MSE 450 Project: Robot Pen

Submitted To:	Dr. Mehrdad Moallem
Class:	MSE 450, Spring 2015, SFU
Submitted On:	April 12th 2015

Submitted By:	
canvas.sfu.ca - Group 16	
Yang Liu	301161741
Oskar Valdemarsson	301268778
Sohail Sangha	301186636

Table of Contents

1 Introduction

Learning how to control a physical system in real time that meets deadlines at a set amount of time is the whole purpose of MSE 450. As a method to learn the relevant topics, a course project was prescribed to satisfy this purpose. Since it is the student's responsibility to learn, it was our combined ambition to proceed with a challenging project.

1.1 Platform overview

Students were given full design control over the project by the professor as long as the final product met some certain criteria such as the use of embedded platform for system control, real time control system development, low level programming for microcontrollers etc. The project demonstrated in this report borrows from the said criteria to deliver a mechatronics products which involves a real time control system based on a Tiva-C-Series microcontroller board, and controls two brushed dc motors in a closed loop using quadrature encoders to bring out the action of writing on a horizontal surface.

1.2 Purpose

As said in the platform overview the intent and purpose of this project was to satisfy the requirements set by the class project. The implicit reason for going with this project was simply due to the simplicity and ease of construction of the hardware. Further, by limiting the 2 degrees of freedom to the horizontal not only reduces the stress on the hardware but also results in simpler kinematic equations. Discussion was held to attach an electromagnet at the end of the arm to produce a 2 degree of freedom "pick and place" robot, but the idea was abandoned in lieu of actuators which would be readily available. Hence the final product supports an extremely simplistic servo mechanism at the end which helps to move the pen up or down. The idea of using a pen was derived from earlier tests on the systems, where the designers were interested in plotting the physical path taken by the robotic arm by using a piece of paper.

2 Method

The "Robot Pen" project proved to be an involved project from both hardware and software perspective, hence the report has been sectioned to deliver an easier understanding of the project. The method has been sectioned into four parts:

- Hardware — Deals with the chronological development of the final design
- Software — Code run on the MCU with help of visual aids
- Controller — Mathematical design information and implementation results
- Control Interface — Java software designed for issuing commands to MCU from a laptop

2.1 Hardware

The hardware is section into two components, Arm I and Arm II with roman numerals standing for both the chronological order of design and the number of degrees of freedom offered by each system type.

2.1.1 Arm I

The initial project was the 1 degree of freedom arm and can be seen in figure 1. The motor was mounted on a tripod made from 3d printed parts. An arm was attached to the motor. Limiting switches were placed on the side of the base that functions as interrupt input. The motor was attached to an H-bridge that allowed for bidirectional rotation: counter clockwise and clockwise. The motor has a built in digital encoder that provided positional feedback to the Tiva microcontroller.

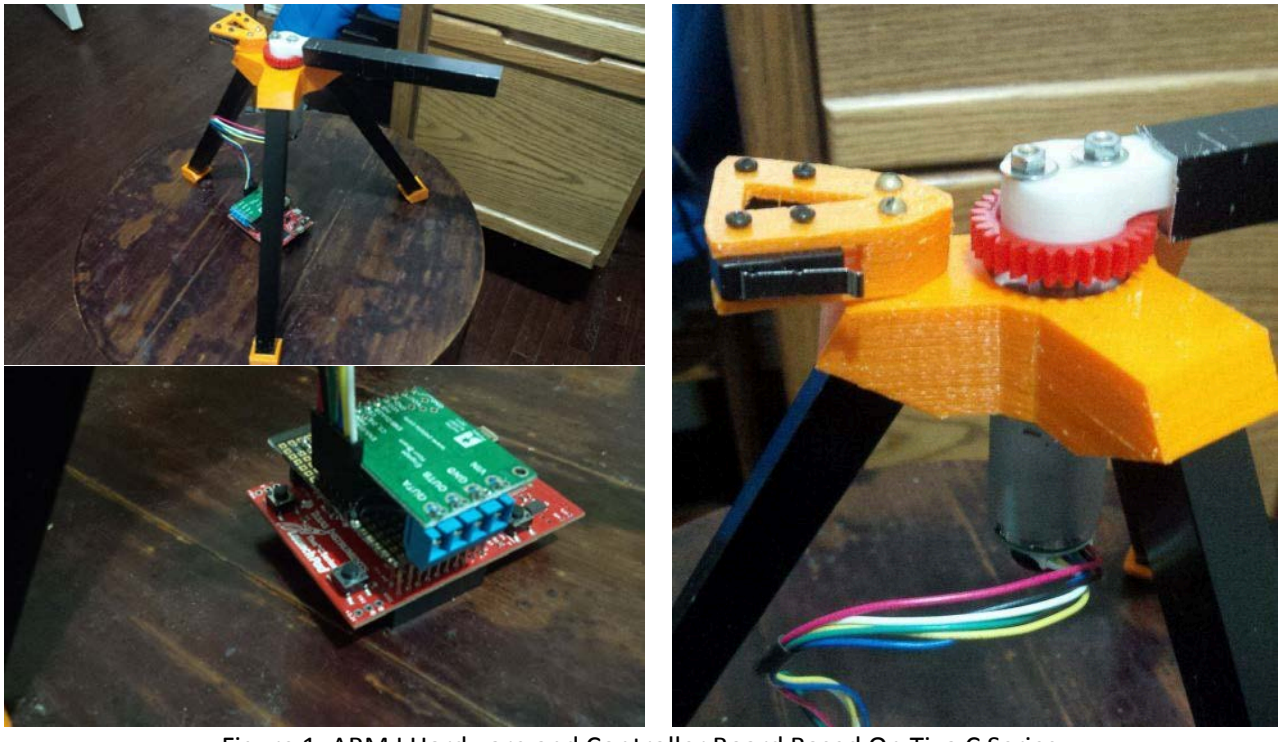


Figure 1; ARM I Hardware and Controller Board Based On Tiva C Series

2.1.2 Arm II

The 1 degree of freedom arm was easily constructed as the lab materials were either already available or was readily 3d printed. Through unanimous decision, a second arm was attached to the end of the first arm. The second arm would be exactly the same configuration as the first arm with the motors attached to the end of the first arm allowing for 2 degrees of freedom through two 1 degree of freedom arms. A motor with a pen attached is connected to the end of the second arm. Images of Arm II can be seen in figure 2.

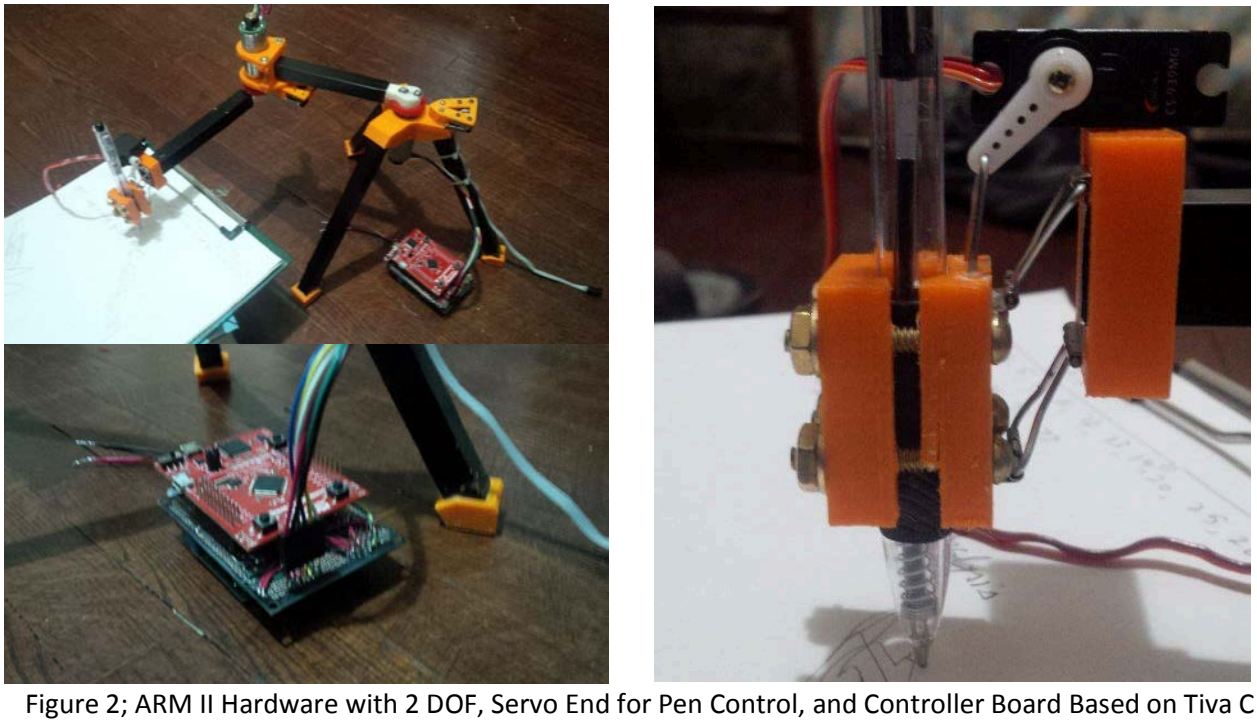


Figure 2; ARM II Hardware with 2 DOF, Servo End for Pen Control, and Controller Board Based on Tiva C

2.2 Software

2.2.1 Arm I

Due to the lower requirements set by the limited hardware components of Arm I (1 dc brushless motor, 2 limit switches, 1 encoder module) the software designed for Arm I served both as learning and testing program for the group members. As such, the program was designed as a plug and play system with a unified code to which new functions were added on a requirement and feature evolution basis. A flowchart of the software in Arm I can be seen in figure 3 and a UML diagram can be seen in figure 4.

The Tiva microcontroller has specific pins that can be configured as either input or output. At the beginning of the program, the algorithm instructs the motor to rotate the arm until either a left or right limit switch is hit, the motor is instructed to stop at this point. The position where the arm hits the limit switch will be set as 35 degrees from the base. This will be later employed as an angular coordinate system for the PID controller. Through a PID controller, the arm rotates to 180 degrees.

The PID controller is a simple feedback controller with a proportional term. The reference input is provided by the user, the process variable is provided by the motor encoder.

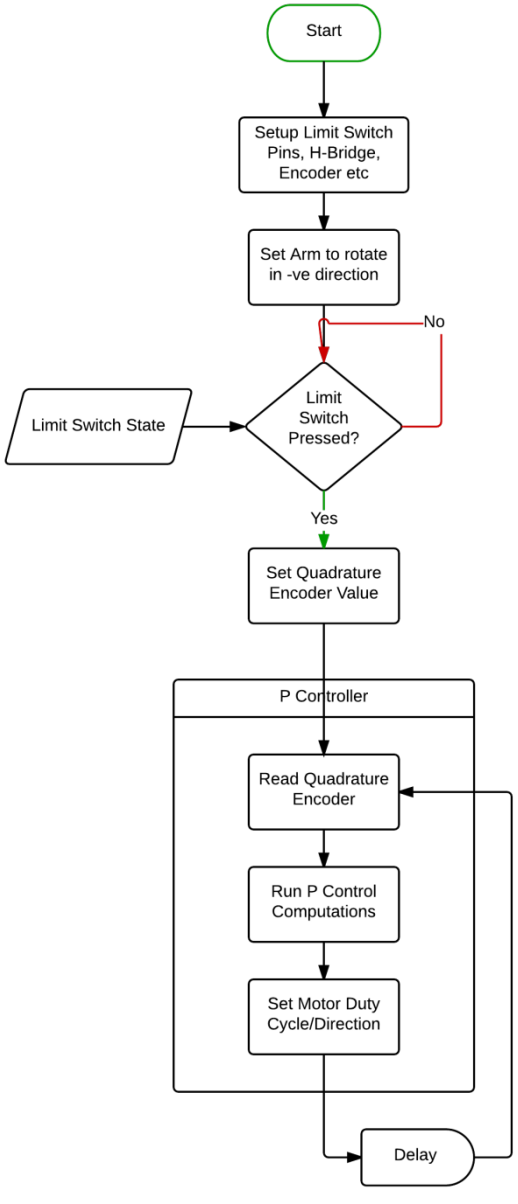


Figure 3; Software controller flowchart for ARM I

ARM I Controller

• **Public**
const int rightPwm = PF_2;
const int rightINA = PE_1;
const int rightINB = PF_4;
const int flip1right = PB_1;
const int flip1left = PB_5;
const int flip2right = PE_4;
const int flip2left = PE_5;
const int button1 = PUSH1;
const int button2 = PUSH2;
int QEIpulses = 0;
long int time = 0;
boolean rightFlipHigh = false;
boolean leftFlipHigh = false;
uint32_t duty = 0;
unsigned int freq = 5000;
uint32_t res = 1000;

• **Public** setup(void): void encoder_init(void): void calibrate(void): void loop(void): void duty_Pcontroller(int position): int duty stopMotor(void): void leftFlip(void): void rightFlip(void): void

Figure 4; UML class equivalent for ARM I Controller

2.2.2 Arm II

While Arm II was structurally more complex than its predecessor, it offered the team members a good opportunity to learn about object oriented programming, as the behavior of both the dc motors in the arms could be approximated as simple 1 degree of freedom arms. But it offered a few challenges from software side, as not only it required the handling of two instances of "controller" object to manage the two dc motors, but also with the addition of the consideration that the system was to be designed as computer controlled device. Where the MCU itself would be responsible for lower level handling of the system such as through PID, PWM, Quadrature encoder modules, H Bridge etc., but the compute would be responsible for bringing about a complex action by issuing simpler commands to the MCU.

2.2.2.1 Top-Level Program

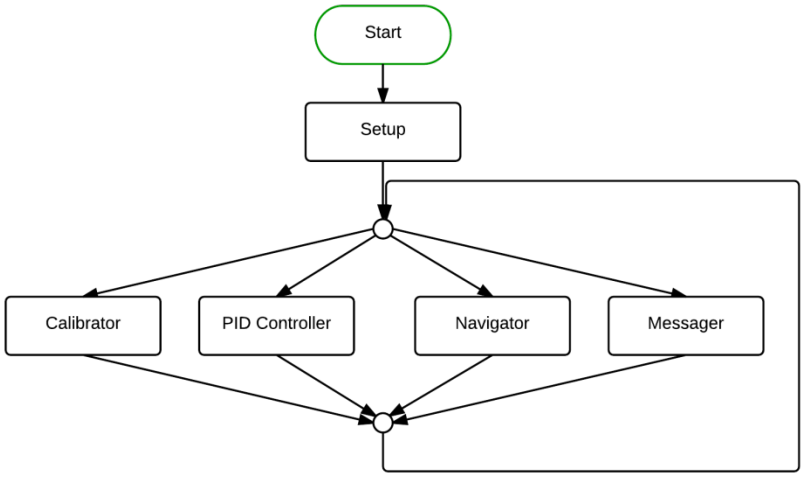


Figure 5; Main program flowchart for ARM II

From figure 5, the architecture is divided into 4 modules: the calibration, PID controller, the navigator and the messenger. When the microprocessor is turned on with the algorithm pre-uploaded, the calibration is run first which resets the quadrature encoders once the flip switches have been hit. This is achieved through turning on the base motor until the flip switch is activated. The program then continuously runs the PID controller, the navigator and the messenger acts as pseudo processes that are checked to see if they have been called by the user. The processes are not parallel; they are processed at a very fast speed with a guarantee by the developer such that none of the processes would occupy more than 5 milliseconds of runtime during each independent call.

2.2.2.2 Calibrator

Calibrator
• **Private** int calibration_stage = CALIBRATION_COMPLETE;
• **Private** calibration_stateChange(int newStage): void • **Public** calibration_run(void): void calibration_init(void): void calibration_interruptHandler(void): void

Figure 5; UML class equivalent for Calibrator

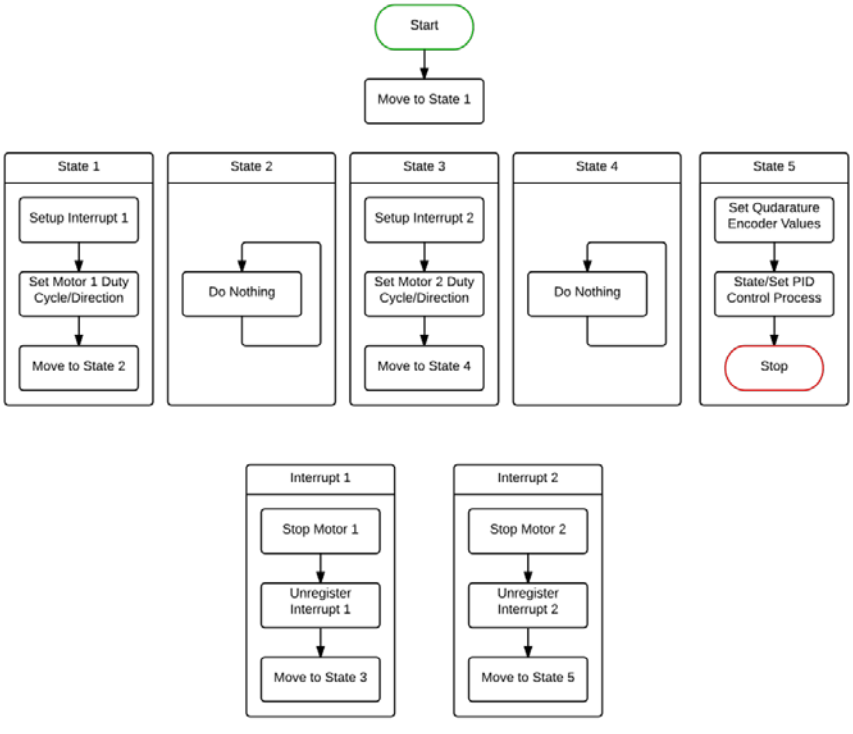


Figure 6; Flowchart of the calibration Process

Refer to figure 6 of the flowchart and figure 5 for a UML diagram over the calibrator. The calibration resets the encoders. The encoders can only measure positional change. Therefore, it cannot obtain its current position without a reference. The signals from the quadrature encoders are processed by the microcontroller; encoder value at the algorithm design level is a counter value that changes as the arms rotate back and forth. The calibration module is designed as a finite state machine, i.e. the program has a fixed set of states it can be in. This achieves the desired behavior that the program transitions from one state to another.

The program begins with the base arm rotating in one direction at a relative slow speed until the base limit switch is hit. Until Interrupt 1 is activated, the messenger is checked for update. The base arm stops at this point. The extended arm begins rotating until the extended limit switch is hit which activates interrupt 2. The extended also stops after this point. Afterwards, the two arms both rotate to 180 degrees which is at a straight line for the arm. The encoders are reset when the switches are hit. The switch triggers an interrupt which can be seen as a block in figure 3.

2.2.2.3 Messenger

To communicate with the microcontroller a communication protocol layer called messenger has been built. Refer to figure 7 for the flow chart and figure 8 for the UML diagram.

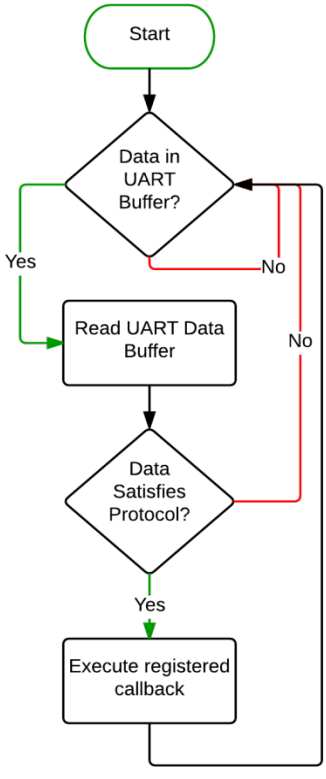


Figure 7; flowchart of the messenger process

Messenger

• **Private**
byte messageRxBuffer[MESSAGE_RX_BUFFER_LENGTH];
int messageRxBuffer_ReadPos = 0;
int messageRxBuffer_StorePos = 0;
void (*messageHandlerCallback)(int, char*);
char messData[MESSAGE_ID_MAX_NUM];
char tempBuffer[MESSAGE_RX_BUFFER_LENGTH];

• **Private** messager_numBytesInBuffer(void): int messager_readSerialBuffer(void): void messager_readByteFromRxBuffer(void): int messager_peakByteFromRxBuffer(byte data): int messager_extractByteFromRxBuffer(int relativeLoc): int messager_trashRxBytesFromRxBuffer(unsigned int num): void messager_doLengthCheck(int len): void messager_processReadBuffer(void): void • **Public** messager_run(void): void messager_add(uint16_t messageId, uint16_t length): void

Figure 8; UML class equivalent for Messenger

The messenger enables message-based communication between the microcontroller and the computer during runtime. The messenger greatly reduces control complexity during runtime; steering the robotic arm to a final position could be achieved through executing a command through the graphical user interface. This messenger feature also makes the debugging process easier since multiple runs can be executed without resetting the microcontroller; saving both time and effort. The messenger also enables a neat feature of logging data from the PID controller. Graphs can be plotted for the behavior of the PID controller which greatly simplifies tuning the PID controller values. The graphical depiction of the system's response is an ideal summary for the systems behavior than the actual physical response since the physical system is prone to physical limitations such as whiplash, dead band and friction.

2.2.2.4 Navigator

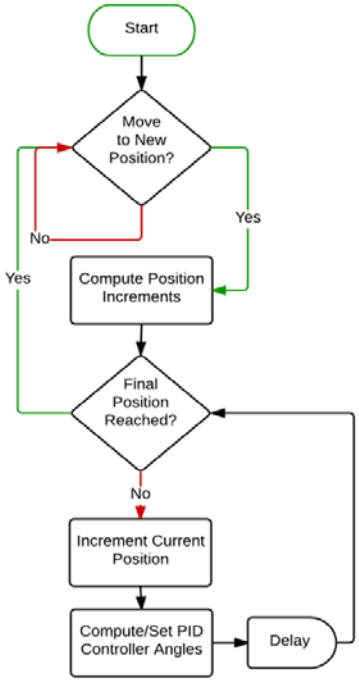


Figure 9; flowchart of Navigator

Navigator

• **Private**
double xStep = 0;
double yStep = 0;
• **Public**
double current_x;
double current_y;

• **Private** navigator_update(double x, double y): void • **Public** navigator_run(int id): void navigator_set(double x, double y): void

Figure 10; UML class equivalent for Navigator

As mentioned above, the arms are controlled by the PID controller by sending an angle to it, since that is the measurements received from the encoders. The arm position is controlled through the PID controller; input is fed in from the user, the motors rotate the entire toward the desired position. However, this robot contains two degree of freedom which makes it unintuitive to control the two arms by just their angle. A more intuitive control method would be giving them a position in a 2d Cartesian coordinate system. This is where the navigator becomes useful. The navigator takes a desired 2d Cartesian coordinate as input, converts them into the angles through inverse kinematics and finally passes the desired angle to the PID controller as a set point. The endpoint for the two degree of freedom arm, the pen, will move towards the final destination in a straight line. Moving the pen to the straight line all comes down to the mathematic calculations that converts coordinates from one system to another. Making the pen move in a straight line also requires creating smaller increments between the starting point and the desired goal point; trying to make the pen follow these smaller steps gives the illusion of a straight line. A simplified block diagram of the system showing the flow from the user input to the result of the pen can be seen in figure 18.

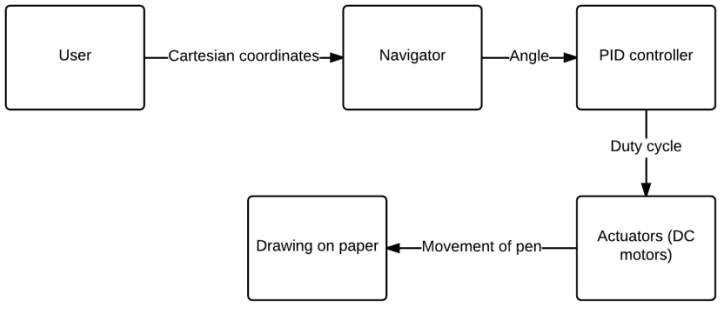


Figure 18; Data/Action flowchart for ARM II

2.3 Controller

The arm was first designed with a PID controller. However, after much testing and adjusting of the PID controller it turned out that the linear PID controller wasn't sufficient to control the arm with satisfying results. Thus a nonlinear controller was introduced which is the current running controller. Below, the designs of these controllers are presented.

2.3.1 PID Controller

The PID controller flowchart is in figure 11 and its UML diagram can be seen in figure 12 and 13. The PID controller runs in a loop with a frequency about 1 kHz with half the frequency allocated to the control of each PID subsystem. A desired angle of the arm is taken as input. An appropriate duty cycle is the output which is calculated from feedback error. Calculating the correct duty cycle for the PWM, it uses values from the quadrature encoder module which gives the current position of the arms. The general procedure for a feedback loop is then implemented with the error passed through the proportional, derivative, and integral term for a controller output to the final control element.

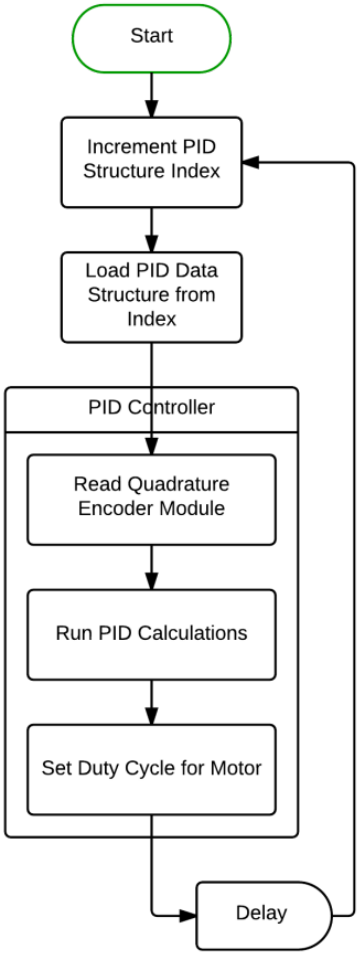


Figure 11; Flowchart of the PID controller

PID Controller Data
• **Public** uint32_t (*read_encoder)(void); uint32_t QEI_base; uint32_t inA_pin; uint32_t inB_pin; uint32_t PWM_pin; uint32_t previous_time; float previous_error; float integral; float prop; float Td; float Ti; float final_pos; bool reverse_dir;

Figure 12; UML class equivalent for PID Controller Data

PID Controller
• **Public** pid_data_tag pid_data[MAX_PID_CONTROLLERS];
• **Private** pidCont_runCont(int pid_num): void • **Public** pidController_set(int pid_num, float final_pos): void pidController_run(int timerId): void pidController_init(void): void

Figure 13; UML class equivalent for PID Controller

To test the PID controller and adjust its parameters the inner arm was set to go from 90 to 270 degrees. The angle of the inner arm as it were traveling from 90 to 270 degrees can be seen in figure 14 and the PID value at the same motion can be seen in figure 15.

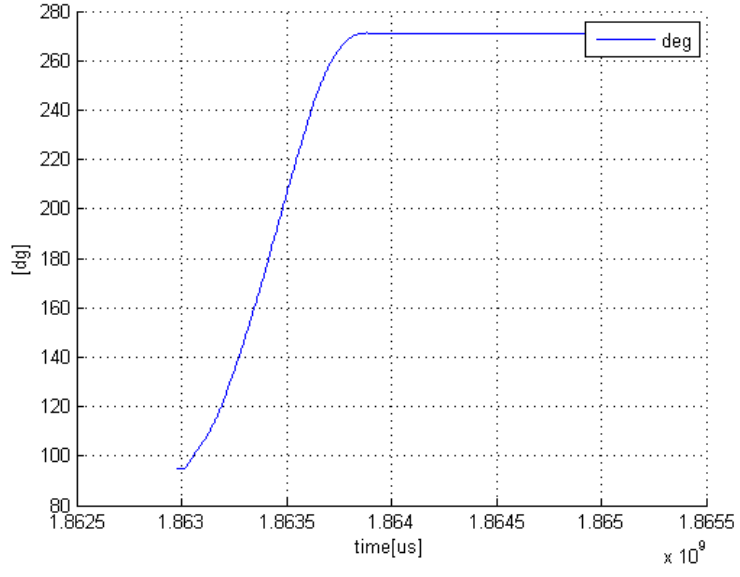
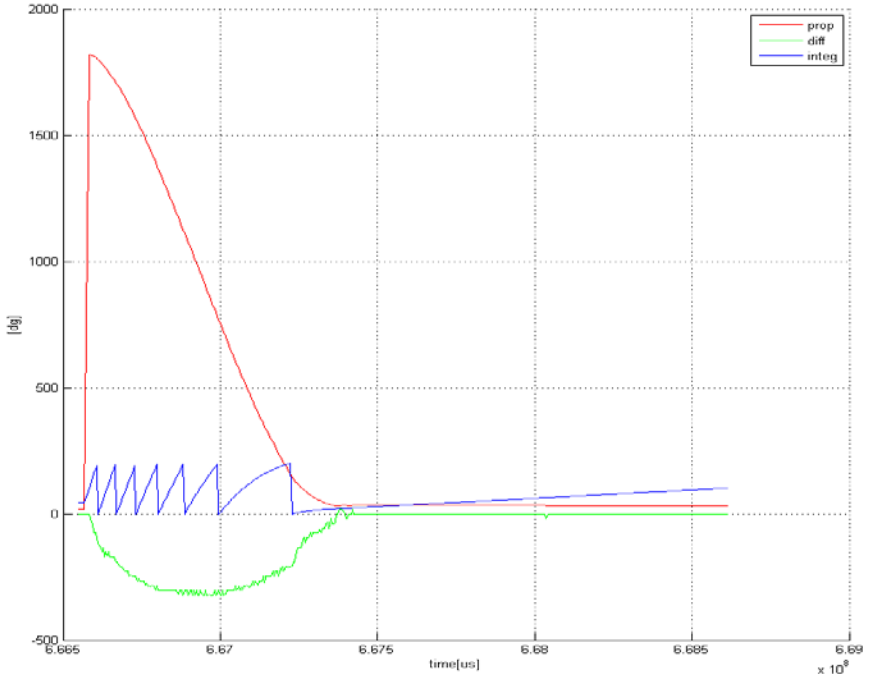


Figure 14; Angle as arm travels from 90 to 270 degrees



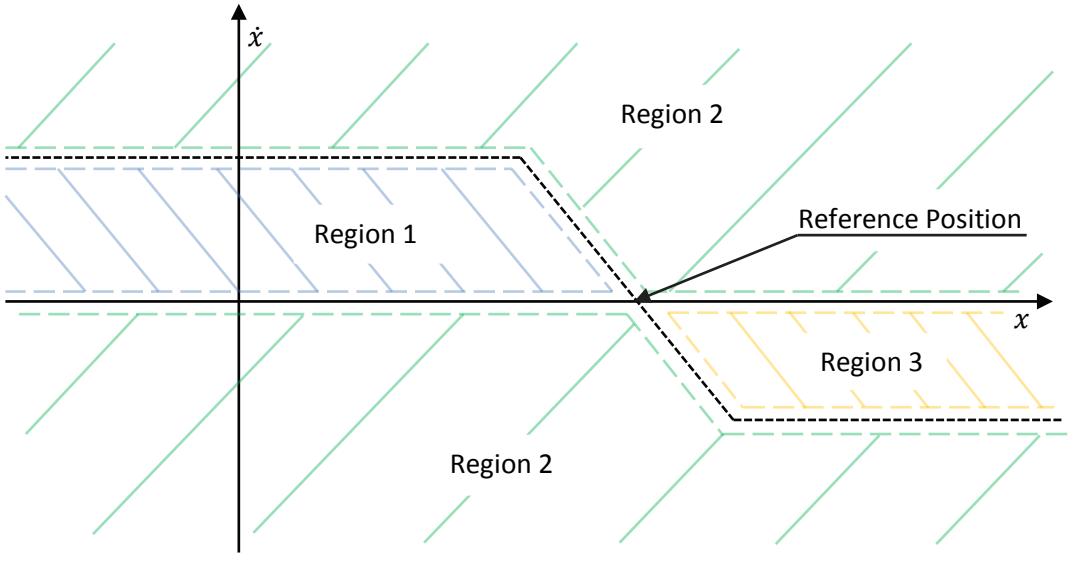
Figur 15; PID values as arms travels from 90 to 270 degrees.

The motion of the arm as it was traveling from 90 to 270 degrees was quite smooth. However it usually didn't stop at exactly 270 degrees. In some tests it stopped at as much as 279 degrees, so the error could be as much as +-9 degrees. The PID values seen in figure 11 are summed and results in the duty cycle sent to the DC motor. The proportional value has the major part of the duty cycle but as the arm is closing in on its goal the negative derivative part gets bigger influence which results in that the arm slows down considerable as it closes in on its goal. As seen in figure 11 the integral part is keep resetting once it

reaches a value of 200. This is the windup guard kicking in. Without the windup guard the integral would keep on building up as long as the arm wasn't at the exact goal position. And since the motor often was incapable of reaching an exact position, this would result in an undesirable behavior with the arm jumping back and forth over the goal position. This was because the motor wasn't able to move at a low speed. To be able to handle this problem a more advanced controller than the PID was needed. Thus in the end, the PID controller was abandoned for a nonlinear controller.

2.3.2 Non Linear Controller

The need of nonlinear controller arose due to the limited accuracy provided by the PID controller due to its inability to operate cheap dc motors with high dead band gap. The nonlinear controller is based on slider mode control theory with the addition of features to limit the operational speed of motor. The nonlinear controller operates by the principle of navigation through the state space of 1-degree of freedom arm. The controller exhibits pre-described behaviour depending on the region of the state space it is in. A visual description of the pre-programmed behaviour is given in Figure 16.



Region 1	Spin Motor In Positive Direction
Region 2	Apply Braking
Region 3	Spin Motor In Negative Direction

Figure 16: Non Linear Control System Behaviour in State Space

To test the functionality of the code structure, a Matlab version was designed first. On achieving credible results the c-code equivalent was designed for "Robot Pen" by adapting the code and data structures of the existing PID controller. The accuracy and precision of the nonlinear controller was done in the same manner as for PID controller, where the data pertaining to the control structure was logged on a computer. The data was plotted with matlab to observe the results, which showed remarkable enhancement in performance, as the nonlinear controller was able to accurately guide the motors to within 1 degree whereas for the PID controller the accuracy was variable from ±5 degrees to ±9 degrees. Secondly due to the use of speed limiting feature of the nonlinear controller better results were obtained while drawing as the motor are able to produce more torque to cope with nonlinear forces generated by the dragging of a pen on a paper.

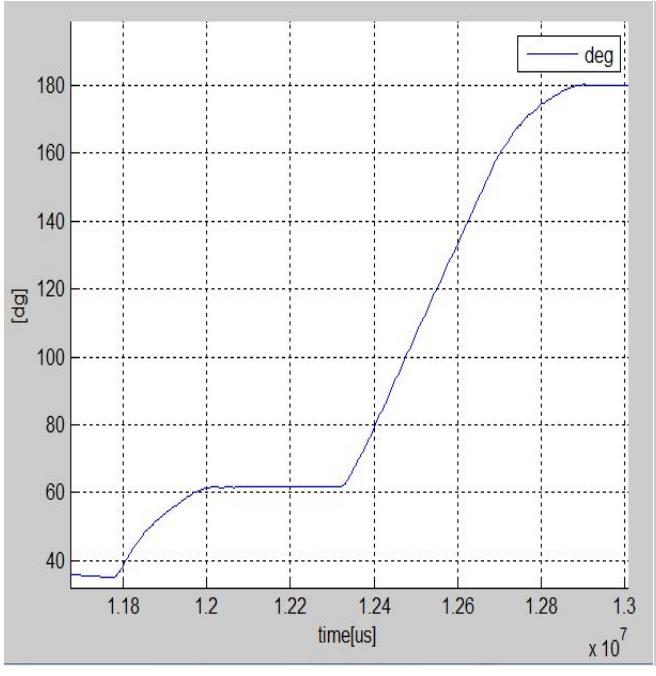


Figure 17; NLCS, angular position Vs. time

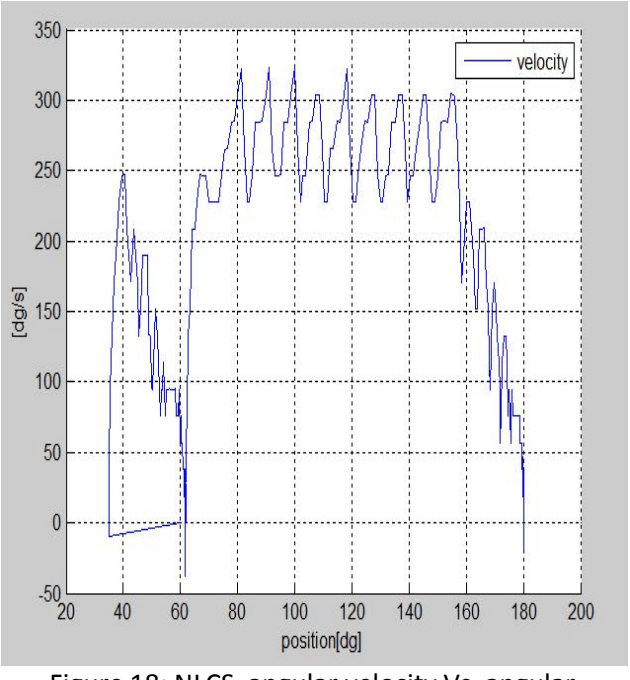


Figure 18; NLCS, angular velocity Vs. angular position

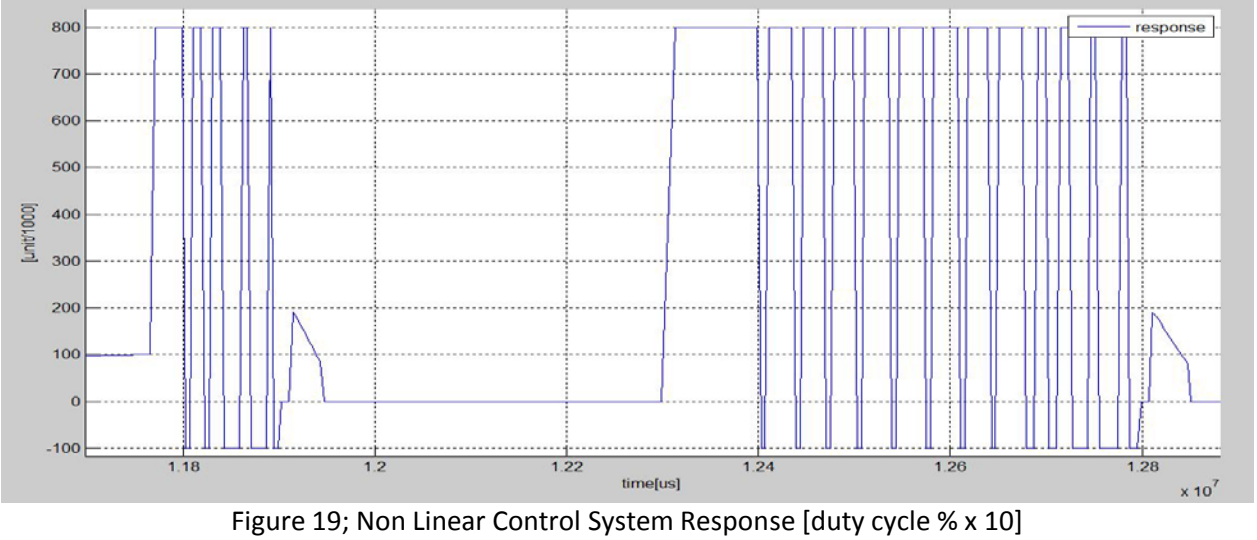


Figure 19; Non Linear Control System Response [duty cycle % x 10]

The results gathered from the nonlinear controller are given in Figure 17 through 19, where the controller attempts from 35 degrees to 60 degrees in the first step and then navigates from 60 degrees to the 180 degrees. The position vs. time graph can be seen in Figure 17, while the controller's eye view can be seen in Figure 19 where the saw patterns are being generated due to the controller jumping from one region to the other.

2.4 Control Interface

The purpose of the control system on the MCU was to perform actions specified by a remote host such as a computer. The communication protocol loosely follows the G-Code structure, as given in figure 20 below, using which the computer can issue a string form of the command. The MCU is responsible for extracting data from the command and executing the associated action.

'$'	0 - 255	Data 0	…	Data N_M - 1	';'
Message Start	Message ID (M)	Fixed Data length N_M for ID M			Message End

Figure 20; Message data format for sending commands to MCU over serial UART.

In a traditional setting, the command being sent to the MCU would be meticulously typed in by the user over serial client such as Putty. Control interface is a java based GUI which was designed to automate this process. When supplied with a script containing commands and data, it is able to sequentially transmit them to the MCU over a serial port. The software has been designed with future expansion in mind such as in case of any other future projects, and thus offers a generalized functionality which remains independent of the command set and data being sent to the MCU. A sample of the script read by the control interface is as follows:

```
mess add PEN 71
mess add NAVIGATION 82
mess send PEN 0
mess send NAVIGATION -10,010
script pause 2000
mess send PEN 1
```

In figure 21 the control interface windows can be seen.

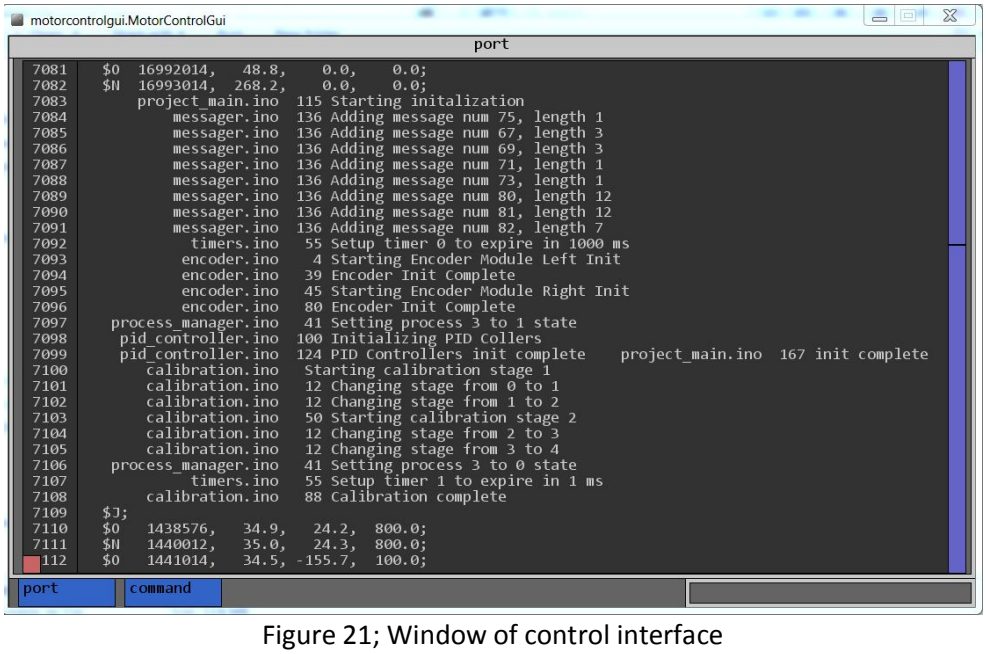


Figure 21; Window of control interface

The control interface and its mechanisms are not covered in the scope of MSE 450 and thus require no further elaboration.

3 Results

3.1 Arm I

The process executed as planned with the arm rotating to 180 degrees. However, the proportional controller rendered the system unstable as the system does not settle to the final point completely. Apart from the inadequacy in the controller design, unanimous team decision was to enhance the challenge presented by the project, as the Arm I hardware and software design was achieved within one week.

3.2 Arm II

When the PID controller was designed for the arm a couple of tests were done to check the performance. A test where the arm with the PID controller has drawn several stars is seen in figure 22.

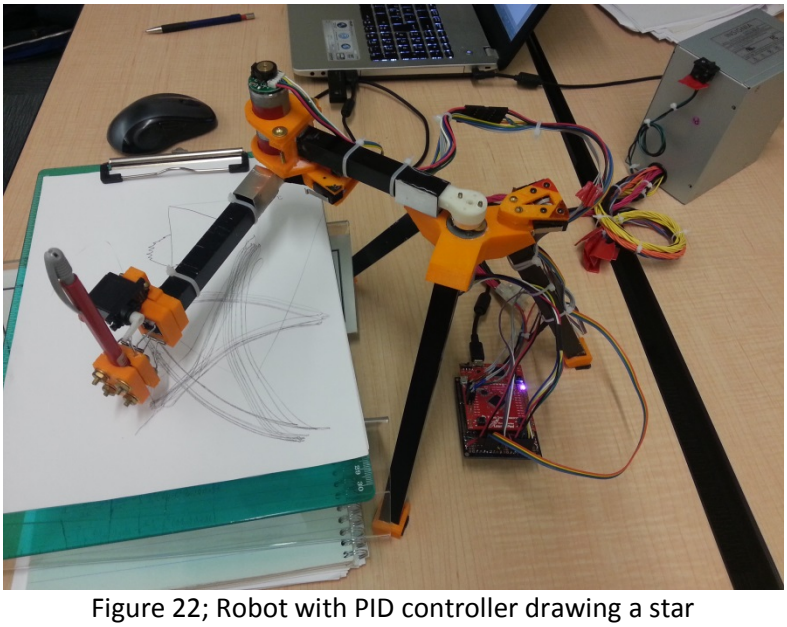


Figure 22; Robot with PID controller drawing a star

As seen the PID controller worked quite fine when drawing a star. As long as the step between two points is large, like in the star, the PID controller works ok. However when trying to move shorter steps it will be unable to move. This made it impossible to draw straight lines. As can be seen in the star of figure 21 the lines of the star are far from straight.

The system worked perfectly for the initial calibration stage with both arms extending to their respective 180 degrees. A pitfall for the economical motors was system inaccuracy. This caused backlash in the system, leading to degraded system performance. The slight deviation from the intended two dimensional Cartesian coordinate was the only flaw that prohibited the system from operating smoothly. The PID algorithm had to be tuned to improve the system performance.

After multiple attempts at system tune-up and PID controller feature updates, it was realized that the dead band gap of the dc motors was excessively large. Often the motors were unable to move at a 30% duty cycle when operated on 5 volts. This was due to multiple factors such as the cheapness of the motors, as well as the fact that the motors were intended for high torque application as opposed to precision applications, and the preferred operational voltage being more than 6 volts.

Since it was highly unlikely to source new dc motors, the final solution was to attempt upgrades to the software in the available time before the project demonstration. The result was the design and integration of nonlinear control system and feature updates to the navigator process. Using the new software components the team was able to remarkably improve the performance of the system. Although the final results are far from perfect, they do prove the capability of the nonlinear control system to perform

precision actions even when dealing with highly inaccurate actuators as is seen in Figure 23 where the robot has written “MSE” on a paper. Compared to the star in figure 22, you can see that the lines are straighter, and it can also be seen that the robot can make shorter lines with better precision.

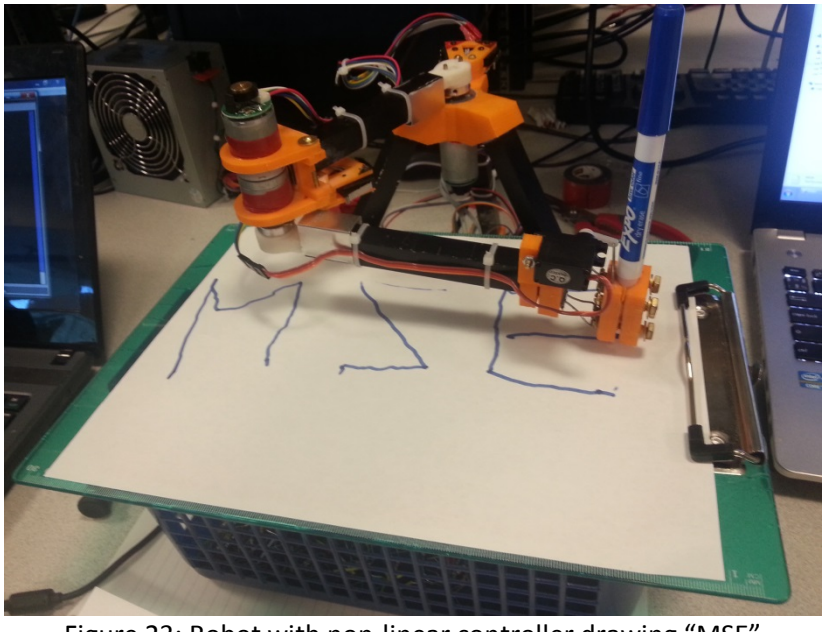


Figure 23; Robot with non-linear controller drawing “MSE”

4 Further Development

As mentioned earlier, one of the major concerns of the robot are the DC motors, which were hard to control due to the backlash in the gearbox and their inability to move at low speed. This is because they were not intended for precision application such as controlling a robot. To be able to improve the system further these motors would need to be changed to a pair more suitable for precision applications. If the motors were changed the current nonlinear controller could be changed to a more suitable controller, maybe a PID controller.

Another module which could need improvement is the navigator. When it moves from point to point in the drawing, it expects to reach the next point within a set time. When that time is up it moves to the next point. So if the pen reaches the point ahead of that time, the pen is going to stand still on that point until the time is up, resulting in a stepping behavior of the pen’s movement. Instead, is a feedback system that could measure when the pen reaches the point is implemented, and could move on to the next point directly as the pen reaches that point, it would result in a much smoother motion of the pen without having to stop the pen at every point.

5 Conclusions

In the end the final result of the project can be seen as successful. The goal was to be able to write the letters "MSE" on a paper with a robot arm and that is what was achieved as seen in figure 23 under results. The purpose of the project mentioned under chapter 1.2 was achieved and the project in nature followed true to the principle behind Mechatronic Systems Engineering. A great deal was learned through numerous real world challenges with valuable experiences were acquired regarding team dynamics in the real world setting.

6 Group Members & Contributions

6.1 Yang Liu

4th year undergraduate student at Simon Fraser University pursuing mechatronics systems engineering.

Project Contributions:

- Software
  - Arm I architecture design
  - Arm I component design
    - Calibration function
    - Interrupt Handling functions
  - Arm II component design
    - Calibrator
    - Navigator
    - Pen-Nib servo control API
- Project Report
  - Writing, editing, and formatting
  - Editor in Chief

6.2 Oskar Valdemarsson

4th year exchange student from Chalmers University of Technology in Sweden studying mechatronic engineering, doing my fourth year at Simon Frasier University.

Project Contributions:

- Software
  - Arm I architecture design
  - Arm I component design
    - Proportional controller
  - Arm II component design
    - PID controller data structure
    - PID controller
    - PID message structure
  - Matlab GUI for generating scripts for Control Interface
  - Scripts for Control Interface
- Project Report
  - Writing, editing, and formatting
  - Images
  - PID controller figures

6.3 Sohail Sangha

4th year undergraduate student at Simon Fraser University pursuing mechatronics systems engineering.

Project Contributions:

- Hardware
  - CAD component design and 3D printing
  - Hardware and electronic assembly
  - Tiva-C-Series based controller board
- Software
  - Arm II architecture design
  - Arm II component design
    - Messenger
    - Calibrator
    - Non-Linear Controller
    - H bridge control API
    - APIs for process management and serial USB logging
  - Java based Control Interface
- Project Report
  - Writing, editing, and formatting
  - Flowcharts and UML classes
  - Nonlinear controller figures