Department of Mechanical Engineering

MECH 458: Mechatronics

Final Project Report

Due Date: December 17, 2024

Carmina Rocheleau, V01021553

Jesse Bertucci, V01007526

Date: December 17, 2024

Lab Section B04 and Group 5

Table of Contents

List of Figures

List of Tables

I. INTRODUCTION

I.A Statement of the Problem

Efficient and accurate sorting systems are vital in modern industries, particularly in applications requiring the classification of items based on specific characteristics. Traditional methods often involve labor-intensive processes or reliance on expensive and specialized equipment. These approaches are either time-consuming or cost-prohibitive, limiting their scalability for smaller-scale operations. The challenge lies in developing a cost-effective, automated solution capable of high-speed and precise sorting while overcoming constraints such as limited processing power, mechanical complexity, and integration of hardware and software components.

I.B Purpose of the design and design overview

The purpose of this project is to design and implement an automated sorting system that addresses these challenges by utilizing a microcontroller-based architecture. The system inspects and classifies objects on a conveyor belt based on their reflective and positional characteristics, ensuring efficiency and accuracy within a 60-second sorting window.

The design features a conveyor belt driven by a DC motor that transports items through inspection stations. At Station S1, a reflective sensor measures the visual reflectivity of each item, while Station S2 signals when an item reaches the sorting point. A stepper motor-controlled sorting tray then rotates to direct items into one of four designated bins. The system employs a trapezoidal acceleration profile for the stepper motor to reduce mechanical strain and optimize operation. Interrupt-driven sensor signals ensure real-time classification without disrupting the conveyor's movement.

By integrating hardware and software components seamlessly, using modular programming practices, and optimizing algorithms for speed and accuracy, the project demonstrates the feasibility of microcontroller-based automation in high-performance sorting applications. This system not only eliminates the need for ferromagnetic and OI sensors but also serves as a practical and scalable solution for industries requiring high-speed, high-accuracy sorting systems.

II. METHODS AND DESIGN APPROACH

II.A Method Overview

The inspection and sorting system was successfully completed and demonstrated by December 03, 2024, following a structured four-week timeline. Including the project report the timeline started November 1st, 2024 and finished December 17, 2024 providing a six-week timeline. This timeline broke down the project into manageable phases, allowing the team to progress methodically through design, assembly, coding, and testing. Each week builds on the previous one, moving from initial setup and hardware integration to advanced software implementation and final system testing. Milestones at the end of each phase kept the project on track, with sufficient time allocated for troubleshooting and optimization. This approach delivered a fully operational and reliable system for demonstration.

Table 1: Task to Week Overview

Step	Task	Week
1	Initial Setup and Planning	1
2	System Assembly, code structure setup, interrupts, and pin allocation	1,2
3	Reflective sensor (RL) and optical sensor (OR) integration	2

4	Code for RL, OR interrupts, and object identification cases	2
5	Optical sensor (EX) and end-of-belt integration	3
6	Hall-effect sensor (HE) and stepper motor initialization	3
7	Full integration of all components	3
8	Trapezoid integration	3
9	Push button setup and LCD code and set up	4
10	Optimization	4
11	Final Testing	4
12	Demonstration	5
13	Documentation	5

II.A.1 Step Break down

II.A.1.a Initial Setup and Planning

The final project commenced on November 1, 2024. First, the project specification document and marking scheme were thoroughly examined and key steps were highlighted. The code from all previous labs was reviewed and the provided skeleton code was inspected. Finally, a project schedule gantt chart was created to divide the workload and set deliverable deadlines. The gantt chart can be seen in Appendix A.

II.A.1.b System Assembly, code structure setup, interrupts, and pin allocation

Functions from previous labs used to control the timer, initialize the stepper motor, initialize the DC motor, initialize the PWM, control the stepper motor, and manipulate the linked list were pulled into the skeleton code main file.

Code was written to initialize four interrupt service routines, the optical sensor interrupt (OR) used to detect objects in front of the reflective sensor, the optical sensor interrupt (EX) used to detect objects at the end of the conveyor belt, the user button interrupt used to start and stop the belt, and the Hall Effect sensor interrupt (HE) used to initialize the stepper motor position. The physical position of these sensors can be seen in Figure # below.

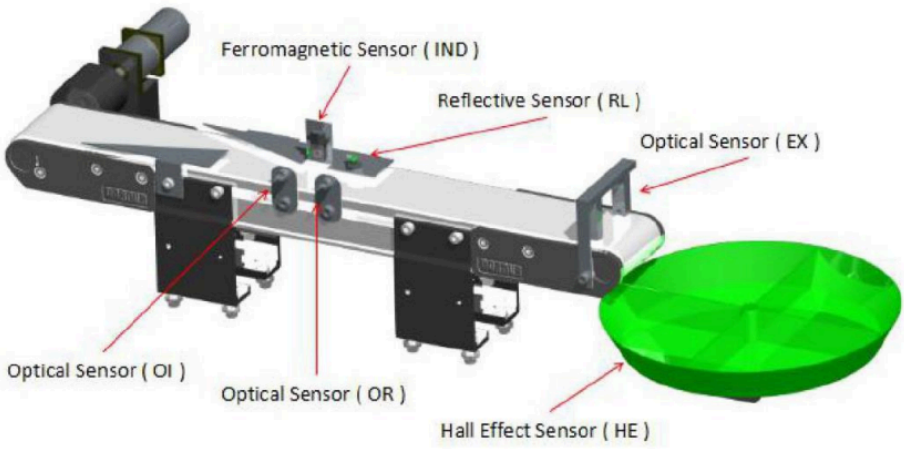


Figure 1: Sensor Positions [1]

I/O pins on the Arduino Mega 2560 were allocated to receive from all sensors and output to motors. These allocations can be seen in circuit wiring schematic found in Appendix B.

II.A.1.c Reflective sensor (RL) and optical sensor (OR) integration

The sensor triggering modes can be seen in Figure 2 below.

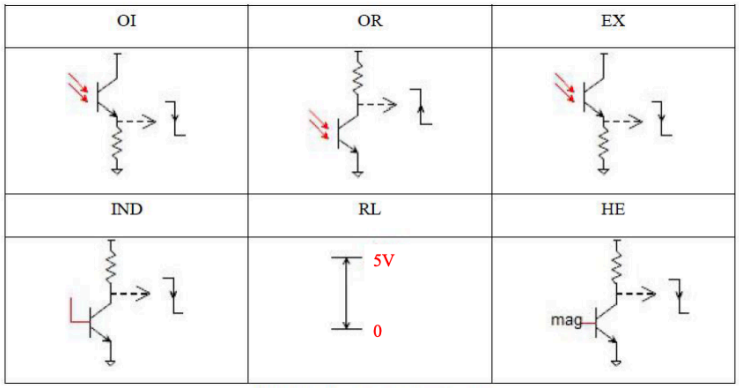


Figure 2: Sensor Trigger Modes [1]

The optical sensor (OR) is used to detect if an object is in front of the reflective sensor (RL). A rising edge interrupt on the OR pin begins a repeating ADC conversion process reading multiple values from the RL sensor and determining the lowest value. While the OR sensor is HIGH the reflective sensor continues to take measurements and the ADC converts each of those measurements. This whole process takes place within the REFLECTIVE_STAGE of the main function.

The object is assigned an object identification code depending on the lowest value read from the RL and ADC. The object codes can be seen in Table 1 below.

Table 2: Object Identification Codes

Object	Code
Black Plastic	0
Aluminum	1
White Plastic	2
Steel	3

Finally, the determined object code is enqueued into the linked list.

II.A.1.d Code for RL, OR interrupts, and object identification cases

As an object approaches the reflective sensor a rising edge OR interrupt is triggered. The code below details how this is handled. First, the ISR checks if an object is still present at the sensor using a mask within an ‘if’ statement. This feature helps protect the ISR from external interference from the motors. Once it has been confirmed that an object is at the OR sensor, the ADSC bit within the ADCSRA register is set, starting an ADC conversion. Next, the state variable which directs the program is set to move to the reflective state. Finally, the variable that tracks the lowest recorded ADC value is reset to its highest possible value.

```
// optical sensor OR
ISR(INT0_vect)
{
   if((PIND & 0x01) == 0x01)
   {
       ADCSRA |= _BV(ADSC); // start conversion
       STATE = 2; // go to reflective state after this interrupt is done
       lowestADCvalue = 1023; // reinitializing lowest 10-bit value to highest possible
value
   }
}
```

Setting the ADSC bit in the ADCSRA register triggers and ADC IRQ. Here the value from the ADC is set to a global variable and the result flag is set. The result flag is used in the reflective stage to ensure that the value obtained from the ADC is valid.

```
// ADC end of interrupt
ISR(ADC_vect)
{
   ADC_result = ADC;
   ADC_result_flag = 1;
}
```

In the reflective stage, while the object that triggered the OR interrupt is in view of the reflective sensor and the ADC has provided a valid value, the ADC result is continually updated by resetting the ADC result flag and resetting the ADSC bit in the ADCSRA register. The lowest value obtained from the ADC is continually updated if the current value is lower than the previous value. A new link is initialized in preparation for the object type to be queued. Four ‘if’, ‘if else’, and ‘else’ statements are used to determine the object type based on the lowest recorded ADC value. Finally, the object is enqueued, the global variable used to track the number of objects on the belt is updated, and state is set to direct the program back to the polling stage.

```
   /* shows that an object has been detected by the optical sensor and than is given a
   value by the based of the of the reflective sensor. these values are than categorized and
placed into a q
    */
   REFLECTIVE_STAGE:
   {
       while(((PIND & 0x01)) == 0x01) // while optical sensor is high
       {
           if(ADC_result_flag == 1) // checks if the ADC has been triggered and a value had
been read
           {
               if(ADC_result < lowestADCvalue)  // if the value you of ADC is lower than
previous value it will reset the lowest value you and store it. this value you is given by
the reflective sensor and indicated what type of material we have
               {
                   lowestADCvalue = ADC_result;
               }

               ADC_result_flag =  0x00; // re sets the flag
               ADCSRA |= _BV(ADSC); // start conversion analog to digital
           }
       }

       // Bl = 0
       // Al = 1
       // Wt = 2
       // St = 3

       initLink(&newLink); // Initialize new link "gives new box"
       /*these if statements allow us to categorize the material based on measured
reflective values */
       if(lowestADCvalue < 200) // this is Al
```

```
        {
            newLink->e.itemCode = 1; // this adds a int value from the tail pointer in FIFO
        }
        else if( (lowestADCvalue > 200) && (lowestADCvalue < 700) ) // this is steel
        {
            newLink->e.itemCode = 3; // this adds a int value from the tail pointer in FIFO
        }
        else if( (lowestADCvalue >= 700) && (lowestADCvalue < 915) ) // this is white
        {
            newLink->e.itemCode = 2; // this adds a int value from the tail pointer in FIFO
        }
        else // this is black
        {
            newLink->e.itemCode = 0; // this adds a int value from the tail pointer in FIFO
        }

        enqueue(&head, &tail, &newLink); // this passes values to the FIFO
        sizeOfQueue=size(&head, &tail);
        STATE = 0;
        goto POLLING_STAGE;
    }
```

II.A.1.e Optical sensor (EX) and end-of-belt integration

Once an object is detected at the end of the belt by the EX sensor using a falling edge interrupt the EX ISR is triggered. The code below shows the steps taken within this ISR. First, the ISR checks if an object is still present at the sensor using a mask within an ‘if’ statement. Next, an EX flag is set, the purpose of this flag will be discussed later. The belt is stopped by braking and a global variable tracking the belts movement is updated. The program is then directed to the end of belt stage by setting the state.

```
// optical sensor EX
ISR(INT1_vect)
{
        if((PIND & 0x02) == 0x00)
        {
            EX_flag = 1; // used to track EX interrupts globally
            PORTB |= (1 << PB0) | (1 << PB1); // brake
            stopGo = 0; // belt is now stopped
            STATE = 1; // goto end END_OF_BELT_STAGE
        }
```

```
}
```

The first action taken in the end of belt stage is to reset the EX. This variable is used late to determine if a second EX interrupt has been triggered while in the end of belt stage. The object is dequeued and its object code is extracted. The global variable tracking the number of objects on the belt is updated. Concluding the queue tasks, the link is freed. Four ‘if’, ‘if else’, and ‘else’ statements update global variable that track the number of sorted object in the bucket. An algorithm is then used to determine how the bucket is to be moved to match the object type. More on this algorithm later in the report. The position of the bucket is updated in a global variable, the belt is restarted, and a global variable tracking the belt movements is updated. A delay is used to account for the time it takes for the object to fall from the belt to the bucket, ensuring that the bucket does not move again until the object has securely landed.

If the user has triggered the stop button then the program will wait one second and then direct itself to stop stage with by setting the state variable. If another EX interrupt has been triggered while in the end of belt stage, the state will be updated to move back to the beginning of the end of belt stage with the global state variable. If no other EX interrupt has been triggered while in the end of belt stage then the program returns to the polling stage.

```
//identify objects when EX sensor is triggered buy pulling objects form the FIFO
END_OF_BELT_STAGE:
{
    EX_flag = 0;

    dequeue(&head, &tail, &rtnLink); // remove item from FIFO
    unsigned int objectType = rtnLink->e.itemCode; // stores value pulled from FIDO
    sizeOfQueue=size(&head, &tail);
    free(rtnLink); // free memory location

    if(objectType == 0)
    {
        black_inBucket++;
```

```
        }
        else if(objectType == 1)
        {
            aluminium_inBucket++;
        }
        else if(objectType == 2)
        {
            white_inBucket++;
        }
        else
        {
            steel_inBucket++;
        }

         int targetPosition = objectType; // pulling object form the q

              // modular math to remove negative values and return a value that can be easily
sorted into what direction it should turn.
               int turn = (targetPosition - currentPosition + 4) % 4;
              if (turn == 0)
               {
                      // Do nothing
               }
               else if (turn <= 2)          {
                // move same direction already going by 50 or 100 steps
                      trapezoidalMove( (turn * 50), 1);
                }
               else
               {
                    //move opposite direction by 50 steps
                      trapezoidalMove( ((4 - turn) * 50), 0);
                      }

        currentPosition = targetPosition;
        PORTB = 0b00001110;    // start belt in CW direction
        mTimer(120);
        stopGo = 1; // belt is now running

        if((stop_flag == 1) && (sizeOfQueue == 0))
        {
            mTimer(1000);
            STATE = 4; // END STAGE
        }
        else
        {
            if(EX_flag == 1) // If another EX interupt is triggered
            {
                STATE = 1; // END_OF_BELT STAGE
            }
```

```
            else
            {
                STATE = 0; // POLLING_STAGE
            }
        }

        goto POLLING_STAGE;
    }
```

II.A.1.f Hall-effect sensor (HE) and stepper motor initialization

The main function calls a function to initialize the stepper motor before entering the polling stage. Here a home flag is initialized to zero. This is used to track if the hall effect sensor in the bucket has triggered an interrupt indicating that the bucket has been homed correctly. While waiting for that interrupt, the stepper motor is incremented by one step. Once home has been detected by the changing of the home flag the stepper motor stops and is then moved once more as a calibration to center the bucket with the location that an object will fall. The interrupt for the hall effect sensor is then disabled. The code below details this process.

```
/* Initializes current position */
void initializeStepper()
{

    home_flag = 0;
    while(home_flag == 0)
    {
        stepCW(1, 20); // move motor to set next position
    }

    stepCCW(10, 20); // adjust home for station

    EIMSK &= ~(_BV(INT3)); // disable INT3 external interrupt mask register for HE sensor

    return;
}
```

As previously explained, the bucket is moving by steps of one waiting for the home flag to be updated. The hall effect sensor detects home and triggers a rising edge interrupt. This ISR sets the home flag and updates the current position global variable used to track bucket position.

```
// optical sensor HE
ISR(INT3_vect)
{
   home_flag = 1;
   currentPosition = 0;
}
```

II.A.1.g Full integration of all components

All components are integrated together with the switch statement used to direct the program to certain blocks depending on interrupt action. This code can be seen below.

```
   // POLLING STATE is default state and contains the different cases that can be called
within our program
   POLLING_STAGE:
   {
       switch(STATE)
       {
           case (0) :
           goto POLLING_STAGE;
           break;
           case (1) :
           goto END_OF_BELT_STAGE;
           break;
           case (2) :
           goto REFLECTIVE_STAGE;
           break;
           case (3) :
           goto PAUSE_STAGE;
           break;
           case (4) :
           goto END;
           default :
           goto POLLING_STAGE;
       }
   }
```

II.A.1.h Trapezoid integration

The logic of the trapezoidal acceleration curve is described in great detail in the Design Logic section. The code below shows the function in its entirety. The function has four parts, the first part contains variables used for acceleration calculations, the second part is the acceleration phase, the third part is the constant speed phase, and the final part is the deceleration phase.

```
void trapezoidalMove(int steps, int direction)
{
   int accelSteps = steps / 8; // acceleration steps
   int decelSteps = steps / 8; // deceleration steps
   int constSpeedSteps = steps - (accelSteps + decelSteps); // constant speed steps

   int maximumDelayConst = 20;
   int maximumDelay = maximumDelayConst; // slowest speed
   int minimumDelay = 8; // fastest speed
   float accelDelayIncrements = (float)(maximumDelay - minimumDelay) / accelSteps;
   float decelDelayIncrements = (float)(maximumDelay - minimumDelay) / decelSteps;

   // Acceleration Phase
   for (int i = 0; i < accelSteps; i++)
   {
       if (direction == 1)
       {
           stepCW(1, maximumDelay);
       }
       else
       {
           stepCCW(1, maximumDelay);
       }

       maximumDelay -= (int)accelDelayIncrements;
       if(maximumDelay < minimumDelay)
       {
           maximumDelay = minimumDelay;
       }
   }

   // constant speed
   if (direction == 1)
   {
       stepCW(constSpeedSteps, minimumDelay);
   }
   else
```

```
    {
        stepCCW(constSpeedSteps, minimumDelay);
    }

    // Deceleration Phase
    for (int i = 0; i < decelSteps; i++)
    {
        if (direction == 1)
        {
            stepCW(1, maximumDelay);
        }
        else
        {
            stepCCW(1, maximumDelay);
        }

        maximumDelay += (int)decelDelayIncrements;  // Increase delay to reduce speed
        if(maximumDelay > maximumDelayConst)
        {
            maximumDelay = maximumDelayConst;
        }
    }
}
```

II.A.1.i Push button setup and LCD code

The user can stop and start the program by activating a push button. This triggers an interrupt. The start and stop ISR contains debouncing code. Within that debouncing code the program is directed to move the pause stage.

```
// button interrupt to start and stop belt
ISR(INT2_vect)
{
    mTimer(20);
    if(((PIND & 0x04)) == 0x04)
    {
        STATE = 3; // PAUSE_STAGE

        while(((PIND & 0x04)) == 0x04);
        mTimer(20);
    }
}
```

In the pause stage an ‘if’ and ‘else’ statement is used to determine if the belt is currently running and the user wants to stop it, or if the belt is currently stopped and the user

wants to start it. This is achieved with the previously mentioned start and stop global variable. If the belt is stopped and the user wants to move the belt, output ports are updated to start the belt and the start and stop global variable is updated. If the the belt is running and the user wants to stop it, output ports are updated to stop the belt, the global start and stop variable is updated, the LCD is cleared, and object counts in the bucket are written to the LCD. The program then returns to the polling stage.

```
   PAUSE_STAGE:
   {
       if(stopGo == 0) // if belt stopped
       {
           PORTB = 0b00001110;    // start belt in CW direction
           stopGo = 1; // belt is now running

           LCDClear();
       }
       else // if belt running
       {
           PORTB |= (1 << PB0) | (1 << PB1); // brake
           stopGo = 0; // belt is now stopped

           LCDClear();

           LCDWriteStringXY(0, 0, "B:   A:   OB:");
           LCDWriteIntXY(2, 0, black_inBucket, 2);
           LCDWriteIntXY(7, 0, aluminium_inBucket, 2);
           LCDWriteIntXY(13,0, sizeOfQueue, 2);

           LCDWriteStringXY(0,1, "W:   S:   T1:");
           LCDWriteIntXY(2, 1, white_inBucket, 2);
           LCDWriteIntXY(7, 1, steel_inBucket, 2);
           LCDWriteIntXY(13, 1, (black_inBucket + aluminium_inBucket + white_inBucket +
steel_inBucket), 2);
       }

       STATE = 0;
       goto POLLING_STAGE;
   }
```

II.A.1.j Optimization

The speed of the entire system depends greatly on three parameters, the belt speed, the bucket's maximum speed, and how fast the bucket accelerates and decelerates to and from that maximum speed.

The belt speed can be adjusted by updating the Output Compare Register A. This is the final line of code in the PWM initialization below.

```
void initializePWM()
{
 /*
 Timer/Counter Control Register A
 - Timer 0 set to Fast PWM mode
 - TOP set to 0xFF
 - OCRA update set to TOP
 - TOV flag set to MAX
 - Page 126 to 128
 */
 TCCR0A |= (1 << WGM00) | (1 << WGM01);

 /*
 Timer/Counter Interrupt Mask Register
 - Enable output compare interrupt (OCIE0A) for Timer 0
 - Page 131
 */
 //TIMSK0 |= (1 << OCIE0A);

 /*
 Timer/Counter Control Register A
 - Set compare match output mode to clear on compare match
 - Clear OC0A on Compare Match, set at BOTTOM
 - Page 126
 */
 TCCR0A |= (1 << COM0A1);

 /*
 Timer/Counter Control Register B
 - Set prescaler in TCCR0B to 64 to get 3.9 kHz
 - Page 129 to 130
 */
 TCCR0B |= (1 << CS01) | (1 << CS00);

 /*
```

```
 Output Compare Register A
 - Set to 50% duty cycle
 - 0xFF (127 out of 255)
 - Page 130
 */
 OCR0A = 0x78; // initialize to 120(dec)
}
```

The maximum speed of the bucket is determined by a delay variable in the trapezoidal acceleration and deceleration function. It was found that a delay of 8 ms between bucket step increments provided a healthy balance of speed and reliability. The amount of steps the bucket accelerates and decelerates is determined by dividing the total number of steps in acceleration, deceleration, and constant speed blocks. It was found that accelerating and deceleration for ⅛ of the total number of steps balanced speed and reliability.

```
void trapezoidalMove(int steps, int direction)
{
   int accelSteps = steps / 8; // acceleration steps
   int decelSteps = steps / 8; // deceleration steps
   int constSpeedSteps = steps - (accelSteps + decelSteps); // constant speed steps

   int maximumDelayConst = 20;
   int maximumDelay = maximumDelayConst; // slowest speed
   int minimumDelay = 8; // fastest speed
   float accelDelayIncrements = (float)(maximumDelay - minimumDelay) / accelSteps;
   float decelDelayIncrements = (float)(maximumDelay - minimumDelay) / decelSteps;
```

II.A.1.k Final Testing

Final testing was completed with a 48 piece set of coloured objects. A lot of time was devoted to determining the ideal belt speed, bucket's maximum speed, and bucket acceleration and deceleration steps. In addition to optimization, calibration required numerous runs carefully recording ADC values for given object colours.

Many issues were encountered such as overlapping ADC values between white and black objects, belt freezing from multiple interrupts happening in quick succession, and objects loaded the bucket causing its acceleration performance to degrade. These issues were resolved by carefully cleaning the objects, implementing an EX flag, and decreasing the maximum speed of the bucket to a reliable level.

II.A.1.l Demonstration

The program was demonstrated on December 3, 2024 at 2:00 pm. The first attempt at sorting the 48 piece set completed in 38 seconds with two errors resulting from missorted pieces. The second attempt saw an increase in speed to 37 seconds and had no errors.

II.A.1.m Documentation

The documentation is the assembly of this report due December 17, 2024.

II.B Gantt Chart

See Appendix A for the Gantt Chart.

The Gantt chart provided a visual representation of the project timeline, breaking down each phase into specific tasks and milestones. It outlined the sequence and duration of activities, ensuring clarity in task delegation and progress tracking, which helped the team stay organized and on schedule.

II.C Flowchart

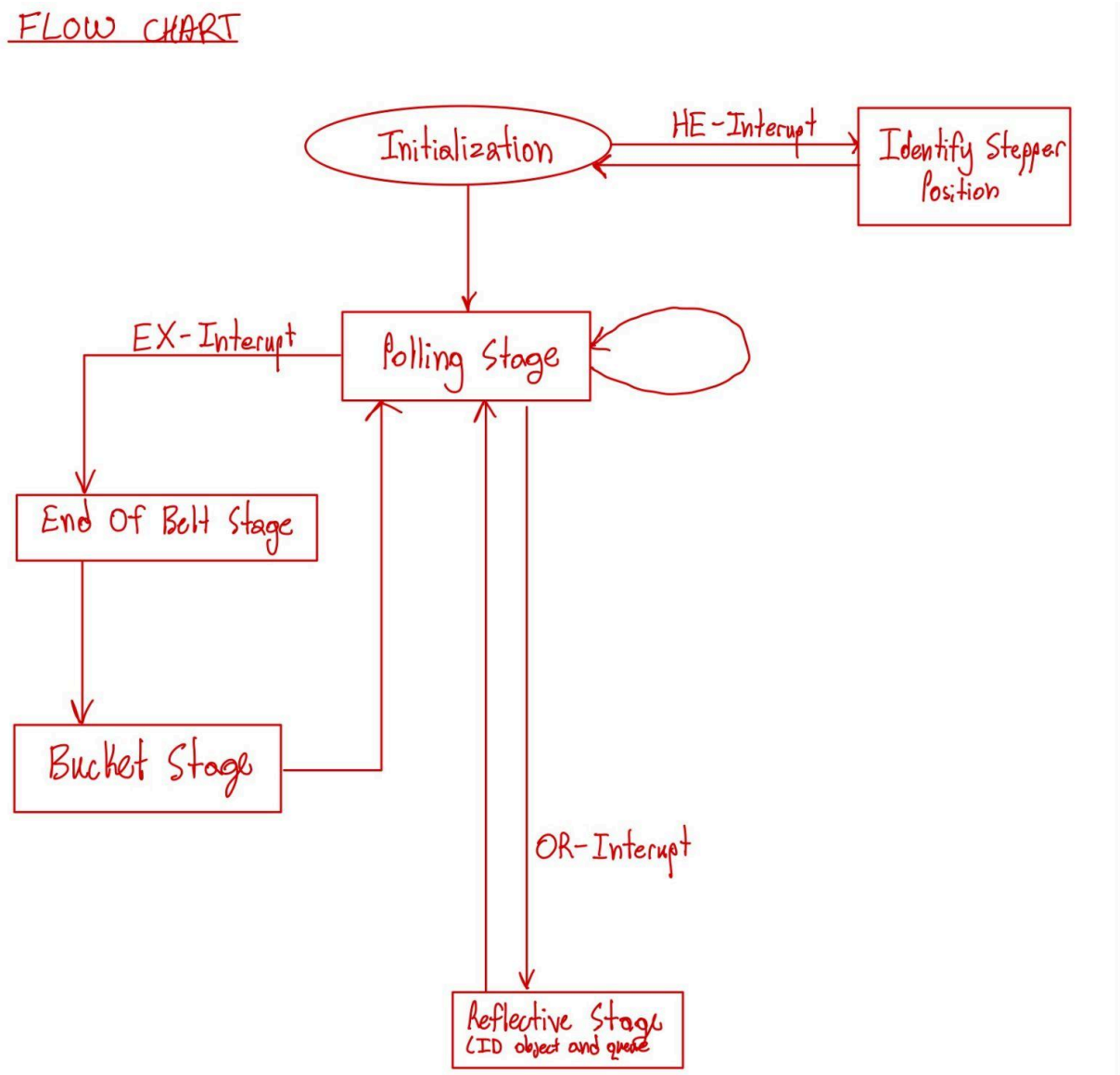


Figure 3: System Flowchart

II.D Description of the Flowchart

The following section details the main function and interrupt flow of the Arduino Mega 2560 MCU. The program starts with initializations of the clock, I/O pins, interrupts, LCD,

PWM, stepper motor, DC motor, and linked list. An HE sensor interrupt will trigger once during the stepper motor initialization for motor position identification.

After initializations are complete, the program enters the Polling Stage. Once an OR interrupt is triggered, the program moves to the Reflective Stage for object identification and enqueuing, returning to the Polling Stage once complete.

While in the Polling Stage, if the EX interrupt is triggered the program moves to the End-Of-Belt Stage where object identification and dequeuing occurs. The program then moves to the Bucket Stage for physical sorting. Once Bucket Stage is complete the program returns to the Polling Stage [#].

II.E Circuit diagram predesign

See Appendix B for the Circuit Diagram.

The circuit diagram illustrated the electrical connections and components of the system, including the microcontroller, motors, sensors, and power supply. It served as a guide for hardware assembly and an essential resource for troubleshooting and documenting the project's electrical design.

II.F Design Research

II.F.1 Bucket math

To determine the shortest path for alignment, we assigned a numerical value to each type of material as follows:

- Black (B) = 0
- Aluminum (A) = 1
- White (W) = 2

- Steel (S) = 3

We then built a table with all possible pairs of materials, as shown below. The values represent the difference between the positions of the materials shown in table 3

Table 3: Possible Values of Various Pairs of Materials for $f = (cp - x)\%4$

		B	A	W	S
		0	1	2	3
B	0	0	-1	-2	-3
A	1	1	0	-1	-2
W	2	2	1	0	-1
S	3	3	2	1	0

Initially, this approach seemed straightforward. However, the presence of negative values complicated the process of determining whether to turn clockwise or counterclockwise. To resolve this, we revisited modular arithmetic concepts [2]. The initial formula, $f = (cp - x)$, where cp is the current position and x is the target position, yielded inconsistent directions results due to negative values, shown in figure 4 below.

Figure 4: $f = (cp - x)$ modular arithmetic

Adjusting the formula to include a positive offset is done by adding four to the equation:

$$f = ((cp - x) + 4)\%4$$

This adjustment ensured all values remained positive, flipping the sequence from [0, 1, 2, 3] to [3, 2, 1, 0]. Applying this adjustment to our material values produced the following table.

Table 4: Values of Various Pairs of Materials of $f = ((cp - x) + 4)\%4$

		B	A	W	S
		0	1	2	3
B	0	0	3	2	1
A	1	1	0	3	2
W	2	2	1	0	3
S	3	3	2	1	0

where all values are now positive and represent consistent directional movement as shown in figure 5 below.

Figure 5: $f = ((cp - x) + 4)\%4$ modular arithmetic

With all values positive, we could now determine the shortest path for alignment. Here we implement three various conditions.

1. Turn == 0
    - Do nothing as it is already at that position
2. Turn <= 2
    - Move the servo turn* 50 (50 or 100) steps in the direction it is already going.
3. Turn > 2
    - For anything greater than two so a value of three we subtract it from four and rotate the servo in the opposite direction as it is the shortest distance. As 50*3 = 150 but (4-3)*50 = 50 and as 50 is smaller we must rotate the opposite direction.

```
int targetPosition = objectType; // pulling object form the q

 // modular math to remove negative values and return a value that can be easily sorted into
what direction it should turn.
               int turn = (targetPosition - currentPosition + 4) % 4;
               if (turn == 0)
               {
                      // Do nothing
               }
               else if (turn <= 2)
               {
                 // move same direction already going by 50 or 100 steps
                      trapezoidalMove( (turn * 50), 1);
                }
               else
               {
                    //move opposite direction by 50 steps
                      trapezoidalMove( ((4 - turn) * 50), 0);
               }
                 currentPosition = targetPosition;
```

For more detail see Appendix C.

II.F.2 Trapezoid

The speed of the stepper motor is controlled by varying the delay between individual steps. One step of the stepper motor corresponds to an angular displacement of 1.8°. Two hundred steps corresponds to one full rotation of the stepper motor, 360°.

In testing it was found that the stepper motor's maximum starting speed was relatively slow. This maximum starting speed was achieved with a delay of 20 ms. The stepper motor's speed would have to be increased to meet the project specifications. Decreasing the stepper motor delay below 20 ms resulted in the stepper motor locking up intermittently, becoming unreliable. To solve this problem, it was decided that the stepper motor angular speed would increase and decrease using a trapezoidal acceleration curve. All motor movements except the initialization movements will be controlled by calling the function *void trapazoidalMove(int steps, int direction).*

Procedure

1. The trapezoidal function divides steps ‘x’ by a predetermined value ‘a’. This value controls what fraction of the total number of steps is being allocated for acceleration and deceleration.
2. The number of steps at top speed is the total number of steps subtracted by the number of acceleration and deceleration steps.
3. Maximum delay is a constant within the function and is set to 20 ms.
4. Minimum delay ‘y’ will be determined experimentally.
5. The slope of the acceleration curve will be determined using equation $m = \frac{20-y}{(\frac{x}{a})}$.
6. In the acceleration phase, after each motor step movement the maximum delay is decreased by subtracting the magnitude of the slope, effectively increasing the motor speed.
7. In the constant speed phase, each motor step has a delay corresponding to the minimum delay.
8. In the deceleration phase, after each motor step the maximum delay is increased by adding the magnitude of the slope, effectively decreasing the speed.

Figure 6 below shows the trapezoidal acceleration graphically.

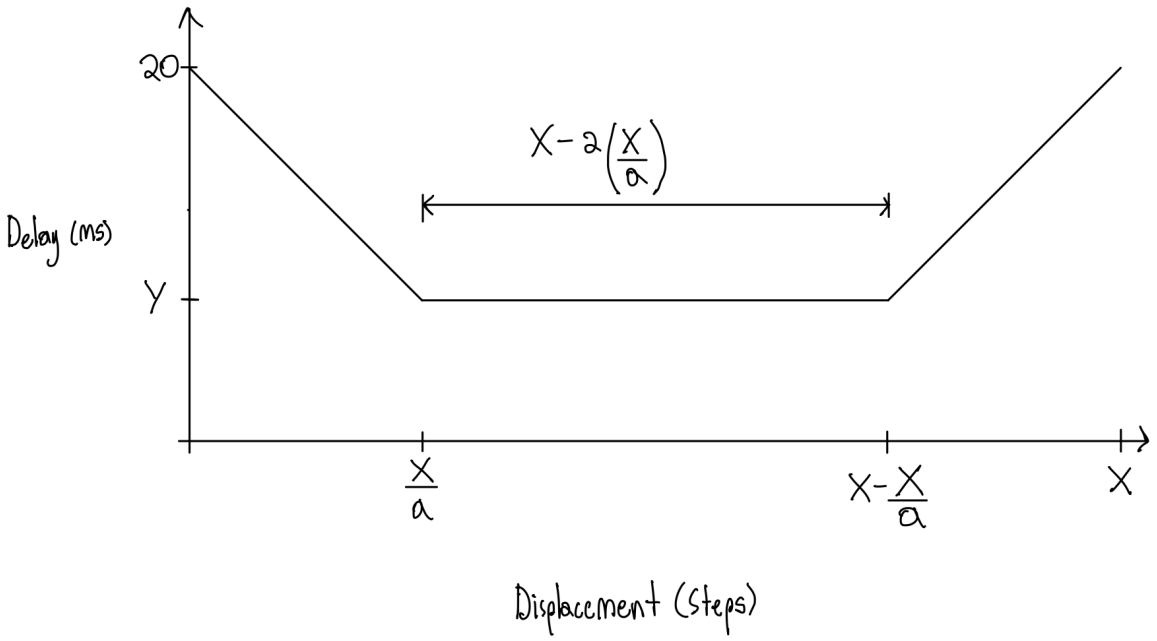


Figure 6: Trapezoidal Acceleration and Deceleration Curve

II.F.3 Pulling From Different Labs

II.F.3.a Lab 1

The following libraries, functions, processes, and logic from Lab 1 used in the final project are as follows.

- Inclusion of header files containing the standard library and I/O library.
- Initialization of I/O ports.
- Setting output ports to ‘high’ or ‘low’.

II.F.3.b Lab 2

The following libraries, functions, processes, and logic from Lab 2 used in the final project are as follows.

- Inclusion of header files containing the interrupt library and LCD library.

- mTimer function (includes timer register initializations, interrupt, and logic used to track specific amounts of time in milliseconds).
- System clock initializations.
- LCD initialization.
- LCD commands to write strings and integers to specific locations.

II.F.3.c Lab 3

The following libraries, functions, processes, and logic from Lab 3 used in the final project are as follows.

- Inclusion of header files containing the linked list library.
- Declaration and initialization of linked list pointers.
- Logic used to add and remove items containing information from the linked list.
- Functions to set up linked list, initialize linked list, enqueue items, dequeue items, clear linked list, check linked list size, check if linked list is empty, and return the first value of the linked list.

II.F.3.d Lab 4(a)

The following libraries, functions, processes, and logic from Lab 4(a) used in the final project are as follows.

- Functions to move the stepper motor clockwise and counter-clockwise.
- Function used to initialize stepper motor position.
- Function used to initialize PWM.

II.F.3.e Lab 4(b)

The following libraries, functions, processes, and logic from Lab 4(a) used in the final project are as follows.

- Button activated interrupt service routine.

- ISR register initializations.
- Function used to initialize DC motor.
- Logic used to set DC motor direction, brake, and control speed.
- ADC interrupt service routine.
- ADC register initializations.
- ADC logic used to obtain measured values with register manipulation and flags.

II.F.4 Errors Found in Lab Code

II.F.4.a Linked List

A linked list is used to dynamically track the identified objects between the OR and EX sensors. The provided linked list code was found to be missing a key step in the dequeue function.

The dequeue function provided works by first checking if the head pointer passed into the function is NULL. If the pointer is valid, the head pointer is moved to the next item in the queue. However, the provided code does not check if the linked list is empty after the head pointer is moved. This can result in errors in later uses of the linked list functions as the tail pointer had not been updated to account for the linked list being emptied.

To fix this error, an additional if statement was added to check if the head pointer is NULL after it is moved to the next item in the linked list. If the head pointer is NULL then the tail pointer will be set to NULL.

II.F.5 Binary code table update:

In order to update the table first we had to understand What E and I were.

1. E(Enable)

Purpose: Acts as a global on/off switch for the motor driver or specific motor phases.

Function: It controls whether the stepper motor driver is active or idle.

- When EEE is 1 (or high), the driver is enabled, allowing current to flow to the motor coils and enabling movement.
- When EEE is 0 (or low), the driver is disabled, and the motor is not energized (might freewheel or hold its position, depending on the driver setup).

2. I (Input or Signal)

   Purpose: Specifies the state (on/off) of each coil or phase in the motor to dictate its movement.

   Function:

   - I1, I2, I3, I4, typically represent the control signals for different windings or coils of the stepper motor.
   - These signals determine the stepper motor's position by energizing the windings in a particular sequence (e.g., full-step, half-step, or microstepping).

Once we understood what E and I did we looked into how to effectively control a stepper motor. We found that it's essential to understand the key components involved, often referred to by the acronym SID:

- Stepper Motor (S): The electromechanical device that converts electrical pulses into discrete mechanical movements.
- Indexer/ Controller (I): Generates step pulses and direction signals to dictate the motor's movement.
- Driver/ Amplifier (D): Converts the controller's signals into the necessary power to energize the motor windings.

This combination ensures precise control over the stepper motor's position and speed. [3][4] [5]

Braking that down into steps:

1. Stepper Motor (S):
    a. The physical stepper motor receives current through its coils based on the sequence of input signals (I1, I2, I3, I4) to move in discrete steps.
    b. The motor itself is ready to move once enabled and provided with current through the driver.
2. Indexer (I):
    a. The Indexer generates the step sequence and direction control signals.
    b. This sequence controls which coils are energized and in what order.
3. Driver (D):
    a. The Driver receives the signals (I1, I2, I3, I4) from the indexer and translates them into power signals that energize the stepper motor's coils.
    b. The Enable signal (E1, E2) ensures the driver is powered and ready to pass current to the motor.
        i. When E = 1, the driver powers the motor.
        ii. When E = 0, the driver disables power to the motor, and it may freewheel or hold its position depending on the design.

There are various sequences that can be used:

- Full-Step Sequence:
    - Energizes two coils at a time. [6]
    - Used for simplicity and torque.
- Half-Step Sequence:
    - Alternates between energizing one coil and two coils, effectively doubling the resolution of a full-step sequence. [6]
    - Used for smoother motion without much complexity.
- Microstepping:

    - Gradually adjusts the current in each coil to divide a full step into many smaller steps.[7]
    - Used for precision and quiet operation.
- Wave Drive (Single-Coil Excitation):
    - Energizes only one coil at a time. [6]
    - Used for low-power or basic applications.

Table 5: Original Table from Lab 4.a [8]

STEP	E1	I1	I2	E2	I3	I4	BINARY
1	1	1	0	0	0	0	110000
2	0	0	0	1	1	0	000110
3	1	0	1	0	0	0	101000
4	0	0	0	1	0	1	000101

For this project we changed it to the Full-Step Sequence.

As a full-step instead of half-step provides:

1. Higher torque to handle loads.
2. Simpler control for straightforward systems.
3. Improved stability in noisy environments or at rest.

Half-step is preferable when resolution or smoothness is more important, but for applications prioritizing torque and simplicity, full-step is often the best choice.

Table 6: Full-Step Sequence

STEP	E1	I1	I2	E2	I3	I4	BINARY
1	1	1	0	1	0	1	110101

2	1	0	1	1	0	1	101101
3	1	0	1	1	1	0	101110
4	1	1	0	1	1	0	110110

III. RESULTS

III.A - Technical Description

The system is designed to automate the sorting of items based on their material and reflectivity, leveraging a combination of sensors, motors, and microcontroller-based processing. Below is a detailed description of the primary components:

1. Optical Sensor (OR)
    - Product Name: SFH 310 NPN Silicon Phototransistor
    - Purpose: Detects the presence or absence of objects as they move along the conveyor belt.
    - Operation: Emits infrared light and detects its reflection or interruption to signal the presence of an object. This triggers the Reflective Stage for further processing.
    - Specifications:
        - Light type: Infrared
        - Sensitivity: Operates effectively in the 400 nm to 1100 nm wavelength range.
        - Operating Voltage: 3V to 24V DC
        - Response Time: 5 µs (typical rise and fall time)
2. Reflectivity Sensor (RL)
    - Product Name: E2R-A01 Amplified Proximity Sensor

- Purpose: Measures the reflectivity of materials for classification into one of four categories: aluminum, steel, white plastic, and black plastic.
- Operation: Projects light onto the object, measuring the intensity of the reflected light to determine its material composition. Integrated with the ADC for precise readings during the Reflective Stage.
- Specifications:
  - Sensing Range: Approximately 0–5 mm
  - Response Frequency: Up to 5 kHz
  - Output: NPN transistor, open collector

3. Optical Sensor (EX)
   - Product Name: OPB819Z Slotted Optical Switch
   - Purpose: Detects when objects reach the end of the conveyor belt for dequeuing and sorting.
   - Operation: Infrared emitter and phototransistor identify when objects pass through the slot, triggering the End-of-Belt Stage.
   - Specifications:
     - Slot Dimensions: 1.25" width
     - Output Type: NPN transistor
     - Operating Temperature: -40°C to +85°C
4. Hall-Effect Sensor (HE)
   - Product Name: SS400 Series Hall-Effect Sensors
   - Purpose: Ensures the stepper motor bucket is correctly aligned (homed) for sorting operations.
   - Operation: Detects the presence of a magnetic field and signals when the bucket has reached its home position.
   - Specifications:
     - Magnetic Sensitivity: Operates with magnetic fields > 20 Gauss
     - Power Requirements: 4.5–24 V DC
     - Response Time: < 1 µs

5. Stepper Motor
    - Product Name: Soyo SY42STH38-0806A Unipolar Stepper Motor
    - Purpose: Provides precise control for rotating the sorting bucket to designated positions.
    - Operation: Moves in discrete 1.8° steps, controlled using trapezoidal acceleration for smooth operation and reduced mechanical strain.
    - Specifications:
        - Step Angle: 1.8°
        - Holding Torque: 2.6 kg-cm
        - Current/Phase: 0.8 A
        - Calibrated delays: Maximum = 20 ms, Minimum = 8 ms
6. DC Motor
    - Product Name: Banebots RS-540 Planetary Gear Motor
    - Purpose: Drives the conveyor belt for transporting items through inspection stations.
    - Operation: Converts electrical energy into continuous rotation with adjustable speed via PWM signals.
    - Specifications:
        - Voltage Range: 4.5V to 12V
        - No-Load Speed: 263 RPM
        - Stall Torque: 2527 oz-in
        - Peak Current: 42 A
7. Microcontroller
    - Product Name: Pololu Microcontroller-Compatible Motor Driver
    - Purpose: Central processing unit for coordinating sensor readings, motor movements, and overall system operation.
    - Operation: Executes programmed instructions to process sensor data, control motor drivers, and handle user inputs.
    - Specifications:

        - Operating Voltage: 5.5–16 V
        - PWM Frequency: Up to 20 kHz
        - GPIO pins: 16–40
        - Communication Protocols: UART, I2C, SPI

III.B - System Performance Specifications

Below are the performance specifications of each component based on system requirements:

1. Optical Sensor (OR)
    - Sensitivity: Operates effectively in the 400 nm to 1100 nm wavelength range
    - Response Time: 5 µs (typical rise and fall time)
    - Operating Voltage: 3V to 24V DC
    - Operating Temperature: -40°C to +100°C
2. Reflectivity Sensor (RL)
    - Sensing Range: 0–5 mm
    - Response Frequency: Up to 5 kHz
    - Power Supply: 12–24 V DC
    - Threshold Values for Reflective Stages:
        - Aluminum < 200
        - 200 ≤ Steel < 700
        - 700 ≤ White Plastic < 915
        - Black Plastic ≥ 915
3. Optical Sensor (EX)
    - Slot Dimensions: 1.25" width
    - Output Type: NPN transistor
    - Operating Voltage: 30V (collector-emitter breakdown)
    - Response Current: 0.5–12 mA (typical)

4. Hall-Effect Sensor (HE)
    - Magnetic Sensitivity: Operates with magnetic fields > 20 Gauss
    - Response Time: < 1 µs
    - Power Requirements: 4.5–24 V DC
    - Output Configuration: Open collector, digital output
5. Stepper Motor
    - Step Angle: 1.8° per step
    - Holding Torque: 2.6 kg-cm
    - Current/Phase: 0.8 A
    - Operating Temperature: -20°C to +50°C
    - Trapezoidal Movement Calibration:
        - Maximum Delay: 20 ms
        - Minimum Delay: 8 ms
6. DC Motor
    - No-Load Speed: 263 RPM
    - Stall Torque: 2527 oz-in
    - Peak Current: 42 A
    - Voltage Range: 4.5–12 V
7. Microcontroller
    - Operating Voltage: 5.5–16 V
    - PWM Frequency: Up to 20 kHz
    - Continuous Current: 14 A per channel
    - Control Interfaces: PWM, INA/INB for direction control

III.C - System Algorithm

```
START
  INITIALIZE sensors, motors, and interrupts
  ALIGN stepper motor bucket using Hall-Effect Sensor
  SET system state to POLLING_STAGE
```

```
POLLING_STAGE:
  IF Optical Sensor (OR) detects an object:
    MOVE to REFLECTIVE_STAGE

REFLECTIVE_STAGE:
  MEASURE reflectivity using RL Sensor
  CLASSIFY object and enqueue object code
  RETURN to POLLING_STAGE

IF Optical Sensor (EX) detects object:
  MOVE to END_OF_BELT_STAGE

END_OF_BELT_STAGE:
  DEQUEUE object code
  CALCULATE bucket position using modular math
  ROTATE bucket to target position using Stepper Motor
  SORT object into designated bin
  RETURN to POLLING_STAGE

USER INTERACTIONS:
  IF push button pressed:
    TOGGLE between PAUSE and RESUME states
    UPDATE LCD with current statistics

ERROR HANDLING:
  IF error detected:
    HALT system and DISPLAY diagnostic information

SHUTDOWN:
  COMPLETE sorting all queued objects
  DISPLAY final statistics
  STOP all components

END
```

III.D - Operation

The sorting system begins operation by initializing all sensors, motors, and interrupts, ensuring the system is ready for use. The DC motor drives the conveyor belt, transporting objects toward the inspection stations.

At Station S1, the Optical Sensor (OR) detects an incoming object and triggers the Reflective Sensor (RL) to measure the object's reflectivity. Based on the ADC readings, the object is classified into one of four categories (e.g., aluminum, steel, white plastic, black plastic) and enqueued for sorting.

The conveyor belt moves the object to the end-of-belt station, where the Optical Sensor (EX) detects its presence. The microcontroller dequeues the object’s classification and calculates the required movement for the Stepper Motor. Using trapezoidal acceleration, the motor aligns the bucket with the appropriate bin. The object is then sorted into its designated bin, and the process repeats for subsequent objects.

Real-time statistics, such as the count of each material type, are displayed on the LCD. The user can pause or resume the system using the push button, ensuring operational flexibility. In case of errors or interruptions, the system halts and displays diagnostic information for troubleshooting.

III.E - Testing and Calibration

III.E.1 Material calibration

To set the necessary values for the reflective stage, we began by calibrating the system using a sample size large enough to provide reliable expectations. Each material type was run through the sensor 10 times, enabling us to identify any outliers and establish a clear range for each material.

However, the calibration process for the black and white Delrin materials required additional attention due to their closely clustered values. For the white Delrin, we polished the surface and gradually adjusted the upper limit until we were satisfied with the readings. To ensure accuracy, we ran all 12 pieces of both black and white Delrin through the sensor four times each, refining the calibration to achieve reliable and consistent results.

The limits for each piece were found to be:

- Aluminum < 200
- 200 >= Steel < 700
- 700 > = White < 915
- 915< Black

III.E.2 Motor noise

The motor's operation involves switching polarity to drive its motion, which also alternates the magnetic field. This switching generates high-frequency electrical noise, or "sparks," which can interfere with the performance of the HE, EX, and OR sensors. To mitigate this noise, adding resistors and, where necessary, capacitors to filter out the unwanted frequencies, as shown in figure 7 below.

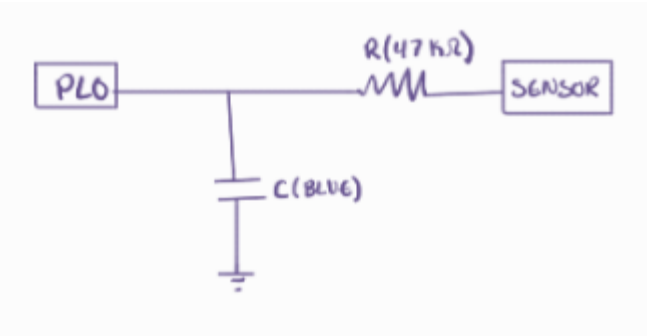


Figure 7: Resistor and Capacitor circuit for motor noise reduction

In our setup, we found it essential to include resistors on each sensor. Additionally, capacitors were required specifically for the HE and OR sensors to effectively eliminate noise and ensure reliable sensor operation.

III.E.3 Bucket Stage q Math test

To verify the functionality of our bucket stage, we began by performing manual calculations, as shown in the accompanying figure 8 below. Following this, we ran test pieces through the system, with each piece's name displayed as it passed through the sensor.

This process involved numerous iterations of code before finalizing the version implemented in our system. Detailed hand calculations for various methods are provided in the appendix D for reference.

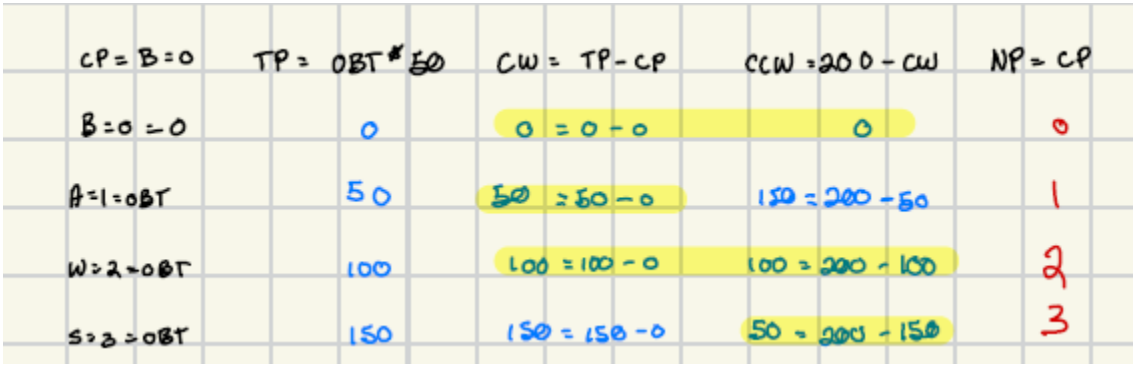


Figure 8: Bucket Stage q test

III.E.4 Servo steps

III.E.4.a Servo steps Calibration

To calibrate the servo at each station, we observed where the pieces landed on the platform. If a piece was off-center, we adjusted the servo's "home" position by the necessary number of steps to ensure the piece consistently fell onto the center of the platform. This iterative process ensured precise alignment at every station.

III.E.4.b Servo steps Testing

To ensure the servo moved to the correct positions and that the logic functioned as intended, we conducted a series of trials by running parts through specific sequences. Each sequence was tested 12 times to evaluate the system's performance under various turning patterns. The tested sequences included:

1. [B, W, B, W] - Four steps, each 100 units counterclockwise (CCW).
2. [S, A, S, A] - Four steps, each 100 units clockwise (CW).
3. [A, W, S, B] - Four steps, each 50 units CCW.
4. [S, W, A, B] - Four steps, each 50 units CW.
5. [A, S, W, B] - A mixed pattern: 50 CCW, 100 CCW, 50 CW, 100 CW.

While most sequences worked as expected, errors were observed primarily in sequence 5, which highlighted inconsistencies in the logic when handling mixed-direction movements. Some of the specific issues encountered included:

- HE Sensor Glitch: The piece would initially fail to drop, then fall later, resulting in an extra piece being left on the belt.
- Stepper Motor Errors: The motor occasionally moved to completely incorrect positions.

Upon further investigation, it became clear that these issues were caused by two main factors:

1. Trapezoidal Movement Calibration: Errors in the movement logic during trapezoidal acceleration and deceleration.
2. Piece Falling Delay: Insufficient delay to account for the time it takes for a piece to drop onto the tray.

These problems are addressed in greater detail in Sections III.E.5 (Trapezoid Calibration) and III.E.6 (Piece Falling Delay Calibration), where additional debugging and refinements were implemented.

III.E.5 Trapezoid calibration

Calibration of the trapezoidal acceleration and deceleration function was divided into three parts, the maximum speed, the slope of increasing velocity versus time (acceleration), and the slope of decreasing velocity versus time (deceleration).

Before determining maximum speed, the slope of acceleration and deceleration were both programmed to be very shallow. This ensured that maximum speed could be determined independently of any acceleration or acceleration issues. It was decided that a slope that ran for ¼ of the total number of steps to be turned would be used as the starting point for maximum speed calibration. Next, the stepper motor was loaded with four D type alkaline batteries. These batteries simulated the weight of the 48 objects with a comfortable margin of error. Finally, the minimum delay variable was decreased incrementally with multiple test runs completed at each increment. It was found that a minimum delay of 6 ms or less could cause the stepper motor to jam. This was likely due to the motors inability to turn that quickly under load and by interrupts that could have happened between motor steps. The minimum delay that provided the best balance of speed and reliability was found to be 8 ms. The code below shows the

variables controlling the acceleration and deceleration run as well as the minimum delay variable.

```
int accelSteps = steps / 8; // acceleration steps
int decelSteps = steps / 8; // deceleration steps
int constSpeedSteps = steps - (accelSteps + decelSteps); // constant speed steps

int maximumDelayConst = 20;
int maximumDelay = maximumDelayConst; // slowest speed
int minimumDelay = 8; // fastest speed
float accelDelayIncrements = (float)(maximumDelay - minimumDelay) / accelSteps;
float decelDelayIncrements = (float)(maximumDelay - minimumDelay) / decelSteps;
```

A similar process was followed with the acceleration and deceleration run calibration. The starting point for testing was set to ¼ of the total number of steps for both acceleration and deceleration. This was incrementally deceased until the stepper motor became unreliable. It was found that a run consisting of ⅛ of the total number of steps for both acceleration and deceleration provided a healthy balance of reliability and speed. Figure 9 below shows how the delay between motor speeds decreased and increased, allowing for reliable acceleration and deceleration.

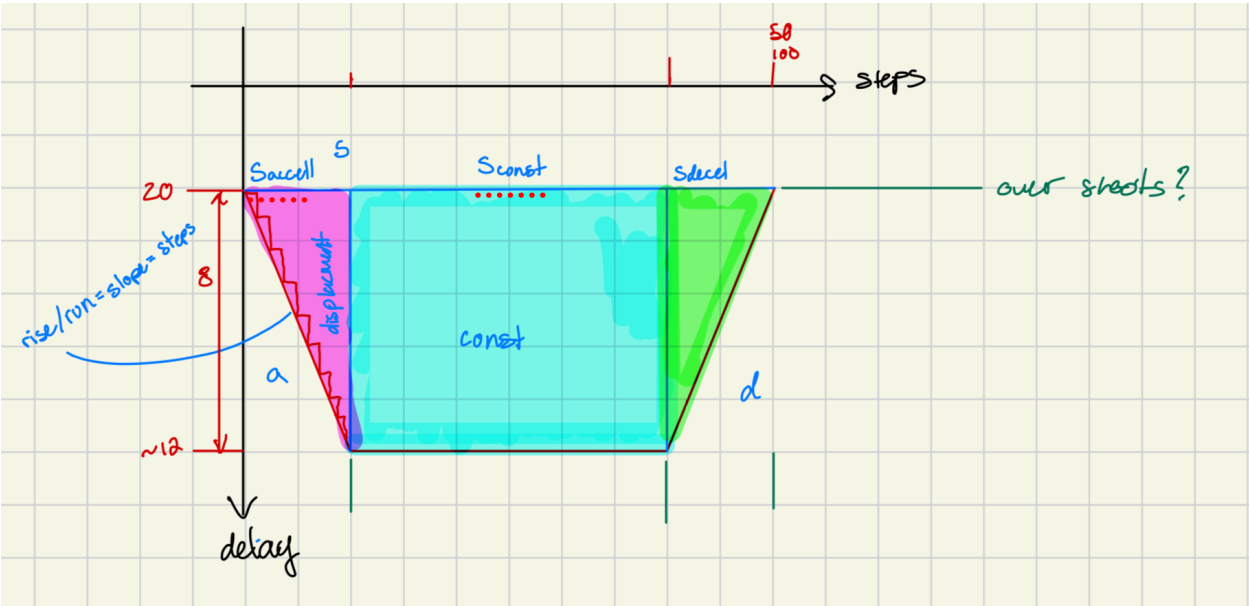


Figure 9: Trapezoidal acceleration and deceleration curve

III.E.6 Piece falling delay calibration

It was discovered early in testing that the program needed to account for the time it takes an object to fall from the belt to the bucket before the bucket begins to move into position for the next object. Increasing this delay significantly slowed the sorting of the 48 pieces but was essential to prevent objects from falling onto the divider of the bucket or into a slot of the bucket that it was not intended for. A delay of 120 ms between bucket repositioning allowed for the object to reliably fall into the correct slot. A slow motion camera was used to confirm this operation and determine the length of the delay. The code used to account for the time it takes for an object to fall from the belt to the bucket can be seen below.

```
int targetPosition = objectType;
int turn = (targetPosition - currentPosition + 4) % 4;
if (turn == 0)
{
    // Do nothing
}
else if (turn <= 2)
{
    trapezoidalMove( (turn * 50), 1);
}
else
{
    trapezoidalMove( ((4 - turn) * 50), 0);
}
currentPosition = targetPosition;
PORTB = 0b00001110;    // start belt in CW direction
mTimer(120);
stopGo = 1; // belt is now running
```

III.F - Limitations and system tradeoff

Limitations preventing the program from reliably operating faster were the bucket speed and belt speed. To increase the speed of the program a number of tasks are recommended. The bucket must move into position in advance of the object reaching the belt and the belt must accelerate and decelerate using an ‘S’ curve or trapezoidal curve.

Currently the program moves the bucket into the correct position for a given object once the object has reached the end of the belt and the belt has stopped. Implementing code that could look ahead into the object queue and begin to move the bucket into position before the first object in the queue reaches the end of the belt could significantly increase the execution time of the program.

The greatest issue preventing the belt speed from increasing too much was objects tumbling off the end of the belt. If the belt was moving too fast and abruptly stopped once an object was detected at the end of the belt that object could tumble onto its side and roll off the belt. This is due to increased momentum of the object. Implementing an ‘S’ curve or trapezoid to accelerate and decelerate the belt more smoothly could allow the belt to run at a greater maximum speed and more reliability.

IV. CONCLUSIONS

The successful completion of this project demonstrated the group's understanding of the electrical, mechanical, and software fundamentals needed to design an automated sorting system. The system effectively integrated numerous hardware components with a microcontroller and software to achieve precise object classification and sorting within a constrained time frame.

Through rigorous testing, calibration, and optimization, a reliable and scalable code base was created, facilitating the high-performance operation of the system. This software integrated a stepper motor, DC motor, optical sensors, reflective sensor, and hall effect sensor. Efficient algorithms were used to accelerate and decelerate the bucket, and to move the bucket from one position to the next in the least amount of time.

Successful noise reduction strategies were implemented in hardware. These utilized resistors and capacitors which greatly increased the reliability of the interrupts.

Although the project was a success, limitations on the bucket speed and belt speed highlighted areas for future improvements. Incorporating algorithms to accelerate and decorate the belt more smoothly, and move the bucket in advance of an approaching object could enhance the systems performance.

V. REFERENCES

[1] Y. Shi, “Final Project: Milestone 5,” *Brightspace.com*, 2024. https://uvic.brightspace.com/ (accessed Nov. 01, 2024).

[2] YouTube, https://www.youtube.com/watch?v=LeKeZdWaIGs (accessed Dec. 6, 2024).

[3] TB6600 Stepper Motor Driver User Guide, https://www.makerguides.com/wp-content/uploads/2019/10/TB6600-Manual.pdf (accessed Dec. 7, 2024).

[4] “Stepper Motor,” Wikipedia, https://en.wikipedia.org/wiki/Stepper_motor (accessed Dec. 6, 2024).

[5] E. Technology, “Stepper motor - types, construction, and Operation,” ELECTRICAL TECHNOLOGY, https://www.electricaltechnology.org/2016/12/stepper-motor-construction-types-and-modes-of-operation.html (accessed Dec. 6, 2024).

[6] Johnson Electric, https://www.johnsonelectric.com/resources-for-engineers/stepper-motors/stepper-motors-operation-modes (accessed Dec. 6, 2024).

[7] E. Staff, “Stepper Motor Basics, types, modes, wiring, questions,” Inst Tools, https://instrumentationtools.com/stepper-motor/ (accessed Dec. 6, 2024).

[8] Y. Shi, “Interfacing: Milestone 4,” *Brightspace.com*, 2024. https://uvic.brightspace.com/ (accessed Dec. 6, 2024).

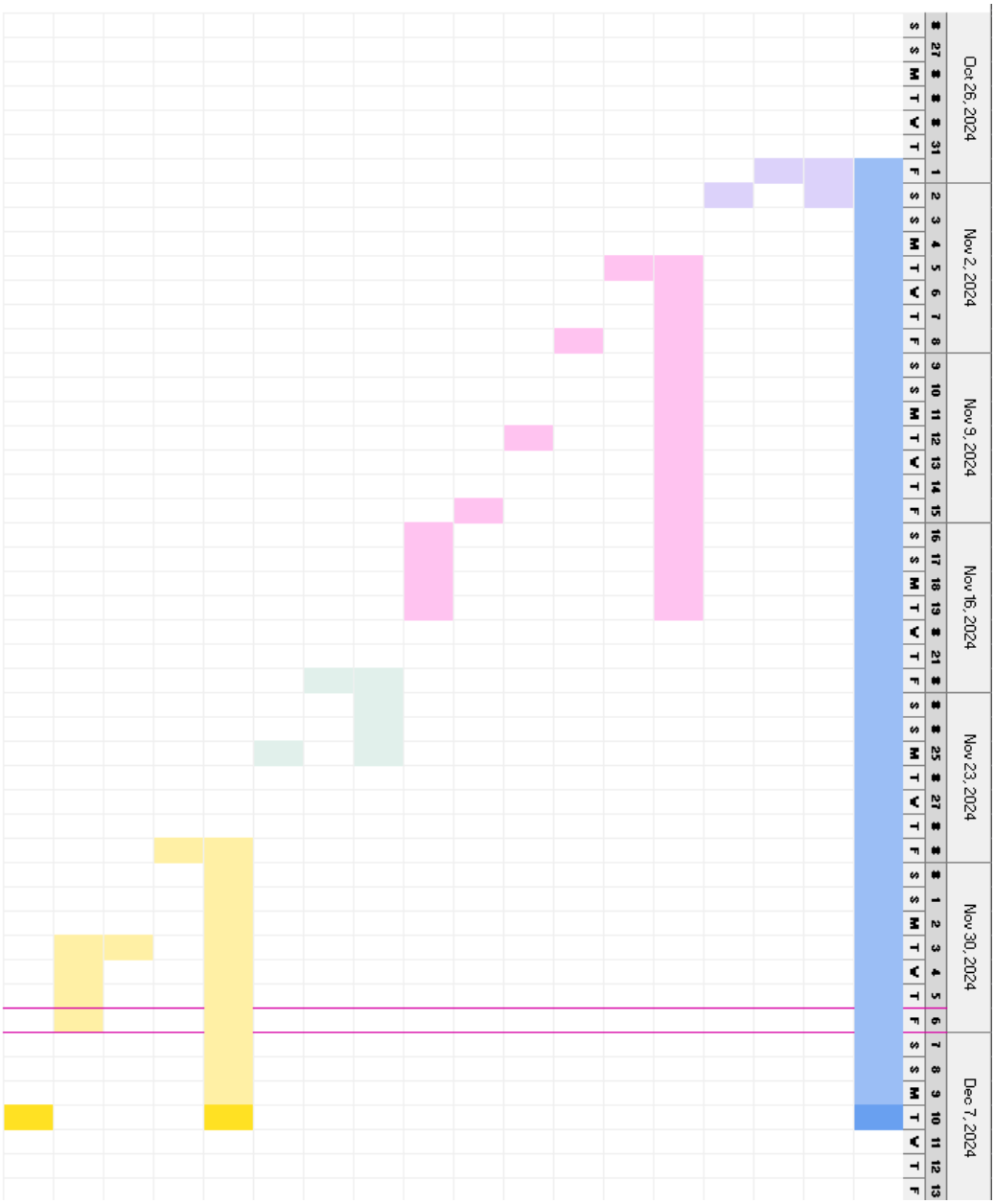
VI. APPENDICES

Appendix A - Gantt Chart
Appendix B - Circuit wiring schematic
Appendix C - Bucket Math
Appendix D - Bucket Stage q Math test
Appendix E - Full Code

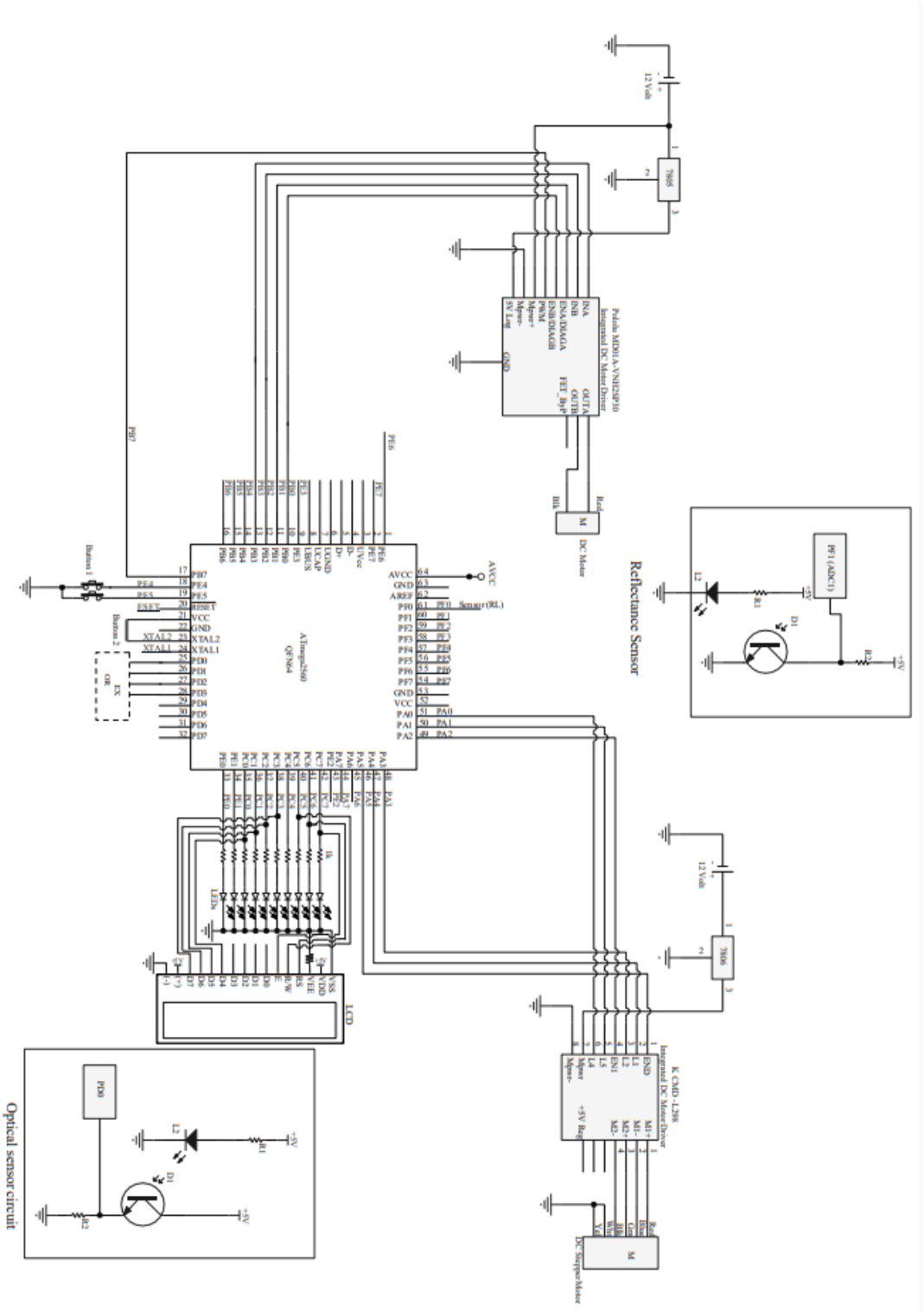
Appendix A - Gantt Chart

Lab Section B04 Carmina Rocheleau, V01021553

TASK	DISCRIPTION	PROGRESS	START	END
PROJECT OVERVIEW		**99%**	**11/1/24**	**12/10/24**
Initial Setup and Planning		**100%**	**11/1/24**	**11/2/24**
Review Project Documentation		100%	11/1/24	11/1/24
Define a Plan of Attack		100%	11/2/24	11/2/24
System Assembly and Basic Code Development		**100%**	**11/5/24**	**11/19/24**
Code structure step up		100%	11/5/24	11/5/24
Reflective sensor (RL) and optical sensor (DR)		100%	11/8/24	11/8/24
Optical Sensor (EX)		100%	11/12/24	11/12/24
Hall-Effect Sensor (HE)		100%	11/15/24	11/15/24
Full integration for all components		100%	11/16/24	11/19/24
Optimization and System Testing		**100%**	**11/22/24**	**11/25/24**
Optimization		100%	11/22/24	11/22/24
System Testing		100%	11/25/24	11/25/24
Final Testing, Debugging, Demonstration, and Documentation		**95%**	**11/29/24**	**12/10/24**
Final Testing		100%	11/29/24	11/29/24
Debugging		100%	12/3/24	12/3/24
Demonstration		100%	12/3/24	12/6/24
Documentation		80%	12/10/24	12/10/24


Oct 26, 2024
Nov 2, 2024
Nov 9, 2024
Nov 16, 2024
Nov 23, 2024
Nov 30, 2024
Dec 7, 2024

Appendix B - Circuit wiring schematic



Appendix C - Bucket Math

TURN TABLE VIEW				
		B		
		0		
A	50		150	S
		100		
		W		

		B	A	W	S
		0	1	2	3
B	0	0	-1	-2	-3
A	1	1	0	-1	-2
W	2	2	1	0	-1
S	3	3	2	1	0

TURN TABLE VIEW				
		B		
		150		
A	0		100	S
		50		
		W		

		B	A	W	S
		0	1	2	3
B	0	4	3	2	1
A	1	5	4	3	2
W	2	6	5	4	3
S	3	7	6	5	4

TURN TABLE VIEW				
		B		
		100		
A	150		50	S
		0		
		W		

		B	A	W	S
		0	1	2	3
B	0	0	3	2	1
A	1	1	0	3	2
W	2	2	1	0	3
S	3	3	2	1	0

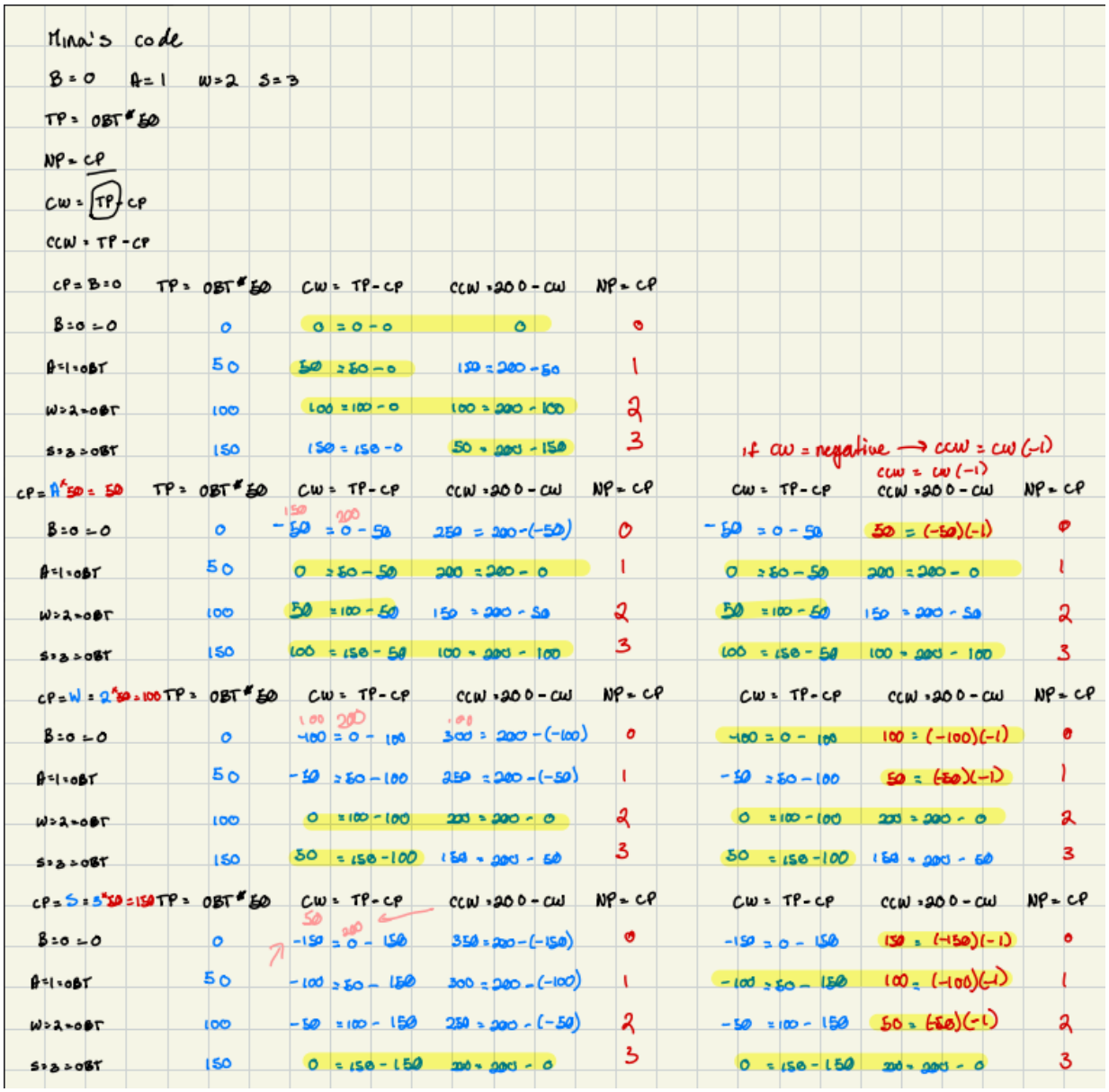
TURN TABLE VIEW				
		B		
		50		
A	100		0	S
		150		
		W		

		B	A	W	S
		0	1	2	3
B	0	0	50	100	50
A	1	50	0	50	100
W	2	100	50	0	50
S	3	50	100	50	0

POSITION VALUES			TURN VALUE								0	< = 2	>2
B	Current Position	0	TP	CP	TP-CP	(TP-CP)*50	(TP-CP)+4	((TP-CP)+4)*50	((TP-CP)+4)%4	(((TP-CP)+4)%4)*50	0	TURN*50	(4-TURN)*50
Tartget Position	B	0	0	0	0	0	4	200	0	0	NOTHING		
	A	1	1	0	1	50	5	250	1	50		50	
	W	2	2	0	2	100	6	300	2	100		100	
	S	3	3	0	3	150	7	350	3	150			50
POSITION VALUES			TURN VALUE								0	< = 2	>2
A	Current Position	1	TP	CP	TP-CP	(TP-CP)*50	(TP-CP)+4	((TP-CP)+4)%4	((TP-CP)+4)%4	(((TP-CP)+4)%4)*50	0	TURN*50	(4-TURN)*50
Tartget Position	B	0	0	1	-1	-50	3	150	3	150			50
	A	1	1	1	0	0	4	200	0	0	NOTHING		
	W	2	2	1	1	50	5	250	1	50		50	
	S	3	3	1	2	100	6	300	2	100		100	
POSITION VALUES			TURN VALUE								0	< = 2	>2
W	Current Position	2	TP	CP	TP-CP	(TP-CP)*50	(TP-CP)+4	((TP-CP)+4)%4	((TP-CP)+4)%4	(((TP-CP)+4)%4)*50	0	TURN*50	(4-TURN)*50
Tartget Position	B	0	0	2	-2	-100	2	100	2	100		100	
	A	1	1	2	-1	-50	3	150	3	150			50
	W	2	2	2	0	0	4	200	0	0	NOTHING		
	S	3	3	2	1	50	5	250	1	50		50	
POSITION VALUES			TURN VALUE								0	< = 2	>2
S	Current Position	3	TP	CP	TP-CP	(TP-CP)*50	(TP-CP)+4	((TP-CP)+4)%4	((TP-CP)+4)%4	(((TP-CP)+4)%4)*50	0	TURN*50	(4-TURN)*50
Tartget Position	B	0	0	3	-3	-150	1	50	1	50		50	
	A	1	1	3	-2	-100	2	100	2	100		100	
	W	2	2	3	-1	-50	3	150	3	150			50
	S	3	3	3	0	0	4	200	0	0	NOTHING		

```
IF TURN == 0
    NOTHING
ELSE IF TURN < = 2
    MOVE CCW BY TURN*50
ELSE
    MOVE CCW BY (4 - TURN)*50
```

Appendix D - Bucket Stage q Math test



Mina's code

B = 0 R = 1 W = 2 S = 3

TP = OBT * 50

NP = CP

CW = TP - CP

CCW = TP - CP

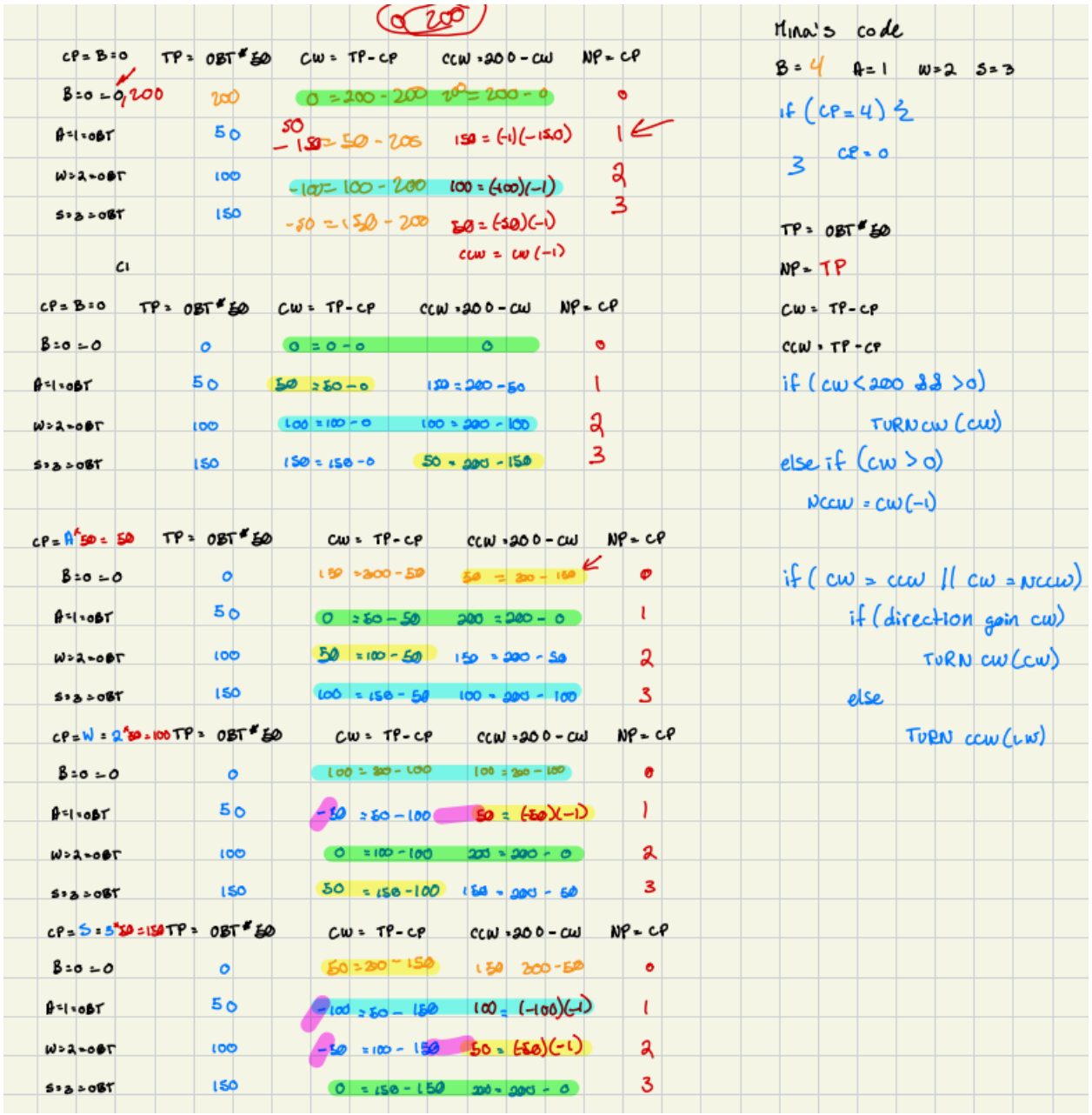
CP = B = 0	TP = OBT * 50	CW = TP - CP	CCW = 200 - CW	NP = CP
B = 0 = 0	0	0 = 0 - 0	0	0
R = 1 = OBT	50	50 = 50 - 0	150 = 200 - 50	1
W = 2 = OBT	100	100 = 100 - 0	100 = 200 - 100	2
S = 3 = OBT	150	150 = 150 - 0	50 = 200 - 150	3

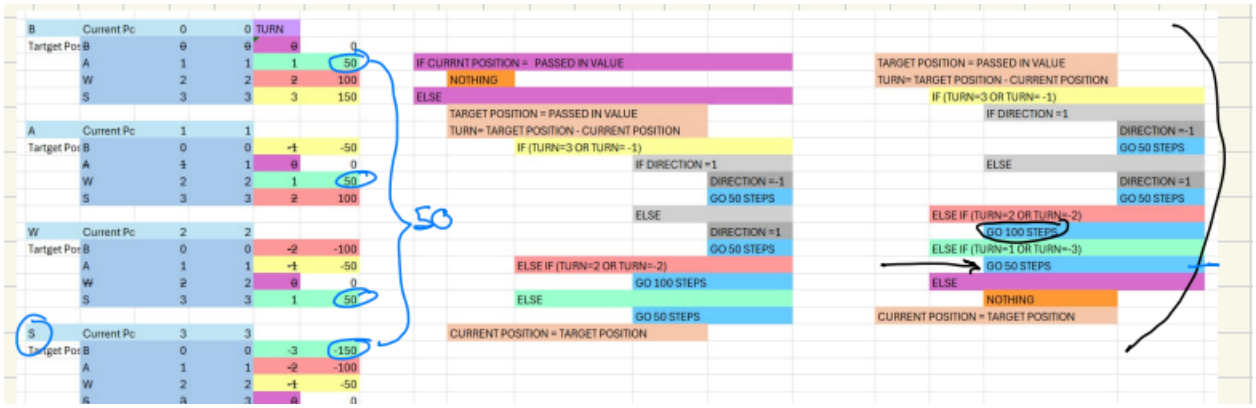
if CW = negative → CCW = CW (-1)

CP = R * 50 = 50	TP = OBT * 50	CW = TP - CP	CCW = 200 - CW	NP = CP	CW = TP - CP	CCW = CW (-1)	NP = CP
B = 0 = 0	0	-50 = 0 - 50	250 = 200 - (-50)	0	-50 = 0 - 50	50 = (-50)(-1)	0
R = 1 = OBT	50	0 = 50 - 50	200 = 200 - 0	1	0 = 50 - 50	200 = 200 - 0	1
W = 2 = OBT	100	50 = 100 - 50	150 = 200 - 50	2	50 = 100 - 50	150 = 200 - 50	2
S = 3 = OBT	150	100 = 150 - 50	100 = 200 - 100	3	100 = 150 - 50	100 = 200 - 100	3

CP = W = 2 * 50 = 100	TP = OBT * 50	CW = TP - CP	CCW = 200 - CW	NP = CP	CW = TP - CP	CCW = 200 - CW	NP = CP
B = 0 = 0	0	-100 = 0 - 100	300 = 200 - (-100)	0	-100 = 0 - 100	100 = (-100)(-1)	0
R = 1 = OBT	50	-50 = 50 - 100	250 = 200 - (-50)	1	-50 = 50 - 100	50 = (-50)(-1)	1
W = 2 = OBT	100	0 = 100 - 100	200 = 200 - 0	2	0 = 100 - 100	200 = 200 - 0	2
S = 3 = OBT	150	50 = 150 - 100	150 = 200 - 50	3	50 = 150 - 100	150 = 200 - 50	3

CP = S = 3 * 50 = 150	TP = OBT * 50	CW = TP - CP	CCW = 200 - CW	NP = CP	CW = TP - CP	CCW = 200 - CW	NP = CP
B = 0 = 0	0	-150 = 0 - 150	350 = 200 - (-150)	0	-150 = 0 - 150	150 = (-150)(-1)	0
R = 1 = OBT	50	-100 = 50 - 150	300 = 200 - (-100)	1	-100 = 50 - 150	100 = (-100)(-1)	1
W = 2 = OBT	100	-50 = 100 - 150	250 = 200 - (-50)	2	-50 = 100 - 150	50 = (-50)(-1)	2
S = 3 = OBT	150	0 = 150 - 150	200 = 200 - 0	3	0 = 150 - 150	200 = 200 - 0	3


200
CP = B = 0
TP = OBT * 50
CW = TP - CP
CCW = 200 - CW
NP = CP
B = 0 - 0, 200
A = 1 - OBT
W = 2 - OBT
S = 3 - OBT
CCW = CW (-1)
Mina's code
B = 4 A = 1 W = 2 S = 3
if (CP = 4) {
3
CP = 0
TP = OBT * 50
NP = TP
CW = TP - CP
CCW = TP - CP
if (CW < 200 && > 0)
TURN CW (CW)
else if (CW > 0)
NCCW = CW (-1)
if (CW = CCW || CW = NCCW)
if (direction gain CW)
TURN CW (CW)
else
TURN CCW (CW)


B
Current Pc
0
0
TURN
Target Pos
B
0
0
0
A
1
1
1
50
W
2
2
2
100
S
3
3
3
150
A
Current Pc
1
1
Target Pos
B
0
0
-1
-50
A
1
1
0
0
W
2
2
1
-50
S
3
3
2
100
W
Current Pc
2
2
Target Pos
B
0
0
-2
-100
A
1
1
-1
-50
W
2
2
0
0
S
3
3
1
50
S
Current Pc
3
3
Target Pos
B
0
0
-3
-150
A
1
1
-2
-100
W
2
2
-1
-50
IF CURRNT POSITION = PASSED IN VALUE
NOTHING
ELSE
TARGET POSITION = PASSED IN VALUE
TURN= TARGET POSITION - CURRENT POSITION
IF (TURN=3 OR TURN= -1)
IF DIRECTION =1
DIRECTION =-1
GO 50 STEPS
ELSE
DIRECTION =1
GO 50 STEPS
ELSE IF (TURN=2 OR TURN=-2)
GO 100 STEPS
ELSE
GO 50 STEPS
CURRENT POSITION = TARGET POSITION
50
TARGET POSITION = PASSED IN VALUE
TURN= TARGET POSITION - CURRENT POSITION
IF (TURN=3 OR TURN= -1)
IF DIRECTION =1
DIRECTION =-1
GO 50 STEPS
ELSE
DIRECTION =1
GO 50 STEPS
ELSE IF (TURN=2 OR TURN=-2)
GO 100 STEPS
ELSE IF (TURN=1 OR TURN=-3)
GO 50 STEPS
ELSE
NOTHING
CURRENT POSITION = TARGET POSITION

Appendix E - Full Code

```
/* ###########################################################################
# PROJECT: Final Project
# GROUP: 5
# NAME 1: Jesse Bertucci Bertucci V01007526
# NAME 2: Carmina Rocheleau V01021553
# DESC: This program rotates a stepper motor and initializes PWM
########################################################################### */


#include <stdlib.h> // header of the general-purpose standard library of C
programming language
#include <avr/io.h> // header of I/O port
#include <util/delay_basic.h> // header of LCD display
#include <avr/interrupt.h> // header of interrupts
#include "lcd.h"
#include "LinkedQueue.h"


// Global Variable
volatile char STATE;

volatile int currentPosition; // used to keep track of stepper motor position

//uint8_t stepperArray[] = {0b00110101, 0b00101101, 0b00101110, 0b00110110};
uint8_t stepperArray[] = {0b00110101, 0b00110110, 0b00101110, 0b00101101};

volatile unsigned int ADC_result; // initialization for 10-bit ADC result value
volatile unsigned int ADC_result_flag;

volatile int stopGo; // 0 = stop, 1= go

volatile unsigned int lowestADCvalue; // stores lowest ADC value for a given
object

volatile unsigned int home_flag;
volatile unsigned int stop_flag;
volatile int direction;

volatile int stepPosition;
```

```
volatile int itemsOnBelt;
volatile int black_inBucket;
volatile int white_inBucket;
volatile int aluminium_inBucket;
volatile int steel_inBucket;

volatile int sizeOfQueue;

volatile unsigned int EX_flag;


/* System clock configured to 8Mhz using prescaler value. Timer
runs on CPU Clock. System clock has been pre-scaled by 2 */
void mTimer(int count)
{
	int i;				// integer to keep track of how many
times the timer counter has reset
	i = 0;				// initialize integer to 0

	TCCR1B |= _BV(WGM12);		// setting WGM to Mode 4: Clear Timer
on Compare Match (CTC) Mode
	OCR1A = 0x03E8;			// setting Output Compare Register to
1000
	TCNT1 = 0x0000;			// setting initial value of timer
counter to 0
	//TIMSK1 = TIMSK1 | 0b00000010;	  // enable output compare interrupt
	TIFR1 |= _BV(OCF1A);		// clears the Output Compare A match
flag

	while(i < count)
	{
		if((TIFR1 & 0x02) == 0x02)	// checks to see if Output Compare A
match flag has been set (timer counter reacted max value)
		{
			TIFR1 |= _BV(OCF1A);	// clears the Output Compare A
match flag indicating event has been handled
			i++;			// increment integer to track
number of clock cycles
		}
	}

	return;
```

```
}

/* Moves the stepper motor in clock-wise direction for a user
specificed number of steps, 1 step == 1.8 degrees */
void stepCW(int steps, int delay)
{
	for (int i = 0; i < steps; i++) // sets the amount of steps the motor
increment
	{
		//stepsPosition
		stepPosition++; // update current position
		if (stepPosition == 4) // current position cannot exceed '3'
		{
			stepPosition = 0; // current position rolls over to '0' if
current position exceeds '3'
		}

		PORTA = stepperArray[stepPosition]; // send value to stepper motor
controller

		mTimer(delay); // delay to allow for stepper motor to complete
previous command
	}

	return;
}

/* Moves the stepper motor in counter-clock-wise direction for a user
specificed number of steps, 1 step == 1.8 degrees */
void stepCCW(int steps, int delay)
{
	for (int i = 0; i < steps; i++) // sets the amount of steps the motor
increment
	{

		stepPosition--; // update current position
		if (stepPosition == -1) // current position drop below '0'
		{
			stepPosition = 3; // current position rolls back to '3' if
current position drops below '0'
		}
```

```
            PORTA = stepperArray[stepPosition]; // send value to stepper motor
controller

            mTimer(delay); // delay to allow for stepper motor to complete
previous command
      }

      return;
}

/* Initializes current position */
void initializeStepper()
{

      home_flag = 0;
      while(home_flag == 0)
      {
            stepCW(1, 20); // move motor to set next position
      }

      stepCCW(10, 20); // adjust home for station

      EIMSK &= ~(_BV(INT3)); // enable INT3 external interrupt mask register
for HE sensor

      return;
}

void initializePWM()
{
  /*
  Timer/Counter Control Register A
  - Timer 0 set to Fast PWM mode
  - TOP set to 0xFF
  - OCRA update set to TOP
  - TOV flag set to MAX
  - Page 126 to 128
  */
  TCCR0A |= (1 << WGM00) | (1 << WGM01);

  /*
  Timer/Counter Interrupt Mask Register
```

```
  - Enable output compare interrupt (OCIE0A) for Timer 0
  - Page 131
  */
  //TIMSK0 |= (1 << OCIE0A);

  /*
  Timer/Counter Control Register A
  - Set compare match output mode to clear on compare match
  - Clear OC0A on Compare Match, set at BOTTOM
  - Page 126
  */
  TCCR0A |= (1 << COM0A1);

  /*
  Timer/Counter Control Register B
  - Set prescaler in TCCR0B to 64 to get 3.9 kHz
  - Page 129 to 130
  */
  TCCR0B |= (1 << CS01) | (1 << CS00);

  /*
  Output Compare Register A
  - Set to 50% duty cycle
  - 0xFF (127 out of 255)
  - Page 130
  */
  OCR0A = 0x78; // initialize to 120(dec)
}

void initializeMotor()
{
      DDRB = 0xff; // sets ENA and ENB to high

      PORTB = 0b00001110;    // Set PB1 to low

      stopGo = 1;
}

void trapezoidalMove(int steps, int direction)
{
      int accelSteps = steps / 8; // acceleration steps
      int decelSteps = steps / 8; // deceleration steps
```

```
      int constSpeedSteps = steps - (accelSteps + decelSteps); // constant
speed steps

      int maximumDelayConst = 20;
      int maximumDelay = maximumDelayConst; // slowest speed
    int minimumDelay = 8; // fastest speed
    float accelDelayIncrements = (float)(maximumDelay - minimumDelay) /
accelSteps;
    float decelDelayIncrements = (float)(maximumDelay - minimumDelay) /
decelSteps;

    // Acceleration Phase
      for (int i = 0; i < accelSteps; i++)
      {
          if (direction == 1)
            {
                  stepCW(1, maximumDelay);
            }
            else
            {
                  stepCCW(1, maximumDelay);
          }

            maximumDelay -= (int)accelDelayIncrements;
            if(maximumDelay < minimumDelay)
            {
                  maximumDelay = minimumDelay;
            }
    }

      // constant speed
      if (direction == 1)
      {
            stepCW(constSpeedSteps, minimumDelay);
      }
      else
      {
            stepCCW(constSpeedSteps, minimumDelay);
      }

    // Deceleration Phase
    for (int i = 0; i < decelSteps; i++)
```

```
      {
          if (direction == 1)
            {
                stepCW(1, maximumDelay);
            }
            else
            {
                stepCCW(1, maximumDelay);
          }

          maximumDelay += (int)decelDelayIncrements;  // Increase delay to
reduce speed
             if(maximumDelay > maximumDelayConst)
             {
                    maximumDelay = maximumDelayConst;
             }
    }
}









// optical sensor OR
ISR(INT0_vect)
{
      if((PIND & 0x01) == 0x01)
      {
            ADCSRA |= _BV(ADSC); // start conversion
            STATE = 2; // go to reflective state after this interrupt is done
            lowestADCvalue = 1023; // reinitializing lowest 10-bit value to
```

```
highest possible value
	}
}

// optical sensor EX
ISR(INT1_vect)
{
		if((PIND & 0x02) == 0x00)
		{
			EX_flag = 1; // used to track EX interrupts globally
			PORTB |= (1 << PB0) | (1 << PB1); // brake
			stopGo = 0; // belt is now stopped
			STATE = 1; // goto end END_OF_BELT_STAGE
		}
}

// button interrupt to start and stop belt
ISR(INT2_vect)
{
	mTimer(20);
	if(((PIND & 0x04)) == 0x04)
	{
		STATE = 3; // PAUSE_STAGE

		while(((PIND & 0x04)) == 0x04);
		mTimer(20);
	}
}

// optical sensor HE
ISR(INT3_vect)
{
	home_flag = 1;
	currentPosition = 0;
}

// shutdown button
ISR(INT4_vect)
{
	mTimer(20);
	if(((PINE & 0x10)) == 0x10)
	{
```

```
            stop_flag = 1;
            LCDWriteStringXY(0, 0, "SHUTTING DOWN ... :");
            while(((PINE & 0x10)) == 0x10);
            mTimer(20);
      }
}

// ADC end of interrupt
ISR(ADC_vect)
{
      ADC_result = ADC;
      ADC_result_flag = 1;
}




/*************************************************************************
********
* DESC: initializes the linked queue to 'NULL' status
* INPUT: the head and tail pointers by reference
*/
void setup(link **h,link **t)
{
      *h = NULL;         /* Point the head to NOTHING (NULL) */
      *t = NULL;         /* Point the tail to NOTHING (NULL) */
      return;
}

/*************************************************************************
********
* DESC: This initializes a link and returns the pointer to the new link or NULL
if error
* INPUT: the head and tail pointers by reference
*/
void initLink(link **newLink)
{
      //link *l;
      *newLink = malloc(sizeof(link));
      (*newLink)->next = NULL;
      return;
```

```
}

/*************************************************************************
**********
*  DESC: Accepts as input a new link by reference, and assigns the head and
tail
*  of the queue accordingly
*  INPUT: the head and tail pointers, and a pointer to the new link that was
created
*/
void enqueue(link **h, link **t, link **nL)
{

      if (*t != NULL)
      {
            /* Not an empty queue */
            (*t)->next = *nL;
            *t = *nL; //(*t)->next;
      }/*if*/
      else
      {
            /* It's an empty Queue */
            //(*h)->next = *nL;
            //should be this
            *h = *nL;
            *t = *nL;
      }/* else */
      return;
}

/*************************************************************************
********
* DESC : Removes the link from the head of the list and assigns it to
deQueuedLink
* INPUT: The head and tail pointers, and a ptr 'deQueuedLink'
*            which the removed link will be assigned to
*/
void dequeue(link **h, link **t, link **deQueuedLink)
{
      *deQueuedLink = *h;      // Will set to NULL if Head points to NULL

      /* Ensure it is not an empty queue */
```

```
	if (*h != NULL)
	{
		*h = (*h)->next;

		if (*h == NULL) // If the queue becomes empty after this dequeue,
set the tail to NULL
		{
			*t = NULL;
		}
	}

	return;
}

/*****************************************************************************
********
* DESC: Peeks at the first element in the list
* INPUT: The head pointer
* RETURNS: The element contained within the queue
*/
element firstValue(link **h)
{
	return((*h)->e);
}

/*****************************************************************************
********
* DESC: deallocates (frees) all the memory consumed by the Queue
* INPUT: the pointers to the head and the tail
*/
void clearQueue(link **h, link **t)
{

	link *temp;

	while (*h != NULL)
	{
		temp = *h;
		*h=(*h)->next;
		free(temp);
	}/*while*/
```

```
	/* Last but not least set the tail to NULL */
	*t = NULL;

	return;
}

/*************************************************************************
********
* DESC: Checks to see whether the queue is empty or not
* INPUT: The head pointer
* RETURNS: 1:if the queue is empty, and 0:if the queue is NOT empty
*/
char isEmpty(link **h)
{
	/* ENTER YOUR CODE HERE */
	return(*h == NULL);
}

/*************************************************************************
********
* DESC: Obtains the number of links in the queue
* INPUT: The head and tail pointer
* RETURNS: An integer with the number of links in the queue
*/
int size(link **h, link **t)
{

	link	*temp;		/* will store the link while traversing the
queue */
	int	numElements;

	numElements = 0;

	temp = *h;		/* point to the first item in the list */

	while(temp != NULL)
	{
		numElements++;
		temp = temp->next;
	}/*while*/

	return(numElements);
```

```
}




int main(int argc, char *argv[])
{
	// config clock ============================================================
	CLKPR = 0x80;
	CLKPR = 0x01;			//  sets system clock to 8MHz
	TCCR1B |= _BV(CS11); // configure timer counter control register 1


	// config i/o ==============================================================
	DDRA = 0b11111111; // servo
	DDRB = 0b11111111; // stepper motor?
	DDRC = 0b11111111; // LCD
	DDRD = 0b11110000; // Going to set up INT2 & INT3 on PORTD
	DDRL = 0b11111111; // used for LEDs to troubleshoot


	// config inturupts
==========================================================
	cli();		// Disables all interrupts

	EIMSK |= (_BV(INT0)); // enable INT0 external interrupt mask register for
OR sensor
	EICRA |= (_BV(ISC01) | _BV(ISC00)); // external interrupt control
register for bit 0 with the rising edge of INTn generates asynchronously an
interrupt request

	EIMSK |= (_BV(INT1)); // enable INT1 external interrupt mask register for
EX sensor
	EICRA |= (_BV(ISC11)); // Falling edge for INT1

	EIMSK |= (_BV(INT2)); // enable INT2 external interrupt mask register for
OR sensor
	EICRA |= (_BV(ISC21) | _BV(ISC20));  // external interrupt control
register for bit 2 with the rising edge of INTn generates asynchronously an
interrupt request
```

```
	EIMSK |= (_BV(INT3)); // enable INT3 external interrupt mask register for
HE sensor
	EICRA |= (_BV(ISC31) | _BV(ISC30)); // external interrupt control
register for bit 3 with the rising edge of INTn generates asynchronously an
interrupt request

	EIMSK |= (_BV(INT4)); // enable INT4 external interrupt mask register for
OR sensor
	EICRB |= (_BV(ISC41) | _BV(ISC40)); // external interrupt control
register for bit 0 with the rising edge of INTn generates asynchronously an
interrupt request

	// config ADC ============================================================
	// by default, the ADC input (analog input is set to be ADC0 / PORTF0
	//ADCSRA is the ADC control and status register, we are using register A
	ADCSRA |= _BV(ADEN); // enable ADC
	ADCSRA |= _BV(ADIE); // enable interrupt of ADC
	ADMUX |= _BV(REFS0); // configures reference voltage for ADC
	lowestADCvalue = 0; // initialize ADC values

	sei();// Enable all interrupts


	// config LCD ============================================================
	InitLCD(LS_BLINK|LS_ULINE); //Initialize LCD module
	LCDClear();


	// config motors
=========================================================
	PORTB = 0b00001111; // brake, idk why?
	initializePWM();
	initializeStepper();
	initializeMotor();



	// initialize FIFO
=========================================================
	link *head;              /* The ptr to the head of the queue */
	link *tail;              /* The ptr to the tail of the queue */
	link *newLink;           /* A ptr to a link aggregate data type (struct)
```

```
*/
	link *rtnLink;			/* same as the above */
	element eTest;			/* A variable to hold the aggregate data type
known as element */

	rtnLink = NULL; // initialize pointers
	newLink = NULL; // initialize pointers

	setup(&head, &tail); // setup FIFO pointers

	// initialize FIFO
============================================================
	STATE = 0; // POLLING stage
	goto POLLING_STAGE; // start



	// POLLING STATE is default state and contains the different cases that
can be called within our program
	POLLING_STAGE:
	{
		switch(STATE)
		{
			case (0) :
			goto POLLING_STAGE;
			break;
			case (1) :
			goto END_OF_BELT_STAGE;
			break;
			case (2) :
			goto REFLECTIVE_STAGE;
			break;
			case (3) :
			goto PAUSE_STAGE;
			break;
			case (4) :
			goto END;
			default :
			goto POLLING_STAGE;
		}
	}
```

```
      //identify objects when EX sensor is triggered buy pulling objects form
the FIFO
      END_OF_BELT_STAGE:
      {
            EX_flag = 0;

            dequeue(&head, &tail, &rtnLink); // remove item from FIFO
            unsigned int objectType = rtnLink->e.itemCode; // stores value
pulled from FIDO
            sizeOfQueue=size(&head, &tail);
            free(rtnLink); // free memory location

            if(objectType == 0)
            {
                  black_inBucket++;
            }
            else if(objectType == 1)
            {
                  aluminium_inBucket++;
            }
            else if(objectType == 2)
            {
                  white_inBucket++;
            }
            else
            {
                  steel_inBucket++;
            }

            int targetPosition = objectType; // pulling object form the q

 // modular math to remove negative values and return a value that can be
easily sorted into what direction it should turn.
            int turn = (targetPosition - currentPosition + 4) % 4;
            if (turn == 0)
            {
                  // Do nothing
            }
            else if (turn <= 2)
            {
                // move same direction already going by 50 or 100 steps
                  trapezoidalMove( (turn * 50), 1);
```

```
            }
           else
           {
                  //move opposite direction by 50 steps
                 trapezoidalMove( ((4 - turn) * 50), 0);
            }
               currentPosition = targetPosition;
           PORTB = 0b00001110;     // start belt in CW direction
           mTimer(120);
           stopGo = 1; // belt is now running

           if((stop_flag == 1) && (sizeOfQueue == 0))
           {
                 mTimer(1000);
                 STATE = 4; // END STAGE
                 LCDWriteStringXY(0,0,"here5");
           }
           else
           {
                 if(EX_flag == 1) // If another EX interupt is triggered
                 {
                       STATE = 1; // END_OF_BELT STAGE
                 }
                 else
                 {
                       STATE = 0; // POLLING_STAGE
                 }
           }

           goto POLLING_STAGE;
     }




     /* shows that an object has been detected by the optical sensor and than
is given a
     value by the based of the of the reflective sensor. these values are than
categorized and placed into a q
      */
     REFLECTIVE_STAGE:
```

```
      {
            while(((PIND & 0x01)) == 0x01) // while optical sensor is high
            {
                  if(ADC_result_flag == 1) // checks if the ADC has been
triggered and a value had been read
                  {
                        if(ADC_result < lowestADCvalue)  // if the value you
of ADC is lower than previous value it will reset the lowest value you and
store it. this value you is given by the reflective sensor and indicated what
type of material we have
                        {
                              lowestADCvalue = ADC_result;
                        }

                        ADC_result_flag =  0x00; // re sets the flag
                        ADCSRA |= _BV(ADSC); // start conversion analog to
digital
                  }
            }

            // Bl = 0
            // Al = 1
            // Wt = 2
            // St = 3

            initLink(&newLink); // Initialize new link "gives new box"

            /*these if statements allow us to categorize the material based on
measured reflective values */
            if(lowestADCvalue < 200) // this is Al
            {
                  newLink->e.itemCode = 1; // this adds a int value from the
tail pointer in FIFO
            }
            else if( (lowestADCvalue > 200) && (lowestADCvalue < 700) ) //
this is steel
            {
                  newLink->e.itemCode = 3; // this adds a int value from the
tail pointer in FIFO
            }
            else if( (lowestADCvalue >= 700) && (lowestADCvalue < 915) ) //
this is white
```

```
            {
                  newLink->e.itemCode = 2; // this adds a int value from the
tail pointer in FIFO
            }
            else // this is black
            {
                  newLink->e.itemCode = 0; // this adds a int value from the
tail pointer in FIFO
            }

            enqueue(&head, &tail, &newLink); // this passes values to the FIFO
            sizeOfQueue=size(&head, &tail);
            STATE = 0;
            goto POLLING_STAGE;
      }

      PAUSE_STAGE:
      {
            if(stopGo == 0) // if belt stopped
            {
                  PORTB = 0b00001110;    // start belt in CW direction
                  stopGo = 1; // belt is now running

                  LCDClear();
            }
            else // if belt running
            {
                  PORTB |= (1 << PB0) | (1 << PB1); // brake
                  stopGo = 0; // belt is now stopped

                  LCDClear();

                  LCDWriteStringXY(0, 0, "B:   A:   OB:");
                  LCDWriteIntXY(2, 0, black_inBucket, 2);
                  LCDWriteIntXY(7, 0, aluminium_inBucket, 2);
                  LCDWriteIntXY(13,0, sizeOfQueue, 2);

                  LCDWriteStringXY(0,1, "W:   S:   T1:");
                  LCDWriteIntXY(2, 1, white_inBucket, 2);
                  LCDWriteIntXY(7, 1, steel_inBucket, 2);
                  LCDWriteIntXY(13, 1, (black_inBucket + aluminium_inBucket +
white_inBucket + steel_inBucket), 2);
```

```
            }

            STATE = 0;
            goto POLLING_STAGE;
      }









      END:
      {
            LCDClear();

            PORTB |= (1 << PB0) | (1 << PB1); // brake
            stopGo = 0; // belt is now stopped

            LCDWriteStringXY(0, 0, "B:   A:   OB:");
            LCDWriteIntXY(2, 0, black_inBucket, 2);
            LCDWriteIntXY(7, 0, aluminium_inBucket, 2);
            LCDWriteIntXY(13,0, sizeOfQueue, 2);

            LCDWriteStringXY(0,1, "W:   S:   T1:");
            LCDWriteIntXY(2, 1, white_inBucket, 2);
            LCDWriteIntXY(7, 1, steel_inBucket, 2);
            LCDWriteIntXY(13, 1, (black_inBucket + aluminium_inBucket +
white_inBucket + steel_inBucket), 2);

            return(0);
      }
}


// If an unexpected interrupt occurs (interrupt is enabled and no handler is
installed,
// which usually indicates a bug), then the default action is to reset the
device by jumping
```

```
// to the reset vector. You can override this by supplying a function named
BADISR_vect which
// should be defined with ISR() as such. (The name BADISR_vect is actually an
alias for __vector_default.
// The latter must be used inside assembly code in case <avr/interrupt.h> is
not included.
ISR(BADISR_vect)
{
	// user code here
}
```