University of Victoria

Department of Mechanical Engineering

MECH 458 - Mechatronics

Final Report

Jeremy Kissack -V00893340

Connor Jones - V00898091

2022-04-21

Table of Contents

List of Figures

List of Tables

Abstract

This project consisted of a real-time classification system that sorts 48 cylinders in the fastest time possible on a conveyor belt. The cylinders were of four different types: black plastic, white plastic, steel, and aluminum. An ATmega2560 microcontroller was used to program a DC motor, a stepper motor, 4 sensors, two buttons, and an LCD display. The DC motor, stepper motor, and the 4 sensors were programmed to perform the classification and sorting. The LCD was used to display the number of sorted items upon the pressing of either the pause button or ramp down button. The system was evaluated based on the time it took to manually place 6 cylinders at a time on the conveyor belt and correctly sort them until all 48 had been sorted. The implementation described in this report successfully sorted 47 pieces correctly in 29 seconds, which means 1 piece was sorted incorrectly. This report outlines the specific algorithms used to achieve this and how it was tested.

1 Introduction

The goal of this project is to use an ATmega2560 Development board to control a DC motor and stepper motor to sort a given number of cylinders, of various material types, into a circular sorting tray that is traveling down a conveyor belt. The DC motor is used to power the conveyor belt and the stepper motor is used to rotate the circular sorting tray. Once the pieces have been correctly sorted, the Liquid Crystal Display (LCD) should display the number of sorted cylinders in each category.

Upon the pressing of a pause switch, the conveyor belt should stop moving and the LCD should show the current number of sorted parts that have been either fully processed or partially processed. Once the pause switch is pressed a second time, the conveyor belt should start moving again and the sorting should continue. Lasty, upon the pressing of a ramp down switch, the final cylinders should finish being processed and sorted and the conveyor belt should come to a halt. The total number of sorted cylinders in each category should be displayed.

1.1 Requirement Specifications [1]

R-1: Move a minimum number of 48 cylinders on the conveyor belt through all 3 sensors in less than 60s.	
Rationale:	The ultimate measure of performance of the system is the number of cylinders it can correctly sort in the shortest amount of time.
R-2: Determine the characteristics of each cylinder moved through the sensors.	
Rationale:	Quantitative data regarding each cylinder must be acquired to accurately apply a classification algorithm to each.
R-3: Classify each cylinder into 1 of 4 categories and sort into the corresponding holding tray	
Rationale:	To accurately sort each of the four discrete types of cylinders, (white ABS, black ABS, Aluminum, and Steel), infra-red reflectivity can be used as a proxy for material type.
R-4: Freeze the system and display all cylinders which have been processed or are in progress when a system pause is signaled.	
Rationale:	This will allow for easy debugging and testing to ensure that the software algorithm is keeping track of the sorted items correctly.
R-5: Complete the processing of remaining in-progress cylinders on the conveyor belt and display the result of each item on an LCD when a system ramp down is signaled.	
Rationale:	This allows the system to finish processing the remaining cylinders in the queue and stop the timer immediately upon completion of the last cylinder, displaying the results on the processed items.

Table 1 - System Requirement Specifications.

2 System Specifications

The hardware used for this project consists of an ATmega2560 microcontroller, an LCD display, a DC motor, a stepper motor, two optical sensors, a reflective sensor, two buttons, and a conveyor belt. A circuit diagram outlining the connections made between all the hardware

components is shown below in Figures 1 and 2. It shows the interconnection of actuators, such as the stepper motor and DC motor, and sensors, to the ATmega2560 Development board [2][3][4]. Note that the red LEDs are not included in the diagram since they were not used in the final implementation.

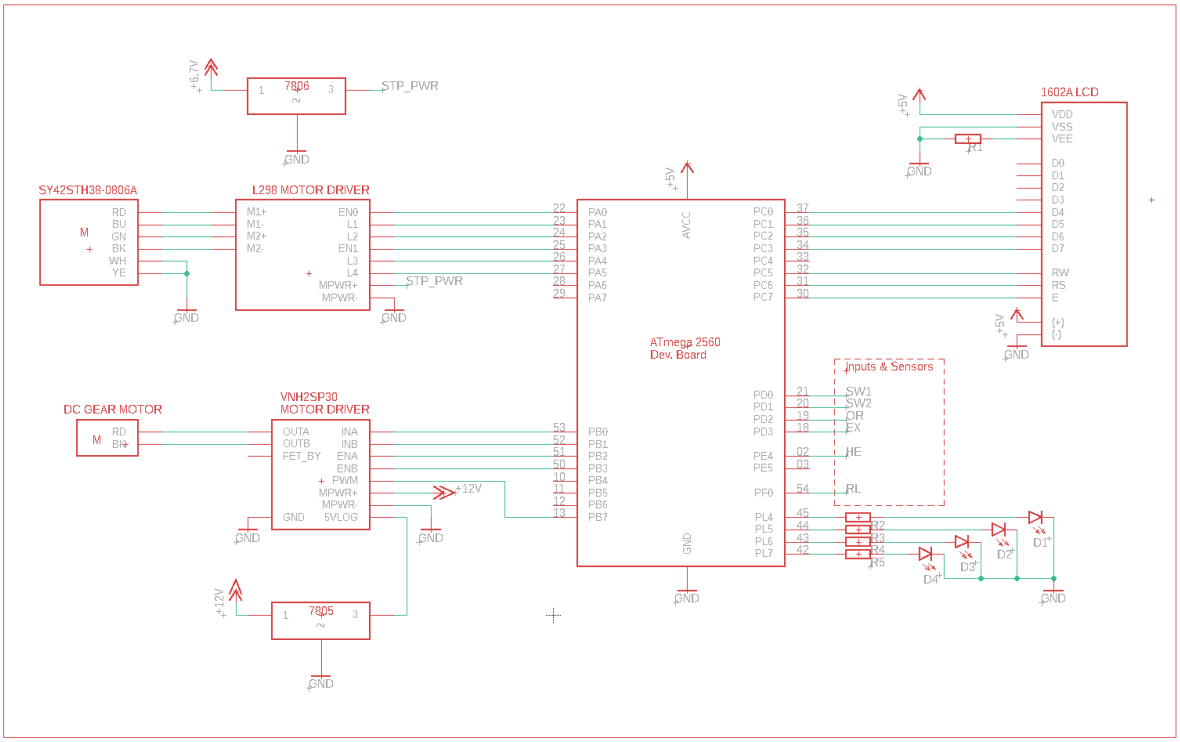

Figure 1 - System circuit diagram overview.

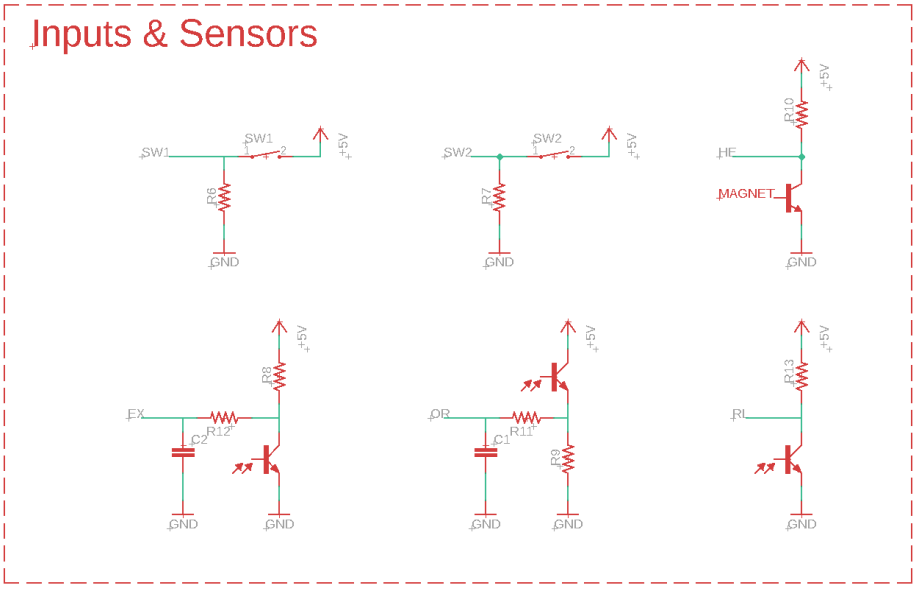

Figure 2 - System circuit diagram inputs and sensors.

2.1 System Algorithm

A flowchart of the software and hardware co-design algorithm is shown in Figure 3. The process for each step is outlined in detail below.

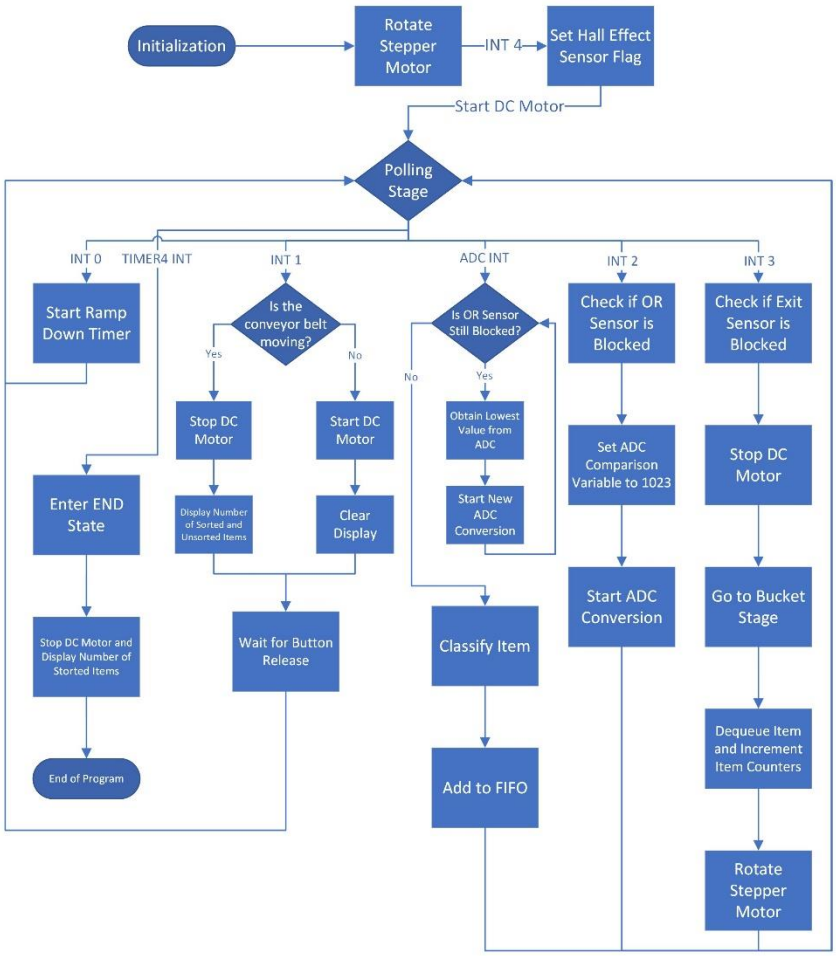

Figure 3 - Flowchart of main system algorithm.

2.1.1 Initialization & Polling Stage

The initialization stage of the program is where include statements, global flags, and registers are set to prepare for the operation of the main state machine. The clock must be initialized and a prescaler set so that the clock runs at 8 MHz. Timer 1 is used as a delay and the output compare register is initialized to be 0x0A to allow for delays in the microsecond range. A global variable STATE keeps track of the location of the program in the state machine, and this is set to 0 to start.

Once global variables and the clock and timer are set, the different ports are enabled as either inputs or outputs. Port B controls the DC motor and the PWM and is set as an output. Port C controls the LCD and is set as an output. The least significant 4 bits of Port D are set as inputs

for interrupts 0-3 which handle 2 buttons and 2 sensors. Port E is set as an input to handle the hall effect sensor input.

After the ports are configured, the interrupts are enabled to either trigger on a rising or falling edge and the corresponding interrupt flags are set. The interrupts connected to the ramp down and pause buttons as well as the OR sensor are configured for triggering on a rising edge. The exit and hall effect sensors are configured for triggering on a falling edge. The ADC and its interrupt are enabled as well, and the ADC is set to be right adjusted. Also, the PWM is enabled and configured to Fast PWM. Timer 0 is used for the PWM and the prescaler is set to 1/256 to provide a frequency of 500Hz. Initially the duty cycle is set to 0 and gets updated later in the program when the DC motor is required to move.

The last stage of the initialization is determining the absolute position of the stepper motor. The stepper motor gets rotated until the magnet on the stepper motor is directly in front of the hall effect sensor, triggering INT 4. Once this interrupt fires, the absolute position of the stepper motor is known, and the program can continue to the polling stage.

In the polling stage, the program is continuously checking if the pause button has been pressed. If the system not paused, the belt speed is set to 50% duty cycle if there are no items in the FIFO. If there are items in the FIFO, then the belt speed is set to 25% duty cycle. These values are stored in constants called EMPTYSPEED and SORTINGSPEED, respectively. The specifics of the system while it is paused is discussed later in this report.

2.1.2 Item Classification

Upon the breaking of the OR sensor beam, INT 2 is triggered. There is an initial 3ms delay added at the beginning of the ISR to handle any sensor bouncing that may occur from noise introduced into the system by the DC motor. Once the delay has expired, there is a software filter to check if the beam is still blocked. If the beam is still blocked, an ADC conversion is started to begin the classification process.

After the initial ADC conversion is completed, the ADC interrupt is triggered. The current value in the ADC register is stored in a variable to be used in a comparison. If the OR sensor is still blocked after the latest ADC value has been stored, it is compared with the previous value stored in a variable called ADC_Val, which is initialized to be the maximum value that can be read by the ADC, 1023; if the latest value is lower than the previous value, then ADC_Val gets updated to hold the new lowest value. A new ADC conversion is then started, and the process repeats until the OR sensor is no longer blocked. Once this becomes the case, the piece is classified.

To classify four different item types, three different range thresholds are defined according to the following 3 equations:

$$Range1 = \frac{(Steel_{Min} + Alum._{Max})}{2}$$

$$Range2 = \frac{(White_{Min} + Steel_{Max})}{2}$$

$$Range3 = \frac{(Black_{Min} + White_{Max})}{2}$$

This classification algorithm creates a buffer zone for each item type that lies directly between two different item types. This maximizes the chances of an outlier item to be classified correctly if it happens to land outside of the measured calibration limits.

By comparing the resulting ADC value to these defined ranges, the items can be accurately classified by the following logic:

$$ADC\,Value > Range3 \rightarrow Black$$

$$Range3 > ADC\,Value > Range2 \rightarrow White$$

$$Range2 > ADC\,Value > Range1 \rightarrow Steel$$

$$Range1 > ADC\,Value \rightarrow Alum.$$

In the software, a variable called itemClass is assigned an integer ranging from 0-4, 0 corresponding to black, 1 to white, 2 to steel, and 3 to aluminum. This information is then wrapped up into a new link for the FIFO and the new link is queued.

2.1.3 Exit Sensor & Stepper Rotation

When the exit sensor beam is blocked, INT 3 is triggered. Just like with the OR sensor, a software filter is in place to ensure that there is no false trigger of the sensor. If the exit sensor is in fact blocked by a piece, the DC motor is stopped, the speed of the belt is reduced to 25% duty cycle for when it starts again, and the state variable is changed to direct the program to the bucket stage. There is also a 3ms delay at the end of the ISR to limit sensor bouncing.

The first thing that happens in the bucket stage is items are dequeued from the FIFO. Based on the item code attached to the link, the target position for the stepper motor is set and the item counters are incremented. For example, if the item code attached to the dequeued link is 2, this item is steel. The target position for the stepper motor is set to –90 and the numST counter is incremented by 1. After, the memory is freed up and the drive_stepper function is called to rotate the stepper motor. A flow chart of the drive_stepper function is shown below in Figure 4.

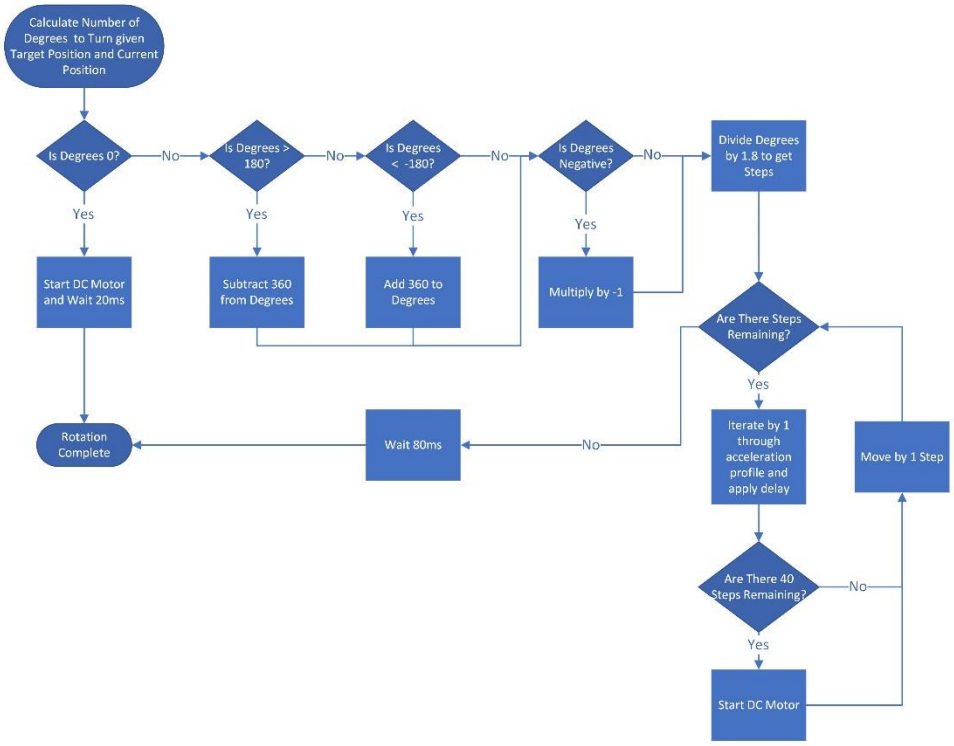

Figure 4 - Flowchart of stepper motor driving algorithm.

Initially, a series of conditions are checked within the function to modify the inputs. A variable called degrees stores the number of degrees the stepper is required to turn. It is calculated by taking the difference between the target position and the current position. Recall that the target position was set in the bucket stage when items are dequeued. The current position is a pointer to a variable that keeps track of the position of the stepper motor after initialization.

The first condition that is checked is if degrees is 0. If this is the case, the DC motor is turned on and a delay of 20ms is started to ensure the stepper motor does not begin turning too early for the next piece before the current piece has fallen into the bucket. If degrees is not 0, then it is checked for being greater than 180 or less than –180. If either of these conditions are true, then 360 is subtracted from degrees or 360 is added to degrees, respectively. This is to ensure that the stepper motor takes the fastest possible path for rotation.

The last condition that gets checked before any rotation occurs is if the degrees variable is negative. If this is the case, it is multiplied by negative 1 to obtain a positive value since the next section of code computes the number of steps needed to rotate the stepper motor.

Once the number of steps has been calculated, the program enters a for loop that decrements by 1 each iteration to indicate that the stepper motor has moved by 1 step. An iterator variable is initialized to 0 that keeps track of where in the stepper acceleration profile the program is. If the iterator is less than the size of the acceleration profile array, then the delay at the current position is applied and the iterator is incremented by 1. For deceleration, if the number of steps

remaining is equal to the iterator position, then the delay is set to the element at position i-1 and the iterator is decremented by 1 until the stepper motor has been completely decelerated.

When there are exactly 40 steps remaining in the rotation, the DC motor is turned on at a speed of 35%. This allows for the pieces falling off the conveyor belt to fall directly in the middle of the bucket every time. The value of 40 was fine-tuned after several iterations of testing and was deemed the best value.

Once all the steps have been completed, an 80ms delay is started to give the belt time to unload the item into the correct position.

2.1.4 Pause & Ramp Down Functions

When the pause button is pressed, INT 1 gets triggered, and the DC motor is halted. The current number of sorted items in each category is displayed on the LCD and any items between sensors on the conveyor belt is included in this count by pulling information from the FIFO.

When the ramp down button is pressed, the function rampdown() is called and this initialized timer 4 to being counting to a specified value. Upon the timer value matching the value stored in the compare register, TIMER4_INT is triggered, and this changes the state variable to enter the END state. In this state, the DC motor is stopped, and the total count values are displayed on the LCD.

2.1.5 Operating Parameters

Reliability	**Empty-Queue Belt PWM**	**Sorting Belt PWM**	**Stepper Profile**			**Continuous Dequeue**	**Load Size**
			Method	**Length**	**Min Delay**		
Stable	25%	50%	Trapezoidal	22 steps	4.68 ms	No	6
Max Speed	35%	80%	Trapezoidal	20 steps	4.08 ms	Yes	8

Table 2 - System Operating Parameters.

Table 2 describes two different modes of operation of the system. The stable mode takes a conservative approach to the speed while maintaining very high accuracy. The Max Speed mode prioritizes the speed and disregards the accuracy. The table describes the systems key operating parameters which dictate the overall speed. The belt PWM values control how fast the conveyor belt feeds in items. The stepper profile parameters control the acceleration curve of the stepper motor, where the Min. Delay defines the period of rotation of the stepper motor at maximum speed. The Continuous Dequeue parameter defines whether the conveyor belt must stop at every exit sensor interrupt, or if the belt can continuously spin when the stepper is in the correct position already. The final parameter, Load Size, defines the number of items added to the conveyor belt upon each loading cycle.

3 Results

Upon final demonstration, the system achieved the required tasks as defined in Table 1. This was done using the following techniques described below.

3.1 Sensor Calibration

In a complex mechatronic system, correctly calibrated sensors are necessary to ensure that accurate data is being received by the system.

3.1.1 Reflectivity Sensor

Due to variation of the different item sets, it was important to have a robust calibration routine which would account for these subtle differences. On top of this, the analog reflectivity sensor is also prone to variation to due temperature and supply voltage variation.

The calibration was completed by running each of an item type through the sensor, E.G., all 12 white items. The maximum and minimum values for item type were stored in an excel spreadsheet and used in further calculations. Once the min and max values were found for all 4 of the item types, the classification ranges were calculated as specified in section 2.1.2.

3.1.2 Exit Sensor

The position and timing of the exit sensor played a critical role in the correct sorting of the items. Variation in belt speed, depending on the station, as well as belt tension and other factors such as temperature, could affect the time to stop the belt upon an item dequeue. It was critical to mechanically calibrate the position of the exit sensor on the belt to ensure that items would not fall off too soon.

3.2 System Validation

To validate the performance and reliability of the system, given an unknown sequence of items, many different edge-cases were tested. This included running various combinations of item types through the system as well as various item spacing to thoroughly stress test the stepper motor and exit sensor interconnections.

The stepper motor was also heavily validated to ensure that the aggressive trapezoidal acceleration profile would not stall the motor. This was achieved by filling the tray with many items to increase the inertia of the load, simulating a worst-case scenario. The stepper was then tasked to sort items, doing many different +90, -90, +180, and -180 degree movements.

The pause and ramp down functions were also validated on many different edge-cases to ensure that the displayed results were accurate, and the performance of the system was not impacted by these features.

3.3 System Performance Specifications

Using the recommended stable operating parameters, defined in Table 2.1.5, this system satisfied all performance requirements, as specified in section 1.1. The system sorted all 48 items in 29 seconds with one error, due to miscalibration of the black and white items. This yields an SPI index of 1.59.

When using the Max Speed operating parameters in Table 2.1.5, the system frequently misclassified items, stalled the stepper motor, or developed FIFO errors. For this reason, an SPI index was not calculated because the results were not consistent.

3.4 Novel Additions

This system better optimized the loading speed of the conveyor belt by maximizing the belt speed while the queue is empty. A limiting factor to the belt speed is the time it takes for the belt to decelerate when the exit sensor is triggered. If the belt speed is too high, and an item comes to the exit sensor, it will fall off the belt before the stepper motor is ready to sort it. Implementing both an empty-queue speed, and a sorting speed greatly improves the time it takes to initially load the items. The only limiting factor to the empty-queue speed is the accuracy of the RL sensor. Higher belt speed leads to a faster transit time of the item across the sensor, which ultimately reduces the ADC sample count and reduces the accuracy of the item classification. From extensive testing, it was found that a PWM value of 25% allowed reliable item sorting for the sorting speed, and 50% for the empty-queue speed ensured reliable object classification.

Another addition that greatly assisted the outcome was increasing the accuracy of mTimer. Initially, the output compare register for the timer was set to 1ms, but this was reduced to 10µs to allow for higher precision timing. This was used specifically for the stepper acceleration profile so that delays that were fractions of milliseconds apart could be used.

3.5 Limitations & Trade-offs

The main limitation of this system is the trade-off between speed and accuracy. The following methods could be used to greatly improve the systems speed, but would reduce the accuracy, leading to misclassifications.

3.5.1 Stepper Speed

Although higher stepper speeds were achievable using a more aggressive acceleration curve, it would infrequently cause the stepper motor to stall out under high inertial loads. This causes a catastrophic failure of the sorting process, so a lower stepper speed was settled on.

3.5.2 Belt Speed & Continuous Item Dequeue

Another way to improve the system speed is to increase the belt speed, but this could only be done up to a limit, as described in section 3.4. In addition to this, the average belt speed can be increased by reducing the number of times that the belt must come to a stop. For example, if the stepper motor is already in the correct position, the belt has no need to stop. This technique was thoroughly explored, but ultimately could not be tuned reliably and would often lead to subsequent items falling of the belt. For this reason, the belt was stopped every time an item was dequeued, but only for a very short period if the stepper was already at the right position.

4 Experience & Recommendations

While the overall system was quite effective, completing the task in just 29 seconds, there are several factors which could further improve the system in future.

The stepper motor was quite fast, but it could have been improved by spending more time tuning it, or even implementing an s-curve as opposed to a trapezoidal profile.

A major obstacle in the testing of this system was the item loading. Due to user discoordination, it was very difficult to load more than 6 items at a time. An auto-feeder system which automatically loads items onto the conveyor belt would eliminate this problem, at the cost of increased complexity.

The DC motor did not provide the most consistent speed output, creating a bit of unknown in the system. This DC motor needs a PI or PID feedback loop so that it can be driven more predictably, and more complex timing techniques can be used in the project.

Lastly, a predictive algorithm, which, given the belt speed, item queue, and item spacing could theoretically generate the ideal combination of movements of both the conveyor belt and stepper motor to maximize the rate at which items are sorted. However, the ATmega2560 used in this project is likely not powerful enough to optimize these types of movements. A more powerful processor could take advantage of modern machine learning and AI techniques to fully optimize this system.

References

[1] K. Zang, "Final Project: Milestone 5", University of Victoria, 2022.

[2] Atmel Corporations, “8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable Flash,” Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V datasheet, 2014.

[3] Solarbotics Ltd, “The Compact L298 Motor Driver,” L298 Motor Driver datasheet, June 2007.

[4] Pololu, “Pololu High Current Motor Driver,” Pololu High Current Motor Driver datasheet.

Appendix A - Software

```
// Include statements
#include <stdlib.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#include "mtimer.h"
#include "lcd.h"
#include "LinkedQueue.h"

// Define Ranges for part differentiation
#define RANGE_1 179 // Limit between Aluminum and Steel
#define RANGE_2 711 // Steel and White
#define RANGE_3 923 // White and Black

// Define stepper port values for each position
#define STEPINCREMENT 1.8
#define POS_0 0b00011011
#define POS_1 0b00011101
#define POS_2 0b00101101
#define POS_3 0b00101011

// Size of stepper acceleration array
#define timerArraySize 22

// Different belt speeds depending if parts are in FIFO or not
#define SORTINGSPEED 0x40
#define EMPTYSPEED 0x80

// Initial delay for stepper motor homing
volatile int TIMER_DELAY = 1500;

// Stepper acceleration profile
int stepDelay[] = {1800, 1400, 1185, 1045, 946, 870, 810, 761, 720, 685, 654, 628, 604, 583, 564, 546,
530, 516, 502, 490, 479, 468};

// Track which state is active
volatile char STATE;

// Track number of times exit sensor has been triggered so
// as not to miss an item
volatile int exitCounter = 0;

// Minimum value detected by ADC
volatile int ADC_Val = 1023;

// Flag to indicate whether the pause button has been pressed
volatile int isPaused = 0;

// Store classification of each item
volatile int itemClass = 0;

// Store each value output by ADC for comparison
volatile int ADC_result;

// Variable to track the phases of the stepper
volatile int curPhase = 0;

// Flag to indicate hall effect sensor triggered
volatile int HE_Found = 0;
```

```
// Track current position of stepper
volatile int Position = 0;

// Target position of stepper
volatile int targPos = 0;

// Counters to track number of sorted parts
volatile int numBL = 0;
volatile int numWH = 0;
volatile int numST = 0;
volatile int numAL = 0;

// Linked list elements
link *head = NULL;
link *tail = NULL;
link *newLink = NULL;
link *rtnLink = NULL;
element e;

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

        // Initialize clock and mTimer
        initClockandTimer();

        // Initialize LCD
        InitLCD(LS_BLINK|LS_ULINE);
        LCDClear();

        // Initialize State
        STATE = 0;

        // Disables all interrupts
        cli();

        // Initialize Ports
        DDRD = 0b11110000;
        DDRE = 0x00;
        DDRC = 0xFF;
        DDRA = 0xFF;
        DDRB = 0xFF;
        DDRL = 0xFF;

        // Set up INT0, INT1, INT2 for rising edge
        EICRA |= _BV(ISC01) | _BV(ISC00);
        EICRA |= _BV(ISC11) | _BV(ISC10);
        EICRA |= _BV(ISC21) | _BV(ISC20);

        // Set up INT3, INT4 for falling edge
        EICRA |= _BV(ISC31);
        EICRB |= _BV(ISC41);

        // Enable External Interrupt Flags
        EIMSK |= (_BV(INT0));
        EIMSK |= (_BV(INT1));
        EIMSK |= (_BV(INT2));
        EIMSK |= (_BV(INT3));
        EIMSK |= (_BV(INT4));

        // Initializing PWM and setting duty cycle
        TCCR0A |= 0x03;
        //TIMSK0 |= 0x02;
        TCCR0A |= 0x80;
        TCCR0B |= 0x03;
        OCR0A = 0x00;
```

```
// Configure ADC
ADCSRA |= _BV(ADEN);
ADCSRA |= _BV(ADIE);
ADMUX |= _BV(REFS0);

// Enable all interrupts
sei();

// Rotate stepper until HE sensor is triggered
initStepper(&curPhase);

// Start conveyor belt
PORTB = 0x0E;

goto POLLING_STAGE;

// POLLING STATE
POLLING_STAGE:
// If the pause button has not been pressed
if (!isPaused) {
        mTimer(5000);
        LCDClear();
        // If queue is empty, speed up belt
        if(size(&head, &tail)==0) OCR0A = EMPTYSPEED;
        else OCR0A = SORTINGSPEED;
}
// Otherwise display sorted items and any unprocessed items
else{
        mTimer(5000);
        LCDWriteStringXY(0,0,"BL");
        LCDWriteIntXY(0,1,numBL, 2);
        LCDWriteStringXY(3,0,"WH");
        LCDWriteIntXY(3,1,numWH, 2);
        LCDWriteStringXY(6,0,"ST");
        LCDWriteIntXY(6,1,numST, 2);
        LCDWriteStringXY(9,0,"AL");
        LCDWriteIntXY(9,1,numAL, 2);

        LCDWriteStringXY(12,0,"BELT");
        LCDWriteIntXY(12,1,size(&head, &tail), 2);
}

// State Machine
switch(STATE){
        case (0) :
        goto POLLING_STAGE;
        break;
        case (1) :
        goto MAGNETIC_STAGE;
        break;
        case (2) :
        goto REFLECTIVE_STAGE;
        break;
        case (3) :
        goto BUCKET_STAGE;
        break;
        case (5) :
        goto END;
        default :
        goto POLLING_STAGE;
}//switch STATE
```

```
        MAGNETIC_STAGE:
        STATE = 0;
        goto POLLING_STAGE;

        REFLECTIVE_STAGE:
        STATE = 0;
        goto POLLING_STAGE;

        BUCKET_STAGE:
        // Keep track of the number of times the exit sensor has been triggered
        // and ensure they have all been handled
        while (exitCounter>0){
                // dequeue item
                dequeue(&head, &tail, &rtnLink);

                // Classify item based on item code given and set target position
                if(rtnLink->e.itemCode==0) {
                        targPos = 0;
                        numBL++;
                }
                else if(rtnLink->e.itemCode==1) {
                        targPos = 180;
                        numWH++;
                }
                else if(rtnLink->e.itemCode==2) {
                        targPos = -90;
                        numST++;
                }
                else if(rtnLink->e.itemCode==3) {
                        targPos = 90;
                        numAL++;
                }
                // free memory and drive stepper to target position
                free(rtnLink);
                drive_stepper(targPos, &Position, &curPhase);
                // decrement counter
                exitCounter--;
        }
        STATE = 0;
        goto POLLING_STAGE;

        END:
        // Brake motor and display total sorted items
        PORTB = 0x0F;
        LCDClear();

        LCDWriteStringXY(0,0,"BL");
        LCDWriteIntXY(0,1,numBL, 2);
        LCDWriteStringXY(3,0,"WH");
        LCDWriteIntXY(3,1,numWH, 2);
        LCDWriteStringXY(6,0,"ST");
        LCDWriteIntXY(6,1,numST, 2);
        LCDWriteStringXY(9,0,"AL");
        LCDWriteIntXY(9,1,numAL, 2);
        return(0);

}

// Ramp Down
ISR(INT0_vect){
        mTimer(3000);
        if(!(PIND & 0b00000001) == 0){
                rampDown();
        }
```

```
}

// Pause button
ISR(INT1_vect){
	mTimer(3000);
	if(!(PIND & 0b00000010) == 0){
		if (!isPaused) {
			PORTB = 0x0F;
			isPaused = 1;
		}
		else {
			PORTB = 0x0E;
			isPaused = 0;
		}
	}
	while(!(PIND & 0b00000010) == 0);
	mTimer(3000);
}

// OR Sensor
ISR(INT2_vect){
	// added 3ms delay to handle sensor bouncing
	mTimer(300);
	if((PIND & 0x04) != 0){
		// reset ADC_Val to max value each time and start a conversion
		ADC_Val = 1023;
		ADCSRA |= (1 << ADSC);
		STATE = 2;
	}
}

// Exit Sensor
ISR(INT3_vect){
	if((PIND & 0x08) == 0) {
		// Brake motor, reduce speed of belt, and increment counter
		PORTB = 0x0F;
		OCR0A = SORTINGSPEED; //25%
		STATE = 3;
		exitCounter++;
	}
	mTimer(300);
}

ISR(INT4_vect) {
	// Set flag to indicate stepper motor is in correct position
	HE_Found = 1;
}

ISR(ADC_vect) {
	// Store current ADC value
	ADC_result = ADC;
	// if OR sensor is still blocked update ADC_Val with new lowest value
	if ((PIND & 0x04) != 0) {
		if (ADC_result < ADC_Val) {
			ADC_Val = ADC_result;
		}
		// Start a new conversion
		ADCSRA |= (1 << ADSC);
	}
	// if OR sensor is no longer blocked, classify item and add it to the queue
	else {
		if (ADC_Val > RANGE_3) itemClass = 0;
	// Black
		else if (ADC_Val > RANGE_2 && ADC_Val < RANGE_3) itemClass = 1; // White
```

```
                else if (ADC_Val > RANGE_1 && ADC_Val < RANGE_2) itemClass = 2; // Steel
                else itemClass = 3;
                        // Aluminum
                initLink(&newLink);
                newLink->e.itemCode = itemClass;
                enqueue(&head, &tail, &newLink);
        }
}

ISR(BADISR_vect) {
}

// Function to rotate the stepper motor until the hall effect sensor interrupt
// is triggered
void initStepper(int *curPhase) {
        while(HE_Found == 0) {
                switch(*curPhase) {
                        case 0:
                        PORTA = POS_1;
                        mTimer(TIMER_DELAY);
                        *curPhase = 1;
                        break;

                        case 1:
                        PORTA = POS_2;
                        mTimer(TIMER_DELAY);
                        *curPhase = 2;
                        break;
                        case 2:
                        PORTA = POS_3;
                        mTimer(TIMER_DELAY);
                        *curPhase = 3;
                        break;
                        case 3:
                        PORTA = POS_0;
                        mTimer(TIMER_DELAY);
                        *curPhase = 0;
                        break;
                }
        }
}

// This function moves the stepper motor based on three inputs: target position,
// the current position, and the current phase that is active
void drive_stepper(int targPos,int *curPos, int *curPhase) {
        // calculate the number of degrees the stepper needs to turn
        int degrees = targPos - (*curPos);
        // if it does not need to move, start the belt and wait for 20ms so as not
        // to start turning too early
        if(degrees == 0){
                PORTB = 0x0E;
                mTimer(2000);
        }
        // These two conditions ensure that the fastest possible route to the next
        // position is taken
        if(degrees>180) degrees-=360;
        else if(degrees<-180) degrees+=360;
        // If degrees is negative, then we multiply by -1 and make the stepper Rotate
        // counter-clockwise
        if (degrees < 0) {
                degrees *= -1;
                isCW = 0;
        }
        // calculate the number of steps needed to move
```

```
	float rawSteps = degrees/STEPINCREMENT;
	// If the difference between rawSteps and discreteSteps is greater than 0.5
	// then we increment by 1, since the float rounds down automatically
	int discreteSteps = rawSteps;
	if ((rawSteps - discreteSteps) > 0.5) {
		discreteSteps++;
	}
int isCW = 1;
	int i = 0;
	// for loop to rotate desired number of steps that decrements each time
	for(discreteSteps; discreteSteps > 0; discreteSteps--) {
		// Condition to accelerate the stepper motor
		if(i<timerArraySize && i<(rawSteps/2)){
			TIMER_DELAY = stepDelay[i];
			i++;
		}
		// condition to decelerate the stepper motor
		else if(i==discreteSteps){
			TIMER_DELAY = stepDelay[i-1];
			i--;
		}
		// if there are 40 steps left, start the belt so that the piece falls off
		// into the center of the bucket
		if(discreteSteps==40){
			PORTB = 0x0E;
			OCR0A = 0x60;
		}
		// when there is one step left, slow the belt down
		else if(discreteSteps==1){
			OCR0A = SORTINGSPEED;
		}

		// Switch case to keep track of the phases of the stepper motor
		switch(*curPhase) {
			case 0:
			if(isCW) {
				PORTA = POS_1;
				mTimer(TIMER_DELAY);
				*curPhase = 1;
				break;
			}
			else {
				PORTA = POS_3;
				mTimer(TIMER_DELAY);
				*curPhase = 3;
				break;
			}
			case 1:
			if (isCW) {
				PORTA = POS_2;
				mTimer(TIMER_DELAY);
				*curPhase = 2;
				break;
			}
			else {
				PORTA = POS_0;
				mTimer(TIMER_DELAY);
				*curPhase = 0;
				break;
			}
			case 2:
			if (isCW) {
				PORTA = POS_3;
				mTimer(TIMER_DELAY);
```

```
                                *curPhase = 3;
                                break;
                        }
                        else {
                                PORTA = POS_1;
                                mTimer(TIMER_DELAY);
                                *curPhase = 1;
                                break;
                        }
                        case 3:
                        if (isCW) {
                                PORTA = POS_0;
                                mTimer(TIMER_DELAY);
                                *curPhase = 0;
                                break;
                        }
                        else {
                                PORTA = POS_2;
                                mTimer(TIMER_DELAY);
                                *curPhase = 2;
                                break;
                        }
                }
                *curPos=targPos;
        }
        // 80ms delay to allow for the stepper to finish turning before the belt
        // starts moving again
        mTimer(8000);
}

// Function to initialize Timer 4 and start counting for an arbitrary amount
// of time
void rampDown(){

        cli();

        TCCR4A = 0x00;

        TCNT4  = 0;

        OCR4A = 0xC800;

        TCCR4B |= (1 << WGM12);

        TCCR4B |= (1 << CS12) | (1 << CS10);

        TIMSK4 |= (1 << OCIE4A);

        sei();

}

// When Timer 4 reaches the expected count, this interrupt fires and takes the
// program to the END state
ISR(TIMER4_COMPA_vect){
        STATE = 5;
}


void initClockandTimer() {

    CLKPR = 0x80;

    CLKPR = 0x01;       // sets system clock to 8MHz
```

```
    TCCR1B |= _BV(CS11);    // sets timer/counter control register prescaler to 8
}


void mTimer(int count) {
    int i = 0;


    TCCR1B |= _BV(WGM12);


    //OCR1A = 0x03E8; //Sets output compare register to 1ms
    OCR1A = 0x0A; //Sets output compare register to 10us


    TCNT1 = 0x0000; //Sets initial value of timer to 0


    //TIMSK1 = TIMSK1 |= 0b00000010; //This needs to be disabled if global interrupts enabled


    TIFR1 |= _BV(OCF1A);


    while(i < count)
    {
        if((TIFR1 & 0x02) == 0x02) {
            TIFR1 |= _BV(OCF1A);
            i++;
        }
    }
    return;
}
```