In the previous lesson, you learned how to make the robot follow the right wall. Now you will add coding to make the robot turn at corners.

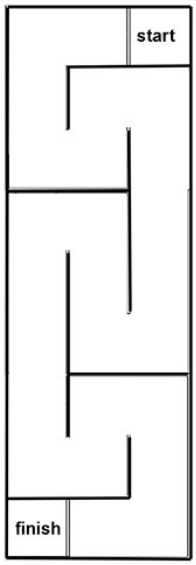

Figure 1 National Robotics Challenge maze - elementary and middle school contestants

Let us look at the maze plan in Figure 1. Imagine your robot at the start and then proceeding to travel through the maze. It is not long before a wall in front blocks the robot's progress. The robot must turn left to continue.

Your current code does not include a method for turning left. You have two Ping sensors mounted on the robot. One of these can determine when the robot must turn left, but which one? The forward-facing sensor could do the job, but not the right-facing sensor. Why will the right-facing sensor not work? As the robot encounters a wall in front of it, there is no significant event on the right to signal a left turn. However, a significant event happens in front of the robot. That event is the wall blocking forward progress. When the robot approaches a wall in front of it, and there is a wall on the right side, then the robot must turn left.

For the first part of this lesson, you will not be testing your robot in the maze. Instead, use the temporary walls placed on the floor of the meeting room to replicate the structure of the maze at the first turn.

There are two approaches for determining the need for a turn (which does not mean there are only two). One approach would be to track the distance the robot has traveled since the start. If you know the distance from the starting point to the left turn, then you could write code that would cause the robot to turn once it has traveled a sufficient distance. With this approach, the robot would NOT be relying on a Ping sensor to determine when it must turn. In the second approach, the robot would use a Ping sensor to determine when it has reached a certain distance to the blocking front wall.

Approach 1 – tracking the distance to travel before the turn

There are two drive functions that track distance traveled by the robot: **drive_goto()** and **drive_getTicks()**. Another way to travel a specified distance is to make the robot drive at a certain speed for a certain amount of time. The **drive_ramp()** function specifies speed without a specification for time. But you can use **drive_ramp()** followed by a **pause()** function to make the robot travel a specific speed for a specific amount of time.

What about **drive_goto()**? It works for making turns, but is problematic if used as the drive method in the wall following procedure. It is much easier to code with **drive_ramp()** as the drive method for wall following. Since **drive_goto()** is not a good option, that leaves us with these:
1) use **drive_ramp()** with a **pause ()** function
2) use **drive_ramp()** with a **drive_getTicks()** function

In some cases **pause ()** is easier to code than **drive_getTicks()**, but **drive_getTicks()** is more accurate. In the case where you are using a wall following procedure, using either **pause ()** or **drive_getTicks()** is complicated. So let us save these options for another lesson and concentrate on Approach 2 for now.

Approach 2 – using the sensor to detect the front wall

When the robot is executing a wall following procedure, a sensor is a good option for detecting the location of a turn. Try to imagine the robot following along the first wall to the right. A Ping sensor points at the wall. As the robot travels, every 20 milliseconds it does a distance measurement with the Ping sensor and makes corrective steering if it is too far or too close to the wall. As the robot approaches the wall in front at the first turn, the front sensor determines when to execute the turn. How would you do that?

You have already used an **if – else if – else** structure for creating the wall following part of your code. You could expand that to an **if – else if – else if – else** structure. The additional element in the structure would be for the code for detecting the front wall.

Below is the code of the while loop for the wall following lesson appended to include the left turn.

```
while(1)
   {
   int wallDistR = ping_mm(17); // take measurement to right wall
   int wallDistF = ping_mm(16);  //take measurement to front wall
   if(wallDistF < ___)           //robot turn left at corner
      {
         drive_ramp(____, 64);
         pause(____);
      }
   else if(wallDistR < 118)      //turn slightly left
      {
         drive_ramp(_____, 64);
      }
   else if(wallDistR > 128)      //turn slightly right
      {
         drive_ramp(64, ____);
      }
   else                          //drive straight forward l
      {
         drive_ramp(64, 64);
      }
   pause(20);
   }                            }
```

Notice the new line:

```
int wallDistF = ping_mm(16);
```

The front ping sensor is attached to pin 16 and this code line causes this ping sensor to take a measurement to the front wall and store it in the variable named **wallDistF**.

Notice the new line:

```
if(wallDistF < ___)
```

I left the number in the argument blank because I want you to discover which numbers work well.

Suppose you want the robot to start turning when it is closer than 1000 mm to the front wall. Then the proper code is:

```
if(wallDistF < 1000) //turn when robot is closer than 1000 mm
```

The value of 1000 is obviously too large, I just used it as an example. In any case, when the robot comes closer to the front wall than the value, then the code lines inside the curly braces {} below the **if()** statement will be executed. I would suggest that you place the robot at the location in the maze where you want it to start turning left. Then measure the distance from the front sensor to the front wall. Place that value in the code line above as your starting point.

The next code line specifies the left turn. The drive command for turning is:

```
drive_ramp(____, 64);
```

Here again I have not supplied one number for you. That number is the speed of the left wheel. Since the robot must turn left, the left wheel must turn slower than the right wheel (i.e. less than 64).

After the drive_ramp() for turning left, there is a pause. The length of the pause will determine how long the robot will turn left before the while loop will continue. Adjust this value until the robot turns about 90 degrees.

Experiment with various values in the code until your robot makes a good left turn at the first intersection. When you have access to the maze, test your robot again to see if it will properly negotiate the next two turns to the left (combined they are a U-turn, or 180 degree turn). If the robot does not negotiate the first three turns properly, you will need to alter your code. Watch the robot carefully as it moves through the maze and try to identify any navigation errors. How can you fix those? Ask for an advisor for help if you are stuck.

You may also want to look at the papers linked below that cover some mathematical aspects of turns with the robot:

http://lafavre.us/robotics/C_language_ActivityBot.pdf

http://lafavre.us/robotics/Work_sheet_driving_ActivityBot.pdf

After you have your robot properly negotiating the first three left turns, you will move on to the next lesson, making the robot turn right.