Department of Mechanical Engineering

MECH 458/554: Mechatronics

Kunwu Zhang(Professor)

Final Project

Tom Boddington - V00932207
Colton Hyland - V00806034

Abstract

For this term project we propose our solution to the specified sorting problem for the provided apparatus by way of a microcontroller. The variety of parts to be sorted and were differentiated and processed using apparatus components sequentially. The apparatus consists of a conveyor belt, stepper motor and multiple sensors. Each of these systems were implemented over the span of the course. The challenge ultimately was the implementation of these systems in the sorting code to achieve the specified requirements.

The software and hardware interfacing are detailed to outline the solution of the project challenge. Specifically, the MCU interface with the various apparatus components is displayed along with a discussion of the sorting algorithm and its implementation.

Our solution to this problem is outlined and discussed within certain limitations and conventions of the apparatus and environment. The results of the solution are analyzed and discussed to provide some insight on how the solution could be improved for the future.

Table of Contents

List of Figures

Problem Description

The main goal of this project was to implement a sorting algorithm on a provided apparatus by using an microcontroller to sort multiples of different part types in the shortest amount of time and the most accuracy. The 48 parts are small cylinders equally batched into aluminum, steel, black and white plastic. An apparatus is provided with a conveyor belt powered by a DC motor, various sensors for identifying pieces and a stepper motor which rotates a plate with four different bins, one for each piece. The Atmel ATmega2560 microcontroller is used to control the separate sorting systems. To understand and generate a working solution, we tested our implementation of the C code needed by the board to interface with the stepper motor, brushed DC motor, and various sensors.

System Layout

The layout of the apparatus and the microcontroller are shown in figures 1 and 2 below. These figures include the following components:

- ATMEGA2560 microcontroller
  Connected via USB to PC for uploadable programming, the Atmel interfaces with each component with input/output pins
- Conveyer belt
- Brushed DC motor
  Controls the speed of the conveyor belt. The Polulu driver sends signa;s as a PWM wave pattern effectively setting the rpm
- Two Polulu VNH2SP30 high current DC motor driver
  Interfaces the DC motors to the microcontroller. The drivers accept logic levels from the board and drives induction.
- Stepper motor
  Soyo 6V 0.8A 36oz-in Unipolar Stepper Motor orientates the sorting tray based on a given part
- Sorting tray
  Partitioned by dividers, 4 tray types allow enough space for the 12 respective parts
- Analog-to-digital converter (ADC)
- Optical sensor (OR)
  Detects the presence of an object (the part) as it travels along the conveyor belt.
- Optical sensor (EX)
- Hall effect-sensor (HE)
- Reflectivity Sensor (RL)

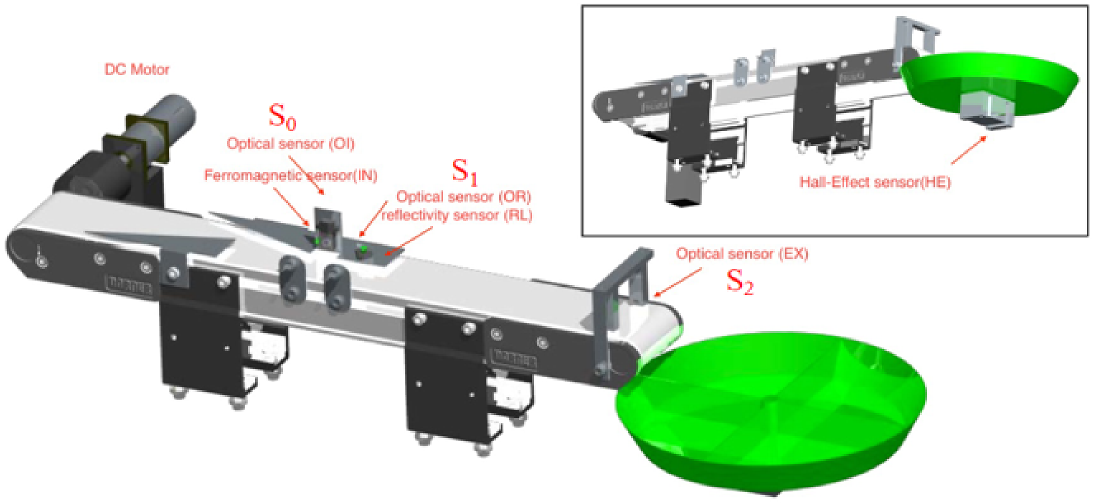


Figure 1 – System overview

Three sensors are needed to successfully implement a sorting algorithm. These sensors, as seen on figure 1, are S1, S2 and S3. The optical sensor (OR) is used to identify the piece, this is done via a laser and reflective sensor which can classify a part depending on the amount of reflection it sees, it can then send the identity of the part into a linked list to be accessed later. The optical sensor (EX) tells the system when a piece is at the end of the belt and by accessing the linked list it can hold the piece in position until the corresponding bin is ready to accept the piece. The hall-effect sensor (HE) is used to tell the system the current position of the bin and move it to an allocated starting position. Once in its starting position the system will know how to get to every other position from there.

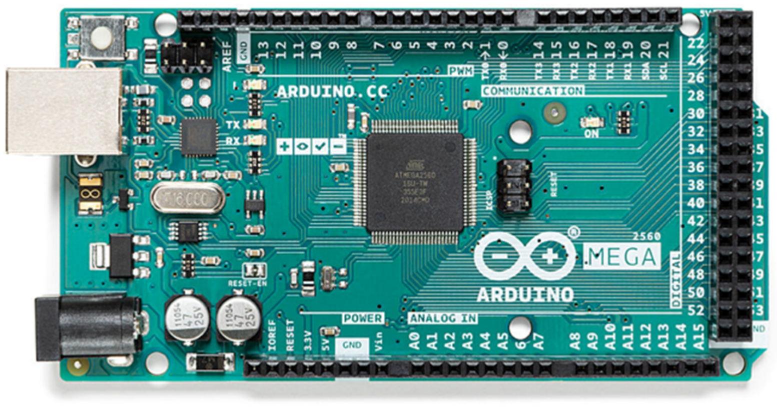


Figure 2 – ATmega2560

Technical Description

The ATmega2560 seen in figure 2 is the main piece of hardware of which we could alter details. The details in question are pin positions, using our circuit board and classifications in the code it is specified which port operates what. In our board portC (30-37) operated red LED's, portB (50-53 and 10-13 in pwm) operated the DC motor, portA (22-29) operated the stepper motor and portL (42-49) operated the LCD display.

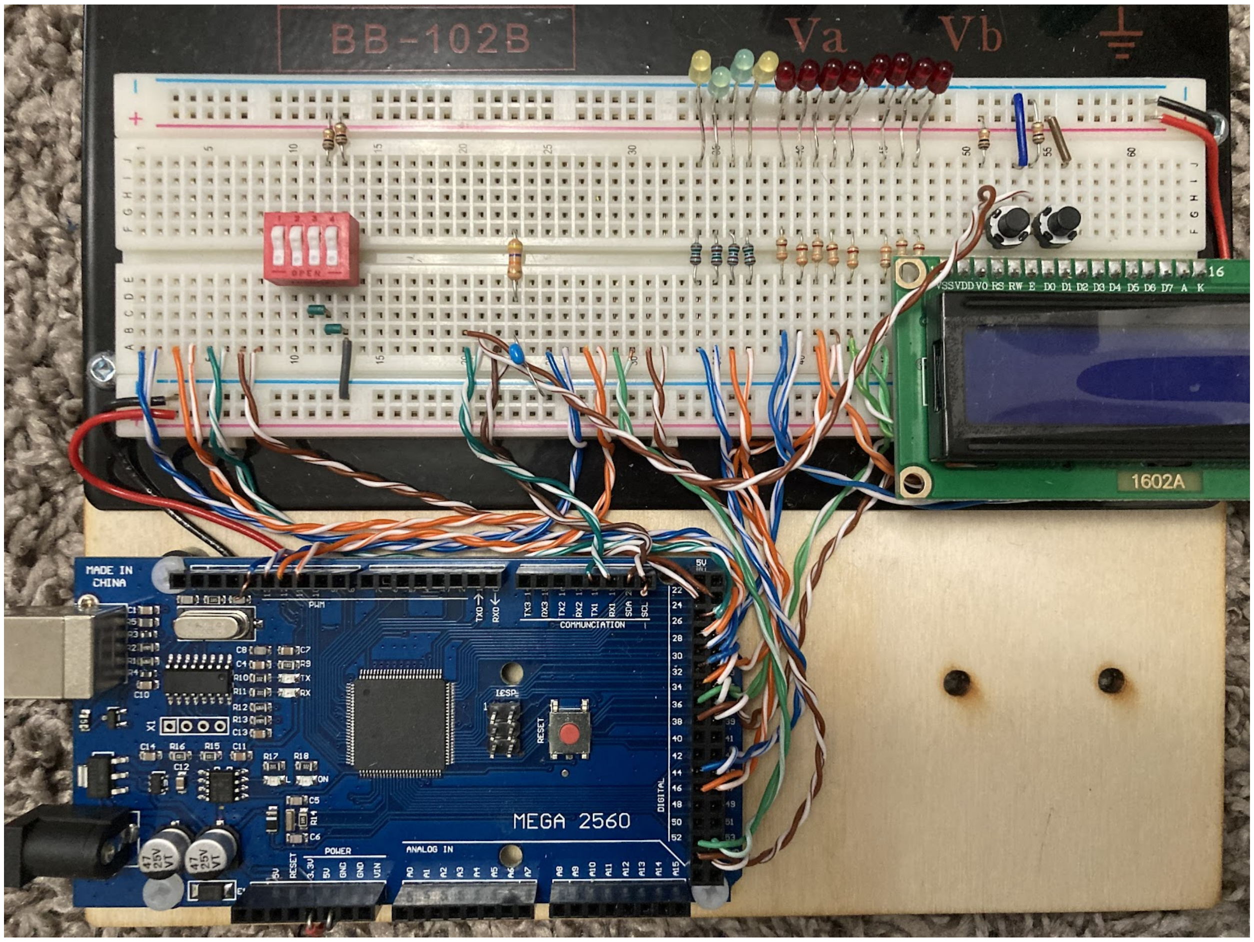


Figure 3 - Final circuit

Methods and Design

The final code has a linked list at the core and multiple functions and cases to bounce off it. At the start the list is empty and remains that way until the optical sensor (OR) is triggered, once the OR has identified the piece it's item code is stored in the head of the linked list, any pieces that go through the OR after will be attached to the head. The conveyor continues running and adding pieces to the linked list until the other optical sensor (EX) is triggered. The EX sensor indicates a piece at the end of the belt and when activated will pause the belt and search the linked list for the item code of the head, this is then used to rotate the stepper moving the bin into the correct position. Once the system is told the bin is correct it will resume the belt and remove the item that was just sorted from the head of the list. This creates a new head of the list which will be activated the next time EX triggers.

The circuit for this project execution uses an ATmega 2560 and generic circuit board. Wiring between the controller and board is executed over the course of the labs in the term and features LED’s and an LCD which can be used for debugging and indicating what code the system is currently executing.

Sorting Algorithm

As the system starts, the sorting tray is located using the hall-effect (HE) sensor. This position is the reference we use for the position of the stepper. The home is found by rotating the stepper until a low signal is picked up by the sensor. For our purposes, home is the black bin. Then the DC motor will be instructed to commence. As parts are loaded and pass the OR, an external interrupt is raised and corresponding ISR is triggered. The RE sensor likewise triggers its own ISR and returns an ADC measurement to determine the unique part type. The program adds the part to a linked list. As the belt continues, the part is brought to the EX sensor. The ISR this sensor triggers brakes our DC motor. The head of our link list has the corresponding parts type which will be sent into the rotation command of our stepper. The stepper rotates on an accelerated curve to the appropriate position and the belt is allowed to continue. This process is looped until a ramp down is initiated by the external interrupt sent by the toggle switch on the arduino.

Results

Over a series of time trials, our solution could reliably sort parts in just over 40 seconds with errors only stemming from loading (human error). Until capacitors were added to the external switches and sensors, our system was experiencing noise from the DC motors which interfered with our stepping algorithm.

Our final performance was able to achieve a similar result from our own testing. By increasing the load batch, we could shave down the time it took to complete in 39 seconds with only a single error. Our pause and ramp down switches also worked successfully. Although, the placement and spacing of the parts had to be small enough for ramp down to work in all cases.

Conclusion

Our solution met all the specifications of the project with much time to spare. Although it was not the fastest solution, we could experience no errors with our algorithm. If this process were to be improved, the acceleration of the stepper motor would have to be converted to allow for an

‘S-curve’ to save on sorting time. We can also alter the speed of the conveyor belt to allow items to move through sensors faster. Our ramp down check would also need to be improved to add a longer window for incoming parts. Overall, we are happy with the way our solution executed.

Appendix A - Code

```
/* ##########################################################################
# PROJECT: Final Project
# NAME 1: Tom Boddington, V00932207
# NAME 2: Colton Hyland, V00806034
# DESC: The final project for mech 458, must be able to identify 4 different parts using an OR
sensor and ADC conversion then sort them using a linked list into designated bins
# 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
track pause condition

/* DC motor belt variables */
volatile unsigned char dcBeltBrake    =  0b11111111; // stops belt by stopping DC motor
volatile unsigned char dcBeltFwd        =  0b10001110; // rotates belt forward with DC motor

/* turntable stepper variables */
int stepRotation[4] =  {0b00110110, 0b00101110, 0b00101101, 0b00110101};                  //
create array with 4 different PWM steps, corresponding to each bin
volatile unsigned char dutyCycle = 0x80;
                    // controls belt speed, set at 50%
volatile int stepCounter = 0;
                              // step counter varies from 0->3, keeps track of stepper location


// 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


/*Sensor Variables*/
// 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;                  // used to initialize stepper position

// 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;
volatile unsigned int totalCount    = 0;

volatile unsigned int inQueue           = 0; //keeps track of items between reflective sensor and
sorted stage

int flag = 0; //used to indicate when ramp down is triggered

// variables to track the maximum ADC value for each part
int ALUM_MAX = 300;                              // for station 8
int STL_MAX = 750;                        // for station 8
int WHITEPLASTIC_MAX =  940; // for station 8
int BLACKPLASTIC_MAX = 1010; // for station 8

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

/* Function declarations */
void mTimer();
void mTimer2();
void generalConfig();
void pwmConfig();
int stepperControl(int direction, int steps);
```

```
void stepperHome();
void rotateDC(char beltState);
int identifyPart(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;                      // starts at polling stage


   InitLCD(LS_BLINK|LS_ULINE);                          //Initialize LCD module
   LCDClear();                                                  //Clear any text that
many be on LCD

   /*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();                          // Enable all interrupts
        stepperHome();                  // get stepper homed at position BLACKPLASTIC

        /*while(1){ //stepper motor test
                stepperControl(CCW, 50);
                mTimer(1000);
                stepperControl(CW, 50);
                mTimer(1000);
                stepperControl(CCW, 100);
                mTimer(1000);
                stepperControl(CW, 200);
                mTimer(1000);
        }*/
```

```
	/*while(1){ //DC motor test
		rotateDC(dcBeltFwd);
		mTimer(1000);
		rotateDC(dcBeltBrake);
		mTimer(1000);
	}*/

		goto POLLING_STAGE;

		// POLLING STAGE
		POLLING_STAGE:
		rotateDC(dcBeltFwd);		//start DC motor


		switch(STATE){
			case (0) :
			goto POLLING_STAGE;
			break;
			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 2, stop when button pressed and display counts, resume when
pressed again

			rotateDC(dcBeltBrake); //stop DC motor

			// print off the current count
			LCDWriteStringXY(0,0,"S:");
			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);



        while(pauseButton == 0){ //hold in pause state until button pressed again
            LCDWriteStringXY(0,1,"Paused");
            LCDWriteStringXY(7,2,"US:");
            LCDWriteIntXY(11,1,inQueue,2);
            }
            LCDClear();

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

        REFLECTIVE_STAGE: // State 2

            initLink(&newLink); //initialize new link connection
            newLink->e.itemCode = (identifyPart(ADC_RESULT)); // identify item and
put value into itemCode
            enqueue(&head, &tail, &newLink); //take data and create new link
            ADC_RESULT=2000;        // arbitrary value to reset ADC

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


        SORTING_STAGE:  //State 3, feed part into correct bin

            rotateDC(dcBeltBrake);
            mTimer(100);                    // stop the belt

            //increments counter of item when EX sensor triggered
            if(head->e.itemCode == ALU){
                aluminumCount+=1;
            }else if(head->e.itemCode == STL){
                steelCount+=1;
            }else if(head->e.itemCode == WHT){
                whiteCount+=1;
            }else{
                blackCount+=1;
            }

            rotateStepper(tablePosition, head->e.itemCode); //move stepper into
correct position
            mTimer(20);
            dequeue(&tail, &head, &rtnLink);        //remove sorted item from linked
list
            free(rtnLink);                  //free up memory after removing list item

            if(flag == 1){ //check if ramp down has been triggered
                if(inQueue == 0){//if triggered and list is empty then send to END
state
                        STATE = 5;
                        goto END;
                }
            }

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

        RAMP_DOWN: // State 4, when button is pressed sort remaining parts, display
count and stop belt


        PORTC = 0b01010101;  //visual indicator that ramp down has been triggered
        LCDClear();

        LCDWriteString("Ramp down");
        flag = 1; //indicate to system that ramp down has been triggered


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


        END: // State 5, stop everything and display results

        rotateDC(dcBeltFwd); //run belt in case piece is stuck in EX sensor
        mTimer(1000);
        rotateDC(dcBeltBrake);              // stop the belt
        PORTC = 0xFF;                                   // visual
indicator that program ended
```

```
                    //display final count
                    LCDClear();
                    LCDWriteStringXY(0,0,"S:");
                    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);

                    STATE = 5;
                    return(0);
}

//Interrupt for exit sensor, EX
ISR(INT0_vect){
          mTimer(5);
     if((PIND &0b00000001) == 0b00000000){                  //detect sensor trigger

                              inQueue -=1; //decrement inQueue since piece is leaving belt
                              STATE = 3;     // go to sorting State



           }
}
//Interrupt for optical sensor, OR
ISR(INT1_vect){
   mTimer(5);
  if ((PIND & 0x02) == 0x02){ //if sensor is detecting a part
                    ADCSRA |= _BV(ADSC);    //start single ADC conversion
                    inQueue +=1;                    //increment inQueue since piece is entering OR
   }
}

//Interrupt for pause condition
ISR(INT2_vect){

      if((PIND &0b0000100) == 0b00000100){                  // initial compare statement to detect
button press
          mTimer(20);
```

```
                              pauseButton = (pauseButton+1)%2;           // flip button state, allows
program to hold until button pressed again
                              STATE = 1;                                                  // go to
PAUSE state

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

}

//Interrupt for ramp down condition
ISR(INT3_vect){
          /* Toggle PORTC bit 3 */

   //RAMP DOWN
     if((PIND &0b00001000) == 0b00000000){                                  // initial compare
statement to detect button press
        mTimer(20);
        STATE = 4;                             // go to RAMP_DOWN state
                     PORTL = 0xFF;
     }
     while((PIND &0b00001000) == 0b00001000);                        // check to see if button is
released
     mTimer(20);
}

//ISR4 removed as wasn't in use

// interrupt for Hall Effect sensor, HE
ISR(INT5_vect){
          STAGE_4_HE_FLAG = 1;                // trigger to indicate turntable HE sensor is in
position
}

// Interrupt for timer3
ISR(TIMER3_COMPA_vect){
   STATE = 3; //handle parts based on reflectiveness
}

// interrupt for ADC value
ISR(ADC_vect){
          if ((ADC) < ADC_RESULT){
                ADC_RESULT = ADC;                                        // pass ADC values to
ADC_RESULT
        }
        if((PIND & 0x02) == 0x02){                           // object is in front of ADC sensor
                ADCSRA |= _BV(ADSC);                                // start new ADC
measurement

                //Print out ADC value for system calibration
                //LCDWriteIntXY(0,1,ADC_RESULT,4);
                //LCDWriteStringXY(7,1,"<-ADC VAL");

        } else if((PIND & 0x02) == 0x00){               // object is not in front of sensor
                STATE = 2;                                               // back to reflective
stage
        }
}

// LEDs start flashing if bug encountered
ISR(BADISR_vect){

        while(1){
                PORTC = 0b10101010;
                mTimer(100);
                PORTC = 0b01010101;
                mTimer(100);
        }

}//end ISR BADISR_vect

void generalConfig(){
        // IO configuration
        DDRA = 0xFF;                //stepper motor, set to output
        DDRB = 0xFF;                //DC motor drive, set to output
        DDRC = 0xFF;                        //LCD/LEDs used for debugging, set to output
        DDRD = 0xF0;                //buttons for interrupts, 0-3 output, 4-7 input
        DDRE = 0x00;                //other interrupts, set to input
        DDRF = 0x00;                //ADC interrupts, set to input
   DDRL = 0xFF;         //other LEDs, set to output

        // interrupt configuration
   EIMSK |= _BV(INT0)|_BV(INT1)|_BV(INT2)|_BV(INT3)/*|_BV(INT4)*/|_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(ISC41) | _BV(ISC40);
					// INT4 rising edge, not in use
	*/
	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
   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(CS00)	// not sure whether we need this or not
   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;
}

//Set DC motor to either forward or brake
void rotateDC(char beltState){
	PORTB = beltState;
}

// keeps rotating stepper until HE is satisfied, initial position set to black
void stepperHome(){
        while (STAGE_4_HE_FLAG == 0 ){
                stepperControl(CW,1);
        }
        tablePosition = BLK;      //tell system initial position
    return;
}

 //function to control stepper rotation
 int stepperControl(int direction, int steps){

     if (steps == 100){   //180 degree
                    int activeArr[100] =
{15,14,13,12,11,10,9,8,7,7,6,6,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5
,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6,6,7,7,7,8,9,10,11,
12,13,14,15};

                    if (direction == 0){                                        //run clockwise
routine
                            for (int i=0; i< steps; i++){
                                    stepCounter++;                                      //increment
step counter
                                    if(stepCounter > 3){                        //reset array index
when it reaches the end
                                            stepCounter = 0;
                                    }//end step counter if
                                    PORTA = stepRotation[stepCounter]; //cycle through the step
rotation array
                                    mTimer(activeArr[i]);
                            }//close for loop
                    } else if (direction == 1){                     //run counterclockwise routine
                            for (int j=0; j < steps; j++){
                                    stepCounter--;
//decrement step counter
                                    if(stepCounter < 0 ){                       //reset array index
when it reaches the end
                                            stepCounter = 3;
                                    } //end step counter if
                                    PORTA = stepRotation[stepCounter];          //cycle through the
step rotation array
                                    mTimer(activeArr[j]);
                            }//close for loop

                    }//end if direction = 0
```

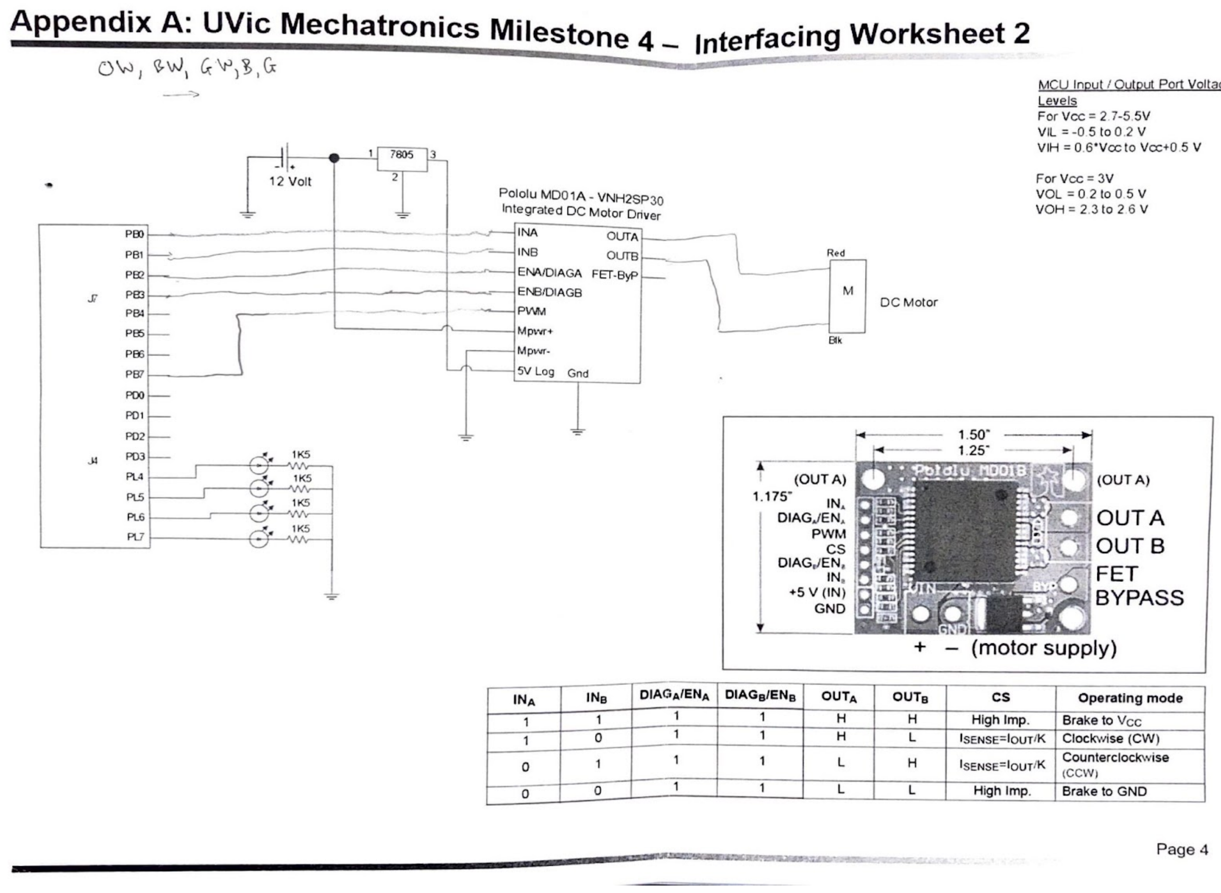
```
            } else if (steps == 50){ //90 degree
                    int activeArr[50] =
{15,14,13,12,11,10,9,8,7,7,6,6,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6,6,7,7,8,9,10,1
1,12,13,14,15};

                              if (direction == 0){                                                    //run
clockwise routine
                                        for (int i=0; i< steps; i++){
                                                  stepCounter++;
//increment step counter
                                                  if(stepCounter > 3){                              //reset array
index when it reaches the end
                                                            stepCounter = 0;
                                                  }//end step counter if
                                                  PORTA = stepRotation[stepCounter]; //cycle through the
step rotation array
                                                  mTimer(activeArr[i]);
                                        }//close for loop
                               } else if (direction == 1){                         //run counterclockwise
routine
                                        for (int j=0; j < steps; j++){
                                                  stepCounter--;
//decrement step counter
                                                  if(stepCounter < 0 ){                             //reset array
index when it reaches the end
                                                            stepCounter = 3;
                                                  } //end step counter if
                                                  PORTA = stepRotation[stepCounter];          //cycle through
the step rotation array
                                                  mTimer(activeArr[j]);
                                        }//close for loop

                              }//end if direction = 0

                    //reaches here then homing condition
            }else {
                               if (direction == 1){                                                    //run
clockwise routine
                                         for (int i=0; i< steps; i++){

                                                   stepCounter++;
//increment step counter
                                                   if(stepCounter > 3){                                      //reset
array index when it reaches the end
```

```
                                                stepCounter = 0;
                                        }//end if
                                        PORTA = stepRotation[stepCounter];        //cycle through
the step rotation array
                                        mTimer(20);
                                }//close for loop

                        }
                        }
    return (0);

 } //end of stepperControl function

//function to determine how to rotate stepper dependent on current and next position
int rotateStepper(int currentPosition, int desiredPosition){
        int delta = (desiredPosition - currentPosition);
        if ((delta == 3) || (delta == -1)){                                     // if the difference is
90 degrees CCW away
                stepperControl(CCW,50);                                         //
rotate 50 ticks to desired position
      rotation = CCW;
        } else if ((delta == -3 ) || (delta == 1)){                     // if the difference is 90
degrees CW away
                stepperControl(CW,50);
// rotate 50 ticks to desired position
      rotation = CW;
        } else if ((delta == -2) || (delta == 2)){              // if the difference is 180 degrees
away
                stepperControl(rotation,100);                                   // rotate 100
ticks to desired position, direction do not matter here
        } else {                                                                //
already in correct position
                stepperControl(CW,0);
// do nothing
        }// end if else
        tablePosition = desiredPosition;                                // update current
position
}// end rotateStepper

// function to determine item type based off ADC values
int identifyPart(int ADC_RESULT){
        int classPart = 0;
        if ((ADC_RESULT <= ALUM_MAX)){                                  // part is
aluminum
```

```
                classPart = ALU;
                totalCount++;
                } else if(ADC_RESULT <= STL_MAX){                              // part is steel
                classPart = STL;
                totalCount++;
                } else if (ADC_RESULT <= WHITEPLASTIC_MAX){          // part is white
                classPart = WHT;
                totalCount++;
                } else if (ADC_RESULT <= BLACKPLASTIC_MAX){                   // part is black
                classPart = BLK;
                totalCount++;
        }
        return classPart;                                                                    //
return part type to allow storage in linked list
}// end identifyPart

// usual timer function to delay by "count" milliseconds
void mTimer(int count){

        int i=0;                                              //counting variable
        TCCR1B |= _BV(CS11);                                  //setup timer
        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){ //time comparison if
                        TIFR1 |= _BV(OCF1A);        //set timer/counter interrupt flag so
the interrupt can execute
                        i++;                                  //increment
                }//end if
        }//end while loop comparing our count up case
        return; //exit timer function
}// end clock function 1

// secondary timer that relies on a separate interrupt status, and does not need to be function
called
void mTimer2(){
        TCCR3B |= _BV(WGM32);                                 //clear OCR3 on compare match, set
output low
        TCCR3B |= _BV(CS30) | _BV(CS32);          //clock prescalar by clk/1024 from prescalar
        OCR3A = 0x0BB8;                                   // write output compare register to hex
value of 3000
        TCNT3 = 0x0000;                                   //set the initial timer/counter value to 0
        TIMSK3 =TIMSK3|0x002;                             //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

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

Appendix B - Wiring Diagrams



Appendix A: UVic Mechatronics Milestone 4 – Interfacing Worksheet 2

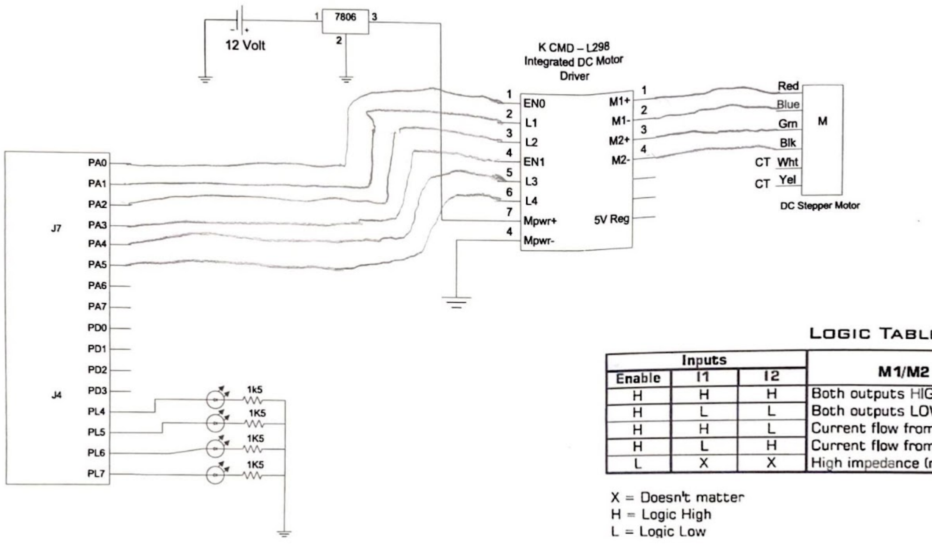
<u>MCU Input / Output Port Voltage Levels</u>
For Vcc = 2.7-5.5V
VIL = -0.5 to 0.2 V
VIH = 0.6*Vcc to Vcc+0.5 V

For Vcc = 3V
VOL = 0.2 to 0.5 V
VOH = 2.3 to 2.6 V

IN_A	IN_B	$DIAG_A/EN_A$	$DIAG_B/EN_B$	OUT_A	OUT_B	CS	Operating mode
1	1	1	1	H	H	High Imp.	Brake to V_{CC}
1	0	1	1	H	L	$I_{SENSE}=I_{OUT}/K$	Clockwise (CW)
0	1	1	1	L	H	$I_{SENSE}=I_{OUT}/K$	Counterclockwise (CCW)
0	0	1	1	L	L	High Imp.	Brake to GND

FIgure 4 - Wiring for DC motor

Appendix A: UVic Mechatronics Milestone 4 – Interfacing Worksheet 1



MCU Input / Output Port Voltage Levels
For Vcc = 2.7-5.5V
VIL = -0.5 to 0.2 V
VIH = 0.6*Vcc to Vcc+0.5 V

For Vcc = 3V
VOL = 0.2 to 0.5 V
VOH = 2.3 to 2.6 V

Logic Table

Inputs			M1/M2 or M3/M4 Outputs
Enable	I1	I2	
H	H	H	Both outputs HIGH (motor brake)
H	L	L	Both outputs LOW (motor brake)
H	H	L	Current flow from M1 to M2 (rotates dir 1)
H	L	H	Current flow from M2 to M1 (rotates dir 2)
L	X	X	High impedance (motor coasts)

X = Doesn't matter
H = Logic High
L = Logic Low

See manufacturer documentation for the pin-out of the K-CMD kit (course website)

Tom Boddington V00932207
Colton Hyland V00806034

Page 3

Figure 5 - Wiring for stepper motor