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

Report of laboratory work №5

Theme: Control

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). regulation of temperature or humidity with the application of the On-Off driving method with hysteresis and with relay operation

b). engine speed adjustment with the application of the PID method with an encoder as a sensor, and L298 driver for the application of power to the engine

NOTE: in p (b) you can choose another control parameter, with the constraint that the drive will have a resolution of at least 8 bits.

The set point (control setpoint) will be set from one of the sources of your choice:

- a potentiometer
- two buttons for UP / Down
- encoder sensor
- keypad
- serial interface

The Setpoint and Current values will be displayed on the LCD.

THE PROGRESS OF THE WORK

To solve this task, it is used a heater, LM35 analog temperature sensor and potentiometer (to set the target temperature).

```
currentTemperature = lm.readData();
potentiometerValue = analogRead(potentiometerPin);
targetTemperature = map(potentiometerValue, 0, 1023, 0, 100);
if ((currentTemperature > targetTemperature + hysteresis) && heaterState)
{
  relay.turnOff();
  heaterState = false;
}
if ((currentTemperature < targetTemperature - hysteresis) && !heaterState)
{
  relay.turnOn();
  heaterState = true;
}
```

Fig 1 Regulation of the temperature using the On-Off driving method

It is read the data from the sensor and compared with the target + or – hysteresis. If it crosses the border, the heater is turned on or off and the PID regulator is started.

The task of the PID controller is to ensure that the temperature of the sensor is equal to the target value, which will be set programmatically. Direct adjustment will occur by changing the fan speed, changing the duty cycle of the PWM signal.

If the current error is positive, and at the same time the value of the integrating component does not exceed the maximum value of the PWM signal pulse duration, then it is accumulated the error in *integralError*, otherwise not. The same is true for a negative error value. Next, it is differentiated the residual and calculated the output.

```
differentialError = (currentError - previousError) / timeIntervalSec;
pwmDutyCycle = -Kp * currentError + Ki * integralError + Kd * differentialError;
if (pwmDutyCycle < pid_duty_cycle_min)
{
  pwmDutyCycle = pid_duty_cycle_min;
}
if (pwmDutyCycle > pid_duty_cycle_max)
{
  pwmDutyCycle = pid_duty_cycle_max;
}
motor.drive(pwmDutyCycle);
```

Fig 2 Calculating the output

Also, it is needed to check if *pwmDutyCycle* doesn't exceed bounds.

The meaning of using a PID controller is that it will provide control of the parameter, regardless of changes in external uncontrolled factors. Mathematically, the principle of operation can be represented in a simple way: $y(t)=f(e(t))$

Where:

- $y(t)$ - input signal
- $e(t)$ - difference between the target and the current value of the controlled variable.

And the PID controller gives a mechanism to calculate *y(t)* from *e(t)*. For now, it should have been systematically moving to consider how exactly these calculations occur. The controller output signal is defined as follows:

$$y(t) = P + I + D = K_p \times e(t) + K_i \times \int_0^t e(\tau)d\tau + K_e \times \frac{de}{dt}$$

It is an algebraic sum of three components, which gave the name to the controller - PID:

- $K_p \times e(t)$ - proportional component
- $K_i \times \int_0^t e(\tau)d\tau$ - integrating component
- $K_{\square} \times \frac{de}{dt}$ - differential component

It should be noted right away that not all components can be used, but only a part of them, then the regulator will be called proportionally differentiating, proportionally integrating, etc. The logic of forming names here is simple and obvious.

The formula has three uncertain values, the selection of which is the setting of the PID controller. Here it is talked about the gains of the proportional, integrating and differentiating components (K_p, K_i , $K_{\square}$) that can be calculated using different methods, for example Ziegler-Nichols method.

The Ziegler-Nichols method in the sequential execution of the following operations:

To reset all coefficients of the regulator.

- To set some target value of the controlled parameter (for example, temperature).
- To begin to increase gradually the proportional coefficient and monitor the reaction of the system.
- At a certain value of K_p, undamped oscillations of the controlled variable will occur.
- To fix this value, as well as the oscillation period of the system.

This concludes the practical part of the method. From the obtained values we calculate the coefficients:

$$K_p = 0.6 \times K$$

$$K_i = (2 \times K_p) / T$$

$$K_d = (K_p \times T) / 8$$

Here K is the same coefficient of the proportional component at which oscillations occurred, and T is the period of these oscillations.

Electrical scheme made in Proteus

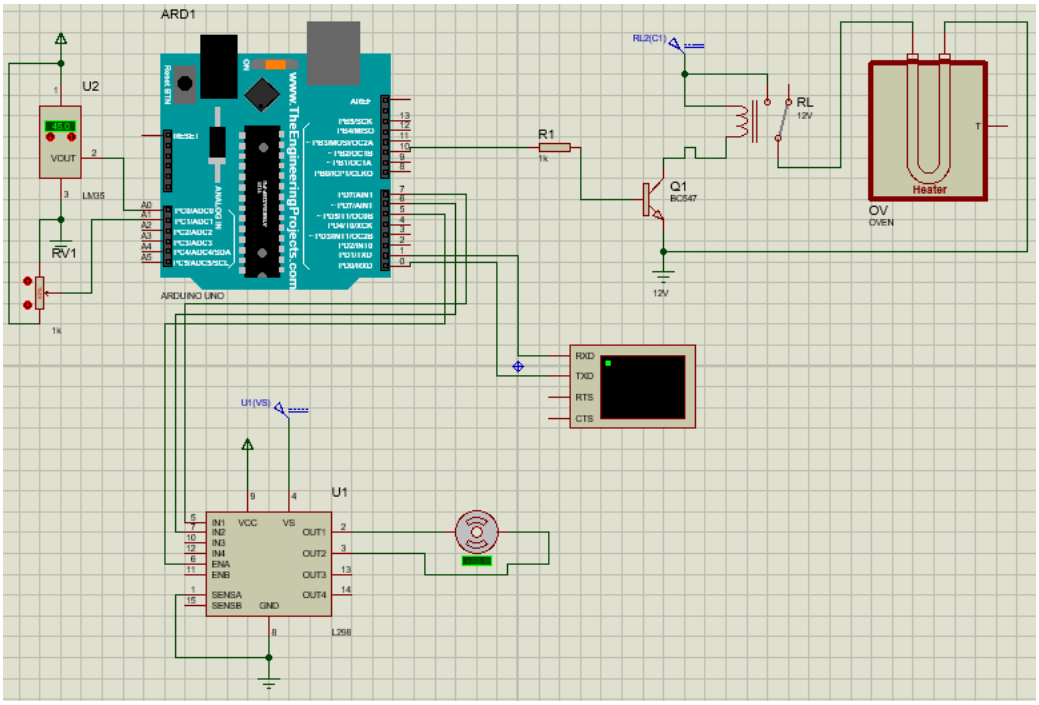

Fig 3 Electrical scheme

Code

// main.cpp

```
#include "LM35.h"
#include "MotorDriver.h"
#include "Relay.h"

#include <Arduino.h>

#define lm35Pin A0
#define potentiometerPin A1
#define relayPin 10
#define IN1 7
#define IN2 6
#define ENA 5

float targetTemperature = 50;
int currentTemperature = 0;
int potentiometerValue = 0;
int hysteresis = 3;
bool heaterState = false;

float pwmDutyCycle = 0;

int pid_duty_cycle_min = 0;
int pid_duty_cycle_max = 100;

float Kp = 10;
float Ki = 1;
float Kd = 1;

float previousError = 0;
float currentError = 0;
float integralError = 0;
float differentialError = 0;

int lastUpdateTime = 0;
int updateIntervalMs = 100;

LM35 lm(lm35Pin);
Relay relay(relayPin);
MotorDriver motor(IN1, IN2, ENA);

void setup()
{
```

```
  relay.init();
  pinMode(potentiometerPin, INPUT);
  Serial.begin(9600);
  while (!Serial);
}

void loop() {
  delay(2000);

  int currentTime = millis();
  if (currentTime - lastUpdateTime >= updateIntervalMs)
  {
    // Task 1
    lastUpdateTime = currentTime;
    currentTemperature = lm.readData();
    potentiometerValue = analogRead(potentiometerPin);
    targetTemperature = map(potentiometerValue, 0, 1023, 0, 100);
    if ((currentTemperature > targetTemperature + hysteresis) && heaterState)
    {
      relay.turnOff();
      heaterState = false;
    }
    if ((currentTemperature < targetTemperature - hysteresis) && !heaterState)
    {
      relay.turnOn();
      heaterState = true;
    }

    // Task 2
    float timeIntervalSec = (float)updateIntervalMs / 1000;
    currentError = currentTemperature - targetTemperature;
    integralError += currentError * timeIntervalSec;
    if (integralError < pid_duty_cycle_min / Ki)
    {
      integralError = pid_duty_cycle_min / Ki;
    }
    else if (integralError > pid_duty_cycle_max / Ki)
    {
      integralError = pid_duty_cycle_max / Ki;
    }
    differentialError = (currentError - previousError) / timeIntervalSec;
    pwmDutyCycle = -Kp * currentError + Ki * integralError + Kd * differentialError;
    if (pwmDutyCycle < pid_duty_cycle_min)
    {
      pwmDutyCycle = pid_duty_cycle_min;
    }
    if (pwmDutyCycle > pid_duty_cycle_max)
```

```
    {
      pwmDutyCycle = pid_duty_cycle_max;
    }
    motor.drive(pwmDutyCycle);
    previousError = currentError;

    float dutyCiclePercentage = pwmDutyCycle;

    Serial.print("Target temperature: ");
    Serial.print(targetTemperature);
    Serial.println(" C");
    Serial.print("Current temperature: ");
    Serial.print(currentTemperature);
    Serial.println(" C");

    Serial.print("Duty cycle percentage: ");
    Serial.print(dutyCiclePercentage);
    Serial.println("%");

    Serial.println(" ");

    delay(2000);
    }
}
```

// LM35.h

```
#ifndef LM35_H
#define LM35_H

#include <Arduino.h>

class LM35
{
    private:
        int sensorPin;
    public:
        LM35(int pin);
        float readData();
};

#endif
```

// LM35.cpp

```cpp
#include "LM35.h"

LM35::LM35(int pin)
{
    sensorPin = pin;
}

float LM35::readData()
{
    return analogRead(sensorPin) * 0.48875;
}
```

// Relay.h

```cpp
#ifndef RELAY_H
#define RELAY_H

#include <Arduino.h>

class Relay
{
    private:
        int relayPin;
    public:
        Relay(int pin);
        void init();
        void turnOn();
        void turnOff();
};

#endif
```

// Relay.cpp

```cpp
#include "Relay.h"

Relay::Relay(int pin)
{
    relayPin = pin;
}

void Relay::init()
{
    pinMode(relayPin, OUTPUT);
    digitalWrite(relayPin, LOW);
```

```
}

void Relay::turnOn()
{
    digitalWrite(relayPin, HIGH);
}

void Relay::turnOff()
{
    digitalWrite(relayPin, LOW);
}

// MotorDriver.h

#ifndef MOTORDRIVER_H
#define MOTORDRIVER_H

#include "Motor.h"

#include <Arduino.h>

class MotorDriver
{
    private:
        Motor motor;
    public:
        MotorDriver(int pin1, int pin2, int pin3);
        void drive(int speed);
};

#endif

// MotorDriver.cpp

#include "MotorDriver.h"

MotorDriver::MotorDriver(int pin1, int pin2, int pin3)
{
    motor.init(pin1, pin2, pin3);
}

void MotorDriver::drive(int speed)
{
    motor.changeSpeed((speed >= 0), (abs(speed) * 2.55 + 0.5));
}
```

```
// Motor.h

#ifndef MOTOR_H
#define MOTOR_H

#include <Arduino.h>

class Motor
{
    private:
        byte motorPin1;
        byte motorPin2;
        byte controlPin;

    public:
        Motor();
        void init(byte pin1, byte pin2, byte pin3);
        void changeSpeed(bool direction, int speed);
};

#endif

// Motor.cpp

#include "Motor.h"

Motor::Motor()
{
}

void Motor::init(byte pin1, byte pin2, byte pin3)
{
    motorPin1 = pin1;
    motorPin2 = pin2;
    controlPin = pin3;

    pinMode(motorPin1, OUTPUT);
    pinMode(motorPin2, OUTPUT);
    pinMode(controlPin, OUTPUT);
}

void Motor::changeSpeed(bool direction, int speed)
{
    if (direction)
    {
        digitalWrite(motorPin1, HIGH);
        digitalWrite(motorPin2, LOW);
```

```
        analogWrite(controlPin, speed);
    }
    else
    {
        digitalWrite(motorPin1, LOW);
        digitalWrite(motorPin2, HIGH);
        analogWrite(controlPin, speed);
    }
}
```

CONCLUSIONS

In this laboratory work, I gained an understanding of the fundamental principles behind controlling embedded systems using various approaches and the benefits of employing different control methods. This knowledge will be crucial in developing complex and practical embedded systems that can be used in significant projects. Efficiently managing various control options is critical for engineers, as it necessitates the development of appropriate algorithms and control methods to ensure that the system functions optimally and reaches its full potential. The effective control of embedded systems is critical for the successful implementation of algorithms within the systems, and it is necessary to ensure that the system's performance and potential are not compromised.