result message for 2.5 seconds, then auto-returns to IDLE.

FSM Key Processing Flowchart

The `processKey()` method implements the FSM transition logic. For each incoming key, it evaluates the current state and the key value to determine the appropriate action (state transition, buffer modification, command execution, or error display).

Figure 2.5 *– Flowchart of the FSM processKey() function*

2.3. Electrical Schematics

The circuit integrates five components connected to the Arduino Mega 2560: the I2C LCD display, the 4x4 membrane keypad, and two LED indicator circuits.

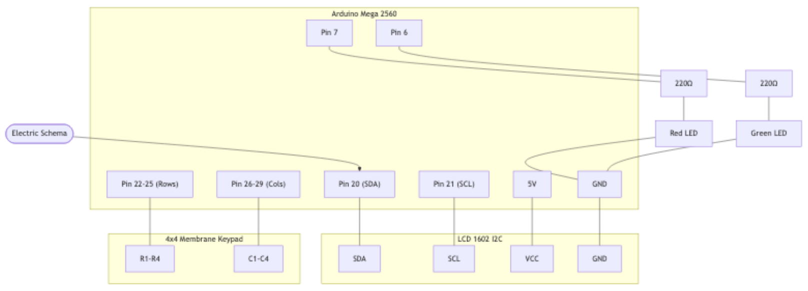


Figure 2.6 *– Electrical schematic of the lock system*

Component Specification

The circuit consists of the following components. The Arduino Mega 2560 is the central microcontroller board (ATmega2560, 5V logic, 16 MHz). The LCD 1602 with I2C backpack is a 16×2 character display at I2C address `0x27`, powered at 5V. The 4x4 membrane keypad has 16 keys arranged in a row-column matrix requiring 8 GPIO pins. Two LEDs (red and green, 5mm) serve as lock state indicators with forward voltage approximately 2V, connected through 220 Ω current-limiting resistors providing approximately 13.6 mA each.

Circuit Connections

The I2C LCD connects using 4 wires: VCC to 5V, GND to ground, SDA to pin 20, and SCL to pin 21. The Arduino Mega's internal pull-up resistors on the I2C lines eliminate the need for external pull-ups. The keypad connects using 8 wires: rows R1–R4 to pins 22–25, and columns C1–C4 to pins 26–29. The Keypad library configures internal pull-ups on column pins and drives row pins as outputs during scanning. The red LED connects from pin 7 through a 220 Ω resistor to ground. The green LED connects from pin 6 through a 220 Ω resistor to ground.

Hardware Configuration

The Wokwi simulation implements the circuit virtually. The following configuration defines the virtual components and their connections:

Listing 2.1 *– Wokwi diagram.json — Virtual circuit definition for Lab 1.2*

```json
{
  "version": 1,
  "author": "Vremere Adrian",
  "editor": "wokwi",
  "parts": [
    {"type": "wokwi-arduino-mega", "id": "mega",
     "top": 0, "left": 0, "attrs": {}},
    {"type": "wokwi-lcd1602", "id": "lcd",
     "top": -280, "left": 100,
     "attrs": {"pins": "i2c"}},
    {"type": "wokwi-membrane-keypad", "id": "keypad",
     "top": -380, "left": -250,
     "attrs": {"keys": ["123A","456B","789C","*0#D"]}},
    {"type": "wokwi-resistor", "id": "r1",
     "top": -50, "left": 210, "rotate": 90,
     "attrs": {"resistance": "220"}},
    {"type": "wokwi-led", "id": "led_red",
     "top": -120, "left": 200,
     "attrs": {"color": "red", "label": "LOCKED"}},
    {"type": "wokwi-resistor", "id": "r2",
     "top": -50, "left": 250, "rotate": 90,
     "attrs": {"resistance": "220"}},
    {"type": "wokwi-led", "id": "led_green",
     "top": -120, "left": 240,
     "attrs": {"color": "green", "label": "UNLOCKED"}}
  ],
  "connections": [
    ["mega:7","r1:1","red",["v0"]],
    ["r1:2","led_red:A","red",["v0"]],
    ["led_red:C","mega:GND.1","black",["v0"]],
    ["mega:6","r2:1","green",["v0"]],
    ["r2:2","led_green:A","green",["v0"]],
    ["led_green:C","mega:GND.1","black",["v0"]],
    ["lcd:GND","mega:GND.2","black",["v0"]],
    ["lcd:VCC","mega:5V","red",["v0"]],
    ["lcd:SDA","mega:20","blue",["v0"]],
    ["lcd:SCL","mega:21","purple",["v0"]],
    ["keypad:R1","mega:22","orange",["v0"]],
    ["keypad:R2","mega:23","orange",["v0"]],
    ["keypad:R3","mega:24","orange",["v0"]],
    ["keypad:R4","mega:25","orange",["v0"]],
    ["keypad:C1","mega:26","yellow",["v0"]],
    ["keypad:C2","mega:27","yellow",["v0"]],
    ["keypad:C3","mega:28","yellow",["v0"]],
    ["keypad:C4","mega:29","yellow",["v0"]]
  ]
}
```

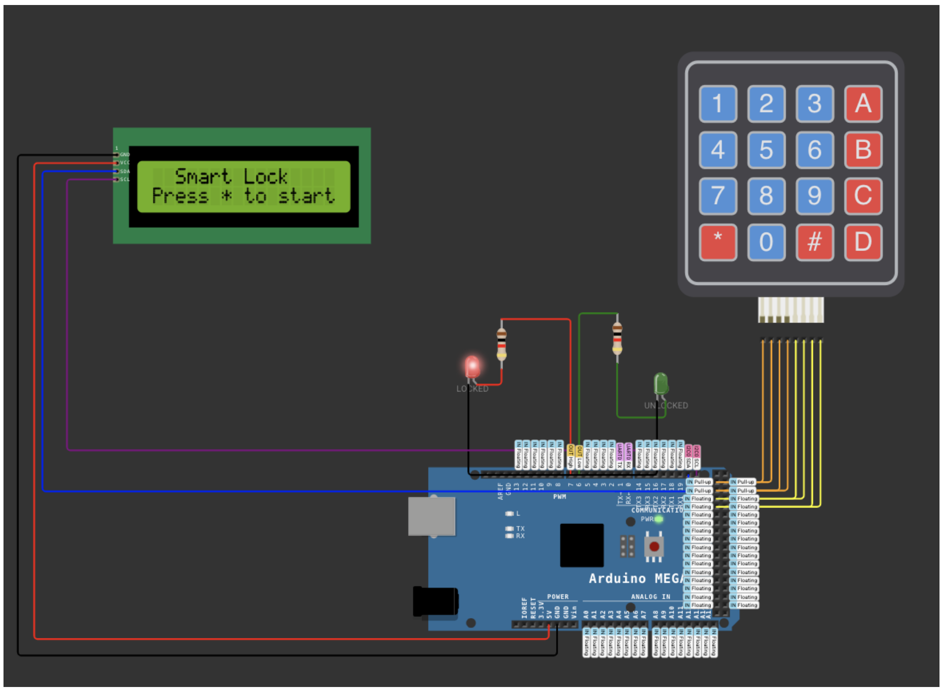


Figure 2.7 *– Wokwi simulation circuit — Arduino Mega with LCD, keypad, and LEDs*

2.4. Project Structure

The project extends the Lab 1.1 structure by adding three new library modules and a second lab entry point, while preserving all existing code:

Listing 2.2 *– Project directory structure for Lab 1.2*

```
labs/
|-- platformio.ini              # Build config (env:lab1_1, env:lab1_2)
|-- src/
|   |-- main.cpp                # Entry point (preprocessor lab selector)
|-- lab/
|   |-- lab1_1/
|   |   |-- lab1_1_main.h       # Lab 1.1 interface (preserved)
|   |   |-- lab1_1_main.cpp     # Lab 1.1 implementation (preserved)
|   |-- lab1_2/
|       |-- lab1_2_main.h       # Lab 1.2 interface
|       |-- lab1_2_main.cpp     # Lab 1.2 implementation
|-- lib/
|   |-- Led/                    # MCAL: LED driver (reused from Lab 1.1)
|   |   |-- Led.h
|   |   |-- Led.cpp
|   |-- StdioSerial/            # ECAL: STDIO redirection (reused)
|   |   |-- StdioSerial.h
|   |   |-- StdioSerial.cpp
|   |-- CommandParser/          # SRV: Command parser (Lab 1.1 only)
|   |   |-- CommandParser.h
|   |   |-- CommandParser.cpp
|   |-- LcdDisplay/             # ECAL: I2C LCD driver (new)
|   |   |-- LcdDisplay.h
|   |   |-- LcdDisplay.cpp
|   |-- KeypadInput/           # ECAL: Matrix keypad driver (new)
|   |   |-- KeypadInput.h
|   |   |-- KeypadInput.cpp
|   |-- LockFSM/               # SRV: Lock state machine (new)
|       |-- LockFSM.h
|       |-- LockFSM.cpp
|-- wokwi/
    |-- lab1.1/                # Wokwi circuit for Lab 1.1 (preserved)
    |   |-- diagram.json
    |   |-- wokwi.toml
    |-- lab1.2/                # Wokwi circuit for Lab 1.2 (new)
        |-- diagram.json
        |-- wokwi.toml
```

Lab isolation is achieved through PlatformIO's `build_src_filter` and preprocessor defines (`-DLAB1_1`, `-DLAB1_2`). The `main.cpp` entry point uses conditional compilation to include and call the correct lab module, eliminating the need to manually comment/uncomment code when switching between labs.

2.5. Modular Implementation

MCAL Layer: Led Driver (Reused from Lab 1.1)

The Led driver, implemented in Lab 1.1, is reused without modification. It provides GPIO-based LED control through an object-oriented interface: `init()`, `turnOn()`, `turnOff()`, `toggle()`, and `isOn()`. In Lab 1.2, two Led instances are created — one for the red (locked) indicator on pin 7 and one for the green (unlocked) indicator on pin 6.

***Listing 2.3** – Led.h — LED driver interface (reused from Lab 1.1)*

```cpp
#ifndef LED_H
#define LED_H
#include <Arduino.h>

class Led {
public:
    Led(uint8_t pin);
    void init();
    void turnOn();
    void turnOff();
    void toggle();
    bool isOn() const;
private:
    uint8_t ledPin;
    bool state;
};
```

```
#endif
```

ECAL Layer: LcdDisplay Driver

The LcdDisplay module wraps the LiquidCrystal_I2C library behind a simplified interface focused on line-based text operations. The key design decision is the `printLine()` method, which writes text to a specific row and pads the remaining columns with spaces. This eliminates display artifacts from previous content without requiring a full `clear()` call (which causes visible flickering).

***Listing 2.4** – LcdDisplay.h — LCD display driver interface*

```cpp
#ifndef LCD_DISPLAY_H
#define LCD_DISPLAY_H
#include <Arduino.h>
#include <LiquidCrystal_I2C.h>

class LcdDisplay {
public:
    LcdDisplay(uint8_t i2cAddress, uint8_t cols, uint8_t rows);
    void init();
    void clear();
    void setCursor(uint8_t col, uint8_t row);
    void print(const char *text);
    void printLine(uint8_t row, const char *text);
    void showTwoLines(const char *line1, const char *line2);
    void backlight(bool on);
private:
    LiquidCrystal_I2C _lcd;
    uint8_t _cols;
    uint8_t _rows;
};
#endif
```

***Listing 2.5** – LcdDisplay.cpp — LCD display implementation (key sections)*

```cpp
#include "LcdDisplay.h"
#include <string.h>

LcdDisplay::LcdDisplay(uint8_t i2cAddress, uint8_t cols, uint8_t rows)
    : _lcd(i2cAddress, cols, rows), _cols(cols), _rows(rows) {}

void LcdDisplay::init() {
    _lcd.init();
    _lcd.backlight();
    _lcd.clear();
}

```

```
void LcdDisplay::printLine(uint8_t row, const char *text) {
    _lcd.setCursor(0, row);
    uint8_t len = strlen(text);
    if (len > _cols) len = _cols;
    for (uint8_t i = 0; i < len; i++) {
        _lcd.write(text[i]);
    }
    // Pad with spaces to clear previous content
    for (uint8_t i = len; i < _cols; i++) {
        _lcd.write(' ');
    }
}

void LcdDisplay::showTwoLines(const char *line1, const char *line2) {
    printLine(0, line1);
    printLine(1, line2);
}
```

The `printLine()` function's approach of writing character-by-character and padding with spaces is more efficient than calling `lcd.clear()` (which takes approximately 2ms for the HD44780 to process) and prevents the visible flicker that occurs when the entire display is blanked before new content is written.

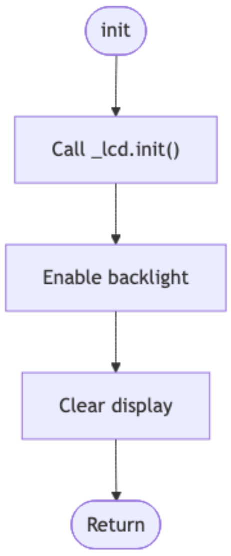


Figure 2.8 *– Flowchart: LCD initialization*

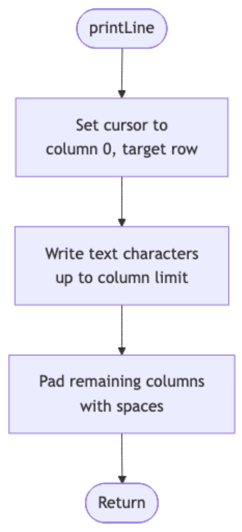

printLine
Set cursor to column 0, target row
Write text characters up to column limit
Pad remaining columns with spaces
Return

Figure 2.9 *– Flowchart: LCD printLine function*

ECAL Layer: KeypadInput Driver

The KeypadInput module wraps the Keypad library with a fixed 4x4 key layout and configurable pin assignments. The standard keypad layout (1-9, 0, *, #, A-D) is defined as a static array in the implementation file, while the GPIO pin assignments are passed through the constructor from the application layer.

Listing 2.6 *– KeypadInput.h — Keypad driver interface*

```
#ifndef KEYPAD_INPUT_H
#define KEYPAD_INPUT_H
#include <Arduino.h>
#include <Keypad.h>

static const byte KEYPAD_ROWS = 4;
static const byte KEYPAD_COLS = 4;

class KeypadInput {
public:
    KeypadInput(byte *rowPins, byte *colPins);
    void init();
    char getKey();
private:
    Keypad _keypad;
};
#endif
```

Listing 2.7 *– KeypadInput.cpp — Keypad driver implementation*

```
#include "KeypadInput.h"

static char keymap[KEYPAD_ROWS][KEYPAD_COLS] = {
    {'1', '2', '3', 'A'},
    {'4', '5', '6', 'B'},
    {'7', '8', '9', 'C'},
    {'*', '0', '#', 'D'}
};

KeypadInput::KeypadInput(byte *rowPins, byte *colPins)
    : _keypad(makeKeymap(keymap), rowPins, colPins,
              KEYPAD_ROWS, KEYPAD_COLS) {}

void KeypadInput::init() {
    _keypad.setDebounceTime(20);
}

char KeypadInput::getKey() {
    return _keypad.getKey();
}
```

The debounce time of 20ms was chosen to balance reliability (filtering mechanical bounce) against responsiveness (staying well within the 100ms latency requirement). The Keypad library's internal state machine ensures that each physical key press produces exactly one `getKey()` event, regardless of contact bounce.

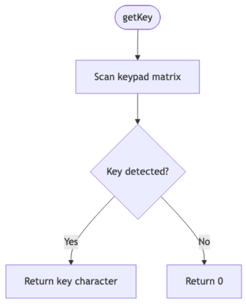


***Figure 2.10** – Flowchart: KeypadInput getKey function*

SRV Layer: LockFSM

The LockFSM is the core service module that implements all lock system logic through a 10-state finite state machine. It is designed to be hardware-independent: the FSM receives key characters via `processKey()` and outputs display content through a `LockDisplay` structure containing two 16-character strings. It does not directly interact with any peripheral — the application layer reads the display data and drives the LCD accordingly.

***Listing 2.8** – LockFSM.h — Lock FSM interface (key sections)*

```
#ifndef LOCK_FSM_H
#define LOCK_FSM_H
#include <Arduino.h>

static const uint8_t MAX_PWD_LEN = 8;
static const unsigned long RESULT_DISPLAY_MS = 2500;

enum LockFSMState {
    STATE_IDLE, STATE_MENU,
    STATE_LOCK_CONFIRM,
    STATE_UNLOCK_WAIT_STAR, STATE_UNLOCK_PWD,
    STATE_CHANGE_WAIT_STAR, STATE_CHANGE_OLD_PWD,
    STATE_CHANGE_NEW_PWD,
    STATE_STATUS_CONFIRM,
    STATE_SHOW_RESULT
};

struct LockDisplay {
    char line1[17];  // 16 chars + null
    char line2[17];
};

class LockFSM {
public:
    LockFSM();
    void init();
    void processKey(char key);
    void update();
    LockFSMState getState() const;
    bool isLocked() const;
    const LockDisplay& getDisplay() const;
    bool displayChanged() const;
    void clearDisplayChanged();
private:
    LockFSMState _state;
    bool _locked;
    char _password[MAX_PWD_LEN + 1];
    char _inputBuffer[MAX_PWD_LEN + 1];
    char _oldPwdBuffer[MAX_PWD_LEN + 1];
    uint8_t _inputLen;
    LockDisplay _display;
    bool _displayChanged;
    unsigned long _resultStartTime;
    void setState(LockFSMState newState);
    void updateDisplay();
    void setResult(const char *line1, const char *line2);
    void clearInput();
    void appendDigit(char digit);
};
#endif
```

The implementation processes each key press through a switch-case on the current state. The critical operations are password verification (using `strcmp()` against the stored password), password update (replacing the stored password after verifying the old one), and display content generation (building masked password strings with * characters for security). The `setResult()` method transitions to `STATE_SHOW_RESULT` and records the timestamp; the `update()` method checks if the display timeout (2.5 seconds) has elapsed and automatically returns to `STATE_IDLE`.

Listing 2.9 *– LockFSM.cpp — processKey implementation (UNLOCK_PWD state excerpt)*

```
case STATE_UNLOCK_PWD:
    if (key >= '0' && key <= '9') {
        appendDigit(key);
        updateDisplay();
    } else if (key == '#') {
        if (strcmp(_inputBuffer, _password) == 0) {
            _locked = false;
            setResult("Access Granted!", "Door is OPEN");
        } else {
            setResult("Wrong Password!", "Access Denied");
        }
        clearInput();
    } else if (key == '*') {
        clearInput();
        updateDisplay();
    }
    break;
```

APP Layer: Lab 1.2 Main

The application layer ties all modules together in a clean, readable structure. It defines the hardware pin mapping as static constants (single source of truth), creates instances of all drivers and the FSM, and implements the non-blocking main loop.

Listing 2.10 *– lab1_2_main.cpp — Application layer (key sections)*

```
#include "lab1_2_main.h"
#include <Arduino.h>
#include <stdio.h>
#include "Led.h"
#include "LcdDisplay.h"
#include "KeypadInput.h"
#include "LockFSM.h"
#include "StdioSerial.h"

// Pin Configuration (single source of truth)
static const uint8_t RED_LED_PIN   = 7;
static const uint8_t GREEN_LED_PIN = 6;
static const uint8_t LCD_I2C_ADDR  = 0x27;
static byte rowPins[4] = {22, 23, 24, 25};
static byte colPins[4] = {26, 27, 28, 29};

// Module instances
```

```
static Led redLed(RED_LED_PIN);
static Led greenLed(GREEN_LED_PIN);
static LcdDisplay lcd(LCD_I2C_ADDR, 16, 2);
static KeypadInput keypad(rowPins, colPins);
static LockFSM lockFSM;

void lab1_2Setup() {
    stdioSerialInit(9600);
    lcd.init();
    keypad.init();
    redLed.init();
    greenLed.init();
    lockFSM.init();
    redLed.turnOn();  // Start locked
    // Display initial FSM state
    lcd.showTwoLines(lockFSM.getDisplay().line1,
                     lockFSM.getDisplay().line2);
}

void lab1_2Loop() {
    char key = keypad.getKey();
    if (key) lockFSM.processKey(key);
    lockFSM.update();
    if (lockFSM.displayChanged()) {
        lcd.showTwoLines(lockFSM.getDisplay().line1,
                         lockFSM.getDisplay().line2);
        lockFSM.clearDisplayChanged();
    }
    // Update LEDs on state change (tracked via prevLocked)
    // Red=locked, Green=unlocked
}
```

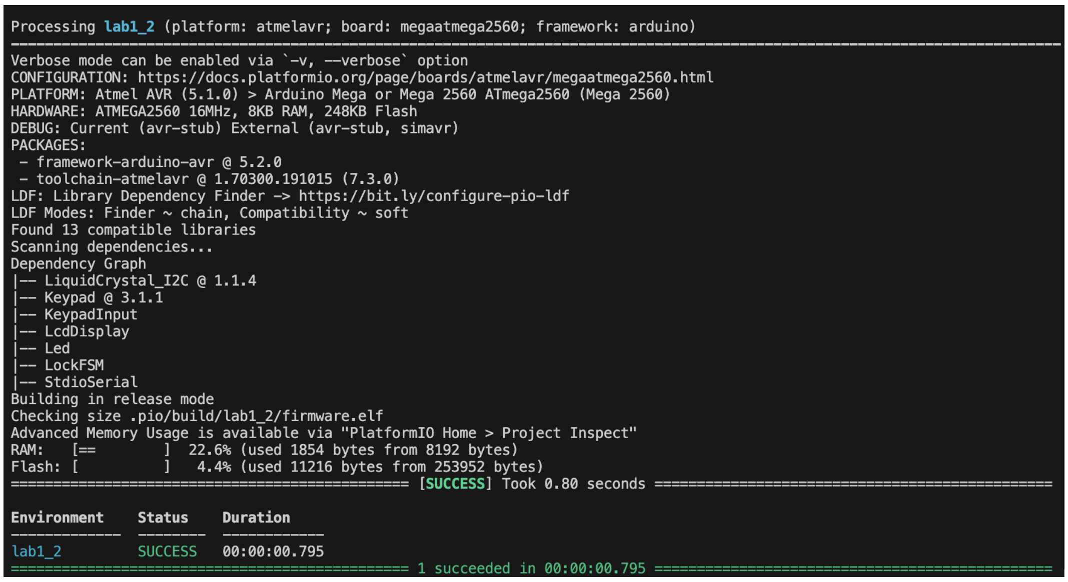
The application code is concise because all complexity is encapsulated in the FSM service layer. The main loop's responsibilities are limited to I/O routing: reading input from the keypad, passing it to the FSM, and reflecting the FSM's output on the LCD and LEDs. This separation means that changing the display technology (e.g., replacing the LCD with an OLED) would require modifying only the display driver and the application's update code, without touching the FSM logic.

3. Obtained Results

3.1. Build Process

The project was compiled using PlatformIO with the `lab1_2` environment targeting the Arduino Mega 2560 (ATmega2560). PlatformIO automatically resolved and installed the external library dependencies (LiquidCrystal_I2C 1.1.4 and Keypad 3.1.1) along with the Wire library required for I2C communication. The build completed successfully with no errors or warnings.

Resource utilization remains modest: the firmware occupies approximately 11.2 KB of Flash (4.4% of 248 KB) and 1.85 KB of RAM (22.6% of 8 KB). The RAM usage increase compared to Lab 1.1 is primarily due to the LCD library's internal buffer and the FSM's password and display buffers.



```
Processing lab1_2 (platform: atmelavr; board: megaatmega2560; framework: arduino)
--------------------------------------------------------------------------------------------------------------------------------
Verbose mode can be enabled via `-v, --verbose` option
CONFIGURATION: https://docs.platformio.org/page/boards/atmelavr/megaatmega2560.html
PLATFORM: Atmel AVR (5.1.0) > Arduino Mega or Mega 2560 ATmega2560 (Mega 2560)
HARDWARE: ATMEGA2560 16MHz, 8KB RAM, 248KB Flash
DEBUG: Current (avr-stub) External (avr-stub, simavr)
PACKAGES:
 - framework-arduino-avr @ 5.2.0
 - toolchain-atmelavr @ 1.70300.191015 (7.3.0)
LDF: Library Dependency Finder -> https://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Found 13 compatible libraries
Scanning dependencies...
Dependency Graph
|-- LiquidCrystal_I2C @ 1.1.4
|-- Keypad @ 3.1.1
|-- KeypadInput
|-- LcdDisplay
|-- Led
|-- LockFSM
|-- StdioSerial
Building in release mode
Checking size .pio/build/lab1_2/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [==        ]  22.6% (used 1854 bytes from 8192 bytes)
Flash: [          ]   4.4% (used 11216 bytes from 253952 bytes)
============================================== [SUCCESS] Took 0.80 seconds ==============================================

Environment    Status    Duration
-------------  --------  ------------
lab1_2         SUCCESS   00:00:00.795
============================================== 1 succeeded in 00:00:00.795 ==============================================
```

Figure 3.1 *– Successful PlatformIO build output for Lab 1.2*

Both lab environments (`lab1_1` and `lab1_2`) build successfully in parallel, confirming that lab isolation is maintained. The preprocessor-based lab selection mechanism (`-DLAB1_1`, `-DLAB1_2`) allows each environment to compile independently without code conflicts.

3.2. Simulation Initialization

Upon starting the Wokwi simulation, the Arduino Mega 2560 boots and initializes all peripherals. The LCD display shows the welcome screen ("Smart Lock" on line 1, "Press * to start" on line 2), the red LED illuminates (indicating LOCKED state), and the green LED remains off. The serial terminal displays the debug banner:

Listing 3.1 *– Serial debug output on startup*

```
========================================
  Lab 1.2: LCD + Keypad Lock System
  MCU: Arduino Mega 2560
========================================

Commands (via keypad):
  *0#          - Lock
  *1*pwd#      - Unlock with password
  *2*old*new#  - Change password
  *3#          - Show lock status

```

```
Default password: 1234
```

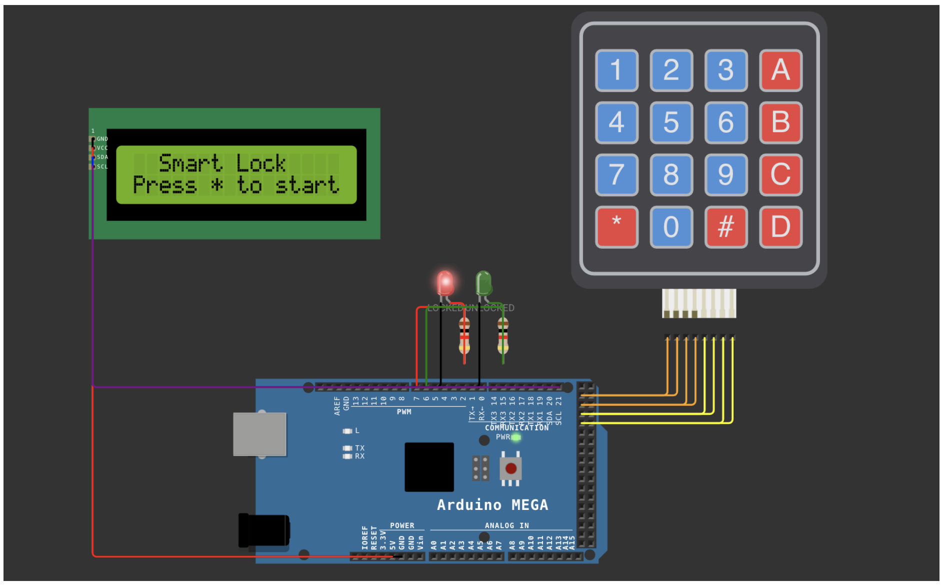


Figure 3.2 – Initial simulation state — LCD shows welcome, red LED on

3.3. Command Interface Validation

Test: Menu Display (Press *)

Pressing the * key transitions from IDLE to MENU state. The LCD updates to show the available commands on both lines: "0:Lock 1:Unlock" and "2:ChPwd 3:Status". The serial debug output confirms the state transition.

Listing 3.2 – Serial debug output — menu display

```
[KEY] '*' in state 0
[FSM] -> state 1
```

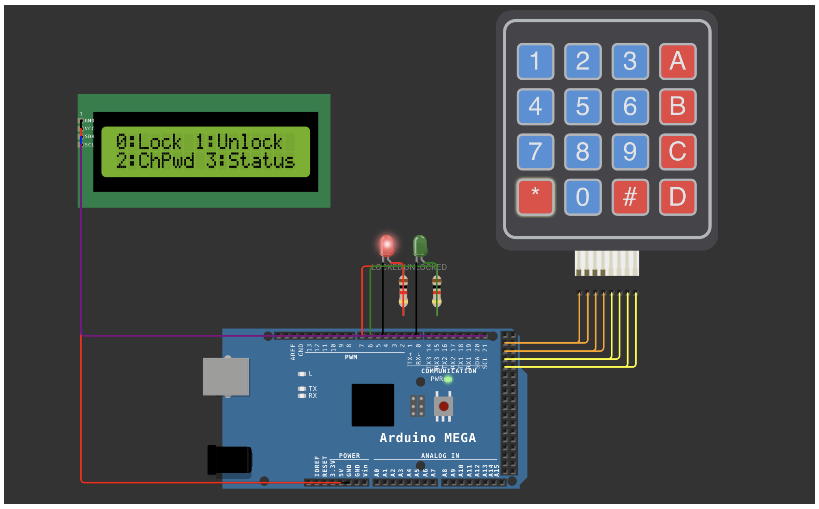


*Figure 3.3 – Menu display after pressing * key*

Test: Lock Command (*0#)

From the menu, pressing 0 shows the lock confirmation screen (“CMD: Lock” / “Press # to exec”). Pressing # executes the lock command: the lock state is set to LOCKED, the red LED turns on, and the LCD shows “Lock Activated” / “Door is LOCKED” for 2.5 seconds.

Listing 3.3 – Serial debug output — lock command

```
[KEY] '0' in state 1
[FSM] -> state 2
[KEY] '#' in state 2
[LOCK] Locked unconditionally
[FSM] Result: Lock Activated | Door is LOCKED
```

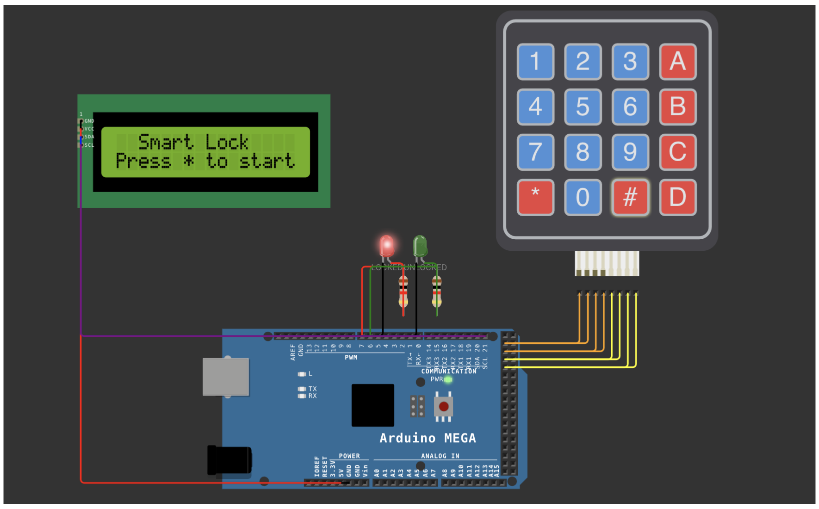


Figure 3.4 – Lock command execution — LCD confirmation and red LED

Test: Unlock with Correct Password (*1*1234#)

The unlock sequence demonstrates the full command flow: * enters the menu, 1 selects unlock, * initiates password entry, digits 1234 are entered (displayed as masked asterisks on LCD), and # executes the verification. With the correct password, the lock opens, the green LED turns on, and the LCD shows “Access Granted!” / “Door is OPEN”.

Listing 3.4 – Serial debug output — successful unlock

```
[KEY] '*' in state 0
[FSM] -> state 1
[KEY] '1' in state 1
[FSM] -> state 3
[KEY] '*' in state 3
[FSM] -> state 4
[KEY] '1' in state 4
[KEY] '2' in state 4
[KEY] '3' in state 4
[KEY] '4' in state 4
```

```
[KEY] '#' in state 4
[LOCK] Unlocked successfully
[FSM] Result: Access Granted! | Door is OPEN
[LED] Red OFF, Green ON (UNLOCKED)
```

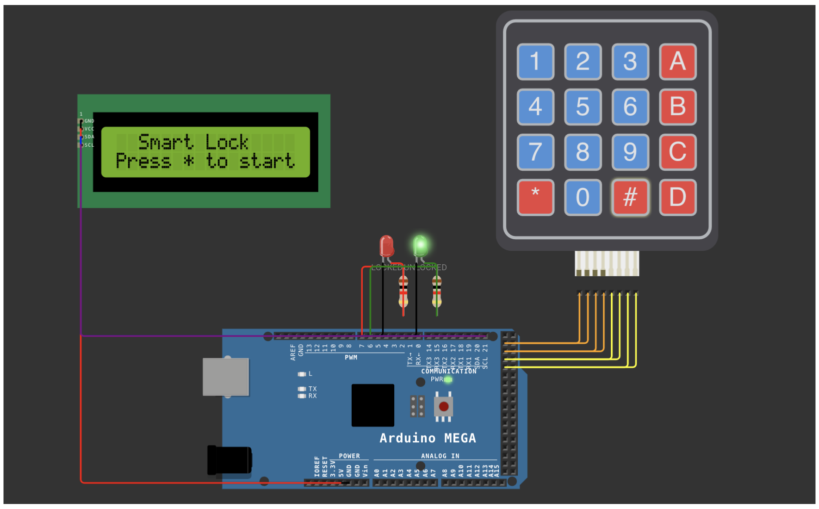


Figure 3.5 *– Successful unlock — green LED on, LCD shows "Access Granted!"*

Test: Unlock with Wrong Password (*1*9999#)

Entering an incorrect password (9999) triggers the error response. The lock remains in its current state, and the LCD shows "Wrong Password!" / "Access Denied" for 2.5 seconds.

Listing 3.5 *– Serial debug output — failed unlock attempt*

```
[KEY] '#' in state 4
[LOCK] Wrong password entered
[FSM] Result: Wrong Password! | Access Denied
```

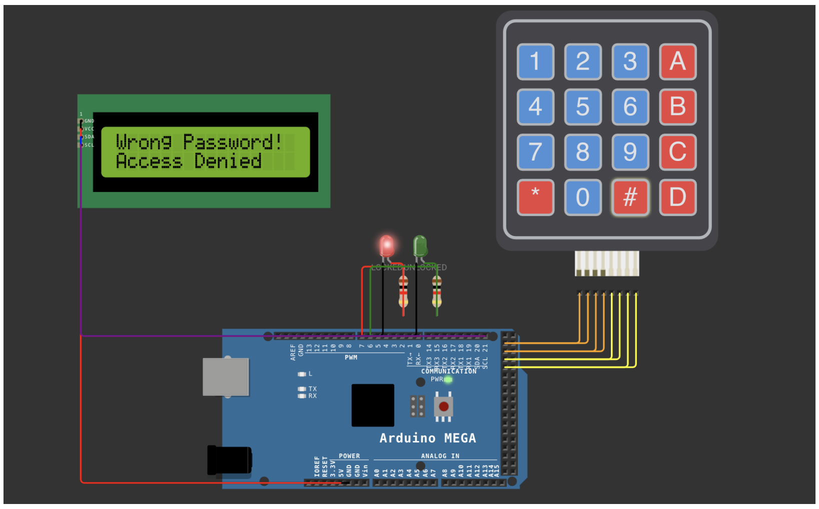


Figure 3.6 *– Failed unlock attempt — "Wrong Password!" message*

Test: Change Password (*2*1234*5678#)

The password change command requires three delimited fields: old password and new password. After entering the correct old password (1234) and a new password (5678), the system updates the stored password and confirms with “Pwd Changed!” / “Successfully”. Subsequent unlock attempts must use the new password.

Listing 3.6 *– Serial debug output — password change*

```
[KEY] '*' in state 0
[FSM] -> state 1
[KEY] '2' in state 1
[FSM] -> state 5
[KEY] '*' in state 5
[FSM] -> state 6
[KEY] '1' in state 6
[KEY] '2' in state 6
[KEY] '3' in state 6
[KEY] '4' in state 6
[KEY] '*' in state 6
[FSM] -> state 7
[KEY] '5' in state 7
[KEY] '6' in state 7
[KEY] '7' in state 7
[KEY] '8' in state 7
[KEY] '#' in state 7
[LOCK] Password changed to: 5678
[FSM] Result: Pwd Changed! | Successfully
```

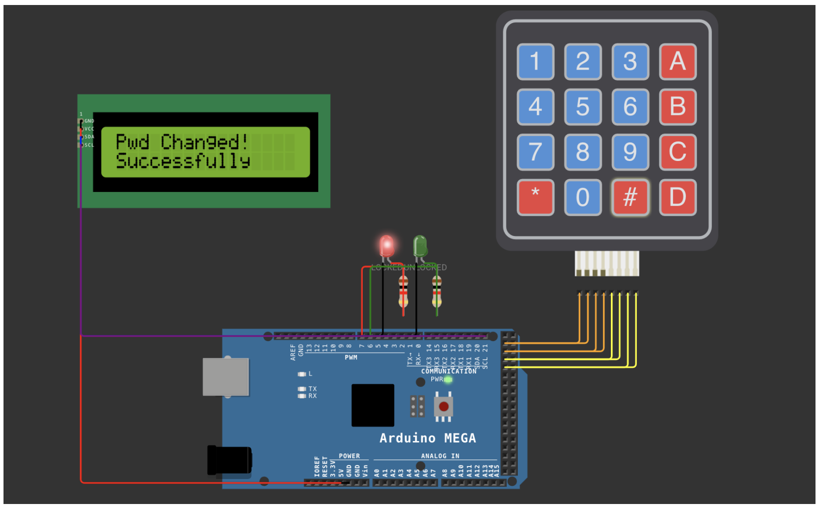


Figure 3.7 *– Password change confirmation*

Test: Status Query (*3#)

The status command displays the current lock state. When locked, the LCD shows “Lock Status:” / “** LOCKED **”; when unlocked, it shows “** UNLOCKED **”.

***Listing 3.7** – Serial debug output — status query*

```
[KEY] '*' in state 0
[FSM] -> state 1
[KEY] '3' in state 1
[FSM] -> state 8
[KEY] '#' in state 8
[LOCK] Status: LOCKED
[FSM] Result: Lock Status: | ** LOCKED **
```

Test: Invalid Command

Pressing an invalid key (e.g., `5`) in the MENU state produces an error message: "Invalid option!" / "Press * to start". The system recovers gracefully and returns to IDLE after the timeout.

Test: Password Masking

During password entry, each digit pressed appears as an asterisk (`*`) on the LCD, preventing visual eavesdropping. The first line shows the context ("Enter password:", "Old password:", or "New password:") while the second line accumulates mask characters.

3.4. System Observations

The system demonstrates several important operational characteristics. Response time is consistently well below the 100ms requirement: the non-blocking main loop executes in approximately 1–2ms when no key is pressed, and key-to-display update latency is imperceptible to the user. The keypad debouncing (20ms) effectively prevents false triggers while maintaining responsive feel.

The LCD update strategy (overwriting lines with space padding instead of calling `clear()`) prevents visible flickering during transitions. The 2.5-second result display timeout provides sufficient time for the user to read confirmation messages before the system returns to the idle state. Users can bypass this timeout by pressing any key.

Memory usage remains efficient: the FSM maintains three small character buffers (password, input, old password) of 9 bytes each, plus the 34-byte display structure. The total static RAM footprint is approximately 1.85 KB, leaving over 6 KB available for future extensions.

The serial debug output provides complete traceability of all key presses, state transitions, and command results, which is valuable for both development and troubleshooting.

3.5. Verification Summary

- **Compilation:** Both `lab1_1` and `lab1_2` environments build successfully with no errors.
- **Initialization:** LCD, keypad, LEDs, and FSM initialize correctly. Welcome screen displays on startup.

- **Lock command (*0#):** Correctly activates lock with confirmation message and red LED.
- **Unlock command (*1*pwd#):** Correct password unlocks (green LED); wrong password produces error.
- **Change password (*2*old*new#):** Updates password after verifying old; rejects incorrect old password.
- **Status query (*3#):** Correctly displays current lock state.
- **Menu display:** Each * press shows context-appropriate options on LCD.
- **Error handling:** Invalid keys, incomplete commands, and wrong passwords produce clear error messages.
- **Password masking:** Digits displayed as * characters during entry.
- **LED feedback:** Red LED for locked, green for unlocked, transitions correctly.
- **Debouncing:** 20ms debounce prevents false key triggers.
- **Response time:** Well below 100ms latency requirement.
- **Lab isolation:** Lab 1.1 builds and runs independently of Lab 1.2.

4. Conclusion

4.1. Achievement of Objectives

All primary objectives of Laboratory Work 1.2 were successfully accomplished. The I2C communication protocol was used to interface a 16x2 LCD display with the Arduino Mega 2560, enabling clear visual feedback through context-aware menu displays, password masking, and confirmation messages. The 4x4 matrix keypad was integrated with 20ms software debouncing for reliable key detection, using the asterisk (*) as a field delimiter and hash (#) as a command executor. A 10-state finite state machine was designed and implemented to manage the complete lock system logic — including unconditional locking, password-based unlocking, password changes with verification, and status queries — with clean separation between the FSM service and hardware drivers. The modular software architecture follows a five-layer design (APP, SRV, ECAL, MCAL, HW) where two library modules from Lab 1.1 (Led, StdioSerial) are reused without modification, and three new modules (LcdDisplay, KeypadInput, LockFSM) extend the library collection. Lab isolation is maintained through PlatformIO's multi-environment build system with preprocessor-based lab selection, ensuring all previous labs remain fully functional.

4.2. Performance Analysis and System Limitations

The system performs well within its design parameters. The non-blocking main loop achieves sub-millisecond cycle times, providing input-to-display latency well below the 100ms requirement. Memory usage is efficient at 22.6% RAM and 4.4% Flash, leaving substantial headroom for additional features. The keypad debouncing reliably filters contact bounce while maintaining responsive input.

However, the system has several limitations. The password is stored in volatile SRAM and resets to the default "1234" on every power cycle or reset, as there is no EEPROM persistence. The maximum password length is limited to 8 digits, and passwords can only contain numeric characters (0–9), reducing the keyspace. The system does not implement lockout protection after multiple failed unlock attempts, which would be essential in a real security application. The LCD's 16-character line width constrains message length, requiring abbreviated text in some states. The FSM uses blocking `printf()` calls for debug output, which could introduce minor latency in the serial transmission path (though this is negligible at 9600 baud for short messages).

4.3. Proposed Improvements

Several enhancements could address the current limitations and extend functionality. EEPROM password persistence would retain the password across power cycles by storing it in the ATmega2560's 4 KB EEPROM. A failed attempt counter with exponential lockout (e.g., 30-second lockout after 3 failed attempts, doubling with each subsequent failure) would significantly improve security. Alphanumeric password support using the keypad's A–D keys would expand the keyspace from 10^8 to 14^8 combinations. An audible buzzer could provide additional feedback for key presses, successful operations, and error conditions. A real-time clock module could enable time-based access control (automatic lock/unlock schedules) and event logging with timestamps. Multi-user support with separate passwords for different access levels (admin for password changes, user for unlock only) would add practical utility. Finally, a timeout mechanism in the FSM's intermediate states would automatically return to IDLE if no key is pressed within a configurable period, preventing the system from remaining in a half-entered command state indefinitely.

4.4. Reflections on the Learning Experience

This laboratory work provided valuable experience in several key areas of embedded systems development. Designing the 10-state FSM required careful planning of all possible state transitions, including error conditions and edge cases, which reinforced the importance of thorough upfront design before implementation. The FSM pattern proved extremely effective for managing complex interactive behavior — each state's logic is self-contained and independently testable, making debugging straightforward.

The transition from Lab 1.1's blocking I/O model (`fgets()` waiting for complete input) to Lab 1.2's non-blocking event-driven model (`getKey()` returning immediately) highlighted a fundamental design pattern in embedded systems: responsive user interfaces require cooperative multitasking where each task voluntarily yields control quickly. This concept scales directly to professional embedded firmware where multiple subsystems must share a single CPU.

Working with the I2C protocol demonstrated the power of standardized communication buses — connecting the LCD required only two signal wires and a well-established library, compared to the 12+ wires that would be needed for a parallel interface. The experience of wrapping third-party libraries behind project-specific interfaces (LcdDisplay wrapping LiquidCrystal_I2C, KeypadInput wrapping Keypad) illustrated the adapter pattern, which isolates application code from library-specific APIs and makes future library replacements straightforward.

4.5. Impact of Technology in Real-World Applications

The keypad-plus-LCD interface pattern implemented in this lab is among the most widely deployed human-machine interfaces in embedded systems. Security panels from manufacturers like DSC, Honeywell, and Bosch use nearly identical command protocols where * accesses function menus and # confirms operations. ATM machines use numeric keypads with LCD displays for PIN entry and transaction selection, employing the same password masking technique demonstrated here. Industrial equipment (CNC machines, PLCs, HVAC controllers) commonly uses membrane keypads with character LCDs for parameter configuration and status monitoring.

The FSM architecture used in this lab directly maps to the state machine designs used in professional security and access control systems, where deterministic state transitions and comprehensive error handling are essential for reliable operation. The modular software architecture (with hardware drivers separated from application logic through abstraction layers) reflects the AUTOSAR methodology used in automotive embedded systems, where standardization of software components enables reuse across different vehicle platforms and hardware variants.

5. Note Regarding Usage of AI

During the preparation of this laboratory report, the author utilized artificial intelligence tools to enhance the quality and presentation of the documentation. Specifically, the following AI tools were employed:

- **Report Formatting:** Assistance with LaTeX document structure, formatting consistency, and adherence to academic writing standards.
- **Content Rephrasing:** Refinement of technical descriptions and explanations to ensure grammatical accuracy and clarity without altering the technical substance.
- **Code Documentation:** Generation of inline comments and Doxygen-style documentation within the source code to improve readability and maintainability.
- **Diagram Generation:** Creation of Mermaid diagram source files for system architecture, FSM state diagrams, and flowcharts.

- **Development Environment Setup:** Configuration assistance for PlatformIO multi-environment builds with I2C LCD and Keypad library integration.

All AI-generated content was thoroughly reviewed, validated, and adjusted by the author to ensure technical accuracy, alignment with project requirements, and adherence to embedded systems best practices. The core technical implementation, circuit design, FSM architecture decisions, and experimental results represent the author's original work and understanding of the subject matter. The AI tools served as assistive technologies to enhance documentation quality and development workflow efficiency, while the author retained full responsibility for the content, correctness, and conclusions presented in this report.

6. Bibliography

[1] Bragarenco, A., Astafi, V., *Sisteme Electronice Încorporate: Indicaţii metodice pentru lucrări de laborator*, Universitatea Tehnică a Moldovei, Chişinău, 2024.

[2] Atmel Corporation, *ATmega2560 Datasheet — 8-bit AVR Microcontroller with 256K Bytes In-System Programmable Flash*, Document 2549Q–AVR–02/2014, 2014.

[3] NXP Semiconductors, *I2C-bus Specification and User Manual*, Document UM10204, Rev' 7.0, `https://www.nxp.com/docs/en/user-guide/UM10204.pdf`, Accessed: 2026-02-18.

[4] Arduino, *Arduino Mega 2560 Rev3 Documentation*, `https://docs.arduino.cc/hardware/uno-rev3/`, Accessed: 2026-02-18.

[5] Schwartz, M., *LiquidCrystal_I2C Library for Arduino*, `https://github.com/johnrickman/LiquidCrystal_I2C`, Accessed: 2026-02-18.

[6] Stanley, M., Brevig, A., *Keypad Library for Arduino*, `https://github.com/Chris--A/Keypad`, Accessed: 2026-02-18.

[7] PlatformIO, *Professional Collaborative Platform for Embedded Development*, `https://platformio.org/`, Accessed: 2026-02-18.

[8] Wokwi, *Online Electronics Simulator — Arduino, ESP32, Raspberry Pi Pico*, `https://wokwi.com/`, Accessed: 2026-02-18.

7. Appendix

The complete source code for Laboratory Work 1.2 is organized below by architectural layer. The full project is available in the GitHub repository: `https://github.com/mcittkmims/es-labs`.

7.1. Application Layer

Entry Point: main.cpp

Listing 7.1 – *src/main.cpp — Application entry point (preprocessor lab selector)*

```
/**
 * @file main.cpp
 * @brief Application Entry Point - Lab Selector
 *
 * The active lab is selected via preprocessor defines set in
 * platformio.ini (-DLAB1_1, -DLAB1_2). Each PlatformIO environment
 * builds and runs exactly one lab without code changes.
 */

#include <Arduino.h>

#if defined(LAB1_1)
    #include "lab1_1_main.h"
#elif defined(LAB1_2)
    #include "lab1_2_main.h"
#else
    #error "No lab selected! Add -DLABx_y to build_flags."
#endif

void setup() {
#if defined(LAB1_1)
    lab1_1Setup();
#elif defined(LAB1_2)
    lab1_2Setup();
#endif
}

void loop() {
#if defined(LAB1_1)
    lab1_1Loop();
#elif defined(LAB1_2)
    lab1_2Loop();
#endif
}
```

Lab 1.2 Main Module

Listing 7.2 – *lab/lab1_2/lab1_2_main.h — Lab 1.2 interface*

```
#ifndef LAB1_2_MAIN_H
#define LAB1_2_MAIN_H
```

```
void lab1_2Setup();
void lab1_2Loop();

#endif // LAB1_2_MAIN_H
```

Listing 7.3 – lab/lab1_2/lab1_2_main.cpp — Lab 1.2 implementation

```
#include "lab1_2_main.h"
#include <Arduino.h>
#include <stdio.h>
#include "Led.h"
#include "LcdDisplay.h"
#include "KeypadInput.h"
#include "LockFSM.h"
#include "StdioSerial.h"

// Pin Configuration
static const uint8_t RED_LED_PIN   = 7;
static const uint8_t GREEN_LED_PIN = 6;
static const uint8_t LCD_I2C_ADDR  = 0x27;
static const uint8_t LCD_COLS      = 16;
static const uint8_t LCD_ROWS      = 2;
static const unsigned long BAUD_RATE = 9600;
static byte rowPins[4] = {22, 23, 24, 25};
static byte colPins[4] = {26, 27, 28, 29};

// Module instances
static Led redLed(RED_LED_PIN);
static Led greenLed(GREEN_LED_PIN);
static LcdDisplay lcd(LCD_I2C_ADDR, LCD_COLS, LCD_ROWS);
static KeypadInput keypad(rowPins, colPins);
static LockFSM lockFSM;
static bool prevLocked = true;

void lab1_2Setup() {
    stdioSerialInit(BAUD_RATE);
    lcd.init();
    keypad.init();
    redLed.init();
    greenLed.init();
    lockFSM.init();
    redLed.turnOn();
    greenLed.turnOff();
    prevLocked = true;

    const LockDisplay& disp = lockFSM.getDisplay();
    lcd.showTwoLines(disp.line1, disp.line2);
    lockFSM.clearDisplayChanged();

```

```cpp
    printf("\r\n");
    printf("========================================\r\n");
    printf("  Lab 1.2: LCD + Keypad Lock System\r\n");
    printf("  MCU: Arduino Mega 2560\r\n");
    printf("========================================\r\n");
    printf("\r\n");
    printf("Commands (via keypad):\r\n");
    printf("  *0#         - Lock\r\n");
    printf("  *1*pwd#     - Unlock with password\r\n");
    printf("  *2*old*new# - Change password\r\n");
    printf("  *3#         - Show lock status\r\n");
    printf("\r\n");
    printf("Default password: 1234\r\n");
    printf("\r\n");
}

void lab1_2Loop() {
    char key = keypad.getKey();
    if (key) {
        lockFSM.processKey(key);
    }

    lockFSM.update();

    if (lockFSM.displayChanged()) {
        const LockDisplay& disp = lockFSM.getDisplay();
        lcd.showTwoLines(disp.line1, disp.line2);
        lockFSM.clearDisplayChanged();
    }

    bool currentLocked = lockFSM.isLocked();
    if (currentLocked != prevLocked) {
        if (currentLocked) {
            redLed.turnOn();
            greenLed.turnOff();
            printf("[LED] Red ON, Green OFF (LOCKED)\r\n");
        } else {
            redLed.turnOff();
            greenLed.turnOn();
            printf("[LED] Red OFF, Green ON (UNLOCKED)\r\n");
        }
        prevLocked = currentLocked;
    }
}
```

7.2. Service Layer

LockFSM Module

***Listing 7.4** – lib/LockFSM/LockFSM.h — Lock FSM interface*

```
#ifndef LOCK_FSM_H
#define LOCK_FSM_H
#include <Arduino.h>

static const uint8_t MAX_PWD_LEN = 8;
static const unsigned long RESULT_DISPLAY_MS = 2500;

enum LockFSMState {
    STATE_IDLE,
    STATE_MENU,
    STATE_LOCK_CONFIRM,
    STATE_UNLOCK_WAIT_STAR,
    STATE_UNLOCK_PWD,
    STATE_CHANGE_WAIT_STAR,
    STATE_CHANGE_OLD_PWD,
    STATE_CHANGE_NEW_PWD,
    STATE_STATUS_CONFIRM,
    STATE_SHOW_RESULT
};

struct LockDisplay {
    char line1[17];
    char line2[17];
};

class LockFSM {
public:
    LockFSM();
    void init();
    void processKey(char key);
    void update();
    LockFSMState getState() const;
    bool isLocked() const;
    const LockDisplay& getDisplay() const;
    bool displayChanged() const;
    void clearDisplayChanged();
private:
    LockFSMState _state;
    bool _locked;
    char _password[MAX_PWD_LEN + 1];
    char _inputBuffer[MAX_PWD_LEN + 1];
    char _oldPwdBuffer[MAX_PWD_LEN + 1];
    uint8_t _inputLen;
    LockDisplay _display;
    bool _displayChanged;
    unsigned long _resultStartTime;
    void setState(LockFSMState newState);
    void updateDisplay();
    void setResult(const char *line1, const char *line2);
    void clearInput();
    void appendDigit(char digit);
};
#endif
```

Listing 7.5 – lib/LockFSM/LockFSM.cpp — Lock FSM implementation

```
#include "LockFSM.h"
#include <string.h>
#include <stdio.h>

static const char DEFAULT_PASSWORD[] = "1234";

LockFSM::LockFSM()
    : _state(STATE_IDLE), _locked(true), _inputLen(0),
      _displayChanged(true), _resultStartTime(0) {
    strncpy(_password, DEFAULT_PASSWORD, MAX_PWD_LEN);
    _password[MAX_PWD_LEN] = '\0';
    _inputBuffer[0] = '\0';
    _oldPwdBuffer[0] = '\0';
    memset(&_display, 0, sizeof(_display));
}

void LockFSM::init() {
    _state = STATE_IDLE;
    _locked = true;
    strncpy(_password, DEFAULT_PASSWORD, MAX_PWD_LEN);
    _password[MAX_PWD_LEN] = '\0';
    clearInput();
    _oldPwdBuffer[0] = '\0';
    updateDisplay();
    printf("[FSM] Initialized. Default password: %s\r\n",
           DEFAULT_PASSWORD);
}

void LockFSM::processKey(char key) {
    if (key == 0) return;
    printf("[KEY] '%c' in state %d\r\n", key, (int)_state);

    switch (_state) {
        case STATE_IDLE:
            if (key == '*') setState(STATE_MENU);
            break;
        case STATE_MENU:
            if (key == '0') setState(STATE_LOCK_CONFIRM);
            else if (key == '1') setState(STATE_UNLOCK_WAIT_STAR);
            else if (key == '2') setState(STATE_CHANGE_WAIT_STAR);
            else if (key == '3') setState(STATE_STATUS_CONFIRM);
            else if (key == '*') updateDisplay();
            else setResult("Invalid option!", "Press * to start");
            break;

        case STATE_LOCK_CONFIRM:
            if (key == '#') {
                _locked = true;
                printf("[LOCK] Locked unconditionally\r\n");
                setResult("Lock Activated", "Door is LOCKED");
            } else if (key == '*') setState(STATE_MENU);
            break;

        case STATE_UNLOCK_WAIT_STAR:
            if (key == '*') {
                clearInput();
                setState(STATE_UNLOCK_PWD);
            } else if (key == '#')
                setResult("Error: need pwd", "Use *1*pwd#");
            break;

        case STATE_UNLOCK_PWD:
            if (key >= '0' && key <= '9') {
                appendDigit(key);
                updateDisplay();
            } else if (key == '#') {
                if (strcmp(_inputBuffer, _password) == 0) {
                    _locked = false;
                    printf("[LOCK] Unlocked successfully\r\n");
                    setResult("Access Granted!", "Door is OPEN");
                } else {
                    printf("[LOCK] Wrong password entered\r\n");
                    setResult("Wrong Password!", "Access Denied");
                }
                clearInput();
            } else if (key == '*') {
                clearInput();
                updateDisplay();
            }
            break;

        case STATE_CHANGE_WAIT_STAR:
            if (key == '*') {
```

```
            clearInput();
            setState(STATE_CHANGE_OLD_PWD);
        } else if (key == '#')
            setResult("Error: need pwd", "Use *2*old*new#");
        break;

    case STATE_CHANGE_OLD_PWD:
        if (key >= '0' && key <= '9') {
            appendDigit(key);
            updateDisplay();
        } else if (key == '*') {
            strncpy(_oldPwdBuffer, _inputBuffer, MAX_PWD_LEN);
            _oldPwdBuffer[MAX_PWD_LEN] = '\0';
            clearInput();
            setState(STATE_CHANGE_NEW_PWD);
        } else if (key == '#')
            setResult("Error: need new", "Use *2*old*new#");
        break;

    case STATE_CHANGE_NEW_PWD:
        if (key >= '0' && key <= '9') {
            appendDigit(key);
            updateDisplay();
        } else if (key == '#') {
            if (strcmp(_oldPwdBuffer, _password) == 0) {
                if (_inputLen > 0) {
                    strncpy(_password, _inputBuffer,
                            MAX_PWD_LEN);
                    _password[MAX_PWD_LEN] = '\0';
                    printf("[LOCK] Password changed to: %s\r\n",
                           _password);
                    setResult("Pwd Changed!", "Successfully");
                } else {
                    setResult("Error: empty pw", "Try again");
                }
            } else {
                printf("[LOCK] Wrong old password\r\n");
                setResult("Wrong Old Pwd!", "Change Denied");
            }
            clearInput();
            _oldPwdBuffer[0] = '\0';
        } else if (key == '*') {
            clearInput();
            updateDisplay();
        }
        break;

    case STATE_STATUS_CONFIRM:
            if (key == '#') {
                if (_locked) {
                    printf("[LOCK] Status: LOCKED\r\n");
                    setResult("Lock Status:", "** LOCKED **");
                } else {
                    printf("[LOCK] Status: UNLOCKED\r\n");
                    setResult("Lock Status:", "** UNLOCKED **");
                }
            } else if (key == '*') setState(STATE_MENU);
            break;

        case STATE_SHOW_RESULT:
            if (key == '*') setState(STATE_MENU);
            else setState(STATE_IDLE);
            break;
    }
}

void LockFSM::update() {
    if (_state == STATE_SHOW_RESULT) {
        if (millis() - _resultStartTime >= RESULT_DISPLAY_MS) {
            setState(STATE_IDLE);
        }
    }
}

LockFSMState LockFSM::getState() const { return _state; }
bool LockFSM::isLocked() const { return _locked; }
const LockDisplay& LockFSM::getDisplay() const {
    return _display;
}
bool LockFSM::displayChanged() const {
    return _displayChanged;
}
void LockFSM::clearDisplayChanged() {
    _displayChanged = false;
}

void LockFSM::setState(LockFSMState newState) {
    _state = newState;
    printf("[FSM] -> state %d\r\n", (int)_state);
    updateDisplay();
}

void LockFSM::updateDisplay() {
    _displayChanged = true;
    switch (_state) {
        case STATE_IDLE:
            strncpy(_display.line1, "  Smart Lock    ", 17);
            strncpy(_display.line2, "Press * to start", 17);
            break;
        case STATE_MENU:
            strncpy(_display.line1, "0:Lock 1:Unlock ", 17);
            strncpy(_display.line2, "2:ChPwd 3:Status", 17);
            break;
        case STATE_LOCK_CONFIRM:
            strncpy(_display.line1, "CMD: Lock       ", 17);
            strncpy(_display.line2, "Press # to exec ", 17);
            break;
        case STATE_UNLOCK_WAIT_STAR:
            strncpy(_display.line1, "CMD: Unlock     ", 17);
            strncpy(_display.line2, "Press * for pwd ", 17);
            break;
        case STATE_UNLOCK_PWD:
        case STATE_CHANGE_OLD_PWD:
        case STATE_CHANGE_NEW_PWD: {
            if (_state == STATE_UNLOCK_PWD)
                strncpy(_display.line1, "Enter password: ", 17);
            else if (_state == STATE_CHANGE_OLD_PWD)
                strncpy(_display.line1, "Old password:   ", 17);
            else
                strncpy(_display.line1, "New password:   ", 17);
            char mask[17];
            uint8_t i;
            for (i = 0; i < _inputLen && i < 16; i++)
                mask[i] = '*';
            for (; i < 16; i++)
                mask[i] = ' ';
            mask[16] = '\0';
            strncpy(_display.line2, mask, 17);
            break;
        }
        case STATE_CHANGE_WAIT_STAR:
            strncpy(_display.line1, "CMD: Change Pwd ", 17);
            strncpy(_display.line2, "Press * for pwd ", 17);
            break;
        case STATE_STATUS_CONFIRM:
            strncpy(_display.line1, "CMD: Status     ", 17);
            strncpy(_display.line2, "Press # to exec ", 17);
            break;
        case STATE_SHOW_RESULT:
            break; // Already set by setResult()
    }
}

void LockFSM::setResult(const char *line1, const char *line2) {
    _state = STATE_SHOW_RESULT;
    _resultStartTime = millis();
    strncpy(_display.line1, line1, 16);
    _display.line1[16] = '\0';
    strncpy(_display.line2, line2, 16);
    _display.line2[16] = '\0';
    _displayChanged = true;
    printf("[FSM] Result: %s | %s\r\n",
           _display.line1, _display.line2);
}

void LockFSM::clearInput() {
    _inputBuffer[0] = '\0';
    _inputLen = 0;
}

void LockFSM::appendDigit(char digit) {
    if (_inputLen < MAX_PWD_LEN) {
        _inputBuffer[_inputLen] = digit;
        _inputLen++;
        _inputBuffer[_inputLen] = '\0';
    }
}
```

7.3. ECU Abstraction Layer

LcdDisplay Module

Listing 7.6 – lib/LcdDisplay/LcdDisplay.h — LCD driver interface

```
#ifndef LCD_DISPLAY_H
#define LCD_DISPLAY_H
#include <Arduino.h>
#include <LiquidCrystal_I2C.h>

class LcdDisplay {
public:
    LcdDisplay(uint8_t i2cAddress, uint8_t cols, uint8_t rows);
    void init();
    void clear();
    void setCursor(uint8_t col, uint8_t row);
    void print(const char *text);
    void printLine(uint8_t row, const char *text);
    void showTwoLines(const char *line1, const char *line2);
    void backlight(bool on);
private:
    LiquidCrystal_I2C _lcd;
    uint8_t _cols;
    uint8_t _rows;
};
#endif
```

***Listing 7.7** – lib/LcdDisplay/LcdDisplay.cpp — LCD driver implementation*

```
#include "LcdDisplay.h"
#include <string.h>

LcdDisplay::LcdDisplay(uint8_t i2cAddress, uint8_t cols,
                       uint8_t rows)
    : _lcd(i2cAddress, cols, rows),
      _cols(cols), _rows(rows) {}

void LcdDisplay::init() {
    _lcd.init();
    _lcd.backlight();
    _lcd.clear();
}

void LcdDisplay::clear() { _lcd.clear(); }

void LcdDisplay::setCursor(uint8_t col, uint8_t row) {
    _lcd.setCursor(col, row);
}

void LcdDisplay::print(const char *text) { _lcd.print(text); }

void LcdDisplay::printLine(uint8_t row, const char *text) {
    _lcd.setCursor(0, row);
    uint8_t len = strlen(text);
    if (len > _cols) len = _cols;
    for (uint8_t i = 0; i < len; i++) _lcd.write(text[i]);
    for (uint8_t i = len; i < _cols; i++) _lcd.write(' ');
}

void LcdDisplay::showTwoLines(const char *line1,
                              const char *line2) {
    printLine(0, line1);
    printLine(1, line2);
}

void LcdDisplay::backlight(bool on) {
    if (on) _lcd.backlight();
    else _lcd.noBacklight();
}
```

KeypadInput Module

Listing 7.8 – lib/KeypadInput/KeypadInput.h — Keypad driver interface

```
#ifndef KEYPAD_INPUT_H
#define KEYPAD_INPUT_H
#include <Arduino.h>
#include <Keypad.h>

static const byte KEYPAD_ROWS = 4;
static const byte KEYPAD_COLS = 4;

class KeypadInput {
public:
    KeypadInput(byte *rowPins, byte *colPins);
    void init();
    char getKey();
private:
    Keypad _keypad;
};
#endif
```

Listing 7.9 – lib/KeypadInput/KeypadInput.cpp — Keypad driver implementation

```
#include "KeypadInput.h"

static char keymap[KEYPAD_ROWS][KEYPAD_COLS] = {
    {'1', '2', '3', 'A'},
    {'4', '5', '6', 'B'},
    {'7', '8', '9', 'C'},
    {'*', '0', '#', 'D'}
};

KeypadInput::KeypadInput(byte *rowPins, byte *colPins)
    : _keypad(makeKeymap(keymap), rowPins, colPins,
              KEYPAD_ROWS, KEYPAD_COLS) {}

void KeypadInput::init() { _keypad.setDebounceTime(20); }

char KeypadInput::getKey() { return _keypad.getKey(); }
```

StdioSerial Module (Reused from Lab 1.1)

Listing 7.10 – lib/StdioSerial/StdioSerial.h — STDIO redirection interface

```
#ifndef STDIO_SERIAL_H
#define STDIO_SERIAL_H
#include <Arduino.h>
#include <stdio.h>
```

```
void stdioSerialInit(unsigned long baudRate);
#endif
```

Listing 7.11 – lib/StdioSerial/StdioSerial.cpp — STDIO redirection implementation

```
#include "StdioSerial.h"

static int serialPutChar(char c, FILE *stream) {
    Serial.write(c);
    return 0;
}

static int serialGetChar(FILE *stream) {
    while (!Serial.available()) { }
    char c = Serial.read();
    if (c == '\r') {
        Serial.write('\r');
        Serial.write('\n');
        return '\n';
    }
    if (c == '\b' || c == 127) {
        Serial.write('\b');
        Serial.write(' ');
        Serial.write('\b');
        return c;
    }
    Serial.write(c);
    return c;
}

static FILE serialStream;

void stdioSerialInit(unsigned long baudRate) {
    Serial.begin(baudRate);
    while (!Serial) { ; }
    fdev_setup_stream(&serialStream,
        serialPutChar, serialGetChar, _FDEV_SETUP_RW);
    stdout = &serialStream;
    stdin  = &serialStream;
}
```

7.4. Microcontroller Abstraction Layer

Led Driver Module (Reused from Lab 1.1)

Listing 7.12 – lib/Led/Led.h — LED driver interface

```
#ifndef LED_H
#define LED_H
#include <Arduino.h>

class Led {
public:
    Led(uint8_t pin);
    void init();
    void turnOn();
    void turnOff();
    void toggle();
    bool isOn() const;
private:
    uint8_t ledPin;
    bool state;
};
#endif
```

***Listing 7.13** – lib/Led/Led.cpp — LED driver implementation*

```
#include "Led.h"

Led::Led(uint8_t pin) : ledPin(pin), state(false) {}

void Led::init() {
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);
    state = false;
}

void Led::turnOn()  { digitalWrite(ledPin, HIGH); state = true;  }
void Led::turnOff() { digitalWrite(ledPin, LOW);  state = false; }

void Led::toggle() {
    if (state) turnOff();
    else turnOn();
}

bool Led::isOn() const { return state; }
```

7.5. Configuration Files

***Listing 7.14** – platformio.ini — PlatformIO project configuration*

```
; Lab 1.1
[env:lab1_1]
platform = atmelavr
board = megaatmega2560
```

```
framework = arduino
monitor_speed = 9600
build_src_filter = +<*> +<../lab/lab1_1/*>
build_flags = -I lab/lab1_1 -DLAB1_1

; Lab 1.2
[env:lab1_2]
platform = atmelavr
board = megaatmega2560
framework = arduino
monitor_speed = 9600
build_src_filter = +<*> +<../lab/lab1_2/*>
build_flags = -I lab/lab1_2 -DLAB1_2
lib_deps =
    marcoschwartz/LiquidCrystal_I2C@^1.1.4
    chris--a/Keypad@^3.1.1
```