Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Embedded Systems

Laboratory work work #1.2

User interaction. STDIO. LCD. Keypad.

During the preparation of this report, the author used various AI Tools such as ChatGPT, Gemini, Claude to generate/augment the content, generate the code and structure the directory. The resulting information was reviewed, validated, and adjusted to meet the requirements of the laboratory assignment.

Author: Timur Cravtov
std. gr. FAF-231

Verified by:
Alexei Martîniuc
asist. univ

Chișinău, 2026

Contents

1 Objective and Technical analysis

This laboratory work aims at designing a embedded system that allows user interaction using a keypad. The user will be able to input a pin code using the keypad, and the system will validate the code and provide feedback through an LCD display. The system will be implemented on an Arduino board, utilizing the serial communication interface for user interaction. The main components of the system include a keypad for user input, an LCD display for output, and an LED to indicate the status of the pin code validation. The system will be designed with a layered architecture to ensure modularity and separation of concerns, making it easier to maintain and extend in the future. The implementation will involve writing drivers for both the keypad and the LCD display, as well as a main program that handles the logic of validating the pin code and controlling the LED based on the validation results.

- Get familiar with the basics of user interaction peripherals, such as keypads and LCD displays.
- Utilize the STDIO library for handling text-based input and output operations.
- Design an application that interprets commands sent through a keypad and displays output on an
- Develop a modular solution with separate functionalities for controlling the peripherals, ensuring maintainability and scalability of the system.

1.1 LCD display

LCD (Liquid Crystal Display) is a flat-panel display technology that uses liquid crystals to produce images [13]. It is commonly used in embedded systems for displaying information to the user. The LCD display in this project will be used to show messages related to the pin code validation process, such as prompts for input and feedback on whether the entered pin code is correct or incorrect. LCD displays differ in size, resolution, and interface types, but for this project, a common 16x2 character LCD with an I2C interface will be used.

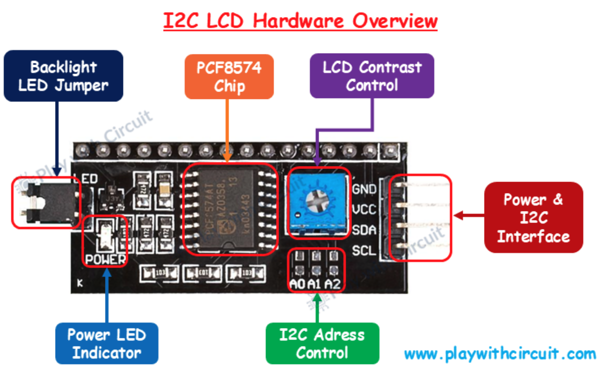

Figure 1: 16x2 character LCD with I2C interface

This type of LCD allows for easy communication with the Arduino board using the I2C protocol, which reduces the number of pins required for connection and simplifies the wiring.

1.2 I2C protocol

I2C (Inter-Integrated Circuit) is a communication protocol that allows multiple devices to communicate with each other using just two wires: SDA (Serial Data Line) and SCL (Serial Clock Line) [14]. It is commonly used in embedded systems for connecting peripherals such as sensors, displays, and other modules. In this project, the I2C protocol will be used to interface the LCD display with the Arduino board. The I2C protocol supports multiple devices on the same bus, allowing for easy expansion of the system in the future if additional peripherals are needed. The LCD display will be assigned a unique address on the I2C bus, and the Arduino will send commands and data to the LCD using this address to control what is displayed.

1.3 Keypad

A keypad is an input device that consists of a set of buttons arranged in a matrix format, allowing users to input data by pressing the buttons. In this project, a 4x4 matrix keypad will be used, which provides 16 buttons for user input. To uniquely identify each button, the keypad is organized in a matrix of rows and columns. When a button is pressed, it connects a specific row to a specific column, allowing the microcontroller to detect which button was pressed by scanning the rows and columns. The keypad will be used to input a pin code, which will then be validated by the system. The implementation will involve writing a driver to read the state of the buttons and interpret the user input accordingly. The circuit of a keypad is presented in Figure 2.

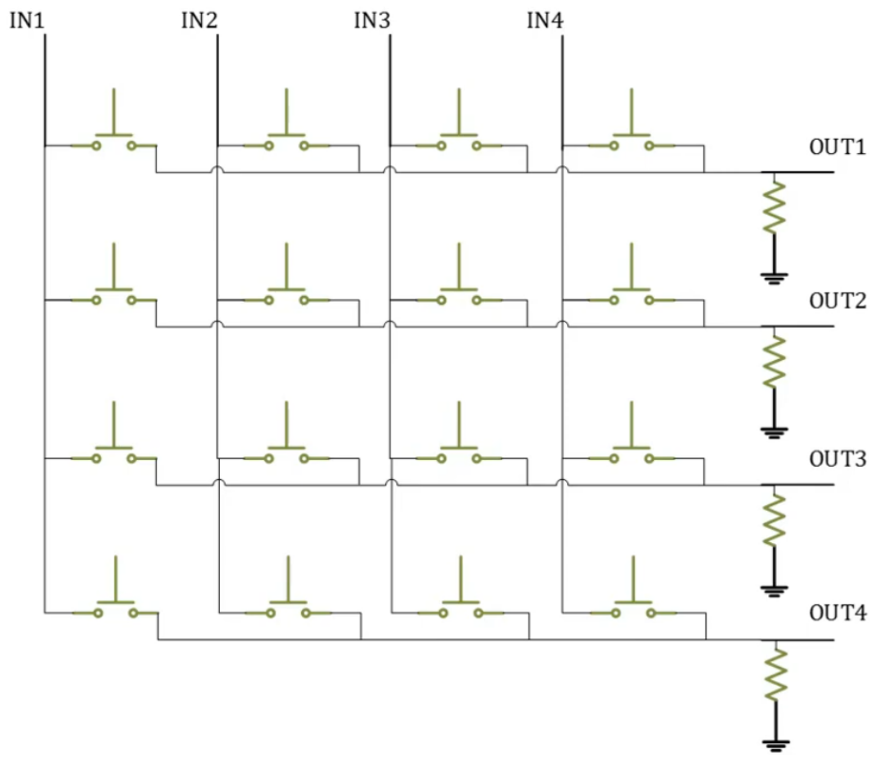

Figure 2: 4x4 matrix keypad

For this laboratory work, `Keypad.h` library will be used, which provides an easy-to-use interface for reading the state of the buttons and handling user input from the keypad [7].

1.4 STDIO library

As in previous laboratory work [?], the `stdio.h` library will be utilized for handling text-based input and output operations. However, the input and output streams will be different. Instead of using the serial monitor for input and output, the system will read user input from the keypad and display output on the LCD display. To achieve this, the standard input and output streams will be redirected to the appropriate interfaces. The `stdio` library provides functions such as `printf()` for output and `scanf()` for input, which can be used to interact with the user through the LCD display and keypad. By redirecting these streams, the program can use familiar I/O functions while still providing a seamless user experience with the hardware components.

1.5 Wokwi emulator

For testing and simulating the system, the Wokwi emulator will be used. Wokwi is an online platform that allows users to create and simulate embedded systems using a variety of components, including microcontrollers, sensors, and displays [2]. It provides a visual interface for designing circuits and writing code, making it easier to test and debug the system before deploying it on actual hardware. Moreover, Wokwi is avaliable as a plugin for Visual Studio Code, which allows for a seamless development experience without the need to switch between different tools. The Wokwi emulator will be used to simulate the behavior of the system, allowing for testing of the pin code validation logic and user interaction with the keypad and LCD display. One of the advantages of Wokwi over SimulIDE is a built-in support for a wider range of components, including the specific LCD display and keypad used in this project, which may not be available in SimulIDE.

2 System design

2.1 System architecture

The system is designed to read user input from a keypad, validate it against a predefined pin code, and provide feedback through an LCD display and an LED and presented on figure 3. The architecture of the system is structured in a layered design to ensure modularity and separation of concerns. The main components of the system include:

- **Keypad** - responsible for capturing user input in the form of a pin code.
- **LCD Display** - responsible for displaying messages to the user, such as prompts for input and feedback on the validation results.
- **LED** - provides a visual indication of whether the entered pin code is correct (green LED) or incorrect (red LED).
- **Microcontroller (Arduino)** - serves as the central processing unit that manages the interactions between the keypad, LCD display, and LED. It reads input from the keypad, processes the validation logic, and controls the LCD display and LED based on the results.

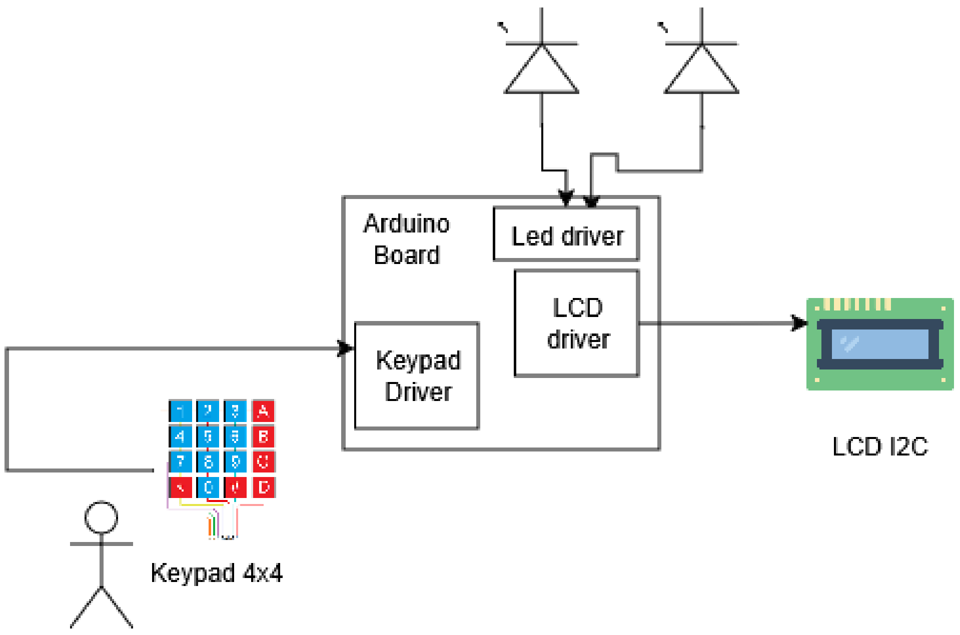

Figure 3: System Architecture Diagram

2.2 Flowchart of the program

The flowchart in Figure 4 illustrates the logic of the program running on the Arduino board.

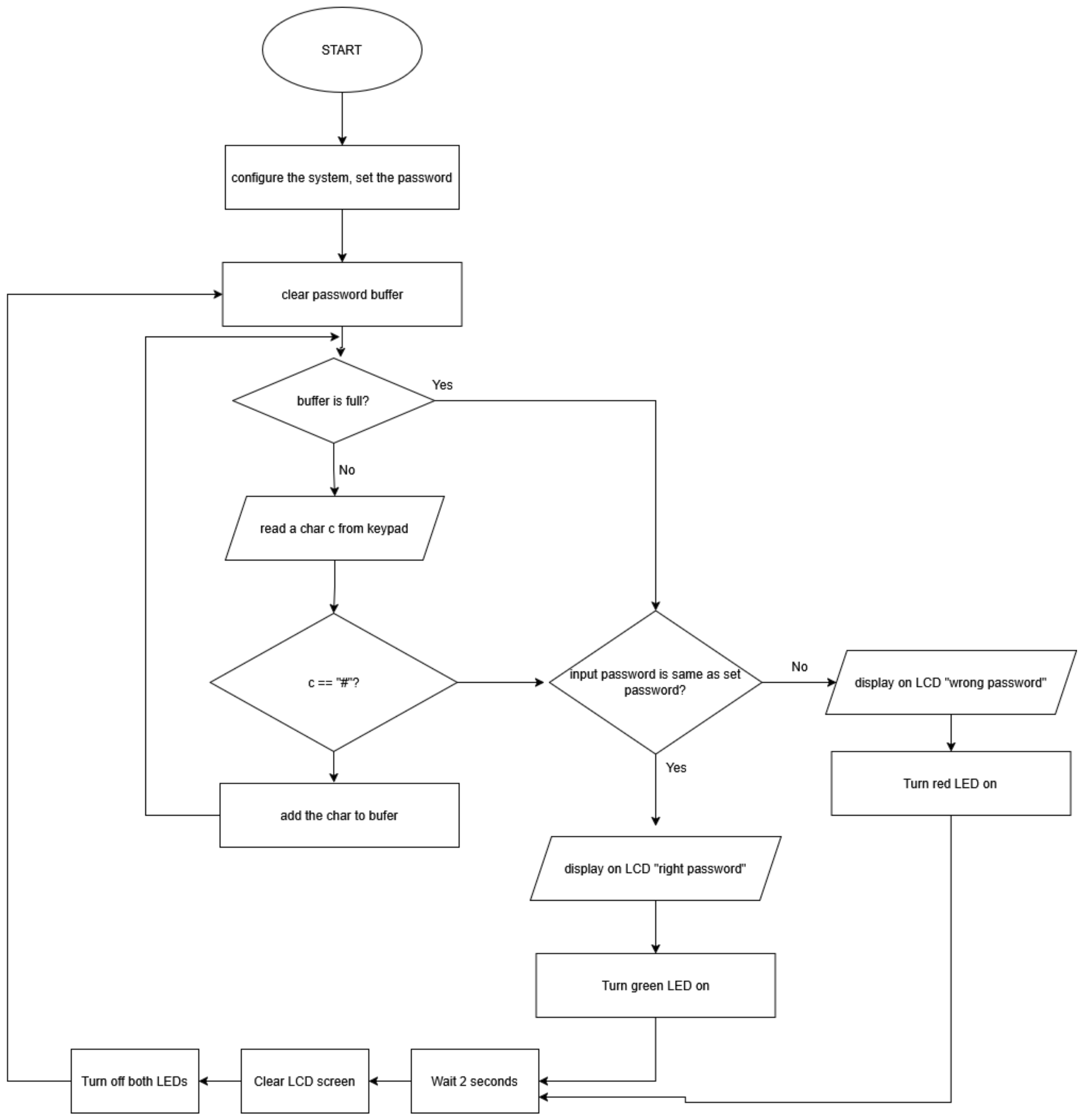
START
configure the system, set the password
clear password buffer
buffer is full?
Yes
No
read a char c from keypad
c == "#"?
add the char to bufer
input password is same as set password?
No
Yes
display on LCD "wrong password"
Turn red LED on
display on LCD "right password"
Turn green LED on
Wait 2 seconds
Clear LCD screen
Turn off both LEDs

Figure 4: Program Flowchart

The program starts waiting for user input via keypad char by char. If the char is #, the program considers the input as a complete command and processes it, otherwise adds this char to the buffer and waits for the next char. When it's read enough characters and the terminal one is not read, the input buffer is compared to predefined password. If the password is correct, the program turns on the green LED and displays the corresponding success message on the LCD display. Otherwise, the message is failure message and the LED is red. After processing the command, the program clears the input buffer, turns off the LEDs and waits for the next user input.

2.3 Hardware required

The following hardware components were used in this laboratory and are recommended for reproducing the experiments and simulations:

- **Microcontroller:** Arduino Uno or Nano (ATmega328P) — 1 unit (for development and deployment).
- **Keypad:** 4x4 matrix keypad — 1 unit (user input).
- **LCD:** 16x2 character LCD with I2C backpack (PCF8574 or compatible) — 1 unit (display output).
- **LEDs and resistors:** Red and green LEDs with 220Ω current-limiting resistors — 2 LEDs + resistors.
- **Wiring and prototyping:** Breadboard, male-to-male jumper wires, USB cable for power/programming.
- **Optional / useful extras:** External EEPROM (for persistent PIN storage), buzzer for audible feedback, and an I2C analyzer or logic probe for debugging.

2.4 Electronic circuit

Figure 5 illustrates the electronic circuit designed for this laboratory work. The circuit consists of an Arduino board connected to LCD display and a keypad. The LCD display is connected to the Arduino using the I2C interface, which requires only two wires (SDA and SCL) for communication. The keypad is connected to the Arduino using digital pins, with rows and columns arranged in a matrix format. Additionally, two LEDs are connected to the Arduino, along with a current-limiting resistor to ensure that the LEDs operates within its safe current limits [9]. The circuit is designed to allow for user interaction through the keypad, with feedback provided on the LCD display and visual indication through the LED.

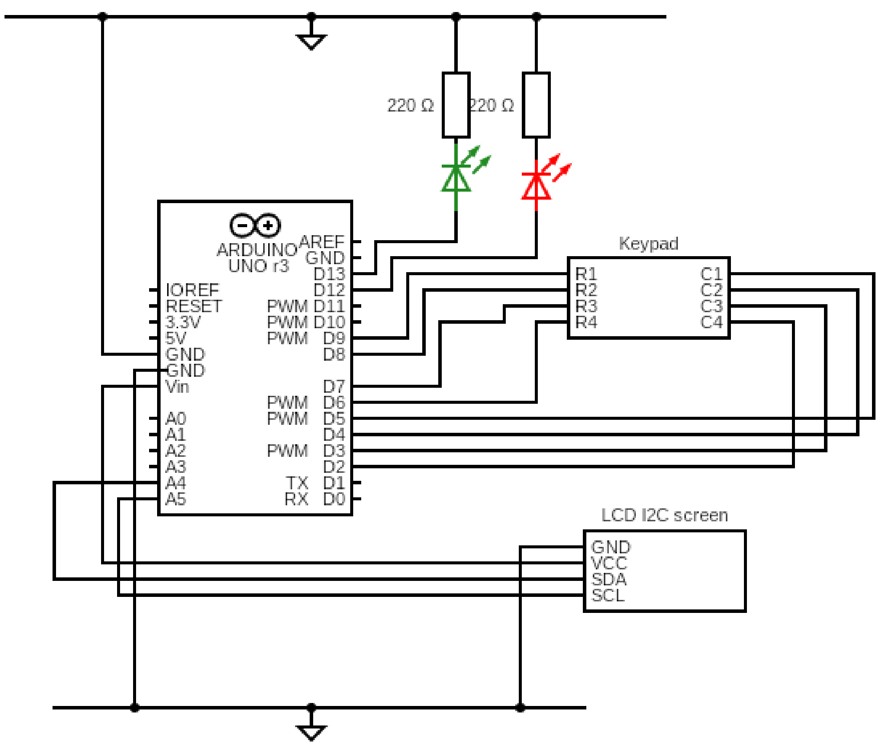

Figure 5: Circuit diagram of the LED control system

2.5 Layered design (hardware/software)

2.5.1 LCD Layered design

Figure 6 presents the layered design for the LCD control module. The design is structured into six layers:

- **APP** - outermost application layer, uses higher-level functions to interact with the LCD such as `lcd.print()`
- **SRV** - service layer, provides services to the application layer and interacts with the lower layers. Redirects `stdout` via `#include <stdio.h>` so that `printf()` writes to the LCD display.
- **ECAL** - embedded component abstraction layer, abstracts the hardware specifics of the LCD module, providing a uniform interface for higher layers via `#include <LiquidCrystal_I2C.h>`.
- **MCAL** - microcontroller abstraction layer, interfaces directly with the microcontroller's I2C/TWI hardware peripheral to transmit data to the LCD controller via `#include <Wire.h>`.

- **MCU** - microcontroller layer, represents the actual microcontroller hardware managing I2C communication. Here - A4 (SDA) and A5 (SCL) pins of Arduino.
- **ECU** - embedded component unit layer, represents the physical 16x2 (or 20x4) I2C LCD module connected to the microcontroller.

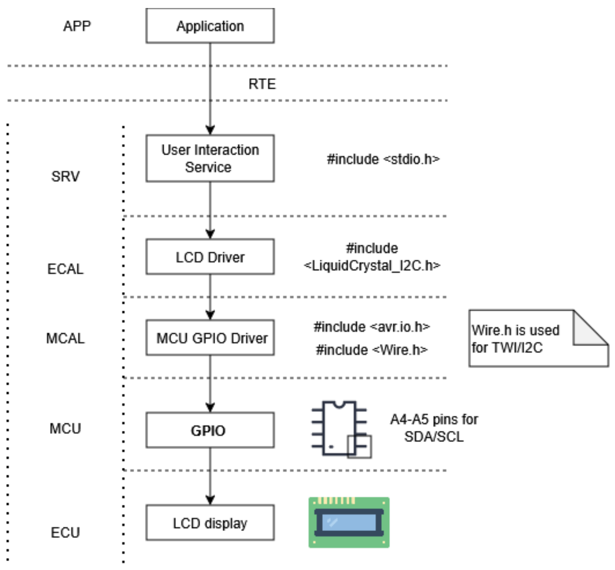

Figure 6: LCD Layered Design

Note: The arrows in the diagrams indicate the direction of data flow and interaction between layers. As one can see, all the arrows point upwards, since LCD is an output device.

2.5.2 Keypad Layered design

Figure 7 illustrates the layered design for the keypad control module. The design is structured into six layers:

- **APP** - outermost application layer, uses higher-level functions to read user input such as `getchar()` or `scanf()`
- **SRV** - service layer, provides services to the application layer and interacts with the lower layers. Redirects `stdin` via `#include <stdio.h>` so that `getchar()` reads from the keypad.
- **ECAL** - embedded component abstraction layer, abstracts the hardware specifics of the keypad matrix, providing a uniform interface for higher layers via `#include <Keypad.h>`.

- **MCAL** - microcontroller abstraction layer, interfaces directly with the microcontroller's GPIO hardware registers to perform matrix scanning via `#include <avr/io.h>`. Communicates bidirectionally with the MCU layer — writing to row pins to activate each row and reading from column pins to detect key presses.
- **MCU** - microcontroller layer, represents the actual microcontroller hardware managing GPIO communication. Here - D2-D9 pins of Arduino (4 row pins + 4 column pins).
- **ECU** - embedded component unit layer, represents the physical 4x4 matrix keypad connected to the microcontroller.

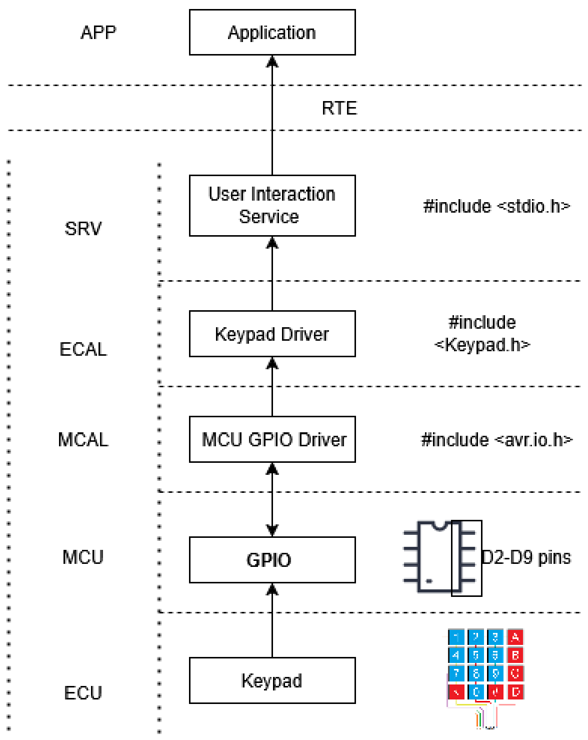

Figure 7: Keypad Layered Design

Note: As one can see, all the arrows point upwards, since Keypad is an input device. However, due to physical connection of the keypad, the microcontroller needs to write to the row pins to activate each row and read from the column pins to detect key presses, which is represented by a bidirectional arrow between MCAL and MCU layers.

3 Implementation and results

3.1 Driver implementation

3.1.1 Keypad driver implementation

Keypad driver implementation involves writing code to read the state of the buttons on the keypad and interpret user input. The driver will utilize the `Keypad.h` library to handle the matrix scanning and provide an interface for reading key presses. The driver will be responsible for initializing the keypad, scanning for button presses, and providing a way for the main application to read the input as characters or strings. Moreover, `KeypadStdioManager` will be implemented to redirect the standard input stream to read from the keypad, allowing the main application to use familiar I/O functions for user input. Figure 8 illustrates the class diagram for the keypad driver implementation, showing the relationship between the `KeypadStdioManager` and the underlying `Keypad` library.

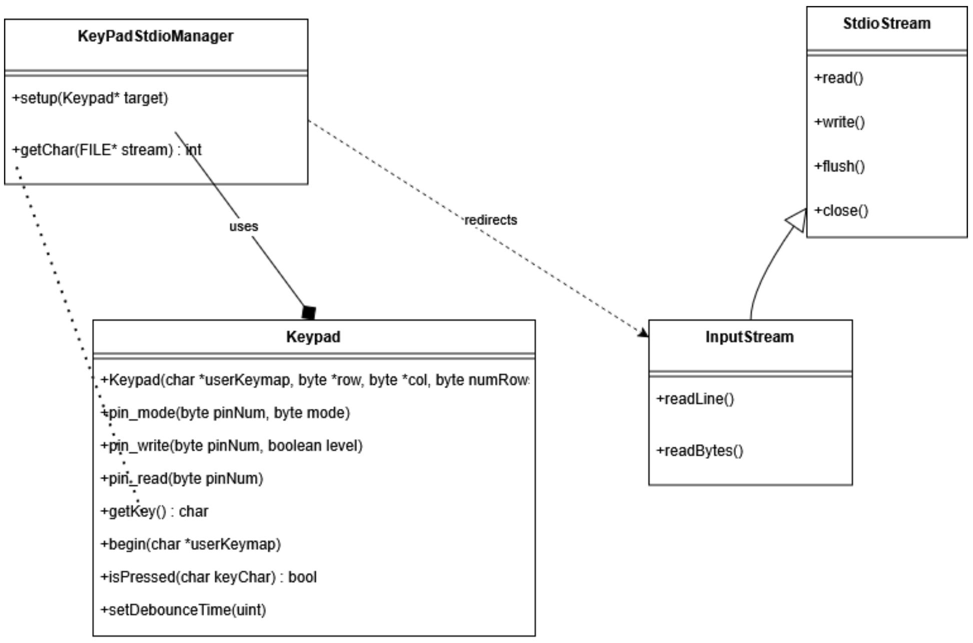

Figure 8: Keypad driver class diagram

3.1.2 LCD driver implementation

The LCD driver implementation involves writing code to control the LCD display and provide an interface for displaying messages. The driver will utilize the `LiquidCrystal_I2C.h` library to handle communication with the LCD over the I2C interface. The driver will be responsible for initializing the LCD, sending commands and data to control what is displayed, and providing a way for the main application to print messages to the LCD. Additionally, `LcdStdioManager` will be implemented to redirect the standard output stream to write to the LCD, allowing the main application to use familiar I/O functions for displaying output. Figure 9 illustrates the class diagram for the LCD driver implementation, showing the relationship between the `LcdStdioManager` and the underlying `LiquidCrystal_I2C` library.

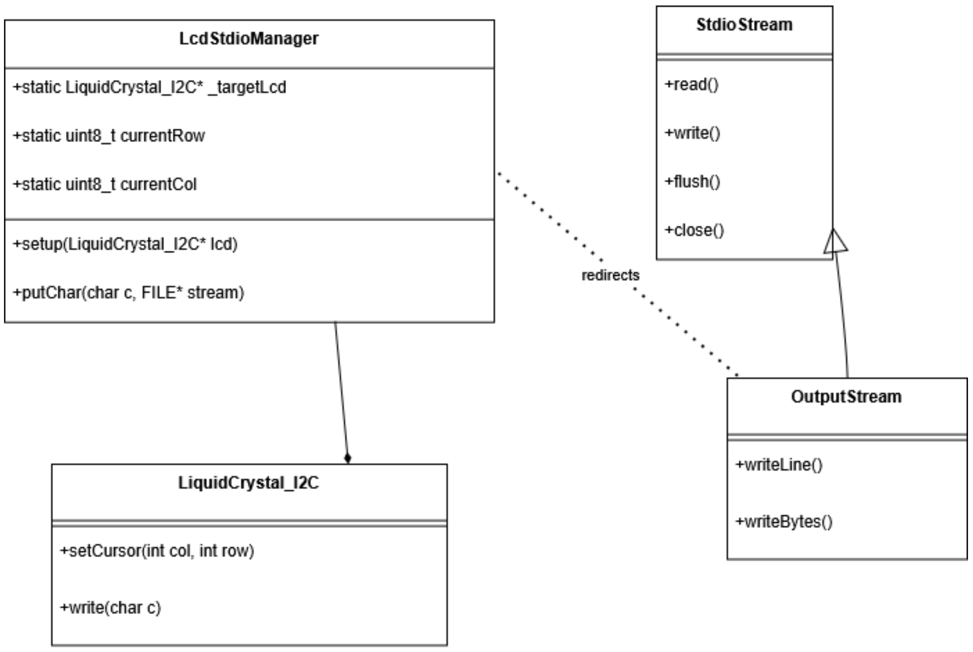
LcdStdioManager
+static LiquidCrystal_I2C* _targetLcd
+static uint8_t currentRow
+static uint8_t currentCol
+setup(LiquidCrystal_I2C* lcd)
+putChar(char c, FILE* stream)
StdioStream
+read()
+write()
+flush()
+close()
redirects
OutputStream
+writeLine()
+writeBytes()
LiquidCrystal_I2C
+setCursor(int col, int row)
+write(char c)

Figure 9: LCD driver class diagram

3.2 Main program implementation

For main program implementation, the logic of validating the pin code and controlling the LCD display and LED based on the results will be implemented. The program will read user input from the keypad, compare it to a predefined pin code, and provide feedback through the LCD display and LED. The main program will utilize the drivers implemented for the keypad and LCD display to interact with the hardware components. The program will also include logic to handle user input, such as buffering input characters until a complete command is received (e.g., when the user presses a specific key to indicate the end of input). Based on whether the entered pin code is correct or incorrect, the program will control the LEDs accordingly and display appropriate messages on the LCD. The main program will be structured to ensure clarity and maintainability, with separate functions for handling input, validating the pin code, and controlling the output devices.

3.3 Simulation

The simulation below uses Wokwi online simulator. The steps shown in the simulation are as follows:

- Basic setup of the circuit designed with accordance to the circuit diagram 10.
- Introducing a pin command in the serial monitor 11.
- The system responds with "Nope" if the password is incorrect 12 and lights a red LED.
- The system responds with "Yep" if the password is correct 13 and lights a green LED.

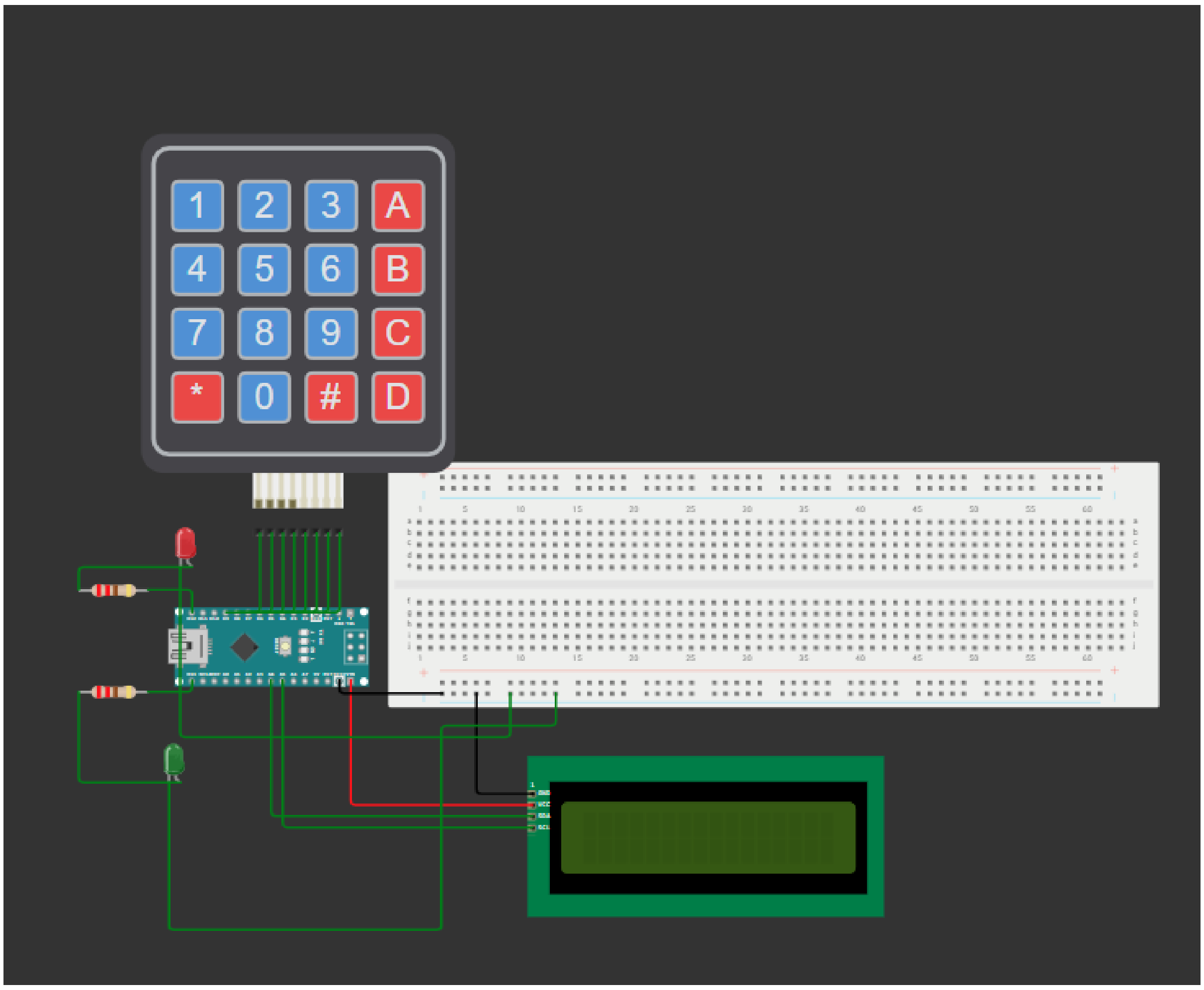

Figure 10: Simulation of the system using Wokwi emulator

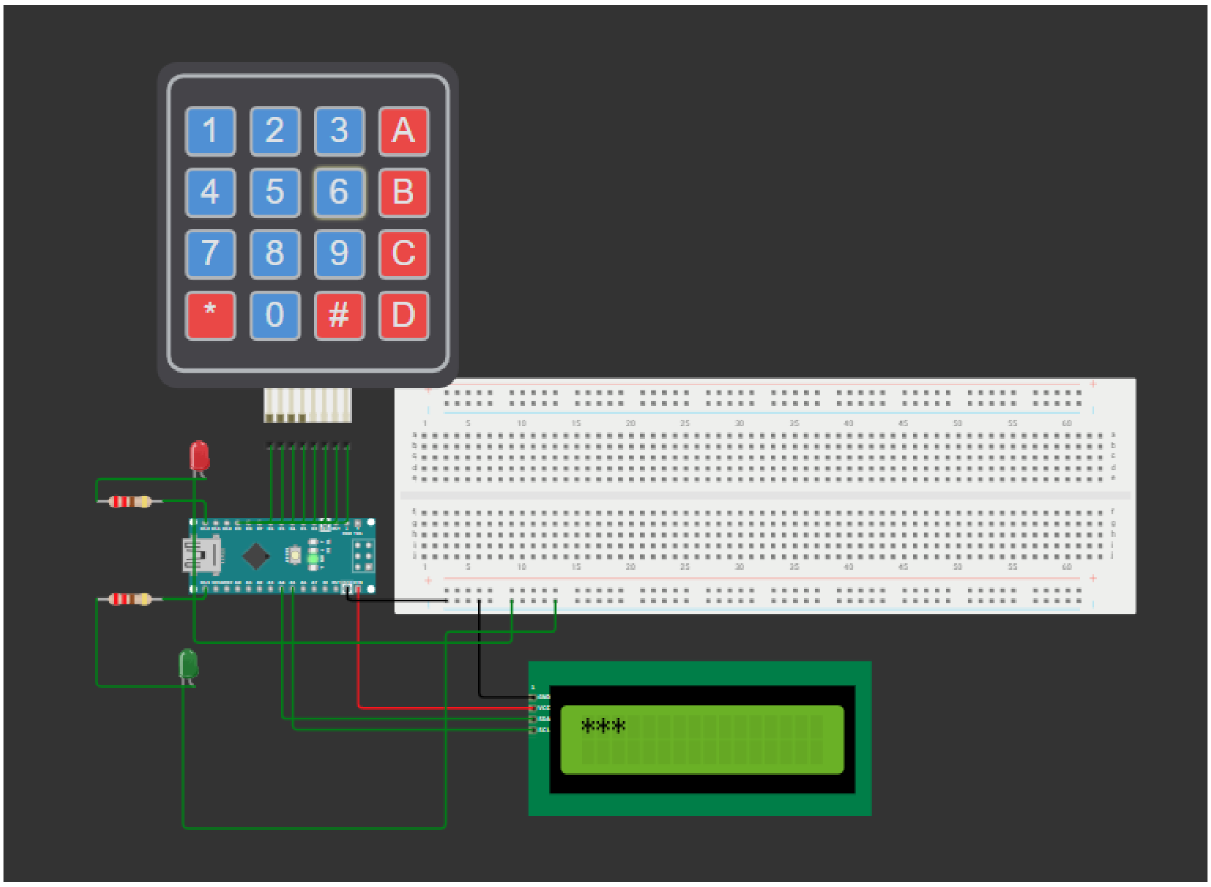

Figure 11: Introducing a pin

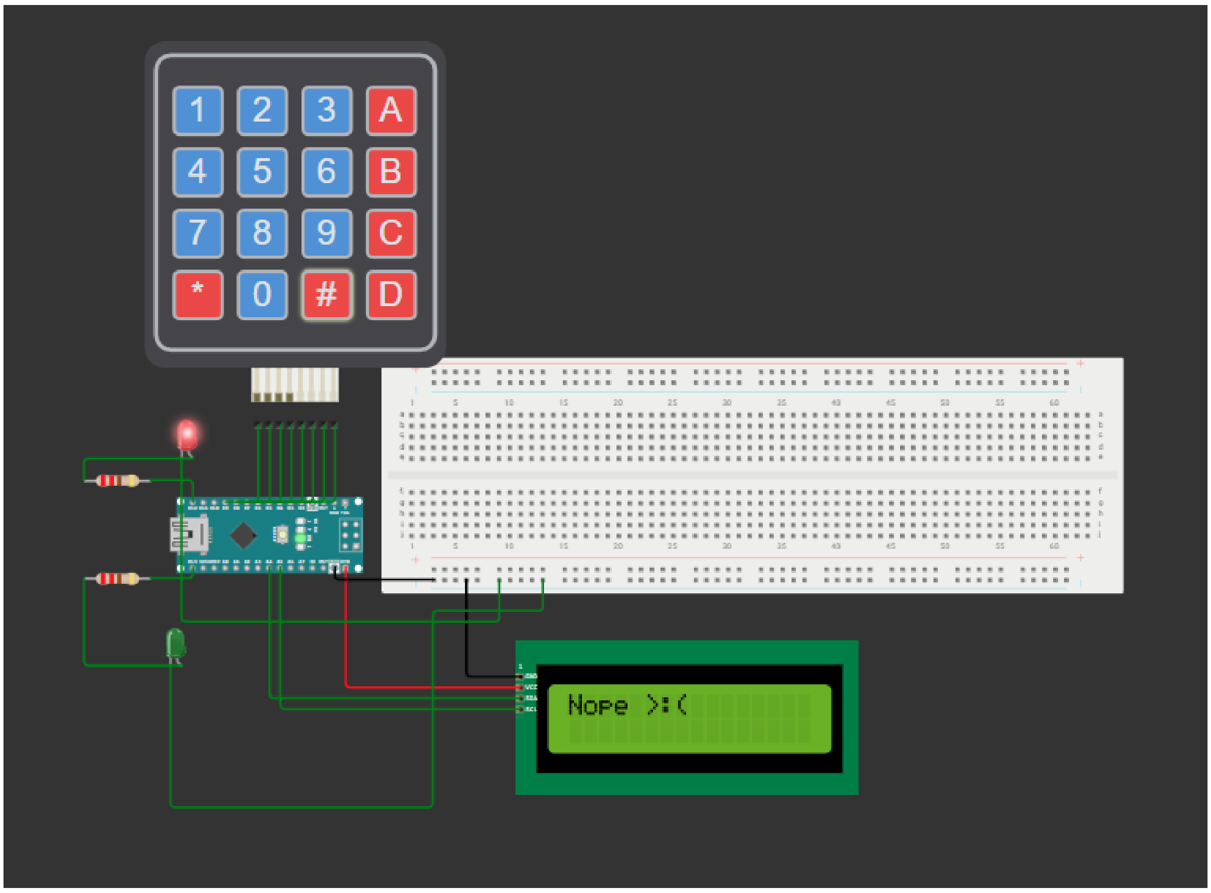

Figure 12: The password is incorrect

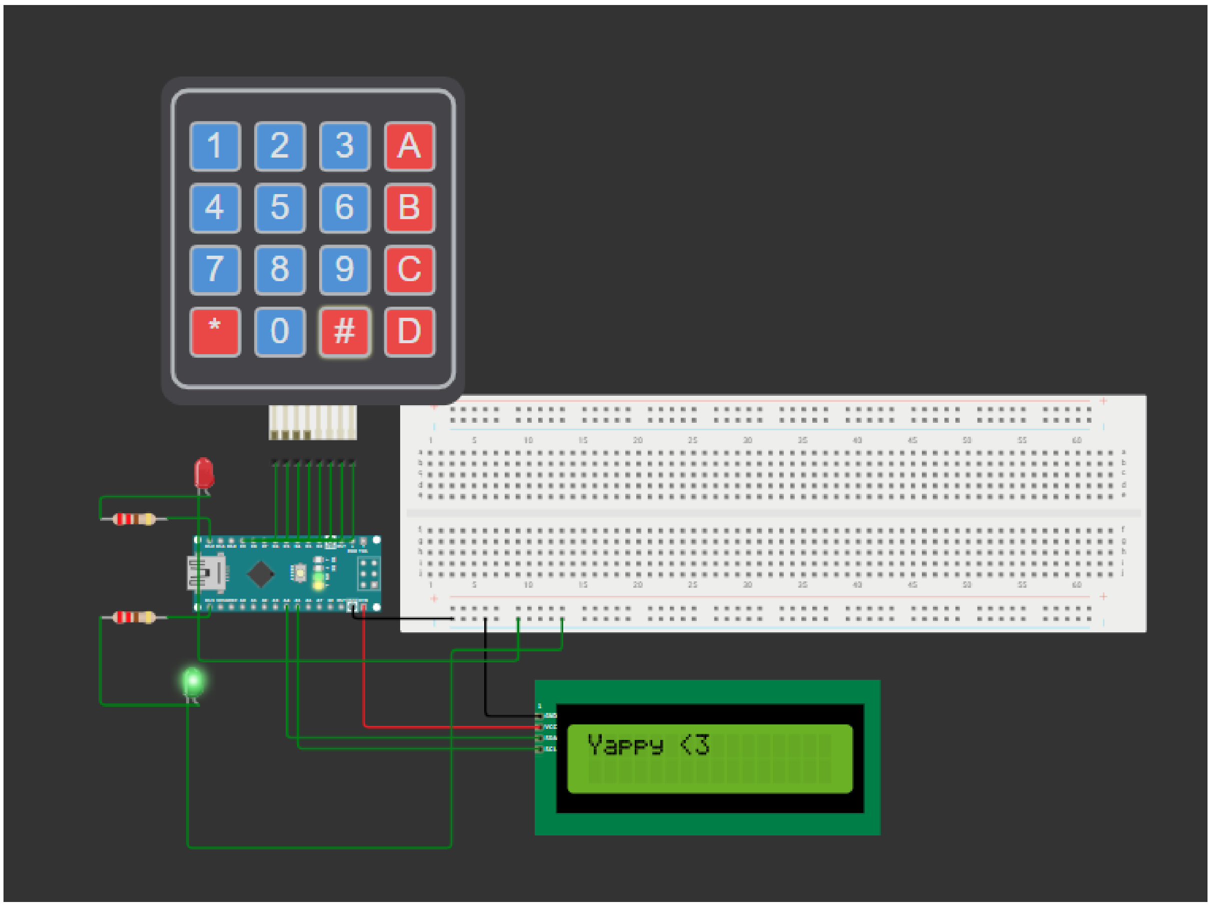

Figure 13: The password is correct

4 Conclusion

This laboratory demonstrated a complete, modular implementation of a pin-code entry and validation system targeted at small microcontroller platforms. The final design couples a simple, testable application core (‘PinCodeSystem‘) with thin hardware adapters for the keypad, LCD and LEDs. Redirecting standard input/output to ‘KeypadStdioManager‘ and ‘LcdStdioManager‘ proved effective: it let the application use familiar C-style I/O routines while remaining hardware-agnostic, which simplified both local testing and emulator runs on Wokwi.

Key outcomes and observations: - Functionality: the system reliably reads key presses, collects user input into a command buffer, and performs PIN comparison when the terminator key is pressed. Correct and incorrect entries trigger distinct LCD messages and LED indicators. - Design quality: the layered architecture enforces separation of concerns — application logic, service adapters, and low-level drivers are distinct and independently testable. - Reuse: several components (serial/stdout adapters, LED abstraction) were reused from the shared library, reducing duplication and easing future labs. - Limitations observed: the current implementation uses a fixed, plaintext PIN and lacks debounce/filtering and rate-limiting; these are acceptable for the lab exercise but insufficient for production use.

Recommendations and next steps: - Add input debouncing and simple anti-brute-force measures (temporary lockout after N failures) to harden the system. - Migrate PIN storage to a small, non-volatile mechanism (EEPROM or encrypted storage) and add secure comparison techniques to avoid timing leaks. - Extend the UI to support administrative functions (PIN change, multi-user) and add unit tests for 'PinCodeSystem' logic to validate edge cases.

In summary, the lab met its learning goals: it demonstrated practical peripheral integration, STDIO redirection for embedded I/O, and a clean layered design that will scale to future enhancements and more robust security features.

References

[1] Serial Communication `https://learn.sparkfun.com/tutorials/serial-communication/all`

[2] Wokwi - Online Arduino Simulator `https://wokwi.com/`

[3] AVR-Libc Reference Manual - Standard I/O `https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html`

[4] GitHub repository `https://github.com/TimurCravtov/EmbeddedSystemsLabs`

[5] Arduino Memory specification and architecture `https://docs.arduino.cc/learn/programming/memory-guide/`

[6] Arduino LED Control `https://roboticsbackend.com/arduino-led-complete-tutorial/`

[7] Keypad library for Arduino `https://docs.oyoclass.com/unoeditor/Libraries/keypad/`

[8] Circuit Diagram Builder `https://www.circuit-diagram.org/docs`

[9] Standard 5mm LED specification `https://www.make-it.ca/5mm-led-specifications/`

[10] Platform.IO - Embedded Development Ecosystem `https://platformio.org/`

[11] SimulIDE - Simple real time electronics simulator `https://simulide.com/p/mcus-/l`

[12] Serial terminal emulator overview - Toradex Developer Center `https://developer.toradex.com/software/development-resources/serial-terminal-emulator/`

[13] 16x2 character LCD with I2C interface `https://www.instructables.com/Interfacing-I2C-LCD-With-Arduino-UNO/`

[14] I2C Protocol Overview `https://circuitcrush.com/i2c-tutorial/`

A Source Code

Besides this appendix, the source code is available in the GitHub repository [4] with all build instructions.

The following files are the primary sources for Laboratory 1.2. Each listing shows the file used in this lab and a short description of its purpose.

Main application: Implements the command loop, reads characters from the keypad (via the keypad STDIO adapter), forwards input to the `PinCodeSystem` for validation, and controls the LCD and LEDs based on the result.

`main.cpp`

```cpp
#include <Arduino.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <lcd/LcdStdioManager.h>
#include <keypad/KeypadStdioManager.h>
#include <Keypad.h>
#include <led/led.h>
#include "PinCodeSystem.h"
// #include <serialio/serialio.h>

// ketpadd stuff
constexpr byte ROWS = 4;
constexpr byte COLS = 4;

// Keymap must be in RAM: Keypad library reads it with keymap[i
    ], not pgm_read_byte()
const char keys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
```

```
};

constexpr byte rowPins[ROWS] = {9, 8, 7, 6};
constexpr byte colPins[COLS] = {5, 4, 3, 2};

Keypad keypad = Keypad(makeKeymap(keys), const_cast<byte*>(
    rowPins), const_cast<byte*>(colPins), ROWS, COLS);


// lcd
constexpr uint8_t LCD_ADDRESS = 0x27;
constexpr uint8_t LCD_COLS = 16;
constexpr uint8_t LCD_ROWS = 2;

LiquidCrystal_I2C lcd(LCD_ADDRESS, LCD_COLS, LCD_ROWS);

// leds
constexpr uint8_t redLedPin = 12;
Led redLed(redLedPin);

constexpr uint8_t greenLedPin = 13;
Led greenLed(greenLedPin);

// PASSWORD FOR ACTIVATING THE SYSTEM (must be numeric and not
    exceed maxPasswordLength)
constexpr uint8_t maxPasswordLength = 10;
const char PASSWORD[] PROGMEM = "123653";

void setup() {
    lcd.init();

    if (!isConfiguredPasswordValid(maxPasswordLength)) {
        printf_P(PSTR("Invalid password\n"));
        while (true) {
        delay(1000);
        }
    }
    // Serial.begin(9600);

    lcd.backlight();
    redLed.init();

    // redirectErrorToSerial();
    greenLed.init();
    LcdStdioManager::setup(&lcd);
    KeypadStdioManager::setup(&keypad);
    // fprintf_P(stderr, PSTR("System initialized\n\rTest"));
}

void loop() {

    handleThisSupremeSecuredSystem(redLed, greenLed,
       maxPasswordLength);
}
```

Listing 1: main.cpp (lab1_2)

Pin code logic: Encapsulates the pin-code state machine and validation logic. Keeps the application logic separate from peripheral drivers.

`PinCodeSystem.h`

```
#pragma once

#include <led/led.h>
#include <stdint.h>
#include <stdbool.h>
#include <avr/pgmspace.h>

// Password stored in program memory (defined in main.cpp)
extern const char PASSWORD[] PROGMEM;

void handleThisSupremeSecuredSystem(Led& red, Led& green,
    uint8_t maxPasswordLength);

byte* readPassword(uint8_t maxPasswordLength);

bool isConfiguredPasswordValid(uint8_t maxPasswordLength);
void clearline();
```

Listing 2: PinCodeSystem.h

PinCodeSystem.cpp

```
#include "PinCodeSystem.h"
#include <Arduino.h>
#include <avr/pgmspace.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

// PASSWORD is defined in main.cpp (stored in PROGMEM)

// Internal maximum buffer for readPassword()
static constexpr uint8_t INTERNAL_BUFFER_SIZE = 32;


void handleThisSupremeSecuredSystem(Led& red, Led& green,
    uint8_t maxPasswordLength) {

    byte* passwordAttempt = readPassword(maxPasswordLength);

    // if password is correct
    if (strcmp_P((char*)passwordAttempt, PASSWORD) == 0) {

        // turn on green and print happy message
        green.on();
        red.off();
        clearline();
        printf_P(PSTR("Yappy <3"));
        delay(2000);
        green.off();
        clearline();
    } else {

        // turn on red and print sad message
        red.on();
        green.off();
        clearline();
        printf_P(PSTR("Nope >:("));
        delay(2000);
        red.off();
        clearline();
    }
}

// password should be numeric and not exceed maxPasswordLength
bool isConfiguredPasswordValid(uint8_t maxPasswordLength) {
    uint8_t length = strlen_P(PASSWORD);
    if (length == 0 || length > maxPasswordLength) {
        return false;
    }
    for (uint8_t i = 0; i < length; i++) {
        char c = (char)pgm_read_byte(&PASSWORD[i]);
        if (!isdigit((unsigned char)c)) {
            return false;
        }
    }
    return true;
}

// prints 16 spsaces from the beginning of the line and returns the cursor
void clearline() {
    putchar('\r');
    // default to a 16-column LCD (matches project's LCD_COLS)
    for (uint8_t i = 0; i < 16; ++i) {
        putchar(' ');
    }
    putchar('\r');
}

// reads password until '#' or is the length succeds maxPassword length
byte* readPassword(uint8_t maxPasswordLength) {
    static byte input[INTERNAL_BUFFER_SIZE + 1];
    memset(input, 0, sizeof(input)); // clear the buffer

    if (maxPasswordLength > INTERNAL_BUFFER_SIZE) {
        maxPasswordLength = INTERNAL_BUFFER_SIZE;
    }

    uint8_t i;
    for (i = 0; i < maxPasswordLength; i++) {
        char c;
        scanf("%c", &c);
        if (c == '#') {
            input[i] = '\0';
            break;
        }
        input[i] = c;
        putchar('*');
    }
    input[i] = '\0';
    return input;
}
```

Listing 3: PinCodeSystem.cpp

Shared drivers used by this lab: The lab uses several shared components (std-in/stdout adapters and drivers) located in the shared module. These are reused across labs to provide consistent STDIO redirection and peripheral abstractions.

`serialio.h`

```
#pragma once

#include <Arduino.h>

// this function does exactly what do you think it does
void redirectSerialToStdio();

void redirectErrorToSerial();
```

Listing 4: serialio.h

`serialio.cpp`

```
#include <Arduino.h>
#include <serialio/serialio.h>
```

```
#include <stdio.h>

#include <stdlib.h>


// add a char to serial output
int serialPutchar(char c, FILE* stream) {
  Serial.write(c);
  return 0;
}

// get a char from serial input
int serialGetchar(FILE* stream) {
  while (!Serial.available());
  return Serial.read();
}

// Helper functions for redirecting Serial to stdio
void redirectSerialToStdio() {
  static FILE uartinout;

  fdev_setup_stream(&uartinout, serialPutchar, serialGetchar,
      _FDEV_SETUP_RW);

  stdout = stdin = stderr = &uartinout;
}


void redirectErrorToSerial() {
  static FILE uartout;

  fdev_setup_stream(&uartout, serialPutchar, NULL,
      _FDEV_SETUP_WRITE);

  stderr = &uartout;
}
```

Listing 5: serialio.cpp

`KeypadStdioManager.h`

```
#pragma once

#include <Keypad.h>

class KeypadStdioManager {
public:
    static void setup(Keypad* target);

private:
    static Keypad* _targetKeypad;
    static int getChar(FILE* stream);
};
```

Listing 6: KeypadStdioManager.h

`KeypadStdioManager.cpp`

```
#include <keypad/KeypadStdioManager.h>

Keypad* KeypadStdioManager::_targetKeypad = nullptr;

void KeypadStdioManager::setup(Keypad* target) {
    _targetKeypad = target;

    static FILE keypadStream;

    fdev_setup_stream(&keypadStream, nullptr, getChar,
      _FDEV_SETUP_READ);

    stdin = &keypadStream;
}

int KeypadStdioManager::getChar(FILE* stream) {

    if (_targetKeypad == nullptr) {
        return EOF;
    }

    int key = NO_KEY;
    while (!(key = _targetKeypad->getKey()));
```

```cpp

    return key;

}
```

Listing 7: KeypadStdioManager.cpp

`LcdStdioManager.h`

```cpp
#pragma once

#include <LiquidCrystal_I2C.h>
#include <stdio.h>
#include <stdlib.h>

class LcdStdioManager {
public:
    static void setup(LiquidCrystal_I2C* lcd);

private:
    static LiquidCrystal_I2C* _targetLcd;
    static int putChar(char c, FILE* stream);
};
```

Listing 8: LcdStdioManager.h

`LcdStdioManager.cpp`

```cpp
#include <lcd/LcdStdioManager.h>

LiquidCrystal_I2C* LcdStdioManager::_targetLcd = nullptr;
static uint8_t currentRow = 0;
static uint8_t currentCol = 0;

void LcdStdioManager::setup(LiquidCrystal_I2C* lcd) {
    _targetLcd = lcd;
    static FILE lcdout;
    fdev_setup_stream(&lcdout, putChar, NULL, _FDEV_SETUP_WRITE)
        ;
    stdout = &lcdout;
}

```

```cpp
// remember: setCursor(x,y) means x columns and y rows
int LcdStdioManager::putChar(char c, FILE* stream) {
    if (_targetLcd) {

        // that's peak performance
        if (c == '\n') {

            currentRow = (currentRow + 1) % 2;
            currentCol = 0;
            _targetLcd->setCursor(currentCol, currentRow);

        } else if (c == '\r') {
            currentCol = 0;
            _targetLcd->setCursor(0, currentRow);
        } else {
            _targetLcd->write(c);
            currentCol++;

        }
    }
    return 0;
}
```

Listing 9: LcdStdioManager.cpp

`led.h`

```cpp
#pragma once

#include <Arduino.h>

/// @brief A simple LED control class
class Led {
public:
  explicit Led(uint8_t pin);

  void on();
  boolean isOn();
  void init();
  void off();
  void toggle();

private:
  uint8_t pin_;
};
```

Listing 10: led.h

`led.cpp`

```
#include <led/led.h>

Led::Led(uint8_t pin) : pin_(pin) {}

void Led::init() {
  pinMode(pin_, OUTPUT);
}

void Led::on() {
  digitalWrite(pin_, HIGH);
}

boolean Led::isOn() {
  return digitalRead(pin_) == HIGH;
}

void Led::off() {
  digitalWrite(pin_, LOW);
}

void Led::toggle() {
  digitalWrite(pin_, !digitalRead(pin_));
}
```

Listing 11: led.cpp