#include <pololu/3pi.h>

#include "follow-segment.h"

#include "turn.h"

char path[100] = "";

unsigned char path\_length = 0; // the length of the path

// Displays the current path on the LCD, using two rows if necessary.

void display\_path()

{

// Set the last character of the path to a 0 so that the print()

// function can find the end of the string. This is how strings

// are normally terminated in C.

path[path\_length] = 0;

clear();

print(path);

if(path\_length > 8)

{

lcd\_goto\_xy(0,1);

print(path+8);

}

}

// This function decides which way to turn during the learning phase of

// maze solving. It uses the variables found\_left, found\_straight, and

// found\_right, which indicate whether there is an exit in each of the

// three directions, applying the "left hand on the wall" strategy.

char select\_turn(unsigned char found\_left, unsigned char found\_straight, unsigned char found\_right)

{

// Make a decision about how to turn. The following code

// implements a left-hand-on-the-wall strategy, where we always

// turn as far to the left as possible.

if(found\_left)

return 'L';

else if(found\_straight)

return 'S';

else if(found\_right)

return 'R';

else

return 'B';

}

// Path simplification. The strategy is that whenever we encounter a

// sequence xBx, we can simplify it by cutting out the dead end. For

// example, LBL -> S, because a single S bypasses the dead end

// represented by LBL.

void simplify\_path()

{

// only simplify the path if the second-to-last turn was a 'B'

if(path\_length < 3 || path[path\_length-2] != 'B')

return;

int total\_angle = 0;

int i;

for(i=1;i<=3;i++)

{

switch(path[path\_length-i])

{

case 'R':

total\_angle += 90;

break;

case 'L':

total\_angle += 270;

break;

case 'B':

total\_angle += 180;

break;

}

}

// Get the angle as a number between 0 and 360 degrees.

total\_angle = total\_angle % 360;

// Replace all of those turns with a single one.

switch(total\_angle)

{

case 0:

path[path\_length - 3] = 'S';

break;

case 90:

path[path\_length - 3] = 'R';

break;

case 180:

path[path\_length - 3] = 'B';

break;

case 270:

path[path\_length - 3] = 'L';

break;

}

// The path is now two steps shorter.

path\_length -= 2;

}

// This function is called once, from main.c.

void maze\_solve()

{

// Loop until we have solved the maze.

while(1)

{

// FIRST MAIN LOOP BODY

follow\_segment();

// Drive straight a bit. This helps us in case we entered the

// intersection at an angle.

// Note that we are slowing down - this prevents the robot

// from tipping forward too much.

set\_motors(50,50);

delay\_ms(50);

// These variables record whether the robot has seen a line to the

// left, straight ahead, and right, whil examining the current

// intersection.

unsigned char found\_left=0;

unsigned char found\_straight=0;

unsigned char found\_right=0;

unsigned char junction\_past=0;

// Now read the sensors and check the intersection type.

unsigned int sensors[5];

read\_line(sensors,IR\_EMITTERS\_ON);

// Check for left and right exits.

if(sensors[0] > 100)

{

found\_left = 0;

clear();

print("Junction");

delay\_ms(300);

junction\_past=1;

follow\_segment();

}

if(sensors[4] > 100)

{

found\_right = 0;

clear();

print("Junction");

delay\_ms(300);

junction\_past=1;

follow\_segment();

}

// Drive straight a bit more - this is enough to line up our

// wheels with the intersection.

set\_motors(40,40);

delay\_ms(200);

// Check for a straight exit.

read\_line(sensors,IR\_EMITTERS\_ON);

if(sensors[1] > 200 || sensors[2] > 200 || sensors[3] > 200)

found\_straight = 1;

// Check for the ending spot.

// If all three middle sensors are on dark black, we have

// solved the maze.

if(sensors [0] > 600 && sensors[1] > 600 && sensors[2] > 600 && sensors[3] > 600 && sensors[4] > 600);

{

if(junction\_past==1);

{

set\_motors(0,0);

clear();

print("Finished!");

}

break;

}

// Intersection identification is complete.

// If the maze has been solved, we can follow the existing

// path. Otherwise, we need to learn the solution.

unsigned char dir = select\_turn(found\_left, found\_straight, found\_right);

// Make the turn indicated by the path.

turn(dir);

// Store the intersection in the path variable.

path[path\_length] = dir;

path\_length ++;

// You should check to make sure that the path\_length does not

// exceed the bounds of the array. We'll ignore that in this

// example.

// Simplify the learned path.

simplify\_path();

// Display the path on the LCD.

display\_path();

}

// Solved the maze!

// Now enter an infinite loop - we can re-run the maze as many

// times as we want to.

/\* while(1)

{

// Beep to show that we finished the maze.

set\_motors(0,0);

play(">>a32");

// Wait for the user to press a button, while displaying

// the solution.

while(!button\_is\_pressed(BUTTON\_B))

{

if(get\_ms() % 2000 < 1000)

{

clear();

print("Solved!");

lcd\_goto\_xy(0,1);

print("Press B");

}

else

display\_path();

delay\_ms(30);

}

while(button\_is\_pressed(BUTTON\_B));

delay\_ms(1000);

// Re-run the maze. It's not necessary to identify the

// intersections, so this loop is really simple.

int i;

for(i=0;i<path\_length;i++)

{

// SECOND MAIN LOOP BODY

follow\_segment();

// Drive straight while slowing down, as before.

set\_motors(50,50);

delay\_ms(50);

set\_motors(40,40);

delay\_ms(200);

// Make a turn according to the instruction stored in

// path[i].

turn(path[i]);

}

// Follow the last segment up to the finish.

follow\_segment();

// Now we should be at the finish! Restart the loop.

} \*/

}

// Local Variables: \*\*

// mode: C \*\*

// c-basic-offset: 4 \*\*

// tab-width: 4 \*\*

// indent-tabs-mode: t \*\*

// end: \*\*