Making ActivityBot360 Follow a Wall

J. La Favre
May 21, 2025

Your project for this year will be to program the ActivityBot360 to solve the National Robotics Challenge Robot Maze (Figure 1). The maze is composed of 50 cells, each 25.4 cm by 25.4 cm. The walls are 1.9 cm thick and 7.6 cm high. Therefore, the lane width is 23.5 cm. The robot must execute thirteen 90 degree turns, seven right and six left. Your robot is equipped with two distance sensors (Ping sensors), one pointing forward and one pointing to the right. The Ping sensors are used to assist in determining when to turn and in which direction.

Figure 1: Middle School Maze, 2025 NRC

In addition to making turns, the robot must be able to keep roughly centered in lanes. Otherwise, it will crash into the walls, which should be avoided. You might think that if you point the robot in an exact forward direction, it will travel down the center of the lane. But this is not the case, especially if the robot must travel down a relatively long lane before turning. A common solution is to provide wall-following code in your program, which is the subject of this lesson.

Wall-following code causes the robot to travel parallel to a wall, using a sensor to maintain a specific distance from the wall. Since your robot is equipped with a sensor pointing to the right, you will program your robot to maintain a given distance from the wall on the right side of the robot.

A long lane can be created in the maze for testing your program. When you are successful in making your robot follow the wall at roughly a specified separation distance, without crashing into the wall or moving too far away from the wall, you will have achieved your first step in an effort to make the robot solve the maze.

Your robot will use the two Ping sensors for measuring distances between the robot and walls. One sensor points to the right side of the robot and is connected to port 16 of the microcontroller board. The second sensor points forward and is connected to port 17.

Start Writing Your Code!

You should now start working on your maze code. Open the SimpleIDE software and click the new project button . Save your new project with the file name **follow right wall** in the **My Projects** folder. Then copy the code on the next page to your code file and save.

```
#include "simpletools.h"
#include "ping.h"
#include "abdrive360.h"

int main()
   { // start of main()code block
     // Put your code here to make the robot follow the right wall
   } // end of main() code block
```

All of the remaining code you will write in this lesson must be placed between the start brace **{** and end brace **}** of the main() code block. Let me provide some help with the remainder of the code.

First, you need to include a function that will make the robot do distance measurements with the right-facing Ping sensor. In addition, you should store the distance value in a variable. Suppose we devise the name **wallDistR** as the variable name to store the Ping measurements to the right wall. In addition, this Ping sensor should be connected to port 16 of the microcontroller board. Here is a proper code statement:

int wallDistR = ping_cm(16);

Add the above line of code to your code file on the next line below the open brace **{** of the main() code block. Then save your file.

The above code could be translated to pseudocode like this: *"make the Ping sensor connected to port 16 send out a burst of sound and then determine the amount of time in microseconds for the sound to return. Then convert the time measurement to a distance in centimeters."*

The distance to the right wall in centimeters has been stored in the variable named **wallDistR**. This is a very important piece of information, which is needed if you want the robot to follow the wall at a specific distance of separation.

Now we need to select a distance of separation between robot sensor and wall. **For this lesson, let's try to make the robot follow the right wall with a sensor separation distance of 6 to 10 cm.** How can you write the code to make that happen?

First, use the function **drive_speed()** for driving the robot. Remember that **drive_speed()** takes two arguments inside the parentheses. The first argument is the speed of rotation for the left wheel and the second argument is the speed of rotation for the right wheel. The maximum speed allowed is 128. For this lesson you will use a maximum speed of 64. To make the robot move forward in a straight line, both wheels should spin at the same speed. Code like this:

drive_speed(64, 64); //drive forward at a speed of 64 ticks per second

You will need to make the robot turn slightly left or slightly right if the robot is too close or too far from the wall. To make the robot turn left, the right wheel should spin faster:

drive_speed(_____, 64); //turn slightly left, slow wheel speed not provided

To make the robot turn right, the left wheel should spin faster:

drive_speed(64, _____); //turn slightly right, slow wheel speed not provided

Notice that I have left the slow wheel speed blank in the above code examples. **I want you to experiment with different values for the slower wheel to find a value that works well** (you will do that later in this lesson).

So far you have learned how to store the distance to the right wall in a variable. You have also reviewed how to make the robot move straight forward and how to turn slightly. Now, let's consider some logic for making the robot follow the wall at a specified distance of separation (6 to 10 cm).

Let us suppose that you set your robot down next to the wall carefully, making sure it is 8 cm from the sensor to the wall. You do your best to point the robot in a direction that will parallel the wall. However, of course, as the robot attempts to run parallel to the wall, it will most likely start either to move a little closer to the wall or move a little farther away from the wall. Since the robot is in the maze, then in either case it will eventually crash into a wall. You must devise some method of corrective steering when the robot wanders away from the desired wall separation distance. If the robot is too far away from the wall, it must make a steering adjustment to move toward the wall. If the robot is too close to the wall, it must steer away from the wall. This is easy to state, but more difficult to code for in C language.

There are three situations for the robot: **1)** it is too close to the wall, **2)** it is too far away from the wall or **3)** it is within the correct separation range from the wall. Your code must include a method for making a decision between the three different cases. This can be done with an **if – else if – else** sequence. Here is the pseudocode for that structure:

if the robot is too close to the wall
steer the robot away from the wall
else if the robot is too far away from the wall
steer the robot toward the wall
else the robot is within the correct range from the wall
steer straight forward (both wheels with same speed)

Let us review exactly how an **if – else if – else** structure must be coded (on next page):

```
if (put your condition here)
  {
    drive_speed(put wheel speeds here);
  }
else if (put your condition here)
  {
    drive_speed(put wheel speeds here);
  }
else // no condition needed here, if above two conditions not
     // met, then this will happen
  {
    drive_speed(put wheel speeds here);
  }
```

Now you are wondering what the conditions should be inside the parentheses, right? Okay, I will try to help with that without giving you everything you need to produce the complete code. We have selected a sensor distance range of 6 to 10 cm from the wall. Then once the robot is outside these limits, steering corrections must be applied. Essentially, you are posing questions inside the parentheses. For the if() line, you may wish to ask: “is the robot too close to the wall?” How would you code for that? You could use the **less than** symbol (**<**). In other words, is the distance to the wall less than 6 cm? The code below poses the question properly in C language:

if(wallDistR < 6)

Remember that we decided to store the distance determined by the sensor in the variable named **wallDistR**. Therefore, the question being asked is this: “is the value stored in **wallDistR** less than 6?” If this condition evaluates to true, then code placed directly below the i**f()** will be executed. The drive code placed within the braces **{ }** below the **if()** line will be executed. That drive code should make the robot steer away from the wall. Also keep in mind that this symbol (**>**) means **greater than**. So use that when you want to test if a variable stores a value greater than a specified number (hint: greater than 10).

You now have enough information to code the necessary lines of an **if-else if-else** structure. Go ahead and write that code and place it inside the main() function below the **int wallDistR = ping_cm(16);** line. You will need to make a guess for the values to be used for the slower wheel in the code where the robot turns (hint: make the value a little less than 64, but I am not going to tell you just how little).

When you are finished writing your code, **SAVE** it. Now ask an advisor to check your code. If you are thinking you are done with your code, you are not quite right. Assuming you wrote your code properly to this point, you have a couple of items to finish before the code is complete. As your code exists now, the robot will run through each code line only one time. That means it will take one distance measurement, and follow one drive command. That does not constitute a control system. The robot needs to keep taking distance measurements as it moves forward and make corrective steering as needed. The microcontroller needs to loop through the lines of code repeatedly. First, the robot makes a distance measurement, then

decides what steering correction must be done, then runs for a short period, then repeats the process. How can you accomplish that in code? Remember that a **while()** loop will cause code to repeat. If you want it to repeat as long as the program runs, then code it with an argument of 1 like this: **while(1)**. Here is the proper structure for a **while(1)** loop:

```
int main()
  { //start of main() code block
    while(1)
      { //start of while(1) code block
        // Put your distance measurement and drive code here
      } //end of while(1) code block
  } //end of main() code block
```

The above code will cause your lines of code inside the **while(1)** block to repeat as long as the program runs. You can replace the argument of **1** inside the parentheses so that the loop runs only under certain conditions, but for now you just want the loop to run all the time.

Now add the **while(1)** line to your code and its start and end braces **{ },** enclosing all measurement and drive code lines.

There is just one element left that is needed to complete your code for making the robot follow the right wall. You need to use a **pause()** function at the bottom of your code, the line just above the end brace of the **while(1)** block. If you don't do this, you will cause the robot to do excessive distance measurements and calculations. By inserting the pause, the microprocessor pauses for a short time before continuing on to make the next distance measurement and applying any needed corrective steering. A good time to pause is about 20 milliseconds, which is coded like this:

pause(20);

Insert the above line as the line just above the end brace **}** of the **while(1)** block. Save your code. Have an advisor check your code.

Load the code into the EEPROM of the robot and test it. Does your robot follow a wall properly? I hope that it will. Then you have completed your first challenge in developing a maze-solving program for ActivityBot. If your robot is turning too sharply it will crash into the wall or wander far from the wall. If your robot turns very little it may also crash into the wall or wander far from the wall. In either case, you need to change the speed of the slower wheel in your code to adjust the turn to a proper amount. Then test again until your robot is following the wall properly. You want the robot to follow the wall without moving excessively close to or excessively far from the wall.

File name: FollowingWall.pdf
online location: https://www.lafavre.us/robotics/FollowingWall.pdf
last update: 5/24/25