San Jose State University
Department of Computer Engineering

CMPE 127 Fall 2018

Project Report

RoboARM

12/15/18

by

Salvatore Nicosia, SID 012013599

Nickolas Schiffer, SID 012279319

I. INTRODUCTION

This report will discuss the design and development of a 3D printed robotic arm which makes use of the ARM based Tiva TM4C123 microcontroller. The arm is capable of recording a sequence of movements from a joystick and playing it back to repeatedly accomplish the same task.

II. DESIGN METHODOLOGY

The RoboARM utilizes a TM4C123GH6PM microcontroller to interpret inputs from an old analog joystick to control the servos that drive the arm; while at the same time making sure that the arm abides by the set movement bounds of the hardware. The microcontroller implements the added functionality of being able to record, store, and ultimately repeat routines that the operator performs. The robotic arm was designed and built in five different phases.

- Phase 1: 3D-printing the Robotic Arm and Claw
- Phase 2: Reverse Engineering the Joystick
- Phase 3: Ports selection and µController wiring
- Phase 4: Programming the TI TM4C123GH6PM µController
- Phase 5: Testing and Adjusting Functionality

For the arm several components were used to satisfy the objectives of the project. The components used are the following:

- TM4C123GH6PM
- 4 servos MG995
- 3D printed arm (https://www.thingiverse.com/thing:1454048)
- 3D printed claw (https://www.thingiverse.com/thing:1480408)
- Various resistors
- 1 Power supply 5V 3A (servo power)
- 1 Power supply 5V 1A (joystick power)
- 1 DB15 D-SUB Female Jack 15 Pins Port
- Prototyping boards
- 16x2 LCD Module
- Analog Joystick
- 10k Potentiometer
- DuPont Wires

In addition, to program the TM4C123GH6PM µController the following IDE and programming language were used:

- Energia (IDE)
- C
- Liquid Crystal Library (https://github.com/energia/Energia/tree/master/libraries/LiquidCrystal)

Phase 1: 3D Printing the Robotic Arm and Claw

During phase 1, the structure of the robotic arm was 3D-printed using the TEVO Tarantula 3D-printer. For the purpose of this project the arm was printed using PLA as it is a cheap and reliable 3D-printing material. The arm was found on the popular website thingiverse.com which is an open source website where people share free 3D model objects. *Figure 1*, shows the 3D-printed structure of the arm.

Figure 1: Structure of RoboARM

Furthermore, the claw was 3D-printed in the same manner and the model was also found on thingiverse.com. *Figure 2* shows the 3D-printed claw.

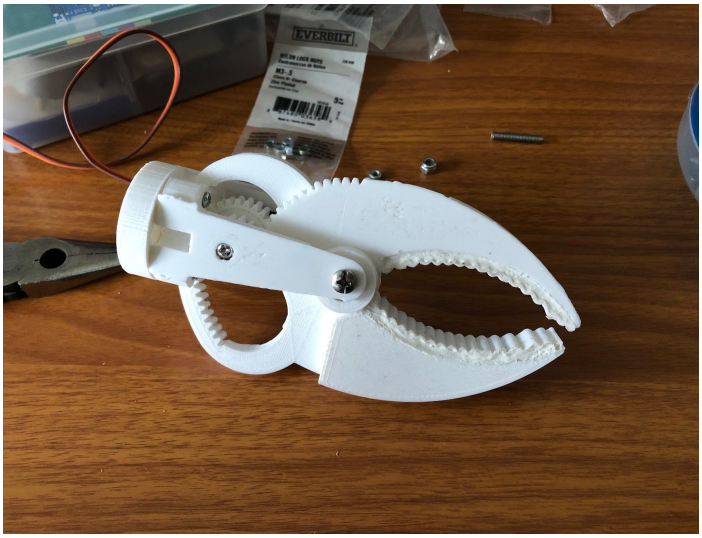

Figure 2: Claw Side-view

After printing both the arm and the claw the four MG995 servos were installed as shown in *Figures 3* and *4*.

Figure 3: Arm with servos installed

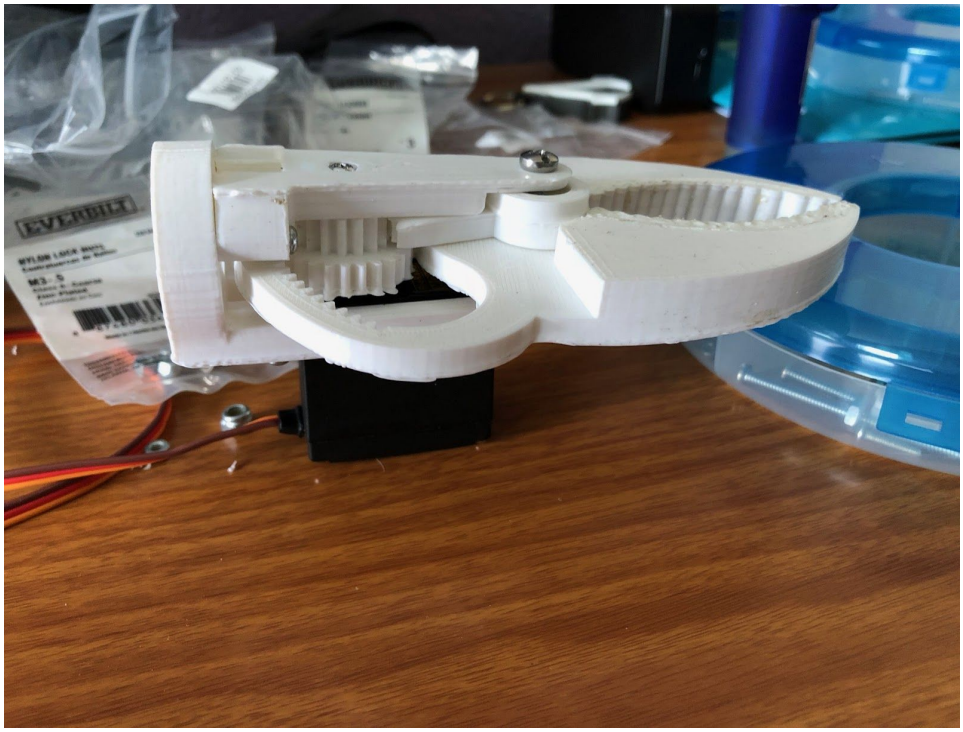

Figure 4: Claw with servo installed

Phase 2: Reverse Engineering the Joystick

The first goal of phase 2 was to determine how the joystick works internally in order to be able to utilize its outputs in a predictable manner. This involves powering the joystick while manipulating its controls in order to document the functionality. The pinout of the joystick was determined to be as shown in *Figure 5*.

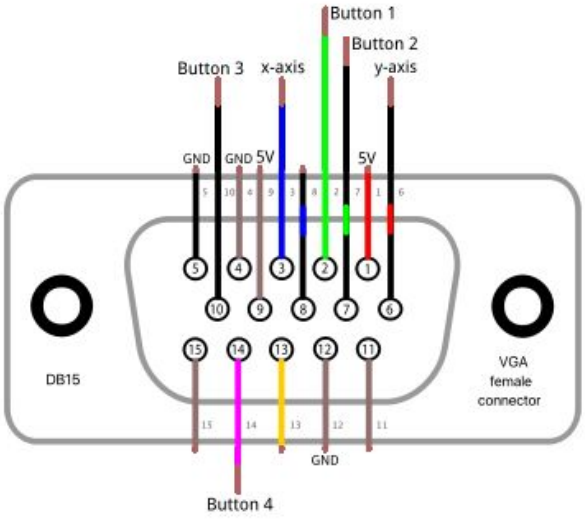

Figure 5: DB15 VGA Pinout

This is where the first challenge was encountered. The joystick works best at 5V and therefore outputs at 5V logic for both the digital buttons and analog joysticks. However, the TM4C123GH6PM microcontroller's digital inputs and ADCs function at 3.3V logic levels as shown in *Figure 6*. Therefore, it was called-for to design circuitry that maps the 0V-5V outputs from the joystick into the 0-3.3V range. These calculations are shown in *Equations 1 - 4* below, with the resulting circuits being shown in *Figures 7 & 8*. The first implementation of this circuit was then created on a breadboard for testing and tuning purposes as shown in *Figure 9*.

$$V_{x-axis} = 5V\left(\frac{10k\Omega}{10k\Omega + 5k\Omega + 0}\right) = 5V\left(\frac{10}{3}\right) \simeq 3.3V \quad (1)$$

$$V_{x-axis} = 5V\left(\frac{10k\Omega}{10k\Omega + 5k\Omega + 100k\Omega}\right) = 5V\left(\frac{10}{115}\right) \simeq 0V \quad (2)$$

$$V_{y-axis} = 5V\left(\frac{10k\Omega}{10k\Omega + 7.7k\Omega + 0}\right) = 5V\left(\frac{10}{17.7}\right) \simeq 3.3V \quad (3)$$

$$V_{y-axis} = 5V\left(\frac{10k\Omega}{10k\Omega + 7.7k\Omega + 100k\Omega}\right) = 5V\left(\frac{10}{117.7}\right) \simeq 0V \quad (4)$$

Table 24-33. ADC Electrical Characteristics[ab]

Parameter	Parameter Name	Min	Nom	Max	Unit
POWER SUPPLY REQUIREMENTS					
V_{DDA}	ADC supply voltage	2.97	3.3	3.63	V
GNDA	ADC ground voltage	-	0	-	V
VDDA / GNDA VOLTAGE REFERENCE					
C_{REF}	Voltage reference decoupling capacitance	-	1.0 // 0.01[c]	-	µF
ANALOG INPUT					
V_{ADCIN}	Single-ended, full- scale analog input voltage, internal reference[de]	0	-	V_{DDA}	V
	Differential, full-scale analog input voltage, internal reference[df]	$-V_{DDA}$	-	V_{DDA}	V
VIN_{CM}	Input common mode voltage, differential mode[g]	-	-	(VREFP + VREFN) / 2 ± 25	mV

Figure 6: Datasheet table used for analog input

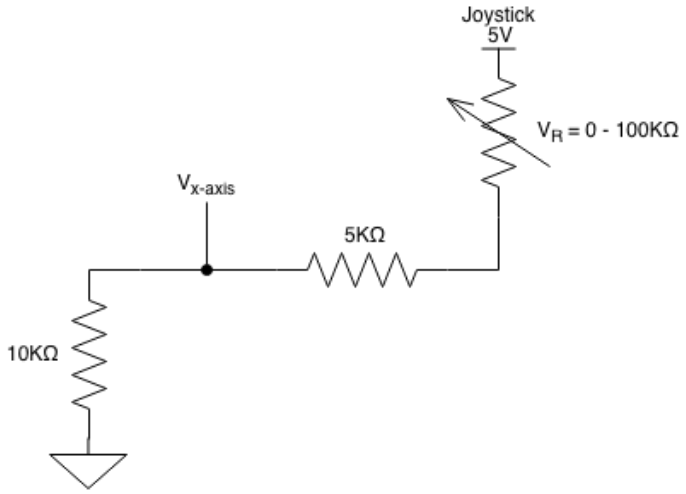

Figure 7: X-Axis Mapping Circuit

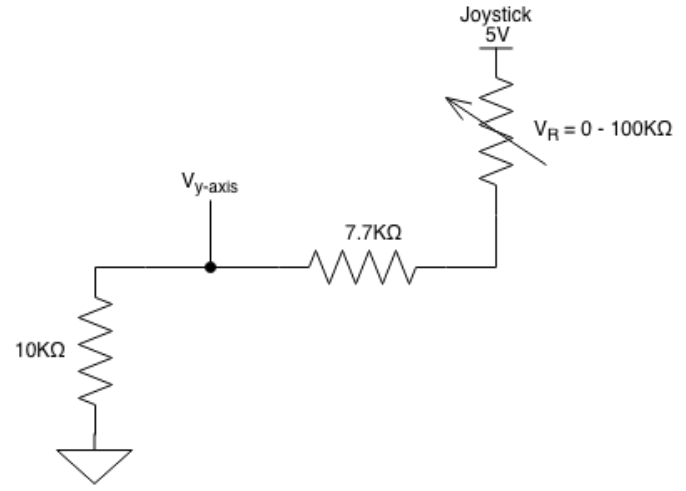

Figure 8: Y-Axis Mapping Circuit

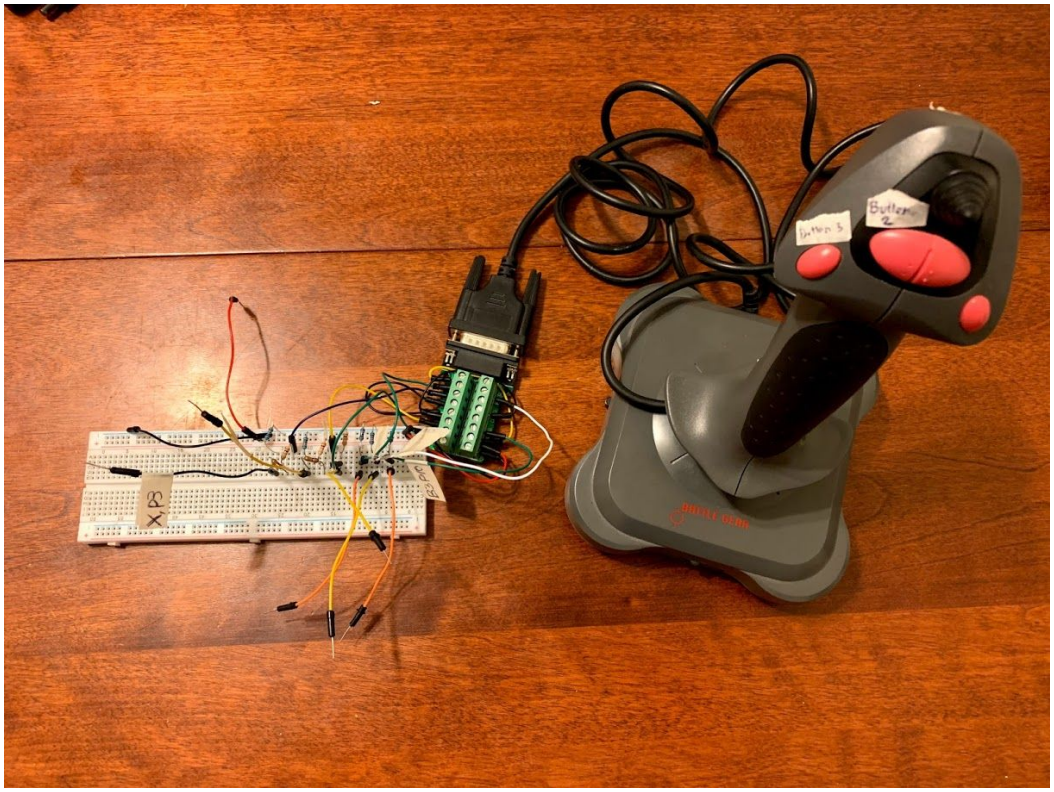

Figure 9: Mapping Circuits on BreadBoard

After wiring the circuit, a small testing program was written to confirm that all the buttons and axes were working properly as shown in *Figure 10.*

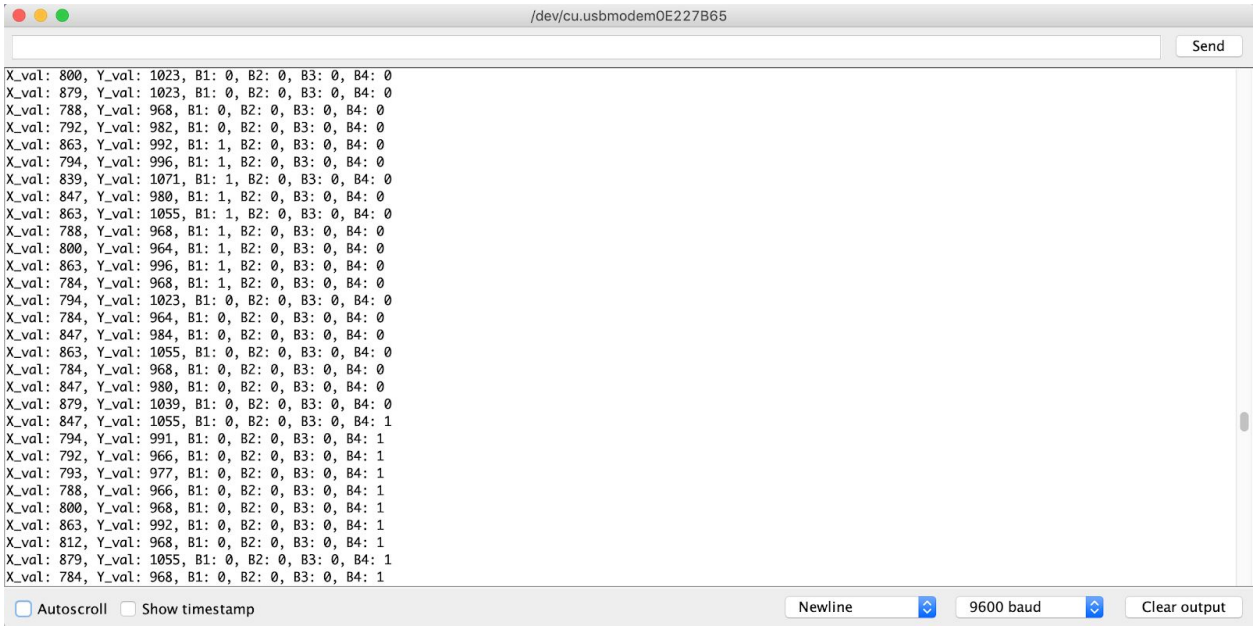

Figure 10: Initial Joystick Output Verification

Next, after verifying the proper functionality of the buttons and axes based on the resistors chosen, the circuit was designed on a prototyping board as shown in *Figure 11*.

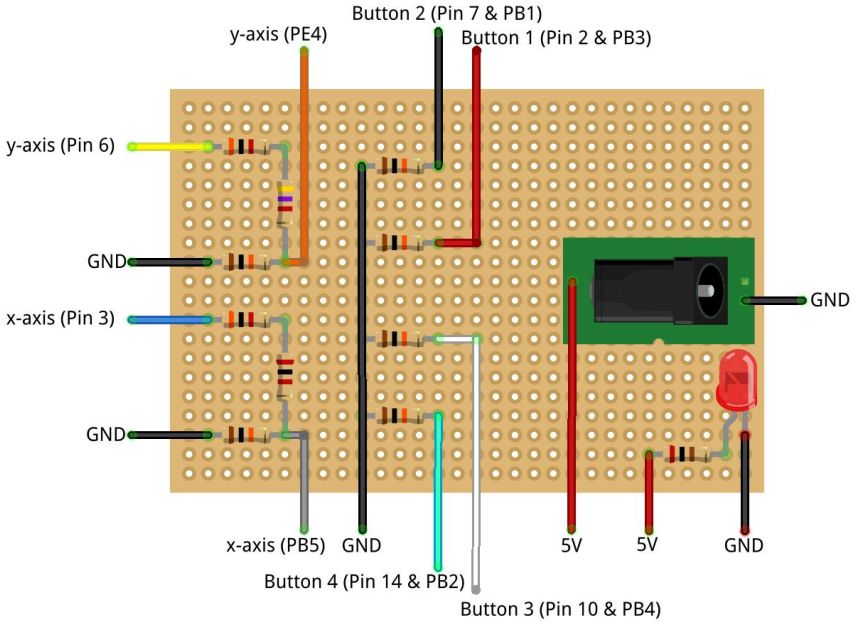

Figure 11: Schematic for VGA connector circuit

With the schematic shown in *Figure 11* the circuit for the joystick was built and connected to the VGA connector as shown in *Figures 12 and 13*.

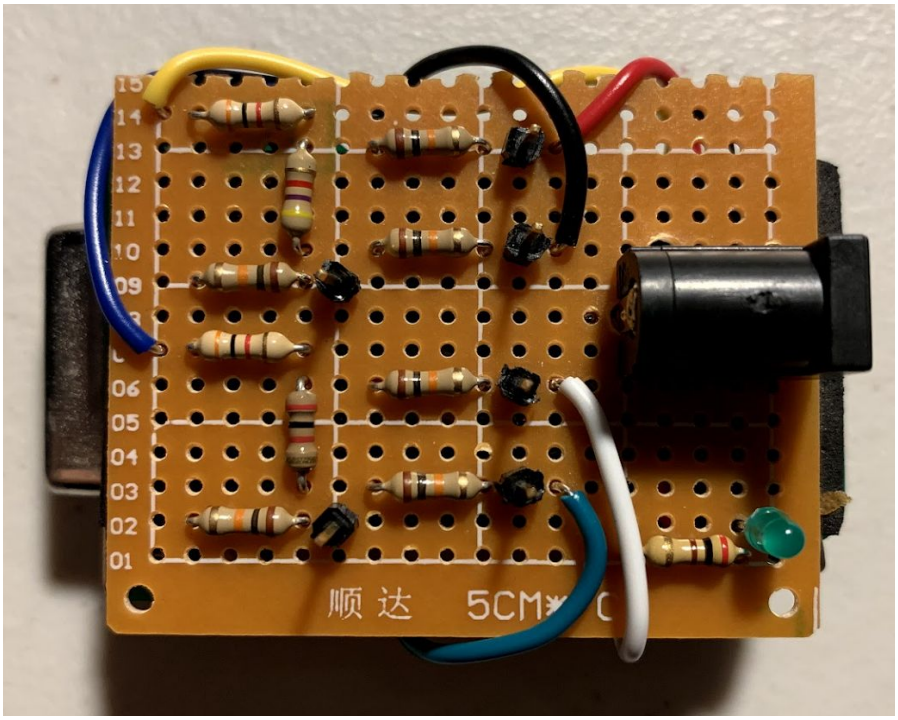

Figure 12: Circuit board installed on VGA connector

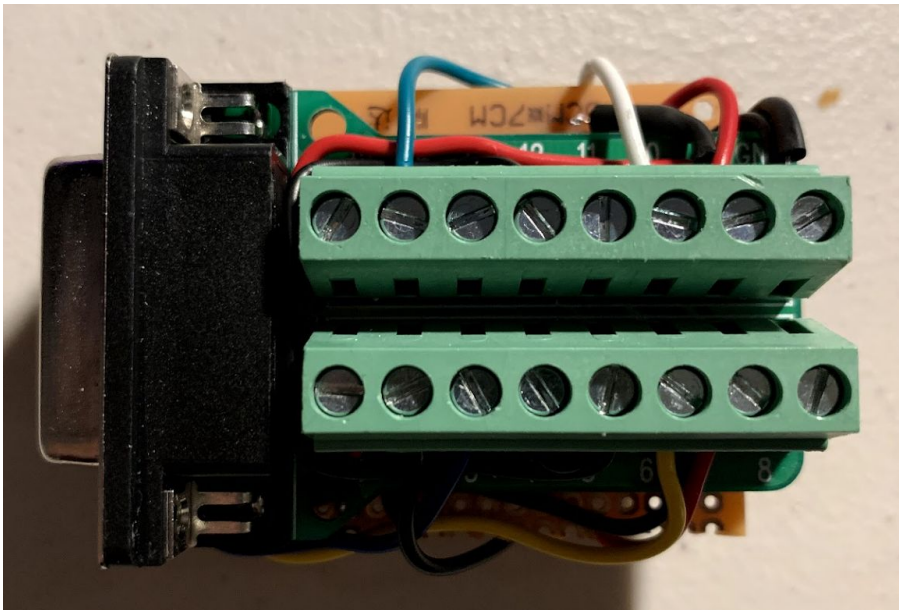

Figure 13: Circuit board installed on VGA connector

Phase 3: Port selection and µController wiring

During phase 3, the ports on the microcontroller were chosen based on the functions that those ports support. *Figure 14* shows the pinout of the TM4C123G microcontroller labeling which pins are analog and digital.

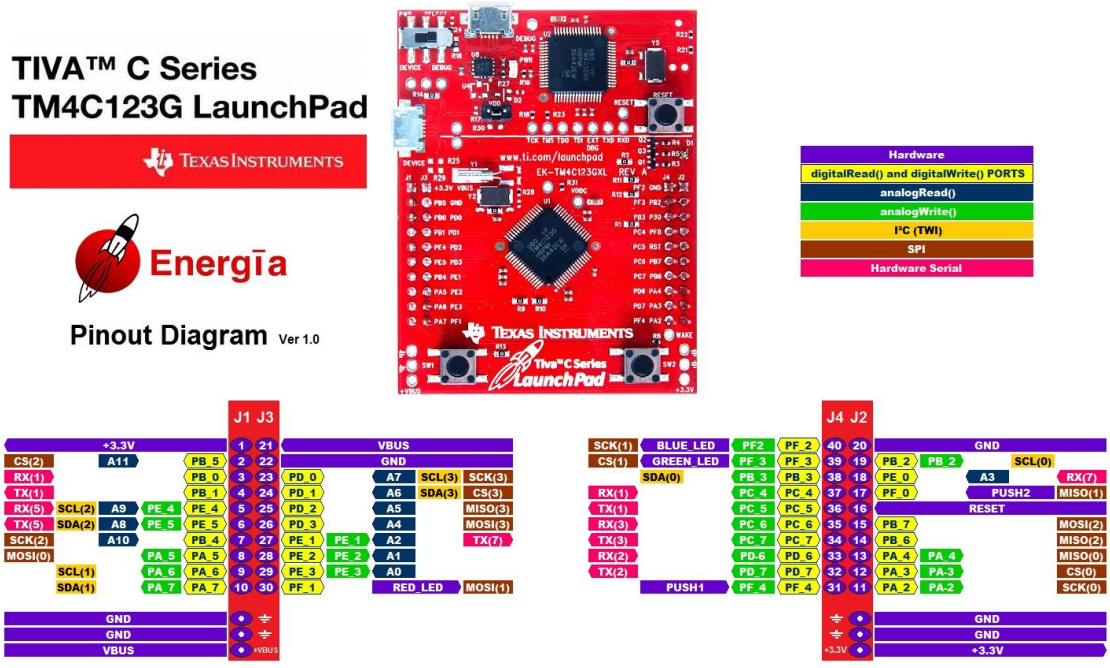

Figure 14: TM4C123G Pinout

Based on the pinouts in *Figure 14,* the ports used for the joystick and arm are listed in *Table 1* and *Table 2*.

Table 1: microcontroller ports used for joysticks

Device	Port	Axis/Button	Function
Joystick	PB5	X1	Analog
	PE4	Y1	Analog
	PB3	B1	Digital
	PB1	B2	Digital
	PB4	B3	Digital
	PB2	B4	Digital

Table 2: microcontroller ports used for arm

Device	Port	Function	Wire Color	Type
Arm	PE0	Up/Down	Black	PWM
	PE5	Forward/Back	Green	PWM
	PC5	Claw	Yellow	PWM
	PC4	Rotation	Blue	PWM

After defining the ports for the arm and joystick, the wires were connected to the microcontroller as shown in *Figure 15.*

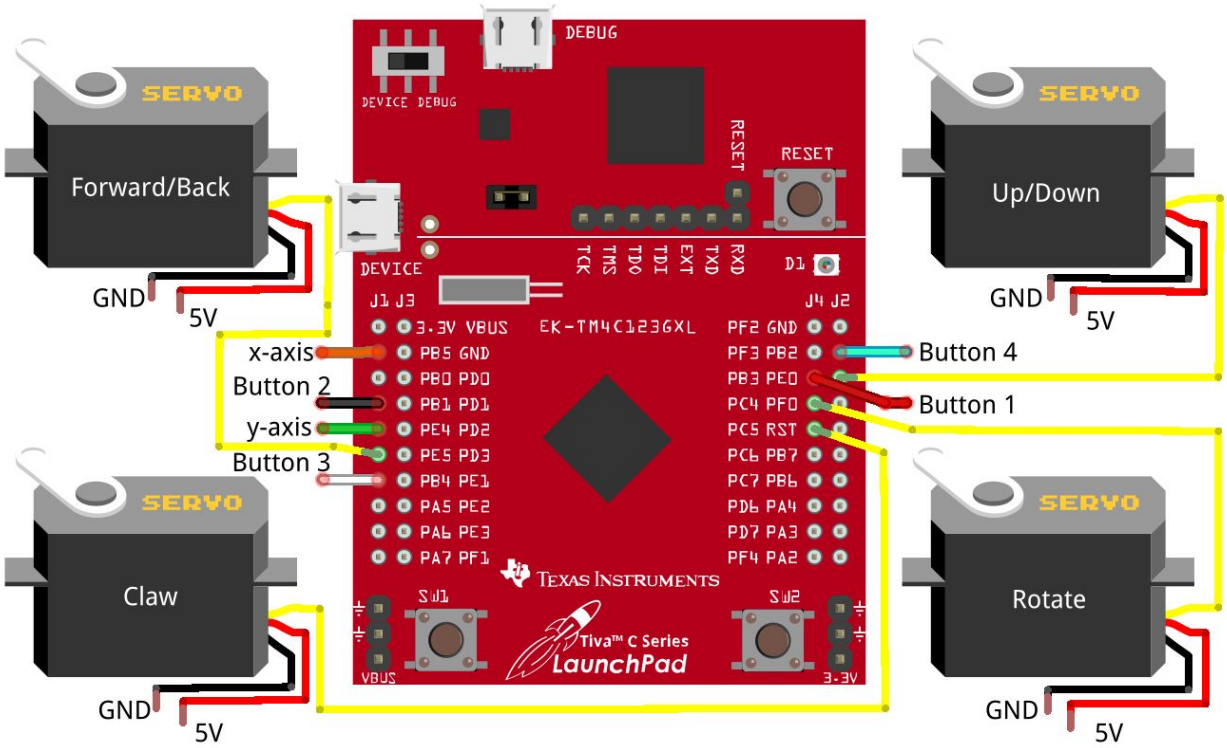

Figure 15: Connection diagram for the arm and joystick

Furthermore, the ports for the LCD display were also chosen based on functionality. *Table 3* shows the ports chosen on the microcontroller for each Pin of the LCD Display.

Table 3: LCD ports used on microcontroller

Device	Port	Pin
LCD	GND	VSS
	VBUS	VDD
	Potentiometer	VO
	PE2	RS
	GND	R/W
	PE1	Enable
	PD0	D4
	PD1	D5
	PD2	D6
	PD3	D7
	VBUS	A
	GND	K

Finally, the LCD was also connected to the microcontroller with its circuit setup as shown in *Figure 16*.

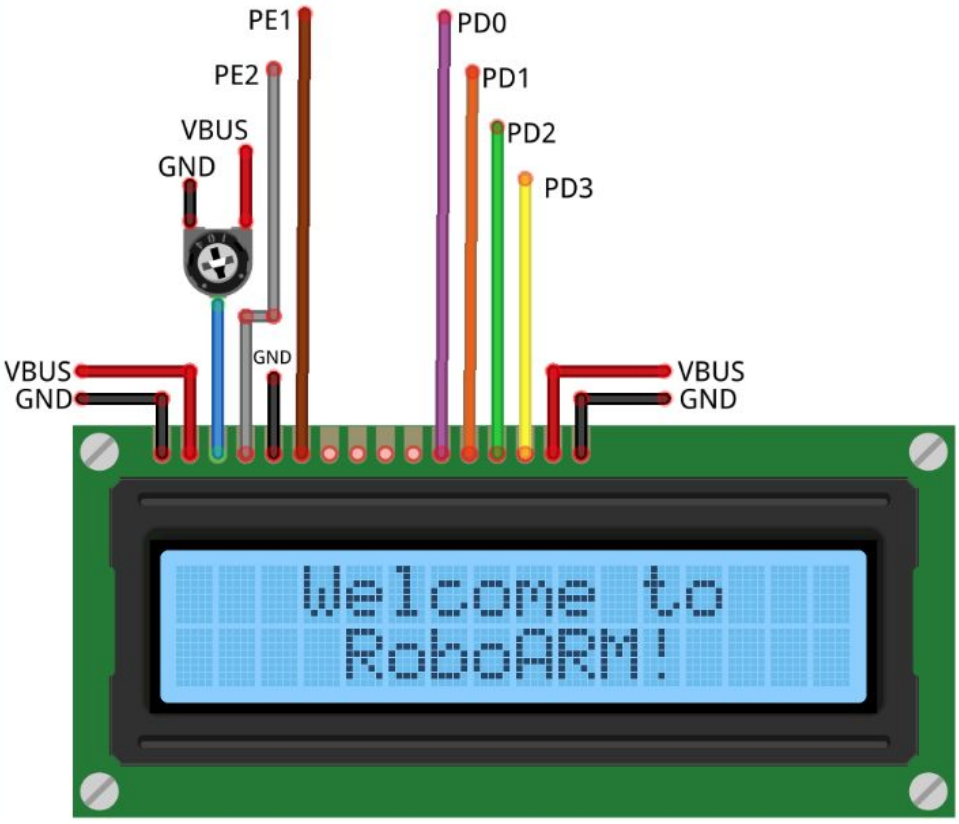

Figure 16: Connection diagram for the LCD Display

Phase 4: Programming the TI TM4C123GH6PM µController

Joystick Calibration:

After the circuitry for the joystick was created and tested to work properly, programs were created to aid in the software calibration process.

The first goal was to adjust the trim such that there was equal resolution on both sides of each axis. In order to accomplish this goal, the *x_delta_low_mid, x_delta_high_mid, y_delta_low_mid,* and *y_delta_high_mid* outputs were created. To use these, a starting point was established upon start and the joystick was moved to the max position in each direction. The corresponding variable would indicate the maximum change in that direction from the starting point. The trim linear potentiometers were then used to calibrate the axes such that they had approximately the same total change in each direction; ie the starting point was in the center of the range and not biased to one side.

The next goal was to account for the high sensitivity of the analog joystick inputs. Because the ADCs are 12 bit, naturally there is more resolution than required for the desired behavior. This caused the arm to constantly twitch and move upon initial testing. The *delta* variables confirm that even when not touched, the joystick axes values fluctuate significantly without user input. It was then decided that to address this issue, a deadzone would be needed so that inputs within a specified range of the starting point would be ignored and the arm would only move if the joystick was moved outside of this range. A deadzone requires a starting point and threshold on either side. To determine the starting point, variables *X_avg* and *Y_avg* were created that keep track of a rolling average of the starting position. After letting this program run for a minute or two we were confident with our starting positions. The *delta* variables were then monitored to determine the natural fluctuation of the values. With some trial and error, values were decided upon that eliminated accidental movement. The output of this program is shown in *Figure 17*.

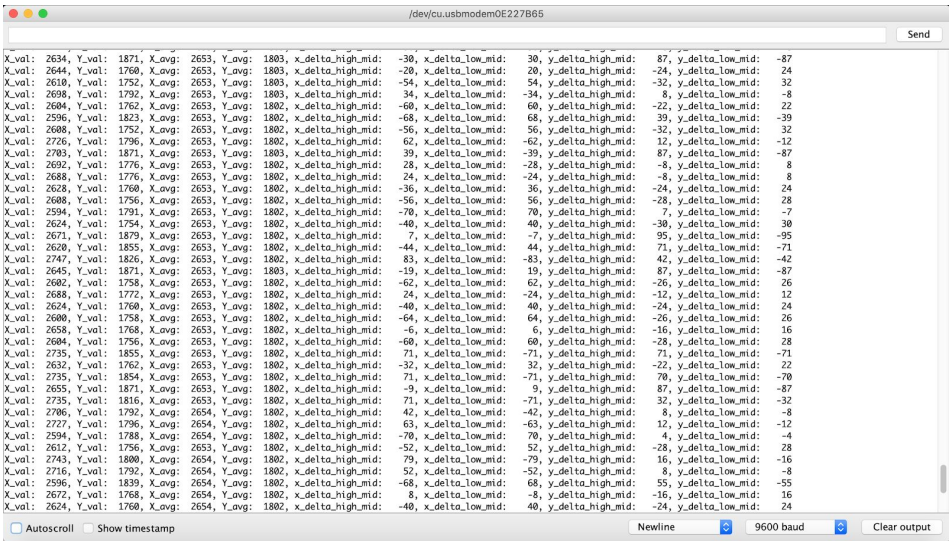

Figure 17: Calibration Program Output

Movement Bounds Calibration:

The next goal was to ensure that the servos would not move the arm past its physical limits allowed by the structure. In order to accomplish this, a DEBUG preprocessor directive was created. When this variable is defined and the program recompiled, the microcontroller will print to the serial monitor the position of each servo along with other debugging information. This is only enabled when needed because repeated "Serial.print" calls drastically reduce performance. Each axis was carefully moved to the maximum in each direction that the physical structure would allow. These values were then recorded and defined in the program as constants that could easily be changed and adjusted later. Throughout the program, the movement bounds-checking refers to these constants so that their values only need to be changed once. This greatly simplifies the maintainability of the code.

All of the movement bounds are constant except for one, which leads to the next development challenge. It was observed that the lower bound in the Up/Down directions would change depending on the Forward/Back position of the arm. This is because the arm structure can move further down when it is extended forward than when it is retracted back. This lead to an interesting problem that required dynamically changing the downward boundary of the arm. A simple illustration is shown in *Figure 18*. A function that performs this calculation is called when moving the arm down. It is also called when moving the arm backwards. If the arm moving backwards will cause the Up/Down position to violate the downward bound, the arm will be adjusted upwards slightly.

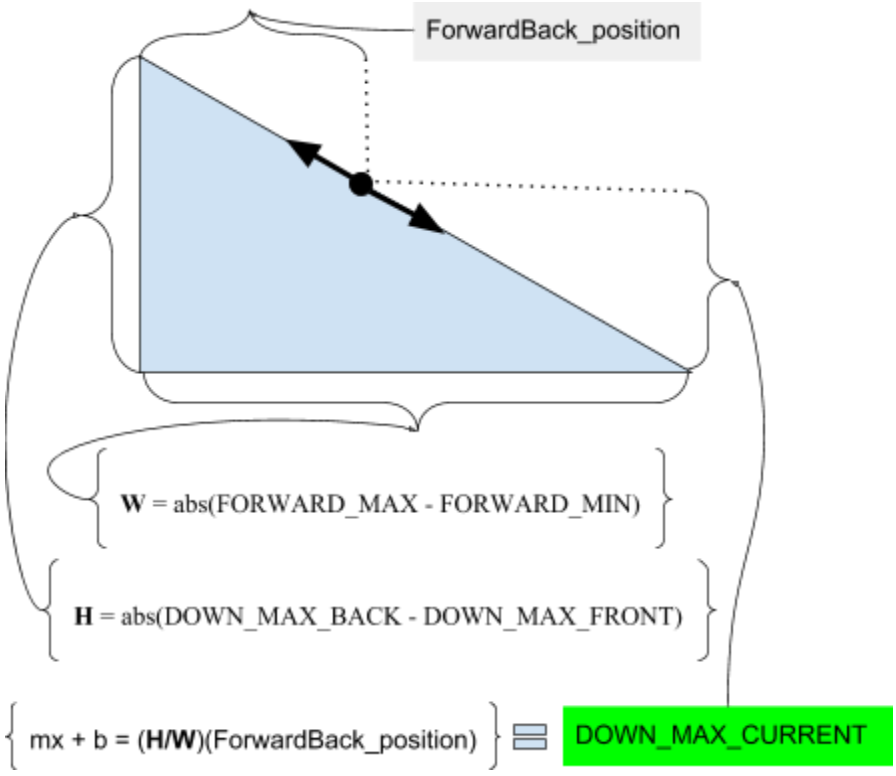

Figure 18: Dynamic Lower Bound Calculation

Initially, the implementation of this function caused lag and poor performance in Downward and Backwards movement. This is because calculating the height, width, and slope repeatedly with double data types uses a lot of processor time. Therefore, this function needed to be heavily optimized for the movement to be smooth and performant. The first step was realizing that the height and width of this triangle only needs to be calculated once. Therefore, this initial calculation was moved to the setup() function. Upon further research it was also realized that using floats instead of doubles would also greatly improve performance. The final version of the function performs a single calculation, multiplying the previously calculated slope of the triangle (float) and the current ForwardBack position (casted as a float). The result is then casted back to an integer type to be used as the current downward bound.

After some testing, it was realized that with the arm fully extended it became difficult to control rotational movement. This is due to the nature of concentric circles. As the radius (extension of the arm) is increased, the arc length of its rotation scales by 2π. To address this problem, a change was made that reduced the movement speed of the rotational axis based on its forward/back position such that it rotated less to keep the arc rotation constant.

Functionality Programming:

The functionality of the robot controls is described in *Table 4*.

Table 4: Joystick buttons and axis assigned to each movement

Movement	Axis/Button
Close Claw	Button 1
Up	Button 2
Down	Button 3
Open Claw	Button 4
Rotation	X-axis
Forward/Back	Y-axis
Record	PUSH1
Playback	PUSH2
Reset	RST

Upon starting, the setup() function is called. This function first enables the Serial Communication functionality. Then it calls the printWelcome() function which prints a welcome message routine to the LCD Screen. After the message routine completes, the pin modes and relevant pullup resistors are initialized. Then, interrupts are initialized to the recording and playback pins for a FALLING event because the onboard switches are active low. When this event occurs, the appropriate IVR functions are called which take care of enabling and disabling playback and recording modes. After this is done, the 4 servos are attached to their appropriate control pins. Next comes the Position Initializations where each Servo is moved to the midpoint of its high and low bounds. As mentioned earlier, the height, width, and slope are calculated once for the dynamic bounds adjustment to use later.

The program now enters the main() loop. If the arm is not recording or playing back a recording, the loop will repeatedly read the inputs and call the functions that deal with moving each axis. If a movement for that axis is requested, these functions will check if this movement is safe to perform without moving out of bounds. Then, the function will increment or decrement the position for that axis by the movement speed and write that location to the appropriate servo.

If the program receives an interrupt on the recording pin, it will then check the current mode, update the display, and enable or disable recording. If recording mode is enabled, the main loop will still perform the same functionality, however it will store the position of each servo in an array entry. It will continue to do so until there is no more space to record, or the recording is stopped. A progress bar will be shown to the user indicating the amount of time they have recorded and how much time they have left to record.

The playback function works similarly in that it is triggered by a hardware interrupt. It will check the state, check to see if there is a recording to playback, and update the displayed mode. It will then return to the main loop which will now instead of polling the inputs, retrieve the recorded positions of the servos and write them to their corresponding servo. Bounds checking does not need to be performed because this was already done during the recording phase. The user will be shown that they are in playback mode along with a progress bar indicating their playback progress. Once playback has concluded, the state will be set back to manual mode and the display will be updated accordingly.

The source code of the files shown in *Table 5* can be found in the *Appendix* section.

Table 5: File Descriptions

File Name	Description
RoboARM.ino	Main Program
InputTest.ino	Displays Outputs and Statistics for Calibration

III. TESTING PROCEDURE

Phase 5: Testing and Adjusting Functionality

During the testing procedure, the programmed movements were tested in order to understand the proper speeds of each axis. Initially it was realized that the speeds were too high and therefore made it hard to precisely control the arm. To decide on the appropriate speeds, a bottle was used as a sample object to verify the strength of the claw and the precision required to easily move this bottle from one location to another as shown in *Figure 19*. Extensive testing was done until the desired results were achieved.

Figure 19: Testing speed of axes and strength of the claw

The three different operating modes were able to be designed as a state machine. The three modes are Manual, Playback, and Record. Proper transitions between any combinations of the modes were programmed for and subsequently tested such that there would never be an undefined state. The modes are shown in *Figures 20 - 22.*

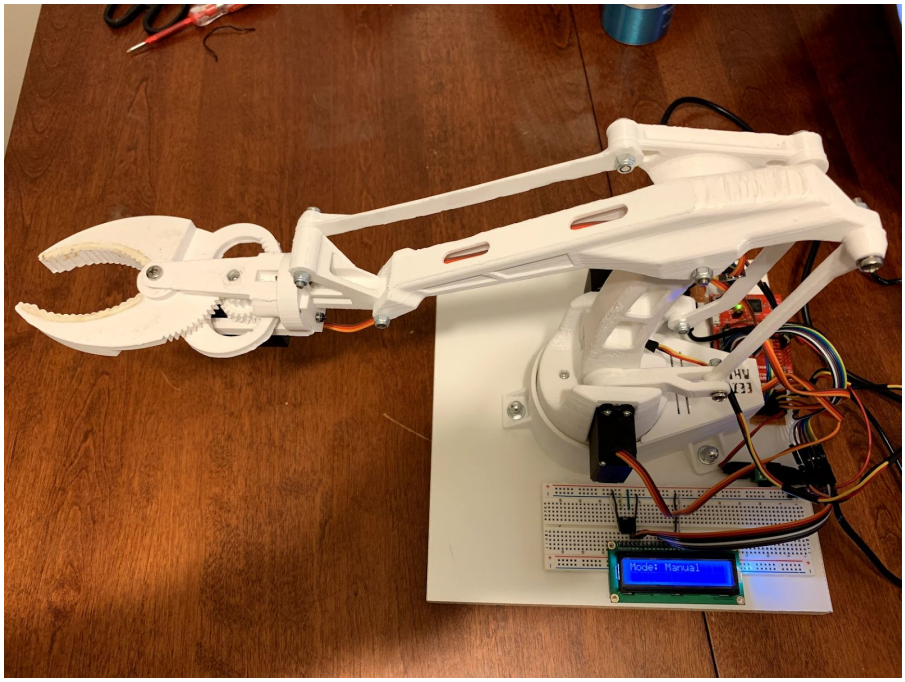

Figure 20: Manual Mode

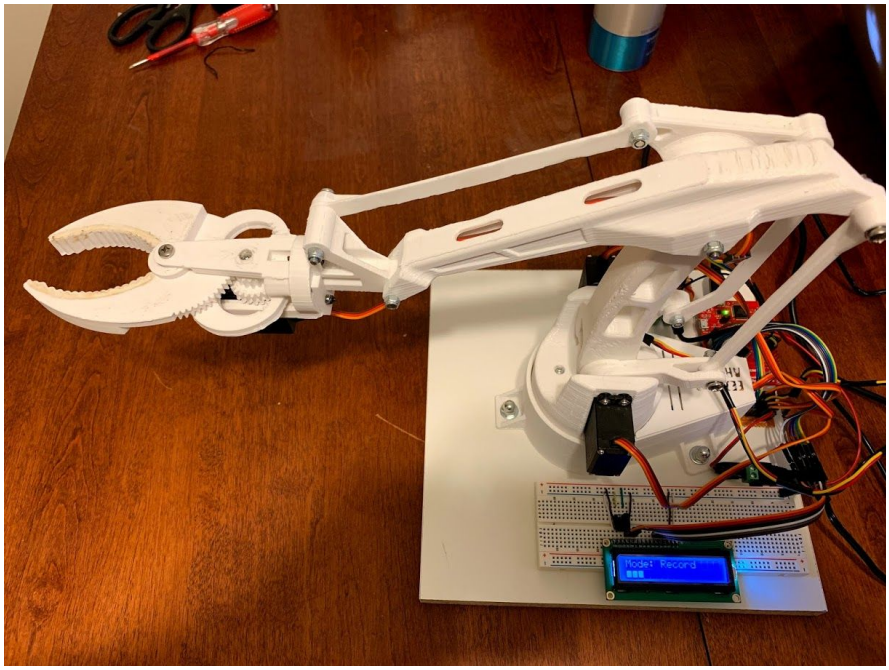

Figure 21: Recording Mode

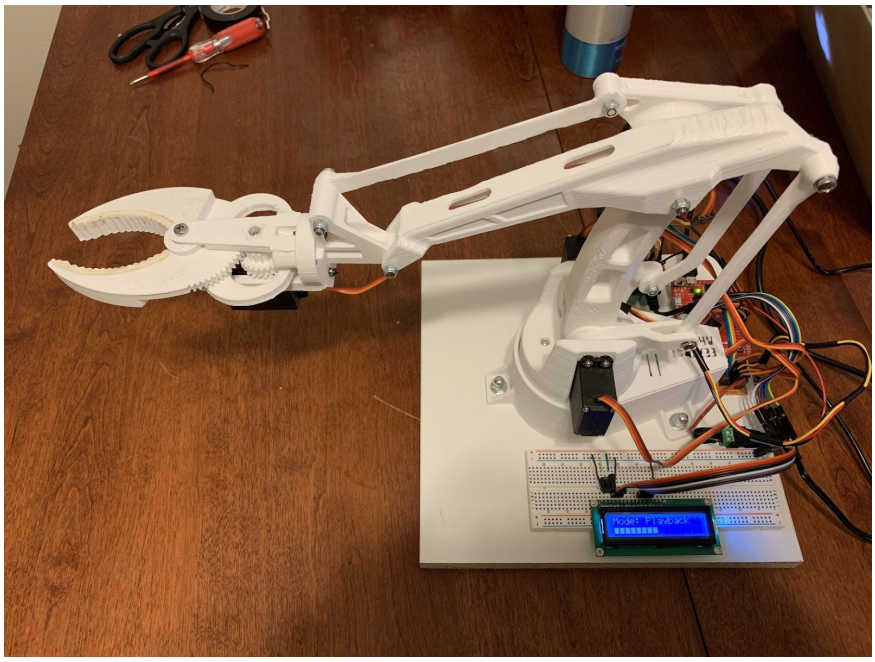

Figure 22: Playback Mode

Further along the testing procedures, it was determined that the maximum recording time was too short to be useful and needed to be improved. Because the recording is stored in RAM, there is not an abundance of space to work with. Initially, the 4 servo positions were stored in a 1024 by 4 array of integers which allowed for approximately 23 seconds of recording time. An integer on this platform is 4 bytes which allows for a range of -2,147,483,648 to 2,147,483,647. This range is far more than needed for this purpose and therefore was wasting valuable space. The

next attempt involved using 2 byte unsigned short integers which allowed for 3072 entries due to data alignment. The positions stored for each servo are in the range 0 to 180 which finally allowed for the usage of an unsigned char data type. This data type is only 1 byte (range 0 - 255) which greatly increased the storage efficiency. This lead to safely being able to store around 7000 entries which equates to a recording time approaching 3 minutes; a drastic improvement compared to the original 23 seconds.

IV. CONCLUSION

This project originally started as an open-ended implementation of the TI TM4C123GH6PM microcontroller with the purpose of becoming familiar with the platform including the onboard ADCs and interrupts. It eventually evolved into that and much more. The design ended up being problem/solution driven; fixing one problem after the next. Some of these problems include but are not limited to servo power delivery, memory optimization, circuit analysis, 3D space calculations, computation optimization, and physical structure limitations.

Further considerations can include using more capable hardware in regards to SBC for more computational power, stepper-motors for increased accuracy, strength, and control, and structural design for more fluid and consistent movement characteristics. Adding additional sensors such as a camera for visual learning and safety, proximity sensors and more could also improve the possible software functionality of the device.

Overall this project lead to a greatly beneficial learning experience in regards to hardware design, software design, physical design, and troubleshooting skills that will be useful in future projects.

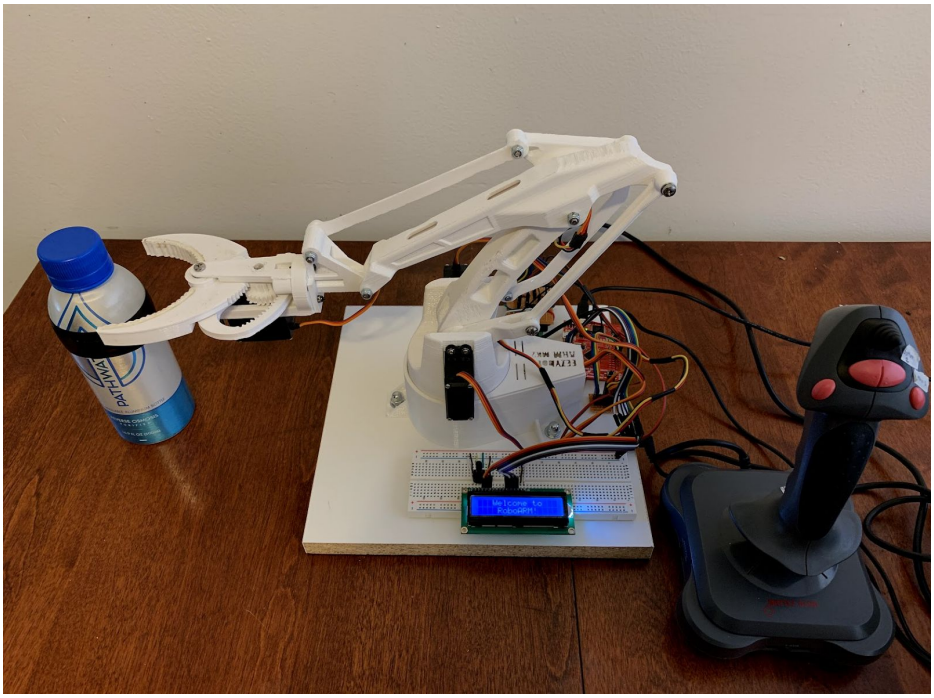

Figure 23: Final Implementation

V. APPENDIX

SOURCE CODE:

RoboARM.ino

```
/*
  RoboARM driver code:
  Pin Mappings:
    Joystick:
      (Port) : (Function) : (Signal)
       PB5   :    X1      : Analog
       PE4   :    Y1      : Analog
       PB1   :    B2      : Digital
       PB4   :    B3      : Digital
       PB3   :    B1      : Digital
       PB2   :    B4      : Digital
    Arm:
      (Port)   : (Function)   (Color)  : (Signal)
        PE0    : Up/Down      (Black)  :   PWM
        PE5    : Forward/Back (Green)  :   PWM
        PC5    : Claw        (Yellow) :   PWM
        PC4    : Rotation     (Blue)   :   PWM

    Movement:
      Button 1: Close Claw
      Button 2: Up
      Button 3: Down
      Button 4: Open Claw

      x-axis: Rotation
      y-axis: Forward/Back

    Deadzone:
      x-avg = 1900 +/- 600
      y-avg = 1832 +/- 500

 The circuit for LCD:
 * LCD RS pin to digital pin 28 (PE2)
 * LCD Enable pin to digital pin 27  (PE1)
 * LCD D4 pin to digital pin 23 (PD0)
 * LCD D5 pin to digital pin 24 (PD1)
 * LCD D6 pin to digital pin 25 (PD2)
 * LCD D7 pin to digital pin 26 (PD3)
 * LCD R/W pin to ground
 * 10K resistor:
 * ends to +5V and ground
 * wiper to LCD VO pin (pin 3)

 */
#include <stdio.h>
#include <Servo.h>
#include <LiquidCrystal.h>


// #define DEBUG /* uncomment for Serial Debugging Output (causes severe lag with actual
movements...Prints are expensive) */

//Servo Bounds
#define CLAW_OPEN_MAX 1
#define CLAW_CLOSED_MAX 150
#define CLAW_MOVEMENT_SPEED 5
```

```
#define ROTATE_LEFT_MAX 0
#define ROTATE_RIGHT_MAX 179
#define ROTATION_SPEED 2


#define UP_MAX 120
#define DOWN_MAX 74
#define UPDOWN_MOVEMENT_SPEED 2

#define DOWN_MAX_BACK 74
#define DOWN_MAX_FORWARD 0
int   downmax_height = 0;
int   downmax_width  = 0;
float downmax_slope  = 0;

#define FORWARD_MAX 179
#define BACK_MAX 89
#define FORWARD_BACK_SPEED 1
#define SENSITIVE_OFFSET 20

//Joystick Calibration
#define X_START 1900 //X Middle Point
#define Y_START 1832 //Y Middle Point
#define X_RANGE 600  //X Deadzone Threshold
#define Y_RANGE 500  //Y Deadzone Threshold

//recording
//#define RECORD_LENGTH (BUFSIZ + 500)
//#define RECORD_LENGTH (3*BUFSIZ + 500)
#define RECORD_LENGTH (6*BUFSIZ + 600)
int record_length =  RECORD_LENGTH;
//int record[RECORD_LENGTH][4];
//unsigned short int record[RECORD_LENGTH][4];
unsigned char record[RECORD_LENGTH][4];
int count = 0; //current recording position
bool recording = false;
bool playing   = false;
int cursor = 0;//current playback position

int progress_cursor = 0; // used for progress bar for recording and playback

//Inputs (used to store input values)
int x_val = 0;
int y_val = 0;

int B1_val = 0;
int B2_val = 0;
int B3_val = 0;
int B4_val = 0;


/* Analog */
int X1 = PB_5;
int Y1 = PE_4;

/* Digital */
int B2_pin = PB_1;
int B3_pin = PB_4;
int B1_pin = PB_3;
int B4_pin = PB_2;

/* Servos */
int UpDown_pin      = PE_0;
int ForwardBack_pin = PE_5;
int Claw_pin        = PC_5;
int Rotate_pin      = PC_4;
```

```
/* Positional Variables */
int ClawPosition        = 0;
int RotationPosition    = 0;
int UpDownPosition      = 0;
int ForwardBackPosition = 0;

char str[200];

int current_down_max = 0;
bool adjusting_back  = false;

//Set up Servos
Servo UpDown;
Servo ForwardBack;
Servo Claw;
Servo Rotate;

// initialize the LCD library with the numbers of the interface pins (see mapping at top of
file)
LiquidCrystal lcd(28, 27, 23, 24, 25, 26);

void moveClaw(char OpenOrClose){

  switch (OpenOrClose){
    case 'o' :
        if (ClawPosition > CLAW_OPEN_MAX + CLAW_MOVEMENT_SPEED){
            ClawPosition -= CLAW_MOVEMENT_SPEED;
            Claw.write(ClawPosition);
            #ifdef DEBUG
            sprintf(str, "Opening Claw. Claw Position: %d\n", ClawPosition);
            Serial.print(str);
            #endif

        }
        break;
    case 'c' :
        if (ClawPosition < CLAW_CLOSED_MAX - CLAW_MOVEMENT_SPEED){
            ClawPosition += CLAW_MOVEMENT_SPEED;
            Claw.write(ClawPosition);
            #ifdef DEBUG
            sprintf(str, "Closing Claw. Claw Position: %d\n", ClawPosition);
            Serial.print(str);
            #endif
        }
        break;
    case 'i':
        int default_val;
        default_val = (CLAW_CLOSED_MAX - CLAW_OPEN_MAX)/2;
        ClawPosition = default_val;
        Claw.write(default_val);
        #ifdef DEBUG
        sprintf(str, "Initializing CLaw. Claw Position: %d\n", ClawPosition);
        Serial.print(str);
        #endif
        break;
    default:
        break;
  }
}

void moveUpDown(char UpOrDown){

  switch (UpOrDown){
    case 'u' : //move up
        if (UpDownPosition + UPDOWN_MOVEMENT_SPEED < UP_MAX ){
```

```
            UpDownPosition += UPDOWN_MOVEMENT_SPEED;
            UpDown.write(UpDownPosition);
            #ifdef DEBUG
            sprintf(str, "Moving Up. UpDown Position: %d\n", UpDownPosition);
            Serial.print(str);
            #endif

        }
        break;
    case 'd' : //move down
            if (!adjusting_back){
              calcDownMax();
            }
        if (UpDownPosition - UPDOWN_MOVEMENT_SPEED > current_down_max){
            UpDownPosition -= UPDOWN_MOVEMENT_SPEED;
            UpDown.write(UpDownPosition);
            #ifdef DEBUG
            sprintf(str, "Moving Down. UpDown Position: %d, Current DownMax: %d\n",
UpDownPosition, current_down_max);
            Serial.print(str);
            #endif
        }
        break;
    case 'i': //initialize
        int default_val;
        //default_val = abs(UP_MAX - DOWN_MAX)/2;
        default_val = 100;
        UpDown.write(default_val);
        UpDownPosition = default_val;
        #ifdef DEBUG
        sprintf(str, "Initializing UpDown. UpDown Position: %d\n", UpDownPosition);
        Serial.print(str);
        #endif
        break;
    default:
        break;
  }
}

void moveArmRotate(){
    //Rotate
    if (x_val > X_START + X_RANGE){ //Move if input value is outside of middle deadzone
      //Rotate Left
      if (ForwardBackPosition >= (FORWARD_MAX - SENSITIVE_OFFSET)){ //If close to the max
front position, move more slowly
        if (RotationPosition - (ROTATION_SPEED - 1) >= ROTATE_LEFT_MAX){
          RotationPosition -= (ROTATION_SPEED - 1);
          Serial.println("Sensitive Left");
          Rotate.write(RotationPosition);
          #ifdef DEBUG
          sprintf(str, "Rotating Left. Rotation Position: %d\n", RotationPosition);
          Serial.print(str);
          #endif
        }
      }

      else if (RotationPosition - ROTATION_SPEED >= ROTATE_LEFT_MAX){ //check to see if
movement will go out of bounds
        RotationPosition -= ROTATION_SPEED;
        Rotate.write(RotationPosition);
        #ifdef DEBUG
        sprintf(str, "Rotating Left. Rotation Position: %d\n", RotationPosition);
        Serial.print(str);
        #endif
      }
    }
```

```
    if (x_val < X_START - X_RANGE){ //Move if input value is outside of middle deadzone
      //Rotate Right
      if (ForwardBackPosition >= (FORWARD_MAX - SENSITIVE_OFFSET)){ //If close to the max
front position, move more slowly
        if (RotationPosition + (ROTATION_SPEED - 1) <= ROTATE_RIGHT_MAX){ //check to see if
movement will go out of bounds
          RotationPosition += (ROTATION_SPEED - 1);
          Serial.println("Sensitive Right");
          Rotate.write(RotationPosition);
          #ifdef DEBUG
          sprintf(str, "Rotating Right. Rotation Position: %d\n", RotationPosition);
          Serial.print(str);
          #endif
        }
      }
      else if (RotationPosition + ROTATION_SPEED <= ROTATE_RIGHT_MAX){ //check to see if
movement will go out of bounds
        RotationPosition += ROTATION_SPEED;
        Rotate.write(RotationPosition);
        #ifdef DEBUG
        sprintf(str, "Rotating Right. Rotation Position: %d\n", RotationPosition);
        Serial.print(str);
        #endif
      }
    }
}

void moveArmForwardBack(){
  //Forward or Back
    if (y_val > Y_START + Y_RANGE){ //Move if input value is outside of middle deadzone
      //Move Forward
      if (ForwardBackPosition + FORWARD_BACK_SPEED <= FORWARD_MAX){ //check to see if
movement will go out of bounds
        ForwardBackPosition += FORWARD_BACK_SPEED;
        ForwardBack.write(ForwardBackPosition);
        #ifdef DEBUG
        sprintf(str, "Moving Forward. ForwardBack Position: %d\n", ForwardBackPosition);
        Serial.print(str);
        #endif
      }
    }
    if (y_val < Y_START - Y_RANGE){ //Move if input value is outside of middle deadzone
      //Move Back
      if (ForwardBackPosition - FORWARD_BACK_SPEED >= BACK_MAX){ //check to see if movement
will go out of bounds

        ForwardBackPosition -= FORWARD_BACK_SPEED;
        ForwardBack.write(ForwardBackPosition);
        #ifdef DEBUG
        sprintf(str, "Moving Back. ForwardBack Position: %d\n", ForwardBackPosition);
        Serial.print(str);
        #endif
        calcDownMax();
        /* If the arm is moved back and is lower than the lowest allowed position for that
ForwardBackPosition, we need to move the arm up.*/
        if (UpDownPosition < current_down_max){
          #ifdef DEBUG
          Serial.println("adjusting\n");
          #endif
          adjusting_back = true;
          moveUpDown('u');
          adjusting_back = false;
        }
      }
    }
```

```
}

/* Used to calculate the lowest point the arm can go depending on its current
ForwardBackPosition (Think Triangle, y=mx+b) */
void calcDownMax(){
  current_down_max = (int)((float)downmax_slope*(float)abs(FORWARD_MAX -
ForwardBackPosition));
  #ifdef DEBUG
  sprintf(str, "height: %d, width: %d, slope: %g, currentdownmax: %d\n", downmax_height,
downmax_width, downmax_slope, current_down_max);
  Serial.print(str);
  #endif

}

/* Prints the Welcome Message when the bot is powered on or reset */
void printWelcome(){
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("   Welcome to");
  delay(2000);
  lcd.setCursor(0, 1);
  lcd.print("    RoboARM!");
  delay(2000);
  lcd.setCursor(0,0);
  lcd.print("                ");
  lcd.setCursor(0,1);
  lcd.print("   Welcome to");
  delay(750);
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Initializing");
  int lcd_cursor = 12;
  for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
      lcd.setCursor(lcd_cursor++, 0);
      lcd.print(".");
      delay(600);
    }
    lcd.setCursor(12,0);
    lcd.print("   ");
    lcd_cursor = 12;
    delay(500);
  }
  byte block[8] = {
    0b00000,
    0b11111,
    0b11111,
    0b11111,
    0b11111,
    0b11111,
    0b11111,
    0b00000
  };
  lcd.createChar(0, block);
  byte play[8] = {
    0b01000,
    0b01100,
    0b01110,
    0b01111,
    0b01110,
    0b01100,
    0b01000,
    0b00000
  };
```

```
  lcd.createChar(1, play);
  byte rec[8] = {
    0b00000,
    0b01110,
    0b11111,
    0b11111,
    0b11111,
    0b01110,
    0b00000,
    0b00000
  };
  lcd.createChar(2, rec);
}

void setup() {



  Serial.begin(9600);
  Serial.print("Starting\n");
  printWelcome();
  lcd.setCursor(0,0);
  lcd.print("Mode: Manual");
  //Setup digital inputs
  pinMode(B1_pin,INPUT);
  pinMode(B2_pin,INPUT);
  pinMode(B3_pin,INPUT);
  pinMode(B4_pin,INPUT);
  pinMode(PUSH1, INPUT_PULLUP);
  pinMode(PUSH2, INPUT_PULLUP);

  //recording and playback Interrupt Initialization
  attachInterrupt(digitalPinToInterrupt(PUSH1), toggleRecord, FALLING);
  attachInterrupt(digitalPinToInterrupt(PUSH2), togglePlayback, FALLING);

  //Attach Servos
  UpDown.attach(UpDown_pin);
  ForwardBack.attach(ForwardBack_pin);
  Claw.attach(Claw_pin);
  Rotate.attach(Rotate_pin);

  //Set Servos to Default
  moveClaw('i');
  moveUpDown('i');

  //intitialize Rotation to middle point of bounds
  RotationPosition = abs(ROTATE_LEFT_MAX - ROTATE_RIGHT_MAX)/2;
  Rotate.write(RotationPosition);
  sprintf(str, "Initializing Rotate. RotationPosition: %d\n", RotationPosition);
  Serial.print(str);

  //Initialize ForwardBack
  //ForwardBackPosition = abs(FORWARD_MAX - BACK_MAX)/2;
  ForwardBackPosition = 100;
  ForwardBack.write(ForwardBackPosition);
  sprintf(str, "Initializing ForwardBack. ForwardBackPosition: %d\n", ForwardBackPosition);
  Serial.print(str);


  downmax_height = abs(DOWN_MAX_BACK - DOWN_MAX_FORWARD);
  downmax_width  = abs(FORWARD_MAX - BACK_MAX);
  downmax_slope  = (float)downmax_height/(float)downmax_width;
  calcDownMax();

}
```

```
/* Recording is toggled via hardware interrupt if record button is pressed */
void toggleRecord(){

  if (recording){
    #ifdef DEBUG
    sprintf(str, "Stopping Recording... Length: %d\n", count);
    Serial.print(str);
    #endif
    lcd.setCursor(6,0);
    lcd.print("Manual    ");
    recording = false;
    lcd.setCursor(0,1);
    lcd.print("                ");
    progress_cursor = 0;
  }
  else{
    #ifdef DEBUG
    Serial.println("Starting Recording...");
    #endif
    lcd.setCursor(6,0);
    lcd.print("Record ");
    lcd.write((byte) 2);
    recording = true;
    count = 0;
  }



}

/* Playback is toggled via hardware interrupt if playback button is pressed */
void togglePlayback(){

  if (recording) //If recording, the recording needs to be concluded before playback can
occur
      recording = false;
      lcd.setCursor(0,1);
      lcd.print("                ");
      progress_cursor = 0;
  if (!playing){
    if (count != 0){
      #ifdef DEBUG
      Serial.println("Playing");
      #endif
      lcd.setCursor(6,0);
      lcd.print("Playback ");
      lcd.write((byte) 1);
      playing = true;
      cursor = 0;
      }
      else {
        #ifdef DEBUG
        Serial.println("No recording to play back");
        #endif
      }
    }
    else{
      playing = false;
      #ifdef DEBUG
      Serial.println("Playback Finished");
      #endif
      lcd.setCursor(0,1);
      lcd.print("    Finished");
      delay(1000);
      lcd.setCursor(0,1);
```

```
      lcd.print("                ");
      lcd.setCursor(6,0);
      lcd.print("        ");
      lcd.setCursor(6,0);

      lcd.print("Manual    ");
      cursor = 0;
    }

}


void loop() {

  // read the values from the sensor if playback is not active:
  if (!playing){
    x_val = analogRead(X1);
    y_val = analogRead(Y1);

    B1_val = digitalRead(B1_pin);
    B2_val = digitalRead(B2_pin);
    B3_val = digitalRead(B3_pin);
    B4_val = digitalRead(B4_pin);

    /* Claw */
   if (B1_val == HIGH){
      moveClaw('c'); //close claw
   }
   else if (B4_val == HIGH){
      moveClaw('o'); //open claw
   }

    if (B2_val == HIGH){
      moveUpDown('u'); //move up
    }
    else if (B3_val == HIGH){
      moveUpDown('d'); //move down
    }

    moveArmRotate();
    moveArmForwardBack();

    /* If recording and still have room to record, store current positions into record array
*/
    if (recording){
        if (count != record_length - 1){
          #ifdef DEBUG
          Serial.print("count:");
          Serial.println(count);
          #endif
          record[count][0] = ClawPosition;
          record[count][1] = RotationPosition;
          record[count][2] = UpDownPosition;
          record[count][3] = ForwardBackPosition;
          count++;

          /* Map recording position to [0 : 16] and print status bar accordingly */
          int progress_block = map(count, 0, record_length, 0, 17);
          if (progress_block > progress_cursor){
            lcd.setCursor(progress_cursor, 1);
            lcd.write((byte) 0);
            progress_cursor = progress_block;
          }
      }
      else {
        toggleRecord();
```

```
      }
    }
  }
  else {
    /* If Playing Back, write recorded position values directly to Servos */
    Claw.write(record[cursor][0]);
    Rotate.write(record[cursor][1]);
    UpDown.write(record[cursor][2]);
    ForwardBack.write(record[cursor][3]);

    /* Map current playback position out of record length into 0 : 16. Then write to status
bar */
    int progress_block = map(cursor, 0, count, 0, 17);
    if (progress_block > progress_cursor){
      lcd.setCursor(progress_cursor, 1);
      lcd.write((byte) 0);
      progress_cursor = progress_block;
    }
    if (cursor < count - 1){
      cursor++;
    }
    else {
      togglePlayback(); //stop playback once it has reached the end
    }
  }
   delay(20);
}
```

InputTesting.ino

```
/*
  Analog Input
 Used for Calibration/Statistics for analog inputs from joystick.

 */
#include <stdio.h>


int x_val = 0;
int y_val = 0;

int B1_val = 0;
int B2_val = 0;
int B3_val = 0;
int B4_val = 0;

//For delta tuning
bool firstpass = true;
int firstpass_x = 0;
int firstpass_y = 0;
int x_delta_high_mid = 0;
int x_delta_low_mid = 0;
int y_delta_high_mid = 0;
int y_delta_low_mid = 0;


/* Analog */
int X1 = PB_5;
int Y1 = PD_2;
/* Digital */
int B2_pin = PB_1;
int B3_pin = PB_4;
int B1_pin = PB_3;
int B4_pin = PB_2;
```

```
 //Average
 long long sum_x = 0;
 long long sum_y = 0;
 long count = 0;
 int avg_x, avg_y;

char str[2048];

void setup() {
  pinMode(B1_pin,INPUT);
  pinMode(B2_pin,INPUT);
  pinMode(B3_pin,INPUT);
  pinMode(B4_pin,INPUT);

  Serial.begin(9600);
}

void loop() {

  // get middle values for x and y (for tuning)
  if (firstpass){
    firstpass_x = analogRead(X1);
    firstpass_y = analogRead(Y1);
    firstpass = false;
  }

  // read the value from the sensor:
  x_val = analogRead(X1);
  y_val = analogRead(Y1);
  // turn the ledPin on
  B1_val = digitalRead(B1_pin);
  B2_val = digitalRead(B2_pin);
  B3_val = digitalRead(B3_pin);
  B4_val = digitalRead(B4_pin);

  //Average Calculation
  sum_x += x_val;
  sum_y += y_val;
  count++;
  avg_x = sum_x / count;
  avg_y = sum_y / count;

  // Calculate Deltas (for tuning)
  x_delta_high_mid = x_val - firstpass_x;
  x_delta_low_mid  = firstpass_x - x_val;

  y_delta_high_mid = y_val - firstpass_y;
  y_delta_low_mid  = firstpass_y - y_val;


  //sprintf(str, "X_val: %d, Y_val: %d, B1: %d, B2: %d, B3: %d, B4: %d\n", x_val, y_val,
B1_val, B2_val, B3_val, B4_val);
  sprintf(str, "X_val: %5d, Y_val: %5d, X_avg: %5d, Y_avg: %5d, x_delta_high_mid: %5d,
x_delta_low_mid: %5d, y_delta_high_mid: %5d, y_delta_low_mid: %5d\n", x_val, y_val, avg_x,
avg_y, x_delta_high_mid, x_delta_low_mid, y_delta_high_mid, y_delta_low_mid);

  Serial.print(str);

  delay(20);
}
```