Robot Code for Project 2
March 31, 2003
Team 4 – Justin Fuller, Rahul Kotamaraju, Matthew Lawrence

```
//==========================================================
//  Project 2
//  Last Edited on 03/28/03, 13:15
//==========================================================
//  PURPOSE:
//    Cause designed hardware to navigate through 8x12
//    arena, collecting points as it hits a goal (in this
//    case, a light source).  Also, avoid high objects,
//    which cause the loss of all points.  Ensure that the
//    robot is robust enough to handle situations like
//    locked movement and lack of stimuli.
//==========================================================
//  BEHAVIORS:
//    1) Cruise - cruise
//    2) Goal-seeking - goal
//    3) Stop Slipping - slip
//    4) High Object Avoid - high
//    There was another behavior proposed to avoid low
//    objects, but this was never implemented.  It would
//    rank between 1 and 2 in order.
//==========================================================
//  STRATEGY NOTES:
//    Maintain separation between sensor and motor code
//    Emulate schema behavior theory where prudent
//==========================================================

// M A C R O S ///////////////////////////////////////////
// Logic constants
#define TRUE -1
#define FALSE 0

// Base direction constants
#define LEFT 0
#define RIGHT 1

// Thread wait constants
#define SLEEP_TIME 0.12
#define SLEEPTHREAD_TIME 0.25
#define SLEEPWALK_TIME 0.9

// Motor "directions," commands to motor_control
#define POWER_FORWARD -3.
#define BACK -2.
#define SPIN_LEFT -1.5
#define SPIN_LEFT_1 -1.25
#define SPIN_RIGHT -1.
#define SPIN_RIGHT_1 -0.75
#define ANGLE_LEFT 0.
#define ANGLE_FRONT_LEFT 0.5
#define FRONT_LEFT 1.
#define FORWARD 1.5
#define FRONT_RIGHT 2.
#define ANGLE_FRONT_RIGHT 2.5
#define ANGLE_RIGHT 3.

#define HIGH_DIR ANGLE_RIGHT

// G L O B A L S  ////////////////////////////////////////
// releaser flags
int cruise_rel_present;
int low_rel_present; // relic from unused behavior
int goal_rel_present;
int slip_rel_present;
int high_rel_present;
```

```
// string for behavior tracking with LCD
char b[2];

// global flags for determining special conditions
int spinning;
int guide_direction;
int goal_on;
int goal_toggle;

// F U N C T I O N S ////////////////////////////////////
// utility function for determining absolute value
int abs(int a)
{
    if(a < 0)
      return -a;
    return a;
}

// CRUISE:
//  Special behavior that really doesn't require any input or
//  releaser, though it did have dummy functions at first.
//  They were deemed a waste of time, and so deleted.

float cruise_direction;
float cruise_guide = SPIN_LEFT_1;

void cruise_act()
{
    float rand_time;
    // since this is the only function used in cruising,
    // it holds the infinite loop for the cruising thread
    while (1)
      {
        // cruise has two components: spin and leg.
        // After spinning left for a random amount of time,
        // walk forward for a random amount of time
        // SPIN
        cruise_guide = SPIN_LEFT_1;
        cruise_direction = cruise_guide;
        rand_time = (float)random(244);
        rand_time = rand_time/1000.0 + 0.256;
        sleep(rand_time);
        defer();

        // LEG
        cruise_direction = FORWARD;
        rand_time = (float)random(1044);
        rand_time = rand_time/1000.0 + 0.256;
        sleep(SLEEPWALK_TIME+rand_time);
        defer();
    } // while (1)
} // void cruise_act()

void cruise()
{
    start_process(cruise_act());
} // void cruise()

// LOW:
//  Since this behavior may not be implemented, will assume
//  a sensor scheme that never detects objects

float low_direction = FORWARD;

// GOAL:
//  Releaser: Goal is in sight
//  Model: Direction of goal
//  Action: Move toward goal

// bound for looping
```

```
#define NUM_GOAL_SENSORS 4

// ports for goal sensors (here, light sensors)
#define GOAL_SENSOR_PORT_0 3
#define GOAL_SENSOR_PORT_1 4
#define GOAL_SENSOR_PORT_2 5
#define GOAL_SENSOR_PORT_3 6

// thresholds for triggering releaser
#define GOAL_SENSOR_THRESHOLD 80
#define GOAL_SENSOR_DIFF_THRESHOLD 2

// threshold for determining if light is straight ahead
#define FORWARD_THRESHOLD 2

// globals for indicating direction of travel and sensor values
float goal_direction;
int goal_sensor[NUM_GOAL_SENSORS];

// globals for releaser that may need use in other functions
int goal_thresh, goal_diff;

void goal_sense()
{
    // gather sensor values
    goal_sensor[0] = analog(GOAL_SENSOR_PORT_0);
    goal_sensor[1] = analog(GOAL_SENSOR_PORT_1);
    goal_sensor[2] = analog(GOAL_SENSOR_PORT_2);
    goal_sensor[3] = analog(GOAL_SENSOR_PORT_3);
} // void goal_sense()

int goal_sensors_above_threshold()
{
    int intCount;
    // Detect if sensors are on active side of thresholds
    for (intCount = 0; intCount < NUM_GOAL_SENSORS; intCount++)
      {
        if (goal_sensor[intCount] < GOAL_SENSOR_THRESHOLD)
          {
            return TRUE;
        } // if (goal_sensor[intCount] > GOAL_SENSOR_THRESHOLD)
    } // for (int intCount = 0; intCount < NUM_GOAL_SENSORS; intCount++)
    return FALSE;
} // int goal_sensors_above_threshold

int goal_sensors_different()
{
    int intOuter, intInner;
    // Detect if any two sensors have difference in values great enough
    // to indicate a discernable signal
    for (intOuter=0; intOuter < NUM_GOAL_SENSORS-1; intOuter++)
      {
        for (intInner=intOuter+1; intInner < NUM_GOAL_SENSORS; intInner++)
          {
            if (abs(goal_sensor[intOuter]-goal_sensor[intInner]) >
                GOAL_SENSOR_DIFF_THRESHOLD)
              {
                return TRUE;
            } // if (abs(goal_sensor[intOuter]-goal_sensor[intInner]) >
            //     GOAL_SENSOR_DIFF_THRESHOLD)
        } // for (int intInner=intOuter+1; intInner < NUM_GOAL_SENSORS; intInner++)
    } // for (int intOuter=0; intOuter < NUM_GOAL_SENSORS; intOuter++)
    return FALSE;
} // int goal_sensors_different()

// Set releaser if conditions are met
void goal_rel()
{
    // Detect if sensors are above thresholds
    goal_thresh = goal_sensors_above_threshold();
    goal_diff = goal_sensors_different();
```

```
    if (!goal_thresh)
      {
        goal_rel_present = FALSE;
        return;
    } // if (!goal_sensors_above_threshold())
    // Detect if sensors are different enough to signal light is not ambient
    else if (!goal_diff)
        {
          goal_rel_present = FALSE;
      } // else if (!goal_sensors_different())

      else goal_rel_present = TRUE;
} // int goal_rel()

int forward = FALSE;

// generate direction information
void goal_perceive()
{
    int temp_index, intCount;
    float temp_direction;

    // evaluate direction without changing global
    temp_direction = 0.;
    // determine direction by calculating maximum sensor value
    for (intCount = 1; intCount < NUM_GOAL_SENSORS; intCount++)
      {
        if (goal_sensor[intCount] < goal_sensor[(int)temp_direction])
          {
            temp_direction = (float)intCount;
        } // if (goal_sensor[intCount] > goal_sensor[goal_direction])
    } // for (int intCount = 1; intCount < NUM_GOAL_SENSORS; intCount++)

    // this code added to deal with a problem;
    // bot wouldn't turn hard enough when light was slightly to one side
    if (temp_direction > FORWARD) temp_direction = ANGLE_RIGHT;
    else temp_direction = ANGLE_LEFT;

    // this code added to cut down on wobble as bot approached target
    if (abs(goal_sensor[1]-goal_sensor[0]) < FORWARD_THRESHOLD &&
        (goal_sensor[1]+goal_sensor[2]) < 30)
      {
        temp_direction = FORWARD;
    } // if (abs(goal_sensor[1]-goal_sensor[0]) < FORWARD_THRESHOLD &&
    //     (goal_sensor[1]+goal_sensor[2]) < 30)

    // set target
    goal_direction = temp_direction;

    // set global guide_direction variable to help with slippage recovery
    // intelligence
    if (goal_direction > FORWARD) guide_direction = RIGHT;
    else guide_direction = LEFT;
} // int goal_perceive()

// goal-seeking thread
void goal()
{
    while(1)
      {
        goal_sense();
        goal_rel();
        goal_perceive();
    } // while (1)
} // void goal()

// SLIP:
//  Behavior that compensates for being lodged against an obstacle
//  Releaser: Encoder difference is not high enough
//  Action: Spin randomly
```

```
// macro for port on which encoder lives
#define ENCODER_PORT 1

// macros for thresholds to determine slippage recovery actions
#define ENCODER_THRESHOLD 5
#define SLIP_COUNT_THRESHOLD 2

// 'stuck' is perception, really
int stuck;
float slip_direction;

// variables to determine type of slippage recovery action
int slip_attempt = 0;
int slip_counter = 0;

void slip_rel()
{
    int encoder_count;

    // determine if encoder count is too low
    encoder_count = read_encoder(ENCODER_PORT);
    if (encoder_count < ENCODER_THRESHOLD)
      {
        slip_rel_present = TRUE;
    } // if (encoder_count < ENCODER_THRESHOLD)
    else
      {
        slip_rel_present = FALSE;
        // keep track of conditions controlling slippage recovery behavior
        slip_attempt = 0;
        slip_counter++;
    } // else
    reset_encoder(ENCODER_PORT);

    // Give status of light sensors, encoder, behavior code, and direction
    printf("%d,%d,%d,%d; %d; %s; %f\n",
           goal_sensor[0],
           goal_sensor[1],
           goal_sensor[2],
           goal_sensor[3],
           encoder_count,
           b,
           motor_direction);
} // int slip_rel()

// if we're not engaged in spinning, then we're obviously stuck
void slip_perceive()
{
    if (!spinning && slip_rel_present)
      {
        stuck = TRUE;
    } // if (!spinning)
    else stuck = FALSE;
} // int slip_perceive()

void slip_act()
{
    // if we perceived that we're stuck,
    if (stuck)
      {
        // to reduce random movement; forgot to change this back
        // for demo; could have reduced some headaches
        guide_direction = 0;
        // based on guide_direction and slip_counter, behave differently
        if (guide_direction == LEFT)
          {
           if (slip_counter < SLIP_COUNT_THRESHOLD)
             {
               // if slip has failed recently, spin backwards
               slip_direction = SPIN_LEFT_1;
               sleep(SLEEP_TIME*10.);
```

```
            } // if (slip_counter < SLIP_COUNT_THRESHOLD)
            else
              {
                // if slip has not failed enough recently, spin in place
                slip_direction = SPIN_LEFT;
                sleep(SLEEP_TIME*6.);
            } // else
        } // if (guide_direction == LEFT)
        else
          {
            if (slip_counter < SLIP_COUNT_THRESHOLD)
              {
                slip_direction = SPIN_RIGHT_1;
                sleep(SLEEP_TIME*10.);
            } // if (slip_counter < SLIP_COUNT_THRESHOLD)
            else
              {
                slip_direction = SPIN_RIGHT;
                sleep(SLEEP_TIME*6.);
            } // else
        } // else
        slip_direction = FORWARD;
        sleep(SLEEP_TIME/3.);
        slip_counter = 0;

        defer();
    } // if (!slip_rel_present)
} // void slip_act()

// slip thread
void slip()
{
    stuck = 0;
    enable_encoder(ENCODER_PORT);
    reset_encoder(ENCODER_PORT);
    while(1)
      {
        slip_rel();
        slip_perceive();
        slip_act();
        sleep(SLEEPTHREAD_TIME*5.);  //ML 12:20,3/27
        defer();
    } // while (1)
} // void slip()

// HIGH:
//  If there's a high object, avoid it in the strongest possible terms

// macro for loops
#define NUM_HIGH_SENSORS 2

// macros for ports
#define HIGH_SENSOR_LEFT 16
#define HIGH_SENSOR_RIGHT 17

// macros for sensor thresholds
#define HIGH_SENSOR_THRESHOLD 60
#define HIGH_SENSOR_DIFF_THRESHOLD 1
#define HIGH_SENSOR_DIFF_THRESHOLD_2 5

// globals for sensors and motor command
float high_direction;
int high_sensor[NUM_HIGH_SENSORS];

void high_sense()
{
    // gather sensory info
    high_sensor[LEFT] = analog(HIGH_SENSOR_LEFT);
    high_sensor[RIGHT] = analog(HIGH_SENSOR_RIGHT);
} // void high_sense()
```

```
void high_rel()
{
    // Set the releaser if sensors are active and different
    if (high_sensor[LEFT] < HIGH_SENSOR_THRESHOLD &&
        high_sensor[RIGHT] < HIGH_SENSOR_THRESHOLD)
      {
        high_rel_present = FALSE;
    } // if (high_sensor[0] > HIGH_SENSOR_THRESHOLD || high_sensor[1] >
HIGH_SENSOR_THRESHOLD)
    else if (abs(high_sensor[LEFT] - high_sensor[RIGHT]) < HIGH_SENSOR_DIFF_THRESHOLD)
        {
          high_rel_present = FALSE;
      } // else if (abs(high_sensor[0] - high_sensor[1]) > HIGH_SENSOR_DIFF_THRESHOLD)
      else high_rel_present = TRUE;
} // int high_rel()

// return a direction to travel given the releaser and sensed information
void high_perceive()
{
    if ((!spinning) && high_rel_present)
      {
        if (abs(high_sensor[LEFT] - high_sensor[RIGHT]) < HIGH_SENSOR_DIFF_THRESHOLD_2)
          {
            high_direction = -1.5 + (float)guide_direction*0.5;
        } // else if (abs(high_sensor[0] - high_sensor[1]) > HIGH_SENSOR_DIFF_THRESHOLD)
if ((!spinning) && high_rel_present) sleep(SLEEPTHREAD_TIME);
        if (high_sensor[LEFT] > high_sensor[RIGHT])
          {
            high_direction = SPIN_RIGHT_1;//ANGLE_RIGHT;
        } // if (high_sensor[LEFT] > high_sensor[RIGHT])
        else high_direction = SPIN_LEFT_1;//ANGLE_LEFT;
    } // if ((!spinning) && high_rel_present)
} // int high_perceive()

// high object avoidance thread
void high()
{
    while (1)
      {
        high_sense();
        high_rel();
        high_perceive();
        defer();
    } // while (1)
} // void high()

float motor_direction;

// arbitrate should determine motor_direction based on highest active releaser
void arbitrate()
{
    goal_on = FALSE;
    if (high_rel_present)
      {
        b[0] = 'H';
        motor_direction = high_direction;
    } // if (high_rel_present)
    else if (stuck)
        {
          b[0] = 'S';
          motor_direction = slip_direction;
      } // else if (stuck)
      else if (goal_rel_present)
          {
            b[0] = 'G';
            beep();
            motor_direction = goal_direction;
            goal_on = TRUE;
        } // else if (goal_rel_present)
        else if (low_rel_present)
            {
```

```
              b[0] = 'L';
              motor_direction = low_direction;
          } // else if (low_rel_present)
          else
            {
              b[0] = 'C';
              motor_direction = cruise_direction;
          } // else
} // void arbitrate()

// macros for motor constants
#define SPIN_AHEAD 60
#define FULL_AHEAD 60
#define FULL_BACK -60
#define MOSTLY_AHEAD 40
#define HALF_AHEAD 20
#define QUARTER_AHEAD 0

// macros for infrared sensor that was not fully implemented
#define IR_PORT 20
#define IR_THRESHOLD 220

// control motors according to desire of highest-level module
// if goal_seeking, stop motors for 1/3 of the time
void motor_control()
{
    int forward_value=5;
    if (goal_on)
      {
        goal_toggle++;
        if ((goal_toggle%6)==0 && (goal_toggle%6)==1)
          {
            motor(LEFT, 0);
            motor(RIGHT, 0);
            return;
        } // if ((goal_toggle%6)==0 && (goal_toggle%6)==1)
    } // if (goal_on)

    if (motor_direction == POWER_FORWARD)
      {
        spinning = FALSE;
        motor(LEFT, 100);
        motor(RIGHT, 100);
    } // if (motor_direction == POWER_FORWARD)
    if (motor_direction == BACK)
      {
        spinning = FALSE;
        motor(LEFT, FULL_BACK);
        motor(RIGHT, FULL_BACK);
    } // if (motor_direction == BACK)
    if (motor_direction == SPIN_RIGHT)
      {
        spinning = TRUE;
        motor(LEFT, SPIN_AHEAD);
        motor(RIGHT, FULL_BACK);
    } // if (motor_direction == SPIN)
    if (motor_direction == SPIN_LEFT)
      {
        spinning = TRUE;
        motor(LEFT, FULL_BACK);
        motor(RIGHT, SPIN_AHEAD);
    } // if (motor_direction == SPIN_LEFT)
    if (motor_direction == SPIN_RIGHT_1)
      {
        spinning = TRUE;
        motor(LEFT, forward_value);
        motor(RIGHT, -80);
    } // if (motor_direction == SPIN)
    if (motor_direction == SPIN_LEFT_1)
      {
        spinning = TRUE;
```

```
        motor(LEFT, -80);
        motor(RIGHT, forward_value);
    } // if (motor_direction == SPIN_LEFT)
    if (motor_direction == ANGLE_LEFT)
      {
        spinning = FALSE;
        motor(LEFT, QUARTER_AHEAD);
        motor(RIGHT, FULL_AHEAD);
    } // if (motor_direction == ANGLE_LEFT)
    if (motor_direction == ANGLE_FRONT_LEFT)
      {
        spinning = FALSE;
        motor(LEFT, HALF_AHEAD);
        motor(RIGHT, FULL_AHEAD);
    } // if (motor_direction == ANGLE_FRONT_LEFT)
    if (motor_direction == FRONT_LEFT)
      {
        spinning = FALSE;
        motor(LEFT, MOSTLY_AHEAD);
        motor(RIGHT, FULL_AHEAD);
    } // if (motor_direction == FRONT_LEFT)
    if (motor_direction == FORWARD)
      {
        spinning = FALSE;
        motor(LEFT, FULL_AHEAD);
        motor(RIGHT, FULL_AHEAD);
    } // if (motor_direction == FORWARD)
    if (motor_direction == FRONT_RIGHT)
      {
        spinning = FALSE;
        motor(LEFT, FULL_AHEAD);
        motor(RIGHT, MOSTLY_AHEAD);
    } // if (motor_direction == FRONT_RIGHT)
    if (motor_direction == ANGLE_FRONT_RIGHT)
      {
        spinning = FALSE;
        motor(LEFT, FULL_AHEAD);
        motor(RIGHT, HALF_AHEAD);
    } // if (motor_direction == ANGLE_FRONT_RIGHT)
    if (motor_direction == ANGLE_RIGHT)
      {
        spinning = FALSE;
        motor(LEFT, FULL_AHEAD);
        motor(RIGHT, QUARTER_AHEAD);
    } // if (motor_direction == ANGLE_RIGHT)
} // void motor_control()

// M A I N ///////////////////////////////////////////////
void main()
{
    b[0] = '?';
    b[1] = '\0';
    guide_direction = 0;
    printf("Start me!\n");
    while (!start_button());
    printf("Started.\n");
    cruise();
    start_process(goal());
    start_process(slip());
    start_process(high());
    sleep(SLEEPTHREAD_TIME * 8.);  //ML 12:20,3/27
    while (1)
      {
        arbitrate();
        motor_control();
        // sleep(SLEEP_TIME);
    } // while (1)
} // void main()
```