Automated Paint Mixing Machine Using PID Control

Submitted by

Youssef Khaled Mahran

Ibrahim Emadeldin Elsayed

Lina Tamer Ghonim

Supervised by

Prof. Ayman A. El-Badawy

A Project Report

Submitted to Mechatronics Department

In Fulfillment of Mechatronics Engineering Course (MCTR601)

The German University in Cairo

June 2023

Table of Contents

Table of Figures

I. Abstract

This report presents the design and implementation of a paint mixing machine based on the CMYK color model. The device utilizes a color sensor to measure color based on the RGB color model, which is then transformed into the CMYK color model. The system utilizes individual pumps for each pigment, starting with Key, and calculates the volume occupied by each pigment based on the desired color. The pump speed is controlled using Pulse Width Modulation (PWM) and is adjusted based on the difference between the current and desired heights of the paint. The design process involves Solidworks modeling, component selection, obtaining the transfer function, prototyping, digital control, and implementing FreeRTOS tasks. The report also covers component testing, prototype implementation, tuning of PID controllers, and final model development. Various paint materials are tested, including CMYK inks, white lacquer, white plastic matte, and plastic matte paint. Task periods are calculated, and a Windows application is developed for machine communication. The report concludes with finalizing the model, including printing team logo stickers, hardware installation, creating CMYK pigments, and conducting comprehensive testing and tuning of the system.

II. Introduction

In today's world, the demand for customized and precise color mixing has significantly increased across various industries such as printing, automotive, and interior design. Achieving accurate and consistent colors is crucial for ensuring customer satisfaction and maintaining product quality. Manual color mixing processes can be time-consuming, prone to errors, and lack the desired level of precision. To address these challenges, automated paint mixing machines have emerged as efficient solutions.

This report presents the design and implementation of a paint mixing machine that utilizes the CMYK color model. The CMYK model, which stands for Cyan, Magenta, Yellow, and Key (Black), is widely used in color printing and reproduction. By accurately measuring and mixing these primary colors, a wide range of custom colors can be achieved to meet specific requirements. The key concept behind the paint mixing machine is to convert the user's desired color input into the corresponding CMYK values, calculating the volume needed for each pigment and then pumping the correct amount. This is achieved through digital PID control where each pigment pump has a PID Controller that controls its speed by controlling it Pulse Width Modulation (PWM).

The methodology employed for designing and implementing the paint mixing machine follows a step-by-step approach. It includes the initial selection of the project idea, outlining the implementation plan, identifying the required components, deriving the transfer function, developing a Solidworks preliminary design, testing individual components, prototyping, writing FreeRTOS tasks, tuning PID controllers, testing different paint materials, and finalizing the model.

The outcomes of this project aim to provide an automated and efficient paint mixing solution that can be utilized in various industries. The report presents the detailed steps taken to design, develop, and test the paint mixing machine, with a focus on achieving accurate color reproduction and ease of use.

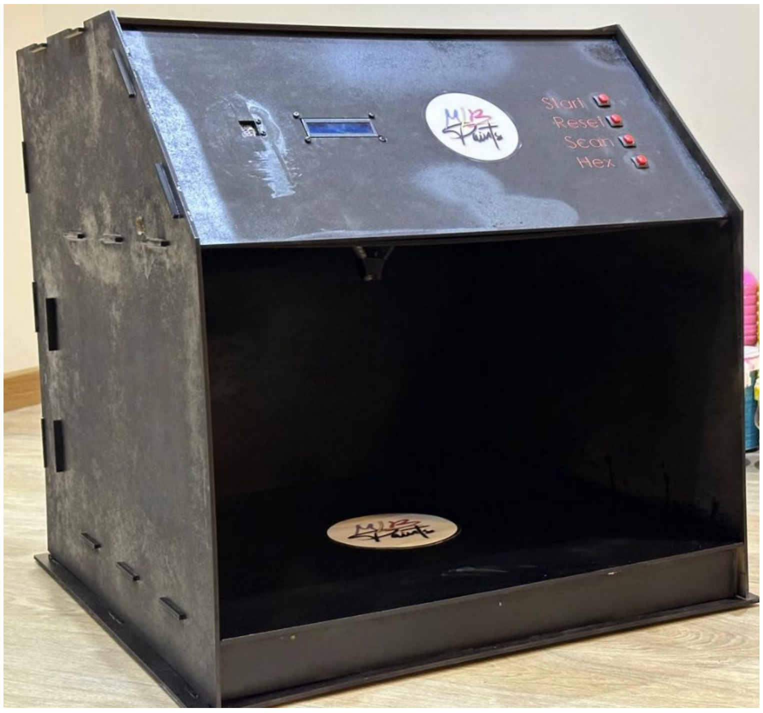


Figure 1: Paint Mixing Machine

III. System Details

1. Device Concept:

The device is based on the CMYK color model. The user inputs the color desired through the TCS34725 color sensor that measures color based on an RGB color model (red, green, blue). The accuracy of color measurements of TCS3472 color sensor is much improved due the IR block filter which blocks the IR spectrum from reaching the sensor and only passes the visible spectrum. Since the human eyes cannot see the IR spectrum, this makes the color measurements very close to the true color detected by human eye. The RGB color model is then transformed to CMYK model through the following algorithm:

1. The R, G, B values are divided by 255 to change the range from 0-255 to 0-1.

2. The black key (K) color is calculated from the red (R'), green (G') and blue (B') colors:

$K = 1 - \max(R', G', B')$

3. The cyan color (C) is calculated from the red (R') and black (K) colors:

$C = (1-R'-K) / (1-K)$

4. The magenta color (M) is calculated from the green (G') and black (K) colors:

$M = (1-G'-K) / (1-K)$

5. The yellow color (Y) is calculated from the blue (B') and black (K) colors:

$Y = (1-B'-K) / (1-K)$

Each pigment value is now a percentage of 100% of each pigment. Each pigment has 5 cm maximum height to be added if and only if there is 100% of this pigment. Each pigment occupies a percentage of its 5 cm allowance depending on their values.

A draft for the control loop that will be controlling each pump speed:

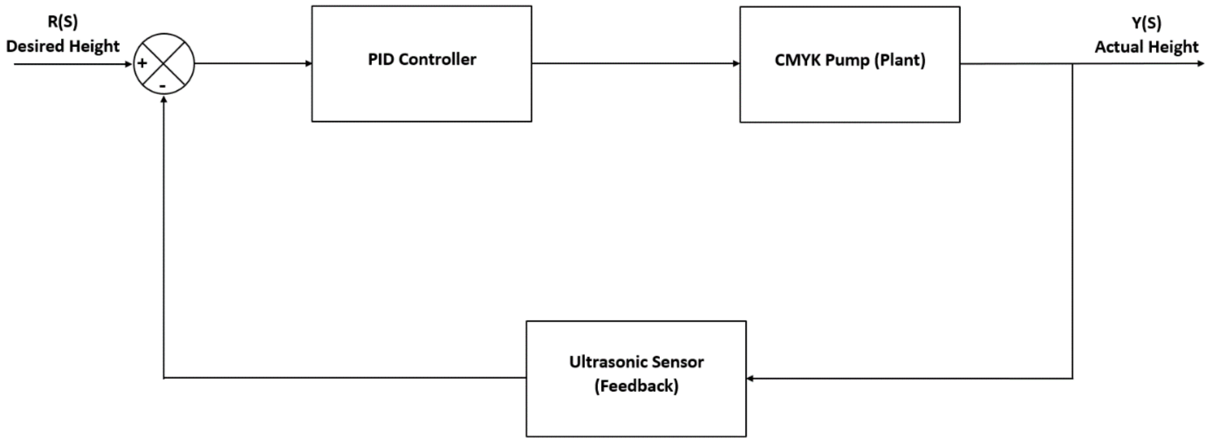


Figure 2: Control Loop Draft

Each pigment will be pumped individually one after one starting with the Key. An ultrasonic sensor will be fitted in the machine to read the current paint height in the container. The Key percentage calculated will be applied to the allowed height of the key to calculate the desired height.

The Key pump is then activated at a speed calculated from the difference between the ultrasonic reading of the current height and the desired height. The closer the paint reaches the desired calculated height the slower the pump speed is until it is turned off when the liquid reaches the desired height. The pump speed will be controlled using the Pulse Width Modulation (PWM). The same method is repeated for each of the remaining pigments until the container is full.

2. Solidworks Preliminary Design:

A draft for the device is implemented on Solidworks giving the following results:

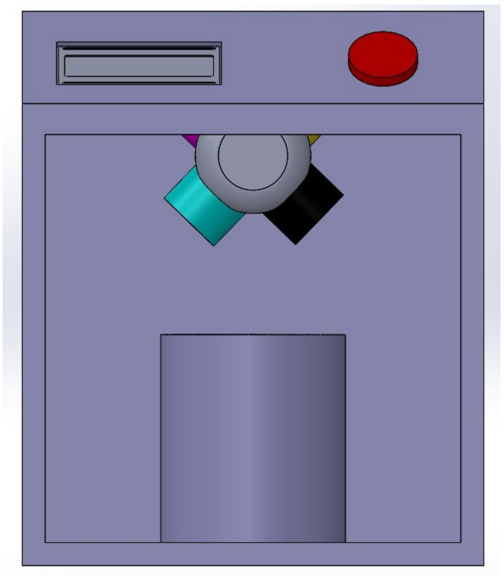


Figure 3: Model Draft Front View

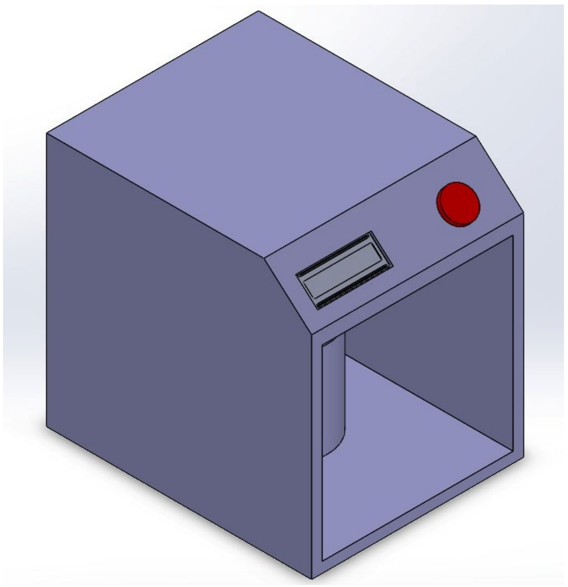


Figure 4: Model Draft Isometric View

IV. Methodology

The following methodology outlines a systematic approach for designing and implementing our paint mixing machine. The process involves a step-by-step progression, as described below:

1. **Idea Selection:** The initial phase of the project involved researching various project ideas utilizing PID Control and selecting the most suitable one for our vision.
2. **Outline for Idea Implementation:**
   - **2.1. Preliminary Design of Machine:** Draft design for the machine was created to outline the overall functionality and looks of the project determining its key features.
   - **2.2. Identifying Components**: Identifying the needed components for the project to work and for the idea to be implemented effectively.
3. **Obtaining Transfer Function:**
   - **3.1. Deriving Transfer Function:** Deriving the transfer function of the system that controls the paint pumps.
   - **3.2. Simulink Model:** Modeling the system on Simulink to simulate its behavior.
   - **3.3. PID control gains:** Identifying the PID gains needed theoretically and obtaining Bode Plots of the system.
   - **3.4. Root Locus and Bode Plot:** Obtaining the Root Locus and Bode Plot of the system.
   - **3.5. Discretizing the Model:** discretizing the transfer function to obtain a discrete model and do discrete control over it.
4. **Solidworks Model of Prototype:** Doing a 3D model for a functioning machine.

5. **Component Testing:** Testing each component individually and identifying how they work including:
    - **5.1. Arduino Uno**
    - **5.2. Color Sensor**
    - **5.3. Ultrasonic Sensor**
    - **5.4. LCD I2C Interfacing**
    - **5.5. LCD and Color Sensor I2C Interfacing**
    - **5.6. Water Pump**
6. **Prototype Implementation:**
    - **6.1. Laser Cutting:** Cutting the model and implementing it in real life.
    - **6.2. Spray Painting:** Painting the MDF.
    - **6.3. Model Assembly:** Assembling the individual components according to the finalized design.
7. **Writing Individual FreeRTOS Tasks and Testing Them:**
    - **7.1. Color Sensor Task:** Writing task for reading color sensor and verifying it reads the right color.
    - **7.2. Ultrasonic Sensor Task:** Writing task for reading ultrasonic sensor and making sure it reads the right distance.
    - **7.3. PID Control of Each Pump:** Writing a task that reads the ultrasonic sensor and sets a new PWM.
    - **7.4. Task Priorities:** Defining the priority order of tasks.
8. **Tuning Each PID Controller Individually:** Testing each motor individually and tuning its gains.

9. **Prototype Testing:** Testing the full model with all FreeRTOS Tasks.

10. **Solidworks of Final Model:**

    10.1. **3D Model:** Modeling the machine on Solidworks.

    10.2. **Model Dimensions:** Identifying final dimensions of the model.

11. **Final Model Implementation:**

    11.1. **Laser Cutting:** Cutting the model and implementing it in real life.

    11.2. **Model Assembly:** Assembling the model.

    11.3. **Spray Painting:** Painting the MDF.

12. **Testing Different Paint Materials:**

    12.1. **Printer Ink:** Testing CMYK Inks as pigments.

    12.2. **White Lacquer:** Testing White Lacquer as base material.

    12.3. **White Plastic Matte:** Testing White Plastic Matte as base material.

    12.4. **Plastic Matte Paint:** Testing CMYK colors made from Plastic Matte Paint.

13. **Calculating Task Periods:** Calculate each task's period.

14. **Testing the Code:** Testing the code functionality and performance.

15. **Writing Aperiodic Tasks:** Rewriting the tasks to run non periodically to better utilize the processor.

16. **Developing Application:** Developing a windows app to communicate with the machine.

17. Finalizing Model:

17.1. Printing Team Logo Stickers: Printing and adding the team logo to the model.

17.2. Screwing Hardware in Place: Screwing the motors and H-Bridges in place.

17.3. Creating CMYK Pigments: Creating pigments out of paints.

18. Final Testing: Doing final tests and tunings to the system.

V. Results

1. Idea Selection:

Various project ideas were researched including but not limited to: Self-balancing robot, ball plate balance, smart irrigation system, GPS navigation robot, and hand mimicking robot. However, after consulting the TAs the paint mixing machine project was chosen to be implemented.

2. Outline for Idea Implementation:

As the idea is still fresh, it was drawn out on paper and a preliminary design was done. The needed components were identified and listed in order to start working on the project.

2.1. Preliminary Design of Machine:

Paint mixing machine project was heavily researched to identify how it will be designed, how it will work, what will the control loop look like, how paint mixing machines work and identifying the device concept.

2.2. Identifying Components:

After knowing how the project will function various components were selected to implement the project:

a. **MDF Wood:** It was chosen to implement the body of the machine due to its low cost, easy cutting and availability.

b. **Ultrasonic sensor:** It was chosen to read the current height of the liquid in the container.

c. **LCD Display:** It was chosen to communicate with the user.

d. **Water Pump:** It was chosen to pump the paint from the tanks to the container.

e. **TCS3472 Color Sensor:** It was chosen due to its accuracy of reading the colors and its availability in the Egyptian market.

3. Obtaining Transfer Function:

Since the project is based on PID control, the system was modeled mathematically and a transfer function for the system was obtained along with the root locus and bode plots.

3.1. Deriving Transfer Function:

After researching and looking at different research papers, the transfer function was derived as the following:

3.1.1. System Model:

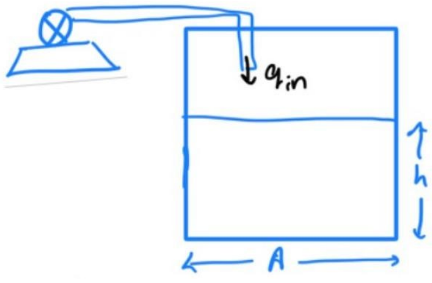


Figure 5: System Model

- $Q_{in} = A \frac{dh}{dt}$

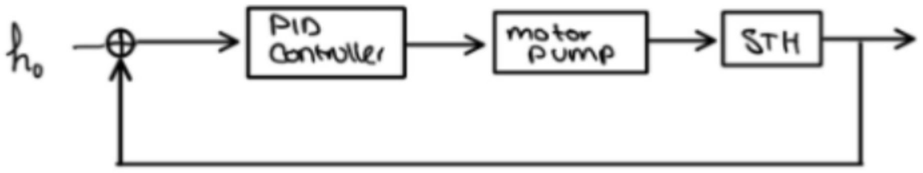


Figure 6: Block Diagram

- Speed of motor (w(t)) [rad/s] is directly related to water flow rate (Q_{in}) [m^2/s] supplying the tank.
- STH is a block that relates speed of flow rate to the height (h).
- Assume a linear relationship where: $w(t) = K_f * Q_{in}(t)$

3.1.2. Modeling the Motor Pump and the PID Controller:

- Input to motor is a voltage source applied by PWM from the H-bridge to its terminals.
- Output is rotational speed of the shaft $w(t) = \frac{d\theta}{dt}$
- Assume motor and shaft are rigid.
- Assume a viscous friction model, friction torque is proportional to shaft angular velocity.

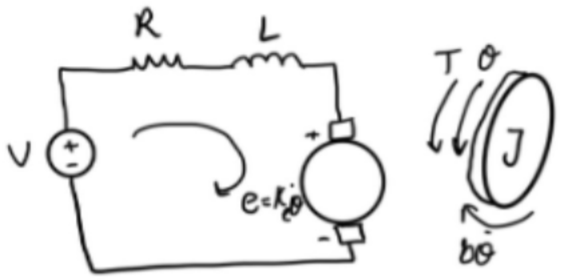


Figure 7: Motor Model

- **Physical parameters of motor:**
    a. Moment of Inertia (J) [kg.m^2]
    b. Motor Viscous Friction Constant (b) [N.m.s]
    c. Electromotive Force Constant (K_e) [N.m/Amp]
    d. Electric Resistance (R) [Ω]
    e. Electric Inductance (L) [H]
- Assuming Armature Controlled Motor $T = K_t * i$
- Back emf is proportional to angular velocity of shaft.
- $e = K_e * w$
- $(J * \ddot{\theta}) + (b * \dot{\theta}) = T = K_t * i$ (1)
- $L \frac{di}{dt} + R_i = e = V - K_e * \dot{\theta}$ (2)
- $\dot{\theta} = \frac{d\theta}{dt} = w$
- Taking Laplace of (1) and (2)

$$s^2 * J * \theta(s) + s * b * \theta(s) = K_t I(s) \quad (3)$$

$$s * L * I(s) + R * I(s) = V(s) - s * K_e \theta(s) \quad (4)$$

- From (3)

$$I(s) = \frac{s^2 * J * \theta(s) + s * b * \theta(s)}{K_t}$$

- Sub in (4)

$$s * L * \frac{s^2 * J * \theta(s) + s * b * \theta(s)}{K_t} + R * \frac{s^2 * J * \theta(s) + s * b * \theta(s)}{K_t} = V(s) - s * K_e * \theta(s)$$

- Replace [s * θ(s)] with [w(s)]

$$s * L * \frac{s^2 * J * \theta(s) + b * w(s)}{K_t} + R * \frac{s * J * w(s) + b * w(s)}{K_t} = V(s) - K_e * w(s)$$

- $w(s) \frac{s^2 * L * J + s * L * b + R * s * J + R * b + K_e * K_t}{K_t} = V(s)$

- $P_m(s) = \frac{w(s)}{V(s)} = \frac{K_t}{s^2 * L * J + s * L * b + R * s * J + R * b + K_e * K_t}$

3.1.3. PID Controller:

- $v(t) = K_p * e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$

- $C(s) = \frac{V(s)}{e(s)} = K_p + \frac{K_i}{s} + K_d \frac{de(t)}{dt}$

3.1.4. STH Block:

- $g(t) = \frac{h(t)}{w(t)}$ (1)
- $w(t) = K_f * Q_{in}(t)$
- $Q_{in}(t) = \frac{w(t)}{K_f}$ (2)
- (2) in (1)

  $w(t) = K_f * A \frac{dh}{dt}$
- Take Laplace

  $W(s) = s * K_f * A * h(s)$
- $G(s) = \frac{h(s)}{w(s)} = \frac{1}{s * A * K_f}$

3.1.5. Overall Transfer Function:

- **Variables to assume:**
  a. $J = 0.01$ kg.m^2
  b. $b = 0.1$ N.m.s
  c. $K_t = 0.1$ N.m/A
  d. $K_e = 0.01$ V.s/rad
  e. $R = 1\ \Omega$
  f. $R_f = 0.5$ s/m^2
  g. $K_f = 1$
  h. $r = 0.05$ m
- $A = \pi * (0.05)^2 = 7.854\text{x}10^{-3}$ m^2
- $F(s) = \frac{h(s)}{w(s)} = C(s) * P_m(s) * G(s)$(Open Loop)
- $G_{sys}(s) = \frac{F(s)}{1 + F(S)}$ (Closed Loop)

3.2. Simulink Model:

After obtaining the system transfer function, it was modeled on Simulink to tune the PID gains as follows:

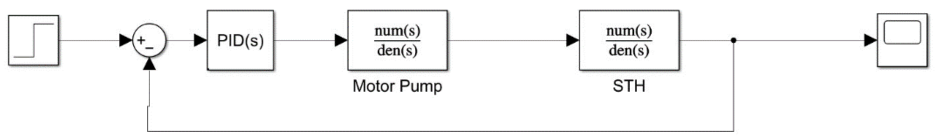


Figure 8: Simulink Closed Loop Model

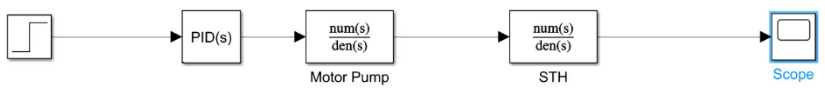


Figure 9: Simulink Open Loop Model

To confirm the linear relationship assumed in (Section 3.1.1.), the open loop model was simulated giving the following results confirming the linear relationship:

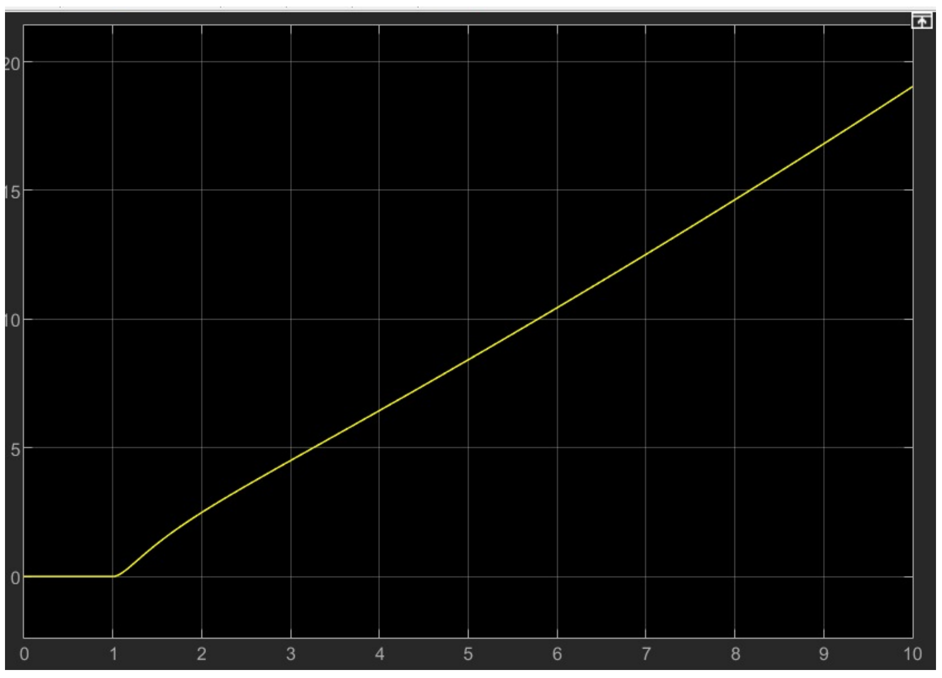
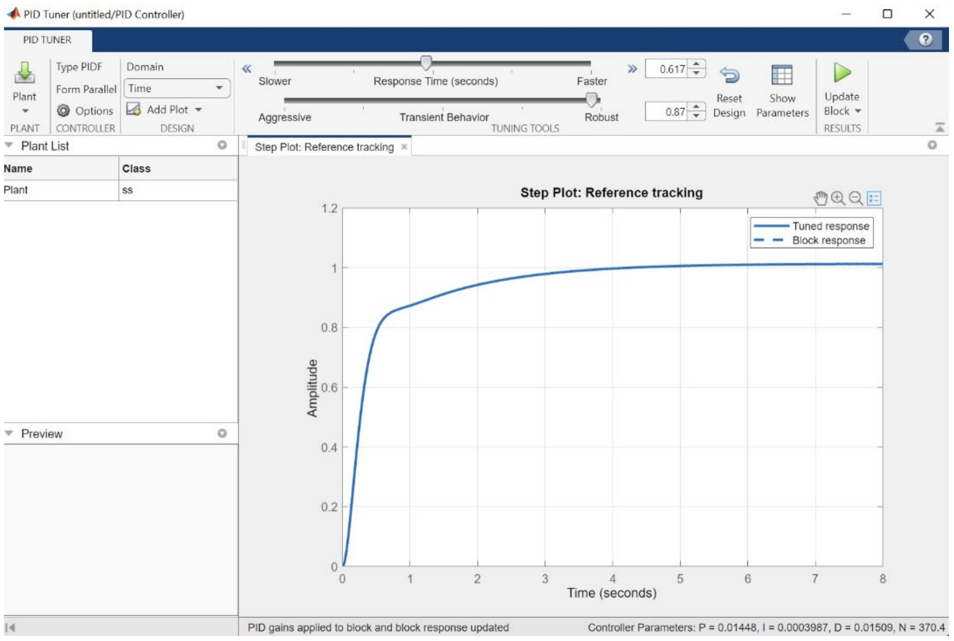


Figure 10: Open Loop Simulation

3.3. PID control gains:

After modeling the system on Simulink, built in PID Tuner was used to tune the PID gains to give the required system behavior. The output of the PID Tuner was the following:

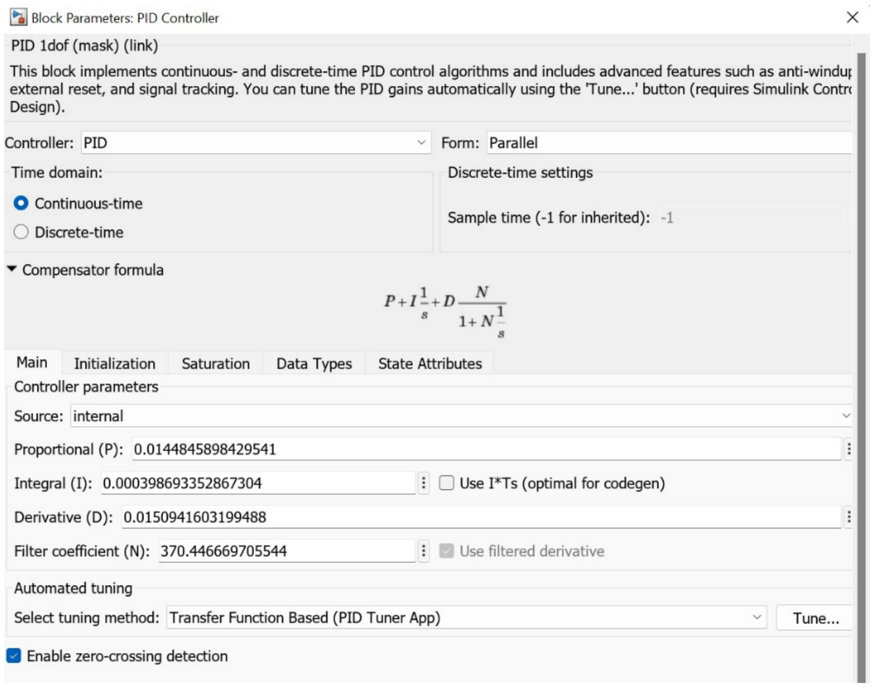


Type PIDF
Form Parallel
Domain: Time
Response Time (seconds): 0.617
Transient Behavior: 0.87

Name	Class
Plant	ss

PID gains applied to block and block response updated
Controller Parameters: P = 0.01448, I = 0.0003987, D = 0.01509, N = 370.4

Figure 11: PID Tuner Step Response



Block Parameters: PID Controller

PID 1dof (mask) (link)

This block implements continuous- and discrete-time PID control algorithms and includes advanced features such as anti-windup external reset, and signal tracking. You can tune the PID gains automatically using the 'Tune...' button (requires Simulink Contro Design).

Controller: PID
Form: Parallel

Time domain:
[✓ Continuous-time]
Discrete-time

Discrete-time settings
Sample time (-1 for inherited): -1

Compensator formula

$$P + I\frac{1}{s} + D\frac{N}{1 + N\frac{1}{s}}$$

Main | Initialization | Saturation | Data Types | State Attributes

Controller parameters
Source: internal
Proportional (P): 0.0144845898429541
Integral (I): 0.000398693352867304
Use I*Ts (optimal for codegen)
Derivative (D): 0.0150941603199488
Filter coefficient (N): 370.446669705544
Use filtered derivative

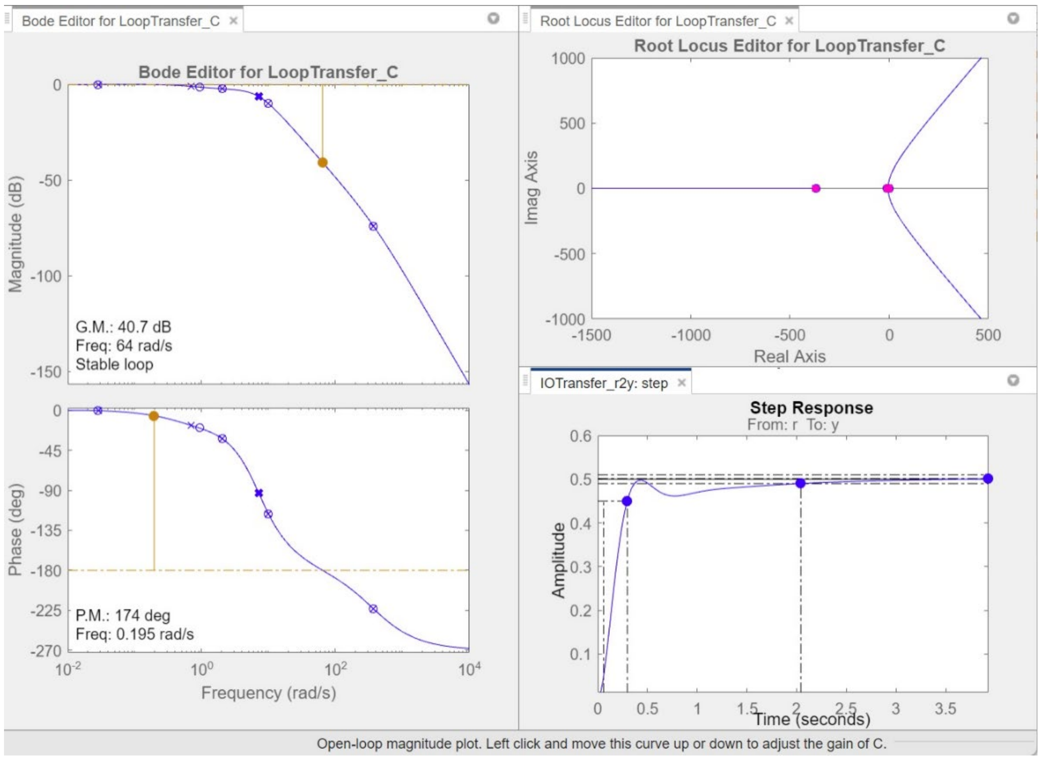
Automated tuning
Select tuning method: Transfer Function Based (PID Tuner App)
Tune...

[✓] Enable zero-crossing detection

Figure 12: PID Tuner Output Gains

3.4. Root Locus and Bode Plot:

The transfer function was transferred to MATLAB to utilize the Control System Designer toolbox to obtain the following Root Locus and Bode Plots:



G.M.: 40.7 dB
Freq: 64 rad/s
Stable loop

P.M.: 174 deg
Freq: 0.195 rad/s

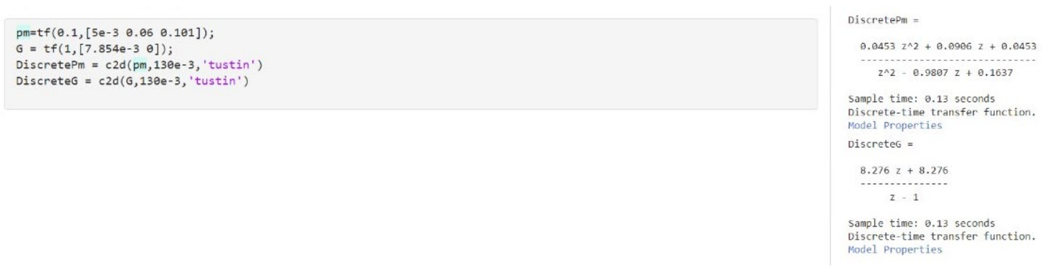
Open-loop magnitude plot. Left click and move this curve up or down to adjust the gain of C.

Figure 13: Root Locus and Bode Plot

In the Root Locus all poles are in the left half plane which confirms the stability of the system. Bode plots also agrees with the Root Locus in terms of stability.

3.5. Discretizing the Model:

The previous sections dealt with modeling and controlling system using continuous control methods. However, when dealing with microcontrollers, continuous signals are impossible for the processor to analyze. The signals have to be discretized in order for the microcontroller to be able to process it and control it. This technique is called digital control which our project utilizes. In order to discretize the system, the c2d() MATLAB command was used and the system was modeled on Simulink giving the following results:



```
pm=tf(0.1,[5e-3 0.06 0.101]);
G = tf(1,[7.854e-3 0]);
DiscretePm = c2d(pm,130e-3,'tustin')
DiscreteG = c2d(G,130e-3,'tustin')
```

```
DiscretePm =

  0.0453 z^2 + 0.0906 z + 0.0453
  ------------------------------
     z^2 - 0.9807 z + 0.1637

Sample time: 0.13 seconds
Discrete-time transfer function.
Model Properties
DiscreteG =

  8.276 z + 8.276
  ---------------
       z - 1

Sample time: 0.13 seconds
Discrete-time transfer function.
Model Properties
```

Figure 14: c2d MATLAB Command

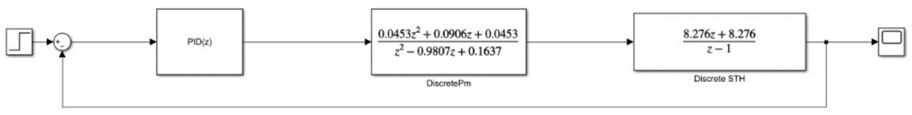

PID(z)
$\frac{0.0453z^2+0.0906z+0.0453}{z^2-0.9807z+0.1637}$
DiscretePm
$\frac{8.276z+8.276}{z-1}$
Discrete STH

Figure 15: Discretized Simulink Model

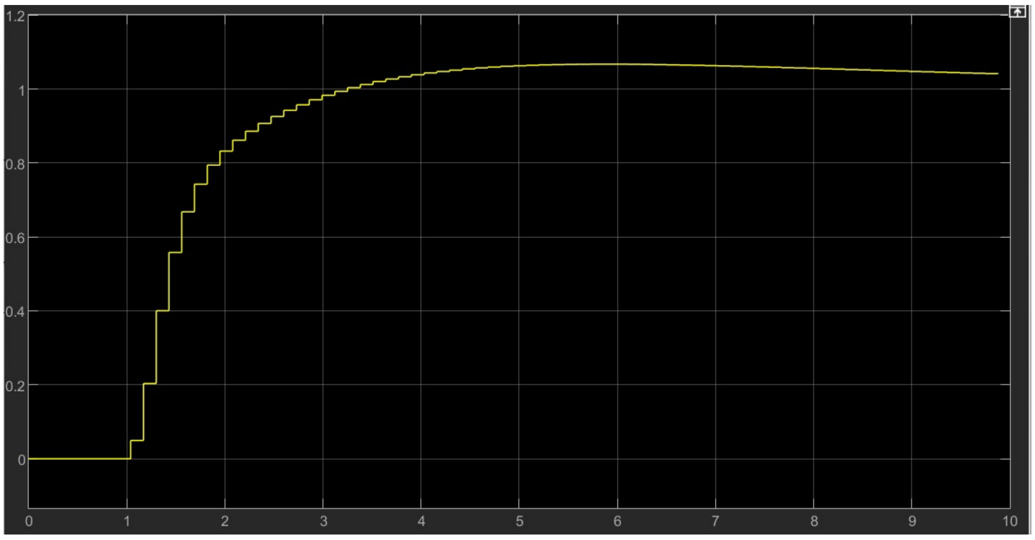


Figure 16: Discretized Step Response

The discretized step response matches the continuous step response obtained in (Section 3.3.). This shows that the system is can be discretized and digitally controlled using Arduino.

4. Solidworks Model of Prototype:

After modeling the system, a 3D model of the actual machine was implemented. The machine was designed to be made from MDF wood and laser cut to the desired shape. Knowing that it will be laser cut, the model was designed to be assembled like a puzzle with each plate containing grooves and parts to be easily assembled together. The 3D model is as follows:

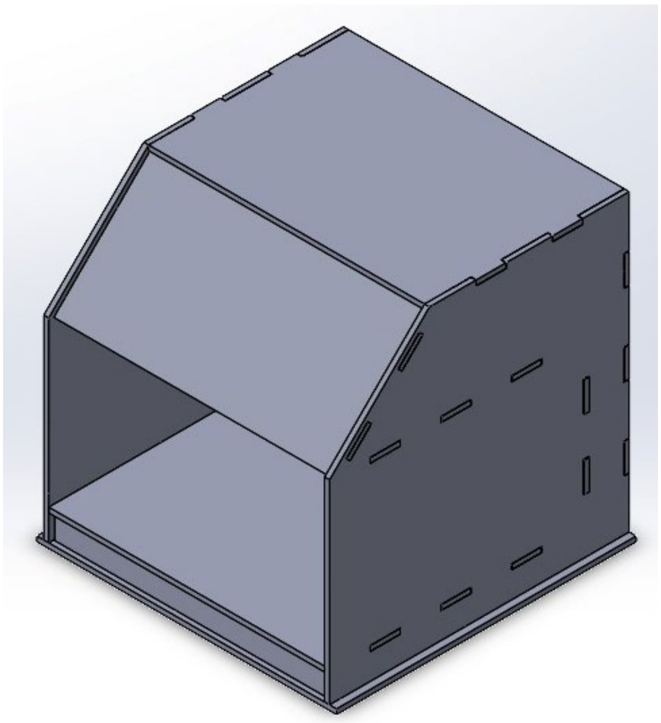


Figure 17: Prototype Solidworks Model

5. Component Testing:

Each component was tested individually to identify how it works and what is the needed code for it to operate properly.

5.1. Arduino Uno:

Initially, the project was to be done using Arduino Uno (Figure 18) due to its wide availability in the Egyptian Market and its cheap price. However, after reviewing the components it appeared that the 13 digital pins of the Arduino Uno are not enough as a minimum of 32 digital pin is needed for the project to function. So, the Arduino Mega 2560 (Figure 19) was chosen to be the brains of the project.

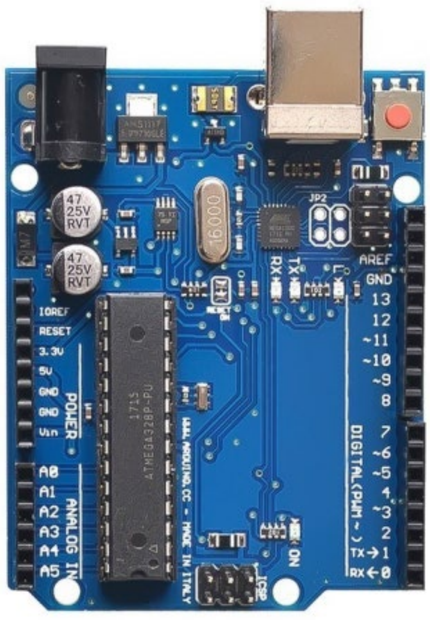


Figure 18: Arduino UNO

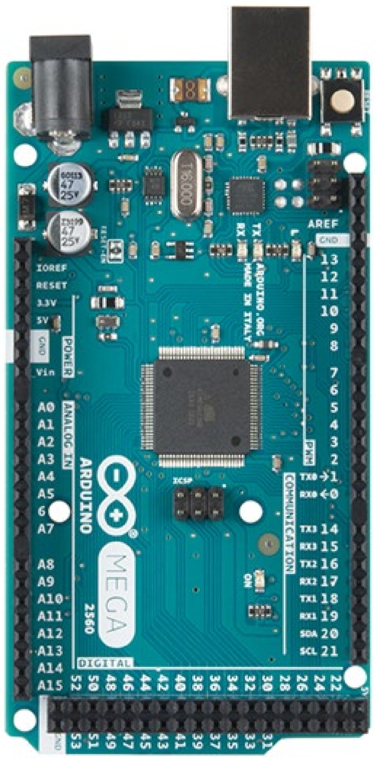


Figure 19: Arduino MEGA 2560

5.2. Color Sensor:

The TCS34725 color sensor (Figure 20) contains 7 pins however after testing only 4 of those are needed; the 3.3V, GND, SDA and SCL pins. It was then connected as the following schematic:

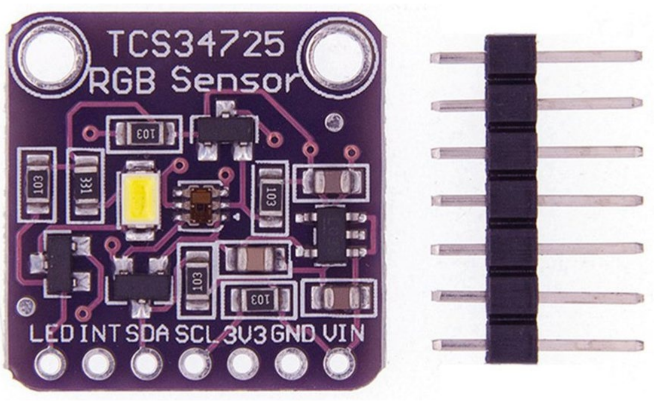


Figure 20: TCS34725 Color Sensor

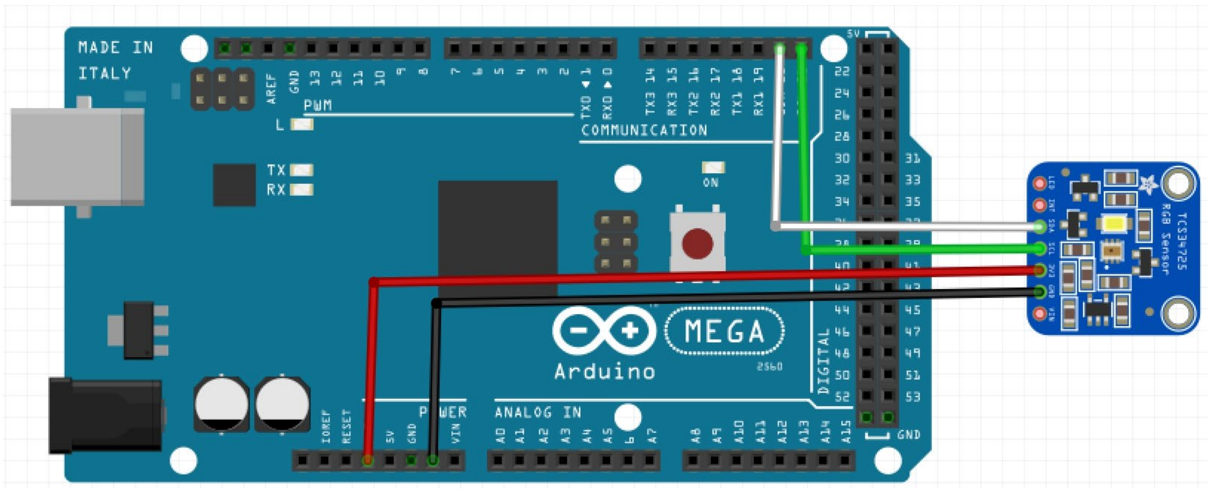


Figure 21: TCS34725 Color Sensor Connection

The idea of how it works is that it sends the obtained data through I2C connection to the Arduino board. There is a library “Adafruit_TCS34725.h” that initializes the I2C connection and receives the data through predefined functions. The following code was used to obtain different RGB values:

```
#include <Wire.h>
#include "Adafruit_TCS34725.h"
/* Initialize with specific int time and gain values */
Adafruit_TCS34725 tcs = Adafruit_TCS34725(TCS34725_INTEGRATIONTIME_614MS,
TCS34725_GAIN_1X);

void setup(void) {
  Serial.begin(9600); //Begin Serial Communication
  tcs.begin();  //Initialize Color Sensor
}

void loop(void) {
  uint16_t r, g, b; //Variables to store values
  tcs.getRGB(&r, &g, &b, &c); //Get Sensor Readings

  /*Print the values to the serial monitor*/
  Serial.print("R: "); Serial.print(r, DEC); Serial.print(" ");
  Serial.print("G: "); Serial.print(g, DEC); Serial.print(" ");
  Serial.print("B: "); Serial.print(b, DEC); Serial.print(" ");
  Serial.println(" ");
}
```

Figure 22: TCS34725 Color Sensor Tester Code

The obtained colors were then compared to the actual color to see if it matches the scanned color. A limitation has risen here as the color sensor cannot scan Black and White colors it only scans the color within the Black-White range but not the extremes. Another limitation was that the color sensor doesn't read the exact value of each RGB color but reads values within an offset of these colors. To avoid this limitation, a code was written so that the color sensor returns the values only if the offset between the current reading and the previous reading is smaller than or equal to one 2 consecutive times. The following code was used to the effectiveness of it:

```
void readColors(){
  int consuctiveReads = 0;  //Consecutive reads = 0
  int prevR = 0;
    while(consuctiveReads < 2){
      colorSensor.getRGB(&r, &g, &b); //Read RGB values
      //if the offset between previous red and curr red is 1 or 0
      if(abs((int)prevR - (int)r) <=1)
        consuctiveReads++;  //Increase the counter
      else
        consuctiveReads = 0;  //Otherwise clear it
      prevR = r;
    }
    consuctiveReads = 0;
    int prevG = 0;
    while(consuctiveReads < 2){
      colorSensor.getRGB(&r, &g, &b); //Read RGB values
      //if the offset between previous red and curr red is 1 or 0
      if(abs((int)prevG - (int)g) <=1)
        consuctiveReads++;  //Increase the counter
      else
        consuctiveReads = 0;  //Otherwise clear it
      prevGreen = g;
    }
    consuctiveReads = 0;
    int prevB = 0;
    while(consuctiveReads < 2){
      colorSensor.getRGB(&r, &g, &b); //Read RGB values
      //if the offset between previous red and curr red is 1 or 0
      if(abs((int)prevB - (int)b) <=1)
        consuctiveReads++;  //Increase the counter
      else
        consuctiveReads = 0;  //Otherwise clear it
      prevBlue = b;
    }
}

void loop(void) {
  readColors();

  /*Print the values to the serial monitor*/
  Serial.print("R: "); Serial.print(r, DEC); Serial.print(" ");
  Serial.print("G: "); Serial.print(g, DEC); Serial.print(" ");
  Serial.print("B: "); Serial.print(b, DEC); Serial.print(" ");
  Serial.println(" ");
}
//setup and initialize in figure 10
```

Figure 23: Color Sensor Second Tester Code

Upon using this tester code more accurate reading were returned from the color sensor and the color sensor readings were stable enough.

5.3. Ultrasonic Sensor:

The HC-SR04 Ultrasonic sensor contains 4 pins; VCC, GND, TRIG, ECHO. They were connected as the following schematic:

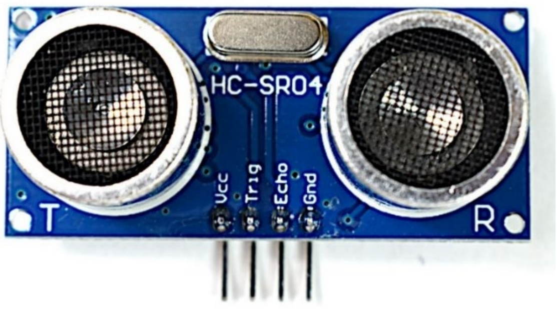


Figure 24: HC-SR04 Ultrasonic Sensor

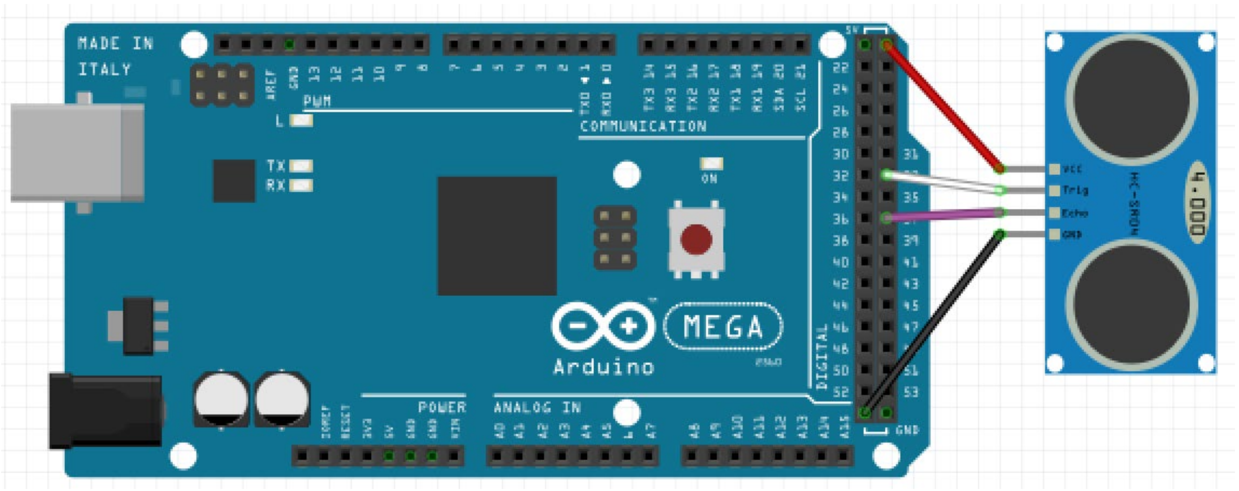


Figure 25: HC-SR04 Ultrasonic Sensor Connection

The idea of the ultrasonic sensor is that it sends an ultrasonic pulse for 10 microseconds (us) by setting the TRIG pin HIGH for 10us then setting it LOW again. Then the sent wave hits an obstacle and bounces back and the sensor receives the signal. It then sets the ECHO pin HIGH for the amount of time the signal took to bounce back. Then by multiplying the duration the ECHO pin was high for by the speed of sound we can calculate the distance between the ultrasonic sensor and the obstacle. The following code was used to test the ultrasonic sensor reading:

```
#define TRIGGER_PIN 33   // Trigger pin of ultrasonic sensor
#define ECHO_PIN 37      // Echo pin of ultrasonic sensor

void setup() {
  Serial.begin(9600); //Begin Serial Communication
  pinMode(TRIGGER_PIN, OUTPUT); //Set Trigger pin as output
  pinMode(ECHO_PIN, INPUT); //Set Echo pin as input
}

void loop() {
  /*Send a trigger pulse to the ultrasonic sensor*/
  digitalWrite(TRIGGER_PIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIGGER_PIN, LOW);
  /*Send a trigger pulse to the ultrasonic sensor*/


  unsigned long duration = pulseIn(ECHO_PIN, HIGH);//Measure the duration of
the echo pulse
  float distance = duration * 0.034 / 2.0;// Calculate the distance based on
the speed of sound

  /*Print the distance on the serial monitor*/
  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");
  /*Print the distance on the serial monitor*/

  delay(1000);  //Delay for 1 second
}
```

Figure 26: HC-SR04 Ultrasonic Sensor Tester Code

A small limitation was discovered as the ultrasonic sensor reading are not stable and it reads within a range of values. This limitation was solved by using the “NewPing.h” library where it has a predefined functions that does 5 consecutive readings and returns the median value of them. The code that utilizes this library and tests its effectiveness is as follows:

```
#include <NewPing.h>
#define TRIGGER_PIN 33   //Trigger pin of ultrasonic sensor
#define ECHO_PIN 39      //Echo pin of ultrasonic sensor

NewPing sensor(trigPin, echoPin, 200);  //Create a NewPing object

void setup() {
  //Initialize serial communication
  Serial.begin(9600);
}

void loop() {
  float duration = sensor.ping_median();  //Send 5 pings and get the median duration
  float distance = duration * 0.034/2;  //Calculate distance

  /*Print the distance to the serial monitor*/
  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");
  /*Print the distance to the serial monitor*/

  delay(1000);  //Delay before next measurement
}
```

Figure 27: NewPing Test Code

The NewPing library gave more accurate and stable readings of the ultrasonic sensor.

5.4. LCD I2C Interfacing:

The 16x2 LCD display (Figure 28) is an easy and effective way to communicate with the user and display messages to him. However, one downside is that it takes 16 pins to communicate with the Arduino. Luckily, there is an I2C module (Figure 29) that takes the message needed to be displayed from the Arduino through I2C connection and displays it on the LCD. This reduced the needed pins from 16 pins to only 4 pins as shown in the following schematic:

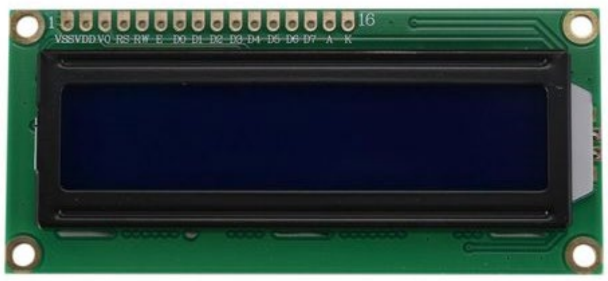


Figure 28: 16x2 LCD Display

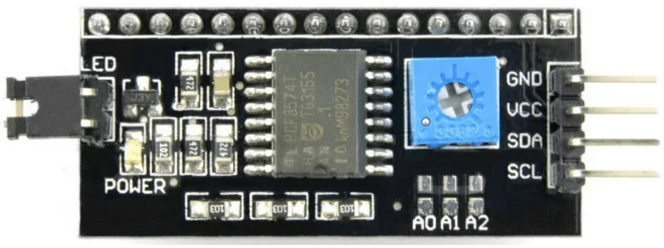


Figure 29: LCD I2C Module

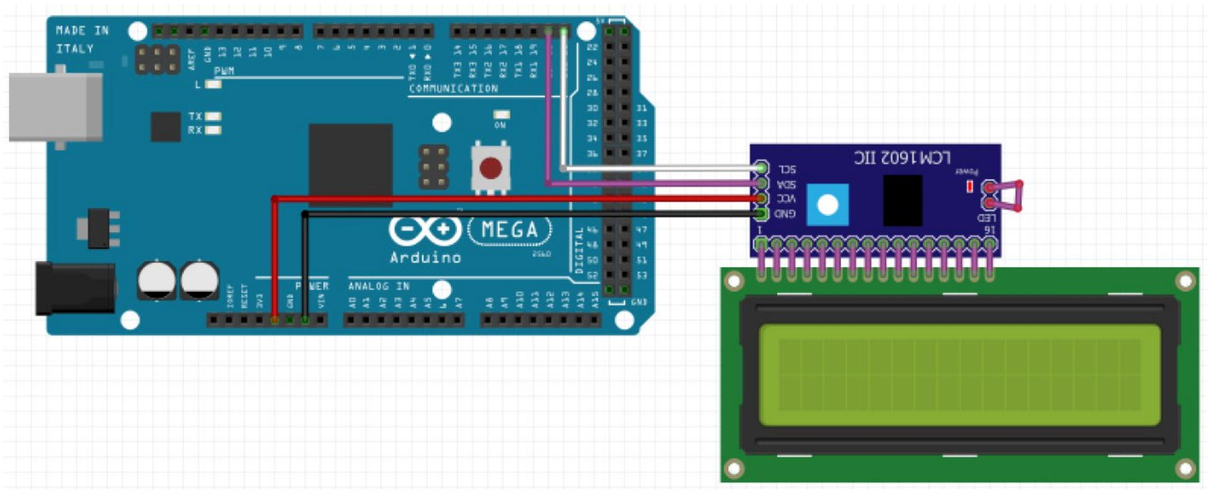


Figure 30: LCD I2C Module Connection

Before testing the LCD display, its I2C address must be known to ensure effective connection as there are 2 default I2C addresses for the LCD (0x27 and 0x32). This wasn't a problem with the Color Sensor as it has only 1 default I2C address. The following scanner code was used to know which I2C address the LCD uses:

```
#include <Wire.h>
#define WIRE Wire // Set I2C bus to use: Wire, Wire1, etc.
void setup() {
  WIRE.begin();
  Serial.begin(9600);
  while (!Serial)
     delay(10);
  Serial.println("\nI2C Scanner");
}
void loop() {
  byte error, address;
  int nDevices;
  Serial.println("Scanning...");
  nDevices = 0;
  for(address = 1; address < 127; address++ )
  {
    WIRE.beginTransmission(address);
    error = WIRE.endTransmission();
    if (error == 0)
    {
      Serial.print("I2C device found at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.print(address,HEX);
      Serial.println("  !");
      nDevices++;
    }
    else if (error==4)
    {
      Serial.print("Unknown error at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.println(address,HEX);
    }
  }
  if (nDevices == 0)
    Serial.println("No I2C devices found\n");
  else
    Serial.println("done\n");
}
```

Figure 31: I2C Scanner Code

Upon knowing the LCD I2C address (0x27), the following test code was used to test the LCD:

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

/*Initialize the LCD address to 0x27 for a 16 chars and 2 line display*/
LiquidCrystal_I2C lcd(0x27,16,2);

void setup()
{
  Serial.begin(9600); //Begin Serial connection
  lcd.init(); //Initialize the lcd
  lcd.backlight();  //Turn on the Backlight
  lcd.setCursor(3,0); //Set the cursor to (3,0)
  lcd.print("Hello, world!"); //Print Hello World
}

void loop()
{
}
```

Figure 32: LCD I2C Module Tester Code

5.5. LCD and Color Sensor I2C Interfacing:

Now the project contains 2 I2C devices utilizing the I2C bus. Testing whether the 2 devices will be able to utilize the I2C bus simultaneously is crucial for the project.

The Color Sensor and the LCD were connected to the Arduino as the follows:

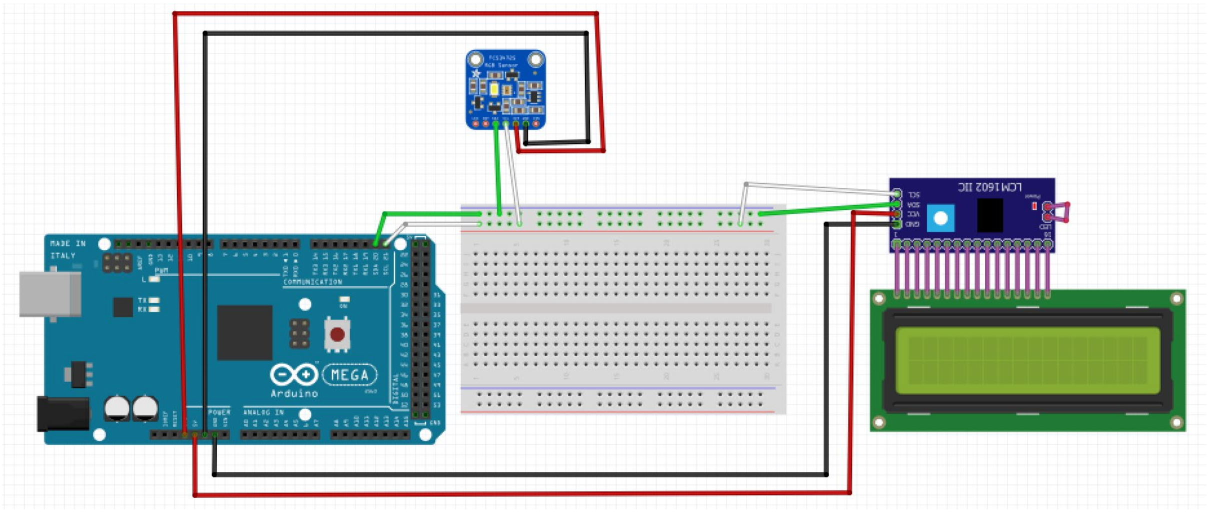


Figure 33: LCD I2C Module and Color Sensor Connection

A test code was then written where the Color Sensor would obtain a color and then display its HEX code on the LCD as follows:

```
#include <Wire.h>
#include "Adafruit_TCS34725.h"
#include <LiquidCrystal_I2C.h>

/*Initialize the LCD address to 0x27 for a 16 chars and 2 line display*/
LiquidCrystal_I2C lcd(0x27,16,2);

/* Initialize with specific int time and gain values */
Adafruit_TCS34725 colorSensor =
Adafruit_TCS34725(TCS34725_INTEGRATIONTIME_614MS, TCS34725_GAIN_1X);
int r, g, b;

void readColors(){
  //Same as figure 11
}
void printColors(){
  /*display color hex code on LCD*/
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Color is: ");
  lcd.setCursor(0,1);
  lcd.print("#");
  lcd.print((int)r, HEX);
  lcd.print((int)g, HEX);
  lcd.print((int)b, HEX);
  /*display color hex code on LCD*/
}

void setup(void) {
  Serial.begin(9600); //Begin Serial Communication
  colorSensor.begin();  //Initialize Color Sensor
  lcd.init(); //Intialize LCD
  lcd.backlight();  //Turn on the backlight
}

void loop(void) {
  readColors();
  printColors();
}
```

Figure 34: LCD I2C Module and Color Sensor Tester Code

The above code worked as expected and gave the following results:

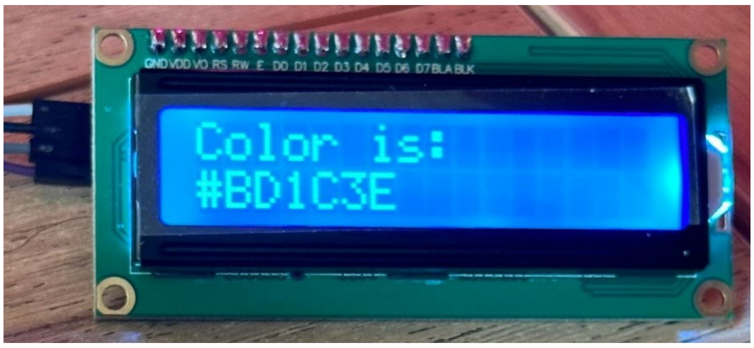


Figure 35: LCD I2C Module and Color Sensor Tester Code Results

5.6. Water Pump:

Water pumps are the corner stone of this project so different water pumps were tested to choose the most effective pump for this project. At first, a mini water pump (Figure 36) was tested using different PWM values to find its threshold. The following code was used to test it:

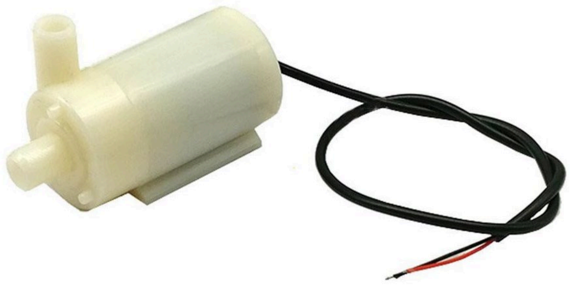


Figure 36: Mini Water Pump 2.5V-6V

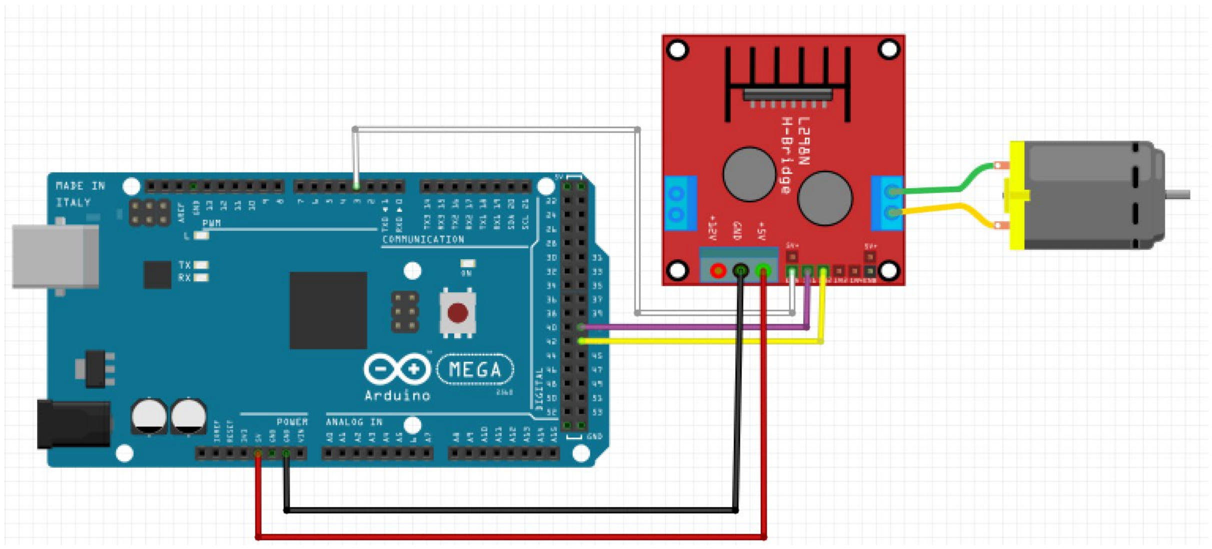


Figure 37: Mini Water Pump Connections

```
// Motor control pins
const int motorPin = 3;
const int high = 41;
const int low = 43;

void setup() {
  pinMode(motorPin, OUTPUT);
  pinMode(high, OUTPUT);
  pinMode(low, OUTPUT);
  digitalWrite(high, HIGH);
  digitalWrite(low, LOW);
}

void loop() {
  // Increase motor speed gradually from 0 to 255
  for (int speed = 0; speed <= 255; speed++) {
    analogWrite(motorPin, speed);
    delay(10); // Delay for a smoother transition
  }

  // Decrease motor speed gradually from 255 to 0
  for (int speed = 255; speed >= 0; speed--) {
    analogWrite(motorPin, speed);
    delay(10); // Delay for a smoother transition
  }
}
```

Figure 38: Mini Water Pump Tester Code

A limitation appeared as the least PWM value the pump operates at is 150 under which the pump stops working. Another limitation was that the pump is not self-priming meaning it has to be fully submerged inside the water. Another connection was tested where the H-Bridge was not used and an NPN Transistor (TP120) was used instead as shown:

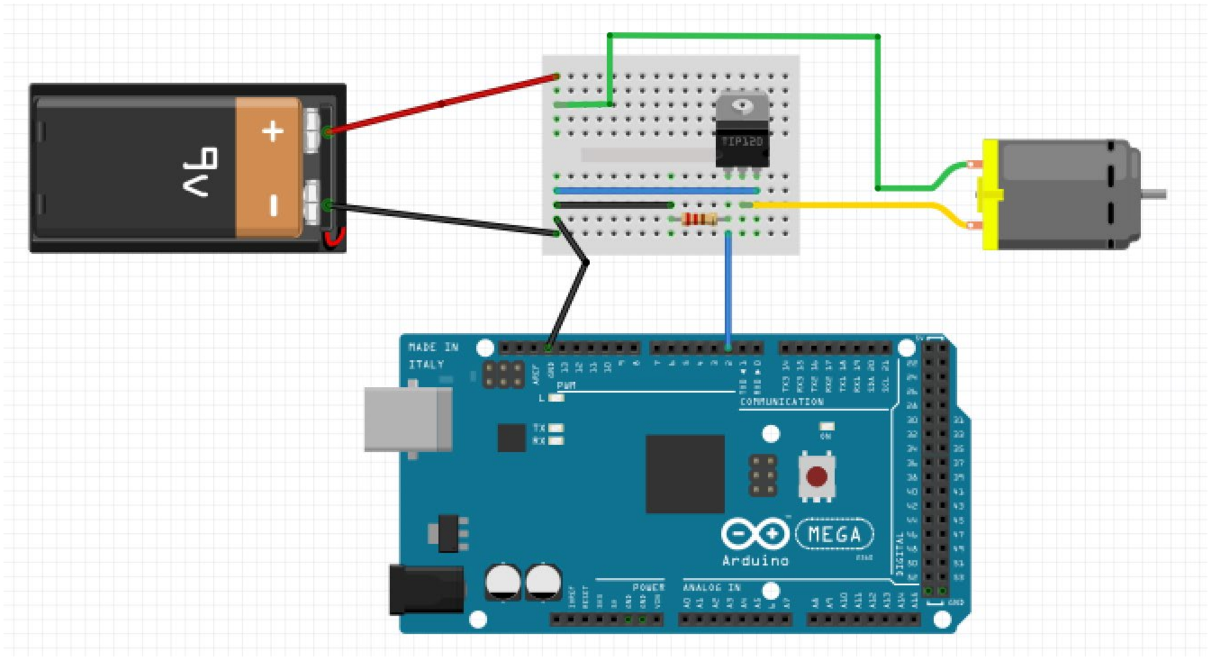


Figure 39: Mini Water Pump Second Trial Connection

The same code was used (Figure 38) and it returned the same result with a PWM threshold of 150. A brushless pump was used instead as it has higher operating voltage (12V) and it was connected as shown:

Figure 40: 12V Brushless Pump

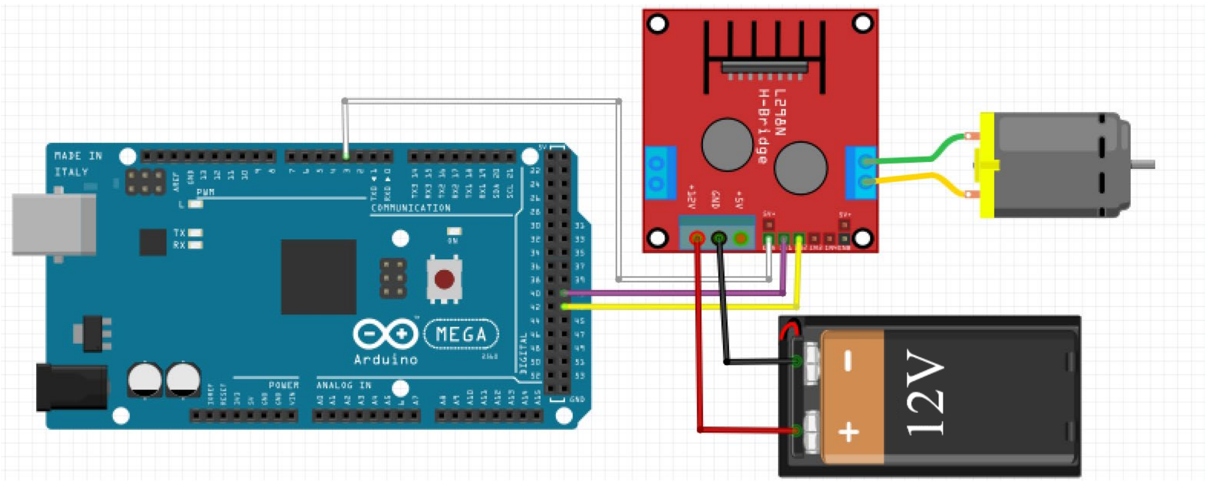


Figure 41: Brushless Pump Connection

The same code was used (Figure 38) and it returned a slightly better PWM threshold of 135. However, it has the same limitation of being a submersible pump and not a self-priming pump. Also, another limitation is that brushless motors do not work effectively with PWMs. The last pump tested was a 12V self-priming pump which had the same connections as the brushless pump (Figure 40) and used the same code (Figure 38).

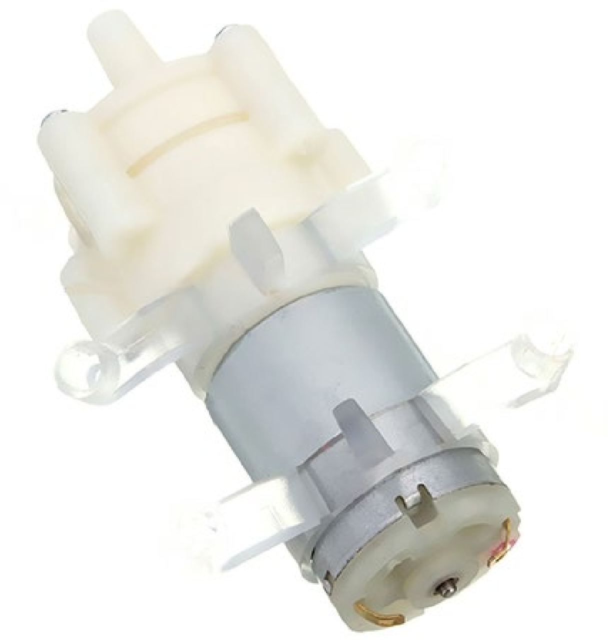


Figure 42: 12V Self-Priming Pump

This pump gave the best PWM threshold of 95. It had the ability to suck the liquid and push it with a head of 3 meters. Lastly, this pump was selected for the project.

6. Prototype Implementation:

A prototype was implemented to help in testing and debugging errors in both the code and hardware.

6.1. Laser Cutting:

The Solidworks model was transformed into a .dwg file to be ready for laser cutting. MDF wood with 5mm thickness was selected with a frame size of 120x80mm.

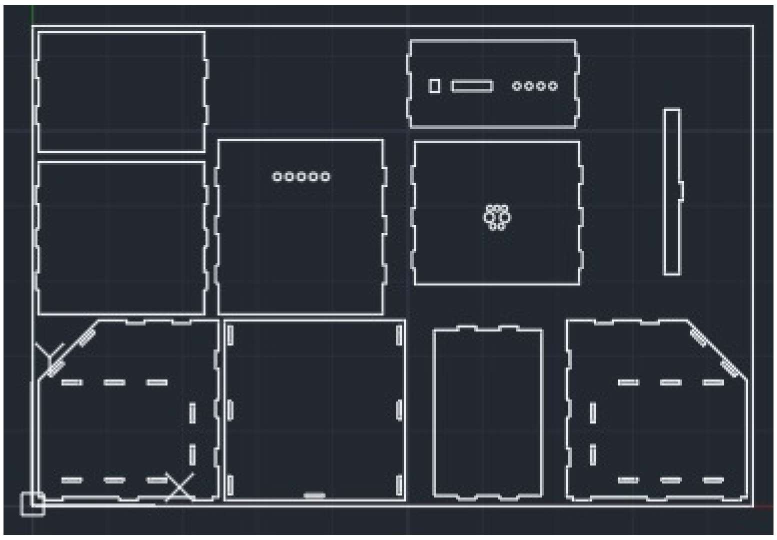


Figure 43: Laser Cutting Drawing

6.2. Spray Painting:

To give the model a cleaner look it was spray painted using Matte Black spray paint.

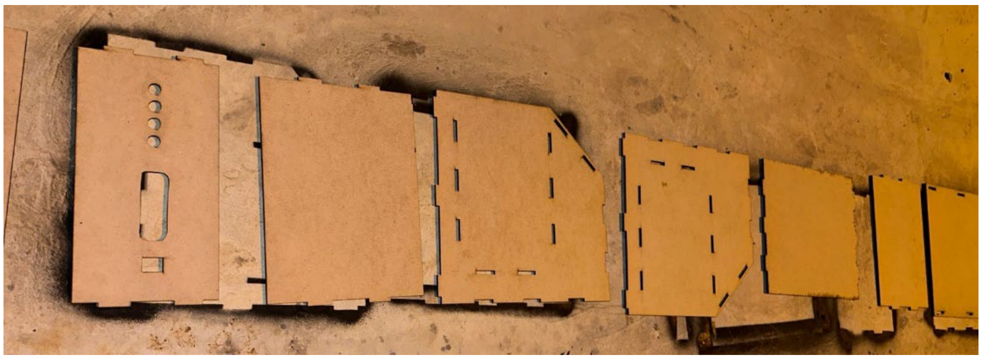


Figure 44: Protype Before Spray Painting

6.3. Model Assembly:

The prototype was finally assembled using the puzzle technique and was ready to start the testing and tuning phase.

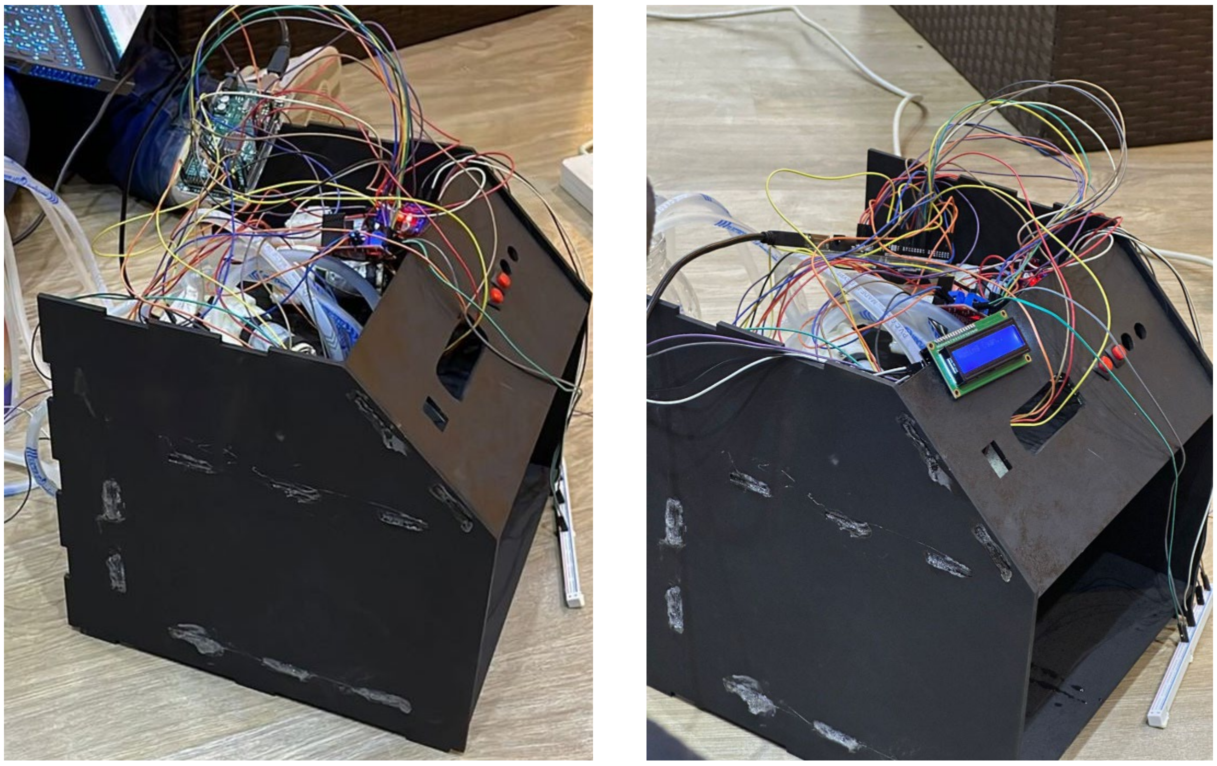


Figure 45: Protype Final Model

7. Writing Individual FreeRTOS Tasks and Testing Them:

FreeRTOS is used to avoid writing bare metal code and utilize the processor effectively using preemptive multitasking.

7.1. Color Sensor Task:

The color sensor tester code (Figure 22) was transformed into a freeRTOS task and started transforming the RGB values to CMYK values and percentages. The color sensor task was written as follows:

```
void readColorSensor(void *pvParameters){
  vTaskSuspendAll();  //Suspend the scheduler
      /*print scanning on LCD*/
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("Scanning...");
      /*print scanning on LCD*/

      readColors(); //Figure 11
      printColors();  //Figure 21

      /*claculate CMYK values using formula in section 3.1.
      and show it is calculating on LCD*/
      calculateCMYK();
      calculatePercentage();
      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.print("Calculating");
      lcd.setCursor(0, 1);
      lcd.print("CMYK....");
      vTaskDelay(pdMS_TO_TICKS(3000));
      //dummy delay to allow user to know it is calculating
      lcd.clear();
      lcd.setCursor(0, 0);
      /*claculate CMYK values and show it is calculating on LCD*/

      xTaskResumeAll(); //Resume the scheduler
      vTaskDelay(pdMS_TO_TICKS(20000*4 + 30000));
    }
  }
```

Figure 46: Color Sensor Task

In this task, the scheduler was suspended as this task's period is mainly dependent on the user and it has no fixed period.

7.2. Ultrasonic Sensor Task:

The ultrasonic sensor tester code in (Figure 27) was transformed into a freeRTOS task to read the distance periodically. The task code is as follows:

```
void readUltrasonicSensor(void *pvParameters){
  TickType_t xLastWakeTime ;// variable that stores last time task unblocked
  const TickType_t xPeriod = pdMS_TO_TICKS(130); // task should execute every 130ms
  xLastWakeTime = xTaskGetTickCount();
  while(1){
      duration = liquidHeight.ping_median();
      actualHeight = duration*0.034/2.0;
      vTaskDelayUntil(&xLastWakeTime, xPeriod);
  }
}
```

Figure 47: Ultrasonic Sensor Task

7.3. PID Control of Each Pump:

The PID control utilized the “PID_v1.h” library to automatically discretize and tune the system. The task takes the ultrasonic reading compares it with the desired height and calculates the new PWM value. The tasks code is as follows:

```
void PIDcontrolWhite(void *pvParameters){
  TickType_t xLastWakeTime ;// variable that stores last time task unblocked
  const TickType_t xPeriod = pdMS_TO_TICKS(20000*4 + 30000);
  xLastWakeTime = xTaskGetTickCount();
  while(1){
    while(actualHeight > desiredHeight){ //while error is positive
      whitePID.Compute(); //compute
      analogWrite(whiteEnable,pwmValue); //set new pwm value
    }
        digitalWrite(whiteEnable,LOW); //disable motor
        desiredHeight = cyanHeight; //change setpoint to next color’s setpoint
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Adding Cyan.."); //display adding the next color
        vTaskDelayUntil(&xLastWakeTime, xPeriod); //suspend
    }
}
```

Figure 48: Paint adding Task

7.4. Task Priorities:

After writing each task, prioritizing the tasks is the most crucial part of the programming. Assuming the tasks run periodically the following priorities were assigned:

```
xTaskCreate(readColorSensor, "readColorSensor", 1000, NULL, 7, NULL);
xTaskCreate(readUltrasonicSensor, "UltrasonicSensor", 1000, NULL, 6, NULL);
xTaskCreate(PIDcontrolKey, "Key", 1000, NULL, 5, NULL);
xTaskCreate(PIDcontrolMagenta, "Magenta", 1000, NULL, 4, NULL);
xTaskCreate(PIDcontrolYellow, "Yellow", 1000, NULL, 3, NULL);
xTaskCreate(PIDcontrolCyan, "Cyan", 1000, NULL, 2, NULL);
```

Figure 49: Task Priorities

Reading the Color Sensor is the first task to be executed and should not be preempted, consequently it was given the highest priority. After obtaining the colors, the actual height should be sampled after a certain amount of time preempting any PID control. Therefore, the ultrasonic sensor has a higher priority than all PID tasks. Lastly, the colors priorities are ordered in descending order in order of which colors is deployed first.

8. Tuning Each PID Controller Individually:

Water was used instead of paint and iterative testing was carried out tuning the gains of each color's PID controller to satisfy the design needs of 0 overshoot and steady state error.

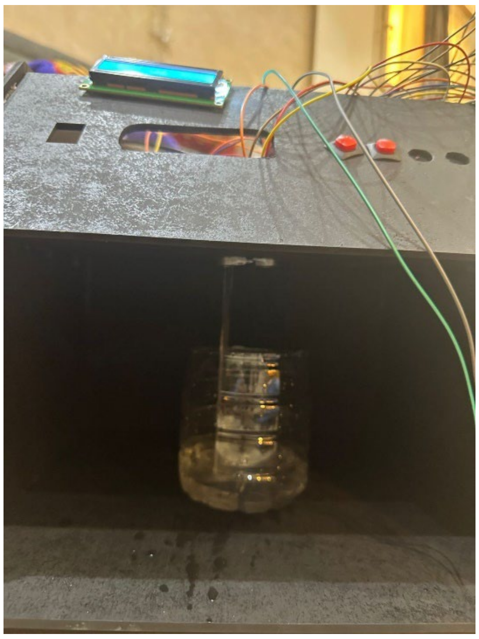


Figure 50: PID Controller Testing

9. Prototype Testing:

The prototype was assembled completely with all 5 pumps working together and the source code uploaded to the Arduino MEGA 2560. The prototype failed with a lot of overshoots due to its small size. This size made the PID controller control pigments in heights less than 0.2cm which was impossible to be done. A newer model was needed with a larger scale.

10. Solidworks of Final Model:

A final model was modeled on Solidworks and accurate dimensions were taken to ensure maximum efficiency.

10.1. 3D Model: Modeling the machine on Solidworks.

After the prototype failed to deliver the needed outcome, a newer model with twice the scale was modeled on Solidworks. The 3D model obtained was as follows:

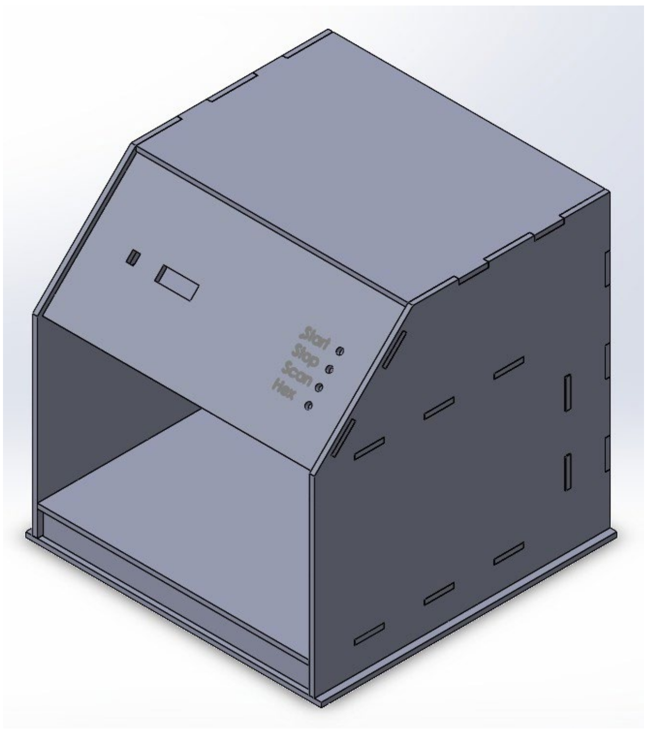


Figure 51: Final Model Solidworks

10.2. Model Dimensions:

The model was 600x600x600 mm. It had many features including a front panel with 4 button holes, LCD hole and a color sensor hole. Like the prototype, the final model was built to be assembled like a puzzle. Detailed dimensions is as follows:

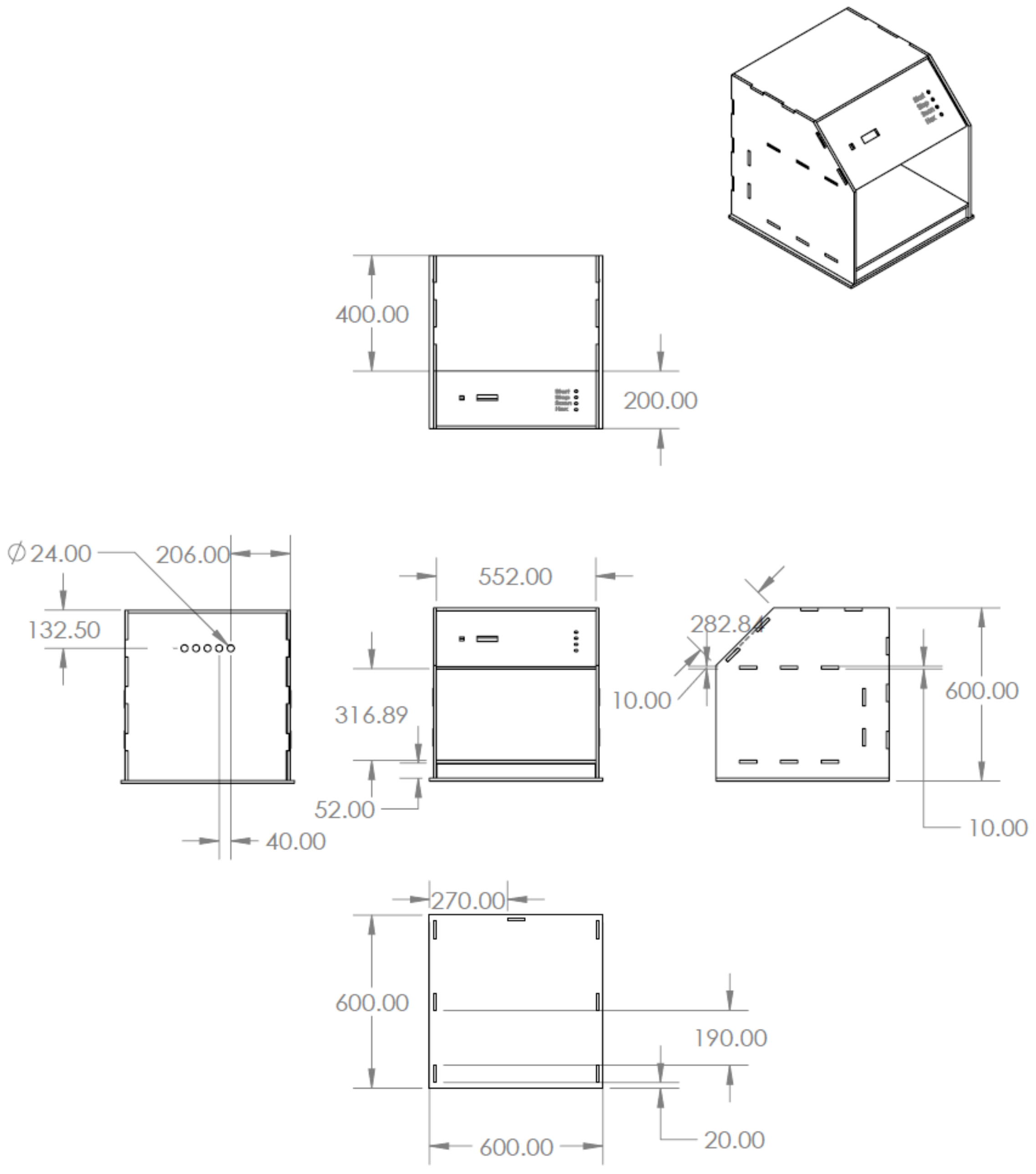


Figure 52: Final Model Dimensions

11. Final Model Implementation:

11.1. Laser Cutting:

The Solidworks model was transformed into a .dwg file to be ready for laser cutting. MDF wood with 5mm thickness was selected with a frame size of 120x80mm.

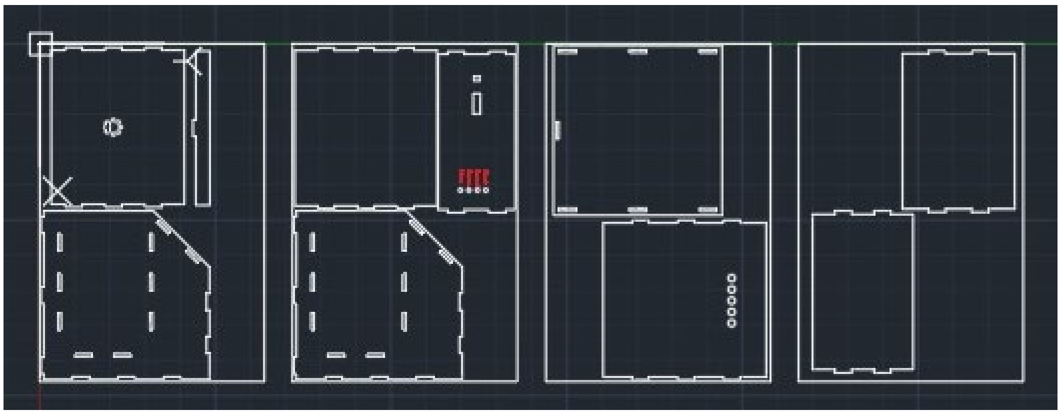


Figure 53: Laser Cutting Drawing

11.2. Model Assembly:

The final model was assembled using the puzzle technique and was ready to start the final testing and tuning phase.

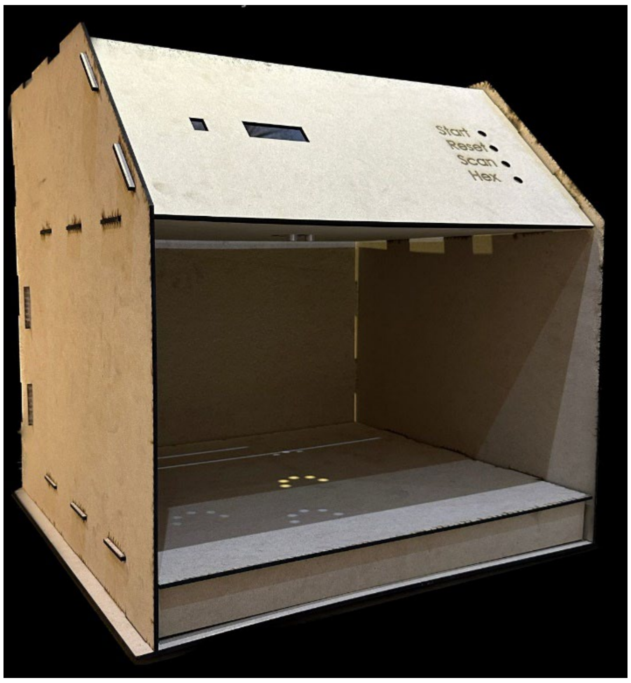


Figure 54: Final Model Before Spray Painting

11.3. Spray Paint

To give the model a cleaner look it was spray painted using Matte Black spray paint.

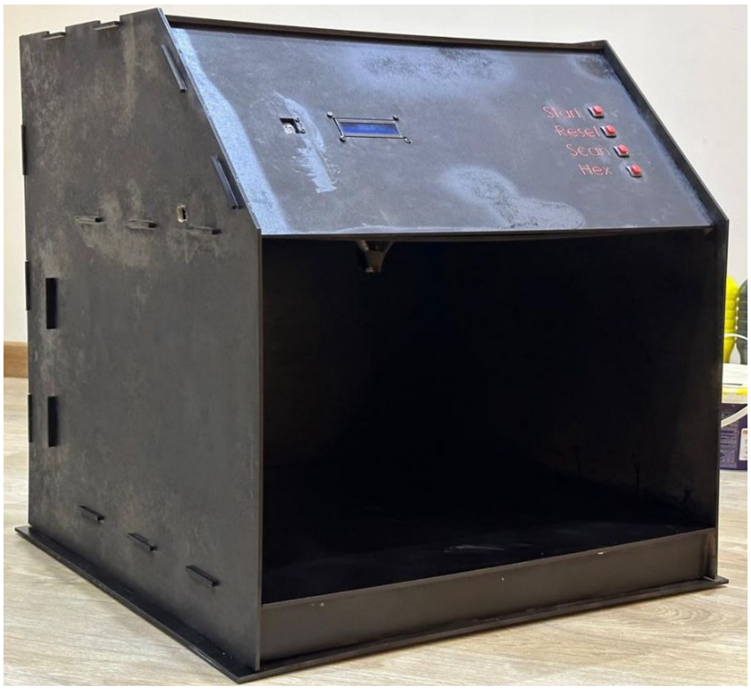


Figure 55: Final Model Spray Painted

12. Testing Different Paint Materials:

Different paint materials were experimented to find the most efficient material. This was done by doing extensive testing with different materials to try and obtain different colors.

12.1. Printer Ink:

The idea of using printer ink was the most logical as the printer already utilizes CMYK model to acquire different colors

Figure 56: Printer Inks

12.2. White Lacquer:

First trial was using thinned down white lacquer as base material and deploying inks in it. No accurate colors were obtained as the inks did not mix with lacquer but was instead suspended within it.

Figure 57: White Lacquer

12.3. White Plastic Matte:

During the second testing round, white plastic matte paint was used as the base material The same issue occurred and the inks did not mix with it but was suspended inside.

Figure 58: White Plastic Matte Paint

12.4. Plastic Matte Paint:

The final testing round was the most successful. Instead of using CMYK inks, Plastic Matte Paint was used with the colors Cyan, Yellow, Magenta and Black. This eliminated the need for base material as the pigments themselves are mixed with a base material already integrated inside the bottle. This testing round gave the most successful and accurate results.

13. Calculating Task Periods:

Assuming the tasks run periodically at certain frequencies, each task should have the right period in order to run. A simple code was run to know how long it takes for the task to run. It depended on getting the number of milliseconds the CPU has been running for twice, once when the task wakes up and again when it finishes executing. The difference between the two values is the runtime of the task. A sample code is as follows:

```
void task(void *pvParameters) {
int timeWaken = millis();

// Code to be executed

Serial.println("Runtime: " + String(millis() - timeWaken));
`
```

Figure 59: Calculating Runtime Code

After knowing the task periods, a time line was sketched by hand and the tasks were ordered in it by their runtime and priority. Each task's period was then calculated.

14. Testing the Code:

At first each pigment was given the same 5 cm maximum allowance to calculate the height required for each pigment. The obtained colors were darker than needed each time the machine ends its operation. This was due to the key color being added in huge quantities. The key allowance was adjusted to only 1.5 cm. This gave more accurate shades of color. However, one final problem stood. When the code finishes executing it does not loop back to the start and allow the user to choose another color. In order for the user to choose another color, the board had to be reset. This was due to periodic scheduling for aperiodic tasks. For example, the read color sensor task doesn't have a fixed period of execution as it purely depends on the time taken by the user to choose a color. Trying to schedule these aperiodic tasks periodically gave lots of problems.

15. Writing Aperiodic Tasks:

Another synchronization method was found to better utilize the CPU and let the tasks. Binary Semaphores were used to synchronize the tasks. The idea of a binary semaphore is that when the semaphore is given the semaphore counter is set to one. When a task tries to take the semaphore, the counter is set to zero. However, if the counter is already zero the task is blocked until another task gives the semaphore. This was a better way to synchronize the tasks and let the tasks run one by one then loop back to the first task. However, task priorities changed to the following to utilize the binary semaphores:

```
xTaskCreate(readColorSensor, "readColorSensor", 1000, NULL, 1, NULL);
xTaskCreate(readUltrasonicSensor, "UltrasonicSensor", 1000, NULL, 6, NULL);
xTaskCreate(PIDcontrolKey, "Key", 1000, NULL, 5, NULL);
xTaskCreate(PIDcontrolMagenta, "Magenta", 1000, NULL, 4, NULL);
xTaskCreate(PIDcontrolYellow, "Yellow", 1000, NULL, 3, NULL);
xTaskCreate(PIDcontrolCyan, "Cyan", 1000, NULL, 2, NULL);
```

Figure 60: New Task Priorities

```
SemaphoreHandle_t colrSensorSemaphore = xSemaphoreCreateCounting(1, 1);
SemaphoreHandle_t keySemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t yellowSemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t magentaSemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t cyanSemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t ultrasonicSemaphore = xSemaphoreCreateCounting(1, 0);
```

Figure 61: Semaphores Initialization

```
void PIDcontrolMagenta(void *pvParameters){
  while(1){
    while(1){
        if(xSemaphoreTake(magentaSemaphore, pdMS_TO_TICKS(3000)) == pdTRUE){
          xSemaphoreGive(yellowSemaphore);
          break;
        }
    }
      //rest of code
  }
}
```

Figure 62: Semaphores Handling

The idea of the semaphores is that each task that runs signals to the next task to get ready to run next as in (Figure 62). The order of task executions is

1) Color Sensor
2) Key Addition
3) Magenta Addition
4) Yellow Addition
5) Cyan Addition

At the beginning of initializing the semaphores all task semaphores are initialized with initial count of 0 except the color sensor semaphore. That means when other high priority tasks try to acquire their semaphore, they will get blocked. However, the color sensor semaphore count is one so the color sensor task runs. When it finishes execution, it gives the key semaphore to signal to the key task to run. When the key finishes it gives the magenta semaphore to signal to the magenta task to run. This happens till we reach the cyan task which gives the color sensor semaphore at the end to signal to the color sensor task to run again. The only periodic task in this code is the ultrasonic sensor task with the highest priority and a period of 130ms which is equal to the sampling time.

16. Developing Application:

Part of the vision for the project is having an application to let the user enter the desired hex code and review the color on. The following application was built using Windows Forms App (C#) on Visual Studio and utilizes Serial Communication with the Arduino board:

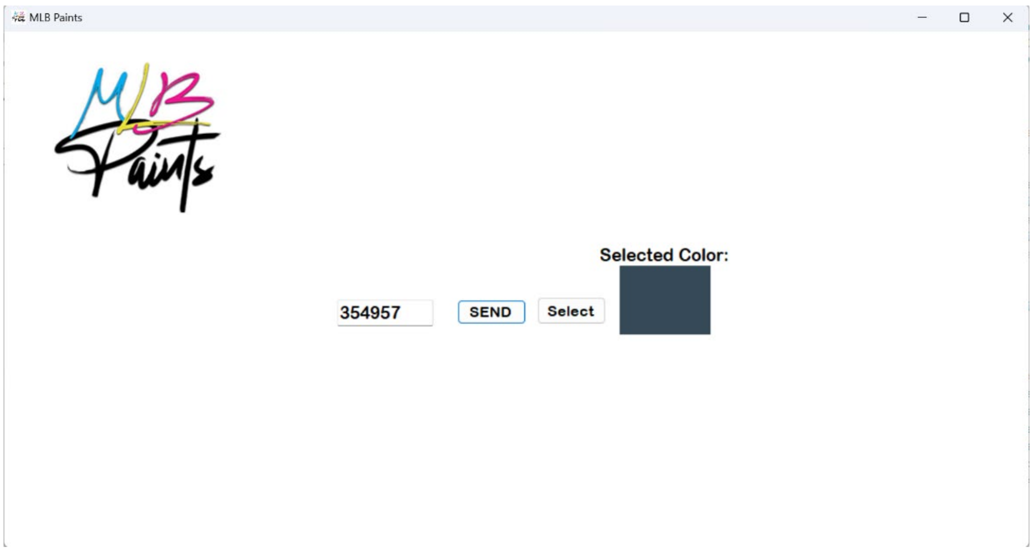


Figure 63: App Home Page

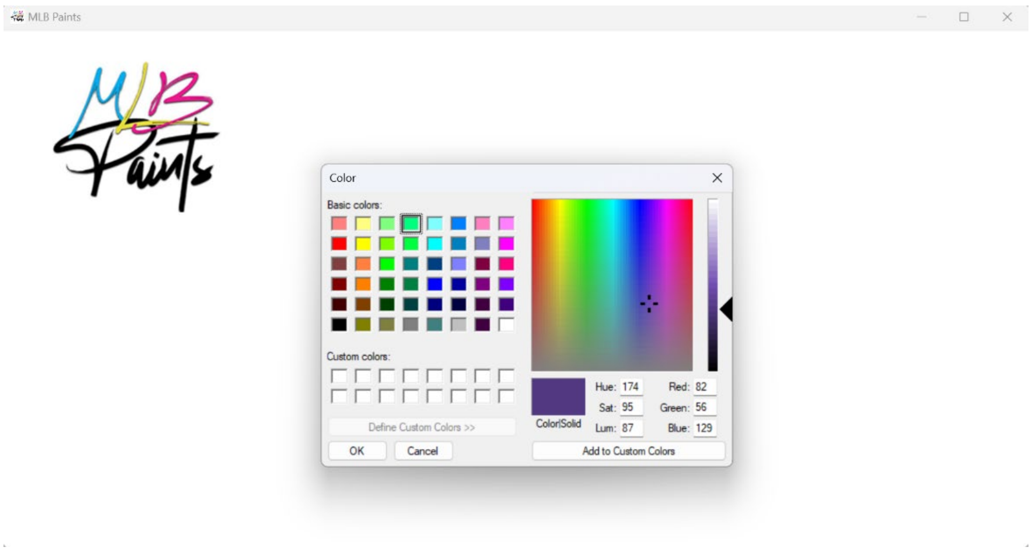


Figure 64: App Color Picker

This app starts serial communication between the PC and the Arduino board using serial communication. The app sends the color as a string and the Arduino receives and obtains the RGB values from it through the following code:

```
while(hexIsPressed){
      if(!print){
      /*print scanning on LCD*/
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("Waiting For");
      lcd.setCursor(0,1);
      lcd.print("HEX Code...");
      /*print scanning on LCD*/
        print = true;
      }
      if(Serial.available()){
        String data = Serial.readString().c_str();
        currRed = strtol(data.substring(0, 2).c_str(), NULL, 16);
        currGreen = strtol(data.substring(2, 4).c_str(), NULL, 16);
        currBlue = strtol(data.substring(4, 6).c_str(), NULL, 16);
        hexIsPressed = false;
      }
    }
```

Figure 65: Arduino RGB Receiver Code

```
private void sendButton_Click(object sender, EventArgs e)
        {
            string s = hexCode.Text;
            s.ToUpper();
            if(s.Length() == 6)
                serialPort1.Write(s);
            else
                errorBox.Text = "Invalid HEX Code!";
        }
```

Figure 66: App RGB Sender Code

Detailed app source code in Appendix.

17. Finalizing Model:

17.1. Printing Team Logo Stickers:

The team logo is a part of the identity of this project. Part of finalizing the model is printing the team logo sticker and sticking it on the model to give the team identity.

Figure 67: Team Logo Stickers

17.2. Screwing Hardware in Place:

Screwing the hardware in place is a crucial part to ensure no hardware conflicts occur due to hardware moving around

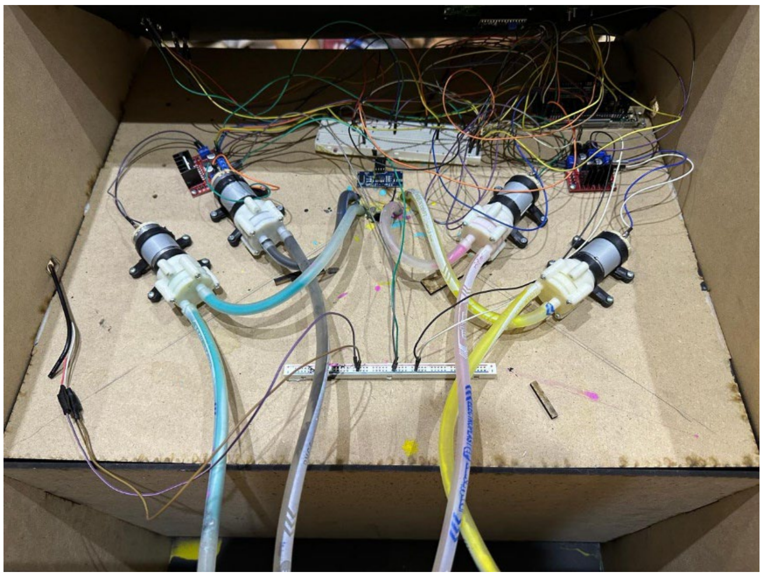


Figure 68: Hardware Connections

17.3. Creating CMYK Tanks:

Creating enough CMYK pigments to accommodate final testing phase to ensure that the machine is working properly

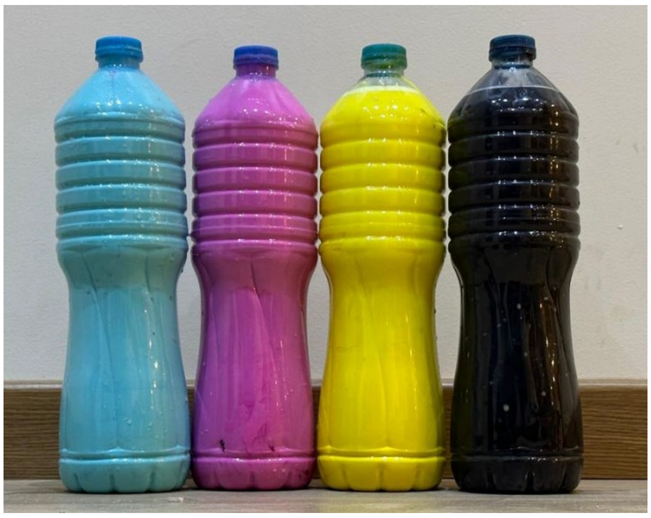


Figure 69: CMYK Pigments

18. Final Testing:

The machine went through final tests and it returned the color successfully using full PID control with no human intervention. The code loops back to the starts when it finishes without resetting. A color of HEX code #354957 was given to itand it gave near accurate color:

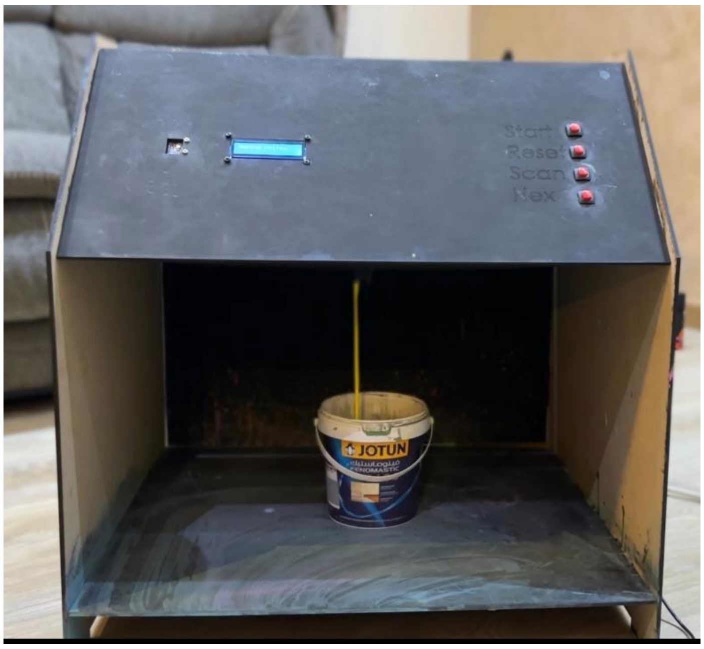


Figure 70: Final Testing

Figure 71: #354957 Color

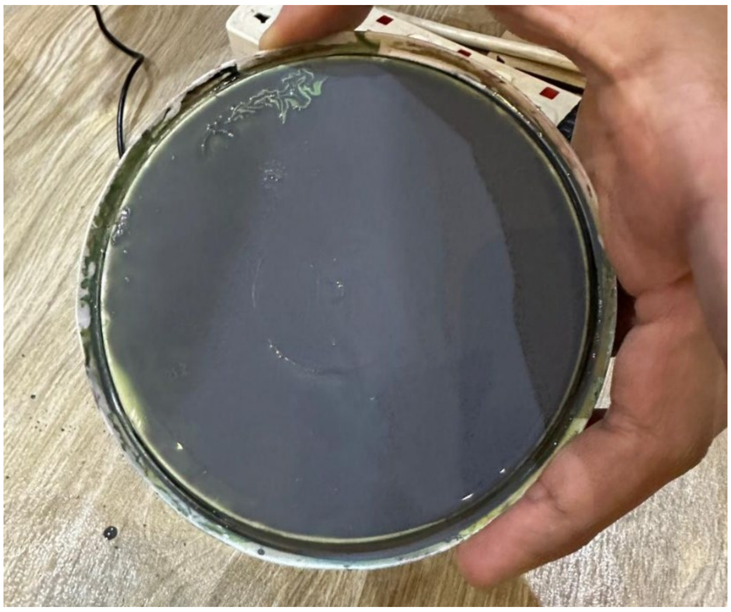


Figure 72: Machine Output

The machine gave near accurate results and operated as intended. Final touches were added to the machine and it was ready.

VI. Discussion

The machine gave the intended results but it was far away from correct. Various limitations were present that stopped the machine from working with accurate precision. One limitation is the absence of pure CMYK pigments. The inability of inks to mix with white paint forced the usage of dyed paint. However, paint shops have only predefined colors and they don't have pure CMYK colors. The absence of pure pigments made the machine give the right color but a different shade to it due to the difference between the pure pigments and used paint. Another limitation was the usage of ultrasonic sensor as it gives unstable readings. These unstable readings hinder the computation of the PID controller and gives wrong PWM values to the motor. This leads to overshooting in some cases and undershooting in other cases which gives different shades of the color not the accurate shade.

VII. Future Work

Several tasks can be made to develop the machine furthermore. Starting with the first limitation, purchasing pure CMYK pigments that are mixable with white paint will help with getting the correct shade of color. To solve the second limitation, a more powerful ultrasonic sensor could be fitted for more stable reading or changing the sensor type with another sensor that is more stable like water level sensor. Adding a mixing mechanism to the machine would make it fully automated for a better user experience. Implementing a more user-friendly GUI for the windows application would boost the overall user experience. Adding an option to the user to choose the desired amount of paint would ease the process on the user. Choosing material other than MDF as it is easily damaged by water would increase the lifetime of the machine. Adding a stopping mechanism that detects when the user entered his hand during mixing and halts the mixing process.

VIII. References

- S. C. Pratama, E. Susanto and A. S. Wibowo, "Design and implementation of water level control using gain scheduling PID back calculation integrator Anti Windup," *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, Bandung, Indonesia, 2016, pp. 101-104, doi: 10.1109/ICCEREC.2016.7814981.
- Mondal, Bikas & Rakshit, Sourav & Sarkar, Rajan & Mandal, Nirupama. (2016). Study of PID and FLC based Water Level Control Using Ultrasonic Level Detector. 10.1109/ICCECE.2016.8009560.
- Modeling, simulation and PID control of water tank model using MATLAB ... (n.d.). https://www.scs-europe.net/dlib/2019/ecms2019acceptedpapers/0177_mct_ecms2019_0069.pdf
- Getu, B. N. (n.d.). Water level controlling system using PID Controller. https://aurak.ac.ae/publications/Water-Level-Controlling-System-Using-Pid-Controller.pdf
- Diana has been an artist for over 26 years and has training in drawing. (2023, March 17). *Can you convert RGB to CMYK without losing color*. Retrieved March 26, 2023, from https://adventureswithart.com/convert-rgb-to-cmyk/
- *RGB color sensor with IR filter and white LED - TCS34725*. Future Electronics Egypt. (n.d.). Retrieved March 26, 2023, from https://store.fut-electronics.com/collections/image-rtc/products/color-sensor-module
- Technologies, F. L., & Instructables. (2022, December 9). *Mesomix - automated paint*

mixing machine. Retrieved March 26, 2023, from https://www.instructables.com/MESOMIX-Automated-Paint-Mixing-Machine/

- *Water pump - in/out outlet (12VDC)*. RAM Electronics. (2020, March 16). https://ram-e-shop.com/product/dc-pump-12vdc/
- *Fritzing*. electronics made easy. (n.d.). https://fritzing.org/learning/tutorials/building-circuit

IX. Appendix

1. Arduino Source Code

```
#include <Arduino_FreeRTOS.h>
#include <Wire.h>
#include <PID_v1.h>
#include <NewPing.h>
#include <Adafruit_TCS34725.h>
#include <LiquidCrystal_I2C.h>
#include "task.h"
#include "semphr.h"

/*  Define Pins */
#define startButton 24
#define hexButton 25
#define scanButton 26
#define cyanIn3 30
#define cyanIn4 31
#define magentaIn1 32
#define magentaIn2 33
#define yellowIn3 34
#define yellowIn4 35
#define keyIn1 36
#define keyIn2 37
#define cyanEnable 3
#define magentaEnable 4
#define yellowEnable 5
#define keyEnable 6
#define Trigger 22
#define Echo 23
/*  Define Pins */

/*  Define Flags  */
boolean scanIsPressed = false;
boolean hexIsPressed = false;
boolean startIsPressed = false;
boolean addedC = false;
boolean addedM = false;
boolean addedY = false;
boolean addedK = false;
/*  Define Flags  */

int consuctiveReads = 0;
float prevRed = 0, prevGreen = 0, prevBlue = 0;
float currRed = 0,currGreen = 0, currBlue = 0;
double Cyan;
double Magenta;
double Yellow;
double Key;
double sum;
double duration = 0;
double actualHeight = 50;
double cyanHeight = 0;
double magentaHeight = 0;
double yellowHeight = 0;
double keyHeight = 0;
double cupHeight = 20;
double pwmValue = 0x00;
double desiredHeight = 0;
double pigmentKp = 120, pigmentKi = 5, pigmentKd = 150;

/* Initialise Color Sensor */
Adafruit_TCS34725 colorSensor =
Adafruit_TCS34725(TCS34725_INTEGRATIONTIME_614MS, TCS34725_GAIN_1X);
/* Initialise Color Sensor */

/* Initialise LCD */
LiquidCrystal_I2C lcd(0x27,16,2);
/* Initialise LCD */

/* Initialise Ultrasonic */
NewPing liquidHeight(Trigger, Echo, 35);
/* Initialise Ultrasonic */

/* Initialise Controller */
PID pigmentPID(&actualHeight, &pwmValue, &desiredHeight, pigmentKp, pigmentKi,
pigmentKd, REVERSE);
/* Initialise Controller */

SemaphoreHandle_t colorSensorSemaphore = xSemaphoreCreateCounting(1, 1);
SemaphoreHandle_t keySemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t yellowSemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t magentaSemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t cyanSemaphore = xSemaphoreCreateCounting(1, 0);
SemaphoreHandle_t ultrasonicSemaphore = xSemaphoreCreateCounting(1, 0);

byte smiley[8] = {
  0b00000,
  0b01010,
  0b00000,
  0b10001,
  0b01110,
  0b00000,
  0b01010,
  0b00100
```

```
};

void PIDcontrolCyan(void *pvParameters){
  while(1){
    while(1){
        if(xSemaphoreTake(cyanSemaphore, pdMS_TO_TICKS(3000)) == pdTRUE){
//Wait for signal from Yellow
          break;
        }
    }
    while(actualHeight > desiredHeight && !addedC){ //While error is +ve and
cyan was not added
      pigmentPID.Compute(); //Compute PWM
      analogWrite(cyanEnable, pwmValue);  //Set new PWM
    }
      digitalWrite(cyanEnable,LOW); //Turn off motor
      if(!addedC){  //if cyan not added before
        addedC= true;

        /* Print Shake.. */
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Shake It Well");
        lcd.write(0);
        vTaskDelay(pdMS_TO_TICKS(5000)); //Dummy delay to allow user to see
the msg
        /* Print Shake.. */

        /* Print Thank You... */
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Thank You");
        vTaskDelay(pdMS_TO_TICKS(5000)); //Dummy delay to allow user to see
the msg
        /* Print Thank You... */

        xSemaphoreGive(sensorSemaphore);  //Signal to color sensor to start
        }
    }
}

void PIDcontrolYellow(void *pvParameters){
  while(1){
    while(1){
        if(xSemaphoreTake(yellowSemaphore, pdMS_TO_TICKS(3000)) == pdTRUE){
//Wait for signal from Magenta
          xSemaphoreGive(cyanSemaphore);  //Signal to cyan to start
          break;
        }
    }
    while(actualHeight > desiredHeight && !addedY){ //While error is +ve and
yellow was not added
      pigmentPID.Compute(); //Compute PWM
      analogWrite(yellowEnable, pwmValue);  //Set new PWM
    }
        digitalWrite(yellowEnable,LOW); //Turn off motor
        if(!addedY){  //if yellow not added before
        desiredHeight = cyanHeight; //Set setpoint to cyan setpoint

        /* Print Adding Cyan.. */
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Adding CYAN..");
        /* Print Adding Yellow.. */

        addedY = true;
        }
    }
}

void PIDcontrolMagenta(void *pvParameters){
  while(1){
    while(1){
        if(xSemaphoreTake(magentaSemaphore, pdMS_TO_TICKS(3000)) == pdTRUE){
//Wait for signal from Key
          xSemaphoreGive(yellowSemaphore);  //Signal to yellow to start
          break;
        }
    }
    while(actualHeight > desiredHeight && !addedM){ //While error is +ve and
magenta was not added
      pigmentPID.Compute();   //Compute PWM
      analogWrite(magentaEnable, pwmValue); //Set new PWM
    }
        digitalWrite(magentaEnable,LOW);  //Turn off motor
        if(!addedM){ //if magenta not added before
        desiredHeight = yellowHeight; //Set setpoint to yellow setpoint

        /* Print Adding Yellow.. */
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Adding Yellow..");
        /* Print Adding Yellow.. */

        addedM = true;
        }
    }
}

void PIDcontrolKey(void *pvParameters){
  while(1){
    while(1){
        if(xSemaphoreTake(keySemaphore, pdMS_TO_TICKS(3000)) == pdTRUE){
//Wait for signal from color sensor
          xSemaphoreGive(magentaSemaphore); //Signal to magenta to start
          break;
        }
    }
    while(actualHeight > desiredHeight && !addedK){ //While error is +ve and
key was not added
      pigmentPID.Compute(); //Compute PWM
      analogWrite(keyEnable, pwmValue); //Set new PWM
    }
        digitalWrite(keyEnable,LOW);  //Turn off motor
        if(!addedK){  //If key not added before
        desiredHeight = magentaHeight;  //Set setpoint to magenta setpoint

        /* Print Adding Magenta.. */
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Adding Magenta..");
        /* Print Adding Magenta.. */

        addedK = true;
        }
    }
}

void readUltrasonicSensor(void *pvParameters){
  TickType_t xLastWakeTime ;// variable that stores last time task unblocked
  const TickType_t xPeriod = pdMS_TO_TICKS(130);// task should execute every
130 ms
  xLastWakeTime = xTaskGetTickCount();  //Get current ticks
  while(1){
          duration = liquidHeight.ping_median();  //Read 5 consuctive reads
and return median
          actualHeight = duration*0.034/2.0;  //Calculate height
        vTaskDelayUntil(&xLastWakeTime, xPeriod); //Sleep
  }
}

void readColors(){
  consuctiveReads = 0;
    while(consuctiveReads < 2){
```

```
      colorSensor.getRGB(&currRed, &currGreen, &currBlue);
      if(abs((int)prevRed - (int)currRed) <=1)
        consutiveReads++;
      else
        consutiveReads = 0;
      prevRed = currRed;
    }
    consutiveReads = 0;
    while(consutiveReads < 2){
      colorSensor.getRGB(&currRed, &currGreen, &currBlue);
      if(abs((int)prevGreen - (int)currGreen) <=1)
        consutiveReads++;
      else
        consutiveReads = 0;
      prevGreen = currGreen;
    }
    consutiveReads = 0;
    while(consutiveReads < 2){
      colorSensor.getRGB(&currRed, &currGreen, &currBlue);
      if(abs((int)prevBlue - (int)currBlue) <=1)
        consutiveReads++;
      else
        consutiveReads = 0;
      prevBlue = currBlue;
    }
}

void calculateCMYK(){
  /*Calculate CMYK values*/
  currRed /= 255.0;
  currGreen /= 255.0;
  currBlue /= 255.0;
  Key = 1 - max(max(currRed, currGreen), currBlue);
  Cyan = (1.0-currRed-Key)/(1.0-Key);
  Magenta = (1.0-currGreen-Key)/(1.0-Key);
  Yellow = (1.0-currBlue-Key)/(1.0-Key);
  /*Calculate CMYK values*/
}

void calculatePercentage(){
  /*Calculate desired height of each pigment*/
  keyHeight = 29.5  - Key * 1.5;
  magentaHeight = keyHeight  - Magenta * 5;
  yellowHeight = magentaHeight  - Yellow * 5;
  cyanHeight = yellowHeight  - Cyan * 5;
  /*Calculate desired height of each pigment*/
}
```

```
void readColorSensor(void *pvParameters){

  while(1){
    while(1){
        if(xSemaphoreTake(sensorSemaphore, pdMS_TO_TICKS(3000)) == pdTRUE){
//Wait for signal from cyan to start
          /* Falsify booleans */
          addedC = false;
          addedM = false;
          addedY = false;
          addedK = false;
          startIsPressed = false;
          hexIsPressed = false;
          scanIsPressed = false;
          break;
          /* Falsify booleans */
        }
      }

    /* Print Hello */
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("Hello, Please");
    lcd.setCursor(0,1);
    lcd.print("Choose Mode");
    /* Print Hello */

    while(!scanIsPressed && !hexIsPressed){
      if(digitalRead(scanButton)==1){ //if scan is not pressed before and it
was pressed
        vTaskDelay(pdMS_TO_TICKS(30)); //debouncing
        if(digitalRead(scanButton) == 1){ //if still pressed
          scanIsPressed = true; //scan is pressed
        }
      }
      if(digitalRead(hexButton)==1){ //if scan is not pressed before and it
was pressed
        vTaskDelay(pdMS_TO_TICKS(30)); //debouncing
        if(digitalRead(hexButton) == 1){ //if still pressed
          hexIsPressed = true; //scan is pressed
        }
      }
    }
    if(scanIsPressed){

      /*print scanning on LCD*/
      lcd.clear();
      lcd.setCursor(0,0);
```

```
        lcd.print("Scanning...");
        /*print scanning on LCD*/

        readColors(); //scan colors
        scanIsPressed = false;
      }
      boolean print = false;
      while(hexIsPressed){
        if(!print){
        /*print scanning on LCD*/
        lcd.clear();
        lcd.setCursor(0,0);
        lcd.print("Waiting For");
        lcd.setCursor(0,1);
        lcd.print("HEX Code...");
        /*print scanning on LCD*/
          print = true;
        }
        if(Serial.available()){ //Wait for data
          String data = Serial.readString().c_str();  //Capture data
          currRed = strtol(data.substring(0, 2).c_str(), NULL, 16); //Convert
first 2 letters from hex string to int
          currGreen = strtol(data.substring(2, 4).c_str(), NULL, 16); //Convert
second 2 letters from hex string to int
          currBlue = strtol(data.substring(4, 6).c_str(), NULL, 16);  //Convert
last 2 letters from hex string to int
          hexIsPressed = false;
        }
      }
        /*display color hex code on LCD*/
        lcd.clear();
        lcd.setCursor(0,0);
        lcd.print("Color is: ");
        lcd.setCursor(0,1);
        lcd.print("#");
        if(currRed < 16)
          lcd.print('0');
        lcd.print((int)currRed, HEX);
        if(currGreen < 16)
          lcd.print('0');
        lcd.print((int)currGreen, HEX);
        if(currBlue < 16)
          lcd.print('0');
        lcd.print((int)currBlue, HEX);
        /*display color hex code on LCD*/

        while((!startIsPressed)){ //while start is not pressed wait till it is
pressed
```

```
          if(digitalRead(startButton) == 1){ //if start is pressed
            vTaskDelay(pdMS_TO_TICKS(30)); //debouncing
          if(digitalRead(startButton) == 1){ //if still pressed
            startIsPressed = true; //scan is pressed
        }
      }
     }
      startIsPressed = false;

      /*claculate CMYK values and show it is calculating on LCD*/
      calculateCMYK();
      calculatePercentage();
      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.print("Calculating");
      lcd.setCursor(0, 1);
      lcd.print("CMYK....");
      vTaskDelay(pdMS_TO_TICKS(3000)); //dummy delay to allow user to know it
is calculating
      lcd.clear();
      lcd.setCursor(0, 0);
      /*claculate CMYK values and show it is calculating on LCD*/

      lcd.print("Adding KEY...");   //Show user that white paint is being
added
      desiredHeight = keyHeight;  //set desired height for PID control
      xSemaphoreGive(keySemaphore); //Signal to key to start
    }
  }

void setup(){
  Serial.begin(9600);  //start serial connection with LCD and Color Sensor
  lcd.init();   //Intialize the LCD
  colorSensor.begin();  //Intialize the Color Sensor
  lcd.backlight();  //Turn on the LCD Backlight
  lcd.createChar(0, smiley);
  pigmentPID.SetMode(AUTOMATIC);   //Intialize PID
  pigmentPID.SetTunings(pigmentKp, pigmentKi, pigmentKd);  //Set PID gains

  /*Intialize buttons as inputs*/
  pinMode(startButton,INPUT);
  pinMode(hexButton,INPUT);
  pinMode(scanButton,INPUT);
  /*Intialize buttons as inputs*/

  /*Intialize motors direction and enable pins*/
  pinMode(cyanIn3,OUTPUT);
  pinMode(cyanIn4, OUTPUT);
  pinMode(magentaIn1,OUTPUT);
  pinMode(magentaIn2, OUTPUT);
  pinMode(yellowIn3,OUTPUT);
  pinMode(yellowIn4, OUTPUT);
  pinMode(keyIn1,OUTPUT);
  pinMode(keyIn2, OUTPUT);
  pinMode(cyanEnable, OUTPUT);
  pinMode(magentaEnable,OUTPUT);
  pinMode(yellowEnable,OUTPUT);
  pinMode(keyEnable, OUTPUT);
  /*Intialize motors direction and enable pins*/

  /*Set motor directions*/
  digitalWrite(cyanIn3, HIGH);
  digitalWrite(cyanIn4, LOW);
  digitalWrite(magentaIn1, HIGH);
  digitalWrite(magentaIn2, LOW);
  digitalWrite(yellowIn3, HIGH);
  digitalWrite(yellowIn4, LOW);
  digitalWrite(keyIn1, HIGH);
  digitalWrite(keyIn2, LOW);
  digitalWrite(cyanEnable, LOW);
  digitalWrite(magentaEnable,LOW);
  digitalWrite(yellowEnable,LOW);
  digitalWrite(keyEnable, LOW);
  /*Set motor directions*/

  pinMode(Trigger,OUTPUT);  //Intialize Ultrasonic sensor trigger pin
  pinMode(Echo,INPUT);  //Intialize Ultrasonic sensor echo pin

  /*Intialize tasks*/
  xTaskCreate(readColorSensor, "readColorSensor", 1000, NULL, 1, NULL);
  xTaskCreate(readUltrasonicSensor, "readUltrasonicSensor", 1000, NULL, 6,
NULL);
  xTaskCreate(PIDcontrolKey, "Key", 1000, NULL, 5, NULL);
  xTaskCreate(PIDcontrolMagenta, "Magenta", 1000, NULL, 4, NULL);
  xTaskCreate(PIDcontrolYellow, "Yellow", 1000, NULL, 3, NULL);
  xTaskCreate(PIDcontrolCyan, "Cyan", 1000, NULL, 2, NULL);
  /*Intialize tasks*/

  vTaskStartScheduler();  //Start Scheduler
}

void loop(){

}
```

2. App Source Code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Reflection;
using System.Drawing.Drawing2D;

namespace MLB_Paints
{
    public partial class Form1 : Form
    {
        Boolean success = false;
        public delegate void d1(string indata);
        public Form1()
        {
            InitializeComponent();
            errorBox.Text = "";
            pictureBox1.BackColor = Color.Black;
            serialPort1.Open();
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void sendButton_Click(object sender, EventArgs e)
        {
            string s = hexCode.Text;
            s.ToUpper();
            if(success)
                serialPort1.Write(s);
            else
                errorBox.Text = "Invalid HEX Code!";
        }

        private void hexCode_TextChanged(object sender, EventArgs e)
        {
            success = true;
            errorBox.Text = "";
            string hex = hexCode.Text;
            hex.ToUpper();
            if (hexCode.Text.Length > 6) {
                errorBox.Text = "Invalid HEX Code!";
                pictureBox1.BackColor = Color.Transparent;
                success = false;
            }
            for (int i = 0; i < hex.Length; i++)
            {
                if (!((hex[i] >= '0' && hex[i] <= '9') || (hex[i] >= 'A' &&
hex[i] <= 'F') || (hex[i] >= 'a' && hex[i] <= 'f')))
                {
                    errorBox.Text = "Invalid HEX Code!";
                    success = false;
                    break;
                }
            }
            if (success)
            {
                for(int i = hex.Length-1; i < 6; i++)
                {
                    hex += "0";
                }
                int red = int.Parse(hex.Substring(0, 2),
System.Globalization.NumberStyles.HexNumber);
                int green = int.Parse(hex.Substring(2, 2),
System.Globalization.NumberStyles.HexNumber);
                int blue = int.Parse(hex.Substring(4, 2),
System.Globalization.NumberStyles.HexNumber);

                // Create a Color object using the extracted RGB values
                Color color = Color.FromArgb(red, green, blue);
                errorBox.Text = "";
                // Set the PictureBox's background color
                pictureBox1.BackColor = color;
                success = true;
              }
        }

        private void serialPort1_DataReceived(object sender,
System.IO.Ports.SerialDataReceivedEventArgs e)
        {
            string indata = serialPort1.ReadLine();
            d1 writeit = new d1(Write2Form);
            Invoke(writeit, indata);
        }
        public void Write2Form(string indata)
        {
            hexCode.Text = indata;
        }
        private void selectorButton_Click(object sender, EventArgs e)
        {
            ColorDialog colorDialog = new ColorDialog();

            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                Color selectedColor = colorDialog.Color;
                string hexValue = ColorTranslator.ToHtml(selectedColor);
                pictureBox1.BackColor = selectedColor;
                hexCode.Text = hexValue.Substring(1,hexValue.Length-1);
            }
        }

        private void pictureBox2_Click(object sender, EventArgs e)
        {

        }

        private void errorBox_Click(object sender, EventArgs e)
        {

        }

        private void pictureBox1_Paint(object sender, PaintEventArgs e)
        {
        }
    }
}
```