Systemy Mikroprocesorowe
LABORATORIUM

PROJEKT: MIKROPROCESOROWY SYSTEM STEROWANIA I POMIARU. REGULATOR PID

Przygotowali:
Miłosz Wiertel, 151139
Kamil Szady, 151098

Poznań 2024

Spis treści

1 Wstęp

W dzisiejszym świecie rozwój systemów automatyki i sterowania odgrywa kluczową rolę w optymalizacji procesów, poprawie efektywności energetycznej oraz zapewnieniu stabilności różnych obiektów lub procesów technologicznych.

Zamiarem niniejszego projektu było zrealizowanie regulatora temperatury opartego na mikrokontrolerze STM32 osadzonego na płytce prototypowej Nucleo F746ZG, który kontroluje przepływ prądu przez rezystor ceramiczny w celu osiągnięcia i utrzymania zadanej temperatury. Cel projektu nie tylko skupia się na zbudowaniu funkcjonalnego regulatora temperatury, ale także na zrozumieniu procesu projektowania systemów regulacyjnych opartych na mikrokontrolerach oraz metodach wizualizacji i wymiany danych.

Projekt ten ma na celu podsumowanie, wykorzystanie i utrwalenie wiedzy zdobytej podczas zajęć laboratoryjnych. Przedsięwzięcie pozwala na przykładowe zastosowanie zaawansowanych technologii mikrokontrolerowych do skonstruowania systemu regulacji temperatury w oparciu o regulator PID, który może znaleźć zastosowanie w różnych dziedzinach, takich jak przemysł, nauka czy też automatyka domowa.

W kolejnych sekcjach tego sprawozdania szczegółowo przedstawiono etapy konstrukcji oraz implementacji tego regulatora temperatury, omówiono zastosowane komponenty elektroniczne, programowanie mikrokontrolera oraz przeprowadzono analizę wyników uzyskanych podczas testów i eksperymentów.

2 Model

Jak już wspomniano we wstępie, obiektem ogrzewanym będzie rezystor ceramiczny, który w w tym konkretnym przypadku ma 47Ω oraz pozwala na wydzielenie mocy $5W$. Z fizycznego punktu widzenia, rezystor możemy zamodelować jako obiekt inercyjny pierwszego rzędu z opóźnieniem transportowym.

$$G_R = \frac{k}{sT + 1} \cdot e^{-sT_d}, \tag{1}$$

gdzie:

- k - wzmocnienie
- T - stała czasowa
- T_d - opóźnienie transportowe

3 Realizacja fizyczna układu

3.1 Założenia projektowe

Mikrokontroler odpowiedzialny jest za realizację algorytmu PID (obliczenia) oraz komunikację: odbiór danych konfiguracyjnych z komputera przez interfejs UART (nastawy dla regulatora PID, temperatura zadana), przesyłanie danych przez interfejs UART do komputera w celu wizualizacji zmian temperatury w czasie (realizacja z wykorzystaniem biblioteki matplotlib języka Python), odczyt aktualnej temperatury poprzez interfejs SPI z czujnika BMP280 oraz wyświetlanie temperatury zadanaje i aktualnej na wyświetlaczu LCD 2X16 wyposażonym w konwerter I2C LCM1602. Elementem wykonawczym jest tranzystor bipolarny BD135-16, który pracuje jako klucz w wyniku podawania na jego bazę sygnału PWM z mikrokontrolera. Do zasilania rezystora wykorzystano zewnętrzny zasilacz impulsowy $12V/2.5A$.

3.2 Spis elementów

- płytka prototypowa Nucleo F746ZG
- rezystor ceramiczny $47\Omega, 5W$
- tranzystor bipolarny BD135-16
- wyświetlacz LCD 2x16 + konwerter I2C LCM1602
- czujnik temperatury BMP280 I2C/SPI
- rezystor $330\Omega, 0.25W$
- zasilacz impulsowy 12V/2.5A - 100V-240V - wtyk DC 5.5/2.5mm
- moduł do płytek stykowych z gniazdem DC 5.5x2.1mm
- płytka stykowa
- przewody łączeniowe

3.3 Niezbędne obliczenia

Element wykonawczy stanowi tranzystor bipolarny BD135-16, który pełni rolę klucza umożliwiając przepływ prądu z zasilacza impulsowego, przez rezystor, złącze kolektor-emiter to masy. W tego typu zastosowaniach tranzystor w momencie przewodzenia (stan wysoki na bazie pochodzący ze sygnału PWM wystawiany przez mikrokontroler zgodnie z algorytmem regualcji PID) powinien znajdować się w stanie nasycenia w celu ograniczenia do minimum spadku napiecia na złączu kolektor-emiter tranzystora, co prowadzi do minimalizacj strat mocy wydzielanych na tranzystorze. Stan nasycenia tranzystora uzyskuje

się w wyniku założenia znacznie mniejszego wzmocnienia prądowego niż wynika z noty katalogowej. Najpierw należy obliczyć prąd płynacy przez kolektor tranzystora (jest on równy prądowi rezystora):

$$I_C = I_R = \frac{U_{ZAS}}{R} = \frac{12V}{47\Omega} = 0.255A \tag{2}$$

Następnie korzystajac ze wzoru na wzmocnienie tranzystora bipolarnego:

$$\beta = \frac{I_C}{I_B} \tag{3}$$

oraz przyjmując wzmocninie prądowe dla nasycenia (dla tranzystora BD135-16 można przyjać $\beta_s = 30$) , należy obliczyć oczekiwany orąd bazy:

$$I_B = \frac{I_C}{\beta_s} = 0.0085A \tag{4}$$

Na podstawie oczekiwanego prądu bazy należy dobrać rezystor ograniczający prąd bazy na podstawie zależności:

$$R_B = \frac{U_{ZAS} - U_{BE}}{I_B}, \tag{5}$$

gdzie:

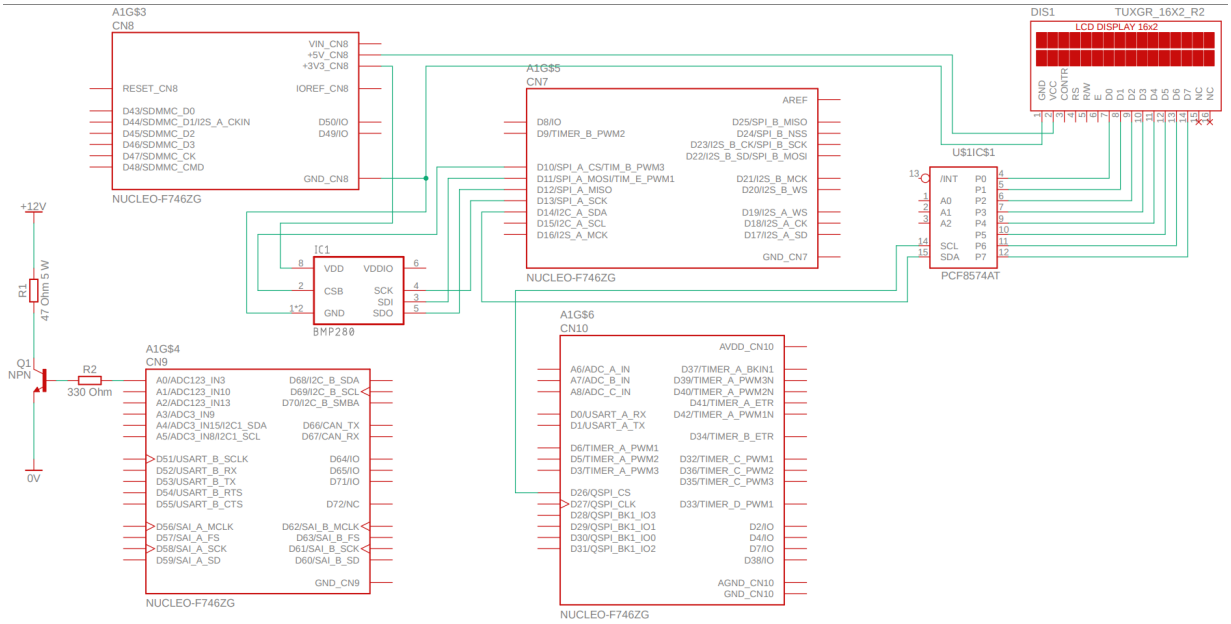
- $U_{ZAS} = 3.3V$ - napięcie wyjściowe z pinu mikrokontrolera realizującego sterowanie PWM
- $U_{BE} = 0.7V$ - spadek napięcia na złączu baza-emiter tranzystora bipolarnego

Zatem rezystor ograniczającey prąd bazy powinien mieć wartosć:

$$R_B = \frac{3.3 - 0.7}{0.0085} = 305.88\Omega, \tag{6}$$

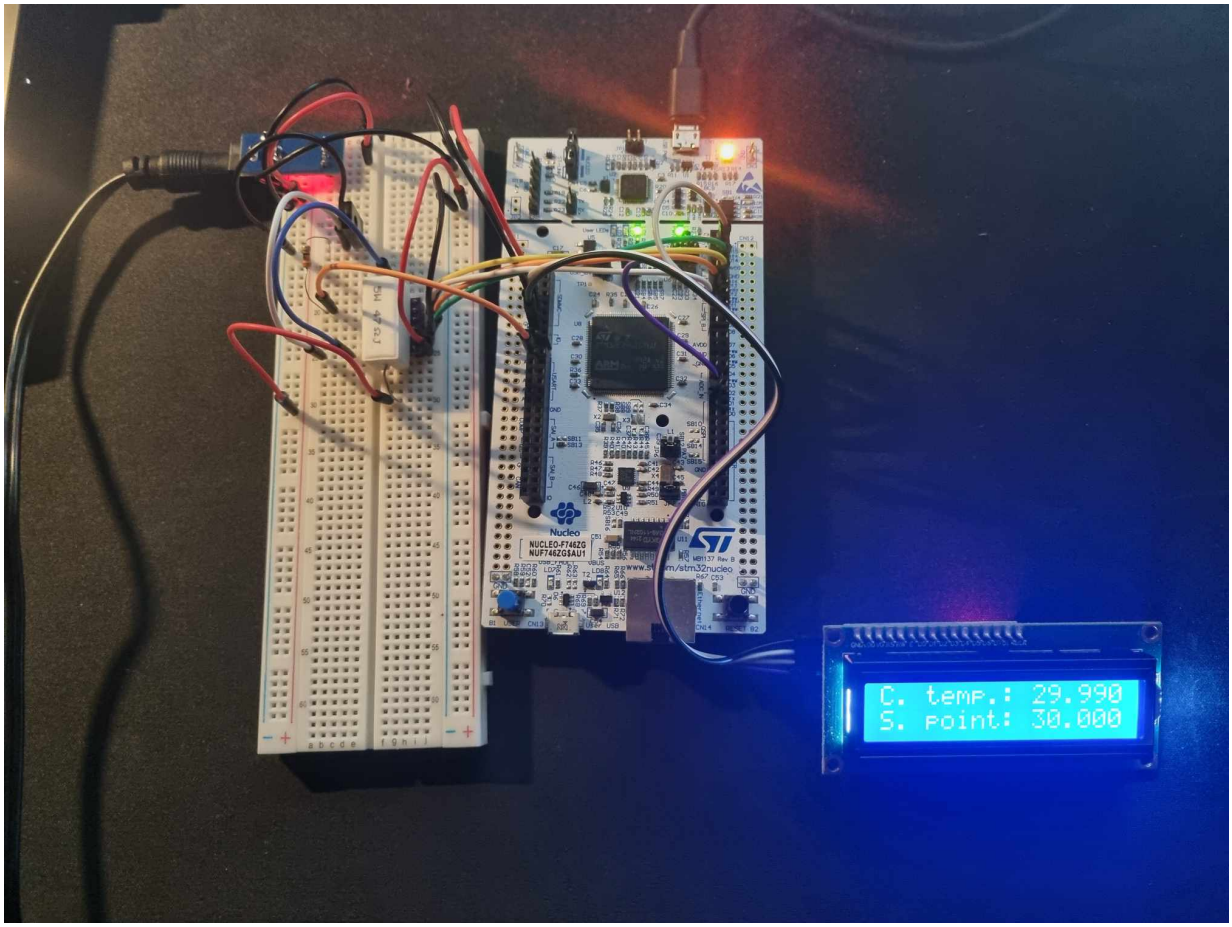
Biorąc pod uwagę szereg wartosci rezystorów E24 wybrano rezystor o rezystancji 330Ω.

3.4 Schemat układu



Rysunek 1: Schemat elektryczny układu

3.5 Fizyczna realizacja układu



Rysunek 2: Fizyczne wykonanie układu.

4 Realizacja algorytmu PID

4.1 Dobór nastaw

Jak wspomniano we wstępie niniejszego opracowania, do regualcji temperatury wykorzystano regualtor PID zaimplementowany na mikrokontrolerze STM32. Wstępnego doboru nastaw dokonano na podstawie odpowiedzi skokowej obiektu rzeczywistego oraz modelu obiektu w programie Matlab. Następnie nastawy poddano korekcji w celu skompensowania niedokładności modelowania i uwzględnienia rzeczywistych niedoskonałosci obiektu (np. wynikające ze strat ciepła w szczelinie pomiędzy rezystorem a sensorem). Ostatecznie wybrano nastawy:

- $K_P = 0.5$
- $K_I = 0.001$
- $K_D = 0.0$

4.2 Kod realizujący działanie regualtora PID

Listing 1: Import niezbędnych bibliotek

```
/* USER CODE BEGIN Includes */
#include "BMPXX80.h"
#include "i2c-lcd.h"
#include <stdio.h>
#include <string.h>
/* USER CODE END Includes */
```

Listing 2: Definicja typów danych dla regulatora PID

```
/* USER CODE BEGIN PTD */
typedef float float32_t;

typedef struct{
    float32_t Kp;
    float32_t Ki;
    float32_t Kd;
    float32_t dt;
}pid_parameters_t;

typedef struct{
    pid_parameters_t p;
    float32_t prev_error, prev_int;
}pid_t;
/* USER CODE END PTD */
```

Listing 3: Deklaracja zmiennych prywatnych

```
/* USER CODE BEGIN PV */
float temperature;
float set_point = 30.0;
float duty;
char lcd_temp[16];
char lcd_set_point[16];
uint8_t Received;
uint8_t uart_temp[64];
/* USER CODE END PV */
```

Listing 4: Funkcja realizująca algorytm regulatora PID

```
/* USER CODE BEGIN 0 */
float32_t calculate_discrete_pid(pid_t* pid, float32_t setpoint, float32_t
    measured){
    float32_t u=0, P, I, D, error, integral, derivative;

    error = setpoint - measured;

    P = pid->p.Kp * error;

    integral = pid->prev_int + (error+pid->prev_error);
    pid->prev_int = integral;
    I = pid->p.Ki * integral * (pid->p.dt/2.0);

    derivative = (error - pid->prev_error)/pid->p.dt;
    pid->prev_error = error;
    D = pid->p.Kd * derivative;

    u = P + I + D;

    return u;
}
```

Listing 5: Definicja regulatora o zadanych nastawach

```
pid_t pid = {.p.Kp = 0.5, .p.Ki = 0.001, .p.Kd = 0.0, .p.dt=1 , .
   prev_error = 0, .prev_int = 0};
```

Listing 6: Inicjalizacja czujnika, wyświetlacza, generatora PWM, generatora podstawy czasu oraz odbioru UART w trybie przerwań

```
  /* USER CODE BEGIN 2 */
  BMP280_Init(&hspi1, BMP280_TEMPERATURE_16BIT, BMP280_STANDARD,
     BMP280_FORCEDMODE);
  lcd_init();
  lcd_clear();
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4);
  HAL_TIM_Base_Start_IT(&htim7);
  HAL_UART_Receive_IT(&huart3, &Received, 2);
  /* USER CODE END 2 */
```

Listing 7: Odczyt UART z wykorzystaniem przerwań wraz z sygnalizacją błędnej wartości

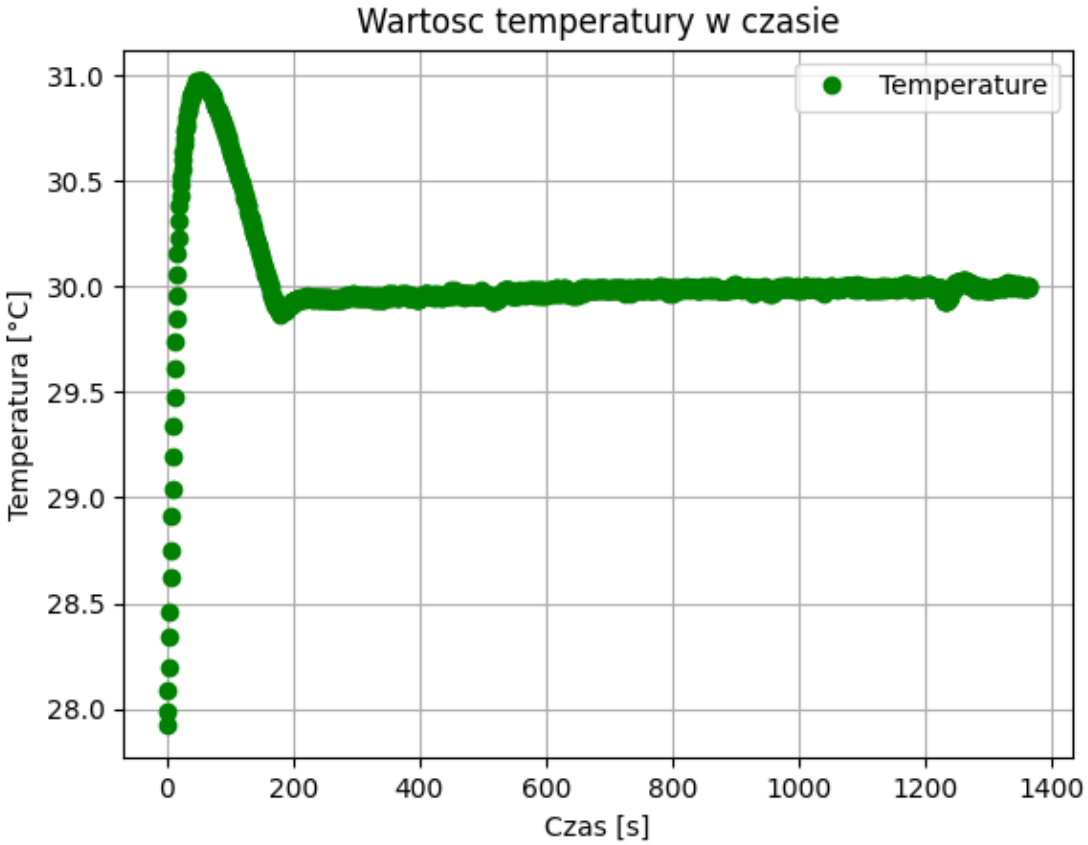
```
/* USER CODE BEGIN 4 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(is_int(&Received))
    {
        set_point = atoi(&Received)/10.0;
        HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
    }
    else
        HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
    HAL_UART_Receive_IT(&huart3, &Received, 2);
}
```

Listing 8: Sygnalizacja działania, odczyt temperatury, obliczenie wypełnienia dla sygnału PWM, wysterowanie generatora PWM, obsługa wyświetlacza, wysyłanie wartości temperatury przez UART

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    HAL_GPIO_TogglePin(LD1_GPIO_Port, LD1_Pin);
    temperature = BMP280_ReadTemperature();

    float pwm_duty_f = (999.0*calculate_discrete_pid(&pid, set_point,
       temperature));
    uint16_t pwm_duty = 0;
    if(pwm_duty_f<0)
        pwm_duty = 0;
    else if(pwm_duty_f>999.0)
        pwm_duty = 999;
    else
        pwm_duty = (uint16_t)pwm_duty_f;
    duty = pwm_duty;
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, pwm_duty);
    sprintf(lcd_temp, "C. temp.: %f", temperature);
    sprintf(lcd_set_point, "S. point: %f", set_point);
    lcd_put_cur(0, 0);
    lcd_send_string(lcd_temp);
    lcd_put_cur(1, 0);
    lcd_send_string(lcd_set_point);
    sprintf((char*)uart_temp, "\n%f", temperature);
    HAL_UART_Transmit(&huart3, uart_temp, strlen((char*)uart_temp), 50);
}
/* USER CODE END 4 */
```

4.3 Działanie zbudowanego i zaprogramowanego układu sterowania



Rysunek 3: Przykładowy efekt regulacji temperatury przy wartości zadanej 30°C

Jak można zauważyć na rysunku 3, wystąpiło przeregulowanie o 1°C. Ze względu na inercyjny charakter obiektu grzejnego, trudno jest w pełni wyeliminować przeregulowanie w procesie regulacji, zwłaszcza dążąc do możliwie krótkiego czasu narastania. Głównym kryterium przy doborze nastaw regulatora było natomiast zminimalizowanie wartości uchybu w stanie ustalonym, które udało się uzyskać dla dobranych nastaw.

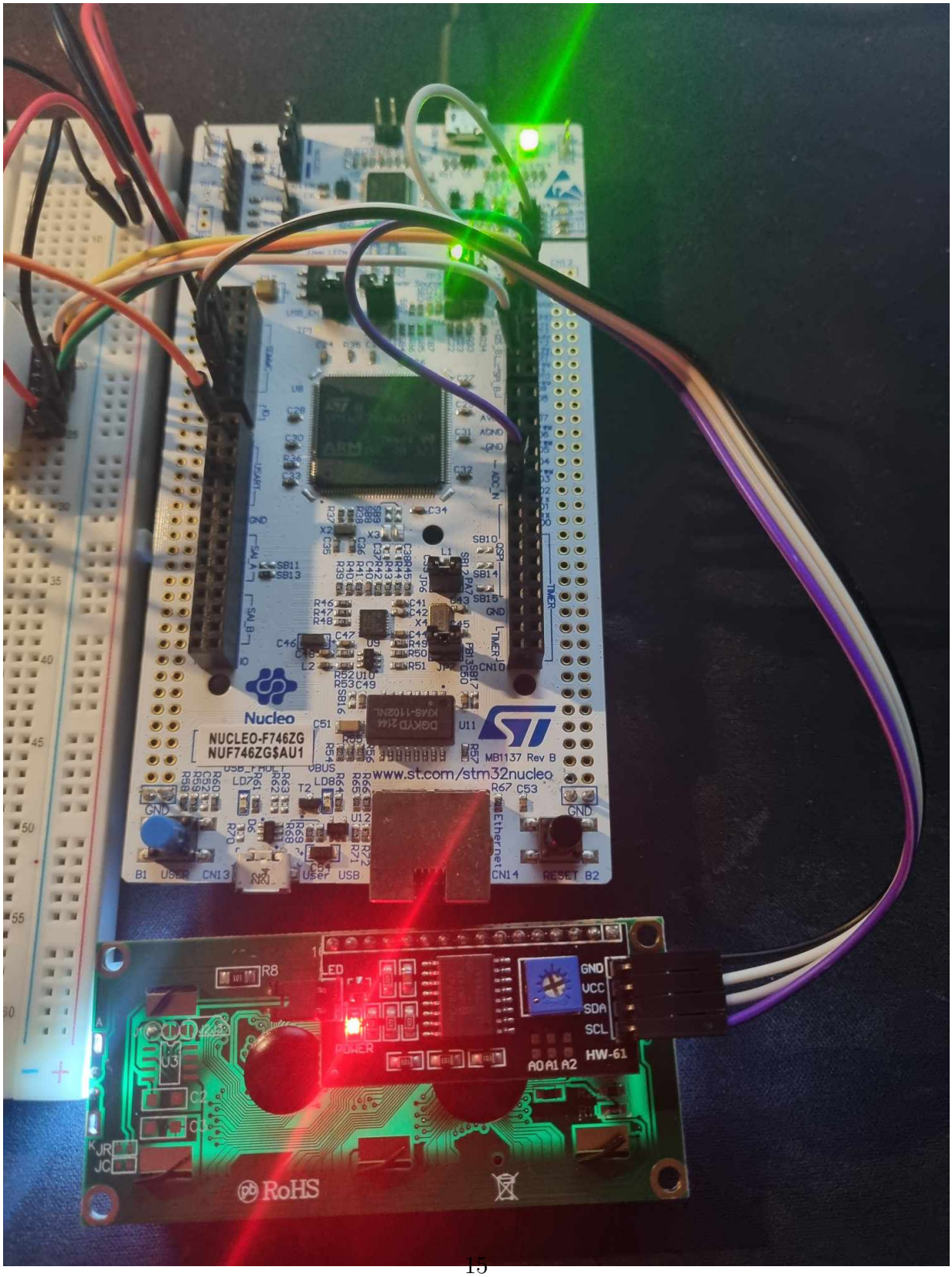
5 Wizualizacja

Zgodnie z wymogami projektu, w celu zdobycia dodatkowych punktów należało rozszerzyć projekt o dodatkowe elementy wejsciowe/wyjsciowe oraz wizualizację w czasie rzeczywistym. Ta realizowana jest z wykorzystaniem wyświetlacza LCD oraz skryptu języka Python rysującego przebieg temperatury w czasie rzeczywistym przy użyciu protokołu komunikacyjnego UART.

5.1 Wyświetlacz LCD

W celu bieżącego monitorowania temperatury rezystora rozbudowano układ o wyświetlacz LCD 2X16 z konwerterem I2C LCM1602. Wykorzystanie konwertera umożliwiło ograniczenie ilości przewodów połączeniowych pomiędzy mikrokontrolrem a wyświetlaczem co zwiększyło przejrzystość i niezawodność po stronie realizacji elektrycznej. Wyświetlacz posiada dwie linie po szesnaście znaków, dzięki czemu jednocześnie zrealizowano podgląd temperatury zadanej i temperatury aktualnej.

5.1.1 Realizacja połączeń elektrycznych



Rysunek 4: Podłączenie wyświetlacza LCD z użyciem konwertera I^2C

5.1.2 Fragmenty kodu odpowiedzialne za obsługę wyświetlacza

Listing 9: Fragment kodu odpowiedzialny za inicjalizację i wyczyszczenie wyświetlacza

```
lcd_init();
lcd_clear();
```

Listing 10: Fragment kodu odpowiedzialny za aktualizację danych prezentowanych na wyświetlaczu

```
    sprintf(lcd_temp, "C. temp.: %f", temperature);
    sprintf(lcd_set_point, "S. point: %f", set_point);
    lcd_put_cur(0, 0);
    lcd_send_string(lcd_temp);
    lcd_put_cur(1, 0);
    lcd_send_string(lcd_set_point);
```

5.2 Wizualizacja w czasie rzeczywistym

Podgląd temperatury chwilowej na wyświetlaczu nie pozwala na obserwację całego horyzontu czasowego pracy układu co uniemożliwa chociażby detekcję zaburzeń funkcjonowania całego obiektu (w rzeczywistych maszynach np. zacinanie lub wydłużenie się czasów otwierania czy zamykania serwozaworów świadczących o ich awarii). Z tego względu niekiedy konieczna jest ciagła wizualizacja wartości parametrów umożliwiająca wnikliwą ich analizę.

5.2.1 Fragmenty kodu odpowiedzialne za obsługę wizualizacji

Listing 11: Skrypt wysyłający przez UART wartość zadaną, odbierający bieżące wartości temperatury i rysujący wykres temperatury w czasie rzeczywistym

```
import serial
import matplotlib.pyplot as plt
from drawnow import drawnow
import time
import numpy as np
from datetime import datetime

# Inicjalizacja portu szeregowego
ser = serial.Serial('COM4', baudrate=115200, timeout=1)
ser.write(b'30')
time.sleep(0.5)

# Inicjalizacja danych
temperature_data = []

# Inicjalizacja pliku do zapisu danych
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
filename = f"temperature_data_{timestamp}.txt"

# Funkcja do aktualizacji wykresu
def update_plot():
    plt.plot(temperature_data, 'go', label='Temperature')
    plt.title('Wartosc temperatury w czasie')
    plt.xlabel('Czas [s]')
    plt.ylabel('Temperatura [C]')
    plt.legend()
    plt.grid()

    # Dodaj zapis do pliku
    with open(filename, 'a') as file:
        for i, temp in enumerate(temperature_data):
```

```
            file.write(f"{i+1}\t{temp}\n")

# Gwna ptla programu
while True:
    try:
        # Odczyt danych z portu szeregowego
        line = ser.readline().decode('utf-8').strip()

        # Pominicie pustej linii
        if not line:
            continue

        # Konwersja na float i dodanie do listy danych
        temperature = float(line)
        temperature_data.append(temperature)

        # Rysowanie wykresu w czasie rzeczywistym
        drawnow(update_plot)

    except KeyboardInterrupt:
        # Zatrzymaj ptl w przypadku przerwania klawiszem

        # Dodaj zapis ostatnich danych do pliku przed zakoczeniem
            programu
        with open(filename, 'a') as file:
            for i, temp in enumerate(temperature_data):
                file.write(f"{i+1}\t{temp}\n")

        break
    except ValueError:
        # Obsuga bdu konwersji na float (jeli otrzymamy niepoprawn
            warto)
        print("Error: Could not convert to float:", line)

# Zamknicie poczenia z portem szeregowym
ser.close()
```

5.2.2 Efekt końcowy

Rysunek 5: Efekt regulacji w stanie ustalonym dla wartości zadanej temperatury 30°C

6 Github

Link do repozytorium Github: `https://github.com/Mily1333/SM_Project`

Spis rysunków