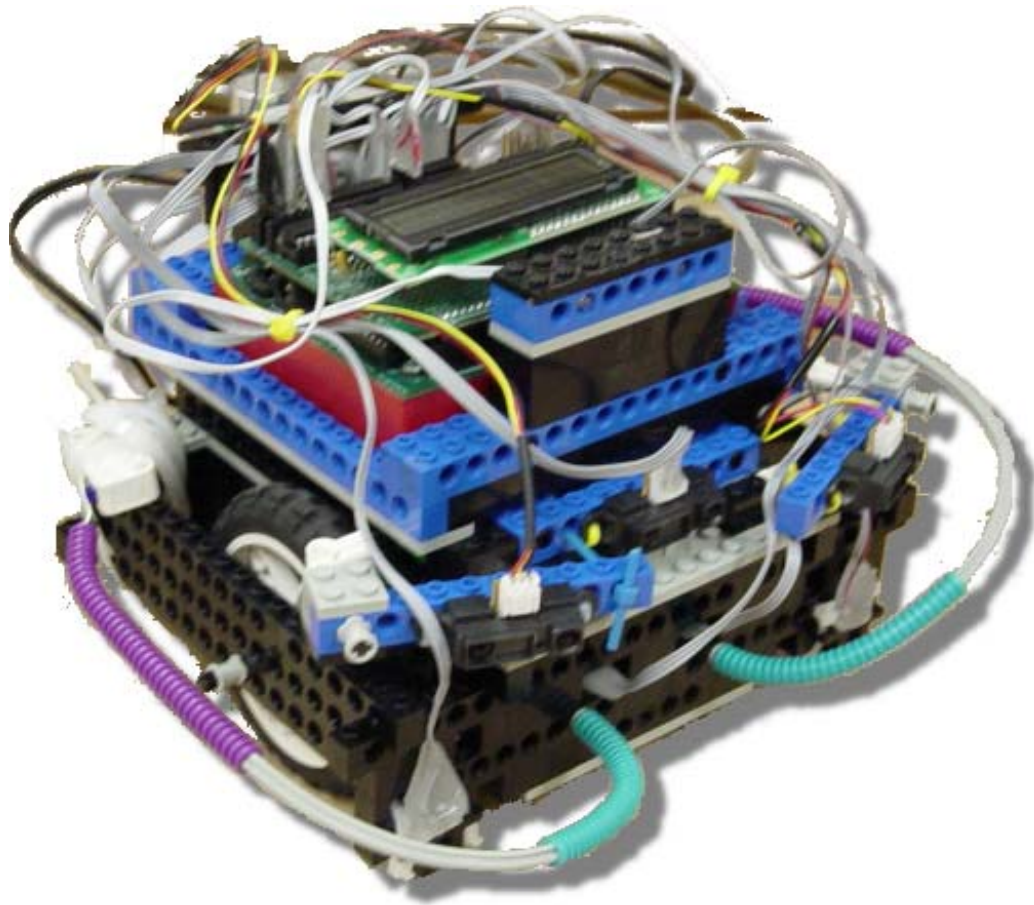


## Project 3 Report

### Team Speedy

Mark III: Actually Speedy!



#### Members:

Michael Noland

Lance Maddex

## Table of Contents

• Overview .....	3
• Wander Behavior.....	4
• Boundary Avoidance Behavior .....	5
• Avoiding and Following Behaviors .....	6
• Bump sensors .....	8
• Killswitch .....	8
• Motor Speed Controller .....	9
• Assumptions .....	10
• Analysis of group and individual behavior ....	11
• Lessons learned .....	13

## Appendices

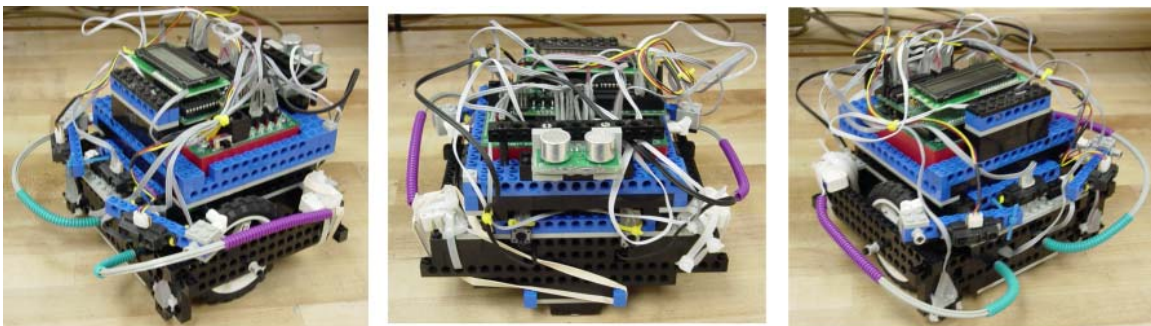
• Appendix 1, Behavior arbitration .....	14
• Source code listing .....	15

## Overview of Speedy mark III

### Vital Statistics

- Powered by a Handyboard with expansion board (68HC11 MCU).
- Reactive Model used (see Appendix 1 for the arbitration scheme)
- Sensors used:
  - 2 'tophat' IR sensors
  - 3 Sharp IR rangefinders
  - 2 breakbeam IR sensors
  - 2 lever arm touch switches
  - 2 rod touch switches (unused in final form)
  - 2 light sensors
  - 1 SRF-04 ultrasonic rangefinder
- Velocity of 12.0 cm/sec when heading in a straight line.
- 10.0 cm/sec when turning.

### Speedy the Robot



*We'd like to thank the makers of zip-ties and rubber bands. Speedy would have not been possible without their gracious contributions.*

## Wander Behavior

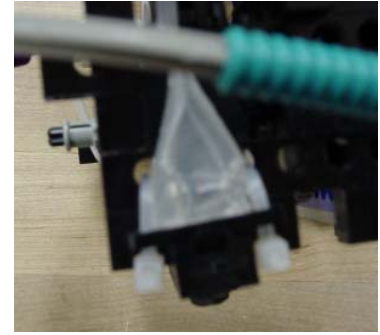
Speedy has a combination of wandering and light seeking as its lowest priority behavior. The function of this behavior is controlled by a pair of light sensors mounted on the top of the robot, in front of the Handyboard. The sensors are baffled by a two unit thick Lego hole, which makes the sensors considerably directional. The light was originally meant to be polarized, but everyone's sensors proved directional enough to make it unnecessary.



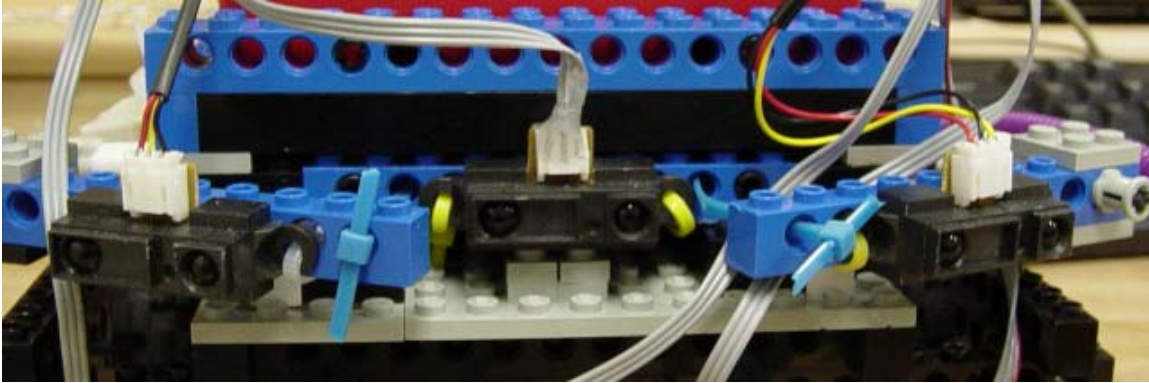
When Speedy sees light in only one of the two 'eyes' (i.e. one sensor is below the threshold), it turns in that direction, attempting to get light in both eyes. Once light is seen in both eyes, Speedy will head forwards towards the light at 12.0 cm/sec and set a flag so it will avoid other robots rather than follow them (see *Appendix 1* for the behavior hierarchy). Conversely, if light is not seen in either eye, the behavior will head forwards. Since this is the lowest priority behavior, Speedy seldom ends up actually going directly forward if no light is seen, instead the follow behavior usually subsumes this one.

## Boundary Avoidance Behavior

Speedy has two ‘tophat’ style IR sensors mounted facing downward on its front corners. These sensors are imprinted when the robot is turned on, they record the average value sensed when over a normal floor. The behavior itself compares this ambient value against the actual value constantly, looking for a change greater than a threshold (50.0 is currently used, but any value over 30.0 works well). When a change is detected on the left sensor, speedy will turn to the right for a period of time or until the black line is no longer detected, whichever is greater. Conversely, Speedy will turn to the left to avoid a right contact. The left sensor has a higher priority in the event where both sensors detect black at the exact same time. Overall, this behavior has the second highest priority (the only higher being the killswitch, which stops all motion completely), which is needed to keep the robot within the proscribed environment. We have not had any problems with keeping Speedy in the environment like some groups have, I believe this is because all of our behaviors are taken care of in the primary process in a continual loop, rather than making each behavior a separate process and *hoping* that the handyboard will service all of them in a reasonable time.



## Avoiding and following other robots



The heart of this project was sensing other robots sharing the environment and reacting to them. How Speedy should react depended on which trial was being conducted, but the final decision was to attempt to follow other robots when no light was detected, and to avoid other robots when trying to reach the light after sensing it. In the early trials, avoid worked fine, but following resulted in accidental aggregation for almost all of the robots, including Speedy. Due to the invalid range on the sharp IR sensors, the aggregation often turned into a Battle Royale, with the stronger robots accidentally pushing other robots out of the range or ripping sensors off.

Speedy has three Sharp IR sensors to detect other robots for either behavior; two are angled slightly to either side with the third pointing directly forward.

During following, Speedy attempts to maintain a reasonable distance between other robots, and keep the robot centered. This is done by turning towards a contact on either of the angled sensors, and moving forward as long as the center sensor reads within the 'happy' range. If the center sensor reads too far away, Speedy will stop following and allow the other lower precedence behaviors to take effect (this is to prevent Speedy from

attempting to follow a stationary object like a wall or someone's leg outside of the environment). However, if the sensor reads too close, Speedy will stop moving for at least 1 second. After this time, if the center sensor is ok, Speedy will continue moving forward, otherwise Speedy will turn completely around after an additional random timeout. The addition of this behavior to all robots was fairly effective in getting rid of the aggregation effect.

The avoid behavior is comparatively much simpler than the follow behavior. Speedy will turn away from an angled contact if the reading crosses one threshold and turn completely around if a closer threshold on the center sensor is crossed. This proved to be surprisingly effective, and with a large enough threshold to prevent speedy from crossing the invalid point on the Sharp IR sensors, Speedy never hit stationary objects. Occasionally Speedy and another robot would collide when both were turning away from something else, or one of the two crossed the other's invalid range.

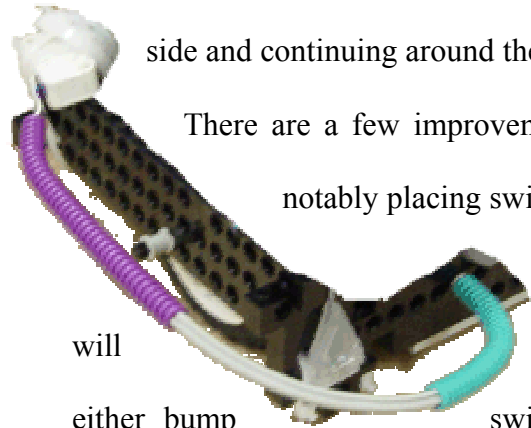
*Table of IR rangefinder tolerances (higher values indicate an object is closer):*

<b>Tolerances</b>	Follow behavior		Avoid Behavior
	Minimum reading for a contact	Maximum reading for comfort	Minimum reading for a contact
Left/Right sensors	80	N/A	60
Center sensor	40	100	80

*Note: All of these tolerances are C-style constants, and can be easily changed if desired.*

## Avoiding Physical Contact

Speedy has two bump sensors, one on either side. These have taken a number of forms over the course of project 3, but the final form is a flexible tube starting near the back side and continuing around the front and anchored in place on the front of the robot.



There are a few improvements that could be made to this arrangement, most notably placing switches on both ends of the tube, rather than just one, as a light contact near the front of the tube assembly will not trigger the switch. Speedy turns away from either bump switch contacts. There are also two legacy bump switches on Speedy's rear end, but the current code never makes Speedy reverse.

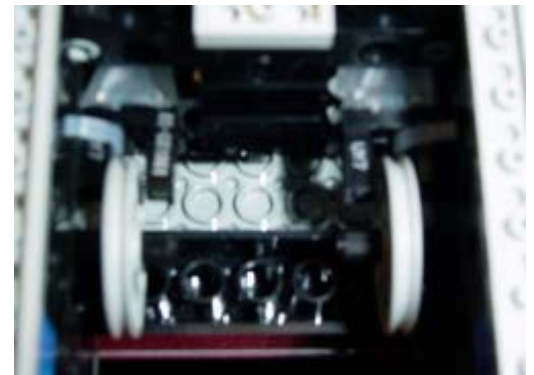
## Killswitch

Speedy has the SRF-04 ultrasonic rangefinding module mounted pointing upwards. This sensor acts as a killswitch or master off for all of Speedy's motion. When a hand is placed over this sensor, Speedy will continue to perform all of its other behaviors and print out any debug information, but will stop moving. This behavior is completely optional and was included mainly as a debugging convenience, so we would not have to chase Speedy across the room trying to read the dim LCD display. During the actual trials, we disabled the behavior by unplugging the digital lead from the module, to prevent another robot's wires from accidentally triggering it and making Speedy look dead.



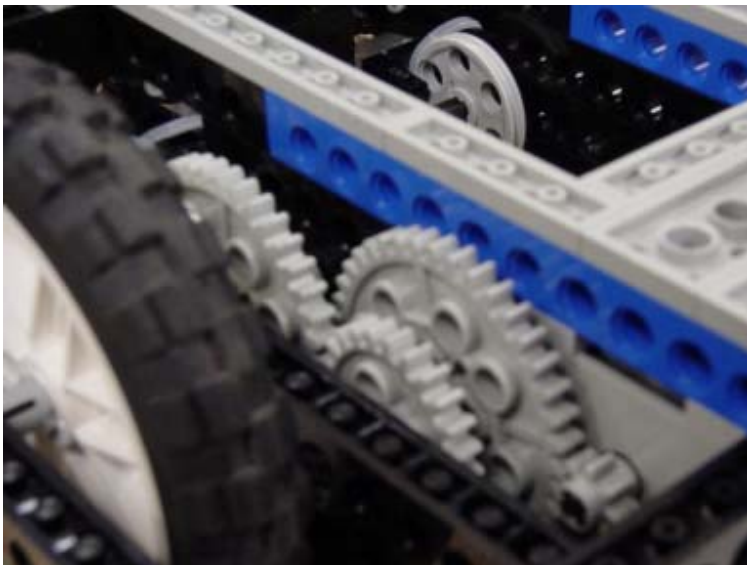
## Motor Speed Controller

On our previous projects, the encoder mountings were always an issue. Either an encoder would slip backwards and always read 1, or angle upwards and always read 0. For project 3, we designed the undercarriage with the encoder mounts in mind making a cavity underneath just for them. A careless hand



can no longer disturb the encoder when picking the robot up, and rather than keep them in place by pressure, each is also zip-tied to the wall as can be seen in the picture. The actual motor % power is adjusted approximately every 300 ms by a separate process running on the Handyboard, which measures the actual number of clicks on the encoders and adjusts the motor % power according to the difference between measured and desired speed:

$$\text{Desired clicks} = \text{desired velocity} * \text{clicks per revolution} / \text{wheel diameter} * \text{time elapsed}$$



Underside view of gear train

With the newly designed gear train, Speedy gets 60 clicks per revolution of a tire. In both of the previous projects, Speedy had a resolution of 100 clicks per revolution, but in practice we noticed no change in straight-line behavior.

## **Environmental Assumptions**

The primary assumption is that the environment consists only of other robots, the light, and the boundaries of the universe (black Gaffer's tape). In addition, no effort is made to distinguish between the light and the other robots for the purposes of collision or following (indeed, we were told that any other obstacles in the environment could be considered robots). However, in the final test, Speedy did not try to follow the light and slam into it because of the decision to make robots avoid when they see the light rather than follow, so it worked out quite well (as long as the light was at the correct height for the Sharp IR sensors to detect it, otherwise the light sensors might see it and the collision avoidance code not).

## **Assumptions about other robots**

Other robots were assumed to be traveling in the proscribed speed range of 10 to 15 cm / sec, and to be at least approximately 3 inches tall, so our IR sensors could detect them.

## **Distinguishing Characteristics**

Speedy is slightly schizophrenic, not entirely sure what to do when placed in an otherwise empty environment. Without a light source or boundaries, it will wander around and chase ghosts, sometimes spinning in place for no particular reason.

## Analysis of isolated and group behavior

Speedy acts much more interesting in a group than in isolation. When there are no robots, Speedy will rotate until it finds the light in both eyes and head towards it, altering course only if it hits the boundary of the environment. When Speedy is placed in a group environment, the other behaviors become relevant and the net result is a much more interesting and animated robot.

### Trial 1

Boundary Avoidance: Worked fine (unless we were pushed out by another robot)

Avoid Robots: Worked fairly well, although two robots coming head on often collided because of the Sharp IR danger zone, as they would be too close to each other when they tried to turn around and all of a sudden no longer desired to turn around (as the sensors returned far away values).

Follow Robots: Ended up as a remarkably successful Aggregation trial, too bad that's not what it was *supposed* to be.

### Trial 2

Boundary Avoidance, Avoid Robots: Still great.

Light Detection: Speedy really isn't particularly interested in the light and only detects it when very close (about 1m away). This was a problem with the source code; the threshold was set far too low.

Narrow pathway: Speedy only made it through the pathway by chance, he wasn't really following during trial 2.

## **Trial 2 (continued)**

Follow Robots: With tweaked tolerances and the new timeout (where a robot has been stopped for a few seconds will turn 180 and continue onwards), this worked significantly better. The robots no longer did an aggregation impression, but most didn't really follow either (including Speedy). Speedy may as well have been doing random wandering.

## **Trial 3**

Boundary Avoidance, Avoid Behavior: Both worked fine.

Light detection: An adjusted threshold and much better code handling how to turn towards the light made Speedy find and lock onto it whenever it wasn't occluded by another robot. This working well set up the stage for Speedy to find other robots on its way to the light and follow them.

Following: We completely rewrote the follow code for trial 3, and it seemed to do the trick, as Speedy followed other robots to the light, and even through the narrow passageway.

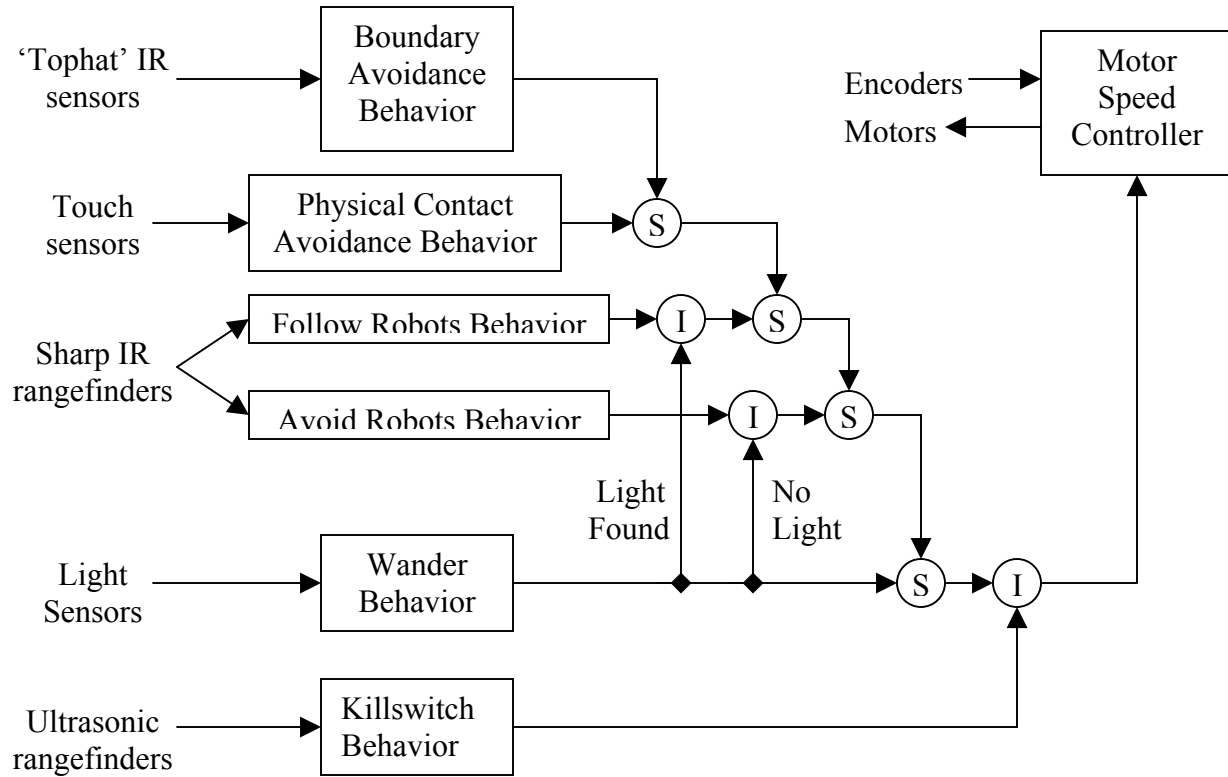
## Lessons learned from the project

Quite a few problems in project 2 were caused by rewiring the robot but not updating all of the places where sensors were read, so for project 3, we moved all of the sensors to #defines (not just the port, but the entire call to read a sensor). This uncovered problems (e.g. our left and right IRs were switched) and probably prevented some others.

Sharp IR rangefinders are interesting sensors, but their two to one mapping (ranges closer than 5 cm will look increasingly far away) can cause no end of hassle! In our first design for project 3, the IR sensors stuck out of the front by over an inch, meaning that any robot that came fairly close to Speedy would appear far enough away that Speedy would continue forward and plow right into them. For the final test, I moved the IR sensors back 3 Lego squares, and added an extension to the front to keep other robots from coming too close, but it didn't help that much (other robots wouldn't detect the plow and would come too close too fast, being in the danger range before Speedy had detected them at the stopping range). I would design future robots with these sensors in mind, keeping them as close to the center of the robot as possible, with as much of the invalid range inside the robot as possible.

A faster robot is indeed a better robot; you don't have to worry about old age kicking in before your robot reaches the other side of the environment. One small disadvantage of the chassis used for project 3 is the direct gearing from motor to wheel, instead of the single belt connecting the motor to the rest of the drive train on the previous two projects. Speedy will now make many grinding and gnashing sounds when forcefully stopped (e.g. by another robot), signaling potential damage to the gears and motor, and if left in this state, to the H-Bridge driver chip on the Handyboard unless it has overheat detection.

## Appendix 1: Behavior Arbitration



## Appendix 2: Source Code Listing

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Project 3 for CECS 373:
//   Winter Semester 2003
//   University of Missouri, Columbia
//
// Team Members:
//   Michael Noland
//   Lance Maddex
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Transcendental numbers
#define PI 3.14159

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Physical robot constants //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Clicks per rotation of output axle
#define gConstant 60.0

// Wheel circumference in centimeters
#define dConstant 26.1

// Wheel seperation in centimeters
#define lConstant 13.5

// Clicks per centimeter
#define gOverD (gConstant/dConstant)

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Motor state variables //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Motor process state variables
int motorPID;
float motorLastTime;
int motorRunning;

// Last actual power (-100..100%) written to the handyboard
int motorL, motorR;

// Desired velocity in cm/s
float velL, velR;

// Desired velocity in clicks/s
float clickL, clickR;

// Callibration factors
float callibrationL, callibrationR;

// A flag that indicates when a motor speed is out of feasible range
int motorError;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Sensors //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#define LEFT_ENCODER 2
#define RIGHT_ENCODER 3
#define LEFT_MOTOR 1
#define RIGHT_MOTOR 3
#define LEFT_ET analog(19)
#define MIDDLE_ET analog(18)
#define RIGHT_ET analog(17)
#define LEFT_TOPHAT analog(3)
#define RIGHT_TOPHAT analog(2)
#define LEFT_LIGHT analog(22)
```

```
#define RIGHT_LIGHT    analog(23)
#define RIGHT_TOUCH    digital(15)
#define LEFT_TOUCH     digital(14)
#define BACK_RIGHT_TOUCH    digital(10)
#define BACK_LEFT_TOUCH    digital(9)
#define LEFT_LIGHT     analog(22)
#define RIGHT_LIGHT    analog(23)

////////////////////////////////////
// Surface Color Constants //////////////////////////////////////
////////////////////////////////////

#define BLACK_RIGHT 1
#define BLACK_LEFT  2
#define COLOR_BROWN 0

#define BLACK_LINE_TOLERANCE 50.0
#define BOUNDARY_TIMEOUT 0.4

////////////////////////////////////
// IR distance tolerances //////////////////////////////////////
////////////////////////////////////

#define LOW_LR_TOLERANCE 60
#define TURN_TOWARDS_TOLERANCE 80

#define HIGH_TOLERANCE 250

#define LOW_AVOID_TOLERANCE 80
#define HAPPY_TOLERANCE 40

#define TOO_CLOSE_TOLERANCE 100

#define AVOIDANCE_TIMEOUT 0.9

////////////////////////////////////
// Bump sensors //////////////////////////////////////
////////////////////////////////////

#define BUMP_TIMEOUT 0.2
#define STOP_TURN_TIMEOUT 2

#define KILLSWITCH_THRESHOLD 150

////////////////////////////////////
// Velocity constants //////////////////////////////////////
////////////////////////////////////

#define FULL_AHEAD_SPEED 12.0
#define TURN_SPEED 10.0
#define INITIAL_VEL 10.0

////////////////////////////////////
// Light and killswitch constants //////////////////////////////////////
////////////////////////////////////

#define LIGHT_FOUND 15

////////////////////////////////////
// State variables //////////////////////////////////////
////////////////////////////////////

float leftHatAverage, rightHatAverage;
float stoppingEvent = 0.0;
int followStopping = 0;
int avoidanceRotating = 0;
float avoidanceEvent = 0.0;
float rotationEvent = 0.0;
int rotatingDir = COLOR_BROWN;
int bumpRotating = 0;
float bumpEvent = 0.0;
float stallEvent = 0.0;
float wanderEvent = 0.0;
```

```
int wanderMode = 0;
float lightSign = -1.0;
int lightFound = 0;

/////////////////////////////////////////////////////////////////
// Math utility functions ///////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////

float abs(float a) {
    if (a < 0.0) return -a;
    return a;
}

/////////////////////////////////////////////////////////////////

float min(float a, float b) {
    if (a < b) return a;
    return b;
}

/////////////////////////////////////////////////////////////////

void updateMotors(void) {
    // Compute the new power percentages and apply them
    motorL = (int)(long)(callibrationL*clickL);
    motorR = (int)(long)(callibrationR*clickR);
    motor(LEFT_MOTOR, motorL);
    motor(RIGHT_MOTOR, motorR);

    // Check for an out of range power percentage
    motorError = ((motorL < -100) || (motorL > 100) ||
                 (motorR < -100) || (motorR > 100));
}

/////////////////////////////////////////////////////////////////
// The motor process: handles automatic recalibration ///////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////

void motorCallibrate(int junk) {
    float curTime;
    float delta;
    float actualL, actualR;

    // Recalibrate the motor
    while (1) {
        if (motorRunning) {
            // Get the time
            curTime = seconds();
            delta = curTime - motorLastTime;
            if (delta > 0.3) {
                motorLastTime = curTime;

                // Read the actual encoder values
                actualL = (float)read_encoder(LEFT_ENCODER);
                actualR = (float)read_encoder(RIGHT_ENCODER);

                // Reset encoders
                reset_encoder(LEFT_ENCODER);
                reset_encoder(RIGHT_ENCODER);

                // Recompute the calibration factors
                if (actualL > 0.0) callibrationL = abs((clickL * delta) / actualL);
                if (actualR > 0.0) callibrationR = abs((clickR * delta) / actualR);
                // printf("(%d,%d) (%d,%d) (%d,%d)\n",
                //         (int)(callibrationL*100.0), (int)(callibrationR*100.0),
                //         (int)actualL, (int)actualR, motorL, motorR);

                // Set the motors to the desired speed (factor f * clicks / sec)
                updateMotors();
            }
        }
    }

    // Relinquish control
}
```

```
        defer();
    }
}

////////////////////////////////////////////////////////////////

void newMotion(float newL, float newR) {
    // Stop the motor process
    motorRunning = 0;

    // Copy the desired velocities
    velL = newL;
    velR = newR;

    // Compute the desired number of clicks / second
    clickL = velL * gOverD;
    clickR = velR * gOverD;

    updateMotors();

    // Reset the encoders
    reset_encoder(LEFT_ENCODER);
    reset_encoder(RIGHT_ENCODER);

    // Start the motor recalibration process
    motorRunning = (abs(velL) > 0.0) && (abs(velR) > 0.0);
}

////////////////////////////////////////////////////////////////

void initMotors(void) {
    motorLastTime = seconds();
    motorRunning = 0;
    motorPID = start_process(motorCalibrate(0), 50);

    calibrationL = 1.0;
    calibrationR = 1.0;
    newMotion(0.0, 0.0);
}

////////////////////////////////////////////////////////////////

void destroyMotors(void) {
    if (motorPID) kill_process(motorPID);
    motorRunning = 0;
    alloff();
}

////////////////////////////////////////////////////////////////

void stopMotion(void) {
    // Stop the motor process
    motorRunning = 0;

    // Kill the motors
    alloff();

    // Set the velocities to 0
    velL = 0.0;
    velR = 0.0;
}

////////////////////////////////////////////////////////////////

void waitForStart(void) {
    printf("Press start\n");
    while (!start_button()) sleep(0.1);
}

////////////////////////////////////////////////////////////////
//
// Called on startup to "imprint" the line-detecting behavior
```

```
//  
////////////////////////////////////  
  
void callibrateTophats(void) {  
    leftHatAverage = (float)LEFT_TOPHAT;  
    rightHatAverage = (float)RIGHT_TOPHAT;  
}  
  
////////////////////////////////////  
//  
// Reads the tophat sensors and returns one of three values depending  
// on the light detected: black, white, or brown  
//  
////////////////////////////////////  
  
int colorDetected(void) {  
    float left = (float)LEFT_TOPHAT;  
    float right = (float)RIGHT_TOPHAT;  
  
    if (left >= leftHatAverage + BLACK_LINE_TOLERANCE)  
        return BLACK_LEFT;  
    if (right >= rightHatAverage + BLACK_LINE_TOLERANCE)  
        return BLACK_RIGHT;  
  
    return COLOR_BROWN;  
}  
  
////////////////////////////////////  
//  
// avoid robots behavior:  
// attempts to avoid other robots, by turning away from side IR  
// contacts, and turning completely around in the event of a center  
// contact.  
//  
////////////////////////////////////  
  
void avoidRobots(float * leftVel, float * rightVel) {  
    int left = LEFT_ET;  
    int middle = MIDDLE_ET;  
    int right = RIGHT_ET;  
    int centerContact = (middle < HIGH_TOLERANCE) && (middle > LOW_AVOID_TOLERANCE);  
    int leftContact = (left < HIGH_TOLERANCE) && (left > LOW_LR_TOLERANCE);  
    int rightContact = (right < HIGH_TOLERANCE) && (right > LOW_LR_TOLERANCE);  
  
    // Turn right  
    if (leftContact && !rightContact) {  
        *leftVel = -TURN_SPEED;  
        *rightVel = TURN_SPEED;  
        printf("Left contact (%d,%d,%d)\n", left, middle, right);  
    }  
  
    // Turn left  
    if (rightContact && !leftContact) {  
        *leftVel = TURN_SPEED;  
        *rightVel = -TURN_SPEED;  
        printf("Right contact (%d,%d,%d)\n", left, middle, right);  
    }  
  
    // Turn around if we get a center contact  
    if (centerContact && !avoidanceRotating) {  
        avoidanceEvent = seconds() + AVOIDANCE_TIMEOUT;  
        avoidanceRotating = 1;  
        if (leftContact) avoidanceRotating = -1;  
    }  
  
    if (avoidanceRotating) {  
        if (avoidanceRotating > 0)  
            *leftVel = -TURN_SPEED;  
        else  
            *leftVel = TURN_SPEED;  
  
        *rightVel = -*leftVel;  
    }  
}
```

```

        if (avoidanceEvent < seconds()) avoidanceRotating = 0;
        printf("Center rotation (%d,%d,%d)\n", left, middle, right);
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// follow robots behavior:
// attempts to keep things within a certain range of the center IR
// sensor, by turning towards things that are too close to either of
// the angled IRs and stopping if it gets too close.
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void followRobots(float * leftVel, float * rightVel) {
    int left = LEFT_ET;
    int middle = MIDDLE_ET;
    int right = RIGHT_ET;

    int additional;

    int centerContact = (middle < HIGH_TOLERANCE) && (middle > TOO_CLOSE_TOLERANCE);
    int centerHappy = (middle < HIGH_TOLERANCE) && (middle > HAPPY_TOLERANCE);
    int leftContact = (left < HIGH_TOLERANCE) && (left > TURN_TOWARDS_TOLERANCE);
    int rightContact = (right < HIGH_TOLERANCE) && (right > TURN_TOWARDS_TOLERANCE);

    if (centerHappy && !centerContact) {
        leftContact = 0;
        rightContact = 0;
    }

    if (centerHappy && !centerContact && !followStopping) {
        *leftVel = FULL_AHEAD_SPEED;
        *rightVel = FULL_AHEAD_SPEED;
        printf("Full Speed Ahead(%d,%d,%d)\n", left, middle, right);
    }

    if (leftContact && !rightContact) {
        *leftVel = TURN_SPEED;
        *rightVel = -TURN_SPEED;
        printf("Left contact (%d,%d,%d)\n", left, middle, right);
    }

    if (rightContact && !leftContact) {
        *leftVel = -TURN_SPEED;
        *rightVel = TURN_SPEED;
        printf("Right contact (%d,%d,%d)\n", left, middle, right);
    }

    // Turn around (eventually) if we get a center contact
    if (centerContact && !followStopping) {
        stoppingEvent = seconds() + 1.0;
        stallEvent = stoppingEvent + (float)random(STOP_TURN_TIMEOUT) + 1.0;
        followStopping = 1;
    }

    if (followStopping) {
        *leftVel = 0.0;
        *rightVel = 0.0;

        if (stoppingEvent < seconds()) {
            if (centerContact) {
                if (stallEvent > seconds()) {
                    *leftVel = -TURN_SPEED;
                    *rightVel = TURN_SPEED;
                } else
                    followStopping = 0;
            } else {
                followStopping = 0;
            }
        }
    }
    printf("Center turning (%d,%d,%d)\n", left, middle, right);
}

```

```

    }
}

/////////////////////////////////////////////////////////////////
//
// Out of Boundary Avoidance Behavior:
// Input sensors: Tophat line detector pair
// Output actuators: Drive motors
// Detects the black lines which denote the boundaries.
// If a black line is detected, it will turn away from it for
// BOUNDARY_TIMEOUT seconds.
//
// rotatingDir is the direction we're rotating in, or 0
// rotationEvent is the target time to stop rotating (relative to a seconds() call)
//
/////////////////////////////////////////////////////////////////

void boundaryAvoidanceBehavior(float * leftVel, float * rightVel) {
    int color = colorDetected();

    // Check the color of the ground
    //   if (rotatingDir == COLOR_BROWN) {
    if ((color == BLACK_RIGHT) || (color == BLACK_LEFT)) {
        rotatingDir = color;
        rotationEvent = seconds() + BOUNDARY_TIMEOUT;
    } //else
    //rotatingDir = COLOR_BROWN;
    // }

    // Set the correct velocity if we are turning
    if (rotatingDir == BLACK_RIGHT) {
        *leftVel = -10.0;
        *rightVel = 10.0;
        lightSign = 1.0;
    }
    if (rotatingDir == BLACK_LEFT) {
        *leftVel = 10.0;
        *rightVel = -10.0;
        lightSign = -1.0;
    }

    // Check to see if we should stop rotating
    if (rotatingDir != COLOR_BROWN) {
        printf("Detected black l(c=%d, d=%d)!\n", color, rotatingDir);
        if (rotationEvent < seconds()) {
            rotatingDir = COLOR_BROWN;
        }
    }
}

/////////////////////////////////////////////////////////////////
//
// ADDED BY LANCE (Whole function)
//
/////////////////////////////////////////////////////////////////

void bumpBehavior(float * leftVel, float * rightVel){
    int right, left;
    left = right = 0;

    if (RIGHT_TOUCH) {
        right = 1;
        bumpRotating = 1;
        bumpEvent = seconds() + BUMP_TIMEOUT;
    }
    if (LEFT_TOUCH) {
        left = 1;
        bumpRotating = 1;
        bumpEvent = seconds() + BUMP_TIMEOUT;
    }

    if (right) {
        *leftVel = -TURN_SPEED;
        *rightVel = TURN_SPEED;
        printf("Compensating for right bump\n");
    }
}

```

```

    }
    if (left) {
        *leftVel = -TURN_SPEED;
        *rightVel = TURN_SPEED;
        printf("Compensating for left bump\n");
    }
}

/////////////////////////////////////////////////////////////////
//
// Wander behavior:
// Makes the robot head towards light if seen in both 'eyes', turn
// towards the light if only one eye sees it, and merely go
// straight forward when there is no light at all.
//
/////////////////////////////////////////////////////////////////

#define TOL 0

void wanderBehavior(float * leftVel, float * rightVel) {
    int right = RIGHT_LIGHT;
    int left = LEFT_LIGHT;

    *leftVel = FULL_AHEAD_SPEED;
    *rightVel = FULL_AHEAD_SPEED;

    // Keep scanning if needed
    lightFound = 1;
    if (left > LIGHT_FOUND) {
        lightFound = 0;
        if (right > LIGHT_FOUND) {
            // Both are too dim, turn away from the black line
            *leftVel = -lightSign * FULL_AHEAD_SPEED;
            *rightVel = -*leftVel;
        } else {
            // Left is too dim, turn right
            *leftVel = FULL_AHEAD_SPEED;
            *rightVel = -*leftVel;
        }
    } else {
        lightFound = 0;
        if (right > LIGHT_FOUND) {
            // Right is too dim, turn left
            *leftVel = -FULL_AHEAD_SPEED;
            *rightVel = -*leftVel;
        }
    }
}

/////////////////////////////////////////////////////////////////
//
// Stall behavior:
// Kills all motion when the robot detects a hand over its
// ultrasonic rangefinder, mounted upwards.
//
/////////////////////////////////////////////////////////////////

void stallBehavior(float * leftVel, float * rightVel) {
    if (sonar() < KILLSWITCH_THRESHOLD) {
        *leftVel = 0.0;
        *rightVel = 0.0;
        printf("Killswitch\n");
    }
}

/////////////////////////////////////////////////////////////////
//
// The behavior handler:
// Coordinates priorities of the different behaviors.
//
/////////////////////////////////////////////////////////////////

void behaviorHandler(float elapsed) {

```

```

float leftVel, rightVel;

// Locomotion behavior hierarchy
leftVel = rightVel = 0.0;

wanderBehavior(&leftVel, &rightVel);
if (lightFound) {
    avoidRobots(&leftVel, &rightVel);
} else {
    followRobots(&leftVel, &rightVel);
}
bumpBehavior(&leftVel, &rightVel);
boundaryAvoidanceBehavior(&leftVel, &rightVel);
stallBehavior(&leftVel, &rightVel);

// Set the plan into motion
newMotion(leftVel, rightVel);

// Debugging printout
//printf("l(%d,%d) th(%d,%d) dh(%d,%d) m(%d,%d)\n", /*LEFT_ET, MIDDLE_ET, RIGHT_ET,*/
//      LEFT_LIGHT, RIGHT_LIGHT, LEFT_TOPHAT, RIGHT_TOPHAT,
//      (int)leftHatAverage, (int)rightHatAverage, motorL, motorR);
}

////////////////////////////////////////////////////////////////

void main(void) {
    float lastTime, curTime;
    float leftVel, rightVel;

    // Hardware initialization
    enable_encoder(LEFT_ENCODER);
    enable_encoder(RIGHT_ENCODER);

    // Motor initialization

    // Wait for user input
    waitForStart();

    // Auto-calibration
    initMotors();
    calibrateTophats();
    calibrateTophats();

    // Motor calibration: run straight for a few secs to get the motor
    // calibration factors accurate (they start at 1.0, 1.0)
    newMotion(20.0, 20.0);
    sleep(2.0);
    newMotion(0.0, 0.0);
    sleep(1.0);

    // Main loop
    lastTime = seconds();
    while (1) {
        // Run the behaviours
        curTime = seconds();
        behaviorHandler(curTime-lastTime);
        lastTime = curTime;

        // Sleepage
        sleep(0.2);

        // Look for a stop button press
        if (stop_button()) {
            newMotion(0.0, 0.0);
            waitForStart();
        }
    }
}

```