COSC343 Assignment 1 - Report

Reuben Crimp, Henry Barnett

March 29, 2014

1 The Algorithm

1.1 Simple

1. align the robot from the start square onto the first track;
2. travel across 14 black tiles.
3. align the robot for the last run
4. travel across 8 black tiles.
5. align onto the finish square

1.2 Across the black tiles

How we traverse the black tiles, is pretty much the same as everyone else. When on a black tile, compare the arc distance from either side, if the arc distances are imbalanced, then balance them with a small rotation proportional to the imbalance. Then drive straight until we hit another black tile.

2 Problems

2.1 Motor Syncing

We found that syncing the motors together had unintentional side effects, like random jerky rotations. to combat these side effects we reset both motors and waited a few milliseconds between commands, this prevented sequential motor commands conflicting with eachother.

2.2 No Floats

No floating point data type for algebra 3.14159 so we take advantage of the int data types large range i.e (x*4.1889) becomes (x*41889)/10000 not as accurate as floating point arithmetic.

2.3 Ambient lighting

Since the values for white and black can vary depending on the ambient light and the sensor. We used the sensor in ”active mode” so we measure reflectivity. And we don’t try and identify shades of grey, we only check if it’s probably back, or not.

3 The Code

```
  /*
  * Assignment1_v8.nxc - March 29, 2014
  * Reuben, Henry, Leo
  */
#define TURN_SPEED 40
#define STRAIGHT_SPEED 50
#define TURN_LEFT_PIVOT OUT_B
#define TURN_RIGHT_PIVOT OUT_C
#define ARC_INC 5                       //Arc distance incrementor
#define LOCKUP_WAIT 50                  //Wait time between sequential Motor
    commands, to prevent motor lockup

#define ON_TILE(a) (SENSOR_1<50)==a // 50 is the bound between black and white
#define BLACK_TILE 1                    //
#define WHITE_TILE 0                    //

//the following could/should be inline functions, but meh...
#define GO_BACKWARDS OnRevSync(OUT_BC, STRAIGHT_SPEED, 0)
#define GO_FORWARDS OnFwdSync(OUT_BC, STRAIGHT_SPEED, 0)
#define STOP_MOTORS OffEx(OUT_BC, RESET_ALL); Wait(LOCKUP_WAIT)
#define CURRENT_TILE (ON_TILE( BLACK_TILE ) ? BLACK_TILE : WHITE_TILE)

sub move(int cm){
    RotateMotorEx(OUT_BC, STRAIGHT_SPEED, cm*20, 0, true, true);
    STOP_MOTORS;
}
sub donutTurn(int degrees, byte pivot){
    RotateMotor(pivot , TURN_SPEED, degrees * 4);
    STOP_MOTORS;
}

void rotateArcDistance(byte pivot, int arcLength, int speed){
    int currArcLength = 0;

    OnFwd( pivot, speed);
    while(currArcLength <= arcLength){
        Wait(ARC_INC);
        currArcLength++;
    }
    STOP_MOTORS;
}
int countArcDistance(byte pivot, int currentTile, int speed){
    int arcLength = 0;

    OnFwd( pivot, speed);
    while( ON_TILE ( currentTile ) ){
        Wait(ARC_INC);
        arcLength++;
    }
    STOP_MOTORS;
    return arcLength;
}

void alignOnTile(int currentTile ){
    int leftArcDist, rightArcDist, correctionArcDist;
    byte correctionPivot;

    leftArcDist = countArcDistance(TURN_LEFT_PIVOT, currentTile, TURN_SPEED);
    rotateArcDistance(TURN_LEFT_PIVOT, leftArcDist, -TURN_SPEED);
    rightArcDist = countArcDistance(TURN_RIGHT_PIVOT, currentTile, TURN_SPEED);
    rotateArcDistance(TURN_RIGHT_PIVOT, rightArcDist, -TURN_SPEED);

    if (rightArcDist > leftArcDist){
        correctionArcDist = rightArcDist - leftArcDist;
        correctionPivot = TURN_LEFT_PIVOT;
    }else{
        correctionArcDist = leftArcDist - rightArcDist;
        correctionPivot = TURN_RIGHT_PIVOT;
    }
    rotateArcDistance(correctionPivot, (correctionArcDist * 3 )/ 10 , -TURN_SPEED);
}
void crossTiles(int tileColour, int tileLimit){
     int tileCount = 0;
     while(tileCount < tileLimit){
         if (CURRENT_TILE == tileColour){
            PlayTone(500, 400);
            tileCount++;
            alignOnTile( BLACK_TILE );
            GO_FORWARDS; until( ON_TILE( WHITE_TILE ) );
        } else {
            GO_FORWARDS; until( ON_TILE ( BLACK_TILE ) );
            Wait(100); STOP_MOTORS;
        }
    }
}

inline void alignForEnd(){
    move(14); STOP_MOTORS;
    donutTurn(90, TURN_RIGHT_PIVOT); STOP_MOTORS;
    Wait(1000);
    OnFwdSync(OUT_BC, 100, 100);
    Wait(5000);
}
inline void alignForMiddle(){
    GO_FORWARDS; until( ON_TILE( BLACK_TILE ) ); STOP_MOTORS;
    move(5);
    Wait(100);
    donutTurn(90, TURN_RIGHT_PIVOT);

    GO_BACKWARDS; until( ON_TILE( BLACK_TILE ) );
    GO_BACKWARDS; until( ON_TILE( WHITE_TILE ) ); STOP_MOTORS;
    GO_FORWARDS; until( ON_TILE( BLACK_TILE ) ); STOP_MOTORS;
}
```

```
inline void alignForStart(){
     int currentTile = CURRENT_TILE;
     GO_FORWARDS; until( ON_TILE( WHITE_TILE ) );
     GO_FORWARDS; until( ON_TILE( BLACK_TILE ) );
     PlayTone(500, 400);
     move(4);
     donutTurn(90, TURN_RIGHT_PIVOT);
}

task main(){
    SetSensorLight(IN_1, true);

    alignForStart();
    crossTiles(BLACK_TILE, 13);
    alignForMiddle();
    crossTiles(BLACK_TILE, 8);
    alignForEnd();
}
```