Ball on Inclined Plane

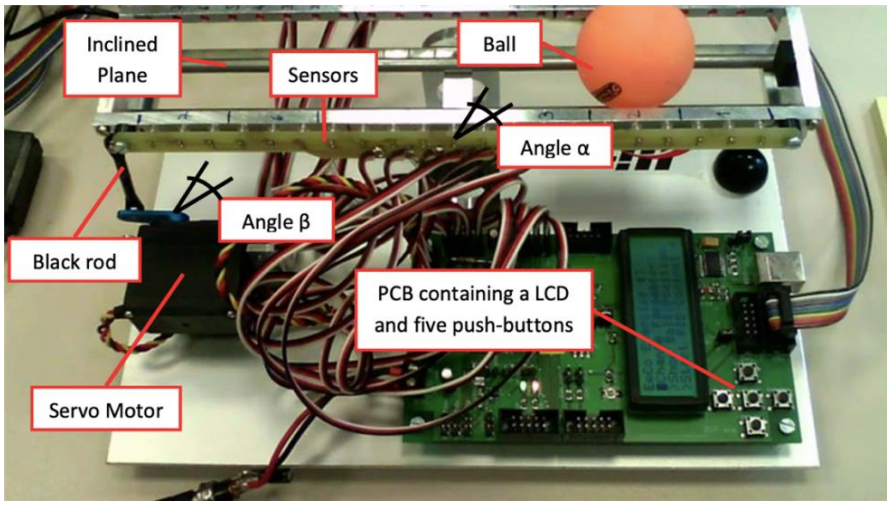


Fig.1: Ball-on-inclined-plane with sensors, an actuator (servo motor) and a PCB

1. Objectives

1) Run/build successfully the current project code with the delivered simple P control task.

2) In the servo-motor task

   a) Calculate correct value for the middle-position (MIDPOS) of the inclined plane with respect to pulse length, timer and pre-scaler.
   b) Find the correct value of the variable DELTA, which represent the gain necessary to move the arm to the right or left position.

3) Modify the welcome message of the LCD, print the following message: " Group31 Ex 3 2019".

4) Implement your solution for the discrete PID control task in embedded-C (in PID.c file)

   a) Test your code on the real model.
   b) Find appropriate PID parameters to control the system.
   c) Goal: The ball must be settled at the reference position within 10 seconds.

5) Use graphical representation (e.g. UML diagrams) to describe the following:
   a) Identify and explain the tasks and timings in the program.
   b) Identify and explain the queues in the program.
   c) Identify and explain the exchanged data between different tasks and queues.

2. Introduction

2.1 Embedded C

- Embedded C is a set of language extensions for the C programming language by the C Standards Committee to address commonality issues that exist between C extensions for different embedded systems.[1]

- Embedded C is an extension to C programming language that provides support for developing efficient programs for embedded devices.

- Embedded C programming typically requires nonstandard extensions to the C language in order to support enhanced microprocessor features such as fixed-point arithmetic, multiple distinct memory banks, and basic I/O operations [1].

- It includes a number of features not available in normal C, such as fixed-point arithmetic, named address spaces and basic I/O hardware addressing [1].

- Embedded C uses most of the syntax and semantics of standard C, e.g., main() function, variable definition, datatype declaration, conditional statements (if, switch case), loops (while, for), functions, arrays and strings, structures and union, bit operations, macros, etc.[2]

2.2 Free RTOS

- It is a Real-time operating system for microcontrollers distributed freely under the MIT open source license, hence the name Free RTOS.

- It is a real-time operating system (RTOS) for microcontrollers and small microprocessors.

- It has been developed in partnership with the world leading chip companies over a 15-year period.

- It is very reliable and is the most widely used RTOS.

- With proven robustness, tiny footprint, and wide device support, the FreeRTOS kernel is trusted by world-leading companies as the de facto standard for microcontrollers and small microprocessors [3]

2.3 ATMEL/AVR Studio

AVR is a family of microcontrollers developed since 1996 by Atmel. These are modified Harvard architecture 8-bit RISC single-chip microcontrollers. AVR was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to one-time programmable ROM, EPROM, or EEPROM used by other microcontrollers at the time [4].

Studio 7 is the integrated development platform (IDP) for developing and debugging all AVR® and SAM microcontroller applications. The Atmel Studio 7 IDP gives you a seamless and easy-to-use environment to write, build and debug your applications written in C/C++ or assembly code. It also connects seamlessly to the debuggers, programmers and development kits that support AVR® and SAM devices [5].

2.4 ATMEGA 128

The high-performance, low-power Microchip 8-bit AVR RISC-based microcontroller combines 128KB of programmable flash memory, 4KB SRAM, a 4KB EEPROM, an 8-channel 10-bit A/D converter, and a JTAG interface for on-chip debugging. The device supports throughput of 16 MIPS at 16 MHz and operates between 4.5-5.5 volts [6].

Name	Value
Program Memory Type	Flash
Program Memory Size (KB)	128
CPU Speed (MIPS/DMIPS)	16
SRAM (B)	4,096
Data EEPROM/HEF (bytes)	4096
Digital Communication Peripherals	2-UART, 1-SPI, 1-I2C
Capture/Compare/PWM Peripherals	2 Input Capture, 2 CCP, 8PWM
Timers	2 x 8-bit, 2 x 16-bit
Number of Comparators	1
Temperature Range (°C)	-40 to 85
Operating Voltage Range (V)	2.7 to 5.5
Pin Count	64

Table 1: Atmega 128 Parameters

By executing instructions in a single clock cycle, the device achieves throughputs approaching 1 MIPS per MHz, balancing power consumption and processing speed.

2.5 PWM – Pulse Width Modulation

Pulse width modulation (PWM) is a powerful technique for controlling analog circuits with a microprocessor's digital outputs.

In a nutshell, PWM is a way of digitally encoding analog signal levels. Through the use of high-resolution counters, the duty cycle of a square wave is modulated to encode a specific analog signal level.

3. Process Calibration

- We are using a Servo Motor to control the plane position in order to balance the ball.

- The motion of servo motor arm is controlled by using different width pulse voltage output from microcontroller. The servo motor recognizes a pulse of certain width or duty cycle and rotates to a desired angle.

- PWM signal is applied to its control pin, the shaft rotates to a specific angle depending on the duty cycle of the pulse.

This is how the various positions of servo motor are controlled.

In order to achieve a particular angle and/or motion, we need to determine the gain and frequency value.

Master Clock rate of atmega128 is 16 MHz [7]

The Timer/Counter Control register is used to set the timer mode, pre-scaler and other options.

In our Embedded C program files, the pre-scaler set in the timer TCCR3B = 0b00000011 [7].

From table 62 and 64 in atmega 128 datasheet [7], we get the pre-scaler value:

$$P = 64$$

So,
Reduced (or operated) frequency = 16/64 MHz

$$f_r = 250\ KHz$$

Tick time,

$$T_r = 1/f_r = 0.004\ ms$$

As seen in section xx of this report:

As, a pulse length of 1.5 ms should be generated in terms of operating clock ticks (which should be assigned to the variable MIDPOS.

So, the MIDPOS value will be given by,

$$MIDPOS = (pulse\ length) / (operating\ clock\ tick\ rate) = (1.5\ ms) / (0.004\ ms)$$

MIDPOS = 375

Similarly,

Left (L) position has a pulse length of 1ms.

LPOS = (pulse length) / (operating clock tick rate) = (1 ms) / (0.004 ms)

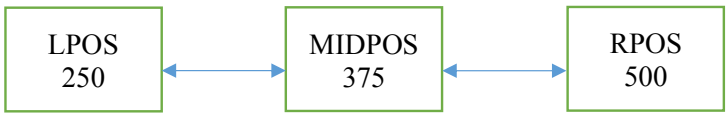
LPOS = 250

&

Right (R) position has a pulse length of 2ms.

RPOS = (pulse length) / (operating clock tick rate) = (2 ms) / (0.004 ms)

RPOS = 500



We can see to go from one position to another e.g. from MIDPOS to RPOS a gain of 125 is required.

DELTA = 125

This gain representing the necessary gain to move the arm from the middle-position to the left or right position is represented by DELTA.

Due to the assembly of the real model, the inclined plane is not exactly horizontal when the pulse length 1.5 ms is given. Thus, after a few iterations of varying MIDPOS and measurements using Vernier Calipers an Actual MIDPOS value of 352 was reached.

Master Clock rate of atmega128	**16 MHz**
Pre-scaler	64
Operating Frequency	250 KHz
Tick Time	0.004 ms
MIDPOS	375
LPOS	250
RPOS	500
DELTA	125
Actual MIDPOS	352

Table 2: Process Calibration Summary Table

3.1 Programming

In EmCoEx_Wippe_v10.c file, we identify the value in TCCR3B and then after determining the pre-scalar from datasheet and after calculation of MIDPOS value. We assign this MIDPOS value to the OCR3A.

```
// Init timer 3
  TCCR3A = 0;
  TCCR3B = 0b00000011; //
  TCCR3C = 0;
  TCNT3 = 0;
  OCR3A = 375;
```

In the SensorServo.c file, we have to use actual value of MIDPOS. Also, the value of DELTA is also defined here. It is used to calculate value actual of OCR3A value.

```
void  vServo ( void * pvParameters)
{ unsigned int  MIDPOS = 352;
  float  DELTA = 125.0, 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 of task servo
```

4. Welcome Message

In order to modify the welcome message, we make a few changes in the HMI.c file. The modified code is as shown.

```
// Init of HMI menus
const char  Main_Menu [80] PROGMEM =
 "GROUP31 EX 3 2020 >Change Parameter   >Show Ball Position >Start PID Control ";
```

5. PID Controller

In the figure below a schematic of a system with a PID controller is shown. The PID controller compares the measured process value y with a reference setpoint value, y0. The difference or error, e, is then processed to calculate a new process input, u. This input will try to adjust the measured process value back to the desired setpoint.

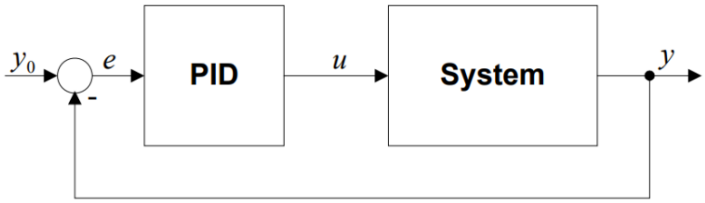


Figure 2: Closed Loop System with PID Controller

Unlike simple control algorithms, the PID controller is capable of manipulating the process inputs based on the history and rate of change of the signal. This gives a more accurate and stable control method.

The basic idea is that the controller reads the system state by a sensor. Then it subtracts the measurement from a desired reference to generate the error value. The error will be managed in three ways; to handle the present through the proportional term, recover from the past using the integral term, and to anticipate the future through the derivate term.

The figure below shows the PID controller schematics, where Tp, Ti, and Td denote the time constants of the proportional, integral, and derivative terms respectively.

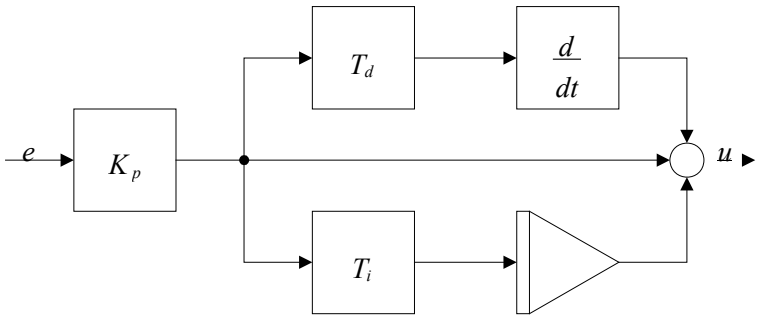


Figure 3: PID Controller Schematic

5.1 Discrete PID Controller

A discrete PID controller will read the error, calculate and output the control input at a given time interval, at the sample period T. The sample time should be less than the shortest time constant in the system.

5.2 Algorithm

Unlike simple control algorithms, the PID controller is capable of manipulating the process inputs based on the history and rate of change of the signal. This gives a more accurate and stable control method [8].

Figure 3 shows the PID controller schematics, where Tp, Ti, and Td denotes the time constants of the proportional, integral, and derivative terms respectively. The transfer function of this system is:

$$\frac{u}{e}\left(s\right) = H\left(s\right) = K_p\left(1 + \frac{1}{T_i s} + T_d s\right)$$

This gives u with respect to e in the time domain:

$$u\left(t\right) = K_p\left(e\left(t\right) + \frac{1}{T_i}\int_0^t e(\sigma)\,d\sigma + T_d\frac{de(t)}{dt}\right)$$

Approximating the integral and the derivative terms to get the discrete form, using:

$$\int_0^t e(\sigma)\,d\sigma \approx T\sum_{k=0}^{n} e(k)$$

$$\frac{de(t)}{dt} \approx \frac{e(n) - e(n-1)}{T}$$

$$t = nT$$

Where n is the discrete step at time t.

This gives the controller:

$$u\left(n\right) = K_p e\left(n\right) + K_i\sum_{k=0}^{n} e(k) + K_d\left(e\left(n\right) - e\left(n-1\right)\right)$$

Where,

$$K_i = \frac{K_p T}{T_i} \qquad K_d = \frac{K_p T_d}{T}$$

To avoid that changes in the desired process value makes any unwanted rapid changes in the control input, the controller is improved by basing the derivative term on the process value only:

$$u\left(n\right) = K_p e\left(n\right) + K_i\sum_{k=0}^{n} e(k) + K_d\left(y\left(n\right) - y\left(n-1\right)\right)$$

Proportional, Integral, and Derivative Terms Together

Using all the terms together, as a PID controller usually gives the best performance. The figure below compares the P, PI, and PID controllers. PI improves the P by removing the stationary error, and the PID improves the PI by faster response and no overshoot [9].

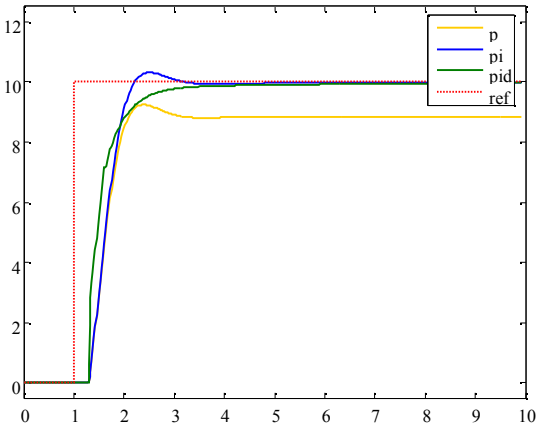


Figure 4: Step Response P, PI, and PID Controller [9]

5.3 PID Parameter Tuning

We started with a Kp, Ki and Kd 2.5,0.03 and 1 respectively as obtained in Exercise 2 of this lab.

Then by using trial and error technique we reached a Kp, Ki and Kd value of 0.6,0.9 and 0.4.

5.4 Implementation

The PID equation is implemented in PID.c file using the following Embedded C Code.

```
// PID.c
// 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 = 0.5, y = 1.0 , st = 0.015, error , lasterror = 0.0 , errortotal = 0.0 ;
  // Super loop of task
  for (;;)
  { // Get position value from sensor queue
    xQueueReceive (QueueSensor, &x, portMAX_DELAY);

        error = (reference -x);
        errortotal  =  errortotal + error  ;

        y = ( proportional * error ) + integral*(errortotal * st ) + derivative * ((error - lasterror) / st);

        lasterror = error ;
        // Write angle value to servo queue
        xQueueSend (QueueServo, &y, portMAX_DELAY);
    } // end for loop
} // end of task PID
```

6. Tasks and Queues

UML stands for Unified Modeling Language and is a way of visualizing a software program using a collection of diagrams.

6.1 Tasks

1. There are total six tasks in our BoiP program as follows: -
2. vSensor: It sense the ball's position and sends to PID controller (QueueSensors).
3. vTaster: It is PCB containing LCD and five Pushbuttons (up, dawn, left, right and Enter). It sends signal to QueueTaster.
4. vHMI: It deals with all Human interaction related work. eg. Parameter change or Start PID control
5. vPID: It generates the error signal comparing ball's position (QueueSensors) with reference position and sends signal to servo motor (Queueservo).
6. vServo: It controls the movement of servo arm depends on error signal received from vPID.
   LED: This task deal with controlling of LEDs based on Value in vIO_SRAM.

```
// Start tasks
 ret_val = xTaskCreate (vSensor, (const signed portCHAR *)\
  "Sensor", configMINIMAL_STACK_SIZE  + (30 * sizeof (float)),\
   NULL, tskIDLE_PRIORITY+2, &vSensor_Handle);
 if (ret_val != pdPASS)  RTOS_Allright = 0;

 ret_val = xTaskCreate (vIO_SRAM_to_LCD, (const signed portCHAR *) \
  "LCD", configMINIMAL_STACK_SIZE  + (20 * sizeof (char)),\
   NULL, tskIDLE_PRIORITY+1, &vIO_SRAM_to_LCD_Handle);
 if (ret_val != pdPASS)  RTOS_Allright = 0;

 ret_val = xTaskCreate (vTaster, (const signed portCHAR *) \
  "Taster", configMINIMAL_STACK_SIZE  + (20 * sizeof (char)),\
   NULL, tskIDLE_PRIORITY+1, &vTaster_Handle);
 if (ret_val != pdPASS)  RTOS_Allright = 0;

 ret_val = xTaskCreate (vHMI, (const signed portCHAR *) \
  "HMI", configMINIMAL_STACK_SIZE  + (100 * sizeof (char)),\
   NULL, tskIDLE_PRIORITY+1, &vHMI_Handle);
 if (ret_val != pdPASS)  RTOS_Allright = 0;

 ret_val = xTaskCreate (vServo, (const signed portCHAR *) \
  "Servo", configMINIMAL_STACK_SIZE  + (5 * sizeof (float)),\
   NULL, tskIDLE_PRIORITY+2, &vServo_Handle);
 if (ret_val != pdPASS)  RTOS_Allright = 0;

 ret_val = xTaskCreate (vPID, (const signed portCHAR *) \
  "PID", configMINIMAL_STACK_SIZE  + (20 * sizeof (float)),\
   NULL, tskIDLE_PRIORITY+2, &vPID_Handle);
 if (ret_val != pdPASS)  RTOS_Allright = 0;
```

6.2Queues

We have used three queues in BoiP. These are as follows,

- QueueSensor: It has one item space type of “float” (32bit). It sense or measure the ball position and store it (ServoSensor.c) . Generate error signal with respect ball’s reference position (PID.c).
- QueueTaster: It has 10 item space type of “Character” (8bit). QueueTaster stores the value of pushbuttons which have been pressed.
- QueueServo: It has one item space type of “float” (32bit). It stores the output of PID.c and send signal to vSensor(ServoSensor.c) and calculate Beta.

```
// Create queues
QueueTaster = xQueueCreate (10, sizeof(char));
 if (QueueTaster == 0)  RTOS_Allright = 0;
 QueueSensor = xQueueCreate (1, sizeof(float));
 if (QueueSensor == 0)  RTOS_Allright = 0;
 QueueServo = xQueueCreate (1, sizeof(float));
 if (QueueServo == 0)  RTOS_Allright = 0;
 lcd_puts_p (strQue);
```

6.3 UML Diagram

The following diagram shows the general idea about queues communication.

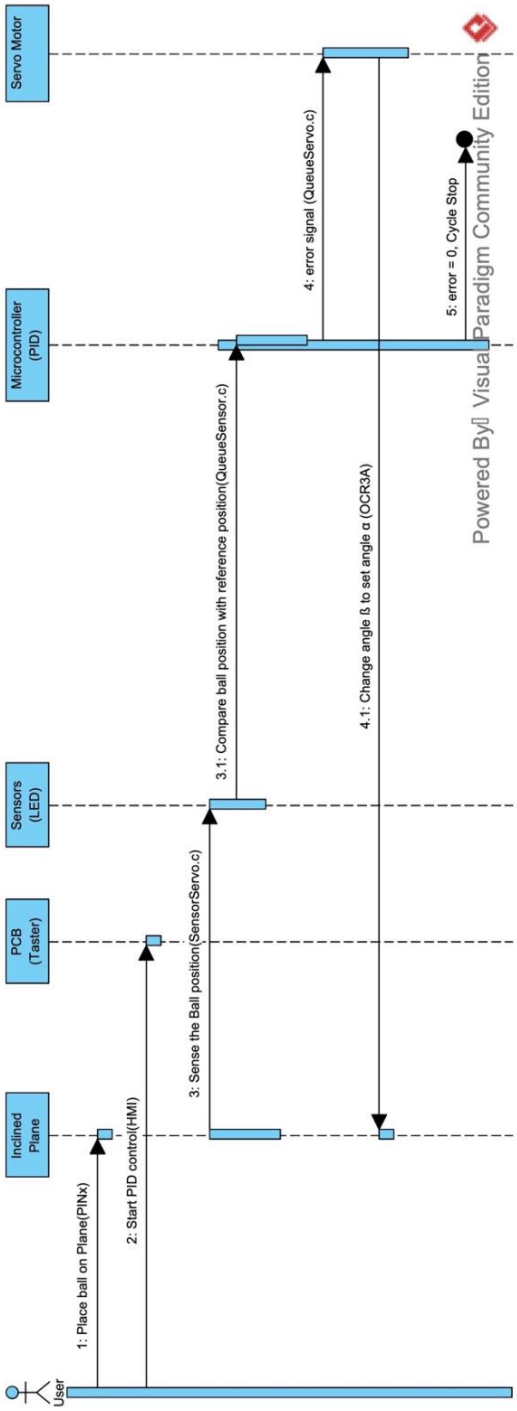


Fig 5: UML Diagram

7. Conclusion

i) Created an executable C project on Atmel Studio 7.
ii) Imported the given files into the Project.
iii) Made some changes in the environment to fir our requirements.
iv) Calculated MIDPOS and DELTA Values with the help of Atmega128 Datasheet.
v) Calibrated the model to acquire actual value of MIDPOS.
vi) Made necessary changes in the code related to MIDPOS and OCR3A.
vii) Implemented PID equation in embedded C language.
viii) Through various iterations reached an appropriate value of Sampling time and PID values
ix) The model successfully stopped the ball within 10 seconds.

Iteration Number	Stopping Time (Seconds)
1	10
2	5.33
3	30+
4	10
5	9.65
6	5.95
7	8.15
8	18
9	12
10	30+
11	9.75
12	8.84

Table 3: Stopping time iteration table.

Note:

All the calibration and Stopping time Iterations were done on Model #2.

References

1) https://en.wikipedia.org/wiki/Embedded_C
2) Skansholm, Jan. Vägen till C [The road to C]. Studentlitteratur. pp. 237–774. ISBN 91-44-01468-6.
3) https://www.freertos.org/
4) https://en.wikipedia.org/wiki/AVR_microcontrollers
5) https://www.microchip.com/mplab/avr-support/atmel-studio-7
6) https://www.microchip.com/wwwproducts/en/ATMEGA128
7) Atmega 128 Datasheet
8) K. J. Astrom & T. Hagglund, 1995: PID Controllers: Theory, Design, and Tuning. International Society for Measurement and Con.
9) Atmel-2558C-Discrete-PID-Controller-on-tinyAVR-and-megaAVR_AVR221_Application Note-09/2016.
10) https://en.wikipedia.org/wiki/Queue_(abstract_data_type)