Poznań University of Technology

Faculty of Control, Robotics
and Electrical Engineering

Institute of Robotics and Machine Intelligence

Division of Control and Industrial Electronics

Microprocessor Systems Final Task

Microprocessor Systems

Nihal Suri, 143208
Nihal.Suri@put.poznan.pl

Maciej Mirecki, 144652
Maciej.Mirecki@put.poznan.pl

Adrian Wójcik, M.Sc.
Adrian.Wojcik@put.poznan.pl

04-02-2022

Contents

Obstacle Balancing 1 Degree of Freedom Beam

1.1 Specification

The idea behind this project is to develop a system which can be measured and controlled. It's a 1 Degree of Freedom system where there's an ultrasonic sensor(HCSRO4) that measures the distance from the object and servo motor(MG995) that controls the pitch of the beam on which the obstacle is balancing, depending on the distance from the object.

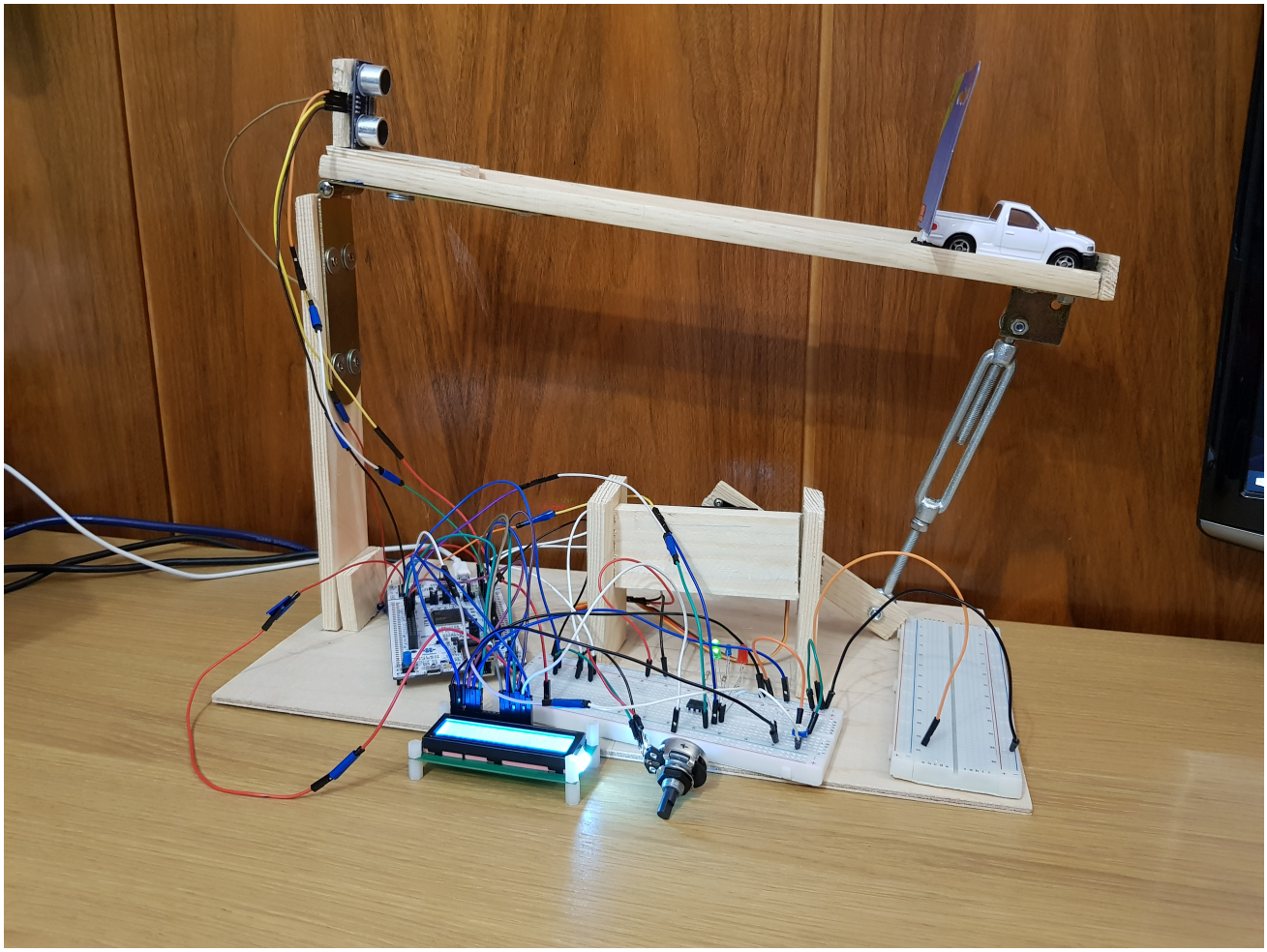


Fig. 1. Obstacle Balancing Position Control System

1.2 Software Implementation

All the software implementations are as follows:

1 Sensor

It's ultrasonic in nature and works in such a way that you need a short 10us pulse to the trigger input to start the ranging, and then the module will send out an 8 cycle burst of ultrasound at 40 kHz and raise its echo. The module itself has 4 pins: `VCC, TRIG, ECHO, GND` as seen in figure 2.

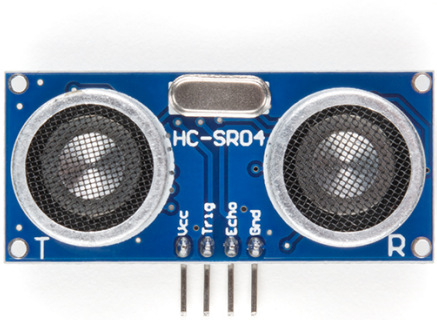


Fig. 2. Ultrasonic Sensor: HCSR04

It's been configured by using TIM8, in the `Input Capture` mode. This timer is a advanced one and is connected to the APB2 timer clocks(MHz). It's set in such a way so that it detects the rising edge input.

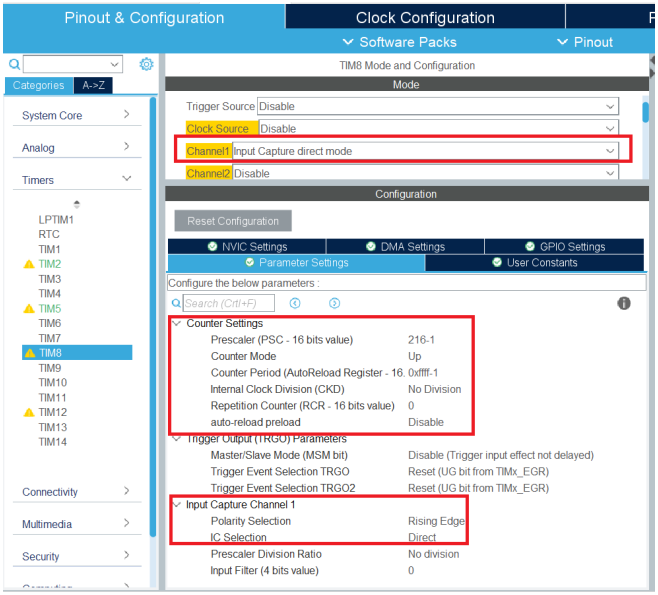


Fig. 3. Input Capture Mode on TIM8

The driver for this sensor has been developed keeping in mind the use case of the timer callback function and how its used to calculate the difference in time between the transmitted and received pulses so that this difference can be then divided by velocity to calculate the actual distance.
The code for the header file is as follows:

```
01. /**
02.   ******************************************************************************
03.   * @file    hcsro4.h
04.   * @authors  NS        Nihal.Suri@student.put.poznan.pl     MM          Maciej.
        Mirecki@student.put.poznan.pl
05.   * @version 2.0
06.   * @date    28-12-2021
07.   * @brief   Driver for ultrasonic distance sensor with Timer Input Capture Mode:
        HCSR04.
08.   *
09.   ******************************************************************************
10.   */
11. 
12. #ifndef INC_HCSR04_H_
13. #define INC_HCSR04_H_
14. 
15. /* Config ------------------------------------------------------------------*/
16. 
17. /* Includes ----------------------------------------------------------------*/
18. #include "stm32f7xx_hal.h"
19. 
20. /* Define ------------------------------------------------------------------*/
21. #define TRIG_PIN GPIO_PIN_7
22. #define TRIG_PORT GPIOC
23. 
24. 
25. /* Macro -------------------------------------------------------------------*/
26. 
27. /* Public variables --------------------------------------------------------*/
28. 
29. uint32_t IC_Val1 = 0;
30. uint32_t IC_Val2 = 0;
31. uint32_t Difference = 0;
32. uint8_t Is_First_Captured = 0;  // is the first value captured ?
33. float Distance  = 0;
34. 
35. /* Public function prototypes ----------------------------------------------*/
36. 
37. 
38. /**
39.  * @brief delay procedure.
40.  * @note delay between transmitting and receiving pulses
41.  * @param[in] time :  time in microseconds
42.  * @param[in] htim8 :  Input Capture timer handler
43.  * @return None
44.  */
45. 
46. void delay_us(uint16_t time, TIM_HandleTypeDef *htim8);
47. 
48. /**
49.  * @brief Input Capture Timer Callback.
50.  * @note Calculates distance by difference in time between pulses
51.  * @param[in] htim8 :  Input Capture timer handler
52.  * @return None
53.  */
54. 
55. void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim8);
56. 
```

```
57. /**
58.  * @brief Reads sensor values
59.  * @note Changes state of TRIG pin and enables interrupt for timer
60.  * @param[in] htim8 :  Input Capture timer handler
61.  * @return None
62.  */
63. 
64. void HCSR04_Read (TIM_HandleTypeDef *htim8);
65. 
66. 
67. 
68. 
69. #endif /* INC_HCSR04_H_ */
```

The code for the source file is as follows:

```
01. /**
02.   ******************************************************************************
03.   * @file    hcsro4.h
04.   * @authors  NS        Nihal.Suri@student.put.poznan.pl     MM          Maciej.
        Mirecki@student.put.poznan.pl
05.   * @version 3.0
06.   * @date    29-12-2021
07.   * @brief   Driver for ultrasonic distance sensor with Timer Input Capture Mode:
        HCSR04.
08.   *
09.   ******************************************************************************
10.   */
11. 
12. /* Includes ------------------------------------------------------------------*/
13. #include"hcsro4.h"
14. 
15. 
16. /* Typedef -------------------------------------------------------------------*/
17. 
18. /* Define --------------------------------------------------------------------*/
19. 
20. /* Macro ---------------------------------------------------------------------*/
21. 
22. /* Private variables ---------------------------------------------------------*/
23. 
24. /* Public variables ----------------------------------------------------------*/
25. 
26. /* Private function prototypes -----------------------------------------------*/
27. 
28. /* Private function ----------------------------------------------------------*/
29. 
30. /* Public function -----------------------------------------------------------*/
31. 
32. 
33. 
34. 
35. /**
36.  * @brief delay procedure.
37.  * @note delay between transmitting and receiving pulses
38.  * @param[in] time :  time in microseconds
39.  * @param[in] htim8 :  Input Capture timer handler
40.  * @return None
41.  */
42. 
43. void delay_us(uint16_t time, TIM_HandleTypeDef *htim8)
44. {
45.         __HAL_TIM_SET_COUNTER(htim8, 0);
46.         while (__HAL_TIM_GET_COUNTER (htim8) < time);
```

```
47.
48. }
49.
50.
51. /**
52.  * @brief Input Capture Timer Callback.
53.  * @note Calculates distance by difference in time between pulses
54.  * @param[in] htim8 :  Input Capture timer handler
55.  * @return None
56.  */
57.
58.
59. void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim8)
60. {
61.         if (htim8->Channel == HAL_TIM_ACTIVE_CHANNEL_1)  // if the interrupt
                source is channel1
62.         {
63.                 if (Is_First_Captured==0) // if the first value is not captured
64.                 {
65.                         IC_Val1 = HAL_TIM_ReadCapturedValue(htim8, TIM_CHANNEL_1)
                            ; // read the first value
66.                         Is_First_Captured = 1;  // set the first captured as true
67.                         // Now change the polarity to falling edge
68.                         __HAL_TIM_SET_CAPTUREPOLARITY(htim8, TIM_CHANNEL_1,
                            TIM_INPUTCHANNELPOLARITY_FALLING);
69.                 }
70.
71.                 else if (Is_First_Captured==1)   // if the first is already
                    captured
72.                 {
73.                         IC_Val2 = HAL_TIM_ReadCapturedValue(htim8, TIM_CHANNEL_1)
                            ;  // read second value
74.                         __HAL_TIM_SET_COUNTER(htim8, 0);  // reset the counter
75.
76.                         if (IC_Val2 > IC_Val1)
77.                         {
78.                                 Difference = IC_Val2-IC_Val1;
79.                         }
80.
81.                         else if (IC_Val1 > IC_Val2)
82.                         {
83.                                 Difference = (0xffff - IC_Val1) + IC_Val2;
84.                         }
85.
86.                         Distance = Difference * .034/2;
87.                         if((Distance>40)){
88.                                 Distance = 40;
89.                         }
90.                         Is_First_Captured = 0; // set it back to false
91.
92.                         // set polarity to rising edge
93.                         __HAL_TIM_SET_CAPTUREPOLARITY(htim8, TIM_CHANNEL_1,
                            TIM_INPUTCHANNELPOLARITY_RISING);
94.                         __HAL_TIM_DISABLE_IT(htim8, TIM_IT_CC1);
95.                 }
96.         }
97. }
98.
99.
100. /**
101.  * @brief Reads sensor values
102.  * @note Changes state of TRIG pin and enables interupt for timer
103.  * @param[in] htim8 :  Input Capture timer handler
```

```
104.  * @return None
105.  */
106.
107.
108.
109. void HCSR04_Read (TIM_HandleTypeDef *htim8)
110. {
111.         HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_SET);  // pull the TRIG
                 pin HIGH
112.         delay_us(10, htim8);  // wait for 10 us
113.         HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_RESET);  // pull the TRIG
                  pin low
114.         __HAL_TIM_ENABLE_IT(htim8, TIM_IT_CC1);
115. }
```

This driver allows us to calculate the distance in real time, and to call it in the main file the following steps have to be carried out, first the timer should be started in the correct mode as follows, for example `HAL_TIM_IC_Start_IT(htim8, TIM_CHANNEL_1);` and after this implementation in the infinite loop the following the read function must be called, `HCSR04_Read(htim8);`.

2 Servo Motor (Actuator)

The servo has been configured by generating a PWM signal from TIM2 and the prescaler and counter period have been set up in such a way so that the frequency of the timer is 50 Hz, as that is what is mentioned in the data-sheet. The same can be observed in figure 5.

Fig. 4. Servo Motor: MG995

The servo motor is controlled by changing the counter value of the timer, this is done in the control algorithm but before that the timer has to be started in the following way, `HAL_TIM_PWM_Start(htim2, TIM_CHANNEL_1);` the counter value is changed only in a certain range so that the angle of the motor can be controlled accurately according to the needs of the position control algorithm.

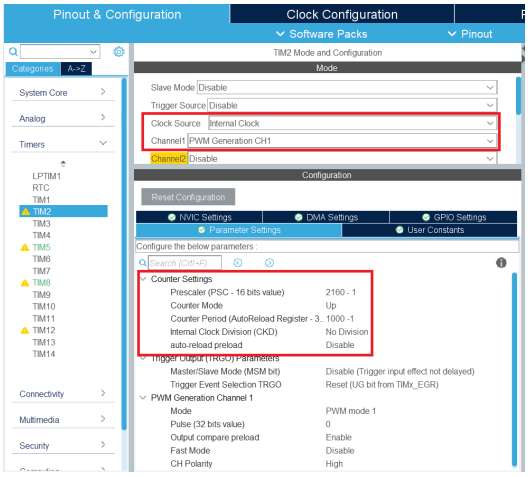


Fig. 5. PWM Generation on TIM2

3 Serial Communication

Serial communication is used in various areas for our task, the two places mainly being:

1. To set the set-point of the system at which the obstacle is supposed to balance on the beam.
2. Communication with the application, for various reasons such as:
   (a) Plotting time response.
   (b) Configuring the ideal set-point, for the obstacle to be at on the beam etc

The code for the following can be observed in the lisiting below:

Listing 1. Implementation of setting a reference value (set-point) with serial port

```
/**
 * @brief setting set point through UART
 * @note : range of input: 10 - 20, input 'SP:00' - switches control to ADC
 * @param[in] huart :  uart handler
 * @return None
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){

        if(huart->Instance == USART3){
          HAL_UART_Receive_IT(&huart3, (uint8_t *) msg_str, strlen("SP:10"));
          sscanf(msg_str, "SP:%d", &sp_temp);
          if(sp_temp == 0)
          {
                  sp_adc = 1;
          }
          else{
                  if(sp_temp >10 && sp_temp < 20)
                      sp = sp_temp;
                          sp_adc = 0;
          }
        }
}
```

4 Control Algorithm

Control algorithm of choice was PID. It was implemented with CMSIS DSP library. The input to the controller is error (set point - current position) and the output was transformed to values of duty cycle, which correspond to the specific angle of the beam. Duty cycle - angle relationship is presented in figure:

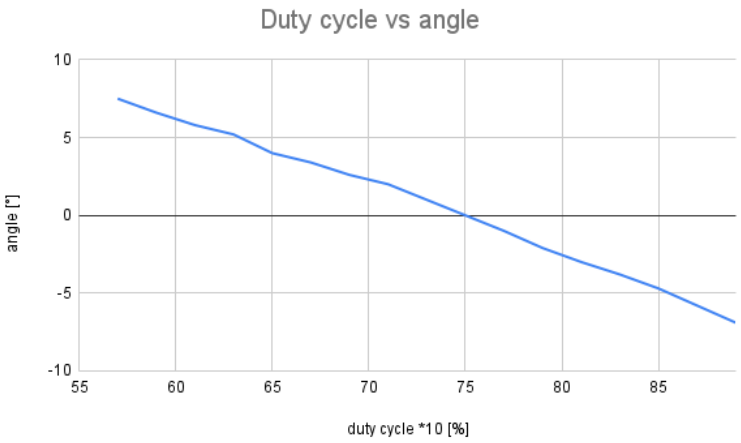


Fig. 6. Duty cycle and angle - linear relationship

To translate the output of PID controller to duty cycle value we defined **linear transform**. Apart from that we defined maximums and minimums for duty cycle and angles. HOME_POS defines the duty cycle corresponding to leveled beam - angle = 0° All mentioned are defined in control_algorithm.h file.

PID controller parameters were selected by applying Ziegler - Nichols method. We set the **Ki** and **Kd** gains to zero and started increasing **Kp** part until we reached constant oscillations. The value of **Kp** at that moment was our ultimate gain **Ku**. Then by plotting the output we determined the period of oscillations **Tu**. After finding two mentioned parameters we used Ziegler - Nichols table. Our choice was control type with some overshoot.

Control type	Kp	Ki	Kd
P	Ku / 2	-	-
PI	Ku / 2.2	1.2Kp / Tu	-
PID	0.6Ku	2Kp / Tu	KpTu / 8
some overshoot	0.33Ku	2Kp / Tu	KpTu / 3
no overshoot	0.3Ku	2Kp / Tu	KpTu / 3

After applying the gains from table, we had to still adjust them slightly to get satisfactory results.

Listing 2. Header file of control algorithm

```
01. /* Includes ------------------------------------------------------------------*/
02. #include "stm32f7xx_hal.h"
03. #include <arm_math.h>
04.
05. /* Define ------------------------------------------------------------------*/
06. #define LINEAR_TRANSFORM(x,amin,amax,bmin,bmax) (((x-amin)/(amax-amin))*(bmax-
    bmin)+bmin)
07. #define DUTY_MAX 87
08. #define DUTY_MIN 55
09. #define ANGLE_MAX 7.5
10. #define ANGLE_MIN -6.9
```

```
#define HOME_POS 73
/* Macro ---------------------------------------------------------------------*/

/* Private variables ---------------------------------------------------------*/

/* Public variables ----------------------------------------------------------*/
uint32_t duty_val = 0;
float32_t SWV_VAR = 0.0f;
float32_t error = 0.0f;
float d = 0.0f;
int sp=10;
arm_pid_instance_f32 pid = {.Kp = 1, .Ki = 0.000147368, .Kd =0.35};

/* Public function prototypes -----------------------------------------------*/


/**
 * @brief PID controller implementation.
 * @note calculation of duty cycle fed to servo
 * @param[in] dist :  current distance of obstacle
 * @param[in] setp :  set point
 * @return duty cycle [% * 10]
 */
int pid_control(float dist, int setp);
```

And the PID control function in control_algorithm.c:

Listing 3. Implementation of pid_control function in control_algorithm.c

```

/**
 * @brief PID controller implementation.
 * @note calculation of duty cycle fed to servo
 * @param[in] dist :  current distance of obstacle
 * @param[in] setp :  set point
 * @return duty cycle [% * 10]
 */
int pid_control(float dist, int setp){

                error = (float32_t)(setp-dist);

                SWV_VAR = arm_pid_f32(&pid, error);

                duty_val = LINEAR_TRANSFORM(SWV_VAR, ANGLE_MIN, ANGLE_MAX,
                    DUTY_MIN, DUTY_MAX);

                if(duty_val <= DUTY_MIN){ // saturation
                        duty_val = DUTY_MIN;
                }
                else if(duty_val >= DUTY_MAX){
                        duty_val = DUTY_MAX;
                }
                if(dist < setp + setp*0.01 && dist > setp- setp*0.01 ){ //
                    deadband
                        duty_val=HOME_POS;
                }
                return duty_val;
        }
```

5 Desktop App

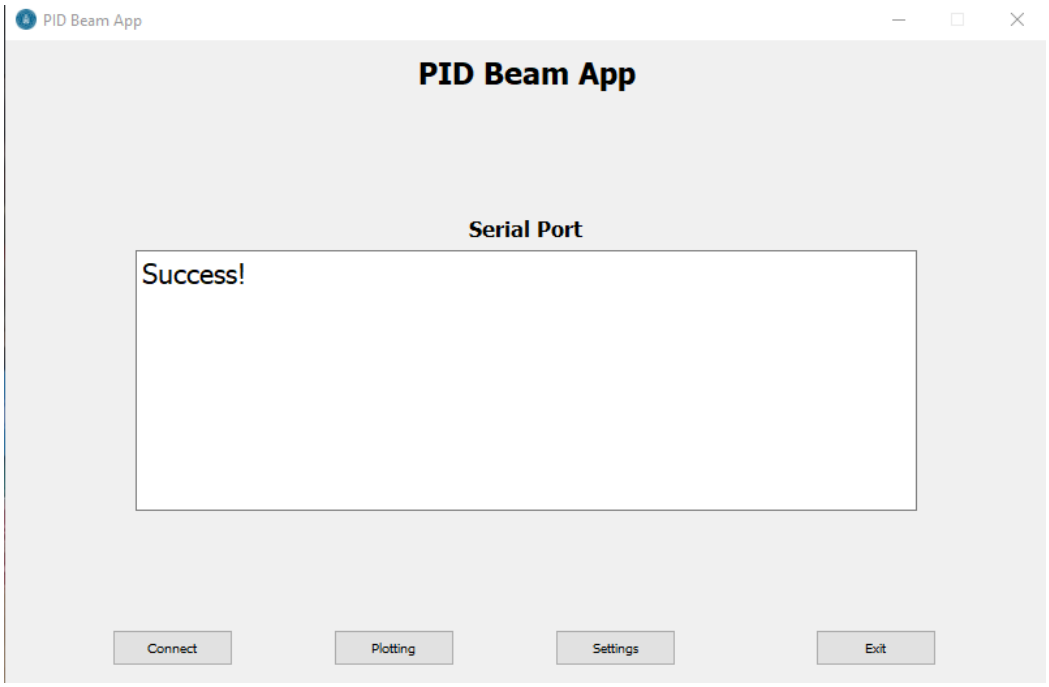
Dedicated application for reading real-time position of object on beam and current set point was written in C++ (QtCreator). Graphical layout was designed in QtDesigner. Application consists of three pages:

- home
- plotting
- settings

Source code of app is available at our github

Home page

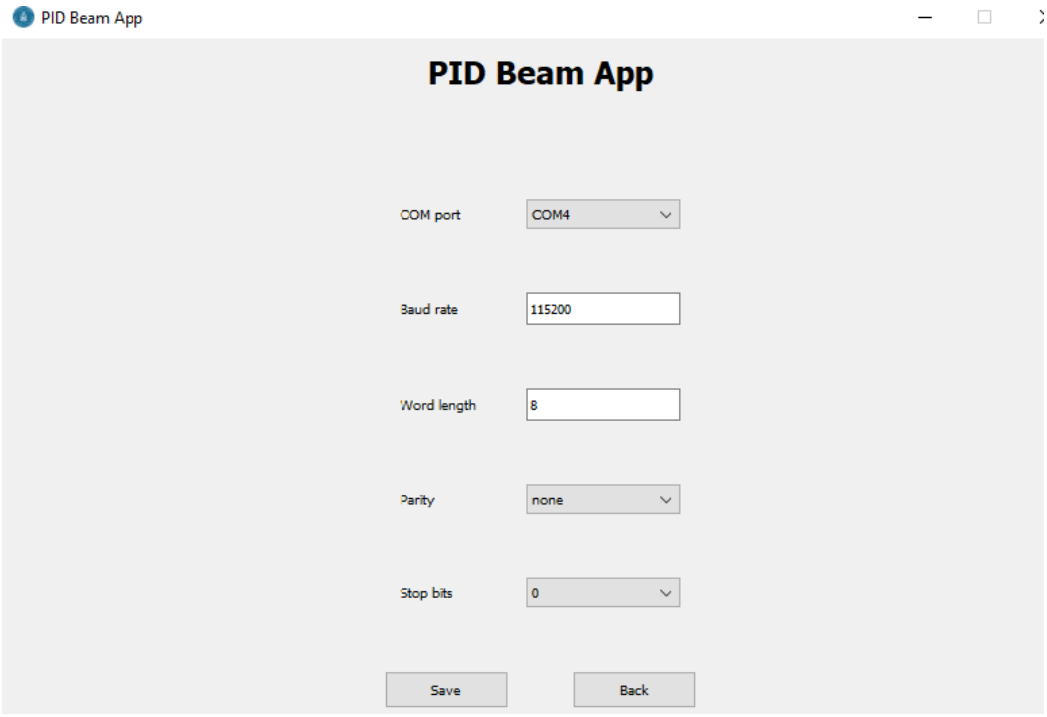
In home page we can connect to default COM port (COM4), read current UART messages, go to plotting page and go to settings

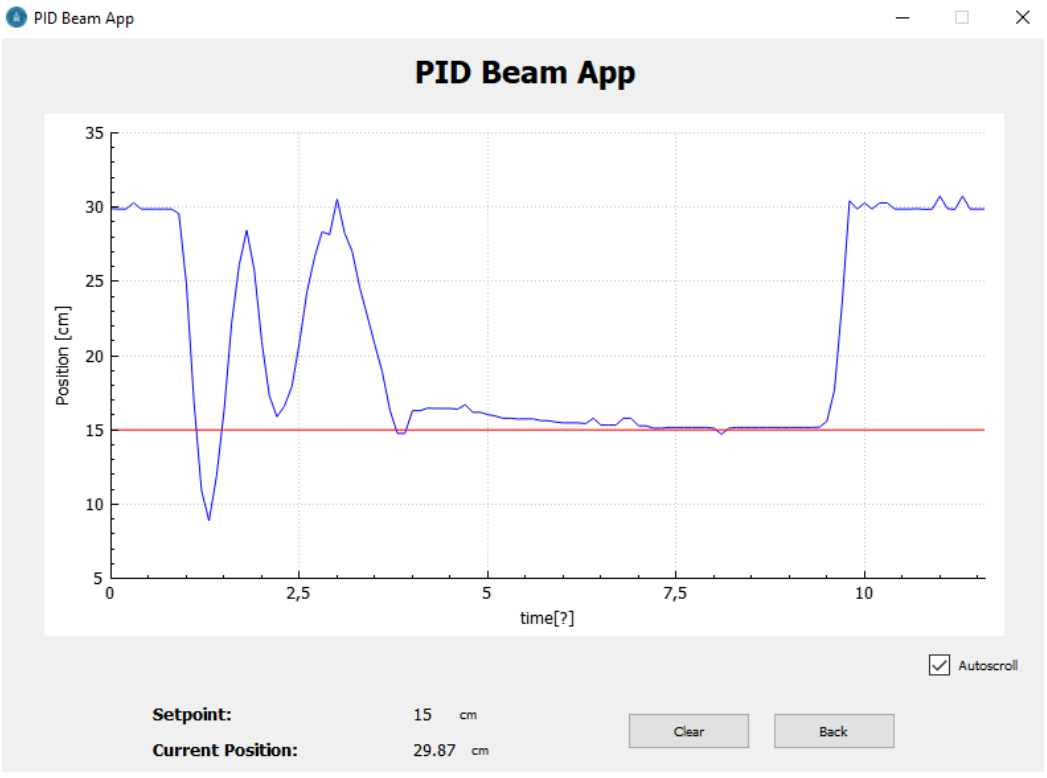


Settings page

In settings page we can select UART connectivity parameters: **Plotting page**

In plotting page we can observe plots of real time position of an object on beam and current set point. Moreover, we have these values presented in numeric form as well.


PID Beam App
PID Beam App
COM port
COM4
Baud rate
115200
Word length
8
Parity
none
Stop bits
0
Save
Back


PID Beam App
PID Beam App
Position [cm]
time[?]
Autoscroll
Setpoint:
15 cm
Current Position:
29.87 cm
Clear
Back

6 Additional User Interface (Potentiometer)

The potentiometer was set up in such a way so that the user could change the setpoint anywhere in the given range of the beam's length, that being 0 to 30 cms. A transformation was done so that the ADC register values could be used. The implementation of the same can be observed in the `ui.c` file.

Listing 4. Implementation of setpoint control function in ui.c

```
01. /**
02.  * @brief Reading ADC register value
03.  * @note reading ADC register value and converting it into setpoint value
04.  * @param[in] hadc : adc handler
05.  * @return setpoint value in [cm]
06.  */
07. int ADC_SETPOINT(ADC_HandleTypeDef* hadc){
08.
09.         if(hadc->Instance == ADC1){
10.
11.          HAL_ADC_Start(&hadc1);
12.
13.          if(HAL_ADC_PollForConversion(&hadc1, 500) == HAL_OK){
14.                          sample_in = HAL_ADC_GetValue(&hadc1);
15.           }
16.         }
17.
18.         return abs((sample_in/100) - 20)+10;
19. }
```

7 Additional User Output devices (LCD, LEDs)

As additional user output devices a set of three LEDs have been set up and they function in the following way, such that if the specific LED is switched on the the following is indicated to the user:

- Green LED: Less than 1% steady-state error.
- Blue LED: Less than 5% steady-state error.
- Red LED: Over 5% steady-state error.

An LCD display has also been used which displays to the user two most crucial values of the system, which is the desired setpoint and the current position of the obstacle on the beam.
The implementation of the same in the `ui.h` file is as follows:

Listing 5. Implementation of LCD and LEDs in ui.h file

```
01. /**
02.   ******************************************************************************
03.   * @file    ui.h
04.   * @authors  NS        Nihal.Suri@student.put.poznan.pl     MM          Maciej.
        Mirecki@student.put.poznan.pl
05.   * @version 2.0
06.   * @date    28-12-2021
07.   * @brief   User Interface: led control, lcd control, ADC input.
08.   *
09.   ******************************************************************************
10.   */
11.   /* Config ------------------------------------------------------------------
        */
12.
13. /* Includes ------------------------------------------------------------------*/
14. #include "stm32f7xx_hal.h"
15. #include <math.h>
16. #include <arm_math.h>
17. #include <lcd.h>
18. #include <adc.h>
19. /* Define ------------------------------------------------------------------*/
20.
21. /* Macro -------------------------------------------------------------------*/
22.
23. /* Private variables ---------------------------------------------------------*/
24.
25. char duty_print[10];
26. char pos_dist[10];
27. char sp_str[];
28. uint32_t sample_in;
29. /* Public variables ----------------------------------------------------------*/
30. float32_t error;
31. float d;
32. uint32_t duty_val;
33. int sp;
34. /* Public function prototypes -----------------------------------------------*/
35.
36.
37. /**
38.  * @brief LED control
39.  * @note green (<0.1% error) blue (0.3 - 0.5% error) red (>0.5% error)
40.  * @param[in] none
41.  * @return None
42.  */
43. void led_routine();
44.
45.
```

```
/**
/**
 * @brief Prints data on LCD display
 * @note printing real-time position and current setpoint
 * @param[in] hlcd1 : lcd handler
 * @return None
 */
void lcd_routine(LCD_HandleTypeDef* hlcd1);
```

The implementation of the same in the `ui.c` file is as follows:

Listing 6. Implementation of LCD and LEDs in ui.c file

```
/**
  ******************************************************************************
  * @file    ui.c
  * @authors  NS        Nihal.Suri@student.put.poznan.pl     MM          Maciej.
      Mirecki@student.put.poznan.pl
  * @version 2.0
  * @date    28-12-2021
  * @brief   User Interface: led control, lcd control, ADC input.
  *
  ******************************************************************************
  */
  /* Config ------------------------------------------------------------------
      */

/* Includes ------------------------------------------------------------------*/
#include <ui.h>

/* Define ------------------------------------------------------------------*/

/* Macro ------------------------------------------------------------------*/


/* Public variables ---------------------------------------------------------*/


/* Public function prototypes -----------------------------------------------*/


/**
 * @brief LED control
 * @note green (<0.1% error) blue (0.3 - 0.5% error) red (>0.5% error)
 * @param[in] none
 * @return None
 */
void led_routine(){

          if(fabs(error) <= 0.3){
                  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 0);
                  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_3, 0);
                  HAL_GPIO_WritePin(GPIOD, GPIO_PIN_7, 1);
                  }

          else if( fabs(error) > 0.31 && fabs(error) < 0.6){
                        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 0);
                        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_3, 1);
                        HAL_GPIO_WritePin(GPIOD, GPIO_PIN_7, 0);
                }
          else    if( fabs(error) > 0.6){
                        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 1);
                        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_3, 0);
                        HAL_GPIO_WritePin(GPIOD, GPIO_PIN_7, 0);
```

```
50.                 }
51. }
52.
53. /**
54.  * @brief Prints data on LCD display
55.  * @note printing real-time position and current setpoint
56.  * @param[in] hlcd1 : lcd handler
57.  * @return None
58.  */
59. void lcd_routine(LCD_HandleTypeDef* hlcd1){
60.           sprintf(pos_dist, "%.02f", d);
61.           sprintf(duty_print, "%lu", duty_val);
62.           sprintf(sp_str, "%d", sp);
63.           LCD_SetCursor(hlcd1, 0, 0);
64.           LCD_printStr(hlcd1, "DISTANCE: ");
65.           LCD_printStr(hlcd1, pos_dist);
66.           //LCD_printStr(hlcd1, "cm");
67.           LCD_SetCursor(hlcd1, 1, 0);
68.           LCD_printStr(hlcd1, "SETPOINT: ");
69.           LCD_printStr(hlcd1, sp_str);
70.
71. }
```

8 Additional Simulations

MATLAB and Simulink was used to better understand the working of our system and run simulations for the tuning purposes. The PID gain values obtianed from these simulations were not actually used for our system, but this helped us in the visualization purposes.
The MATLAB code is as follows:

```
clear all;
close all;

% PARAMETERS
m = 0.11;    % mass of the  object[kg]
R = 0.015;    % radius of the object[m]
d = 0.03;    % lever arm offset [m]
g = 9.8;     % gravitational acceleration
L = 1.0;     % length of the beam
J = 9.99e-6;     % object's moment of inertia
r = 0;       % object position coordinate
alpha = 0;   % beam angle (radians)
theta = 0;   % servo angle (radians)

% TRANSFER FUNCTION
num = m*g*d;
den = [L*((J/R^2) + m) 0 0];
Gs = tf(num, den) % [m/rad]

% STATE SPACE REPRESENTATION
H = -m*g/(J/(R^2)+m);
A = [0 1 0 0; 0 0 H 0; 0 0 0 1; 0 0 0 0]
B = [0 0 0 1]';
C = [1 0 0 0];
D = [0];
object_ss = ss(A,B,C,D)
```

The simulink model designed to tune the PID gains can be observed in figure 7.

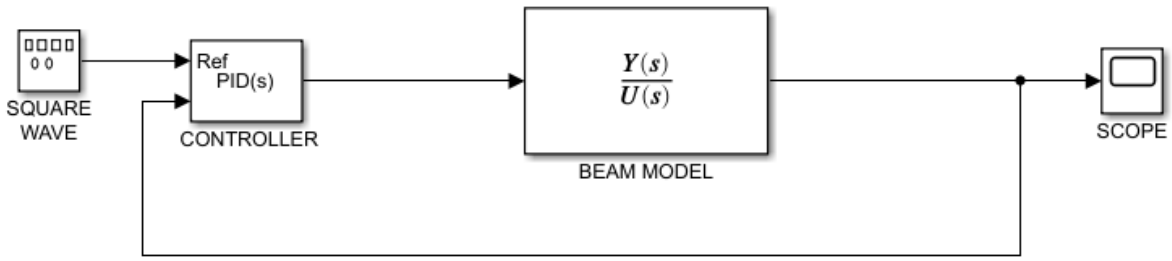


Fig. 7. Simulink Model to tune PID values

1.3 Hardware Applications

1 CAD Models

SolideEdge 2022 was used to design the CAD models for this project, because of the CAD models we were able to understand how and were to place the sensor on our mechanical structure, the movement of the obstacle on the surface and the overall dimensions of the structure. The models can be observed in figure 8, 9 and 10.

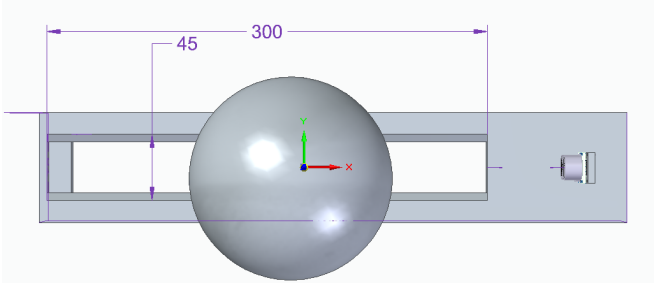


Fig. 8. Top View of the Model

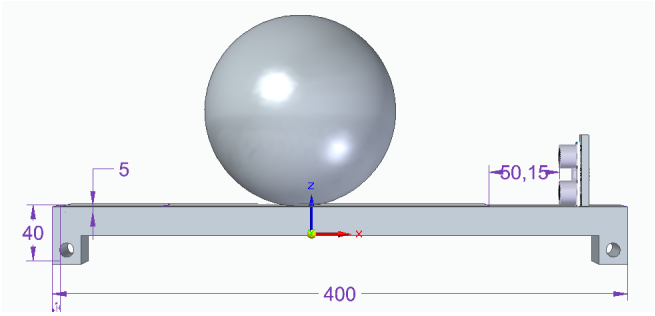


Fig. 9. Front View of the Model

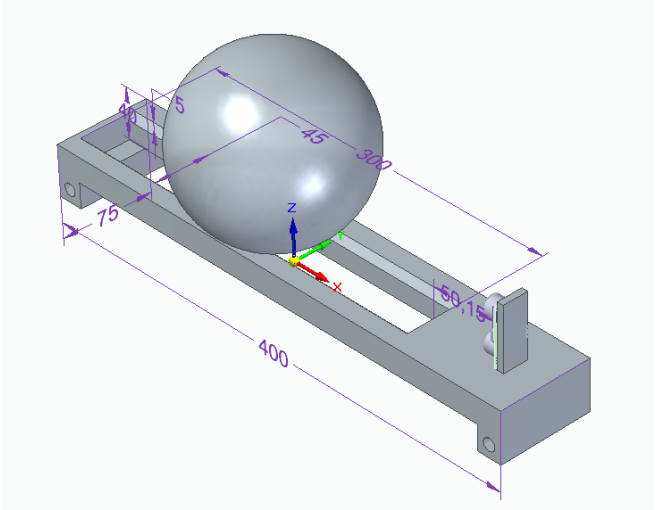


Fig. 10. View from home position of the Model

2 Mechanical Structure

The idea of the mechanical structure came from several online resources and a crucial one being the following, from this point on we understood that our model has to be a four-bar linkage and we started to think about how we can go about the build for this system, we chose wood as our material for the frame as it would provide us with efficient and fast prototyping results. The photos and videos of how our system operates can be viewed in the results section.

1.4 Results

Google drive with photos of videos

1.5 Conclusion

The following where the most challenging parts of the task for us:

- Mechanical Structure: As we didn't have much knowledge about linkage mechanisms it was crucial for us to study and build the system as soon as possible, because only then we could start with testing our software applications.
- Control Algorithm: We had studied about various control algorithms in previous subjects but only theoretically, this was the first time we had implemented a popular algorithm like PID on an actual system and see how tuning the parameters affected a live system.
- Desktop App: We had some experience with C++ from first semester programming course, so developing the app with user interface was straightforward, but the challenge came when we had to implement serial port interface. There were some minor problems with overflowing buffer and the expected messages were sometimes sent in parts, which caused faulty values on real time plot. We overcame that, by adding small delay, specifying the length and correct form of message - **sscanf()** turned out to be very useful.

Summary

This project was an idea that we wanted to build for sometime and bringing it to realization was a great learning experience. In general, development of project's software part was quite fast, as we had great experience from laboratory classes with timer callbacks, handling GPIOs and using ADC on STM32. Thanks to that we could focus additional features like Doxygen standard or developing dedicated application. PID control algorithm implementation for real system was challenging, but it helped us to understand its operation even better. We hope to develop this project into something bigger in the future, maybe as our engineering thesis.