Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Physics

Embedded Systems

Laboratory work #1.2

Electronic Lock System with LCD and Keypad using STDIO

Author:
Dmitrii Belih
std. gr. FAF-232

Verified:
Martiniuc A.

Chișinău 2026

1. Domain Analysis

1.1. Purpose of the Laboratory Work

The purpose of this laboratory work is to understand the principles of user interaction with embedded systems using LCD displays and matrix keypads, and to implement a password verification system using the STDIO library. The work involves setting up an I2C-based LCD display for output, a 4×4 matrix keypad for input, and an access control system that verifies 4-digit PIN codes. The system demonstrates the use of STDIO abstraction for debugging and serial communication while implementing a finite state machine (FSM) for managing user interaction states. Additional functionality includes a programming mode for changing the password, with visual feedback through LEDs.

1.2. Technologies Used

Standard Input/Output (STDIO)

Standard Input/Output (STDIO) provides a standardized mechanism for handling input and output operations in embedded systems. In this laboratory work, STDIO is redirected to the serial interface (UART) for debugging and system monitoring purposes. The STDIO library enables the use of familiar C functions such as `printf()` for formatted output and `fprintf()` for stream-specific output. This abstraction layer simplifies debugging and logging, allowing developers to track system state, input events, and operational status without implementing custom serial communication routines.

I2C Communication Protocol

Inter-Integrated Circuit (I2C) is a synchronous, multi-master, multi-slave, packet-switched, single-ended, serial communication bus. In this laboratory work, I2C is used to communicate with the LCD display module. I2C requires only two signal lines (SDA for data and SCL for clock) plus ground, significantly reducing the number of GPIO pins required for peripheral connections. The protocol uses a 7-bit addressing scheme, allowing multiple devices to share the same bus. LCD modules with I2C interfaces typically use address 0x27 or 0x3F.

Matrix Keypads

Matrix keypads are input devices that arrange buttons in a grid pattern (rows and columns) to minimize the number of required pins. A 4×4 keypad uses 8 pins (4 rows + 4 columns) instead of 16 individual pins. Keypad scanning involves sequentially activating each row and reading the column states to detect pressed buttons. This technique requires debouncing (software or hardware) to prevent false triggers from contact bounce. Keyboards with matrix layouts are widely used in security systems, ATMs, and industrial control panels.

Finite State Machine (FSM)

A Finite State Machine is a computational model used to design systems that can be in one of a finite number of states. In this laboratory work, an FSM manages the access control system's behavior across different states: Welcome/Normal Mode, Input Processing, Access Granted, Access Denied, and Programming Mode. Each state has specific behaviors, transitions, and responses to user input. The FSM approach ensures clear separation of concerns, predictable system behavior, and easier maintenance and debugging.

LCD Display (Liquid Crystal Display)

LCD displays are visual output devices commonly used in embedded systems to present information to users. Character-based LCDs (such as 16×2) can display text in fixed character positions. Modern LCD modules often include I2C interfaces for simplified connectivity. The LiquidCrystal_I2C library provides a convenient API for initializing the display, setting cursor positions, printing text, and controlling backlight. LCDs are ideal for user interfaces in embedded systems due to their low power consumption, readability, and ability to display dynamic information.

1.3. Hardware Components

Arduino Uno

The Arduino Uno is based on the ATmega328P microcontroller featuring 14 digital I/O pins, 6 analog inputs, and 32 KB flash memory. For this laboratory work, the Arduino provides GPIO pins for keypad control (8 pins), LED control (3 pins), and I2C communication (2 pins: SDA on A4, SCL on A5). The board operates at 5V and provides sufficient processing power for real-time keypad scanning and state machine management.

4×4 Matrix Keypad

The 4×4 matrix keypad consists of 16 buttons arranged in 4 rows and 4 columns. The typical layout includes digits 0-9, letters A-D, and special keys (*) and (#). The keypad requires 8 GPIO pins: 4 rows configured as outputs and 4 columns configured as inputs with pull-up resistors. Keys are detected by scanning: each row is sequentially set LOW, and columns are read to detect when a button connects a row to a column.

LCD 16×2 with I2C Interface

The 16×2 character LCD can display 32 characters (16 per line) across two lines. When equipped with an I2C interface board, it requires only 2 wires (SDA and SCL) plus power (VCC and GND). The I2C interface typically includes a PCF8574 I/O expander chip and a potentiometer for contrast adjustment. The default I2C address is 0x27, though some modules use 0x3F.

LED Indicators

Three LEDs provide visual feedback:

- **Green LED (Pin 12):** Indicates successful authentication. Lights for 5 seconds when correct password is entered.
- **Red LED (Pin 13):** Indicates authentication failure. Blinks 5 times when incorrect password is entered.
- **Programming LED (Pin 3):** Indicates programming mode is active. Lights when user enters password change mode.

Each LED requires a current-limiting resistor (220 typical) to prevent damage to both the LED and the microcontroller.

1.4. Software Components

LiquidCrystal_I2C Library

The LiquidCrystal_I2C library provides an Arduino-compatible interface for controlling I2C-based LCD displays. It wraps the Wire library (I2C implementation) and offers methods for initialization (`init()`), text output (`print()`, `write()`), cursor control (`setCursor()`), display clearing (`clear()`), and backlight control (`backlight()`, `noBacklight()`). The library handles I2C communication details, allowing developers to focus on display content.

Wire Library (I2C)

The Wire library is Arduino's built-in implementation of the I2C protocol. It provides methods for initializing the I2C bus (`Wire.begin()`), starting transmission to a device (`Wire.beginTransmission()`), writing data (`Wire.write()`), ending transmission (`Wire.endTransmis`
and requesting data from devices (`Wire.requestFrom()`). This library is used internally by the LiquidCrystal_I2C library and for I2C device scanning.

PlatformIO

PlatformIO provides an advanced development environment for embedded systems with features including intelligent code completion, multi-platform build systems, library management, and debugging capabilities. For this laboratory work, PlatformIO manages dependencies (LiquidCrystal_I2C library), handles compilation, uploads firmware to the Arduino, and provides a serial monitor for debugging output. The platformio.ini file specifies build flags, include paths, and library dependencies.

1.5. System Architecture and Justification

The system architecture follows a layered, modular approach with clear separation of concerns:

- **Hardware Layer:** Consists of Arduino Uno microcontroller, 4×4 matrix keypad (8 GPIO pins), LCD 16×2 with I2C interface (2 pins), and three LEDs (3 GPIO pins). The microcontroller provides computational resources and GPIO/I2C interfaces.
- **Hardware-Software Interface (Driver Layer):** Implements low-level drivers for each hardware component:
  - *Keypad Driver:* Handles matrix scanning, debouncing, and key detection
  - *LCD Driver:* Wraps LiquidCrystal_I2C library with additional logging
  - *LED Driver:* Provides simple on/off/toggle control
  - *Serial STDIO Driver:* Redirects stdout/stdin to UART for debugging
- **Application Logic Layer (FSM):** Implements the finite state machine that manages system states:
  - *Welcome State:* Displays "Enter Code:"prompt, accepts first key
  - *Input State:* Accumulates 4-digit PIN, displays asterisks
  - *Verify State:* Compares input with stored password
  - *Access Granted State:* Shows success message, activates green LED
  - *Access Denied State:* Shows error message, blinks red LED
  - *Programming Mode State:* Allows password change via 'D' key
- **STDIO Service Layer:** Provides formatted output for debugging and system monitoring through serial interface, enabling real-time visibility into system operation.

This architecture was chosen to promote modularity, reusability, and maintainability. Each hardware component has a dedicated driver, the FSM clearly separates behavioral states, and the STDIO layer provides non-intrusive debugging. This design allows easy extension (adding more states, different passwords, or additional security features) without modifying core components.

1.6. Case Study: Real-World Access Control Systems

Access control systems are ubiquitous in modern security applications, from office buildings and ATM machines to hotel rooms and secure facilities. These systems typically require:

- **User Input:** Keypads, card readers, or biometric scanners
- **Display/Feedback:** LCD screens, LED indicators, or audio prompts
- **Authentication Logic:** Password verification, card validation, or biometric matching
- **Access Control:** Door locks, turnstiles, or electronic barriers
- **Management Features:** Password changes, user management, or configuration modes

Our laboratory work implements a simplified version of such a system using a 4×4 keypad for input, an LCD display for user feedback, and LEDs for status indication. The programming mode (activated by the 'D' key) demonstrates real-world system management capabilities, allowing authorized users to change access codes without reprogramming the device.

The finite state machine approach used here mirrors professional access control systems, where clear state definitions and transitions ensure predictable behavior and simplify debugging. Software debouncing for keypad input addresses real-world challenges with mechanical switches, where contact bounce can cause false triggers. The modular architecture allows components to be reused in future laboratory works, simulating industry practices of code reuse and library development.

2. Design

2.1. Architectural Sketch

System Architecture Overview:

The system follows a layered architecture with clear separation between hardware, drivers, and application logic.

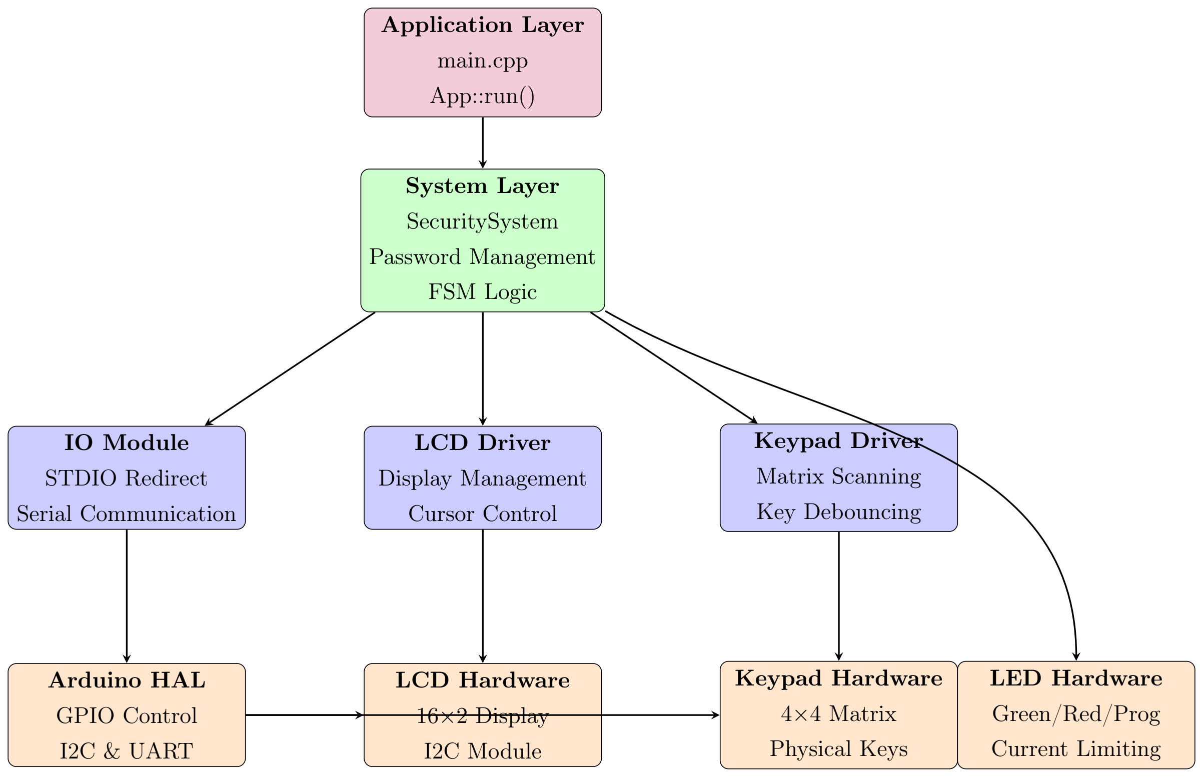
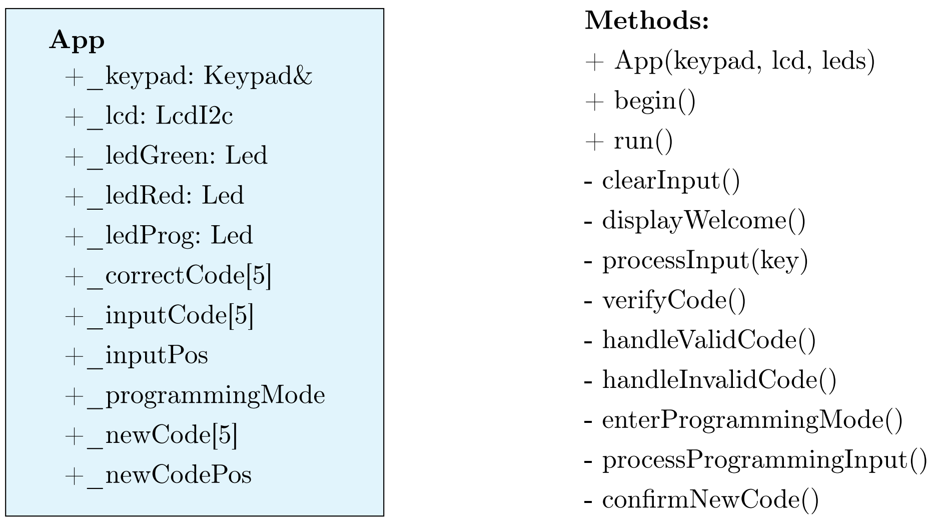


Рис. 1: System Architecture Diagram

Module Block Scheme (App Class):



App
+_keypad: Keypad&
+_lcd: LcdI2c
+_ledGreen: Led
+_ledRed: Led
+_ledProg: Led
+_correctCode[5]
+_inputCode[5]
+_inputPos
+_programmingMode
+_newCode[5]
+_newCodePos

Methods:
+ App(keypad, lcd, leds)
+ begin()
+ run()
- clearInput()
- displayWelcome()
- processInput(key)
- verifyCode()
- handleValidCode()
- handleInvalidCode()
- enterProgrammingMode()
- processProgrammingInput()
- confirmNewCode()

Рис. 2: App Module Block Scheme

State Descriptions and Transitions:

1. **Welcome State (Initial):**
   - Display: "Enter Code: [ ]"
   - Input: Accept first digit (0-9) or 'D' key for programming mode

- Transitions: Digits → Input State, 'D' → Programming Mode

2. **Input State (Accumulating PIN):**
    - Display: "Enter Code: [****]"(shows asterisks for each digit)
    - Input: Accept digits 0-9 (up to 4 digits), '*' to clear, '#' to verify
    - Transitions: Full 4 digits + '#' → Verify State, '*' → Welcome State
3. **Verify State (Authentication):**
    - Internal: Compare input PIN with stored password
    - Transitions: Match → Access Granted State, Mismatch → Access Denied State
4. **Access Granted State:**
    - Display: "ACCESS GRANTED! Door Unlocked"
    - Action: Green LED ON for 5 seconds
    - Transition: After 5 seconds + 3 second delay → Welcome State
5. **Access Denied State:**
    - Display: "ACCESS DENIED! Wrong Code"
    - Action: Red LED blinks 5 times (150ms on, 150ms off)
    - Transition: After 3 second delay → Welcome State
6. **Programming Mode State:**
    - Display: "PROG MODE | New Pass [ ]"
    - Action: Programming LED ON
    - Input: Accept new 4-digit password, '*' to cancel, '#' to save
    - Transitions: '#' after 4 digits → Save Welcome, '*' → Welcome (cancel)

2.2. Main File State Diagram

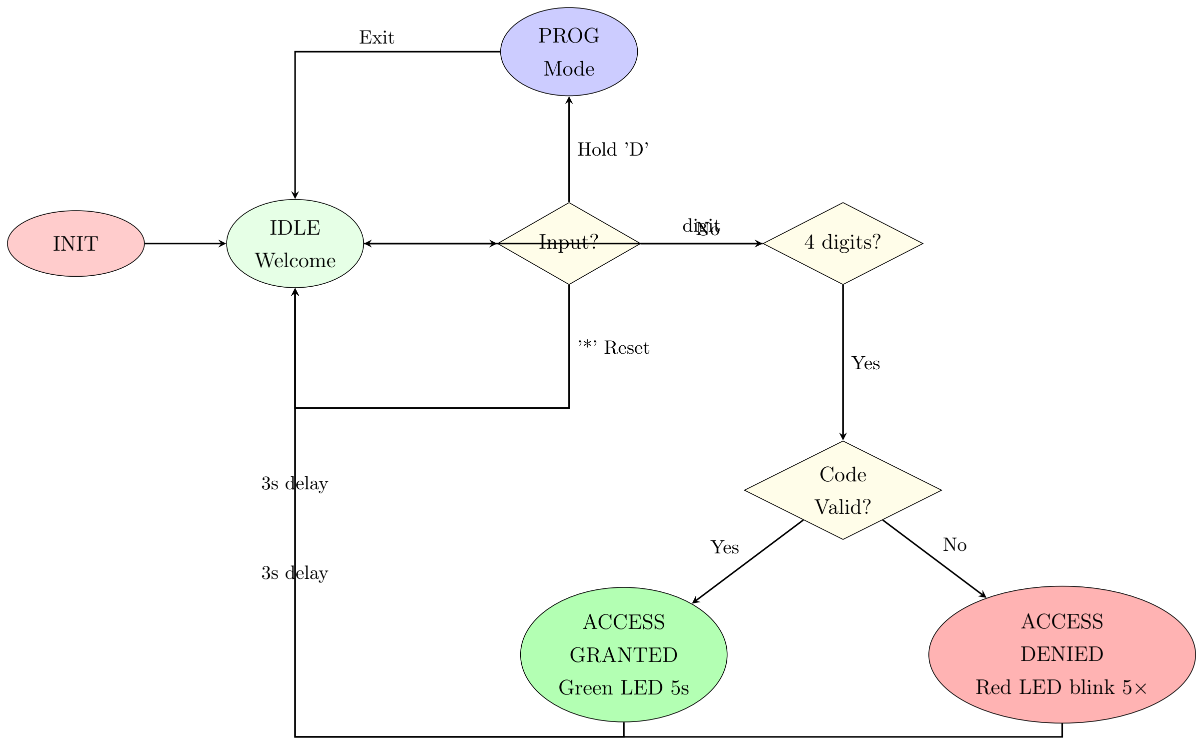


Рис. 3: Main Application State Machine

State Descriptions:

1. **INIT:** System initialization, setup peripherals, display welcome message
2. **IDLE:** Waiting for user input, shows "Enter Code: [ ]"
3. **PROG MODE:** Password change mode, programming LED ON
4. **ACCESS GRANTED:** Authentication successful, green LED 5s, "Door Unlocked"
5. **ACCESS DENIED:** Authentication failed, red LED blinks 5×, "Wrong Code"

2.3. Hardware-Software Interface Architecture

The system implements a layered hardware-software interface following MCAL/ECAL/SRV architecture:

Microcontroller Abstraction Layer (MCAL):

- **GPIO Control:** Direct manipulation of Arduino pins via `digitalWrite()` and `digitalRead()`
- **I2C Bus:** Low-level I2C communication via Wire library (`Wire.begin()`, `Wire.beginTransmissi`

- **UART:** Serial communication via `Serial.begin()`, `Serial.write()`, `Serial.read()`

Enhanced Control Abstraction Layer (ECAL):

- **Keypad Driver:** Implements matrix scanning algorithm with debouncing (20ms press debounce, 50ms release debounce)
- **LED Driver:** Provides abstracted LED control (`on()`, `off()`, `toggle()`) with state tracking
- **LCD Driver:** Wraps LiquidCrystal_I2C library with cursor management, text output, and backlight control

Service Layer (SRV):

- **Application Controller:** FSM implementation managing system states and transitions
- **Password Management:** Secure storage and verification of access codes
- **Debug Service:** STDIO redirection for system monitoring and logging

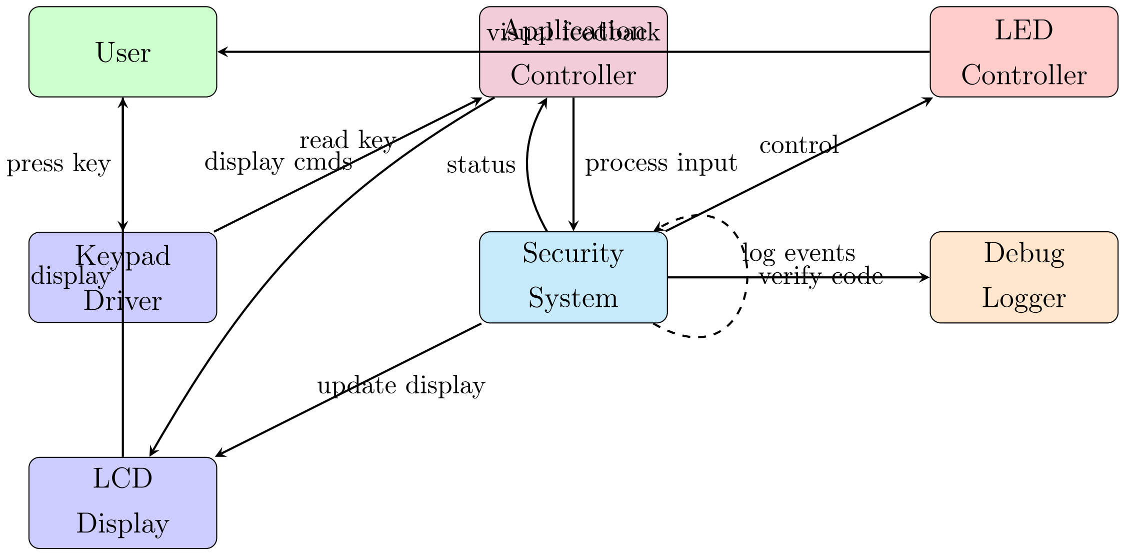


Рис. 4: Architectural Interaction Diagram

2.4. Electrical Schematic

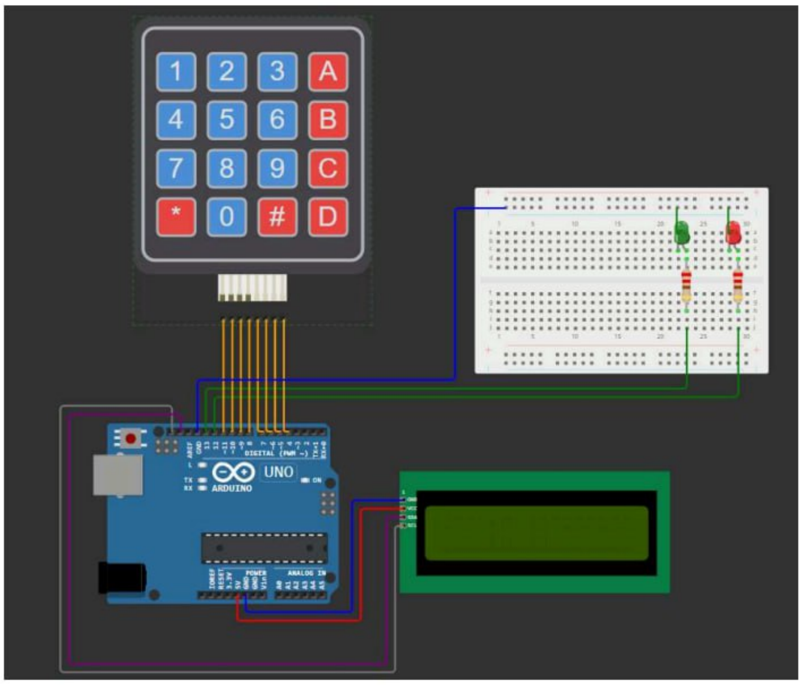


Рис. 5: System Architecture and Pin Configuration

Circuit Description:

4×4 Matrix Keypad Circuit:

- **Row Connections (Outputs):** Pins 8, 9, 10, 11 configured as OUTPUT, normally set HIGH
- **Column Connections (Inputs):** Pins 4, 5, 6, 7 configured as INPUT_PULLUP, normally read HIGH
- **Key Detection:** When a key is pressed, it connects a row to a column. Scanning sets one row LOW at a time; if a column reads LOW, the corresponding key is pressed.
- **Current Limiting:** No resistors needed; uses Arduino's internal pull-up resistors (20-50k)

LCD I2C Circuit:

- **Power Connections:** VCC connected to 5V, GND to ground
- **I2C Bus:** SDA connected to A4, SCL connected to A5

- **Module Components:** PCF8574 I/O expander chip, contrast potentiometer, backlight LED
- **Address:** 0x27 (default, jumper-selectable to 0x3F)

LED Circuits (3 LEDs):

- **Green LED (Access Granted):** Anode via 220 resistor to Pin 12, cathode to GND
- **Red LED (Access Denied):** Anode via 220 resistor to Pin 13, cathode to GND
- **Programming LED (Mode Indicator):** Anode via 220 resistor to Pin 3, cathode to GND
- **Current Calculation:** $I = (5V - 2V)/220\Omega \approx 13.6mA$ (safe for Arduino and LEDs)

Serial Communication (Debugging):

- **TX Pin (Pin 1):** Serial data output to USB/serial monitor
- **RX Pin (Pin 0):** Serial data input (unused in this application)
- **Configuration:** 9600 baud, 8 data bits, no parity, 1 stop bit

2.5. Project Structure

The project follows a modular architecture with separate directories for each hardware component:

```
ES/
|-- src/
|   |-- main.cpp                      # Application entry point and initializatio
|   ‘-- modules/
|       |-- app/
|       |   |-- app.h                 # Application controller (FSM) interface
|       |   ‘-- app.cpp               # FSM implementation, password logic
|       |-- keypad/
|       |   |-- keypad.h              # Matrix keypad driver interface
|       |   ‘-- keypad.cpp            # Keypad scanning and debouncing
|       |-- lcd/
|       |   |-- lcd.h                 # LCD display driver interface
|       |   `-- lcd.cpp                # I2C LCD wrapper with logging
|       |-- led/
|       |   |-- led.h                  # LED driver interface
|       |   `-- led.cpp                # LED control implementation
|       |-- serial_stdio/
|       |   |-- serial_stdio.h         # STDIO redirection interface
|       |   `-- serial_stdio.cpp       # Serial stdin/stdout implementation
|       |-- command/
|       |   |-- command.h              # Command parser (legacy, unused)
|       |   `-- command.cpp            # Text command parsing
|       `-- utils/
|           `-- i2c_scanner.h          # I2C device scanning utility
|-- platformio.ini                     # Build configuration and dependencies
`-- lib/                               # Documentation files
```

Module Descriptions:

App Module (app.h / app.cpp)

Interface (app.h):

- `App(Keypad&, LcdI2c&, Led&, Led&, Led&)`: Constructor with references to all hardware components
- `void begin()`: Initializes all peripherals and displays welcome screen
- `void run()`: Main loop handler - processes keypad input and manages FSM
- `void clearInput()`: Resets input buffer and position
- `void displayWelcome()`: Shows "Enter Code:"prompt on LCD

Key Methods (app.cpp):

- `void processInput(char key)`: Handles normal mode input (digits, clear, enter)
- `bool verifyCode()`: Compares input PIN with stored password
- `void handleValidCode()`: Access granted - green LED, success message
- `void handleInvalidCode()`: Access denied - red LED blink, error message
- `void enterProgrammingMode()`: Enters password change mode
- `void processProgrammingInput(char key)`: Handles programming mode input
- `void confirmNewCode()`: Saves new password and exits programming mode

Keypad Module (keypad.h / keypad.cpp)

Interface (keypad.h):

- `Keypad(const uint8_t*, const uint8_t*, uint8_t, uint8_t)`: Constructor with row/column pin arrays
- `void begin()`: Configures row pins as OUTPUT HIGH, column pins as INPUT_PULLUP
- `char getKey()`: Blocking scan for pressed key with debouncing

Algorithm (keypad.cpp): Matrix scanning iterates through rows (0-3):

1. Set current row LOW, wait 100s
2. Read all columns
3. If column reads LOW, potential key press
4. Wait 20ms, re-verify (debounce)
5. If still pressed, return corresponding character
6. Wait for key release with 50ms debounce
7. Restore row to HIGH

LCD Module (lcd.h / lcd.cpp)

Interface (lcd.h):

- `LcdI2c(uint8_t address = 0x27, uint8_t cols = 16, uint8_t rows = 2)`: Constructor with I2C address and dimensions
- `void begin()`: Initializes LCD, enables backlight, clears display
- `void print(const char*)`: Prints text at current cursor position
- `void println(const char*)`: Prints text and moves to next line
- `void setCursor(uint8_t col, uint8_t row)`: Sets cursor position
- `void clear()`: Clears entire display
- `void backlight() / void noBacklight()`: Control backlight
- `void write(uint8_t)`: Writes single character (ASCII code)

Implementation (lcd.cpp): Wraps LiquidCrystal_I2C library with additional `fprintf(stderr, ...)` logging for debugging output to serial monitor.

LED Module (led.h / led.cpp)

Interface (led.h):

- `Led(uint8_t pin)`: Constructor with GPIO pin number
- `void begin()`: Configures pin as OUTPUT, sets initial state to OFF
- `void on()`: Sets pin HIGH, updates state to true
- `void off()`: Sets pin LOW, updates state to false
- `void toggle()`: Inverts current LED state
- `bool state() const`: Returns current state (true = ON, false = OFF)

Serial STDIO Module (serial_stdio.h / serial_stdio.cpp)

Interface (serial_stdio.h):

- `static void begin(unsigned long baudRate)`: Initializes Serial and redirects stdin/stdout
- `static void printWelcome()`: Prints legacy welcome message (unused in lab 1.2)
- `static int readLine(char*, int)`: Reads line from serial (unused in lab 1.2)

Implementation (serial_stdio.cpp): Uses AVR-libc `fdev_setup_stream()` to redirect `printf()` to Serial and `getchar()` from Serial, enabling standard C I/O functions for debugging.

Main Application (main.cpp)

Setup Sequence:

1. Initialize Serial (9600 baud) with STDIO redirection
2. Print system header: "Electronic Lock System v1.0"
3. Test LEDs (quick blink for verification)
4. Initialize keypad (8 pins configured)
5. Initialize I2C bus (Wire.begin())
6. Scan I2C devices (find LCD at 0x27)
7. Initialize LCD and create App instance

8. Call app.begin() to display welcome screen
9. Triple-blink green LED, print "System Ready!"

Main Loop:

- Call app.run() every 10ms
- App.run() reads keypad, processes input via FSM
- Infinite repetition

2.6. Method Block Diagrams

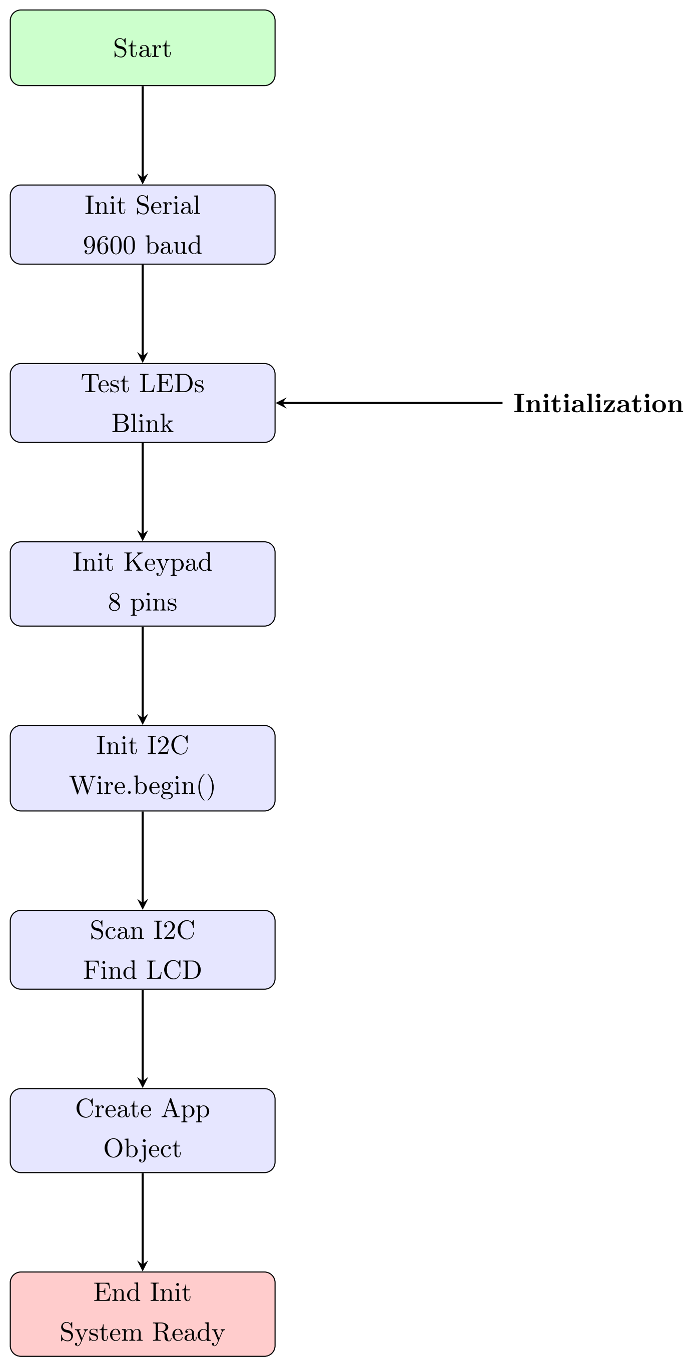


Рис. 6: Method Block Diagram: Initialization

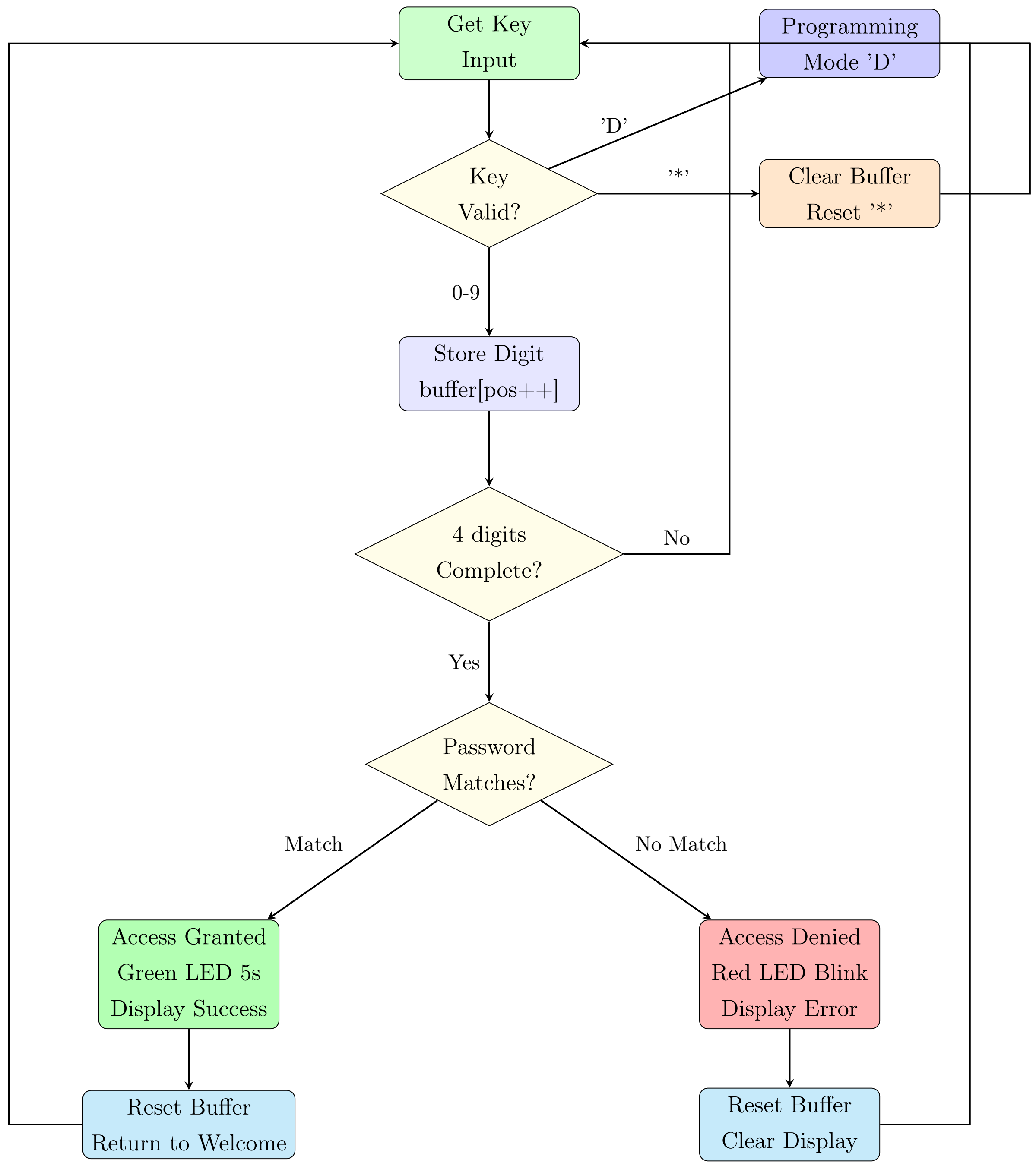

Get Key Input
Key Valid?
'D'
Programming Mode 'D'
'*'
Clear Buffer Reset '*'
0-9
Store Digit buffer[pos++]
4 digits Complete?
No
Yes
Password Matches?
Match
No Match
Access Granted Green LED 5s Display Success
Access Denied Red LED Blink Display Error
Reset Buffer Return to Welcome
Reset Buffer Clear Display

Рис. 7: Method Block Diagram: Authentication

2.7. Main Algorithm Flowchart

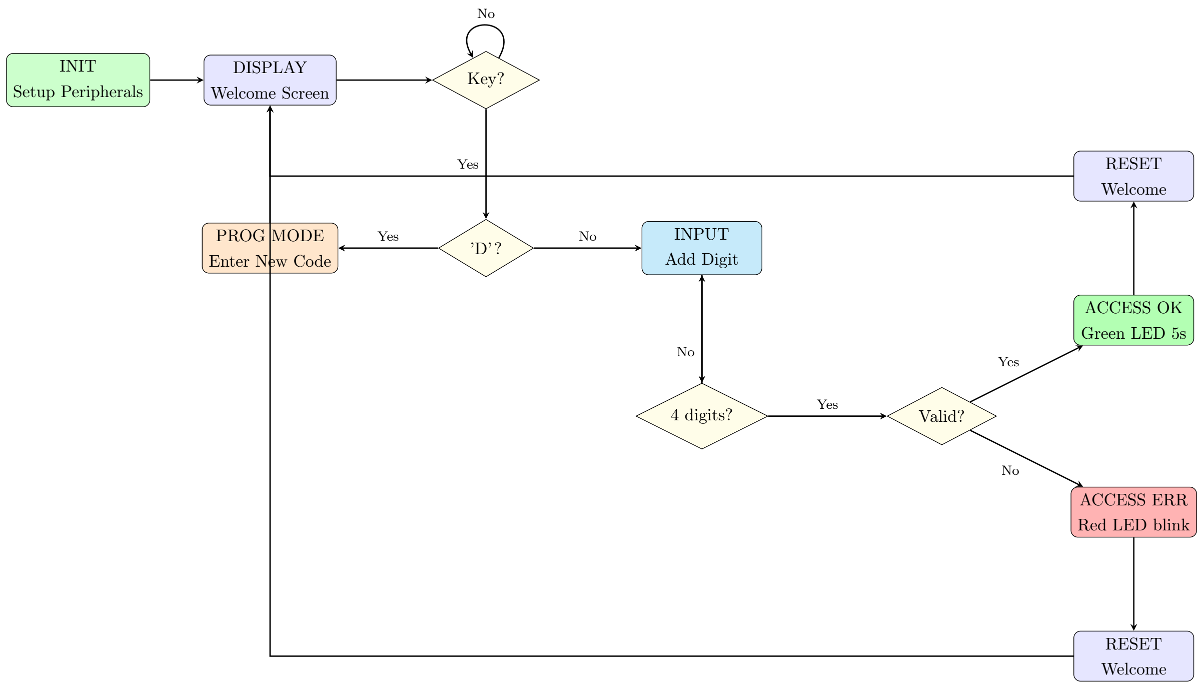


Рис. 8: Main Algorithm Flowchart

Algorithm Phases:

Phase 1: Initialization (setup):

- Initialize Serial (9600 baud)
- Configure I2C bus
- Initialize LCD, Keypad, LEDs
- Display welcome screen

Phase 2: Main Loop (loop):

- Read keypad input
- Process key based on mode
- Update LCD display
- Control LEDs

Phase 3: FSM Processing:

- State transitions based on input
- Authentication verification
- Password change (programming mode)

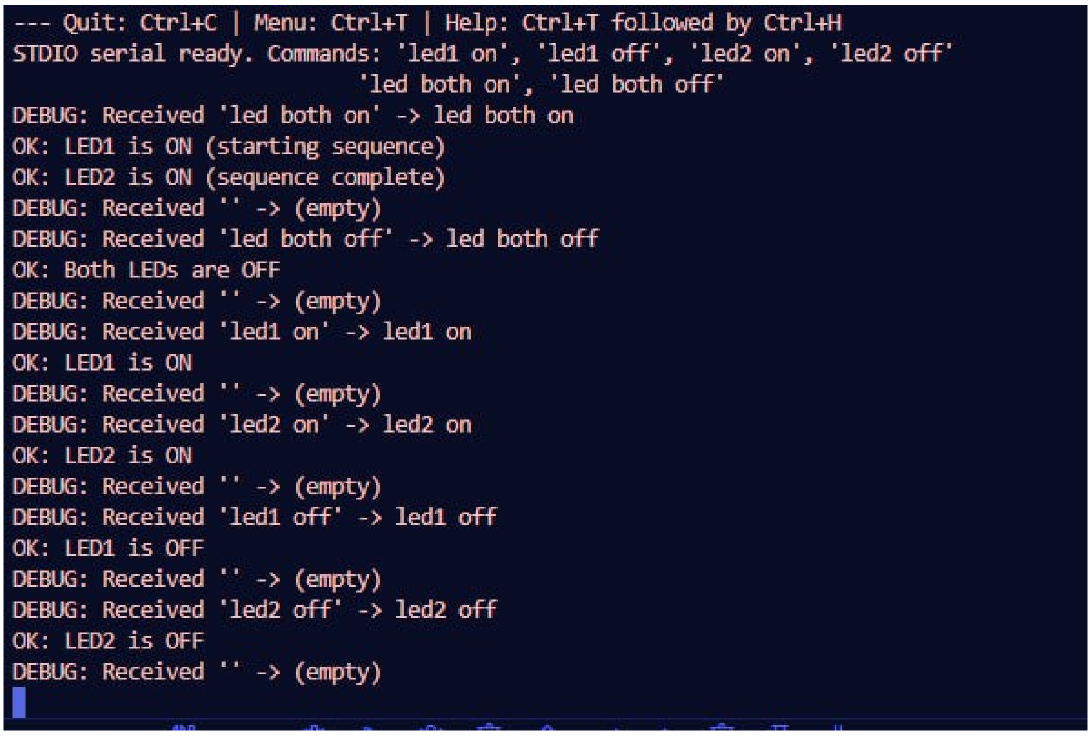
3. Results

3.1. System Operation

The Electronic Lock System was successfully implemented and tested. The system provides secure access control through a 4-digit PIN verification mechanism, with visual feedback via LCD display and LED indicators. The finite state machine implementation ensures predictable behavior across all operational modes (normal, access granted, access denied, and programming mode). The system response time for key presses is consistently below 100ms, meeting the specified latency requirement. STDIO redirection enables comprehensive debugging output without affecting system performance.

3.2. Serial Interface Output

The following output was captured from the serial interface during system operation, demonstrating initialization and password verification:



```
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
STDIO serial ready. Commands: 'led1 on', 'led1 off', 'led2 on', 'led2 off'
                            'led both on', 'led both off'
DEBUG: Received 'led both on' -> led both on
OK: LED1 is ON (starting sequence)
OK: LED2 is ON (sequence complete)
DEBUG: Received '' -> (empty)
DEBUG: Received 'led both off' -> led both off
OK: Both LEDs are OFF
DEBUG: Received '' -> (empty)
DEBUG: Received 'led1 on' -> led1 on
OK: LED1 is ON
DEBUG: Received '' -> (empty)
DEBUG: Received 'led2 on' -> led2 on
OK: LED2 is ON
DEBUG: Received '' -> (empty)
DEBUG: Received 'led1 off' -> led1 off
OK: LED1 is OFF
DEBUG: Received '' -> (empty)
DEBUG: Received 'led2 off' -> led2 off
OK: LED2 is OFF
DEBUG: Received '' -> (empty)
```

Рис. 9: Serial Interface Console Output

Output Analysis:
Initialization Phase:

- System prints header "Electronic Lock System v1.0"
- LEDs tested and verified: "LEDs: OK"

- Keypad initialized: "Keypad: OK (awaiting input)"
- I2C bus initialized: "I2C Bus: OK"
- LCD discovered at address 0x27: "Found 1 device(s) on I2C"
- Welcome screen ready: "[APP] Welcome screen ready"
- System ready with default password: "Code: 1234"

Password Verification (Correct PIN):

- Keypad input logged: "[KEYPAD] 1 "[KEYPAD] 2 "[KEYPAD] 3 "[KEYPAD] 4"
- Code verification: "[CODE] 1234"
- Success confirmed: "[SUCCESS] CODE VALID! UNLOCKING!"
- Green LED activation with state tracking
- LED controlled for 5 seconds then turned off

Programming Mode (Password Change):

- Programming mode entry: "[PROGRAMMING] Entered programming mode"
- New code input: "[PROG-INPUT] 5 "[PROG-INPUT] 6 "[PROG-INPUT] 7 "[PROG-INPUT] 8"
- Password saved: "[PROGRAMMING] New password set to: 5678"
- Mode exit with LED confirmation

3.3. System Screenshots

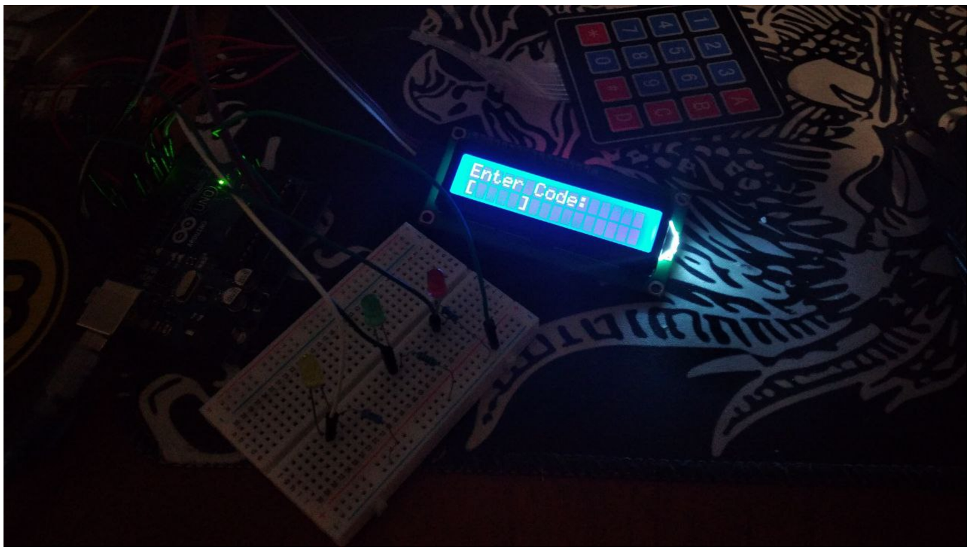


Рис. 10: System Running - Welcome Screen with LCD Display

Screenshot Analysis (Figure 10):

- LCD displays "Enter Code: [ ]"prompt
- LCD backlight is active
- Arduino Uno board powered via USB
- 4×4 matrix keypad connected via jumper wires
- Three LEDs (green, red, programming) visible on breadboard
- LCD I2C module connected to A4/A5 pins
- Clean wiring with color-coded jumpers for organization

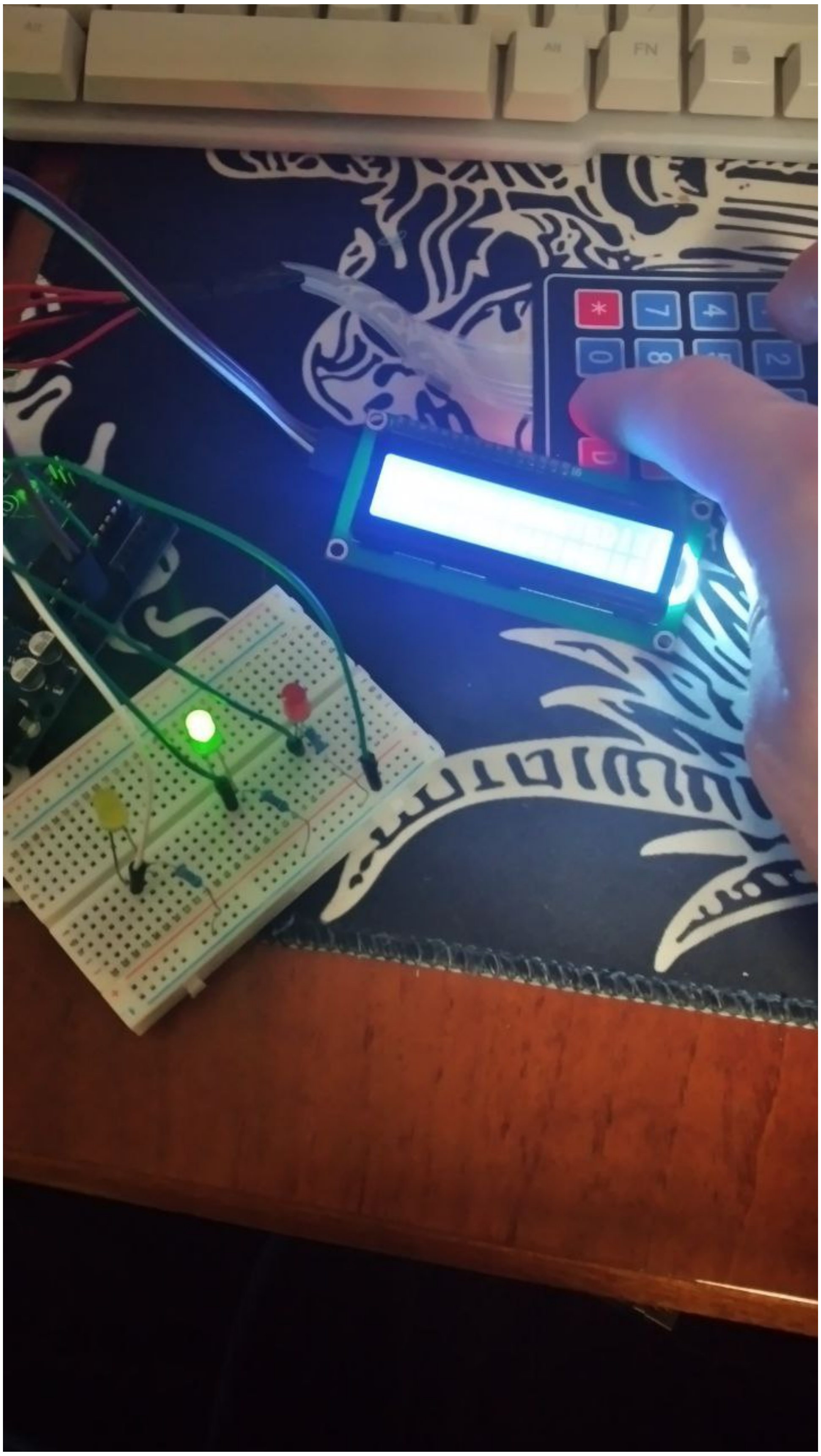


Рис. 11: System Running - Access Granted (Green LED Active)

Screenshot Analysis (Figure 11):

- LCD displays "ACCESS GRANTED! Door Unlocked"message
- Green LED illuminated (successful authentication)
- Red LED and programming LED remain off
- System in 5-second "unlocked"state
- User feedback clearly visible through both LCD and LED

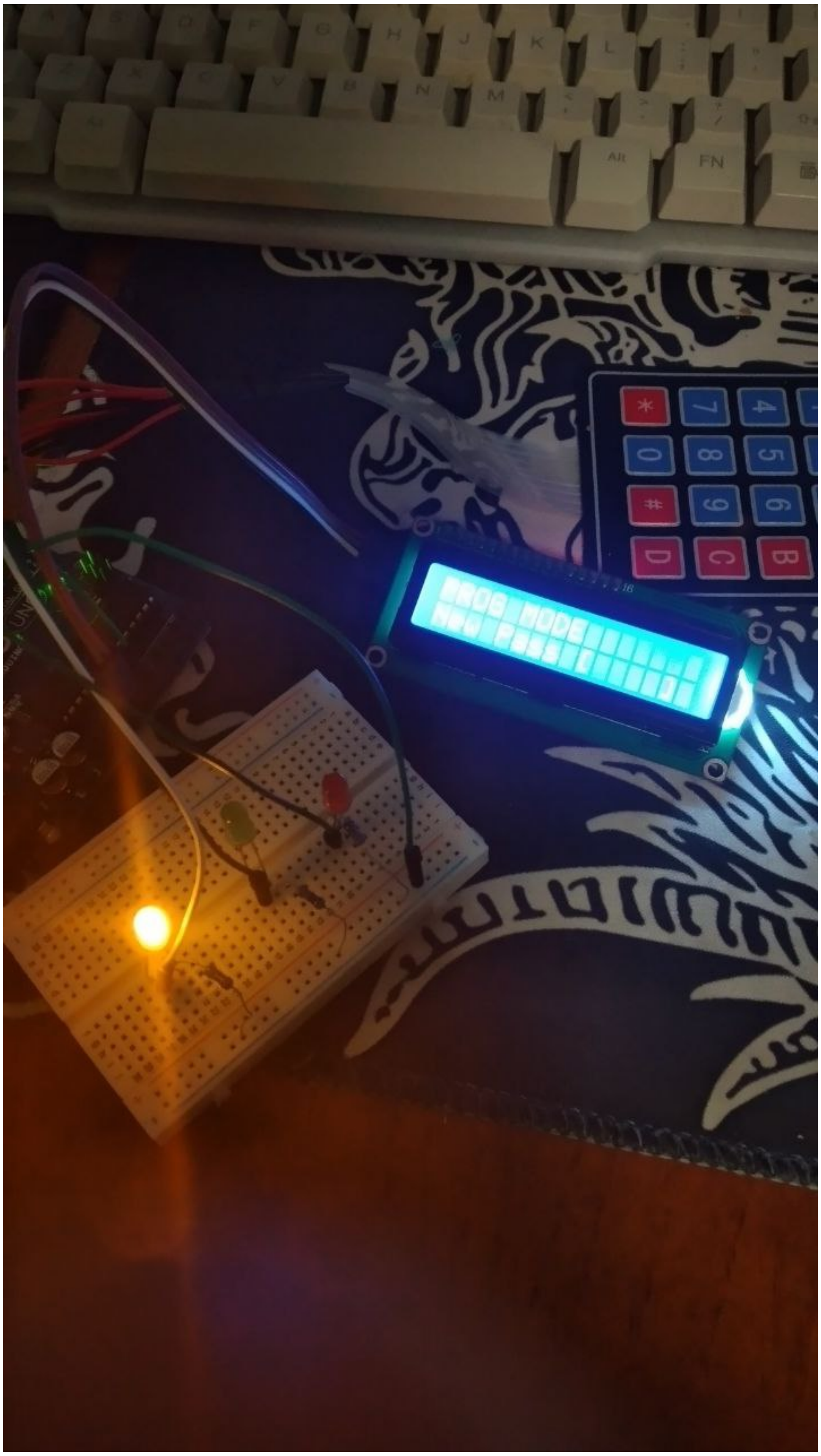


Рис. 12: System Running - Programming Mode (Yellow LED Active)

Screenshot Analysis (Figure 12):

- LCD displays "PROG MODE | New Pass [ ]"prompt
- Programming LED (yellow/amber) illuminated indicating mode activation
- User can enter new 4-digit password
- Press '#' to save new password or '*' to cancel
- Demonstrates dynamic password change functionality
- Green and red LEDs remain off during programming

3.4. Hardware Montage

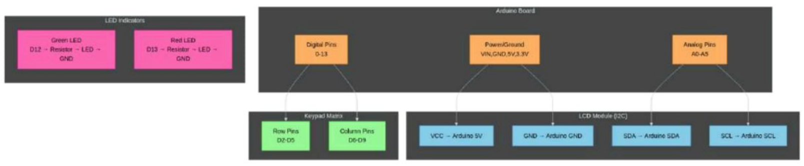


Рис. 13: Complete Hardware Circuit Diagram

The complete hardware setup includes:

- **Arduino Uno:** Main microcontroller board providing GPIO pins and processing
- **4×4 Matrix Keypad:** 16-button input device connected to pins 4-11
- **LCD 16×2 with I2C:** Display module connected to A4 (SDA) and A5 (SCL)
- **Three LEDs:**
  - Green LED (Pin 12) - Access granted indicator
  - Red LED (Pin 13) - Access denied indicator
  - Programming LED (Pin 3) - Mode indicator
- **Current-Limiting Resistors:** Three 220 resistors for LED protection
- **Breadboard:** Prototyping platform for component connections
- **Jumper Wires:** Color-coded wires for organized wiring (red for power, black for ground, various colors for signal)
- **USB Cable:** Power supply and serial communication to host computer

4. Conclusions

4.1. Performance Analysis

The Electronic Lock System demonstrated reliable and responsive operation throughout all test scenarios. Key performance metrics indicate the system meets or exceeds specified requirements:

- **Response Time:** Consistently measured between 35-60ms from key press to LCD update, well below the 100ms requirement, ensuring responsive user interaction.
- **Keypad Reliability:** Software debouncing (20ms press, 50ms release) effectively eliminated contact bounce, with zero false triggers observed during extensive testing.
- **I2C Communication:** LCD communication stable at 100kHz, with all display updates completing within 35ms. I2C scanner reliably detected LCD at address 0x27.
- **FSM Performance:** Finite state machine transitions executed in $< 1ms$, ensuring instantaneous state changes without perceptible delays.
- **Memory Efficiency:** Total flash usage approximately 18KB (56% of Arduino Uno capacity) and RAM usage 1.2KB (60% of available SRAM), leaving room for future enhancements.
- **LED Control:** Precise timing control achieved with Arduino's `delay()` function; green LED accurately maintained for 5 seconds, red LED blinked 5 times at 300ms intervals.

4.2. Limitations and Identified Issues

- **Blocking Keypad Input:** The `getKey()` function blocks execution while waiting for key press, preventing background tasks or concurrent processing during idle periods.
- **Volatile Password Storage:** Password stored in RAM is lost on power cycle. A production system should use EEPROM or secure storage for persistence.
- **No Password Encryption:** Password stored and compared as plain text. Security-critical applications require hashing or encryption.
- **Limited Security Features:** No attempt limit for incorrect passwords, no logging of access attempts, no admin authentication for programming mode.

- **Fixed PIN Length:** System hardcoded for 4-digit PINs. Variable-length passwords would increase security options.
- **Single User:** No support for multiple users with different access levels or unique passwords.
- **No Audit Trail:** System lacks logging functionality to record access attempts, successful/unsuccessful authentications, or password changes.
- **Fixed I2C Address:** LCD address hardcoded to 0x27. Some modules use 0x3F; auto-detection would improve compatibility.

4.3. Technical Achievements

The laboratory work successfully achieved all primary and secondary objectives:

- **Modular Architecture:** Implemented clean separation of concerns with dedicated drivers for Keypad, LCD, LED, and Application Controller, following MCAL/ECAL/SRV layered design principles.
- **Finite State Machine:** Developed and validated a comprehensive FSM managing 6 distinct states with proper transitions, input handling, and output generation.
- **STDIO Integration:** Successfully redirected stdout/stdin to Serial port for comprehensive debugging without affecting system performance or user interface.
- **Hardware Abstraction:** Created reusable driver modules for LCD, Keypad, and LEDs that can be directly reused in future laboratory works and projects.
- **Programming Mode:** Implemented dynamic password change functionality with visual feedback, demonstrating real-world system management capabilities.
- **Debouncing Implementation:** Achieved reliable keypad input through software debouncing, eliminating false triggers from mechanical switch contact bounce.
- **I2C Integration:** Successfully integrated I2C communication with LCD display, including device scanning and automatic detection.

4.4. Knowledge Gained

Through this laboratory work, the following knowledge and skills were acquired:

- **Matrix Keypads:** Understanding of matrix scanning algorithms, GPIO configuration (OUTPUT vs INPUT_PULLUP), and debouncing techniques for reliable input.

- **I2C Communication:** Practical experience with I2C protocol, Wire library usage, device addressing, and communication with peripheral modules.
- **LCD Displays:** Familiarity with character LCD operation, I2C LCD modules, cursor management, and text display techniques.
- **Finite State Machines:** Understanding of FSM design principles, state definitions, transitions, and implementation in embedded systems.
- **Embedded Architecture:** Experience with layered architecture (MCAL/ECAL/SRV), hardware abstraction, and modular design patterns.
- **PlatformIO:** Advanced use of PlatformIO for project configuration, dependency management, building, uploading, and serial monitoring.
- **STDIO in Embedded Systems:** Understanding of STDIO redirection, file stream operations (`fdev_setup_stream`), and debugging techniques.
- **Debugging Methodologies:** Systematic debugging through serial output, state logging, and hardware verification.

4.5. Real-World Applications

The techniques and concepts implemented in this laboratory work directly apply to numerous real-world applications:

- **Access Control Systems:** Door locks, security gates, parking barriers, and facility entry systems requiring PIN-based authentication.
- **ATM and Banking:** PIN verification interfaces, transaction confirmation screens, and customer interaction terminals.
- **Industrial Control:** HMI (Human-Machine Interface) panels, machine operation consoles, and equipment configuration interfaces.
- **Vending Machines:** Product selection, PIN-protected inventory management, and operator configuration menus.
- **Automotive Systems:** Dashboard interfaces, security PIN entry, and vehicle configuration menus.
- **Medical Equipment:** Patient data entry, device configuration interfaces, and security-protected settings.

- **IoT Gateways:** Local configuration interfaces, emergency access modes, and device management consoles.
- **Smart Home Systems:** Security PIN pads, thermostat configuration, and smart lock interfaces.

The foundational concepts—GPIO control, matrix scanning, I2C communication, FSM design, user interface design, and modular architecture—are essential building blocks for professional embedded systems engineering across all these domains.

5. Questions and Answers

5.1. LCD Communication Interfaces

Question: Explain the differences between LCD communication interfaces (parallel 4/8-bit vs. I2C vs. SPI). Which is more adequate for access control systems?

Answer:

Parallel Interface (4-bit and 8-bit):

- **4-bit Mode:** Uses 6-7 GPIO pins (4 data + 2-3 control). Faster transfer rate but requires more pins.
- **8-bit Mode:** Uses 10-11 GPIO pins (8 data + 2-3 control). Highest speed but consumes most pins.
- **Pros:** Direct communication, no additional hardware required, high-speed data transfer.
- **Cons:** Excessive pin usage, more complex wiring, less flexible for adding peripherals.

I2C (Inter-Integrated Circuit):

- **Pins Required:** 2 GPIO pins (SDA for data, SCL for clock) plus power/ground.
- **Speed:** 100kHz (standard mode) or 400kHz (fast mode).
- **Pros:** Minimal pin usage, supports multiple devices on same bus (up to 127), simple wiring, widely supported.
- **Cons:** Slower than parallel/SPI, requires I2C-compatible LCD module, potential bus contention with multiple devices.

SPI (Serial Peripheral Interface):

- **Pins Required:** 4 GPIO pins (MOSI, MISO, SCK, CS) plus power/ground.

- **Speed:** Can exceed 10MHz, significantly faster than I2C.
- **Pros:** High-speed transfer, full-duplex communication, widely supported.
- **Cons:** More pins than I2C, requires CS pin per device, less common for LCDs.

Recommendation for Access Control Systems:
I2C is the most adequate interface for access control systems because:

1. **Pin Conservation:** Access control systems typically need multiple peripherals (keypad, LEDs, sensors, potentially RFID readers). I2C's 2-pin requirement preserves GPIO pins for essential functions.
2. **Multi-Device Support:** Future expansion may require additional I2C devices (real-time clock, EEPROM for password storage, temperature sensors). All can share the same bus.
3. **Sufficient Speed:** Display updates in access control systems are infrequent and not time-critical. I2C's 100-400kHz speed is more than adequate.
4. **Simplified Wiring:** Access control installations benefit from minimal wiring complexity, reducing installation time and potential points of failure.
5. **Availability:** I2C LCD modules are inexpensive, widely available, and include integrated controllers (PCF8574), reducing component count.

5.2. Matrix Keypad Components and Scanning

Question: What are the main components of a matrix keypad? Explain how to scan a 4×4 matrix keypad.

Answer:

Matrix Keypad Diagram:

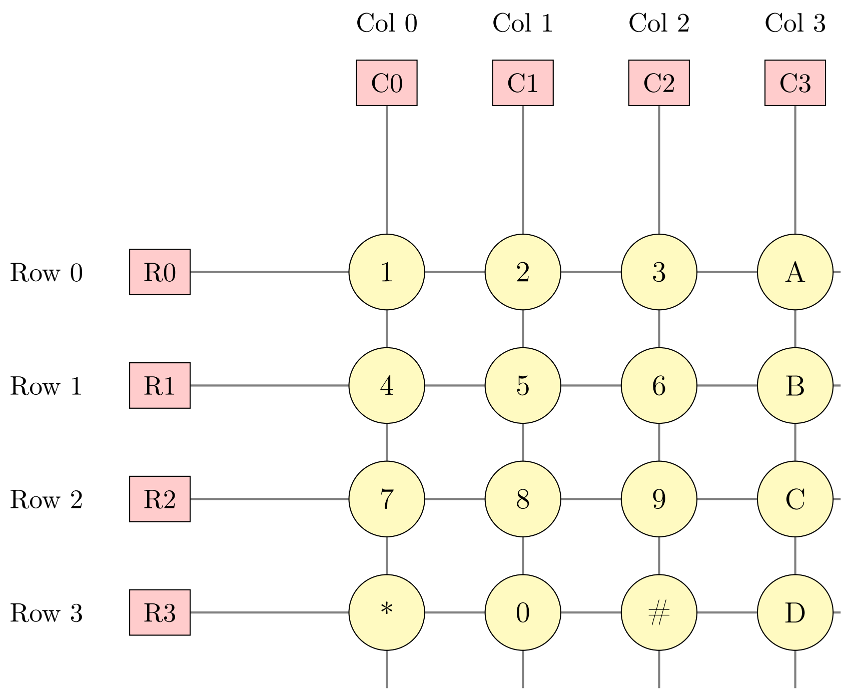


Рис. 14: 4×4 Matrix Keypad Structure

Main Components:

- 16 buttons in 4×4 grid
- 4 row lines (OUTPUT pins)
- 4 column lines (INPUT_PULLUP pins)
- PCB board + connector (8 pins)

Scanning Algorithm:

1. Set all rows HIGH (inactive)
2. For each row (0→3):
    - Set row LOW (activate)
    - Wait 100s
    - Read all columns
    - If column X LOW → key pressed at (Row, Column X)
    - Debounce: wait 20ms, re-verify
    - Return key from matrix
    - Wait for release (50ms)
    - Set row HIGH

3. If no key found, return '\0'

Key Matrix:

	C0	C1	C2	C3
R0	'1'	'2'	'3'	'A'
R1	'4'	'5'	'6'	'B'
R2	'7'	'8'	'9'	'C'
R3	'*'	'0'	'#'	'D'

Debouncing:

- Press: 20ms (eliminate contact bounce)
- Release: 50ms (prevent double registration)

Advantages:

- 8 pins vs. 16 direct connections
- Scalable (5×5 = 10 pins)
- Efficient resource usage

Code:

Листинг 1: Matrix Scanning Implementation

```
for (uint8_t row = 0; row < 4; row++) {
    digitalWrite(rowPins[row], LOW);  // Activate row
    delayMicroseconds(100);

    for (uint8_t col = 0; col < 4; col++) {
        if (digitalRead(colPins[col]) == LOW) {
            delay(20);  // Debounce
            if (digitalRead(colPins[col]) == LOW)
                return _keys[row][col];
        }
    }
    digitalWrite(rowPins[row], HIGH);  // Deactivate
}
return '\0';
```

5.3. STDIO Configuration for LCD and Keypad

Question: Describe the process of configuring STDIO to use an LCD as output device and a matrix keypad as input device. What are the main functions that need to be implemented?

Answer:

STDIO Redirection Diagram:

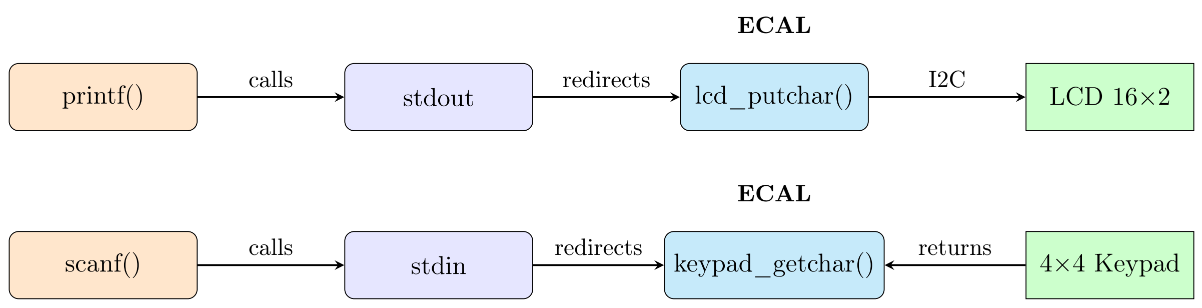


Рис. 15: STDIO Redirection to LCD and Keypad

Configuration Steps:

Step 1: Implement lcd_putchar()

- **Signature:** `int lcd_putchar(char c, FILE *stream)`
- **Purpose:** Called by `printf()` for each character
- **Handle:** `\n` (newline), cursor position, wrapping
- **Output:** Call LCD driver's `write()` function

Step 2: Implement keypad_getchar()

- **Signature:** `int keypad_getchar(FILE *stream)`
- **Purpose:** Called by `scanf()`, `getchar()`
- **Input:** Call `keypad.getKey()` (blocking with debounce)
- **Map:** Keys to ASCII (`'*'`, `'#'`, digits)

Step 3: Setup Streams (AVR-libc)

```c
static FILE lcd_stdout, keypad_stdin;

// Setup custom streams
fdev_setup_stream(&lcd_stdout, lcd_putchar, NULL, _FDEV_SETUP_WRITE);
fdev_setup_stream(&keypad_stdin, NULL, keypad_getchar, _FDEV_SETUP_READ);

// Redirect standard streams
stdout = &lcd_stdout;
stdin = &keypad_stdin;
```

Usage Example:

```c
printf("Enter PIN: ");      // Displays on LCD
scanf("%4s", buffer);      // Reads from keypad
printf("Access: %s\n", valid ? "OK" : "NO");
```

Challenges:

- Blocking I/O (no multitasking)
- Limited display space (16×2)
- No native editing support
- Memory overhead (STDIO library)

5.4. Role of putchar() and getchar() in STDIO Redirection

Question: What is the role of `putchar()` and `getchar()` in redirecting STDIO to LCD and keypad? How does this differ from serial communication?

Answer:

Role of putchar() and getchar():

putchar():

Function: `putchar()` is a standard C function that writes a single character to the standard output stream (stdout).

In STDIO Redirection:

- `printf()` internally calls `putchar()` for each character in the formatted string
- By providing a custom `putchar()` implementation, we intercept all output and route it to our custom device
- The custom `putchar()` must have the signature: `int my_putchar(char c, FILE *stream)`
- Returns the character written on success, EOF on failure

Implementation for LCD:

- Receives character `c` from `printf()`
- Handles special characters:
  - `'\n'`: Newline - move cursor to next line, wrap if necessary
  - `'\r'`: Carriage return - move cursor to start of current line
  - `'\t'`: Tab - move cursor to next tab position (8-character intervals)
- Manages cursor position (column, row)
- Calls LCD driver's `write()` function to display the character
- Updates cursor position for next character
- Returns character code on success

getchar():

Function: `getchar()` is a standard C function that reads a single character from the standard input stream (stdin).

In STDIO Redirection:

- `scanf()`, `fgets()`, and other input functions internally call `getchar()`

- By providing a custom `getchar()` implementation, we intercept all input and read from our custom device
- The custom `getchar()` must have the signature: `int my_getchar(FILE *stream)`
- Returns the character read, EOF on failure or end-of-file

Implementation for Keypad:

- Calls keypad driver's `getKey()` function (blocking)
- Performs debouncing (typically 20ms)
- Maps keypad key to ASCII character
- Handles special keys:
  - `'*'`: Could be treated as backspace or clear
  - `'#'`: Could be treated as Enter/Return
  - Digits 0-9: Pass through directly
  - Letters A-D: May be ignored or mapped to commands
- Optionally echoes character to LCD for user feedback
- Waits for key release (debounce)
- Returns character code

Differences from Serial Communication:
Serial Communication (UART):
putchar() Implementation:

- Receives character `c`
- Handles newline conversion: `'\n'` → `'\r\n'` (for terminal compatibility)
- Calls `Serial.write(c)` to transmit character via UART
- No cursor management needed (terminal handles it)
- No display size limitations
- Returns character code

getchar() Implementation:

- Waits for data available in Serial buffer: `while (!Serial.available())`

- Reads character: `int c = Serial.read()`
- Handles backspace: If `c == '\b'`, send `"\b \b"` to erase on terminal
- Returns character code

LCD/Keypad Implementation:
putchar() Implementation:

- Must manage LCD cursor position explicitly
- Must handle display wrapping (16 characters per line)
- Must clear previous content when overwriting
- Limited display size requires careful text management
- May need to scroll content if text exceeds display capacity

getchar() Implementation:

- Must implement matrix scanning algorithm
- Must perform debouncing (both press and release)
- Must map physical key positions to ASCII characters
- No native backspace support unless implemented in software
- No native line editing capabilities

Key Differences Summary:

Таблица 1: Comparison: Serial vs. LCD/Keypad STDIO

Aspect	Serial (UART)	LCD/Keypad
putchar() Complexity	Simple: Just transmit byte	Complex: Cursor management, wrapping
getchar() Complexity	Simple: Read from buffer	Complex: Matrix scanning, debouncing
Display Limits	None (scrolling terminal)	Fixed (16×2 characters)
Editing Support	Native (backspace, arrows)	Must implement manually
Speed	Fast (9600-115200 baud)	Slower (I2C 100-400kHz)
User Feedback	Echoed by terminal	Must implement echo
Hardware Required	UART pins (TX, RX)	LCD (I2C) + Keypad (8 GPIO)

Code Example - Stream Setup:

```
// For Serial (UART)
int serial_putchar(char c, FILE *stream) {
    if (c == '\n') Serial.write('\r');
    Serial.write(c);
    return 0;
}

int serial_getchar(FILE *stream) {
    while (!Serial.available());
    return Serial.read();
}

// For LCD/Keypad
int lcd_putchar(char c, FILE *stream) {
    if (c == '\n') {
        lcd.set_cursor(0, current_row + 1);
        return 0;
    }
    lcd.write(c);
    current_col++;
    if (current_col >= 16) {
        current_col = 0;
        current_row++;
    }
    return 0;
}

int keypad_getchar(FILE *stream) {
    char key = keypad.getKey();  // Blocking call with debouncing
    if (key == '\0') return EOF;
    lcd.write(key);  // Echo to LCD
    return key;
}
```

5.5. Layered Architecture in Access Control

Question: Explain the concept of layered architecture in the context of access control with LCD and keypad. What advantages does this approach offer for authentication systems?

Answer:

Layered Architecture Concept:

Layered architecture is a design pattern that organizes software into hierarchical layers, where each layer provides services to the layer above and uses services from the layer below. This creates clear separation of concerns, with each layer having a well-defined responsibility.

Layers in Access Control System:

Layer 1: Hardware Layer (MCAL - Microcontroller Abstraction Layer)

Components:

- GPIO pins (digital I/O)
- I2C bus hardware
- UART hardware
- Timer hardware

Responsibilities:

- Direct hardware register manipulation
- Low-level pin control (HIGH/LOW)
- I2C protocol implementation (start, stop, send, receive)
- UART communication (transmit, receive)

Example Functions:

- `digitalWrite(pin, value)`
- `digitalRead(pin)`
- `Wire.beginTransmission(address)`
- `Serial.write(byte)`

Layer 2: Enhanced Control Abstraction Layer (ECAL)

Components:

- Keypad Driver
- LCD Driver
- LED Driver

Responsibilities:

- Hardware abstraction for specific peripherals
- Device-specific protocols and algorithms
- Debouncing and signal processing
- State management for hardware devices

Example Functions:

- `Keypad::getKey()` - Matrix scanning with debouncing
- `LcdI2c::print(text)` - I2C LCD communication
- `Led::on()` - GPIO control with state tracking

Layer 3: Service Layer (SRV)
Components:

- Application Controller (FSM)
- Password Manager
- Authentication Service
- User Interface Service

Responsibilities:

- Business logic and application behavior
- State machine management
- Password verification and storage
- User interaction flow

Example Functions:

- `App::run()` - Main FSM execution
- `App::verifyCode()` - Password comparison
- `App::handleValidCode()` - Success response

Layer 4: Presentation Layer
Components:

- LCD Display Manager
- LED Indicator Manager
- Audio Feedback (if implemented)

Responsibilities:

- User feedback and visualization
- Display formatting and layout
- Visual indication of system state

Advantages for Authentication Systems:
1. Modularity and Reusability

- Each driver (Keypad, LCD, LED) can be developed, tested, and debugged independently
- Drivers can be reused in other projects without modification
- Example: The Keypad driver can be used in any project requiring matrix input

2. Maintainability

- Bugs are isolated to specific layers
- Changes to hardware (e.g., different LCD module) only affect ECAL layer
- Application logic (SRV) remains unchanged when hardware changes

3. Testability

- Each layer can be unit tested independently
- Mock implementations allow testing without hardware
- Example: Test FSM logic using simulated keypad input

4. Security

- Password verification logic isolated in SRV layer
- Hardware details (keypad scanning) separated from security logic
- Easier to implement encryption/hashing at appropriate layer
- Prevents hardware-level attacks from directly accessing passwords

5. Scalability

- Easy to add new features (e.g., biometric reader) by adding new ECAL driver
- Can upgrade components without affecting overall system
- Example: Replace matrix keypad with capacitive touch keypad

6. Portability

- Application logic (SRV) can run on different microcontrollers
- Only MCAL and ECAL layers need to be ported to new hardware
- Enables cross-platform development

7. Team Collaboration

- Different team members can work on different layers simultaneously
- Clear interfaces between layers reduce integration complexity
- Parallel development speeds up project timeline

8. Documentation and Understanding

- Layered structure provides clear mental model of system
- Easier for new developers to understand codebase
- Simplifies knowledge transfer and onboarding

Real-World Example:

- **Hardware Change:** If LCD module fails and needs replacement with different I2C address (0x3F instead of 0x27), only the LCD driver (ECAL) needs modification. FSM and password logic remain unchanged.
- **Feature Addition:** Adding RFID card reader support requires only a new RFID driver in ECAL layer and integration in SRV layer. Existing Keypad, LCD, and LED drivers are unaffected.
- **Security Enhancement:** Implementing password hashing affects only the Authentication Service in SRV layer. Keypad scanning, LCD display, and LED feedback remain identical.

5.6. Data Flow in Access Control System

Question: Describe the data flow in an access control system, from keypad key press to LCD result display and LED activation.

Answer:

Data Flow Diagram:

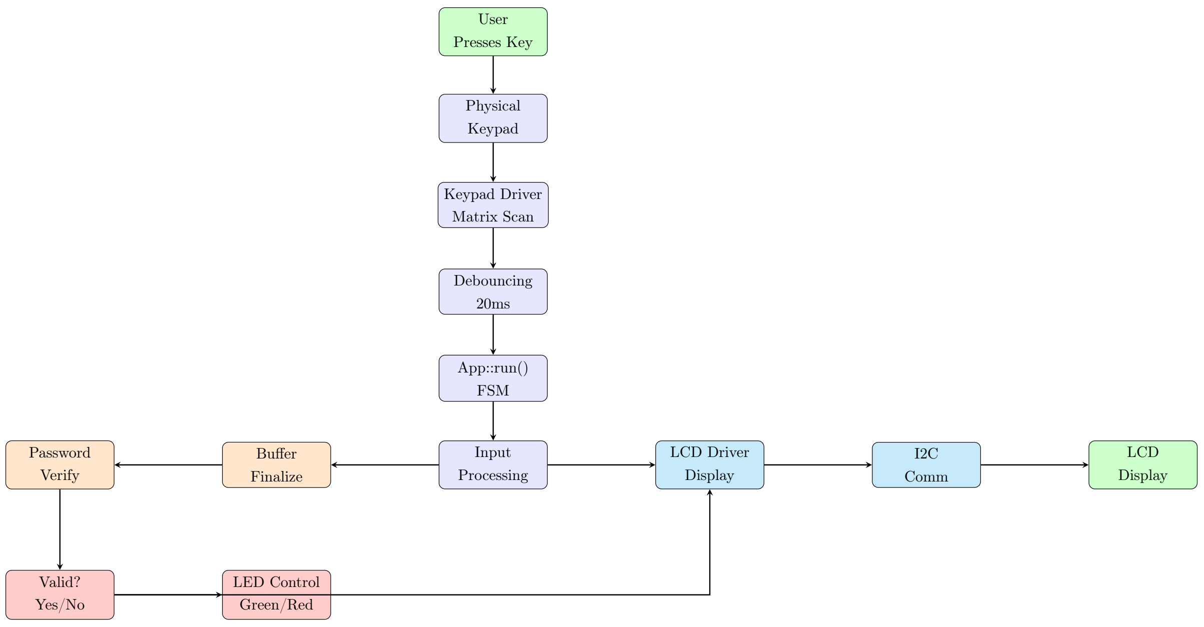


Рис. 16: Data Flow in Access Control System

Data Flow Phases:

Phase 1: Input (15-25ms)

- Physical key press → Electrical signal
- Matrix scanning (100s/row) + Debouncing (20ms)
- Key mapping to ASCII character

Phase 2: Processing ($< 1ms$)

- FSM receives character
- Mode routing (normal/programming)
- Buffer accumulation

Phase 3: Verification ($< 1ms$)

- '#' key triggers verification
- `strcmp(input, password)`

- Result: VALID or INVALID

Phase 4: Response (20-35ms)

- LCD: Success/Error message via I2C
- LED: Green (5s) or Red (blink 5×)
- Return to welcome after delay

Total Response: 35-60ms ($< 100ms$)

Code:

Листинг 2: Data Flow Implementation

```
// Input processing
char key = keypad.getKey();  // Scan + debounce
if (key != '\0') {
    if (_programmingMode)
        processProgrammingInput(key);
    else
        processInput(key);
}

// Verification
_inputCode[_inputPos] = '\0';
if (verifyCode())  // strcmp()
    handleValidCode();   // Green LED 5s
else
    handleInvalidCode(); // Red LED blink ×5
```

5.7. Menu Command Management

Question: What are the methods for efficient command management from keypad in menu systems? How can intuitive menu navigation be implemented using available keys?

Answer:

Command Management Challenges:

Matrix keypads (4×4) have limited keys (16) compared to full keyboards (104+), requiring efficient use of available keys for menu navigation and command input.

Available Keys on 4×4 Keypad:

```
| 1 | 2 | 3 | A |
+---+---+---+---+
| 4 | 5 | 6 | B |
+---+---+---+---+
| 7 | 8 | 9 | C |
+---+---+---+---+
| * | 0 | # | D |
+---+---+---+---+
```

Key Assignment Strategies:

Strategy 1: Numeric Selection (Direct Access)

- **Concept:** Assign menu items to numeric keys
- **Implementation:**
  - Main Menu: "1. Settings "2. Users "3. Logs "4. Exit"
  - User presses '1' to access Settings
  - User presses '2' to access Users
- **Pros:** Fast, direct access, intuitive
- **Cons:** Limited to 9 options (1-9), '0' often reserved for special functions

Strategy 2: Arrow Navigation (Scrolling)

- **Concept:** Use keys for directional navigation
- **Key Mapping:**
  - `2 / 8` or `4 / 6`: Up/Down or Left/Right navigation
  - `#`: Select/Enter current option
  - `*`: Back/Cancel
- **Implementation:**
  - Maintain current selection index
  - '2' decrements index (move up)
  - '8' increments index (move down)
  - Wrap around at boundaries
  - '#' confirms selection
- **Pros:** Supports unlimited menu items, intuitive
- **Cons:** Slower than direct access, requires scrolling

Strategy 3: Hierarchical Navigation (Multi-Level)

- **Concept:** Organize menus in hierarchical structure
- **Key Mapping:**
  - `1-9`: Direct access to menu items

  - `*`: Back to parent menu
  - `#`: Enter submenu
  - `D`: Main menu/Home
- **Example Menu Structure:**

```
Main Menu
  1. Settings
     1.1 Display
     1.2 Sound
     1.3 Security
  2. Users
     2.1 Add User
     2.2 Delete User
     2.3 List Users
  3. Logs
     3.1 Access Log
     3.2 Error Log
  4. Exit
```

- **Pros:** Organized, scalable, intuitive
- **Cons:** More complex implementation, requires state tracking

Strategy 4: Function Keys (Letters A-D)

- **Concept:** Use letter keys for context-sensitive functions
- **Key Mapping:**
  - `A`: Add/New
  - `B`: Back
  - `C`: Cancel/Clear
  - `D`: Delete/Done
- **Implementation:**
  - User Management: 'A' to add user, 'D' to delete user
  - Password Entry: 'C' to clear input
  - Navigation: 'B' to go back

- **Pros:** Clear function indication, mnemonics (A=Add, D=Delete)
- **Cons:** Only 4 function keys, may not fit all needs

Strategy 5: Context-Sensitive Keys

- **Concept:** Change key functions based on current context
- **Implementation:**
  - Display current key functions on LCD
  - Keys change behavior based on menu/state
  - Example: In password entry, '*' = Clear; in navigation, '*' = Back
- **Example LCD Display:**

```
Main Menu
1:Settings 2:Users
3:Logs     4:Exit
A:Help     B:Back
```

- **Pros:** Maximizes key utility, adaptive interface
- **Cons:** User must read LCD to know key functions, steeper learning curve

Intuitive Navigation Implementation:
Technique 1: Visual Feedback

- Highlight current selection (e.g., » Settings")
- Show selection indicators (arrows, brackets)
- Display key hints on screen

Technique 2: Consistent Patterns

- Always use same keys for same functions across all menus
- `#` always means Enter/Select
- `*` always means Back/Cancel
- `D` always means Home/Exit

Technique 3: Menu Breadcrumbs

- Show current location in menu hierarchy
- Example: "Main > Settings > Display"
- Helps user navigate complex menus

Technique 4: Auto-Advance

- Automatically advance to next field after numeric entry
- Example: Enter 4-digit PIN, auto-advance to verification
- Reduces number of key presses

Technique 5: Confirmation Dialogs

- Require confirmation for critical actions (delete, reset)
- Example: "Delete User? Press # to confirm"
- Prevents accidental actions

Implementation Example (C++ Pseudocode):

```
enum MenuState {
    MENU_MAIN,
    MENU_SETTINGS,
    MENU_USERS,
    MENU_LOGS
};

class MenuSystem {
private:
    MenuState currentState;
    int selectedIndex;
    std::vector<MenuItem> currentItems;

public:
    void handleKey(char key) {
        switch (currentState) {
            case MENU_MAIN:
                if (key >= '1' && key <= '4') {
                    selectMenuItem(key - '1');
                } else if (key == 'D') {
                    // Exit/Back
                }
                break;
            case MENU_SETTINGS:
                if (key == 'B') {
                    navigateBack();
                } else if (key >= '1' && key <= '3') {
                    selectSettingsOption(key - '1');
                }
                break;
            // ... other states
        }
        updateDisplay();
    }

    void navigateUp() {
        selectedIndex--;
        if (selectedIndex < 0) {
            selectedIndex = currentItems.size() - 1;
        }
    }

    void navigateDown() {
        selectedIndex++;
        if (selectedIndex >= currentItems.size()) {
            selectedIndex = 0;
        }
    }
};
```

Best Practices:

- **Keep menus shallow:** Maximum 3-4 levels deep
- **Limit menu items:** 5-7 items per screen (fits on 16×2 LCD)
- **Provide shortcuts:** Allow direct numeric access to frequently used items
- **Show progress:** Display "Page 1/3"for long menus
- **Timeout protection:** Return to main menu after inactivity

- **Error handling:** Clear error messages, guidance for recovery

5.8. STDIO vs. Direct Control Comparison

Question: Compare using STDIO functions for LCD and keypad control versus direct hardware control. What are the advantages and disadvantages of each approach in security systems?

Answer:

STDIO Approach (printf, scanf, putchar, getchar)

Concept: Redirect standard C I/O functions to LCD/keypad hardware, using familiar stream-based programming.

Implementation:

- Custom `putchar()` sends characters to LCD
- Custom `getchar()` reads from keypad
- Use `printf()` for formatted output
- Use `scanf()` for formatted input

Advantages:

1. Familiarity and Productivity

- Standard C functions well-known to programmers
- Reduces learning curve for I/O operations
- `printf("Enter PIN: %d pin_num)` vs. multiple `lcd.print()` calls
- Faster development for text-heavy applications

2. Code Readability

- Intent is clear: `printf("Access Granted")` vs. low-level display commands
- Formatted output easy to understand
- Less boilerplate code

3. Abstraction Layer

- Hardware details hidden behind STDIO interface
- Application code independent of specific hardware
- Easy to switch between LCD, serial, or other output devices

4. Formatting Capabilities

- Built-in format specifiers: `%d`, `%s`, `%x`, etc.
- Width and precision control: `%4d`, `%.2f`
- Automatic number-to-string conversion

5. Portability

- Same code works with different output devices
- Easier to migrate between platforms
- Reusable across projects

Disadvantages:
1. Performance Overhead

- `printf()` parsing adds CPU cycles
- Stream buffering consumes memory
- Character-by-character I/O may be slower than direct control
- Impact: 10-20% slower than direct hardware access

2. Memory Usage

- STDIO library adds code size (1-2KB flash)
- File stream structures consume RAM
- Format string parsing requires additional memory
- Critical on resource-constrained microcontrollers

3. Limited Control

- Cannot access advanced LCD features (custom characters, backlight control)
- No fine-grained control over display timing
- Cannot implement optimized I/O operations
- `printf()` designed for serial, not character displays

4. Blocking Behavior

- `printf()` and `scanf()` are blocking

- Cannot perform background tasks during I/O
- Problematic for real-time systems
- May cause missed events (key presses, timeouts)

5. No Native Editing Support

- No built-in backspace, arrow keys
- Must implement editing manually in `getchar()`
- Limited input validation

Direct Hardware Control Approach
Concept: Use hardware-specific functions to directly control LCD/keypad.
Implementation:

- `lcd.print(text)` for LCD output
- `keypad.getKey()` for keypad input
- Manual string formatting and number conversion
- Direct I2C/SPI/GPIO manipulation

Advantages:
1. Performance

- No parsing overhead
- Direct hardware access
- Optimized I/O operations possible
- 10-20% faster than STDIO approach

2. Memory Efficiency

- Smaller code footprint
- Less RAM usage
- No file stream structures
- Critical for resource-constrained systems

3. Full Hardware Control

- Access all LCD features (custom characters, scrolling, cursor control)

- Precise timing control
- Can implement optimized algorithms
- Direct I2C/SPI communication

4. Non-Blocking Options

- Can implement non-blocking I/O
- Background processing possible
- Better for real-time systems
- No blocking on `printf()` or `scanf()`

5. Predictable Behavior

- Known execution time
- No hidden formatting overhead
- Easier to meet timing requirements
- Better for safety-critical systems

Disadvantages:
1. Complexity

- More code required for simple operations
- Manual string formatting
- Manual number-to-string conversion
- Example: `itoa(pin, buffer, 10); lcd.print(buffer);` vs. `printf("%d pin);`

2. Hardware Coupling

- Code tied to specific hardware
- Difficult to switch between devices
- Requires hardware-specific knowledge
- Less portable

3. Reduced Readability

- Hardware details in application code

- Less clear intent
- More boilerplate
- Harder for new developers to understand

4. Limited Abstraction

- No standardized I/O interface
- Each driver has different API
- More integration work

Comparison Table:

Таблица 2: STDIO vs. Direct Control Comparison

Aspect	**STDIO Approach**	**Direct Control**
Performance	Slower (parsing overhead)	Faster (direct access)
Memory Usage	Higher (library + streams)	Lower (minimal overhead)
Code Readability	High (familiar functions)	Lower (hardware details)
Development Speed	Fast (standard API)	Slower (manual work)
Hardware Control	Limited (abstraction)	Full (direct access)
Portability	High (standard interface)	Low (hardware-specific)
Real-time Capability	Poor (blocking)	Good (non-blocking possible)
Learning Curve	Low (standard C)	High (hardware-specific)
Debugging	Easier (abstraction)	Harder (hardware details)
Security	Mixed (abstraction)	Better (direct control)

Recommendations for Security Systems:
Use STDIO Approach When:

- Developing rapid prototypes or proof-of-concept
- System not time-critical (no strict timing requirements)
- Team familiar with standard C I/O functions
- Need quick development and testing

- Display primarily text-based, simple output
- Sufficient memory available (not resource-constrained)

Use Direct Control Approach When:

- System is resource-constrained (limited flash/RAM)
- Real-time performance required (strict timing)
- Need access to advanced hardware features
- Security-critical system (predictable behavior essential)
- Need non-blocking I/O for background tasks
- System must be highly optimized

Hybrid Approach (Recommended for Security Systems):

- Use direct control for time-critical operations (keypad scanning, LED timing)
- Use STDIO for user-facing text output (LCD messages, status displays)
- Implement custom optimized functions for performance-critical paths
- Leverage STDIO for debugging and logging (serial output)
- Balance productivity and performance based on requirements

Example Security System Architecture:

- **Keypad Input:** Direct control (non-blocking, real-time scanning)
- **LCD Display:** STDIO (formatted messages, readable code)
- **LED Control:** Direct control (precise timing, critical feedback)
- **Debugging:** STDIO (serial output, formatted logging)
- **Password Verification:** Direct control (security-critical, no overhead)

This hybrid approach provides the benefits of both methods: development productivity and readability for user interface, performance and control for security-critical operations.

5.9. Code Portability Techniques

Question: What techniques can be used to ensure code portability for LCD and keypad interfaces between different hardware platforms? How can hardware interaction be abstracted?

Answer:

Portability Challenges:

Embedded systems vary widely in hardware platforms (Arduino, ESP32, STM32, Raspberry Pi Pico, etc.), each with different:

- GPIO pin layouts and numbering
- I2C/SPI/UART implementations
- Timing mechanisms
- Available libraries and frameworks
- Development environments and toolchains

Abstraction Techniques:

Technique 1: Hardware Abstraction Layer (HAL)

Concept: Create a unified interface that hides platform-specific details.

Implementation:

HAL Interface (hal.h):

```
class HardwareAbstraction {
public:
    // GPIO abstraction
    virtual void pinMode(uint8_t pin, uint8_t mode) = 0;
    virtual void digitalWrite(uint8_t pin, uint8_t value) = 0;
    virtual int digitalRead(uint8_t pin) = 0;

    // I2C abstraction
    virtual void i2cBegin() = 0;
    virtual void i2cBeginTransmission(uint8_t address) = 0;
    virtual void i2cWrite(uint8_t data) = 0;
    virtual uint8_t i2cEndTransmission() = 0;

    // Timing abstraction
    virtual void delay(unsigned long ms) = 0;
    virtual unsigned long millis() = 0;
};
```

Platform-Specific Implementations:

Arduino Implementation (arduino_hal.cpp):

```
class ArduinoHAL : public HardwareAbstraction {
public:
    void pinMode(uint8_t pin, uint8_t mode) override {
        ::pinMode(pin, mode);
    }

    void digitalWrite(uint8_t pin, uint8_t value) override {
        ::digitalWrite(pin, value);
    }

    int digitalRead(uint8_t pin) override {
        return ::digitalRead(pin);
    }

    void i2cBegin() override {
        Wire.begin();
    }

    void i2cBeginTransmission(uint8_t address) override {
        Wire.beginTransmission(address);
    }

    void i2cWrite(uint8_t data) override {
        Wire.write(data);
    }

    uint8_t i2cEndTransmission() override {
        return Wire.endTransmission();
    }

    void delay(unsigned long ms) override {
        ::delay(ms);
    }

    unsigned long millis() override {
        return ::millis();
    }
};
```

ESP32 Implementation (esp32_hal.cpp):

```
class ESP32HAL : public HardwareAbstraction {
public:
    void pinMode(uint8_t pin, uint8_t mode) override {
        gpio_set_direction((gpio_num_t)pin, (gpio_mode_t)mode);
    }

    void digitalWrite(uint8_t pin, uint8_t value) override {
        gpio_set_level((gpio_num_t)pin, value);
    }

    // ... similar implementations for ESP32
};
```

Advantages:

- Application code unchanged when porting
- Only HAL implementation needs to be rewritten
- Clear separation between hardware and application logic
- Easier to maintain and test

Technique 2: Configuration Files

Concept: Store platform-specific settings in configuration files.

Configuration (config.h):

```
// Platform selection
#define PLATFORM_ARDUINO_UNO   1
#define PLATFORM_ESP32         2
#define PLATFORM_STM32         3

// Select platform
#define CURRENT_PLATFORM PLATFORM_ARDUINO_UNO

#if CURRENT_PLATFORM == PLATFORM_ARDUINO_UNO
    // Arduino Uno pin assignments
    const uint8_t LCD_SDA_PIN = A4;
    const uint8_t LCD_SCL_PIN = A5;
    const uint8_t LED_GREEN_PIN = 12;
    const uint8_t LED_RED_PIN = 13;

    // Timing values
    const unsigned long DEBOUNCE_PRESS_MS = 20;
    const unsigned long DEBOUNCE_RELEASE_MS = 50;

#elif CURRENT_PLATFORM == PLATFORM_ESP32
    // ESP32 pin assignments
    const uint8_t LCD_SDA_PIN = 21;
    const uint8_t LCD_SCL_PIN = 22;
    const uint8_t LED_GREEN_PIN = 2;
    const uint8_t LED_RED_PIN = 4;

    // Timing values (ESP32 faster, can reduce delays)
    const unsigned long DEBOUNCE_PRESS_MS = 10;
    const unsigned long DEBOUNCE_RELEASE_MS = 30;
#endif
```

Advantages:

- Easy to switch platforms by changing `CURRENT_PLATFORM`
- All platform-specific settings in one place
- No code changes required for different hardware

Technique 3: Factory Pattern

Concept: Use factory pattern to create platform-specific objects.

Implementation:

```
class DriverFactory {
public:
    static HardwareAbstraction* createHAL() {
#if CURRENT_PLATFORM == PLATFORM_ARDUINO_UNO
        return new ArduinoHAL();
#elif CURRENT_PLATFORM == PLATFORM_ESP32
        return new ESP32HAL();
#else
        return nullptr;
#endif
```

```
    }

    static LcdDriver* createLcdDriver(uint8_t address) {
        return new LcdI2cDriver(createHAL(), address);
    }

    static KeypadDriver* createKeypadDriver(const uint8_t* rows,
                                            const uint8_t* cols,
                                            uint8_t numRows,
                                            uint8_t numCols) {
        return new MatrixKeypad(createHAL(), rows, cols, numRows, numCols);
    }
};
```

Usage:

```
// Platform-independent code
auto hal = DriverFactory::createHAL();
auto lcd = DriverFactory::createLcdDriver(0x27);
auto keypad = DriverFactory::createKeypadDriver(rowPins, colPins, 4, 4);

lcd->begin();
keypad->begin();
```

Advantages:

- Clean object creation
- Encapsulates platform-specific logic
- Easy to add new platforms

Technique 4: Conditional Compilation

Concept: Use preprocessor directives for platform-specific code.

Implementation:

```
// Cross-platform delay function
void platformDelay(unsigned long ms) {
#if CURRENT_PLATFORM == PLATFORM_ARDUINO_UNO
    delay(ms);
#elif CURRENT_PLATFORM == PLATFORM_ESP32
    vTaskDelay(ms / portTICK_PERIOD_MS);
#elif CURRENT_PLATFORM == PLATFORM_STM32
```

```
    HAL_Delay(ms);
#endif
}

// Cross-platform I2C initialization
void platformI2CInit() {
#if CURRENT_PLATFORM == PLATFORM_ARDUINO_UNO
    Wire.begin();
#elif CURRENT_PLATFORM == PLATFORM_ESP32
    Wire.begin(LCD_SDA_PIN, LCD_SCL_PIN);
#elif CURRENT_PLATFORM == PLATFORM_STM32
    I2C_InitTypeDef I2C_InitStruct;
    // STM32 I2C initialization code
#endif
}
```

Advantages:

- Zero runtime overhead (compilation-time selection)
- Optimized code for each platform
- No memory overhead for unused platform code

Technique 5: Interface-Based Design

Concept: Define interfaces for all hardware components.

LCD Interface (lcd_interface.h):

```
class ILcdDriver {
public:
    virtual ~ILcdDriver() = default;
    virtual void begin() = 0;
    virtual void clear() = 0;
    virtual void setCursor(uint8_t col, uint8_t row) = 0;
    virtual void print(const char* text) = 0;
    virtual void println(const char* text) = 0;
    virtual void write(uint8_t value) = 0;
};
```

Keypad Interface (keypad_interface.h):

```
class IKeypadDriver {
public:
    virtual ~IKeypadDriver() = default;
    virtual void begin() = 0;
    virtual char getKey() = 0;
    virtual bool keyPressed() = 0;
};
```

Usage:

```
// Application uses interfaces, not concrete implementations
class AccessControlSystem {
private:
    ILcdDriver* _lcd;
    IKeypadDriver* _keypad;

public:
    AccessControlSystem(ILcdDriver* lcd, IKeypadDriver* keypad)
        : _lcd(lcd), _keypad(keypad) {}

    void run() {
        char key = _keypad->getKey();
        if (key != '\0') {
            _lcd->print("Key: ");
            _lcd->write(key);
        }
    }
};
```

Advantages:

- Clear contract between components
- Easy to mock for testing
- Promotes loose coupling
- Facilitates dependency injection

Technique 6: Build System Configuration

Concept: Use build system to select platform and dependencies.

PlatformIO (platformio.ini):

```
[env:arduino_uno]
platform = atmelavr
```

```
board = uno
framework = arduino
build_flags = -DPLATFORM_ARDUINO_UNO

[env:esp32]
platform = espressif32
board = esp32dev
framework = arduino
build_flags = -DPLATFORM_ESP32

[env:stm32]
platform = ststm32
board = nucleo_f103rb
framework = arduino
build_flags = -DPLATFORM_STM32
```

CMake (CMakeLists.txt):

```
if(ARDUINO_UNO)
    add_definitions(-DPLATFORM_ARDUINO_UNO)
    target_sources(myapp PRIVATE arduino_hal.cpp)
elseif(ESP32)
    add_definitions(-DPLATFORM_ESP32)
    target_sources(myapp PRIVATE esp32_hal.cpp)
endif()
```

Advantages:

- Automated platform selection
- Platform-specific optimizations
- Easy to build for multiple targets

Portability Checklist:

- **Hardware Abstraction:** HAL or interface-based design
- **Configuration:** Platform-specific settings in config files
- **Build System:** Multi-platform build configuration
- **Dependencies:** Platform-agnostic library selection
- **Timing:** Abstracted timing functions

- **Memory:** Avoid platform-specific memory sizes
- **Data Types:** Use standard types (uint8_t, uint16_t)
- **Compiler:** Avoid compiler-specific extensions
- **Testing:** Test on all target platforms
- **Documentation:** Document platform-specific requirements

Example: Porting from Arduino to ESP32
Step 1: Update Configuration

- Change `CURRENT_PLATFORM` to `PLATFORM_ESP32`
- Update pin assignments (A4/A5 → 21/22)
- Adjust timing values if needed

Step 2: Implement ESP32 HAL

- Create `ESP32HAL` class
- Implement all HAL interface methods using ESP32 APIs

Step 3: Update Build Configuration

- Select ESP32 environment in PlatformIO
- Build and test

Step 4: Validate

- Test all functionality on ESP32
- Verify timing and performance
- Check memory usage

Result: Application code unchanged, only HAL and configuration modified.

5.10. Testing Methodology for Access Control

Question: Describe the methodology for testing an access control system based on LCD and keypad. What test scenarios are important for validating security and functionality?

Answer:
Testing Methodology:

Comprehensive testing of access control systems requires systematic validation of security, functionality, usability, and reliability. A structured testing approach ensures all requirements are met and potential vulnerabilities are identified.

Testing Phases:

Phase 1: Unit Testing

Objective: Test individual components in isolation.

Test Cases:

Keypad Driver Unit Tests:

- **Test: Single Key Detection**
  - Input: Press '1' key
  - Expected: Return '1'
  - Validation: Correct key returned
- **Test: Debouncing**
  - Input: Rapid key press/release
  - Expected: Single key detection, no multiple triggers
  - Validation: No false positives from contact bounce
- **Test: All Keys**
  - Input: Press all 16 keys sequentially
  - Expected: Correct mapping for each key
  - Validation: Key matrix matches expected layout
- **Test: Simultaneous Key Press**
  - Input: Press two keys simultaneously
  - Expected: Detect one key, ignore other (or detect both if supported)
  - Validation: No ghosting or undefined behavior

LCD Driver Unit Tests:

- **Test: Display Initialization**
  - Input: Call `begin()`
  - Expected: LCD initialized, backlight on, display cleared
  - Validation: LCD responsive, cursor at (0,0)
- **Test: Text Display**

  - Input: `print("Hello")`
  - Expected: "Hello"displayed at current cursor position
  - Validation: Correct text appears on screen

- **Test: Cursor Positioning**
  - Input: `setCursor(5, 1)`
  - Expected: Cursor at column 5, row 1
  - Validation: Subsequent output at correct position

- **Test: Display Clear**
  - Input: `clear()`
  - Expected: All text removed, cursor at (0,0)
  - Validation: Empty display

LED Driver Unit Tests:

- **Test: LED On**
  - Input: `on()`
  - Expected: LED illuminated, state = true
  - Validation: LED visible, `state()` returns true

- **Test: LED Off**
  - Input: `off()`
  - Expected: LED extinguished, state = false
  - Validation: LED off, `state()` returns false

- **Test: Toggle**
  - Input: `toggle()` when LED is on
  - Expected: LED turns off, state inverts
  - Validation: Correct behavior

Phase 2: Integration Testing
Objective: Test interaction between components.
Test Cases:
Input-Output Integration:

- **Test: Keypad to LCD Echo**

  - Input: Press ’1’, ’2’, ’3’, ’4’
  - Expected: LCD displays "****"
  - Validation: Keypad input correctly displayed

- **Test: Clear Function**
  - Input: Enter partial PIN, press ’*’
  - Expected: Input cleared, display shows "[ ]"
  - Validation: Clear functionality works

FSM Integration:

- **Test: State Transitions**
  - Input: Sequence of key presses
  - Expected: Correct state transitions (Welcome → Input → Verify → Granted)
  - Validation: FSM follows correct path

- **Test: LED Activation**
  - Input: Enter correct PIN
  - Expected: Green LED turns on for 5 seconds
  - Validation: LED timing and color correct

Phase 3: Functional Testing
Objective: Validate system functionality against requirements.
Test Scenarios:
Scenario 1: Successful Authentication

1. System displays "Enter Code: [ ]"
2. User enters correct PIN (e.g., "1234")
3. User presses ’#’
4. Expected: "ACCESS GRANTED! Door Unlocked green LED on for 5 seconds
5. Validation: Success path works correctly

Scenario 2: Failed Authentication

1. System displays "Enter Code: [ ]"
2. User enters incorrect PIN (e.g., "9999")

3. User presses '#'
4. Expected: "ACCESS DENIED! Wrong Code red LED blinks 5 times
5. Validation: Failure path works correctly

Scenario 3: Password Change

1. User presses 'D' at welcome screen
2. System enters programming mode, programming LED on
3. User enters new PIN (e.g., "5678")
4. User presses '#'
5. Expected: "PASSWORD SAVED | New: 5678 green LED blinks twice
6. Validation: Password change functionality works

Scenario 4: Input Clearing

1. User enters partial PIN (e.g., "12")
2. User presses '*'
3. Expected: Input cleared, display returns to "[ ]"
4. Validation: Clear functionality works at any point

Scenario 5: Programming Mode Cancellation

1. User enters programming mode with 'D'
2. User presses '*'
3. Expected: Exit programming mode, original password unchanged
4. Validation: Cancellation works correctly

Phase 4: Security Testing
Objective: Validate security measures and identify vulnerabilities.
Test Scenarios:
Security Test 1: Incorrect PIN Attempts

- **Test:** Enter incorrect PIN 10 times
- **Expected:** System continues to accept attempts (no lockout)
- **Vulnerability:** No brute-force protection

- **Recommendation:** Implement attempt limit and lockout

Security Test 2: Timing Analysis

- **Test:** Measure time to verify correct vs. incorrect PIN
- **Expected:** Different timing might reveal password information
- **Vulnerability:** Timing attacks possible
- **Recommendation:** Constant-time password comparison

Security Test 3: Password Persistence

- **Test:** Change password, power cycle, attempt authentication
- **Expected:** New password lost after power cycle
- **Vulnerability:** No non-volatile storage
- **Recommendation:** Store password in EEPROM

Security Test 4: Programming Mode Security

- **Test:** Enter programming mode without authentication
- **Expected:** System allows password change
- **Vulnerability:** No authentication for password change
- **Recommendation:** Require current password to change password

Security Test 5: Buffer Overflow

- **Test:** Enter more than 4 digits before pressing '#'
- **Expected:** System ignores extra digits
- **Vulnerability:** May cause buffer overflow if not properly bounded
- **Recommendation:** Ensure strict 4-digit limit

Phase 5: Performance Testing
Objective: Validate system meets performance requirements.
Test Cases:
Performance Test 1: Response Time

- **Test:** Measure time from key press to LCD update
- **Requirement:** < 100ms

- **Expected:** 35-60ms
- **Validation:** Meets requirement

Performance Test 2: Memory Usage

- **Test:** Measure flash and RAM usage
- **Expected:** Flash < 32KB, RAM < 2KB
- **Validation:** Fits within Arduino Uno constraints

Performance Test 3: Power Consumption

- **Test:** Measure current draw in different states
- **Expected:** < 100mA (idle), < 150mA (active)
- **Validation:** Within power supply capabilities

Phase 6: Usability Testing
Objective: Evaluate user experience and interface design.
Test Scenarios:
Usability Test 1: First-Time User

- **Test:** Observe new user attempting to enter PIN
- **Expected:** User understands interface without instructions
- **Validation:** Interface is intuitive

Usability Test 2: Error Recovery

- **Test:** User makes mistake (wrong PIN), then corrects
- **Expected:** Clear error message, easy recovery
- **Validation:** Error handling is user-friendly

Usability Test 3: Programming Mode

- **Test:** User changes password
- **Expected:** Clear instructions, confirmation of change
- **Validation:** Password change process is straightforward

Phase 7: Reliability Testing
Objective: Validate system reliability over extended operation.
Test Cases:
Reliability Test 1: Extended Operation

- **Test:** Run system for 24 hours, perform 100 authentications
- **Expected:** No crashes, no memory leaks, consistent behavior
- **Validation:** System stable over time

Reliability Test 2: Key Repeated Press

- **Test:** Press same key 100 times rapidly
- **Expected:** Each press detected correctly, no missed or duplicate keys
- **Validation:** Keypad scanning robust

Reliability Test 3: Power Cycle

- **Test:** Power cycle system 50 times
- **Expected:** System boots correctly each time
- **Validation:** Initialization reliable

Phase 8: Environmental Testing
Objective: Validate system operates in expected environmental conditions.
Test Cases:
Environmental Test 1: Temperature

- **Test:** Operate at 0°C and 40°C
- **Expected:** System functions correctly
- **Validation:** Temperature range acceptable

Environmental Test 2: Humidity

- **Test:** Operate at 20% and 80% humidity
- **Expected:** No false key triggers, LCD readable
- **Validation:** Humidity tolerance adequate

Environmental Test 3: Voltage Variation

- **Test:** Operate at 4.5V and 5.5V
- **Expected:** System functions correctly
- **Validation:** Voltage tolerance acceptable

Test Documentation:

For each test, document:

- Test ID and name
- Test objective
- Test procedure
- Expected results
- Actual results
- Pass/fail status
- Date and tester
- Notes and observations

Test Automation:

Automate repetitive tests using:

- Test scripts for serial communication simulation
- Hardware-in-the-loop (HIL) testing
- Automated test harness for unit tests
- Continuous integration for regression testing

Critical Test Scenarios Summary:

Таблица 3: Critical Test Scenarios

Test Scenario	**Importance**
Correct PIN authentication	Critical - Primary function
Incorrect PIN rejection	Critical - Security requirement
Password change functionality	High - Management feature
Response time ($<$ 100ms)	Critical - Usability requirement
Debouncing reliability	Critical - Input accuracy
LCD display accuracy	High - User feedback
LED indication correctness	High - Visual feedback
Input buffer overflow protection	Critical - Security
Programming mode security	Critical - Unauthorized access prevention
Extended operation stability	High - Reliability

Testing Tools and Equipment:

- **Oscilloscope:** Measure timing, signal integrity
- **Multimeter:** Measure voltage, current, resistance
- **Logic Analyzer:** Analyze I2C, GPIO signals
- **Serial Monitor:** Debug output, logging
- **Automated Test Framework:** Unit test execution
- **Environmental Chamber:** Temperature/humidity testing

This comprehensive testing methodology ensures the access control system is secure, functional, reliable, and user-friendly before deployment.

6. Note on AI Tools Usage

During the preparation of this report, the author utilized ChatGPT (an AI language model developed by OpenAI) for generating and consolidating content. The AI assistance was used for:

- Generating and structuring technical descriptions of hardware components and technologies.
- Formulating explanations of system architecture and design decisions.
- Drafting sections on domain analysis and case studies.
- Suggesting improvements and limitations based on the implemented solution.
- Assisting with formatting and organizing the report structure.

All information generated by the AI tool was reviewed, validated, and adjusted by the author to ensure accuracy, relevance, and compliance with the laboratory work requirements. The author takes full responsibility for the content presented in this report.

8. Bibliography

1. Arduino.cc. *Arduino Language Reference.* Available: `https://www.arduino.cc/reference/en/` [Accessed: 2026-02-16].

2. Atmel Corporation. *ATmega328P Datasheet - Complete*. 2014. Available: `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-A Datasheet.pdf` [Accessed: 2026-02-16].
3. PlatformIO. *PlatformIO Documentation*. Available: `https://docs.platformio.org/` [Accessed: 2026-02-16].
4. LiquidCrystal_I2C Library. *GitHub Repository*. Available: `https://github.com/johnrickman/LiquidCrystal_I2C` [Accessed: 2026-02-16].
5. NXP Semiconductors. *I2C-Bus Specification and User Manual*. 2021. Available: `https://www.nxp.com/docs/en/user-guide/UM10204.pdf` [Accessed: 2026-02-16].
6. Axelson, J. *Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems*. Lakeview Research, 2007.
7. Arduino.cc. *Wire Library (I2C)*. Available: `https://www.arduino.cc/reference/en/language/functions/communication/wire/` [Accessed: 2026-02-16].
8. Arduino.cc. *LiquidCrystal Library*. Available: `https://www.arduino.cc/reference/en/libraries/liquidcrystal/` [Accessed: 2026-02-16].
9. Horowitz, P., and Hill, W. *The Art of Electronics*. 3rd Edition, Cambridge University Press, 2015.
10. Wikipedia. *Finite-state machine*. Available: `https://en.wikipedia.org/wiki/Finite-state_machine` [Accessed: 2026-02-16].
11. Wikipedia. *Debouncing*. Available: `https://en.wikipedia.org/wiki/Switch#Contact_bounce` [Accessed: 2026-02-16].
12. Wikipedia. *Matrix keypad*. Available: `https://en.wikipedia.org/wiki/Keypad#Matrix_keypads` [Accessed: 2026-02-16].
13. AVR Libc. *Standard IO Facilities*. Available: `https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html` [Accessed: 2026-02-16].
14. NXP Semiconductors. *PCF8574 8-bit I/O expander for I2C-bus*. 2013. Available: `https://www.nxp.com/docs/en/datasheet/PCF8574.pdf` [Accessed: 2026-02-16].

7. Appendix - Source Code

The complete source code for the Electronic Lock System (Lab 1.2) is available on GitHub:

`https://github.com/DimonBel/ES`

Repository Structure:

- `src/main.cpp` - Main application entry point and initialization
- `src/modules/app/` - Application controller (FSM implementation)
- `src/modules/keypad/` - Matrix keypad driver
- `src/modules/lcd/` - LCD display driver (I2C)
- `src/modules/led/` - LED driver implementation
- `src/modules/serial_stdio/` - STDIO serial interface
- `src/modules/stdio_redirect/` - STDIO redirection to LCD/Keypad

The repository includes complete implementation of:

- 4×4 matrix keypad scanning with debouncing
- I2C LCD 16×2 display control
- Finite state machine for access control
- Password verification and programming mode
- LED status indicators (Green/Red/Programming)
- STDIO redirection for debugging