TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Report of laboratory Work №4

Actuators

Student: Pogorevici Daniel, FAF-202

Teacher: univ.lecturer Moraru Dumitru

Chisinau 2023

1. The Task of the Laboratory Work

For understanding principles of work with actuators and setting interaction of user with actuator-based embedded system in current laboratory work will be implemented motor and led control system, based on command line input from user. So, work requires implementation of four steps:

1. Read commands from the command line interface via keyboard (serial input);
2. Perform action based on command over the Relay and therefore the bulb;
3. Perform action based on command over the motor;
4. Allow user to set parameters of how to perform action via motor.

Considering the complexity of the task, it will be the best practice to divide all code parts into layers that will be implemented inside the main file. It means writing library/driver for actuators to simplify interaction with them.

2. Implementation

For this laboratory work I set the default speed of the motor which is gonna be 100.

```
void setup()
{
  Serial.begin(19200);

  // Set initial motor speed
  motor.setSpeed(100);

  attachInterrupt(digitalPinToInterrupt(2), rpmInterrupt, RISING);
}
```

Fig. 1 - The speed of the motor and attaching an interrupt

After that I have the loop function which takes user inputs and depending on what you inserted a command is executed.

```
void loop()
{
  while (Serial.available() == 0) {}
  userInput = Serial.readStringUntil('\r');
  userInput.toLowerCase();

  String command = userInput.substring(0, userInput.indexOf(' '));
  int value = userInput.substring(userInput.indexOf(' ') + 1).toInt();

  if (command == RELAY_ON) ···
  else if (command == RELAY_OFF) ···
  else if (command == RPM_CHANGE) ···
  else if (userInput == RPM_GET) ···
  else if (command == TURN_LEFT) ···
  else if (command == TURN_RIGHT) ···
  else
  {
    Serial.println("Command unknown.");
  }
}
```

Fig. 2 - Commands issued by the user input

We can see how these are performed by observing the Relay and the Motor classes:

```cpp
#include "Relay.h"

Relay::Relay(int pin) : _pin(pin) {
    pinMode(_pin, OUTPUT);
}

void Relay::on() {
    digitalWrite(_pin, HIGH);
    Serial.println("Bulb turned on.");
}

void Relay::off() {
    digitalWrite(_pin, LOW);
    Serial.println("Bulb turned off.");
}
```

Fig. 3 - Relay class

```cpp
#include "Motor.h"

Motor::Motor(int in1, int in2, int enable) : _in1(in1), _in2(in2), _enable(enable) {
    pinMode(_in1, OUTPUT);
    pinMode(_in2, OUTPUT);
    pinMode(_enable, OUTPUT);

    // Set initial motor rotation clockwise
    digitalWrite(_in1, HIGH);
    digitalWrite(_in2, LOW);
}

void Motor::setSpeed(int speed) {
    analogWrite(_enable, speed);
    Serial.print("Motor speed set to: ");
    Serial.println(speed % 256);
}

double Motor::getSpeed() {
    return _rpm;
}

void Motor::updateRpm(double rpm) {
    _rpm = rpm;
}

void Motor::setDirection(int direction) {
  if (direction == 1) {
    digitalWrite(_in1, HIGH);
    digitalWrite(_in2, LOW);
  } else if (direction == -1) {
    digitalWrite(_in1, LOW);
    digitalWrite(_in2, HIGH);
  }
}
```

Fig. 4 - Motor class

Additionally, the interruption is also worth mentioning about in order to determine the rpm:

```
void rpmInterrupt()
{
  interrupt_flag = !interrupt_flag;

  if (interrupt_flag == 1)
  {
    currentMicros = micros();
  }
  else
  {
    // Calculate pulse interval between current and previous micros() values
    unsigned long pulseInterval = currentMicros - previousMicros;

    updatePulseIntervalFilter(pulseInterval);

    // Calculate average pulse interval
    unsigned long avgPulseInterval = sumPulseIntervals / filterSize;

    if (avgPulseInterval > 0)
    {
      calculateRpm(avgPulseInterval);
    }

    previousMicros = currentMicros;
  }

  // Update the motor RPM value
  motor.updateRpm(rpm);
}
```

Fig. 5 - rpmInterrupt function

We also can see the presence of the helper functions in order to determine an average for the rpm as well as the formulas for the frequency and the computation:

```
// Update the pulse interval filter with the new pulse interval
void updatePulseIntervalFilter(unsigned long pulseInterval)
{
  // Remove oldest pulse interval from sum and add new pulse interval
  sumPulseIntervals -= pulseIntervals[filterIndex];
  pulseIntervals[filterIndex] = pulseInterval;
  sumPulseIntervals += pulseIntervals[filterIndex];

  // Update filter index for the next sample
  filterIndex = (filterIndex + 1) % filterSize;
}

// Calculate motor RPM based on average pulse interval
void calculateRpm(unsigned long avgPulseInterval)
{
  // Calculate frequency from average pulse interval (taking half of the interval due to rising and falling edges)
  frequency = 1000000.0 / (double)(0.5 * avgPulseInterval);

  // Calculate RPM from frequency and pulses per rotation
  rpm = (60.0 * frequency) / (double)pulsesPerRotation;
}
```

Fig. 6 - Compute the rpm

Some clarification might appear to be needed regarding the formulas.

The original formula for frequency uses *micros()* instead of *millis()*, hence the larger constant *1000000.0*. Also, the original formula accounts for both rising and falling edges by multiplying the *avgPulseInterval* by 0.5.

The original formula for rpm directly multiplies the frequency by 60 and then divides by the number of pulses per rotation. The provided formula calculates the period of the pulses first, and then divides 60 by the product of the period and the number of pulses per rotation. Both formulas yield the same result for RPM:

```
// Calculate frequency from average pulse interval (taking half of the interval due to rising and falling edges)
// frequency = (1.0/(double)(abs(currentMillis-previousMillis)))*1000.0;
frequency = 1000000.0 / (double)(0.5 * avgPulseInterval);

// Calculate RPM from frequency and pulses per rotation
// rpm = 60.0 / ((1.0/frequency) * 24.0);
rpm = (60.0 * frequency) / (double)pulsesPerRotation;
```

Fig. 7 - Computing rpm and frequency

3. Simulated electrical schema

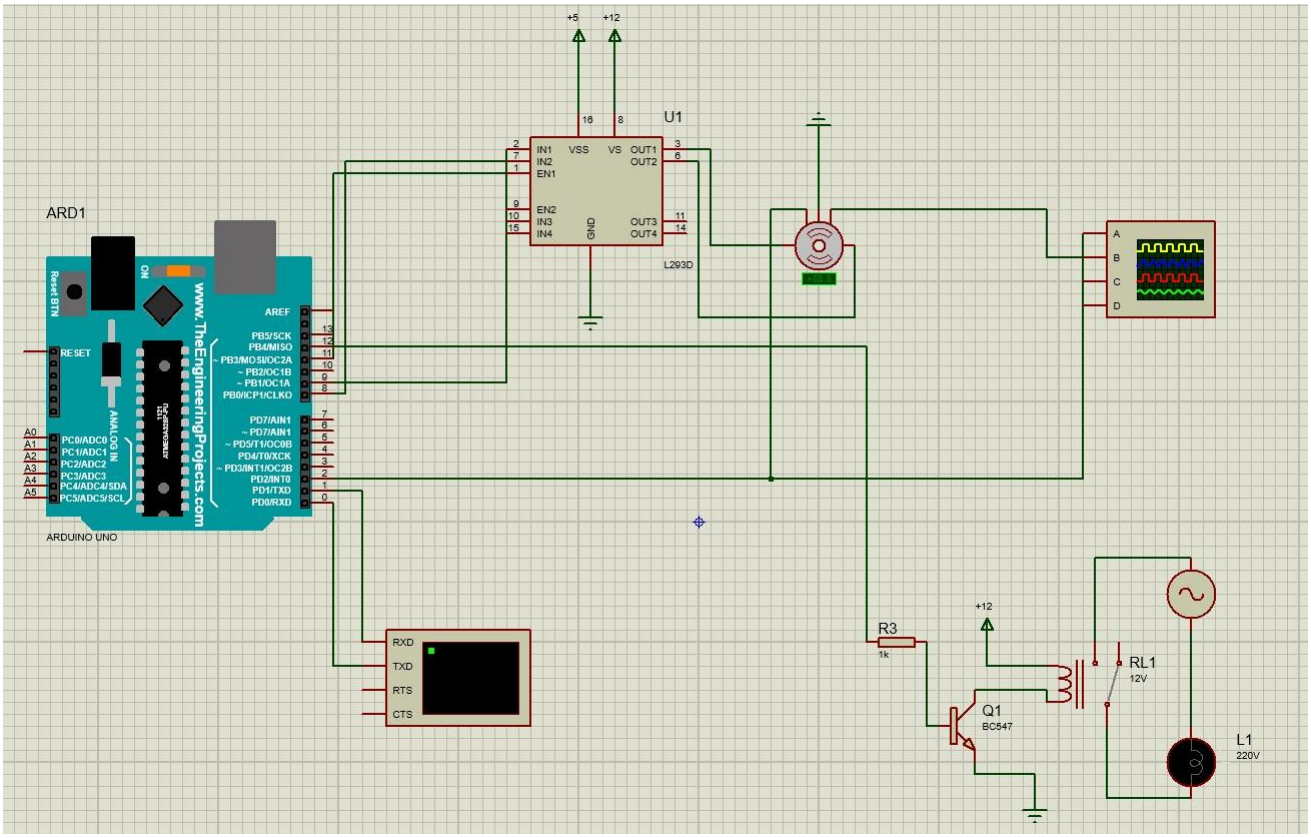

Fig. 8 - Simulated electrical schematic.

4. Conclusions

In conclusion, my laboratory project provided me with valuable insights into working with actuators in embedded systems, a critical aspect of IoT. I learned that breaking down the code into modules is an effective way to develop a complex yet user-friendly system. This approach also allows for seamless integration of new features in the future.

Additionally, programming actuators requires a comprehensive understanding of various aspects, making it a crucial skill for aspiring engineers.