

PID Temperature Controller

**Presented To:**

Sarmad Hassan

**Presented By:**

Muhammad Aliyan Jamshaid

(FA21-BCE-042)

Saad Ali Qamar

(FA21-BCE-044)

Table of Contents

[1. INTRODUCTION 3](#_Toc167682003)

[1.1 Introduction: 3](#_Toc167682004)

[1.2 Aims and Objectives: 3](#_Toc167682005)

[1.3 Expected Outcomes: 4](#_Toc167682006)

[2. PROJECT IMPLEMENTATION 3](#_Toc167682003)

[2.1 Hardware Components: 4](#_Toc167682007)

[2.2 Software Components: 7](#_Toc167682008)

[2.3 Block Diagram: 9](#_Toc167682009)

[2.4 Methodology: 9](#_Toc167682011)

[Hardware Configuration: 9](#_Toc167682012)

[Schematic: 10](#_Toc167682013)

[PID Control Implementation: 11](#_Toc167682014)

[Integration with BLYNK: 12](#_Toc167682015)

[Data analysis and Visualization: 12](#_Toc167682016)

3. EXPERIMENTS AND RESULTS 13

4. CONCLUSION 14

5. REFERENCES 15

. 6. FINAL HARDWARE 16

7. BLYNK DASHBOARD 17

8. CODE 18

## INTRODUCTION

## Introduction:

Temperature control is a critical aspect in various applications, ranging from industrial processes to environmental comfort systems. In this project, we focus on developing a Temperature PID Controller using the Blynk platform with ESP32, aimed at efficiently regulating the temperature of a box effectively.

The fundamental idea behind our project is a closed-loop control system, which continuously modifies control actions to reach and maintain the intended temperature setpoint based on feedback from the system's output. One of the main benefits of feedback control is that it can guarantee precise tracking of the target temperature setpoint, especially in systems with high loop gain. Additionally, feedback greatly improves the control system's responsiveness, allowing for quick adjustments in response to outside variations or disturbances.

Our project aims to demonstrate the practical implementation of temperature control using a PID (Proportional-Integral-Derivative) controller within a room environment, facilitated by the Blynk platform and ESP32 microcontroller. Temperature sensing is achieved using DS18B20 temperature sensor. The ESP32's built-in Pulse Width Modulation (PWM) capabilities enable it to regulate a 12V DC heating bulb or cooling fan by adjusting the duty cycle of the PWM signal. This, in turn, controls the average power delivered, efficiently adjusting the room temperature. This project showcases a practical temperature control system for a room environment.

## Aims and Objectives:

Our project addresses the critical need for real-time temperature monitoring in data center design. Traditional methods of estimating cooling capacity reserves often lead to inefficient resource allocation, with some areas experiencing excessive heat while others remain underutilized. Our aim is to develop a solution using PID control techniques that leverages real-time data to optimize cooling efficiency, prevent equipment overheating, and minimize energy wastage, thereby ensuring the reliability and efficiency of data center operations.

#### Objectives

1. Develop a real-time temperature monitoring system using ESP32 microcontroller to gather data from multiple points within a room.
2. Implement PID (Proportional-Integral-Derivative) control algorithms to regulate the temperature based on collected sensor data and the set point.
3. Integrate an IoT platform like Blynk or Thing Speak for remote monitoring and control of the temperature control system.
4. Using Python libraries for data analysis and visualization to identify temperature distribution patterns and assess system performance.
5. Conduct rigorous testing and validation of the temperature control system to ensure reliability and effectiveness.
6. Document the project's design, implementation, and evaluation processes, providing insights and recommendations for future deployments in real-world data center settings.

## Expected Outcomes:

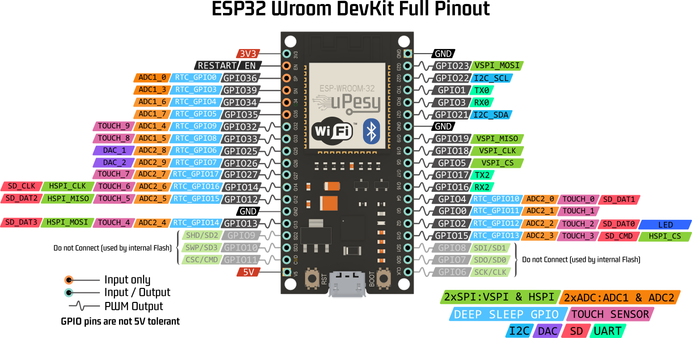
The temperature control system operates by continuously measuring the room temperature within the enclosed chamber and comparing it with the predefined setpoint. In a feedback loop with a temperature sensor, the system detects any change in temperature from the setpoint and calculates an error value. Subsequently, the PID (Proportional-Integral-Derivative) controller dynamically adjusts the operation of the heater or fan to minimize this error. Subsequently, the PID controller dynamically uses PWM signals to adjust the operation of the heater or fan to minimize this error. If the current temperature is less than the setpoint, the heater is activated to raise the temperature, while if it is greater than the setpoint, the fan is turned on to cool the chamber. The PID controller's effectiveness is determined by its tuning parameters - Kp, Ki, and Kd – By changing these parameters we can influence the system’s characteristics such as overshoot, settling time, and rise time, fine tuning the response to changes in temperature, optimizing system performance.

1. PROJECT IMPLEMENTATION

## Hardware Components:

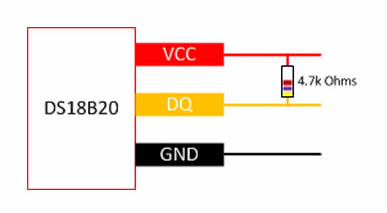
#### ESP32 Node MCU

We are using ESP32 Node MCU microcontroller for our project implementation because of its versatile features that perfectly align with the requirements of our project. While Arduino is great for basic projects, the ESP32 provides built-in Wi-Fi and Bluetooth connectivity, which is essential for remote monitoring and control. Additionally, the ESP32 has more processing power and memory, enabling us to implement complex algorithms like PID control more effectively.



#### DS18B20

For temperature sensing, we've selected the DS18B20 sensor, complemented by the Dallas Temperature library for interfacing with the ESP32 microcontroller. This sensor is popular for its accuracy and simplicity, making it a reliable choice for our project. It operates using the One Wire protocol, allowing multiple sensors to be connected to a single digital pin on the ESP32 microcontroller. Additionally, the DS18B20 provides digital output. We use it along with a 4.7k Ohm Resistor.

A black wire with a white and red wire

Description automatically generated

#### 16x2 LCD with I2C

For our project's display needs, we're using a standard 16x2 character LCD screen. The official name for the 16x2 LCD is **HD44780.** To simplify the connection with the ESP32 microcontroller and reduce the number of required pins, we've opted for an LCD module equipped with an I2C interface. This interface allows for seamless communication between the ESP32 and the LCD screen using just a few wires.

A black and blue electronic device

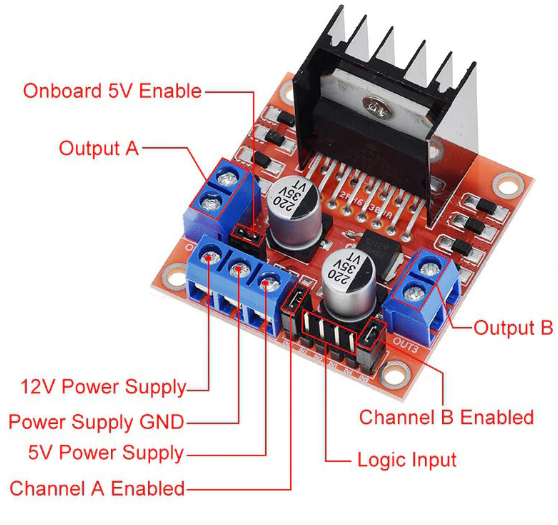
Description automatically generatedA computer screen with many different colored wires

Description automatically generated with medium confidence

With this setup, we can easily display temperature readings, system status, and other relevant information in real-time, enhancing the user experience and usability of our temperature control system.

#### L298 H Bridge Motor Driver

We're utilizing the L298 motor driver to control the fan and the bulb. The L298 is a dual H-bridge motor driver that allows us to drive both DC motors (for the fan) and other high-current loads like the bulb. By using analog values from the PID Controller, we can adjust the speed of the fan and the brightness of the bulb according to the temperature readings. This provides us with precise control over these components, allowing us to maintain optimal conditions in our temperature control setup.



#### 12 Volts DC Fan and Filament Bulb

We've opted for a 12V filament bulb and a DC fan to regulate temperature conditions. The 12V filament bulb provides a controllable heat source, allowing us to adjust its brightness to influence the ambient temperature. Meanwhile, the DC fan helps dissipate excess heat by increasing airflow when necessary. By combining these components, we can effectively manage temperature variations within our system.

A black fan with wires

Description automatically generatedA light bulb with a filament

Description automatically generated

## Software Components:

#### Arduino IDE

We're developing our project using the Arduino IDE, a popular integrated development environment (IDE) widely used for programming Arduino and compatible microcontroller boards. The Arduino IDE provides a user-friendly interface for writing, compiling, and uploading code to our ESP32 microcontroller.

In our project, we're leveraging several libraries to streamline development and enhance functionality:

1. **DallasTemperature.h**: Communicates with the DS18B20 temperature sensor.
2. **Wire.h**: Enables I2C communication for interfacing with devices like the LCD screen.
3. **WiFi.h**: Establishes a connection to wireless networks for remote monitoring and control.
4. **BlynkSimpleEsp32.h**: Integrates the Blynk IoT platform for smartphone app control.
5. **LiquidCrystal\_I2C.h**: Simplifies communication with I2C-enabled LCD displays.

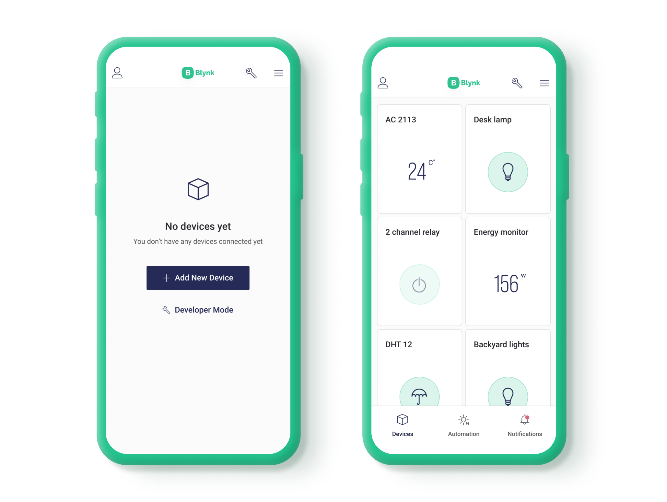
A blue and white logo

Description automatically generated

#### Blynk

Blynk is an IoT (Internet of Things) platform that simplifies the process of building connected projects. With Blynk, developers can create custom smartphone apps to monitor and control their hardware remotely.

In our project, we're incorporating the Blynk platform using the BlynkSimpleEsp32.h library. This allows us to establish a communication link between our ESP32 microcontroller and the Blynk cloud servers. Through the Blynk app, we can create a user interface with customizable widgets such as buttons, sliders, and graphs to interact with our temperature control system.

With Blynk, we can monitor temperature readings in real-time, adjust control parameters, and receive notifications/alerts directly on our smartphones. This remote control and monitoring capability adds flexibility and convenience to our project, enabling us to manage our temperature control system from anywhere with an internet connection.

A green and white logo

Description automatically generated

#### Python

We're harnessing the power of Python for data visualization. Using Python, specifically with libraries like Matplotlib and Pandas, we can generate insightful graphs and plots based on the temperature data collected by our system. By collecting data from our ESP32 microcontroller, including temperature readings and control outputs, we can analyze the effectiveness of the PID algorithm in maintaining the desired temperature setpoint.

A blue and white logo with a circular chart

Description automatically generated with medium confidence

## Block Diagram:

## 

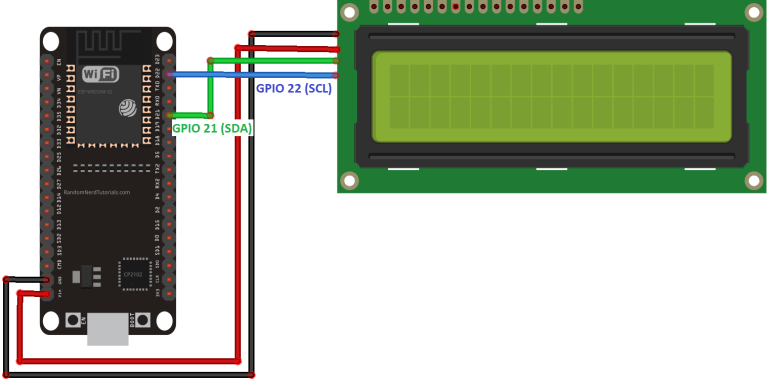
## Methodology:

### Hardware Configuration:

Set up the experimental environment with the necessary hardware components, including the ESP32 microcontroller, which is supplied with 5 volts using a laptop, temperature sensor (DS18B20), LCD display, motor driver (L298 H Bridge), fan, and filament bulb in the following way.

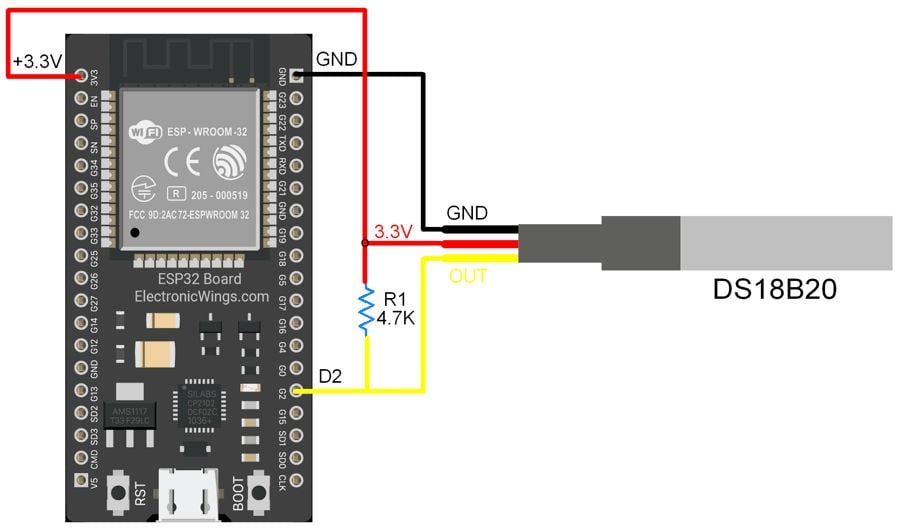
**LCD Interface**

* Connect the I2c IC with LCD.
* Connect the ground of the IC with the ground of esp32.
* Connect SCL terminal with GPIO22 and SDA terminal with GPIO21.
* Give the 5V supply to the VCC terminal of the IC using ESP32 Vin pin.



**Temperature Sensor (DS18B20) Interface**

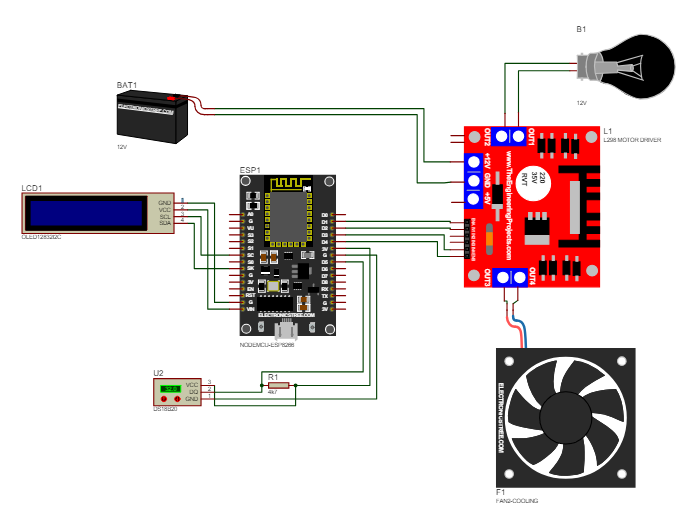
* Connect the ground and VCC of the sensor with the ESP32.
* The DQ pin is attached with GPIO 15 of ESP32.
* The resolution of the sensor is set to 12 for more variation.



**L298 Motor Driver Interface**

* A fan and filament bulb are connected to L298 to screw terminal blocks.
* Motor driver is powered with a 12 V power supply.
* Enable pins for the fan and bulb are connected to GPIO 12 and 13.
* The Enable A and Enable B pins are used for enabling and controlling the speed of the motor and brightness of the bulb.
* Using PWM signals from the ESP32 we can control the voltage supplied to our components.
* The ground of 12V volt supply is common with ESP32 ground.

## Schematic:

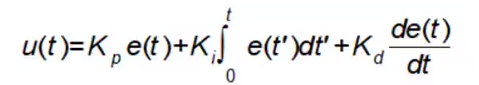


### PID Control Implementation:

A PID controller is a feedback control mechanism widely used in engineering and automation systems to regulate processes and maintain desired setpoints. It operates based on three main control actions: proportional, integral, and derivative, which are applied to the error signal between the desired setpoint and the actual process variable. We establish the controller using the ESP32.

Firstly, we read the current temperature from the ESP32 and subtract it from the setpoint to get the error value which is then fed into the PID algorithm to get an output which is a value from 0-255. This analog value is used to driver the bulb and fan.

We can express PID control mathematically with the following equation. P, I, and D are represented by the three terms that add together here. Kp, Ki, and Kd are constants that tune how the system reacts to each factor:



In many situations, it's expedient to plug in a dedicated PID controller to your process, but you can make your own with an Arduino or other similar dev board. You can even write our own PID routine. We expressed each term in our code in a manner like:

* P:instanteneousError = setpoint – input;
* I:cumulativeError = += error \* elapsed Time;
* D:rateOfError = (error – errorLastCalculation)/elapsed Time

### Integration with BLYNK:

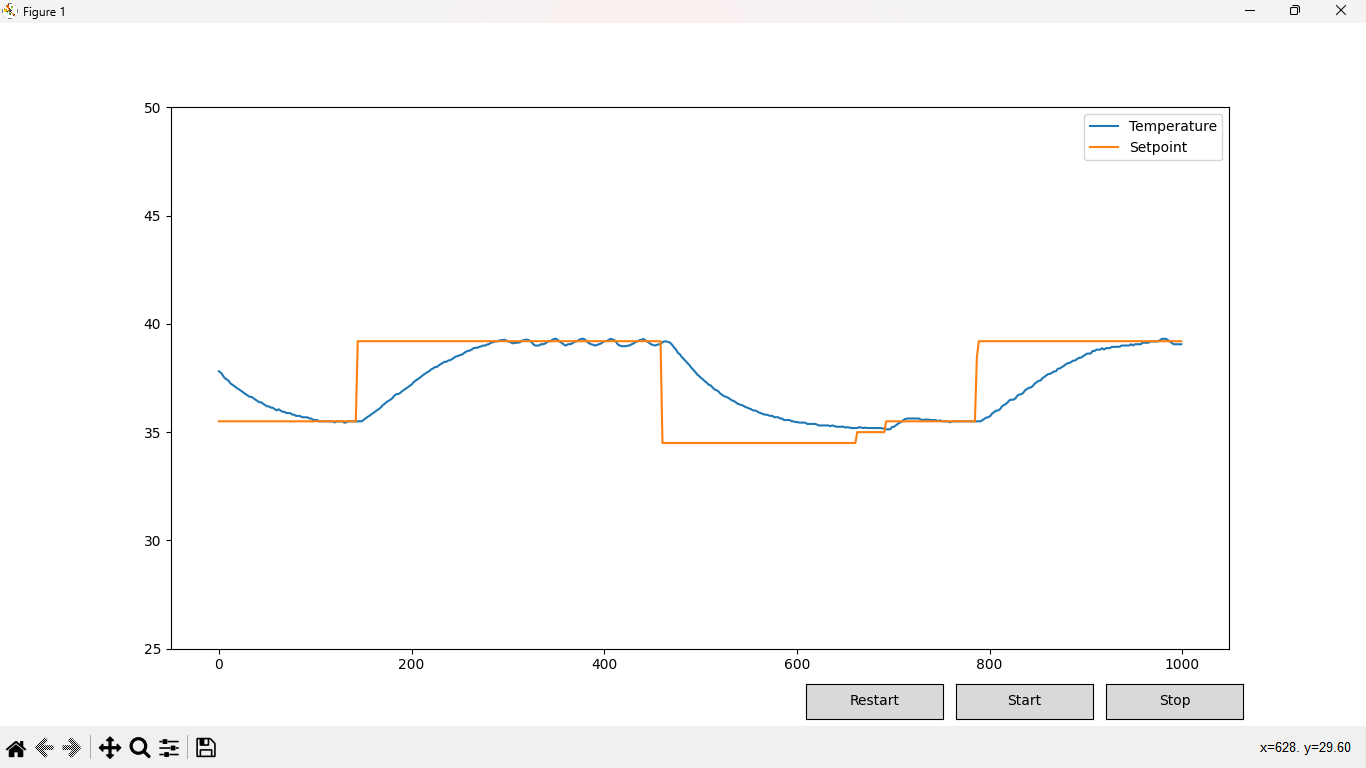
* ESP32 has a built in WIFI module which we can use to integrate IOT platform like Blynk.
* We add our BLYNK credentials to our code to establish the connection.
* A web and mobile dashboard is created for our requirements.
* We are using virtual pins for our data visualization like current temperature and output value we can change the value of threshold, KP, KI and KD using BLYNK sliders.

### Data analysis and Visualization:

* A python code is written using MATPLOT LIB to visualize our data in form of animated graphs.
* Python establishes communication with ESP using PORT 5 with baud rate of 9600.
* Button to stop, start and restart the graph are added using Python GUI libraries.

1. EXPERIMENTS AND RESULTS

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **CURRENT**  **TEMPERATURE** | **SET**  **TEMPERATURE** | **Kp** | **Ki** | **Kd** | **RISE**  **TIME** | **OVERSHOOT** | **STEADY**  **STATE**  **ERROR** |
| 25.86 | 30 | 10 | 10 | 10 |  |  |  |
| 29.23 | 25 | 10 | 60 | 10 |  |  |  |
| 34.33 | 40 | 60 | 10 | 10 |  |  |  |
| 23.45 | 20 | 10 | 10 | 60 |  |  |  |
| 27.43 | 35 | 30 | 60 | 90 |  |  |  |



1. CONCLUSION

We are pleasure to conclude our project on the topic “Temperature PID Controller”. The set temperature, Kp, Ki, Kd is defined by the user and the control unit continuously compares the current temperature followed by DS18B20 and set point and maintains the temperature inside chamber by varying the fan and heater respectively followed by PID controller. The PID control loop controls the transient response and steady state error of the system. By changing the value of Kp the rise time will reduce and it also reduces, but never eliminate, the steady-state error. Integral controller (Ki) decreases the rise time, increases both the overshoot and the settling time, and eliminates the steady-state error. A derivative control (Kd) will have the effect of increasing the stability of the system, reducing the overshoot, and improving the transient response.

The detail of this project has been made by both members of our team sincerely with the inspiration of out tutors. Through this project we have done, sensing, and maintaining the temperature using. DS18B20 which is very sensitive.

This module is ideal for interfacing with ESP 32. According to the program installed in the ESP32, it regulates or works based on the comparison of current temperature with set point. And the working of the system can be observed on the LCD and BLYNK.

1. REFERENCES

[1] J.G. Ziegler and Nichols, “Optimal Settings for Automatic Controllers”, Trans. ASME, vol. 64 P.P. 759- 768, 1942.

[2] “Introduction: PID Controller Design." Control Tutorials - Introduction:

PID Controller. “http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction§ion=ControlPID”.

[3} “National Instruments” <http://www.ni.com/white-paper/3782/en/>

1. FINAL HARDWARE
2. BLYNK DASHBOARD

A computer with a black screen

Description automatically generated

A screenshot of a computer

Description automatically generated

A black rectangular cellphone with blue trim

Description automatically generated

A screenshot of a phone

Description automatically generated

1. CODE

#define BLYNK\_TEMPLATE\_ID "TMPL6CVf0NrSH"

#define BLYNK\_TEMPLATE\_NAME "Temperature Controller"

#define BLYNK\_AUTH\_TOKEN "IZ8jXtnXLysYnoM2gsY1BCbbK0wmUcsZ"

#define BLYNK\_PRINT Serial

#include <DallasTemperature.h>

#include <Wire.h>

#include <WiFi.h>

#include <BlynkSimpleEsp32.h>

#include <LiquidCrystal\_I2C.h>

#include <PID\_v1.h>

LiquidCrystal\_I2C lcd(0x27, 16, 2);

#define ONE\_WIRE\_BUS 15

OneWire oneWire(ONE\_WIRE\_BUS);

DallasTemperature sensors(&oneWire);

char ssid[] = "PTCL.3154";

char pass[] = "786786786";

double start,Setpoint;     //v1

double temp=0;     //v2

double Kp=1;        //v3

double Ki=0;        //v4

double Kd=0;       //v5

int heaterEnable = 12;  //enable  heater

int fanEnable = 13;  //enable  fan

double Input, Output1,Output2;

double dt, last\_time;

double integral, previous = 0;

BlynkTimer timer;

BLYNK\_WRITE(V0)

{

start = param.asInt();

}

BLYNK\_WRITE(V1)

{

   Setpoint = param.asDouble();

}

BLYNK\_WRITE(V3)

{

   Kp = param.asDouble();

}

BLYNK\_WRITE(V4)

{

   Ki = param.asDouble();

}

BLYNK\_WRITE(V5)

{

   Kd = param.asDouble();

}

void setup() {

  last\_time = millis();;

  previous = 0;

  integral = 0;

  Serial.begin(9600);

   pinMode(heaterEnable, OUTPUT);

   pinMode(26, OUTPUT);

   pinMode(fanEnable, OUTPUT);

   pinMode(27, OUTPUT);

  lcd.init();

  lcd.backlight();

  lcd.setCursor(0, 0);

  Blynk.begin(BLYNK\_AUTH\_TOKEN, ssid, pass);

   while (Blynk.connect() == false) {

  }

  digitalWrite(26, HIGH);

  digitalWrite(27, HIGH);

   sensors.begin();

   sensors.setResolution(12);

   timer.setInterval(1000L, readTemperature);

}

void loop() {

  Blynk.run();

  timer.run();

}

void readTemperature() {

  sensors.requestTemperatures();

  temp=sensors.getTempCByIndex(0);

  Input=temp;

  Blynk.virtualWrite(V2, temp);

  double now = millis();

  dt = (now - last\_time)/10000.00;

  last\_time = now;

  double error1 = Setpoint - Input;

  double error2 = Input - Setpoint;

   if(Input > Setpoint){

        Output2 =  pid(error2);

        analogWrite(fanEnable,Output2);

        analogWrite(heaterEnable,LOW);

        Serial.println(Output2);

   }

   else{

        Output1 =  pid(error1);

        analogWrite(heaterEnable,Output1);

        analogWrite(fanEnable,LOW);

       Serial.println(Output1);}

 }

//------------------------------Priniting and Plotting-----------------------------//

  Serial.print(temp);

  Serial.print(",");

  Serial.print(Setpoint);

  Serial.println();

  lcd.setCursor(0, 0);

  lcd.print("T:");

  lcd.print(temp);

  lcd.print(" P:");

  lcd.print(Kp);

  lcd.setCursor(0, 1);

  lcd.print("I:");

  lcd.print(Ki);

  lcd.print(" D:");

  lcd.print(Kd);

}

 Serial.print(temp);

  Serial.print(",");

  Serial.print(Setpoint);

  Serial.println();

  lcd.setCursor(0, 0);

  lcd.print("T:");

  lcd.print(temp);

  lcd.print(" P:");

  lcd.print(Kp);

  lcd.setCursor(0, 1);

  lcd.print("I:");

  lcd.print(Ki);

  lcd.print(" D:");

  lcd.print(Kd);

}

double pid(double error)

{

  double proportional = error;

   if (Input == Setpoint || Ki==0) {

    integral = 0;

  } else {

    integral += error \* dt;

    integral = constrain(integral, 0, 10);

  }

  Serial.print("KI");

  Serial.println(integral);

  double derivative = (error - previous) / dt;

  previous = error;

  double output = (Kp \* proportional) + (Ki \* integral) + (Kd \* derivative);

  output = constrain(output, 0, 255);

  return output;

}

#### Python Code:

*import* time

*import* serial

*import* matplotlib.pyplot *as* plt

*import* matplotlib.animation *as* animation

*import* numpy *as* np

*from* scipy.interpolate *import* interp1d

*from* matplotlib.widgets *import* Button

def *animate*(i, temperatures, setpoints, ser, ax):

*if* not ser.is\_open:

*return*

    line = ser.readline().decode('latin1').strip().rstrip('\r\n')

    data = line.split(',')

*if* len(data) == 2:

*try*:

            temperature, setpoint = map(float, data)

*if* 0 <= temperature <= 50 and 0 <= setpoint <= 50:

                temperatures.append(temperature)

                setpoints.append(setpoint)

*except* ValueError:

*pass*

    setpoints = setpoints[-1000:]

    temperatures = temperatures[-1000:]

    # *Interpolate data*

*if* len(temperatures) > 1:

        x = range(len(temperatures))

        f\_temp = interp1d(x, temperatures, kind='linear')

        f\_setpoint = interp1d(x, setpoints, kind='linear')

        x\_new = np.linspace(0, len(temperatures) - 1, 500)

        ax.clear()

        ax.plot(x\_new, f\_temp(x\_new), label='Temperature')

        ax.plot(x\_new, f\_setpoint(x\_new), label='Setpoint')

        ax.set\_ylim([25, 50])

        ax.legend()  # *Add legend here*

def *start\_animation*(event):

*global* ani

    ani.event\_source.start()

*global* ani

    ani.event\_source.stop()

def *stop\_animation*(event):

*global* ani

    ani.event\_source.stop()

def *restart*(event):

*global* temperatures, setpoints

    temperatures = []

    setpoints = []

    ax.clear()

temperatures = []

setpoints = []

fig, ax = plt.subplots()

ax.set\_ylim([20, 60])

ser = serial.Serial("COM5", 9600)

time.sleep(2)

ani = animation.FuncAnimation(fig, animate, fargs=(temperatures, setpoints, ser, ax), interval=100, save\_count=500)

# *Add start, stop, and restart buttons*

ax\_start = plt.axes([0.7, 0.01, 0.1, 0.05])

ax\_stop = plt.axes([0.81, 0.01, 0.1, 0.05])

ax\_restart = plt.axes([0.59, 0.01, 0.1, 0.05])

btn\_start = Button(ax\_start, 'Start')

btn\_stop = Button(ax\_stop, 'Stop')

btn\_restart = Button(ax\_restart, 'Restart')

btn\_start.on\_clicked(start\_animation)

btn\_stop.on\_clicked(stop\_animation)

btn\_restart.on\_clicked(restart)

plt.show()

ser.close()