ABSTRACT

A conveyor belt sorting system was designed with the intent of sorting aluminum, steel, black, and white pieces accurately in an efficient manner. The primary apparatus was constructed prior to this project, therefore, with the exception of minor hardware requirements, this project was almost entirely focused on generating an efficient algorithm. The microcontroller was based on a ATmega 2560 chip, with software written and tested from ATmel Studio 7 and based on the C programming language. The final design utilizes an almost entirely interrupt driven algorithm to detect and sort each of these distinct pieces. By utilizing this method, computational time is only spent on tasks that must be completed, allowing for readings and conversions to occur as fast as possible. The final system meets all requirements, and was found to accurately sort 48 pieces in a time of 23 seconds. Further improvements to both hardware and software are expected to produce better results if pursued in the future.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This project intended to optimize the sorting process of a predesigned conveyor belt system by leveraging hardware and software capabilities.

As per project specifications a set of 48 cylindrical pieces must be sorted into corresponding spaces on a rotating platform controlled by a stepper motor, within a time no greater then 60 seconds. A conveyor belt is powered by a DC motor and controls the position of the cylindrical pieces and the stage of processing that is occurring. To appropriately sort and detect the position of each piece a number of sensors must be used.

Two optical sensors (OR and EX) at different stages of the process trigger interrupts within the MCU that can be leveraged to trigger different processes and track the position of the pieces along the conveyor belt. A reflective sensor (RL) connected to an analog port relays analog readings to the MCU, which are converted to a digital reading and used to sort the cylinders through distinct readings of reflectivity as required by project specifications.

Additional specifications of the project required that a system ramp down and pause system be put in place to stop the processes either after all processes have finished (ramp down) or immediately after a button has been pressed (pause). Once the system has come to a complete stop feedback on the state of sorting will be provided on an LCD screen. This will indicate how many pieces have been sorted for each type of piece, along with how many pieces are on the conveyor but need to be sorted.

The goal of the project is to utilize these sensor and control systems to optimize the sorting process of the cylinders to sort a set of 48 pieces in an efficient and accurate way.

2 DESIGN APPROACH AND METHODOLOGY

2.1 DESIGN APPROACH

To achieve optimal performance and adaptation to the asynchronous nature of the project, the sorting algorithm was designed to run almost entirely through interrupts. Features that were not time dependent, such as the LCD display during operation were not controlled by interrupts and occurred when time was available in the main function.

The first bottleneck that needed to be optimized was the stepper motor speed and acceleration profile. If the stepper motor was not instructed to operate at a maximum, or close to maximum, considerable time would be lost during the sorting stage. For this reason, the stepper had to be controlled through a timer ISR, which controlled the delay between steps by adjusting the compare register. This method reduced any additional delay and allowed the stepper to begin moving as soon as possible. The parameters which define the acceleration profile of the stepper motor are based on calibration results. The process for this is described in the calibration section to follow.

Optimization of the steppers motor path is dependent upon the pattern of pieces to be sorted, and therefore some computation is required to accurately react to these patterns. Since there was a finite number of drop patterns for any two parts only these two pieces needed to be compared

when optimizing the steppers motion. By comparing the two relative locations of the two pieces to be dropped, both the speed, direction of rotation, and rotation traveled can be optimized. Precise sorting was required in order to maximize the speed of the sorting system. Therefore, there needed to be some link between the conveyor belt and the stepper motor to ensure that processes were occurring at the precise instance in time. Since the precise time required for the stepper to reach a given position can be calculated within the software fairly efficiently, this can be coupled with a calibrated estimate of drop time. This in turn allows for maximum control between the stepper and the conveyor belt and ensures that parts are dropped as close to the absolute minimum amount of time required.

Further control of the conveyor belt itself was also unfortunately required due to slipping of pieces at higher speeds. Operating the conveyor belt at maximum speed was simply impractical, however reducing the speed of the conveyor also created a significant amount of time wasted just on transport time from the loading area to the end of the conveyor. In an attempt to find a middle ground of this issue two methods were implemented to address slipping. The conveyor was controlled with a form of speed control and deceleration was applied to the conveyor to minimize the jolt of a sudden stop. A flag system was also created to indicate if a slip had been detected for a piece that was supposed to be within the final exit sensor (EX). This was completed through a form of checking system between the conveyor and stepper motor to determine if the piece should have in fact dropped.

The final constraint that had to be addressed was for the sorting system allocating enough time for a high number of (analog to digital conversion) ADC readings to be completed by the reflective sensor (RL) for each part. If the number of readings were significantly reduced at any point during testing, parts would be identified incorrectly. In an attempt to achieve optimal results a PCINT interrupt was linked to GPIO pins and toggled by the software (PORT calls) to split up longer interrupts, ideally increasing time for ADC conversion of RL readings.

2.2 METHODOLOGY

To efficiently debug and test the sorting system four pieces were used to test different patterns that tested cases implemented in the software algorithm. The primary patterns that represent the methods of testing for this project are given below. For some cases, combinations of these test cases were used, however these are the underlying patterns.

1. Doubles
2. 90 Degree Variations
3. 180 Degree Variations
4. Clockwise/Counterclockwise Spin

The double pattern tested the algorithm's ability to address and efficiently sort two pieces of the same classification. In theory the stepper motor would not have to stop for any pieces dropped, and continuous rotation to the following part would optimize the sorting process. A double pattern challenges this, since the time allocation for each region would have to be longer for two pieces to drop, meanwhile the software needs to react in a way so as to not allow the sorting tray to come to rest.

The 90-degree variation test was used to benchmark the system's ability to maximize the sorting speed of pieces that require a 90 degree back and forth sorting process. Ideally, as little time as possible is required between each drop. This tested the software's ability to react to these situations, while not dropping to earlier or causing the stepper motor to lose steps during the process. Testing of this functionality focused on the parameters linking the control of the stepper and conveyor belt.

The 180-degree variation test was completed with similar purposes to that of the 90 degree variation test. The difference being that the system would have to be optimized to ensure that when these cases occur the stepper does not change direction, but rather maintains speed and direction of rotation for the next drop. Accurate and precise sorting was required for results to be adequate.

The final test was the Clockwise/Counterclockwise spin test, which accounted for a specific order in which the system spins in a full circle with a part being dropped at each 90 degrees of rotation. This test was used to optimize the algorithm so that the sorting system does not have to slow the stepper down at any point. This requires the conveyor belt and stepper motor to operate in precise unison to prevent mistakes.

It should be noted that each of these tests also provided debugging opportunities for unrecognized errors and edge cases. These tests do not come close to covering the variety of patterns of which the system would experience, however they are easily distinguishable patterns that can be used to trace errors within the software with relative ease. All testing patterns did not only provide results to optimize the sorting process at the stepper, but errors that occurred with other sensors such as the RL sensor and corresponding ADC conversion readings could also be determined.

The testing of the final product was completed using a preset pattern of 48 pieces. The end goal being each piece is sorted correctly and in the fastest time. Testing with this many pieces and at a high rate of loading could produce errors in a system that would go unnoticed in the previously mentioned tests. These errors were primarily a result of reduced ADC readings reducing the accuracy of differentiating parts. This test was ultimately the test used to define the system's ability to meet the requirements of the project.

3 RESULTS

3.1 TECHNICAL DESCRIPTION

The conveyor belt sorting system was constructed around a predesigned conveyor belt apparatus and hardware. The microcontroller used for this project was built around an ATmel 2560 microchip.

The conveyor belt of the sorting system used a Banebots 12V DC motor and was controlled by a Pololu High-Current Motor Driver coupled with an external power source [1, 2]. The stepper motor was a Soyo 6V Unipolar Stepper coupled with a L298 motor driver and external power supply [3, 4].

A total of four sensors were used on the sorting apparatus. There were two optical sensors (OR, EX), one reflective sensor (RL), and one hall effect sensor (HE). The optical sensor OR was used to detect incoming pieces into the RL sensor, to trigger the beginning or end of readings. The OR sensor was an Omron E2R-A01 [5]. The EX detected pieces at the end of the conveyor when they

were to exit and be sorted. This sensor was a OPB819Z Slotted Optical Switch [6]. In order to differentiate and classify pieces the RL sensor was used to measure the reflectance of the piece. For the RL sensor a Optek OPB740W Reflective Object Sensor was used in conjunction with an NPN Silicon Phototransistor laser [7, 8]. Finally, the HE sensor was used for initial calibration of the stepper motor, a Microswitches SS400 series sensor was used for this sensor [9].

3.2 System Performance Specifications

The following parameters in Table 1 were set for the final stage of testing and official timing tests. As noted in the table it was found that the parameters for fast operation did not sort the pieces appropriately and errors were significant and consistent. A reduction in these parameters to the safe setting produced results that were more desirable, at the expense of time. It's projected that the optimal parameter values for the given system are somewhere in the middle of the two modes.

Parameters	Safe Parameters	Fast Parameters	Projected Best Parameters
Conveyor DC Motor			
Conveyor Speed (%)	58%	63%	61%
Stepper Motor			
Jerk Steps	4	4	4
Minimum Delay	0x02F0	0x02F0	0x02F0
Maximum Delay	0x0A00	0x0A00	0x0A00
Maximum Acceleration	0x0090	0x00F0	0x00F0
Other			
Drop Region	20	14	16
Test Results	23 sec/0 errors	DNF	19-21 sec (estimated)

Table 1 - Test Parameters and Results

Although the separation in values between the conveyor speed is minimal for safe and fast parameters, slight changes were found to significantly impact the system's ability to properly sort the pieces. As can be noted by the DNF for the Fast test parameters. The minimum delay and jerk steps did not need to be changed between steps. The stepper motor's acceleration profile was kept conservative to avoid the motor jamming at higher loading further along in the test. A small difference between the stepper profiles was the maximum acceleration parameter, which was slightly higher for the Fast test. Since the stepper motor did not appear to cause problems during the test, a higher value for this parameter is acceptable.

Finally, the drop region was slightly lower between the Fast and Slow tests, simply due to the increased speed of the conveyor and stepper motor. This meant that pieces needed to be dropped more precisely. Testing results would suggest that the best value for the drop region is somewhere in between the two tests, since the conveyor belt will be moving slightly faster.

3.3 SYSTEM ALGORITHM

3.3.1 Block and Data Diagrams

The following block diagram in Figure 1 shows the functionality and data flow of the software and hardware modules of the system implementation on the MCU. This diagram shows the system in normal sorting operation. The Pause and Ramp Down buttons override the control flow. However, the Pause button reverses the system to its previous state after when it calls for resume and the Ramp Down button completely shuts down the system after a brief period. Therefore, this diagram summarizes the operation of the designed system in near entirety.

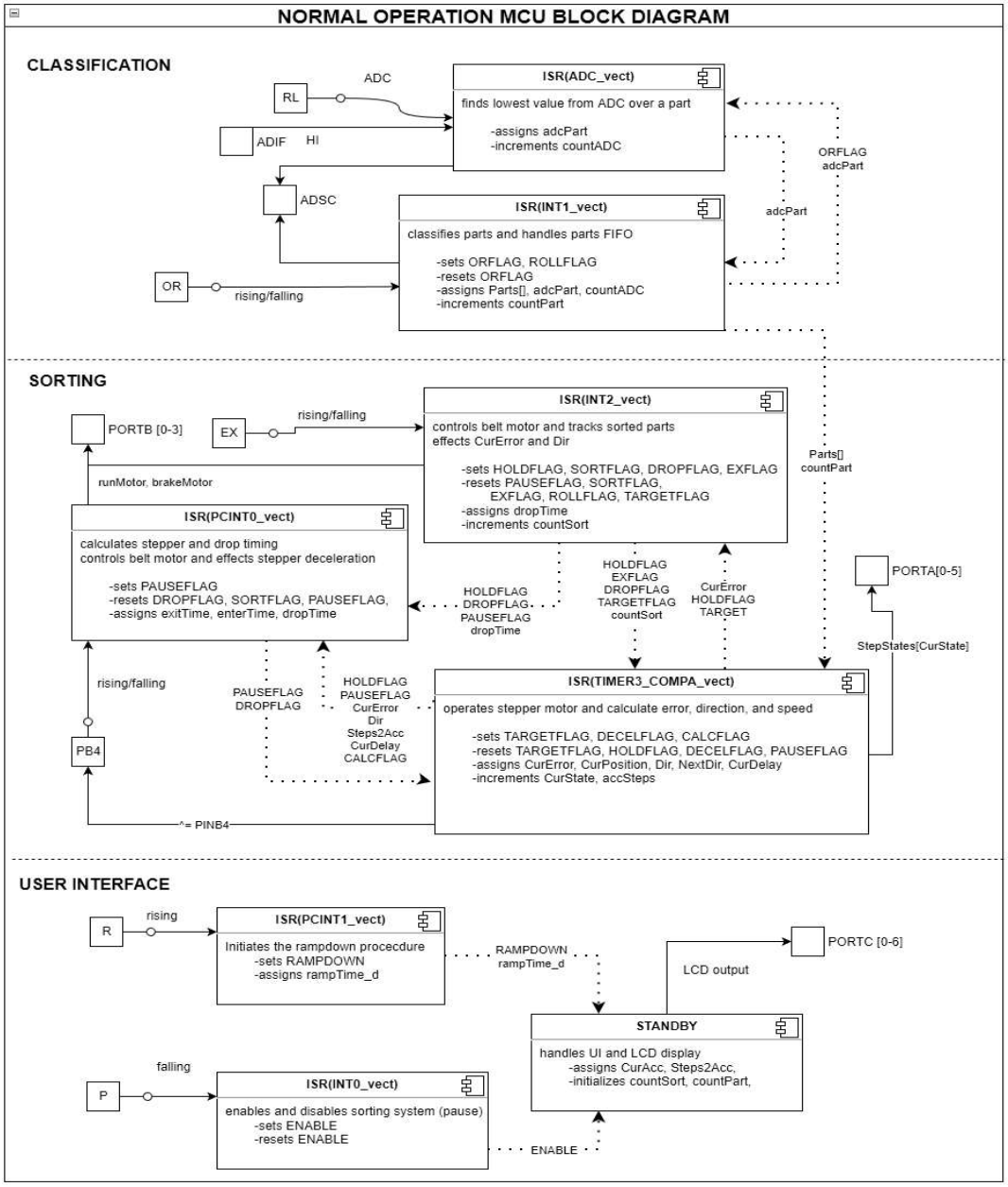


Figure 1 - MCU Block Diagram Normal Operation

3.3.2 Circuit Diagram

The image below illustrates the circuit used for this project. An image of the microcontroller and wiring can be found in Appendix C – Microcontroller Wiring.

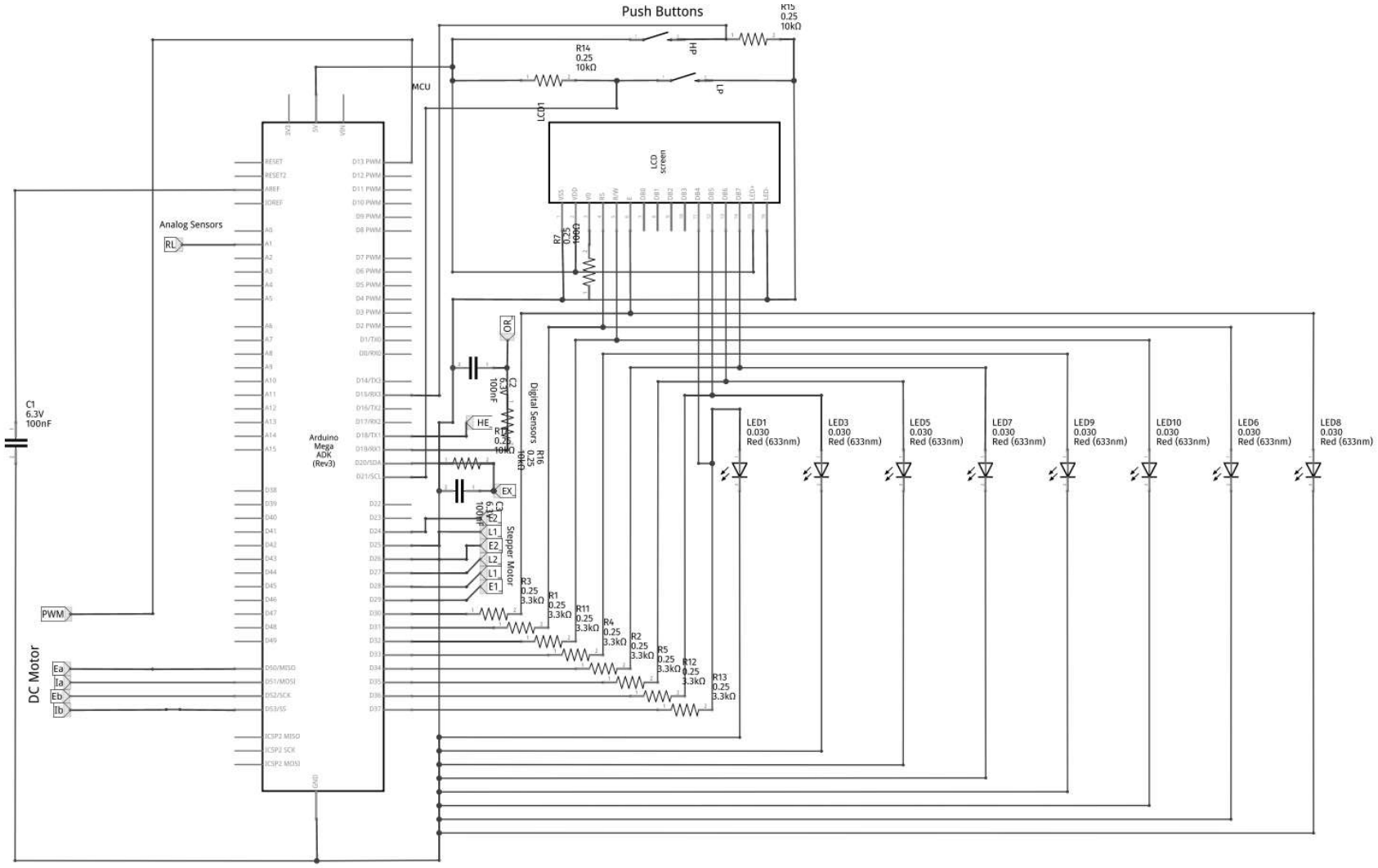


Figure 2 - Circuit Diagram

3.3.3 Flow Charts

The following flow chart in Figure 3shows the operation of functions and ISR implemented in the design of the sorting system in the main function. Specific flow charts relating to distinct functions such as motor control can be found in the B-2 Flow Charts.

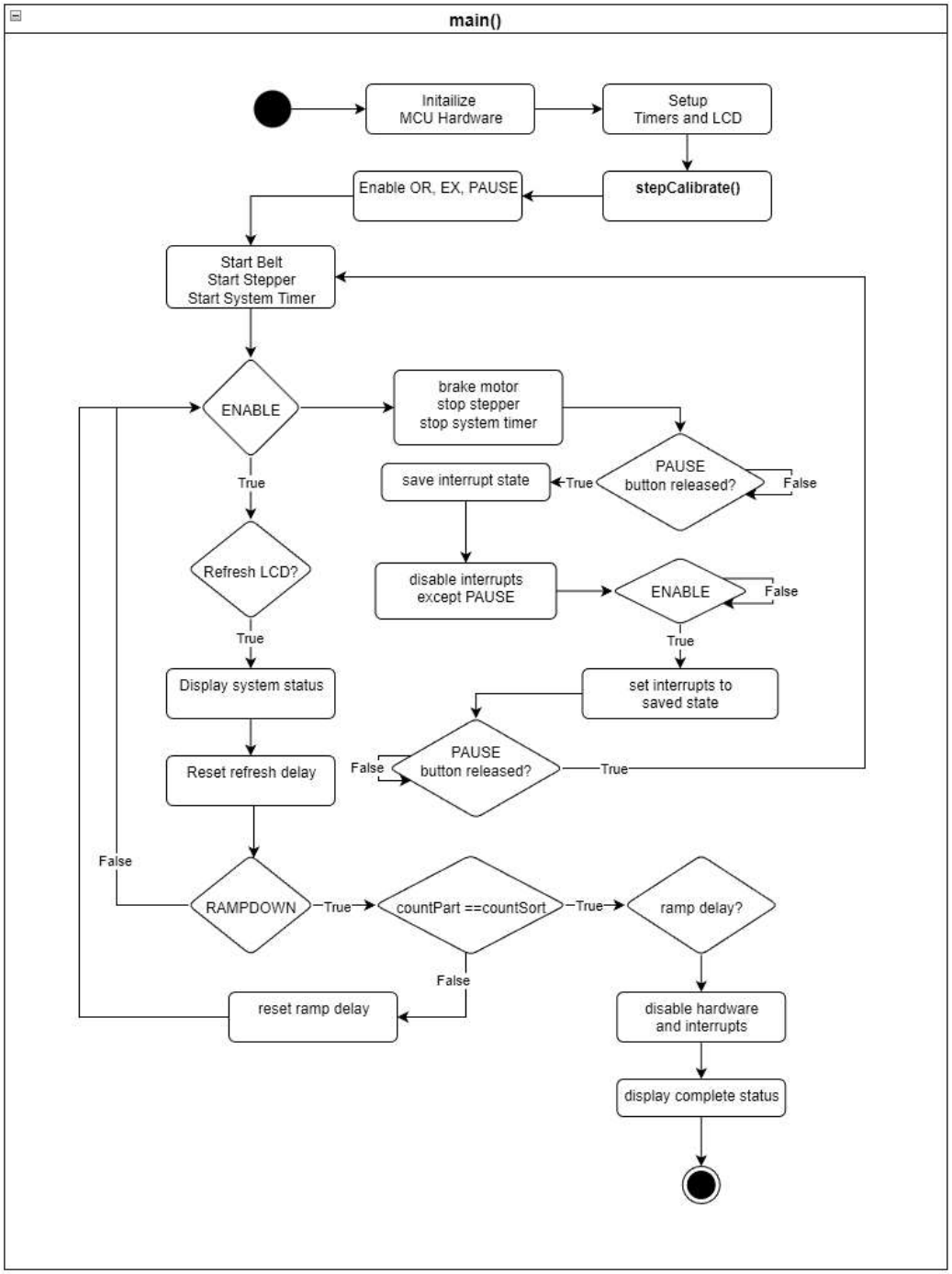


Figure 3 - main.c program flow chart

3.3.4 State Diagram

Supplementary to the flow diagrams, the state diagrams illustrate how the system reacts to input depending on the state it is in. For this system there are a total of four states: Classification, Sorting, User Interface, and Stepper. The states related to the general process (Sorting and Classifying) are given in Figure 4 and Figure 5 below. For state diagrams of the User Interface and Stepper states, see B-1 State Charts.

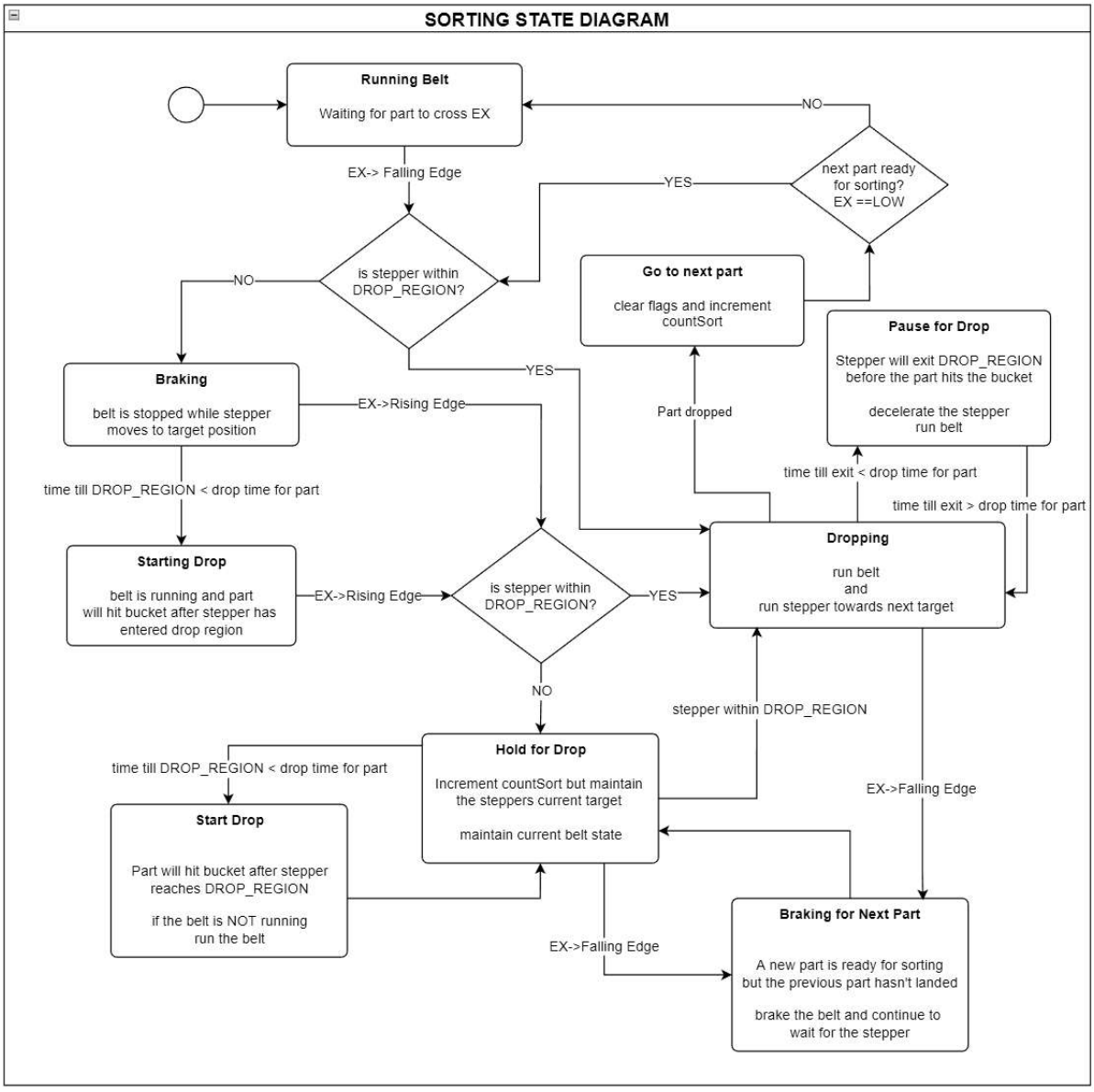


Figure 4 - Sorting State Diagram

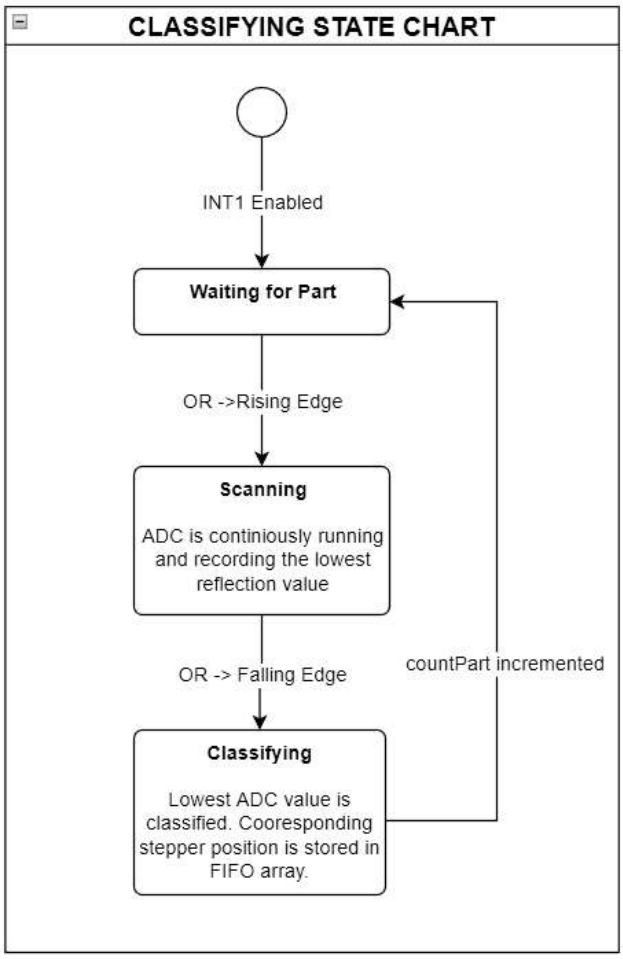


Figure 5 - Classify State Chart

3.4 TESTING AND CALIBRATION

For testing and calibration of the system a number of different tests were required to optimize their values for a given apparatus. The parameters to be discussed were often dependent upon each other, so sorting tests had to be conducted throughout to ensure proper functionality. The following calibration tests were completed prior to testing:

- ADC Reading Ranges
- Conveyor Speed and Exit Sensor Position
- Sorting Drop Regions and Timing
- Stepper Motor Acceleration Profile

3.4.1 ADC Reading

The calibration of the ADC Readings required ranges to be set for digital reflective values that would be expected to be returned for different pieces. To complete this calibration the digital value from the reading was displayed on the LCD when the conveyor was operating at normal speeds. Multiple pieces of the same type were placed on the conveyor and repeatedly passed through the

reflective sensor. Based calibrated value for that type of piece would then be changed to match that of the lowest displayed reading.

3.4.2 Conveyor Speed and Exit Sensor Position

The conveyor speed and the exit sensor position significantly impacted the sorting system. Conveyor belt speeds that were too high would cause parts to slip at the end of the conveyor belt, causing significant errors that could carry over to the sorting of the remaining pieces. If the conveyor was operating too slow, there could be significant loss of time. The exit sensor position also required tuning as it impacted the final sorting. If the exit sensor was too far forward, parts could slip off or rock back and forth, causing the interrupt to trigger multiple times. If the exit sensor was too far back, pieces could get stuck at the end and not be properly sorted.

To properly calibrate the speed and the exit sensor position, pieces were run through in the patterns discussed in the Methods section of the report. The speed could be easily set, based on iterative tests until slipping was no longer observed. Once the speed was set, the tests would be run further until all pieces were sorted accurately and no false readings or stuck pieces were observed.

3.4.3 Drop Regions and Timing

The drop regions and the timing of the sorting system was difficult to accurately calibrate so that pieces were dropping at the optimal time. These parameters were dependent upon the stepper and conveyor belt speeds, so inconsistent performance from different apparatus made it challenging to determine a calibration range. These parameters dictate when the pieces are dropped from the conveyor along with how much time is required to be spent in the given drop region.

The calibration of these parameters was completed in a similar way to that of the conveyor speed and exit sensor positions. The tests outlined in the method section were run through the system until results were found to be as expected. At this point in the calibration the stepper would be running at full speed, therefore it was required to film these tests in slow motion. Upon completion of this calibration, box set testing could then be completed to further validate and troubleshoot the system.

3.4.4 Stepper Motor Acceleration Profile

The stepper motor's acceleration profile was defined by a number of parameters. Each of which was often required to be changed if a different apparatus was used. For the calibration of the stepper, no pieces were passed through to test, as was completed in previous calibration stages. For these stage two batteries were placed on the top of the sorting plate. A number of test angles were then sent to the stepper motor to test the acceleration profile. If, at any point, the stepper lost steps or became jammed the parameters were reduced. This was completed until results were found to be consistent.

4 LIMITATIONS

Throughout testing there were several clear hardware limitations that are believed to have drastically reduced the final capabilities of the sorting system.

The limitation of the steppers acceleration directly dictated the speed in which the system could sort pieces from the conveyor. Quick changes in direction, with significant weight applied to the top of the sorting dish, caused the stepper to often lose steps as a result of the momentum causing excessive rotation in a direction. This also created troubleshooting problems as errors caused by missed steps were not always clear.

When running the conveyor at high speeds in order to reduce excess time wasted between the first and second optical sensors, slipping was often found to occur as previously discussed. This not only resulted in parts leaving the conveyor belt before they could be sorted appropriately, but parts would also begin to collect behind parts with higher frictional force, such as steel. By collecting close together, incorrect differentiation of parts caused significant errors. Alterations to the belt itself to reduce slipping, may allow for the system to operate slightly faster, assuming that an algorithm is in place that can keep up.

5 CONCLUSION

The final system was optimized to a point where final results were acceptable and met the initial specifications of the design. It was ultimately difficult to further optimize the system without hardware changes. While operating with safe parameters the system can complete the sorting of 48 pieces in 23 seconds, which was a more than acceptable result.

6 RECOMMENDATIONS

As stated previously in the Limitations section of this report hardware prevented major improvement beyond what the final system was able to achieve. It is recommended that if further improvements of the system are required that the conveyor belt be swapped out for a belt with more friction to prevent slipping of the pieces. It is also recommended that the stepper motor be swapped out for a higher quality motor, if one can be found.

For the software portion of the system, it is recommended that an algorithm be implemented to reset or extend the size of the array used for storing the identity of the pieces. The use of the array instead of the linked list was thought to increase speed and reduce issues of memory for the project. Since the final test only used a known number of pieces (48), this could easily be accounted for. If, however, the number of pieces were to exceed the size of the pre-allocated array errors would occur. It is believed that there are simple ways to account for this issue, however if none are to be found then conversion to a linked list system is a straightforward alternative.

7 REFERENCES

[1] Robotshop, *Banebots 12V 263 RPM 2527oz-in Planetary Gearmotor w/RS-540 Motor Specifications,* Robot Shop.

[2] Pololu, *Pololu High-Current Motor Driver Carrier Pinouts and Dimensions,* Pololu.

[3] Robot Shop, *Soyo 6V 0.8A 36oz-in Unipolar Stepper Motor Specfications,* Robot Shop.

[4] Solarbotics, *The Compact L298 Motor Driver,* 2007.

[5] OMRON, *E2R-A01 Compact Low Profile Inductive Proximity Sensor,* OMRON.

[6] OPTEK Technology, *Slotted Optical Switch.*

[7] Osram - Optosemiconductors, *Silicon NPN Phototransistor - SFH 310,* Osram, 2007.

[8] OPTEK, *Reflective Object Sensors Types OPB740W,* OPTEK, 1996.

[9] Honeywell - Microswitch, *SS400 Seris Charts.*

APPENDIX B – ALGORITHM SPECIFICATIONS

B-1 STATE CHARTS

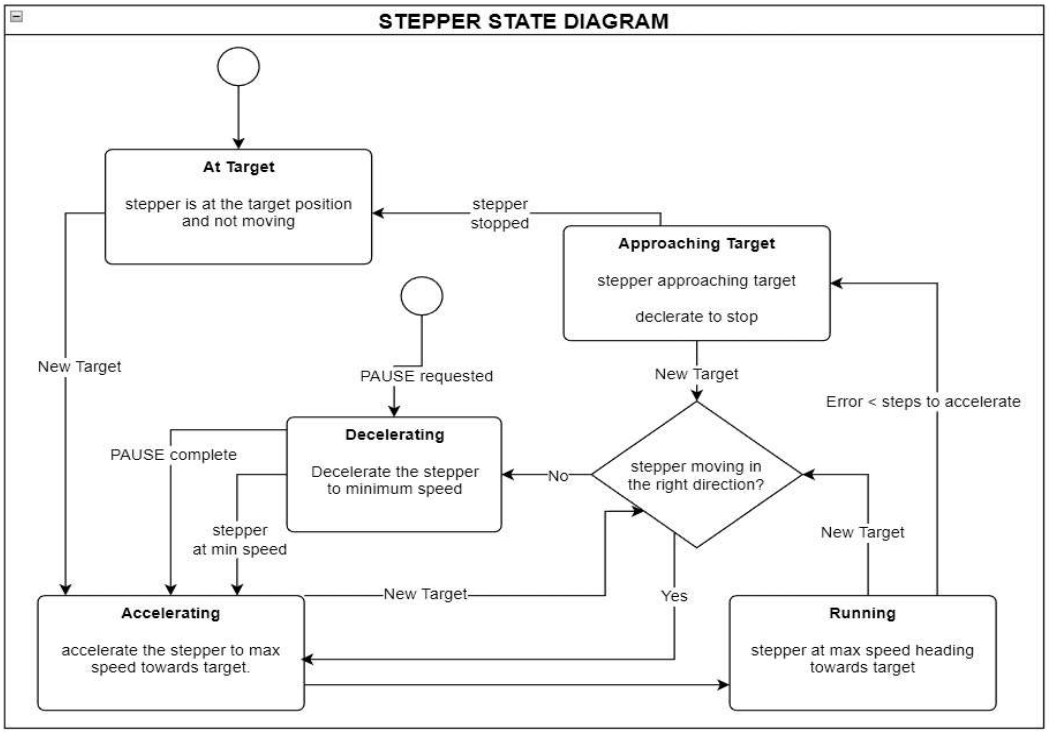


Figure 6 - Stepper State Diagram

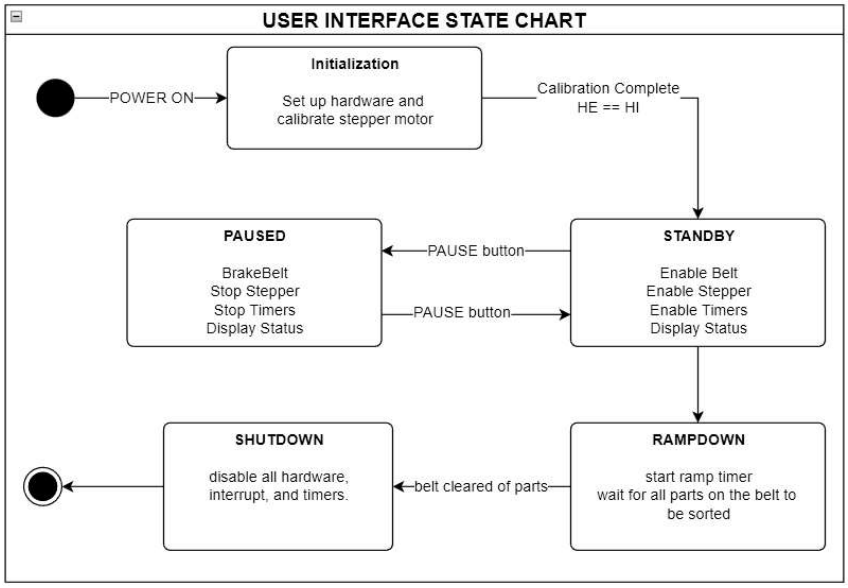
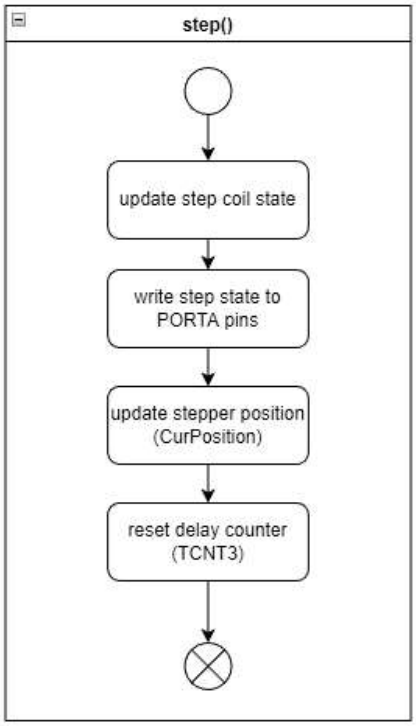
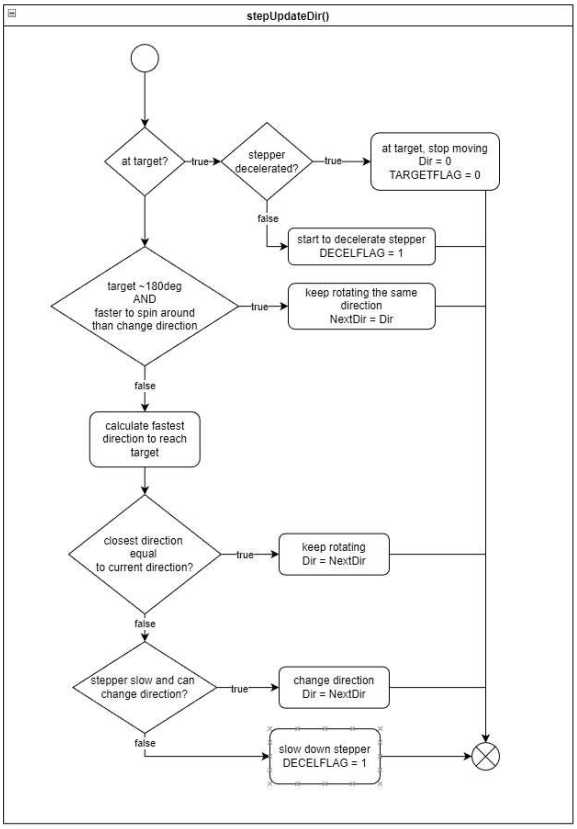
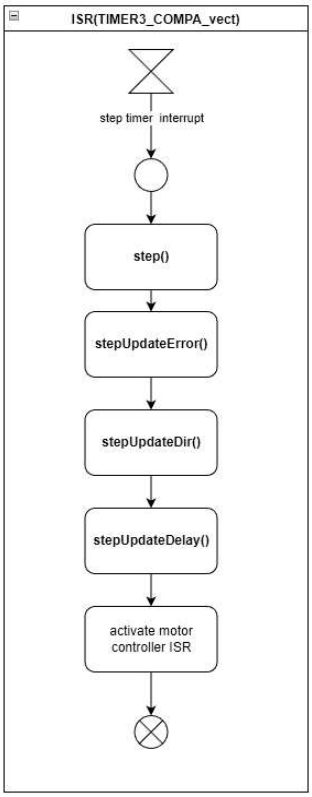


Figure 7 - User Interface State Diagram

B-2 FLOW CHARTS







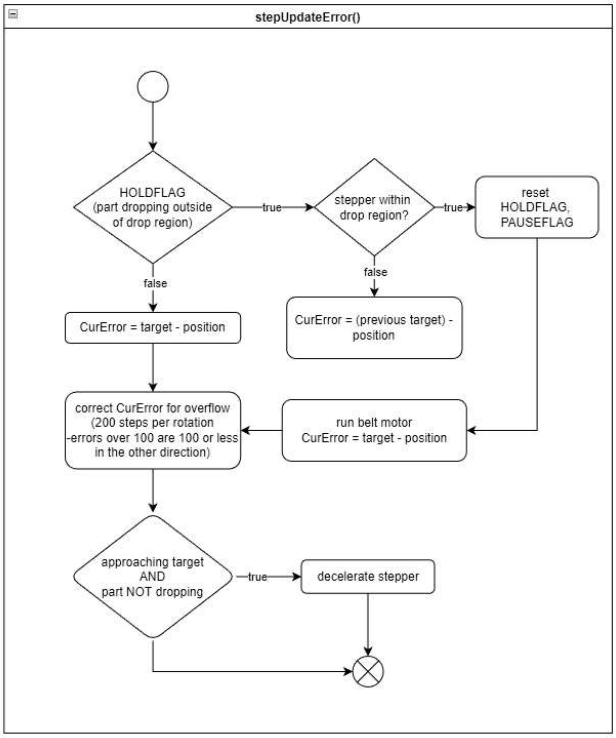
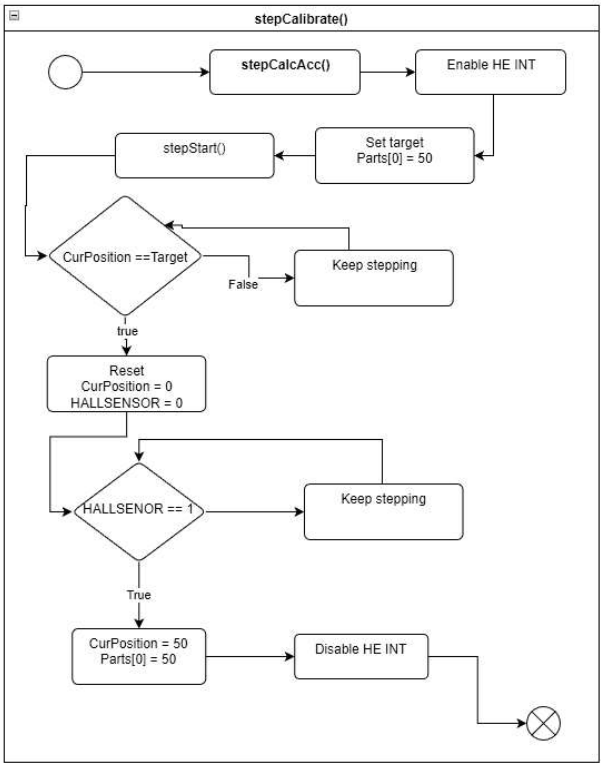
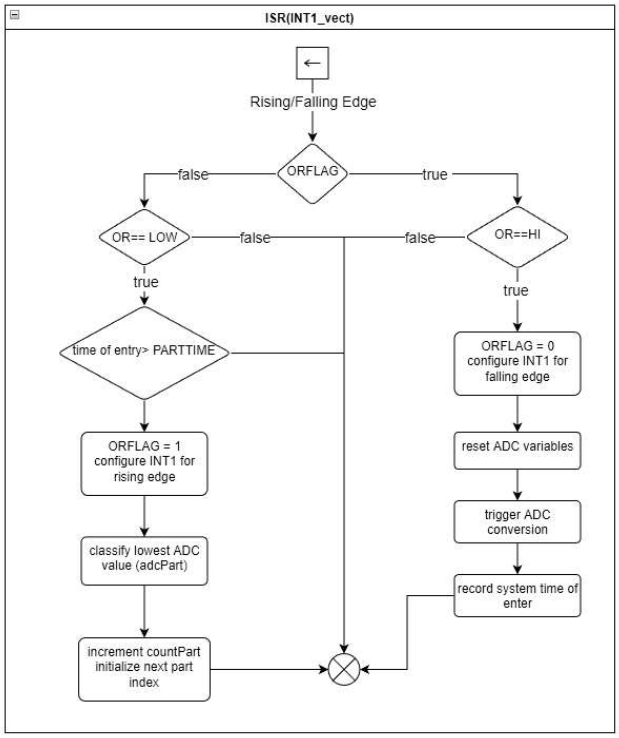
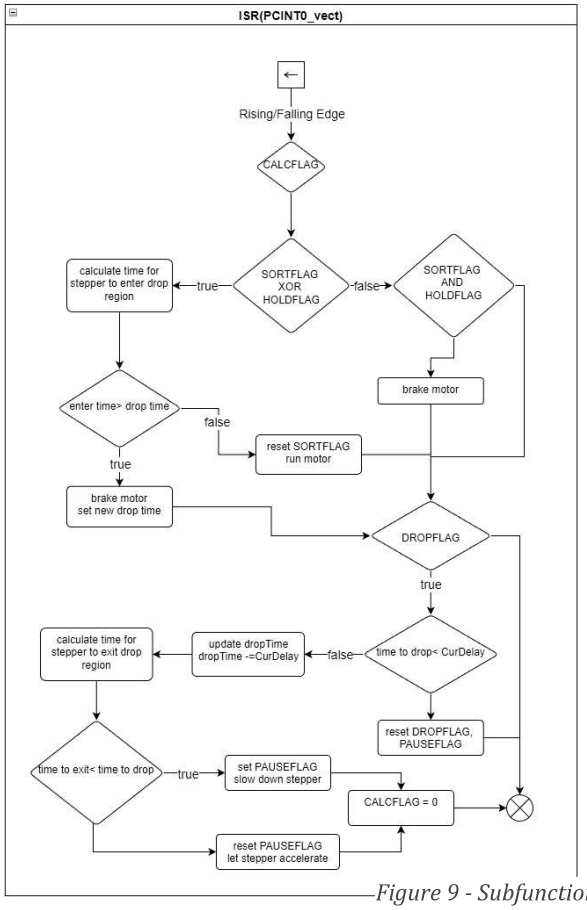


Figure 8 - Sub Function Flow Charts Page 1







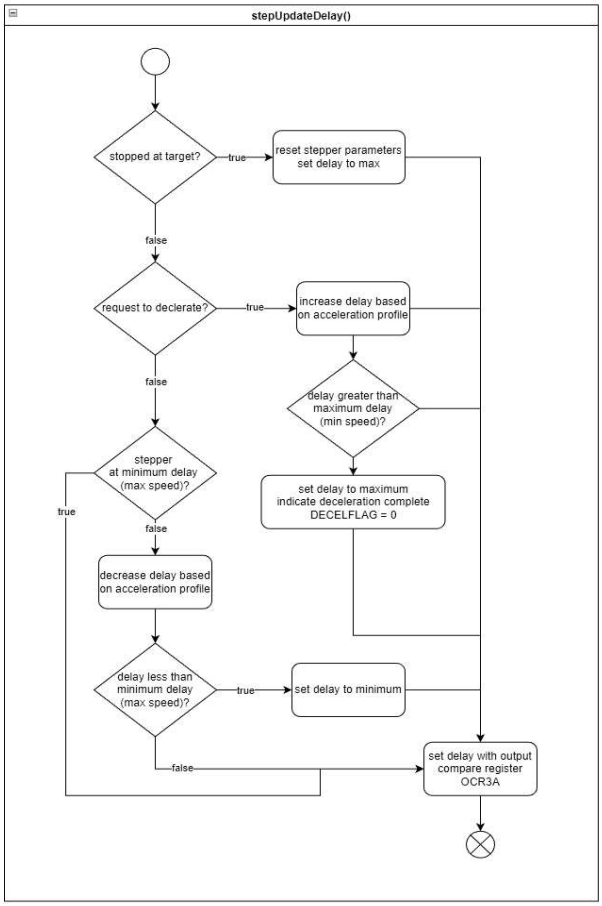
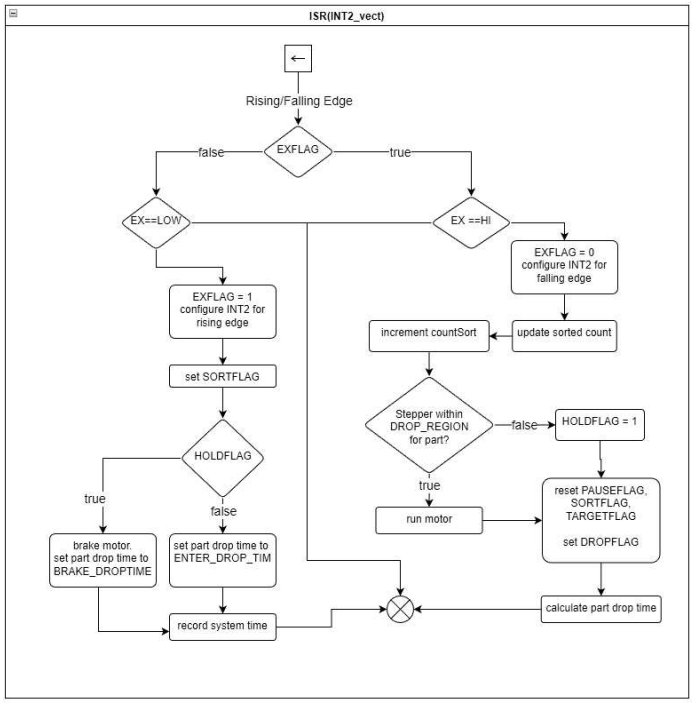
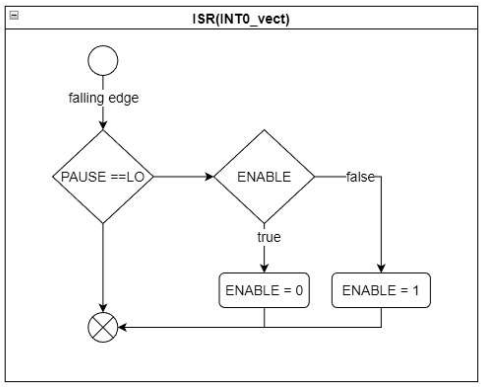
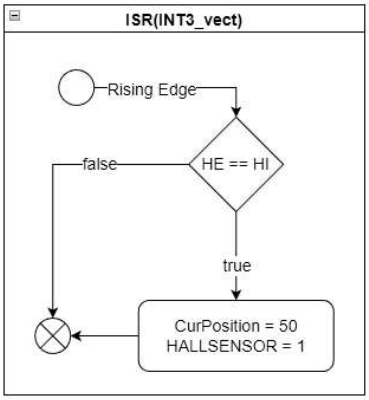
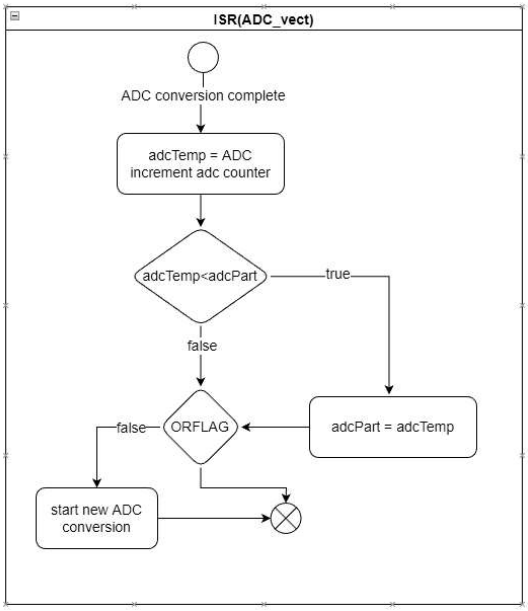


Figure 9 - Subfunction Flow Charts Page 2









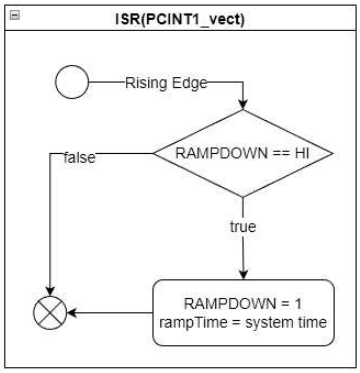


Figure 10 - Subfunction Flow Charts Page 3

APPENDIX C – CODE

C-1 MAIN.C

```
/*
 * main.c
 *
 * Created: 2022-12-09
 * Author: Matthew Ebert V00884117; Scott Griffioen V00884133
 * For: MECH 458, University of Victoria
 *
 * Dependencies: main.h, control.c, stepper.c
 *
 * BOARD: ATMEGA 2560
 *
 * Description: This program operates the sorting machine used in
        the course MECH 458 at the University of Victoria. The task of
        the machine is to identify cylindrical parts of different
        materials and sort them into labeled bins. A full description
        can be found in course documentation

        This program is interrupt based. All functional code can be found in the ISRs
        at the bottom of this file. The main function is used for initialization
        and the user interface. Since all code related to sorting is contained within
        ISRs, it will receive priority over any code in main. Consequently, the
        STANDBY section within main fill up 'idle' CPU time and should not effect the
        functionality or performance of the ISR processes.

        The primary functionality of this program is contained within the following ISRs

        ISR(INT1_vect): Handles functionality relating to OR and RL sensors
```

```
        ISR(INT2_vect): Handles functionality relating to EX sensor and Belt motion
        ISR(TIMER3_COMPA_vect): Controls stepper
        ISR(PCINT0_vect): Controls belt motor
*
*/


#include "main.h"
//FLAGS
volatile char HALLSENSOR=0; //Set when HE sensor is active
volatile char ENABLE=1; //reset/set with PAUSE button push
volatile char RAMPDOWN=0;// Set with RAMPDOWN button push
volatile char EXFLAG=0;//Set when part is within EX sensor
volatile char ORFLAG = 1;//Reset when part is within OR sensor
volatile char MOTORFLAG = 0;//Set when the motor is running
volatile char DECELFLAG = 0;//Set when stepper must decelerate
volatile char ROLLFLAG = 0;//Indicates when roll over for Parts array is in progress
/*Set when a part leave EX before the
stepper has reached the corresponding drop zone*/
volatile char HOLDFLAG = 0;

/*Set when stepper is within Steps2Acc of the target position
 and the part is NOT already dropping into the bin */
volatile char TARGETFLAG = 0;

/*Set when the stepper need to slow down to allow a part to fall into the bin */
volatile char PAUSEFLAG = 0;

//Set when the belt motor speed need to be updated
volatile char CALCFLAG = 0;
```

```
//Set when a piece is off the belt and falling towards the bin
volatile char DROPFLAG = 0;

//Set when a part enters EX and need to be sorted
volatile char SORTFLAG = 0;



//GLOBALS




/*
This array records all the parts detected by the system.
The parts are stored as a stepper position. The array is
parsed by 2 counters which indicate how many parts have been
detected and how many have been sorted
*/
uint8_t Parts[PARTS_SIZE+1];// leave 1 extra space for rollover handling



/*The 2 counters below need to 'roll over' when they approach PARTS_SIZE
The size of the array limits how many parts are stored in the system at any time*/



/*
This counter indicates the number of parts scanned and classified
by the system. It is incremented every time a new part enters OR
*/
```

```
volatile uint8_t countPart =0;
/*
This counter indicates which part is currently being sorted.
It is incremented every time a part leaves EX.
*/
volatile uint8_t countSort =0;



//ADC variables
volatile uint16_t adcValue = 1023;
volatile uint16_t adcPart =1023;
volatile uint16_t adcTemp =1023;
volatile uint16_t adcDisp =1023;
volatile uint16_t countADC=0;



/*
Variables used to record timing of events based on the
system timer which updates runtTime_d every 1ms.
*/
volatile uint16_t runTime_d =0;
volatile uint16_t refreshTime =0;
volatile uint16_t rampTime_d =0;
volatile uint16_t ORTime_s=0;
volatile uint16_t EXTime_s=0;

//EXTERNALS
extern volatile uint8_t CurPosition;
extern volatile uint16_t dropTime;
```

```
extern volatile uint16_t exitTime;
extern volatile int16_t CurError;
extern volatile uint16_t enterTime;
extern volatile uint16_t dropTime;
extern volatile uint16_t CurDelay;
extern volatile uint16_t enterdropTime;
extern volatile int8_t Dir;



int main(int argc, char *argv[]){
//INITIALIZATION
        //Limit Clock to 8MHz
        CLKPR = 0x80;
        CLKPR = 0x01;



        cli();//Disable Interrupts

        //GPIO setup
        DDRA = 0xFF; //OUTPUT for stepper
        DDRB = 0xFF; //OUTPUT for motor
        DDRC = 0xFF; //OUTPUT for LCD
        DDRD = 0x00;//INPUT for EX, OR, HE, PAUSE
        DDRJ &= ~_BV(PINJ0); //INPUT

        //EXT INTERRUPTS
        EICRA |= _BV(ISC01);//PAUSE
        EICRA |= _BV(ISC11) |_BV(ISC10);//OR
        EICRA |= _BV(ISC21);//EX
```

```
EICRA |= _BV(ISC31) | _BV(ISC30);//HE

PCICR |= _BV(PCIE1);//Enable PCINT1
PCMSK1 |= _BV(PCINT9);//RAMPDOWN

PCICR |= _BV(PCIE0);//Enable PCINT 0
PCMSK0 |= _BV(PCINT4);//Motor Controller



ADC_Init();
mTimer_init();
stepTimer_init();
InitLCD(LS_BLINK|LS_ULINE);
LCDClear();
EIMSK |= 0x08;//Enable HE
sei();// Enable global interrupts
// Calculate the stepper acceleration profile and calibrate position
stepCalibrate();
EIMSK |= 0x07;//Enable OR, EX, and PAUSE
EIMSK &= ~(0x08);//Disable HE

//Initialize the belt motor PWM and pins
Motor_init();



//reset counters
countPart=0;
countSort = 0;
startMotor();//Start Belt motor
runTimerStart();//Start System Timer
```

```
//MAIN OPERATION

/*****************************************************************************/
/* MAIN OPERATION: The following code contained in the labels STANDBY, DISABLE, and SHUTDOWN
        handle the user interface which includes the LCD screen and 2 buttons.

                                        It DOES NOT perform sorting or control any other hardware.

The sorting algorithm is handle entirely by ISR's which can be found below main() in this file.
*/
/*****************************************************************************/


STANDBY://Display the system status and handle button push events

        while (1)
        {
                if(ENABLE)
                {//if the system is enabled
                        if((runTime_d-refreshTime)>REFRESH_PERIOD)
                        {//if LCD needs to be updated
                                dispStatus();//display system information
                                refreshTime = runTime_d;//control refresh rate of LCD
                        }
                }else
                {//else go to disabled state
                        goto DISABLE;
                }
```

```
            if(RAMPDOWN)
            {//if RAMPDOWN button is pushed
                  if(countSort != countPart)
                  {//if there are parts which need sorting
                        rampTime_d = runTime_d;//reset RAMPDOWN timer

                  }else if((runTime_d-rampTime_d)>RAMPDOWN_DELAY)
                  {//else if the belt has been cleared of parts
                        goto SHUTDOWN;//shutdown the system
                  }
            }

      }//while ENABLE



DISABLE:// when the ENABLE is reset by ISR(INT0_vect)
      brakeMotor();//stop the belt
      while((PIND & 0x01) == 0x00);//wait for PAUSE button to be released
      stepStop();//stop the stepper motor
      runTimerStop();//stop the run timer
      uint8_t INTState = EIMSK; //save current interrupt state

      //Disable all interrupts except PAUSE button
      EIMSK = 0x01;
      PCMSK1 &= ~_BV(PCINT9);
      PCMSK0 &= ~_BV(PCINT4);

      brakeMotor();//insure motor is stopped (for edge case)
```

```
        stepRes();//Reset the stepper acceleration
        dispPause();//Display pause information

        //Wait for PAUSE button to be pushed again
        while(!ENABLE);
        while((PIND & 0x01) == 0x00);//wait for PAUSE button release

        //return interrupts to previous state
        EIMSK = INTState;
        PCMSK1 |= _BV(PCINT9);
        PCMSK0 |= _BV(PCINT4);
        runTimerResume();//start system timer
        stepStart();//start stepper
        runMotor();//start motor

goto STANDBY;//return to STANDBY mode


SHUTDOWN://When RAMPDOWN has been pushed and no part is on belt
        cli();//disable all interrupts
        PORTB = 0x00;//disable belt motor
        PORTA = 0x00;//disable stepper motor
        dispComplete();//display complete information
        while(1)
        {
                //wait until hardware reset
        }
```

```
        return(0);
}
//*************MAIN***************//



//*************ISR***************//

/*************************************************************************/
/* DESCRIPTION: This ISR handles events with the OR sensor.
Initially, the interrupt is triggered on a rising edge which occurs when
a part enters OR. It is then configured for a falling edge to detect when
the part leaves. This cycle repeats.

This ISR controls detecting and classifying parts. It also starts the ADC.

Filters have been implemented to prevent errors due to noise in the system. One part makes use
the function debounce() to implmenent a software lowpass filter. Another part compares the time
since the part has entered against a minimum expected time to block potential double reads cause
rocking or sliding of the parts.

RUNTIME ~500cc
                                        */
/*************************************************************************/
ISR(INT1_vect)
{//OR has triggered falling or rising edge
        if(ORFLAG)
        {//if Part is entering OR
```

```
        if(debounce(1, 1, NOISECHECK))
        {//FILTER noise

                ORFLAG  = 0; //Part has entered OR

                //set to falling edge
                EIMSK &= ~_BV(INT1);
                EICRA &= ~_BV(ISC10);
                EIMSK |= _BV(INT1);

                //reset adc variables
                countADC = 0;
                adcPart = 1023;

                ADCSRA |=_BV(ADSC);//start first ADC conversion

                ORTime_s = runTime_d;//record time part entered
                EIFR |= _BV(INT1);//reset interrupt flag (for edge case)
        }//HI

}else//!ORFLAG
{//if Part is leaving OR


        if(debounce(1, 0, NOISECHECK) && ((runTime_d - ORTime_s) > PARTTIME))
        {//FILTER noise and double edge detection
                ORFLAG  = 1;//Part has cleared OR

                //Turn on rising edge
                EIMSK &= ~_BV(INT1);
```

```
                    EICRA |= _BV(ISC10);
                    EIMSK |= _BV(INT1);

                    adcDisp = adcPart;//set display ADC variable

                    //FILTER bad reads from ADC
                    if((adcPart<HI_Reflect) && countADC>50)
                    {//if a reflect value was recorded and the adc got more than minimum reads
                        Parts[countPart] = classify(adcPart);//classify the part and add to array
                        Parts[countPart+1] = Parts[countPart];//Initialize next array index
                        countPart +=1;//increment part counter
                        if(countPart==PARTS_SIZE)
                        {//roll over if at parts size
                            Parts[0] = Parts[countPart-1];
                            Parts[1] = Parts[0];
                            countPart = 1;
                            ROLLFLAG = 1;
                        }
                    }
                    EIFR |= _BV(INT1);//reset interrupt flag (for edge case)
                }//LO
            }//else
}//OR



/*****************************************************************/
/* DESCRIPTION: This ISR handles events with the EX sensor.
Initially, the interrupt is triggered on a falling edge which occurs when
a part enters EX. It is then configured for a rising edge to detect when
```

the part leaves. This cycle repeats.

This ISR set flags and variable to effectively sort parts as they
reach the end of the belt. It also starts and brakes the belt in 2 cases.

This ISR also increment the countSort variable every time a part leaves the sensor.

Filters have been implemented to prevent errors due to noise in the system. One part makes use the function debounce() to implmenent a software lowpass filter. Another part compares the time since the part has entered against a minimum expected time to block potential double reads cause rocking or sliding of the parts.

RUNTIME ~800cc

```
                                        */
/*************************************************************************/
]{

        if(!EXFLAG)
        {//Part is entering EX

                if(debounce(2, 0, NOISECHECK))
                {//FILTER noise
                                EXFLAG =1;
                                // Turn on rising edge
                                EIMSK &= ~_BV(INT2);
                                EICRA |= _BV(ISC20);
                                EIMSK |= _BV(INT2); //Enable Interrupt
                                EIFR |= _BV(INT2);

                                SORTFLAG = 1;//Part need sorting
```

```
                        if(HOLDFLAG)
                        {//if the previous part has not finished sorting
                                brakeMotor();//stop the belt
                                enterdropTime = BRAKE_DROP_TIME;//set the drop time
                        }else
                        {//else keep the belt moving
                                enterdropTime = ENTER_DROP_TIME;//set the drop time
                        }
                        EXTime_s = runTime_d;//record time part entered
        }//LO
}else
{//Part is leaving EX

        if(debounce(2,1, NOISECHECK) && ((runTime_d - EXTime_s)>SORTTIME))
        {//FILTER noise and double edge detection
                        EXFLAG = 0;
                        //Turn on falling edge
                        EIMSK &= ~_BV(INT2);
                        EICRA &= ~(_BV(ISC20));
                        EIMSK |= _BV(INT2);
                        EIFR |= _BV(INT2);

                        updateCount(Parts[countSort]);//Update the sorted count for display



                        if(countSort<countPart || ROLLFLAG)
                        {//if still parts to sort
                                countSort+=1;//go to next part
                                if(countSort==PARTS_SIZE)
```

```
                                        {//roll over if at parts size
                                                countSort = 1;
                                                ROLLFLAG = 0;
                                        }
                                        TARGETFLAG =0;//New target; reset flag
                                }

                                if(abs(CurError)>DROP_REGION)
                                {//if stepper hasn't reached the drop zone for previous part
                                        HOLDFLAG = 1;//set hold flag to keep moving to previous target
position
                                }else
                                {//else start the belt to drop the part
                                        runMotor();
                                }
                                //reset flag
        PAUSEFLAG=0;
                                SORTFLAG = 0;

                                DROPFLAG = 1;//part is now dropping into the bin

                                //record time for part to hit bucket.
                                //Correct for next time ISR(TIMER3_COMPA_vect) runs
                                dropTime = DROP_TIME - (OCR3A - TCNT3);

                        EXTime_s = runTime_d;//record time part exited
                }//HI
        }
}//EX
```

```
/*****************************************************************************/
/* DESCRIPTION: This is the primary ISR of the sorting system. This ISR runs
every time a step is required from the stepper motor. Consequently, the faster
the stepper is moving, the more this ISR runs.

The ISR controls writing to the stepper driver, calculating the variable CurError,
calculating the best direction to turn, and determining the next delay held in CurDelay.
CurDelay, controls the speed and acceleration of the stepper.

The ISR also triggers the ISR for motor control, ISR(PCINT0_vect).

RUNTIME ~400cc                                          */
/*****************************************************************************/
ISR(TIMER3_COMPA_vect){
//CONTROL STEPPER
        step();//step stepper and update the position
        stepUpdateError(); //calculate the new stepper position error (CurError)
        stepUpdateDir(); //update the stepper direction
        stepUpdateDelay(); //update the stepper speed
//CONTROL STEPPER

        //trigger motor controller.
        CALCFLAG = 1;
        PORTB ^= _BV(PINB4);
}//stepTimer
```

```
/*************************************************************************/
/* DESCRIPTION: This ISR handles the ADC.
after initially being triggered by OR, this function runs every time
an ADC conversion completes. When it runs it compares the current ADC value
to the lowest of the sequence and replaces it if it is lower. If the part being
scanned is still within OR it triggers another conversion. Otherwise, it exits
without restarting the ADC.                                          */
/*************************************************************************/
ISR(ADC_vect){

        //if ADC is lower than value
        adcTemp = ADCL;
        adcTemp+= (ADCH<<8);
        countADC+=1;

        if(adcTemp<adcPart){
                adcPart = adcTemp;// set value to ADC
        }

        if(!ORFLAG){
                ADCSRA |=_BV( ADSC);
        }
}//ADC



/*************************************************************************/
/* DESCRIPTION: This ISR is connected to the HE sensor. When triggered
it calibrates the stepper position to the known position of the HE sensors.
Its also sets the HALLSENSOR flag                                    */
```

```
/*********************************************************************/
ISR(INT3_vect){
        if(debounce(3, 1, NOISECHECK)){
                //stepStop();
                CurPosition = B_ID;
                HALLSENSOR= 1;
        }
}//HE


/*********************************************************************/
/* DESCRIPTION: This ISR is connected to the PAUSE button. When pressed
it is debounced and then toggle enable                              */
/*********************************************************************/
ISR(INT0_vect){
        if(debounce(0, 0, BOUNCECHECK)){
                if(ENABLE)
                {
                        ENABLE = 0;
                }else
                {
                        ENABLE = 1;
                }
        }
}//ISR Pause Button

/*********************************************************************/
/* DESCRIPTION: This ISR is connected to the RAMPDOWN button. It begins
then ramp down sequence by setting the RAMPDOWN flag.               */
/*********************************************************************/
```

```
ISR(PCINT1_vect)
{
	if(debouncePINJ(0, 1, BOUNCECHECK)){
		RAMPDOWN = 1;
		rampTime_d = runTime_d;
	}
}//ISR Ramp Button



/*****************************************************************************/
/* DESCRIPTION: This ISR is triggered by toggling the pin connected to
PCINT4 (PB4). This occurs every time ISR(TIMER3_COMPA_vect) runs.

This function controls the belt motor. It determines when to start or stop
a part drop based on the time it will take the stepper motor to reach a
drop zone and the time it will take the part to fall. Some of these timings are
calibrated outside of run time.

This function also can request the stepper motor to slow down should a part
require more time to drop into the zone.

NOTE: The reason this function is separate from ISR(TIMER3_COMPA_vect) is
to allow the ADC ISR to run in between these functions. This achieves higher
precision classification for each part by allowing more ADC reads.                */
/*****************************************************************************/
ISR(PCINT0_vect)
{
	if(CALCFLAG)
	{//if motor control is requested (NOISE filter)
```

```
if(SORTFLAG ^ HOLDFLAG)
{//If a piece needs sorting XOR a the stepper is holding target for a part

        //Calculate the time till stepper enters the drop zone
        if(CalcEnterTime())
        {//if stepper will not reach the drop zone before part falls
                brakeMotor();//brake motor to slow or stop part
                enterdropTime = BRAKE_DROP_TIME;//adjust new drop time
        }else
        {//else stepper will reach drop zone in time for the part to fall
                SORTFLAG = 0;//the part will be sorted correctly, reset flag
                runMotor();//start the motor
        }
}else if(SORTFLAG && HOLDFLAG)
{//if a new pieces needs to be sorted and a part is currently dropping
        brakeMotor();//stop the motor and wait for the part to finish dropping
}



if(DROPFLAG)
{//if a part has left EX and is dropping
        if(dropTime<CurDelay)
        {//if the part will hit the bucket before the next step
                //reset flags; the part is sorted or missed
                DROPFLAG = 0;
                PAUSEFLAG = 0;
        }else
        {//else
```

```
                        dropTime -=CurDelay;//update the drop time
                        //calculate the time until the stepper leaves the drop zone
                        if(CalcExitTime())
                        {//if the stepper will exit before the parts drop
                                PAUSEFLAG = 1;//slow down the stepper
                        }else
                        {
                                PAUSEFLAG = 0;//allow stepper to move as normal
                        }
                }
        }
        CALCFLAG = 0;//reset flag for next call

}
```

C-2 CONTROL.C

```
/*
 * control.c
 *
 * Created: 2022-12-09
 * Author: Matthew Ebert V00884117; Scott Griffioen V00884133
 * For: Sorting Machine, MECH 458, University of Victoria
 *
 * Dependencies: main.h
 *
 * BOARD: ATMEGA 2560
 *
 * Description: This file contains functions used in main.c
 *
 */

#include "main.h"



//GLOBALS
volatile uint8_t countB =0;//counts number of black parts sorted
volatile uint8_t countW =0;//counts number of white parts sorted
volatile uint8_t countS=0;//counts number of steel parts sorted
volatile uint8_t countA =0;//counts number of aluminum parts sorted

volatile uint8_t motorDecSpeed =MOTOR_SPEED;//motor speed while decelerating
volatile uint16_t exitTime =0;//time for stepper to exit drop zone
volatile uint16_t enterTime =0;//time for stepper to enter drop zone
volatile uint16_t dropTime = DROP_TIME;//time for part to hit bucket from leaving EX
volatile uint16_t enterdropTime = ENTER_DROP_TIME;//Time for part to hit bucket from entering EX
```

```
volatile uint16_t motorTime_d = 0;//current running time of motor



volatile uint8_t Steps2Exit = 0;//number of steps until stepper exits drop zone
volatile uint8_t Steps2MIN = 0;//number of steps until stepper is at max speed
volatile uint8_t Steps2Enter = 0;//number of steps until stepper enters drop zone

//EXTERNALS
extern volatile uint16_t runTime_d;
extern volatile char MOTORFLAG;
extern volatile uint8_t countPart;
extern volatile uint8_t countSort;
extern uint8_t Parts[PARTS_SIZE];
extern volatile uint16_t adcDisp;
extern volatile char PAUSEFLAG;
extern volatile char TARGETFLAG;
extern volatile char DECELFLAG;
extern volatile char HOLDFLAG;
extern volatile uint8_t Steps2Acc;
extern volatile uint16_t CurDelay;
extern volatile uint8_t accSteps;
extern volatile uint8_t CurPosition;
extern volatile int16_t CurError;
extern volatile int8_t Dir;
extern volatile char EXFLAG;



/*********************************************************************/
/* DESCRIPTION: Initializes the PWM and pins for belt motor
                                        */
```

```
/*************************************************************************/
void Motor_init(void){
        //Set control register to fast PWM mode
        //Clear OC0A on compare match, set at BOTTOM
        TCCR0A |= _BV(COM0A1) | _BV(WGM01)| _BV(WGM00);
        //prescale the timer by 8 (3.9kHz)
        TCCR0B |= _BV(CS01);//|_BV(CS00) ;
        //reset the output compare register flag
        TIFR0 |= _BV(OCF0A);
        //set the output compare register value
        OCR0A = 0;
        //Stop
        stopMotor();
}


/*************************************************************************/
/* DESCRIPTION: This function calculates the time until the stepper motor exits
the drop region defined by DROP_REGION. The function uses the current speed
and direction of the stepper to calculate the EXACT time it will take for
the stepper to reach an edge of the drop region.

The function returns a 1 if the stepper motor will exit the drop region before the
current part hits the bucket. It returns a 0 if the parts will hit before the
stepper exits.

The part drop time is estimated based on calibrated variables and updated every time
ISR(PCINT0_vect) runs if necessary. This time is stored in the variable
dropTime.
NOTE: dropTime is an approximation and of the actual part drop time which will
```

```
vary with every drop.We cannot know the exact time the part will take to drop
since we do not know the precise speed and position of the part past the EX sensor.
                                        */
/*************************************************************************/
uint8_t CalcExitTime(void)
{
        if(HOLDFLAG)
        {//if part is past EX
                return 0;
        }

        //Calculate the number of steps to edge of drop zone
        Steps2Exit = DROP_REGION - abs(CurPosition - Parts[countSort-1]);
        //Calculate number of steps until maximum acceleration
        Steps2MIN = Steps2Acc-accSteps;

        if(((CurError*Dir)>0) || (CurDelay>=MAXDELAY))
        {//If the stepper is turning in the right direction or able to switch directions

                if(Steps2Exit<Steps2Acc)
                {// if stepper will exit before reaching max speed
                        exitTime = (CurDelay - MINDELAY)/2 * Steps2Exit;
                }else
                {//if stepper will reach max speed before exiting drop region
                        exitTime = (CurDelay - MINDELAY)/2 * Steps2MIN + (Steps2Exit -
Steps2Acc)*MINDELAY;
                }

        }else
        {//if stepper is turning in the wrong direction
```

```
                exitTime = (MAXDELAY - CurDelay)/2 * (Steps2Acc)//time to decelerate
                +(MAXDELAY-MINDELAY)/2 * Steps2Acc//time to accelerate
                + (Steps2Exit -(Steps2Acc-Steps2MIN))*MINDELAY;//time to reach edge at max speed after
acceleration
        }

        if(exitTime<dropTime)
        {//if stepper will exit region before current part drops
                return 1;
        }else
        {//else
                return 0;
        }

}



/*****************************************************************************/
/* DESCRIPTION: This function calculates the time until the stepper motor enters
the drop region defined by DROP_REGION. The function uses the current speed
and direction of the stepper to calculate the EXACT time it will take for
the stepper to reach an edge of the drop region.

The function returns a 1 if the stepper motor will enter the drop region after the
current part hits the bucket. It returns a 0 if the parts will hit after the
stepper enters.

The enterdropTime is an approximation of the time it will take the part to hit the bucket
```

```
when it enters or is in EX. This variable is set to 1 of two values which are calibrated
outside of regular run time. enterdropTime is set to ENTER_DROP_TIME if the belt is running
when its drop sequence begins or to BRAKE_DROP_TIME if the belt is stopped in the same situation.
                                        */
/*************************************************************************/
uint8_t CalcEnterTime(void)
{

        if(abs(CurError)<DROP_REGION)
        {//if stepper is within drop region
                return 0;
        }

        //calculate steps to enter drop region
        Steps2Enter =  abs(CurError) - DROP_REGION;
        //calculate steps to reach max speed
        Steps2MIN = Steps2Acc-accSteps;

        if(Steps2Enter>40)
        {//if stepper is far from the drop region
        //this is needed else enterTime may exceed range of 16 bit number
                return 1;
        }

        if(((CurError*Dir)>0) || (CurDelay>=MAXDELAY))
        {//if stepper is moving in the right direction or can change directions

                if(Steps2MIN > Steps2Enter)
                {//if stepper will enter drop region before reaching max speed
                        enterTime = (CurDelay - MINDELAY)/2 * Steps2MIN;
```

```
                }else
                {//if stepper will reach max speed before entering region
                        //Note: Calculation is broken up to prevent truncation and overflow
                        enterTime =   (Steps2Enter- Steps2MIN); //time to reach edge after acceleration
                        enterTime = enterTime*MINDELAY;
                        enterTime  += (CurDelay - MINDELAY)/2 * Steps2MIN;//time to accelerate

                }

        }else
        {

                enterTime = (Steps2Enter- Steps2MIN);
                enterTime = enterTime*MINDELAY;
                enterTime += (MAXDELAY - CurDelay)/2 * Steps2Acc;
                enterTime += (MAXDELAY-MINDELAY)/2 * Steps2Acc;

        }


        if(enterTime>enterdropTime)
        {//if part will drop before stepper reaches region
                return 1;
        }else
        {//if stepper will reach region before part drops
                return 0;
        }
}

/*********************************************************************/
```

```
/* DESCRIPTION: Starts the motor belt if required. Sets up a timer which will
decelerate the motor when it reaches the MOTOR_START_DELAY value.
Reset the motor timer and counter variables. This function should only be called
on startup or after an extended pause. Use runMotor() in other cases.                    */
/*************************************************************************/
uint8_t startMotor(){
        PORTB &= 0x80;
        PORTB |= 0b00001011;
        TCNT0 = 0;
        OCR0A = MOTOR_START_SPEED;
        //TCCR0B |= _BV(CS01);
        if(!MOTORFLAG)
        {
                MOTORFLAG = 1;
                motorTimerStart();//start the motor timer to handle deceleration
                OCR5A =MOTOR_START_DELAY ;//Initial delay of the motor before deceleration
                motorTime_d = runTime_d;
        }
        TCNT5 = 0x0000;//restart max motor run time
        return MOTORFLAG;
}


/*************************************************************************/
/* DESCRIPTION: Runs belt motor is required. Starts motor timer if motor is off.
Sets MOTORFLAG to indicate belt is running.                              */
/*************************************************************************/
uint8_t runMotor(){
        //set to run forward, leave PWM signal as is.
        PORTB &= 0x80;
```

```
	PORTB |= 0b00001011;
	TCNT0 = 0;//reset PWM counter
	OCR0A = MOTOR_SPEED;//set initial motor speed

	if(!MOTORFLAG)
	{
		//start motor time which handles deceleration
		motorTimerStart();
		MOTORFLAG = 1;
		motorTime_d = runTime_d;//save motor run time
	}
	return MOTORFLAG;

}


/*************************************************************************/
/* DESCRIPTION: brakes belt motor. Resets MOTORFLAG to indicate motor
stopped.                                        */
/*************************************************************************/
uint8_t brakeMotor(){
	PORTB &= 0x80;
	PORTB |= 0b00001111;
	MOTORFLAG = 0;
	return MOTORFLAG;
}


/*************************************************************************/
/* DESCRIPTION: stops belt motor. Resets MOTORFLAG to indicate motor
```

```
stopped.                                    */
/*****************************************************************************/
uint8_t stopMotor(){
        PORTB = 0x00;
        MOTORFLAG = 0;
        return MOTORFLAG;
}

/*****************************************************************************/
/* DESCRIPTION: set up the motor timer which handles motor deceleration.                    */
/*****************************************************************************/
void motorTimerStart(void){
        TCCR5B |= _BV(WGM52); // Configure counter for CTC mode;
        OCR5A = MOTOR_TIMER; //1s timer
        TCNT5 = 0x0000; //Counter value register; Reset to 0
        TIMSK5 |= _BV(OCIE5A);  //Enable Interrupt
        TCCR5B |= _BV(CS52)| _BV(CS50);//Set prescaler to 1024
        TIFR5 |= _BV(OCF5A);//reset interrupt flag
        motorDecSpeed = MOTOR_SPEED;
}//mTimer_init

/*****************************************************************************/
/* DESCRIPTION: Disables the motor timer                                     */
/*****************************************************************************/
void motorTimerStop(void){
        TCCR5B &= ~_BV(CS52)& ~_BV(CS50);
}



/*****************************************************************************/
```

```
/* DESCRIPTION: Decelerates the motor based on the control parameters.
When the motor timer is active, the motor will run for a given period. This ISR
will then run and slow down the motor by an increment. This will repeat until the
motor reaches a certain speed. This gives a motor response with an initial fast pulse before
rapidly decelerating to a constant speed.                                    */
/*****************************************************************************/
ISR(TIMER5_COMPA_vect){
        //if motor needs to slow down
        motorDecSpeed -= MOTOR_DEC;
        OCR5A = MOTOR_DEC_RATE;
        if(motorDecSpeed < MOTOR_SLOW_SPEED)
        {//if less than slowest motor speed
                motorDecSpeed = MOTOR_SLOW_SPEED; //set as lowest speed
                MOTORFLAG = 0;
                motorTimerStop();//disable timer
        }
        //Change motor speed by adjusting the PWM
        TCNT0 = 0;
        OCR0A = motorDecSpeed;
}//ISR

/*****************************************************************************/
/* DESCRIPTION: Initializes the ADC                                          */
/*****************************************************************************/
void ADC_Init(void){
        // Set reference voltage to internal 5V (need capacitor)
        ADMUX |= _BV(REFS0);
        // Set Channel to ADCC1 (channel 1)
        ADMUX |= _BV( MUX0);
        // Enable ADC
```

```
	ADCSRA |= _BV(ADEN);
	// Enable Interrupt;
	ADCSRA |= _BV( ADIE);
	//ADCSRA |= _BV(ADLAR);
	// Set up prescaler to 128
	ADCSRA |= _BV( ADPS0) | _BV( ADPS1);// | _BV( ADPS2);
}


/*************************************************************************/
/* DESCRIPTION: Returns a part classification based on ADC value         */
/*************************************************************************/
uint8_t classify(uint16_t reflectVal){
	if(reflectVal >= B_Reflect){
		//countB+=1;
		return B_ID;
	}
	else if((reflectVal >= W_Reflect)){
		//countW+=1;
		return W_ID;
	}
	else if(reflectVal >= S_Reflect){
		//countS+=1;
		return S_ID;
	}else
	{
		//countA+=1;
		return A_ID;
	}
}
```

```
volatile uint16_t countCheck = 0;
volatile uint8_t mask = 0;


/*****************************************************************************/
/* DESCRIPTION: This function is used as a debounce and filter. Unlike other
debounce functionality this function DOES NOT use time delays. Instead,
consecutive reads of a pin on PORTD are taken. If the read is true, the program
continues until number of reads equals checkNum. If the read if false, the program
immediately returns false to the call function.

The logic behind this method take advantage of the fact that for this application,
sensors or buttons change state relatively slowly compared to the processor speed.
This means that should a button bounce and give a false reading when pushed,
the program will revisit the calling function again (since all inputs are interrupt based),
until button has stopped bouncing.

In the case of noise, this function acts like a low pass filter. Any frequency that
changes value at a lower period than the time it take to run this function
(controlled by checkNum) will be discarded. Any frequency or signal which has a
period greater than this function time will be accepted.

EX: An active high button is pushed by the user. The input signal begins bouncing.
The MCU reads a rising edge, enters this functions, and reads several true readings.
Then a falling edge happens and a false reading is received. The program can now safely
```

```
exit the calling interrupt because, since the button is actually pushed, there will
be another rising edge to trigger the interrupt and enter the debounce function again.
When bouncing has stopped, the last rising edge will trigger the interrupt and
this function will be called and return a true reading after reading the now stable
input checkNum times.                                   */
/*********************************************************************/
uint8_t debounce(uint8_t pin, uint8_t level, uint16_t checkNum){
        mask = (1<<pin); //create pin read mask
        level = (level<<pin);
        countCheck = 0;
        for(countCheck = 0; countCheck<checkNum; countCheck++)
        {//read the pin a number of times
                if((PIND & mask)!=level)//if any of the reads are false
                {
                        return 0;//return false
                        //the next edge will trigger interrupt if actually true
                        //and this function will be called again
                }
        }
        return 1;//return true
}

//same as above but on PORTJ
uint8_t debouncePINJ(uint8_t pin, uint8_t level, uint16_t checkNum){
        mask = (1<<pin); //create pin read mask
        level = (level<<pin);
        countCheck = 0;
        for(countCheck = 0; countCheck<checkNum; countCheck++)//read the pin a number of times
        {
                if((PINJ & mask)!=level)//if any of the reads are false
```

```
                {
                        return 0;//return false
                }
        }
        return 1;//return true
}

/*****************************************************************************/
/* DESCRIPTION: Updates part count based on input position value.
Usually called as updateCount(Parts[countSort]) to update the counter of the
part being sorted.                                  */
/*****************************************************************************/
uint8_t updateCount(uint8_t pos){

        if(pos==200)
        {
                countS++;
        }else if(pos==150)
        {
                countW++;
        }else if(pos==100)
        {
                countA++;
        }else
        {
                countB++;
        }
        return 1;
}
```

```
/*************************************************************************/
/* DESCRIPTION: Initialize mTimer. This function is only used for debugging.                    */
/*************************************************************************/
void mTimer_init(){
        TCCR1B |= _BV(CS11);//Set prescaler to 8
        TCCR1B |= _BV(WGM12); // Configure counter for CTC mode;
        OCR1A = 0x03E8; //Set top value for Timer counter
}//mTimer_init



/*************************************************************************/
/* DESCRIPTION: This is a millisecond timer. This function is only used for debugging.
*/
/*************************************************************************/
void mTimer(int count){
        int i; //counter for ms
        i = 0;
        TCNT1 = 0x0000; //Counter value register; Reset to 0
        TIFR1 |= _BV(OCF1A); //Set the OC interrupt flag by writing 1
        while(i<count){
                if((TIFR1 & 0x02) == 0x02){
                        TIFR1 |= _BV(OCF1A);//reset interrupt flag
                        i++;      //increment counter to count milliseconds
                }
        }
        return;
}//mTimer
```

```
/*********************************************************************/
/* DESCRIPTION: Initialize system timer which counts in ms                    */
/*********************************************************************/
void runTimerStart(void){
        TCCR4B |= _BV(WGM42); // Configure counter for CTC mode;
        OCR4A = 0x0007; //0.01s timer
        TCNT4 = 0x0000; //Counter value register; Reset to 0
        TIMSK4 |= _BV(OCIE4A);  //Enable Interrupt
        TCCR4B |= _BV(CS42)| _BV(CS40);//Set prescaler to 1024
        TIFR4 |= _BV(OCF4A);//reset interrupt flag
}//mTimer_init


//Stops System Timer
void runTimerStop(void){
        TCCR4B &= ~_BV(CS42)& ~_BV(CS40);
}

//resumes system timer
void runTimerResume(void){
        TCCR4B |= _BV(CS42) | _BV(CS40);
}

//Updates System time
ISR(TIMER4_COMPA_vect){
        runTime_d +=1;//add 1/1000 seconds to system time
```

```
}//ISR



//BAD ISR
ISR(BADISR_vect)
{
        PORTC = 0xFF;
}//BADISR


////////////////////////////////////////////////////////////////////////
//DISPLAY FUNCTIONS

void dispComplete (void)
{
        LCDClear();
        LCDClear();
        LCDWriteString("B  A  W  S  C");
        LCDWriteIntXY(0,1, countB, 2);
        LCDWriteString(" ");
        LCDWriteIntXY(0,1, countA, 2);
        LCDWriteString(" ");
        LCDWriteIntXY(0,1, countW, 2);
        LCDWriteString(" ");
        LCDWriteIntXY(0,1, countS, 2);
        LCDWriteString(" ");
        LCDWriteIntXY(0,1,countSort, 2);
}
```

```
void dispStatus(void)
{
	LCDClear();
	LCDWriteIntXY(0, 0, countSort, 2);
	LCDWriteStringXY(2,0,"/");
	LCDWriteIntXY(3,0, countPart, 2);
	LCDWriteStringXY(5,0, "(");
	LCDWriteIntXY(6,0, countB, 1);
	LCDWriteIntXY(7,0, countA, 1);
	LCDWriteIntXY(8,0, countW, 1);
	LCDWriteIntXY(9,0, countS, 1);
	LCDWriteStringXY(10,0, ")");
	LCDWriteStringXY(12,0, "T");
	LCDWriteIntXY(13,0, runTime_d/100, 3);
	LCDWriteIntXY(0, 1, CurPosition, 3);
	LCDWriteStringXY(3,1, ">");
	LCDWriteIntXY(4, 1, Parts[countSort], 3);
	LCDWriteIntXY(12, 1, adcDisp, 4);

}

void dispFLAGS(void){
	LCDClear();
	LCDWriteString("M");
	LCDWriteInt(MOTORFLAG,1);
	LCDWriteString(" P");
	LCDWriteInt(PAUSEFLAG,1);
	LCDWriteString(" T");
	LCDWriteInt(TARGETFLAG,1);
	LCDWriteStringXY(0,1," ");
```

```
        LCDWriteString("D");
        LCDWriteInt(DECELFLAG,1);
        LCDWriteString(" S");
        LCDWriteInt(HOLDFLAG,1);
}

extern volatile char ROLLFLAG;
void dispPause(void)
{
        LCDClear();
        LCDWriteString("B  A  W  S  O");
        LCDWriteIntXY(0,1, countB, 2);
        LCDWriteString(" ");
        LCDWriteIntXY(0,1, countA, 2);
        LCDWriteString(" ");
        LCDWriteIntXY(0,1, countW, 2);
        LCDWriteString(" ");
        LCDWriteIntXY(0,1, countS, 2);
        LCDWriteString(" ");
        if(ROLLFLAG)
        {
                LCDWriteIntXY(0,1, countPart + (PARTS_SIZE- countSort), 2);
        }else
        {
                LCDWriteIntXY(0,1, countPart - countSort, 2);
        }

}
```

B-3 STEPPER.C

```
/*
 * stepper.c
 *
 * Created: 2022-12-09
 * Author: Matthew Ebert V00884117; Scott Griffioen V00884133
 * For: Sorting Machine, MECH 458, University of Victoria
 *
 * Dependencies: main.h
 *
 * BOARD: ATMEGA 2560
 *
 * Description: This file contains functions used in main.c
 *
 */

#include "main.h"

//GLOBALS
volatile uint8_t Steps2Acc= 50;//number of steps to accelerate from MAXDELAY to MINDELAY
volatile uint8_t accSteps= 0;//indicates current step of acceleration profile
volatile uint16_t CurAcc[50];//hold acceleration profile

volatile uint16_t CurDelay = 0;//the current delay for stepper
volatile int8_t Dir = 1;//the current direction of the stepper
volatile int8_t NextDir = 1;//the next calculated direction needed
volatile uint8_t CurPosition = 50;// the current stepper position
volatile int16_t CurError = 0;//the current position error of the stepper
```

```
//Coil excitation pattern and state counter
//L1,L2                    //L1,L4                    //L4,L3                    //L2,L3
volatile int8_t StepStates[] = {0b11011000,0b10111000,0b10110100, 0b11010100};
volatile int8_t CurState = 0;



volatile uint8_t StepsDelta = 0;//legacy variable
//EXTERNAL GLOBAL
extern uint8_t Parts[PARTS_SIZE];
extern volatile uint8_t countSort;
extern volatile char HALLSENSOR;//Needs to be false
extern volatile char DECELFLAG;
extern volatile char EXFLAG;
extern volatile char PAUSEFLAG;
extern volatile char TARGETFLAG;
extern volatile char HOLDFLAG;
extern volatile uint8_t Steps2Exit;
extern volatile char DROPFLAG;




/*****************************************************************************/
/* DESCRIPTION: This function writes to the stepper driver, updates
the position and state variables, and resets the delay counter. It also
handles roll over for CurState and CurPosition.
                                        */
/*****************************************************************************/
uint8_t step(void){
        CurState = CurState + Dir;//Update CurState based on Dir
```

```
        //stepper roll over
        if (4 <= CurState){CurState = 0;}
        else if (-1 >= CurState){CurState = 3;}

        PORTA = StepStates[CurState]; //Step
        CurPosition = CurPosition + Dir;//Update CurPosition base on Dir
        //protect against roll over
        if(CurPosition > 225 && Dir==1){CurPosition -=  200;}
        else if(CurPosition < 25 && Dir==-1){CurPosition += 200;}

        TCNT3 = 0x0000;//Reset Counter

        return 1;          //return step;
}//step




/*****************************************************************************/
/* DESCRIPTION: This function updates the variable CurError based on the
current position and the current target position indicated by
Parts[countSort].
This function also handles several edge cases indicated by the flags
HOLDFLAG, PAUSEFLAG, and TARGETFLAG (see variable initialization for meaning).
                                        */
/*****************************************************************************/
uint8_t stepUpdateError(void)
{
        if(HOLDFLAG)
        {
```

```
                if(abs(CurError)<DROP_REGION)//We may need to check the time since slip to see if the part
fell
                {//Maybe a reduced drop region and a delay to ensure piece hits
                        HOLDFLAG = 0;
                        PAUSEFLAG = 0;
                        runMotor();
                        CurError = Parts[countSort] - CurPosition;
                }else
                {
                        CurError = Parts[countSort-1] - CurPosition;
                }
        }else
        {
                CurError = Parts[countSort] - CurPosition;
        }



        if(CurError>100)
        {
                CurError = CurError - 200;
        }else if(CurError<-100)
        {
                CurError = CurError + 200;
        }


        if(abs(CurError) < Steps2Acc && !DROPFLAG)//change if slowing down to quickly at zone; may
cause oscillation
        {
                TARGETFLAG = 1;
        }else
        {
```

```
                TARGETFLAG = 0;
        }
        return 1;
}



/*************************************************************************/
/* DESCRIPTION: This function updates the stepper direction (Dir).
Dir is updated from the calculated variable NextDir. NextDir is set based on the
stepper speed and distance to target. To avoid stalling the stepper, if NextDir
does not equal Dir, the stepper must be slowed to MAX delay before Dir is updated.
This is controlled by the DECELFLAG.
                                        */
/*************************************************************************/
uint8_t stepUpdateDir(void){
        if(CurError == 0)
        {// if stepper is at target
                if(CurDelay > (MAXDELAY-MINDELAY))
                {// if stepper can stop
                        Dir = 0; //stop stepping
                        TARGETFLAG = 0; //clear target flag
                        return 1;
                }else
                {//Decelerate stepper
                        DECELFLAG = 1;
                        return 0;
                }
        }else if((abs(CurError)>SPIN_ROUND_LIMIT) && (CurDelay<MAXDELAY))
        {//Next target is close in same direction and at high speed
```

```
                DECELFLAG = 0;//Don't slow down
                if(Dir != 0)
                {//Keep direction
                        NextDir = Dir;
                }else
                {//edge case where Dir might be zero
                        Dir = 1;
                        return 1;
                }
        }else
        {//Calculate closest direction
                NextDir = (CurError>0) - (CurError<0);
        }

        if(NextDir == Dir)
        {//next direction is the same
                Dir = NextDir;
                return 1;
        }else if(CurDelay >= MAXDELAY)
        {//stepper is can change direction
                Dir = NextDir;
                return 1;
        }else
        {//Decelerate stepper to switch directions
                DECELFLAG = 1;
                return 0;
        }
        return 1;
}
```

```
/*****************************************************************************/
/* DESCRIPTION: This function updates the stepper delay, CurDelay, which
controls the speed of the stepper. This function uses the acceleration profile
 in CurAcc[] to accelerate the stepper and keep track of the current
 acceleration step.
 The operation of this function can be summarized into two steps
 1. if requested by DECELFLAG, PAUSEFLAG, or TARGETFLAG
        - Decelerate the stepper until MAXDELAY is reached
2. Otherwise accelerate the stepper until MINDELAY

This creates a spring like action for the stepper as it is always trying to reach
max speed unless instructed otherwise.
                                        */
/*****************************************************************************/
uint8_t stepUpdateDelay(void)
{

        if(Dir==0)
        {//if stepper is not stepping:
                stepRes();//reset stepper
        }else if(TARGETFLAG || DECELFLAG || PAUSEFLAG)
        {//Decelerate if prompted
                CurDelay = CurDelay + CurAcc[accSteps];
                if (CurDelay > MAXDELAY)
                {
                        accSteps = 0;
                        CurDelay = MAXDELAY;
                        DECELFLAG = 0;
```

```
		}else if(accSteps>0)
		{
			accSteps--;
		}

	}else if(CurDelay>MINDELAY)
	{//Accelerate if able
		CurDelay = CurDelay -  CurAcc[accSteps];
		if (CurDelay <= MINDELAY || CurDelay > MAXDELAY)
		{//overflow protection
			CurDelay = MINDELAY;
		}
		if(accSteps<Steps2Acc)
		{//acceleration increase
			accSteps++;
		}
	}else
	{
		return 0;
	}

	OCR3A = CurDelay;//set the new delay
	return 1;
}


//Resets stepper parameters when the stepper is stopped
void stepRes(void){
	accSteps = 0;
```

```
        StepsDelta = 0;
        CurDelay = MAXDELAY;
}



//Initializes the hardware timer used to control the stepper
void stepTimer_init (void)
{
        TCCR3B |= _BV(WGM32);//Set CTC mode
        OCR3A = 0xFFFF; //Clear compare register A
        TCNT3 = 0x0000; //Clear count register
        TIMSK3 |= _BV(OCIE3A);  //Enable Interrupt
        return;
} //stepTimer_init

//enable the stepper timer and reset the stepper.
void stepStart(void){
        TCNT3 = 0x0000;//Reset counter
        OCR3A = MAXDELAY;//Set compare value
        TCCR3B |= _BV(CS31) | _BV(CS30);//Enable Stepper with prescaler
        TIFR3 |= _BV(OCF3A);//Reset interrupt flag
        CurDelay = MAXDELAY;//Reset CurDelay
}//stepStart

//stop the stepper timer
void stepStop(void){
        TCCR3B &= ~_BV(CS31);//Disable timer
        TCCR3B &=~_BV(CS30);
}//stepStop
```

```
/*************************************************************************/
/* DESCRIPTION: This function sets up the stepper for operations. First it
calls for the stepper profile calculation. Next it calibrates the stepper position
and state with the HE sensor.
                                        */
/*************************************************************************/
int8_t stepCalibrate(void){

        //Calculate the acceleration profile
        stepCalcAcc();

        //set stepper to slowest speed
        CurDelay = MAXDELAY;
        CurPosition = 0;//set CurPosition

        //move 50 steps to align poles and steps
        Parts[0] = 50;//Set stepper to move 50 steps
        stepStart();//Start stepper
        while(CurError !=0)
        {
                //prevent stepper from accelerating
                DECELFLAG = 1;
        }

        //move until HE triggers ISR(INT3_vect)
        HALLSENSOR = 0;
        CurPosition = 0;
        while(!HALLSENSOR)
        {
```

```
                //keep stepper moving forwards
                if(abs(CurError)<30 && !HALLSENSOR){
                        CurPosition = 0;
                }
        }//Wait for hall sensor to trigger

        //move stepper to face black region
        Parts[0] = B_ID;
        return 1;
}


/*********************************************************************/
/* DESCRIPTION: Calculates the stepper acceleration profile based on
        - the minimum and maximum delay between steps
        - the maximum acceleration
        - the number of steps to apply jerk (increase/decrease acceleration)
        The profile follows an S-curve shape.
        First stage: acceleration is increased every step until MAXACC is reached
        Second stage: acceleration is held constant until near max speed
        Third stage: Decrease acceleration as max speed is approached

        The profile is saved in the array CurAcc[]. This array has the size Steps2Acc
        which indicated how many steps it take to go from rest to max speed.
                                                */
/*********************************************************************/
void stepCalcAcc(void){

        uint16_t JERK = MAXACC/JERKSTEPS;
        uint16_t steps = 0;
```

```
uint16_t delay = MAXDELAY;

//FIRST STAGE: positive jerk
CurAcc[steps] = 0;
for(steps = 1; steps<JERKSTEPS; steps++){
        delay -=CurAcc[steps-1];
        CurAcc[steps] = CurAcc[steps-1]+JERK;
        if(CurAcc[steps]>MAXACC){
                CurAcc[steps] = MAXACC;
        }
}//Increase Acceleration


//Second Stage: Constant Acceleration
CurAcc[steps] = MAXACC;
while((delay -MAXACC -JERK*JERKSTEPS*JERKSTEPS/2)>MINDELAY){
        delay -=CurAcc[steps-1];
        if(delay<MINDELAY){
                delay = MINDELAY;
        }
        steps++;
        CurAcc[steps] = MAXACC;
}//Constant Acceleration

//Third Stage: Negative jerk to Max Speed -> MINDELAY
while(delay >MINDELAY){
        steps++;

        delay -=CurAcc[steps-1];
        if(JERK> CurAcc[steps-1]){
```

```
                        CurAcc[steps] = 0;
                        break;
                        }else{
                        CurAcc[steps] = CurAcc[steps-1]-JERK;

                }

        }//Decrease Acceleration

        //Record how many steps it take to reach maximum speed from rest
        Steps2Acc = steps;
}
```

APPENDIX C – MICROCONTROLLER WIRING

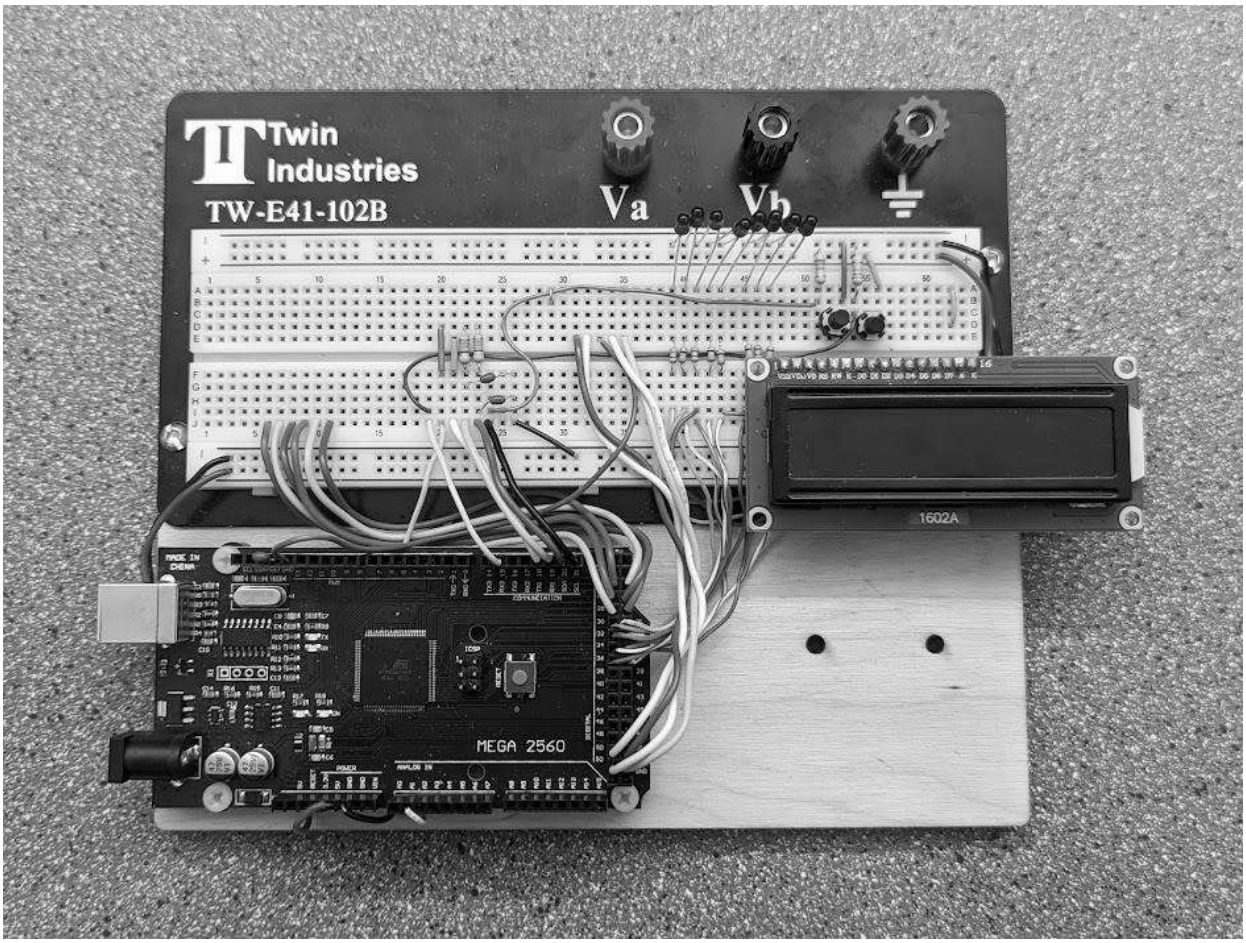


Figure 11 - MCU Wiring Diagram

APPENDIX D – PROJECT PROPOSAL

MECH 458
Milestone 5 Report
University of Victoria

Final Project Initial Proposal

Group 6 - B03
Scott Griffioen - V00884133
Matthew Ebert - V00884117

Submitted: November 15th, 2022

Table of Contents

Table of Figures

Objectives

A set of cylindrical pieces must be sorted into corresponding spaces on a rotating platform controlled by a stepper motor. A conveyor belt is powered by a DC motor and controls the position of the cylindrical pieces and the stage of processing that is occurring. To appropriately sort and detect the position of each piece several sensors must be used.

Two optical sensors at different stages of the process trigger interrupts within the MCU that can be leveraged to trigger different processes and track the position of the pieces along the conveyor belt. A reflective sensor connected to an analog port relays reflective information to the MCU, which can be converted to digital reading and used to sort the cylinders through distinct readings of reflectivity. In addition, a ferromagnetic sensor can also be used to determine the material, however this will not be used for completion of this project.

The goal of the project is to utilize these sensor and control systems to optimize the sorting process of the cylinders to sort a set of 48 pieces in an efficient and accurate way.

Timeline

The following Gantt chart outlines the remaining work that is required to meet project goals and requirements before the final demonstration. To date, work has been completed that has allowed for a simple sorting system to operate as expected that could meet the minimum efficiency requirements. However, significant room for further optimization remains in the coming weeks. The Gantt chart below describes these stages.

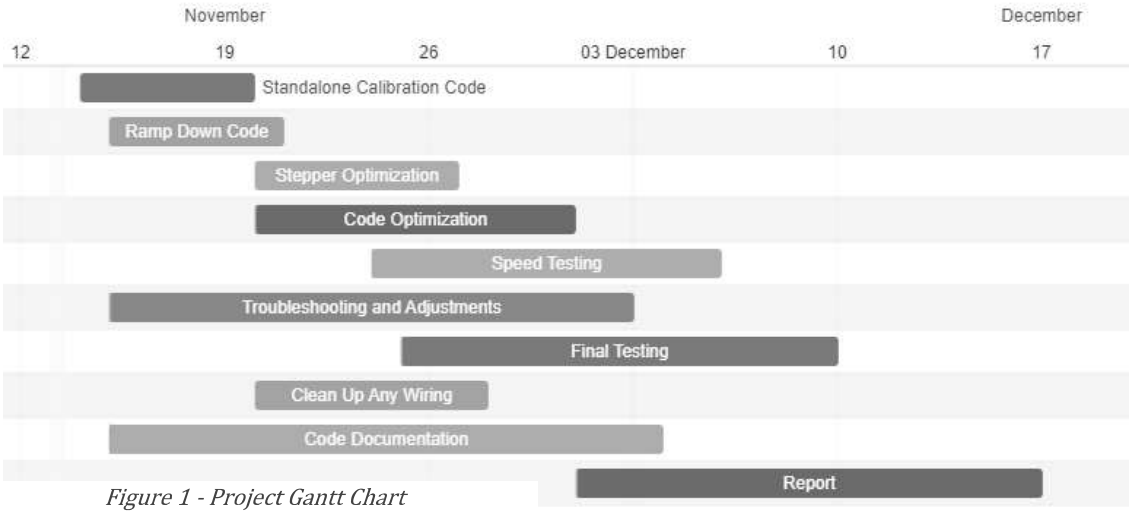


Figure 1 - Project Gantt Chart

System Design

Main Program

The main program contains 2 sections: initialization and run-time. The initialization portion runs once on system startup. The hardware is configured, and the stepper motor is calibrated to a known position. The program then starts the DC belt motor and enters a standby mode. The program exits the standby mode when certain flags are sets. These flags send the program to handle the following events

- **Add Part**: a part has passed through the reflectivity sensor and is ready for classification
- **Sort Part:** a part is at the end of the belt and ready to be sorted into the correct bin
- **Pause:** The user has pressed the pause button
- **RunDown:** The user has set the system to sort the remaining unsorted part and shutdown

It should be noted that most of the system processes are handled by interrupts. These are described in the following section.

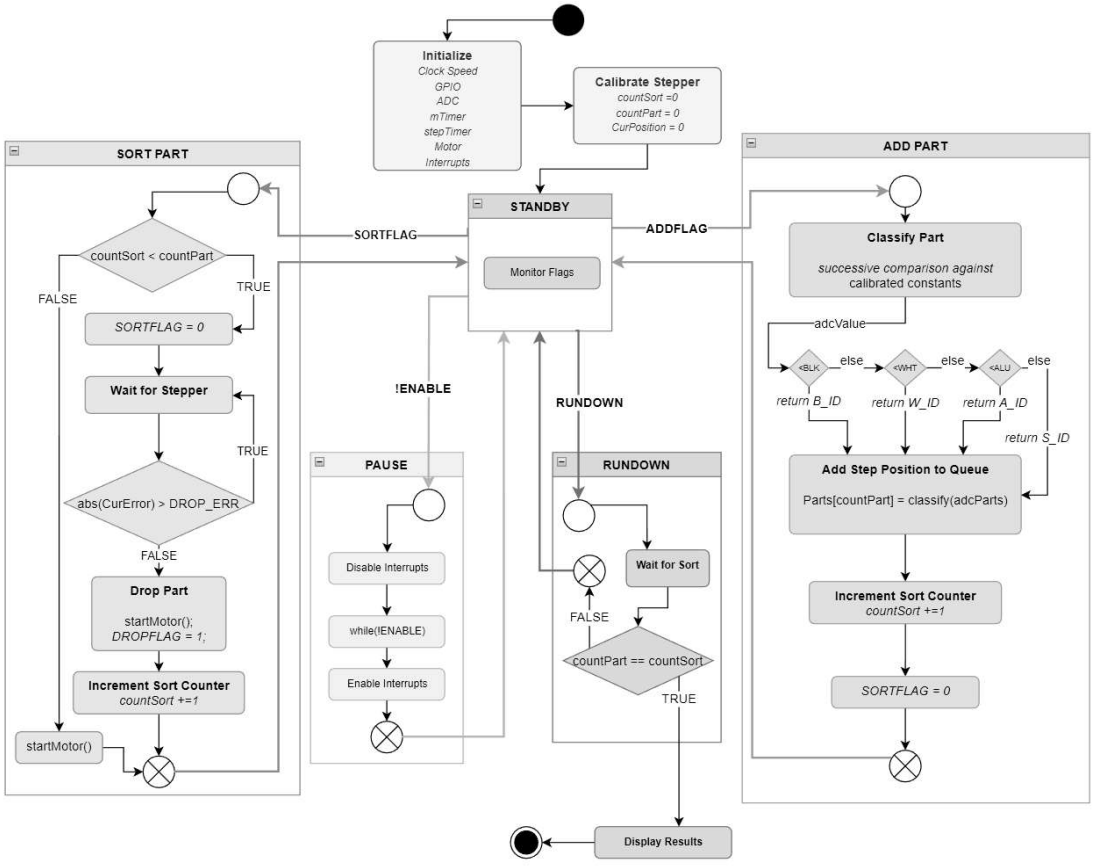


Figure 2 The state description of the sorting system's main program.

Interrupt Service Routines

The primary component of the control software is the stepper motor ISR. The sub-module is attached to TIMER3 on the 2560 board and runs near constantly during system operation. The timer is configured for output compare mode and the compare register adjusted to control the step timings of the motor.

The *Parts* array is the driving parameter of the stepper ISR. Every execution of the TIMER3 ISR, the position error of the stepper is calculated from its current known position and a target position given by the value stored in *Parts* at index *countSort.* The *countSort* global variable will change the stepper target when it is updated on the next execution of the TIMER3 ISR.

Other variables including CurDelay, CurAcc, and Steps Delta are used to implement the acceleration profile on the stepper motor.

The other ISR serve to handle the system sensor and set/reset flags for the main program.

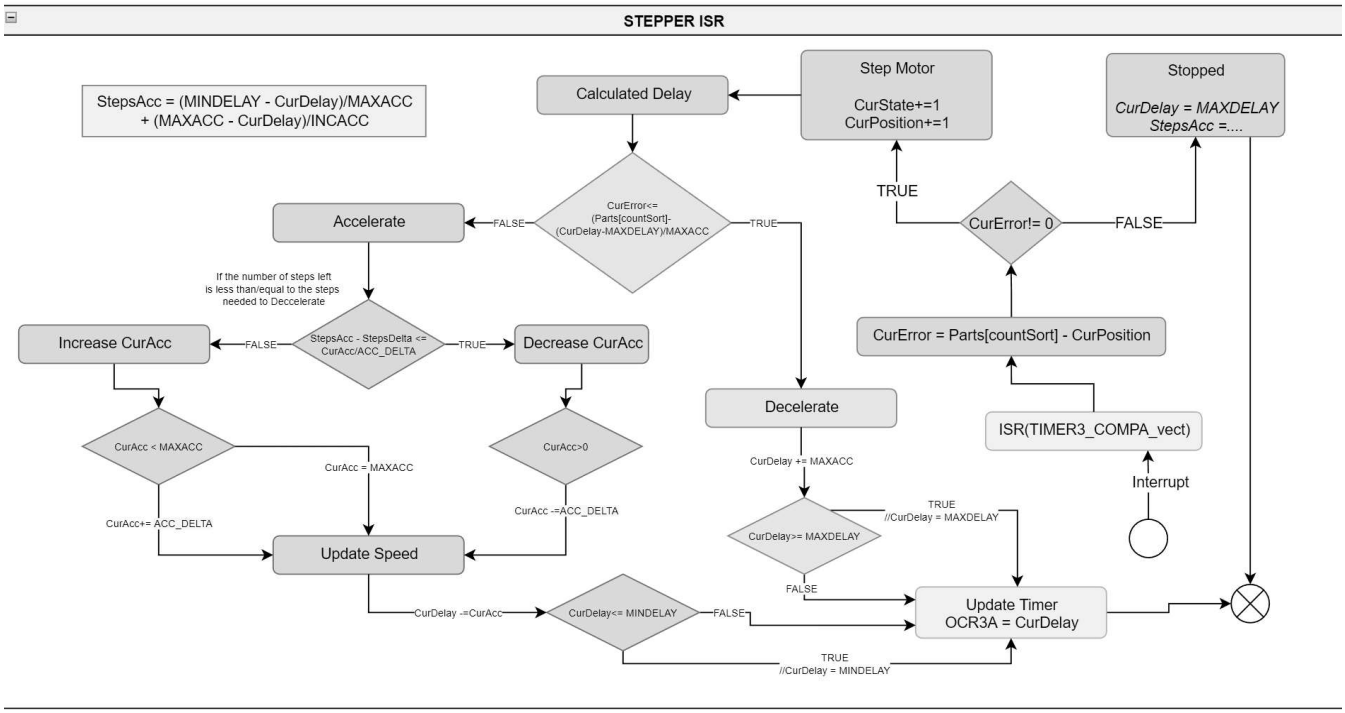


Figure 3 A flow chart of the STEPPER ISR implementation

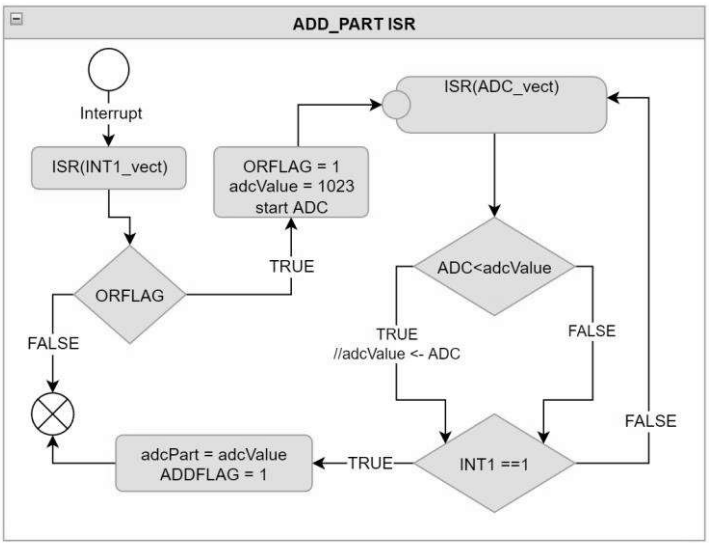


Figure 4 The ADC interrupt module. This implementation combines INT1 external interrupt and the internal ADC interrupt ISRs.

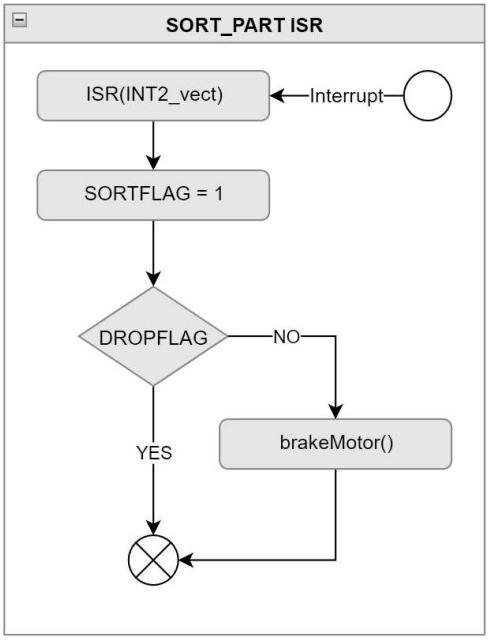


Figure 5 The INT2 external interrupt which signals the controller a part is at the end of the belt

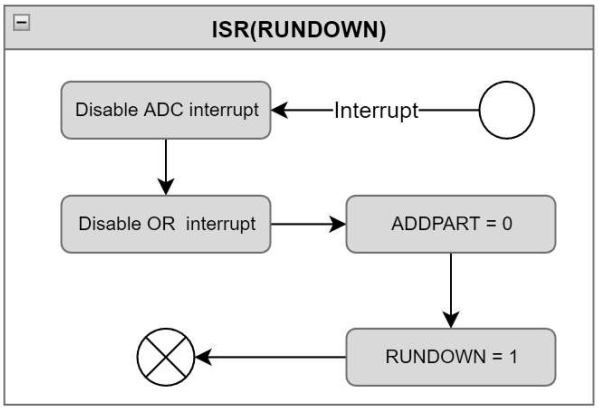


Figure 6 RUNDOWN ISR; This interrupt will connect to an external button

Circuit Diagram

A detailed preliminary circuit diagram is given on the following page, see Figure 7. It describes the components and features that will be utilized for completion of this project. Omitted from the diagram is any circuitry/wiring such as motor controllers and specific sensor details that were already completed on the apparatus prior to this project beginning.

For troubleshooting purposes there are 8 x LED's along with a 16x2 LCD screen connected by pins 30:37. The stepper motor is connected to pins 24:29, while the DC motor is connected to pins 50:53 with a PWM connection to pin 13. Sensor and switch input is wired to 18:21 (INT3:INT0). The first optical sensor of the apparatus is connected to pin 20 (INT1), while the second optical sensor at the end of the conveyor is connected to pin 19 (INT2). The hall-effect sensor is then connected to pin 18 (INT3) and a low pass push button is connected at pin 21(INT0). A second pushbutton is also shown in the circuit diagram below; however, it is not currently in use. Finally, the reflective sensor used to differentiate and sort the cylindrical pieces is wired to pin A1 of the analog port.

Figure 7 - Circuit Diagram