```
/* ####################################################################
# PROJECT: Final Project
# NAME 1: Brodie Gould, V00973591
# NAME 2: Nick Gee, V00862631
# DESC: *add description*
# STARTED March 15th, 2022############################################## */

/* included libraries */
#include <avr/interrupt.h>
#include <avr/io.h>
#include <stdlib.h>
#include <stdio.h>
#include "LinkedQueue.h"
#include "lcd.h"

/* Global Variables */
volatile char STATE;
volatile unsigned char pauseButton   = 1;                                    //
  variable to treat the ESTOP like a pause button for purposes of running

/* DC motor belt variables */
volatile unsigned char dcBeltBrake    =  0b11111111;                         //
  bits that correspond to DC motor brake using VCC only, not GND
volatile unsigned char dcBeltBwd      =  0b10001101;                         //
  bits that correspond to DC motor rotating forwards
volatile unsigned char dcBeltFwd      =  0b10001110;                         //
  bits that correspond to DC motor rotating backwards
volatile unsigned char motorDirection =  0b10001101;                         //
  initialize the current belt motor direction to forwards

/* turntable stepper variables */
int stepRotation[4] =  {0b00110110, 0b00101110, 0b00101101, 0b00110101};     //
  create array with 4 different PWM steps, pulses two poles at once
volatile int stepCounter =0;                                                 //
  step counter varies from 0->3
volatile int turntableSteps      = 0;                                        // set
   variable to store the state of the current step
volatile unsigned char dutyCycle = 0xC0;                                     // set
   PWM. Alternative values are(50%=0x80, 60%=9A, 65%=41 70%=B4,75%=C0 80%=CD, 85%
  =D8,90%=E6,95%=F0, 100%==FF)

// global variables to use for the linked list
#define CW 1
#define CCW 0
#define BLK 1
#define STL 2
#define WHT 3
#define ALU 4



// Arrays modelling the Trapazoidal function. deprecated as S Curve had better
```

```
  runtime. Model done via excel
//const float accProfile100[100] =
  {154.44,148.89,143.33,137.78,132.22,126.67,121.11,115.56,110.00,104.44,98.89,93.33,8
  7.78,82.22,76.67,71.11,65.56,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,6
  0.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,6
  0.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,6
  0.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,6
  0.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,6
  0.00,65.56,71.11,76.67,82.22,87.78,93.33,98.89,104.44,110.00,115.56,121.11,126.67,13
  2.22,137.78,143.33,148.89,154.44};
//const float accProfile50[50] =
  {154.44,148.89,143.33,137.78,132.22,126.67,121.11,115.56,110.00,104.44,98.89,93.33,8
  7.78,82.22,76.67,71.11,65.56,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,60.00,6
  0.00,60.00,60.00,60.00,60.00,60.00,60.00,65.56,71.11,76.67,82.22,87.78,93.33,98.89,1
  04.44,110.00,115.56,121.11,126.67,132.22,137.78,143.33,148.89,154.44};


// Arrays modelling the S Curve we chose. Model done via excel
const float accProfile50[50] =
  {145.81,143.20,140.22,136.85,133.11,129.00,124.57,119.87,114.98,110.00,105.02,100.13
  ,95.43,91.00,86.89,83.15,79.78,76.80,74.19,71.92,69.98,68.32,66.91,65.73,64.74,64.74
  ,65.73,66.91,68.32,69.98,71.92,74.19,76.80,79.78,83.15,86.89,91.00,95.43,100.13,105.
  02,110.00,114.98,119.87,124.57,129.00,133.11,136.85,140.22,143.20,145.81};
const float accProfile100[100] =
  {149.72,148.08,146.22,144.11,141.76,139.14,136.25,133.11,129.71,126.08,122.25,118.26
  ,114.16,110.00,105.84,101.74,97.75,93.92,90.29,86.89,83.75,80.86,78.24,75.89,73.78,7
  1.92,70.28,68.84,67.59,66.50,65.55,64.74,64.04,63.44,62.93,62.49,62.12,61.80,61.53,6
  1.30,61.10,60.93,60.79,60.67,60.57,60.48,60.41,60.34,60.29,60.25,60.25,60.29,60.34,6
  0.41,60.48,60.57,60.67,60.79,60.93,61.10,61.30,61.53,61.80,62.12,62.49,62.93,63.44,6
  4.04,64.74,65.55,66.50,67.59,68.84,70.28,71.92,73.78,75.89,78.24,80.86,83.75,86.89,9
  0.29,93.92,97.75,101.74,105.84,110.00,114.16,118.26,122.25,126.08,129.71,133.11,136.
  25,139.14,141.76,144.11,146.22,148.08,149.72};


// reflective sensor RL
volatile unsigned int  ADC_RESULT_FLAG;
volatile unsigned int  ADC_RESULT =2000;            //set the ADC result to some
  arbitrarily large number to start

// Hall Effect sensor HE in turntable
volatile unsigned int STAGE_4_HE_FLAG = 0;          // Hall Effect sensor flag to
  indicate table position for table initialization

// count variables for each part (on screen A, S, W, B)
volatile unsigned int aluminumCount = 0;
volatile unsigned int steelCount    = 0;
volatile unsigned int whiteCount    = 0;
volatile unsigned int blackCount    = 0;

// count variables for the pause state count
volatile unsigned int pausealuminumCount = 0;
volatile unsigned int pausesteelCount   = 0;
```

```
volatile unsigned int pausewhiteCount   = 0;
volatile unsigned int pauseblackCount   = 0;
volatile unsigned int totalCount    = 0;
volatile unsigned int onBelt = 0;                   // variable used to track values
  still in the linked list, not yet sorted

// size of linked list variable
int sizeOfList = 0;

// variables to track the maximum ADC value for each part. Varies for each station
int ALUM_MAX = 100;
int STL_MAX = 700;
int WHITEPLASTIC_MAX =  930;
int BLACKPLASTIC_MAX = 1000;


// variable to track what position the table is in
volatile unsigned char tablePosition =0;    // empty variable
volatile unsigned char rotation =CW;        // for turntable efficiencies, set it
  initially to clockwise rotation

/* Function declarations */
void mcuTimer(int count);
void rampTimer();
void milliTimer(int count);
void generalConfig();
void pwmConfig();
int sortControl(int direction, int steps);
void sortStepper(int currentPosition, int desiredPosition);
void stepperHome();
void DCMotorControl(char beltState);
int classifyPart(int ADC_RESULT);


int main(int argc, char *argv[]){
    CLKPR = 0x80;                       // allow cpu clock to be adjusted
    CLKPR = 0x01;                       // sets system clock to 8MHz clk/2
    STATE = 0;                          // ready to poll

    // LCD initialization and get port readouts. note its a 16W * 2H display
    InitLCD(LS_BLINK|LS_ULINE);         // initialize LCD module
    LCDClear();                         // cler display

    /*FIFO Function*/
    link *head;                         // The ptr to the head of the queue
    link *tail;                         // The ptr to the tail of the queue
    link *newLink;                      // A ptr to a link aggregate data type
      (struct)
    link *rtnLink;                      // same as the above
    rtnLink = NULL;
    newLink = NULL;
    setup(&head, &tail);
```

```
    /*END FIFO Function*/

    cli();                              // Disables all interrupts
    generalConfig();                    // initialize port settings, interrupt
      settings, and ADC settings
    pwmConfig(dutyCycle);               // initialize the PWM to the duty cycle set in
       the global variable declarations
    sei();                              // Global Enable for all interrupts
    stepperHome();                      // get stepper homed at position BLACKPLASTIC
    DCMotorControl(dcBeltFwd);          // start belt in forward direction

        goto POLLING_STAGE;

        // POLLING STATE
        POLLING_STAGE: // State 0

        switch(STATE){
            case (0) :
            goto POLLING_STAGE;

            break;  // not needed but syntax is correct
            case (1) :
            goto PAUSE;
            break;
            case (2) :
            goto REFLECTIVE_STAGE;
            break;
            case (3) :
            goto SORTING_STAGE;
            break;
            case (4) :
            goto RAMP_DOWN;
            case (5):
            goto END;
            default :
            goto POLLING_STAGE;
        }//switch STATE

        PAUSE: //State 1
            DCMotorControl(dcBeltBrake);                    // stop belt

            onBelt = size(&head, &tail);                    // determine size of
              linked list
            pauseblackCount = blackCount;                   // pass total count values
               to the pause state values
            pausewhiteCount = whiteCount;
            pausesteelCount = steelCount;
            pausealuminumCount = aluminumCount;

            for (int j = 0; j<onBelt; j++){                 // loop through items in
              the linked list
                int listHead;
```

```
            listHead = head->e.itemCode;                // peak at head of linked list
            head = head->next; // go to next list item
            if (listHead == BLK){                       // remove that item from the sorted list, as it's still on the belt
                pauseblackCount--;
            } else if(listHead == WHT){
                pausewhiteCount--;
            }else if (listHead == STL){
                pausesteelCount--;
            }else if (listHead == ALU){
                pausealuminumCount--;
            }
        }// end for

        LCDWriteStringXY(0,0,"S:");                     // print off the current count
        LCDWriteIntXY(2,0,pausesteelCount,2);
        LCDWriteStringXY(4,0,"A:");
        LCDWriteIntXY(6,0,pausealuminumCount,2);

        LCDWriteStringXY(8,0,"W:");
        LCDWriteIntXY(10,0,pausewhiteCount,2);
        LCDWriteStringXY(12,0,"B:");
        LCDWriteIntXY(14,0,pauseblackCount,2);

        LCDWriteStringXY(0,1,"ON BELT:");               // print off the current size of the linked list (unsorted pieces)
        LCDWriteIntXY(9,1,onBelt,1);

        while(pauseButton == 0);                        // wait to reset
        ADC_RESULT =2000;                               // set ADC result high so it reads properly after pausing
        DCMotorControl(dcBeltFwd);                      // restart belt

        STATE = 0;                                      // back to polling, restart belt
    goto POLLING_STAGE;

    REFLECTIVE_STAGE: // State 2
    // logic for characterizing the item and storing it in the linked list
        initLink(&newLink);                             // initialize new link connection
        newLink->e.itemCode = (classifyPart(ADC_RESULT));   // classify item and put value into itemCode
        enqueue(&head, &tail, &newLink);                // take data and create new link
        // LCDWriteIntXY(0,1,ADC_RESULT,4);             // ADC READOUT VALUES FOR TESTING
         //LCDWriteStringXY(7,1,"<-ADC VAL");           // ADC READOUT VALUES FOR TESTING
        ADC_RESULT=2000;                                // use some
```

```
            arbitrarily large number to reset the ADC high

        STATE = 0;          // reset state once part is done being classified
        goto POLLING_STAGE;


        SORTING_STAGE:  //State 3
        // belt needs to slow, turntable needs to move to correct position, and belt
          needs to feed the part in
            DCMotorControl(dcBeltBrake);                        // stop the belt

            sortStepper(tablePosition, head->e.itemCode);       // call sorting
              function with current list head, and previous list head

            dequeue(&tail, &head, &rtnLink);                    // remove link @ head
              of linked list
            free(rtnLink);                                      // free up memory
              after removing list item
            DCMotorControl(dcBeltFwd);                          // restart belt in
              forward direction

        STATE = 0;      // go back to POLLING_STAGE
        goto POLLING_STAGE;

        RAMP_DOWN: // State 4
        // process the remaining parts on the belt, then readout the total count of
          parts
            cli();                                              // disable all
              interrupts
            DCMotorControl(dcBeltBrake);                        // stop the belt
            LCDClear();
            LCDWriteStringXY(0,0,"S:");                         // print count of all
              sorted items
            LCDWriteIntXY(2,0,steelCount,2);
            LCDWriteStringXY(4,0,"A:");
            LCDWriteIntXY(6,0,aluminumCount,2);

            LCDWriteStringXY(8,0,"W:");
            LCDWriteIntXY(10,0,whiteCount,2);
            LCDWriteStringXY(12,0,"B:");
            LCDWriteIntXY(14,0,blackCount,2);


            onBelt = size(&head, &tail);                        // calculate sizeof
              list, return size as an int
            LCDWriteStringXY(0,1,"ON BELT:");
            LCDWriteIntXY(10,1,onBelt,1);

        STATE = 5;  // go back to END
        goto END;

        END: // State 5
```

```
        // Stop everything here...'MAKE SAFE'
            PORTC = 0xFF;                                   // all red LED's to
              indicate program ended

        STATE = 5;
        return(0);                                          // end
}

// ISR0 for exit optical sensor, EX, uses PD0, pin21
// exit optical sensor is active low
ISR(INT0_vect){
      if((PIND &0b00000001) == 0b00000000){                 // initial compare
        statement to detect input
          STATE = 3;                                        // go to SORTING State

      }
}
// ISR1 for entry optical sensor, OR. uses PD1, pin 20
ISR(INT1_vect){
   if ((PIND & 0x02) == 0x02){                              // if sensor is
     detecting a part
         ADCSRA |= _BV(ADSC);                               // start single ADC
           conversion
    }
}

// ISR2 for right PB. uses PD3, pin 19 *RISING EDGE*
// ISR for generating an PAUSE condition
ISR(INT2_vect){
       //PAUSE LOGIC
        if((PIND &0b0000100) == 0b00000100){                // initial compare
          statement to detect button press
            mcuTimer(20);                                   // debounce delay
            pauseButton = (pauseButton+1)%2;                // flip button state
            STATE = 1;                                      // go to PAUSE state

            while((PIND &0b00000100) == 0b00000100);        // check to see if
              button is released
            mcuTimer(20);                                   // debounce delay
        }
}

// ISR3 for left PB. uses PD3, pin 18 *FALLING EDGE*
// function to handle RAMPDOWN condition. let belt run until the linked list is empty,
   then end the program
ISR(INT3_vect){
    //RAMP DOWN
        if((PIND &0b00001000) == 0b00000000){               // initial compare
          statement to detect button press
            mcuTimer(20);                                   // debounce delay
            rampTimer();                                    // use separate timer
              to let belt run idle before stopping
```

```
            while((PIND &0b00001000) == 0b00001000);           // check to see if
              button is released
            mcuTimer(20);                                      // debounce delay

        }
}

// ISR5 for Hall effect sensor (HE) built into the turntable PE5, pin 3
ISR(INT5_vect){
    STAGE_4_HE_FLAG = 1;                    // trigger to indicate turntable HE sensor
       is in position
}
// Timer3 interrupt routine for the rampdown delay
ISR(TIMER3_COMPA_vect){
    STATE = 4;                              // when TIM3 OCR == set value, goes to
      rampdown state
}

// ADC interrupt for classifying parts
ISR(ADC_vect){
    if ((ADC) < ADC_RESULT){
        ADC_RESULT = ADC;                   // maintain the lowest ADC reading for
          part detection
    }
    if((PIND & 0x02) == 0x02){              // object is in front on ADC sensor
        ADCSRA |= _BV(ADSC);                // start new ADC measurement
    } else if((PIND & 0x02) == 0x00){       // object is not in front of sensor
        STATE = 2;                          // goto reflective state
    }
}

// in case of bugs, this ISR is called to display flashing lights
ISR(BADISR_vect){
    PORTC = 0b10101010;     //blink state 1
    mcuTimer(100);          //time delay
    PORTC = 0b01010101;     //blink state 2
    mcuTimer(100);          //time delay
}//end ISR BADISR_vect

void generalConfig(){
    // IO configuration
    DDRA = 0xFF;                                                    // set all port A
      pins as output for the stepper motor
    DDRB = 0xFF;                                                    // set all port B
      pins as output for DC motor drive
    DDRC = 0xFF;                                                    // set all port C
      pins as output, LCD/LED's for debugging
    DDRD = 0xF0;                                                    // set rightmost 4
       pins, PORTD(0,3) as output, rightmost pins (4-7) = input for INT 2 & INT 3
    DDRE = 0x00;                                                    // set all port E
      pins as input, for interrupts
    DDRF = 0x00;                                                    // set all port F
```

```
      pins as inputs, with the ADC using port F for interrupts
    DDRL = 0xFF;                                                    // use port L to
      display blinky lights

    // interrupt configuration
    EIMSK |= _BV(INT0)|_BV(INT1)|_BV(INT2)|_BV(INT3)|_BV(INT5);     // enable
      interrupts 0-5
    EICRA |= _BV(ISC01);                                            // INT0 falling
      edge
    EICRA |= (_BV(ISC11)| _BV(ISC10));                              // INT1 rising
      edge
    EICRA |= (_BV(ISC20) | _BV(ISC21));                             // INT2 rising
      edge
    EICRA |= _BV(ISC31);                                            // INT3 falling
      edge
    EICRB |= _BV(ISC51);                                            // INT5 falling
      edge

    // ADC configuration
    ADCSRA |= _BV(ADEN);                                            // enable ADC
    ADCSRA |= _BV(ADIE);                                            // enable ADC
      interrupts
    ADMUX  |= _BV(REFS0);                                           // AVcc with
      external cap at AREF pin, Table 26-3 has more options
    ADMUX  |= _BV(MUX0);                                            // select ADC1
      channel
    return;
}//end interruptConfig

//PWM configuration
void pwmConfig(int dutyCycle){
    TCCR0A |=_BV(WGM01) |_BV(WGM00);                                // set timer
      counter control register A WGM01:0 bit to 1 to enable fast PWM mode
    TCCR0A |=_BV(COM0A1);                                           // enable compare
      output mode for fast PWM to clear output compare OC0A on compare match to set
      OC0A at bottom (non-inverting)
    TCCR0B |=_BV(CS01);                                             // timer counter
      control register B so that the clock is scaled by clk/64 to moderate the DC
      motor speed
    OCR0A = dutyCycle;                                              // set output
      compare register to the ADC's result
    return;
}

// Tidier way of controlling the way the motor runs. Either FWD or STOP
void DCMotorControl(char beltState){
    PORTB = beltState;
}

// function to home the turntable that rotates N number of steps until the HE sensor
  is set, and at the BLACKPLASTIC drop position
void stepperHome(){
```

```
    while (STAGE_4_HE_FLAG == 0 ){
            //run clockwise routine
                stepCounter++;                                  // increment step
                  counter
                if(stepCounter > 3){
                    stepCounter = 0;                            // reset array index
                      when it reaches the end
                }//end if
                PORTA = stepRotation[stepCounter];              // cycle through the
                  step rotation array
                milliTimer(200);
        }//endwhile
    tablePosition = BLK;                                        // table is in black
      position
    return;
}

 //Stepper Control Function
 // function that is passed the desired direction and steps, and runs the S Curve
   array, and drops the part before it finishes its rotation for efficiency
 int sortControl(int direction, int steps){
    //each step is 1.8 degrees, so scale degrees to follow the stepper. adjust for %5
      tolerance

     if (steps == 100){                                         // if a 180 degree
       turn is needed
        //array of values mapped to an S Curve acceleration over 100 steps (180
          degrees)
         if (direction ==CCW){
             for (int j=0; j< steps-5; j++){
                 stepCounter++;                                 // increment step
                   counter
                 if(stepCounter > 3){
                     stepCounter = 0;                           // reset array index
                      when it reaches the end
                 }//end if
                 PORTA = stepRotation[stepCounter];             // cycle through the
                   step rotation array
                 milliTimer(accProfile100[j]);                  // delay using first
                   95 steps of S Curve
             }//close for loop

             DCMotorControl(dcBeltFwd);                         // pulse belt long
               enough to unload part, then finish turntable steps
             mcuTimer(5);
             DCMotorControl(dcBeltBrake);

             for (int k=95; k<steps; k++){
                 stepCounter++;                                 // increment step
                   counter
                 if(stepCounter > 3){
                     stepCounter = 0;                           // reset array index
```

```
                    when it reaches the end
                }//end if
                PORTA = stepRotation[stepCounter];              // cycle through the
                  step rotation array
                milliTimer(accProfile100[k]);                   // delay using last 5
                  steps of S Curve
            }
        } else if (direction ==CW){
            for (int j=0; j< steps-5; j++){
                stepCounter--;                                  // decrement step
                  counter
                if(stepCounter < 0 ){
                    stepCounter = 3;                            // reset array index
                     when it reaches the end
                } //end if stepcount=0
                PORTA = stepRotation[stepCounter];              // cycle through the
                  step rotation array
                milliTimer(accProfile100[j]);                   // delay using first
                  95 steps of S Curve
            }//end for step<j

            DCMotorControl(dcBeltFwd);                          // pulse belt long
              enough to unload part, then finish turntable steps
            mcuTimer(5);
            DCMotorControl(dcBeltBrake);
            for (int k=95; k< steps; k++){
                stepCounter--;                                  // decrement step
                  counter
                if(stepCounter < 0 ){
                    stepCounter = 3;                            // reset array index
                      when it reaches the end
                } //end if stepcount=0
                PORTA = stepRotation[stepCounter];              // cycle through the
                  step rotation array
                milliTimer(accProfile100[k]);                   // delay using last 5
                  steps of S Curve
                 }
         }//end if direction = 0

      } else if (steps ==50){                                   // elseif a 90 degree
        turn is needed
        //array of values mapped to an S Curve acceleration over 50 steps (90 degrees)
        if (direction ==CCW){
                    for (int j=0; j< steps-5; j++){
                        stepCounter++;                          // increment step
                      counter
                        if(stepCounter > 3){
                            stepCounter = 0;                    // reset array index
                      when it reaches the end
                        }//end if
                       PORTA = stepRotation[stepCounter];       // cycle through the
                      step rotation array
```

```
                    milliTimer(accProfile50[j]);
                }// end for k

                DCMotorControl(dcBeltFwd);                  // pulse belt long
                  enough to unload part, then finish turntable steps
                mcuTimer(5);
                DCMotorControl(dcBeltBrake);

                for(int k=45; k<steps; k++){
                    stepCounter++;                          // increment step
                  counter
                    if(stepCounter > 3){
                        stepCounter = 0;                    // reset array index
                  when it reaches the end
                    }//end if
                    PORTA = stepRotation[stepCounter];      // cycle through the
                  step rotation array
                    milliTimer(accProfile50[k]);            // delay using last 5
                  steps of S Curve
                }// end for k

        } else if (direction ==CW){
                //run clockwise routine
                for (int j=0; j< steps-5; j++){
                    stepCounter--;                          // decrement step
                      counter
                    if(stepCounter < 0 ){
                        stepCounter = 3;                    // reset array index
                      when it reaches the end
                    } //end if stepcount=0
                    PORTA = stepRotation[stepCounter];      // cycle through the
                      step rotation array
                    milliTimer(accProfile50[j]);            // delay using first
                      45 steps of S Curve
                }//end for step<j

                 DCMotorControl(dcBeltFwd);                 // pulse belt long
                   enough to unload part, then finish turntable steps
                 mcuTimer(5);
                 DCMotorControl(dcBeltBrake);

                for (int k=45; k< steps; k++){
                    stepCounter--;                          // decrement step
                      counter
                    if(stepCounter < 0 ){
                        stepCounter = 3;                    // reset array index
                      when it reaches the end
                    } //end if stepcount=0
                    PORTA = stepRotation[stepCounter];      // cycle through the
                      step rotation array
                    milliTimer(accProfile50[k]);            // delay using last 5
                      steps of S Curve
```

```
                }//end for step<k
            }// end 50 steps
        }
        return (0);
    } //end of sortControl function

// function to determine how much the stepper motor has to correct, given the difference between the current position and the desired position
void sortStepper(int currentPosition, int desiredPosition){
    int delta = (desiredPosition - currentPosition);
    if ((delta == 3) || (delta == -1)){                 // if the difference is 90 degrees CCW away
        sortControl(CCW,50);                            // rotate 50 ticks (90 degrees) CCW
        rotation = CCW;
        mcuTimer(80);                                   // time delay for gradual S Curve
    } else if ((delta == -3 ) || (delta == 1)){         // if the difference is 90 degrees CW away
        sortControl(CW,50);                             // rotate 50 ticks (90 degrees) CW
        rotation = CW;
        mcuTimer(80);                                   // time delay for gradual S Curve
    } else if ((delta == -2) || (delta == 2)){          // if the difference is 180 degrees away
        sortControl(rotation,100);                      // rotate 100 ticks (180 degrees) CW
        mcuTimer(80);                                   // time delay for gradual S Curve
    } else {
        sortControl(CW,0);                              // if table is at correct position, stay
    }// end if else
    tablePosition = desiredPosition;                    // update delta
    return;
}// end sortStepper

// function to classify item based on ADC reading
int classifyPart(int ADC_RESULT){
    int classPart =0;
    if ((ADC_RESULT <= ALUM_MAX)){
        classPart = ALU;
        aluminumCount +=1;
        totalCount++;
        } else if(ADC_RESULT <= STL_MAX){
        classPart = STL;
        steelCount +=1;
        totalCount++;
        } else if (ADC_RESULT <= WHITEPLASTIC_MAX){
        classPart = WHT;
        whiteCount +=1;
```

```
        totalCount++;
        } else if (ADC_RESULT <= BLACKPLASTIC_MAX){
        classPart = BLK;
        blackCount +=1;
        totalCount++;
    }
    return classPart;
}// end sortedPart

// clock functions
// timer that delays  for the count in millis
void mcuTimer(int count){
    int i=0;                            //counting variable
    TCCR1B |= _BV(CS11);                //clock prescalar by clk/8 = 1MHz
    TCCR1B |= _BV(WGM12);               //clear OCR1 on compare match, set output low
    OCR1A = 0x03E8;                     // write output compare register to hex value
      of 1000
    TCNT1 = 0x0000;                     //set the initial timer/counter value to 0
    while(i<count){                     //loop to check and see if the passed
      millisecond value is equal to our interrupt flag
        if ((TIFR1 & 0x02 ) == 0x02){   //*4* time comparison if
                TIFR1 |= _BV(OCF1A);    //set timer/counter interrupt flag so the
                  interrupt can execute
                i++;                    //increment
        }//*4*end if
    }//*2*end while loop comparing out count up case
    return; //exit timer function
}// end clock function 1

// secondary timer that relies on a separate interrupt status
void rampTimer(){
    PORTL=0xFF;                         //indicate rampTimer is active
    TCCR3B |= _BV(WGM32);               //clear OCR3 on compare match, set output low
    TCCR3B |= _BV(CS30) | _BV(CS32);    //clock prescalar by clk/1024 from prescalar,
      8MHz/1024 = 7.3kHz
    OCR3A = 0xFFFF;                     // write output compare register to hex value
      of 15
    TCNT3 = 0x0000;                     //set the initial timer/counter value to 0
    TIMSK3 |=_BV(OCIE3A);               //set timer/counter output compare A match
      interrupt enable
    TIFR3 |= _BV(OCF3A);                //set timer/counter flag register = 1 so the
      flag executes when interrupt flag TCNT1 == OCRA1
    return; //exit timer function
}//end clock function 1

// third timer downscaled further to allow for smoother operation of the turntable
void milliTimer(int count){
    int i=0;                            //counting variable
    TCCR5B |= _BV(CS51);                //clock prescalar by clk/8
    TCCR5B |= _BV(WGM52);               //clear OCR1 on compare match, set output low
    OCR5A = 0x064;                      // write output compare register to hex value
      of 100
```

```
    TCNT5 = 0x0000;                    //set the initial timer/counter value to 0
    while(i<count){                    //loop to check and see if the passed
      millisecond value is equal to our interrupt flag
        if ((TIFR5 & 0x02 ) == 0x02){  //*4* time comparison if
                TIFR5 |= _BV(OCF5A);   //set timer/counter interrupt flag so the
                  interrupt can execute
                i++;                   //increment
        }//*4*end if
    }//*2*end while loop comparing out count up case
    return; //exit timer function
}// end clock function 1

/*************************** Linked List Functions
  ****************************************/

/
  ********************************************************************************
  **
* DESC: initializes the linked queue to 'NULL' status
* INPUT: the head and tail pointers by reference
*/
void setup(link **h,link **t){
    *h = NULL;      /* Point the head to NOTHING (NULL) */
    *t = NULL;      /* Point the tail to NOTHING (NULL) */
    return;
    }/*setup*/

/
  ********************************************************************************
  **
* DESC: This initializes a link and returns the pointer to the new link or NULL if
  error
* INPUT: the head and tail pointers by reference
*/
void initLink(link **newLink){
    //link *l;
    *newLink = malloc(sizeof(link));
    (*newLink)->next = NULL;
    return;
    }/*initLink*/

/
  ********************************************************************************
  ****
*  DESC: Accepts as input a new link by reference, and assigns the head and tail
*  of the queue accordingly
*  INPUT: the head and tail pointers, and a pointer to the new link that was created
*/
/* will put an item at the tail of the queue */
void enqueue(link **h, link **t, link **nL){
    if (*t != NULL){
        /* Not an empty queue */
```

```
        (*t)->next = *nL;
        *t = *nL; //(*t)->next;
        }/*if*/
        else{
            /* It's an empty Queue */
            //(*h)->next = *nL;
            //should be this
            *h = *nL;
            *t = *nL;
            }/* else */
            return;
    }/*enqueue*/

/ ↵
  ********************************************************************************↵
  **
* DESC : Removes the link from the head of the list and assigns it to deQueuedLink
* INPUT: The head and tail pointers, and a ptr 'deQueuedLink'
*        which the removed link will be assigned to
*/
/* This will remove the link and element within the link from the head of the queue */
void dequeue(link ** t, link **h, link **deQueuedLink){
    /* ENTER YOUR CODE HERE */
    *deQueuedLink = *h; // Will set to NULL if Head points to NULL
    /* Ensure it is not an empty queue */
    if (*h != NULL){
        *h = (*h)->next;
        if (*h == NULL){
            *t = NULL;
             printf("Linked List is Empty");
        }
    } else{
        /* It's an empty Queue */
        *t = NULL;
        printf("Linked List is Empty");
        }/* else */
    return;
}/*dequeue*/

/ ↵
  ********************************************************************************↵
  **
* DESC: Peeks at the first element in the list
* INPUT: The head pointer
* RETURNS: The element contained within the queue
*/
/* This simply allows you to peek at the head element of the queue and returns a NULL ↵
  pointer if empty */
element firstValue(link **h){
    return((*h)->e);
}/*firstValue*/
```

```
/                                                                                ↵
  ********************************************************************************↵
  **
* DESC: deallocates (frees) all the memory consumed by the Queue
* INPUT: the pointers to the head and the tail
*/
/* This clears the queue */
void clearQueue(link **h, link **t){
    link *temp;
    while (*h != NULL){
    temp = *h;
    *h=(*h)->next;
    free(temp);
    }/*while*/
    /* Last but not least set the tail to NULL */
    *t = NULL;
    return;
}/*clearQueue*/

/                                                                                ↵
  ********************************************************************************↵
  **
* DESC: Checks to see whether the queue is empty or not
* INPUT: The head pointer
* RETURNS: 1:if the queue is empty, and 0:if the queue is NOT empty
*/
/* Check to see if the queue is empty */
char isEmpty(link **h){
    /* ENTER YOUR CODE HERE */
    return(*h == NULL);
}/*isEmpty*/

/                                                                                ↵
  ********************************************************************************↵
  **
* DESC: Obtains the number of links in the queue
* INPUT: The head and tail pointer
* RETURNS: An integer with the number of links in the queue
*/
/* returns the size of the queue*/
int size(link **h, link **t){
    link    *temp;          /* will store the link while traversing the queue */
    int     numElements;
    numElements = 0;
    temp = *h;          /* point to the first item in the list */
    while(temp != NULL){
        numElements++;
        temp = temp->next;
        }/*while*/
        return(numElements);
}/*size*/
```

```
/*
Wiring Guide
HE SENSOR - pin3 INT5 something
OR SENSOR - pin20 ISR1
RL SENSOR - pinA1, ADC1
EX SENSOR - Pin21, ISR0
CONNECT GROUND PIN
*/
```