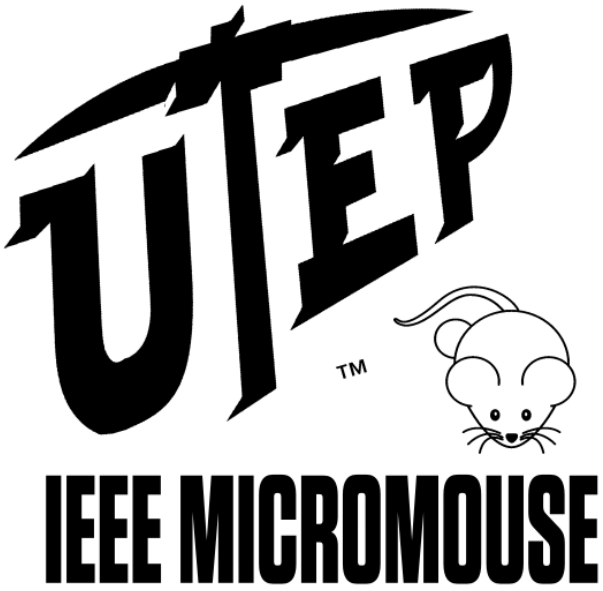

IEEE MICROMOUSE

Workshop Week 12:

Scan walls, best neighbor, and turn to neighbor

Objective:

Finish the micromouse code with the final algorithm to reach the center of the maze by completing the missing code, such as scanning for walls, finding the best neighbor and turning towards the best neighbor.

Background information:

This week we will look at the final portions of code to solve the maze, thus completing the micromouse project. Last workshop, you were able to find the best path possible to the center of the maze by using the flood fill algorithm. The exercise for the last workshop was to manually move the mouse around the maze. Today we will automate the process by writing a set of functions that will find the best neighbor (open neighbor with lowest cell value), turn towards the best neighbor and scan the maze for new walls.

Before tackling those functions, let us start by thinking about the main and final algorithm to solve the maze. First, we want to run the program until we reach the 0 valued cell. Second, we want to always orient and move towards the neighbor that is open and has the lowest cell value. Finally, the mouse must update its position and orientation on the maze as well as physically on the real maze. In addition, the mouse must add walls to the maze as they are encountered.

With those requirements in mind, here is the final algorithm to solve the maze:

```
1.  void loop()
2.  {
3.    /*Get the initial position and heading of the mouse*/
4.    int8_t x = mouse_get_x();
5.    int8_t y = mouse_get_y();
6.    uint8_t heading = mouse_get_heading();
7.
8.    /*Run the algorithm until we reached the zero value cell (Center of the maze)*/
9.    do
10.   {
11.      /*Wait for button press or serial console for debugging (Not needed)*/
12.      while(digitalRead(BUTTON) != HIGH);
13.
14.      /*Scan walls*/
15.      scan_walls(x, y, heading);
16.
17.      /*Update maze using floodfill*/
18.      flood_fill(x, y);
19.
20.      /*Get the best neighbor*/
21.      uint8_t best_neighbor = find_best_neighbor(x, y);
22.
23.      /*Orient mouse towards best neighbor*/
24.      turn_best_neighor(best_neighbor);
25.
26.      /*Move forward*/
27.      move_forward(x, y, best_neighbor);
28.
29.      /*Get the position and heading of the mouse*/
30.      x = mouse_get_x();
```

```
31.       y = mouse_get_y();
32.       heading = mouse_get_heading();
33.
34.       /*Print maze, only for debugging (Not needed)*/
35.       maze_print();
36.    }while(maze_get_value(x, y) != 0);
37.
38.    /*Maze solved trap the CPU*/
39.    while(1);
40. }
41.
```

As you can observe, we are placing the final portion of code in the loop function. This is not required, because once we reached the center of the maze, we want to finish our program. You might want to play around and explore on what to do next after we find the center of the maze. One thing that you might consider is to do a second run with the maze already mapped out and with a valid path to center, but this time the mouse does it as fast as possible.

Scan walls

To scan the walls, the mouse simply must take a reading on all sensors and check if the measurements are greater than the wall thresholds.

Now it is easy to think that if the right or left sensor measurements is greater than a threshold, then we simply add a wall to the EAST or WEST. This in fact will give you the wrong result because the orientation of the mouse is not always aligned to the NORTH of the maze and the walls are relative to it. Here is a visual example of why this happens:

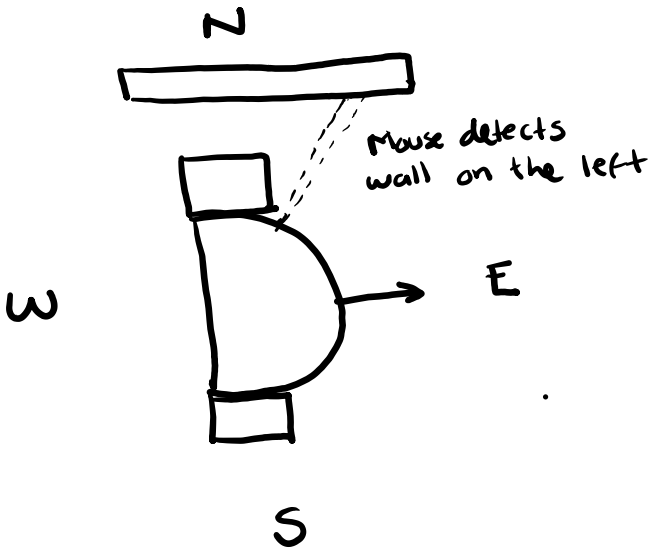

Figure 1 Adding walls to the maze depends on the orientation of the mouse. The mouse is heading EAST, but we detected a wall on our left. If we simply add the left sensor as an EAST wall, then we will get the wrong wall placement

Looking at the image above, you can see that the mouse detected a wall to the left of its sensor. You might be tempted to add the WEST wall of the current cell, but that will be wrong because of the orientation of the mouse. The correct answer is to add a wall to the NORTH. To solve this problem, we are going to shift the orientation of the wall discovered by 1 or 3 depending on the heading of the mouse. So, for the image above, we simply shift the current heading of the mouse heading by 3, so we move from EAST to SOUTH to WEST and the finally to NORTH, which is the correct position of the discovered wall. if we discover a wall to our right, we take the same approach but this time we only shift the wall orientation by one. This is the code to add a wall when the left IR sensor measurement is greater than a threshold:

```
1.    /*Threshold for left sensor*/
2.    if(ir2 > IR_LEFT_WALL_THRESHOLD)
3.    {
4.      /*There is a wall on our right*/
5.      /*Bit shift heading for correct wall positioning*/
6.      uint8_t wall = heading << 3;
7.      /*If bigger than 16, then wrap around back to 1*/
8.      if(wall > 8)
9.      {
10.       wall /= 16;
11.     }
12.
13.     maze_set_wall(x, y, wall);
14.   }
15.
```

Notice that we need to wrap around the wall if it ever gets bigger than 8, this is due to the internal implementation of the maze, and it should be added to your code.

The final code to scan for walls will look like this:

```
1.  const int IR_FRONT_WALL_THRESHOLD = 700;
2.  const int IR_RIGHT_WALL_THRESHOLD = 700;
3.  const int IR_LEFT_WALL_THRESHOLD = 700;
4.
5.  void scan_walls(int x, int y, uint8_t heading)
6.  {
7.    /*Measure all IR sensors*/
8.    int ir0 = ir_0_read(150);
9.    int ir1 = ir_1_read(150);
10.   int ir2 = ir_2_read(150);
11.   int ir3 = ir_3_read(150);
12.
13.   /*Threshold for front sensors*/
14.   int ir_front = (ir0 + ir3)/2;
15.   if(ir_front > IR_FRONT_WALL_THRESHOLD)
16.   {
17.     /*There is a wall at the front*/
18.     /*Just set the wall to the current heading of the mouse*/
19.     maze_set_wall(x, y, heading);
20.   }
21.
22.   /*Threshold for right sensor*/
23.   if(ir1 > IR_RIGHT_WALL_THRESHOLD)
24.   {
25.     /*There is a wall on our left*/
```

```
26.     /*Bit shift heading for correct wall positioning*/
27.     uint8_t wall = heading << 1;
28.     /*If bigger than 16, then wrap around back to 1*/
29.     if(wall > 8)
30.     {
31.       wall /= 16;
32.     }
33.
34.     maze_set_wall(x, y, wall);
35.   }
36.
37.   /*Threshold for left sensor*/
38.   if(ir2 > IR_LEFT_WALL_THRESHOLD)
39.   {
40.     /*There is a wall on our right*/
41.     /*Bit shift heading for correct wall positioning*/
42.     uint8_t wall = heading << 3;
43.     /*If bigger than 16, then wrap around back to 1*/
44.     if(wall > 8)
45.     {
46.       wall /= 16;
47.     }
48.
49.     maze_set_wall(x, y, wall);
50.   }
51. }
52.
```

Find best neighbor

Next up is finding the best neighbor. The best neighbor is always going to be the open cell with the lowest cell value. If you look at the flood fill algorithm, you can observe that there is a section that also tries to find the neighbor with the lowest minimum value. So, we can simply repeat that implementation as follows:

```
1.  uint8_t find_best_neighbor(int x, int y)
2.  {
3.    /*Neighbor offsets for all orientations*/
4.    int8_t neighbor_offset[4][2] =
5.    {
6.      {0, -1},
7.      {1,  0},
8.      {0,  1},
9.      {-1, 0},
10.   };
11.
12.   /*Get current cell*/
13.   cell_t *current = maze_get_cell(x, y);
14.   cell_t *neighbor_cell;
15.
16.   /*Assume lowest value is the highest distance*/
17.   uint8_t lowest_value = 255;
18.   /*Assume best neighbor is to the north*/
19.   uint8_t best_neighbor = NORTH;
20.   for(int i = 0; i < 4; i++)
21.   {
22.     /*If the neighbor is open...*/
23.     if((current->walls & (1 << i)) == 0)
```

```
24.     {
25.       /*Get neighbor coordinates*/
26.       int8_t nx = x + neighbor_offset[i][0];
27.       int8_t ny = y + neighbor_offset[i][1];
28.
29.       /*Get neighbor cell value*/
30.       neighbor_cell = maze_get_cell(nx, ny);
31.       if(neighbor_cell->value < lowest_value)
32.       {
33.         /*If this this lower than the previous cell value, then set it as
34.          * the new minimum
35.          */
36.         lowest_value = neighbor_cell->value;
37.         best_neighbor = (1 << i);
38.       }
39.     }
40.   }
41.
42.   return best_neighbor;
43. }
44.
```

Observe that we always assume that the best neighbor is to the north. We do this so that if we have 2 neighbors of the same value, we always select the one that does not make the mouse turn.

Turn towards best neighbor

The next function we need to tackle is turning towards the best neighbor. The input of this function is the desired heading, and the output will obviously be the mouse physically turning and the maze updating the heading of the mouse. To do this we must consider the current heading of the mouse and the desired heading, which will come in the form of NORTH, EAST, SOUTH and WEST. Once we know this, we can now physically turn the mouse +90.0, -90.0 or 180.0 degrees.

To know the angle the mouse has to turn, we must create a truth table as follows:

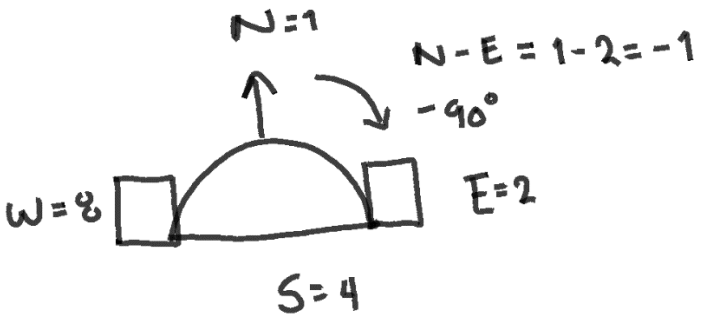

Figure 2 Basic idea on how we find how the angle and direction of rotation

Heading	New Heading	Heading - New Heading	Result
1	1	0	0°
1	2	-1	-90°
1	4	-3	180°
1	8	-7	90°
2	1	1	90°
2	2	0	0°
2	4	-2	-90°
2	8	-6	180°
4	1	3	180°
4	2	2	90°
4	4	0	0°
4	8	-4	-90°
8	1	7	-90°
8	2	6	180°
8	4	4	90°
8	8	0	0°

Figure 3 Truth table to turn the mouse based on current and desired heading

Using the table as our guide, we can create a switch statement to turn the mouse based on the result of the heading minus the new heading. The code looks as follows:

```
1.  void turn_best_neighor(uint8_t best_neighbor)
2.  {
3.    /*Get current heading*/
4.    uint8_t heading = mouse_get_heading();
5.    /*Relative positioning to best neighbor*/
6.    int8_t delta = heading - best_neighbor;
7.
8.    /*Truth table to find how much we need to turn*/
9.    switch(delta)
10.   {
11.     case 7: case -1: case -2: case -4:
12.       /*Physically turn the mouse*/
13.       control_turn(-90.0);
14.       break;
15.     case -7: case 1: case 2: case 4:
16.       /*Physically turn the mouse*/
17.       control_turn(90.0);
18.       break;
19.     case 3: case -3: case 6: case -6:
20.       /*Physically turn the mouse*/
21.       control_turn(180.0);
22.       break;
23.   }
24.
25.   /*Update heading mouse*/
26.   mouse_set_heading(best_neighbor);
27. }
28.
```

Notice that we are using the ***control_turn*** function from workshop 9. If you don't have this function, please go back to and complete this workshop.

Move forward

Lastly, we need to move the mouse forward AFTER we orient the mouse towards the best neighbor. As with the ***turn_best_neighbor*** function, we first physically move the mouse, then update its position on the maze. The function to do this can be coded as follows:

```
1.  void move_forward(int8_t x, int8_t y, uint8_t heading)
2.  {
3.    /*Physically move the mouse forward*/
4.    control_forward();
5.
6.    /*figure out where we need to move the mouse on the map*/
7.    switch(heading)
8.    {
9.      case NORTH:
10.       y--;
11.       break;
12.     case EAST:
13.       x++;
14.       break;
15.     case SOUTH:
16.       y++;
17.       break;
18.     case WEST:
19.       x--;
20.       break;
21.   }
22.
23.   /*Set the mouse new position on the map*/
24.   mouse_set_x(x);
25.   mouse_set_y(y);
26. }
27.
```

Look at how we move the mouse based on the heading. Depending on the orientation of the mouse, if we move forward one cell, then we either increment or decrement the x or y position of the mouse.

Finally, observe that we make use of the ***control_forward*** function, which was introduced in workshop 8. Again, if you don't have this function, please read the workshop and complete it.

Exercise:

Complete the micromouse project by writing the ***scan_walls***, ***find_best_neighbor***, ***turn_best_neighbor*** and ***move_forward*** functions into your main sketch. Use the code template below.

```
1.  #include "micromouse.h"
2.  #include "maze.h"
3.
4.  uint8_t find_best_neighbor(int x, int y)
5.  {
6.  }
7.
8.  void scan_walls(int x, int y, uint8_t heading)
9.  {
10. }
11.
12. void turn_best_neighbor(uint8_t best_neighbor)
13. {
14. }
15.
16. void move_forward(int8_t x, int8_t y, uint8_t heading)
17. {
18. }
19.
20. void setup()
21. {
22.   /*Enable serial port for bluetooth*/
23.   Serial.begin(9600);
24.
25.   /*Initialize maze*/
26.   maze_init(MAZE_WIDTH/2, MAZE_HEIGHT/2);
27.   mouse_set_x(0);
28.   mouse_set_y(MAZE_HEIGHT - 1);
29. }
30.
31. void loop()
32. {
33.   /*Get the initial position and heading of the mouse*/
34.   int8_t x = mouse_get_x();
35.   int8_t y = mouse_get_y();
36.   uint8_t heading = mouse_get_heading();
37.
38.   /*Run the algorithm until we reached the zero value cell (Center of the maze)*/
39.   do
40.   {
41.      /*Wait for button press or serial console for debugging (Not needed)*/
42.      while(digitalRead(BUTTON) != HIGH);
43.
44.      /*Scan walls*/
45.      //scan_walls(x, y, heading);
46.
47.      /*Update maze using floodfill*/
48.      flood_fill(x, y);
49.
50.      /*Get the best neighbor*/
51.      uint8_t best_neighbor = find_best_neighbor(x, y);
52.
53.      /*Orient mouse towards best neighbor*/
54.      turn_best_neighor(best_neighbor);
55.
56.      /*Move forward*/
```

```
57.       move_forward(x, y, best_neighbor);
58.
59.       /*Get the position and heading of the mouse*/
60.       x = mouse_get_x();
61.       y = mouse_get_y();
62.       heading = mouse_get_heading();
63.
64.       /*Print maze, only for debugging (Not needed)*/
65.       maze_print();
66.    }while(maze_get_value(x, y) != 0);
67.
68.    /*Maze solved trap the CPU*/
69.    while(1);
70. }
71.
```