<< IN 1901 - Microcontroller Based Application Development Project>>

PROJECT PROPOSAL REPORT

Level 01

Automated Potato Grading machine

Examiner Mr. B. H. Sudantha

Submitted by : Group 08 / IT

K N Balasooriya 224017L

W G C W Bandara 224022X

M R M Muadh 224127A

S A Gamaarachchi 224054V

M.I. Abdulla 224001H

Bachelor of Information Technology

Faculty of Information Technology

University of Moratuwa

Introduction

Potatoes are a staple food worldwide, and their quality and size are crucial factors for both suppliers andconsumers. However, the manual sorting process is inefficient and prone to errors, leading to dissatisfaction and operational challenges. Our project, the Automated Potato Sorting Machine, aims to address these issues by introducing an innovative solution that streamlines the grading process.

Problem in Brief

The manual sorting of potatoes poses significant challenges in terms of time consumption, labor intensity,and error rates. Current methods are not only inefficient but also prone to inconsistencies, leading to customer dissatisfaction and operational inefficiencies within grocery shops and supermarkets. The inability to quickly and accurately categorize potatoes into small, medium, and large sizes results in suboptimal inventory management and compromises the quality of service provided to customers. Additionally, the reliance on manual labor for sorting tasks contributes to increased labor costs and potential health hazards for workers, especially in environments where hygiene standards must be rigorously maintained. Thus, there exists a pressing need for a modernized solution that automates thepotato grading process, streamlines operations, reduces labor costs, and ultimately enhances overall customer satisfaction and operational efficiency in the potato supply chain.

Significance of Study

The significance of our study lies in its potential to revolutionize the potato supply chain and improve theoverall efficiency and customer satisfaction within the grocery and supermarket industry. By addressing the inherent challenges of manual potato sorting, our automated grading machine promises several key benefits:

- Enhanced Efficiency: Automation of the grading process will significantly reduce the time and laborrequired for sorting potatoes, leading to faster inventory turnover and increased productivity for grocery shops and supermarkets.

- Improved Accuracy: The integration of advanced technologies such as sensors and motors ensure precise and consistent sorting of potatoes into small, medium, and large sizes, minimizing errors and discrepancies in inventory management.

- Cost Reduction: By replacing manual labor with automation, our solution offers potential costsavings for businesses by reducing labor expenses and minimizing losses due to inaccuracies in sorting.

- Enhanced Customer Satisfaction: With potatoes sorted accurately and efficiently, customers will experience improved shopping experiences, as they can easily find potatoes of the desired size, leadingto increased loyalty and repeat business.

- Hygiene and Safety: Automation of the grading process ensures adherence to stringent hygienestandards, reducing the risk of contamination and ensuring the safety of both consumers and workers involved in the potato supply chain.

Aim and Objectives

Aim:

Our primary aim is to develop an automated potato grading machine capable of efficiently sorting potatoes into small, medium, and large sizes, catering to the diverse needs of customers. Additionally, weaim to optimize resource utilization and minimize wastage by accurately measuring and dispensing potatoes based on customer requirements.

Objectives:

- Develop a robust and reliable potato grading machine that accurately differentiates between small, medium, and large potatoes.
- Implement user-friendly interfaces for easy input of desired potato size and quantity by customers.
- Integrate sensors and motors for precise control and automation of the grading process.
- Enhance efficiency and reduce operational costs for grocery shops and supermarkets.
- Ensure the safety and hygiene of the grading process to meet regulatory standards and consumer expectations.
- To develop a system for automatic fertilization and pest control that is safe for ornamental plants and does not harm the environment or human health.

Literature Study

Research in automated sorting systems for agricultural produce, including potatoes, highlights the efficacy of integrating sensors, actuators, and machine learning algorithms to enhance efficiency and accuracy in sorting processes. Technological advancements in sensor technology and automation have enabled the development of sophisticated systems capable of real-time detection and classification based on quality parameters. While these systems have significantly improved product quality and operational efficiency in the agricultural and food processing industries, challenges such as high implementation costs and integration complexities remain. Nonetheless, ongoing research aims to address these challenges and further enhance the capabilities of automated sorting systems to meet the evolving demands of the potato supply chain.

Proposed Solution

Our solution aims to revolutionize the potato sorting process by leveraging advanced technologies and innovative design principles to automate and optimize the grading process. Central to our solution is theintegration of multiple components, each meticulously designed to perform specific functions with precision and efficiency.

Features of the Proposed Solution

- Ultrasonic Sensor: Essential for accurately measuring potato levels within the container, the ultrasonic sensor enables precise control over the dispensing process, ensuring optimal sorting efficiency.

- Servo Motors: Integral to the operation of the servo motor-controlled door, servo motors facilitate controlled and precise dispensing of potatoes based on user input, enhancing the accuracy of the sorting process.

- Wiper Motors: These motors induce vibrations to facilitate the sorting process within the filtering mechanism, ensuring efficient separation of potatoes into small, medium, and large sizes.

- Load Cells: Incorporated within the designated containers, load cells enable accurate measurement and monitoring of potato quantities, optimizing inventory management and reducing wastage.

- Used in conjunction with conveyor belts, stepper motors provide precise control over the movement of potatoes, ensuring smooth and accurate transportation throughout the sorting process.

Nature of the Solution

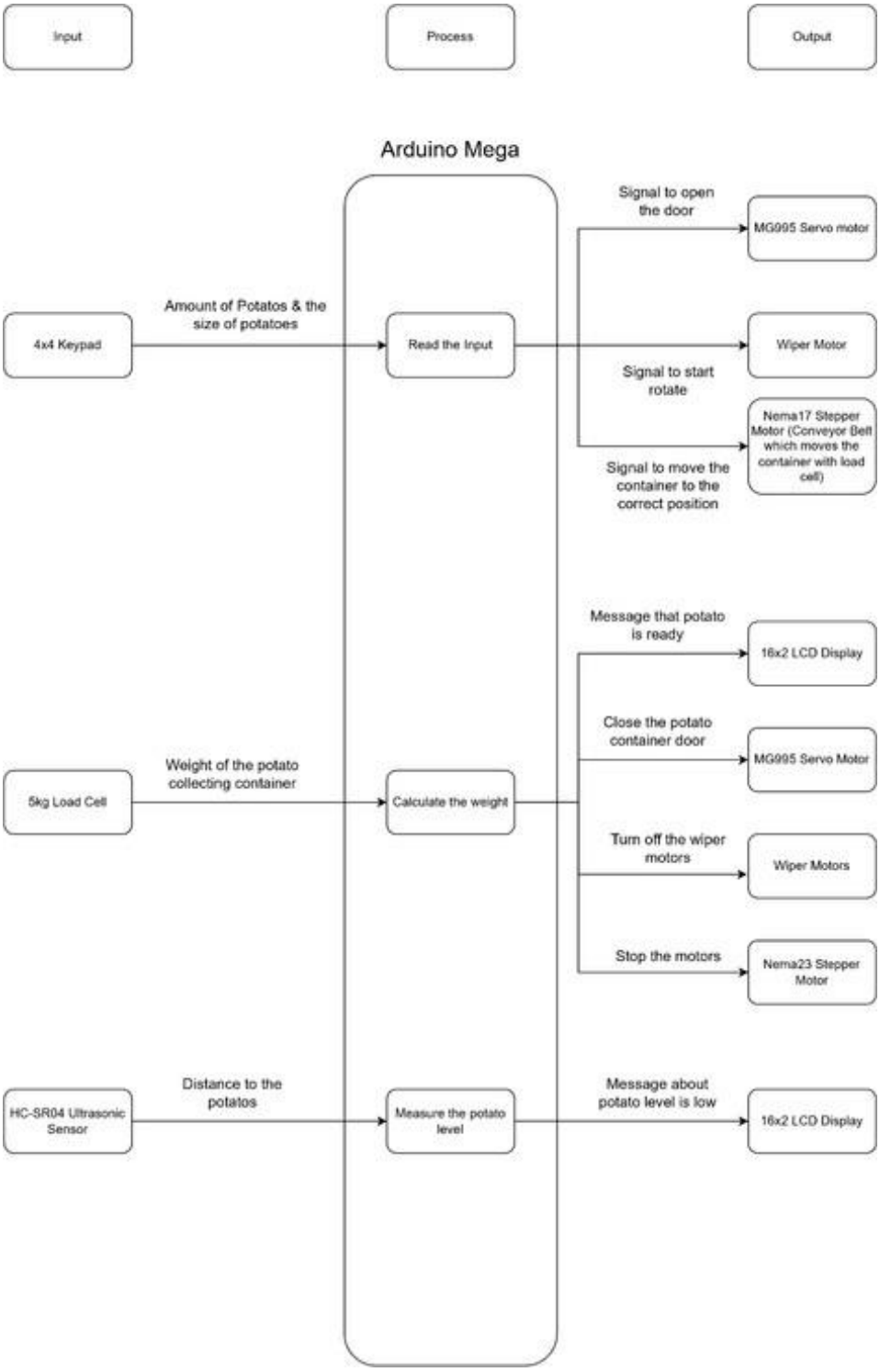


Figure 01: Block diagram of the input, process and output

Solution Design

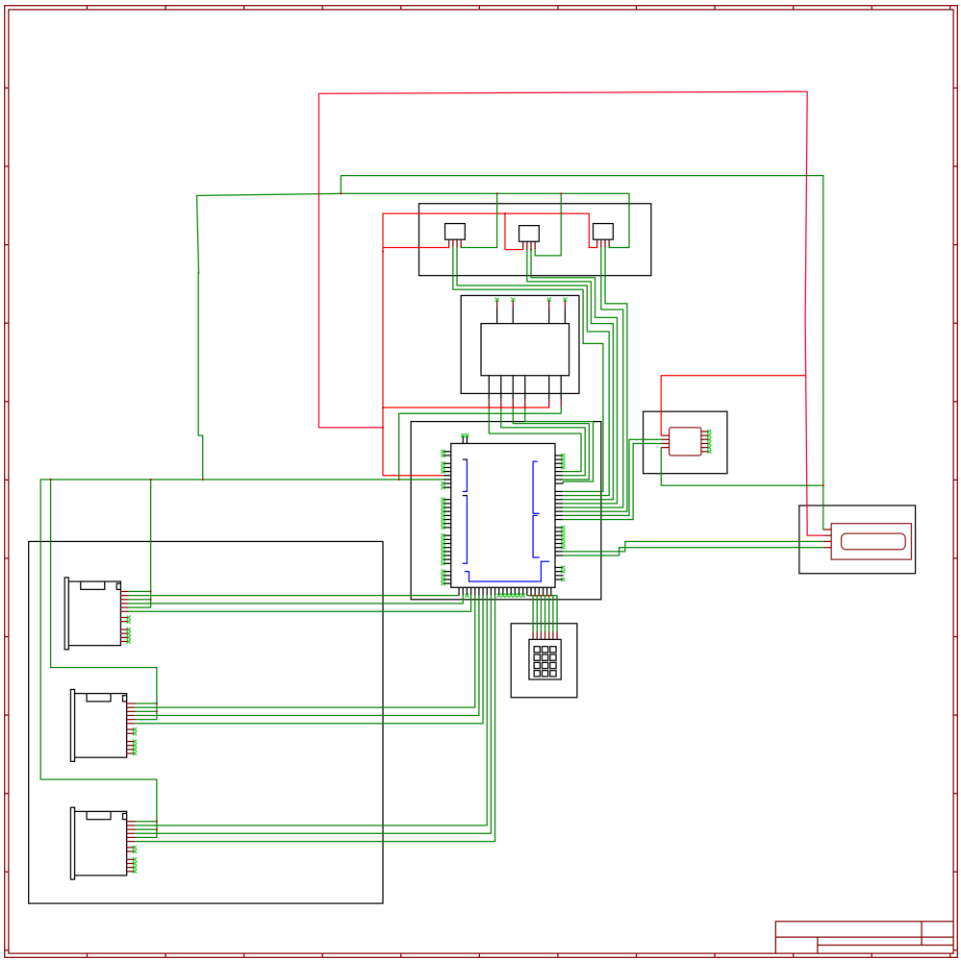


Figure 02: High level design of the product

3D view

Figure 03: A graphical or 3D view of the solution

Structure

Figure 03: Actual Structure

budget allocation

Component	Unit Price	Unit	Total Price
HC-SR04 Ultrasonic Sensor	220.00	1	260
MG995 Servo Motor	990.00	1	990
Wiper Motors	5000.00	3	15000
10kg Load Cell	520.00	1	520
Nema23 Stepper Motors x 3	4000.00	3	12000
4x4 Keypad	180.00	1	180
16x2 LCD Display	380.00	1	380
HX711 Module	260.00	1	260
Arduino Mega	2000.00	1	2000
Nema23 Stepper Motor	1750.00	1	1750
Relay Module	350.00	1	350
TB6600 Motor Driver	2000.00	3	6000
Other	5000.00	1	5000
Total			**43690.00**

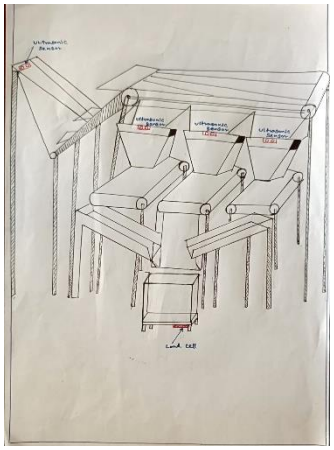
Table 01: Components with budget allocation

Workload

REG.NUMBER	ASSIGNED RESPONSIBILITIES
224017L	• Using an ultrasound sensor, it determines whether the potatoes are in the container or not. • Designing the structure of the machine.
224022X	• Identify the potato level and stop the grading system. • Potato Grading Mechanism. • Design the structure of the machine.
224127A	• After input the weight and size of the potatoes start to measure the weight. • Design the structure of the machine.
224001H	• Stop the stepper motors after measuring the potatoes. • Design the structure of the machine.
224054V	• Get size and weight of the potatoes and display the process. • Design the structure of the machine.

K.N Balasooriya-224017L (Leader)

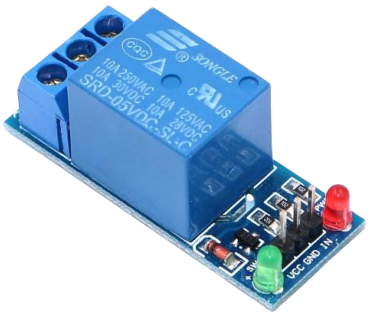
Ultrasonic Sensor & Main Diagram



- Potato level measured by the HC-SR04 ultrasonic sensor.
- Allows the use to check the potato level using the admin panel. Send
- signal to the wiper motor when the level is low or high

W.G.C.W Bandara-224022X

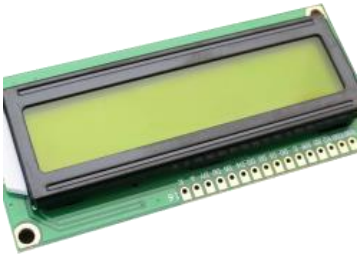
Wiper Motor& Relay Module

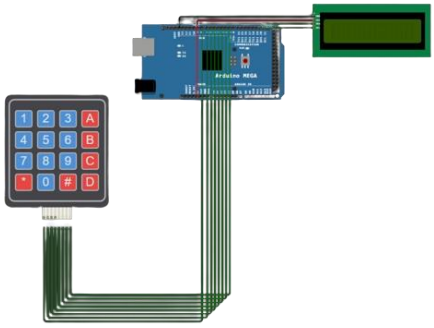


- Used to potato grading mechanism
- Send signal to the wiper motor when the level is low
- process of the wiper motor and ultrasonic sensors

S.A. Gamaarachchi-224054V

Keyboard & Display , Circuit Diagram

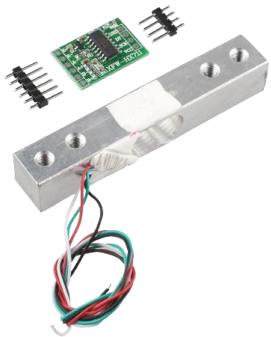


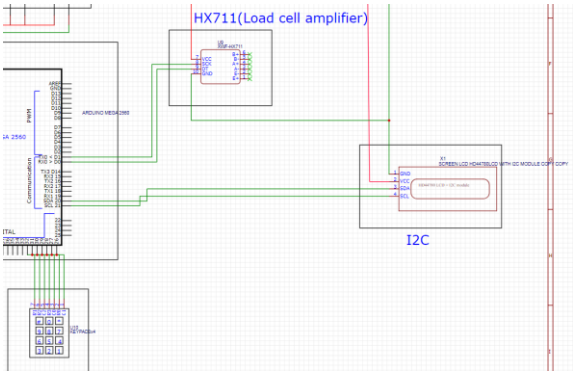


- A load cell is a sensor used to measure weight of potatoes.
- This deformation generates an electrical signal proportional to the force exerted,
- which is then converted into a weight measurement.

M.R.M Muadh-224127A

10kg Load Cell & HX711 Module

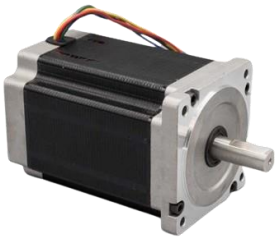




- Input the weight and size of the potatoes start to measure the weight.
- Design the structure of the machine.

M.I. Abdulla-224001H

Stepper Motor and TB6600



- A load cell is a sensor used to measure weight of potatoes.
- This deformation generates an electrical signal proportional to the force exerted, which is then converted into a weight measurement.

Cording Part

```
//*keypad, keyboard and HX711 libraries start #include <Wire.h>
#include <LiquidCrystal_I2C.h> #include <Keypad.h>
#include "HX711.h"

//*keypad, keyboard and HX711 libraries end



//*stepper motor 1 pins start #define PUL_PIN1 10 // Pulse pin
#define DIR_PIN1 11 // Direction pin #define EN_PIN1 12 // Enable pin
//*stepper motor 1 pins end



//*stepper motor 2 pins start #define PUL_PIN2 22 // Pulse pin
#define DIR_PIN2 23 // Direction pin #define EN_PIN2 24 // Enable pin

//*stepper motor 2 pins end



//*stepper motor 3 pins start #define PUL_PIN3 25 // Pulse pin
#define DIR_PIN3 26 // Direction pin #define EN_PIN3 27 // Enable pin
//*stepper motor 3 pins end



//*HX711 pins start #define DOUT1 38
#define CLK1 39

#define DOUT2 40

#define CLK2 41

#define DOUT3 42

#define CLK3 43

//*HX711 pins end
```

```
//*Ultrasonic sensor start const int trigPin1 = 30; const int echoPin1 = 31; const int trigPin2 = 32; const int echoPin2 = 33; const int trigPin3 = 34; const int echoPin3 = 35;
//*Ultrasonic sensor end

//*relay module start

const int relay1 = 46; // IN1 pin of the relay module

//*relay module end


//*display and keypad start

LiquidCrystal_I2C lcd(0x27, 16, 2); // Initialize LCD at address 0x27, 16x2


const byte ROWS = 4; // Four rows const byte COLS = 4; // Three columns

char keys[ROWS][COLS] = {

{ '1', '2', '3', 'A' },

{ '4', '5', '6', 'B' },

{ '7', '8', '9', 'C' },

{ '*', '0', '#', 'D' }

};


byte rowPins[ROWS] = { 5, 4, 3, 2 };

byte colPins[COLS] = { 9, 8, 7, 6 };


Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS);

//*display and keypad end
```

```
//*HX711 start HX711 scale;

float calibration_factor1 = 200000; // Adjust this value for calibration float
calibration_factor2 = 210000; // Adjust this value for calibration float calibration_factor3
= 180000; // Adjust this value for calibration
//*HX711 end



bool displayReady = 0; bool interupted = 0;


void setup() {

//*display and keypad start lcd.init();        // Initialize the LCD
lcd.backlight();                  // Turn on backlight lcd.clear();      // Clear the LCD screen
Serial.begin(9600); // Start serial communication for debugging

//*display and keypad end



//*stepper motor 1 start pinMode(PUL_PIN1, OUTPUT); pinMode(DIR_PIN1, OUTPUT);
pinMode(EN_PIN1, OUTPUT);


// Enable the driver

digitalWrite(EN_PIN1, LOW); // LOW to enable



// Set direction

digitalWrite(DIR_PIN1, LOW); // Set direction to HIGH for one direction

//*stepper motor 1 end



//*stepper motor 2 start pinMode(PUL_PIN2, OUTPUT); pinMode(DIR_PIN2, OUTPUT);
pinMode(EN_PIN2, OUTPUT);


// Enable the driver
```

```
digitalWrite(EN_PIN2, LOW); // LOW to enable


// Set direction

digitalWrite(DIR_PIN2, LOW); // Set direction to HIGH for one direction

//*stepper motor 2 end


//*stepper motor 3 start pinMode(PUL_PIN3, OUTPUT); pinMode(DIR_PIN3, OUTPUT);
pinMode(EN_PIN3, OUTPUT);


// Enable the driver

digitalWrite(EN_PIN3, LOW); // LOW to enable


// Set direction

digitalWrite(DIR_PIN3, LOW); // Set direction to HIGH for one direction

//*stepper motor 3 end

//*Ultrasonic sensor start pinMode(trigPin1, OUTPUT); pinMode(echoPin1, INPUT);
pinMode(trigPin2, OUTPUT); pinMode(echoPin2, INPUT); pinMode(trigPin3, OUTPUT);
pinMode(echoPin3, INPUT);
//*Ultrasonic sensor end


//*relay module start

// Set the relay pin as an output pinMode(relay1, OUTPUT);


// Initialize the relay to be off digitalWrite(relay1, HIGH);
//*relay module end

}
```

```
void loop() { upper_part(); lower_part();
}



//*function to run motor start void runMotor(int pulPin) { digitalWrite(pulPin, HIGH);

delayMicroseconds(833); // Adjust delay for desired speed digitalWrite(pulPin, LOW);
delayMicroseconds(833); // Adjust delay for desired speed

}

//*function to run motor end



//*function for ultrasonic sensor start

long readUltrasonicDistance(int trigPin, int echoPin) { digitalWrite(trigPin, LOW);
delayMicroseconds(2); digitalWrite(trigPin, HIGH); delayMicroseconds(10);
digitalWrite(trigPin, LOW);
long duration = pulseIn(echoPin, HIGH); long distance = duration * 0.034 / 2; return
distance;
}

//*function for ultrasonic sensor end



void lower_part() {



static unsigned long invalidMillis = millis(); static unsigned long processMillis = millis();
static unsigned long weightMillis = millis(); static unsigned long motorMillis = millis();
static unsigned long toleranceMillis = millis();

enum class lower_part_state : uint8_t { IDLE,    // defaults to 0 INTERUPT,
        // defaults to 1 SIZE_SELECT,              // defaults to 2 INVALID,              //
defaults to 3 SIZE_ENTER,                          // defaults to 4 PROCESSING,
                // defaults to 5 SIZE_DELETE,                      // defaults to 6
WEIGHT_SELECT, // defaults to 7 WEIGHT_ENTER,                                          //
defaults to 8 WEIGHT_DELETE, // defaults to 9 CONFIRMATION,
        // defaults to 10 LOAD_CELL_1_ON, // defaults to 11 LOAD_CELL_2_ON, //
```

```
defaults to 12 LOAD_CELL_3_ON, // defaults to 13 MOTOR_1_ON,
                // defaults to 14 MOTOR_2_ON,                    // defaults to
15 MOTOR_3_ON,                    // defaults to 16 MOTOR_1_OFF,
                // defaults to 17 MOTOR_2_OFF,                    // defaults to 18
MOTOR_3_OFF,                    // defaults to 19 PROCESS_END,
                        // defaults to 20
};



// Keep track of the current State

static lower_part_state currState = lower_part_state ::IDLE;



// Keep track of the previous State

static lower_part_state prevState = lower_part_state ::IDLE;



// Variables for user inputs and weight values

static String inputText = ""; // String to store user input

static String motor = ""; // String to store input motor number static float weightValue =
0.0; // float to store input weight value static float weight = 0.0; //float to store current
weight
static long reading = 0; static char key = "";


if (currState == lower_part_state ::IDLE) { displayReady = 1;
} else { displayReady = 0;
}



// Process according to our State Diagram switch (currState) {


// Initial state (or final returned state) case lower_part_state ::IDLE:


if (interupted == 1) {
```

```
currState = lower_part_state ::INTERUPT; break;
}



// Waiting for size input lcd.setCursor(0, 0); lcd.print("Enter size(1/2/3): ");
lcd.setCursor(0, 1);
key = keypad.getKey(); // Read keypad input



if (key != NO_KEY) {

currState = lower_part_state ::SIZE_SELECT; Serial.println("size key pressed");
}

break;



case lower_part_state ::INTERUPT:

key = keypad.getKey(); // Read keypad input interupted = 0;
if (key != NO_KEY) { switch (key) {
case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

case '0':

case '*':
```

```
case '#':

case 'A':

case 'B':

case 'C':

break; case 'D':
lcd.clear(); // Clear the entire display currState = lower_part_state ::IDLE; break;
}

}

break;



case lower_part_state ::SIZE_SELECT: switch (key) {
case '1':

case '2':

case '3':

Serial.println("1/2/3 pressed");

if (inputText.length() < 1) { // Limit to 1 characters

inputText += key;

lcd.print(inputText); // Display the entered digits currState = lower_part_state ::IDLE;
Serial.println("1/2/3 printed");
} else {

lcd.print("not valid"); Serial.println("not valid printed"); invalidMillis = millis();
currState = lower_part_state ::INVALID; prevState = lower_part_state ::SIZE_SELECT;
}

break; case '4':
case '5':

case '6':

case '7':
```

```
case '8':

case '9':

case '0':

case 'A':

case '*':

case '#':

lcd.print("not valid"); invalidMillis = millis();
currState = lower_part_state ::INVALID; prevState = lower_part_state ::SIZE_SELECT;

break; case 'D':
if (inputText != "") {

currState = lower_part_state ::SIZE_ENTER; Serial.println("enter pressed");
} else {

lcd.print("not valid"); invalidMillis = millis();
currState = lower_part_state ::INVALID; prevState = lower_part_state ::SIZE_SELECT;
}

break; case 'C':
currState = lower_part_state ::SIZE_DELETE; break;
case 'B':

currState = lower_part_state ::SIZE_DELETE; break;
default: break;
}

break;



case lower_part_state ::INVALID: if (millis() - invalidMillis >= 1000) {
lcd.clear();     // Clear the entire display

inputText = ""; // Clear the input text lcd.setCursor(0, 0);
if (prevState == lower_part_state ::SIZE_SELECT) { currState = lower_part_state
::IDLE;
} else if (prevState == lower_part_state ::WEIGHT_SELECT) { currState =
lower_part_state ::WEIGHT_SELECT;
} else {
```

```
currState = lower_part_state ::CONFIRMATION;

}

}

break;



case lower_part_state ::SIZE_ENTER: motor = inputText;
lcd.clear();     // Clear the entire display inputText = ""; // Clear the input text
lcd.setCursor(0, 0); lcd.print("Processing...");
processMillis = millis();

currState = lower_part_state ::PROCESSING; prevState = lower_part_state
::SIZE_ENTER; break;


case lower_part_state ::PROCESSING: if (millis() - processMillis >= 1000) { lcd.clear();
// Clear the entire display

lcd.setCursor(0, 0);

if (prevState == lower_part_state ::SIZE_ENTER) { currState = lower_part_state
::WEIGHT_SELECT;
} else if (prevState == lower_part_state ::WEIGHT_ENTER) { currState =
lower_part_state ::CONFIRMATION;
} else {

if (motor == "1") {

currState = lower_part_state ::LOAD_CELL_1_ON;

} else if (motor == "2") {

currState = lower_part_state ::LOAD_CELL_2_ON;

} else {

currState = lower_part_state ::LOAD_CELL_3_ON;

}

}
```

```
}

break;



case lower_part_state ::SIZE_DELETE: lcd.clear();      // Clear the entire display
inputText = ""; // Clear the input text lcd.setCursor(0, 0);
currState = lower_part_state ::IDLE; break;


case lower_part_state ::WEIGHT_SELECT:

// Waiting for weight input

lcd.setCursor(0, 0); lcd.print("Enter weight(kg): "); lcd.setCursor(0, 1);
key = keypad.getKey(); // Read keypad input



if (key != NO_KEY) { switch (key) {
case '1':

case '2':

case '3':

case '4':

if (inputText.length() < 1) { // Limit to 1 characters inputText += key;
lcd.print(inputText); // Display the entered digits

} else {

lcd.print("not valid"); invalidMillis = millis();
currState = lower_part_state ::INVALID;

prevState = lower_part_state ::WEIGHT_SELECT;

}

break; case '5':
case '6':

case '7':
```

```
case '8':

case '9':

case '0':

case 'A':

case '*':

case '#':

lcd.print("not valid"); invalidMillis = millis();
currState = lower_part_state ::INVALID;

prevState = lower_part_state ::WEIGHT_SELECT; break;
case 'D':

currState = lower_part_state ::WEIGHT_ENTER; prevState = lower_part_state
::WEIGHT_SELECT; break;
case 'C':

currState = lower_part_state ::WEIGHT_DELETE; prevState = lower_part_state
::WEIGHT_SELECT; break;
case 'B':

currState = lower_part_state ::SIZE_DELETE; break;
default: break;
}

}

break;

case lower_part_state ::WEIGHT_ENTER: Serial.print("inputText: ");
Serial.println(inputText);
weightValue = inputText.toFloat() * 100; Serial.print("weight value: ");
Serial.print(weightValue); Serial.println();
lcd.clear();     // Clear the entire display inputText = ""; // Clear the input text
lcd.setCursor(0, 0); lcd.print("Processing...");
processMillis = millis();

currState = lower_part_state ::PROCESSING; prevState = lower_part_state
::WEIGHT_ENTER; break;
```

```
case lower_part_state ::WEIGHT_DELETE: lcd.clear();          // Clear the entire display
inputText = ""; // Clear the input text lcd.setCursor(0, 0);
currState = lower_part_state ::WEIGHT_SELECT; break;


case lower_part_state ::CONFIRMATION:

// Waiting for weight input lcd.setCursor(0, 0);

lcd.print("size: "); lcd.print(motor); lcd.setCursor(0, 1); lcd.print("weight: ");
lcd.print(weightValue);
key = keypad.getKey(); // Read keypad input



if (key != NO_KEY) { switch (key) {
case 'D':

lcd.clear();     // Clear the entire display inputText = ""; // Clear the input text
lcd.setCursor(0, 0); lcd.print("Processing...");
processMillis = millis();

currState = lower_part_state ::PROCESSING; prevState = lower_part_state
::CONFIRMATION; break;
case 'B':

currState = lower_part_state ::SIZE_DELETE; break;
case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':
```

```
case '0':

case 'A':

case 'c':

case '*':

case '#':

lcd.print("not valid"); invalidMillis = millis();
currState = lower_part_state ::INVALID; prevState = lower_part_state
::CONFIRMATION; break;
default: break;
}

}

break;



case lower_part_state ::LOAD_CELL_1_ON:

//*HX711 start

Serial.println("HX711 calibration sketch"); Serial.println("Remove all weight from scale");
Serial.println("After readings begin, place known weight on scale"); Serial.println("Press
+ or a to increase calibration factor");

Serial.println("Press - or z to decrease calibration factor");



scale.begin(DOUT1, CLK1); scale.set_scale(calibration_factor1);
//scale.tare(); // Reset the scale to 0

static long zero_factor1 = scale.read_average(); // Get a baseline reading

// Set the scale to this zero factor scale.set_offset(zero_factor1); // Zero out the scale
Serial.print("Zero factor: "); // This can be used to remove the need to tare the scale.
Useful in permanent scale projects.

Serial.println(zero_factor1); weightMillis = millis(); motorMillis = millis();
```

```
lcd.setCursor(0, 0); lcd.print("       wait"); lcd.setCursor(0, 1); lcd.print("weighting began");
currState = lower_part_state ::MOTOR_1_ON; Serial.println("motor 1 selected");
break;

//*HX711 end



case lower_part_state ::LOAD_CELL_2_ON:

//*HX711 start

Serial.println("HX711 calibration sketch");

Serial.println("Remove all weight from scale");

Serial.println("After readings begin, place known weight on scale"); Serial.println("Press + or a to increase calibration factor"); Serial.println("Press - or z to decrease calibration factor");


scale.begin(DOUT2, CLK2); scale.set_scale(calibration_factor2);
//scale.tare(); // Reset the scale to 0

static long zero_factor2 = scale.read_average(); // Get a baseline reading

// Set the scale to this zero factor scale.set_offset(zero_factor2); // Zero out the scale Serial.print("Zero factor: "); // This can be used to remove the need to tare the scale. Useful in permanent scale projects.

Serial.println(zero_factor2); weightMillis = millis(); motorMillis = millis();


lcd.setCursor(0, 0); lcd.print("       wait"); lcd.setCursor(0, 1); lcd.print("weighting began");
currState = lower_part_state ::MOTOR_2_ON; Serial.println("motor 2 selected");
break;

//*HX711 end

case lower_part_state ::LOAD_CELL_3_ON:

//*HX711 start

Serial.println("HX711 calibration sketch"); Serial.println("Remove all weight from scale");
```

```
Serial.println("After readings begin, place known weight on scale"); Serial.println("Press + or a to increase calibration factor"); Serial.println("Press - or z to decrease calibration factor");


scale.begin(DOUT3, CLK3); scale.set_scale(calibration_factor3);
//scale.tare(); // Reset the scale to 0

static long zero_factor3 = scale.read_average(); // Get a baseline reading

// Set the scale to this zero factor scale.set_offset(zero_factor3); // Zero out the scale
Serial.print("Zero factor: "); // This can be used to remove the need to tare the scale. Useful in permanent scale projects.

Serial.println(zero_factor3); weightMillis = millis(); motorMillis = millis();


lcd.setCursor(0, 0); lcd.print("       wait"); lcd.setCursor(0, 1); lcd.print("weighting began");
currState = lower_part_state ::MOTOR_3_ON; Serial.println("motor 3 selected");

break;

//*HX711 end



case lower_part_state ::MOTOR_1_ON:

//*HX711 start

if (millis() - weightMillis >= 1000) {

scale.set_scale(calibration_factor1); // Adjust to this calibration factor weight = scale.get_units();


if (weight < 0) { weight = 0.00;
}

Serial.print("Reading: "); Serial.print("Grams: ");
weight = weight * 1000; // Convert kg to grams Serial.print(weight);
Serial.print(" g");

Serial.print(" calibration_factor: "); Serial.print(calibration_factor1); reading = scale.read(); Serial.print(" Raw reading: "); Serial.print(reading); Serial.println();
```

```
weightMillis = millis();

}

//*HX711 end

//*motor 1 start

if (millis() - motorMillis >= 2000) { runMotor(PUL_PIN1);
}

//*motor 1 end



if (millis() - motorMillis >= 2500) { if (weight >= weightValue) {
lcd.clear(); // Clear the entire display

currState = lower_part_state ::MOTOR_1_OFF; toleranceMillis = millis();
Serial.println("moving to motor 1 off");

}

}

break;



case lower_part_state ::MOTOR_2_ON:

//*HX711 start

if (millis() - weightMillis >= 1000) {

scale.set_scale(calibration_factor2); // Adjust to this calibration factor weight =
scale.get_units();


if (weight < 0) { weight = 0.00;
}

Serial.print("Reading: "); Serial.print("Grams: ");
weight = weight * 1000; // Convert kg to grams Serial.print(weight);
Serial.print(" g");
```

```
Serial.print(" calibration_factor: "); Serial.print(calibration_factor2); Serial.print("weight
value: "); Serial.print(weightValue); Serial.println();
weightMillis = millis();

}

//*HX711 end



//*motor 2 start

if (millis() - motorMillis >= 2000) { runMotor(PUL_PIN2);
}

//*motor 2 end



if (millis() - motorMillis >= 2500) { if (weight >= weightValue) {
lcd.clear(); // Clear the entire display

currState = lower_part_state ::MOTOR_2_OFF; toleranceMillis = millis();
Serial.println("moving to motor 2 off");

}

}

break;



case lower_part_state ::MOTOR_3_ON:

//*HX711 start

if (millis() - weightMillis >= 1000) {

scale.set_scale(calibration_factor3); // Adjust to this calibration factor weight =
scale.get_units();


if (weight < 0) { weight = 0.00;
}
```

```
Serial.print("Reading: "); Serial.print("Grams: ");
weight = weight * 1000; // Convert kg to grams Serial.print(weight);
Serial.print(" g");

Serial.print(" calibration_factor: "); Serial.print(calibration_factor3); Serial.print("weight
value: "); Serial.print(weightValue); Serial.println();
weightMillis = millis();

}

//*HX711 end

//*motor 3 start

if (millis() - motorMillis >= 2000) { runMotor(PUL_PIN3);
}

//*motor 3 end



if (millis() - motorMillis >= 2500) { if (weight >= weightValue) {
lcd.clear(); // Clear the entire display

currState = lower_part_state ::MOTOR_3_OFF; toleranceMillis = millis();
Serial.println("moving to motor 3 off");

}

}

break;



case lower_part_state ::MOTOR_1_OFF: digitalWrite(EN_PIN1, HIGH);
if (millis() - toleranceMillis >= 1000) { if (weight >= weightValue) {
currState = lower_part_state ::PROCESS_END; Serial.println("moving to process end");
} else {

currState = lower_part_state ::MOTOR_1_ON;

}

}
```

```
break;



case lower_part_state ::MOTOR_2_OFF: digitalWrite(EN_PIN2, HIGH);
if (millis() - toleranceMillis >= 1000) { if (weight >= weightValue) {
currState = lower_part_state ::PROCESS_END; Serial.println("moving to process end");
} else {

currState = lower_part_state ::MOTOR_2_ON;

}

}

break;



case lower_part_state ::MOTOR_3_OFF: digitalWrite(EN_PIN3, HIGH);
if (millis() - toleranceMillis >= 1000) { if (weight >= weightValue) {
currState = lower_part_state ::PROCESS_END; Serial.println("moving to process end");
} else {

currState = lower_part_state ::MOTOR_3_ON;

}

}

break;

case lower_part_state ::PROCESS_END:

if (millis() - motorMillis >= 1000) { Serial.println("precess end"); motorMillis = millis();
}

lcd.setCursor(0, 0); lcd.print(" please remove"); lcd.setCursor(0, 1); lcd.print("
        weight");
//*HX711 start if (motor == 1) {
if (millis() - weightMillis >= 1000) {

scale.set_scale(calibration_factor1); // Adjust to this calibration factor weight =
scale.get_units();
```

```
if (weight < 0) { weight = 0.00;
}

Serial.print("Reading: "); Serial.print("Grams: ");
weight = weight * 1000; // Convert kg to grams Serial.print(weight);
Serial.print(" g");

Serial.print(" calibration_factor: "); Serial.print(calibration_factor1);

Serial.print("weight value: "); Serial.print(weightValue); Serial.println();
weightMillis = millis();

}

} else if (motor == 2) {

if (millis() - weightMillis >= 1000) {

scale.set_scale(calibration_factor2); // Adjust to this calibration factor weight =
scale.get_units();


if (weight < 0) { weight = 0.00;
}

Serial.print("Reading: "); Serial.print("Grams: ");
weight = weight * 1000; // Convert kg to grams Serial.print(weight);
Serial.print(" g");

Serial.print(" calibration_factor: "); Serial.print(calibration_factor2); Serial.print("weight
value: "); Serial.print(weightValue); Serial.println();
weightMillis = millis();

}

} else {

if (millis() - weightMillis >= 1000) {

scale.set_scale(calibration_factor3); // Adjust to this calibration factor weight =
scale.get_units();


if (weight < 0) { weight = 0.00;
}
```

```
Serial.print("Reading: "); Serial.print("Grams: ");
weight = weight * 1000; // Convert kg to grams Serial.print(weight);
Serial.print(" g");

Serial.print(" calibration_factor: "); Serial.print(calibration_factor3); Serial.print("weight
value: "); Serial.print(weightValue); Serial.println();
weightMillis = millis();

}

}

//*HX711 end

if (weight <= 10.0) { Serial.println("restart");
lcd.clear(); // Clear the entire display currState = lower_part_state ::IDLE;
}

digitalWrite(EN_PIN1, LOW); // LOW to enable digitalWrite(EN_PIN2, LOW); // LOW to
enable digitalWrite(EN_PIN3, LOW); // LOW to enable break;


default:

Serial.println("'Default' Switch Case reached - Error");

}

}



void upper_part() {



static unsigned long sensor1Millis = millis(); static unsigned long sensor2Millis = millis();
static unsigned long sensor3Millis = millis();


enum class upper_part_state : uint8_t { NORMAL,       // defaults to 0
CONTAINER_1_FULL, // defaults to 1 CONTAINER_2_FULL, // defaults to 2
CONTAINER_3_FULL, // defaults to 3 CONTAINER_1_EMPTY, // defaults to 4
CONTAINER_2_EMPTY, // defaults to 5 CONTAINER_3_EMPTY, // defaults to 6
};
```

```
// Keep track of the current State

static upper_part_state currState = upper_part_state ::NORMAL;



long distance1 = readUltrasonicDistance(trigPin1, echoPin1); long distance2 =
readUltrasonicDistance(trigPin2, echoPin2); long distance3 =
readUltrasonicDistance(trigPin3, echoPin3);


switch (currState) {



case upper_part_state ::NORMAL: if (millis() - sensor1Millis >= 1000) {
Serial.print("Sensor 1: "); Serial.print(distance1); Serial.print(" cm, ");
sensor1Millis = millis();

}



if (millis() - sensor2Millis >= 1000) { Serial.print("Sensor 2: "); Serial.print(distance2);
Serial.print(" cm, ");
sensor2Millis = millis();

}



if (millis() - sensor3Millis >= 1000) { Serial.print("Sensor 3: ");

Serial.print(distance3); Serial.println(" cm"); sensor3Millis = millis();
}

// Check if sensor 1 reads below 10 cm if (distance1 < 10) {
currState = upper_part_state ::CONTAINER_1_FULL;

}



// Check if sensor 2 reads below 10 cm if (distance2 < 10) {
currState = upper_part_state ::CONTAINER_2_FULL;
```

```
}


// Check if sensor 3 reads below 10 cm if (distance3 < 10) {
currState = upper_part_state ::CONTAINER_3_FULL;

}

break;


case upper_part_state ::CONTAINER_1_FULL:

// Stop the motor digitalWrite(relay1, LOW); Serial.println("Motor 1 stopped");
interupted = 1;

if (displayReady == 1) { lcd.clear(); lcd.setCursor(0, 0);
lcd.print(" container 1");

lcd.setCursor(0, 1); lcd.print("       is full");
currState = upper_part_state ::CONTAINER_1_EMPTY;

}

break;


case upper_part_state ::CONTAINER_2_FULL:

// Stop the motor digitalWrite(relay1, LOW); Serial.println("Motor 2 stopped");
interupted = 1;


if (displayReady == 1) { lcd.clear(); lcd.setCursor(0, 0);
lcd.print(" container 2");

lcd.setCursor(0, 1); lcd.print("       is full");
currState = upper_part_state ::CONTAINER_2_EMPTY;

}

break;
```

```
case upper_part_state ::CONTAINER_3_FULL:

// Stop the motor digitalWrite(relay1, LOW); Serial.println("Motor 3 stopped");
interupted = 1;



if (displayReady == 1) { lcd.clear(); lcd.setCursor(0, 0);
lcd.print(" container 3");

lcd.setCursor(0, 1); lcd.print("       is full");
currState = upper_part_state ::CONTAINER_3_EMPTY;

}

break;



case upper_part_state ::CONTAINER_1_EMPTY: if (millis() - sensor1Millis >= 1000) {
Serial.print("Sensor 1: ");
Serial.print(distance1); Serial.println(" cm, "); sensor1Millis = millis();
}



// Check if sensor 1 reads below 10 cm if (distance1 > 25) {

digitalWrite(relay1, HIGH);

currState = upper_part_state ::NORMAL;

}

break;



case upper_part_state ::CONTAINER_2_EMPTY: if (millis() - sensor2Millis >= 1000) {
Serial.print("Sensor 2: ");
Serial.print(distance2); Serial.println(" cm, "); sensor2Millis = millis();
}
```

```
// Check if sensor 1 reads below 10 cm if (distance2 > 25) {
digitalWrite(relay1, HIGH);

currState = upper_part_state ::NORMAL;

}

break;



case upper_part_state ::CONTAINER_3_EMPTY: if (millis() - sensor3Millis >= 1000) {
Serial.print("Sensor 3: ");
Serial.print(distance3); Serial.println(" cm, "); sensor3Millis = millis();

}



// Check if sensor 1 reads below 10 cm if (distance3 > 25) {
digitalWrite(relay1, HIGH);

currState = upper_part_state ::NORMAL;

}

break;



default:

Serial.println("'Default' Switch Case reached - Error");

}

}
```