Generally speaking, the task has been dividend into 2 stage with several smaller tasks:

* Stage1:

1. Start at a tile, towards the lane which is consisted of black and white tiles.
2. Walk in the lane.
3. Count encountered black tiles while making sounds to indicate that.
4. Stopped after encountered 15 black tiles and turn right

* Stage2:

1. The robot should move forward which the tower should be in front of it.
2. It should make contact the target
3. Push tower off the square it sitting on and make a sound to indicate it has finished.

The main associated problems need to be solved are:

1. The robot should distinguish the black and one tile. (stage1)
2. The robot should walk in the boundaries next to its both side. (stage1)
3. The robot should head towards to the tower which its position is not uncertain. (stage2)

When we try to implement the algorithm to finish the tasks. Our first intention is naturally try to solve them as human. We want to use the ROBOTC’s task ability to execute the different task concurrently. It is very nature to think that way. For example, for the task of walking ahead and counting the black tile it encountered. It is just “let it keep walking” and “count when it meets a black tile”. This executing multiple tasks functionality the ROBOTC gives us is very powerful. It works well at the beginning when we just want to finish two small tasks. However, as we develop more functions and add into the previous one, we found some annoying bugs which are very hard to debug. At last we cannot trace the logic in the code and everything seems in a mess.

So instead of executing multiple tasks at the same time, we decide to break the task into multiple same steps, and in each step we let it finish multiple function. For example, in stage1, there are 15 same steps, in each step, the robot should should finish those simple tasks:

1. It moves ahead.
2. Stops and makes a sound once it encountered a black tile.
3. It detects its positon relative in the black tile.
4. It corrects its errors according to its position.

Once it repeated this step for 15 times, it will finish the stage1. This approach makes our code become much simpler to debug (After all, this is the nature way to write programming). Similar for stage2, the robot needs to finish basically two tasks:

1. Moves forward and stop when it encountered a black tile.
2. Scan for a certain degree ahead of it which ended towards the direction of target.

Except for this problem, other problem seems easier to solve. For, example, the movement of robot depends heavily on the moving and turning. But the command for controlling the robot’s movement at specific degree is error prone. So, for keeping consistency and reduce the error, we create some helper functions to wrap those commands. The helper functions can separate the functionality of movement away from the logic of our task. When we implement the turning and moving, we can just focus on manipulating the build-in functions instead of checking the code and the logic scenario back and forth.

If the previous problems are related to the implementation of algorithms, then there are other problems are some how harder to solve. The first problem which is the biggest problem is the debugging. Instead of check some logs or just see the Exception has been thrown during running time, we have very limited debug method, such as using sound or display some data on the LED. These methods are not that useful, specially when we want to improve the consistency of robot performance.

Another big problem we meet is the team work. Obviously, this is the first opportunity for us to work as a team. We lack the experience of version control which make us hard to merge everyone’s work into the code, meanwhile keep everyone develop his own part. So at last, three of us just sit in front of one computer and develop the code into one file. The most backward of this is that it limited other group member’s idea because during the development only one person’s idea is the main idea and other person will just think it is the best and follow it. And because we only have one robot, we just concentrate on implement one algorithm. Nick’s original idea for stage1 has been ignored exactly because of this. We regret so much that we haven’t explore other group member’s idea. We could do better simply do more thinking than coding.

In general, through this assignment, we really appreciate that it is hard to build some robot to finish even simple task. The environment is uncertain and making robot behave based on what it can sense is much more interesting than we expected.

The code is listed as below:

#pragma config(Sensor, S1, leftBumper, sensorEV3\_Touch)

#pragma config(Sensor, S2, rightBumper, sensorEV3\_Touch)

#pragma config(Sensor, S3, colorSensor, sensorEV3\_Color, modeEV3Color\_Color)

#pragma config(Sensor, S4, sonarSensor, sensorEV3\_Ultrasonic)

#pragma config(Motor, motorB, leftMotor, tmotorEV3\_Large, PIDControl, driveLeft, encoder)

#pragma config(Motor, motorC, rightMotor, tmotorEV3\_Large, PIDControl, driveRight, encoder)

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

// GOLOBAL VARIABLES

int totalBlackTile = 0;

int MOVINGSPEED = 30;

float targetDistance = 255.0;

// variables for scan

bool RIGHT = true;

bool LEFT = false;

bool approachingTarget = false;

int TURNINGSPEED = 10;

// variables for test boundary

int MAX\_BOUNDARY\_EXPECTED = 4\*100;

int rightBoundaryDegree = 0;

int leftBoundaryDegree = 0;

int differences = 30;

/\*\*DEFINED FUNCTION \*/

/\*\* in these turning function, the robot do two things:

\* turning and recording the distance it get from the sensor.

\* For turning, it turns in its fixed postion with one wheel roll ahead and the other roll backward.

\* For scanning, it keep geting the distance in front ot it, it the distance should decrease first,

\* and increase as it turns away from the robot.

\* just as the distance is getting larger, it stop the turning.s

\*/

void turn(int degreeToTurn, int TURNINGSPEED){

float previousDistance = getUSDistance(sonarSensor);

float currentDistance = getUSDistance(sonarSensor);

degreeToTurn = degreeToTurn \* 2;

resetMotorEncoder(leftMotor);

resetMotorEncoder(rightMotor);

if (degreeToTurn >= 0) {

setMotorTarget(leftMotor, degreeToTurn, TURNINGSPEED);

setMotorTarget(rightMotor, -degreeToTurn, TURNINGSPEED);

while(getMotorEncoder(leftMotor)!=getMotorTarget(leftMotor) && getTouchValue(leftBumper)!=1 && getTouchValue(rightBumper)!=1) {

setMotorSync(leftMotor, rightMotor, 100, TURNINGSPEED);

currentDistance = getUSDistance(sonarSensor);

if (currentDistance<previousDistance) {

previousDistance = currentDistance; // it means it is turning toward to target;

approachingTarget = true;

} else if (approachingTarget && currentDistance > previousDistance && previousDistance < targetDistance) {

targetDistance = previousDistance;

return;

}

}

} else if (degreeToTurn <= 0) {

degreeToTurn = -1 \* degreeToTurn;

setMotorTarget(leftMotor, -degreeToTurn, TURNINGSPEED);

setMotorTarget(rightMotor, degreeToTurn, TURNINGSPEED);

while(getMotorEncoder(rightMotor)!=getMotorTarget(rightMotor)&& getTouchValue(leftBumper)!=1 && getTouchValue(rightBumper)!=1) {

setMotorSync(leftMotor, rightMotor, -100, TURNINGSPEED);

currentDistance = getUSDistance(sonarSensor);

if (currentDistance<previousDistance) {

previousDistance = currentDistance; // it means it is turning toward to target;

approachingTarget = true;

} else if (approachingTarget && currentDistance > previousDistance && previousDistance < targetDistance) {

targetDistance = previousDistance;

return;

}

}

}

}

/\*\*

\* the robot scan first 45 degree to its left

\* then 90 degree to its right

\* at last 45 degree to its left

\* in any of these three steps, if it updated the target distance, it interupt turning.

\*/

void scan() {

approachingTarget = false;

int scanDegree = 45;

if(getTouchValue(leftBumper)==1 || getTouchValue(rightBumper)==1){

return;

} else {

turn(-scanDegree, TURNINGSPEED);

if (approachingTarget) {return;}

turn(scanDegree\*2,TURNINGSPEED);

if (approachingTarget) {return;}

turn(-scanDegree, TURNINGSPEED);

}

}

/\*\*

\* It is a helper function

\* bool leftMotorOrRightMotor indicates which wheel you want to turn

\* moterEncoder is the encoder you want to turn

\*/

void turnOneMotor(bool leftMotorOrRightMotor, int motorEncoder)

{ // leftMotorOrRightMotor:

// true, turn RIGHT motor,

// false, turn LEFT motor.

// motorEncoder, is the encoder you want the wheel to turn, positive means forward, negative means backward

resetMotorEncoder(leftMotor);

resetMotorEncoder(rightMotor);

if(leftMotorOrRightMotor) { // RightMotor move while LeftMotor stop

if (motorEncoder > 0) { // move forward

setMotorTarget(rightMotor, motorEncoder, TURNINGSPEED);

while(getMotorEncoder(rightMotor)!=getMotorTarget(rightMotor)) {

setMotorSync(leftMotor, rightMotor, -50, TURNINGSPEED);

}

} else if (motorEncoder<0) { // move backward

setMotorTarget(rightMotor, -motorEncoder, TURNINGSPEED);

while(getMotorEncoder(rightMotor)!= -1\*getMotorTarget(rightMotor)) {

setMotorSync(leftMotor, rightMotor, -50, -TURNINGSPEED);

}

}

} else { // else false, LeftMotor move while RightMotor stop

if (motorEncoder > 0) {

setMotorTarget(leftMotor, motorEncoder, TURNINGSPEED);

while(getMotorEncoder(leftMotor)!=getMotorTarget(leftMotor)) {

setMotorSync(leftMotor, rightMotor, 50, TURNINGSPEED);

}

} else if (motorEncoder < 0) {

setMotorTarget(leftMotor, -motorEncoder, TURNINGSPEED);

while(getMotorEncoder(leftMotor)!= -1\*getMotorTarget(leftMotor)){

setMotorSync(leftMotor, rightMotor, 50, -TURNINGSPEED);

}

}

}

}

/\*\*the robot move with one wheel, meet the boundary and move back

\* it returns the encoder in the turning process, so later it can use

\* the differences from right and left encoder to adjust its position.

\*/

int turnOneMotorUntilMeetBoundary(bool leftOrRightMotor) {

resetMotorEncoder(leftMotor);

resetMotorEncoder(rightMotor);

int saved\_leftBoundaryDegree = 0;

int saved\_rightBoundaryDegree = 0;

if(leftOrRightMotor) {

// turn RightMotor

setMotorTarget(rightMotor, MAX\_BOUNDARY\_EXPECTED, TURNINGSPEED);

while(getMotorEncoder(rightMotor)!=getMotorTarget(rightMotor) && getColorName(colorSensor)==colorBlack) {

setMotorSync(leftMotor, rightMotor, -50, TURNINGSPEED);

}

saved\_leftBoundaryDegree = getMotorEncoder(rightMotor);

turnOneMotor(RIGHT, -1\*saved\_leftBoundaryDegree);

return saved\_leftBoundaryDegree;

} else {

setMotorTarget(leftMotor, MAX\_BOUNDARY\_EXPECTED, TURNINGSPEED);

while(getMotorEncoder(leftMotor)!=getMotorTarget(leftMotor) && getColorName(colorSensor)==colorBlack) {

setMotorSync(leftMotor, rightMotor, 50, TURNINGSPEED);

}

saved\_rightBoundaryDegree = getMotorEncoder(leftMotor);

turnOneMotor(LEFT, -1\*saved\_rightBoundaryDegree);

return saved\_rightBoundaryDegree;

}

}

/\*\*it stops on a black tile, tests the black tile's boundary and adjust its postion

\*based on the difference of the encoder of wheels

\*/

void testBoundary() {

leftBoundaryDegree = turnOneMotorUntilMeetBoundary(RIGHT);

rightBoundaryDegree = turnOneMotorUntilMeetBoundary(LEFT);

differences = fabs(rightBoundaryDegree - leftBoundaryDegree);

if(leftBoundaryDegree>rightBoundaryDegree) {

turnOneMotor(RIGHT, differences/4);

// turn(-differences/20);

} else {

turnOneMotor(LEFT, differences/4);

//turn(differences/20);

}

}

/\*\*keep moving with speed until it encountered a black tile\*/

void moveForwardWithSpeed(int movingSpeed) {

TLegoColors previousColor = getColorName(colorSensor); // it should be black at start

TLegoColors currentColor = getColorName(colorSensor); // it should be black at start

int count = 0;

while(count<1 && getTouchValue(leftBumper)==0 && getTouchValue(rightBumper)==0){

motor[leftMotor] = movingSpeed;

motor[rightMotor] = movingSpeed;

//setMotorSync(leftMotor, rightMotor, 0, movingSpeed);

currentColor = getColorName(colorSensor); // now robot enter the white tile.

if (currentColor == colorWhite) {

previousColor = colorWhite;

} else if(currentColor == colorBlack && previousColor == colorWhite) {

count++;

previousColor = colorBlack;

totalBlackTile++;

playSound(soundUpwardTones);

}

}

motor[leftMotor] = 0;

motor[rightMotor] = 0;

sleep(500);

}

void moveToLine(){

moveForwardWithSpeed(MOVINGSPEED);

turnOneMotor(RIGHT, 330);

}

void moveWithSpeedAndTime(int movingSpeed, int movingTime, int withRatio)

{

setMotorSyncTime(leftMotor, rightMotor, withRatio, movingTime\*1000, movingSpeed);

wait1Msec(movingTime\*1000);

}

task main()

{

//stage 1

moveToLine();

while(totalBlackTile<15) {

if (differences<20 && totalBlackTile<14) {

moveForwardWithSpeed(MOVINGSPEED);

moveForwardWithSpeed(MOVINGSPEED);

testBoundary();

} else {

moveForwardWithSpeed(MOVINGSPEED);

testBoundary();

}

}

// End of Stage 1

// Start of stage 2:

moveWithSpeedAndTime(-1\*MOVINGSPEED, 1, 0);

turnOneMotor(LEFT, 345);

moveWithSpeedAndTime(MOVINGSPEED, 4, 0);

while (getTouchValue(leftBumper)==0&&getTouchValue(rightBumper)==0){

moveForwardWithSpeed(MOVINGSPEED);

scan();

if (targetDistance < 100) {

TURNINGSPEED = 7;

}

}

// After it making contacts with targets

moveWithSpeedAndTime(-1\*MOVINGSPEED, 2, 0); // moving back for 2 seconds

moveWithSpeedAndTime(3\*MOVINGSPEED, 2, 0); // then full speed and push 2 seconds

moveWithSpeedAndTime(-1\*MOVINGSPEED, 1, 0); // moving back for 1 second

moveWithSpeedAndTime(MOVINGSPEED, 2, 100); //turning for 2 seconds

playSound(soundUpwardTones);

wait1Msec(1000);

}