|  |
| --- |
| Team 3 - 3 |
| Team Computing Project Document |
|  |

|  |
| --- |
| Michael Dunleavy, C15331521  Eoin Feely, C15470308  Jack Mc Cormack, C15581053  5-4-2016 |

# Strategy Used

We interpreted the project brief as having three distinct phases. Phase 1 is the calibration phase, where the grid is measured and the starting location found. Other calibration tasks are performed in this phase too. Phase 2 covers the mapping of the grid and the colours of its tiles. Phase 3 is the process of finding the obstacle on the grid and is in many ways similar to phase 2 in implementation. These three phases will be expanded upon below.

A key element of our strategy is to avoid hard coding values wherever possible and make the program as adaptable as possible. This means that there are a lot of calibrations that must be made when the program begins, and why we view calibrations as their own phase of the project. First we dynamically generate a light sensor threshold to accommodate different light levels and differently shaded grids. We calibrate how far the motors must turn to make the robot rotate n° in a point turn. This allows for different physical constructions of the robot. We measure the length of one grid tile, so the robot can operate on different sized grids. Then we calibrate the threshold distance between an obstacle and the robot. While these processes take time they make the robot much more versatile and the code much more adaptable. This modularity is what sets our robot apart. Our strategy is to first build a comprehensive library of tools and then use these tools to meet the project specification.

Once these setup routines are complete the robot is placed on the starting position (facing east) and will begin to measure the grid. It travels forward until it hits a double line. The distance it travelled is divided by the width of one square. This figure is added to grid’s width. The robot then backtracks the same distance to the starting position. The robot travels backward (ie. west) until it hits a double line. Again, the distance is divided then added to the grid’s width. This westerly distance is also the x co-ordinate of the starting position and is stored as such. These steps are then repeated for the y axis. With the grid’s dimensions and starting position we can create a blank map of the grid and write it to file. This way there is a partially completed map available if the program has to be terminated early. This completes phase 1.

Phase 2 involves traversing the grid and detecting the colour of each tile. To begin this traversal we move to position (0, 0), the top left corner. From here we move in a snaking motion made up of 4 sub-motions: east w tiles, south 1 tile, west w tiles, south 1 tile; where w is the width of the grid. These sub-motions are cycled in a loop. A counter variable stores the number of tiles mapped so far. Once this equals the area of the grid, the loop stops. We have set up these sub-motions so that the robot will take light readings at set distance intervals as it moves. It does not have to stop each time it reads a value. These light values are converted to a Boolean-like form and stored in a 2D array. Every time a value is read the array gets rewritten to file. Again, this allows the program to be prematurely terminated and still have useful data for debugging purposes. Once all data is collected the robot returns to the starting position and phase 3 begins.

For phase 3 the robot follows the same traversal from phase 2. The difference is that the traversal must be able to stop at any point once it detects an obstacle. This includes during the middle of sub-motions. The functions involved have to return whether or not they were interrupted so the program knows whether or not to continue the traversal. If an obstacle is found, we then write where we found it to file. These co-ordinates are simple to determine, as the robot’s position and orientation are tracked at all times. We then look up the colour of this tile from the 2D grid array from phase 2 and write this information to file. This completes both phase 3 and the entire program.

# Division of Work

We divided our work up on a practical basis, using the knowledge that as a team each member has both strengths and weaknesses. As such we tried to focus on working on those aspects we are strong at and getting help on the areas we found that came harder to us.

Michael focused on the development of high-level coding functions and code that would enable us to tackle any situation with ease, this was instrumental in allowing us to easily understand the code and make the necessary movements or actions, whenever we needed to, without having to write several complicated new lines every time we could simply just call the function.

Jack focused mainly on the build of the robot, sourcing materials online for development of our code and aiding with testing and debugging of newly developed code and functions during the laboratories. Trying to figure out simple and effective solutions to the complex, and not so complex issues we encountered along our development process.

Eoin was instrumental in the final testing and debugging phase of the process, despite being a new addition to our team he joined the team well and worked side by side to aid our efforts as best he could.

Documentation of the project and the writing of design documents and briefs was divided evenly among the team and we each again tried to focus on areas we were strongest at, by splitting the tasks required into groups and each member choosing different tasks to complete, we found this worked well for us.

In the end we found that all team members participated well together, worked to the best of their abilities and glued together well as a team to create a robot which could operate well in any environment.

# Lessons Learned

Throughout our project we learned many things, down below we will list the main takeaway points and lessons we gained from our experience working together in the module of Team Computing.

Working as a team to tackle a difficult assignment the best thing to do before you start is to take a while at the very start is to chat with your team mates, discuss your strengths and weaknesses and come up with a plan of action. Fortunately we did indeed do this and we learned that it was definitely the right way to go, we saw first-hand in our laboratories how easy it is to get lost and confused if you don’t have that plan to rely on.

The next thing we learned to do is to actively ask your teammates for help, a fresh set of eyes on your code or to solve an issue you are having can make all the difference. It is a misconception that the best idea to solve a problem is to stare at it on your screen until you have an epiphany, we found it much more effective and time efficient to call a teammate over and discuss the problem, possible solutions and then to go ahead and plan out how we would go about doing it.

Moving on from that we also realised that sometimes the best thing to do is take a few minutes to just decompress and relax. To clear your head so that when you do return back to the issue or problem or code, you can look at it clearly with a slightly different mind-set. This often led to some of our best ideas for development.

The final main lesson we learned in our experiences working together as teammates is that it is vital to actively listen to your teammates, and trust their judgement. The power of a team is much greater than one single person. Everyone has something to contribute and nobody is always right all the time.

# Test Plan

Our approach towards preparing the robot was a way we believed would work well for us, and it did. It was a simple and effective approach. We basically would code on the weekends and when it came around to the labs on Wednesday mornings, we would debug the code and test it with the robot, seeing what worked well and what didn’t. Having TOO much code sometimes helped us, because we could see what some things did in relation to others. If something didn’t work, we modified it or deleted it.

Taking that approach made sure it wasn’t too difficult to implement the plan of the project into the robot’s brain. We started with getting the robot to go through the whole grid, up and back to the start position. Then we moved onto counting the white squares and all the black squares and finally we coded the robot to find an object on the grid and take into account what square it was on, what colour the square was etc.

We coded this on the weekends and week by week in the labs we would work on debugging one section, e.g debug the robot going through the grid in week 9, debug the white/black square code in week 10.

# Printout of Code

#pragma config(Sensor, S1, touchSensor, sensorTouch)

#pragma config(Sensor, S3, lightSensor, sensorLightActive)

#pragma config(Sensor, S4, sonarSensor, sensorSONAR)

#pragma config(Motor, motorB, motorL, tmotorNXT, PIDControl, driveRight, encoder)

#pragma config(Motor, motorC, motorR, tmotorNXT, PIDControl, driveLeft, encoder)

//\*!!Code automatically generated by 'ROBOTC' configuration wizard !!\*//

#define DEFAULT\_PWR 20

#define SPACING\_THRESH 5.0

#define START 2

#define WHEEL\_WIDTH 5.6

#define MAP\_MAX\_ROWS 100

#define MAP\_MAX\_COLS 100

#define MAP\_PATH "team3-3\_file1.dat"

#define OBST\_FILE\_SZ 100

#define OBST\_PATH "team3-3\_file2.dat"

/\*

Mathmematically correct modulo operation instead of that trucated

nonsense C comes with.

\*/

#define MOD(n, m) ((((n) % (m)) + (m)) % (m))

enum Compass **{**E **=** 0**,** S **=** 90**,** W **=** 180**,** N **=** 270**};**

enum LightBool **{**DARK**,** LIGHT**};**

**typedef** struct Coord **{**

unsigned int x**,** y**;**

**}** Coord**;**

// initital calibrations

void calibrateLightThresh**(**void**);**

void calibrateRotationRatio**(**void**);**

void calibrateProximityTresh**(**void**);**

void calibrateSquareLen**(**void**);**

void initVariables**(**void**);**

// distance conversions

int cmToDeg**(**float cm**);**

float cmToRev**(**float cm**);**

float degToCm**(**int deg**);**

float revToCm**(**float rev**);**

// angle conversions

float degToRad**(**int deg**);**

int radToDeg**(**float rad**);**

// geometric formulae

int angleBetween**(**Coord p1**,** Coord p2**);**

float distBetween**(**Coord p1**,** Coord p2**);**

// turning

void face**(**int theta**);**

void pointTurn**(**int theta**);**

void pointTurn1**(**int theta**);**

// regular motion

void forwardNSquares**(**float n**);**

void moveNCm**(**float n**,** int pwrL**,** int pwrR**);**

void moveTo**(**Coord newPos**);**

// motion dependant on light sensor

float moveTilDark**(**int pwrL**,** int pwrR**);**

float moveTilDoubleLine**(**int spacingThresh**,** int pwrL**,** int pwrR**);**

float moveTilLight**(**int pwrL**,** int pwrR**);**

float moveTilLightChange**(**int pwrL**,** int pwrR**);**

// grid mapping

void mapCurrentPos**(**void**);**

void mapGrid**(**void**);**

void mapNSquares**(**int n**);**

void measureGrid**(**void**);**

void initGrid**(**void**);**

void writeGrid**(**void**);**

// obstacle detection

void setObstaclePos**(**void**);**

bool surveyGrid**(**void**);**

bool surveyNSquares**(**int n**);**

void writeObstaclePos**(**void**);**

// misc functions

void idleTilTouch**(**void**);**

void incrementRobotPos**(**void**);**

Coord robotPos**;**

int orientation**;**

unsigned int defaultPwr**;**

unsigned int wheelWdt**;**

float rotationRatio**;**

unsigned int lightThresh**;**

float proximityThresh**;**

unsigned int sqLen**;**

/\*

RobotC lacks dynamic memory allocation.

Therefore the grid must be of a fixed, excessive size.

\*/

char grid**[**MAP\_MAX\_ROWS**][**MAP\_MAX\_COLS**];**

unsigned int gridRows**,** gridCols**;**

unsigned int gridArea**;**

Coord startPos**;**

Coord obstaclePos**;**

unsigned int whiteC**,** blackC**;**

task main**()** **{**

Coord origin**;**

origin**.**x **=** 0**;**

origin**.**y **=** 0**;**

nxtDisplayTextLine**(**0**,** "Touch to begin"**);**

nxtDisplayTextLine**(**1**,** "phase 1:"**);**

nxtDisplayTextLine**(**2**,** "CALIBRATION"**);**

idleTilTouch**();**

eraseDisplay**();**

initVariables**();**

calibrateLightThresh**();**

calibrateRotationRatio**();**

calibrateSquareLen**();**

calibrateProximityTresh**();**

measureGrid**();**

/\*gridRows = 7;

gridCols = 9;

gridArea = gridRows \* gridCols;

startPos.x = 0;

startPos.y = 4;

robotPos = startPos;\*/

initGrid**();**

writeGrid**();**

moveTo**(**startPos**);**

face**(**E**);**

nxtDisplayTextLine**(**0**,** "Touch to begin"**);**

nxtDisplayTextLine**(**1**,** "phase 2:"**);**

nxtDisplayTextLine**(**2**,** "OBSTACLE"**);**

nxtDisplayTextLine**(**3**,** "DETECTION"**);**

idleTilTouch**();**

eraseDisplay**();**

moveTo**(**origin**);**

face**(**E**);**

nxtDisplayTextLine**(**0**,** "Place obstacle"**);**

nxtDisplayTextLine**(**1**,** "on grid."**);**

idleTilTouch**();**

eraseDisplay**();**

**if** **(**surveyGrid**())**

writeObstaclePos**();**

**}**

/\*

Calculate the light sensor threshold.

Sensor values above this threshold will be considered to be light.

Values below it will be dark.

\*/

void calibrateLightThresh**(**void**)** **{**

int light**,** dark**;**

eraseDisplay**();**

// get light reading

nxtDisplayTextLine**(**0**,** "Put me on a light"**);**

nxtDisplayTextLine**(**1**,** "surface."**);**

idleTilTouch**();**

light **=** SensorValue**(**lightSensor**);**

wait1Msec**(**250**);**

eraseDisplay**();**

// get dark reading

nxtDisplayTextLine**(**0**,** "Put me on a dark"**);**

nxtDisplayTextLine**(**1**,** "surface."**);**

idleTilTouch**();**

dark **=** SensorValue**(**lightSensor**);**

wait1Msec**(**250**);**

eraseDisplay**();**

// calc average of ligh and dark

lightThresh **=** **(**light **+** dark**)** **/** 2**;**

**}**

/\*

Calibrate how close an obstacle should be before it is detected.

\*/

void calibrateProximityTresh**(**void**)** **{**

eraseDisplay**();**

nxtDisplayTextLine**(**0**,** "Place me half a"**);**

nxtDisplayTextLine**(**1**,** "square from an"**);**

nxtDisplayTextLine**(**2**,** "obstacle."**);**

idleTilTouch**();**

eraseDisplay**();**

proximityThresh **=** SensorValue**(**sonarSensor**);**

**}**

/\*

Calibrates a figure called the rotation ratio.

This is the ratio of the wheels rotaion (at default power) to the

rotatation of the robot in a point turn.

ie. the wheels must turn 'rotationRatio' degrees to turn the robot

1 degree.

\*/

void calibrateRotationRatio**(**void**)** **{**

// current and previous light sensor readings

bool currLight**,** prevLight**;**

int encoderMean**;**

// brake

motor**[**motorL**]** **=** 0**;**

motor**[**motorR**]** **=** 0**;**

eraseDisplay**();**

nxtDisplayTextLine**(**0**,** "Put me on the"**);**

nxtDisplayTextLine**(**1**,** "right edge of a"**);**

nxtDisplayTextLine**(**2**,** "straight line."**);**

idleTilTouch**();**

eraseDisplay**();**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// convert light sensor reading to a boolean value

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// do until a new dark region is encountered

**do** **{**

// update sensor readings

prevLight **=** currLight**;**

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// set power

motor**[**motorL**]** **=** defaultPwr**;**

motor**[**motorR**]** **=** **-**defaultPwr**;**

**}** **while** **(**currLight **==** LIGHT **||** prevLight **==** DARK**);**

// brake

motor**[**motorL**]** **=** 0**;**

motor**[**motorR**]** **=** 0**;**

// calc average of the encoder readings

encoderMean **=** **(**nMotorEncoder**[**motorL**]** **+** nMotorEncoder**[**motorL**])** **/** 2**;**

// calc rotation ratio

rotationRatio **=** **(**float**)** encoderMean **/** 180.0**;**

**}**

/\*

Calibrate the length of 1 grid square.

Includes the length of the border.

\*/

void calibrateSquareLen**(**void**)** **{**

eraseDisplay**();**

nxtDisplayTextLine**(**0**,** "Put me on the"**);**

nxtDisplayTextLine**(**1**,** "border in front of"**);**

nxtDisplayTextLine**(**2**,** "2 white squares."**);**

idleTilTouch**();**

eraseDisplay**();**

sqLen **=** 0**;**

// move up to the inner edge of the border

moveTilLight**(**defaultPwr**,** defaultPwr**);**

// measure the length of the white square

sqLen **+=** moveTilDark**(**defaultPwr**,** defaultPwr**);**

// measure the width of the border

sqLen **+=** moveTilLight**(**defaultPwr**,** defaultPwr**);**

**}**

/\*

Initialise a selection of the global variables with default values.

\*/

void initVariables**(**void**)** **{**

orientation **=** 0**;**

defaultPwr **=** DEFAULT\_PWR**;**

wheelWdt **=** WHEEL\_WIDTH**;**

whiteC **=** 0**;**

blackC **=** 0**;**

nMotorPIDSpeedCtrl**[**motorL**]** **=** mtrSpeedReg**;**

nMotorPIDSpeedCtrl**[**motorR**]** **=** mtrSpeedReg**;**

**}**

/\*

Convert from centimeters travelled to degrees rotated by wheels.

\*/

int cmToDeg**(**float cm**)** **{**

**return** **(**int**)** **(**cmToRev**(**cm**)** **\*** 360**);**

**}**

/\*

Convert from centimeters travelled to revolutions rotated by

wheels.

\*/

float cmToRev**(**float cm**)** **{**

**return** cm **/** **(**PI **\*** wheelWdt**);**

**}**

/\*

Convert from degrees rotated by wheels to centimeters travelled.

\*/

float degToCm**(**int deg**)** **{**

**return** revToCm**((**float**)** deg **/** 360**);**

**}**

/\*

Convert from revolutions rotated by wheels to centimeters

travelled.

\*/

float revToCm**(**float rev**)** **{**

**return** rev **\*** **(**PI **\*** wheelWdt**);**

**}**

/\*

Convert an angle from degrees to radians.

\*/

float degToRad**(**int deg**)** **{**

**return** **(**float**)** deg **\*** **(**PI **/** 180**);**

**}**

/\*

Convert an angle from radians to degrees.

\*/

int radToDeg**(**float rad**)** **{**

**return** **(**int**)** **(**rad **\*** 180 **/** PI**);**

**}**

/\*

Return the angle between two points.

\*/

int angleBetween**(**Coord p1**,** Coord p2**)** **{**

**return** radToDeg**(**atan2**(**p2**.**y **-** p1**.**y**,** p2**.**x **-** p1**.**x**));**

**}**

/\*

Return the distance between two points.

\*/

float distBetween**(**Coord p1**,** Coord p2**)** **{**

**return** sqrt**(**pow**(**p2**.**x **-** p1**.**x**,** 2**)** **+** pow**(**p2**.**y **-** p1**.**y**,** 2**));**

**}**

/\*

Face a given orientation (in degrees) by performing a point turn.

\*/

void face**(**int theta**)** **{**

int diff**;**

// difference in orientation

diff **=** MOD**((**theta **-** orientation**),** 360**);**

// turn clockwise if it's faster to do so, else anti-clockwise

**if** **(**diff **<** 180**)**

pointTurn**(**diff**);**

**else**

pointTurn**(-(**360 **-** diff**));**

// update robot's orientation

orientation **=** theta**;**

**}**

/\*

Perform a point turn of a given angle in degrees.

Positive angles yield clockwise rotations, and negative yield

anti-clockwise rotations.

\*/

void pointTurn**(**int theta**)** **{**

**if** **(!**theta**)**

**return;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// set motor encoder targets for more accurate movements

nMotorEncoderTarget**[**motorL**]** **=** abs**(**theta**)** **\*** rotationRatio**;**

nMotorEncoderTarget**[**motorR**]** **=** abs**(**theta**)** **\*** rotationRatio**;**

// set power

**if** **(**theta **>** 0**)** **{**

motor**[**motorL**]** **=** **-**defaultPwr**;**

motor**[**motorR**]** **=** defaultPwr**;**

**}**

**else** **{**

motor**[**motorL**]** **=** defaultPwr**;**

motor**[**motorR**]** **=** **-**defaultPwr**;**

**}**

// idle loop while motor encoders haven't met their target

**while** **(**

nMotorRunState**[**motorL**]** **!=** runStateIdle **||**

nMotorRunState**[**motorR**]** **!=** runStateIdle

**)** **{**

**}**

**}**

/\*

Perform a point turn of a given angle in degrees.

Positive angles yield clockwise rotations, and negative yield

anti-clockwise rotations.

This implementation does not use motor encoder targets and is less

accurate.

\*/

void pointTurn1**(**int theta**)** **{**

float distance**;**

int encoderMean**;**

float travelled**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// distance the wheels must travel and how far they've already gone

distance **=** abs**(**theta**)** **\*** rotationRatio**;**

travelled **=** 0**;**

// while robot has not travelled the required distance

**while** **(**travelled **<** distance**)** **{**

// set power

**if** **(**theta **>** 0**)** **{**

motor**[**motorL**]** **=** **-**defaultPwr**;**

motor**[**motorR**]** **=** defaultPwr**;**

**}**

**else** **{**

motor**[**motorL**]** **=** defaultPwr**;**

motor**[**motorR**]** **=** **-**defaultPwr**;**

**}**

// calc distance travelled so far

encoderMean **=** **(**nMotorEncoder**[**motorL**]** **+** nMotorEncoder**[**motorR**])** **/** 2**;**

travelled **=** degToCm**(**encoderMean**);**

**}**

// brake

motor**[**motorL**]** **=** 0**;**

motor**[**motorR**]** **=** 0**;**

**}**

/\*

Move forward a given distance (measured in grid squares).

\*/

void forwardNSquares**(**float n**)** **{**

int encoderTarget**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// set motor encoder target

encoderTarget **=** n **\*** cmToDeg**(**sqLen**);**

nMotorEncoderTarget**[**motorL**]** **=** encoderTarget**;**

nMotorEncoderTarget**[**motorR**]** **=** encoderTarget**;**

// set motor power

motor**[**motorL**]** **=** defaultPwr**;**

motor**[**motorR**]** **=** defaultPwr**;**

// idle loop while motor encoders haven't met their target

**while** **(**

nMotorRunState**[**motorL**]** **!=** runStateIdle **||**

nMotorRunState**[**motorR**]** **!=** runStateIdle

**)** **{**

**}**

**}**

/\*

Move the robot a given number of centimeters.

The power levels of the motors are given as arguments.

\*/

void moveNCm**(**float n**,** int pwrL**,** int pwrR**)** **{**

int signedEncoderL**,** signedEncoderR**;**

int encoderMean**;**

float travelled**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

travelled **=** 0**;**

// while robot has not travelled the n centimeters

**while** **(**abs**(**travelled**)** **<** abs**(**n**))** **{**

// set power

motor**[**motorL**]** **=** pwrL**;**

motor**[**motorR**]** **=** pwrR**;**

// convert motor encoder values to signed integers

**if** **(**pwrL **<** 0**)**

signedEncoderL **=** **-**nMotorEncoder**[**motorL**];**

**else**

signedEncoderL **=** nMotorEncoder**[**motorL**];**

**if** **(**pwrR **<** 0**)**

signedEncoderR **=** **-**nMotorEncoder**[**motorR**];**

**else**

signedEncoderR **=** nMotorEncoder**[**motorR**];**

// calc distance travelled so far

encoderMean **=** **(**signedEncoderL **+** signedEncoderR**)** **/** 2**;**

travelled **=** degToCm**(**encoderMean**);**

**}**

// brake

motor**[**motorL**]** **=** 0**;**

motor**[**motorR**]** **=** 0**;**

**}**

/\*

Move to a given co-ordinate.

Update the robot's position and orientation aftwerwards.

\*/

void moveTo**(**Coord newPos**)** **{**

int theta**;**

float dist**;**

// turn to face newPos

theta **=** angleBetween**(**robotPos**,** newPos**);**

face**(**theta**);**

orientation **=** theta**;**

// move forward to newPos

dist **=** distBetween**(**robotPos**,** newPos**);**

forwardNSquares**(**dist**);**

robotPos **=** newPos**;**

**}**

/\*

Move until the light sensor detects a change from light to dark.

Robot will still move if it starts on a dark area.

Motor powers are given as arguments.

Return the total distance travelled (in centimeters).

\*/

float moveTilDark**(**int pwrL**,** int pwrR**)** **{**

// current and previous light sensor readings

bool currLight**,** prevLight**;**

int encoderMean**;**

float travelled**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// convert light sensor reading to a boolean value

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// do until a new light region is encountered

**do** **{**

// update sensor readings

prevLight **=** currLight**;**

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// set power

motor**[**motorL**]** **=** pwrL**;**

motor**[**motorR**]** **=** pwrR**;**

**}** **while** **(**currLight **==** LIGHT **||** prevLight **==** DARK**);**

// brake

motor**[**motorL**]** **=** 0**;**

motor**[**motorR**]** **=** 0**;**

// measure distance travelled during function call

encoderMean **=** **(**nMotorEncoder**[**motorL**]** **+** nMotorEncoder**[**motorL**])** **/** 2**;**

travelled **=** degToCm**(**encoderMean**);**

**return** travelled**;**

**}**

/\*

Move robot until it passes two dark lines whose spacing is less

than a given threshold. (This threshold includes the width of the

first line because I'm lazy.)

Return the total distance travelled (in centimeters).

\*/

float moveTilDoubleLine**(**int spacingThresh**,** int pwrL**,** int pwrR**)** **{**

float recentDist**,** totalDist**;**

//moveNCm(sqLen, pwrL, pwrR);

//totalDist = sqLen;

**do** **{**

recentDist **=** abs**(**moveTilDark**(**pwrL**,** pwrR**));**

totalDist **+=** recentDist**;**

eraseDisplay**();**

nxtDisplayTextLine**(**0**,** "Recent dist: %.1f"**,** recentDist**);**

nxtDisplayTextLine**(**1**,** "Total dist: %.1f"**,** totalDist**);**

**}** **while** **(**recentDist **>** spacingThresh **||** totalDist **<** 1.2 **\*** sqLen**);**

**return** totalDist**;**

**}**

/\*

Move until the light sensor detects a change from dark to light.

Robot will still move if it starts on a light area.

Motor powers are given as arguments.

Return the total distance travelled (in centimeters).

\*/

float moveTilLight**(**int pwrL**,** int pwrR**)** **{**

// current and previous light sensor readings

bool currLight**,** prevLight**;**

int encoderMean**;**

float travelled**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// convert light sensor reading to a boolean value

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// do until a new light region is encountered

**do** **{**

// update sensor readings

prevLight **=** currLight**;**

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// set power

motor**[**motorL**]** **=** pwrL**;**

motor**[**motorR**]** **=** pwrR**;**

**}** **while** **(**currLight **==** DARK **||** prevLight **==** LIGHT**);**

// brake

motor**[**motorL**]** **=** 0**;**

motor**[**motorR**]** **=** 0**;**

// measure distance travelled during function call

encoderMean **=** **(**nMotorEncoder**[**motorL**]** **+** nMotorEncoder**[**motorL**])** **/** 2**;**

travelled **=** degToCm**(**encoderMean**);**

**return** travelled**;**

**}**

/\*

Move until the light sensor detects a change from light to dark or

dark to light.

Motor powers are given as arguments.

Return the total distance travelled (in centimeters).

\*/

float moveTilLightChange**(**int pwrL**,** int pwrR**)** **{**

// current and previous light sensor readings

bool currLight**,** prevLight**;**

int encoderMean**;**

float travelled**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// convert light sensor reading to a boolean value

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// do until current and previous readings are different

**do** **{**

// update sensor readings

prevLight **=** currLight**;**

currLight **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

// set power

motor**[**motorL**]** **=** pwrL**;**

motor**[**motorR**]** **=** pwrR**;**

**}** **while** **(**currLight **==** prevLight**);**

// brake

motor**[**motorL**]** **=** 0**;**

motor**[**motorR**]** **=** 0**;**

// measure distance travelled during function call

encoderMean **=** **(**nMotorEncoder**[**motorL**]** **+** nMotorEncoder**[**motorL**])** **/** 2**;**

travelled **=** degToCm**(**encoderMean**);**

**return** travelled**;**

**}**

/\*

Mark the robot's starting position on the grid,

and initialise all other valid squares as white.

\*/

void initGrid**(**void**)** **{**

unsigned int i**,** j**;**

// initialise each square to LIGHT

**for** **(**i **=** 0**;** i **<** gridRows**;** i**++)** **{**

**for** **(**j **=** 0**;** j **<** gridCols**;** j**++)**

grid**[**i**][**j**]** **=** LIGHT**;**

**}**

// mark starting position

grid**[**startPos**.**y**][**startPos**.**x**]** **=** 2**;**

**}**

/\*

Measure the dimensions of the grid and note the robot's starting

position.

\*/

void measureGrid**(**void**)** **{**

int dist**;**

gridRows **=** 0**;**

gridCols **=** 0**;**

nxtDisplayTextLine**(**0**,** "Put me on the"**);**

nxtDisplayTextLine**(**1**,** "starting position."**);**

idleTilTouch**();**

eraseDisplay**();**

// move east

dist **=** moveTilDoubleLine**(**SPACING\_THRESH**,** defaultPwr**,** defaultPwr**);**

moveNCm**(**dist**,** **-**defaultPwr**,** **-**defaultPwr**);**

gridCols **+=** dist **/** sqLen **-** 1**;**

wait1Msec**(**1000**);**

// move west

dist **=** moveTilDoubleLine**(**SPACING\_THRESH**,** **-**defaultPwr**,** **-**defaultPwr**);**

moveNCm**(**dist**,** defaultPwr**,** defaultPwr**);**

gridCols **+=** dist **/** sqLen **-** 1**;**

startPos**.**x **=** dist **/** sqLen**;**

wait1Msec**(**1000**);**

// move south

face**(**S**);**

dist **=** moveTilDoubleLine**(**SPACING\_THRESH**,** defaultPwr**,** defaultPwr**);**

moveNCm**(**dist**,** **-**defaultPwr**,** **-**defaultPwr**);**

gridRows **+=** dist **/** sqLen **-** 1**;**

wait1Msec**(**1000**);**

// move north

dist **=** moveTilDoubleLine**(**SPACING\_THRESH**,** **-**defaultPwr**,** **-**defaultPwr**);**

moveNCm**(**dist**,** defaultPwr**,** defaultPwr**);**

gridRows **+=** dist **/** sqLen **-** 1**;**

startPos**.**y **=** dist **/** sqLen**;**

// update robot's current position

robotPos **=** startPos**;**

// measure area

gridArea **=** gridCols **\*** gridRows**;**

eraseDisplay**();**

nxtDisplayTextLine**(**0**,** "Width: %d"**,** gridCols**);**

nxtDisplayTextLine**(**1**,** "Height: %d"**,** gridRows**);**

nxtDisplayTextLine**(**2**,** "Area: %d"**,** gridArea**);**

wait1Msec**(**3000**);**

**}**

/\*

Take a light reading from the robot's current position and update

the grid and colour counters with it.

\*/

void mapCurrentPos**(**void**)** **{**

bool reading**;**

reading **=** **(**bool**)** **(**SensorValue**(**lightSensor**)** **/** lightThresh**);**

**if** **(**reading **==** LIGHT**)**

whiteC**++;**

**else**

blackC**++;**

// add reading to grid

**if** **(**grid**[**robotPos**.**y**][**robotPos**.**x**]** **!=** START**)**

grid**[**robotPos**.**y**][**robotPos**.**x**]** **=** **(**char**)** **((**int**)** reading**);**

eraseDisplay**();**

nxtDisplayTextLine**(**

0**,** "grid[%d][%d] = %d"**,**

robotPos**.**y**,** robotPos**.**x**,** **(**int**)** reading

**);**

writeGrid**();**

**}**

/\*

Traverse the entire grid and take a light reading on each square.

\*/

void mapGrid**(**void**)** **{**

unsigned int mapped**;**

unsigned int step**;**

mapCurrentPos**();**

mapped **=** 1**;**

step **=** 0**;**

/\*

While the grid is not fully mapped,

perform 1 step of a 4 step snaking traversal.

\*/

**while** **(**mapped **<** gridArea**)** **{**

**switch** **(**step **%** 4**)** **{**

// step 0: east, gridCols squares

**case** 0**:** **{**

face**(**E**);**

mapNSquares**(**gridCols**);**

mapped **+=** gridCols**;**

**break;**

**}**

// step 1: south, 1 square

**case** 1**:** **{**

face**(**S**);**

mapNSquares**(**1**);**

mapped**++;**

**break;**

**}**

// step 2: west, gridCols squares

**case** 2**:** **{**

face**(**W**);**

mapNSquares**(**gridCols**);**

mapped **+=** gridCols**;**

**break;**

**}**

// step 3: south, 1 square

**case** 3**:** **{**

face**(**S**);**

mapNSquares**(**1**);**

mapped**++;**

**break;**

**}**

**default:** **{**

//TODO error log

**}**

**}**

step**++;**

**}**

**}**

/\*

Move forward a given number of squares and take a light reading on

each square passed.

The robot's orientation must be a multiple of 90 degrees.

\*/

void mapNSquares**(**int n**)** **{**

int encoderTarget**;**

int encoderMean**;**

int marker**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// set motor encoder target

encoderTarget **=** n **\*** cmToDeg**(**sqLen**);**

nMotorEncoderTarget**[**motorL**]** **=** encoderTarget**;**

nMotorEncoderTarget**[**motorR**]** **=** encoderTarget**;**

// set motor power

motor**[**motorL**]** **=** defaultPwr**;**

motor**[**motorR**]** **=** defaultPwr**;**

// mark where to take the next light reading

marker **=** cmToDeg**(**sqLen**)** **/** 2**;**

// while motor encoders haven't met their target

**while** **(**

nMotorRunState**[**motorL**]** **!=** runStateIdle **||**

nMotorRunState**[**motorR**]** **!=** runStateIdle

**)** **{**

encoderMean **=** **(**nMotorEncoder**[**motorL**]** **+** nMotorEncoder**[**motorR**])** **/** 2**;**

// if the robot passed the marker

**if** **(**encoderMean **>** marker**)** **{**

// update the robot's position

incrementRobotPos**();**

// take a light reading

mapCurrentPos**();**

// set a new marker

marker **+=** cmToDeg**(**sqLen**);**

**}**

**}**

**}**

/\*

Write the grid array to a text file in a readable format.

\*/

void writeGrid**(**void**)** **{**

const char whiteChar **=** ' '**;**

const char blackChar **=** 'X'**;**

const char startChar **=** 'S'**;**

const char errorChar **=** 'E'**;**

TFileHandle file**;**

TFileIOResult ioRet**;**

short fileSize**;**

unsigned int i**,** j**;**

// delete old file

Delete**(**MAP\_PATH**,** ioRet**);**

// open file

fileSize **=** gridArea **\*** 3 **+** gridRows**;**

OpenWrite**(**file**,** ioRet**,** MAP\_PATH**,** fileSize**);**

// write each sqaure

**for** **(**i **=** 0**;** i **<** gridRows**;** i**++)** **{**

**for** **(**j **=** 0**;** j **<** gridCols**;** j**++)** **{**

WriteByte**(**file**,** ioRet**,** '['**);**

**switch** **(**grid**[**i**][**j**])** **{**

**case** DARK**:** **{**

WriteByte**(**file**,** ioRet**,** blackChar**);**

**break;**

**}**

**case** LIGHT**:** **{**

WriteByte**(**file**,** ioRet**,** whiteChar**);**

**break;**

**}**

**case** START**:** **{**

WriteByte**(**file**,** ioRet**,** startChar**);**

**break;**

**}**

**default:** **{**

WriteByte**(**file**,** ioRet**,** errorChar**);**

//TODO error log

**}**

**}**

WriteByte**(**file**,** ioRet**,** ']'**);**

**}**

WriteByte**(**file**,** ioRet**,** '\n'**);**

**}**

// terminate and close file

Close**(**file**,** ioRet**);**

**}**

/\*

Using the robot's current posiotion and orientation,

and knowing that the obstacle is 1 tile in front of the robot,

set the co-ordinates of the obstacle.

\*/

void setObstaclePos**(**void**)** **{**

obstaclePos **=** robotPos**;**

**switch** **(**orientation**)** **{**

**case** E**:** **{**

obstaclePos**.**x**++;**

**break;**

**}**

**case** S**:** **{**

obstaclePos**.**y**++;**

**break;**

**}**

**case** W**:** **{**

obstaclePos**.**x**--;**

**break;**

**}**

**case** N**:** **{**

obstaclePos**.**y**--;**

**break;**

**}**

**default:** **{**

//TODO error log

**}**

**}**

**}**

bool surveyGrid**(**void**)** **{**

unsigned int surveyed**;**

unsigned int step**;**

bool foundObstacle**;**

mapCurrentPos**();**

surveyed **=** 1**;**

step **=** 0**;**

/\*

While the grid is not fully surveyed,

perform 1 step of a 4 step snaking traversal.

\*/

**while** **(**surveyed **<** gridArea**)** **{**

**switch** **(**step **%** 4**)** **{**

// step 0: east, gridCols squares

**case** 0**:** **{**

face**(**E**);**

foundObstacle **=** surveyNSquares**(**gridCols**);**

surveyed **+=** gridCols**;**

**break;**

**}**

// step 1: south, 1 square

**case** 1**:** **{**

face**(**S**);**

foundObstacle **=** surveyNSquares**(**1**);**

surveyed**++;**

**break;**

**}**

// step 2: west, gridCols squares

**case** 2**:** **{**

face**(**W**);**

foundObstacle **=** surveyNSquares**(**gridCols**);**

surveyed **+=** gridCols**;**

**break;**

**}**

// step 3: south, 1 square

**case** 3**:** **{**

face**(**S**);**

foundObstacle **=** surveyNSquares**(**1**);**

surveyed**++;**

**break;**

**}**

**default:** **{**

//TODO error log

**}**

**}**

**if** **(**foundObstacle**)** **{**

setObstaclePos**();**

**return** true**;**

**}**

step**++;**

**}**

**return** false**;**

**}**

bool surveyNSquares**(**int n**)** **{**

int encoderTarget**;**

int encoderMean**;**

int marker**;**

// reset motor encoders

nMotorEncoder**[**motorL**]** **=** 0**;**

nMotorEncoder**[**motorR**]** **=** 0**;**

// set motor encoder target

encoderTarget **=** n **\*** cmToDeg**(**sqLen**);**

nMotorEncoderTarget**[**motorL**]** **=** encoderTarget**;**

nMotorEncoderTarget**[**motorR**]** **=** encoderTarget**;**

// set motor power

motor**[**motorL**]** **=** defaultPwr**;**

motor**[**motorR**]** **=** defaultPwr**;**

// mark where to take the next ultrasonic reading

marker **=** cmToDeg**(**sqLen**)** **/** 2**;**

// while motor encoders haven't met their target

**while** **(**

nMotorRunState**[**motorL**]** **!=** runStateIdle **||**

nMotorRunState**[**motorR**]** **!=** runStateIdle

**)** **{**

// take an ultrasonic reading

**if** **(**SensorValue**(**sonarSensor**)** **<** proximityThresh**)** **{**

obstaclePos **=** robotPos**;**

**return** true**;**

**}**

encoderMean **=** **(**nMotorEncoder**[**motorL**]** **+** nMotorEncoder**[**motorR**])** **/** 2**;**

// if the robot passed the marker

**if** **(**encoderMean **>** marker**)** **{**

// update the robot's position

incrementRobotPos**();**

// set a new marker

marker **+=** cmToDeg**(**sqLen**);**

**}**

**}**

**return** false**;**

**}**

void writeObstaclePos**(**void**)** **{**

TFileHandle file**;**

TFileIOResult ioRet**;**

short fileSize**;**

// delete old file

Delete**(**OBST\_PATH**,** ioRet**);**

// open file

fileSize **=** OBST\_FILE\_SZ**;**

OpenWrite**(**file**,** ioRet**,** OBST\_PATH**,** fileSize**);**

// write obstacle's co-ordinates

WriteText**(**file**,** ioRet**,** "Obstacle position: "**);**

WriteShort**(**file**,** ioRet**,** obstaclePos**.**x**);**

WriteText**(**file**,** ioRet**,** ", "**);**

WriteShort**(**file**,** ioRet**,** obstaclePos**.**y**);**

WriteByte**(**file**,** ioRet**,** '\n'**);**

// write colour of obstacle's tile

WriteText**(**file**,** ioRet**,** "Tile colour: "**);**

**switch** **(**grid**[**obstaclePos**.**y**][**obstaclePos**.**x**])** **{**

**case** DARK**:** **{**

WriteText**(**file**,** ioRet**,** "Black\n"**);**

**break;**

**}**

**case** LIGHT**:** **{**

WriteText**(**file**,** ioRet**,** "White\n"**);**

**break;**

**}**

**default:** **{**

WriteText**(**file**,** ioRet**,** "Unknown\n"**);**

**}**

**}**

// terminate and close file

Close**(**file**,** ioRet**);**

**}**

/\*

Wait idly until the touch sensor is pressed.

\*/

void idleTilTouch**(**void**)** **{**

**while** **(**SensorValue**(**touchSensor**)** **==** 0**)** **{**

// idle loop

**}**

wait1Msec**(**250**);**

**}**

/\*

Using the robot's orientation, update its current position to

reflect moving forward one square.

\*/

void incrementRobotPos**(**void**)** **{**

**switch** **(**orientation**)** **{**

**case** E**:** **{**

robotPos**.**x**++;**

**break;**

**}**

**case** S**:** **{**

robotPos**.**y**++;**

**break;**

**}**

**case** W**:** **{**

robotPos**.**x**--;**

**break;**

**}**

**case** N**:** **{**

robotPos**.**y**--;**

**break;**

**}**

**default:** **{**

//TODO error log

**}**

**}**

**}**