Department of Mechanical Engineering

MECH 458 – Mechatronics

Course Instructor: Dr Homayoun Najjaran

Lab Instructor: Patrick Chang

# Final Design Report

Prepared by:

**B02– Group 5**

**Cole Manton V00917415**

**Jack Martin V01016907**

## Table of Contents

## Table of Figures and Tables

# 1 Introduction

The primary objective of the MECH458 project is to design, program, and optimize the software and control framework for a conveyor belt sorting system. This system is powered by an Atmega2560 microcontroller (MCU), which processes inputs from a range of sensors, and controls both a DC motor and a stepper motor. Based on data from the sensor array, the software must accurately identify the material of each object and sort it into the correct bin as efficiently as possible. Below is an overview of the system requirements and the role of each major hardware component in this project. [1]

I. A DC motor is used to actuate the conveyor belt and move objects through the sorting system.
II. An optical sensor informs the system when an object is in front of the reflective sensor.
III. A reflective sensor determines the material based on pre-calibrated visual characteristics.
IV. An exit sensor informs the system when an object has reached the end of the conveyor belt.
V. A stepper motor is used to rotate the sorting bins to the correct material.
VI. A hall-effect sensor is used to initialize the stepper motor to a home location.

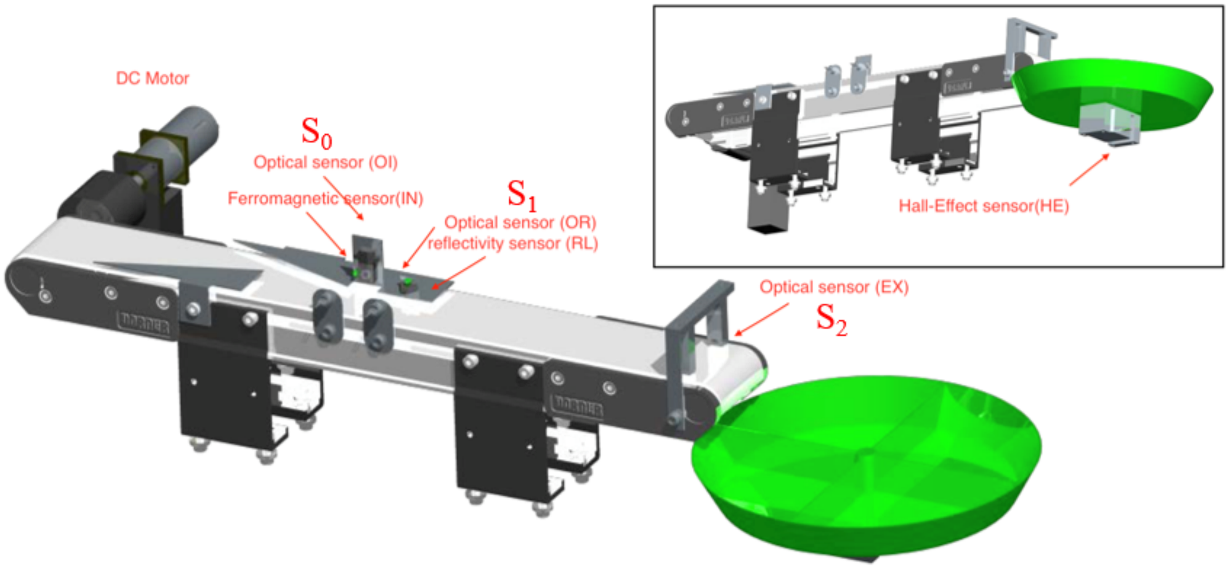


*Figure 1: System Overview [1]*

When the hardware components are integrated with the appropriate software and control logic, the sorting system can accurately scan, identify, and sort each object placed on the conveyor belt. It should be noted that Figure 1 above contains a ferromagnetic sensor which is not used in the current set up.

## 2 System Documentation

This section describes each software and hardware algorithm as well as how they are connected to the performance specifications.

## 2.1 System Tasks

The system is designed to transport an unspecified quantity of items along a conveyor belt, identify each item type, and sort it into the correct compartment within a tray. The items to be sorted include white Delrin, black Delrin, aluminum, and steel. This process must be completed for 48 items in under 60 seconds. Along with sorting, the system is required to perform two additional functions.

First, when the user sends a pause signal (through the use of a push button), the system will stop and display the count of each item that has been fully or processed as well as the count of pieces remaining on the belt. An item is considered fully processed once it has been sorted into the appropriate bin, while a partially processed item is one that has passed the reflective sensor but has not yet been sorted into a bin. Second, when a ramp-down signal is received (again from a push button), the system will complete processing all items currently on the conveyor, then halt and display the total number of each fully processed item.

## 2.2 System Diagram

This section presents visual representations of the software and hardware control logic utilized in this project. Flow diagrams include action and decision blocks to illustrate the progression through the implemented control logic, along with interrupt blocks that indicate points where the control flow is intentionally paused or altered.

## 2.2.1 High Level Diagram

Figure 2 shows the high-level flow chart for the algorithm that is implemented in the system.

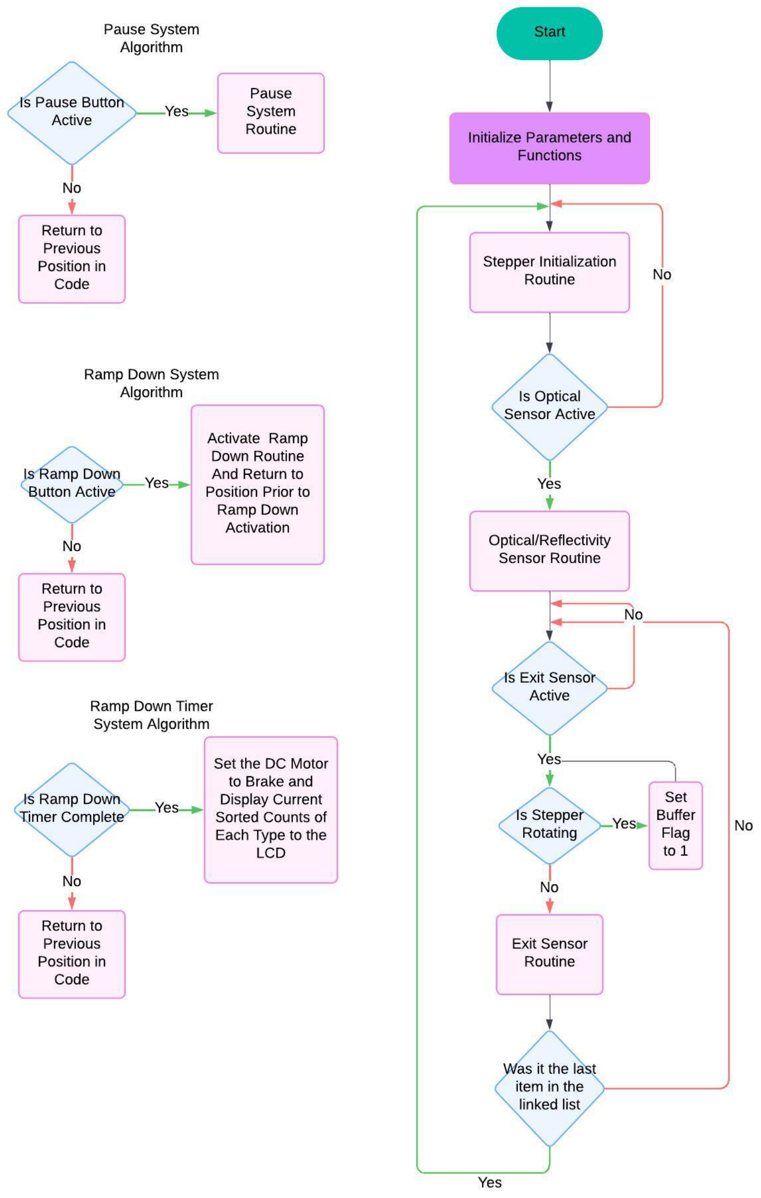

Pause System Algorithm
Is Pause Button Active
Yes
Pause System Routine
No
Return to Previous Position in Code
Ramp Down System Algorithm
Is Ramp Down Button Active
Yes
Activate Ramp Down Routine And Return to Position Prior to Ramp Down Activation
No
Return to Previous Position in Code
Ramp Down Timer System Algorithm
Is Ramp Down Timer Complete
Yes
Set the DC Motor to Brake and Display Current Sorted Counts of Each Type to the LCD
No
Return to Previous Position in Code
Start
Initialize Parameters and Functions
Stepper Initialization Routine
No
Is Optical Sensor Active
Yes
Optical/Reflectivity Sensor Routine
No
Is Exit Sensor Active
Yes
Is Stepper Rotating
Yes
Set Buffer Flag to 1
No
No
Exit Sensor Routine
Was it the last item in the linked list
Yes

*Figure 2: High Level Diagram*

The system starts when the reset button is pressed on the MCU, initializing all required parameters and functions for implementation. Next, it enters the 'Stepper Initialization Routine,' detailed in Section 2.2.2. After that, the system continuously checks if the optical sensor is active. When it is, the 'Optical/Reflectivity Sensor Routine' (Section 2.2.3) is initiated. Then, the exit sensor is monitored continuously, and if active, the system checks if the stepper motor is rotating. If it is, the buffer flag is set to 1; otherwise, the 'Exit Sensor Routine' (Section 2.2.4) begins.

The system then verifies if the sorted item is the last one in the linked list. If true, it returns to the top of the process to check the optical sensor. If not, it continues to monitor the exit sensor.

During sorting, two additional functions may be required. First, the pause function: when the pause button is pressed, the code enters the 'Pause System Routine' (Section 2.2.5). Second, the ramp function is activated if the ramp-down button is pressed. This triggers an interrupt-based timer, and once complete, sets the motor to brake and displays the current and sorted item types on the LCD.

### 2.2.2 Stepper Initialization Routine

Figure 3 Shows the Stepper Initialization Routine.

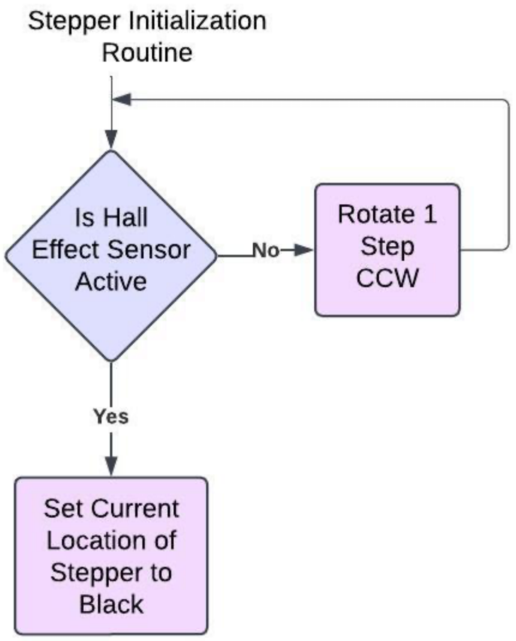

Stepper Initialization Routine
Is Hall Effect Sensor Active
No
Rotate 1 Step CCW
Yes
Set Current Location of Stepper to Black

*Figure 3: Stepper Initialization Routine Diagram*

This routine starts after initializing the necessary parameters and functions for the implementation. It first checks whether the hall effect sensor is active. If not, the stepper motor rotates one step counterclockwise. If the sensor is active, the current position of the stepper is marked as black.

### 2.2.3 Optical/Reflectivity Sensor Routine

Figure 4 shows the Optical/Reflectivity Sensor Routine

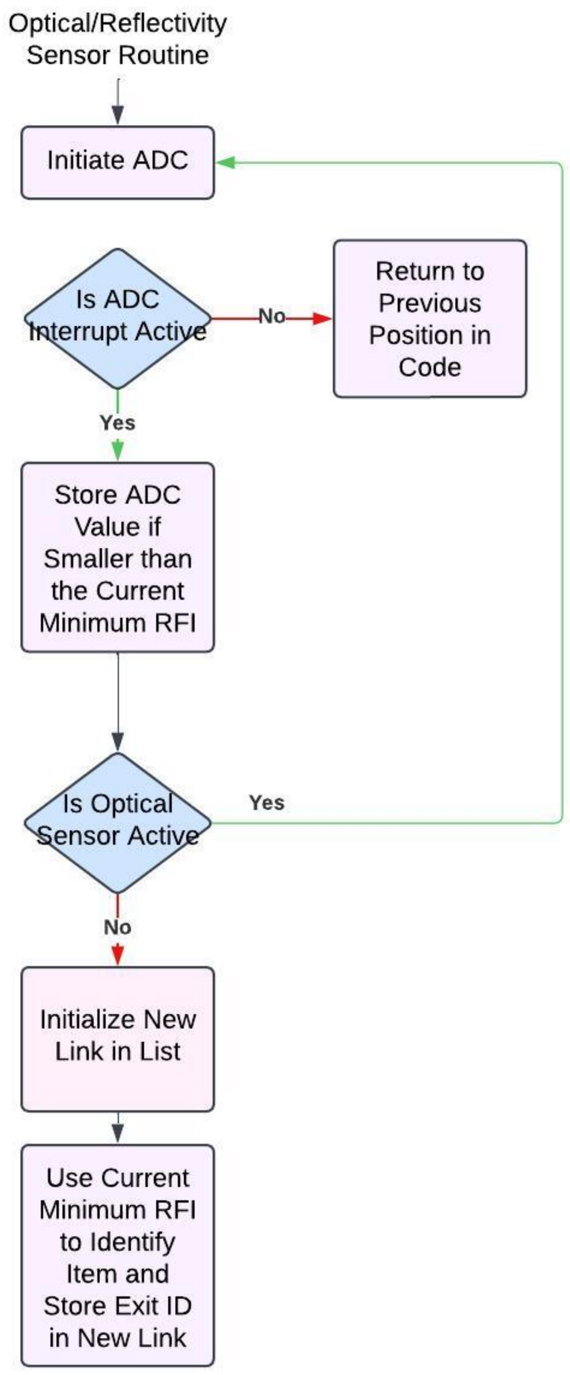

Optical/Reflectivity Sensor Routine
Initiate ADC
Is ADC Interrupt Active
No
Return to Previous Position in Code
Yes
Store ADC Value if Smaller than the Current Minimum RFI
Is Optical Sensor Active
Yes
No
Initialize New Link in List
Use Current Minimum RFI to Identify Item and Store Exit ID in New Link

*Figure 4: Optical/Reflectivity Sensor Routine Diagram*

In this routine, an ADC conversion is initiated upon entry. Then, the routine continuously checks if the ADC interrupt is active, indicating the completion of the conversion. The ADC output represents the reflective index (RFI) of the item. Once complete, the value is stored if it is lower than the current minimum RFI. Next, the optical sensor is checked; if it is active, another ADC conversion starts. If not, a new link is created in the linked list, using the current minimum RFI to identify the item and storing that ID in the new link.

### 2.2.4 Exit Sensor Routine

Figure 5 Shows the Exit Sensor Routine.

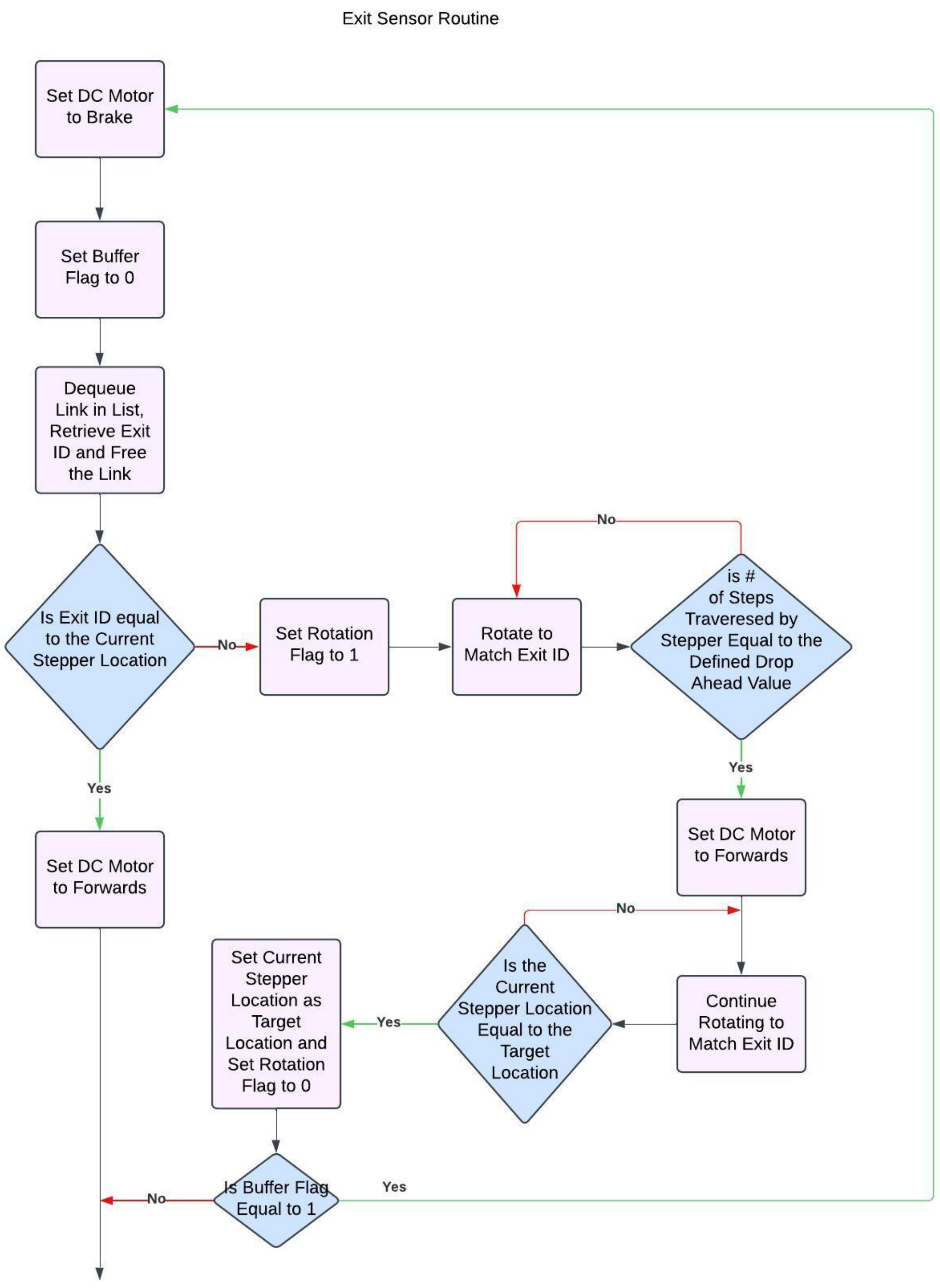

Exit Sensor Routine
Set DC Motor to Brake
Set Buffer Flag to 0
Dequeue Link in List, Retrieve Exit ID and Free the Link
Is Exit ID equal to the Current Stepper Location
No
Set Rotation Flag to 1
Rotate to Match Exit ID
No
is # of Steps Traveresed by Stepper Equal to the Defined Drop Ahead Value
Yes
Set DC Motor to Forwards
Yes
Set DC Motor to Forwards
No
Continue Rotating to Match Exit ID
Is the Current Stepper Location Equal to the Target Location
Yes
Set Current Stepper Location as Target Location and Set Rotation Flag to 0
Is Buffer Flag Equal to 1
No
Yes

*Figure 5: Exit Sensor Routine Diagram*

Upon entering this routine, the conveyor belt stops, and the buffer flag is reset to zero. A link is then dequeued from the list, its exit ID is retrieved, the link is released, and the memory is freed. The system checks if the exit ID matches the stepper motor's current position. If it does, the conveyor belt restarts, and the item is dropped. If not, the rotation flag is set to one, and the tray begins rotating to align with the exit ID.

During rotation, the system monitors whether the stepper has traversed the specified drop-ahead value. If not, rotation continues; if so, the conveyor belt turns on, and the item is dropped. The stepper motor keeps rotating until it reaches the exit ID, at which point the stepper’s position is updated to the target value, and the rotation flag is cleared.

After this, the system checks if the buffer flag is set to 1, indicating that an item passed the exit sensor during the rotation. If true, the routine restarts. If false, the routine exits.

### 2.2.5 Pause System Routine

Figure 6 shows the Pause System Routine.

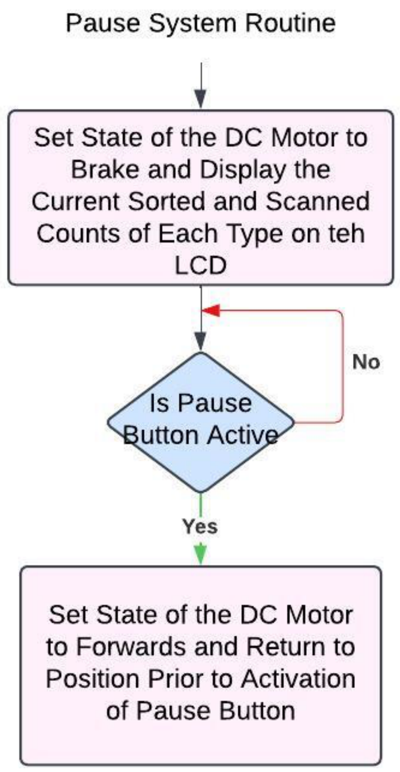

Pause System Routine
Set State of the DC Motor to Brake and Display the Current Sorted and Scanned Counts of Each Type on teh LCD
Is Pause Button Active
No
Yes
Set State of the DC Motor to Forwards and Return to Position Prior to Activation of Pause Button

*Figure 6: Pause System Routine Diagram*

When this routine begins, the conveyor belt pauses, and the LCD displays the current counts of each type that have been fully processed and total scanned items on the belt. The system then checks if the pause button is active. If it is not, the LCD continues to display the object counts. If the pause button is active, the conveyor belt resumes, and the code returns to its position just before the pause button was first pressed.

## 2.3 Software and Hardware Algorithm

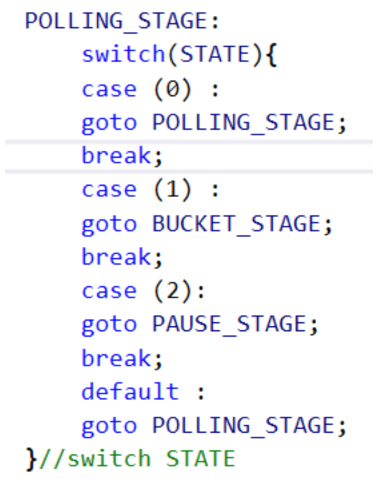
This section offers a comprehensive overview of the software and hardware control logic implemented in this project. It begins by detailing the control logic for each state within the state machine. Next, an in-depth explanation of each function utilized in the state machine is provided. Finally, the section concludes with a description of each Interrupt Service Routine (ISR) and the corresponding sensors.

## 2.3.1 System States

This section provides detailed explanations of the algorithms and logic implemented in each of the four states of the state machine.

### *2.3.1.1 Polling Stage*

Figure 7 shows the code used in the Polling State of the state machine



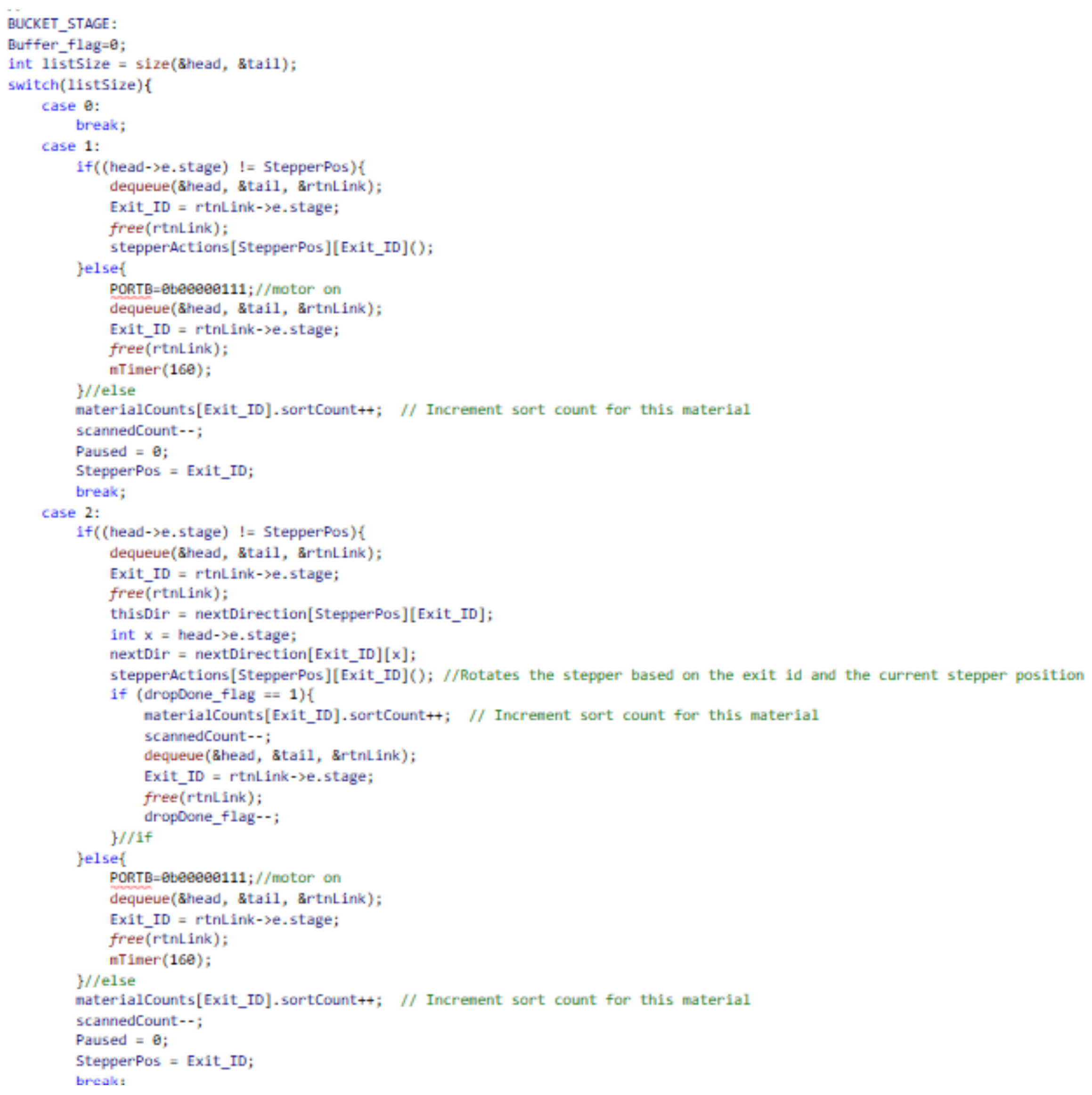
```
POLLING_STAGE:
    switch(STATE){
    case (0) :
    goto POLLING_STAGE;
    break;
    case (1) :
    goto BUCKET_STAGE;
    break;
    case (2):
    goto PAUSE_STAGE;
    break;
    default :
    goto POLLING_STAGE;
}//switch STATE
```

*Figure 7 Overview of Polling Stage*

The polling stage serves as the foundation of the state machine in this project. During variable declaration, the system's state is initialized to 0, placing it in the polling stage until an interrupt occurs. When the interrupt is triggered, the state variable is updated to the corresponding stage, which is then entered upon returning to the switch statement. After completing the necessary stage, the state variable is reset to 0, and the polling stage logic is executed again.

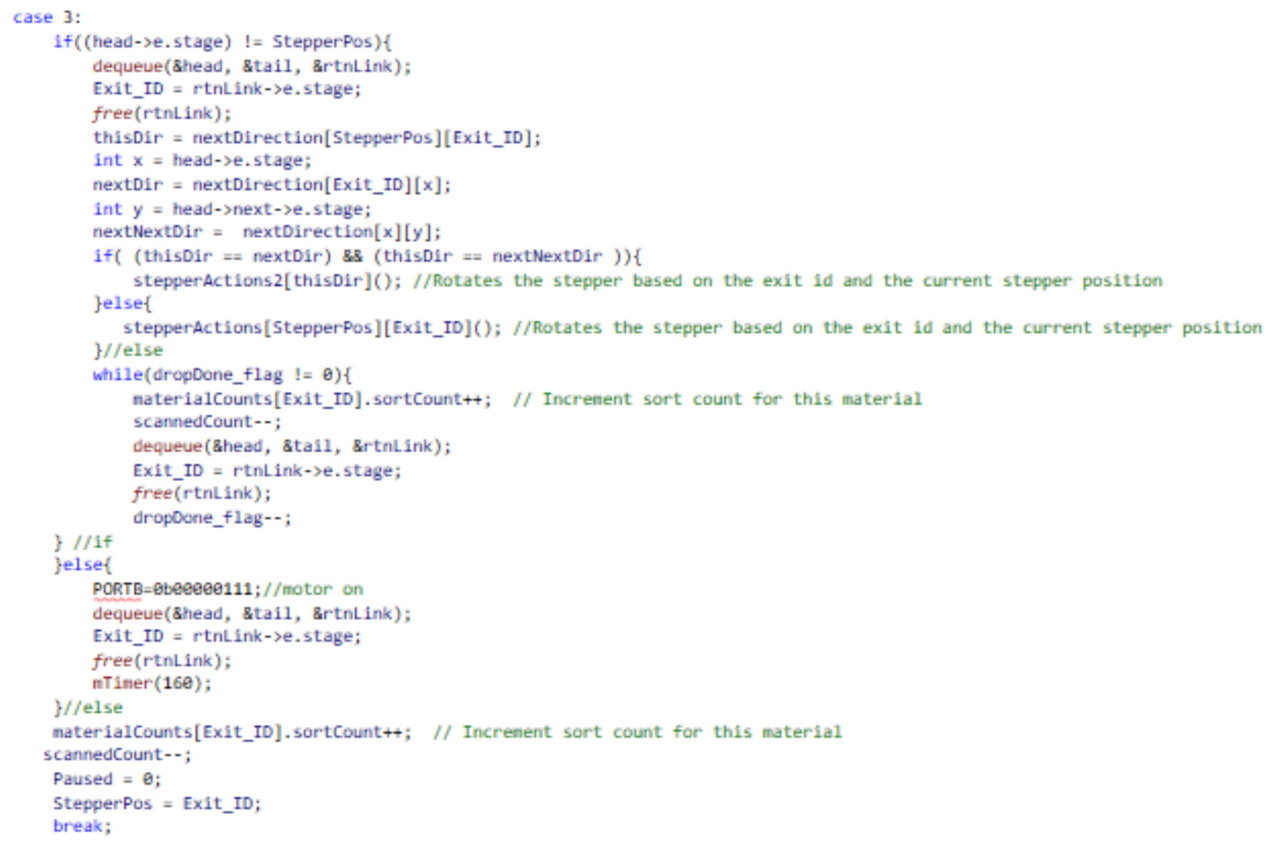
### *2.3.1.2 Bucket Stage*

Figures 8-10 detail the code for the Bucket State.



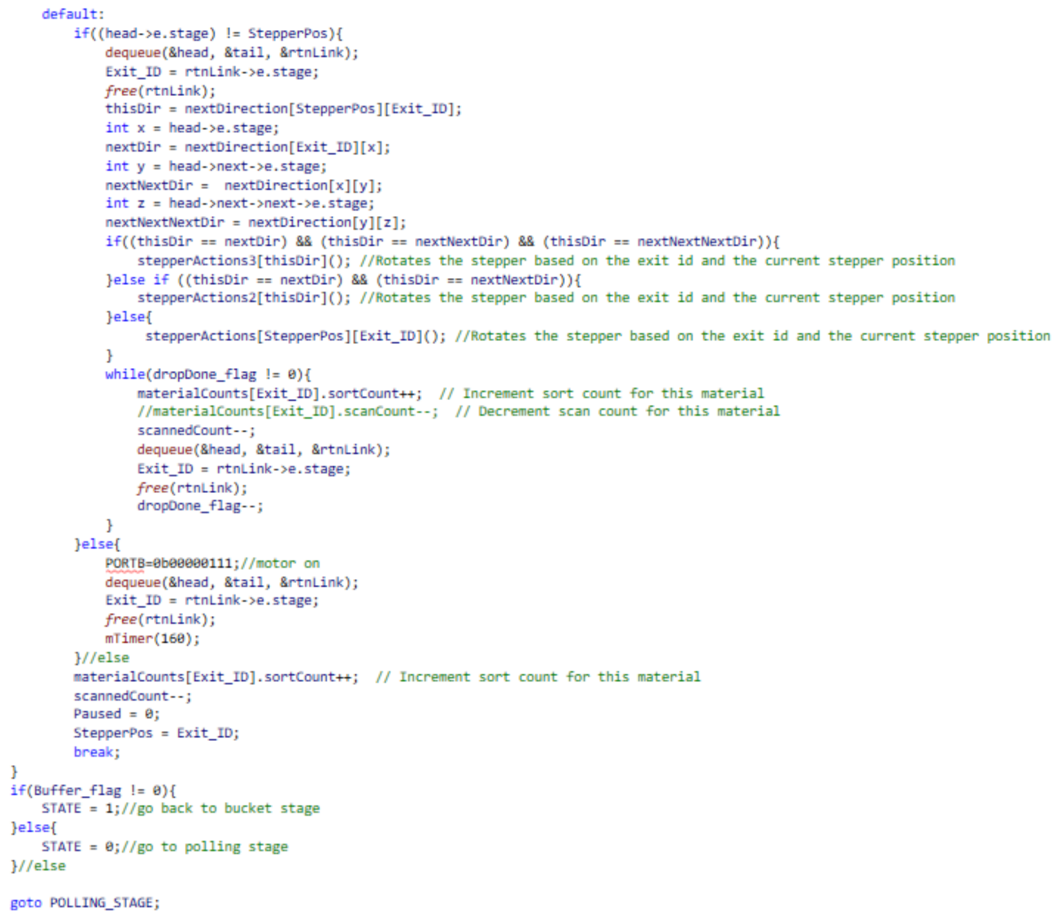
```
BUCKET_STAGE:
Buffer_flag=0;
int listSize = size(&head, &tail);
switch(listSize){
    case 0:
        break;
    case 1:
        if((head->e.stage) != StepperPos){
            dequeue(&head, &tail, &rtnLink);
            Exit_ID = rtnLink->e.stage;
            free(rtnLink);
            stepperActions[StepperPos][Exit_ID]();
        }else{
            PORTB=0b00000111;//motor on
            dequeue(&head, &tail, &rtnLink);
            Exit_ID = rtnLink->e.stage;
            free(rtnLink);
            mTimer(160);
        }//else
        materialCounts[Exit_ID].sortCount++;  // Increment sort count for this material
        scannedCount--;
        Paused = 0;
        StepperPos = Exit_ID;
        break;
    case 2:
        if((head->e.stage) != StepperPos){
            dequeue(&head, &tail, &rtnLink);
            Exit_ID = rtnLink->e.stage;
            free(rtnLink);
            thisDir = nextDirection[StepperPos][Exit_ID];
            int x = head->e.stage;
            nextDir = nextDirection[Exit_ID][x];
            stepperActions[StepperPos][Exit_ID](); //Rotates the stepper based on the exit id and the current stepper position
            if (dropDone_flag == 1){
                materialCounts[Exit_ID].sortCount++;  // Increment sort count for this material
                scannedCount--;
                dequeue(&head, &tail, &rtnLink);
                Exit_ID = rtnLink->e.stage;
                free(rtnLink);
                dropDone_flag--;
            }//if
        }else{
            PORTB=0b00000111;//motor on
            dequeue(&head, &tail, &rtnLink);
            Exit_ID = rtnLink->e.stage;
            free(rtnLink);
            mTimer(160);
        }//else
        materialCounts[Exit_ID].sortCount++;  // Increment sort count for this material
        scannedCount--;
        Paused = 0;
        StepperPos = Exit_ID;
        break;
```

*Figure 8 Bucket Stage Code 1*



```
case 3:
    if((head->e.stage) != StepperPos){
        dequeue(&head, &tail, &rtnLink);
        Exit_ID = rtnLink->e.stage;
        free(rtnLink);
        thisDir = nextDirection[StepperPos][Exit_ID];
        int x = head->e.stage;
        nextDir = nextDirection[Exit_ID][x];
        int y = head->next->e.stage;
        nextNextDir =  nextDirection[x][y];
        if( (thisDir == nextDir) && (thisDir == nextNextDir )){
            stepperActions2[thisDir](); //Rotates the stepper based on the exit id and the current stepper position
        }else{
           stepperActions[StepperPos][Exit_ID](); //Rotates the stepper based on the exit id and the current stepper position
        }//else
        while(dropDone_flag != 0){
            materialCounts[Exit_ID].sortCount++;  // Increment sort count for this material
            scannedCount--;
            dequeue(&head, &tail, &rtnLink);
            Exit_ID = rtnLink->e.stage;
            free(rtnLink);
            dropDone_flag--;
    } //if
    }else{
        PORTB=0b00000111;//motor on
        dequeue(&head, &tail, &rtnLink);
        Exit_ID = rtnLink->e.stage;
        free(rtnLink);
        mTimer(160);
    }//else
    materialCounts[Exit_ID].sortCount++;  // Increment sort count for this material
   scannedCount--;
    Paused = 0;
    StepperPos = Exit_ID;
    break;
```

*Figure 9 Bucket Stage Code 2*



```c
        default:
            if((head->e.stage) != StepperPos){
                dequeue(&head, &tail, &rtnLink);
                Exit_ID = rtnLink->e.stage;
                free(rtnLink);
                thisDir = nextDirection[StepperPos][Exit_ID];
                int x = head->e.stage;
                nextDir = nextDirection[Exit_ID][x];
                int y = head->next->e.stage;
                nextNextDir =  nextDirection[x][y];
                int z = head->next->next->e.stage;
                nextNextNextDir = nextDirection[y][z];
                if((thisDir == nextDir) && (thisDir == nextNextDir) && (thisDir == nextNextNextDir)){
                    stepperActions3[thisDir](); //Rotates the stepper based on the exit id and the current stepper position
                }else if ((thisDir == nextDir) && (thisDir == nextNextDir)){
                    stepperActions2[thisDir](); //Rotates the stepper based on the exit id and the current stepper position
                }else{
                     stepperActions[StepperPos][Exit_ID](); //Rotates the stepper based on the exit id and the current stepper position
                }
                while(dropDone_flag != 0){
                    materialCounts[Exit_ID].sortCount++;  // Increment sort count for this material
                    //materialCounts[Exit_ID].scanCount--;  // Decrement scan count for this material
                    scannedCount--;
                    dequeue(&head, &tail, &rtnLink);
                    Exit_ID = rtnLink->e.stage;
                    free(rtnLink);
                    dropDone_flag--;
                }
            }else{
                PORTB=0b00000111;//motor on
                dequeue(&head, &tail, &rtnLink);
                Exit_ID = rtnLink->e.stage;
                free(rtnLink);
                mTimer(160);
            }//else
            materialCounts[Exit_ID].sortCount++;  // Increment sort count for this material
            scannedCount--;
            Paused = 0;
            StepperPos = Exit_ID;
            break;
    }
    if(Buffer_flag != 0){
        STATE = 1;//go back to bucket stage
    }else{
        STATE = 0;//go to polling stage
    }//else

    goto POLLING_STAGE;
```

*Figure 10 Bucket Stage Code 3*

The bucket stage is activated when an object triggers the exit sensor (EX). The Interrupt Service Routine (ISR) checks if a rotation is in progress. If so, a buffer flag is set to 1, indicating that an object passed the exit sensor while the stepper motor is rotating.

Upon entering the bucket stage, the buffer flag is reset to 0 to signal that the object passing the exit sensor is now being processed. Next it acquires the size of the linked list, as different operations are required if there are less that 4 links in the list. Each case is how many links are in the list 1, 2, 3 and default is entered for 4 or more.

For case 1, the head is examined, if the value stored is the same as the previous value, the belt is started and the head is dequeued, set to Exit_ID and freed. If the head is not the same as the previous piece, the head is dequeued, assigned to Exit_ID, and freed. Next the lookup table stepperActions is called, taking in the Exit_ID and the current stepper position as inputs and rotating to the correct bin for sorting. After sorting the piece, the sortedCount and scannedCount for that piece is incremented and decremented, Paused is set to 0 as the belt is now rotating, and stepperPos is set to Exit_ID.

Case 2 has the potential for a continuous 180 degree rotation, therefore the process slightly differs to case 1. A new variable thisDir is introduced to determine the direction of rotation for the current piece. A second variable nextDir is set by checking the next piece in the queue and determining its rotation direction. The two previously mentioned variables are determined by passing a starting position and target position to a lookup table (nextDirection) that returns an integer. If they are determined to be the same in the rotation function, the system will attempt to drop both during a 180 degree rotation. Similarly to case 1, stepperActions is called to rotate the stepper. After the rotation, if dropDone_flag is non zero, then an additional piece was successfully dropped in a continuous 180 degree rotation, the second link must be dequeued to ensure the list remains in sequence.

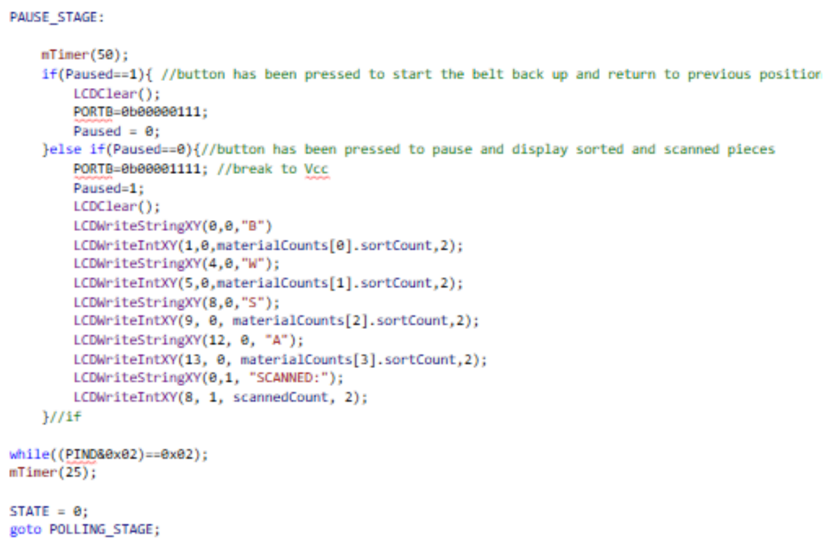
Case 3 is similar to case 2 and has the potential for either a continuous 180 or 270 degree rotation. In addition to variables thisDir and nextDir, a third variable nextnextDir is introduced to determine the direction of rotation between the 2nd and 3rd piece. If thisDir is equal to nextDir and nextnextDir a 270 degree rotation can be completed and stepperActions2 is called to complete the rotation. Otherwise, the same process as case 2 follows, and an additional link must be dequeued for every additional drop that was made to maintain the queue sequence.

The default case is similar to case 3, but has the potential for a continuous 360 degree rotation. Therefore, in addition to variables thisDir, nextDir and nextnextDir, a fourth variable nextnextnextDir is introduced to determine if the rotation between the 3rd and 4th piece. Next, if thisDir is equal to nextDir, nextnextDir and nextnextnextDir, stepperActions3 is called as a continuous 360 degree rotation can be completed. The remainder is the same as case 3.

Lastly, the buffer flag is checked. If it is set to 1, this means that another object passed the exit sensor while the stepper motor was rotating. In this case, the state remains in the bucket stage to process the new item. If the buffer flag is 0, the state is reset to 0, and the system returns to the polling stage.

#### *2.3.1.3 Pause State*

Figure 11 shows the code used in the Pause State.



```
PAUSE_STAGE:

    mTimer(50);
    if(Paused==1){ //button has been pressed to start the belt back up and return to previous positior
        LCDClear();
        PORTB=0b00000111;
        Paused = 0;
    }else if(Paused==0){//button has been pressed to pause and display sorted and scanned pieces
        PORTB=0b00001111; //break to Vcc
        Paused=1;
        LCDClear();
        LCDWriteStringXY(0,0,"B")
        LCDWriteIntXY(1,0,materialCounts[0].sortCount,2);
        LCDWriteStringXY(4,0,"W");
        LCDWriteIntXY(5,0,materialCounts[1].sortCount,2);
        LCDWriteStringXY(8,0,"S");
        LCDWriteIntXY(9, 0, materialCounts[2].sortCount,2);
        LCDWriteStringXY(12, 0, "A");
        LCDWriteIntXY(13, 0, materialCounts[3].sortCount,2);
        LCDWriteStringXY(0,1, "SCANNED:");
        LCDWriteIntXY(8, 1, scannedCount, 2);
    }//if

while((PIND&0x02)==0x02);
mTimer(25);

STATE = 0;
goto POLLING_STAGE;
```

*Figure 11 Code for Pause Stage*

The paused stage is triggered when the pause button is pressed, activating an interrupt that changes the state of the paused flag. If the system is currently active (paused = 0), the ISR sets the paused flag to 1. If the system is already paused, the ISR toggles the paused flag to 0.

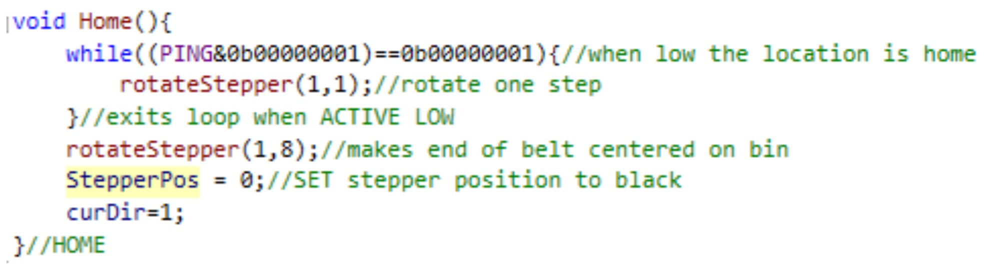
If the paused flag is set to 0, the DC motor is stopped, and the system continuously displays the number of partially sorted and fully sorted items on the LCD. This display continues until the paused flag is reset to 0. Once the while loop is exited, the conveyor belt is restarted, and the system returns to the polling state by setting the system state to 0.

## 2.3.2 Functions

This section provides a comprehensive description of all the functions implemented in the state machine.

### *2.3.2.1 Home()*

Figure 12 contains code for the home function



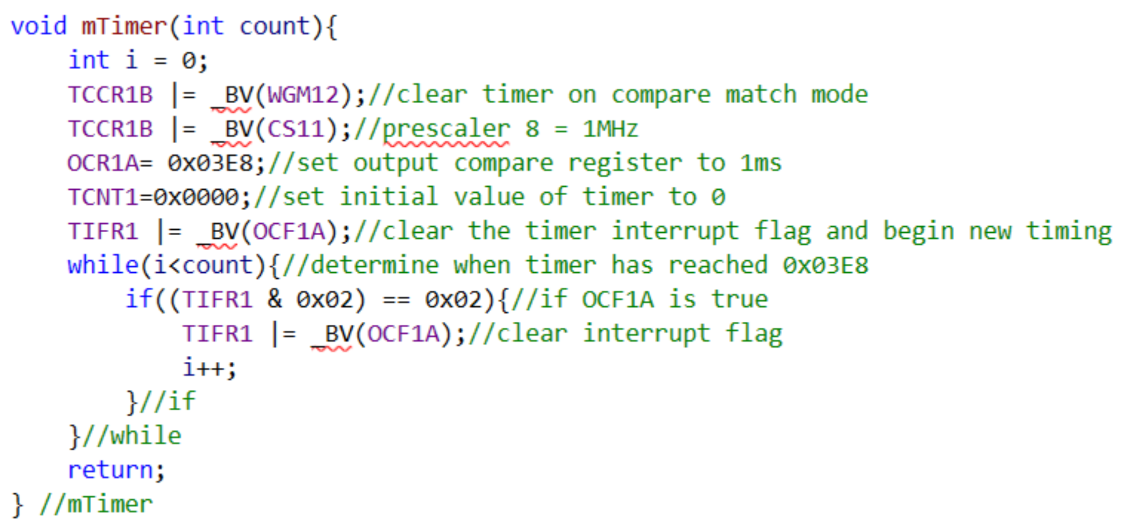
```
void Home(){
    while((PING&0b00000001)==0b00000001){//when low the location is home
        rotateStepper(1,1);//rotate one step
    }//exits loop when ACTIVE LOW
    rotateStepper(1,8);//makes end of belt centered on bin
    StepperPos = 0;//SET stepper position to black
    curDir=1;
}//HOME
```

*Figure 12 Home Function Code*

The home function is used to reset the stepper motor to its home position. The loop checks if the Hall Effect sensor is high, indicating the home position has not been reached. While the location is not home, it rotates the stepper motor by one step in the CCW direction until the home position is reached. Once the home position is reached, the stepper then rotates another 8 steps to center the end of the conveyor belt on the bin, and stepperPos is set to 0 indicating the black bin is at the end of the conveyor belt.

### *2.3.2.2 mTimer(int count)*

The code for the standard timer function is shown in Figure 13.



```
void mTimer(int count){
    int i = 0;
    TCCR1B |= _BV(WGM12);//clear timer on compare match mode
    TCCR1B |= _BV(CS11);//prescaler 8 = 1MHz
    OCR1A= 0x03E8;//set output compare register to 1ms
    TCNT1=0x0000;//set initial value of timer to 0
    TIFR1 |= _BV(OCF1A);//clear the timer interrupt flag and begin new timing
    while(i<count){//determine when timer has reached 0x03E8
        if((TIFR1 & 0x02) == 0x02){//if OCF1A is true
            TIFR1 |= _BV(OCF1A);//clear interrupt flag
            i++;
        }//if
    }//while
    return;
} //mTimer
```

*Figure 13 code for mTimer*

The mTimer function is the standard timer used throughout the project and is used to occupy the processor in intervals of milliseconds which is controlled by the input argument “count”. The timer prescaler is set to 8 so the timer frequency is $f_{timer} = \frac{8[MHz]}{8} = 1[MHz]$. Furthermore, the value stored in OCR1A is equal to 1000, resulting in a timer length of $t_{timer} = \frac{1000}{1[MHz]} = 0.001s$. The while loop executes continuously until the timer has reached the value stored in OCR1A (ie one millisecond has passed). Once reached, the interrupt flag is cleared, and the loop variable is incremented. Therefore, the period occupied by the processor can be controlled using the variable “count”.

#### *2.3.2.3 PWM()*

Figure 14 contains code for the PWM functionality.

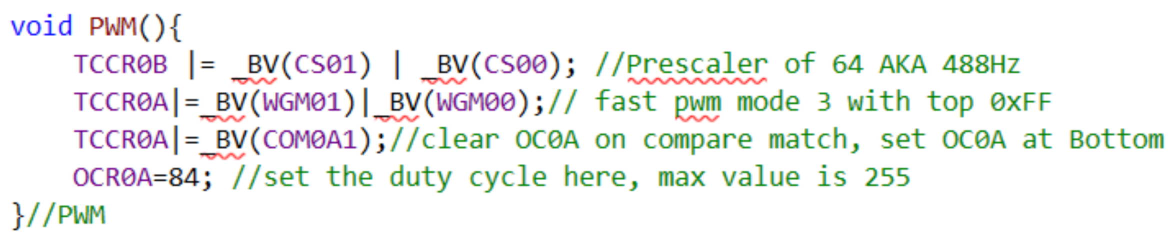

```
void PWM(){
    TCCR0B |= _BV(CS01) | _BV(CS00); //Prescaler of 64 AKA 488Hz
    TCCR0A|=_BV(WGM01)|_BV(WGM00);// fast pwm mode 3 with top 0xFF
    TCCR0A|=_BV(COM0A1);//clear OC0A on compare match, set OC0A at Bottom
    OCR0A=84; //set the duty cycle here, max value is 255
}//PWM
```

*Figure 14 Code for PWM*

The PWM function was designed to keep the main function of the project clean and organized. It simplifies the process of setting up the required registers by setting the necessary bits, allowing the main function to focus on higher-level logic. To control the DC motors using PWM, the function configures the system to operate in fast PWM mode with a top value of 0xFF (the maximum value for the register). The PWM output pin (OC0A) is cleared on a compare match and is set to 0x00 at the bottom, enabling efficient adjustment of the PWM duty cycle by modifying the value stored in the OCR0A register. Furthermore, a pre-scaler of 8 is used such that the frequency of the PWM is calculated to be:

$$f_{PWM} = \frac{f_{clk}}{N * 256} = \frac{8[MHz]}{64 * 256} = 488Hz$$

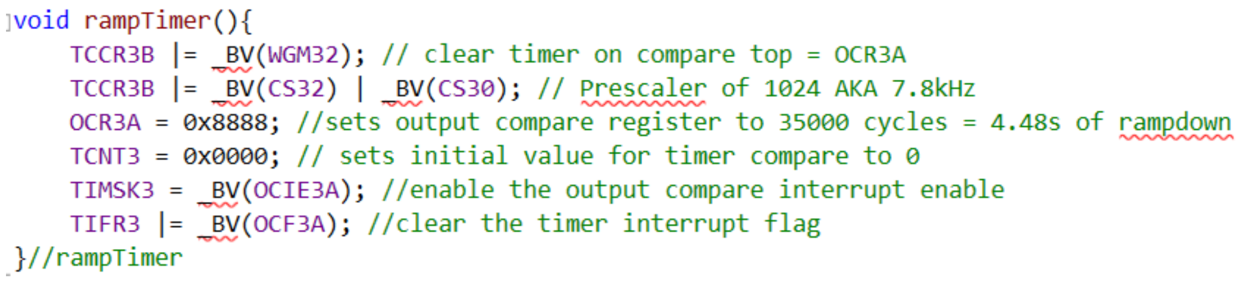
As was specified by the lab technician. Lastly, given that the period, T, of the PWM is 2.05ms, the duty cycle of the PWM can be calculated using:

$$DC = \frac{t}{T}$$

Where t is the time taken to reach the value stored in OCR0A. Therefore, for a required duty cycle, the value required to be set in OCR0A can simply be calculated as t = DC * T.

### 2.3.2.4 rampTimer()

Figure 15 contains code for the rampTimer



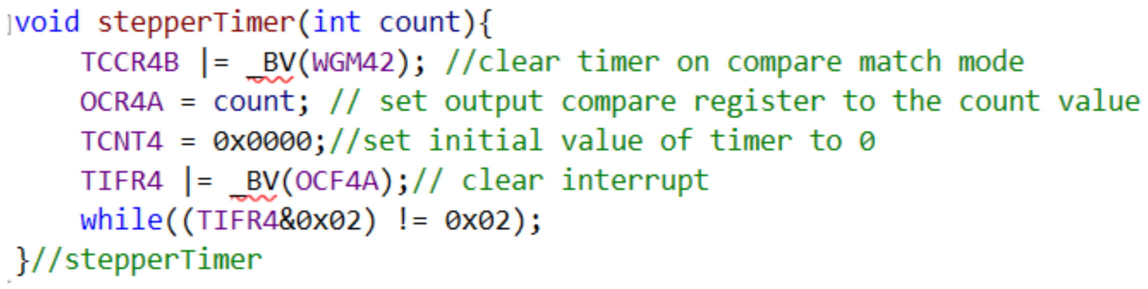
```
void rampTimer(){
    TCCR3B |= _BV(WGM32); // clear timer on compare top = OCR3A
    TCCR3B |= _BV(CS32) | _BV(CS30); // Prescaler of 1024 AKA 7.8kHz
    OCR3A = 0x8888; //sets output compare register to 35000 cycles = 4.48s of rampdown
    TCNT3 = 0x0000; // sets initial value for timer compare to 0
    TIMSK3 = _BV(OCIE3A); //enable the output compare interrupt enable
    TIFR3 |= _BV(OCF3A); //clear the timer interrupt flag
}//rampTimer
```

*Figure 15 Code for rampTimer*

The rampTimer function is called from the ramp-down button ISR and initializes a timer which throws an interrupt upon reaching the value stored in OCR3A. The timer frequency is set using a prescaler of 1024 and thus is $f_{timer} = \frac{8[MHz]}{1024} = 7.8125[kHz]$. The value stored in OCR3A is set to 35000, resulting in a timer length of $t_{timer} = \frac{35000}{7.8125[kHz]} = 4.48[s]$. When the timer reaches the value stored in OCR3A, the output compare interrupt is thrown (TIMER3_COMPA_vect) and is handled by its respective ISR.

### 2.3.2.5 stepperTimer(int count)

Figure 16 contains code for the stepperTimer



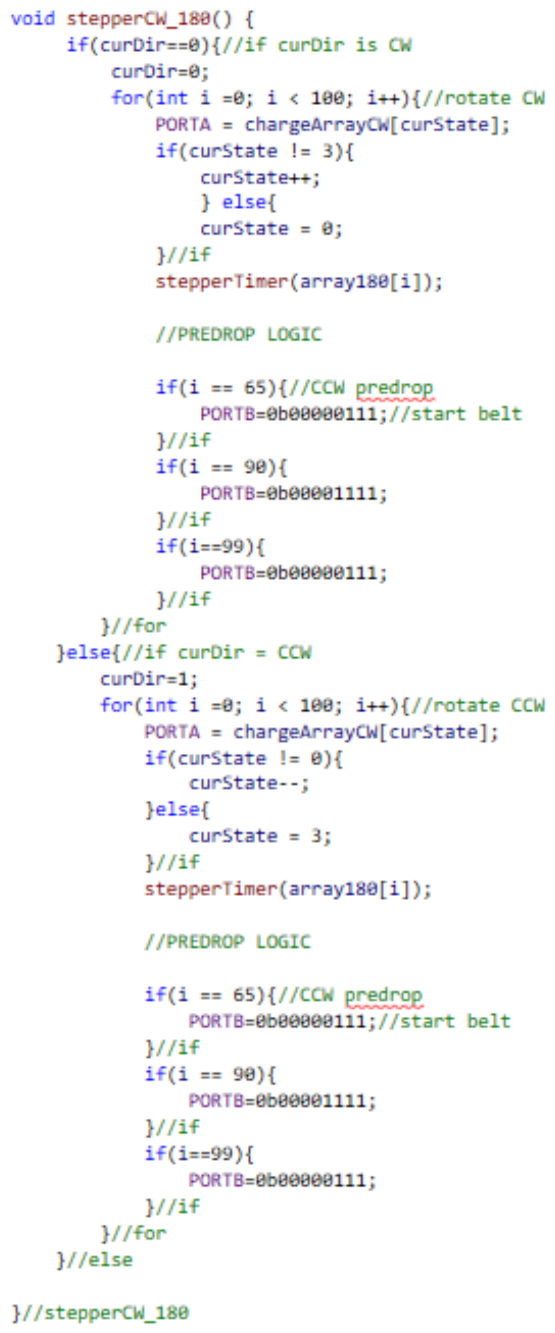
```
void stepperTimer(int count){
    TCCR4B |= _BV(WGM42); //clear timer on compare match mode
    OCR4A = count; // set output compare register to the count value
    TCNT4 = 0x0000;//set initial value of timer to 0
    TIFR4 |= _BV(OCF4A);// clear interrupt
    while((TIFR4&0x02) != 0x02);
}//stepperTimer
```

*Figure 16 Code for stepperTimer*

The stepperTimer function configures and uses Timer/Counter 4 to create a time delay, which is typically used for controlling the stepper motor's timing. First, it sets the timer to operate in "Clear Timer on Compare Match" (CTC) mode by configuring the relevant bits in the Timer/Counter Control Register 4 (TCCR4B). This mode ensures that the timer resets to zero each time it reaches a value specified in the output compare register, OCR4A. The function then loads the value of count into OCR4A, determining the point at which the timer will trigger an interrupt. The timer counter is reset to zero using TCNT4 set to 0x0000, ensuring the timer starts from the beginning. To ensure that any previous interrupt flags are cleared, the function sets the interrupt flag for the compare match in the Timer/Counter Interrupt Flag Register (TIFR4). It then enters a loop, constantly checking the interrupt flag. The loop will continue until the compare match interrupt flag is set, indicating that the timer has reached the value specified in OCR4A. Once the flag is set, the function exits the loop and completes its execution. This setup allows for precise control over the timing of the stepper motor's movements, based on the count value.

### 2.3.2.6 stepper_180()

Figure 17 contains code used to rotate the stepper motor 180 degrees.



```
void stepperCW_180() {
     if(curDir==0){//if curDir is CW
         curDir=0;
         for(int i =0; i < 100; i++){//rotate CW
             PORTA = chargeArrayCW[curState];
             if(curState != 3){
                 curState++;
                 } else{
                 curState = 0;
             }//if
             stepperTimer(array180[i]);

             //PREDROP LOGIC

             if(i == 65){//CCW predrop
                 PORTB=0b00000111;//start belt
             }//if
             if(i == 90){
                 PORTB=0b00001111;
             }//if
             if(i==99){
                 PORTB=0b00000111;
             }//if
        }//for
    }else{//if curDir = CCW
        curDir=1;
        for(int i =0; i < 100; i++){//rotate CCW
            PORTA = chargeArrayCW[curState];
            if(curState != 0){
                curState--;
            }else{
                curState = 3;
            }//if
            stepperTimer(array180[i]);

            //PREDROP LOGIC

            if(i == 65){//CCW predrop
                PORTB=0b00000111;//start belt
            }//if
            if(i == 90){
                PORTB=0b00001111;
            }//if
            if(i==99){
                PORTB=0b00000111;
            }//if
        }//for
    }//else

}//stepperCW_180
```

*Figure 17 Code for stepper180*

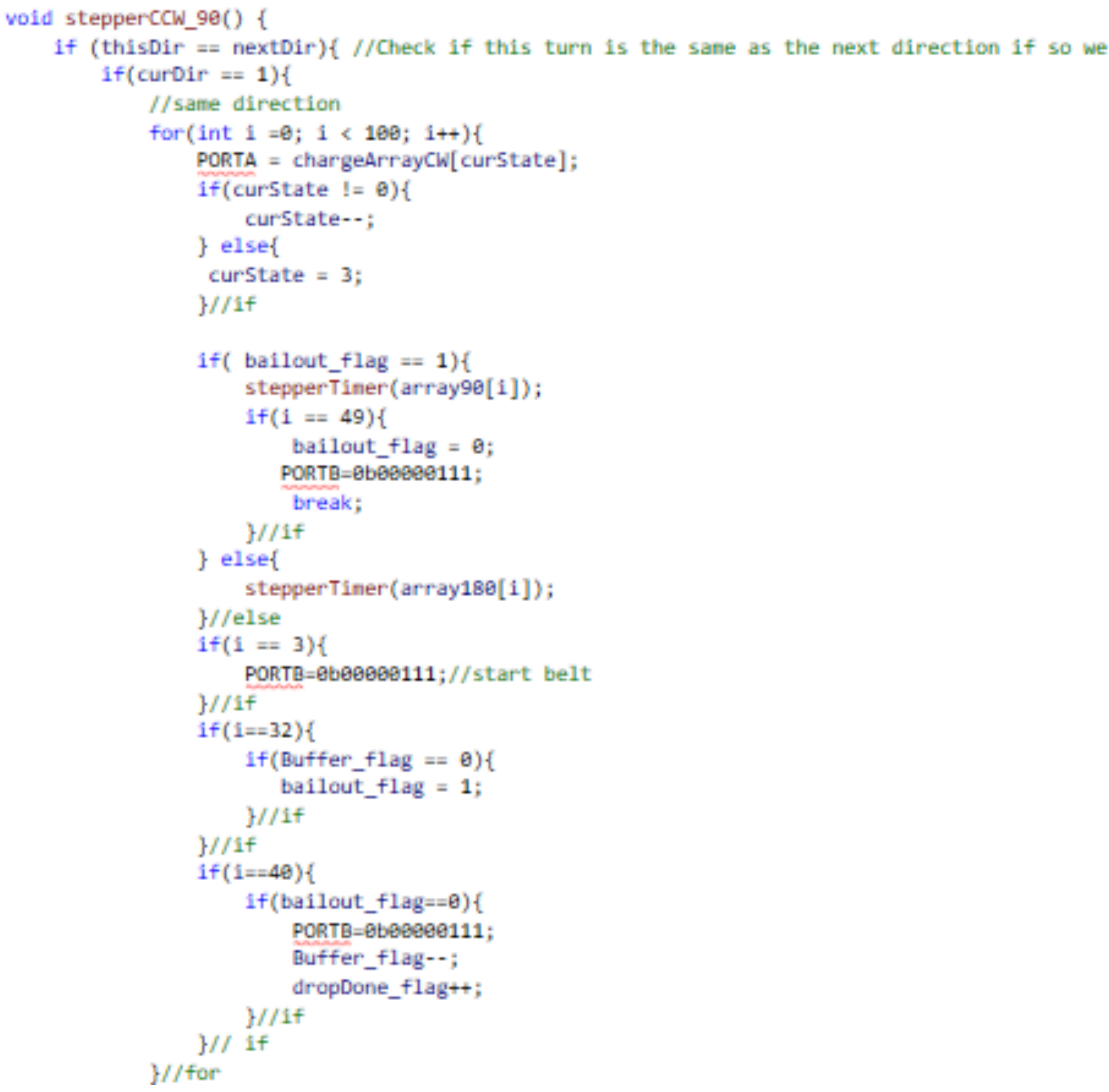
This function iterates for the specified number of steps in either positive or negative increments, depending on the direction set by the global variable curDir. The iteration takes place through both the motor control array and the motor acceleration array. The global variable curState tracks the index of the motor control array, ensuring that the correct step is output to PORTA for the motor. The motor control array is designed for full-step control, where two adjacent coils are energized for each step. This method increases the torque compared to the wave-drive method used in Lab 4.

The variable i tracks the number of steps the stepper motor has completed during the rotation and is used to determine the relative position of the tray. A pre-drop feature was introduced in which pieces are dropped when the stepper motor has traversed a certain number of steps, allowing the motor to reach the correct position just as the piece is placed on the sorting tray. This feature significantly improves the system's speed and efficiency.

Additionally, the acceleration array provides pre-calculated delays between each step of the motor, which are passed as inputs to the stepperTimer() function. The acceleration profile used is an s-curve, which was developed through extensive testing. The details of this testing and the resulting acceleration curve will be discussed in Section 5.

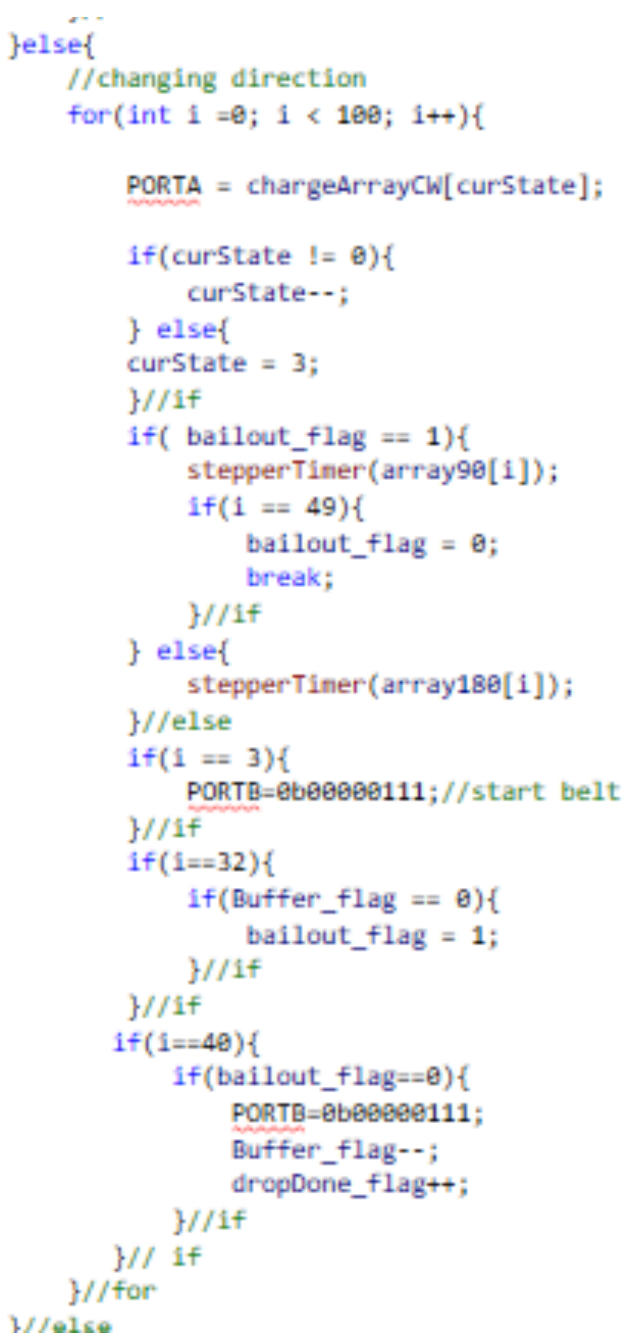
#### *2.3.2.7 stepperCCW_90()*

Figures 18-20 contain the code used to rotate the stepper CCW 90 degrees.



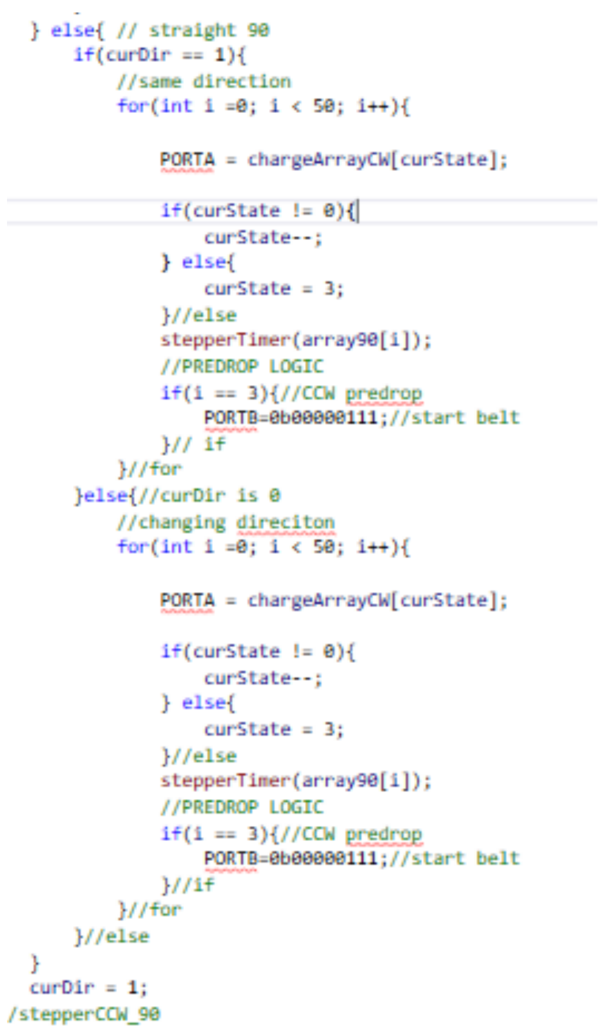
```
void stepperCCW_90() {
    if (thisDir == nextDir){ //Check if this turn is the same as the next direction if so we
        if(curDir == 1){
            //same direction
            for(int i =0; i < 100; i++){
                PORTA = chargeArrayCW[curState];
                if(curState != 0){
                    curState--;
                } else{
                 curState = 3;
                }//if

                if( bailout_flag == 1){
                    stepperTimer(array90[i]);
                    if(i == 49){
                        bailout_flag = 0;
                       PORTB=0b00000111;
                        break;
                    }//if
                } else{
                    stepperTimer(array180[i]);
                }//else
                if(i == 3){
                    PORTB=0b00000111;//start belt
                }//if
                if(i==32){
                    if(Buffer_flag == 0){
                       bailout_flag = 1;
                    }//if
                }//if
                if(i==40){
                    if(bailout_flag==0){
                        PORTB=0b00000111;
                        Buffer_flag--;
                        dropDone_flag++;
                    }//if
                }// if
            }//for
```

*Figure 18 Code for stepperCCW_90 1*



```
}else{
    //changing direction
    for(int i =0; i < 100; i++){

        PORTA = chargeArrayCW[curState];

        if(curState != 0){
            curState--;
        } else{
        curState = 3;
        }//if
        if( bailout_flag == 1){
            stepperTimer(array90[i]);
            if(i == 49){
                bailout_flag = 0;
                break;
            }//if
        } else{
            stepperTimer(array180[i]);
        }//else
        if(i == 3){
            PORTB=0b00000111;//start belt
        }//if
        if(i==32){
            if(Buffer_flag == 0){
                bailout_flag = 1;
            }//if
        }//if
       if(i==40){
           if(bailout_flag==0){
               PORTB=0b00000111;
               Buffer_flag--;
               dropDone_flag++;
           }//if
       }// if
    }//for
}//else
```

*Figure 19 Code for stepperCCW_90 2*



```
} else{ // straight 90
    if(curDir == 1){
        //same direction
        for(int i =0; i < 50; i++){

            PORTA = chargeArrayCW[curState];

            if(curState != 0){
                curState--;
            } else{
                curState = 3;
            }//else
            stepperTimer(array90[i]);
            //PREDROP LOGIC
            if(i == 3){//CCW predrop
                PORTB=0b00000111;//start belt
            }// if
        }//for
    }else{//curDir is 0
        //changing direciton
        for(int i =0; i < 50; i++){

            PORTA = chargeArrayCW[curState];

            if(curState != 0){
                curState--;
            } else{
                curState = 3;
            }//else
            stepperTimer(array90[i]);
            //PREDROP LOGIC
            if(i == 3){//CCW predrop
                PORTB=0b00000111;//start belt
            }//if
        }//for
    }//else
}
curDir = 1;
/stepperCCW_90
```

*Figure 20 Code for stepperCCW_90 3*

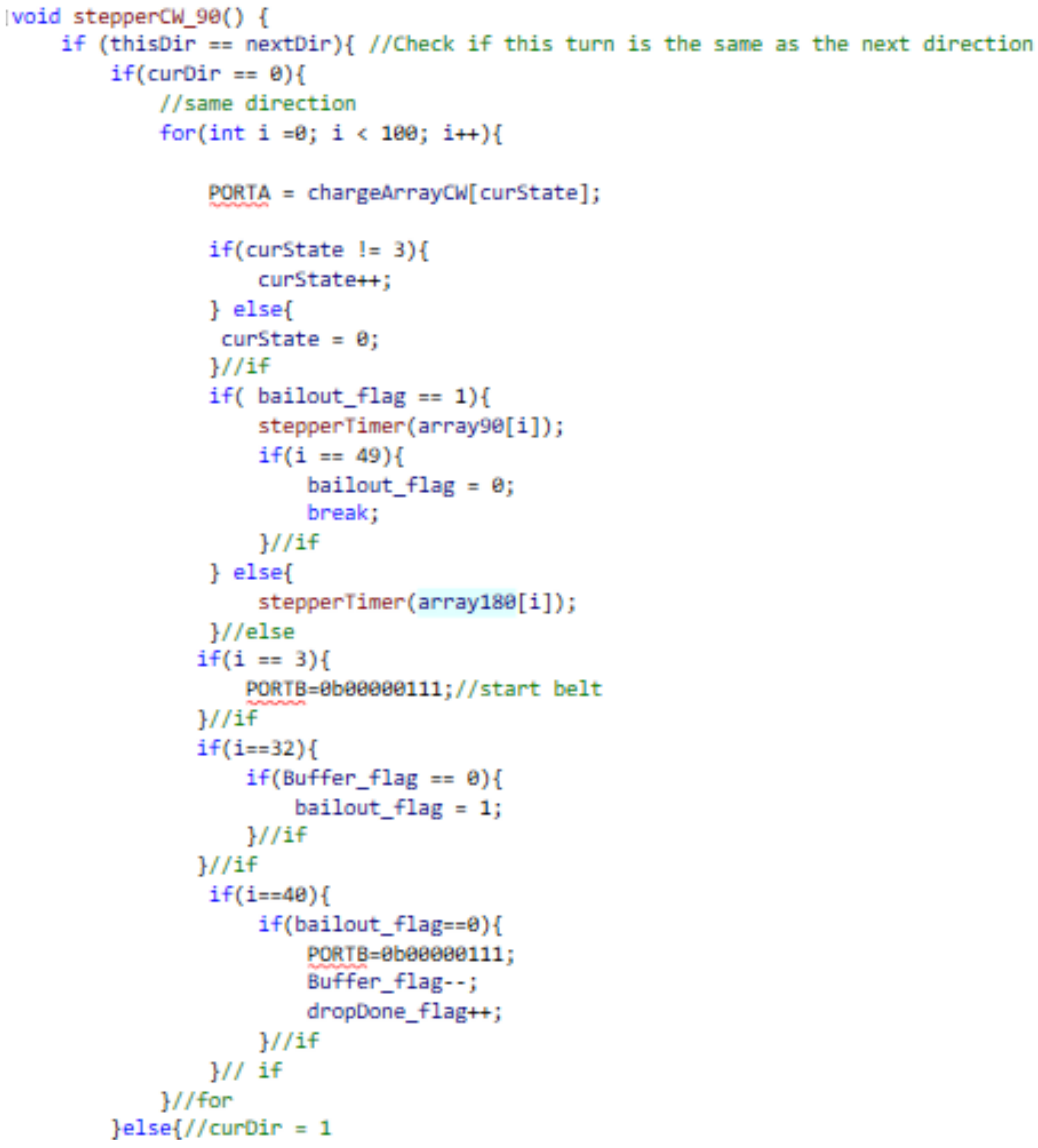
This function iterates for the specified number of steps in negative increments. The iteration takes place through both the motor control array and the motor acceleration array. The global variable curState tracks the index of the motor control array, ensuring that the correct step is output to PORTA for the motor. The motor control array is designed for full-step control, where two adjacent coils are energized for each step. This method increases the torque compared to the wave-drive method used in Lab 4.

It first checks if the direction of rotation (thisDir) matches the next direction of rotation (nextDir). If they are the same, a 100 step, 180 degree rotation can be initiated to rotate to the second part while kicking off the first into its bucket. Timing is managed using either array90 or array180, depending on the state of bailout_Flag, which controls whether the loop exits early at iteration 49. Additional operations also include starting the conveyor belt at iteration 3, allowing the first part time to drop into the center of its bucket as the tray rotates. At iteration 32, Buffer_flag is checked, if it is a 1 then something is at the exit sensor and the double drop can continue, if 0 then the gap between pieces is determined to be too large and the bailout_Flag is set to 1 which indicates that the continuous rotation must be aborted. If no bailout occurs, the conveyor belt is started on iteration 40 to allow the second part to drop, additionally the dropDone_flag is set indicating a second drop has been made requiring an additional dequeue from the linked list. If thisDir did not match nextDir, the function executes a simpler 50 step iteration loop for a direct 90 degree rotation. The specifics of the continuous rotation algorithm are further discussed in section 3.1.

Additionally, the acceleration array provides pre-calculated delays between each step of the motor, which are passed as inputs to the stepperTimer() function. The acceleration profile used is an s-curve, which was developed through extensive testing. The details of this testing and the resulting acceleration curve will be discussed in Section 5.

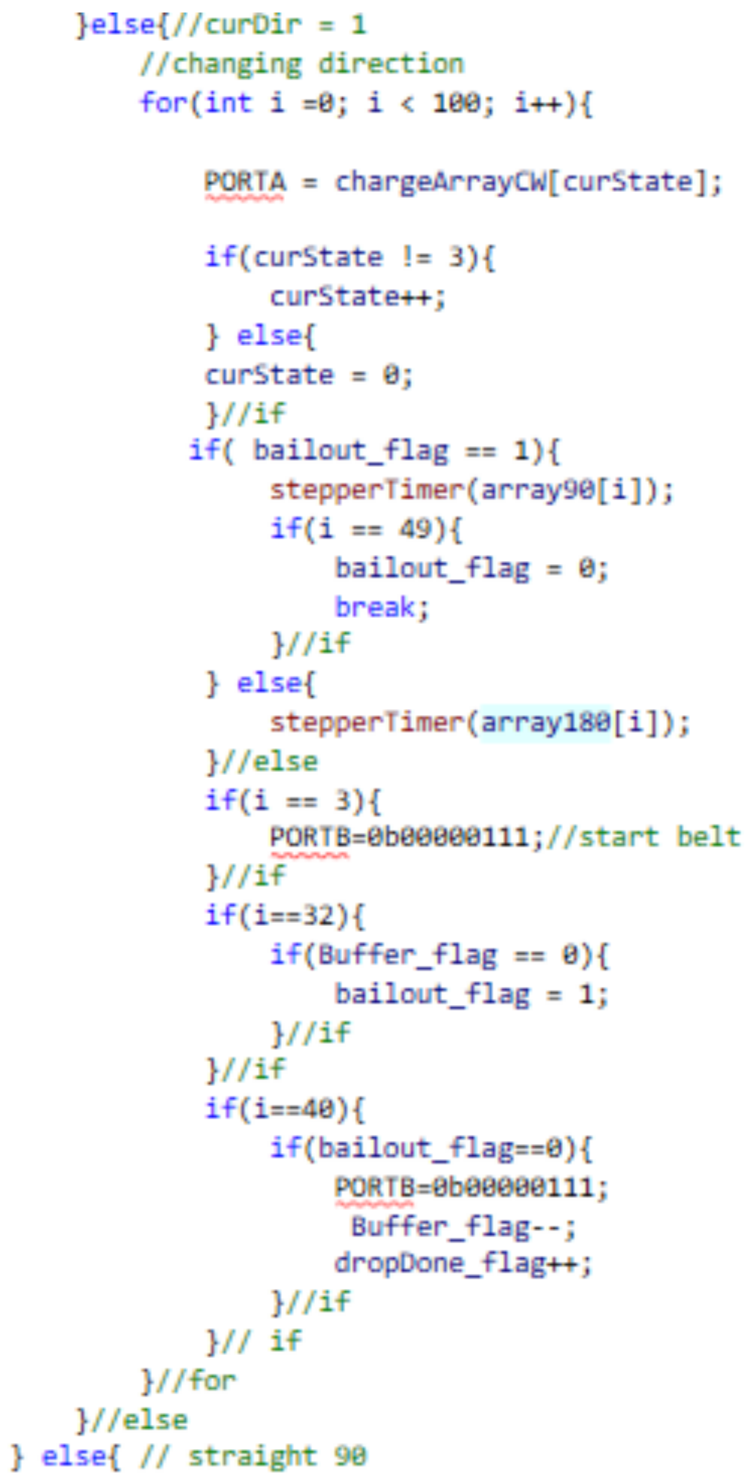
#### *2.3.2.8 stepperCW_90()*

Figure 21-23 contains the code used to rotate the stepper CW 90 degrees.



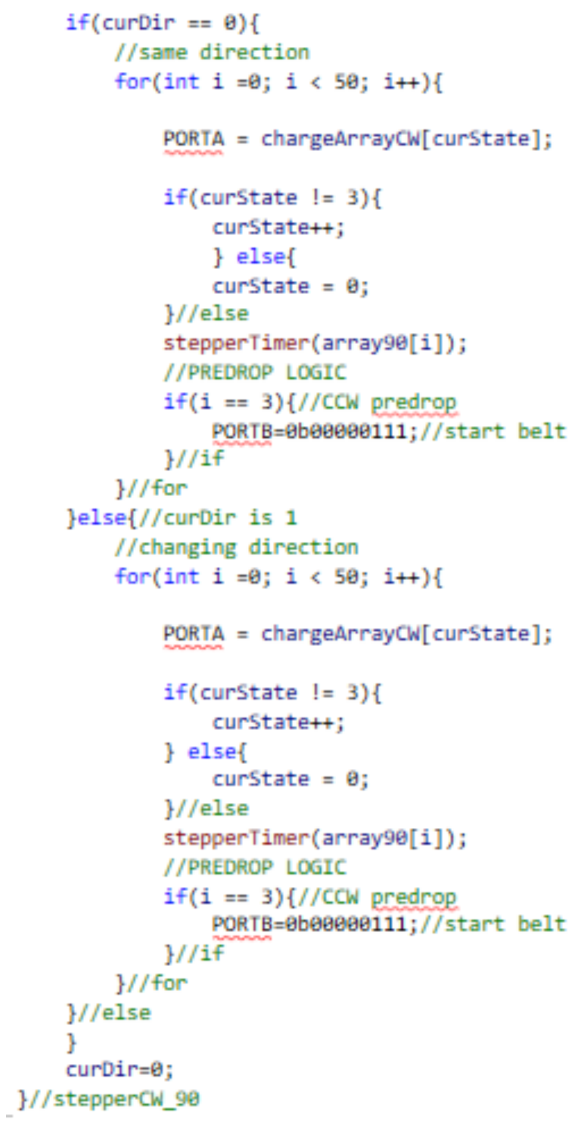
```
void stepperCW_90() {
    if (thisDir == nextDir){ //Check if this turn is the same as the next direction
        if(curDir == 0){
            //same direction
            for(int i =0; i < 100; i++){

                PORTA = chargeArrayCW[curState];

                if(curState != 3){
                    curState++;
                } else{
                 curState = 0;
                }//if
                if( bailout_flag == 1){
                    stepperTimer(array90[i]);
                    if(i == 49){
                        bailout_flag = 0;
                        break;
                    }//if
                } else{
                    stepperTimer(array180[i]);
                }//else
               if(i == 3){
                   PORTB=0b00000111;//start belt
               }//if
               if(i==32){
                   if(Buffer_flag == 0){
                       bailout_flag = 1;
                   }//if
               }//if
                if(i==40){
                    if(bailout_flag==0){
                        PORTB=0b00000111;
                        Buffer_flag--;
                        dropDone_flag++;
                    }//if
                }// if
            }//for
        }else{//curDir = 1
```

*Figure 21 Code for StepperCW_90 1*



```
    }else{//curDir = 1
        //changing direction
        for(int i =0; i < 100; i++){

            PORTA = chargeArrayCW[curState];

            if(curState != 3){
                curState++;
            } else{
            curState = 0;
            }//if
           if( bailout_flag == 1){
                stepperTimer(array90[i]);
                if(i == 49){
                    bailout_flag = 0;
                    break;
                }//if
            } else{
                stepperTimer(array180[i]);
            }//else
            if(i == 3){
                PORTB=0b00000111;//start belt
            }//if
            if(i==32){
                if(Buffer_flag == 0){
                    bailout_flag = 1;
                }//if
            }//if
            if(i==40){
                if(bailout_flag==0){
                    PORTB=0b00000111;
                     Buffer_flag--;
                    dropDone_flag++;
                }//if
            }// if
        }//for
    }//else
} else{ // straight 90
```

*Figure 22 Code for StepperCW_90 2*



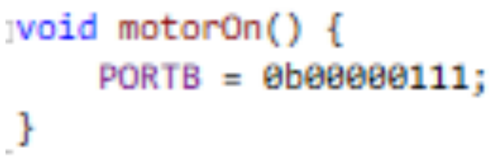
```
    if(curDir == 0){
        //same direction
        for(int i =0; i < 50; i++){

            PORTA = chargeArrayCW[curState];

            if(curState != 3){
                curState++;
                } else{
                curState = 0;
            }//else
            stepperTimer(array90[i]);
            //PREDROP LOGIC
            if(i == 3){//CCW predrop
                PORTB=0b00000111;//start belt
            }//if
        }//for
    }else{//curDir is 1
        //changing direction
        for(int i =0; i < 50; i++){

            PORTA = chargeArrayCW[curState];

            if(curState != 3){
                curState++;
            } else{
                curState = 0;
            }//else
            stepperTimer(array90[i]);
            //PREDROP LOGIC
            if(i == 3){//CCW predrop
                PORTB=0b00000111;//start belt
            }//if
        }//for
    }//else
    }
    curDir=0;
}//stepperCW_90
```

*Figure 23 Code for StepperCW_90 3*

The only variation from the counter-clockwise stepper function is that this function decrements the curState variable such that the motor control array is traversed positively, resulting in a clockwise motion. All other logic remains identical and thus will not be re-stated.

### *2.3.2.9 motorOn()*

Figure 24 contains the code used to turn the DC motor on.



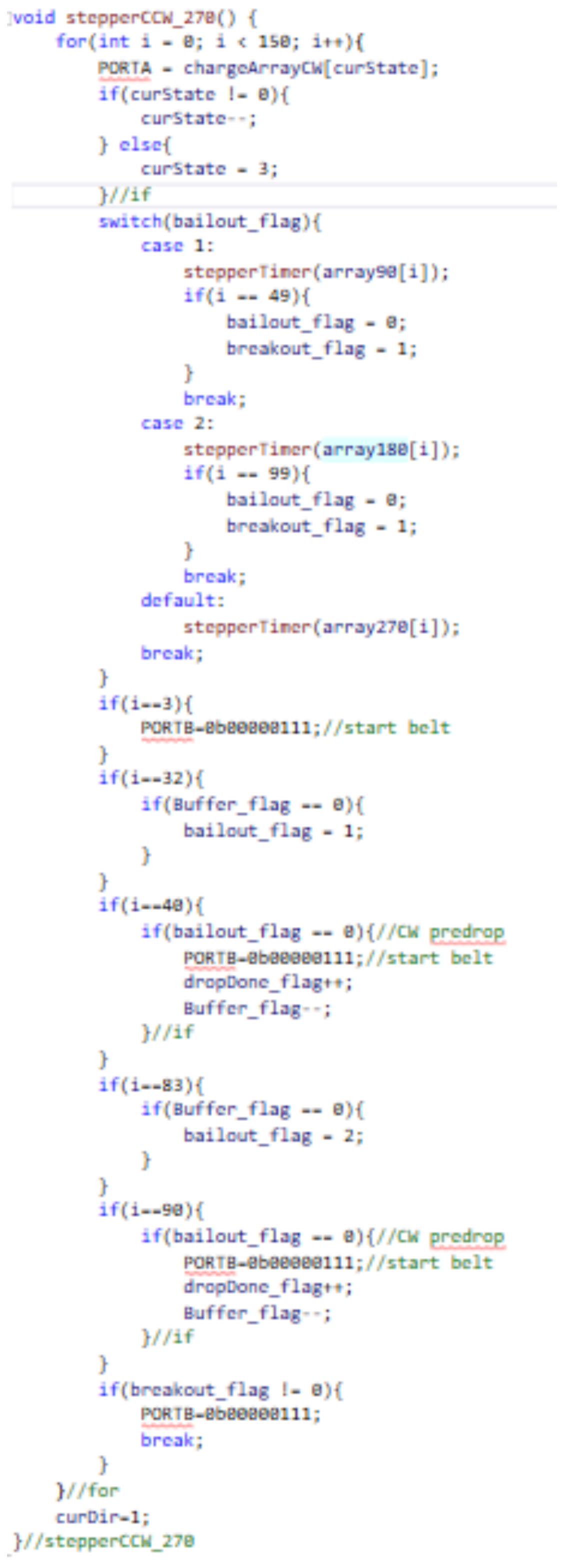
```
void motorOn() {
    PORTB = 0b00000111;
}
```

*Figure 24 Code for motorOn*

This function is designed to simply turn the conveyor belt on if 2 pieces of the same type are one after the other. It is one of the items in the stepperActions look up table and the program is sent here if the Exit_ID matches the current stepper position.

### *2.3.2.10 stepperCCW_270()*

Figure 25 contains the code used to rotate the stepperCCW_270 degrees.



```
void stepperCCW_270() {
    for(int i = 0; i < 150; i++){
        PORTA = chargeArrayCW[curState];
        if(curState != 0){
            curState--;
        } else{
            curState = 3;
        }//if
        switch(bailout_flag){
            case 1:
                stepperTimer(array90[i]);
                if(i == 49){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }
                break;
            case 2:
                stepperTimer(array180[i]);
                if(i == 99){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }
                break;
            default:
                stepperTimer(array270[i]);
            break;
        }
        if(i==3){
            PORTB=0b00000111;//start belt
        }
        if(i==32){
            if(Buffer_flag == 0){
                bailout_flag = 1;
            }
        }
        if(i==40){
            if(bailout_flag == 0){//CW predrop
                PORTB=0b00000111;//start belt
                dropDone_flag++;
                Buffer_flag--;
            }//if
        }
        if(i==83){
            if(Buffer_flag == 0){
                bailout_flag = 2;
            }
        }
        if(i==90){
            if(bailout_flag == 0){//CW predrop
                PORTB=0b00000111;//start belt
                dropDone_flag++;
                Buffer_flag--;
            }//if
        }
        if(breakout_flag != 0){
            PORTB=0b00000111;
            break;
        }
    }//for
    curDir=1;
}//stepperCCW_270
```

*Figure 25 Code for stepperCCW_270*

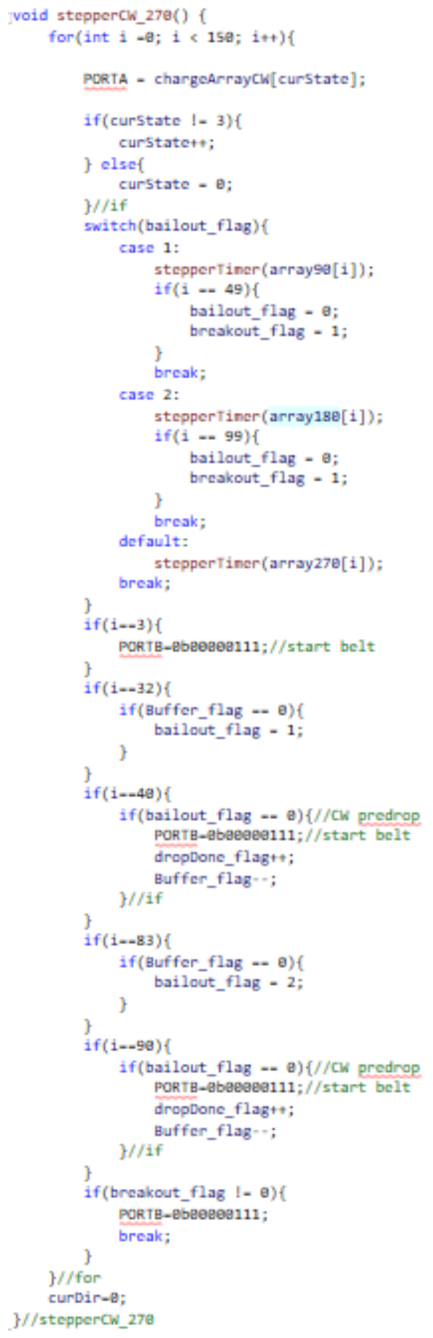
This function iterates for the specified number of steps in negative increments. The iteration takes place through both the motor control array and the motor acceleration array. The global variable curState tracks the index of the motor control array, ensuring that the correct step is output to PORTA for the motor. The motor control array is designed for full-step control, where two adjacent coils are energized for each step. This method increases the torque compared to the wave-drive method used in Lab 4.

The switch statement handles timing based on the value of the bailout_flag. The bailout_flag is designed to be triggered in the event that the spacing between pieces is too large, which would cause an error on the drop, the same as discussed in the 90 degree rotation sction. After dropping the first piece, if by step 32 the buffer_flag has not been triggered, indicating the following piece is at the exit sensor, the bailout _flag is set to 1, the stepperTimer switches to array90 and only completes the 90 degree rotation before bailing out. The same process follows on step 82, if buffer_flag is 0, bailout_flag is set to 2 and the stepperTimer array is set to array180 indicating the sequence must bail out after the 180 degree rotation. If no bailout_flag is set, the stepper will complete the full 270 degree rotation, dropping all 3 pieces.

After completing the drop of pieces on iteration 40 and 90, the dropDone_flag is increased, allowing the pieces to be dequeued from the linked list following the completion of the rotation, ensuring the queue remains intact.

### *2.3.2.11 stepperCW_270()*

Figure 26 contains the code used to rotate the stepperCW_270 degrees.



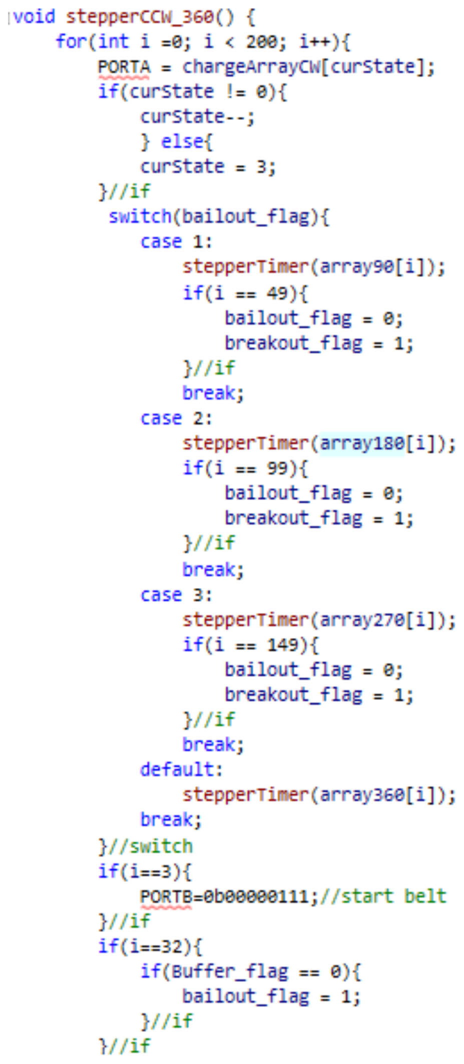
```c
void stepperCW_270() {
    for(int i =0; i < 150; i++){

        PORTA = chargeArrayCW[curState];

        if(curState != 3){
            curState++;
        } else{
            curState = 0;
        }//if
        switch(bailout_flag){
            case 1:
                stepperTimer(array90[i]);
                if(i == 49){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }
                break;
            case 2:
                stepperTimer(array180[i]);
                if(i == 99){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }
                break;
            default:
                stepperTimer(array270[i]);
            break;
        }
        if(i==3){
            PORTB=0b00000111;//start belt
        }
        if(i==32){
            if(Buffer_flag == 0){
                bailout_flag = 1;
            }
        }
        if(i==40){
            if(bailout_flag == 0){//CW predrop
                PORTB=0b00000111;//start belt
                dropDone_flag++;
                Buffer_flag--;
            }//if
        }
        if(i==83){
            if(Buffer_flag == 0){
                bailout_flag = 2;
            }
        }
        if(i==90){
            if(bailout_flag == 0){//CW predrop
                PORTB=0b00000111;//start belt
                dropDone_flag++;
                Buffer_flag--;
            }//if
        }
        if(breakout_flag != 0){
            PORTB=0b00000111;
            break;
        }
    }//for
    curDir=0;
}//stepperCW_270
```

*Figure 26 Code for stepperCW_270*

The only variation from the counter-clockwise stepper function is that this function decrements the curState variable such that the motor control array is traversed positively, resulting in a clockwise motion. All other logic remains identical and thus will not be re-stated.

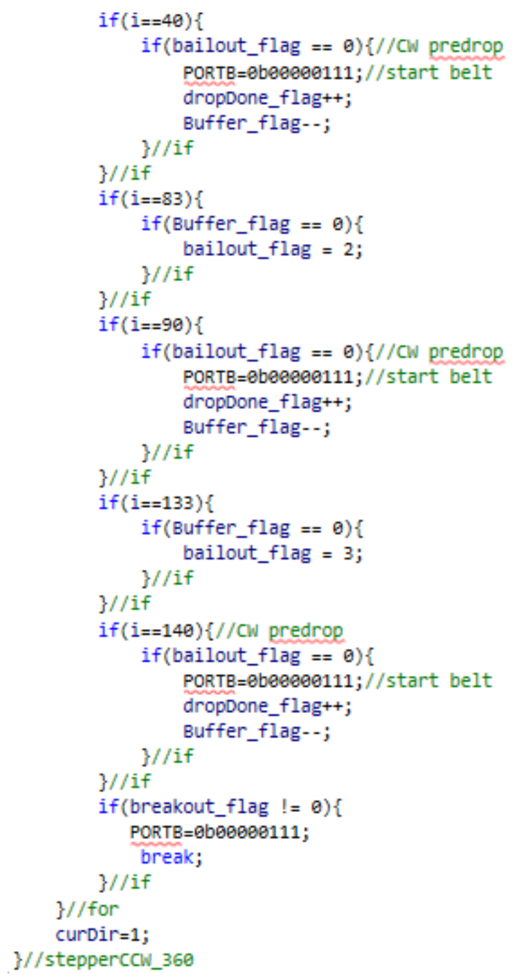
### *2.3.2.12 stepperCCW_360()*

Figures 27 and 28 contain the code used to rotate the stepperCCW_360 degrees.



```
void stepperCCW_360() {
    for(int i =0; i < 200; i++){
        PORTA = chargeArrayCW[curState];
        if(curState != 0){
            curState--;
            } else{
            curState = 3;
        }//if
         switch(bailout_flag){
            case 1:
                stepperTimer(array90[i]);
                if(i == 49){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }//if
                break;
            case 2:
                stepperTimer(array180[i]);
                if(i == 99){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }//if
                break;
            case 3:
                stepperTimer(array270[i]);
                if(i == 149){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }//if
                break;
            default:
                stepperTimer(array360[i]);
            break;
        }//switch
        if(i==3){
            PORTB=0b00000111;//start belt
        }//if
        if(i==32){
            if(Buffer_flag == 0){
                bailout_flag = 1;
            }//if
        }//if
```

*Figure 27 Code for stepperCCW_360 1*



```
        if(i==40){
            if(bailout_flag == 0){//CW predrop
                PORTB=0b00000111;//start belt
                dropDone_flag++;
                Buffer_flag--;
            }//if
        }//if
        if(i==83){
            if(Buffer_flag == 0){
                bailout_flag = 2;
            }//if
        }//if
        if(i==90){
            if(bailout_flag == 0){//CW predrop
                PORTB=0b00000111;//start belt
                dropDone_flag++;
                Buffer_flag--;
            }//if
        }//if
        if(i==133){
            if(Buffer_flag == 0){
                bailout_flag = 3;
            }//if
        }//if
        if(i==140){//CW predrop
            if(bailout_flag == 0){
                PORTB=0b00000111;//start belt
                dropDone_flag++;
                Buffer_flag--;
            }//if
        }//if
        if(breakout_flag != 0){
           PORTB=0b00000111;
            break;
        }//if
    }//for
    curDir=1;
}//stepperCCW_360
```

*Figure 28 Code for stepperCCW_360 2*

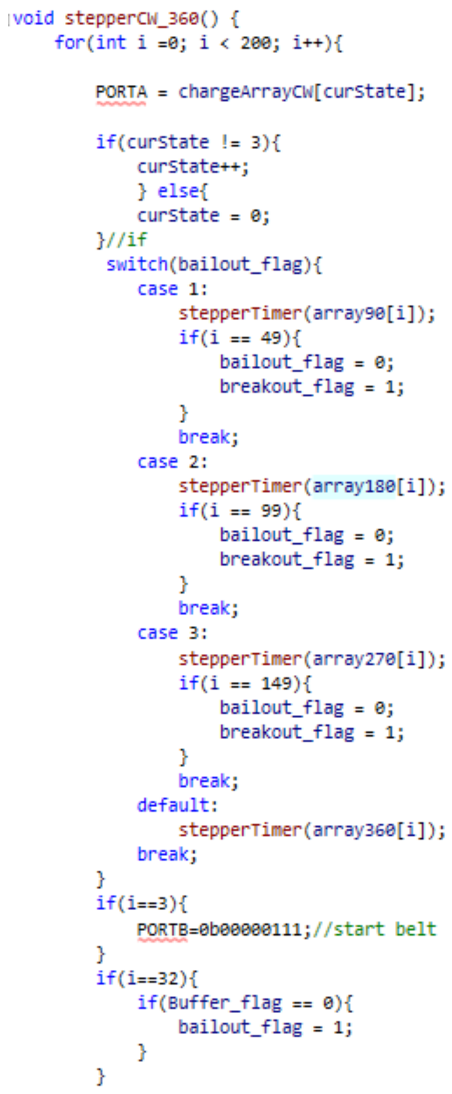
This function iterates for the specified number of steps in negative increments. The iteration takes place through both the motor control array and the motor acceleration array. The global variable curState tracks the index of the motor control array, ensuring that the correct step is output to PORTA for the motor. The motor control array is designed for full-step control, where two adjacent coils are energized for each step. This method increases the torque compared to the wave-drive method used in Lab 4.

The switch statement handles timing based on the value of the bailout_flag. The bailout_flag is designed to be triggered in the event that the spacing between pieces is too large, which would cause an error on the drop. After dropping the first piece, if by iteration 32 the buffer_flag has not been triggered, indicating the following piece is at the exit sensor, the bailout _flag is set to 1, the stepperTimer switches to array90 and only completes the 90 degree rotation before bailing out. The same process follows on iteration 82, if buffer_flag is 0, bailout_is set to 2 and the stepperTimer array is set to array180 indicating the sequence must bail out after the 180 degree rotation. Additionally, on iteration 133, if buffer_flag is 0, bailout_flag is set to 3 and the stepperTimer array is set to array270 indicating the sequence must bail out after the 270 degree rotation. If no bailout_flag is set, the stepper will complete the full 360 degree rotation, dropping all 4 pieces.

After completing the drop of pieces on iteration 40, 90 and 140 the dropDone_flag is increased, allowing the pieces to be dequeued from the linked list following the completion of the rotation, ensuring the queue remains intact.

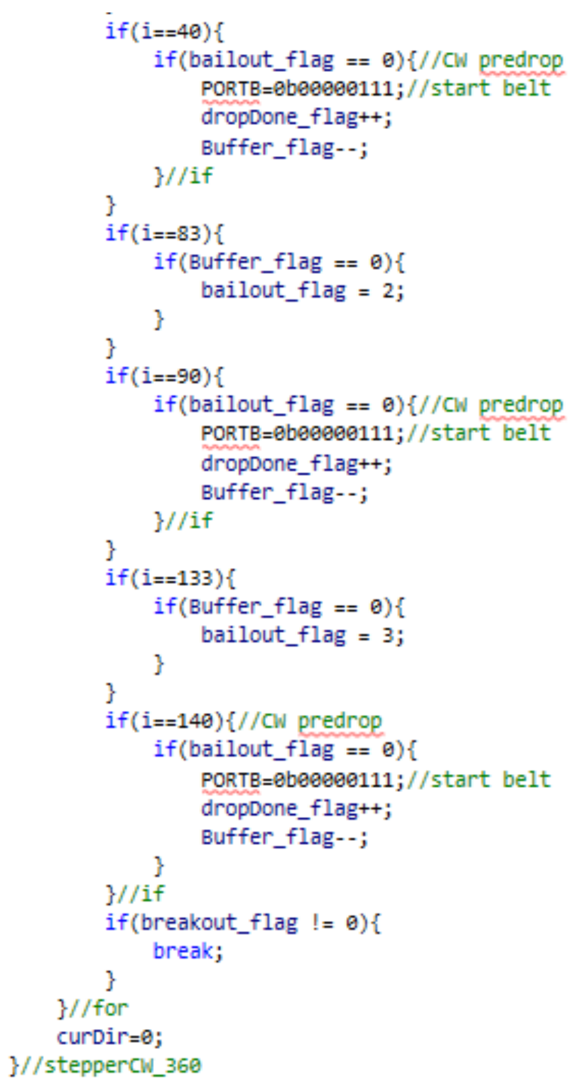
#### *2.3.2.13 stepperCW_360()*

Figures 29 and 30 contain the code used to rotate the stepperCW_360 degrees.



```
void stepperCW_360() {
    for(int i =0; i < 200; i++){

        PORTA = chargeArrayCW[curState];

        if(curState != 3){
            curState++;
            } else{
            curState = 0;
        }//if
         switch(bailout_flag){
            case 1:
                stepperTimer(array90[i]);
                if(i == 49){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }
                break;
            case 2:
                stepperTimer(array180[i]);
                if(i == 99){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }
                break;
            case 3:
                stepperTimer(array270[i]);
                if(i == 149){
                    bailout_flag = 0;
                    breakout_flag = 1;
                }
                break;
            default:
                stepperTimer(array360[i]);
            break;
        }
        if(i==3){
            PORTB=0b00000111;//start belt
        }
        if(i==32){
            if(Buffer_flag == 0){
                bailout_flag = 1;
            }
        }
```

*Figure 29 Code for stepperCW_360 1*



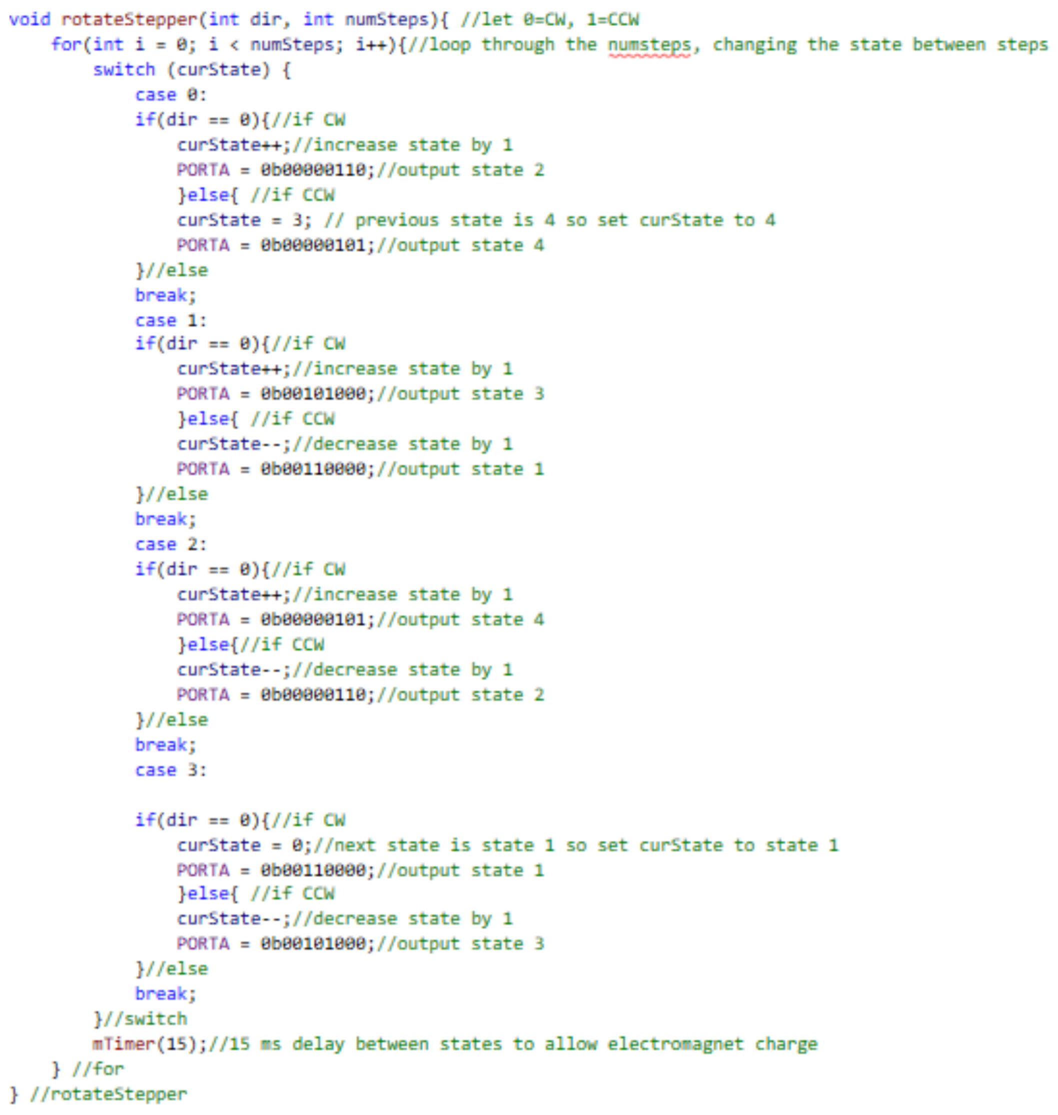
```
    if(i==40){
        if(bailout_flag == 0){//CW predrop
            PORTB=0b00000111;//start belt
            dropDone_flag++;
            Buffer_flag--;
        }//if
    }
    if(i==83){
        if(Buffer_flag == 0){
            bailout_flag = 2;
        }
    }
    if(i==90){
        if(bailout_flag == 0){//CW predrop
            PORTB=0b00000111;//start belt
            dropDone_flag++;
            Buffer_flag--;
        }//if
    }
    if(i==133){
        if(Buffer_flag == 0){
            bailout_flag = 3;
        }
    }
    if(i==140){//CW predrop
        if(bailout_flag == 0){
            PORTB=0b00000111;//start belt
            dropDone_flag++;
            Buffer_flag--;
        }
    }//if
    if(breakout_flag != 0){
        break;
    }
  }//for
  curDir=0;
}//stepperCW_360
```

*Figure 30 Code for stepperCW_360 2*

The only variation from the counter-clockwise stepper function is that this function decrements the curState variable such that the motor control array is traversed positively, resulting in a clockwise motion. All other logic remains identical and thus will not be re-stated.

### *2.3.2.14 rotateStepper(int dir, int numSteps)*

Figure 31 contains the code used to rotate the stepper a given number of steps in either direction.



```
void rotateStepper(int dir, int numSteps){ //let 0=CW, 1=CCW
    for(int i = 0; i < numSteps; i++){//loop through the numsteps, changing the state between steps
        switch (curState) {
            case 0:
            if(dir == 0){//if CW
                curState++;//increase state by 1
                PORTA = 0b00000110;//output state 2
                }else{ //if CCW
                curState = 3; // previous state is 4 so set curState to 4
                PORTA = 0b00000101;//output state 4
            }//else
            break;
            case 1:
            if(dir == 0){//if CW
                curState++;//increase state by 1
                PORTA = 0b00101000;//output state 3
                }else{ //if CCW
                curState--;//decrease state by 1
                PORTA = 0b00110000;//output state 1
            }//else
            break;
            case 2:
            if(dir == 0){//if CW
                curState++;//increase state by 1
                PORTA = 0b00000101;//output state 4
                }else{//if CCW
                curState--;//decrease state by 1
                PORTA = 0b00000110;//output state 2
            }//else
            break;
            case 3:

            if(dir == 0){//if CW
                curState = 0;//next state is state 1 so set curState to state 1
                PORTA = 0b00110000;//output state 1
                }else{ //if CCW
                curState--;//decrease state by 1
                PORTA = 0b00101000;//output state 3
            }//else
            break;
        }//switch
        mTimer(15);//15 ms delay between states to allow electromagnet charge
    } //for
} //rotateStepper
```

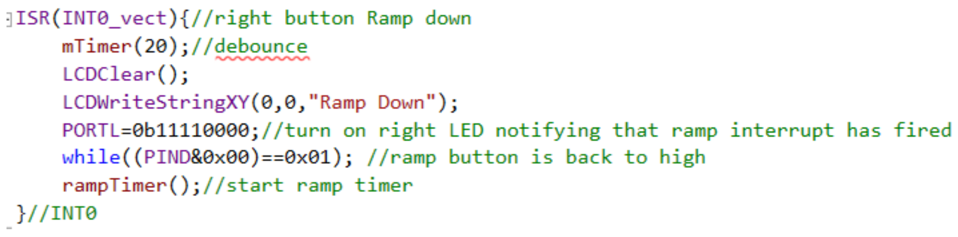
*Figure 31 Code for rotateStepper*

The rotateStepper() function is responsible for controlling the stepper motor's rotation, either clockwise (CW) or counterclockwise (CCW), for a specified number of steps. The direction (dir) is passed as a parameter, where 0 represents clockwise rotation and 1 represents counterclockwise rotation. The function iterates through the number of steps (numSteps), changing the state of the motor at each step, and outputs the corresponding control signal to PORTA. Inside the function, the switch statement determines the motor's current state (curState) and adjusts it based on the direction of rotation. There are four possible states (0 through 3) corresponding to the four step positions of the stepper motor. After each state change, the mTimer(15) function is called, introducing a 15-millisecond delay to allow the electromagnets to fully energize before proceeding to the next state. This process is repeated for the number of steps specified in numSteps, effectively rotating the stepper motor in the desired direction. The function provides precise control over the motor's stepping sequence, allowing for accurate positioning.

### 2.3.3 Interrupt Service Routines (ISRs)

#### *2.3.3.1 Ramp Down State Interrupt Service Routine*

Figure 32 details the code for the Ramp Down State Interrupt Service Routine.



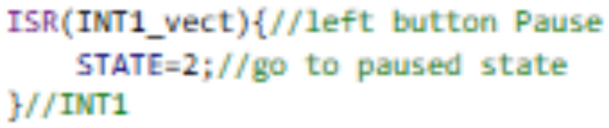
```
ISR(INT0_vect){//right button Ramp down
    mTimer(20);//debounce
    LCDClear();
    LCDWriteStringXY(0,0,"Ramp Down");
    PORTL=0b11110000;//turn on right LED notifying that ramp interrupt has fired
    while((PIND&0x00)==0x01); //ramp button is back to high
    rampTimer();//start ramp timer
}//INT0
```

*Figure 32 Code for Ramp Down State Interrupt Service Routine*

This ISR is triggered when the ramp-down button is pressed, initiating the rampTimer() function. Upon pressing the active-high button, the ISR is entered. Once the button is released, the rampTimer() function is called, and the program resumes execution from its previous position in the code.

### *2.3.3.2 Pause State Interrupt Service Routine*

Figure 33 details the code for the Pause State Interrupt Service Routine



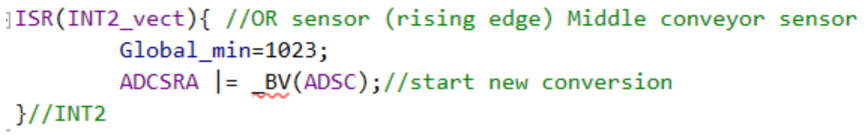
```
ISR(INT1_vect){//left button Pause
    STATE=2;//go to paused state
}//INT1
```

*Figure 33 Pause State Interrupt Routine*

This ISR is triggered when the active-low button is pressed, setting STATE to 2. Once the button is released, the system enters the pause state.

### *2.3.3.3 Optical Sensor interrupt Service Routine*

Figure 34 details the code for the Optical Sensor Interrupt Service Routine.



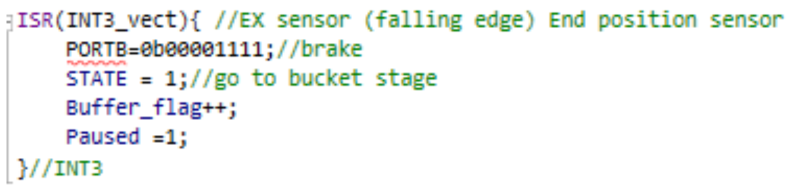
```
ISR(INT2_vect){ //OR sensor (rising edge) Middle conveyor sensor
        Global_min=1023;
        ADCSRA |= _BV(ADSC);//start new conversion
}//INT2
```

*Figure 34 Optical Sensor Interrupt Service Routine*

This ISR is used to start an ADC conversion. The Global_min is set to 1023 and the line " ADCSRA |= _BV(ADSC)" starts the ADC conversion.

### *2.3.3.4 Exit Sensor Interrupt Service Routine*

Figure 35 details the code for the exit Sensor Interrupt Service Routine.



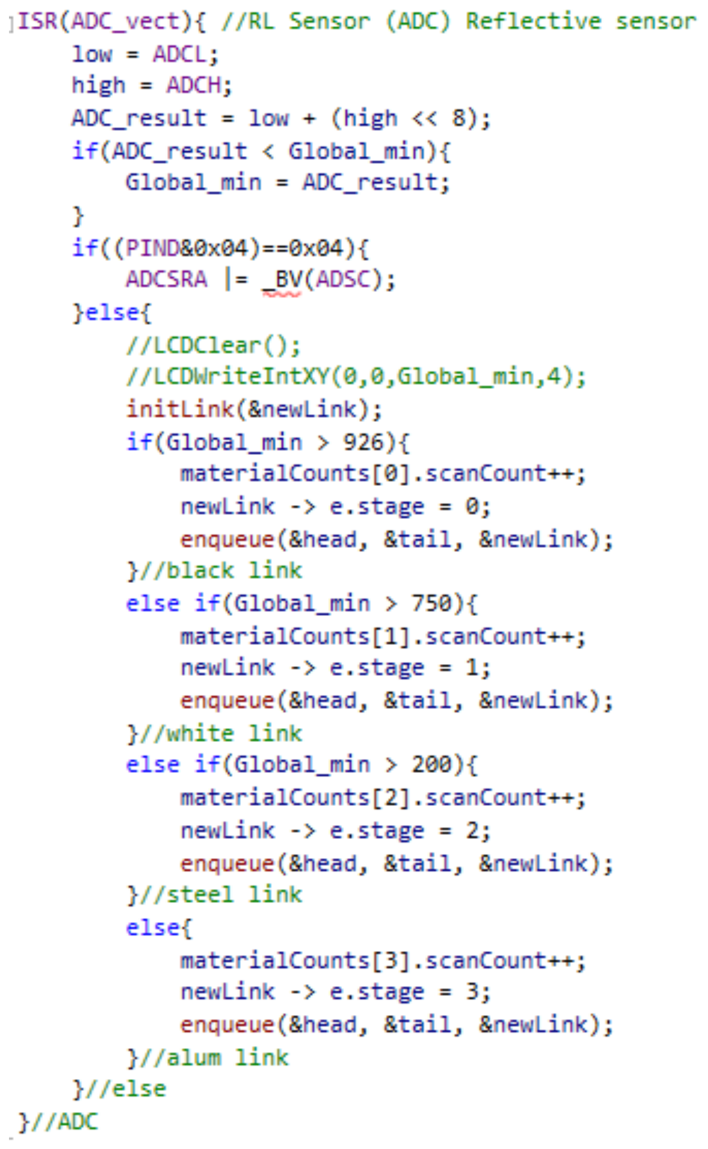
```
ISR(INT3_vect){ //EX sensor (falling edge) End position sensor
    PORTB=0b00001111;//brake
    STATE = 1;//go to bucket stage
    Buffer_flag++;
    Paused =1;
}//INT3
```

*Figure 35 Exit Sensor Interrupt Service Routine*

This ISR is entered when the exit sensor detects an object. The conveyor belt is set to brake, and sets STATE to 1 sending it to the bucket stage. If it is, a buffer flag is increased to signal that an object passed the exit sensor. This flag is used to ensure that no item gets missed during sorting. Paused is set to 1 indicating the conveyor belt is now stopped.

### *2.3.3.5 ADC Interrupt Service Routine*

Figure 36 details the code for the ADC Interrupt Service Routine.



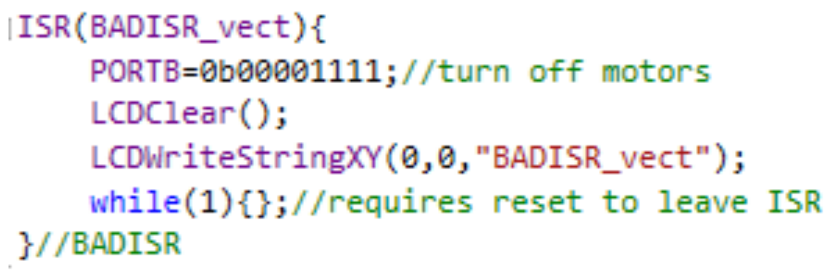
```
ISR(ADC_vect){ //RL Sensor (ADC) Reflective sensor
    low = ADCL;
    high = ADCH;
    ADC_result = low + (high << 8);
    if(ADC_result < Global_min){
        Global_min = ADC_result;
    }
    if((PIND&0x04)==0x04){
        ADCSRA |= _BV(ADSC);
    }else{
        //LCDClear();
        //LCDWriteIntXY(0,0,Global_min,4);
        initLink(&newLink);
        if(Global_min > 926){
            materialCounts[0].scanCount++;
            newLink -> e.stage = 0;
            enqueue(&head, &tail, &newLink);
        }//black link
        else if(Global_min > 750){
            materialCounts[1].scanCount++;
            newLink -> e.stage = 1;
            enqueue(&head, &tail, &newLink);
        }//white link
        else if(Global_min > 200){
            materialCounts[2].scanCount++;
            newLink -> e.stage = 2;
            enqueue(&head, &tail, &newLink);
        }//steel link
        else{
            materialCounts[3].scanCount++;
            newLink -> e.stage = 3;
            enqueue(&head, &tail, &newLink);
        }//alum link
    }//else
}//ADC
```

*Figure 36 ADC Interrupt Service Routine*

This ISR is used to set the reflective global minimum variable equal to the ADC result. The ISR is entered when an ADC conversion is completed, then checks if the ADC result is less than the global minimum. If this is true, the global minimum will be set as the ADC result. Then, the code checks if the OR sensor is active, meaning the object is still in front of it. If this is true, another ADC conversion is started using "ADCSRA |= _BV(ADSC)" and the logic is repeated. In the reflective state, the global minimum is used to identify the material of the object. When the OR sensor is no longer active, the scanned count of the material is increased, and the link is enqueued to the linked list.

#### *2.3.3.6 Unexpected interrupt Handler*

Figure 37 details the code for the Unexpected Interrupt Handler.



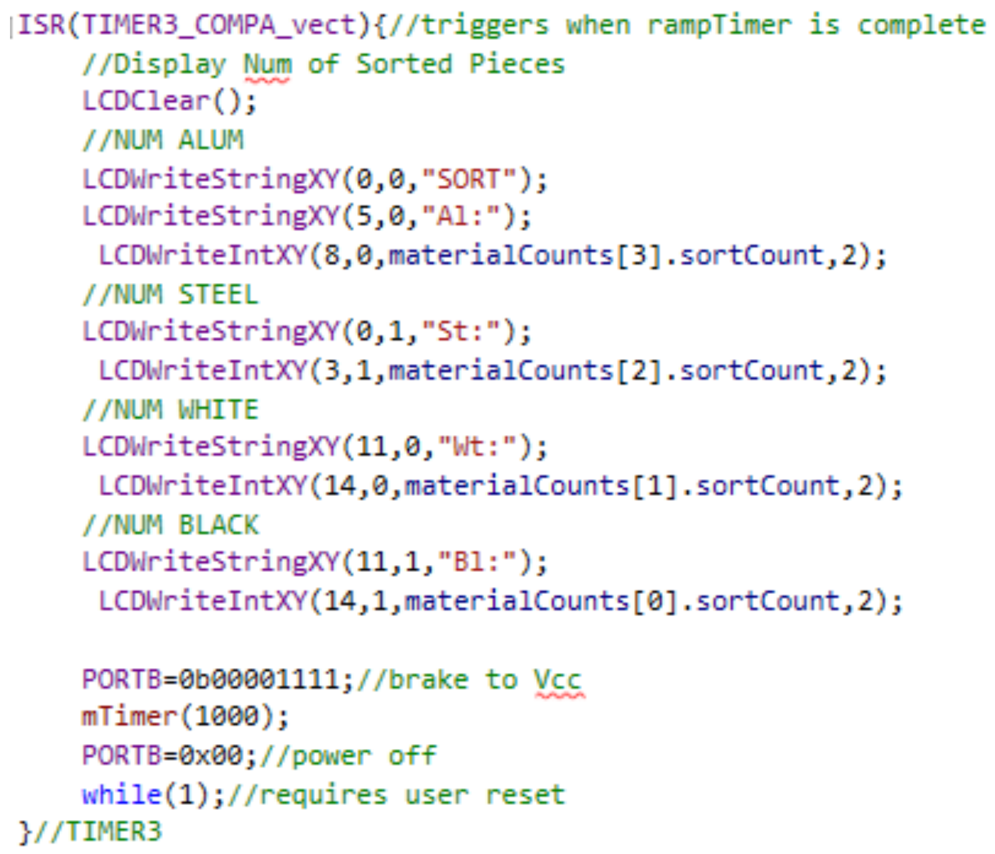
```
ISR(BADISR_vect){
    PORTB=0b00001111;//turn off motors
    LCDClear();
    LCDWriteStringXY(0,0,"BADISR_vect");
    while(1){};//requires reset to leave ISR
}//BADISR
```

*Figure 37 Unexpected interrupt Handler*

This ISR is triggered when an unexpected interrupt is received, initiating a system shutdown. Upon activation, the port controlling the DC motor is set to off, and the message "BADISRVect" is displayed on the LCD to indicate that this ISR has been entered. The program then enters a while(1) loop, ensuring that the ISR remains active until the system is manually reset by the user.

#### *2.3.3.7 Ramp Down Timer Interrupt Service Routine*

Figure 38 details the code for Ramp Down Timer Interrupt Service Routine.



```
ISR(TIMER3_COMPA_vect){//triggers when rampTimer is complete
    //Display Num of Sorted Pieces
    LCDClear();
    //NUM ALUM
    LCDWriteStringXY(0,0,"SORT");
    LCDWriteStringXY(5,0,"Al:");
     LCDWriteIntXY(8,0,materialCounts[3].sortCount,2);
    //NUM STEEL
    LCDWriteStringXY(0,1,"St:");
     LCDWriteIntXY(3,1,materialCounts[2].sortCount,2);
    //NUM WHITE
    LCDWriteStringXY(11,0,"Wt:");
     LCDWriteIntXY(14,0,materialCounts[1].sortCount,2);
    //NUM BLACK
    LCDWriteStringXY(11,1,"Bl:");
     LCDWriteIntXY(14,1,materialCounts[0].sortCount,2);

    PORTB=0b00001111;//brake to Vcc
    mTimer(1000);
    PORTB=0x00;//power off
    while(1);//requires user reset
}//TIMER3
```

*Figure 38 Code for Ramp Down Timer Interrupt Service Routine*

This ISR is used to shut down the system and display the number of sorted objects for each material once the timer started in the function rampTimer() is complete. The ISR is entered once the rampTimer throws an output compare interrupt and subsequently displays the number of sorted objects for each material. The DC motor is first set to brake followed by power off. A while(1) is entered to ensure the ISR is not left without a user reset.

## 2.4 Project Timeline

Figure 39 shows the Project Timeline Chart where blue cells indicate time allocated to work on that item, and red cells indicate due dates.

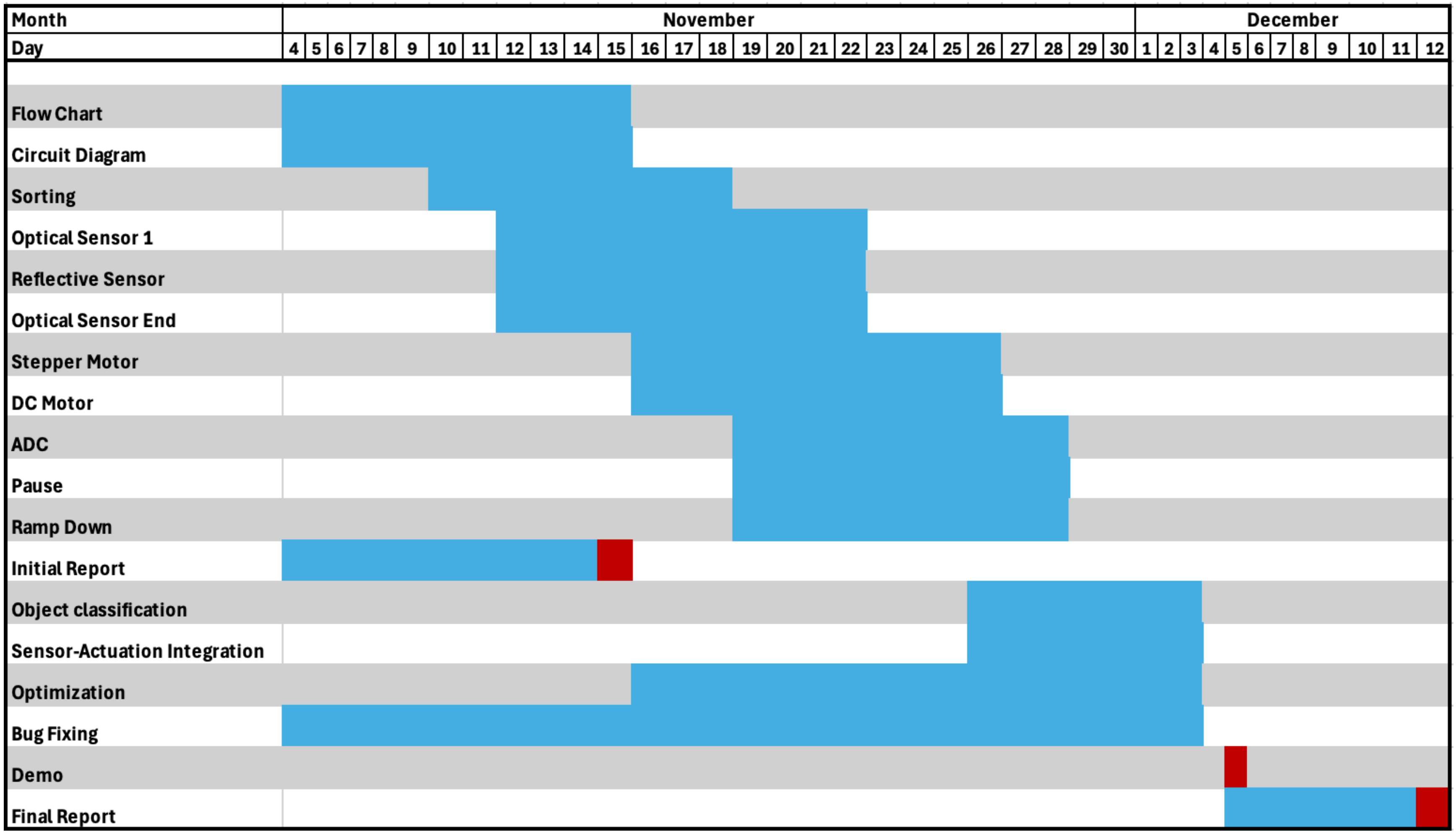


*Figure 39: Project Timeline Chart*

## 2.5 Circuit Diagram

Figure 40 shows the circuit diagram illustrating the connections between the systems hardware components.

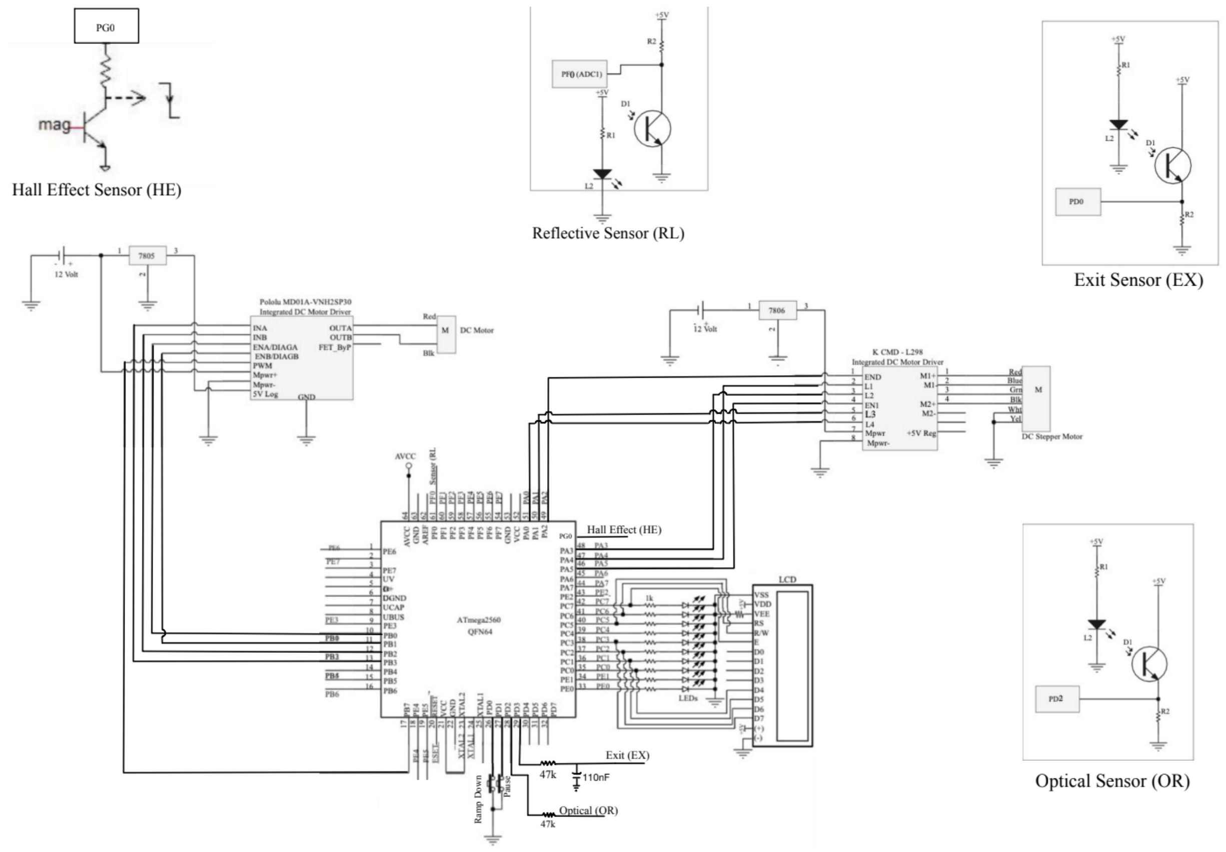


*Figure 40: Circuit Diagram*

# 3 System Performance Specifications

## 3.1 Software optimization

Throughout the project, the group reworked the overall code many times to improve logic, flow, and decrease runtime and complexity. The project required code that could handle many inputs and process lots of information to make decisions quickly. As such, we decided to make our code as efficient as possible before attempting to implement optimizations such as continuous loading or a “look ahead” feature. Early iterations of our code were pieced together from the labs, and included bulky stepper motor excitation sequences, and slow rotation algorithms. To improve our stepper charging algorithm, we implemented arrays containing the charge orders and iterated through it to activate the motor. Similar to how we handled the rotations originally, our early bucket stage contained nested switch statements which would first asses where the bucket needed to turn and then assessed where it currently was. Nested switch statements are inherently slow and cumbersome to deal with, so we created a lookup table that would take in the current position and target position as inputs and call the necessary rotate function. This effectively reduced the runtime of that section from $O(n^2)$ at worst case, to $O(1)$.

The next level of optimization the group decided to address was continuous loading. After many failed attempts at this issue, including changing the order of interrupts and creating many flags, it was discovered that the only solution was to have the sorting and enqueuing of materials take place within the reflective sensor ISR. Without doing so, the code was attempting to run at two different places in main simultaneously, the reflective stage and the bucket stage. By removing the reflective stage and handling the operations within the interrupt, we were able to achieve continuous loading.

The final level of optimization that the group decided to undertake was implementing a "look ahead" feature. The idea of this feature is to look ahead at the list, before pieces reach the exit sensor, and adjust the algorithm to sort the pieces more efficiently. Specifically, we wanted to replace consecutive, same-direction 90 degree turns with a continuous turn and multiple pre-drops. This would save considerable time as the stepper acceleration and deceleration is much slower than continuing at peak speed. Looking ahead at the list and determining if the next couple turns are going to be same direction 90 degree turns was fairly straightforward and just involved creating a lookup table which takes in a start position and end position and returns the magnitude and direction of the required turn. Similarly, dropping multiple pieces during a single rotation was quite simple although tuning the pre-drops is quite tedious, and the dropDone variable was created to keep track of how many pieces were dropped so they could be processed in the bucket stage. The main obstacle with this feature is determining the spacing between pieces and if they will be close enough for the pre-drops to land in the bins. Because the pieces are loaded by hand, consistent spacing cannot be guaranteed. The solution we came up with was to check whether there was a piece at the exit sensor ready to be dropped as late as possible. We first implemented this with just two consecutive 90 degree rotations replaced by a 180 with two drops. The s-curve the group was using required 17 steps to decelerate safely from top speed, therefore we checked if there was an exit sensor flag present 17 steps before the single 90 degree turn was completed. If an exit flag was present, then the algorithm continued with its 180 degree rotation and completed the two drops. If there was no exit flag present, the gap was determined to be too large and the double drop would not be successful, at which point the bailout_flag was set to one. At the next iteration of the stepper rotation for loop, the step delay would then call the value from our 90 degree (50 step) array thus commencing the deceleration and not dropping a second piece. This idea was then scaled to 270 and 360 degrees for more consecutive turns. With more turns, we had to keep track of which drop the bailout was happening at to know which array to switch to for deceleration. The bailout flag was set to 1 if the second piece had too much of a gap, 2 if the third piece had too much of a gap, and so forth. The stepper delay was then pulled from the correct array by using a switch statement with the bailout flag as input. On the last step of the array being used, if a bailout flag was present, a breakout flag (breakout_flag) was set to 1 and the function would break from the for loop.

The look ahead feature could have easily been scaled further to include cases for back to back 180s being replaced by a 360 with two drops, or a 90 and 180 being replaced with a 270, or any other of the countless possible combinations. However, we had our hands full trying to sort out hardware issues, and faulty sensors, so we decided to stick with what we had and tune in the pre-drops.

## 4 System Limitations and Performance Trade-offs

The optimization of the sorting system involved encountering several limitations and making trade-offs between speed and reliability. This section will first address the system's limitations, followed by a discussion of the trade-offs made during the optimization process.

## 4.1 System Limitations

During system optimization, several limitations were identified that could not be exceeded. The first limitation was the conveyor belt speed. At duty cycles exceeding approximately 72.5%, Steel items would fly off the belt upon reaching the exit sensor. This occurred because the inertia of the items surpassed the frictional force holding them in place. Notably, this issue was unique to Steel items as their mass was significantly greater than the others and as such they had much more momentum.

Additionally, significantly increasing the belt speed reduced the accuracy of the reflectivity measurements. This was due to fewer ADC conversions performed at higher speeds compared to slower ones. Consequently, the belt speed was constrained by both the risk of items falling off the conveyor and the need to maintain accurate reflectivity readings.

The next major limitation observed was in the stepper acceleration profile. When weight was added to the tray, simulating steel and aluminum pieces being trapped in the black or white tray. Using this approach, the maximum stepper delay, which is used to start the stepper from a standstill, and the minimum stepper delay, which is applied once the stepper has fully accelerated, were determined experimentally. When the maximum stepper delay was reduced below 15 ms, it was observed that the stepper would often stall during direction changes. Additionally, if a heavier item, such as steel or aluminum, fell onto the stepper during rotation, the impact would frequently cause the stepper to stall. Therefore, 15 ms was determined to be the lowest viable maximum stepper delay.

Similarly, when the minimum stepper delay was reduced below 4.5 ms, the stepper would occasionally stall, depending on direction changes and the load. While the below 4.5 ms delay resulted in an extremely fast stepper, it would often stall under the standard load conditions used during testing. Therefore the s-curve profile could be much more aggressive but with higher risk of stalling, as such, the group opted for a safer profile that was consistent.

## 4.2 System Trade-offs

During the optimization of the sorting system, various performance trade-offs were encountered and analyzed to achieve a fast, reliable, and accurate system. The maximum values discussed in the system limitations served as the foundation for the integrated system design, with each major parameter being adjusted to operate at the fastest stable value. Some parameters, such as belt speed and pre-drop location, did not involve trade-offs but instead had inherent limitations, as explained in the previous section. These parameters did not impact other system functionalities and were optimized independently. However, this level of independence did not apply to all parameters.

Parameters such as the minimum and maximum stepper delays were interdependent with other system functionalities, necessitating the analysis of performance trade-offs. A key trade-off encountered was the reduction in the stepper's acceleration profile to incorporate the pre-drop functionality. Since the stepper did not come to a complete stop using this method, the initial stepper delay had to be significantly increased to generate enough torque to overcome the inertia of the sorting tray, particularly during direction changes

# 5 Testing and Calibration Procedures

This section outlines the testing and calibration procedures conducted to ensure the accuracy of the system.

## 5.1 Reflectivity Calibration

The item type was identified using a reflectivity sensor. A 10-bit ADC converted the analog signal into a digital signal ranging from 0 to 1023. To classify items, the ADC output was compared to predefined reference values obtained through testing. A testing program was used to display the maximum ADC output for each item on an LCD. Each item type was scanned approximately 20 times to ensure a sufficient sample size, and the minimum and maximum reflective index (RFI) values were recorded.

Since there were four objects, three threshold values were determined to classify them effectively. These thresholds were calculated as the midpoints between the maximum RFI of the smaller reflective index and the minimum RFI of the larger reflective index. For example, during the demonstration, the minimum RFI for black was 930, while the maximum RFI for white was 920. To reduce classification errors, items with an RFI greater than 924 were categorized as black. Table 1 summarizes the finalized RFI thresholds used during the demonstration.

*Table 1 Reflective Index Sorting Values*

| Type | Minimum Reflective Index |
|---|---|
| Black | 925 |
| White | 750 |
| Steel | 120 |

## 5.2 Maximum Belt Speed

The primary concern with increasing the belt speed was potential errors in sorting due to ADC timing issues. To evaluate this, a test was conducted by placing two sets of eight pieces on the belt and observing whether errors occurred at higher belt speeds. Surprisingly, the limiting factor was not the ADC timing but the position of the first piece in the stack.

If a Steel piece was at the start of the stack, the belt occasionally had to pause while waiting for the stepper motor to reach its correct position. During this pause, the Steel piece sometimes fell off before being sorted accurately. To address this, the belt speed was set to 50% of its maximum speed, which was found to be a stable value that prevented this error. However, further increases in belt speed disregarding the Steel-related issue did result in ADC timing errors.

## 5.3 Stepper Motor: S-Curve

After optimizing the trapezoidal velocity profile, an S-curve profile, illustrated in Figure 41, was implemented to account for the identified stepper motor limitations.

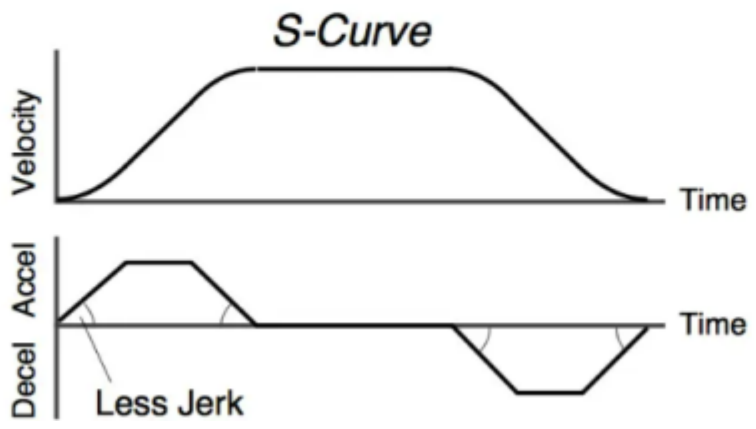
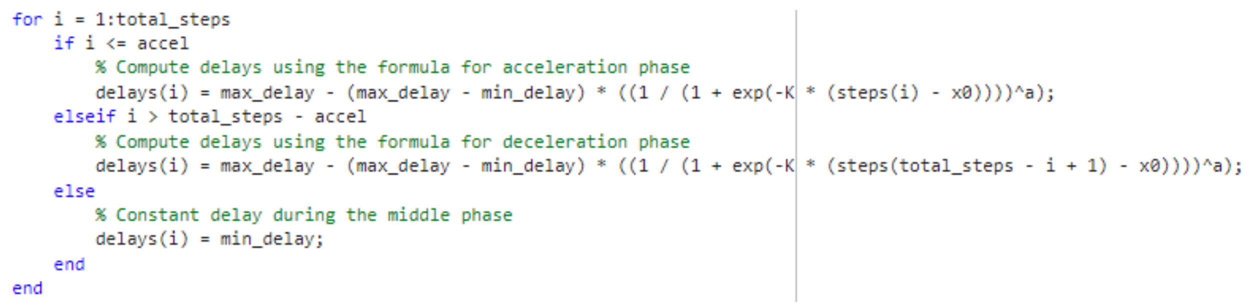


*Figure 41 Standard S-Curve Profile*

The equation used to construct the S-curve profile is presented in Figure 42.



```
for i = 1:total_steps
    if i <= accel
        % Compute delays using the formula for acceleration phase
        delays(i) = max_delay - (max_delay - min_delay) * ((1 / (1 + exp(-K * (steps(i) - x0))))^a);
    elseif i > total_steps - accel
        % Compute delays using the formula for deceleration phase
        delays(i) = max_delay - (max_delay - min_delay) * ((1 / (1 + exp(-K * (steps(total_steps - i + 1) - x0))))^a);
    else
        % Constant delay during the middle phase
        delays(i) = min_delay;
    end
end
```

*Figure 42 S-Curve Equation*

The coil delays were optimized through extensive testing, with the maximum and minimum delays set to 15 ms and 4.5 ms, respectively. The parameter ‘x0’ represents the mean of the time intervals and shifts the curve horizontally. The parameter ‘k’ adjusts the steepness of the curve, with larger values resulting in a steeper slope. Meanwhile, the parameter ‘a’ also influences the curve's shape, with lower values producing a gentler slope. Table 2 details the finalized parameters resulting in the fastest sort time while having no rotational errors.

*Table 2 Finalized S-Curve Parameters*

| S-Curve Parameters | Value |
|---|---|
| Maximum Delay [ms] | 15 |
| Minimum Delay [ms] | 4.5 |
| x0 | 6 |
| k | 0.5 |
| a | 2 |

Figure 43 details the acceleration profile with the above specifications. The curve in figure 43 shows the time delay vs step which it should be noted is inversely proportional to speed. If speed were plotted against the steps the curve would be flipped.

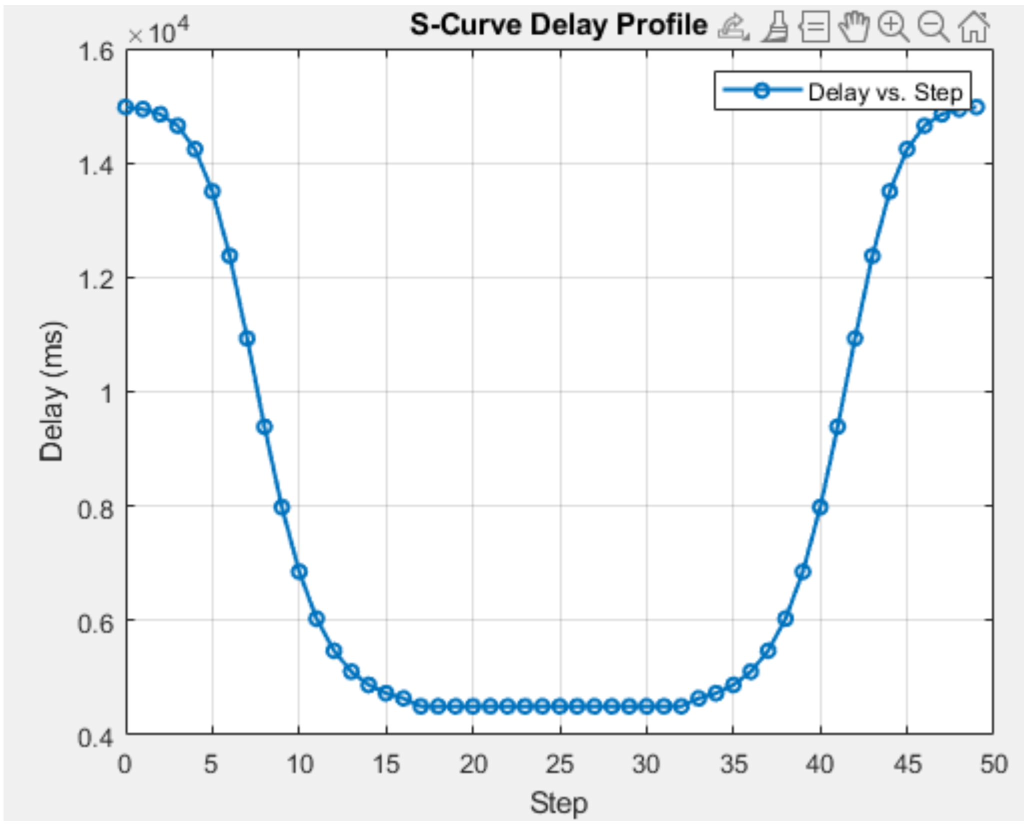


*Figure 43 Delay between Stepper Motor Steps vs. Steps*

## 5.4 Noise Reduction

Noise interference from the DC motor and other electrical appliances affected the performance of the optical sensor and exit sensor. To mitigate this issue, we implemented a filter circuit to suppress the noise and stabilize the sensor signals. For the exit sensor, we added a capacitor in parallel with a 47 kΩ resistor, forming a low-pass filter to attenuate high-frequency noise. For the optical sensor, we introduced a 47 kΩ resistor, which helped stabilize the signal by reducing susceptibility to electrical interference. These modifications significantly improved the reliability and accuracy of the sensor readings, ensuring consistent performance despite the presence of electrical noise sources.

# 6 Demonstration Results

The system's performance during a demonstration is evaluated using the 'System Performance Index' (SPI). This index accounts for both the total sort time and the number of errors. The SPI is calculated using the following formula:

$$SPI = \frac{N_C - N_I}{T}$$

Where $N_C$ is the number of correctly sorted items, $N_I$ is the number of incorrectly sorted items and T is the time it took to sort all items. Table 3 details the demonstration results. The best test was a sort time of 22 [s] and zero errors, resulting in a SPI of 2.18.

*Table 3 Demonstration Results*

| Test # | Sort Time [s] | # of Errors | SPI |
|---|---|---|---|
| 1 | 22 | 0 | 2.18 |
| 2 | 22 | 1 | 2.09 |
| 3 | 22 | 1 | 2.09 |
| 4 | 21 | 2 | 2.1 |

# 7 Experience and Recommendations

The following section offers a comprehensive discussion of the project team's experiences and recommendations.

## 7.1 Project Experience

This project exceeded all expectations the team had at the outset of the course. Initially, the project seemed daunting, especially since the team was not explicitly skilled in coding. However, throughout the five instructional labs, the team rapidly gained confidence in writing, reading, and understanding code, transforming what seemed like a challenging task into a manageable and rewarding experience. The project served as an excellent introduction to mechatronic systems, sparking both interest and curiosity in the field of mechatronics. Throughout the process, both team members gained confidence in their coding abilities and became proficient in debugging. This skill was particularly crucial, as the first iteration of a new block of code rarely worked as expected. Additionally, the team developed a strong understanding of troubleshooting hardware and circuitry. By the project's conclusion, the team was able to quickly pinpoint errors in the system using a systematic troubleshooting routine. Ultimately, the experience not only boosted the team's confidence in working with both circuitry and software but also provided invaluable knowledge in mechatronic systems that will be carried forward into their future careers in industry.

## 7.2 Project Recommendations

While the team's experience with the project was overwhelmingly positive, several recommendations could enhance the experience for future students.

The first recommendation is to design a simple enclosure for all wiring and circuitry that is not meant to be modified by the students. Throughout the project, the team encountered system errors caused by loose wires, often due to the apparatus being accidentally disturbed by students or objects falling off the sorting tray. A simple enclosure to secure these components would significantly improve the experience by reducing the time spent troubleshooting unintentional wiring issues.

The second recommendation involves reducing the reliance on human skill for loading the conveyor belt. Given the high speed of the belt, accurately and reliably placing 8 objects at a time proved to be extremely challenging. Perfecting this manual technique took significant time away from optimizing the system. Automating the loading process would eliminate the human error factor. A simple dropping mechanism, such as a solenoid actuator to release the pieces, could be designed to load the objects in a controlled manner. This addition would create an additional challenge for students, as the loading process would need to be synchronized with the sorting system. Implementing this feature would be relatively straightforward and could serve as a design project for either ENGR110 students or co-op students.

Lastly, it is recommended to provide greater incentives for optimizing the system. Currently, the difference in project grades between teams with a sorting time of 50 seconds versus one of 25 seconds is only a 6% increase, despite the significant difference in the time and effort required to achieve faster sorting times. Teams that spend 50 hours optimizing their system may only gain a small grade advantage compared to teams that spend considerably less time. Increasing the weight of system optimization in the grading rubric would create more motivation for teams to invest additional time and effort into improving their designs. Attempting to improve the code and implement optimizations is where the group learned the most. At the beginning of the class no one in the group was confident in coding or had experience with microcontrollers of any kind, and by the end achieved one of the top times and were able to help other groups debug and improve their code. This was due to countless hours spent trying to optimize our program, and a greater grade weight incentive would encourage all students to do so and could improve the overall learning experience for students. While our team greatly enjoyed the project and wanted to learn the required skills, if we hadn’t, it would have been easy to “scrape by” with minimal effort and still receive a good grade.

# 8 Conclusion

The primary goal of this project was to design, code, and optimize the software and control system for the conveyor belt sorting system. Each system requirement was thoroughly analyzed before proceeding with the integrated system design. After developing an initial working version of the sorting system, extensive testing was conducted to identify the system's maximum and stable operating parameters. These parameters served as a foundation for optimization, and the performance trade-offs of each were carefully evaluated. Once the initial code was fully optimized, additional functionality was integrated, and the optimization process was repeated. This cycle was completed three times, with the final code set achieving a completion time of 22 seconds and no errors. The project provided the team with invaluable experience in mechatronic systems, equipping us with skills and knowledge that will be carried forward beyond graduation.

## 9 References

[1] The University of Victoria, “MILESTONE 5: Final Project,” 2024.

[2] Microchip, ATmega640-1280-1281-2560-2561-Datasheet-DS40002211A, (Accessed 2024)