SPRAWOZDANIE Z LABORATORIUM		Rok akademicki **2021/22**
Przedmiot: **SYSTEMY MIKROPROCESOROWE**		
Temat projektu: **Projekt regulatora PID – regulacja temperatury rezystora**		Ostateczny termin złożenia: **31.01.2021**
Wydział, kierunek, semestr, grupa: **WARiE, AiR, sem. 5, Gr. A1/L2**	Imię i Nazwisko: **1. Jakub Grzesiak**	Punkty:
Data złożenia ćwiczenia: **31.01.2022**		

1. Cel projektu

Celem projektu było zbudowanie i przetestowanie demonstracyjnego systemu sterowania i pomiaru temperatury obiektu cieplnego (rezystora) w oparciu o mikrokontroler z rodziny STM32. Zadaniem zaimplementowanego regulatora PID była regulacja temperatury z uchybem do 1% wartości zakresu regulacji. Do realizacji celu użyto płytkę ewaluacyjną NUCLEO-STM32F767ZI. Do symulacji i sprawdzania pomiarów posłużył Matlab i Python.

2. Spis użytych elementów

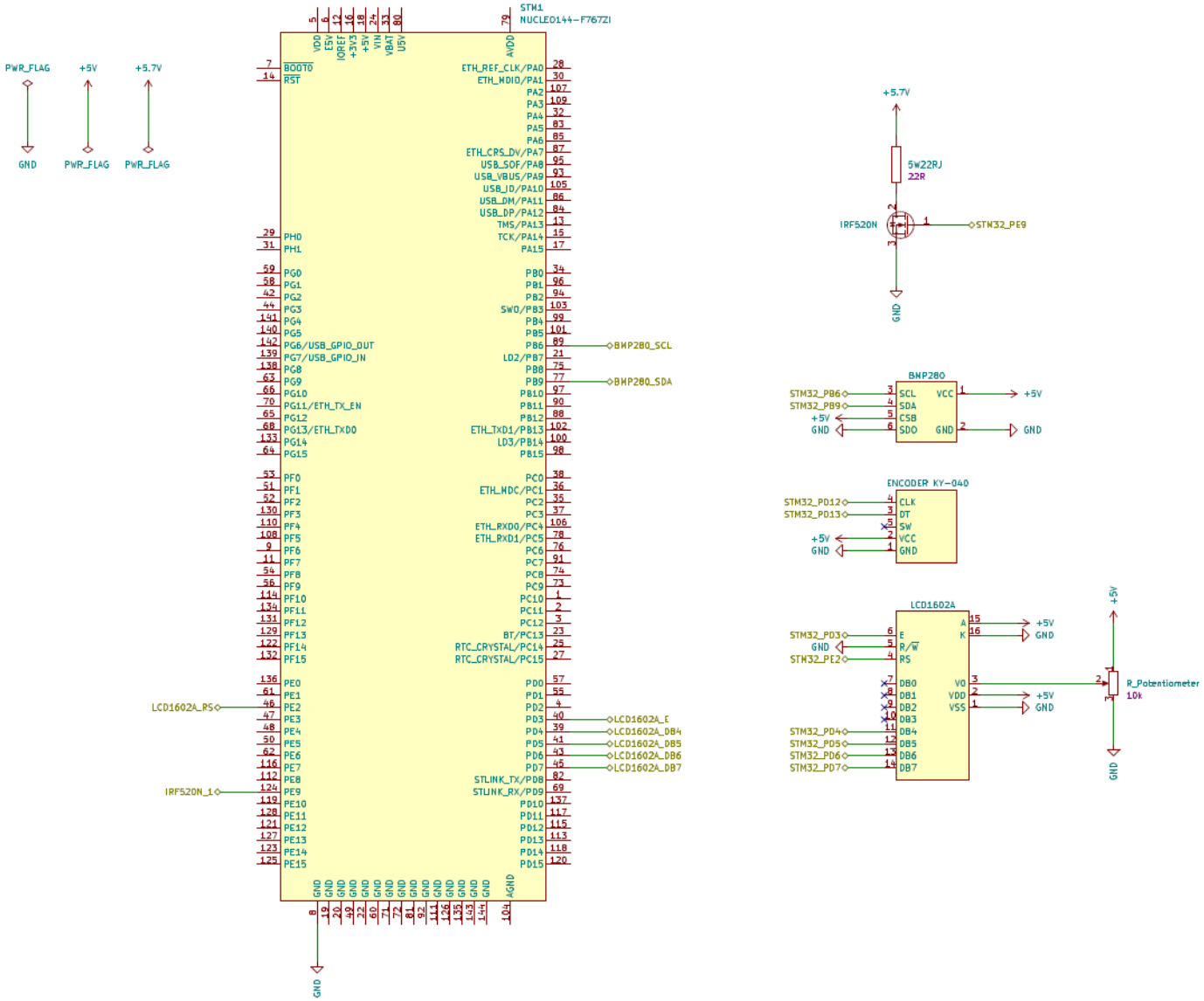
- Płytka ewaluacyjna NUCLEO-STM32F767ZI
- Rezystor ceramiczny 22Ω, 5W
- Zasilacz sieciowy 5,7V/800mA DC
- Moduł zasilania MB102
- Tranzystor MOSFET IRF520N
- Czujnik temperatury BMP280
- Enkoder obrotowy
- Wyświetlacz LCD 1602A

3. Funkcjonalności projektu

- Zadawanie temperatury za pomocą komunikacji szeregowej UART lub enkodera (domyślna temperatura startowa to 20℃, max. 65℃)
- Podgląd aktualnej wartości mierzonej za pomocą komunikacji szeregowej UART
- Wyświetlanie zadawanej oraz aktualnej temperatury na wyświetlaczu LCD
- Skrypt Python do logowania danych na żywo i graficznego przedstawiania sygnałów pomiarowych
- Układ automatycznej regulacji sterowany za pomocą programowo zaprojektowanego regulatora PID
- System kontroli wersji GitHub

4. Schemat elektroniczny

Schemat elektroniczny wykonano w programie KiCAD. Plik PDF schematu znajduje się w głównym folderze z dokumentacją.



Rysunek 1. Schemat elektroniczny projektu

5. Skrypt Python do logowania danych na żywo

Skrypt posiada następujące funkcjonalności:

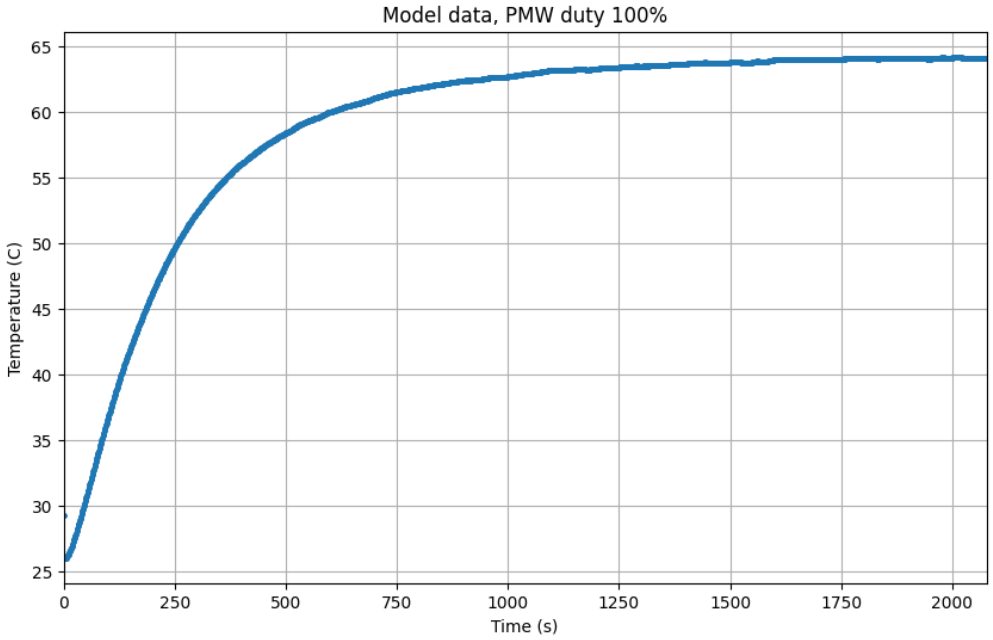
- Pozwala na zadanie żądanej wartości temperatury
- Odczytuje aktualną oraz zadaną wartość temperatury przesyłaną przez port szeregowy COM z mikroprocesora
- Zapisuje czas od uruchomienia, aktualną wartość temperatury z czujnika oraz wartość zadaną temperatury do pliku .csv
- Wykreśla graficzny przebieg temperatury mierzonej przez czujnik BMP280 na żywo

Przy starcie skryptu należy podać numer portu szeregowego COM z którego pobierane będą dane oraz zadać temperaturę jaką ma osiągnąć obiekt (zakres od 20 do 65℃). Do działania

skryptu wymagane jest posiadanie bibliotek w nim użytych – gdy nie są zainstalowane można użyć narzędzia pip install. Do wczytywania danych i wysyłania komend przez port szeregowy użyto biblioteki pyserial. Aby zakończyć pracę programu należy zamknąć okno wykresu ‘krzyżykiem’ bądź nacisnąć dowolny klawisz gdy aktywne jest okno figure. Następnie wykres wyłaczy się, a dane oraz wykres zostaną zapisane. Listing kodu został przedstawiony na końcu dokumentacji.

6. Model Matlab/Simulink i analiza obiektu oraz dobór nastaw regulatora

Do wykonania modelu obiektu wykorzystano dane pomiarowe zebrane przy wymuszeniu wypełnieniem 100%, czyli pełną wartością napięcia zasilania $\sim 5{,}7V$. Poniżej przedstawiono przebieg narastania temperatury rezystora w czasie wykreślony przez skrypt Python:

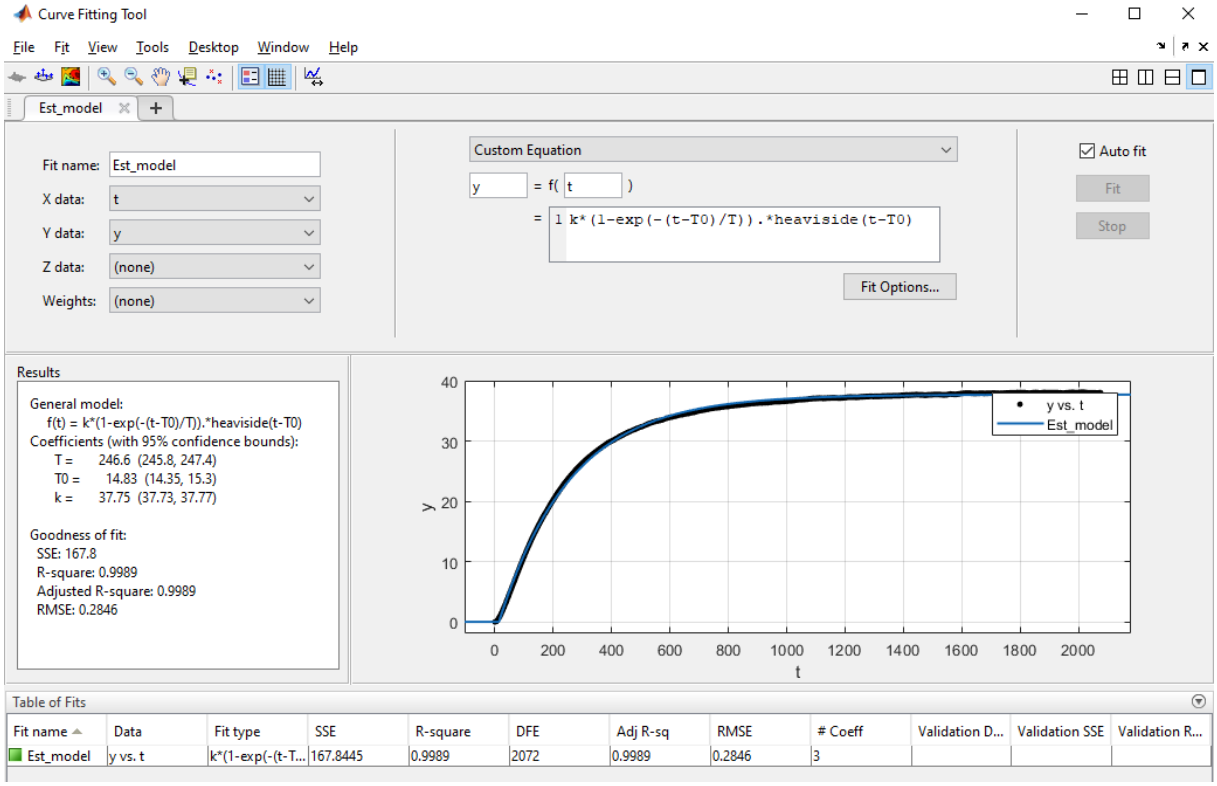


Rysunek 2. Wykres przebiegu temperatury przy wymuszeniu maksymalnym sygnałem sterującym

Przebieg wskazuje na obiekt o charakterze inercyjnym z opóźnieniem transportowym. Przy podaniu na obiekt maksymalnego osiągalnego wymuszenia otrzymujemy maksymalną temperaturę około 64,1℃. Do modelowania, od tej wartości należy odjąć offset ($64{,}1 - 26{,}05 = 38{,}05$℃) aby znormalizować wartości – przebieg będzie zaczynał się wtedy od wartości 0℃.

Do wyznaczenia parametrów obiektu z odpowiedzi skokowej użyto narzędzia Curve Fitting Tool.

Wzór do dopasowania odpowiedzi skokowej do obiektu FOPDT zaczerpnięto z pracy "Curve fitting software for first order plus dead time (FOPDT) model parameter estimation using step or pulse response data: a tutorial" – link. Poniżej przedstawiono efekt działania narzędzia CFT:



Rysunek 3. Identyfikacja parametrów obiektu z odpowiedzi skokowej

Estymowaną transmitancję można zatem przedstawić wzorem:

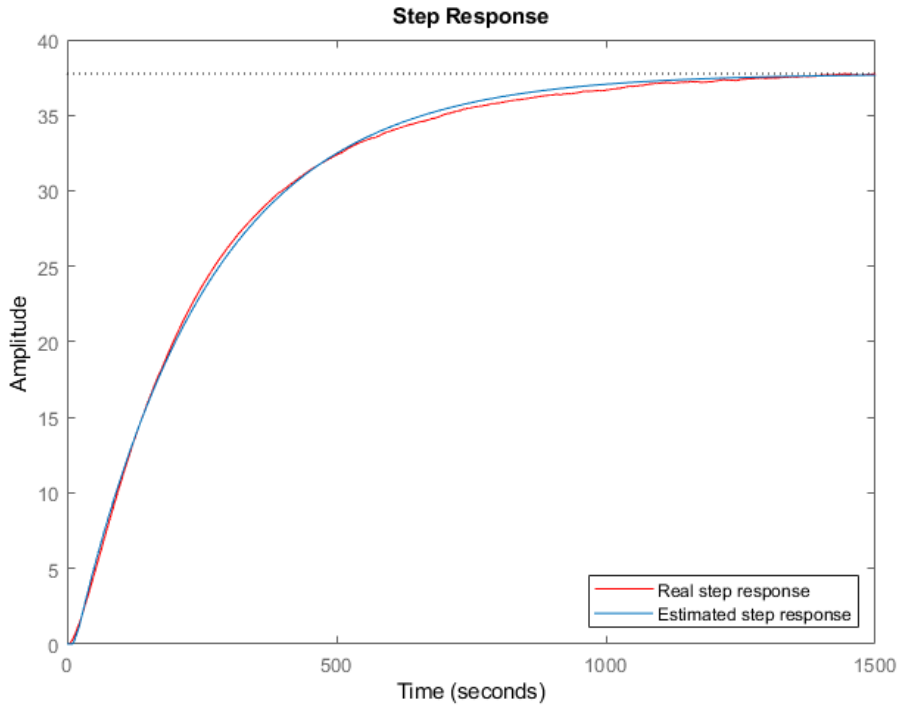
$$G_{est}(s) = \frac{k}{1+sT}e^{-sT_0} = \frac{37{,}75}{1+246{,}6s}e^{-14{,}83s}$$

Wzmocnienie $k = 37{,}75$

Stała czasowa $T = 246{,}6$

Opóźnienie transportowe $T_0 = 14{,}83$

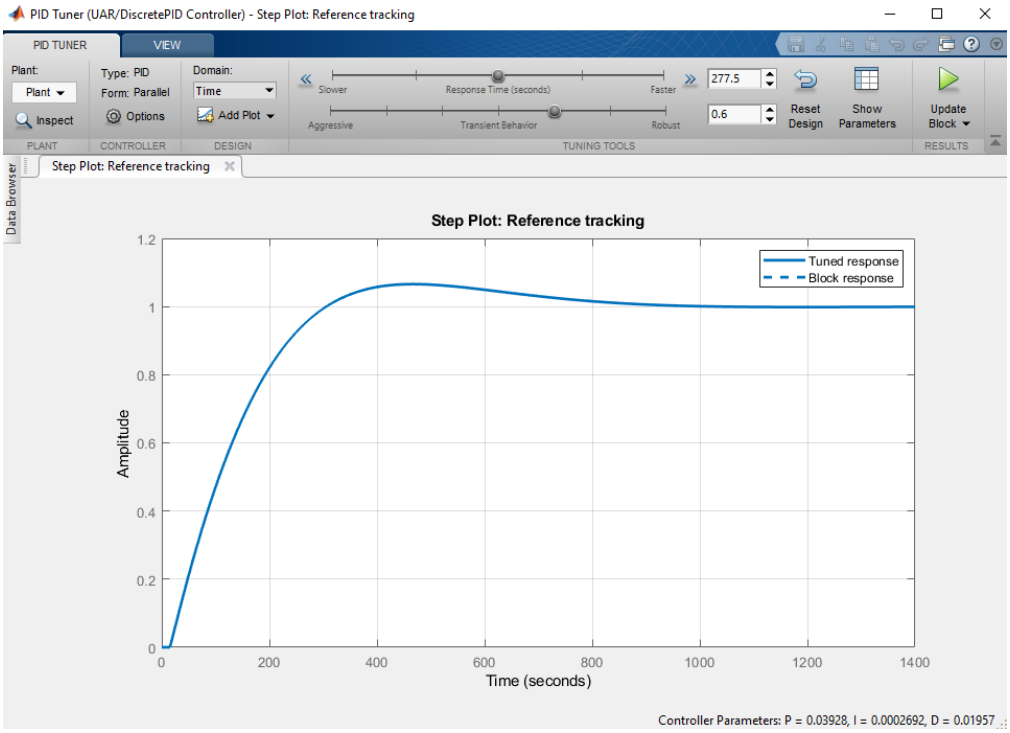
Poniżej przedstawiono porównanie odpowiedzi skokowych obiektu rzeczywistego i estymowanego:



Rysunek 4. Porównanie odpowiedzi skokowych obiektu rzeczywistego i estymowanego

Odpowiedzi odpowiadają sobie, więc identyfikację można uznać za poprawnie wykonaną. Wszystkie pliki (skrypt Matlab i plik .sfit narzędzia CFToolbox znajdują się w folderze Matlab_files oraz na GitHubie). Listing kodu dodano także na końcu dokumentacji.

Doboru nastaw regulatora dokonano przy pomocy bloku Discrete PID Controller i narzędzia PID Tuner.

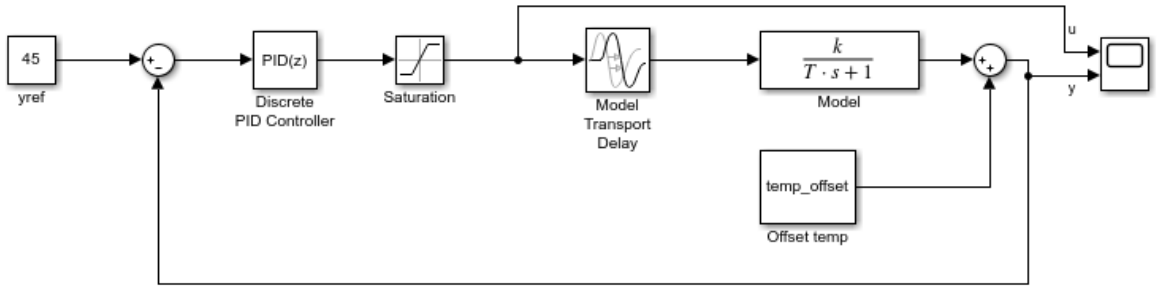


Rysunek 5. Dobór nastaw regulatora PID i odpowiedź obiektu w UAR z zaproponowanym regulatorem

Narzędzie PID Tuner pomogło dobrać nastawy, które pozwalają na spełnienie założenia niskiego uchybu ustalonego. Następnie nastawy zaimplementowano w programie mikroprocesora i zbadano odpowiedź układu. Nastawy jakie dobrano to:

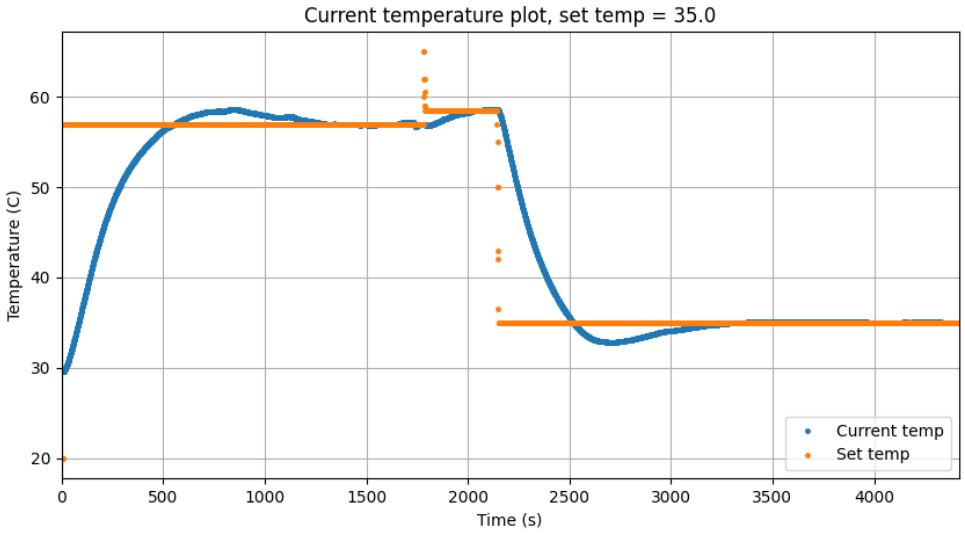
$$K_p = 0{,}03928$$
$$K_i = 0{,}0002692$$
$$K_d = 0{,}01957$$

Schemat blokowy UAR:



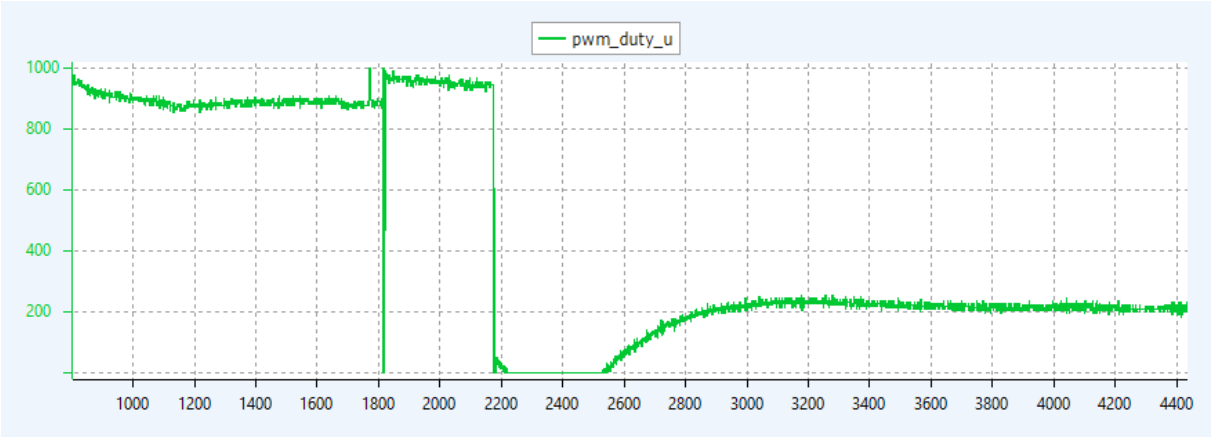
Rysunek 6. Schemat blokowy UAR

Rzeczywista odpowiedź układu z regulatorem PID i wyznaczonymi nastawami:



Rysunek 7. Przebieg wartości wyjściowej podczas testów projektu

Oraz przebieg sygnału sterującego wykreślony w Data Trace Timeline Graph:



Rysunek 8. Przebieg sygnału sterującego podczas testów projektu

Jak widać na przebiegu, regulacja PID temperatury działa zgodnie z wymaganiami – temperatura odczytana czujnikiem stabilizuje się na zadanej wartości. Uchyb ustalony mieści się w granicy błędu zadanego 1%.

7. Opis rozwiązań zastosowanych w programie mikroprocesora

- **Interfejs UART** skonfigurowano w trybie Receive and Transmit oraz w trybie przerwaniowym NVIC i DMA. Przez ten interfejs można pobierać dane od użytkownika (temperatura zadana) oraz wysyłać dane do portu szeregowego w komputerze (aktualna temperatura i temperatura zadana). Poniżej przedstawiono Callback odbioru danych przez mikroprocesor:

  main:
```
// Get setpoint value from user
  HAL_UARTEx_ReceiveToIdle_DMA(&huart3, (uint8_t *)get_UART, 10);
```

  while(1):
```
// Reset data from UART
  memset(get_UART, 0, 10);
```

  poza mainem:
```
// UART callback handling
void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size){
   if(huart->Instance == USART3){
          float tmp = atof(get_UART);
          if(tmp < 20) set_temp_f = 20;
          else if(tmp > 65) set_temp_f = 65;
          else set_temp_f = tmp;

          HAL_UARTEx_ReceiveToIdle_DMA(&huart3, (uint8_t *)get_UART, 10);
   }
}
```

- Skonfigurowano **interfejs I2C** do obsługi czujnika BMP280
- Do obsługi **czujnika BMP280** i **wyświetlacza LCD** użyto zewnętrznych bibliotek załączonych do projektu
- Skonfigurowano **3 timery**:
  - TIM1 do sterowania sygnałem PWM sterującym bramką tranzystora IRF520N (max. wypełnienie = 999)
  - TIM3 w trybie przerwaniowym NVIC do wyzwalania pomiaru temperatury, wysyłania danych, obliczania sygnału sterującego (razem z nasyceniem) i sterowania tranzystorem co okres próbkowania $T_p = 1s$:

  poza mainem:
```
// TIMERS callback handling
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
   if(htim->Instance == TIM3){
          // TEMPERATURE
          BMP280_ReadTemperatureAndPressure(&current_temp_f, &pressure);
          sprintf(current_temp_ch_UART, "Current temperature: %.2f\n\r",
current_temp_f);
          HAL_UART_Transmit(&huart3, (uint8_t *)current_temp_ch_UART,
sizeof(current_temp_ch_UART)-1, 1000);

          sprintf((char*)set_temp_ch_UART, "Set temperature: %.2f\n\r",
set_temp_f);
```

```
            HAL_UART_Transmit(&huart3, (uint8_t*)set_temp_ch_UART,
    strlen(set_temp_ch_UART), 1000);

            pwm_duty_f = (htim1.Init.Period * calculate_PID(&PID1, set_temp_f,
    current_temp_f));

            // Saturation
            if(pwm_duty_f < 0.0) pwm_duty_u = 0;
            else if(pwm_duty_f > htim1.Init.Period) pwm_duty_u =
    htim1.Init.Period;
            else pwm_duty_u = (uint16_t) pwm_duty_f;

            //pwm_duty_u = htim1.Init.Period; // 100% PWM duty for creating model
            __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, pwm_duty_u);
        }
    }
```

- TIM4 w trybie pollingu do obsługi enkodera obrotowego:

main:

```
// Prevents from bugging set_temp_f when encoder counter value goes through 0
  htim4.Instance->CNT = 65535 / 2;
```

while(1):

```
// ENCODER
        enc_uint = __HAL_TIM_GET_COUNTER(&htim4);
        enc_diff_int = enc_uint - prev_enc_uint;
        if(enc_diff_int >= 2 || enc_diff_int <= -2){
              enc_diff_int /= 2;
              set_temp_f += 0.5 * enc_diff_int;
              if(set_temp_f > 65) set_temp_f = 65;
              if(set_temp_f < 20) set_temp_f = 20;
        }
        prev_enc_uint = enc_uint;
```

- Skonfigurowano obsługę **wyświetlacza LCD** w pętli głównej programu:

main:

```
    LCD_init();
    LCD_write_command(LCD_CLEAR_INSTRUCTION);
    LCD_write_command(LCD_HOME_INSTRUCTION);
```

while(1):

```
// LCD
        snprintf(current_temp_ch_LCD, LCD_MAXIMUM_LINE_LENGTH, "Temp:  %.2f",
current_temp_f);
        LCD_write_text(current_temp_ch_LCD);
        LCD_write_data(LCD_CHAR_DEGREE);
        LCD_write_char('C');
        snprintf(set_temp_ch_LCD, LCD_MAXIMUM_LINE_LENGTH, "Set T: %.2f",
set_temp_f);
        LCD_goto_line(1);
        LCD_write_text(set_temp_ch_LCD);
        LCD_write_data(LCD_CHAR_DEGREE);
        LCD_write_char('C');
        HAL_Delay(100);
        LCD_write_text("                ");
        LCD_write_command(LCD_HOME_INSTRUCTION);
```

- Zaimplementowano **regulator PID** oraz funkcję obliczającą sygnał sterujący:

poza mainem:

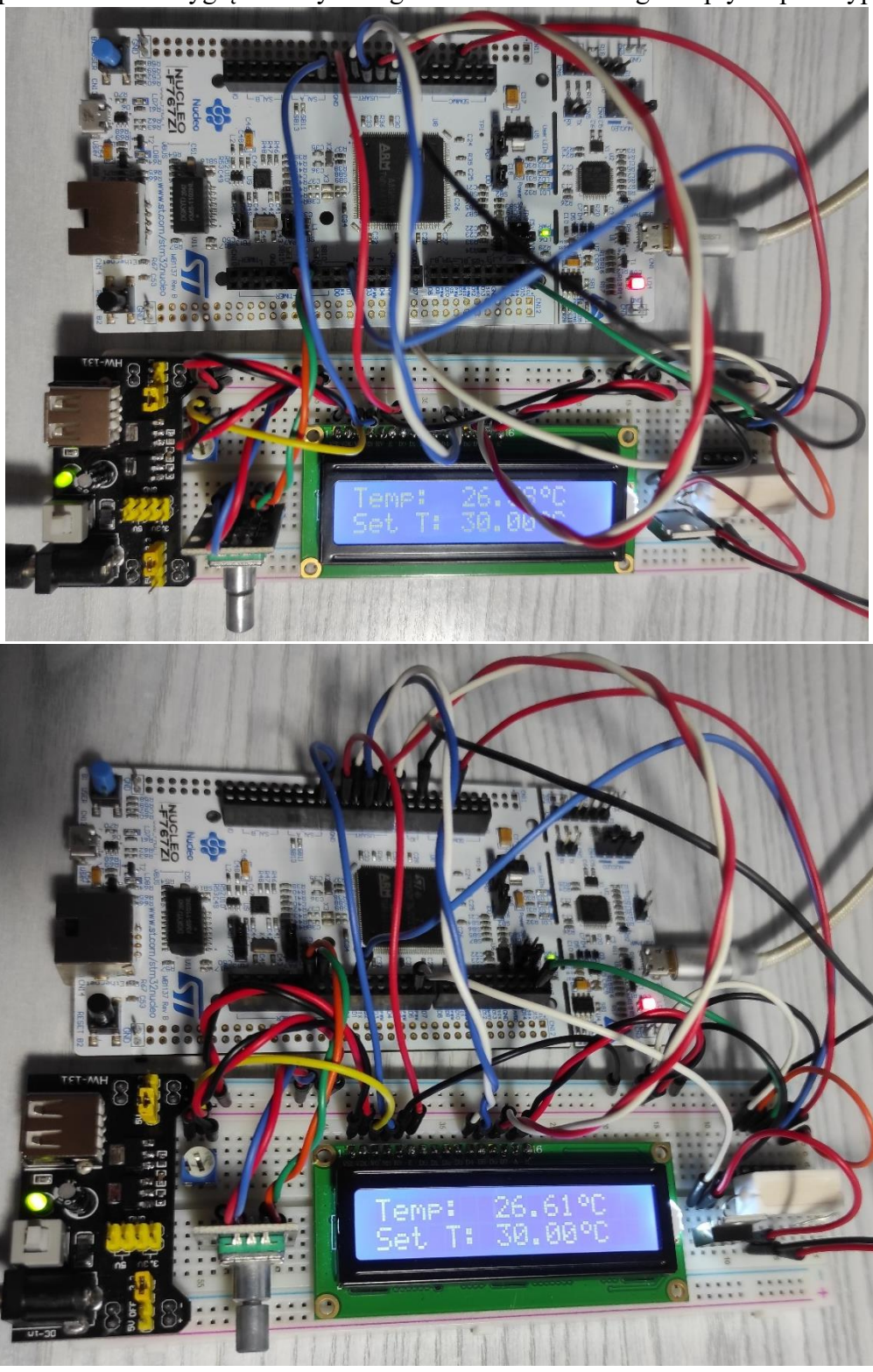
```
struct Controller{
	float Kp;
	float Ki;
	float Kd;
	float Tp;
	float prev_error;
	float prev_u_I;
};

float calculate_PID(struct Controller *PID, float set_temp, float meas_temp){
	float u = 0;
	float error;
	float u_P, u_I , u_D;

	error = set_temp - meas_temp;

	// Proportional
	u_P = PID->Kp * error;

	// Integral
	u_I = PID->Ki * PID->Tp / 2.0 * (error + PID->prev_error) + PID->prev_u_I;
	PID->prev_u_I = u_I;

	// Derivative
	u_D = (error - PID->prev_error) / PID->Tp;

	PID->prev_error = error;

	// Sum of P, I and D components
	u = u_P + u_I + u_D;

	return u;
}

struct Controller PID1;
```

main:

```
  // Initialize PID Controller parameters and init data
  PID1.Kp = 0.03928;
  PID1.Ki = 0.0002692;
  PID1.Kd = 0.01957;
  PID1.Tp = 1;
  PID1.prev_error = 0;
  PID1.prev_u_I = 0;
```

8. Zdjęcia fizycznego układu

Poniżej przedstawiono wygląd rzeczywistego układu zbudowanego na płytce prototypowej:



Rysunek 9. Zdjęcia fizycznego układu

9. Link do repozytorium Github

Kompletny projekt na mikroprocesor STM32, skrypt Python, Matlab oraz pełną dokumentację zamieszczono jako publiczne repozytorium Github pod linkiem:

https://github.com/JakubGrzesiak/SM---Final-Project.git

Listing kodu Python:

```
import serial #pip install pyserial
import matplotlib.pyplot as plt
import time
import csv

# Init variables/arrays
t = 0
Tp = 1                          # sampling time [s]
curr_temp_samples = []
curr_temp_flag = 0
set_temp_samples = []
set_temp_flag = 0
timebase = []
d = bytearray()                 # serial read buffer

# Handling plot close event
close_flag = 1
def handle_close(event):
    global close_flag
    close_flag = 0
    print("Data logging finished!")

# Figure init
fig = plt.figure(figsize=(10,6))
fig.canvas.mpl_connect('key_press_event', handle_close)
fig.canvas.mpl_connect('close_event', handle_close)
plt.ion()

# Saving data to .csv file
timestr = time.strftime("%Y%m%d-%H%M%S")
data = open("data_{}.csv".format(timestr) , 'w', newline='')
writer = csv.writer(data, delimiter=',')
header = ["Time", "Curr_temp", "Set_temp"]
writer.writerow(header)

# For debugging
# UART = serial.Serial("COM4", 115200, timeout=1,
parity=serial.PARITY_NONE)
# set_start_temp = "30.00"

# User input
COM_PORT = int(input("Serial port number: "))
UART = serial.Serial("COM{}".format(COM_PORT), 115200, timeout=1,
parity=serial.PARITY_NONE)
set_start_temp = str(input("Set temperature: "))
UART.write(set_start_temp.encode())

while close_flag:
    if UART.inWaiting() >= 0:
        d += UART.read(1)
        if b"Current temperature: " in d:
            curr_temp = UART.read(5)
            curr_temp = curr_temp.decode()
```

```
            curr_temp = float(curr_temp)
            curr_temp_samples.append(curr_temp)
            d[::] = b""
            curr_temp_flag = 1
            print("Current temperature:", curr_temp)
        elif b"Set temperature: " in d:
            set_temp = UART.read(5)
            set_temp = set_temp.decode()
            set_temp = float(set_temp)
            set_temp_samples.append(set_temp)
            d[::] = b""
            set_temp_flag = 1
            print("Set temperature:", set_temp)

    if curr_temp_flag and set_temp_flag:
        # Prepare x label for plot
        timebase.append(t)
        t += Tp
        print("Timebase:", timebase)

        # Plotting data
        plt.clf()
        plt.grid(True)
        plt.plot(timebase, curr_temp_samples, '.', markersize=5,
label="Current temp")
        plt.plot(timebase, set_temp_samples, '.', markersize=5, label="Set
temp")
        plt.xlim(0, t + 1)
        plt.title("Current temperature plot, set temp =
{}".format(set_temp))
        # plt.title("Model data, PMW duty 100%")     # Used for model
identification
        plt.xlabel("Time (s)")
        plt.ylabel("Temperature (C)")
        plt.legend(loc="lower right")
        plt.show(block=False)
        fig.canvas.flush_events()
        plt.pause(0.0001)
        writer.writerow([timebase[-1], curr_temp_samples[-1],
set_temp_samples[-1]])
        curr_temp_flag = 0
        set_temp_flag = 0

    if close_flag == 0:
        break

fig.savefig("Temp_plot_{}.png".format(timestr))
UART.close()
data.close()
```

Listing kodu Matlab:

```
% clear all; close all; clc;

% Read and parse data
data = dlmread('Model_data.csv', ',', 2, 0);
t = data(:,1);
y = data(:,2);
temp_offset = y(1);
yr = 64.1 - temp_offset;
y = y - y(1); % normalize data (start from 0)
```

```
% Plot step response
plot(t, y, 'r');
title("Normalized model data, PMW duty 100%");
xlabel("Time (s)");
ylabel("Temperature (C)");
xlim([0 max(t)]);

% Estimated transfer function
k = 37.75;
T = 246.6;
T0 = 14.83;
est_model = tf([k], [T 1], 'InputDelay', T0);
hold on;
step(est_model);
legend("Real step response", "Estimated step response", 'Location',
'southeast');

% PID controller in simulink
sim('UAR.slx')
```

Bibliografia/Źródła

- https://os.mbed.com/platforms/ST-Nucleo-F767ZI
- https://msalamon.pl/dziecinnie-prosta-sprzetowa-obsluga-enkodera-na-stm32
- https://www.electronics-tutorials.ws/pl/tranzystor/mosfet-jako-przelacznik.html
- https://msalamon.pl/dostalismy-swietna-obsluge-przerwania-uart-idle-w-halu/
- https://matplotlib.org/stable/index.html
- https://www.researchgate.net/profile/Chris_Cox6/publication/316658102_First_order_plus_dead_time_FOPDT_model_parameter_estimation/links/5b2275ed0f7e9b0e37423cf6/First-order-plus-dead-time-FOPDT-model-parameter-estimation