```
#include "motordriver_4wd.h"
#include "structs.h"
#include <seeed_pwm.h>
#include <StackArray.h>

//#define DEBUG 1
//#define DEBUG_PING_CHECK 1
//#define DEBUG_PATH 1
//#define DELAY_START 1

enum {FORWARD, BACKWARD, TURN_LEFT, TURN_RIGHT, STOPPED}
moveState;
enum {WANDER, WANDEROUT, GOHOME, INCLINE, DONE} machineState;

//Calibration function prototypes
void inline checkTicks();
void inline defaultPan();
void inline tickStopper();
void inline initialTest();
void inline lightInit();
void inline colorLoop();
void inline farthest();

void setup()
{
  //Individually initialize our components
  servoInit();
  moveInit();
  buttonInit();
  lightInit();
  diodeInit();
  worldNavInit();

  initialTest();

  clavInit();

  #ifdef DELAY_START
  lightCycle(250);
  #endif

  //Initialize the serial communications:
  Serial.begin(9600);
```

```
}

void loop()
{
  //forward(20);
  clavBrain();
  //farthest();
  //Serial.println( getDiodeVal() );
}
```

```
//================================================================
======================
// Theta Management
//================================================================
======================

#define THETA_INCREMENTS 4
enum IncrementalTheta {ZERO_PI, PI_O2, ONE_PI, THREE_PI_O2};

IncrementalTheta radianLeft(IncrementalTheta rad){
  switch(rad){
    default:
    case ZERO_PI: return PI_O2;
      break;
    case PI_O2: return ONE_PI;
      break;
    case ONE_PI: return THREE_PI_O2;
      break;
    case THREE_PI_O2: return ZERO_PI;
      break;
  }
}

IncrementalTheta radianRight(IncrementalTheta rad){
  switch(rad){
    default:
    case ZERO_PI: return THREE_PI_O2;
      break;
    case PI_O2: return ZERO_PI;
      break;
    case ONE_PI: return PI_O2;
      break;
    case THREE_PI_O2: return ONE_PI;
      break;
  }
}

double radianToValue( IncrementalTheta rad ){
  switch(rad){
    default:
    case ZERO_PI: return 0;
      break;
```

```
    case PI_O2: return PI / 2;
      break;
    case ONE_PI: return PI;
      break;
    case THREE_PI_O2: return (3 * PI) / 2;
      break;
  }
}

typedef struct {
  int col; //The column of this sector
  int row; //The row of this sector
  int inHeading; //The heading when we entered this sector
  int outAction; //The action we took to leave this sector
} SectorPath;
```

```
//The subsytems - these were originally in separate files, but
have been compiled here for convenience
// Nicolas Fredrickson

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
// Servo Functions and Constants
// Specific to my configuration and servo (HS-422)
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~

#define SERVO_PULSE_MIN   0.4 * 1000 // This should be your 0
degrees
#define SERVO_PULSE_MAX   2.4 * 1000 // This should be your 180
degrees
#define SERVO_PERIOD     25 * 1000  // Total pulse length,
mis-setting can result in erratic behavior!

#define START_ANGLE   0
#define END_ANGLE     180

#define servoPin 13

IncrementalTheta panDir;

void inline defaultPan();

//Initializer Function
void servoInit(){
  pinMode(servoPin, OUTPUT);
}

/*
  --------------- Servo Control Functions ---------------
*/

//Executes a PWM order according to the provided pulse length
void pulsePanTo(float pulseLength) {
  for (int i = 0; i < 200; i++) {
    digitalWrite(servoPin, HIGH);
    delayMicroseconds(pulseLength);
    digitalWrite(servoPin, LOW);
    delayMicroseconds(SERVO_PERIOD - pulseLength);
```

```
  }
}

void radianPanTo( int rad ){
  switch(rad){
    case ZERO_PI:
      panDir = ZERO_PI; anglePanTo(0);
      break;
    default:
    case PI_O2:
      panDir = PI_O2; anglePanTo(90);
      break;
    case ONE_PI:
      panDir = ONE_PI; anglePanTo(180);
      break;
  }
}

//Converts a target angle into a target pulse length, then pans
//Essentially, maps the angle from the angle range (START_ANGLE
to END_ANGLE)
//to the pulse range (SERVO_PULSE_MIN to SERVO_PULSE_MAX)
void anglePanTo(int angle) {
  float newVal = angle - START_ANGLE;
  newVal = newVal * (SERVO_PULSE_MAX - SERVO_PULSE_MIN);
  newVal = newVal / (END_ANGLE - START_ANGLE);
  newVal = newVal + SERVO_PULSE_MIN;

  pulsePanTo(newVal);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
// Grove Ultrasonic Functions and Constants
// Specific to my configuration
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
#define pingPin A0

#define RANGER_LEFT     ZERO_PI
#define RANGER_FORWARD  PI_O2
#define RANGER_RIGHT    ONE_PI
```

```
/*
  ----------Ultrasonic Control Functions ----------------
*/
int pingCM(){
  long duration;

  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW);
  delayMicroseconds(2);
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(5);
  digitalWrite(pingPin, LOW);

  pinMode(pingPin, INPUT);
  duration = pulseIn(pingPin, HIGH);
  delay(100);

  #ifdef DEBUG
  Serial.print("Distance Read: ");
  Serial.print(duration);
  Serial.print(" ---> ");
  Serial.print( (duration / 29) / 2 );
  Serial.println(" cm");
  #endif
  return (duration / 29) / 2;
}

int pingMM(){
  return pingCM() * 10;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
// Motor Functions and Constants
// Specific to my wheel configuration
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~

#define TICK_PIN_LEFT 1
#define TICK_PIN_RIGHT 0

#define TICK_PER_ROT    72.0
```

```
#define WHEEL_RADIUS    42.5
#define WHEEL_DISTANCE  158.0

#define DISTANCE_PER_TICK   (PI * WHEEL_RADIUS * 2) /
TICK_PER_ROT

//The extra speed for our left-side wheels
//If veering left, increase
//If veering right, decrease
#define DIFFERENTIAL_PERCENTAGE  0.0505

//Tick Counts (For Odometry)
volatile int tickCountLeft;
volatile int tickCountRight;

volatile int leftDirection;
volatile int rightDirection;

//Initializer Function
void moveInit(){
  MOTOR.init(); //Initll pin

  pinMode(TICK_PIN_LEFT, INPUT);
  pinMode(TICK_PIN_RIGHT, INPUT);

  tickCountLeft = 0;
  tickCountRight = 0;

  attachInterrupt(TICK_PIN_LEFT, tickLeft, CHANGE);
  attachInterrupt(TICK_PIN_RIGHT, tickRight, CHANGE);
}

/*
  -------------- Tick Counter Functions ----------------
*/
void updateDirection(){
    switch(moveState){
      default:
      case FORWARD:
        leftDirection = 1;
        rightDirection = 1;
        break;
      case TURN_RIGHT:
```

```
        leftDirection = 1;
        rightDirection = -1;
        break;
      case TURN_LEFT:
        leftDirection = -1;
        rightDirection = 1;
        break;
      case BACKWARD:
        leftDirection = -1;
        rightDirection = -1;
        break;
    }
}

void tickLeft(){
  tickCountLeft += leftDirection;
}

void tickRight(){
  tickCountRight += rightDirection;
}
//.375 cm
/*
  -------------- Motor Control Functions ----------------
*/
inline void turnInPlaceLeft(int speeed)
{
  //set right
  MOTOR.setSpeedDir1(speeed, DIRF);
  //set left
  MOTOR.setSpeedDir2(speeed, DIRF);
}

inline void turnInPlaceRight(int speeed)
{
  //set right
  MOTOR.setSpeedDir1(speeed, DIRR);
  //set left
  MOTOR.setSpeedDir2(speeed, DIRR);
}

inline void backward(int speeed)
{
```

```
  //set right
  MOTOR.setSpeedDir1(speeed, DIRR);
  //set left
  MOTOR.setSpeedDir2(speeed + DIFFERENTIAL_PERCENTAGE * speeed,
DIRF);
}

inline void forward(int speeed)
{
  //set right
  MOTOR.setSpeedDir1(speeed, DIRF);
  //set left
  MOTOR.setSpeedDir2(speeed + DIFFERENTIAL_PERCENTAGE * speeed,
DIRR);
}

inline void stopMotion()
{
  //Stop right
  MOTOR.setStop1();
  //Stop left
  MOTOR.setStop2();
}

inline void stopLeft()
{
  //Stop left
  MOTOR.setStop2();
}

inline void stopRight()
{
  //Stop right
  MOTOR.setStop1();
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
// Button Functions and Constants
// Specific to my configuration
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
```

```
#define buttonPin 11

//Initializer Function
void buttonInit(){
  pinMode(buttonPin, INPUT);
}

/*
  -------------- Button Functions ----------------
*/

inline void checkButtonState(){
  if( digitalRead(buttonPin) )
    moveState = STOPPED;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
// RGB Light Functions and Constants
// Specific to my configuration
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~

#include<ChainableLED.h>

#define NUM_LEDS    1

#define PIN_CLOCK   SCL
#define PIN_DATA    SDA

#define DEFAULT_SATURATION  1.0
#define DEFAULT_BRIGHTNESS  0.15

#define COLOR_HUE_ONE   0.0
#define COLOR_HUE_TWO   0.25
#define COLOR_HUE_THREE 0.5
#define COLOR_HUE_FOUR  0.75

//defines the pin used on arduino.
ChainableLED light(PIN_CLOCK, PIN_DATA, NUM_LEDS);

void inline lightInit(){
  light.init();
```

```
}

/*
  -------------- Light Functions ----------------
*/
inline void radianToColor(float radianAngle){
  light.setColorHSB(0, fmod(radianAngle, TWO_PI) / TWO_PI,
DEFAULT_SATURATION, DEFAULT_BRIGHTNESS);
}

inline void degreeToColor(int degree){
  light.setColorHSB(0, (degree % 360) / 360, DEFAULT_SATURATION,
DEFAULT_BRIGHTNESS);
}

inline void radianToSat(float radianAngle, double color){
  light.setColorHSB(0, color, fmod(radianAngle, TWO_PI) / TWO_PI
,DEFAULT_BRIGHTNESS );
}

inline void radianToBright(float radianAngle, double color){
  light.setColorHSB(0, color, DEFAULT_SATURATION,
fmod(radianAngle, TWO_PI) / TWO_PI );
}

inline void specificColor(double color){
  light.setColorHSB(0, color, DEFAULT_SATURATION,
DEFAULT_BRIGHTNESS );
}

inline void rgbColor(int red, int green, int blue){
  light.setColorRGB(0, red, green, blue);
}

inline void lightOff(){
  light.setColorHSB(0, 0, 0, 0);
}

inline void lightCycle(int delayTime){
  for (int i = 0; i < 10; i++)
    {
      rgbColor( 100, 0, 0 );  // red
      delay(delayTime);
```

```
      rgbColor( 0, 100, 0 );  // green
      delay(delayTime);
      rgbColor( 0, 0, 100 );  // blue
      delay(delayTime);
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
// Diode (Light Detector) Functions and Constants
// Specific to my configuration
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~

#define PIN_OUTPUT A2

enum {LEFT, RIGHT} lightState;

void diodeInit(){
  digitalWrite( PIN_OUTPUT, INPUT_PULLUP);
}

int getDiodeVal(){
  return analogRead( PIN_OUTPUT );
}
```

```
// Calibration Functions
// For ensuring that everything is working just right
// Nicolas Fredrickson

void inline initialTest(){
  defaultPan();
  basicColorCycle();
}

/*
  -------------- Calibration Functions ----------------
*/

void inline checkTicks(){
 Serial.print(tickCountLeft);
  Serial.print(" : ");
 Serial.println(tickCountRight);
}

void inline tickStopper(){
    if(tickCountLeft > TICK_PER_ROT )
      stopLeft();
    if(tickCountRight > TICK_PER_ROT * 2)
      stopRight();
}

//Tests the servo and the ultrasonic ranger
//Pans from 0 to 180 and tests each angle, then aims at the
longest distance.
void inline farthest()
{
  int bestAngle = -1;
  int bestPing = -1;

  int ping = 0;

  for (int i = 0; i <= END_ANGLE; i += 10) {
    //Move to angle
    anglePanTo(i);
    //Ping
    ping = pingCM();
    delay(1000);
```

```
    //If ping is greater than bestPing
    if (ping > bestPing) {
      //Set the values appropriately
      bestAngle = i;
      bestPing = ping;
    }
  }

 anglePanTo(bestAngle);
}

//Pans to the middle, then the end, then the beginning
void inline defaultPan() {

  Serial.println("To Default Position");

  anglePanTo( (START_ANGLE + END_ANGLE) / 2 );
  delay(500);

 anglePanTo(END_ANGLE);
  delay(500);

 anglePanTo(START_ANGLE);
  delay(500);
}

void inline colorLoop(){

  float hue = 0;
  boolean up = true;

  for(int i = 0; i < 80; i++){
    light.setColorHSB(0, hue, 1.0, 0.5);
    Serial.println(hue);
    delay(100);

    if (up)
      hue+= 0.025;
    else
      hue-= 0.025;

    if (hue>=1.0 && up)
      up = false;
```

```
    else if (hue<=0.0 && !up)
      up = true;
  }
}

void basicColorCycle(){
  light.setColorRGB(0, 128, 0, 0);
  delay(100);
  light.setColorRGB(0, 128, 128, 0);
  delay(100);
  light.setColorRGB(0, 0, 128, 0);
  delay(100);
  light.setColorRGB(0, 0, 128, 128);
  delay(100);
  light.setColorRGB(0, 0, 0, 128);
  delay(100);
  light.setColorRGB(0, 128, 0, 128);
  delay(100);
  light.setColorRGB(0, 128, 128, 128);
  delay(100);
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
// Worldgrid Constants
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~

#define DEFAULT_CERTAINTY 0.0

#define PASSED_UNSURE   '?' // Haven't crossed it, wouldn't know
#define PASSED_BLOCKED  '#' // Obstacle, space is blocked
#define PASSED_OPENED   '*' // Non-obstacle, space is clear
#define PASSED_MISREAD  'E' // YOU FOOL, WHAT HAVE YOU DONE!?!?!?

#define ROW_COUNT     5
#define COLUMN_COUNT  5

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
// World Navigation Constants
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~

#define SECTOR_WIDTH_CM  30
#define SECTOR_WIDTH_MM  300.0

#define LEFT_BOUNDRY    0.0
#define RIGHT_BOUNDRY   COLUMN_COUNT * SECTOR_WIDTH_MM
#define BOTTOM_BOUNDRY  0.0
#define TOP_BOUNDRY     ROW_COUNT * SECTOR_WIDTH_MM

#define EXCESS 192//187.5

//================================================================
======================
// Variables!
//================================================================
======================

//Tracks our statistical certainty about the existence of an
obstacle
//Used with our distance ranger
floatstatGrid[ROW_COUNT][COLUMN_COUNT];
```

```
//Records our absolute certainty about the existence of an
obstacle
//Used when we pass through a sector.
charpassGrid[ROW_COUNT][COLUMN_COUNT];

IncrementalTheta thetaState;

//Odometry Variables
double dTheta;
double dX;
double dY;

double x;
double y;
double theta;

double xNoSkew;
double yNoSkew;

//================================================================
=======================
// Initialization!
//================================================================
=======================
inline void worldNavInit(){
  for (int i = 0; i < ROW_COUNT; i++) {
    for (int j = 0; j < COLUMN_COUNT; j++) {
      statGrid[i][j] = DEFAULT_CERTAINTY;
      passGrid[i][j] = PASSED_UNSURE;
    }
  }
}

inline void addShift( IncrementalTheta inThet ){
  xNoSkew = x;
  yNoSkew = y;

  switch(inThet){
    default:
    case ZERO_PI:
      //Undo the slight add, the do the excess add
      x = x + (EXCESS / 2) + 15;
      break;
```

```
    case PI_O2:
      y = y + (EXCESS / 2) - 5;
      break;
    case ONE_PI:
      x = x + (EXCESS / 4);
      break;
    case THREE_PI_O2:
      y = y + (EXCESS / 4);
      break;
  }
}

inline void removeShift( IncrementalTheta inThet ){
  switch(inThet){
    default:
    case ONE_PI:
    case ZERO_PI:
      x = xNoSkew;
      break;
    case PI_O2:
    case THREE_PI_O2:
      y = yNoSkew;
      break;
  }
}

//================================================================
=======================
// Grid Coordinate Resolution & Management - for both grids
//================================================================
=======================

// Think of the sectors as a number line
// Out input is, essentially, a point on the number line
inline int valueToSectorCM( double cmVal ){
  return int( cmVal / SECTOR_WIDTH_CM );
}

inline int valueToSectorMM( double mmVal ){
  return int( mmVal / SECTOR_WIDTH_MM );
}

boolean inSectors(int col, int row){
```

```
  return (col >= 0 && row >= 0) && (col < COLUMN_COUNT && row <
ROW_COUNT);
}

//================================================================
======================
// Statistics Grid Management
//================================================================
======================

double getSectorStat( int col, int row ){
  if( inSectors(col, row) )
    return statGrid[ row ][ col ];
  else
    return -1.0;
}

void setSectorStat( int col, int row, double value ){
  if( inSectors(col, row) )
    statGrid[ row ][ col ] = value;
}

void incrementSectorStat( int col, int row, double value ){
  if( inSectors(col, row) )
    statGrid[ row ][ col ] += value;
}

void printStatGrid(){
  for(int i = ROW_COUNT - 1; i >= 0; i--){
    for(int j = 0; j < COLUMN_COUNT; j++){
     Serial.print(statGrid[i][j]); Serial.print(' ');
    }
    Serial.println();
  }
}

//================================================================
======================
// Passed Grid Management
//================================================================
======================

inline char getSectorPass( int col, int row ){
```

```
  if( inSectors(col, row) )
    return passGrid[ row ][ col ];
  else
    return PASSED_MISREAD;
}

inline void setSectorPass( int col, int row, char symbol ){
  if( inSectors(col, row) )
    passGrid[ row ][ col ] = symbol;
}

inline boolean getSectorBlock( int col, int row ){
  if( inSectors(col, row) )
    return passGrid[ row ][ col ] == PASSED_BLOCKED;
  else
    return false;
}

inline void markCurrentPass(){
  setSectorPass( valueToSectorMM( x ), valueToSectorMM( y ),
PASSED_OPENED);
}

inline void markProjectedBlock(){
  setSectorPass( valueToSectorMM( x ) + projectColTheta(),
valueToSectorMM( y ) + projectRowTheta(), PASSED_BLOCKED);
}

inline void printPassGrid(){
  for(int i = ROW_COUNT - 1; i >= 0; i--){
    for(int j = 0; j < COLUMN_COUNT; j++){
     Serial.print(passGrid[i][j]); Serial.print(' ');
    }
    Serial.println();
  }
}

//================================================================
======================
// Theta Management
//================================================================
======================
void incrementTheta(){
```

```
  thetaState = radianLeft(thetaState);
  theta = radianToValue(thetaState);
}

void decrementTheta(){
  thetaState = radianRight(thetaState);
  theta = radianToValue(thetaState);
}

IncrementalTheta getPanTheta(){
  if(panDir == RANGER_LEFT)
    return radianLeft(thetaState);
  else if(panDir == RANGER_RIGHT)
    return radianRight(thetaState);
  else
    return thetaState;
}

double evalPanTheta(){
  return radianToValue( getPanTheta() );
}

//Find our effective theta given our current direction
IncrementalThetamoveEffectTheta(){
  return customEffectTheta(moveState);
}

//Find our effective theta given our current direction
IncrementalTheta customEffectTheta(int inMove){
  if(inMove == BACKWARD)
    return (thetaState + 2) % 4;
  else
    return thetaState;
}

//=================================================================
=======================
// Projections - for finding out what's in front of us.
//=================================================================
=======================

int projectCol(IncrementalTheta inputTheta){
  switch( inputTheta ){
```

```
    default:
    case ZERO_PI: return 1;
    case PI_O2: return 0;
    case ONE_PI: return -1;
    case THREE_PI_O2: return 0;
  }
  return 0;
}

int projectRow(IncrementalTheta inputTheta){
  switch( inputTheta ){
    default:
    case ZERO_PI: return 0;
    case PI_O2: return 1;
    case ONE_PI: return 0;
    case THREE_PI_O2: return -1;
  }
  return 0;
}

int projectColTheta(){
  return projectCol( thetaState );
}

int projectRowTheta(){
  return projectRow( thetaState );
}

int projectColPan(){
  return projectCol( getPanTheta() );
}

int projectRowPan(){
  return projectRow( getPanTheta() );
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Pathfinding Constants
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//These tell us the action we had to take in order to leave the
sector
#define ACTION_FORWARD       0
#define ACTION_REVERSE       1
#define ACTION_STOP          2
#define ACTION_LEFT_FORWARD  3
#define ACTION_RIGHT_FORWARD 4
#define ACTION_NONE          5

//These allow us to use the pass grid in our pathing algorithm
#define PASSED_UNSURE_PLAN '!' // We haven't crossed it, we
don't know what's there, but our pathing algorithm went over it.
#define PASSED_OPENED_PLAN '+' // We know the way is open, and
our pathing algorithm went over it

#define TARGET_IN_PUSH  160

//----- Path Finder specific globals
SectorPath subTarget;

int pathTargetCol, pathTargetRow; // Where are we going?
boolean boxForgive; // are we willing to venture into a space
that may be a box?

inline void worldPassReset(){
  for (int i = 0; i < ROW_COUNT; i++)
    for (int j = 0; j < COLUMN_COUNT; j++){
      if( passGrid[i][j] == PASSED_UNSURE_PLAN )
        passGrid[i][j] = PASSED_UNSURE;
      else if( passGrid[i][j] == PASSED_OPENED_PLAN)
        passGrid[i][j] = PASSED_OPENED;
    }
}

//=============================================================
======================
// Target Management
//=============================================================
```

```
=======================

inline int getStartAction() {
  return subTarget.outAction;
}

inline boolean posInSubTarget() {
  return posInTarget( subTarget.col, subTarget.row );
}

inline boolean posInTarget(int targCol, int targRow) {
  return posInTargetCol( targCol ) && posInTargetRow( targRow );
}

boolean posInTargetCol(int targCol) {
  switch ( moveEffectTheta() ) {
    case ZERO_PI:
      return x > (targCol * SECTOR_WIDTH_MM) + TARGET_IN_PUSH;
      break;
    case ONE_PI:
      return x < (targCol * SECTOR_WIDTH_MM) + (SECTOR_WIDTH_MM
- TARGET_IN_PUSH);
      break;
    case PI_O2:
    case THREE_PI_O2: return valueToSectorMM(x) == targCol;
      break;
  }
}

boolean posInTargetRow(int targRow) {
  switch ( moveEffectTheta() ) {
    case PI_O2: return y > (targRow * SECTOR_WIDTH_MM) +
TARGET_IN_PUSH;
      break;
    case THREE_PI_O2: return y < (targRow * SECTOR_WIDTH_MM) +
(SECTOR_WIDTH_MM - TARGET_IN_PUSH);
      break;
    case ZERO_PI:
    case ONE_PI: return valueToSectorMM(y) == targRow;
      break;
  }
}
```

```
void projectionCheck() {
  int col;
  int row;

  col = valueToSectorMM( x ) + projectCol( moveEffectTheta() );
  row = valueToSectorMM( y ) + projectRow( moveEffectTheta() );

  if ( !inSectors(col, row) )
    return false;
  else if ( getSectorStat( col, row ) > 0.5 ) {
    subTarget.col = valueToSectorMM( x );
    subTarget.row = valueToSectorMM( y );
  }
}

void blockCollision() {
 markProjectedBlock();

  subTarget.col = valueToSectorMM( x );
  subTarget.row = valueToSectorMM( y );
}

//=============================================================
======================
// Boundry Checking! Are we in bounds or out of bounds?
//=============================================================
======================
boolean outOfBoundry() {
  switch ( moveEffectTheta() ) {
    case ZERO_PI: return x > RIGHT_BOUNDRY;
      break;
    case PI_O2: return y > TOP_BOUNDRY;
      break;
    case ONE_PI: return x < LEFT_BOUNDRY;
      break;
    case THREE_PI_O2: return y < BOTTOM_BOUNDRY;
  }
}

boolean inBoundry() {
  switch ( moveEffectTheta() ) {
    case ZERO_PI: return x < RIGHT_BOUNDRY - TARGET_IN_PUSH;
      break;
```

```
    case PI_O2: return y < TOP_BOUNDRY - TARGET_IN_PUSH;
      break;
    case ONE_PI: return x > LEFT_BOUNDRY + TARGET_IN_PUSH;
      break;
    case THREE_PI_O2: return y > BOTTOM_BOUNDRY + TARGET_IN_PUSH;
  }
}

//================================================================
======================
// Path Finder!
//================================================================
======================
boolean pathFindFromCurrent(int targCol, int targRow) {

  pathTargetCol = targCol;
  pathTargetRow = targRow;

  boxForgive = false;

  //(SectorPath){ nextPoint.col, nextPoint.row, firstPoint.
inHeading, firstPoint.outAction };
  subTarget = (SectorPath) {
    valueToSectorMM( x ), valueToSectorMM( y ), thetaState,
ACTION_NONE
  };

  boolean success = pathFindRecursive( &subTarget );
  worldPassReset();

  if (!success) {
    boxForgive = true;
    subTarget = (SectorPath) {
      valueToSectorMM( x ), valueToSectorMM( y ), thetaState,
ACTION_NONE
    };
    success = pathFindRecursive( &subTarget );
    worldPassReset();
  }

 Serial.print("Subtarget: "); Serial.print(subTarget.col);
Serial.println( subTarget.row);
```

```
  if (subTarget.col > 5)
    subTarget.col = 4;
  else if (subTarget.col < 0)
    subTarget.col = 0;

  if (subTarget.row > 5)
    subTarget.row = 4;
  else if (subTarget.row < 0)
    subTarget.row = 0;

  Serial.print("\t Modified Subtarget: ");
Serial.print(subTarget.col);Serial.println(subTarget.row);

  return success;
}

boolean pathFindRecursive(SectorPath *thisSector) {

  //Check if this is even a valid position
  if ( !inSectors(thisSector->col, thisSector->row) ||
getSectorPass(thisSector->col, thisSector->row) ==
PASSED_BLOCKED ) {
   resetSectorPassPlan(thisSector->col, thisSector->row);
    return false;
  }

  //Check if we hit the target
  if ( thisSector->col == pathTargetCol && thisSector->row ==
pathTargetRow) {
    //If we did, set this action as STOP
    thisSector->outAction = ACTION_STOP;
    //Set our subtarget to this
    subTarget.col = thisSector->col;
    subTarget.row = thisSector->row;
   resetSectorPassPlan(thisSector->col, thisSector->row);
    //Spread the word!
    return true;
  }

  //Mark our postion on the pass grid
 setSectorPassPlan( thisSector->col, thisSector->row);

  boolean success = false;
```

```
  //These track the direction indicies
  //Since we came from one direction,
  //it naturally follows that there are only three we can take
  //Thus, three variables
  byte headings[3] = {255, 255, 255};

  //And we only need to track the low and mid scores
  //Since we only care about their positions relative to each
other;
  byte scores[3] = {255, 255, 255};
  byte genScore = 255;

  //Now, probe our four options:
  for (int probeHead = 0; probeHead < THETA_INCREMENTS;
probeHead++) {

    genScore = headingScore(
                thisSector->col + projectCol(probeHead),
thisSector->row + projectRow(probeHead),
                thisSector->col, thisSector->row,
thisSector->inHeading, probeHead
               );

    //if the score is valid
    if ( genScore != 255 ) {
      //if it's lower than the lowest, or equal
      if ( genScore == scores[0] || min(genScore, scores[0]) ==
genScore ) {
        //boot the middle to the highest
        headings[2] = headings[1]; scores[2] = scores[1];
        //boot the lowest to middle
        headings[1] = headings[0]; scores[1] = scores[0];
        //set the lowest
        headings[0] = probeHead; scores[0] = genScore;
      }
      //else, if it's lower than the middle, or equal
      else if ( genScore == scores[1] || min(genScore,
scores[1]) == genScore ) {
        //boot the middle to the highest
        headings[2] = headings[1]; scores[2] = scores[1];
        //set the middle
        headings[1] = probeHead; scores[1] = genScore;
```

```
      }
      //else, if it's lower than the highest, or equal
      else if ( genScore == scores[2] || min(genScore,
scores[2]) == genScore )
        //set the highest
        headings[2] = probeHead; scores[2] = genScore;
    }
  }

  SectorPath nextSector = (SectorPath) {
    0, 0, 0, ACTION_NONE
  };

  for (int i = 0; i < 3 && !success; i++) {
    if (headings[i] != 255 && scores[i] != 255) {
      thisSector->outAction = headingsToAction(
thisSector->inHeading, headings[i] );

      nextSector.col = thisSector->col + projectCol( headings[i]
);
      nextSector.row = thisSector->row + projectRow( headings[i]
);
      nextSector.inHeading = actionToNewHeading(
thisSector->inHeading,thisSector->outAction);
      nextSector.outAction = ACTION_NONE;

      success = pathFindRecursive( &nextSector );
    }
  }

  //If we succeeded
  if (success)
    //And this sector's action is a waypoint action
    if (thisSector->outAction >= ACTION_STOP)
      //And its not the start sector
      if ( thisSector->col != valueToSectorMM( x ) ||
thisSector->row != valueToSectorMM( y ) ) {
        #ifdef DEBUG
        Serial.println("subtargetReassign"); Serial.
print(thisSector->col);Serial.print(thisSector->row);
        #endif
        subTarget.col = thisSector->col; subTarget.row =
thisSector->row;
```

```
      }

  //Finally, undo the marking so we can use the pass grid later
 resetSectorPassPlan(thisSector->col, thisSector->row);

  return success;
}

byte headingScore(int col, int row,  int currCol, int currRow,
int oldHeading, int newHeading) {
  //If a path isn't possible, reject it;
  if ( !isPathPossible( col, row ) )
    return 255;

  //if we are considering high probability squares dangerous,
and this is high prob, reject.
  if ( !boxForgive && statGrid[ row ][ col ] >= 0.50)
    return 255;

  //Base score
  byte score = 1;

  //If the parities of the heading don't match, that means we
have to turn - that's worth two points.
  if ( (oldHeading % 2) != (newHeading % 2) )
    score += 2;

  //If the sector is an unknown, we multiply the blockage
certainty by 10 and add it to the score
  if ( boxForgive && passGrid[ row ][ col ] == PASSED_UNSURE )
    score += byte( 10 * statGrid[ row ][ col ] );

  //If the sector is an unknown, and we are in GoHome or
GoIncline, unknown spaces cost more
  if ( machineState == GOHOME && passGrid[ row ][ col ] !=
PASSED_OPENED )
    score += 3;

  //If, relative to our last position, we're moving away from
the goal, we add 4.
  if ( abs(pathTargetCol - currCol) - abs(pathTargetCol - col )
< 0  ||
       abs(pathTargetRow - currRow) - abs(pathTargetRow - row )
```

```
< 0 )
    score += 4;

  return score;
}

//=============================================================
======================
// Path Finder Utility functions
//=============================================================
======================

//This function flips a symbol from it's regular pass state to a
"plan" state
inline void setSectorPassPlan(int col, int row) {
  if ( inSectors(col, row) ) {
    //If we're not sure about this tile, mark it as such
    if ( passGrid[ row ][ col ] == PASSED_UNSURE )
      passGrid[ row ][ col ] = PASSED_UNSURE_PLAN;

    //If we're not sure about this tile, mark it as such
    else if ( passGrid[ row ][ col ] == PASSED_OPENED )
      passGrid[ row ][ col ] = PASSED_OPENED_PLAN;
  }
}

//This function flips a symbol from a "plan" state to it's
regular pass state
inline void resetSectorPassPlan(int col, int row) {
  if ( inSectors(col, row) ) {
    //If we're not sure about this tile, mark it as such
    if ( passGrid[ row ][ col ] == PASSED_UNSURE_PLAN )
      passGrid[ row ][ col ] = PASSED_UNSURE;

    //If we're not sure about this tile, mark it as such
    else if ( passGrid[ row ][ col ] == PASSED_OPENED_PLAN )
      passGrid[ row ][ col ] = PASSED_OPENED;
  }
}

//This function checks if a symbol is in a plan state
inline boolean isPathPossible( int col, int row ) {
  //Basically, the passGrid cannot:
```

```
  //       - have already been pathed over
  //       - blocked by a box
  // in order for us to path over
  if ( inSectors(col, row) )
    return (passGrid[ row ][ col ] != PASSED_UNSURE_PLAN) &&
           (passGrid[ row ][ col ] != PASSED_OPENED_PLAN) &&
           (passGrid[ row ][ col ] != PASSED_BLOCKED);
  else
    return false;
}

byte headingsToAction(int oldHeading, int moveDirection) {
  if ( oldHeading == moveDirection)
    return ACTION_FORWARD;
  if ( abs(oldHeading - moveDirection) == 2)
    return ACTION_REVERSE;
  if ( radianRight(oldHeading) == moveDirection )
    return ACTION_RIGHT_FORWARD;
  if ( radianLeft(oldHeading) == moveDirection )
    return ACTION_LEFT_FORWARD;

  return ACTION_NONE;
}

int actionToNewHeading(IncrementalTheta oldHeading, int action) {
  switch (action) {
    default:
    case ACTION_FORWARD:
    case ACTION_REVERSE:
    case ACTION_NONE:
    case ACTION_STOP:
      return oldHeading;
      break;
    case ACTION_RIGHT_FORWARD:
      return radianRight(oldHeading);
      break;
    case ACTION_LEFT_FORWARD:
      return radianLeft(oldHeading);
      break;
  }

  return 255;
}
```

```
#define COLOR_HUE_FORWARD   0.0
#define COLOR_HUE_LEFT   0.25
#define COLOR_HUE_BACKWARD 0.5
#define COLOR_HUE_RIGHT  0.75

#define CLAV_BRAIN_TURN_SPEED_LEFT  40
#define NINETY_DEGREE_LEFT  69//67  //Right ticks to turn left

#define CLAV_BRAIN_TURN_SPEED_RIGHT 40
#define NINETY_DEGREE_RIGHT 69  //Left ticks to turn right //66
//70

#define PING_TARGET_CM_STOP   60 //We can see two boxes away!
#define PING_TARGET_CM_MOVE   24
//#define PING_TARGET_MM   500

#define INVESTIGATE_ITERATES   2

#define WANDER_TARGET_COL  4
#define WANDER_TARGET_ROW  4

#define GO_HOME_TARGET_COL  0
#define GO_HOME_TARGET_ROW  0

#define INCLINE_TARGET_COL  -5
#define INCLINE_TARGET_ROW  0

#define INCLINE_TARGET_THETA  ONE_PI

#define INCLINE_SPEED_FAST 16
#define INCLINE_SPEED_SLOW 8
#define COLLISION_SPEED   5
#define WANDER_SPEED      10
#define KNOWN_SPEED       15

int targetCol, targetRow;
boolean parity;

//================================================================
======================
// Initializer
//================================================================
```

```
=======================
inline void clavInit(){
  tickCountLeft = tickCountRight = 0;
  moveState = STOPPED;
  updateDirection();

 specificColor(COLOR_HUE_FORWARD);

  dTheta = 0;
  dX = 0;
  dY = 0;

  x = EXCESS / 2; //0;
  y = EXCESS / 2; //0;//200;//50.0;//80.0;
  thetaState = PI_O2;
  theta = radianToValue(thetaState);
  addShift(thetaState);

  targetCol = WANDER_TARGET_COL;
  targetRow = WANDER_TARGET_ROW;

  radianPanTo( RANGER_FORWARD );
  delay(50);
  parity = false;
  //anglePanTo(90);
  //Straight( 10, 1 );
}

//=============================================================
=======================
// Loop
//=============================================================
=======================
inline void clavBrain()
{

  //if(moveState == FORWARD)
    //pingCheck();

  //delay(100);

  //---- update robot config (x,y,theta)
  dX = PI * WHEEL_RADIUS * cos(theta) * ((double)(tickCountLeft
```

```
+ tickCountRight) / TICK_PER_ROT);
  xNoSkew = xNoSkew + dX;
  x = x + dX;

  dY = PI * WHEEL_RADIUS * sin(theta) * ((double)(tickCountLeft
+ tickCountRight) / TICK_PER_ROT);
  yNoSkew = yNoSkew + dY;
  y = y + dY;


  tickCountLeft = tickCountRight = 0;

  markCurrentPass();

  #ifdef DEBUG
  Serial.print(x); Serial.print(" :x || y: "); Serial.
println(y);Serial.print("\t");
  Serial.print(valueToSectorMM( x )); Serial.print(" :col ||
row: "); Serial.println(valueToSectorMM( y ));
  #endif

  //Change our light
  if(parity)
    radianToColor( radianToValue( moveEffectTheta() ) );
  else
    radianToColor( theta );

  parity = !parity;

  if(machineState == WANDER || machineState == WANDEROUT){
    clavGoPlace(true);
    //---- check if we're completely in the
    if ( posInTarget(targetCol, targetRow) )
     {
       ClavStop();

       #ifdef DEBUG
      Serial.println("======================");
       Serial.println("\t Wander Target Reached!");
      Serial.println("======================");
       #endif

       //---- update state
```

```
    radianPanTo( RANGER_FORWARD );
    delay(100);

    machineState = GOHOME;
    targetCol = GO_HOME_TARGET_COL;
    targetRow = GO_HOME_TARGET_ROW;
   }
}
else if (machineState == GOHOME)
{
  //---- use map to go home
  clavGoPlace(false);

  //---- check if we're completely in the
  if ( posInTarget(targetCol, targetRow) )
   {
    ClavStop();

    #ifdef DEBUG
   Serial.println("======================");
    Serial.println("\t Go Home - Target Reached!");
   Serial.println("======================");
    #endif

    machineState = INCLINE;
    targetCol = INCLINE_TARGET_COL;
    targetRow = INCLINE_TARGET_ROW;

    turnToTheta(ONE_PI);

    lightCycle(50);
   }
}
else if (machineState == INCLINE) {
  if ( getDiodeVal() > 100) {
    if (lightState == LEFT) {
      //Set Right
     MOTOR.setSpeedDir1(INCLINE_SPEED_FAST, DIRF);
      //Set Left
     MOTOR.setSpeedDir2(INCLINE_SPEED_SLOW, DIRR);
      lightState = RIGHT;
      delay(100);
    }
```

```
      else if (lightState == RIGHT){
        //Set Right
       MOTOR.setSpeedDir1(INCLINE_SPEED_SLOW, DIRF);
        //Set Left
       MOTOR.setSpeedDir2(INCLINE_SPEED_FAST, DIRR);
        lightState = LEFT;
        delay(100);
      }
    }

    //---- check if we're completely in the
    if ( x <= -1550 )
    {
      ClavStop();

      #ifdef DEBUG
     Serial.println("======================");
      Serial.println("\t Incline - Target Reached!");
     Serial.println("======================");
      #endif

      machineState = DONE;
    }
  }
  else if (machineState == DONE){
    lightCycle(100);
  }
}

//~~~~~~~~~~~~~~
// Sub-brain: Wander (for wandering, and the like)
//~~~~~~~~~~~~~~
void clavGoPlace(boolean investigate){

  //If we've stopped
  if( moveState == STOPPED ){
    int speedWeDo;
    //Investigate the surrounding area
    if(investigate){
      investigatePing();
      speedWeDo = WANDER_SPEED;
    }
    else
```

```
    speedWeDo = KNOWN_SPEED;

  #ifdef DEBUG
  printPassGrid();
  delay(100);
  printStatGrid();
  delay(100);
  #endif

  //Formulate a plan
 pathFindFromCurrent(targetCol, targetRow);

  tickCountLeft = tickCountRight = 0;

  if(investigate){
    radianPanTo( RANGER_FORWARD ); delay(50);
   }

  #ifdef DEBUG
  Serial.println( getStartAction() );
  #endif

  //Follow the plan
  switch( getStartAction() ){
    default:
    case ACTION_STOP:
    case ACTION_NONE:
      break;

    case ACTION_FORWARD:
      ClavForward(speedWeDo);break;

    case ACTION_REVERSE:
      ClavReverse(speedWeDo); break;

    case ACTION_LEFT_FORWARD:
      TurnLeft90(); ClavForward(speedWeDo); break;

    case ACTION_RIGHT_FORWARD:
      TurnRight90(); ClavForward(speedWeDo); break;
  }
}
//If we've reached our subtarget
```

```
  else if( posInSubTarget() ){
    //Stop!
    ClavStop();
    #ifdef DEBUG
   Serial.println("======================");
    Serial.println("\t Subtarget Reached!");
   Serial.println("======================");
    #endif
  }
  // If we're moving and we're suddenly attacked by a wild box
  else if( moveState == FORWARD && digitalRead(buttonPin) )
  {
    //---- stop
    ClavStop();
    //---- mark that we had a collision, adjust our sights
    blockCollision();

    tickCountLeft = tickCountRight = 0;

    ClavReverse(COLLISION_SPEED);

    #ifdef DEBUG
   Serial.println("======================");
    Serial.println("\t Button Pressed!");
   Serial.println("======================");
    #endif
  }
  //otherwise, if we're moving forward and we go out of bounds
  else if( (moveState == FORWARD || moveState == BACKWARD) &&
outOfBoundry() && machineState == WANDER )//(OUT_OF_BOUNDRY) )
  {
    int oldState = moveState;
    //---- stop
    ClavStop();

    //Mark that we've gone out of bounds
    machineState = WANDEROUT;

    #ifdef DEBUG
   Serial.println("======================");
    Serial.println("\tOut of Bounds!");
   Serial.println("======================");
    #endif
```

```
    if(oldState == FORWARD)
      ClavReverse(WANDER_SPEED);
    else if(oldState == BACKWARD)
      ClavForward(WANDER_SPEED);
  }
  else if ( machineState == WANDEROUT && inBoundry()
)//(IN_BOUNDRY))
  {
    //---- stop
    ClavStop();
    machineState = WANDER;
  }
  /*
  else if(machineState == WANDER || machineState == WANDEROUT){
    projectionCheck();
  }
  */
}

//=================================================================
======================
// Initiate the forward - this code is duplicated so much that
we might as well
//=================================================================
======================
inline void ClavForward(int inSpeed) {
  //---- update state
  moveState = FORWARD;
  updateDirection();
  //---- go forward
  forward(inSpeed);
  //---- R E S T    A E S T H E T I C A L L Y
  delay(100);
}

//=================================================================
======================
// Initiate the reverse - this code is duplicated so much that
we might as well
//=================================================================
======================
inline void ClavReverse(int inSpeed) {
```

```
  //---- update state
  moveState = BACKWARD;
  updateDirection();
  //---- back up
  backward(inSpeed);
  //---- R E S T   A E S T H E T I C A L L Y
  delay(100);
}

//================================================================
=======================
// Stop the car - this code is duplicated so much that we might
as well
//================================================================
=======================
inline void ClavStop() {
  //---- stop
  stopMotion();
  //---- R E S T   A E S T H E T I C A L L Y
  delay(100);
  //---- update state
  moveState = STOPPED;
  updateDirection();
}

//================================================================
=======================
// TurnLeft90
//================================================================
=======================
void TurnLeft90() {
    //Set up for turning left
   specificColor(COLOR_HUE_LEFT);
    moveState = TURN_LEFT;
    updateDirection();
    tickCountLeft = tickCountRight = 0;

    //begin the turn
   turnInPlaceLeft(CLAV_BRAIN_TURN_SPEED_LEFT);

    //while loop to turn as we please
    while (tickCountRight < NINETY_DEGREE_LEFT)
    {
```

```
      delayMicroseconds(1);
    }

    //Stop the turn
    ClavStop();

    #ifdef DEBUG
   Serial.println("======================");
    Serial.println("\t Turning Left!");
   Serial.println("======================");
    #endif

    tickCountLeft = tickCountRight = 0;

    removeShift(thetaState);

    //Update our theta (positively)
    incrementTheta();

    addShift(thetaState);

    //R E S T    A E S T H E T I C A L L Y
    delay(100);
}

//================================================================
======================
// TurnRight90
// dirn is 1 for right, -1 for left
//================================================================
======================
void TurnRight90() {
    //Set up for turning right
   specificColor(COLOR_HUE_RIGHT);
    moveState = TURN_RIGHT;
    updateDirection();
    tickCountLeft = tickCountRight = 0;

    //begin the turn
   turnInPlaceRight(CLAV_BRAIN_TURN_SPEED_RIGHT);

    //while loop to turn as we please
    while (tickCountLeft < NINETY_DEGREE_RIGHT)
```

```
    {
      delayMicroseconds(1);
    }

    //Stop the turn
    ClavStop();

    #ifdef DEBUG
   Serial.println("======================");
    Serial.println("\t Turning Right!");
   Serial.println("======================");
    #endif

    tickCountLeft = tickCountRight = 0;

    removeShift(thetaState);

    //Update our theta (negatively)
    decrementTheta();

    addShift(thetaState);

    //R E S T    A E S T H E T I C A L L Y
    delay(100);
}

//================================================================
=======================
// TurnToTheta
//================================================================
=======================
void turnToTheta(int target){
  switch( target - thetaState ){
    case -1: TurnRight90();
      break;
    case 1: TurnLeft90();
      break;
    case 2:
    case -2: TurnRight90(); TurnRight90();
      break;
    default:
    case 0:
      break;
```

```
  }
}

//Take a look around, see?
void investigatePing(){
  //start left
  int angle = RANGER_LEFT;

  //for each angle
  for(int i = 0; i < 3; i++){
    //Pan to that angle
    radianPanTo(angle);
    delay(50);

    //If whatever we're scanning isn't automatically going to be
out of bounds
    if( inSectors(valueToSectorMM( x ) + projectColPan(),
valueToSectorMM( y ) + projectRowPan()) ){
      //Then ping X times
      for(int j = 0; j < INVESTIGATE_ITERATES; j++){
        pingCheck();
        delay(100);
       }
     }
    angle = radianLeft(angle);
  }
  evalPanTheta();
}

inline void pingCheck(){
  boolean test;
  int dist_cm = pingCM();
  double objX, objY;
  int objRow, objCol;

  if( pingMoveCheck(dist_cm) || pingStopCheck(dist_cm) ) {
    objX = (x / 10) + cos( evalPanTheta() ) * dist_cm;
    objY = (y / 10) + sin( evalPanTheta() ) * dist_cm;

    objCol = valueToSectorCM( objX );
    objRow = valueToSectorCM( objY );

    if( objCol == valueToSectorMM( x ) && objRow ==
```

```
valueToSectorMM( y ) ){
        objCol += projectColPan();
        objRow += projectRowPan();
    }

    if( inSectors( objCol, objRow ) )
    {
      if( getSectorStat( objCol, objRow ) <= 0.0 )
        setSectorStat( objCol, objRow, 0.5);

      incrementSectorStat( objCol, objRow, ( 1.0 -
getSectorStat( objCol, objRow ) ) / 2.0 );
    }
  }
}

inline boolean pingMoveCheck(int dist_cm){
  return (moveState == FORWARD || moveState == BACKWARD) &&
dist_cm < PING_TARGET_CM_MOVE;
}

inline boolean pingStopCheck(int dist_cm){
  return moveState == STOPPED && dist_cm < PING_TARGET_CM_STOP;
}
```