**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-191 Boico Alexandr**

**Controlled: univ. lecturer Moraru Dumitru**

**Chișinău 2022**

**The task of the laboratory work:**

1. 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 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.

1. 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

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

**The progress of the work**

To realize this issue I have used a heater, LM35 analog temperature sensor and varistor( to set target temperature).

**Appendix 1**

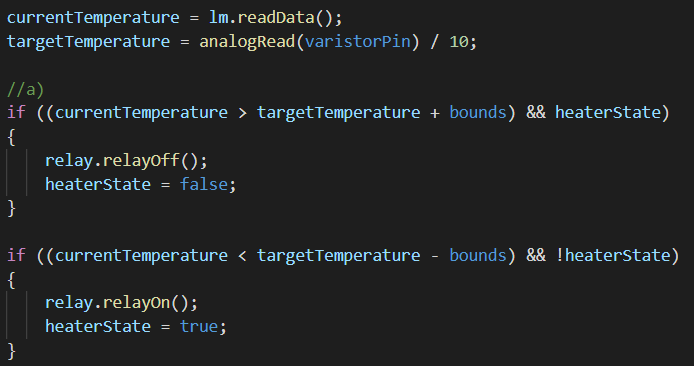


Fig 1. regulation of temperature driving method with hysteresis with relay operation

We read data from the sensor and compare it with the target + or - bounds. If it crosses the border then we turn on the heater or turn off and start the PID regulator.

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, by changing the duty cycle of the PWM signal.

**Appendix 2**

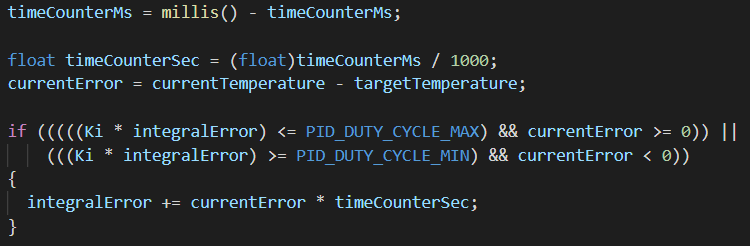


Fig 2. An additional constraint for the integrating component

I introduced an additional constraint for the integrating component here so that too large values do not accumulate, which can lead to increased system inertia( Appendix 2).

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 we accumulate the error in *integralError*, otherwise we do not. The same is true for a negative error value.

Next, we differentiate the residual and calculate the output.

**Appendix 3**

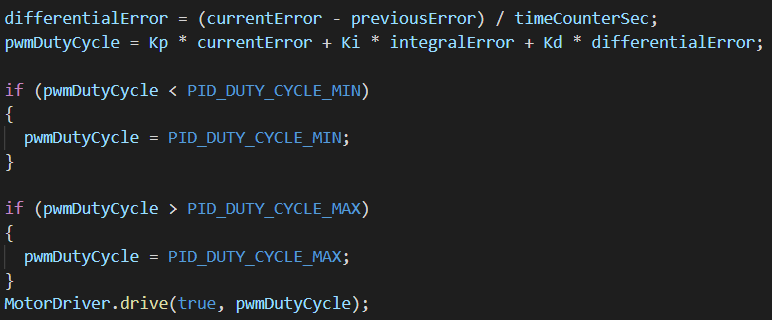


Fig 3. Calculating output

Also we need 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 very simply: *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 us a mechanism to calculate *y(t)* from *e(t)*. Again I refer to a future example in which everything will finally fall into place. In the meantime, we are systematically moving to consider how exactly these calculations occur. The controller output signal is defined as follows:

We have the algebraic sum of three components, which gave the name to the controller - PID:

* - proportional component
* - integrating component
* - 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. We are talking about the gains of the proportional, integrating and differentiating components (, , ) that we can calculate using different methods, for example Ziegler-Nichols method.

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

1. To reset all coefficients of the regulator.
2. To set some target value of the controlled parameter (for example, temperature).
3. To begin to increase gradually the proportional coefficient and monitor the reaction of the system.
4. At a certain value of , undamped oscillations of the controlled variable will occur.
5. 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:

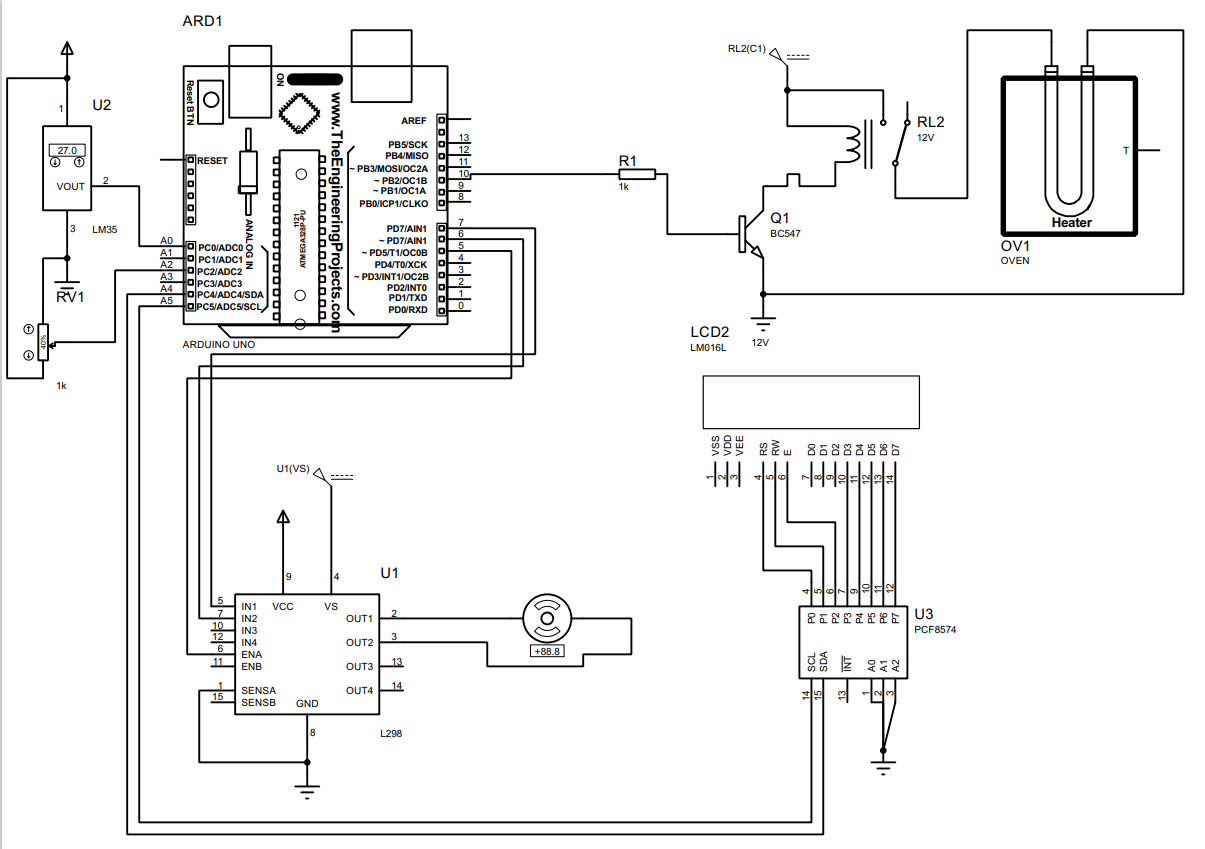
Here *K* is the same coefficient of the proportional component at which oscillations occurred, and *T* is the period of these oscillations.

**Conclusion**

After performing the 5th laboratory work, I have learned a lot of new information about PID controllers, have seen examples of their realizations on other platforms( for example STM32)

**Appendix**

**Electrical scheme in proteus**

****

**Code**

* main.cpp

#include <Arduino.h>

#include "Relay.h"

#include "MotorDriver298.h"

#include "LM35.h"

#include <Wire.h>

#include <LiquidCrystal\_I2C.h>

LiquidCrystal\_I2C lcd(0x20, 16, 4); //for proteus

//Setup data

#define startTemperature 10

#define bounds 3

#define PID\_DUTY\_CYCLE\_MIN 0

#define PID\_DUTY\_CYCLE\_MAX 100

float pwmDutyCycle = 500;

float Kp = 300;

float Ki = 10;

float Kd = 900;

float previousError = 0;

float currentError = 0;

float integralError = 0;

float differentialError = 0;

uint32\_t timeCounterMs = 0;

float targetTemperature = startTemperature;

int currentTemperature = startTemperature;

//Pins

#define lm35Pin A0 //Pin for LM35

#define varistorPin A2 //Pin for temperature setting

#define relayPin 10 //Pin for Relay-heater

#define motorPin1 7

#define motorPin2 6

#define motorPin3 5

Relay relay(relayPin);

MotorDriver298 MotorDriver(motorPin1, motorPin2, motorPin3);

LM35 lm(lm35Pin);

bool heaterState = false;

void dataOnLCD();

void setup() {

lcd.init();

lcd.backlight();

relay.init();

}

void loop() {

currentTemperature = lm.readData();

targetTemperature = analogRead(varistorPin) / 20;

//a)

if ((currentTemperature > targetTemperature + bounds) && heaterState)

{

relay.relayOff();

heaterState = false;

}

if ((currentTemperature < targetTemperature - bounds) && !heaterState)

{

relay.relayOn();

heaterState = true;

}

//b)

timeCounterMs = millis() - timeCounterMs;

float timeCounterSec = (float)timeCounterMs / 1000;

currentError = currentTemperature - targetTemperature;

if (((((Ki \* integralError) <= PID\_DUTY\_CYCLE\_MAX) && currentError >= 0)) ||

(((Ki \* integralError) >= PID\_DUTY\_CYCLE\_MIN) && currentError < 0))

{

integralError += currentError \* timeCounterSec;

}

differentialError = (currentError - previousError) / timeCounterSec;

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;

}

MotorDriver.drive(true, pwmDutyCycle);

previousError = currentError;

dataOnLCD();

}

void dataOnLCD()

{

lcd.setCursor(0,0);

lcd.print("Target:");

lcd.setCursor(10,0);

lcd.print(targetTemperature);

lcd.setCursor(14,0);

lcd.print("C");

lcd.setCursor(0,1);

lcd.print("Сurrent:");

lcd.setCursor(10,1);

lcd.print(currentTemperature);

lcd.setCursor(14,1);

lcd.print("C");

}

* LM35.h

#ifndef LM35\_h

#define LM35\_h

#include <Arduino.h>

class LM35

{

public:

LM35(byte pin);

void initSensor();

float readData();

float convertMillivotsToCelsius(int value);

protected:

byte SensorPin;

};

#endif

* LM35.cpp

#include "LM35.h"

LM35::LM35(byte pin)

{

SensorPin = pin;

}

float LM35::readData()

{

return analogRead(SensorPin)/2;

}

* Relay.h

#ifndef Relay\_h

#define Relay\_h

#include <Arduino.h>

class Relay

{

private:

byte relayPin;

public:

Relay(byte pin);

void init();

void relayOn();

void relayOff();

};

#endif

* Relay.cpp

#include "Relay.h"

Relay::Relay(byte pin)

{

relayPin = pin;

}

void Relay::init()

{

pinMode(relayPin, OUTPUT);

digitalWrite(relayPin, LOW);

}

void Relay::relayOn()

{

digitalWrite(relayPin, HIGH);

}

void Relay::relayOff()

{

digitalWrite(relayPin, LOW);

}

* 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 changeStateWithSpeed(bool direction, int speed);

// direction = true -> clockwise

// direction = false -> anti-clockwise

void changeState(bool direction);

};

#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::changeState(bool direction)

{

if (direction)

{

digitalWrite(MotorPin1, HIGH);

digitalWrite(MotorPin1, LOW);

}

else

{

digitalWrite(MotorPin1, LOW);

digitalWrite(MotorPin1, HIGH);

}

}

void Motor::changeStateWithSpeed(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);

}

}

* MotorDriver.h

#ifndef MotorDriver298\_h

#define MotorDriver298\_h

#include <Arduino.h>

#include "Motor.h"

class MotorDriver298

{

private:

Motor motorA;

Motor motorB;

bool motorQuantity = false; //true = 2 motors, false - 1 motor

public:

MotorDriver298(byte pin1, byte pin2, byte pin3); //only motor A

MotorDriver298(byte pin1, byte pin2, byte pin3, byte pin4, byte pin5, byte pin6); //motorA and motorB

void drive(bool side, int speed);

// motor "A" -> side = true

// motor "B" -> side = false

};

#endif

* MotorDriver.cpp

#include "MotorDriver298.h"

MotorDriver298::MotorDriver298(byte pin1, byte pin2, byte pin3)

{

motorA.init(pin1, pin2, pin3);

}

MotorDriver298::MotorDriver298(byte pin1, byte pin2, byte pin3, byte pin4, byte pin5, byte pin6)

{

motorA.init(pin1, pin2, pin3);

motorB.init(pin4, pin5, pin6);

motorQuantity = true;

}

void MotorDriver298::drive(bool side, int speed)

{

if(!side && motorQuantity)

{

motorB.changeStateWithSpeed((speed >= 0), (abs(speed) \* 2.55 + 0.5));

}

else

{

motorA.changeStateWithSpeed((speed >= 0), (abs(speed) \* 2.55 + 0.5));

}

}