MECH 458 Project Report
Department of Electrical Engineering

Instructor: Kunwu Zhang

Date: April 22nd, 2020

Author:

1. Connor Wiebe

Student Number V00872959

2. MINH DAI TRAN

Student Number V00928014

Group 6

Executive Summary

The purpose of this project is to create a system for sorting objects on a conveyor belt into a sorting tray.

A circuit was designed to interconnect an at90usb1287 microcontroller with user input buttons, the system's sensors, and two motors.

The interface consists of two buttons, to start/stop the sorting process, and to initiate calibration. A DC motor drives the conveyor belt and a stepper motor drives the sorting tray. An analog reflectivity sensor determines the object material, and an optical sensor at the belt exit controls the conveyor belt.

A program was written in C to operate the sorting process.

The system successfully fulfills the design requirements: To sort between four objects types of different materials, in any order, queuing at least 4 items on the belt at a time. This was validated through our testing procedure. At the end of the report, we discuss the system limitations and features that we could have included or improved on.

Table of Contents

List of Figures

1. Project Description

The purpose of this project is to implement an object sorting system. The system consists of a conveyor belt, a sorting tray with four sections for four different materials, an exit sensor at the end of the conveyor belt and a reflective sensor.

Four different types of objects: *black, white, steel, and aluminium* are placed on the conveyor belt and sorted placed onto their respective segment of the sorting tray.

2. System Summary

2a. High level block diagram

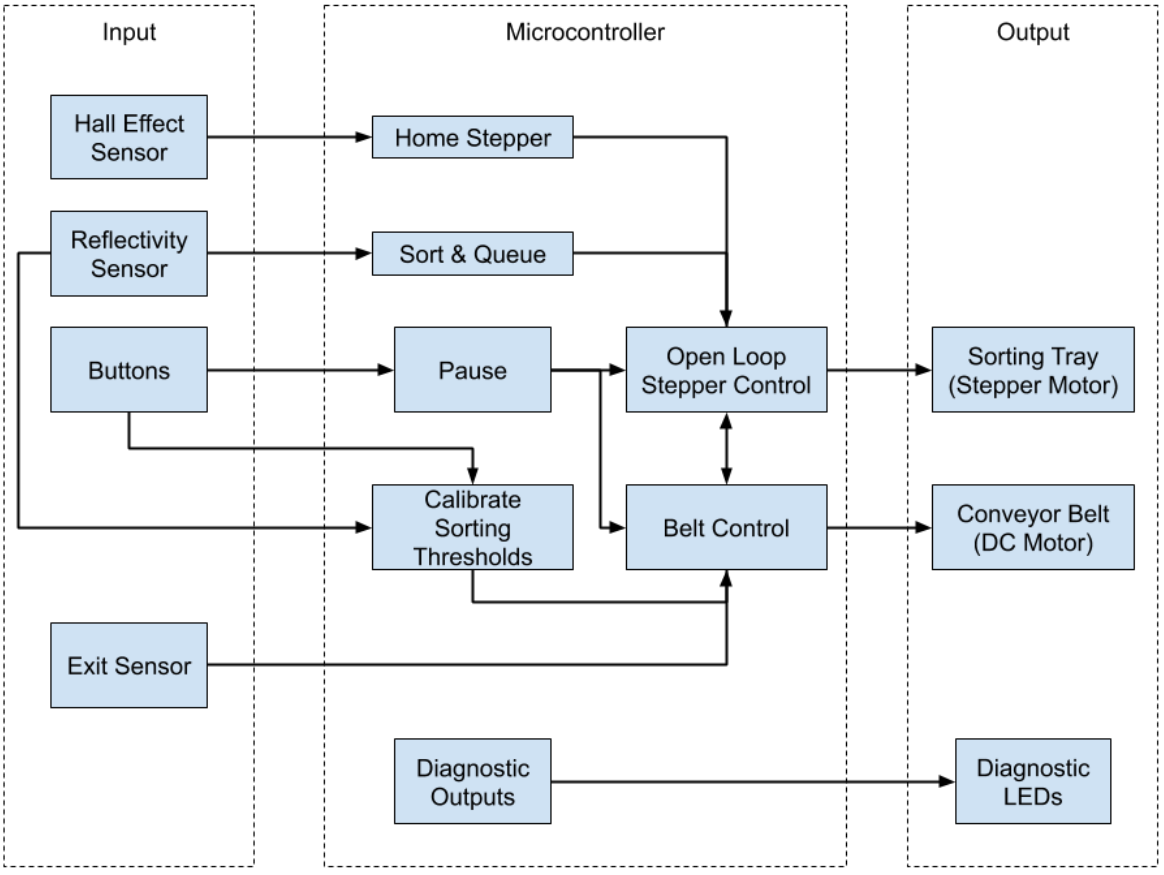

Figure 1: High Level Block Diagram

2b. Circuit Diagram

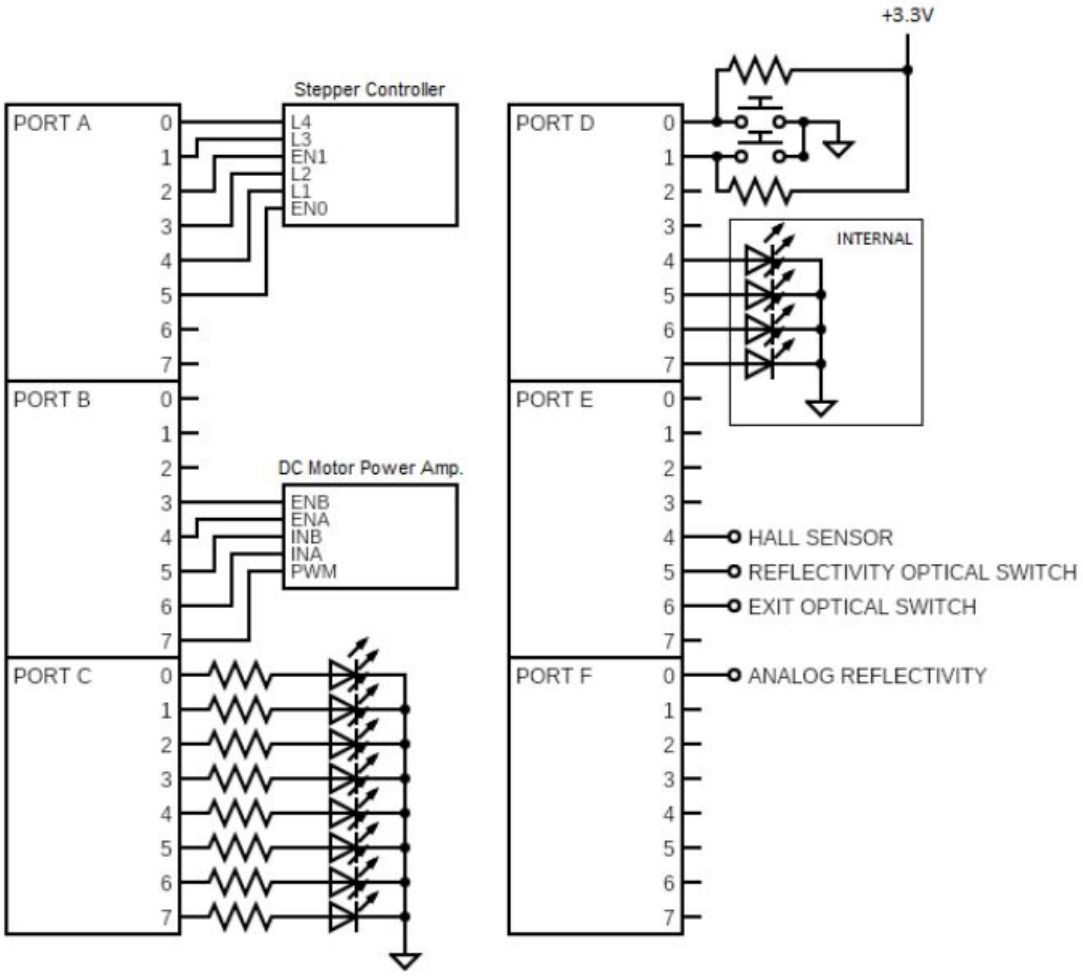

Figure 2: System Circuit Diagram

2c. System Algorithm

- Pause:
  - Button Pressed:
    - Stops Conveyor Belt
    - Moves tray to its destination
    - Wait for the pause button to be pressed again
- Ramp Down:
  - Button Pressed:
    - Set RAMP_DOWN flag
  - When the queue is empties:
    - Stops motors
- Reflectivity Sensor:
  - Laser tripped:

        - Start conversions
        - Continuously store lowest conversion value
    - Laser Closes
        - Stop Conversions
        - Classify based on minimum value
        - Add to main queue
- Exit Sensor:
    - Laser tripped:
        - If tray is not in position, stop conveyor belt
    - Laser closes
        - Read from the queue
        - Run conveyor belt
- Hall Sensor:
    - Homing
        - Turns clockwise until the sensor fires, then sets the home position, and offset to the center of the nearest bin
        - Run-Time adjustment:
            - Resets the step position whenever the sensor fires, if turning clockwise
- Conveyor Belt:
    - Only stops if an item is preset, and the tray is not in position for it to be dropped

2d. Flow Chart

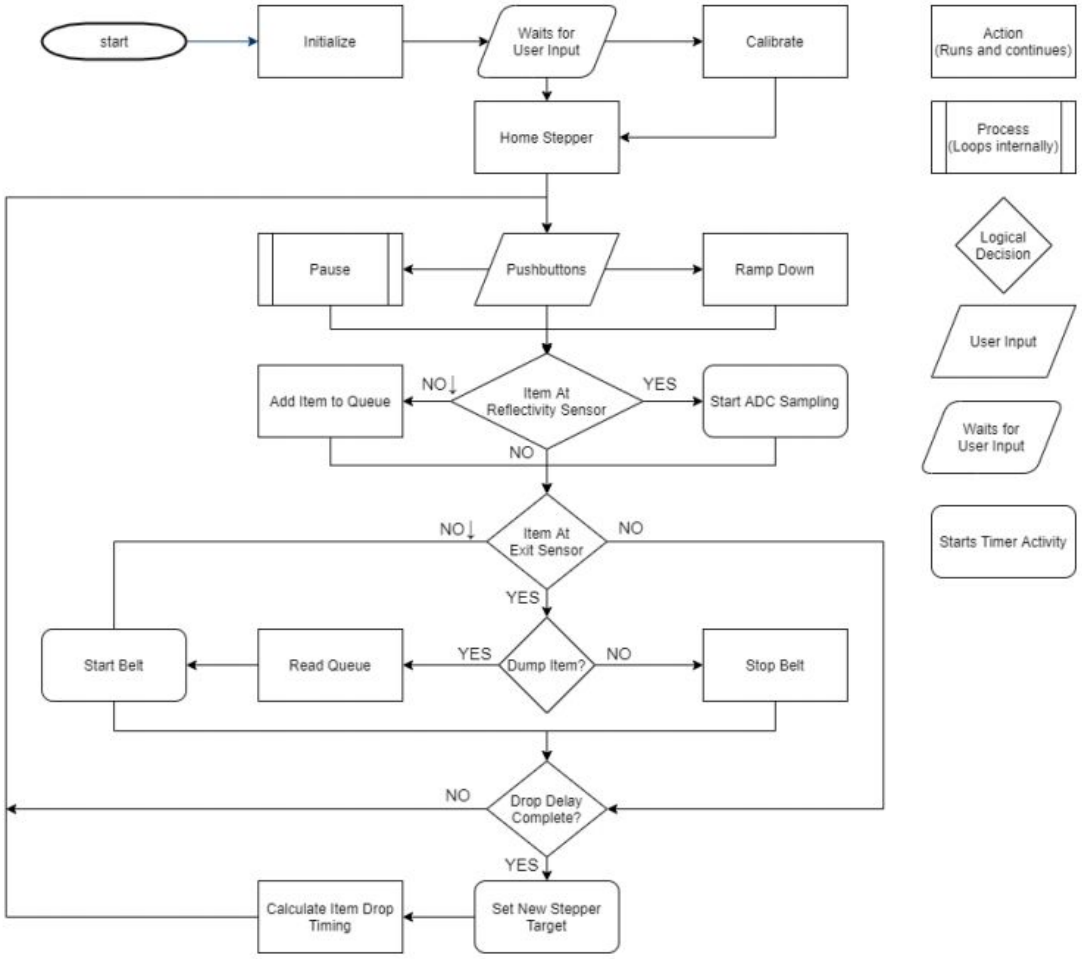

Figure 3: System Flow Chart

3. Performance Specification

The stepper motor position control uses a trapezoidal acceleration curve, so the time to move between two points is directly proportional to the acceleration constant, until the peak speed in the curve reaches maximum speed.

This stepper driver is double stepping, so there are 400 steps per revolution.

After a dispense command dispenses an item from the belt, the item must travel from the belt exit to the tray; a delay which we call the item drop time. During the drop time the stepper is delayed from starting its next movement.

A dispense command starts the belt when the tray has come within a certain number of steps to its destination - this is the angle buffer. At zero, the item is

dispensed only after the tray has stopped, and increasing the buffer allows the item to be dispensed while it is still in motion, to prevent the belt from having to wait for the item to fall from the belt.

The system overall is limited by the speed at which the stepper can move - the conveyor belt is capable of delivering very many items in a short amount of time. Therefore, to maximize performance, the acceleration curve should be optimized for point-to-point speed, and the timing should be calibrated to minimize stepper downtime - i.e. the stepper should always be moving.

3a. Maximum Specification

Acceleration of the stepper was tested to function up to 1700 steps/s^2, or 4.25 r/s^2 beyond which stuttering occurred. This acceleration is a constant value, for simple linear acceleration curves.

The DC motor is capable of running at maximum speed, only being switched on and off. What speed and acceleration this represented was not measured.

The drop time should be set to an accurate measured drop time for a dispense command. This prevents any unnecessary waiting after a dispensation is completed. If the angle buffer is exactly calibrated, this delay will have no effect.

The angle buffer should be set to an angle such that a deceleration over that angle will finish in exactly *drop time*. Therefore the bin will reach its destination at the same time the item lands in the bin, given the trapezoidal acceleration curves.

Instead of being measured, this value can be calculated by summing the elements in the acceleration curve's frequency array - the index where the sum is larger than drop time is the optimal angle buffer.

3b. Recommend Specification (For remote testing)

For stable condition acceleration is set to 1000 steps/s^2, to allow for differences in the stations' stepper motors, with a large margin of error.

The DC motor is set at 62% of its maximum speed, which we subjectively considered to be 'reasonably fast' and which did not cause overshoot problems when braking at the exit sensor.

In the remote test, the drop time is set to 400ms, a deliberate overestimate, to ensure correct sorting. From watching video of the machine, the true value is near ¼

of a second; the difference between these times would be directly subtracted from the sorting time of each item - approximately 2 seconds over 16 items.

To ensure the objects land in the center of their bin, both the angle buffer and drop time should be run with a safety margin - the drop time 100ms above the measured value, and the angle buffer set to 50% of its maximum accurate value.

4. Testing Procedures

Procedure:

1) Calibration:
    a) Start the calibration process by pressing button 2.
    b) Feed several black items through the reflectivity sensor then press button 2. Repeat this step for white then steel items.(Ignore aluminium)
    c) Read the reflectivity value from the LED display. In order, they are black, white and steel. Make sure that these values make sense.
    d) If the reading on the LED display matches with our expected value that means the sensor is working properly. Black and white items are very to each other in values and we need to make sure the system can differentiate between these two items.
2) Press button 1 to start the program:
    a) Watch the stepper motor home, note whether or not it spins clockwise and stops at the center of the black bin.
    b) If the stepper home to the correct position, that means the system has the correct position for each bin for the sorting process later.
3) When both the reflectivity value make sense and the stepper homing works correctly, start the sorting process. Load items on the conveyor belt, 4 at a time to a total of 48 items.
4) After the system finishes sorting, check the tray to make sure that all 48 items are sorted correctly. Check the count variable and make sure that it manages to keep track of all 48 items.
    a) If the system keeps count of all 48 items as well as sort those items correctly. This means that the system fulfills its sorting functionality with no error.

5) Press button 1 to start the system again, place a few items on the sensor. Press 1 again and observe if the system pauses. Press 1 again and make sure the system starts from where it left off.
    a) If the system pauses and continues where it left off with button 1 press, this means that the pause functionality works as intended. The user can pause and continue as they want by pressing a button.

5. System Limitation

The ramping functionality has been removed so that the system can successfully sort the objects in the limited amount of testing now that we have limited testing time due to remote work.

The optimal drop time has been neglected so that the stepper motor can operate in conjunction with the DC motor. We also had to alter our testing procedure to find the optimal speed for both motors. This optimal speed test procedure would involve finding the maximum speed the system could operate while yielding an accurate sorting result. These limitations result in the system unable to operate any faster than it already is.

The system may not also be able to calibrate correctly under extreme humidity and dusty conditions.

6. Reflection

There are many things we could have done to further improve our design, espeicallying in the speed department. Due to testing limitations we were unable to implement our ramping functionality which can improve the speed of the system. Furthermore, we could further tweak our speed value on the conveyor belt and sorting tray to improve the overall speed of the system. This includes finding the optimal drop time of the items. Another feature that can potentially improve our design would be a touch screen interface that has all the values and buttons labeled on the screen.

This project was a valuable experience in both introducing us to mechatronic design and optimization. We may not have been able to do as much with the project as we first expected to but we have learned how to optimize our work load, when having to do remote work with limited testing time.

Appendix A – Object Sorting Program

```
/*
 Milestone 5
 Program 1
 Project: 458 Object Sorter
 Group: 6
 Name 1: Connor Wiebe V00872959
 Name 2: Dai Minh Tran V00928014
 Desc: Sort 4 kinds of items into bins
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <math.h>
#include <stdlib.h>
#include "stepperDriver.h"
#include "myqueue.h"
#include "dcDriver.h"

//      //      //      //      GLOBALS      //      //      //      //

#define DROP_TIME 400
#define DC_SPEED 160

/*      reflectometer   */
int THRESH_BLACK = 941;
int THRESH_WHITE = 699;
int THRESH_STEEL = 350;

volatile int Reflectometer_latch=0;             //ADC stops reading when unlatched
volatile int albedo = 1023;                     //maximum value on an item
int numA=0, numS=0, numW=0, numB=0;             //number of each type sorted

char read_next_item_flag = 1;   //Flag to read the next item from the queue - prevents errors if the queue is empty.
volatile int belt_count = 0;    //Number of items on the belt

/* exit */
volatile int exit_latch = 0;
volatile int exit_unlatch_enable = 0;

/*      debounce        */
volatile int Button1_Flag = 0, Button2_Flag = 0;        //Button presses raise a flag when pressed
volatile int Button1_Enable = 1, Button2_Enable = 1;    //and are disabled for a debounce time
volatile int dispensing = 0;

volatile int t3A=0, t3B=0, t3C=0;                       //The debounce timer uses three channels: A,B,C
volatile int *t3A_ptr, *t3B_ptr, *t3C_ptr;
volatile int t3A_set, t3B_set, t3C_set;

void init_debounce();
void debounce(volatile int* flag_var, int ms, int set);     //Dynamic channel allocation
```

```
void debounceA(volatile int* flag_var, int ms, int set);
void debounceB(volatile int* flag_var, int ms, int set);
void debounceC(volatile int* flag_var, int ms, int set);
void mTimer(int ms);

void initADC();
void Pause();

//        //        //        //          MAIN   //        //        //        //
int main(void)
{

/*        queue   */
Q Item_Q;
Item_Q = Q_new();
material next_item = ERROR;     //the material of the next item to be added to the queue

/*         reflectometer   */
int reflectometer_in = 0;
int rf_smooth = 0;

/*         exit      */
int exit_in = 0;
int exit_smooth = 0;

/*         tray      */
int dest_steps = 0;                 //Steps for calculating movements
material dest_tray = BLACK;         //init as not error, to avoid a sort


/*    initialize Clock       */
CLKPR = (1<<CLKPCE);                //enable clock prescaler editing
CLKPR = 0x00;                       //increase clock speed to 8MHz

/*    initialize Buttons    */
EICRA |= 1<<ISC01;                  //Set INT0 to Falling edge (Button1)
EIMSK |= 1<<INT0;                   //Enable INT0
EICRA |= 1<<ISC11;                  //Set INT1 to Falling edge (Button2)
EIMSK |= 1<<INT1;                   //Enable INT1

/*        initialize IO pins */
DDRC = 0xFF;    //LED output
DDRD = 0x60;    //other LEDs
DDRE = 0x00;    //sensors input
DDRF = 0x00;    //analog input

init_debounce();            //Uses Timer 3 – debounce timer counts in ms
initADC();
initStepper();              //Uses Timer 2
initDCmotor();              //Uses Timer 0 (PWM) - calls sei()
enableDCmotor();
```

```
//      //      //      //      //      //      Startup //      //      //      //      //      //

homeStepper();                  //Home the stepper motor (holds CPU)

//Note: Calibration method removed – the appropriate values were specified to speed up remote testing
// The method would simply run the belt, and store the albedo values as thresholds, In order B->W->S->A

setDCspeed(DC_SPEED);           //Start the belt


//      //      //      //      //      //      MAIN LOOP      //      //      //      //      //      //
  while (1)
  {
        if (Button1_Flag) Pause(&Item_Q);               //PAUSE: stop belt - finish tray move
                                                        //UNPAUSE: restart belt, normal operation

//Reflectivity Sensor

        //smooth out the signal
        reflectometer_in = ((PINE & (1<<PINE5)) == 1<<PINE5);
        if (reflectometer_in){
                if (++rf_smooth > 8) rf_smooth = 8;
        }else{
                if (--rf_smooth < 0) rf_smooth = 0;
        }

        //latch on rising edge
        if ((rf_smooth==8) && (!Reflectometer_latch)){
                Reflectometer_latch = 1;
                ADCSRA |= 0x1<<ADSC;                    //Start conversions


        //unlatch from falling edge
        }else if ((rf_smooth==0) && (Reflectometer_latch)){
                Reflectometer_latch = 0;

                next_item = ALUMINUM;                   //determine the material
                if (albedo > 350) next_item = STEEL;
                if (albedo > 699) next_item = WHITE;
                if (albedo > 941) next_item = BLACK;
                albedo = 1023;                          //Reset albedo limits

                Q_add(&Item_Q, next_item);              //Add the item to the queue
                belt_count++;
        }
```

```
//Exit Sensor

        //smooth out the signal
        exit_in = ((PINE & (1<<PINE6)) == 0);
        if (exit_in){
                if (++exit_smooth > 16) exit_smooth = 16;
        }else{
                if (--exit_smooth < 0) exit_smooth = 0;
        }



        //latch on rising edge
        if (exit_smooth==16 && !exit_latch && belt_count>0){
                exit_latch = 1;
                setDCspeed(0);
        }

        //unlatch from falling edge
        if (exit_smooth==0 && exit_latch && exit_unlatch_enable){
                exit_latch = 0;
                exit_unlatch_enable = 0;
        }



//Item Dispensing

        if (exit_latch  && !read_next_item_flag && !dispensing){        //If item is ready to be sorted
                if (!stepperRUNNINGflag) {                              //Wait for the tray to stop
                        setDCspeed(DC_SPEED);                           //Run belt to dump item

                        dispensing = 1;
                        debounce(&dispensing, DROP_TIME, 0);            //Don't move tray until the item lands
                        debounce(&exit_unlatch_enable,75, 1);           //Don't stop the belt on the same item

                        //There may be only 1 item in the queue, so read and dump aren't always simultaneous
                        read_next_item_flag = 1;
                        belt_count--;

                        switch (dest_tray){
                                case ALUMINUM: numA++; break;           //count the number of each type sorted
                                case STEEL: numS++; break;
                                case BLACK: numB++; break;
                                case WHITE: numW++; break;
                                default: break;
                        }
                }
        }
```

```
//Stepper Move Control

        // When an item is dispensed, the dispensing, and read_next_item flags are set
        // dispensing is reset on a timer, calibrated based on how long the item takes to reach the bin
        // read_next_item is reset here, when another item is on the belt.

        if (read_next_item_flag && Item_Q.size>0 && !dispensing){
                read_next_item_flag = 0;

                dest_tray = Q_read(&Item_Q);            //Remove the old item, get the next item's type

                dest_steps = dest_tray*100;             //ALUMINUM - WHITE  00 - 01 | 0 - 1   This is how it
                                                        //BLACK - STEEL     11 - 10 | 3 - 2   is in the enum

                //get Clockwise distance, and choose the smaller angle of CW and CCW to move

                int dTheta = (dest_steps - SM.pos_steps);
                if (dTheta > STEPS_PER_REV/2) dTheta = dTheta - STEPS_PER_REV;
                else if (dTheta < -STEPS_PER_REV/2) dTheta = STEPS_PER_REV + dTheta;

                turnStepper(dTheta);                    //Command the stepper to move
        }

    }//loop
}//main


void Pause(Q *q){

        //Stops the belt, moves the tray to the destination of the next item
        setDCspeed(0);                                  //Stop belt
        mTimer(200);

        volatile int Display_Delay = 0;
        volatile int Display_Index = 0;
        Button1_Flag=0;

        //Display loop
        while(!Button1_Flag){

                if (!Display_Delay){

                        switch (++Display_Index){
                                case 6: Display_Index = 1;          //restarts loop
                                case 1: PORTC = 0xF0 | belt_count; break;         //# On Belt
                                case 2: PORTC = 0x10 | numB; break;                       //# sorted
                                case 3: PORTC = 0x20 | numS; break;                       //# sorted
                                case 4: PORTC = 0x40 | numW; break;                       //# sorted
                                case 5: PORTC = 0x80 | numA; break;                       //# sorted
                        }
```

```
                Display_Delay = 1;
                debounce(&Display_Delay,1000,0);        //wait 1 second between displays
            }

        }
        PORTC = 0x0;

        while(stepperRUNNINGflag);              //wait for the stepper to stop

        setDCspeed(DC_SPEED);                   //restart the belt

        mTimer(500);                            //debounce after leaving pause
        Button1_Flag=0;
        Button1_Enable = 1;
}


void initADC(){

        //REFS(0,1) = 01 -> +reference AVCC with ext. cap on AREF pin
        //MUX4:0 selects ADC number (ADC1 - PORTF1)
        ADMUX = 0b01000001;

        ADCSRA |= 0x1<<ADEN;            //enable ADC
        ADCSRA |= 0x1<<ADIE;            //enable interrupt
}


ISR(ADC_vect){

        if (ADC < albedo) albedo = ADC;
        //Start a new conversion - maybe
        if (Reflectometer_latch) ADCSRA |= 1<<ADSC;    //start again
}


//Button 1 is on PD0, for INT0)
ISR(INT0_vect){
        if (Button1_Enable){
            Button1_Flag = 1;
            Button1_Enable = 0;
            debounce(&Button1_Enable,300,1);
        }
}


//Button 2 is on PD1, for INT1)
ISR(INT1_vect){
        if (Button2_Enable){
            Button2_Flag = 1;
            Button2_Enable = 0;
```

```
                debounce(&Button2_Enable,300,1);
        }
}


volatile int mCounter;
void mTimer(int ms){

        mCounter= 0;
        TCCR1A = 0x00;
        TCCR1B |= 0x1<<WGM12 | 1<<CS11; //CTC, and CLOCK /8

        OCR1A = 0x03e8;         //Set output compare to 1000 [0x03e8 = 1000] (1ms)
        TCNT1 = 0x0000;         //Reset initial value of Timer Counter 1

        TIMSK1 = 0x2;           //Enable output compare interrupt enable
        TIFR1 |= 0x2;           //Clear interrupt flag and begin timer

        while(mCounter < ms){}

        TIMSK1 = 0;     //Disable interrupt
}

ISR(TIMER1_COMPA_vect){
        mCounter++;
}


//////    DEBOUNCE TIMERS       //////

void init_debounce(){
        //16 bit timer
        TCCR3A = 0x00; //normal compare mode
        TCCR3B = 1<<WGM32 | 1<<WGM33 | 1<<CS32 | 1<<CS30;       //prescaler /1024
        TIMSK3 = 0x01;          //keep Overflow on, so the timer can restart
}

void debounce(volatile int* flag_var, int ms, int set){
        // debounce sets or resets a variable after a given time

        if (!t3A) debounceA(flag_var, ms, set);
        else if (!t3B) debounceB(flag_var, ms, set);
        else if (!t3C) debounceC(flag_var, ms, set);
        //otherwise buffer it - but implement that later, if it ever comes up
}

void debounceA (volatile int* flag_var, int ms, int set){
        //wait ms, then (re)set the flag variable
        t3A=1;                                  //claim channel
        t3A_ptr = flag_var;                     //store reference to flag
        t3A_set = set;                          //set or reset
```

```
	OCR3A = (TCNT3 + ms*8) % 0xFFFF;	//Set compare time in the future
	TIMSK3 |= 1<<OCIE3A | 1;		//enable Channel A interrupt & OVF
}
void debounceB (volatile int* flag_var, int ms, int set){
	//wait ms, then (re)set the flag variable
	t3B=1;					//claim channel
	t3B_ptr = flag_var;			//store reference to flag
	t3B_set = set;				//set or reset
	OCR3B = (TCNT3 + ms*8) % 0xFFFF;	//Set compare time in the future
	TIMSK3 |= 1<<OCIE3B | 1;		//enable Channel B interrupt & OVF
}
void debounceC (volatile int* flag_var, int ms, int set){
	//wait ms, then (re)set the flag variable
	t3C=1;					//claim channel
	t3C_ptr = flag_var;			//store reference to flag
	t3C_set = set;				//set or reset
	OCR3C = (TCNT3 + ms*8) % 0xFFFF;	//Set compare time in the future
	TIMSK3 |= 1<<OCIE3C | 1;		//enable Channel c interrupt
}

ISR(TIMER3_COMPA_vect){
	TIMSK3 &= ~(1<<OCIE3A);		//disable channel interrupt
	*t3A_ptr = t3A_set;		//(re)set flag
	t3A=0;				//relinquish channel
}
ISR(TIMER3_COMPB_vect){
	TIMSK3 &= ~(1<<OCIE3B);		//disable channel interrupt
	*t3B_ptr = t3B_set;		//(re)set flag
	t3B=0;				//relinquish channel
}
ISR(TIMER3_COMPC_vect){
	TIMSK3 &= ~(1<<OCIE3C);		//disable channel interrupt
	*t3C_ptr = t3C_set;		//(re)set flag
	t3C=0;				//relinquish channel
}
ISR(TIMER3_OVF_vect){
	TCNT3 = 0;	//Reset timer on overflow
}
```

Appendix B – DC Driver

```
/*
 * dcDriver.h
 *
 * Created: 2020-02-15 5:20:59 PM
 *  Author: Connor
 */


#ifndef DCDRIVER_H_
#define DCDRIVER_H_

typedef enum {                  //Enum statement to define motor modes
 BrakeVCC = 0b00000000,
 CW =           0b00000010,
 CCW =          0b00000001
 //             ENA,ENB,INA,INB
 } dcm;

//      7       6       5       4       3       2       1       0
//      pwm     x       x       x       ENA     ENB     INA     INB (offset 0)

 #define DCM_PIN_OFFSET 0

 volatile char DCmotorRUNNING;
 volatile unsigned int DCmotorTIME;
 volatile char DCdirection;

 void initDCmotor();
 void enableDCmotor();
 void setDCspeed(unsigned char speed);
 void changeDCdirection();


#endif /* DCDRIVER_H_ */
```

```
/*
 * dcDriver.c
 *
 * Created: 2020-02-11 12:03:56 PM
 *  Author: Connor
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include "dcDriver.h"

//Assumes 8MHz clock

void initDCmotor(){
	stopDCmotor();		//Disable motor
	cli();			//disable global interrupts


	//DC motor output PORTB pins 3-7 (3/4/5/6 control, 7 PWM)
	DDRB = 0b10001111;

	//WGM Fast PWM mode (011)   - bit 3 set to 0 in register B
	//COM0A (10) - OC0A clears on match, and sets at TOP.

	TCCR0A = (0x1<<COM0A1)|(0x1<<WGM01)|(0x1<<WGM00);

	// CS mode /64 (011)
	TCCR0B = (1<<CS01 || 1<<CS00);		//488Hz  (8MHz / (256*prescaler)

	TCNT0 = 0x00;		//Reset Counter
	OCR0A = 0x80;		//default to 50% duty cycle
	TIMSK0 = 0x03;		//Compare A and overflow interrupts enabled

	sei();			//re-enable global interrupts

	TIFR0 = 0x03;		//Clear OCF0A and TOV0 flags, to start timer
	DCmotorTIME = 0;	//Reset runtime timer
	DCdirection = 1;	//set motor direction

}

void enableDCmotor(){
	OCR0A = 0xFF;		//Set minimum speed
	TIMSK0 = 0x02;		//Compare A interrupt enabled
	TIFR0 = 0x02;		//Clear OCF0A and TOV0 flags, and start timer
}
```

```
void setDCspeed(unsigned char speed){
        DCmotorRUNNING = 1;

        dcm MotorMode;                          //Motor mode enum – CW, CCW, or brake
        if (DCdirection==1) MotorMode = CW;     //Set direction
        else MotorMode = CCW;

        if (speed == 0){                        //No speed -> Brake
                MotorMode = BrakeVCC;
                DCmotorRUNNING = 0;
        }

        OCR0A = speed;                          //Set speed via PWM period (1=fast, 255=slow)

        //Set the MotorMode output on the control pins
        PORTB = ((PORTB & ~0x0F) | MotorMode);
}

void changeDCdirection(){
        char oldSpeed = OCR0A;
        setDCspeed(0);
        DCdirection = (DCdirection) ? 0 : 1;
        for (int i=0; i<1000000; i++){ }        //wait a while
        setDCspeed(oldSpeed);
}


ISR(TIMER0_COMPA_vect){
        //Must be written to allow PWM on interrupt
}
ISR(TIMER0_OVF_vect){
        if(DCmotorRUNNING) DCmotorTIME++; //motor runtime counter the runtime - 488Hz, or 2.049ms period
}
```

```
/*
 * stepperDriver.h
*  Author: Connor
 */

#ifndef STEPPERDRIVER_H_
#define STEPPERDRIVER_H_

 #define DEGREES_PER_STEP 0.9
 #define STEPS_PER_REV 400

 typedef enum {                    //Enum statement to define motor modes (four steps, double stepping, plus brake)
          Stepper_BrakeVCC = 0b111111,
            S1 = 0b110000,
           S12 = 0b110110,
            S2  = 0b000110,
           S23 = 0b101110,
            S3  = 0b101000,
           S34 = 0b101101,
            S4  = 0b000101,
           S41 = 0b110101          //        (EN1/L1/L2/EN2/L3/L4)
} StepperMode_t;
StepperMode_t StepperMode;

struct Stepper_Control{
          int pos_steps;
          int dest_steps;
          int stepCount;
          char Direction;
          float Frequency;
} SM;

int ACCEL;
volatile int Hall_in;

volatile int speed_index;
int Target_Period;
int OCR_arr[100];                  //index in steps, output in period

volatile int stepperRUNNINGflag;
volatile int stepperENABLEDflag;

void initStepper();
void enableStepper();
void disableStepper();
void homeStepper();
void turnStepper(int steps);
void Trapezoid();
void createRamp();
 //Note: when called, turnStepper will override a previous command, not queue.
```

```
#endif /* STEPPERDRIVER_H_ */
```

Appendix C – Stepper Motor Control

```
/*
 * stepperDriver.c
 */

 #include <avr/io.h>
 #include <avr/interrupt.h>
 #include <stdlib.h>
 #include "stepperDriver.h"

 //Assumes 8MHz clock - enable that yourself

 void initStepper(){
        SM.pos_steps = 0;
        SM.dest_steps = 0;
        SM.stepCount = 0;

        ACCEL = 1024;           //ticks/s^2 (default 1024)
        createRamp();

        DDRA |= 0x3F;  //Set Port A data direction

        //Initialize pulse timer
        TCCR2A = 0x02;          //CTC mode (WMG bit 1)
        TCCR2B = 0x07;          //prescaler /1024

        OCR2A = 0x255;          //Stepper starts stationary - choose lowest frequency

        TCNT2 = 0x00;           //Reset counter
        TIMSK2 = 0x00;          //Disable interrupts
        PORTA = 0x00;           //Set output to nothing (coast)

        EICRB |= 1<<ISC41;      //Set INT4 to Falling Edge        (Hall)
        EIMSK |= 1<<INT4;       //Enable INT4

        stepperENABLEDflag = 1;
 }


 void enableStepper(){
        stepperENABLEDflag = 1;                 //enable the output
        TCNT2 = 0x00;  //Reset counter
 }

void disableStepper(){
        stepperENABLEDflag = 0;
        SM.stepCount = 0;
        TIMSK2 = 0x00;
        PORTA = 0x00;  //Disable completely - coasts instead of braking
```

```
}

//Homes to BLACK (300)
void homeStepper(){
	EIMSK |= 1<<INT4;	//Enable INT4
	Hall_in = 0;
	while (!Hall_in){
		if (!stepperRUNNINGflag) turnStepper(16);	//move slowly until the sensor fires
	}

	TIMSK2 = 0;			//stop stepper motor
	stepperRUNNINGflag = 0;
	SM.stepCount = 0;

	turnStepper(-16);		//move to the center of BLACK
	SM.pos_steps = 300;		//set the position to BLACK's position
	EIMSK &= ~(1<<INT4);		//Disable INT4
}


//Hall effect sensor input
ISR(INT4_vect){
	Hall_in=1;
}

void turnStepper(int steps){
	//Move the stepper motor a specified number of steps CW (+) or CCW (-)
	//Use Timer2 to generate interrupts, which decrement the stepCount, and track the pos_steps

	if ((stepperENABLEDflag == 0) | (steps == 0)) return;	//Don't try to move 0 steps, or if disabled

	if (steps>0) SM.Direction = 1;		//Set motor direction
	else {
		SM.Direction = 0;
		steps = steps * -1;		//absolute number of steps
	}

	SM.stepCount = steps;

	TCNT2 = 0x00;		//Reset Timer 2
	TIMSK2 = 0x02;		//Enable Timer 2 interrupt
	TIFR2 = 0x03;		//Start Timer 2

	stepperRUNNINGflag = 1;
}
```

```
// Primary Stepper control section

ISR(TIMER2_COMPA_vect){

	Trapezoid();				//Changes period to affect trapezoidal acceleration

	if(SM.stepCount-- <= 0){		//stop timer on completion, and hold position
	********************* Might need to be more sophisstocated
		TIMSK2 = 0x00;
		stepperRUNNINGflag = 0;
	}else{

		if (SM.Direction == 1){		//FWD cycles forward
			switch (StepperMode){
				case S1: StepperMode = S12; break;
				case S12: StepperMode = S2; break;
				case S2: StepperMode = S23; break;
				case S23: StepperMode = S3; break;
				case S3: StepperMode = S34; break;
				case S34: StepperMode = S4; break;
				case S4: StepperMode = S41; break;
				case S41: StepperMode = S1; break;

				default: StepperMode = S1; break;
			}
			SM.pos_steps++;
			}else{
			switch (StepperMode){		//REV cycles backward
				case S4: StepperMode = S34; break;
				case S34: StepperMode = S3; break;
				case S3: StepperMode = S23; break;
				case S23: StepperMode = S2; break;
				case S2: StepperMode = S12; break;
				case S12: StepperMode = S1; break;
				case S1: StepperMode = S41; break;
				case S41: StepperMode = S4; break;

				default: StepperMode = S1; break;
			}
			SM.pos_steps--;
		}

		//Normalize position
		SM.pos_steps = SM.pos_steps % STEPS_PER_REV;
		if (SM.pos_steps < 0) SM.pos_steps += STEPS_PER_REV;

		//PORTC = SM.pos_steps;

		//Set the output to step the motor
		PORTA = StepperMode;
	}
```

```
}



void Trapezoid() {
// Changes period to affect trapezoidal acceleration
//speed_index defines the speed, as an index in the array of periods “OCR_arr”

//On each step the speed index is (in/de)cremented by 1, and is therefore equal to the number of steps needed to stop

//Therefore, the stepper will accelerate until speed_index == step_count, holding at max speed

        if (speed_index < SM.stepCount){
                if (speed_index >= 99) return;

                OCR2A = OCR_arr[++speed_index];
        }else{

                if (SM.stepCount > 99) return;

                OCR2A = OCR_arr[--speed_index];
        }
};


void createRamp(){
// populates an array of periods, such that indexing through at a rate equal to the periods ramps linearly in frequency

        float f_calc = 30.6;        //minimum frequency afforded by the clock

        OCR_arr[0] = 255;
        for (int i = 1; i<100; i++){
                f_calc += ACCEL/f_calc;
                OCR_arr[i] = 7813/f_calc;
        }
}
```

Appendix D – Queue Implementation

```
#ifndef MYQUEUE_H_
#define MYQUEUE_H_

typedef enum {
	STEEL = 2, WHITE = 1, ALUMINUM = 0, BLACK = 3, ERROR = -1
} material;

typedef struct Node {
	struct Node* next;
	material Material;
}Node;

typedef struct Q {
	Node* head;
	Node* tail;
	int size;
}Q;


Q Q_new();
void Q_add(Q* q, material lightness);
material Q_read(Q* q);
material Q_peek(Q* q);

#endif /* MYQUEUE_H_ */
```

```
#include <stdio.h>
#include <stdlib.h>
#include "myqueue.h"

//return a new empty queue
Q Q_new() {
	Q q = {
		.head = NULL,
		.tail = NULL,
		.size = 0
	};
	return q;
}

void Q_add(Q* q, material m) {
	Node* i = (Node*)malloc(sizeof(Node)); //allocates new memory location
	i->Material = m;				//set the value of the material in the new node
	q->size++;					//increase queue size

	if (q->head == NULL) {			//populate queue if it was empty
		q->head = i;
		q->tail = i;
		i->next = NULL;
		return;
	}

	q->tail->next = i;		//Link new node at the end of the queue if it was not empty
	q->tail = i;
}

material Q_read(Q* q) {
	if (q->head == NULL) return ERROR;	//Only read if something is there

	Node* temp = q->head;			//create reference to node at queue head
	material r_val = temp->Material;	//get the material at queue head

	q->head = q->head->next;		//point queue head at next item
	if (--q->size == 0) q->tail = NULL;	//point tail at NULL in empty list

	free(temp);				//free the Node memory

	return r_val;
}

material Q_peek(Q* q) {
	if (q->head == NULL) return ERROR;	//Only read if something is there

	Node* temp = q->head;			//create reference to node at queue head

	return temp->Material;			 //return the material at queue head
}
```