

THE UNIVERSITY OF VICTORIA

DEPARTMENT OF MECHANICAL ENGINEERING



University of Victoria

MECH 458 – Mechatronics

Final Project - Final Report

Ben Claydon - V01005333

Benjamin Kappers - V01009297

Abstract

This report discusses the design, implementation, and optimization of an object sorting mechatronic system. The system utilizes a conveyor belt driven by a DC motor, a sorting mechanism actuated by a stepper motor, and multiple sensors to detect, classify, and sort objects based on their material. An optical sensor and an analog reflective sensor are used to classify the objects, and a second optical sensor is used to determine when an object is ready to be sorted. The system is controlled using an ATmega2560 microcontroller and operates using an interrupt-driven state machine program to achieve real-time responsiveness.

Object classification is achieved by sampling analog reflective results and comparing them against calibrated bounds for aluminum, steel, white plastic, and black plastic. A FIFO data structure is used to track and organize object order and ensure correct sorting. Stepper motor motion is optimized using a precomputed quintic s-curve function to reduce mechanical stress and prevent missed steps.

System performance was evaluated based on sorting speed and accuracy. Through testing and calibration, optimal system parameters were determined to balance speed and reliability. During the final demonstration, the system was able to sort 48 objects with a 98% accuracy in 39 seconds. By restructuring the software to allow for simultaneous interrupts, the system speed could be increased. Additionally, the system has physical limitations that restrict further increases in performance, primarily due to motor torque constraints, sensor sampling rates, and mechanical delays such as object drop time. Increasing conveyor speed or stepper acceleration beyond optimal values introduced a higher risk of misclassification and mechanical errors, including missed steps and improper bucket alignment. As a result, a balance between speed and reliability was necessary to ensure consistent operation, highlighting the trade-offs inherent in real-world mechatronic system design.

Table of Contents

1	Project Overview	1
1.1	Project Description.....	1
1.2	Objectives	1
2	System Technical Documentation.....	3
2.1	Circuit Diagram	4
2.2	Flow Chart and Algorithm	5
2.2.1	Initialization and Polling.....	6
2.2.2	Reflective Stage	7
2.2.3	Bucket Stage	8
2.2.4	Stepper Motor Motion and Acceleration.....	9
2.2.5	Timers and PWM	10
2.2.6	Pause/Resume	13
2.2.7	Ramp Down	13
3	System Performance Specifications.....	14
4	Testing and Calibration Procedures	15
5	Limitations and System Trade-offs	16
6	Experience and Recommendations	17
6.1	Cooling System for the Motor Driver.....	17
6.2	Full Set of Test Pieces for Each Station.....	17
6.3	Closed Buckets and More Powerful Stepper Motor	17
7	Conclusions.....	18
8	References.....	19
9	APPENDIX A – Full Code	20
9.1	Main.c	20
9.2	Functions.C	27
9.3	Functions.h.....	34
9.4	Final_Stepper_Curve.py	35

List of Figures

Figure 1. SPI vs. Tmax and I. Generated using python script by OpenAI, ChatGPT 5 [2].....	2
Figure 2. System Block Diagram.....	3
Figure 3. System Circuit Diagram	4
Figure 4. System Flow Chart	5

List of Tables

Table 1 System Performance Parameters.....	15
Table 2 Optimal Parameters.....	15

List of Listings

Listing 1. Polling stage switch	7
Listing 2. Interrupt 2 ISR	7
Listing 3. Reflective stage general structure	8
Listing 4. Object Classification.....	8
Listing 5. Interrupt 3 ISR	9
Listing 6. Bucket stage stepper logic	9
Listing 7. Counterclockwise stepper function	10
Listing 8. Fifty step delay array	10
Listing 9. PWM Generation.....	11
Listing 10. Change Duty Cycle Function	11
Listing 11. mTimer function	12
Listing 12. Non-blocking millis functions	12
Listing 13. Microsecond delay timer	13
Listing 14. Pause/Resume logic.....	13
Listing 15. Ramp down stage conditions.....	14
Listing 16. Quintic S-Curve.....	14
Listing 17. Calibration bounds.....	16
Listing 18. Drop time buffer	17

1 Project Overview

1.1 Project Description

The objective of this project is to develop a full mechatronic system that is capable of inspecting, classifying, and sorting physical objects with maximum accuracy and speed. For this project, the system must use a conveyor belt to sort specific objects into their corresponding bins at the end of the conveyor belt. The system consists of multiple sensors and actuators that can be used to complete the required tasks. A DC motor is attached to the conveyor belt to give control over speed and direction. The bins are attached to a stepper motor to control the orientation and ensure the correct bin faces the end of the conveyor belt. Two optical sensors and one reflective sensor are used throughout the process. One optical sensor and the reflective sensor are used to identify and classify the objects as they travel down the conveyor belt, and the second optical sensor is used to determine when an object reaches the end of the conveyor belt. Additionally, a hall effect sensor is used for stepper motor orientation calibration. An LCD screen is used to display appropriate information throughout the process, and green LEDs are used for debugging and displaying additional information. The motors, sensors, and the LCD will be controlled by an ATmega2560 MCU.

A given number of objects will be placed on the belt in rapid succession, with a minimum of four objects at a time. There are four different types of objects: aluminum, steel, white plastic, and black plastic. Each object can be categorized based on its reflective properties by creating a corresponding range of lowest sensor values for each object type. After categorizing the objects, the conveyor belt motor drives the belt to transport the objects to the bucket. As they are transported, the stepper motor must rotate 90, or 180 degrees, either clockwise or counterclockwise to place the correct bucket under the belt. When the second optical sensor indicates the object has reached the end of the belt, and the bucket has assumed the correct position, the conveyor belt motor is turned on again to drop the object into the bucket. This process repeats for each piece. For multiple pieces in a set, the system must wait with each piece at the end of the belt until the bucket has time to rotate into the correct position.

1.2 Objectives

Our final design objective is to sort all objects in groups of 8 in under 30 seconds. However, this objective cannot be met until we first meet a set of preliminary criteria. The objectives for milestones in the project are listed below:

- Calibration: A range of values is obtained by passing all objects of one material through the sensor. This is repeated to for each material.
- Sort one object at a time: Categorize the material of that object, move it down the conveyor belt, rotate the bucket, and drop the object into the correct bucket.
- Sort four objects at a time: With four objects placed adjacently on the belt, categorize all four objects with the optical sensor, store the order and category of each object, move all

objects down the belt, then rotate the bucket and drop (repeating for each of the four objects).

- Sort eight objects at a time: Same process, but more data to store, and more possible complications.
- Optimize: optimize the algorithm for faster operation and lower error to achieve the goal of 30s completion time.

In Figure 1 below we graphed the system performance index (SPI) with respect to the maximum task completion time (Tmax) and the number of errors (I). The objective of this project is to achieve the maximum SPI. During the optimization phase, we can use the graph to balance system accuracy and speed for maximum SPI.

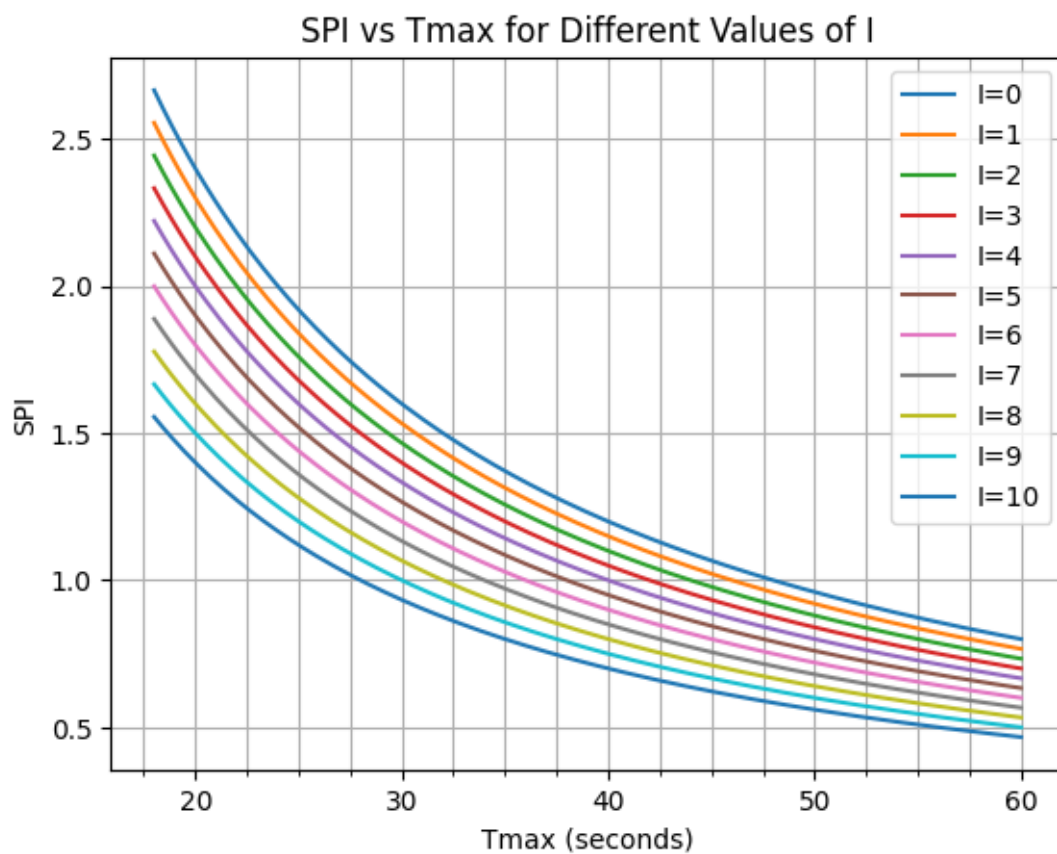


Figure 1. SPI vs. Tmax and I. Generated using python script by OpenAI, ChatGPT 5 [2]

2 System Technical Documentation

The system is built around an AVR microcontroller interfacing with a range of sensors and actuators to enable real-time object sorting. Object detection is performed using an optical reflectance sensor connected to the microcontroller's ADC, allowing material classification based on reflected light intensity. Position and event detection are handled through external interrupt pins connected to mechanical buttons and sensor triggers, ensuring immediate response to object arrival and user inputs. Actuation is achieved through a DC motor driving the conveyor belt via PWM control and a stepper motor used to rotate the sorting plate to defined quadrants. A general block diagram of the system is given in Figure 2 below.

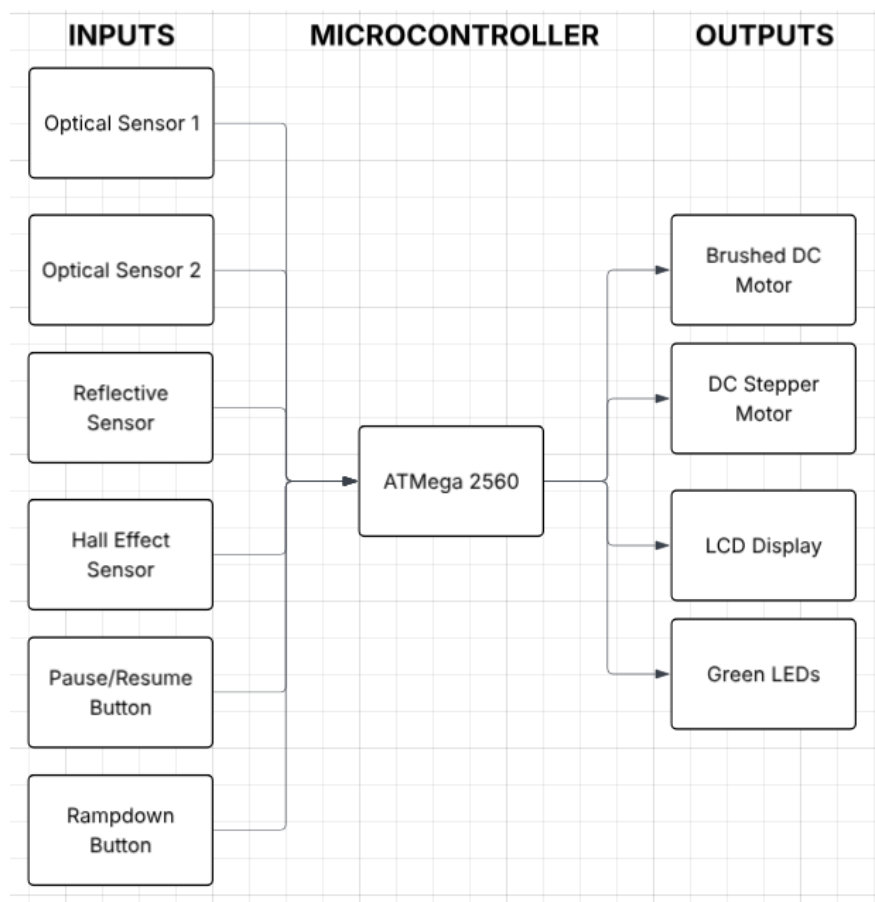


Figure 2. System Block Diagram

2.1 Circuit Diagram

The figure below shows the complete circuit diagram used to make the necessary electrical connections for the functioning system. The diagram was adapted from a figure provided in the Final Project: Milestone 5 slides created by Y. Shi [1]. The original diagram was modified to reflect the implementation used in this project. These modifications included removing unused sensors, adding an additional optical sensor, hall effect sensor, and relocating the buttons.

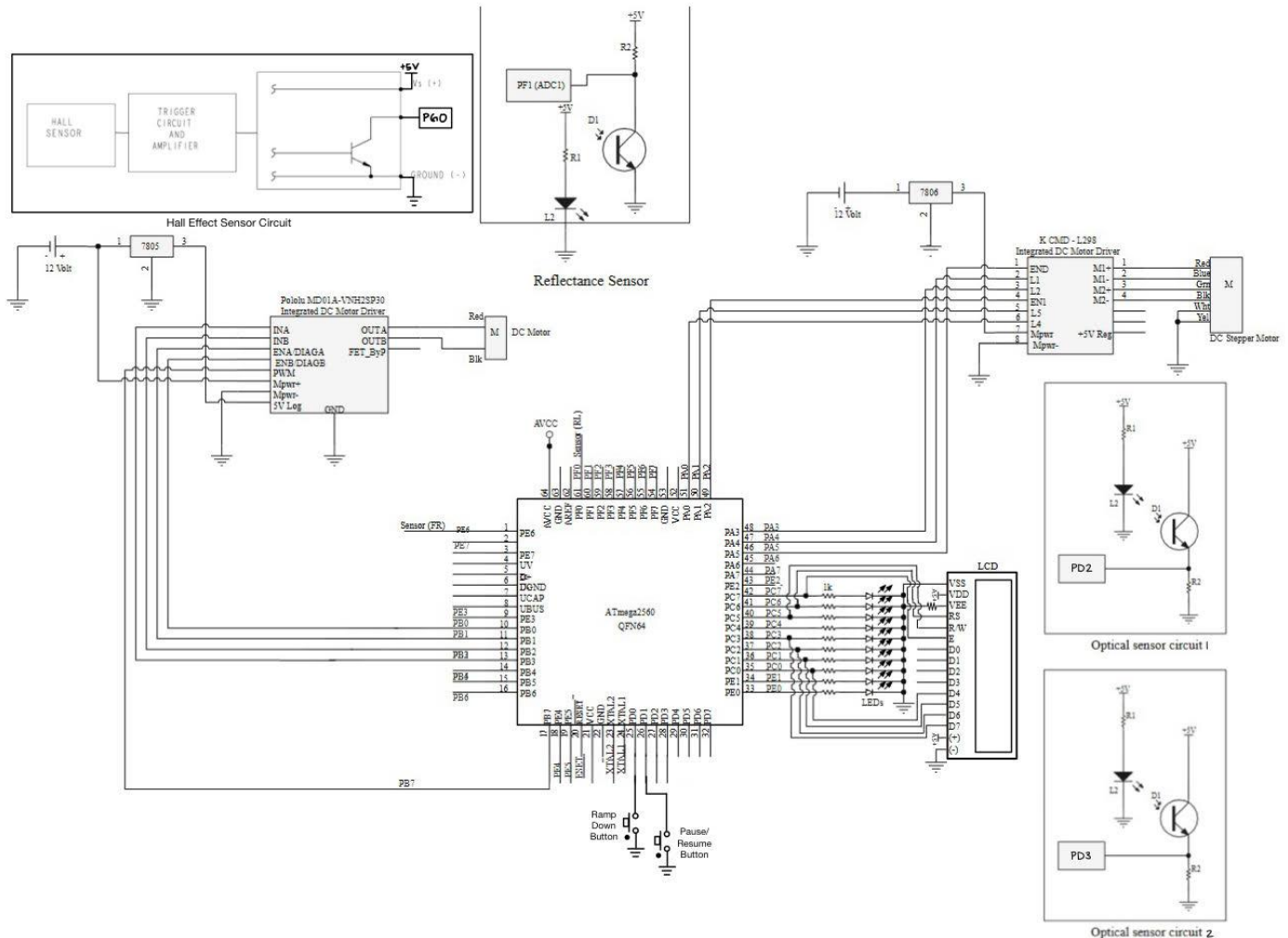


Figure 3. System Circuit Diagram

2.2 Flow Chart and Algorithm

The software is designed as an interrupt-driven state machine, where the main loop handles high level system states while external interrupts manage real time events such as object detection, user input, and sensor conversions. The state machine is broken up into three major stages: polling stage, reflective stage, and bucket stage. There are additional states for a pause/resume function, and a ramp down function. Figure 4 below is a high-level flowchart that provides a simplified and clear visual representation of the algorithm used by the system to complete the object sorting.

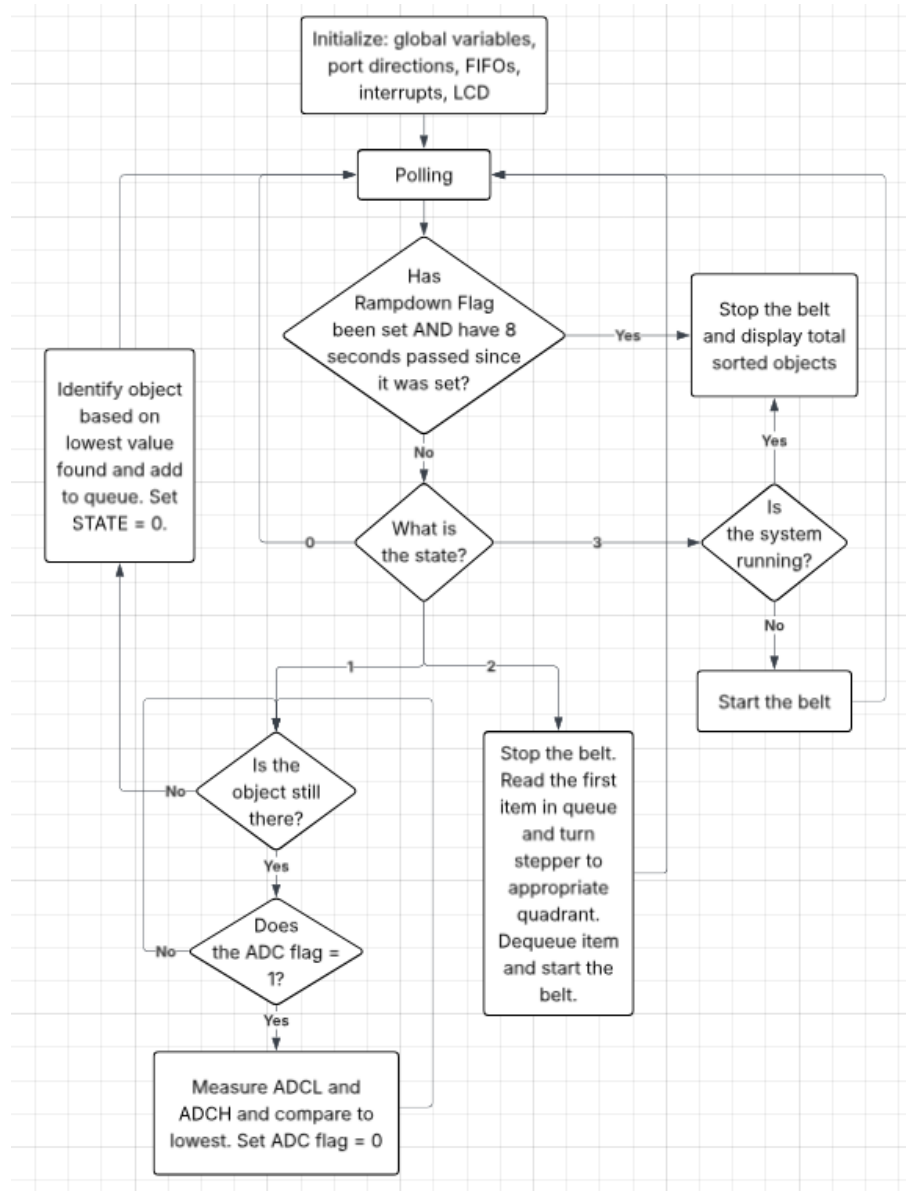


Figure 4. System Flow Chart

2.2.1 Initialization and Polling

The first portions of the code that get executed are initializations. All the global variables that will be used throughout the process need to be declared and set, ports directions need to be configured, timers need to be initialized, the FIFOs must be created, and interrupts need to be set up. After all the necessary initializations have been completed, the system can enter its polling stage.

The global variables declared at the beginning of the program are used for a wide variety of purposes. There are variables for the ADC, Pause/Resume and Ramp down functions, stepper motor, timers, object identification process, object counters, and calibration bounds. Most of these variables are set to zero or simply declared and set later in the program. However, the variables used for the calibration bounds are set to unique values and are tuned before running the system for the first time on a given day.

Ports A, B, C, D, L, and G are configured to be either input or output, depending on what is required. The stepper motor is connected to Port A, so each pin is set to be an output. Port B also has each pin set as an output, as the DC motor is connected to it. Port C is used to send signals to the LCD display, so each of its pins are outputs. The four hardware interrupts used are all connected to Port D, so the first four pins of Port D are set as inputs. Port L is connected to the green debugging LEDs, so each of the pins are set as outputs. Finally, the Hall Effect Sensor is connected to Port G, so each of the pins are set as inputs.

The interrupts and the ADC need to be configured appropriately. The ADC is enabled and configured to have left adjusted bits. The triggers for interrupts 0 and 2 are configured to be falling edges, as the button or sensor is active high. The triggers for interrupts 1 and 3 are configured to be rising edges, as the button or sensor is active low. All four interrupts are then enabled.

The system clock and two different millisecond timers are also initialized. The system clock is set at 8 MHz. mTimer is set up on Timer 1 and is a blocking millisecond timer that delays a given number of milliseconds. To achieve this, the prescale is chosen so that Timer 1 runs at 1 MHz. millis is a non-blocking timer set up on Timer 3 that keeps a running count of how many milliseconds have passed. It uses the same prescale so that it runs at 1 MHz.

The LCD is initialized, two empty FIFO queues are created, and the stepper motor is orientated to have the black quadrant face the conveyer belt. PWM is generated at an appropriate duty cycle, which starts the conveyor belt. Immediately after, the program enters the Polling Stage. The polling stage is the heart of the program. It polls the STATE variable and goes to different sections of the program depending on its value, as shown in Listing 1 below.

```

switch (STATE) {
    case (0) :
        goto POLLING_STAGE;
        break;
    case (1) :
        goto REFLECTIVE_STAGE;
        break;
    case (2) :
        goto BUCKET_STAGE;
        break;
    case (3) :
        goto PauseResume_STAGE;
        break;
    case (4) :
        goto RampDown_STAGE;
        break;
    case (6) :
        goto END;
    default :
        goto POLLING_STAGE;
}

```

Listing 1. Polling stage switch

As shown in the above code, if STATE is set to 0, the program remains in polling stage. If it is set to 1, the program enters reflective stage. If STATE is set to 2 or 3, the program enters pause/resume stage or ramp down stage respectively.

2.2.2 Reflective Stage

The role of the reflective stage is to identify and classify each object that travels down the conveyor belt. An optical sensor is used to determine when an object has arrived, and a reflective sensor is used to classify what the object is. Optical sensor 1 is used during this stage and is connected to interrupt 2. The reflective sensor is connected to an ADC port on the ATmega 2560. As the object passes through the reflective sensor, ADC readings will be taken in rapid succession. The lowest digital value recorded will be kept and used to identify the object. When interrupt 2 gets triggered, the following ISR function gets executed.

```

ISR(INT2_vect) {
    // reset the lowest variable, and start the first ADC reading
    lowest = 1023;
    ADC_count = 0;
    ADCSRA |= _BV(ADSC);
    STATE = 1;
}

```

Listing 2. Interrupt 2 ISR

This interrupt fires when an object initially passes through the optical sensor. The lowest variable gets reset to 1023, the highest 10-bit number, the ADC counter variable gets reset to zero, a new conversion is started, and the STATE variable is changed to 1. After this ISR finishes, the code will return to where it was when it was interrupted, which is polling stage. However, now that STATE is equal to 1, the program will be sent to reflective stage. The general structure of reflective stage is shown in Listing 3 below.

```

if(PIND & 0x04){

    /* KEEP TAKING ADC READINGS */

}else{

    /* CLASSIFY OBJECT AND ADD TO QUEUE */

}

```

Listing 3. Reflective stage general structure

The if statement in Listing 3 checks to see if an object is still in the optical sensor and reflective readings need to be taken. If the object is still there, the ADC continues to complete conversions and start new ones once finished. When a conversion is complete, it is compared against the lowest value recorded and the lower of the two is kept. If the if statement is evaluated as false and the else block is entered, it means that the object has left the optical sensor. The program now has all the information needed to classify the object. It compares the lowest value found to calibration bounds that are predetermined, as shown in Listing 4 below. The method of determining those bounds will be discussed in more detail in Section 5.

```

if(lowest >= White_min && lowest <= White_max){
    CubeType = 0;
}
if(lowest >= Aluminum_min && lowest <= Aluminum_max){
    CubeType = 1;
}
if(lowest >= Black_min && lowest <= Black_max){
    CubeType = 2;
}
if(lowest >= Steel_min && lowest <= Steel_max){
    CubeType = 3;
}

```

Listing 4. Object Classification

The CubeType variable can have four unique values. 0 corresponds to white plastic, 1 is aluminum, 2 corresponds to black plastic, and 3 is steel. Once the object has been identified, a new link is created, its itemCode is updated to be the CubeType, and it is added to the FIFO. The STATE variable is changed back to 0, and the program returns to polling stage.

2.2.3 Bucket Stage

The role of the bucket stage is to physically sort the objects. A plate with four quadrants matching the different types of objects is attached to a stepper motor at the end of the conveyor belt. If the correct quadrant is facing the belt, an object will land in the correct quadrant when it falls off. The bucket stage is triggered by a second optical sensor which is connected to interrupt 3. The sensor is located a few centimeters from the end of the conveyor belt. The ISR function that runs when the optical sensor is triggered is shown in Listing 5 below.

```

ISR(INT3_vect){
    PORTB |= _BV(PB0); // sets INA to 1, stopping the belt
    STATE = 2;         // corresponds to bucket stage
}

```

```
}
```

Listing 5. Interrupt 3 ISR

As shown above, when the second optical sensor is triggered, the conveyer belt stops, and the STATE variable is set to 2. The program will then return to polling stage, and the switch will send the program to bucket stage. The first task that needs to be completed in bucket stage is determining if and by how much the stepper needs to turn. The four quadrants on the stepper are equally spaced, so the stepper will only ever have to turn either 90 degrees clockwise, 90 degrees counterclockwise, or 180 degrees. The code that determines this is shown in Listing 6 below.

```
int target_quadrant;
int diff;
target_quadrant = head->e.itemCode;
diff = target_quadrant - current_quadrant;

if(diff == 1 || diff == -3){
    Stepper_CCW(50);           // counter clockwise 90°
}
else if(diff == 2 || diff == -2){
    Stepper_CW(100);           // clockwise 180°
}
else if(diff == 3 || diff == -1){
    Stepper_CW(50);           // clockwise 90°
}
```

Listing 6. Bucket stage stepper logic

Variables target_quadrant, current_quadrant, and diff are used to determine how much to turn the stepper. The itemCode of the first item in the FIFO is read and set as the target_quadrant. The current_quadrant is subtracted from the target_quadrant and stored in diff. Depending on the value of diff, the stepper turns the appropriate amount. The stepper being used is a 200-step motor, so turning 90 degrees requires 50 steps, and turning 180 degrees requires 100 steps. Once the stepper has turned to the correct quadrant, a completion counter is incremented, the link is dequeued, and the belt is started again. The STATE variable is reset to 0, and the program returns to the polling stage.

2.2.4 Stepper Motor Motion and Acceleration

Controlling the stepper motor effectively and efficiently is not a trivial task. There are many optimizations that can be made to increase the speed of the stepper motor. However, there are also limitations that cannot be exceeded without a large loss of accuracy. The function for turning the stepper counterclockwise is shown in Listing 7 below. The function for turning the stepper clockwise is almost identical; the only difference between the logic that tracks the current position.

```
void Stepper_CCW(int steps){
    // full stepping array
    int array[4] = {0b00110110, 0b00101110, 0b00101101, 0b00110101};

    for(int i = 0; i<steps; i++){
        int delay = 10;
```

```

        current_position = (current_position + 3) % 4;
        PORTA = array[current_position];

        if(steps == 50){
            microTimer(fifty_step_delay_array[i]);
        }
        if(steps == 100){
            microTimer(hundred_step_delay_array[i]);
        }
        if(steps == 1){
            mTimer(delay);
        }
    }
}

```

Listing 7. Counterclockwise stepper function

The above software controls the stepper motor using full-step driving. Full-step driving energizes the motor coils in a discrete sequence, rotating the motor by one full step for each change in the excitation pattern. For a given number of steps, the function advances through the stepping sequence that many times. The current position of the stepper is tracked and either incremented or decremented by 1, depending on whether the stepper is rotating clockwise or counterclockwise. The stepper motor is connected to Port A, so the position in the array that corresponds to the current position is sent to Port A. If the steps are triggered too quickly, the motor cannot physically keep up due to inertia and torque limitations, which can cause missed steps or stalling. This is why delays are introduced between each step. For longer movements, like 50 or 100 steps, the program uses delay arrays to implement an acceleration profile. The motor starts with larger delays to allow the rotor to synchronize with the stepping sequence, then gradually reduces the delay to increase step frequency, before increasing delays again for deceleration. The acceleration array for rotating 50 steps is shown in Listing 8 below.

```

int fifty_step_delay_array[50] = {
    12000, 11992, 11940, 11788, 11483,
    10999, 10348, 9577, 8753, 7938,
    7180, 6504, 5919, 5421, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5421, 5919, 6504, 7180, 7938,
    8753, 9577, 10348, 10999, 11483,
    11788, 11940, 11992, 12000, 12000,
};

```

Listing 8. Fifty step delay array

The method for determining the values to include in the above array will be further discussed in Section 3.

2.2.5 Timers and PWM

The system makes use of multiple hardware timers to support both the motor control and the system timing requirements. These timers are responsible for generating the PWM signal used to

control the conveyor belt, as well as providing both blocking and non-blocking timing functions throughout the system.

A PWM signal is generated using Timer 0, configured in Fast PWM mode as shown in Listing 9. The PWM signal is output on pin PB7 and is used to control the speed of the DC motor driving the conveyor belt. In Fast PWM mode, the timer continuously counts up to a defined value, and the output pin is toggled based on a compare match between the timer counter and the Output Compare Register. The duty cycle of the signal determines the effective power delivered to the motor, and therefore directly controls its speed.

```
void PWM(){

    // Set to Fast PWM mode
    TCCR0A |= _BV(WGM00);
    TCCR0A |= _BV(WGM01);
    TCCR0B = 0b00000000;

    // Set the compare match output mode to clear on a compare match
    TCCR0A |= _BV(COM0A1);

    // Sets the prescale to 64
    TCCR0B |= _BV(CS01);
    TCCR0B |= _BV(CS00);

    // Stop at 127 counts
    OCR0A = 127;

    // Sets the output to pin PB7
    DDRB |= _BV(PB7);
}
```

Listing 9. PWM Generation

The duty cycle can be dynamically adjusted using the `changeDutyCycle()` function, shown in Listing 10. This function first applies safety clamps to ensure that the duty cycle remains within a valid operating range. This prevents the motor from stalling at extremely low duty cycles or being driven beyond safe operating limits. The duty cycle is then converted into a corresponding compare value and written to OCR0A, which updates the PWM output.

```
void changeDutyCycle(char duty_cycle){
    // set safety clamps
    if (duty_cycle < 12){
        duty_cycle = 12;
    }else if(duty_cycle > 100){
        duty_cycle = 100;
    }
    OCR0A = (duty_cycle*255)/100;
}
```

Listing 10. Change Duty Cycle Function

The system includes a blocking millisecond delay function, `mTimer()`, which is used for simple timing operations where program execution does not need to continue during the delay. The timer is based on Timer 1, which is configured in `init_mTimer()` to run at a prescaled frequency of 1 MHz by setting the CPU clock to 8 MHz and applying a prescaler of 8. This configuration

results in a timer tick period of 1 microsecond, allowing the hardware timer to accurately measure millisecond-scale delays. The mTimer function operates by resetting Timer 1 and polling the Timer 1 interrupt flag until a 1 millisecond interval has elapsed. A software counter is incremented each time the flag is set, and this process repeats until the desired delay count is reached. Although this approach is blocking and prevents other code from executing during the delay, it provides a simple and reliable timing method for non-critical delays within the system, such as stepper motor timing or short control pauses. The code for mTimer is shown in Listing 11 below.

```
void mTimer(int count){
    int i;
    i=0;
    TCCR1B |= _BV(WGM12);
    OCR1A = 0x03E8;
    TCNT1 = 0x0000;
    TIFR1 |= _BV(OCF1A);
    while(i<count){
        if((TIFR1 & 0x02) == 0x02){
            TIFR1 |= _BV(OCF1A);
            i++;
        }
    }
}
```

Listing 11. mTimer function

A non-blocking millisecond timer was configured on Timer 3. It used to same prescale settings as mTimer: 1 MHz frequency and would count to one thousand. However, instead of staying in a blocking loop for a given amount of time, it would continually count the number of milliseconds that had passed since the timer was initialized. The three functions used to create this are shown below in Listing 12.

```
void init_millis(){
    CLKPR = 0x80;
    CLKPR = 0x01;
    TCCR3B |= _BV(CS31);
    TCCR3B |= _BV(WGM32);
    OCR3A = 0x03E8;
    TCNT3 = 0x0000;
    TIMSK3 |= _BV(OCIE3A);
    TIFR3 |= _BV(OCF3A);
}

ISR(TIMER3_COMPA_vect) {
    millis_counter++;
    current_time++;
}

long int millis(){
    return millis_counter;
}
```

Listing 12. Non-blocking millis functions

As seen in the previous section, a microTimer function was used for the stepper acceleration profiles. This is to allow for finer tuning abilities, instead of having such coarse options to choose from. The microTimer function utilizes the fact that Timer 3 is already running at 1 MHz. By multiplying the millisecond timer by 1000 and adding the current value of TCNT3, a microsecond timer can easily be implemented, as shown in Listing 13.

```
long int micros(){
    long ms;
    long t;
    cli();
    ms = millis_counter;
    t = TCNT3;
    sei();
    return ms*1000 + t;
}

void microTimer(long microsecond_delay){
    long start = micros();
    while(micros() - start < microsecond_delay){}
}
```

Listing 13. Microsecond delay timer

2.2.6 Pause/Resume

The Pause/Resume feature will be triggered by interrupt 1, which is connected to a button. Software debouncing is used to ensure the interrupt is only triggered once per button press. If the system is running when the button is pressed, it pauses, and if the system is paused when the button is pressed, it resumes. If paused, the LCD displays the number of aluminum, steel, white plastic, and black plastic objects that have been sorted, as well as the number of objects currently on the belt. The software logic used to implement this is shown in Listing 14 below.

```
paused = !paused;
if(paused == 1){
    PORTB |= _BV(PB0)

    /* Display object counts on the LCD */

}else{
    PORTB &= ~_BV(PB0);
}
```

Listing 14. Pause/Resume logic

2.2.7 Ramp Down

The ramp-down feature must continue to sort the objects that are currently on the belt and then turn the system off once they have been sorted. It will be triggered by interrupt 0, which is connected to a button. This button also has software debouncing to prevent the trigger from firing more than once per button push. To achieve the desired effect, a flag will be set, and an eight second timer will start once the button is pushed. The program will be sent to the polling stage and continue as usual. The beginning of the polling stage contains an if statement that checks if certain conditions for ramp down have been met, as shown in Listing 15.

```

if(ramp_down_flag == 1 && millis_counter-ramp_down_time > 8000){
    goto RampDown_STAGE;
}

```

Listing 15. Ramp down stage conditions

Once this condition is met, the belt is stopped, all interrupts are disabled, and the appropriate object counts are displayed on the LCD.

3 System Performance Specifications

The maximum system performance specifications were obtained by adjusting parameters for the conveyor belt DC motor and the bucket stepper motor to achieve maximum operating speed without system failure. The absolute max/min values for system operation are shown in Table 1 below. Using absolute max/min values produces a greater risk of system error, so values should be adjusted to optimal values that balance speed and risk of error. For optimal values, refer to Table 2 below.

The stepper motor acceleration arrays were generated using a Python script to ensure smooth and consistent motion. The script implements a quintic S-curve function to gradually ramp the motor speed up and down, reducing mechanical shock and missed steps. The script was originally generated using ChatGPT 5 [2] and later tweaked to match our needs. The S-curve function is defined in Listing 16 below, and the full script can be found in Appendix A.

```

def smotherstep(t):
    return 6*t**5 - 15*t**4 + 10*t**3

```

Listing 16. Quintic S-Curve

There were three parameters that were varied to find the ideal system performance. The start delay is the delay that produces the minimum motor speed when the motor first starts motion. The minimum delay produces the maximum speed of the motor; this speed is maintained for a period between acceleration and deceleration phases. The acceleration steps parameter is the number of acceleration steps used to accelerate between the start delay and the min delay. If the acceleration steps are too low, the motor will stall. The script converts desired timing values into velocities using the relationship $v = \frac{1}{\text{delay}}$, then scales the acceleration profile so that the total change in velocity matches the required range between the starting delay and minimum delay. The velocity is incrementally integrated to generate a full motion profile consisting of acceleration, constant speed, and deceleration phases. Finally, the velocities are converted back into step delays. By varying these parameters and observing the results, the ideal values could be determined.

The minimum acceleration steps are dependent on the start delay and min delay, so the value provided in Table 1 is the minimum acceleration steps for the respective min delay and start delay values.

For the DC motor, the duty cycle parameter is the percentage of time the motor is powered on in the PWM cycle. A faster duty cycle produces a faster motor speed. The maximum duty cycle is dependent on the reflective sensor readings. The sensor takes readings periodically at a constant

interval of time, so if the object passes faster through the sensor, fewer readings will be taken, and the error in reading becomes larger. If the belt speed is too fast, the peak in curvature of the piece could pass through the sensor between sensor readings, resulting in an inaccurate reading.

Table 1 System Performance Parameters

System Parameter	Value
Accel_Steps	15 steps
Start_Delay	9000 μ s
Min_Delay	4300 μ s
Duty_Cycle	40%

To reduce the risk of the system incorrectly sorting pieces, it is necessary to reduce the bucket and belt speeds to optimal (stable) values.

When operating at the maximum possible belt speed, there is an increased risk of an incorrect reflective sensor reading, which would result in the piece being sorted into the incorrect bucket. At a duty cycle of 40%, the system was capable of correctly identifying pieces, but to reduce risk, we decreased the duty cycle to 35%.

The stepper motor is prone to failure when weight is added to the bucket, as more torque is required to rotate the motor. Metal pieces are typically dispersed from the bucket, but if they bounce into a different bucket section, they will add extra weight to the bucket. If the acceleration of the bucket is too rapid, the extra weight could cause the stepper motor to stall. Other aspects, such as temperature changes and friction, could also increase the required torque and cause the motor to stall. To minimize this risk, the acceleration steps are increased from 15 to 20 steps, Start_Delay is increased from 9000 μ s to 12000 μ s, and Min_Delay is increased from 4300 μ s to 5000 μ s. The optimal (stable) system parameters are shown in Table 2 below.

Table 2 Optimal Parameters

System Parameter	Value
Accel_Steps	20 steps
Start_Delay	12000 μ s
Min_Delay	5000 μ s
Duty_Cycle	35%

4 Testing and Calibration Procedures

A key part of the system is the calibration bounds used to distinguish between the different types of objects. As previously explained, the objects are identified by a reflective sensor connected to an onboard ADC. Ideally, all objects of the same material will exhibit the same reflective properties and result in the same digital result. However, there are many factors that can change the reflective sensor readings. The light level in the room, the smoothness of the object, and any dust or grease can vastly change the reflective properties of an object. To combat this issue, a range of values was chosen for each object type. These values are shown in Listing 17 below.

```
volatile int Aluminum_min = 0;  
volatile int Aluminum_max = 149;  
volatile int Steel_min = 150;  
volatile int Steel_max = 830;  
volatile int White_min = 831;  
volatile int White_max = 918;  
volatile int Black_min = 919;  
volatile int Black_max = 1023;
```

Listing 17. Calibration bounds

To determine the values for these bounds, objects of the same type would be repeatedly fed through the reflective sensor. Variables that store the lowest low and the highest low were created. For each object that passes through the sensor, the lowest ADC reading is kept and compared to the lowest low and highest low. If it is greater than the highest low, it becomes the new highest low, and if it is less than the lowest low, it becomes the new lowest low. After running all the objects through the sensor, the minimum bound is set to the lowest low, and the highest low is set to maximum bound. Once this has been done for all four materials, the bounds are extended so that every possible ADC result falls into one of the bounds.

As the different stages of the software were written, they were continually tested to ensure they were functioning correctly and accurately. The LCD screen was used to display information as objects passed through the system. After every object left the reflective sensor, the number of ADC readings taken, lowest value recorded, and CubeType were displayed on the LCD. This allowed us to constantly monitor the system and make sure that each object was being identified correctly. If an object was sorted incorrectly, it provided an easy way to check if the ADC was the culprit of the mistake. Similarly, when an object passes through the second optical sensor, the LCD displays the stepper motor's current quadrant and the target quadrant. This provided an efficient way to monitor the accuracy of the system as it was being tested.

Once the complete program was written, many tests were run to validate the accuracy and consistency of the system. These tests involved repeatedly running samples of each material in random order through the detection and sorting process and comparing the system's output classification against the expected result. This allowed verification that the calibration bounds correctly separated each material and that the decision logic consistently produced the correct output under normal operating conditions.

The system was also evaluated under non-ideal conditions. This included variations in background electrical noise, differences in object positioning, and surface cleanliness. These tests were used to assess the robustness of the reflective sensor and confirm that the calibration method was resilient to minor environmental changes.

5 Limitations and System Trade-offs

Our system algorithm is limited to sorting pieces in sets of a maximum of 8 pieces because the algorithm is not equipped to handle an interrupt from the reflective sensor while another piece is being sorted in the bucket stage. This is a substantial limitation, as it creates gaps on the belt where no pieces are sorted. When 8 pieces are sorted, occasionally, pieces can be separated as they travel down the belt, causing the first piece to reach the end of the belt before the eighth

piece has passed through the reflective sensor. To manage this risk, we had to make a trade-off between the risk of system failure and system sorting speed; we decided to eliminate the risk of failure by sorting pieces in groups of six pieces.

When the system remains powered on for about ten consecutive minutes, the motor driver starts to overheat, causing the stepper motor to experience unexpected behavior – including stalling, or rotating the incorrect number of steps. To mitigate this, the system must be powered down between sorting runs to allow time for cooling.

The sorting time is limited by the time it takes a piece to fall into the bucket. If the stepper rotates past the correct bucket before the piece falls, it will land in the incorrect bucket. A high stepper acceleration increases the risk of a piece hitting the rim between buckets or falling into the wrong bucket. That risk is reduced by adding a delay of 300 ms before rotating the bucket to allow the piece to fall. This is implemented in the code below, where the current time is a global variable reflecting the total time counted since the start of the program, and last_drop_time is a global variable that is set to equal the current time at the end of each drop. This function increases the time it takes to sort the pieces, but it significantly reduces the risk of pieces landing in the wrong buckets.

```
// ensures enough time has passed in between cubes, and the previous cube
has fallen before starting to turn the stepper again

if(target_quadrant != current_quadrant){
    while(current_time - last_drop_time < 300){}
}
```

Listing 18. Drop time buffer

6 Experience and Recommendations

Recommendations for system improvements are described in the sub-sections below.

6.1 Cooling System for the Motor Driver

As discussed in Section 5, the system is limited by the tendency of the motor driver to overheat and cause unexpected stepper motor behavior. Unplugging the controller makes testing less efficient, and it does not solve the problem of the driver overheating while the system is running. A cooling system could be added to the motor drive to mitigate overheating, allowing the system to stay powered on, removing wait time between testing runs, and preventing the risk of stepper motor failure while the system is running.

6.2 Full Set of Test Pieces for Each Station

Once the system is developed to a point where it can sort four pieces into correct buckets, it is essential to test the system with a full set of 48 pieces. Currently, this set can only be accessed for 30-minute intervals, often limiting the testing time to 30 minutes per session in the lab. If each station had its own set of test pieces, groups could test for substantially more time, debug more easily, and develop their systems to achieve greater performance.

6.3 Closed Buckets and More Powerful Stepper Motor

Currently, the buckets for metal pieces are open, allowing pieces to fall out of the bucket for weight reduction. The point of this is to reduce the weight of the bucket and prevent motor failure, but if this system were to be used in a real application, opening the buckets would defeat the purpose of the system. The open buckets introduce another error; occasionally, metal pieces become jammed between the bottom of the belt and the bucket as it is falling out of the bucket, causing the bucket to jam and lose alignment, resulting in sorting failure.

If the stepper motor were replaced with a more powerful motor capable of delivering more torque, the buckets for metal pieces could be closed, making a more consistent, realistic, and reliable system.

7 Conclusions

The objective of developing a fully functional mechatronic sorting system capable of accurately classifying and sorting objects was successfully achieved. The system integrated sensing and actuating to perform automated sorting with a high degree of reliability. By using an interrupt-driven state machine program, the system can respond efficiently to multiple events such as object detection and user inputs, enabling consistent and coordinated operation.

A key accomplishment of this project was the successful implementation of a reflective sensor and the calibration method that allowed consistent material classification despite variable environments. Additionally, the use of a precomputed stepper motor acceleration profile greatly improved motion smoothness and reduced the likelihood of missed steps or mechanical failure. These optimizations were critical in achieving a stable system.

Testing and optimization revealed important trade-offs between speed and accuracy. While maximum speed parameters allowed faster sorting, they introduced a high rate of misclassification and stepper motor errors. By choosing more conservative parameters, the system was able to achieve a 98% accuracy while sorting 48 objects in 39 seconds.

Despite its success, the system has several limitations. The software is currently able to handle multiple interrupts simultaneously. By restructuring the program to handle synchronized interrupts, the system would improve its sorting speed. Prolonged use of the system results in undesired thermal effects, which lead to unpredictable stepper motor actuation. These limitations show an opportunity for future improvements and testing. Overall, the project demonstrates a successful application of mechatronic design principles, combining hardware and software to complete a real-world task.

8 References

- [1] Y. Shi, “Final Project: Milestone 5,” MECH 458 course lecture slides, University of Victoria, 2026.
- [2] OpenAI, ChatGPT (GPT-5), 2026. Used for assistance in generating stepper motor acceleration profiles and SPI plot
- [3] Atmel, ATmega640/V-1280/V-1281/V-2560/V-2561/V Datasheet, 2014.

9 APPENDIX A – Full Code

9.1 Main.c

```
#include <avr/interrupt.h>
#include <avr/io.h>
#include <stdlib.h>
#include "lcd.h"
#include "Functions.h"

// GLOBAL VARIABLE DECLARATIONS

// State machine variable
volatile char STATE;

// ADC variables
volatile int ADC_count = 0;
volatile unsigned int ADC_result_flag = 0;

// Pause/Resume and Rampdown variables
volatile char paused = 0;
volatile unsigned int ramp_down_flag = 0;
volatile int ramp_down_time = 0;

// Stepper variables
volatile int current_position = 0;
volatile int current_quadrant;
// white = 0, aluminum = 1, steel = 2, black = 3

// Timing variables
volatile long millis_counter = 0;
volatile long last_drop_time = 0;
volatile long current_time = 0;

// Object identification variables
volatile int low;
volatile int high;
volatile int result;
volatile int lowest;
volatile char CubeType = 0;

// Completion tracker variables
volatile char completed_white = 0;
volatile char completed_aluminum = 0;
volatile char completed_black = 0;
volatile char completed_steel = 0;

// Calibration bounds
volatile int Aluminum_min = 0;
volatile int Aluminum_max = 149;
volatile int Steel_min = 150;
volatile int Steel_max = 830;
volatile int White_min = 831;
volatile int White_max = 918;
volatile int Black_min = 919;
volatile int Black_max = 1023;
```

```

int main(int argc, char *argv[]){

    // unlock clock settings and set system clock to 8 MHz. Initialize
mTimer
    CLKPR = 0x80;
    CLKPR = 0x01;
    init_mTimer();

    // set initial state to 0: polling stage
    STATE = 0;

    // set port directions
    DDRD = 0b11110000;      // INT2 & INT3 on PORTD
    DDRC = 0xFF;            // LCD Display on PORTC
    DDRL = 0xFF;            // Green LEDs for debugging
    DDRG = 0;               // Hall effect sensor

    // Initialize LCD and clear the screen
    InitLCD(LS_BLINK|LS_ULINE);
    LCDClear();

    // Set up FIFO queue
    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

    // initialize the pointers
    rtnLink = NULL;
    newLink = NULL;
    setup(&head, &tail);

    // Set up FIFO for the completed queue
    link *completed_head;
    link *completed_tail;
    setup(&completed_head, &completed_tail);

    // Calibration
    int highest_low = 0;
    int lowest_low = 1023;

    // Interrupt Configuration
    cli();                  // Disables all interrupts

    init_millis(); // initialize millisecond timer

    // Configure the ADC
    ADCSRA |= _BV(ADEN); // enable ADC
    ADCSRA |= _BV(ADIE); // enable interrupt of ADC
    ADMUX |= _BV(REFS0); // left adjusted bits in ADCH

```

```

    // Set up the Interrupt 0, 1, 2, and 3 options
    EICRA |= _BV(ISC01) | _BV(ISC00); // Interrupt 0: Rising edge(button is
active low)

    EICRA |= _BV(ISC11); // Interrupt 1: Falling edge(button is active high)
    EICRA &= ~_BV(ISC10);

    EICRA |= _BV(ISC21); // Interrupt 2: Rising edge (sensor is active low)
    EICRA |= _BV(ISC20);

    EICRA |= _BV(ISC31); //Interrupt 3: Falling edge(sensor is active high)
    EICRA &= ~_BV(ISC30);

    EIMSK |= (_BV(INT0)); // enable INT0
    EIMSK |= (_BV(INT1)); // enable INT1
    EIMSK |= (_BV(INT2)); // enable INT2
    EIMSK |= (_BV(INT3)); // enable INT3

    sei(); // Enable all interrupts

    // Initialize the belt motor
    DDRB = 0xff; //sets port B to output for motor driver
    PORTB = 0b00000000;
    PORTB |= _BV(PB2) | _BV(PB3); //sets ENA and ENB to high

    // Initialize the stepper to black quadrant
    current_quadrant = 2;
    while(PING & _BV(PG0)){
        Stepper_CCW(1);
        mTimer(5);
    }

    //start PWM and set it to 30% duty cycle
    PWM(); // starts PWM
    PORTB &= ~_BV(PB0); // sets INA = 0
    PORTB |= _BV(PB1); // sets INB = 1
    changeDutyCycle(35);

    goto POLLING_STAGE;

POLLING_STAGE:

    if(ramp_down_flag == 1 && millis_counter-ramp_down_time > 8000){
        goto RampDown_STAGE;
    }

    switch(STATE){
        case (0) :
            goto POLLING_STAGE;
            break;
        case (1) :
            goto REFLECTIVE_STAGE;
            break;
        case (2) :

```

```

        goto BUCKET_STAGE;
        break;
        case (3) :
        goto PauseResume_STAGE;
        break;
        case (4) :
        goto RampDown_STAGE;
        break;
        case (6) :
        goto END;
        default :
        goto POLLING_STAGE;
    }

REFLECTIVE_STAGE:

if(PIND & 0x04){ //if an object is there

    if(ADC_result_flag == 1){ // if a conversion is finished
        ADC_result_flag = 0;

        if(result < lowest){
            lowest = result;
        }
        ADCSRA |= _BV(ADSC); // Start a new conversion
    }

}else{ // cube has left the sensor

    // check to see if we need to update calibration ranges
    if(lowest > highest_low){
        highest_low = lowest;
    }
    if(lowest < lowest_low){
        lowest_low = lowest;
    }

    // LCD outputs for debugging reflective stage results. Uncomment
as needed.

    /*
    //display highest low and lowest low on LCD
    LCDClear();
    LCDWriteStringXY(0, 0, "HL: ");
    LCDWriteIntXY(4, 0, highest_low, 4);
    LCDWriteStringXY(0, 1, "LL: ");
    LCDWriteIntXY(4, 1, lowest_low, 4);

    LCDWriteStringXY(0, 0, "rslt: ");
    LCDWriteIntXY(6, 0, lowest, 4);

    LCDWriteStringXY(0,1, "adcs:");
    LCDWriteIntXY(5, 1, ADC_count, 4);
    */

```

```

        // Determine what type of cube passed through the sensor
        if(lowest >= White_min && lowest < White_max){
            CubeType = 0;
        }
        if(lowest >= Aluminum_min && lowest <= Aluminum_max){
            CubeType = 1;
        }
        if(lowest >= Black_min && lowest <= Black_max){
            CubeType = 2;
        }
        if(lowest >= Steel_min && lowest <= Steel_max){
            CubeType = 3;
        }

        // More LCD Debugging
        //LCDWriteIntXY(12, 0, CubeType, 1);

        // Add an item to the queue with the cube's information
        initLink(&newLink);
        newLink->e.itemCode = CubeType;
        enqueue(&head, &tail, &newLink);

        // go back to polling stage
        STATE = 0;
    }

    goto POLLING_STAGE;

BUCKET_STAGE:
{
    // variables for determining how much to turn the stepper
    int target_quadrant;
    int diff;
    target_quadrant = head->e.itemCode;
    diff = target_quadrant - current_quadrant;

    // LCD output for debugging bucket stage
    /*
    LCDWriteStringXY(0, 0, "curr: ");
    LCDWriteIntXY(6, 0, current_quadrant, 1);
    LCDWriteStringXY(0,1, "trgt: ");
    LCDWriteIntXY(6, 1, target_quadrant, 4);
    LCDWriteStringXY(7,0, "diff: ");
    LCDWriteIntXY(14, 0, diff, 1);
    */

    // ensures enough time has passed in between cubes, and the previous
    cube has fallen before starting to turn the stepper again
    if(target_quadrant != current_quadrant){
        while(current_time - last_drop_time < 300){}
    }

    if(diff == 1 || diff == -3){
        Stepper_CCW(50);                // counter clockwise 90°
    }
}

```

```

}else if(diff == 2 || diff == -2){
    Stepper_CW(100);          // clockwise 180°

}else if(diff == 3 || diff == -1){
    Stepper_CW(50);          // clockwise 90°
}

// update quadrant tracker
current_quadrant = target_quadrant;

// Update completion counts
if(target_quadrant == 0){
    completed_white++;
}
if(target_quadrant == 1){
    completed_aluminum++;
}
if(target_quadrant == 2){
    completed_black++;
}
if(target_quadrant == 3){
    completed_steel++;
}

// remove item from the in progress FIFO, and add to the completed FIFO
dequeue(&head, &tail, &rtnLink);
enqueue(&completed_head, &completed_tail, &rtnLink);

// sets INA = 0, starts the belt
PORTB &= ~_BV(PB0);

// update time stamp
last_drop_time = current_time;

// go back to polling
STATE = 0;
goto POLLING_STAGE;

} // end bucket stage

PauseResume_STAGE:

paused = !paused; // if system is running, pause. if system is paused,
resume

if(paused == 1){
    PORTB |= _BV(PB0); // sets INA = 1, stopping the belt

    // Display object counts on the LCD
    LCDClear();
    LCDWriteStringXY(0, 0, "C W A B S");
    LCDWriteIntXY(0, 1, size(&head, &tail), 1);
    LCDWriteIntXY(3, 1, completed_white, 1);
    LCDWriteIntXY(6, 1, completed_aluminum, 1);
    LCDWriteIntXY(9, 1, completed_black, 1);
    LCDWriteIntXY(12, 1, completed_steel, 1);
}

```

```

    }else{
        PORTB &= ~_BV(PB0); // sets INA = 0, starts the belt
    }

    // go back to polling stage
    STATE = 0;
    goto POLLING_STAGE;

RampDown_STAGE:

    PORTB |= _BV(PB0); // sets INA = 1, stopping the belt

    // Display completed object counts on the LCD
    LCDClear();
    LCDWriteStringXY(0, 0, "W A B S");
    LCDWriteIntXY(0, 1, completed_white, 2);
    LCDWriteIntXY(3, 1, completed_aluminum, 2);
    LCDWriteIntXY(6, 1, completed_black, 2);
    LCDWriteIntXY(9, 1, completed_steel, 2);

    END:

    PORTB |= _BV(PB0); // sets INA to 1, stopping the belt
    cli();

    return(0);
}

/* Set up the External Interrupt 0 Vector. Used for Ramp Down */
ISR(INT0_vect){

    // button debounce
    mTimer(25);
    while ( (PIND & 0x01) ) {}

    ramp_down_flag = 1;
    ramp_down_time = millis_counter;
}

/* Set up the External Interrupt 1 Vector. Used for Pause/Resume */
ISR(INT1_vect){

    //button debounce
    mTimer(25);
    while ( (PIND & 0x02) ) {}

    STATE = 3;
}

/* Set up the External Interrupt 2 Vector. Used for the First Optical Sensor
Interrupt */
ISR(INT2_vect){

    // reset the lowest variable, and start the first ADC reading

```

```

        lowest = 1023;
        ADC_count = 0;
        ADCSRA |= _BV(ADSC);
        STATE = 1;
    }
    /* Set up the External Interrupt 3 Vector. Used for the Second Optical Sensor
    Interrupt */
    ISR(INT3_vect){
        PORTB |= _BV(PB0); // sets INA to 1, stopping the belt
        STATE = 2;          // corresponds to bucket stage
    }

    // the interrupt will be triggered once the ADC is done a conversion
    ISR(ADC_vect){

        // set ADC result flag and increase the ADC counter
        ADC_result_flag = 1;
        ADC_count++;

        // convert to 10 bit number and store in result
        low = ADCL;
        high = ADCH;
        result = high << 8;
        result = result | low;
    }

    ISR(BADISR_vect){
        while(1){
            PORTL = 0xFF;
        }
    }

```

9.2 Functions.C

```

/* include libraries */
#include <stdlib.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "Functions.h"

// global variables from main
extern volatile int current_position;
extern volatile int millis_counter;
extern volatile long current_time;

// 50 step stepper s-curve acceleration array
int fifty_step_delay_array[50] = {
    12000, 11992, 11940, 11788, 11483,
    10999, 10348, 9577, 8753, 7938,
    7180, 6504, 5919, 5421, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5421, 5919, 6504, 7180, 7938,

```

```

    8753, 9577, 10348, 10999, 11483,
    11788, 11940, 11992, 12000, 12000,
};

// 100 step stepper s-curve acceleration array
int hundred_step_delay_array[100] = {
    12000, 11992, 11940, 11788, 11483,
    10999, 10348, 9577, 8753, 7938,
    7180, 6504, 5919, 5421, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5000, 5000, 5000, 5000, 5000,
    5421, 5919, 6504, 7180, 7938,
    8753, 9577, 10348, 10999, 11483,
    11788, 11940, 11992, 12000, 12000,
};

// turns the stepper clockwise
void Stepper_CW(int steps){

    // full stepping array
    int array[4] = {0b00110110, 0b00101110, 0b00101101, 0b00110101};

    for(int i = 0; i < steps; i++){

        int delay = 10; // used for stepper initialization

        current_position = (current_position + 1) % 4;
        PORTA = array[current_position];

        if(steps == 50){
            microTimer(fifty_step_delay_array[i]);
        }
        if(steps == 100){
            microTimer(hundred_step_delay_array[i]);
        }
        if(steps == 1){
            mTimer(delay);
        }
    }
}

// Turns the stepper motor counter clockwise
void Stepper_CCW(int steps){

    // full stepping array

```

```

int array[4] = {0b00110110, 0b00101110, 0b00101101, 0b00110101};

for(int i = 0; i<steps; i++){

    int delay = 10;

    current_position = (current_position + 3) % 4;
    PORTA = array[current_position];

    if(steps == 50){
        microTimer(fifty_step_delay_array[i]);
    }
    if(steps == 100){
        microTimer(hundred_step_delay_array[i]);
    }
    if(steps == 1){
        mTimer(delay);
    }
}

}

// Generates a PWM signal on pin PB7
void PWM(){

    // Set to Fast PWM mode
    TCCR0A |= _BV(WGM00);
    TCCR0A |= _BV(WGM01);
    TCCR0B = 0b00000000;

    // Set the compare match output mode to clear on a compare match
    TCCR0A |= _BV(COM0A1);

    // Sets the prescale to 64
    TCCR0B |= _BV(CS01);
    TCCR0B |= _BV(CS00);

    // Stop at 127 counts
    OCR0A = 127;

    // Sets the output to pin PB7
    DDRB |= _BV(PB7);
}

// changes the duty cycle of the PWM signal
void changeDutyCycle(char duty_cycle){

    // set safety clamps
    if (duty_cycle < 12){
        duty_cycle = 12;
    }else if(duty_cycle > 100){
        duty_cycle = 100;
    }

    OCR0A = (duty_cycle*255)/100;
}

// returns how many microseconds have passed

```

```

long int micros(){
    long ms;
    long t;

    cli();
    ms = millis_counter;
    t = TCNT1;
    sei();

    return ms*1000 + t;
}

// blocking microsecond timer used for the stepper s-curve delay
void microTimer(long microsecond_delay){
    long start = micros();
    while(micros() - start < microsecond_delay){}
}

// non-blocking millisecond timer initialization
void init_millis(){
    CLKPR = 0x80; //unlocks clock settings
    CLKPR = 0x01; // sets CPU clock to 8MHz
    TCCR3B |= _BV(CS31); // sets prescaler so timer runs at 1MHz

    TCCR3B |= _BV(WGM32); // sets waveform generation mode to 0100

    OCR3A = 0x03E8; // Set the output compare register for 1000 cycles = 1
ms
    TCNT3 = 0x0000; // sets timer counter to start at 0

    TIMSK3 |= _BV(OCIE3A); // set the output compare interrupt enable to 1

    TIFR3 |= _BV(OCF3A); // clear the timer interrupt flag and begin new
timing
}

// returns the number of milliseconds that have passed
long int millis(){
    return millis_counter;
}

// increments the millisecond timer
ISR(TIMER3_COMPA_vect) {
    millis_counter++;
    current_time++;
}

// Initialization for mTimer
void init_mTimer(void){
    CLKPR = 0x80; //unlocks clock settings
    CLKPR = 0x01; // sets CPU clock to 8MHz
    TCCR1B |= _BV(CS11); // sets prescaler so timer runs at 1MHz
}

```

```

// Blocking delay that delays the given number of milliseconds
void mTimer(int count){

    int i;
    i=0;

    TCCR1B |= _BV(WGM12); // sets waveform generation mode to 0100

    OCR1A = 0x03E8; // Set the output compare register for 1000 cycles = 1
ms    TCNT1 = 0x0000; // sets timer counter to start at 0

    //TIMSK1 |= _BV(OCIE1A); // set the output compare interrupt enable to
1    TIFR1 |= _BV(OCF1A); // clear the timer interrupt flag and begin new
    timing

    while(i<count){

        if((TIFR1 & 0x02) == 0x02){

            TIFR1 |= _BV(OCF1A);

            i++;
        } //end if

    } //end while

} // end mTimer

// FIFO FUCNTIONS:

/*****
*****
* 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;
}/*setup*/

/*****
*****
* 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;
    }/*initLink*/

/*****
*****
*   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
*/
/* will put an item at the tail of the queue */
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;
}/*enqueue*/

/*
*****
*****
*   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
*/
/* This will remove the link and element within the link from the head of the
queue */
void dequeue(link **h, link**t, link **deQueuedLink){
    /* ENTER YOUR CODE HERE */
    *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 there is only one element in the queue, the tail pointer
will still be pointing to the dequeued element after its removed.
        // once element is removed, check to see if head == NULL, and if
it is, change tail to NULL as well.

        if (*h == NULL){
            *t = NULL;
        }
    }
}

```

```

        }/*if*/

        return;
    }/*dequeue*/

/*
*****
* DESC: Peeks at the first element in the list
* INPUT: The head pointer
* RETURNS: The element contained within the queue
*/
/* This simply allows you to peek at the head element of the queue and
returns a NULL pointer if empty */
element firstValue(link **h){
    return ((*h)->e);
}/*firstValue*/

/*
*****
* DESC: deallocates (frees) all the memory consumed by the Queue
* INPUT: the pointers to the head and the tail
*/
/* This clears the queue */
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;
}/*clearQueue*/

/*
*****
* 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
*/
/* Check to see if the queue is empty */
char isEmpty(link **h){
    /* ENTER YOUR CODE HERE */
    return(*h == NULL);
}/*isEmpty*/

```

```

/*
*****
* 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
*/
/* returns the size of 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);
}/*size*/

```

9.3 Functions.h

```

//#include "LinkedList.h"
#include <stdlib.h>

typedef struct {
    char itemCode;    /* stores a number describing the element */
    char stage;       /* 0: part is built, 1: part not built, 2: part is
shipped */
} element;

typedef struct link{
    element e;
    struct link *next;
} link;

void setup(link **h, link **t);
void initLink(link **newLink);
void enqueue(link **h, link **t, link **nL);
void dequeue(link **h, link **t, link **deQueuedLink);
element firstValue(link **h);
void clearQueue(link **h, link **t);
char isEmpty(link **h);
int size(link **h, link **t);
void init_mTimer(void);
void mTimer(int count);
void PWM();
void changeDutyCycle(char duty_cycle);

```

```

void Stepper_CW(int steps);
void Stepper_CCW(int steps);
int TrapezoidDelay(int step_num, int total_steps);
void init_millis();
long int micros();
void microTimer(long microsecond_delay);

```

9.4 Final_Stepper_Curve.py

```

import math

def smotherstep(t):
    # Quintic S-curve:  $6t^5 - 15t^4 + 10t^3$ 
    return 6*t**5 - 15*t**4 + 10*t**3

def generate_step_delay_array(total_steps, accel_steps, start_delay,
min_delay):
    if accel_steps * 2 > total_steps:
        raise ValueError("Acceleration + deceleration steps exceed total
steps")

    # Convert delays to velocities (steps per microsecond)
    v_min = 1.0 / start_delay
    v_max = 1.0 / min_delay

    velocities = []

    # --- Build acceleration profile (S-curve) ---
    accel_profile = []
    for i in range(accel_steps):
        t = i / (accel_steps - 1)
        accel_profile.append(smotherstep(t)) # 0 → 1

    # Scale acceleration so total dv matches target
    dv_total = v_max - v_min
    total_a = sum(accel_profile)
    scale = dv_total / total_a
    accel_profile = [a * scale for a in accel_profile]

    # --- Integrate acceleration → velocity ---
    v = v_min

    # Acceleration phase
    for a in accel_profile:
        v += a
        velocities.append(v)

    # Cruise phase
    cruise_steps = total_steps - 2 * accel_steps
    velocities.extend([v_max] * cruise_steps)

    # Deceleration phase (mirror)
    for a in reversed(accel_profile):
        v -= a
        velocities.append(v)

```

```

# --- Convert velocities → delays (microseconds) ---
delays = [int(1.0 / v) for v in velocities]

return delays

def print_c_array(name, array):
    print(f"int {name}[{len(array)}] = {{{")
    for i in range(0, len(array), 5):
        row = ", ".join(f"{val:5d}" for val in array[i:i+5])
        print(f"\t{row},")
    print("};")

# ---- Example usage ----
if __name__ == "__main__":
    total_steps = 50
    accel_steps = 15
    start_delay = 12000
    min_delay = 5000

    delays = generate_step_delay_array(
        total_steps,
        accel_steps,
        start_delay,
        min_delay
    )

    print_c_array("fifty_step_delay_array", delays)

    total_time = sum(delays) / 1_000_000
    print(f"Total time: {total_time} seconds")

```