MAE 3780 Mechatronics
Final Report

Dongze Yue (dy85)
Jiahao Zhang(jz522)
Jiawen Fang(jf442)

September 13, 2016

1 Introduction

This report is a full documentation of the sumobot design, manufacturing and performance for the final project MAE 3780 Mechatronics by group Bin Number 21. The aim of the project is to design and build an 8'by 8' robot that will actively engage the opponents robot and push it out of the 3ft diameter arena in 3 minutes or less. Our robot ranked No.5 pre-competition and thus entered the second round directly. In the following rounds we had 2 wins and 2 losses and was eliminated before the 7th round.

2 Design Breakdown

2.1 Overall Mechanical Design

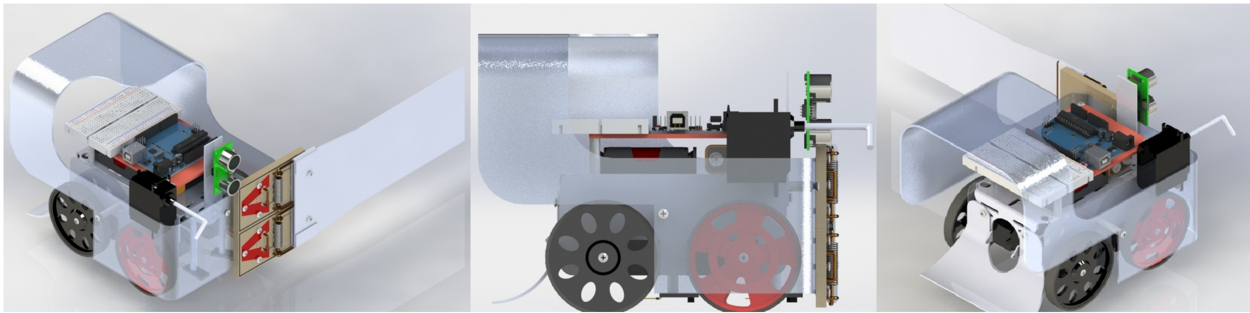

Figure 1: Robot Final Rendering

Mechanics The robot has 4-wheel drive. Its active element is a slapper held by two mousetraps mounted in the front. The trigger for the slapper is a bent shaft on the servo motor attached to the robot skin. The slapper is manually reset by bending it 180 degree and then blocked with the bent shaft. It's released when the shaft moves away. There are 3 QTIs at the bottom of the robot: 2 flanks the front and 1 at the center. Two plat steel plate are customized to extend the body downward to accommodate all four servo motors. Two H-bridges, soldered to 2 separate perfboards are hot-glued to the bottom of the chassis. The 9V and 6V batteries are placed right on top the chassis. The Arduino and the breadboard are placed on the top layer for easy wiring. On the back of the robot we placed a curved steel plate to prevent the robot from tipping over when tackled by flippers.

2.2 Algorithmic Description

There are 5 states in our Algorithm: Initial, search, engage, trigger and stop.

Initial state, as the name suggests, activates all the elements in the robot and offers a 2 seconds break for the robot to get into his role.

Search state is automatically entered after initial state. In this state,our robot is programed to rotate in clock-wise motion until target is found within pre-set distance.

Engage state is entered when the distance detected by the sensor is below pre-set value. In this state, the robot will stop its rotation and move straight forward to approach the target. There is also a "correction" sub-state under engage state. When target is lost in engage state, first the robot will turn to left and right up to 40 degrees twice to relocate the target and improve the

route.However if target is not found after the two turns, the robot will re-enter search state.

Listing 1: Searching Method

```
/*
*The searching method has a global variable, dir that decides the direction
*of rotation of the search.
*The robot keeps rotating and getting value from *the sonar using getSonar() method
*until the distance is close enough.
*Once the searching is done the robot goes out of the loop,
*otherwise it stays in the loop forever until target found.
*/
int Search(void){
    if (!dir){
    MotorAction(RIGHT);
    } else {
    MotorAction(LEFT);
    }
    initSonar();
    while(getSonar() > DIST_FIND) {
    RobotTurn(1,5); //The robot turns for 5 degrees.
    }
}
```

Trigger state is active when the detected distance is under a smaller preset value and the servo that previously pinned the extended arm will let go and the mousetrap will fire out.

Stop state is when the center QTI is triggered. In this state all motion will stop and the red LED light up.

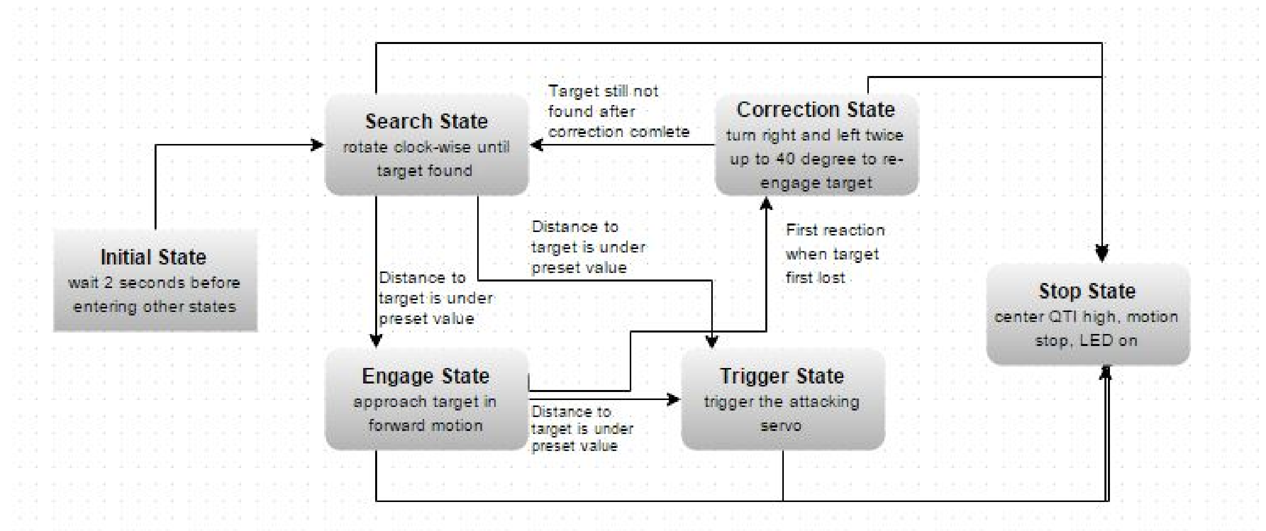

Figure 2: Algorithm Breakdown

2.3 Schematic Description

H-Bridge is designed with "step down resistors" to protect the motor circuit: when Arduino is reset every time, the pins go back to low and turn on the BJT. This makes sure the motor H-bridge won't be shorted.

Our MOSFETs were selected incorrectly. Their threshold voltage range 2-4V and reach peak performance at 10V, which is not achievable through Arduino output, posing challenge to the H-bridge design. We attempted to connect the Arduino and our batteries in series to supply a 11V power on our motors. Although such setup offers high speed and torque on the wheels, it causes the malfunction of our sonar because of the huge current the motor draws from not only the batteries but the Arduino board as well.

In order to achieve ideal turn on voltage of the MOSFET of 10V, we drew an additional wire from the positive terminal of the 9V battery which supplies power to the Arduino board to the H-bridge. We used the Arduino pins to toggle the 9 volts supplied by the battery and then utilized the 9 volts to turn on the MOSFETs. However, the lab only provided us with free NPN BJTs, so it's only possible to build a pull down network using additional resistor. We attempted to to build a pull up network with the same configuration by moving the resistor to the emitter side of the BJT, but the circuit didn't work as expected - the voltage on Vout was extremely low and is insufficient to drive the MOSFETs. Later we adapted the pull down network and identified another problem - wiring the network directly onto Arduino pins causes the H-bridge to short circuit every time Arduino resets for whenever Arduino resets, all the pins are on low, which turns on the BJTs and therefore turns on all the MOSFETS, generating a large short circuit current.

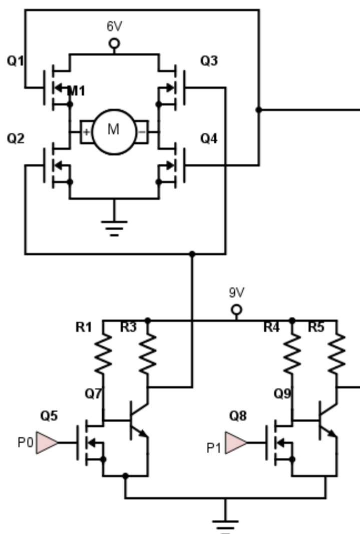

Figure 3: H-Bridge

In order to solve this problem, we added an additional pull down network on top of the original network. By doing so we obtain a pull up network by doubling the negation. We chose MOSFETs over BJTs for the second pull down network because we believed that this will reduce current draw on Arduino pins, increasing the board efficiency.

Motion	mov	turn	pos	P_0	P_1	P_2	P_3
Stop	0	/	/	0	0	0	0
Forward	1	0	1	0	1	0	1
Backward	1	0	0	1	0	1	0
Left	1	1	0	1	0	0	1
Right	1	1	1	0	1	1	0

$$P_0 = (mov) \cdot (\overline{pos})$$

$$P_1 = (mov) \cdot (pos)$$

$$P_2 = (mov) \cdot (\overline{pos \oplus turn})$$

$$P_3 = (mov) \cdot (pos \oplus turn)$$

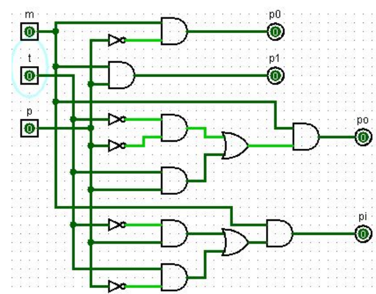

Figure 4: Motor Drive Logic

Motor Drive Logic The truth table and logic equations above explain the motor drive logic. In our plan, the logic was to be implemented by circuit above, which was later replaced by microprocessor program due to time constraint. The function input three parameters: "mov","turn","pos" which indicate if the robot move/not move, turn left/right, go forward/backward(or turn clockwise/anti-

clockwise). It then output P0 and P1 values which will specify the actual direction of motor rotation through H-bridge.

3 Design Motivation

Mechanical Design Our mechanical design went through a major change of attacking mechanism before the actual construction from mousetrap lifter to mousetrap slapper. The original plan was to upgrade the winning design from year 2014 by re-using the servo that triggers the attack to lift up the extended arms on the mousetraps and thus tilt up the opponent trapped in between the arms. This way we lessen the traction and limit the mobility of the opponent and increase traction to our robot by adding the extra weight from the opponent robot. However after observing the flat shovel design adopted by several groups we realized that the biggest challenge is to avoid the shovel instead of to initiate a shovel match.so instead we placed the mousetrap and the extended arm in the front of our robot and the it will be trigger when the distance between two robots is within preset value. This way, we utilize the moment provided by long arm and big momentum to avoid running into the opponent face to face so that we increase our chance of blindsiding the opponent as our robot is programmed to have a quicker reaction after the slap.

Electrical Design One of our major electrical designs was to provide maximum power to the motor. For maximum efficiency, we compared the performances of a MOSFET H-bridge and a BJT H-bridge, both powered by 6V batteries, with the components available and we soon discovered that using the BJT H-bridge model that we adapted in the lab sessions will eventually limit the amount of current flowing down the H-bridge. The motor were weak and rotating slowly. In order to get larger current from the battery, we decided to use MOSFET to control the H-bridge. Besides reducing current loss, we also experimented to increase the voltage supply to the motor to achieve higher torque. However, regarding the risk of damaging the motor, we didn't go far on this approach.

Another important feature that we implemented was to minimize the current loss of the Arduino board, because we thought that drawing too much current from the board will cause the system unstable and glitch the sonar. Therefore, we added two extra MOSFET, Q5 and Q8 on the diagram, to minimize current, because MOSFET will not draw significant current from the gate.

C Program Design Besides the main() function that initializes the program and the while(1) loop and regulates the robot, we wrote 5 additional files containing useful methods and 1 header class containing all the numerical constants and Finite State Machine state information. Within the while(1) loop of the main function, we only call the fsm() function every cycle and we regulate the behavior of the robot in each distinct state of the FSM. This design is to ensure that the robot correctly functions as what we designed to and brings a hint of simplicity. We used the header class to contain all the global constants to easily debug and calibrate our system.

4 Strengths and weaknesses

4.1 Strengths

Power and Servo Control: Our H-bridge design ensures that maximum possible power is delivered to the servo motors to drive the robot robustly and actively engage the opponents. Fur-

thermore, the simple and effective logic expressions of P0, P1, P2 and P3 on H-bridges enable the system to respond even faster.
Active Slapper: There are two main purposes of the slaper attack: first to dodge the opponent's active element and second to increase our chance to blindside the other robot. Our design was greatly different than the majority of other robots and we hoped to take them by surprise.

4.2 Weaknesses

Insufficient Sonars: two sonars were supplied but we only implemented one due to the time constraint. This decreases the accuracy of our searching mechanism and it's likely we'll lose track of short targets and targets with decoys.
Poor Wiring: Our wires were long and disorganized, all connected to one proto-board and then directly to the Arduino pins.There's no insurance to keep the wires connected in place either besides a few drops of hot glue. It would takes us longer to realize or locate circuit failures if the wires got loose or fell out.

5 Performance in Competition

In the **2nd round** we competed with bin 24. Our robot was faster to locate the opponent and drove it out swiftly.

In the **3rd round** we competed with bin 4. Two robots engaged with each other at roughly the same time and the race evolved into a pushing battle. Our robot was eventually pushed out of the arena because the rubber band on our left back wheel fell off and got stuck onto the motor shaft.

In the **5th round** we competed with bin 34. Our robot was significantly faster at locating the opponent and it drove out the opponent right after they engaged with each other.

In the **6th round** we competed with bin 33. The opponent pushed us out of the arena from our back. We missed the target in our first attempt to search because the sensor wasn't turned on. Our guess is our wires either got loose or dropped out.

6 Statement of work

Jiahao Zhang

Designed, CAD and constructed the robot. Implemented the initial FSM and overall algorithm (capable of performing milestone#3 and milestone#4), which was later further optimized for competition.

Dongze Yue

In charge of the overall electrical design. Designed, soldered and configured power management and H-bridge of the robot.Collected the code written by the teammates and integrated them into one file. Straightened out the methods and added the header file to make the program more clear and comprehensive. Re-factored our original design of the Finite State Machine and wrote a sequential switch-case branch logic to actively change the state of the FSM against the change of input.

Jiawen Fang

Assisted in mechanical design and construction, electrical and coding trouble shooting. Managed the schedule, budget and purchase plan. Kept the morale up and ordered midnight pizza.

7 Future Work

All the designs below were actually realised by our code, but they weren't installed onto the robot due to time constraints.

Speed Measurement In the process of designing the Finite State Machine, we developed a possible scenario that our robot gets trapped on the enemy's large shovel (or plate) that although our motor tries to move, we are actually pushed by the enemy. Therefore, we decide to implement a speed measuring system that monitors the speed of rotation of a wheel real time and warns the controller if the robot stops moving while the motor is still on.
In order to measure the speed, we originally thought that we can add a accelerometer on board and use the acceleration information to calculate whether the robot is moving or not. However, a typical accelerometer will cost a large portion of our budget and it has a lot of extra features that we won't use, therefore we decided to look for other solutions.
The second approach we took was to look for a tachometer that measures the speed into voltage. We soon found this very hard to work with, because the voltage provided by tachometer is noisy and tiny, which requires a complicated filter to process. More importantly, voltage level is an analog feature instead of digital. It will be complex to code the Arduino to detect the voltage.
Finally we had an inspiration on an old LEGO robot servo model that gives an digital feedback on the current motor speed. We nobly sacrificed the servo by forcibly opening the seal and breaking the gears. Instead of using a traditional tachometer, the LEGO model uses an "optical" tachometer, which connects a partly-hollow gear to the servo and used an optical sensor to detect if there were any blockade (the un-hollow part of the gear) in front of it. As the gear starts rotating at a given speed, the optical sensor detects the presense of the blockade in a unique frequency. The frequency will be inversely-proportional to the actual speed of the rotation. Therefore, regarding the LEGO design, we used a QTI sensor right next to the wheel, and sticked black-and-white stripes around the wheel. Once the wheel starts rotating, the QTI sensor will frequently detect the white and black parts changing. Therefore we can include speed as a factor in the calculation.

Memory and Coordinate System We also planned to store each movement that the robot made into a size 100 array in the Arduino. Then we will know how far we travelled and where we are in the circle, if we correctly calibrated the system. For each move that the program calls the MotorAction() method, the array updates to record the movement. Then the robot can perform a GoBack() method to rewind all the movements and go out of trouble if it encounters enemy. Also, we can create a coordinate system within the robot if we also input the sonar information into it: we will know where the enemy are, the speed of the enemy, and the route the enemy is walking on.

8 Conclusion

In this report we presented the thoughts and highlights in the design and construction process of our sumobot. Reflect upon our work and the competition result, it's obvious that with limited time to finish the project, it is crucial to balance the efforts put into mechanical, electrical and coding elements of the robot. Through this project, we definitely gained better understanding of the course and the field of Mechatronics. Futher details on the cost and the complete code could be found in the appendix.

9 Appendix

Cost Table

Name	Quantity	Source	Price
Wheel	2	DigiKey	4.26*2=8.52
MOSFET I	4	DigiKey	1.48*4=4.92
MOSFET II	8	DigiKey	0.93*8=7.44
Motor	2	Lab	3*2=6
Mousetrap	2	McMaster	0.85*2=1.7
Scrap steel	1 ib	Mechine Shop	16*0.2=3.2

Total Cost: 31.78 dollars

Commented Code Here is the fully commented code.

Before introducing the implemented methods, it is important to start with the header file first, as it includes crucial state and constant information for the whole program.

Listing 2: constants.h

```
#ifndef _CONSTANTS_H_
#define _CONSTANTS_H_
    // motor states
    #define FORWARD 1
    #define BACKWARD 2
    #define LEFT 3
    #define RIGHT 4
    #define OFF 0
    // fsm states
    #define INIT 0
    #define SEARCH 1
    #define ENGAGE 2
    #define TRIGGER 3
    // useful constants
    #define SONAR_FACTOR 0.0256 //numerical multiplier that gives
    // the sonar measured distance in inches
    #define DIST_TRIGGER 8 // trigger distance 8 inches
    #define SONAR_DELAY 40000 // sonar delays 40ms everytime the system calls
    #define DIST_FIND 20 // target recognition distance 20 inches
    #define SPEED_FACTOR 100000 // normalize the speed
    #define MAX_WAIT_TIME 1000 // speed measurement time-out interval
    #define TURN_FACTOR 7.5 // time multiplier in the while loop when the robot turns
    #define NORMAL_SPEED 20 // normal speed of the robot
#endif
```

The first file, Main.c, contains the crucial method of the Finite State Machine, fsm(). It is called every cycle in the main function to function properly.

Listing 3: Finite State Machine switch, fsm()

```
int STATE = INIT; // STATE is a global register that stores the current state.
void fsm(void){   // this should be infinitely looped
  /*
   *state breakdown:
   *initial state: halt, nothing, just initializing
   *- go to searching state after initialization
   *searching state: search until target found
   *- when target found, go to engage state;
   *engage state: go to the target, will call *CorrectionLeft&Right to adjust its routes
```

```
    *- branch 1: if reach target close within certain *distance, go to active element
    *trigger state
    *- branch 2: if cannot find target, go to *searching state
    *active element trigger state: release trigger, *push enemy
    *- go to engage state
    */
    // each state method called below is implemented in fsm.c.
    // such method not only executes what to be done in that state,
    // it also returns the value of the next state the system should go to.
    switch(STATE){
     case INIT:
       STATE = init_0(STATE);
       break;
     case SEARCH:
       STATE = search_0(STATE);
       break;
     case ENGAGE:
       STATE = engage_0(STATE);
       break;
     case TRIGGER:
       STATE = trigger_0(STATE);
       break;
     default:
       STATE = INIT;
       break;
    }
}
```

Listing 4: Main Function, main()

```
int main(void) {

    DDRB |= 0b00110111; // Setting ports as output, PB0~2 & PB4 are P0 and P1 ports for
    // the H-bridge. PB5 is the debug LED.
    DDRD |= 0b00000010; // PD1 is another debug LED.
    DDRC |= 0b00000011; // PC0 controls the trigger servo,
    // as PC1 is the LED that indicates out of circle.
    PORTC &= ~(1<<0);
    EICRA |= (1<<2)|1;      //Interrupt settings
    EIMSK |= (1<<1)|1;
    PCMSK2 |= (1<<PCINT20);
    PCICR |= (1<<PCIE2);
    sei();                  //enable interrupt

    while(1){               //loop
    fsm();
    }
}
```

The second file, fsm.c, contains all the useful methods of searching the target and the state functions that executes certain actions and returns the next state.

Listing 5: Directional Correction Methods

```
/*
*Before going into complete search, the robot calls correction methods
*in engage state to briefly adjust the position, trying to
*find the target.
*/
volatile int dir = 0; // 0 for right, 1 for left
// global indicator for direction used by search() method.

// setters for the direction variable.
void setLeft(void){
```

```
  dir = 1;
}

void setRight(void){
  dir = 0;
}

// correction methods
int CorrectionRight(void){
MotorAction(RIGHT); // turns clockwise
initSonar(); // initializes the sonar
int counter = 0; // initializes the counter
while(counter < 10){ // stays in the loop until the counter reaches 10
    if (getSonar() < DIST_FIND) {
      return 1; // if the sonar finds the target again, return True.
    }
    counter ++;
}
return 0; // if fails, return false
}

int CorrectionLeft(void){
  MotorAction(LEFT); // the robot rotates counter-clockwise.
  initSonar();
  int counter = 0;
  while(counter < 10){
    if (getSonar() < DIST_FIND) {
      return 1;
    }
    counter ++;
  }
  return 0;
}
```

Listing 6: FSM State Methods

```
int init_0(int state){
  /*
   * initialization state
   * directly goes into searching state at the next cycle
   */
  return SEARCH;
}

int search_0(int state){
  /*
   * searching state
   * calls search method and goes to engage state when done
   * at the end of the state, the robot should face at the opponent
   */
  Search(); // calls the search method
  return ENGAGE;
}

int engage_0(int state){
  /*
   * engage state
   * walks up to opponent
   * if lose target, call correction methods
   * if correction fails, go to searching state
   * if target is very close, go to trigger state
   */
  MotorAction(FORWARD); // moves forward
  initSonar();
```

```
  if (getSonar() > DIST_FIND) {
    int stat; // stat is a Boolean value determining whether the correction succeeded.
    if (!dir) { // using the current direction preference
    stat = CorrectionLeft();
    } else {
      stat = CorrectionRight();
    }
    if (!stat) {return SEARCH;} // if correction fails, go to search state
  }
  if ((getSonar() < DIST_TRIGGER)) { // if target is very close, go to trigger state
    return TRIGGER;
  }
  return ENGAGE; // stays in engage state if nothing happens
}

int trigger_0(int state){
  /*
   * trigger state
   * release the trigger, and go back to engage state
   * this state only gets activated one time
   */
   PORTC |= (1<<PC0); // turn on the trigger servo for 500ms
   _delay_ms(500);
   PORTC &= ~(1<<PC0);
   setLeft(); // direction preference on the left side,
   // since the trigger slaps the enemy to the left
   return ENGAGE; // go back to engage state
}
```

The third file, Motor.c, contains important methods of motor movement, including the logical expression of P0 P3.

Listing 7: Motor Movement Methods

```
volatile int action = OFF; // global variable storing the current action of the motor

int getAction(void) { // getter of action
  return action;
}

/*
 * The MotorAction() method is implemented twice below, taking two different input:
 * 1. The input of three integers indicating movement, turning and direction.
 * 2. The input of a constant state defined in the header
 */
void MotorAction(int mov, int turn, int positive){
  // assign logical expressions to P0~P3
  int p0 = mov & ~positive;
  int p1 = mov & positive;
  int p2 = mov & ~(positive^turn);
  int p3 = mov & (positive^turn);
  // assign P0~P3 to PORTB
  PORTB = (PINB & (0b11101000)) | ((p0<<PB0)|(p1<<PB1)|(p2<<PB2)|(p3<<PB4));
  // save current action
  action = FORWARD*(mov & ~turn & positive) + BACKWARD*(mov & ~turn & ~positive)
  + LEFT*(mov & turn & ~positive) + RIGHT*(mov & turn & positive);
}

void MotorAction(int act){
  action = act;
  if (act == FORWARD) {
    PORTB |= (1<<5);
    PORTB &= (~(1<<PB0)); // forward
    PORTB |= (1<<PB1);
```

```
    PORTB &= (~(1<<PB2));
    PORTB |= (1<<PB4);
  } else if (act == BACKWARD) {
    PORTB &= ~(1<<5);
    PORTB |= (1<<PB0); // backward
    PORTB &= (~(1<<PB1));
    PORTB |= (1<<PB2);
    PORTB &= (~(1<<PB4));
  } else if (act == LEFT) {
    PORTB |= (1<<PB0); // counter-clockwise
    PORTB &= (~(1<<PB1));
    PORTB &= (~(1<<PB2));
    PORTB |= (1<<PB4);
  } else if (act == RIGHT) {
    PORTB &= (~(1<<PB0)); // clockwise
    PORTB |= (1<<PB1);
    PORTB |= (1<<PB2);
    PORTB &= (~(1<<PB4));
  } else if (act == OFF){
    PORTB &= (~(1<<PB0)); // stop
    PORTB &= (~(1<<PB1));
    PORTB &= (~(1<<PB2));
    PORTB &= (~(1<<PB4));
  }
}

// The RobotTurn() method takes in a calibrated factor and turns the robot at a given angle.
void RobotTurn(int positive, double angle){
  double t = angle * TURN_FACTOR;
  MotorAction(1,1,positive);
  for(int i = 0; i < t; i++){
    _delay_ms(1);
  }
}
```

The fourth file, QTI.c, implements the interrupt method that handles the situation when the robot reaches the while circle at the edge of the arena.

Listing 8: QTI Interrupt Methods

```
// Left QTI
ISR(INT1_vect)
{
  setRight();
  if ((PIND >> 3) & 1) { // on color black, pin high
  } else { // on color white, pin low
    while(!((PIND >> 3) & 1)){ // while on color white
      MotorAction(RIGHT); // turn right
      if (!((PIND >> 2) & 1)) { // if both QTI on white, perform emergency backing
        MotorAction(BACKWARD);
        _delay_ms(1200);
      }
    }
  }
}

// Right QTI
ISR(INT0_vect)
{
  setLeft();
  if ((PIND >> 2) & 1) {
  } else {
    PORTB &= ~(1<<5); // on white, pin low
    while(!((PIND >> 2) & 1)){
```

```
            MotorAction(LEFT); // turn left
            if (!((PIND >> 3) & 1)) { // if both QTI on white, perform emergency backing
                MotorAction(BACKWARD);
                _delay_ms(1200);
            }
        }
    }
}

// Center QTI
ISR(PCINT2_vect){
    if(!((PIND >> 4) & 1)){   // on white, pin low
        while(1){ // kill all the movement and stays in the loop forever
            MotorAction(OFF);
            PORTC |= (1<<PC1);
            PORTC &= ~(1<<PC0);
        }
    }
}
```

The fifth file, Sonar.c, contains all the methods to utilize the sonar to get the distance between the robot and enemy.

Listing 9: Sonar Methods

```
// set sonar on PB3 - PCINT3
// if we want to get the distance, simply use getSonar()
// this returns an int as distance proportional to the returning pulse length.

volatile unsigned int flag = 0; // global interrupt flag

void initSonar(void)
{
  PCICR |= (1 << 0); // enable interrupt for PCINT7...0
  sei(); // enable global interrupt
  TCCR1B |= ((1 << 1)|(1 << 0)); // set timer 1's prescaler to 8
}

void startSonarMeasurement(void)
{
  DDRB |= (1 << 3); // set data direction for PB3 as output
  PORTB |= (1 << 3); // toggle PB3 to high for 5 us
  _delay_us(5);
  PORTB ^= (1 << 3); // set PB3 back to low, this is a trigger pulse
  DDRB &= ~(1<<3); // data direction for PB3 as input
  PCMSK0 |= (1 << 3); // enable PCINT3 interrupt mask
}

void resetTimer(void) //resets the timer value to 0
{
  unsigned char sreg;
  sreg = SREG;
  cli();
  TCNT1 = 0;
  SREG = sreg;
}

unsigned int readTimer(void) // reads the current timer value as an int
{
  unsigned char sreg;
  unsigned int i;
  sreg = SREG;
  cli();
  i = TCNT1;
```

```
  SREG = sreg;
  return i;
}

ISR(PCINT0_vect)
{
  if ((PINB & (1<<3)) == 0b00001000) // if PB3 is high - its at rising edge
  // reset timer and start counting, flag = 0
  {
    flag = 0;
    resetTimer();
  }
  else
  {
    flag = 1; // if PB3 is low - at falling edge: toggle the flag.
    PCMSK0 &= ~(1<<3); // disable interrupt
  }
}


int getSonar(void)
{
  volatile unsigned int c = 0; // internal variable for timer
  flag = 0; // set flag to 0
  startSonarMeasurement(); // start the trigger pulse
  while (flag == 0) { // this loop waits for the flag to toggle
  // wait for the return pulse to finish
  }
  c = SONAR_FACTOR*readTimer(); // read the timer value to c once finish.
  _delay_us(SONAR_DELAY);
  return c;
}
```

The last part of the code, Speed.c, implements the function of measureing the current speed of the robot using a QTI sensor. Although we did not put on the QTI during the competition due to time constraints, we still made this part fully working.

Listing 10: Speed Measurement Methods

```
// set QTI on PC0 - PCINT8
// we wrote the file before implementing the trigger. since we didn't use
//the speed measurement, we set the trigger servo to PC0 instead in the competition.
// if we want to get the speed, simply use getSpeed()
// this returns an int proportional to the rotation speed of the wheel.

volatile unsigned int flag_s = 0; // global flag to measure time interval

void initSpeed(void)
{
  PCICR |= (1 << PCIE1); // enable interrupt for PCINT14...8
  sei(); // enable global interrupt
  TCCR1B |= ((1 << CS12)|(1 << CS10)); // set timer 1's prescaler to 1024
}

void startSpeedMeasurement(void) {
  DDRC &= ~(1<<PC0); // data direction for PC0 as input
  PCMSK1 |= (1 << PCINT8); // enable interrupt mask
}

ISR(PCINT1_vect){
  if (PINC & 1) {
    flag_s = 0;
    resetTimer(); //on color black, reset timer
  } else {
```

```
    flag_s = 1; //on color white, toggle flag, counting complete
    PCMSK1 &= ~(1 << PCINT8); //disable interrupt
  }
}

int getSpeed(void)
{
  volatile unsigned int c;
  startSpeedMeasurement();
  int i = 0;
  while (flag_s == 0) { // this loop waits for the flag to toggle
  // wait for the color to change
  i++;
  _delay_ms(1);
  // if the measurement exceeds maximum waiting time, return a speed of 0.
  if (i > MAX_WAIT_TIME) {return 0;}
  }
  c = SPEED_FACTOR/readTimer(); //c is an int proportional to speed
  // therefore inverse proportional to timer value
  return c;
}

// the following method checks whether the current speed is lower than the expected speed.
int checkSpeed(void)
{
  int spd = getSpeed();
  if (spd > NORMAL_SPEED) {
    return 1;
  } else {
    return 0;
  }
}
```