

Embedded Control Laboratory

Ball on inclined plane

Report of lab exercise 3

Calibration and control of the real BoIP system with a PID-controller in Embedded-C language.

WS 2019/2020

Group 04

1536988 Venkata Sai Tarak Padarathi

1536199 Kaushik Goud Chandapet

1533846 Sai Radha Krishna Vangaveeti

Date: 26.01.2020

Contents

1. Introduction -----	3
2. Objectives -----	4
3. Model Calibration -----	5
4. Modifying the Welcome message -----	8
5. PID Equations -----	8
6. Queues and Tasks -----	10
7. PID Tuning -----	12
8. Conclusion -----	12

1. INTRODUCTION:

1.1 Embedded C:

There are different programming languages used for different applications. For embedded systems Embedded C programming is used which is different from C programming. One such difference is Embedded C programming depends upon the hardware architecture whereas C programming doesn't depend upon the hardware architecture. Embedded C is having more header files than C programming. It is used for resources like RAM and ROM^[1].

1.2 FreeRTOS:

FreeRTOS is a real time operating system. It is used to execute tasks in Embedded systems. FreeRTOS has a scheduler where we give the priorities of tasks to be performed. Then it performs the tasks accordingly. FreeRTOS provides the core real time scheduling^[2]. RTOS will use both ROM and RAM. ROM is used to store the program code. This code is copied to RAM for improved performance.

1.3 AVR studio:

AVR studio is an integrated development environment used to write and debug the AVR microcontroller applications. It is easy to use for writing and debugging the applications written in C/C++^[3].

1.4 PWM working:

Pulse width modulation is digital because the DC supply is either fully ON or OFF. PWM signals are square waves. This technique is used to reduce power dissipation. The conventional circuit which when the current increases, the power dissipation also increases. But, in this case we use MOSFET for less power dissipation. This technique also uses the complete current as the voltage occurs completely when the device is on. This technique also has Duty cycle, which can be controlled. If we consider an example of LED. We can switch on and off the LED. The amount of time we switch the LED on and off can be represented in Square wave^[4]. From Fig. 1 shown below, we can say that Duty cycle is the percentage of on time with respect to off time. So, by using the PWM we can control the brightness of LED.

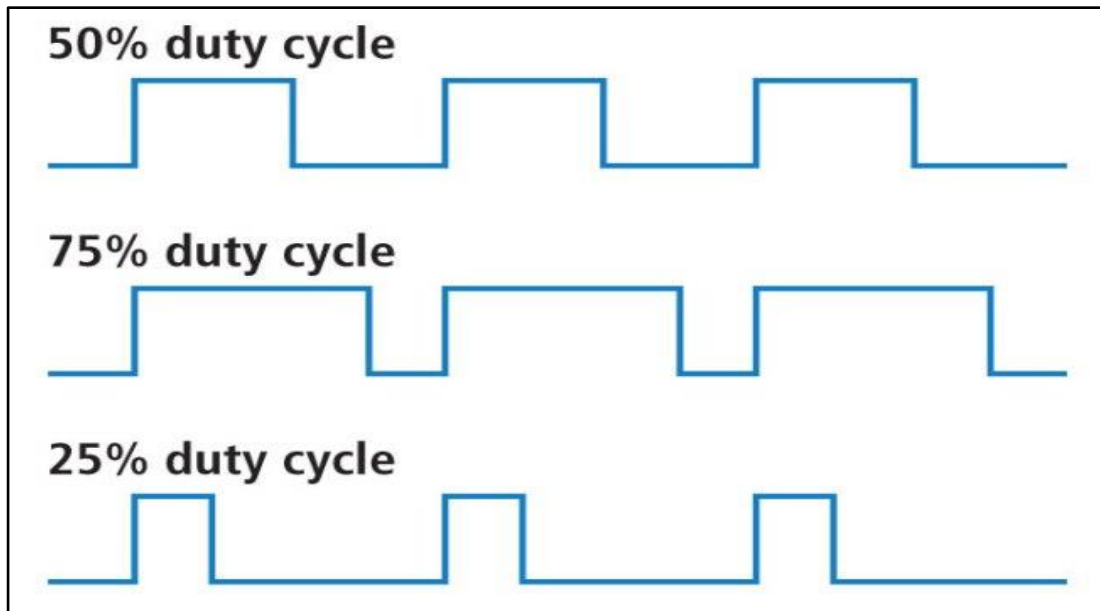


Fig. 1: Pulse Width Modulation

2. OBJECTIVE:

The objective of the experiment is to control the ball position on an inclined plane by using ATMEGA 128 Microcontroller and to develop the program using PID equations.

Tasks:

- Calculation of the theoretical value of MIDPOS and DELTA.
- Calibration the system to find the actual value of MIDPOS.
- Modification of the welcome message of the LCD.
- Programming a discrete PID-controller in Embedded C language and implementation on the real BoIP system.
- Tuning of these PID parameters.

3. MODEL CALIBRATION

Every physical model includes some kind of errors within it (e.g., error due to tolerance). Hence calibration is a necessary step. In this section, calibration of our BoiP model is discussed in detail along with the description of necessary registers. The details, specifications and various registers present in the microcontroller can be found in datasheet of ATmega128. These datasheets are free to access and are provided by the manufacturer of the microcontroller.

Every microcontroller runs with the help of a system clock frequency. It is possible to alter this system clock frequency depending on the requirement and application. The system clock frequency is divided by a pre-scalar value to achieve the operating clock frequency.

3.1 Timer/Counter Control Register (TCCR3B)

TCCR3B is an 8 bit register used to alter the clock frequency to control the Servo motor. The table below shows the pre-scalar values offered by the microcontroller. The selected pre-scalar value is divided with input/output system clock ($\text{clk}_{\text{I/O}}$) to get the required/operating clock frequency. This can be achieved by altering the bits 0, 1 and 2 of the register. In the current case, a pre-scalar value of 8 is selected. So the bit values of 010 are loaded into last three bits of TCCR3B register.

Bit 2	Bit 1	Bit 0	Description
0	0	0	No i/o system clock ($\text{clk}_{\text{I/O}}$)
0	0	1	$\text{clk}_{\text{I/O}} / 1$
0	1	0	$\text{clk}_{\text{I/O}} / 8$
0	1	1	$\text{clk}_{\text{I/O}} / 64$
1	0	0	$\text{clk}_{\text{I/O}} / 256$
1	0	1	$\text{clk}_{\text{I/O}} / 1024$

Table. 1: Pre-scalar values in ATmega128 microcontroller

System Clock frequency = 16 MHz.

Pre-scalar value = 8

Required/Operating frequency = 16 MHz / 8 = 2 MHz (3.1)

3.2 Servo Motor Control using PWM

Direction and angle of rotation of the servo motor arm can be controlled by duration of the pulse width (output signal from microprocessor to control the servo motor arm). Left, middle and right positions are achieved at 1 ms, 1.5 ms and 2 ms respectively. These pulses repeats for every 18 ms (i.e., cycle time is 18 ms). The left and right positions are achieved by giving a gain equivalent to 0.5 ms when the arm is at middle position. This gain is denoted by DELTA.

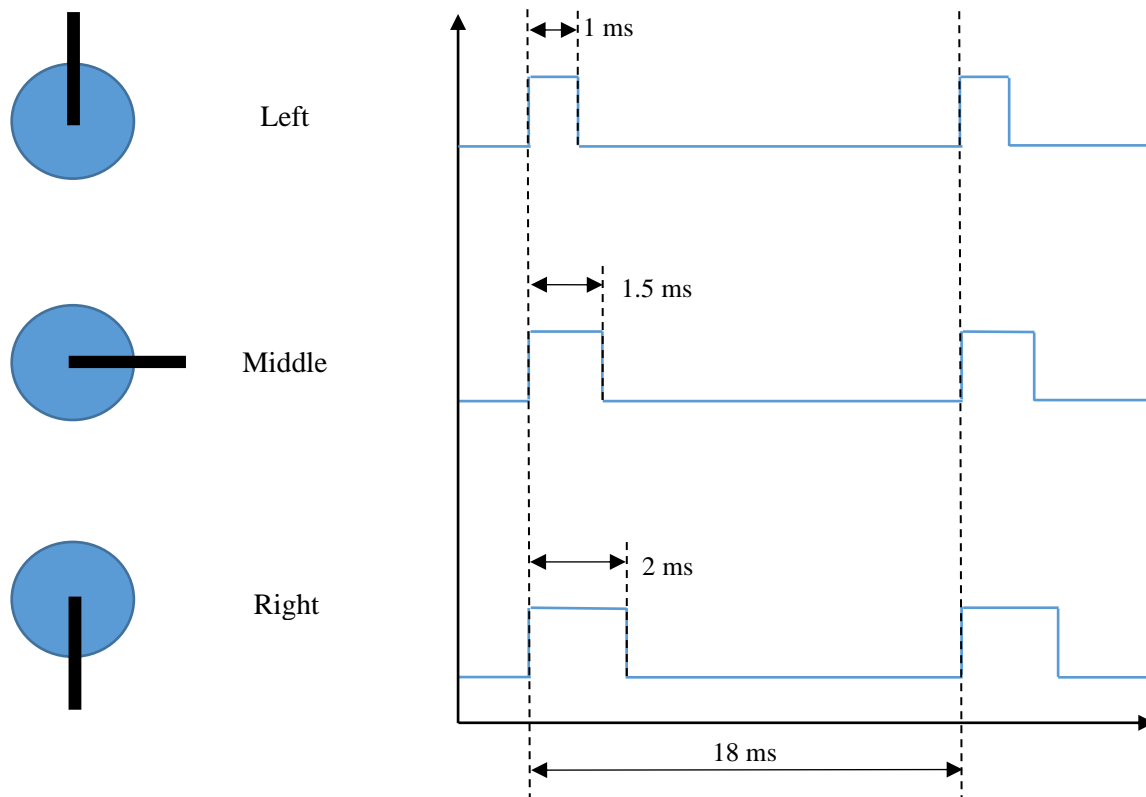


Fig. 2: Servo motor arm positions and respective pulse widths

3.3 Servo Motor Control and Middle Position (MIDPOS)

Pulse width value to attain the middle position of the motor is 1.5 ms and the gain value DELTA is 0.5 ms. The registers *Output Compare Register* (OCR3A) along with *Timer/Counter register* (TCNT3) are used to accurately measure time with the help of pulses. Operating frequency of the controller 2 MHz signifies that in one second, 2 Million pulses are generated. Below is the calculation showing number of pulses generated during 1.5 ms and 0.5 ms

Operating frequency = 2 MHz

Number of pulses generated per second = 2×10^6

Pulse ON time for MIDPOS = 1.5 ms = 1.5×10^{-3} sec

Number of pulses generated during pulse ON time = $(2 \times 10^6) \times (1.5 \times 10^{-3}) = 3000$ (3.2)

Pulse ON time for DELTA = 0.5 ms = 0.5×10^{-3} sec

Number of pulses generated during pulse ON time = $(2 \times 10^6) \times (0.5 \times 10^{-3}) = 1000$ (3.3)

3.4 Timer/Counter register (TCNT3)

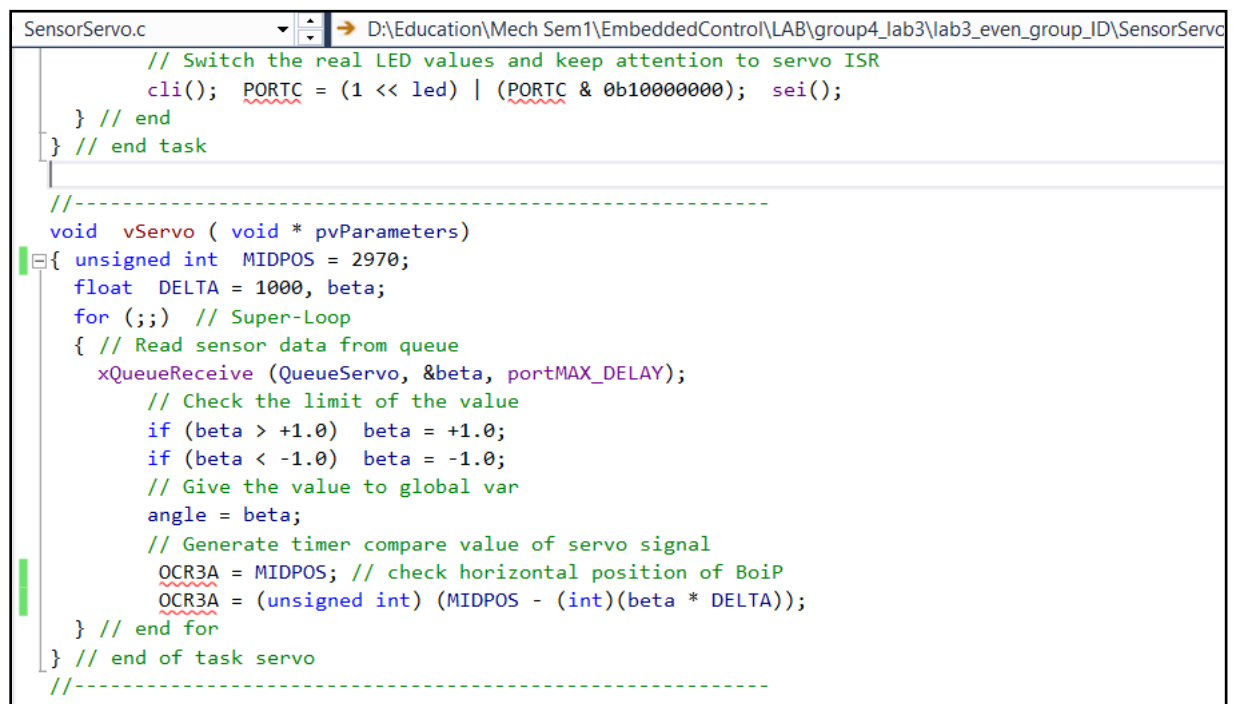
TCNT3 is a 16 bit register used to read the number of pulses. Starting from the value 0, for every pulse, the counter value is increased by 1. Thereby, counting the number of pulses generated during a time period. Reset condition of this register is discussed in the Output Compare Register (OCR3A) section below.

3.5 Output Compare Register (OCR3A)

OCR3A is a 16 bit register used to store the value that needs to be compared with TCNT3 register. As the counter value in TCNT3 increases with each pulse, it will be compared with the given value in OCR3A register. When the values in both the registers becomes equal, an output compare interrupt is generated and the TCNT3 register is reset to 0. In our case (refer eq. 3.2), the previously calculated value of 3000 (hereafter, referred to as theoretical MIDPOS) is loaded into OCR3A register.

3.6 Calibration for actual MIDPOS

After loading the theoretical MIDPOS value, it can be observed that at the middle position of the motor, the plane is not necessarily horizontal. So theoretical MIDPOS value is tuned to achieve horizontal position of the plain, checking with the help of Vernier callipers. This tuned MIDPOS value is called actual MIDPOS and is then loaded into OCR3A register as shown in Fig. 3. The actual MIDPOS value is 2970.



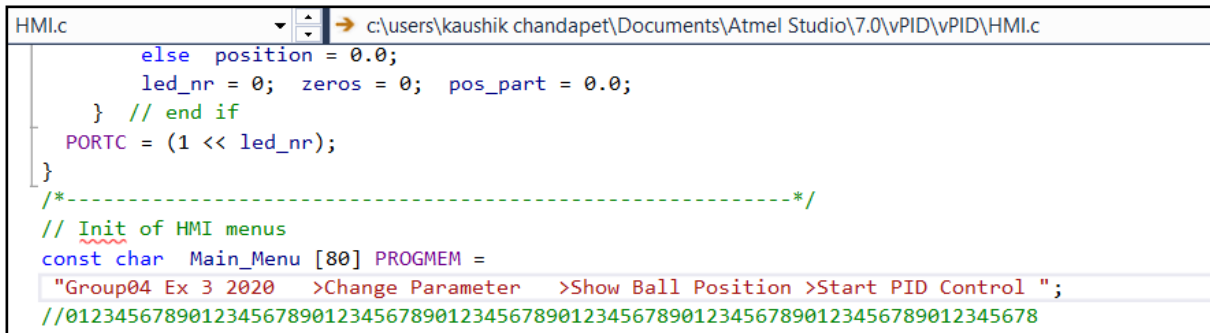
```
SensorServo.c
// Switch the real LED values and keep attention to servo ISR
cli(); PORTC = (1 << led) | (PORTC & 0b10000000); sei();
} // end
} // end task

//-----
void vServo ( void * pvParameters)
{
    unsigned int MIDPOS = 2970;
    float DELTA = 1000, beta;
    for (;;) // Super-Loop
    { // Read sensor data from queue
        xQueueReceive (QueueServo, &beta, portMAX_DELAY);
        // Check the limit of the value
        if (beta > +1.0) beta = +1.0;
        if (beta < -1.0) beta = -1.0;
        // Give the value to global var
        angle = beta;
        // Generate timer compare value of servo signal
        OCR3A = MIDPOS; // check horizontal position of BoiP
        OCR3A = (unsigned int) (MIDPOS - (int)(beta * DELTA));
    } // end for
} // end of task servo
//-----
```

Fig. 3: Loading actual MIDPOS to OCR3A register

4. Modifying the Welcome message of LCD

The welcome message has been modified to “Group04 Ex 3 2020” by editing the HMI.c file as shown in the Fig. 4.



```
HMI.c
c:\users\kaushik chandapet\Documents\Atmel Studio\7.0\vPID\vPID\HMI.c

    else position = 0.0;
    led_nr = 0; zeros = 0; pos_part = 0.0;
  } // end if
  PORTC = (1 << led_nr);
}

/*-----*/
// Init of HMI menus
const char Main_Menu [80] PROGMEM =
"Group04 Ex 3 2020  >Change Parameter  >Show Ball Position >Start PID Control ";
//012345678901234567890123456789012345678901234567890123456789012345678
```

Fig. 4: Modification of welcome message

5. PID EQUATIONS:

Ideal PID equation is given by the following formula^[5]. Here $u(t)$ represents the control signal and $e(t)$ represents the control error.

$$u(t) = k_p * e(t) + k_i * \int e(T) dT + k_d * (de/dt) \dots\dots\dots (5.1)$$

The control signal is the sum of three terms. First term is proportional term which is proportional to the error and integral term is proportional to the integral of error and derivative term will be proportional to the derivative of the error^[5]. The representation of controller in another form is given below.

$$u(t) = k_p * (e(t) + (1/T_i) * \int e(T) dT + T_d * (de(t)/dt)) \dots\dots\dots (5.2)$$

The proportional term acts on present errors. The integral term will represent the average of past errors. The derivative term represents the prediction of future errors.

T_i is integral time. Increment in T_i shows that output responds slower to an error.

T_d is the differential time. This represents speed with which the Process Variable responds to the change in controller output^[5].

By applying Laplace transform to equation 5.1 gives an equation in s domain^[6].

$$U(s)/E(s) = k_p + (k_i/s) + k_d * s \dots\dots\dots (5.3)$$

Derivative term is modified to low pass filter so that it becomes less noisy. Equation 5.3 is modified to equation 5.4 as shown below^[6].

$$C(s) = U(s)/E(s) = k_p + (k_i/s) + (N * k_d / (1 + (N/s))) \dots\dots\dots (5.4)$$

5.1 DISCRETE PID EQUATION:

As computer cannot understand these terms. So, we make a transition from continuous to discrete PID in understandable form. The resulting Discrete PID equation obtained from Continuous PID equation is given below. Here $k_P = k_{PID}$, $k_I = K_{PID}/T_N$ and $k_D = k_{PID} * T_N$

$$u_k = k_{PID} * (e_k + (1/T_N) * \sum_{i=0}^{k-1} e_i * \Delta t + T_N * (e_k - e_{k-1}) / \Delta t) \dots\dots\dots (5.5)$$

Equation 5.5 shows the Discrete PID equation. Using equation 5.5 we implemented the program as shown below.

5.2 IMPLEMENTATION:

We implemented final PID equations in the program and tested it on the real model. The PID.c file is used to implement the program.

```
/*
 * PID.c
 *
 * Created: 10.08.1921 12:40:11
 * Author: Henry Ford
 */
// Compiler includes
#include <math.h>
#include <avr/io.h>
// FreeRTOS includes
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
// Modul includes
#include "global.h"
#include "PID.h"
//-----

void vPID ( void * pvParameters)
{
    float x = 0.0, y = 0.0, p = 0.0, i = 0.0, d = 0.0, t = 0.018 ;
    // Place here your local parameters
    //float proportional = 0.900;
    //float integral = 0.022;
    //float derivative = 0.370;
    float previous_error = 0.0, error = 0.0, total_error = 0.0;

    // Super loop of task
```

```

for (;;)
{ // Get position value from sensor queue

    xQueueReceive (QueueSensor, &x, portMAX_DELAY);

    /*      Compute here your digital PID formula with the
            help of following global variables:
            reference, proportional, integral, derivative
            and please remember the sampling time
    */

    // For example a simple P control startup formula

    error = (reference - x);
    p = proportional * error;
    i = integral * t * (error + total_error);
    d = derivative * (error - previous_error) / t;

    total_error = error + total_error;
    previous_error = error;

    y = p + i + d;

    // Write angle value to servo queue
    xQueueSend (QueueServo, &y, portMAX_DELAY);

} // end for loop

} // end of task PID

```

The program is then flashed to ATMEGA 128 Microcontroller through AVR studio with the help of an ISP.

6. QUEUES AND TASKS:

Every action that needs to be performed is sequentially programmed into Tasks. Their order of execution is decided by task scheduler. Queues are used to communicate between any two tasks. Declarations of all these tasks and Queues is done under main program. At the end of each task, the output of that task is sent to a Queue using **xQueueSend** command with a wait period of some ticks. Then the RTOS will wake **xQueueRecieve** and corresponding task will be initialised^[7].

Queues and tasks that are used in our program are:

Queues: QueueTaster; QueueSensor; QueueServo

Tasks: vSensor; vTaster; vHMI; vServo; vPID.

6.1 Flow of Data exchange:

At the start of the program “vTaskStartScheduler()” starts all the tasks. ”vSensor” task is first called and with the help of photo sensors that are connected to the PORTC of the Microcontroller board, vSensor task determines the position of the ball. The position of the ball is then set to the parameter “smooth_pos” and this value is sent to “QueueSensor”, which is then received by the task “vPID”.

In the vPID task, the PID implementation is done and the output is sent to “QueueServo”. The output is received by “vServo”. Here, the output signal is compared with the Midpos value and generates OCR3A signal as per Pulse Width Modulation. The Servo motor arm moves thereby adjusting the ball position on the inclined plane. This process is continued till the ball is reached to desired position(Steady state).

The other tasks in our program include:

”vTaster” that records the user events such as pressing up/down/side buttons to change the PID parameters or the reference value. The event is then sent to “QueueTaster” and is received by “vHMI”. This task analyses the event and gives related information to the respective pins. The parameters will be updated to the microcontroller, changes display on LCD and sends updated parameter values to “vPID”. And the cycle continues.

6.2 UML Sequence diagram

The UML sequence diagram provides a sequential representation of the system under consideration. Though it does not define the algorithms nor provides a detailed overview of the system, it is enough for universal understanding of the working of the system. The Fig.5 indicates the graphical representation of tasks and queues in a program.

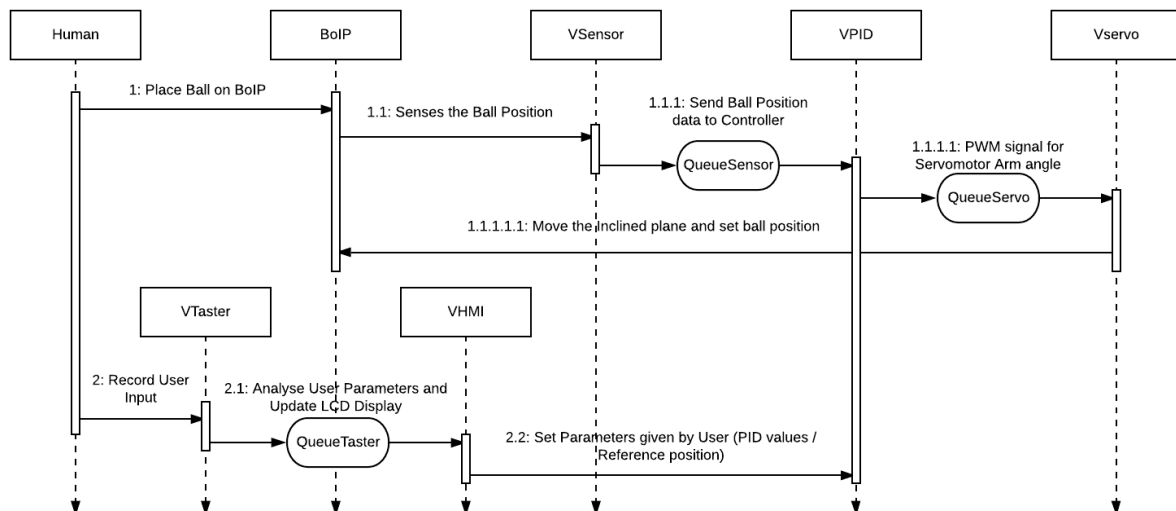


Fig. 5: UML Sequence diagram of BoIP system

7. PID TUNING:

The term PID tuning refers to finding out the good working parameters of a PID controller i.e the values of proportional gain, integral gain and derivative gain. This can be achieved by either manually i.e by trial and error method which involve a lot of experience in this field or by some already established step by step methods.

PID parameters for balancing the ball on desired position in BoiP model :

For balancing the ball, we followed a step by step procedure to settle down the ball on desired position :

- a) Set all gains to zero.
- b) Increase only the proportional gain until the oscillations are stable.
- c) Then start increasing derivative gain to minimise the oscillations
- d) On repeating steps b & c we will get the minimum oscillations for some proportional gain and derivative gain.
- e) Then increase the integral gain to achieve the goal in minimum time.

On following these steps, for the PID parameters $P=0.9$, $I=0.022$, $D=0.37$, an optimum time of 6 seconds is obtained to stabilize the plastic ball at the mid position.

7.1 PID TABLE:

TIME	PID INPUT
6	0.000
7	0.045
10	0.000
7	0.032
5	0.000
4	0.000
4.7	0.000
4.5	0.000
3.8	0.040
6	0.000

Table. 2: Iterations for P, I and D values

8. CONCLUSION:

From the experiment, we found out that for the PID parameters $P=0.9$, $I=0.022$, $D=0.37$, the system took an optimum time of 6 seconds to stabilize the ball at the given reference position.

References:

1. <https://www.elprocus.com/basics-and-structure-of-embedded-c-program-with-examples-for-beginners/>
2. <https://www.freertos.org/about-RTOS.html>
3. <https://www.microchip.com/mplab/avr-support/atmel-studio-7>
4. <https://learn.sparkfun.com/tutorials/pulse-width-modulation/all>
5. http://www.cds.caltech.edu/~murray/books/AM08/pdf/am06-pid_16Sep06.pdf
6. <https://www.scilab.org/discrete-time-pid-controller-implementation>
7. <https://www.freertos.org/Embedded-RTOS-Queues.html>