TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Report of laboratory work №6

Theme: Finite Automata

Fulfilled: st.gr. FAF-202 **Popa Eugeniu**

Controlled: univ. lecturer **Moraru Dumitru**

Chișinău 2023

THE TASKS OF THE LABORATORY WORK

Main task: Develop an MCU-based application that will implement management systems for:
a). design an elevator in a staircase with at least 8 floors, using finite automata
b). design a traffic light with three colors, using finite automata

THE PROGRESS OF THE WORK

To solve the first task, it is used a separate class for the elevator, called ElevatorFSM. The purpose of this class is to implement a finite state machine (FSM) for an elevator system.

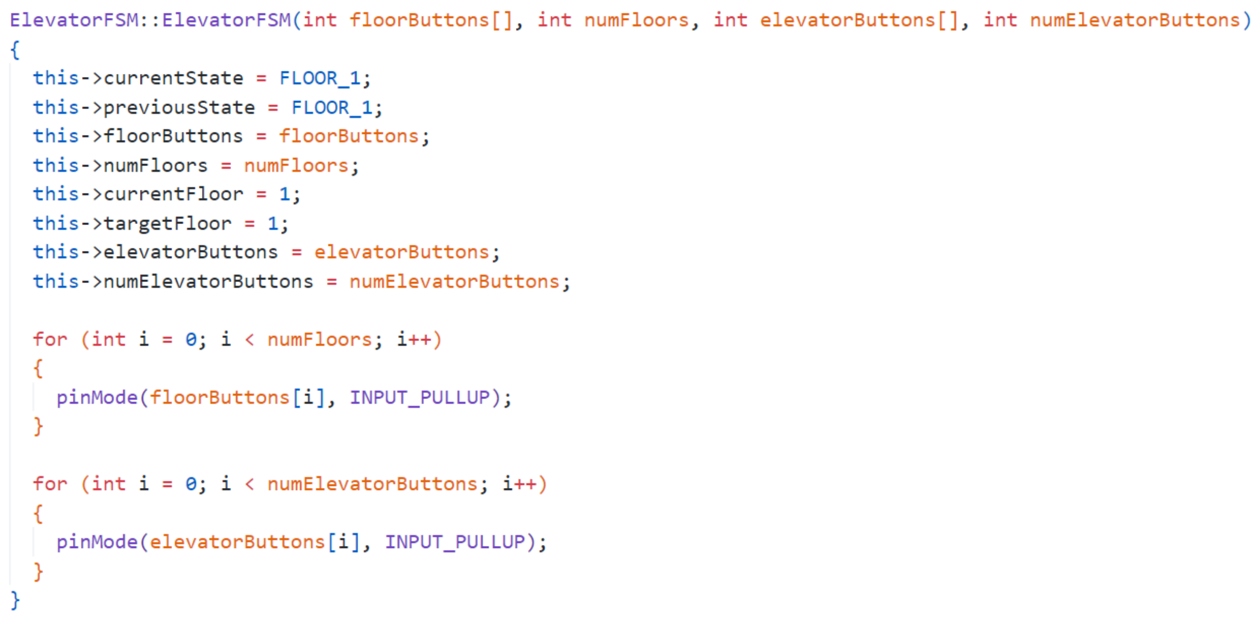
The constructor takes in several parameters: floorButtons[] (an array of integers representing the floor buttons in the elevator), numFloors (the number of floors in the building), elevatorButtons[] (an array of integers representing the buttons inside the elevator) , and numElevatorButtons (the number of buttons inside the elevator).

The constructor initializes the state variables of the elevator FSM. currentState and previousState are initialized to FLOOR_1, indicating that the elevator is initially at the first floor. currentFloor and targetFloor are both set to 1 initially.

The constructor also stores the floor buttons and elevator buttons arrays, as well as the number of floors and elevator buttons, in the corresponding member variables of the class.

In the subsequent code, two for loops are used to set the pin modes of the floor buttons and elevator buttons. Each floor button and elevator button is set as an input with a pull-up resistor enabled. This is a common configuration for reading button inputs in electronic circuits.

Overall, this constructor sets up the initial state and initializes the necessary variables and pin modes for the elevator FSM.



```
ElevatorFSM::ElevatorFSM(int floorButtons[], int numFloors, int elevatorButtons[], int numElevatorButtons)
{
  this->currentState = FLOOR_1;
  this->previousState = FLOOR_1;
  this->floorButtons = floorButtons;
  this->numFloors = numFloors;
  this->currentFloor = 1;
  this->targetFloor = 1;
  this->elevatorButtons = elevatorButtons;
  this->numElevatorButtons = numElevatorButtons;

  for (int i = 0; i < numFloors; i++)
  {
    pinMode(floorButtons[i], INPUT_PULLUP);
  }

  for (int i = 0; i < numElevatorButtons; i++)
  {
    pinMode(elevatorButtons[i], INPUT_PULLUP);
  }
}
```

Fig 1 Constructor which sets up the initial state and does the initialization

The handleState() method of the ElevatorFSM class is responsible for handling the logic and actions based on the current state of the elevator FSM.

Inside the method, a switch statement is used to check the value of the current State variable. The switch cases correspond to the different floors (FLOOR_1, FLOOR_2, FLOOR_3, etc.).

Within each case, two for loops iterate through the floor buttons and elevator buttons. The purpose is to check if any of the buttons have been pressed.

For each floor button or elevator button that is pressed (indicated by a LOW signal from digitalRead()), the following actions are performed:

The LED connected to pin 22 is set to a brightness level of 200 (assuming it's an LED connected via PWM).

The LED connected to pin 23 is turned off (assuming it's an LED connected via PWM).

The target floor is set based on the button that was pressed.

Information about the current floor and target floor is printed to the serial monitor.

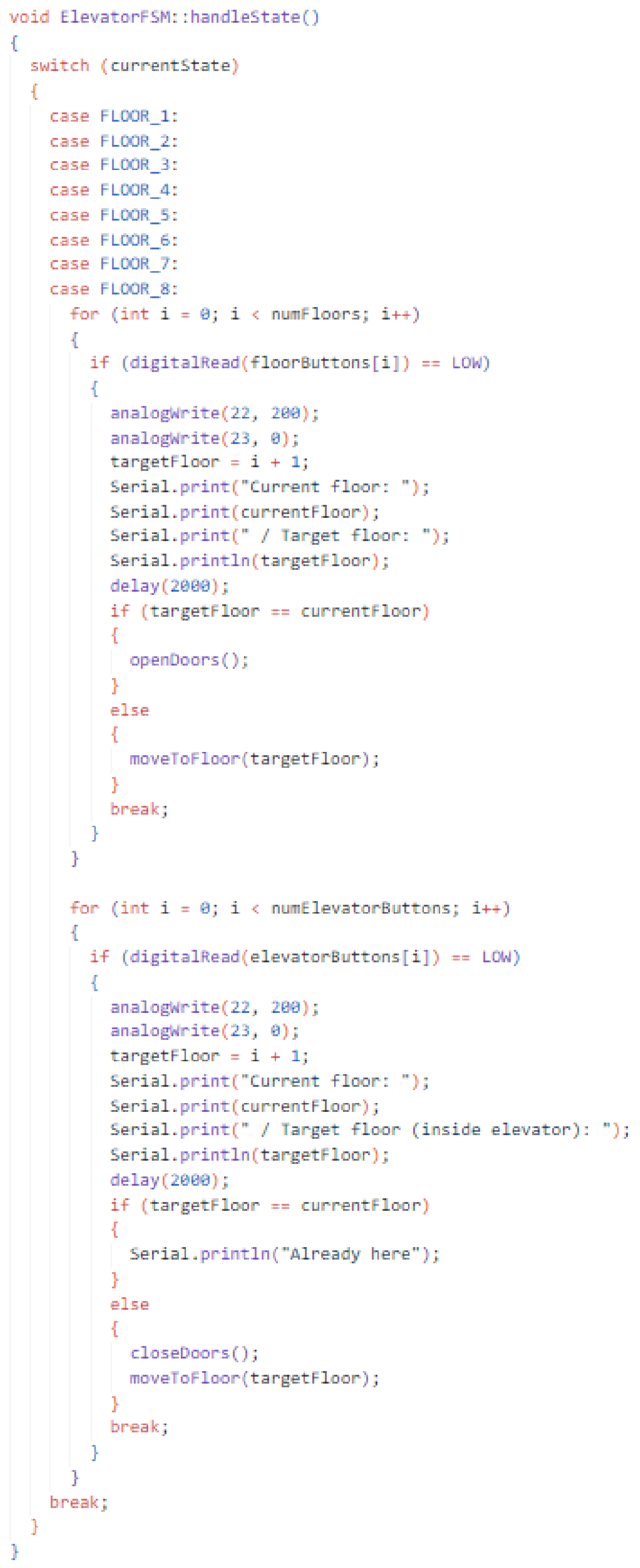
A delay of 2000 milliseconds (2 seconds) is added to provide a pause for visualization purposes.

If the target floor is the same as the current floor, the openDoors() function is called to open the elevator doors.

If the target floor is different from the current floor, the moveToFloor() function is called to move the elevator to the target floor after first closing the doors.

After the for loops and actions are executed, the break statement is used to exit the switch statement.

In summary, the handleState() method handles the button inputs for each floor and inside the elevator. It sets the target floor, updates the LED brightness, prints floor information, delays for visualization, and calls the appropriate functions to open/close doors and move the elevator to the target floor based on the button presses.



```
void ElevatorFSM::handleState()
{
  switch (currentState)
  {
    case FLOOR_1:
    case FLOOR_2:
    case FLOOR_3:
    case FLOOR_4:
    case FLOOR_5:
    case FLOOR_6:
    case FLOOR_7:
    case FLOOR_8:
      for (int i = 0; i < numFloors; i++)
      {
        if (digitalRead(floorButtons[i]) == LOW)
        {
          analogWrite(22, 200);
          analogWrite(23, 0);
          targetFloor = i + 1;
          Serial.print("Current floor: ");
          Serial.print(currentFloor);
          Serial.print(" / Target floor: ");
          Serial.println(targetFloor);
          delay(2000);
          if (targetFloor == currentFloor)
          {
            openDoors();
          }
          else
          {
            moveToFloor(targetFloor);
          }
          break;
        }
      }

      for (int i = 0; i < numElevatorButtons; i++)
      {
        if (digitalRead(elevatorButtons[i]) == LOW)
        {
          analogWrite(22, 200);
          analogWrite(23, 0);
          targetFloor = i + 1;
          Serial.print("Current floor: ");
          Serial.print(currentFloor);
          Serial.print(" / Target floor (inside elevator): ");
          Serial.println(targetFloor);
          delay(2000);
          if (targetFloor == currentFloor)
          {
            Serial.println("Already here");
          }
          else
          {
            closeDoors();
            moveToFloor(targetFloor);
          }
          break;
        }
      }
    break;
  }
}
```

Fig 2 Handling the state of the elevator using handleState()

The moveToFloor() method of the ElevatorFSM class is responsible for moving the elevator to a specified floor.

The method takes an int parameter floor, which represents the target floor to which the elevator should move.

First, the method performs a check to ensure that the floor value is within the valid range of floors (between 1 and numFloors). If the floor is outside this range, the method simply returns, indicating an invalid floor input.

Next, the previousState variable is updated to store the current state before moving.

The method determines the direction in which the elevator needs to move by comparing the floor with the current floor (currentFloor). If the floor is greater than the currentFloor, the direction is set to 1 (upward movement), otherwise, it is set to -1 (downward movement).

A while loop is used to repeatedly update the current floor until it reaches the floor. On each iteration, the current floor is incremented or decremented based on the direction.

Within the loop, the currentFloor is checked again to ensure it remains within the valid floor range. If it falls outside the range, the method returns.

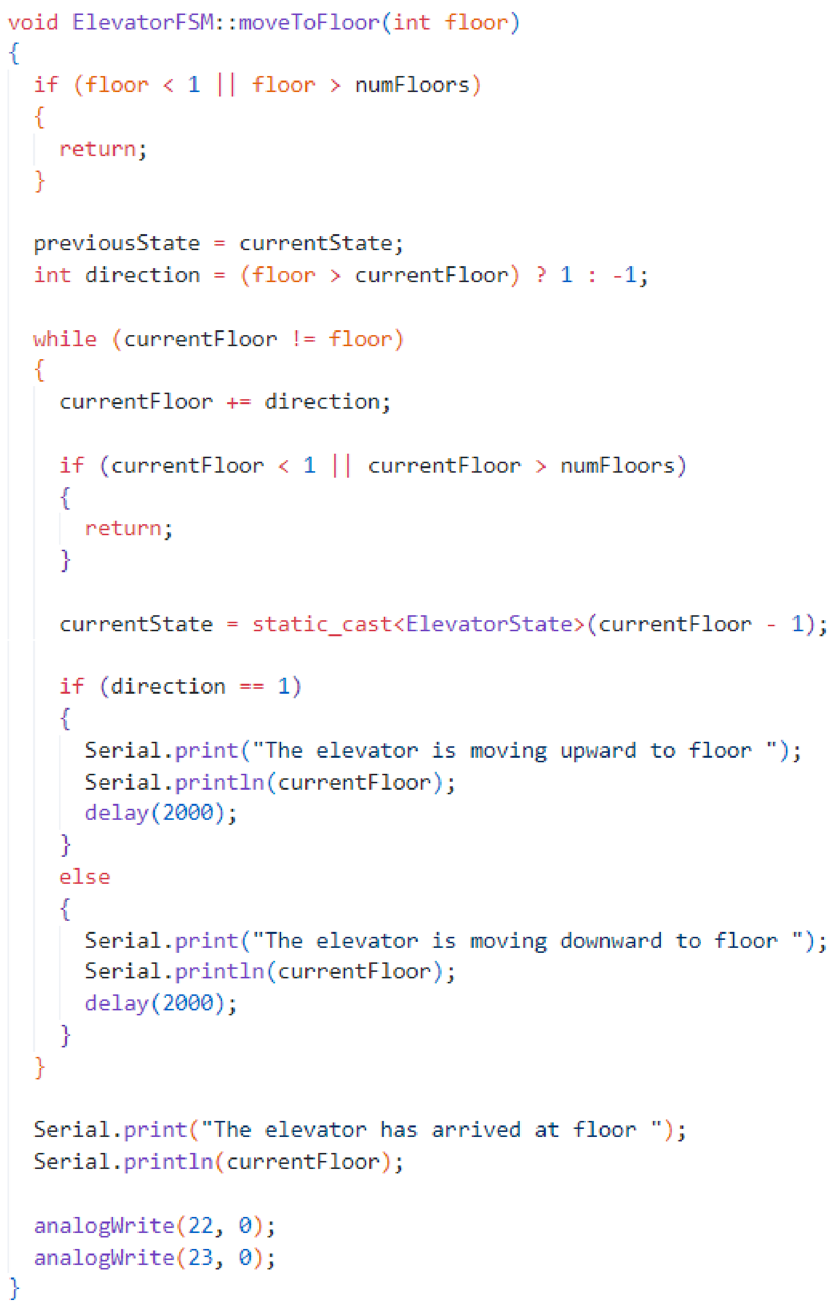
The currentState is updated to match the new currentFloor using a static_cast to convert the integer value to the corresponding ElevatorState enumeration.

Based on the direction, information about the elevator's movement is printed to the serial monitor, indicating whether it is moving upwards or downwards to a particular floor. A delay of 2000 milliseconds (2 seconds) is added for visualization purposes.

Once the while loop exits, it means that the elevator has reached the target floor. A message indicating the arrival at the current floor is printed to the serial monitor.

Finally, the LED connected to pin 22 is turned off by setting the PWM value to 0, indicating that the elevator has stopped moving. The same is done for the LED connected to pin 23.

In summary, the moveToFloor() method moves the elevator from the current floor to a specified target floor. It updates the current floor, prints movement information, and controls the LED indicators.



```
void ElevatorFSM::moveToFloor(int floor)
{
  if (floor < 1 || floor > numFloors)
  {
    return;
  }

  previousState = currentState;
  int direction = (floor > currentFloor) ? 1 : -1;

  while (currentFloor != floor)
  {
    currentFloor += direction;

    if (currentFloor < 1 || currentFloor > numFloors)
    {
      return;
    }

    currentState = static_cast<ElevatorState>(currentFloor - 1);

    if (direction == 1)
    {
      Serial.print("The elevator is moving upward to floor ");
      Serial.println(currentFloor);
      delay(2000);
    }
    else
    {
      Serial.print("The elevator is moving downward to floor ");
      Serial.println(currentFloor);
      delay(2000);
    }
  }

  Serial.print("The elevator has arrived at floor ");
  Serial.println(currentFloor);

  analogWrite(22, 0);
  analogWrite(23, 0);
}
```

Fig 3 Moving to a floor

To solve the second task, it is used a separate class for the semaphore, called TrafficLightFSM. The purpose of this class is to implement a finite state machine (FSM) for an traffic light.

The functions transitionToState() and handleState() represent a simplified implementation of a traffic light finite state machine (FSM). Here's a brief explanation of each function:

void TrafficLightFSM::transitionToState(TrafficLightState nextState)

This function is responsible for transitioning the traffic light FSM to the next state.

It takes a TrafficLightState parameter called nextState, which represents the state to transition to.

Inside the function, the current state variable (currentState) is updated with the nextState value, indicating the transition.

void TrafficLightFSM::handleState()

This function is responsible for handling the behavior of the traffic lights based on the current state.

It uses a switch statement to determine the actions to be taken for each state.

The actions include setting the appropriate digital output pins to control the traffic light LEDs, printing the current state information to the serial monitor, and introducing delays to simulate the timing of traffic light changes.

After performing the actions for the current state, it calls the transitionToState function to transition to the next state based on the defined sequence.

The states mentioned in the code are as follows:

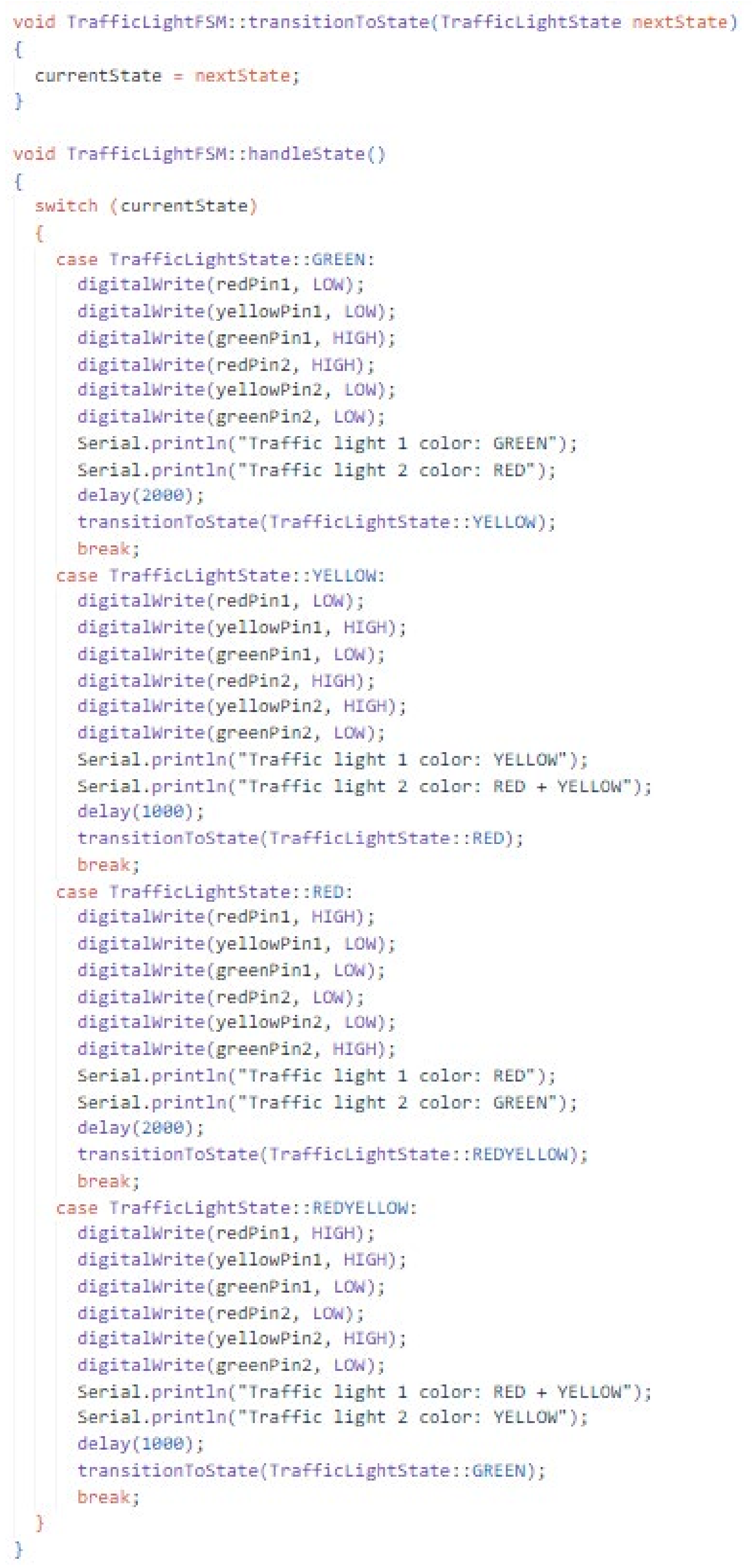
TrafficLightState::GREEN: Represents the state when one traffic light is green and the other is red.

TrafficLightState::YELLOW: Represents the state when one traffic light is yellow and the other is red and yellow.

TrafficLightState::RED: Represents the state when one traffic light is red and the other is green.

TrafficLightState::REDYELLOW: Represents the state when one traffic light is red and the other is yellow.

The provided code snippet shows a simple traffic light FSM with a predefined sequence of state transitions. It controls the digital outputs of two traffic lights to simulate the behavior of traffic lights in a basic intersection scenario.



```
void TrafficLightFSM::transitionToState(TrafficLightState nextState)
{
  currentState = nextState;
}

void TrafficLightFSM::handleState()
{
  switch (currentState)
  {
    case TrafficLightState::GREEN:
      digitalWrite(redPin1, LOW);
      digitalWrite(yellowPin1, LOW);
      digitalWrite(greenPin1, HIGH);
      digitalWrite(redPin2, HIGH);
      digitalWrite(yellowPin2, LOW);
      digitalWrite(greenPin2, LOW);
      Serial.println("Traffic light 1 color: GREEN");
      Serial.println("Traffic light 2 color: RED");
      delay(2000);
      transitionToState(TrafficLightState::YELLOW);
      break;
    case TrafficLightState::YELLOW:
      digitalWrite(redPin1, LOW);
      digitalWrite(yellowPin1, HIGH);
      digitalWrite(greenPin1, LOW);
      digitalWrite(redPin2, HIGH);
      digitalWrite(yellowPin2, HIGH);
      digitalWrite(greenPin2, LOW);
      Serial.println("Traffic light 1 color: YELLOW");
      Serial.println("Traffic light 2 color: RED + YELLOW");
      delay(1000);
      transitionToState(TrafficLightState::RED);
      break;
    case TrafficLightState::RED:
      digitalWrite(redPin1, HIGH);
      digitalWrite(yellowPin1, LOW);
      digitalWrite(greenPin1, LOW);
      digitalWrite(redPin2, LOW);
      digitalWrite(yellowPin2, LOW);
      digitalWrite(greenPin2, HIGH);
      Serial.println("Traffic light 1 color: RED");
      Serial.println("Traffic light 2 color: GREEN");
      delay(2000);
      transitionToState(TrafficLightState::REDYELLOW);
      break;
    case TrafficLightState::REDYELLOW:
      digitalWrite(redPin1, HIGH);
      digitalWrite(yellowPin1, HIGH);
      digitalWrite(greenPin1, LOW);
      digitalWrite(redPin2, LOW);
      digitalWrite(yellowPin2, HIGH);
      digitalWrite(greenPin2, LOW);
      Serial.println("Traffic light 1 color: RED + YELLOW");
      Serial.println("Traffic light 2 color: YELLOW");
      delay(1000);
      transitionToState(TrafficLightState::GREEN);
      break;
  }
}
```

Fig 4 Handling the state of the traffic light using handleState() and transitionToState()

Electrical scheme made in Proteus

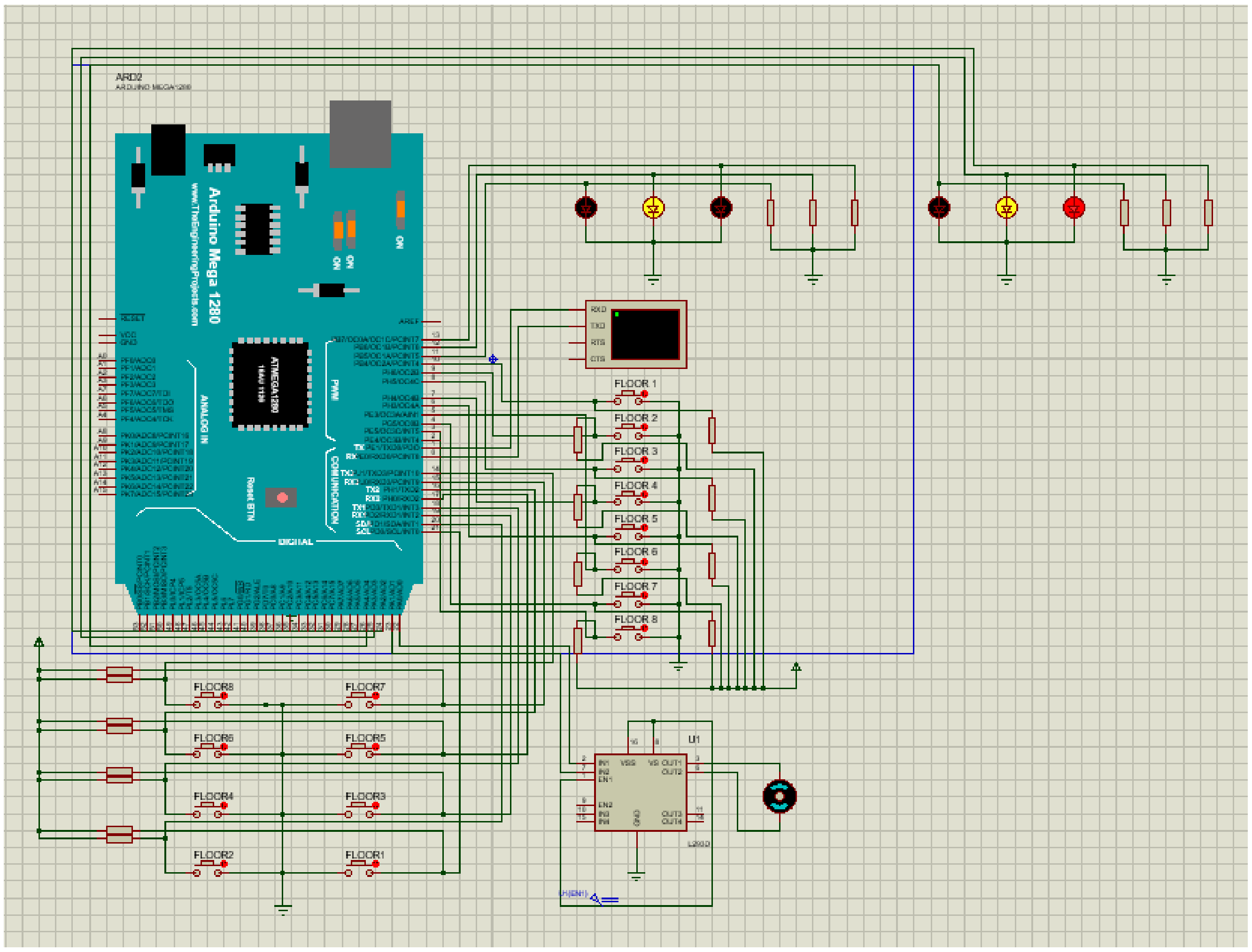


Fig 5 Electrical scheme

CONCLUSIONS

As a student, I have successfully completed the laboratory work of developing an MCU-based application to implement management systems for an elevator in a staircase and a traffic light using finite automata.

Throughout the project, I encountered various challenges that allowed me to deepen my understanding of both finite automata and MCU programming. Designing the elevator system required careful consideration of different states representing floors and transitions representing the elevator's movement. Factors such as user input, safety measures, and door operations had to be incorporated into the system to ensure its proper functioning.

Similarly, developing the traffic light system involved designing an automaton with states representing different colors and transitions indicating the timing sequence between these colors. The system had to switch between colors based on predefined time intervals and respond to external input from sensors or control mechanisms.

Implementing these systems on an MCU required me to write efficient and modular code that could handle the necessary inputs and outputs while ensuring the reliable operation of the systems. I had to consider hardware limitations and make optimal use of the MCU's resources.

Working on this laboratory project provided me with valuable insights into software engineering practices. I learned the importance of proper system design, modular programming, and thorough testing. I also recognized the significance of documentation in conveying my work and making it replicable for others.

Overall, this laboratory work allowed me to apply theoretical concepts to real-world applications and gain practical experience in developing MCU-based systems. It enhanced my problem-solving skills, deepened my understanding of finite automata, and improved my proficiency in MCU programming. I am confident that the knowledge and skills acquired during this project will be valuable in my future endeavors as an engineer.