University of Victoria
Department of Engineering

MECH 458
Object Sorting System
Final Report

Lab Section: B03

Submitted on: Dec 12, 2016

Minghua Li V00801708
Christopher Burgher V00764083

Contents

Tables and Figures

Tables

Figures

Abstract

The objective of this project was to design an object sorting system that is able to inspect key features of various cylindrical objects and sort the objects into separate bins characterized by its material type. Over the past four months, a series of lab exercises were conducted to become familiarized with the hardware and software that will be required to successfully complete this project. The primary challenge of this project was to implement a reliable sorting algorithm into a software application that will run on top of the Atmel AT90USB1287 microcontroller.

This report will detail the necessary hardware and software involved in the implementation of the sorting system. A significant portion of the report will involve a thorough discussion of the implementation of the sorting system. More specifically, this will include interfacing the MCU to the various sensors and drivers of the sorting unit as well as a detailed discussion of the sorting algorithm that will be implemented in software.

A discussion of the testing results of the implemented system as well as the results of the final demonstration will also be included.

The limitations of the sorting system and methods in which the sorting systems performance could be improved are also addressed.

1.0 Introduction

1.1 Problem Description

The objective of this experiment is to design an object sorting system that is able to inspect key features of various cylindrical objects and sort the objects into separate bins characterized by its material type. The sorting system will be used to sort out a random number of small cylindrical objects made of steel, aluminum, black plastic or white plastic. The Atmel AT90USB1287 microcontroller will be used in this project to operate and control the system as a functional object sorting unit. The main sorting system has already been setup beforehand and the primary task of this project is to design the software that will run on top of the Atmel AT90USB1287 microcontroller when sorting out the cylindrical objects. Further, some of the other outcomes of this project will include:

1. Understand how to set up and write software programs that run on the Atmel MCU
2. Understand how to program and execute the application on the microcontroller to gain the desired outputs
3. Learn how to interface with a stepper motor, brushed DC motor, and various sensors.

1.2 System Setup

The basic layout of the sorting system and the microcontroller are displayed in figure 1 and figure 2 below (taken from class slides). The sorting system will include the following components:

- Stepper motor with rotating bin.
  Soyo 6V 0.8A 36oz-in Unipolar Steeper Motor that controls the position of a sorting bin (see figure 1). The rotating bin has four dividers used to separate the different cylinders according to material type.
- Conveyor belt system with a DC motor
- Two motor drivers:
  - One K CMD – L298 Integrated DC Motor Driver is an integrated monolithic circuit in a 15 lead Multiwatt and powerSO20 packages. It can accept standard TTL logic levels and drive inductive loads.
  - One Pololu MD01A - VNH2SP30 Integrated DC Motor Driver
- One Inductive sensor, one reflective sensor, one Hall effect sensor and three optical sensors
- Analog-to-digital converter (ADC)
- AT90USBKey microcontroller. The AT90USBKey is a low-cost demonstration board based on the AT90USB1287 microcontroller. The key simply connects to the USB port of a personal computer to retrieve on-board documentation and run AVR programs.

The hardware components will be discussed in greater detail in section 2.1. To note, it turns out that the objects that will be sorted can be classified based solely on their reflective properties. Thus the inductive sensor (located in station 1) will not be used in this project.

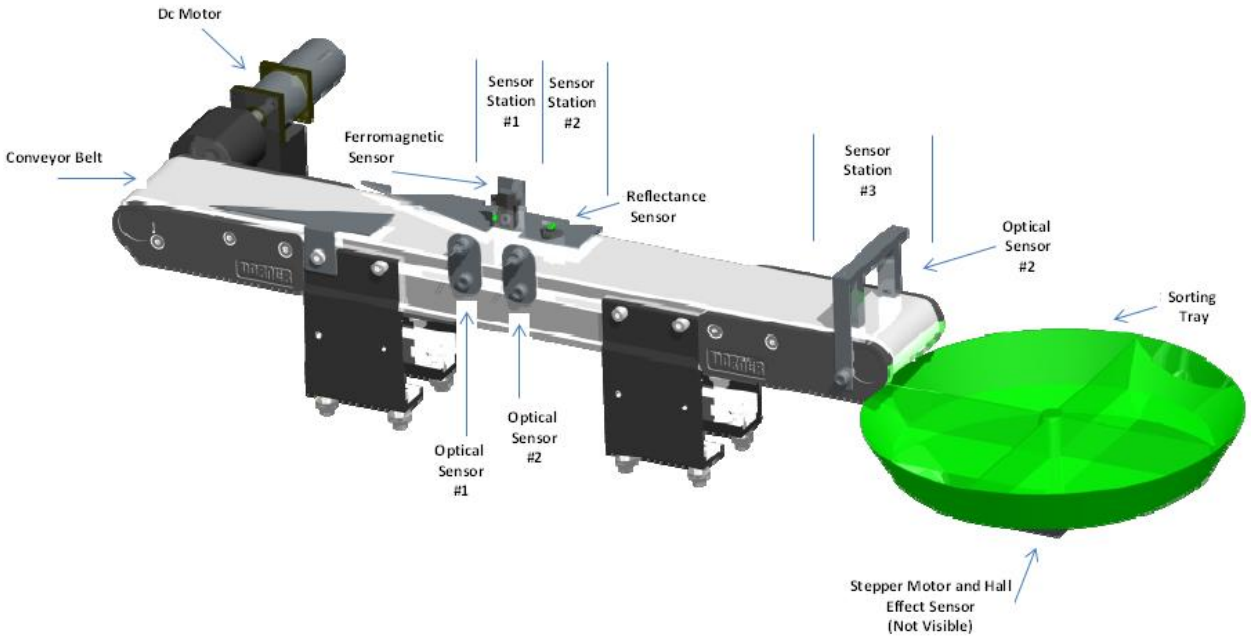

Figure 1: Basic layout of object sorting system

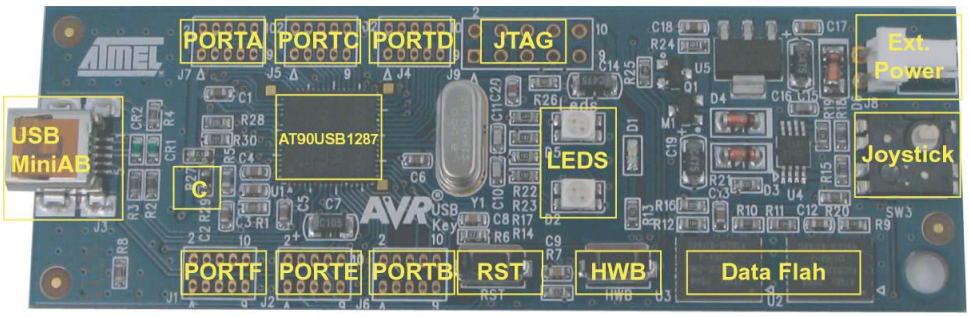

Figure 2: AT90USBKey microcontroller layout**.**

2.0 System Design

2.1 Hardware

The primary hardware components of the sorting system will include the MCU, DC motor, stepper motor, ferromagnetic sensor, reflective sensor, and three optical sensors. To note, only two of the optical sensors will be required for this project. This includes the optical sensor at sensor station 2 and the optical sensor at sensor station 3. The optical sensor at sensor station 1 will not be used since the inductive sensor is not required. The following is a specific description of each component.

2.1.1DC Motor and Motor Driver

DC Motor: The DC motor will be used to control the speed of the conveyor belt. It receives signals from the Pololu MD01A - VNH2SP30 Integrated DC Motor Driver and is driven by a PWM signal.

Motor Drivers: The Pololu MD01A - VNH2SP30 Integrated DC Motor Driver will be used to interface the DC motor to the microcontroller.

The motor was generally run at 47.8% of full speed during our sorting. This motor speed rate was calibrated by applying a few tests. A higher speed will yield less reliable ADC measurements from the reflective sensor because there is a smaller window of time that the object is in front of the sensor. On the other hand, a lower speed will slow down the total soring time. Therefore, the trade-off result of speed was 47.8% (0x78 in the code).

2.1.2 Stepper Motor and Motor Driver

Stepper Motor: The stepper motor is used to rotate the sorting tray to a desired bin position where the cylindrical objects are placed. In order for the stepper motor to rotate, it will receive a sequence of pulses from the K CMD – L298 Integrated DC Motor Driver which is connected to one of the output ports of the microcontroller.

Motor Drivers: The K CMD – L298 Integrated DC Motor Driver will be used to interface the stepper motor to the microcontroller.

The motor can spin in either a clockwise or counter clockwise direction, and has a gross resolution of 200 steps per 360^{o} revolution. For purposes of this project, the sorting tray will only need to spin either 90^{o} or 180^{o}. A 90^{o} rotation will take 50 steps and a 180^{o} rotation will take 100 steps. A basic Hall Effect sensor was set to identify a home position as a reference position for sorting.

The MCU interfaced the stepper motor using pins PORTE [0…5]. The motor was run in the CW and CCW direction throughout the application, depending on the sorting position required and the current position. There was no motor acceleration process for the stepper motor, since after the longer process (above 35 objects processed), the spin tray has a deflection for every spin to cause a system error which is called the cumulative error.

For each step of stepper motor, the time delay was set to 15 ms to allow the sorting tray to spin faster and smoother.

2.1.3 Optical Sensors

The optical sensors will be used to detect the presence of the cylindrical object as it travels down the conveyor belt. The optical sensors that will be used are located at sensor station 2, and sensor station 3 (See figure 1 above). The optical sensor at station 2 sits across from the reflective sensor. This optical sensor is used to identify when a cylindrical object is in front of the reflective sensor. As long as the object is in front of the optical sensor, ADC measurements will be taken from the reflective sensor. The optical sensor at station 3 is used to detect when an object has reached the end of the conveyor belt. When an object is detected in front of this

optical sensor, the corresponding ISR will be called where the conveyor belt is paused and the object is identified using the sorting algorithm.

The two optical sensors (reflective ready, and belt exit) were connected to PORTD [1] and PORTD [2] respectively.

2.1.4 Reflective Sensor

The reflective sensor was set on sensor station 2 and it is used to measure the reflectiveness of the object. The reflective sensor was an analog sensor and required analog to digital conversion to process the input signal. The lower the voltage reading obtained from the reflective sensor, the higher the reflection that was measured. The reflective sensor was triggered 5 ms after the optical sensor was active (active high), since the object has the highest reflection value right at the center. The 5 ms delay was just given the reflective sensor a better reading of the middle part of the object. The process of a cascade of ADC conversions will stop when the object is no longer in front of the optical sensor. The reflection data will be collected for the object evaluation and calibration process in the further step. The reflective sensor was connected to PORTF [1].

2.1.5 Hall Effect Sensor

The Hall Effect sensor was mounted on the bottom of the sorting tray. When the magnetic pickup is at the sensor, the sensor is active and indicates that the home position has been reached. The Hall Effect sensor was connected to PORTD [4].

2.1.6 8 Bit LED Display

PORTC will primarily be used to display ADC results (by lighting up some LED's) from the sensors during the testing and debugging stages. PORTC will also be used to communicate error codes during these stages to simplify things when troubleshooting problems. A circuit diagram of the 8 bit LED display is shown below in figure 3.

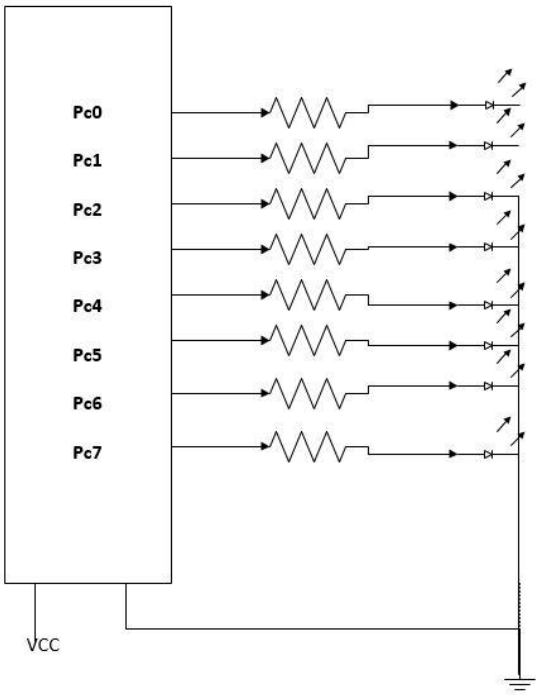

Figure 3. Circuit connection between Port C and 8 LEDs

These 8 LEDs were connected to the system using PORTC [0…7].

2.1.7 4 Bit On Board LED Display

There were two LEDs which were used for the part of indicate function of 10 bit outputs built on AT90USB1287 board. These LEDs are coupled to PORTD [4] and [7] with each bit representing an upper and lower LED red and green color respectively.

2.1.8 Push Button Switches

Two single pole momentary switches were set for the Pause and Ramp Down. These switches were mounted on the proto board. When the pause button is pressed, the conveyor belt will brake and the system will be in a paused state. The system will resume sorting when the pause button is pressed again. When the ramp down button is pressed, the system will shut down after the most recent object to be processed is sorted.

The Pause and Ramp Down buttons were connected to the system using PORTA [0] and PORTD [0] respectively.

2.2 Software

The software that is used in this project was written in C using the Atmel AVR Studio IDE. The developed software is comprised of the following modules:

- A module that configures the I/O status of the selected ports in order to establish communication between the MCU and the rest of the sorting system.
- A module that locates the home position of the sorting tray using input from a Hall effect sensor. See Appendix A.1 for the source code.

- A module that is used to access one of the MCU's system timers. For purposes of this project, the timer will primarily be used to control the rate of rotation of the stepper motor. See Appendix A.6 for the source code.
- A module that utilizes the MCU's pulse width modulation (PWM) capability. PWM will be used to operate the DC motor used to move the conveyor belt. See Appendix A.4 for the source code.
- A module that operates the stepper motor (rotate clockwise/counter clockwise and stop rotation). See Appendix A.5 for the source code.
- A module that operates the DC motor (rotate clockwise and stop rotation). See Appendix A.3 for the source code.
- Modules that configures analog to digital conversion (ADC) capability and obtains ADC measurements from the reflective sensor. See Appendix A.2 for the source code.
- Modules that configure external interrupt capabilities and interrupt service routines (ISR) that respond to external input. There will be an ISR for each of the following external inputs:
    1. Ramp down input. The purpose of the ramp down is to shut down the system after the most recent object is sorted. See Appendix A.1 for the source code.
    2. Rising edge detection from the optical sensor at sensor station 2.
    3. Falling edge detection from the optical sensor at sensor station 3.

It should be noted that the ADC measurements corresponding to the reflectiveness of an object will be stored in an array. The length of the array is set to 48 (the maximum number of objects that can be processed) and will have two counters pointing to a given index of the array. Whenever a new ADC measurement is taken from the reflective sensor, the measurement will be allocated to the memory location pointed to by the tail index. Whenever an object is being sorted at sensor station 3, the head index is used to access the ADC measurement corresponding to the current object being sorted. This ADC measurement is used in the sorting algorithm to identify the object. Below, figure 4 illustrates the structure of the array.

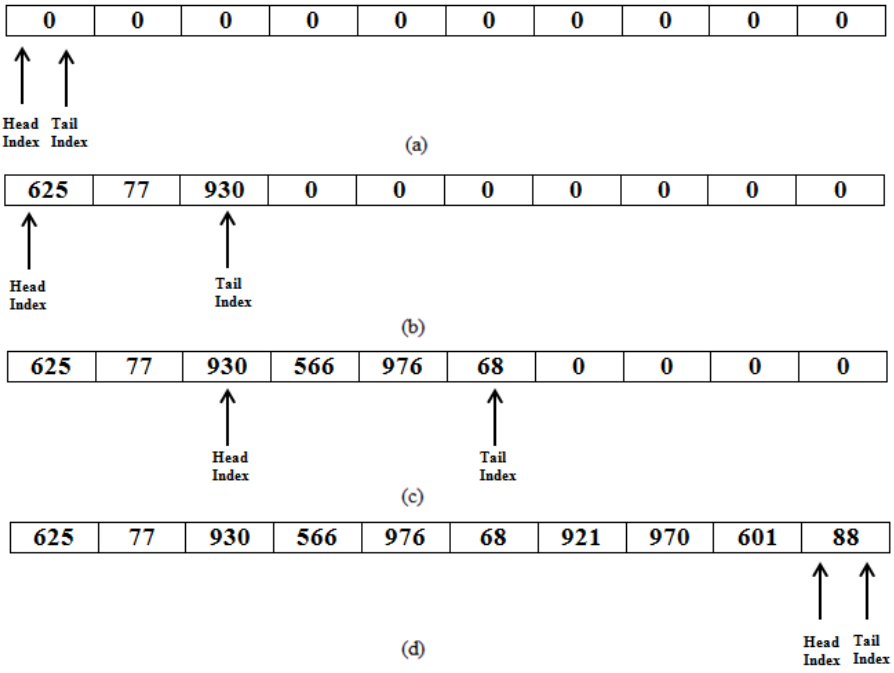

Figure 4: Object array structure. (a) Initial state. (b) 3 ADC values loaded and no object processed. (c) 5 ADC values loaded and 2 objects processed. (d) Final state with all objects processed. Note that the array will have a length of 48.

3.0 System Implementation

3.1 Circuit Connections

Various connections will have to be made between the microcontroller and the rest of the sorting system. In order to control the stepper motor, PORTE will be configured as an output port and will be connected to the various inputs of the K CMD – L298 Integrated DC Motor Driver. This driver is required to interface between the microcontroller and the stepper motor. The output of the driver is sent to the stepper motor. A simple schematic illustrating these connections is shown below in figure 5.

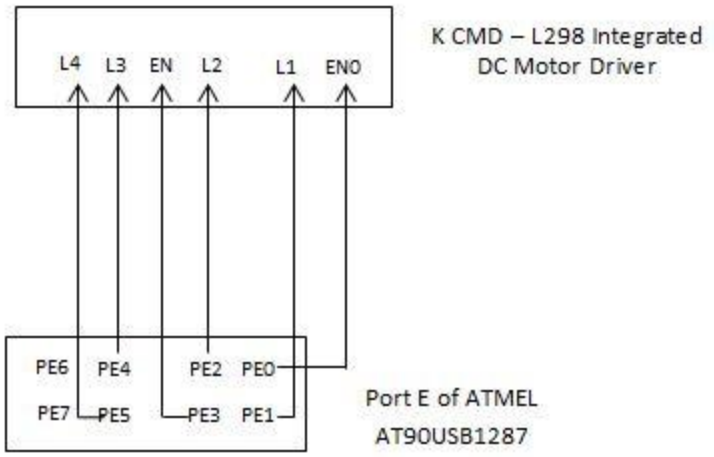

Figure 5: Connections for the stepper motor with directions

In order to control the DC motor, PORTB will be configured as an output port and will be connected to the various inputs of the Pololu MD01A - VNH2SP30 Integrated DC Motor Driver. This driver is required to interface between the microcontroller and the DC motor. The output of the driver is sent to the DC motor. A simple schematic illustrating these connections is shown below in figure 6. Note that pin PB7 is used for PWM.

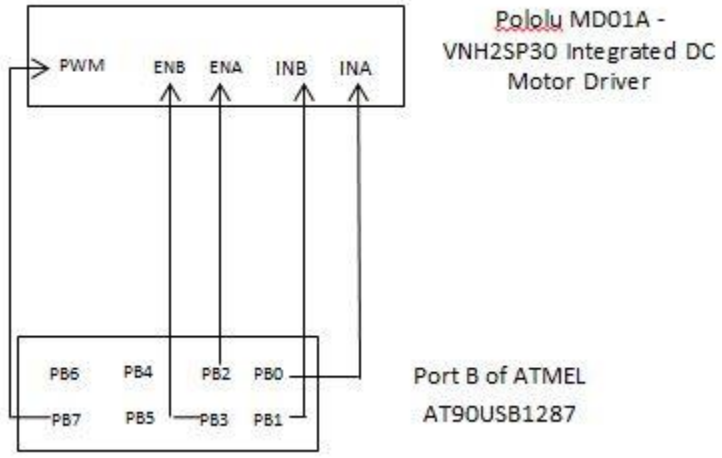

Figure 6: Connections for the DC motor with directions

In order to connect to the various sensors that will be used (optical and reflective), PORTD will be configured as an input port and will be connected to the various sensors. A simple schematic showing these connections is displayed below in figure 7.

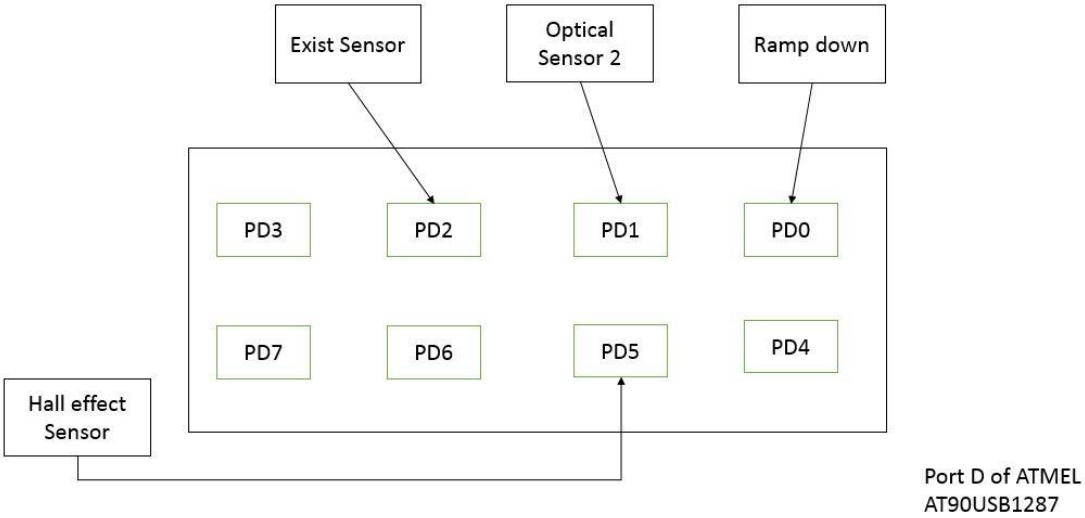

Figure 7: Connections to Various Sensors with directions (all input)

PORTC will primarily be used to display ADC results (by lighting up some LED's) from the sensors during the testing and debugging stages. PORTC will also be used to communicate error codes during these stages to simplify things when troubleshooting problems.

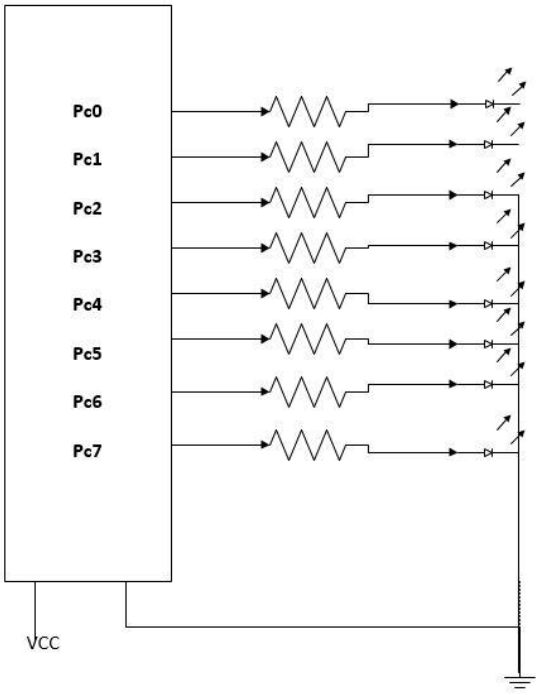

Figure 8. Circuit connection between Port C and 8 LEDs

PORTF will be used for the Reflection sensor to collect the reading of the objects surface reflection.

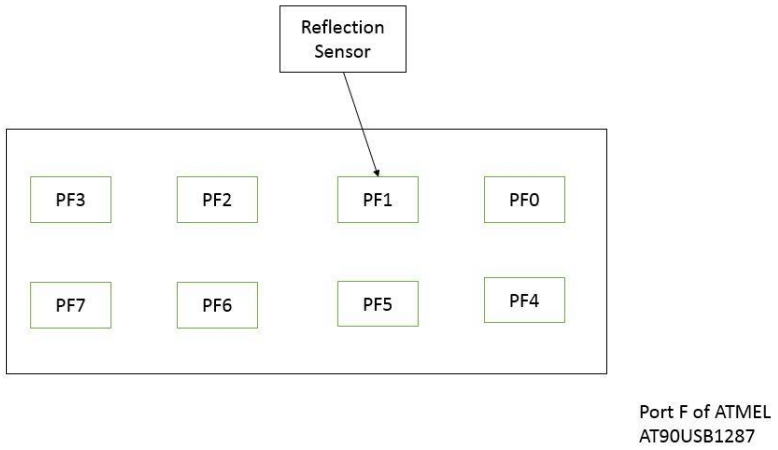

Figure 9. Circuit connection between Port F and reflection sensor

3.2 System Sorting Algorithm

In order to successfully sort all of the cylindrical objects, the following algorithm will be utilized as seen in figure 10: When the system starts up, the home position of the sorting tray must be located. The home position provides a reference position when the system starts up. The home position is found by continuously rotating the sorting tray using the stepper motor until a low signal from the Hall Effect signal is detected. The home position will be identified as the Black plastic bin. A flowchart that illustrates how the home position of the sorting tray is located is provided below in figure 11. The DC motor module will send a command to start the DC motor. When the cylindrical object crosses an optical sensor in one of the two utilized sensor stations, an external interrupt will be raised and the corresponding ISR will be called. If the object crosses the optical sensor located in sensor station 2, the corresponding ISR will be used to determine the optimal ADC measurement from the reflective sensor. A flowchart illustrating how the optimal ADC measurement is determined is provided in figure 12. If the object crosses the optical sensor in sensor station 3, the module that is used to control the DC motor will send the appropriate command to pause the motor since the object is at the end of the conveyor belt. At this point, the object type is determined (steel, aluminum, black plastic or white plastic) based on the optimal reflective sensor reading stored in the head index of the object array. A flowchart illustrating how the object type is determined is provided in figure 13. Once the object type is determined, the module that is used to operate the stepper motor will send the appropriate command (number of steps and direction) to rotate the sorting tray to the necessary position. A flowchart illustrating how the sorting tray is correctly positioned is provided in figure 14. The DC motor will be restarted and the object will be placed into the sorting tray. At this point if either all 48 objects have been processed or the user has pressed the ramp down button, the individual count for each object will be displayed to the user (PORTC[0-3]) and then the system will shut down. If the object type cannot be determined, the system will pause and an error code will be sent out PORTC (0xF0) indicating the error to the user. Further, if an external interrupt is detected that is not from sensor station 2 or sensor station 3, system will pause and an error code will be sent out PORTC (0xFF) indicating the error to the user that the error occurred. The system will then shut down.

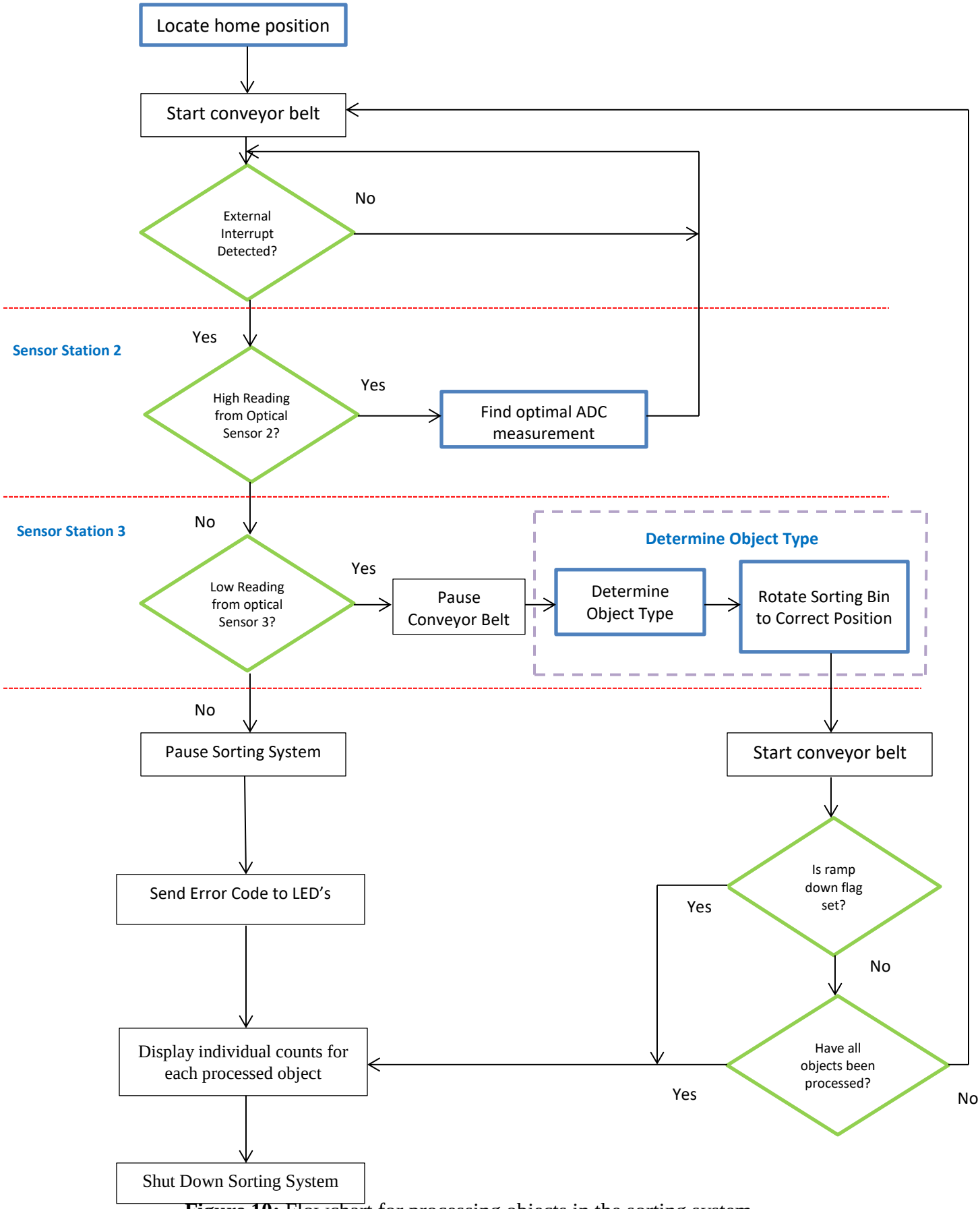

Figure 10: Flowchart for processing objects in the sorting system.

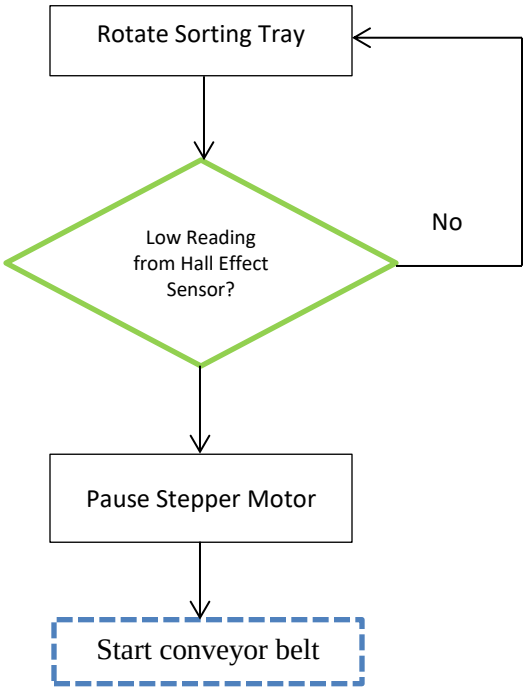

Figure 11: Flowchart for locating home position of sorting tray.

Since the determination of object type relies on the ADC measurement from the reflective sensor, it is crucial that the optimal ADC measurement is determined. The software approach to obtaining the optimal ADC measurement that was utilized is as follows: As soon as an object crosses the optical sensor located in sensor station 2, the corresponding ADC ISR will start a single ADC conversion. Once the conversion is finished, the ADC value will be compared to the current optimal value. The current optimal value is a temporary dummy variable that will store the most reflective (lowest) ADC measurement. If the ADC measurement is lower (more reflective) than the current optimal value, the ADC measurement will be passed to the dummy variable. If the object is still in front of the optical sensor, a new ADC conversion will begin and the optimization process will repeat. Otherwise, the optimal ADC measurement will be loaded into the tail index of the object array (See figure 4 for more details of the object array). The process for obtaining the optimal ADC measurement is illustrated below in figure 12.

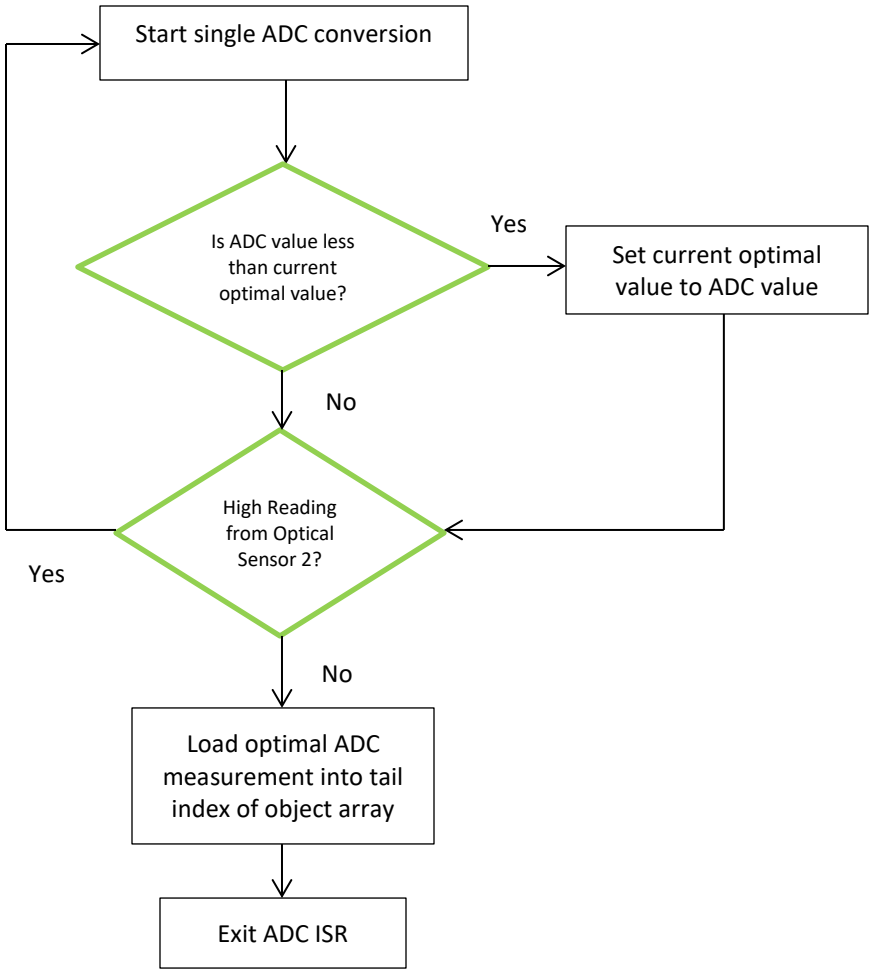

Figure 12: Flowchart for obtaining optimal ADC measurement.

As mentioned above, when the object enters sensor station 3, the conveyor belt is stopped and the object is processed. The processing stage is split up into two smaller stages. First the object type is determined (see figure 13) . Next, the sorting tray must reposition itself to the bin according to the object type determined (see Figure 14). In order to determine object type, the current ADC measurement being processed needs to be compared to a series of ADC measurement intervals. These intervals are determined beforehand during the calibration process and will be discussed in the calibration section. If the ADC measurement falls within one of these intervals, the object is classified according to the object type associated with the interval. Once the object is classified, the sorting tray will rotate to the position corresponding to the classified object. The process of determining the desired position of the sorting tray is described by the flowchart in figure 14 below.

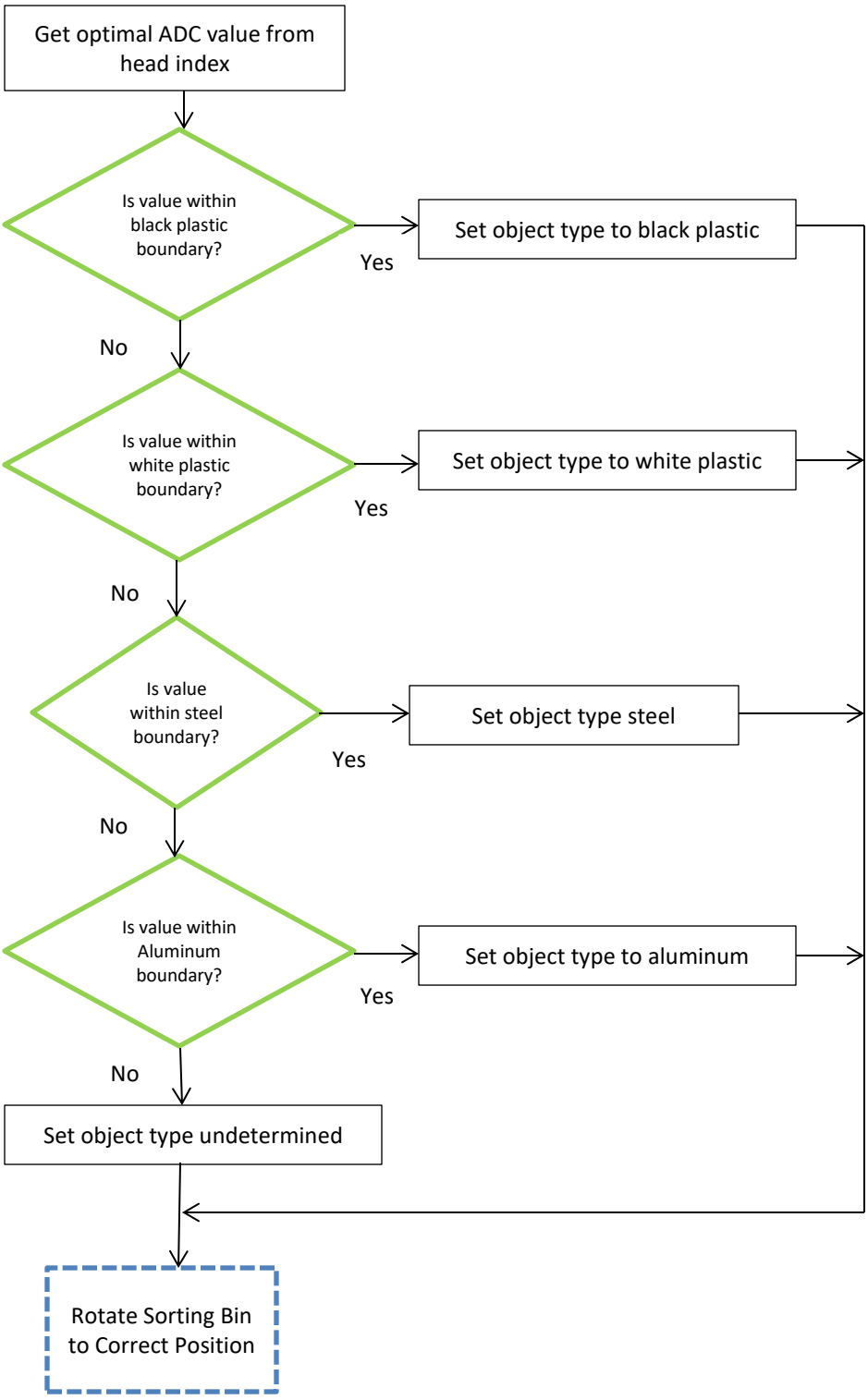

Figure 13: Flowchart for determining object type

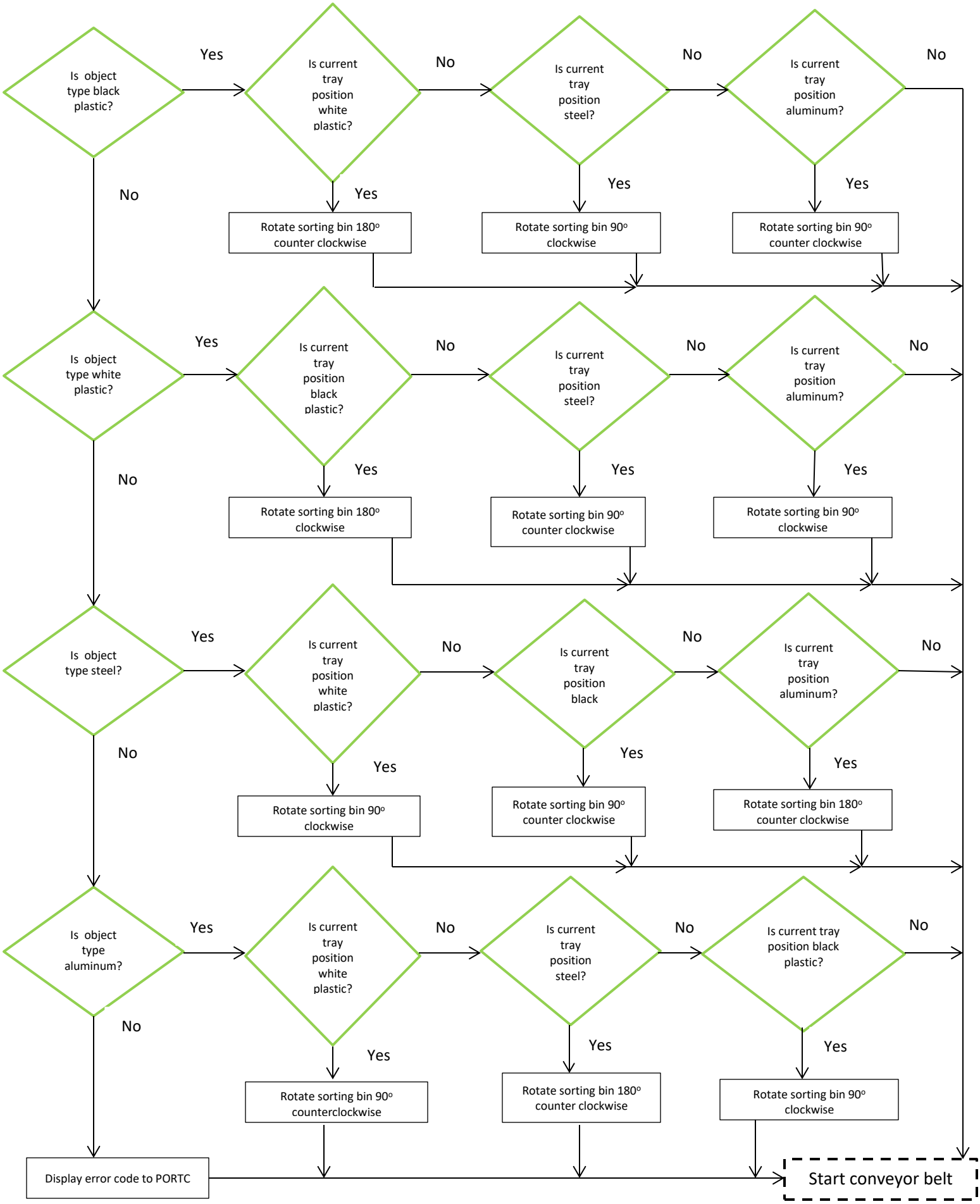
Is object type black plastic?
Yes
Is current tray position white plastic?
No
Is current tray position steel?
No
Is current tray position aluminum?
No
No
Yes
Rotate sorting bin 180° counter clockwise
Yes
Rotate sorting bin 90° clockwise
Yes
Rotate sorting bin 90° counter clockwise
Is object type white plastic?
Yes
Is current tray position black plastic?
No
Is current tray position steel?
No
Is current tray position aluminum?
No
No
Yes
Rotate sorting bin 180° clockwise
Yes
Rotate sorting bin 90° counter clockwise
Yes
Rotate sorting bin 90° clockwise
Is object type steel?
Yes
Is current tray position white plastic?
No
Is current tray position black
No
Is current tray position aluminum?
No
No
Yes
Rotate sorting bin 90° clockwise
Yes
Rotate sorting bin 90° counter clockwise
Yes
Rotate sorting bin 180° counter clockwise
Is object type aluminum?
Yes
Is current tray position white plastic?
No
Is current tray position steel?
No
Is current tray position black plastic?
No
No
Yes
Rotate sorting bin 90° counterclockwise
Yes
Rotate sorting bin 180° counter clockwise
Yes
Rotate sorting bin 90° clockwise
Display error code to PORTC
Start conveyor belt

Figure 14: Flowchart for determining the desired position of sorting tray

3.3 Calibration

In order to determine the type of object that is being sorted, the range of ADC measurements taken from the reflective sensor for each object type needed to be calibrated. In order to do this, a series of ADC measurements were taken for each object over a nuber of calibration trials and the ranges were documented. Each ADC measurement is 10 bits. In order to record each measurement, 8 bits (bit 0 – bit 7) were sent out PORTC and the remaining 2 bits (bit 8 – bit 9) were sent out to one of the LEDs on the MCU board. Below, figure 15 displays a histogram for each object indicating the corresponding ranges of ADC measurements. The raw data is also provided in table 1. The minimum and maximum ADC measurements for each object are also provided below in table 2. The calibration was conducted by taking ADC measurements over 12 trials for each object. It was found that the black plastic ranged from 968-987, white plastic ranged from 919-930, steel ranged from 562-693, and aluminum ranged from 65-90. These ranges are clearly visible in figure 15. Since it is highly possible that ADC measurements can fall outside of these ranges, decision boundaries were calculated using the following formula:

$$X1 = \frac{MinADCObjectA - MaxADCObjectB}{2}$$

where MinADCObjectA > MaxADCObjectB. The decision boundaries form regions that define object type in terms of reflectiveness. If an ADC measurement corresponding to an unidentified object falls within one of these regions, the object will be identified according to the region. To clarify, the decision boundary for black plastic and white plastic is calculated by subtracting the minimum ADC measurement for black plastic from the maximum ADC measurement for white plastic and dividing by 2. The decision boundaries for black plastic-white plastic, Steel - White Plastic, and Aluminum – Steel are highlighted below in table 3. These boundaries are also identified by the vertical dashed lines in figure 15.

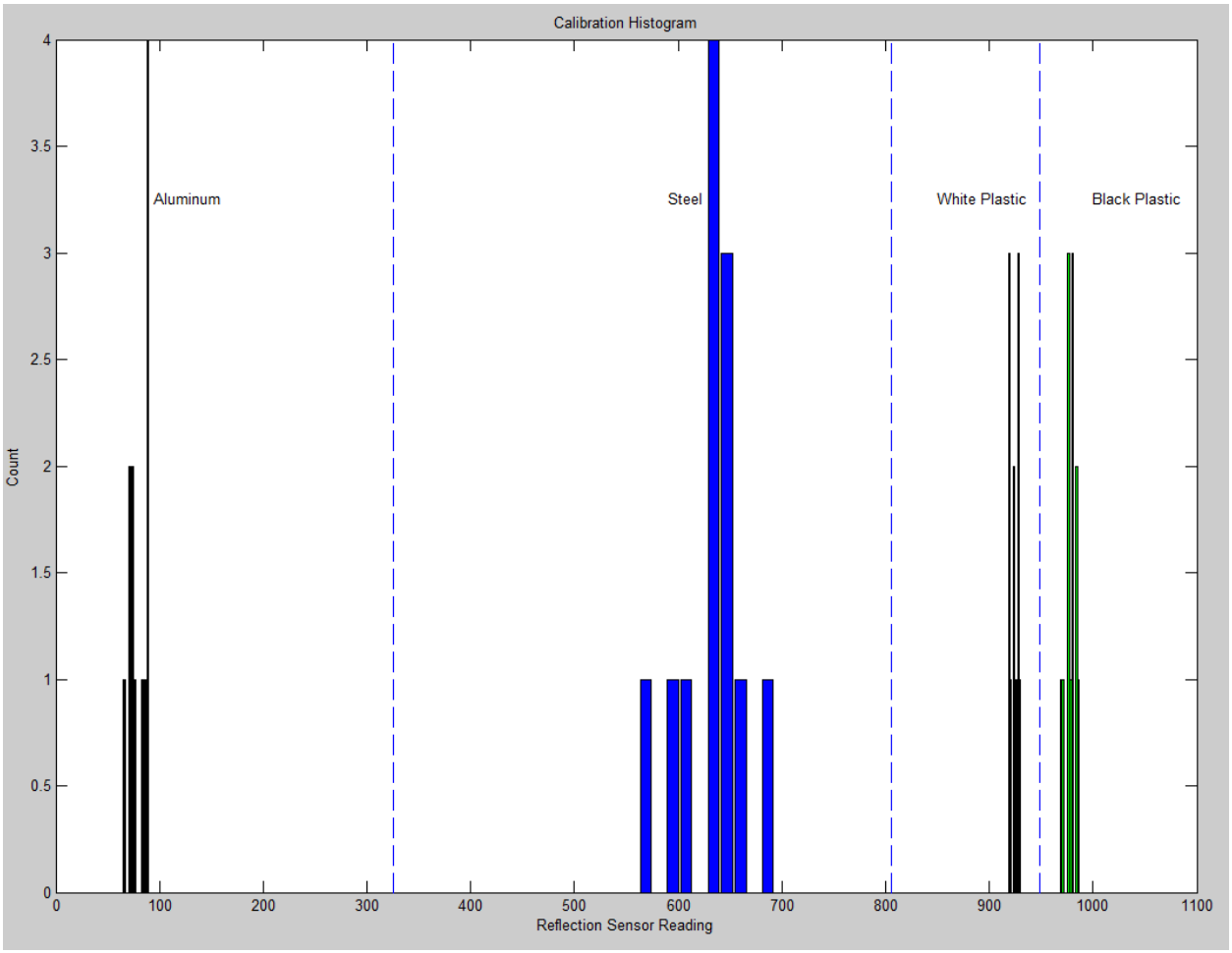

Figure 15: Calibration results over 12 runs for each object.

	Aluminum	Steel	White Plastic	Black Plastic
Trial 1	65	640	920	968
Trial 2	72	693	924	984
Trial 3	90	562	930	976
Trial 4	89	638	926	979
Trial 5	75	607	928	987
Trial 6	88	632	920	980
Trial 7	76	654	919	985
Trial 8	72	644	921	977
Trial 9	74	630	928	976
Trial 10	86	590	928	970
Trial 11	85	652	924	981
Trial 12	88	650	927	980

Table 1: ADC measurements during calibration for each object.

Object Type	Min Value	Max Value
Aluminum	65	90
Steel	562	693
White Plastic	919	930
Black Plastic	968	987

Table 2: Minimum and maximum readings for each object during calibration

Boundary Type	Value
Aluminum - Steel	326
Steel - White Plastic	806
White Plastic – Black Plastic	949

Table 3: Decision boundary values

4.0 Results

4.1 System Testing

The system was tested in numerous trials to validate the systems capability to sort objects based on ADC measurements from the reflective sensor. Each test consisted of measuring the total amount of time it takes to sort all 48 pieces along with the number of errors accumulated. An error is defined as when an object is placed into the wrong bin location. This occurs when an optimal ADC measurement falls within the wrong interval (see table 2). Below, table 4 displays the results for three trials. The average time it took to sort all 48 objects was 53.33 seconds with an average of 4 errors. The errors are due to the inability to sometimes distinguish between the black and white plastic objects. This is due to a small gap that exists between the intervals of black and white plastic. As a result, there is a greater chance that ADC measurements for the two objects will fall into the wrong region. However, the number of errors accumulated during a trial is always less than the maximum amount allowed (no more than 10 errors are allowed).

Trial #	Time (s)	Errors
1	52	3
2	59	6
3	55	3

Table 4: System testing results.

4.2 Final Demonstration

During the final demonstration, the sorting system was able to successfully sort the 48 required pieces. For each trial, the sorting system made less than the maximum amount of sorting errors allowed (10 errors are allowed). The best time for the system to sort 48 pieces was 48 seconds with only 3 accumulated errors. Table 5 displays the completion times and error count for each demonstration trial. Also, the pause button and ramp down switch were successfully demonstrated.

Trial #	Time (s)	Errors
1	51	3
2	48	3
3	50	5

Table 5: Final demonstration results

5.0 Conclusion

5.1 Limitations

During the project demo, the system took 48 seconds to sort 48 objects with three errors. The major bottle neck was the speed of the spin disk, since the delay of each step was set to be 15 ms. As a result, it takes 0.75 seconds for a 90^o rotation (50 steps) and 1.5 seconds for a 180^o rotation (100 steps). This is slower than desired. If the rate of rotation of the spin disk could be increased, then the sorting time could be decreased.

During the demo, the steel objects were brand new and the reflection values were pretty low compared with the white and black objects. Normally, the reflection values for steel objects are around 170 to 350. However, the ADC values for the steel objects were around 45 to 75 during the demo. The ADC values for aluminum were around 12 to 37. As a result, there was a very small gap for the system to distinguish between aluminum and steel. Consequently, one aluminum object was classified as steel during the demo.

5.2 Future Expansion

The stepper motor will apply the acceleration coding to improve the spinning speed installed of using the time delay in order to solve the bottle neck of the whole system.

If given more time, the inductive sensor (located at sensor station 1) would be utilized. The inductive sensor can be used to easily distinguish between steel and aluminum (steel is magnetic while aluminum is not). It can solve the problem of the close reading between steel and aluminum thus decreasing the possibility of classification errors.

Appendix A Code

A.1 Main Source Code

```
#include <stdlib.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#include "LinkedQueue.h"
#include "Timing.h"
#include "StepperMotor.h"
#include "PWM.h"
#include "DCMotor.h"
#include "ADC.h"

//---------------------------------
//used to test the status of a pin
//---------------------------------
enum PushButtonStatus
{
	STATUS_HIGH = 0x00,
	STATUS_LOW = 0x01
}typedef buttonStatus;

enum SystemStatus
{
	SYSTEM_PAUSED = 0x00,
	SYSTEM_ACTIVE = 0x01

}typedef systemStatus;

//-----------------------------
//forward function declarations
//-----------------------------
void SetupIO(void);
void SetupInterrupts(void);
void SetHomePosition(void);
void ProcessObject(void);
void Reset(void);

//---------------------------
//global variable declarations
//---------------------------
volatile buttonStatus pauseButton;
volatile buttonStatus rampDownButton;
volatile systemStatus status;
volatile dcMotorMode  dcMotor;
volatile int          upperReg;
volatile int          lowerReg;
volatile int          currentOptimalVal;
volatile int          counter;
volatile char         STATE;

//----------------
//threshold values
//----------------
volatile int           minWhPlstVal;
volatile int           maxWhPlstVal;
```

```
volatile int           minBlPlstVal;
volatile int           maxBlPlstVal;
volatile int           minStVal;
volatile int           maxStVal;
volatile int           minAlVal;
volatile int           maxAlVal;

//------------
//object array
//------------
int objectList[48];
volatile int head;
volatile int tail;
int          listLength;

//-----------------------
//stepper motor variables
//-----------------------
volatile trayPosition currentTrayPosition;
volatile Direction    currentStepperDirection;

int main()
{
       STATE = 0;

       //------------------------
       //variable initializations
       //------------------------
       pauseButton       = STATUS_HIGH;
       rampDownButton    = STATUS_HIGH;
       status            = SYSTEM_ACTIVE;
       currentOptimalVal = 0xFFFF;
       upperReg          = 0;
       lowerReg          = 0;
       counter           = 0;

       //-----------------------
       //setup the stepper motor
       //-----------------------
       step[0] = 48;
       step[1] = 6;
       step[2] = 40;
       step[3] = 5;
       halfTurn = 100;
       quarterTurn = 50;

       //-------------------------------------------
       //initialize reflective values from calibration
       //-------------------------------------------
       minWhPlstVal = 750;//WHITE PLASTIC
       maxWhPlstVal = 922;

       minBlPlstVal = 923;//BLACK PLASTIC
       maxBlPlstVal = 1100;

       minStVal     = 300;//STEEL
       maxStVal     = 749;
```

```
minAlVal     = 1;//ALUMINUM
maxAlVal     = 299;

//-----------
//object aray
//-----------
listLength = 48;
Reset();

//-------------------------
//setup the reference point
//-------------------------
currentTrayPosition     = POSITION_BLACK;
currentStepperDirection = COUNTERCLOCKWISE_ROTATION;

cli(); // Disables all interrupts

//------------------
//setup IO diretions
//------------------
SetupIO();

//----------------
//setup interrupts
//----------------
SetupInterrupts();

//---------
//setup ADC
//---------
SetupADC();

//---------
//setup PWM
//---------
SetupPWM();

// Enable all interrupts
sei(); // Note this sets the Global Enable for all interrupts

SetHomePosition();

OCR0A = 0x78;//initial duty cycle.Change as necessary

OperateDCMotor(DC_MOTOR_ROTATE_CCW);

goto POLLING_STAGE;

// POLLING STATE
POLLING_STAGE:

       switch(STATE){
              case (0) :
                     //-----------------------------
                     //poll for user to pause system
                     //-----------------------------
                     if(((PINA & 0x01) == 0) && pauseButton == STATUS_HIGH)
                     {
```

```
                                pauseButton = STATUS_LOW;//might be for testing
                                Timer(20);//waitout 'contact bounce'

                                if(status == SYSTEM_ACTIVE)//pause system
                                {
                                        status  = SYSTEM_PAUSED;
                                        dcMotor = DC_MOTOR_BRAKE_VCC;
                                        //-----------------------------
                                        //display the current object count
                                        //-----------------------------
                                        PORTC = head;
                                }
                                else//resume system
                                {
                                        status  = SYSTEM_ACTIVE;
                                        dcMotor = DC_MOTOR_ROTATE_CCW;
                                        //-----------------------------
                                        //display the current object count
                                        //-----------------------------
                                        PORTC = 0;
                                }
                                OperateDCMotor(dcMotor);

                        }
                        else if(((PINA & 0x01) == 1) && (pauseButton == STATUS_LOW))
                        {
                                Timer(20);//waitout 'contact bounce'
                                pauseButton = STATUS_HIGH;
                        }
                        goto POLLING_STAGE;
                        break; //not needed but syntax is correct
                case (1) :
                        goto MAGNETIC_STAGE;
                        break;
                case (2) :
                        goto REFLECTIVE_STAGE;
                        break;
                case (4) :
                        goto BUCKET_STAGE;
                        break;
                case (5) :
                        goto END;
                default :
                        goto POLLING_STAGE;
        }//switch STATE


MAGNETIC_STAGE:
        //Reset the state variable
        STATE = 0;
        goto POLLING_STAGE;

REFLECTIVE_STAGE:


        //Reset the state variable
        STATE = 0;
        goto POLLING_STAGE;
```

```
	BUCKET_STAGE:
		//-----------------------------------------------------
		//determine the type of object currently being processed
		//-----------------------------------------------------
		ProcessObject();

		OperateDCMotor(DC_MOTOR_ROTATE_CCW);

		//-----------------------------------------------------------
		//IF THE OBJECT COUNT IS AT ITS LIMIT (48) END THE PROGRAM
		//-----------------------------------------------------------
		if(head == listLength ||
		  ((rampDownButton == STATUS_LOW) && ((tail - head) == 0)))
		{
			STATE = 5;
		}
		else
		{
			STATE = 0;
		}
		goto POLLING_STAGE;

	END:
		Timer(500);
		OperateDCMotor(DC_MOTOR_BRAKE_VCC);

		//------------------------------
		//display the final object count
		//------------------------------
		PORTC = head;

		//----------------------------------
		//clear the list and other variables
		//----------------------------------
		Reset();
	return(0);

}
//-------------------------------------------------------------------------------------------
-
//---------------
//ISR definitions
//---------------
//-------------------------------------------------------------------------------------------
-
/*
	Set up the External Interrupt 0 Vector
	used to detect FALLING EDGE from ramp down push button

*/
ISR(INT0_vect){

	//-----------------------
	//for debouncing purposes
	//-----------------------
	Timer(5);
	if(((PIND & 0x01) == 0x00) && (rampDownButton == STATUS_HIGH))
```

```
	{
		rampDownButton = STATUS_LOW;
	}
}

/* Set up the External Interrupt 1 Vector
   used to detect RISING EDGE from optical sensor
   from sensor station 2
*/
ISR(INT1_vect){
	//-----------------------
	//for debouncing purposes
	//-----------------------
	//Timer(5);
	if((PIND & 0x02) == 2)
	{
		ADCSRA |= _BV(ADSC);
	}
	else
	{
		STATE = 0;
	}
}


/*
	Set up the External Interrupt 2 Vector
    used to detect FALLING EDGE from optical sensor
    from sensor station 3
*/
ISR(INT2_vect)
{
	Timer(5);
	if((PIND & 0x04) == 0)
	{
		OperateDCMotor(DC_MOTOR_BRAKE_VCC); //interrupt
		STATE = 4;//go to the bucket stage
	}
}

//--------------------------------------------------------------------------------
---------------
// If an unexpected interrupt occurs (interrupt is enabled and no handler is installed,
// which usually indicates a bug), then the default action is to reset the device by
jumping
// to the reset vector. You can override this by supplying a function named BADISR_vect
which
// should be defined with ISR() as such. (The name BADISR_vect is actually an alias for
__vector_default.
// The latter must be used inside assembly code in case <avr/interrupt.h> is not
included.
//--------------------------------------------------------------------------------
---------------
ISR(BADISR_vect)
{
	//stop motor and let user know that something is wrong
    OperateDCMotor(DC_MOTOR_BRAKE_VCC);
	for(int i = 0; i < 4; i++)
```

```
	{
		PORTC  = 0xFF;
		Timer(100);
		PORTC  = 0x00;
		Timer(100);
	}
	PORTC  = 0xFF;
}
//-------------------------------------------------------------------------------------------
//--------------------END OF ISR DEFINITIONS-----------------------------------------------
-------------



//-------------------------------------------------------------------------------------------
-------------
//---------------
//ADC definitions
//---------------
//-------------------------------------------------------------------------------------------
-------------
ISR(ADC_vect)
{
	if((ADC) < currentOptimalVal)
	{
		currentOptimalVal = (ADC);
		upperReg          = (ADCH << 6);
		lowerReg          = ADCL;
	}

	if((PIND & 0x02) == 2)//the object is in front of sensor
	{
		ADCSRA |= _BV(ADSC);//start a new ADC measurements b/c the objet is still
		//infront of the sensor
	}
	else if((PIND & 0x02) == 0)//the object is not in front of sensor
	{
		PORTC = lowerReg;//display ADCH portion
		PORTD = upperReg;//display ADCL portion

		//------------------------
		//store optimal ADC value
		//------------------------
		if(tail < listLength)
		{
			objectList[tail++] = currentOptimalVal;
		}
		currentOptimalVal = 0xFFFF;

	}
}
//-------------------------------------------------------------------------------------------
-------------
//--------------------END OF ADC DEFINITIONS-----------------------------------------------
-------------
```

```
//-------------------------------------------------------------------------------------
//--------------------
//function definitions
//--------------------
//-------------------------------------------------------------------------------------
void SetupIO(void)
{
	//---------------------------------
	//used for user input/ push buttons
	//---------------------------------
	DDRA = 0x00;
	//--------------------
	//pin 0 - 3 used for interrupts.
	//--------------------
	DDRD = 0xC0;
	//---------------------------------
	//used for ADC with optical sensors
	//---------------------------------
	DDRF = 0x00;
	//---------------------------------------
	//just use as a display. Light up the LEDs
	//---------------------------------------
	DDRC = 0xFF;
	//-------------------------------------
	//PORTE is for output to stepper motor
	//-------------------------------------
	DDRE = 0xFF;
	//-------------------------------------------------------------
	//PORT B is output. Used for output PWM and control dc motor
	//-------------------------------------------------------------
	DDRB = 0xFF;
	PORTB = 0x00;
	return;
}

void Reset(void)
{
	//--------------------
	//erase list contents
	//--------------------
	for(int i = 0; i < listLength; i++){objectList[i] = 0;}
	head = 0;
	tail = 0;
	return;
}


void ProcessObject(void)
{
	int        steps  = 0;
	objectType object = OBJECT_UNDEFINED;

	//-------------------------
	//DETERMINE THE OBJECT TYPE
	//-------------------------
	int val = objectList[head];
```

```
   if(objectList[head] >= minBlPlstVal)
   {
          object = PLASTIC_BLACK;
   }
   else if((objectList[head] < maxWhPlstVal) && (objectList[head] > minWhPlstVal))
   {
          object = PLASTIC_WHITE;
   }
   else if ((objectList[head] <= maxAlVal) && (objectList[head] > minAlVal))
   {
          object = METAL_ALUMINUM;
   }
   else if ((objectList[head] <= maxStVal) && (objectList[head] >= minStVal))
   {
      object = METAL_STEEL;
   }

   //-------------------------------------------------
   //DETERMINE WHERE TO POSITION THE SORTING TRAY
   //BASED ON THE CURRENT OBJECT CURRENT TRAY POSITION
   //-------------------------------------------------
if(object == METAL_STEEL)
   {
            if(currentTrayPosition == PLASTIC_BLACK)
            {
            steps = quarterTurn;
                  currentStepperDirection = COUNTERCLOCKWISE_ROTATION;
            }
            else if(currentTrayPosition == PLASTIC_WHITE)
            {
            steps = quarterTurn;
                  currentStepperDirection = CLOCKWISE_ROTATION;
            }
            else if(currentTrayPosition == METAL_ALUMINUM)
            {

            steps = halfTurn;
                  currentStepperDirection = COUNTERCLOCKWISE_ROTATION;
            }
            else
            {
            steps = 0;
            }
         currentTrayPosition = POSITION_STEEL;
   }
   else if (object == METAL_ALUMINUM)
   {
           if(currentTrayPosition == PLASTIC_BLACK)
            {
            steps = quarterTurn;
                  currentStepperDirection = CLOCKWISE_ROTATION;
            }
            else if(currentTrayPosition == PLASTIC_WHITE)
            {
            steps = quarterTurn;
                  currentStepperDirection = COUNTERCLOCKWISE_ROTATION;
            }
            else if(currentTrayPosition == METAL_STEEL)
```

```
          {

          steps = halfTurn;
                currentStepperDirection = COUNTERCLOCKWISE_ROTATION;
          }
          else
          {
          steps = 0;
          }
       currentTrayPosition = POSIITON_ALUMINUM;
}
else if (object == PLASTIC_WHITE)
{
         if(currentTrayPosition == PLASTIC_BLACK)
          {
               steps = halfTurn;
                currentStepperDirection = CLOCKWISE_ROTATION;
          }
          else if(currentTrayPosition == METAL_STEEL)
          {
               steps = quarterTurn;
                currentStepperDirection = COUNTERCLOCKWISE_ROTATION;
          }
          else if(currentTrayPosition == METAL_ALUMINUM)
          {

                steps = quarterTurn;
                currentStepperDirection = CLOCKWISE_ROTATION;
          }
          else
          {
               steps = 0;
          }
       currentTrayPosition = POSITION_WHITE;
}
else if(object == PLASTIC_BLACK)
{
         if(currentTrayPosition == METAL_STEEL)
          {
               steps = quarterTurn;
                currentStepperDirection = CLOCKWISE_ROTATION;
          }
          else if(currentTrayPosition == PLASTIC_WHITE)
          {
               steps = halfTurn;
                currentStepperDirection = COUNTERCLOCKWISE_ROTATION;
          }
          else if(currentTrayPosition == METAL_ALUMINUM)
          {

               steps = quarterTurn;
                currentStepperDirection = COUNTERCLOCKWISE_ROTATION;
          }
          else
          {
               steps = 0;
          }
          currentTrayPosition = POSITION_BLACK;
```

```
	}
	else
	{
		//LET USER KNOW THAT THERE IS AN ERROR. THE OBJECT TYPE IS UNDEFINED
		//THE CALIBRATED BOUNDARIES
		//FLASH SOME LED LIGHTS
		for(int i = 0; i < 5; i++)
		{
			PORTC = 0xF0;
			Timer(50);
			PORTC = 0x00;
			Timer(50);
		}
	}
	//---------------------------------------
	//point to the next index in object array
	//---------------------------------------
	head++;

	//-----------------------------------
	//rotate to the next sorting position
	//-----------------------------------
	RotateStepperMotor(currentStepperDirection,steps,15);
	return;
}


void SetHomePosition(void)
{
	while((PIND & 0b00010000) == 0b00010000)
	{
		RotateStepperMotor(CLOCKWISE_ROTATION,1,20);
	}
	return;
}


void SetupInterrupts(void)
{
	//-------------------------------------------------
	//This is to set up interrupt #1 - ramp down pin D0
	//detects whether user pressed ramp down button
	//-------------------------------------------------
	EICRA |= _BV(ISC01);

	//-------------------------------------------------
	//This is to set up interrupt #1 - optical sensor #2
	//detects a rising edge
	//-------------------------------------------------
	EICRA |= _BV(ISC10) | _BV(ISC11);

	//-------------------------------------------------
	//This is to set up interrupt #2 - optical sensor #3
	//detects a falling edge
	//-------------------------------------------------
	EICRA |= _BV(ISC21);
```

```
	// See page 96 - EIFR External Interrupt Flags...notice how they reset on their
	EIMSK |= 0b00000111;
	return;
}
//-------------------------------------------------------------------------------------
------------
//----------------------END OF FUNCTION DEFINITIONS-----------------------------------
------------
```

A.2 ADC Setup Source Code

```
#include <avr/interrupt.h>
#include "ADC.h"


void SetupADC()
{
	// config ADC ============================================================
	// by default, the ADC input (analog input is set to be ADC0 / PORTF0
	ADCSRA |= _BV(ADEN); // enable ADC
	ADCSRA |= _BV(ADIE); // enable interrupt of ADC

	//Setup three interrupts -> ADC2: activates pins PF0,PF1,PF2
	ADMUX |= _BV(REFS0) | _BV(MUX0);


	ADCSRB |= 0x80;
	return;
}
```

A.3 DC Motor Source Code

```
#include <avr/io.h>
#include "DCMotor.h"


//-------------------------------------------------
//Description: Operates the DC motor in three modes:
//             1) Rotate shaft in CW direction
//                      2) Rotate shaft in CCW direction
//                      3) Brake/Stop motor
//-------------------------------------------------
void OperateDCMotor(dcMotorMode mode)
{
	switch(mode)
	{
		case DC_MOTOR_ROTATE_CW:
			PORTB = 0x02;
		break;

		case DC_MOTOR_ROTATE_CCW:
			PORTB = 0x01;
		break;
```

```
		case DC_MOTOR_BRAKE_VCC:
		default:
			PORTB = 0x00;
		break;
	}
	return;
}
```

A.4 PWM Setup Source Code

```
#include "PWM.h"
#include <avr/interrupt.h>

//-------------------------------------------------------------------------------------------
-------------------------
//Description: Sets up and applies fast PWM.
//		The prescaler value is set to cs/8 as of now. This sets the frequency of
the signal to approx 500Hz
//		The duty cycle must be a fractional value between 0.00 and 1.00
//-------------------------------------------------------------------------------------------
-------------------------
void SetupPWM(void)
{
	//STEP 1
	//----------------
	//set the WGM bits
	//----------------
	TCCR0A |= _BV(WGM00);
	TCCR0A |= _BV(WGM01);

	//STEP 3
	TCCR0A |= _BV(COM0A1);

	//STEP 4
	//-------------------------
	//Set the Clock Select bits
	//-------------------------
	//NOTE THAT THIS VALUE SHOULD BE ADJUSTED DURING TESTING
	//the prescaler value is set to cs/64 for now
	TCCR0B |= _BV(CS01);

	//STEP 5
	//-------------------------------------------------------------------
	//set the value of the output compare register using the duty cycle
	//-------------------------------------------------------------------
	OCR0A = 0x00;//ADC_result

	return;
}
```

A.5 Stepper Motor Source Code

```
#include <avr/io.h>
```

```
#include "StepperMotor.h"


//-------------------------------------------------------------------------------------------
------
//Description: Rotates a stepper motor in the CW and CCW directions.
//             For a 90 degree rotation, pass 50 to numSteps.
//             For a 180 degree rotation, pass 100 to numSteps.
//             It is recommended to pass 20 to delay for a 20 ms rest between consecutive
steps
//-------------------------------------------------------------------------------------------
------
void RotateStepperMotor(Direction direction,int numSteps,int delay)
{
	const int MAX_INDEX = 3;
	const int MIN_INDEX = 0;

	switch(direction)
	{
		case CLOCKWISE_ROTATION:
			for(int i = 0; i < numSteps; i++)
			{
				currentStepperMotorIndex++;
				if (currentStepperMotorIndex > MAX_INDEX)
				{
				currentStepperMotorIndex = MIN_INDEX;

				}
				PORTE = step[currentStepperMotorIndex];
				Timer(delay);
			}
		break;
		case COUNTERCLOCKWISE_ROTATION:
			for(int i = 0; i < numSteps; i++)
			{
				currentStepperMotorIndex--;
				if(currentStepperMotorIndex < MIN_INDEX)
				{
					currentStepperMotorIndex = MAX_INDEX;
				}
				PORTE = step[currentStepperMotorIndex];
				Timer(delay);
			}
		break;
	}
	return;
}
```

A.6 Timer Source Code

```
#include "Timing.h"
#include <avr/interrupt.h>

void Timer(int count)
{
	int currentCount = 0;
```

```
	TCCR1B |= _BV(CS10);//setup timer
	/*
	This will set the WGM bits to 0100
	*/
	TCCR1B |= _BV(WGM12);

	/*
	Set the compare register for 1000 cycles = 1ms
	Timer currently works at 1MHz
	*/
	OCR1A = 0x03E8;

	/*
	Set the timer counter register to 0
	*/
	TCNT1 = 0x0000;

	/*
	Enable the output compare interrupt to be enabled
	*/
	//TIMSK1 = TIMSK1 | 0b00000010;

	/*
	Clear the timer interrupt flag and begin timing
	*/
	TIFR1 |= _BV(OCF1A);

	while(currentCount < count)
	{
		/*
		Test to see whether the Output Compare Match Flag has been set
		Note: This occurs when the value in TCNT1 matches the value in OCR1A
		*/
		if((TIFR1 & 0x02) == 0x02)
		{
			currentCount++;
			TIFR1 |= _BV(OCF1A);
		}
	}
	return;
}
```