Poznań University of Technology
Faculty of Control, Robotics and Electrical Engineering
Institute of Robotics and Machine Intelligence
Division of Control and Industrial Electronics

CPU Fan control system

Microprocessor Systems Laboratory Classes

Project report and documentation

Created by:

Paweł Rozenblut, ID: 140361
Ram Ellanki, ID: 139783
Keshav Sharma, ID: 142862

Submission Date: February 2, 2021
Supervisor: Adrian Wójcik, M.Sc.

Contents

1 Final task . . . **3**

1.1 Specification . . . 3

1.2 Implementation . . . 3

1.2.1 PWM Control Circuit . . . 3

1.2.2 Tachometer Circuit . . . 4

1.2.3 Rotary Encoder Circuit . . . 4

1.2.4 The 'MS' Library . . . 5

1.2.5 Usage of the 'MS' library written for the project . . . 6

1.3 Test results . . . 7

1.4 Summary . . . 9

1 Final task

1.1 Specification

The following section contains information on the project developed for the Microprocessor Systems Laboratory classes at Poznań University of Technology, as a part of first cycle studies in the field of Automatic Control and Robotics. Selected problem - to provide control routines for the F7 series STM32 Nucleo development board, and assemble control circuits; all together allowing one to obtain a complete rotational velocity control system control of the 3-wire 5V CPU fan. The board configuration can be the same as the .ioc file provided by the instructor for fourth laboratory classes. Basing on that, one must perform the following setup:

1. HCLK = 216 MHz Timers 3, 6 and 7 enabled where the last one is additional.
2. TIM3 for PWM generation at PC7 (CH2); prescaler = 8, counter period = 999,
3. TIM6 for periodical RPM updates and data transmission trigger: prescaler = 10799, counter period = 9999, trigger event = Reset (at max), interrupt enabled,
4. Communication - same as in default .ioc for laboratory, with enabled interrupt mode.

1.2 Implementation

The following figure [1] presents the block diagram depicting the operation of the system. Details on physical and program components to the task are presented further in this section.

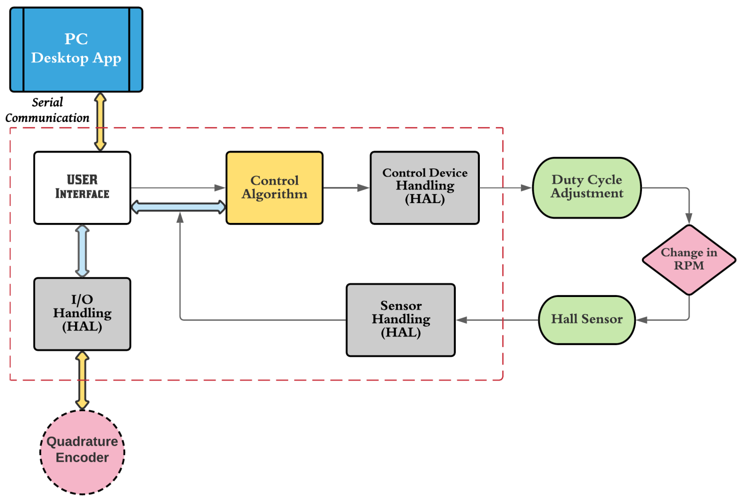


Figure 1: Block Diagram

1.2.1 PWM Control Circuit

We are using Open drain configuration of N-Channel Mosfet(2N7000) and Pin PC7(PC7 in IOC configuration and D21 on board) for our PWM control circuit. PC7 configured as TIM3_CH2 PWM Generation.

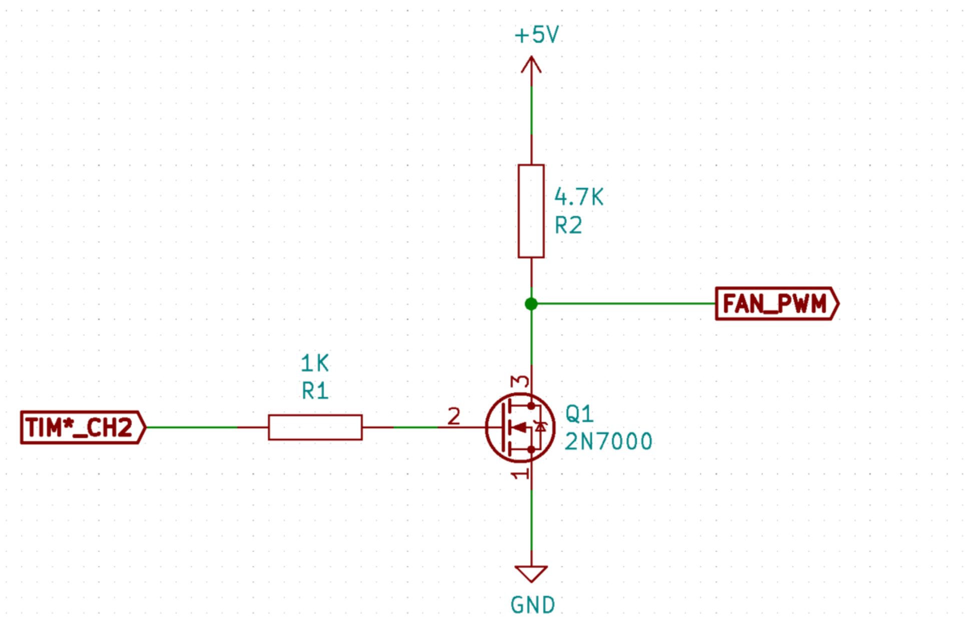


Figure 2: Fan Pwm Circuit with N-Channel Mosfet(2N7000)

1.2.2 Tachometer Circuit

In our Tachometer circuit we connected Fan tachometer pin to the pin PE0(PE0 in IOC configuration and D34 on board) configured as GPIO mode *External interrupt Mode with Rising edge trigger detection* in *Pull-up* configuration with a pull up resistor of 10K connected with 3.3V power supply.

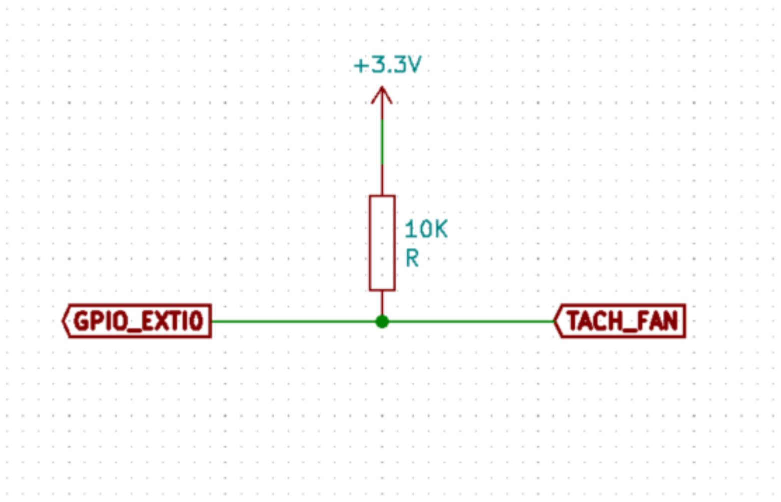


Figure 3: Tachometer Circuit

1.2.3 Rotary Encoder Circuit

For Rotary Encoder circuit we took idea from our PWM lab introduction file with some changes. Instead of TIM*CH* configuration pins we are using GPIO configuration pins. We connected Rotary encoder CLK pin to the pin PE9(PE9 in IOC configuration and D6 on board) configured as GPIO mode *GPIO Input* and DT pin to the PE11(PE11 in IOC configuration and D5 on board) configured as GPIO mode *External interrupt Mode with falling edge trigger detection*. Rotary Encoder circuit contains 2 pull-up resistor of 10K connected with 5V power supply and RC low pass filter.

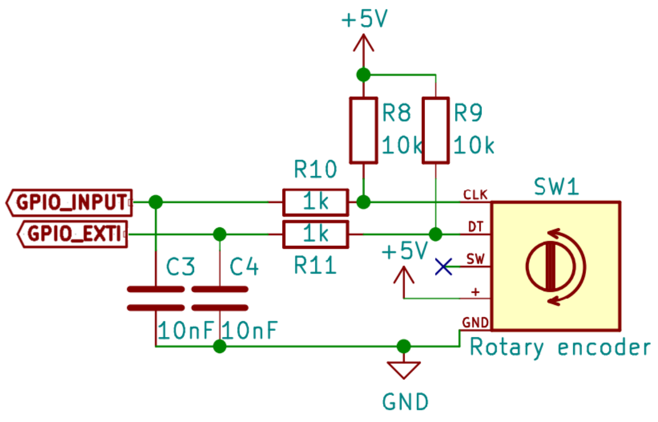


Figure 4: Rotary encoder with pull-up resistors and RC low- pass filter.(Original By Adrian Wójcik, M.Sc)

1.2.4 The 'MS' Library

The code realising the function has been grouped into three sections - the one responsible for UART communication ('MS_UART_trns'), the core module ('MS_core'), and functions realising the control routines ('MS_signal_control'). Those names refer to .c/.h file pairs, and are brought together for including in STM32 project in one header ('MS_proj_al.h'), whose contents are presented in [4]. In case of all the presented code - the doxygen-type comments of the code is omitted in listing (but still exists in the attached files).
'MS_core.h' contains the *ProgramData* struct, and a prototype of the STM32 peripheral interface handlers initialisation function. It also serves the purpose of handling the STM32CubeIDE generated headers to the other functions in the model (since it all is based on its HAL library after all). Its contents are presented in [7], whereas the body of the corresponding .c file - in [8].
Functions responsible for controlling the signal are presented in [9] and [10]. Most of those are 'helper functions', but the 'compensate_error(...)' deserves to be examined. Said function realises the adjustment of PWM duty cycle of signal set on the gate of a MOSFET in the electronic control circuit. In terms of automatic control, and *ON ITS OWN*, its operation could be approximately characterised as a three position relay-type element [5].

$$y_N = \begin{cases} U\,\mathrm{sign}(x) & \text{for } |x| > a \\ 0 & \text{for } |x| \leqslant a \end{cases}$$

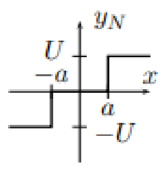
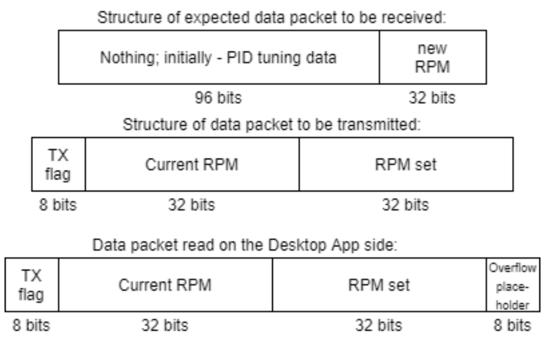


Fig. 2.5. Three-position relay

Figure 5: Characteristics of a relay-type-element with a dead-zone. Source: Lecture 02, Automatic Control II, by dr.hab. inż. Dariusz Horla.

Refering to the presented picture [5], value of U is $\approx 5[mV]$, (the value of PWM duty is decreased but that

follows from the structure of the electronic control circuit). There were plans to incorporate the PID controller instead, but due to the nature of the plant (1-st order system) - the presented solution proved to be easier to implement, and sufficient for our needs.
The contents of the module responsible for data exchange with the PC is presented in [5] and [6]. The structure of data received and transmitted is following:


Structure of expected data packet to be received:
Nothing; initially - PID tuning data
new RPM
96 bits
32 bits
Structure of data packet to be transmitted:
TX flag
Current RPM
RPM set
8 bits
32 bits
32 bits
Data packet read on the Desktop App side:
TX flag
Current RPM
RPM set
Overflow place-holder
8 bits
32 bits
32 bits
8 bits
Figure 6: Structure of data packets.

As can be seen on the figure [6] - the data packet read out by the desktop app is 8 bits longer than the transmitted one; since desktop app does not use any data integrity validation mechanism, we introduced our own method for partial determination, whether to read data or not. In situation where the desktop app has been turned off and on while the control system was operating - the data transmitted are accumulated, and upon resuming the desktop app operation, the data are not valid - in this case we don't want to parse the incoming data - we want to wait until the normal mode of operation is resumed, and this is achieved by checking whether after expected 72 bit of data - we get something else. If not, then the program is considered to be operating properly.

1.2.5 Usage of the 'MS' library written for the project

After copying all the library files into the project workspace, include the "MS_proj_al.h" file in 'main.c', then define the ProgramData structure, as presented in [1].

```c
/* USER CODE BEGIN Includes */
#include "MS_proj_al.h"
(...)
/* USER CODE BEGIN PV */
Program_Data pd;
```

Source code 1: Final task, include and private variable sections.

Having initialised the ProgramData structure, we pass its address to the initialize_STM32_interfaces(...) function to initialise the STM32 interfaces [2].

```c
/* USER CODE BEGIN 2 */
initialize_STM32_interfaces(&pd);
```

Source code 2: Interface intialisation function

Lastly, in appropriate callbacks sections, we call the library functions, like so [3]:

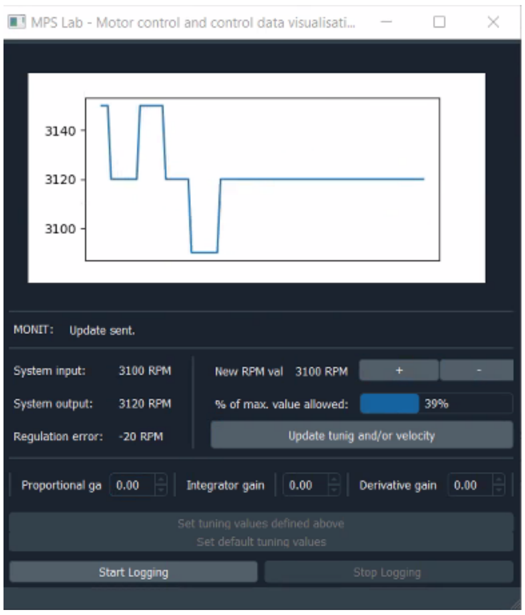
```c
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == TACH_Pin)
  {
    increment_counter_at_tacho_ev(&pd);
  }

  if (GPIO_Pin == ENCODER_A_Pin)
  {
    encoder_RPM_update(&pd);
    set_PWM_from_RPM(&pd);
    update_PWM_duty(&pd);
  }
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM6)
  {
    get_RPM_actual(&pd);
    read_PWM_duty(&pd);
    compensate_error(&pd);
    update_PWM_duty(&pd);
    transmit_data_packet(&pd);
  }

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if (huart->Instance == USART3)
  {
    deparse_received_data(&pd);
    set_PWM_from_RPM(&pd);
    update_PWM_duty(&pd);

    HAL_UART_Receive_IT(&huart3, pd.rx_buffer, 16);
  }
}

/* USER CODE END 4 */
```

Source code 3: Implementation of logic for control of the circuit.

As can be seen, the control and comunication functions are placed in appropriate callbacks. In case of HAL_GPIO_EXTI - for reading the encoder position (for increasing/decreasing RPM), and evaluating current RPM (impulses from HAL sensor in the fan). Timer callback triggers the data-update, and control routines; updated data are transmitted through UART (each one second). Lastly, the 'data received' callback - for updating the RPM value, from data received from the desktop app.

1.3 Test results

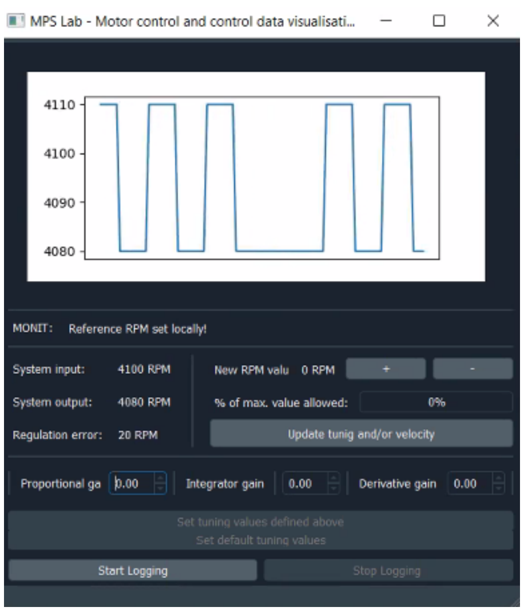
The operation of the circuit has been presented in the pictures in this subsection. In [7] we see the result in sending an update from a desktop application, in [8] we see the monitoring of the value set through a physical interface (encoder regulating the RPM value, observed in the 'System input' field). Figure [9] presents data received from the board. As can be observed, the system is functioning properly. 'Start Log' allows us to record input and output data, and store it as entries in a .txt files appearing in the app's directory (for both input and output - each log session appear in the same file - automatically created if not exists, preceded with current date/hour headers - intended workflow: copy and paste data into Excel/MATLAB workspaces). Other fields in the app's GUI are rather self-explanatory, and aside from the PID section (disabled), all features have been tested and are functional.


MPS Lab - Motor control and control data visualisati...
3140
3120
3100
+
-
Update tunig and/or velocity
Set tuning values defined above
Set default tuning values
Start Logging
Stop Logging

MONIT: Update sent.

System input: 3100 RPM
System output: 3120 RPM
Regulation error: -20 RPM
New RPM val 3100 RPM
% of max. value allowed: 39%
Proportional ga 0.00
Integrator gain 0.00
Derivative gain 0.00

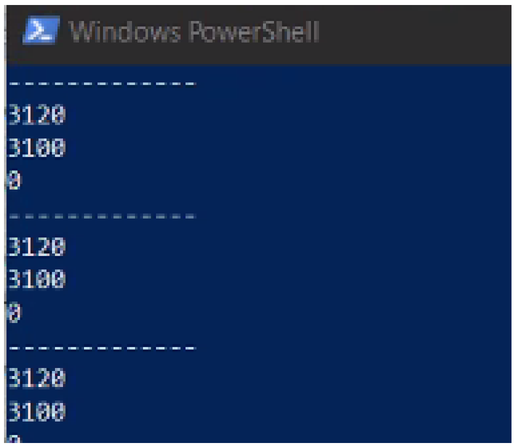
Figure 7: Update sent from the PC - state monitoring.


MPS Lab - Motor control and control data visualisati...
4110
4100
4090
4080
+
-
Update tunig and/or velocity
Set tuning values defined above
Set default tuning values
Start Logging
Stop Logging

MONIT: Reference RPM set locally!

System input: 4100 RPM
System output: 4080 RPM
Regulation error: 20 RPM
New RPM valu 0 RPM
% of max. value allowed: 0%
Proportional ga 0.00
Integrator gain 0.00
Derivative gain 0.00

Figure 8: RPM updated through the rotary encoder - state monitoring.



```
-------------
3120
3100
0
-------------
3120
3100
0
-------------
3120
3100
```

Figure 9: Terminal view of the data received through serial port.

1.4 Summary

Overall, the project has been realised successfully. It fulfills all the minimum, and 5 out of 8 possible additional requirements.

Appendix A: Embedded app, source codes

As stated earlier - presented codes are without comments explaining its operation; those can be found in the original files attached to this report.

```c
#ifndef MS_COMBO_H
#define MS_COMBO_H

#include "MS_core.h"
#include "MS_signal_control.h"
#include "MS_UART_trns.h"

#endif
```

Source code 4: MS_proj_al.h

```c
#ifndef MS_TRANS_H
#define MS_TRANS_H

#include "MS_core.h"

void deparse_received_data(Program_Data *pd);

void transmit_data_packet(Program_Data *pd);

#endif
```

Source code 5: MS_UART_trns.h

```c
#include "MS_UART_trns.h"
#include "MS_core.h"

void transmit_data_packet(Program_Data *pd)
{
  HAL_UART_Transmit(&huart3, (uint8_t *)&(pd->tx_flag), 1, 1000);

  uint8_t a[4] = {
      0,
  };

  memcpy(a, &(pd->RPM_actual), sizeof(pd->RPM_actual));
  HAL_UART_Transmit(&huart3, (uint8_t *)a, 4, 1000);

  memcpy(a, &(pd->RPM_reference), sizeof(pd->RPM_reference));
  HAL_UART_Transmit(&huart3, (uint8_t *)a, 4, 1000);
}

void deparse_received_data(Program_Data *pd)
{
  pd->RPM_reference = pd->rx_buffer[12] | (pd->rx_buffer[13] << 8) | (pd->rx_buffer[14] << 16) | (pd->rx_buffer[15] << 24);
}
```

Source code 6: MS_UART_trns.c

```c
#ifndef MS_CORE_H
#define MS_CORE_H

#include "main.h"
#include "adc.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"

#include "stdio.h"
#include "arm_math.h"
#include "math.h"

typedef struct Program_Data
{
  uint32_t RPM_actual;    // Acutal/Current speed at which the fan is rotating
  uint32_t RPM_reference; // Desired value of speed at which the fan is supposed to rotate.
  uint32_t PWM; // Current value of PWM duty cycle (expressed as permille)
  uint8_t rx_buffer[16];      // Buffer for the received data.
  uint8_t tx_flag;            // Transmission flag, for desktopApp  interfacing purposes
  uint32_t HAL_PULSE_counter; // Amount of pulses generated by HAL sensor-based tachometer, in 1 s.
} Program_Data;
void initialize_STM32_interfaces(Program_Data *pd);

#endif
```

Source code 7: MS_core.h

```c
#include "MS_core.h"
#include "MS_signal_control.h"

void initialize_STM32_interfaces(Program_Data* pd)
{
    HAL_TIM_Base_Start_IT(&htim6);            // Period 1s
    HAL_TIM_Base_Start_IT(&htim7);            // Extra timer
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2); // Frequency 12kHz
    HAL_UART_Receive_IT(&huart3, pd->rx_buffer, 16);

    pd->tx_flag = 68;
    pd->HAL_PULSE_counter = 0;

    set_reference_RPM(pd, 0);
    set_PWM_from_RPM(pd);
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, pd->PWM);
}
```

Source code 8: MS_core.c - initialisation of STM32 peripherials (based on provided ProgramData struct.

```c
#ifndef MS_SIGNAL
#define MS_SIGNAL
#include "MS_core.h"

void compensate_error(Program_Data* pd);
void set_PWM_from_RPM(Program_Data* pd);
void set_reference_RPM(Program_Data* pd, uint32_t new_RPM_ref);
void get_RPM_actual(Program_Data* pd);
void increment_counter_at_tacho_ev(Program_Data* pd);
void update_PWM_duty(Program_Data* pd);
void read_PWM_duty(Program_Data* pd);
void encoder_RPM_update(Program_Data* pd);

#endif
```

Source code 9: MS_signal_control.h - prototypes.

```c
#include "MS_signal_control.h"

void compensate_error(Program_Data *pd)
{
    if ((int)(pd->RPM_actual) > (int)(pd->RPM_reference + 40))
    {
        if (pd->PWM < 1000)
        {
            pd->PWM += 1;
        }
    }

    else if ((int)(pd->RPM_actual) < (int)(pd->RPM_reference - 40))
    {
        if (pd->PWM > 0)
        {
            pd->PWM -= 1;
        }
    }
}

void set_PWM_from_RPM(Program_Data *pd)
{
    if (pd->RPM_reference < 240)
    {
        pd->PWM = 1000;
        return;
    }

    if (pd->RPM_reference > 8000)
    {
        pd->PWM = 0;
        return;
    }

    pd->PWM = (uint32_t)(((float)pd->RPM_reference - 8332.9) / (-8.3707));
}

void set_reference_RPM(Program_Data *pd, uint32_t new_RPM_ref)
{
    pd->RPM_reference = new_RPM_ref;
}

void get_RPM_actual(Program_Data *pd)
{
    pd->RPM_actual = pd->HAL_PULSE_counter * 1 * 60 / 2;
    pd->HAL_PULSE_counter = 0;
}

void increment_counter_at_tacho_ev(Program_Data *pd)
{
    pd->HAL_PULSE_counter++;
}

void update_PWM_duty(Program_Data *pd)
{
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, pd->PWM);
}

void read_PWM_duty(Program_Data *pd)
{
    pd->PWM = __HAL_TIM_GET_COMPARE(&htim3, TIM_CHANNEL_2);
}

void encoder_RPM_duty_update(Program_Data *pd)
{
    GPIO_PinState ENC_B = HAL_GPIO_ReadPin(ENCODER_B_GPIO_Port, ENCODER_B_Pin);
    if ((ENC_B == 1) && (pd->RPM_reference < 7900))
    {
        pd->RPM_reference += 100;
    }
    if ((ENC_B == 0) && (pd->RPM_reference > 0))
    {
        pd->RPM_reference -= 100;
    }
}
```

Source code 10: MS_signal_control.h bodies of the functions.

Appendix B: List of physical components used to realise the task.

1. Evaluation board STM32F746ZG
2. 4020 PWM fan (5V)
3. Quadrature Encoder
4. Power Supply 5V
5. MOSFET 2N7000
6. Resistors (**1k, 4.7k, 10k**)
7. Capacitors (**10nF**)

Appendix C: Physical realisation of the circuit

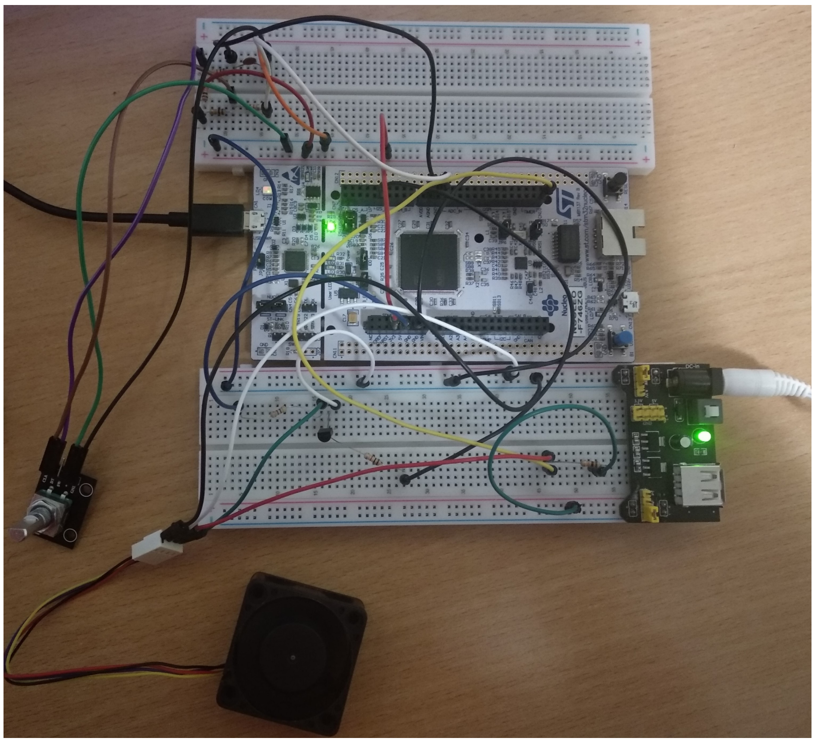


Figure 10: Physical realisation of the circuit