

**Ministerul Educației și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică**

Report

*Laboratory work nr. 6.1  
Control comportamental cu Automate Finite:****Button-Led***

Prepared by:

**Berco Andrei,** FAF-221

Checked by:

**Martîniuc Alexei,** *university assistant*

**Chișinău – 2025**

# **1. OBJECTIVES**

# **Main purpose of the work:** To design and implement a modular microcontroller-based application that controls an LED using a finite state machine (FSM). The system transitions through OFF, ON, and BLINKING states based on user input from a push-button. The control flow and system state are observable in real time via serial output using STDIO.

# **The objectives of the work:**

# To understand the use of Finite State Machines (FSM) in embedded applications.

# To develop modular code by separating hardware abstraction (LED, Button) from control logic (FSM).

# To apply debouncing techniques for reliable push-button input detection.

# To implement serial communication using STDIO for real-time feedback and debugging.

# To demonstrate a simple state-driven behavior using a user interface (button) and output interface (LED).

# To promote reusability and clarity through structured module design.

# To lay the foundation for more advanced control systems, such as ON-OFF hysteresis logic.

# **PROBLEM DEFINITION**

**System Description:**  
The project implements a basic Finite State Machine (FSM) to control an LED based on user interaction through a button. The system cycles through the following states:

* **OFF**: The LED is off.
* **ON**: The LED is turned on.
* **BLINKING**: The LED toggles on and off at a fixed interval.

Each press of the button transitions the system to the next state. The system ensures accurate button press detection using a software debounce mechanism. State changes and current state information are displayed via the serial interface using printf, enabled through a custom STDIO redirection.

**Hardware Involved:**

* Microcontroller: Arduino Uno
* LED: Connected to digital pin 13
* Button: Connected to digital pin 2, using internal pull-up
* Serial Communication: Via USB, 9600 baud

**Key Features:**

* Debounced button input
* FSM with three distinct states
* LED control logic encapsulated in a class
* Button input encapsulated in a class
* STDIO redirection for using printf with the serial monitor

**Use Case:**  
The application serves as a demonstrative tool for embedded systems learners to understand state machines, modular design, and serial communication. It provides a hands-on example of how a microcontroller can react to user input and change behavior in a predictable, testable manner.

**2. DOMAIN ANALYSIS**

**3.1 Application Domain:**  
This project falls under the domain of **Embedded Systems** and **IoT (Internet of Things)**. It specifically addresses user-interactive systems that rely on real-time input and output, such as smart appliances, sensor systems, or control interfaces.

**3.2 Related Concepts:**

* **Finite State Machine (FSM):** A computation model used to design systems with a limited number of specific states and predictable transitions. Common in embedded applications for managing states like OFF, ON, and BLINK.
* **Modular Programming:** The practice of dividing code into independent modules or classes (e.g., LED, Button) that improve reusability and readability.
* **Button Debouncing:** A technique to eliminate false triggers due to mechanical bouncing in push-buttons.
* **Serial Communication & STDIO Redirection:** Redirection of printf to serial output, allowing debugging and feedback without Serial.print().

**3.3 Target Users:**

* Embedded systems students and developers.
* Engineers developing control systems or real-time embedded applications.
* Educators demonstrating FSM-based microcontroller behavior.

**3.4 Constraints:**

* The Arduino Uno has limited memory and processing power.
* The system assumes only one input device (button) and one output device (LED).
* Serial communication is limited to 9600 baud in this setup.

**Architectural Design**

The architectural design emphasizes modularity, reusability, and real-time interaction. Each component of the system is encapsulated in its own module:

* **MCU Core**: Manages overall logic, reads sensor input, processes control algorithms, and updates outputs.
* **Sensor Module**: Reads environmental or positional data (e.g., DHT22 or potentiometer), returning processed values.
* **Control Module**: Implements ON-OFF logic with hysteresis, calculates thresholds, and determines output state.
* **Actuator Module**: Controls relay or motor via L298 driver based on control output.
* **Interface Module**: Displays current and set values on LCD and allows interaction via serial (STDIO) or physical controls (buttons/keypad).

**Hardware Diagram**

A diagram of a button

AI-generated content may be incorrect.

**A diagram of a computer

AI-generated content may be incorrect.**

**Layer Diagram**

**A diagram of a computer

AI-generated content may be incorrect.**

# **Electrical Scheme**

**A circuit board with red text

AI-generated content may be incorrect.**

**A circuit board with wires connected to it

AI-generated content may be incorrect.**

**Project Structure and Modular Implementation**

**Table of states**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| State | Description | Transition Condition | Action Performed | Next State |
| STATE\_OFF | LED is off | Button pressed | LED turned off, state printed | STATE\_ON |
| STATE\_ON | LED is steadily on | Button pressed | LED turned on, state printed | STATE\_BLINKING |
| STATE\_BLINKING | LED blinks at 500ms interval | Button pressed | LED turned off, state printed | STATE\_OFF |

A screenshot of a computer

AI-generated content may be incorrect.

**Block Diagram of System Behaviour**

**A diagram of a computer

AI-generated content may be incorrect.**

The general flowchart outlines the program’s execution flow. In setup(), the FSM, LED, Button, and serial communication are initialized. The loop() continuously updates the FSM, checking for button presses. If a button is pressed, the FSM transitions to the next state (OFF → ON → BLINKING → OFF). Based on the current state, the LED is updated (on, off, or blinking). The current state is printed to the serial monitor, and the loop repeats.

**Functional Block Diagrams**

**A diagram of a button press

AI-generated content may be incorrect.**

**Description:**

The Finite State Machine (FSM) diagram represents the state transitions of the system. The FSM starts in the OFF state. A button press transitions it to the ON state, where the LED is steadily on. Another button press moves it to the BLINKING state, where the LED blinks at a 500ms interval. A final button press returns it to the OFF state, turning the LED off. This cycle repeats with each

**Code flowchart:**

**Main.cpp**

A diagram of a computer

AI-generated content may be incorrect.

# **Description:**

The main.cpp flowchart is simple. In setup(), serial communication, the FSM, LED, and Button are initialized, and initialization messages are printed. The loop() continuously calls fsm.update() to run the FSM, driving the program’s core functionality.

**button.cpp**

A screenshot of a diagram

AI-generated content may be incorrect.

**Description:**

The isPressed() method in button.cpp implements debounced button reading. It reads the button pin and checks if the state has changed. If so, it records the debounce time. After ensuring the debounce delay (50ms) has passed, it updates the button state. If the new state is LOW (button pressed with INPUT\_PULLUP), it returns true; otherwise, it returns false.

**led.cpp**

A diagram of a flowchart

AI-generated content may be incorrect.

**Description:**

The led.cpp flowchart covers the LED class methods. init() sets the pin as OUTPUT and turns the LED off. turnOn() and turnOff() set the pin HIGH or LOW, updating the state. toggle() inverts the state and updates the pin. blink() checks if the specified interval has passed since the last blink; if so, it toggles the LED state and updates the pin and last blink time.

**fsm.cpp**

A diagram of a flowchart

AI-generated content may be incorrect.

# The update() method in fsm.cpp drives the FSM. If a button press is detected, it transitions the state: OFF → ON (turns LED on), ON → BLINKING, or BLINKING → OFF (turns LED off). The new state is printed, and the state change time is updated. If no button is pressed and the state is BLINKING, the LED blinks at the specified interval. Otherwise, no LED action is taken.

**stdio\_redirect.cpp**

A diagram of a diagram

AI-generated content may be incorrect.

The stdio\_redirect.cpp flowchart covers serial I/O functions. stdio\_init() starts serial communication and sets up stdout/stdin streams. uart\_putchar() writes a character to serial, adding a carriage return before a newline. uart\_getchar() waits for and reads a serial character. stdio\_readstring() buffers incoming characters until a newline or carriage return, then returns the string if the buffer isn’t empty, or skips empty lines.

# **Results** A circuit board with wires connected to it AI-generated content may be incorrect.

# A circuit board with wires connected to it AI-generated content may be incorrect.

# **Conclusion**

In this project, we successfully designed and implemented an Arduino-based Finite State Machine (FSM) system to control an LED using a button with serial communication for state monitoring. The system integrates several core components: a button for user input, an LED for output, an FSM to manage state transitions, and a serial communication interface for real-time feedback. Each module was carefully developed and tested to ensure reliable performance in controlling the LED through distinct states (OFF, ON, BLINKING).

**Key Highlights:**

1. **Button Integration and Debouncing**: The Button class interfaces with a push button on digital pin 2, utilizing the Arduino’s internal pull-up resistor to simplify the circuit. The debouncing mechanism, implemented with a 50ms delay, ensures accurate detection of button presses by filtering out noise, allowing the FSM to transition states reliably.
2. **LED Control Logic**: The LED class manages the LED on digital pin 13, providing methods to turn it on, off, toggle, and blink at a 500ms interval. This functionality supports the FSM’s states: steady ON, OFF, and BLINKING. The implementation ensures smooth operation by updating the LED state based on the current FSM state and timing requirements.
3. **Finite State Machine Implementation**: The FiniteStateMachine class orchestrates the system’s behavior, transitioning between OFF, ON, and BLINKING states with each button press. The FSM logic ensures predictable state changes, with the LED reflecting the current state (off, on, or blinking), providing a clear visual indication of the system’s operation.
4. **Serial Interface for Monitoring**: The stdio\_redirect module enables serial communication at 9600 baud, allowing real-time state updates to be printed to the serial monitor. This feature provides visibility into the system’s operation, displaying messages like “Current State: OFF” after each state transition, enhancing user interaction and debugging capabilities.
5. **Modular Design**: The project’s modular structure, with separate files for each component (led.cpp, button.cpp, fsm.cpp, stdio\_redirect.cpp), promotes maintainability and scalability. Each module handles a specific task, reducing complexity and making it easier to modify or extend the system in the future. This design also supports code reusability for other Arduino projects.
6. **User Feedback**: The serial output provides clear feedback on the system’s state, making it easy for users to understand the current operation (e.g., whether the LED is off, on, or blinking). This transparency is crucial for verifying the system’s behavior and troubleshooting any issues.

**Challenges:** Several challenges were encountered during the project:

* **Debouncing Accuracy**: Ensuring reliable button debouncing required fine-tuning the debounce delay (50ms). Initial tests showed occasional false positives, which were resolved by adjusting the timing and verifying the button state transitions.
* **Serial Output Synchronization**: Aligning serial output with state changes required careful timing to avoid delays in the main loop. Proper initialization and buffer management in stdio\_redirect.cpp mitigated this issue.

**Future Improvements:**

* **Enhanced User Interaction**: Adding a more interactive serial interface to allow users to set the blink interval dynamically could improve flexibility.
* **Additional States**: Introducing more states (e.g., a FAST\_BLINKING state with a shorter interval) could expand the system’s functionality.
* **External Display**: Integrating an LCD or OLED display to show the current state visually, alongside serial output, would enhance user experience and make the system more standalone.

This project demonstrates a robust implementation of a state-driven LED control system using Arduino, with a focus on modularity, reliability, and user feedback. The lessons learned provide a strong foundation for future enhancements and more complex Arduino-based projects.

During the preparation of this report, the author used Grok for generating/strengthening the content. The resulting information was reviewed, validated, and adjusted according to the requirements of the laboratory work.

# **References**

1. IoT-24 : Lab 6.1. - <https://www.youtube.com/watch?v=ODJjs82LrtM>
2. Arduino Documentation about Seria - <https://docs.arduino.cc/language-reference/en/functions/communication/serial/>
3. PlatformIO Documentation - <https://docs.platformio.org/en/latest/integration/ide/pioide.html>

# **Appendix - Source Code**

File name: button.cpp

text

Copy

#include "button.h"

#include <Arduino.h>

// Constructor

Button::Button(int buttonPin)

: pin(buttonPin),

lastButtonState(HIGH),

buttonState(HIGH),

lastDebounceTime(0),

debounceDelay(50) // 50ms debounce time

{}

// Initialize the button

void Button::init() {

// Use internal pull-up resistor

pinMode(pin, INPUT\_PULLUP);

}

// Check if button is pressed (debounced)

bool Button::isPressed() {

int reading = digitalRead(pin);

// Check for button state change

if (reading != lastButtonState) {

lastDebounceTime = millis();

}

// Check if debounce delay has passed

if ((millis() - lastDebounceTime) > debounceDelay) {

// If button state has changed

if (reading != buttonState) {

buttonState = reading;

// Return true only on button press (LOW for INPUT\_PULLUP)

if (buttonState == LOW) {

lastButtonState = reading;

return true;

}

}

}

lastButtonState = reading;

return false;

}

// Get current button pin

int Button::getPin() const {

return pin;

}

File name: led.cpp

text

Copy

#include "led.h"

#include <Arduino.h>

// Constructor

LED::LED(int ledPin) : pin(ledPin), state(false) {}

// Initialize the LED

void LED::init() {

pinMode(pin, OUTPUT);

digitalWrite(pin, LOW);

}

// Turn LED on

void LED::turnOn() {

digitalWrite(pin, HIGH);

state = true;

}

// Turn LED off

void LED::turnOff() {

digitalWrite(pin, LOW);

state = false;

}

// Toggle LED state

void LED::toggle() {

state = !state;

digitalWrite(pin, state);

}

// Blink LED with specified interval

void LED::blink(unsigned long interval) {

static unsigned long lastBlinkTime = 0;

unsigned long currentTime = millis();

if (currentTime - lastBlinkTime >= interval) {

toggle();

lastBlinkTime = currentTime;

}

}

// Get current LED state

bool LED::getState() const {

return state;

}

// Get LED pin number

int LED::getPin() const {

return pin;

}

File name: fsm.cpp

text

Copy

#include "fsm.h"

#include "stdio\_redirect.h"

// Constructor

FiniteStateMachine::FiniteStateMachine(LED& ledObj, Button& buttonObj)

: currentState(STATE\_OFF),

lastStateChangeTime(0),

blinkInterval(500), // 500ms blink interval

led(ledObj),

button(buttonObj)

{}

// Initialize the FSM

void FiniteStateMachine::init() {

// Initialize components

led.init();

button.init();

// Initialize stdio

stdio\_init(9600);

}

// Update method to be called in main loop

void FiniteStateMachine::update() {

// Check for button press

if (button.isPressed()) {

// State machine logic

switch (currentState) {

case STATE\_OFF:

currentState = STATE\_ON;

led.turnOn();

break;

case STATE\_ON:

currentState = STATE\_BLINKING;

break;

case STATE\_BLINKING:

currentState = STATE\_OFF;

led.turnOff();

break;

}

// Print state after change

printState();

// Update last state change time

lastStateChangeTime = millis();

}

// Control LED based on current state

if (currentState == STATE\_BLINKING) {

led.blink(blinkInterval);

}

}

// Get current state

FSMState FiniteStateMachine::getState() const {

return currentState;

}

// Print current state to serial

void FiniteStateMachine::printState() const {

switch (currentState) {

case STATE\_OFF:

printf("Current State: OFF\n");

break;

case STATE\_ON:

printf("Current State: ON\n");

break;

case STATE\_BLINKING:

printf("Current State: BLINKING\n");

break;

}

}

File name: stdio\_redirect.cpp

text

Copy

/\*\*

\* stdio\_wrapper.cpp

\* Standard I/O wrapper implementation for Arduino

\*/

#include "stdio\_redirect.h"

// Create a FILE stream structure for stdin and stdout

static FILE uartout = {0};

static FILE uartin = {0};

// Buffer for reading serial input

#define BUFFER\_SIZE 32

static char commandBuffer[BUFFER\_SIZE];

static int bufferIndex = 0;

void stdio\_init(unsigned long baud) {

// Initialize serial

Serial.begin(baud);

// Wait for serial port to connect (needed for native USB port only)

while (!Serial && millis() < 5000);

// Setup stdout stream

fdev\_setup\_stream(&uartout, uart\_putchar, NULL, \_FDEV\_SETUP\_WRITE);

stdout = &uartout;

// Setup stdin stream

fdev\_setup\_stream(&uartin, NULL, uart\_getchar, \_FDEV\_SETUP\_READ);

stdin = &uartin;

}

int uart\_putchar(char c, FILE \*stream) {

// Optionally convert newline to carriage return + newline

if (c == '\n') {

Serial.write('\r');

}

Serial.write(c);

return 0;

}

int uart\_getchar(FILE \*stream) {

// Wait until character is available

while (!Serial.available());

return Serial.read();

}

bool stdio\_available() {

return Serial.available() > 0;

}

bool stdio\_readstring(char\* buffer, size\_t max\_length) {

while (Serial.available()) {

char c = Serial.read();

if (c == '\n' || c == '\r') {

if (bufferIndex > 0) {

commandBuffer[bufferIndex] = '\0'; // Terminate the string

bufferIndex = 0; // Reset for next command

// Copy to output buffer, respecting max\_length

strncpy(buffer, commandBuffer, max\_length - 1);

buffer[max\_length - 1] = '\0'; // Ensure null-termination

// Debug: Print the buffer

printf("DEBUG: stdio\_readstring buffer: '%s'\n", buffer);

return true;

}

// Skip empty lines

} else {

if (bufferIndex < BUFFER\_SIZE - 1) {

commandBuffer[bufferIndex++] = c;

}

// Ignore characters if buffer is full

}

}

return false;

}

File name: main.cpp

text

Copy

#include <Arduino.h>

#include "led.h"

#include "button.h"

#include "fsm.h"

// Pin Definitions

const int BUTTON\_PIN = 2; // Button connected to digital pin 2

const int LED\_PIN = 13; // Built-in LED or external LED on pin 13

// Create hardware components

LED led(LED\_PIN);

Button button(BUTTON\_PIN);

// Custom printf function for serial output

// Create FSM instance

FiniteStateMachine fsm(led, button);

void setup() {

// Initialize FSM

fsm.init();

// Initialize serial communication

Serial.begin(9600);

// Optional: Print initial message

printf("Finite State Machine Initialized\n");

// Initialize LED and button

led.init();

button.init();

// Optional: Print initial state message

printf("Initial State: OFF\n");

}

void loop() {

// Update FSM in each loop iteration

fsm.update();

}