


## 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 shell below to your code file and save.

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

int ping_mm(int pin);

int main()
{
    //start of main() code block

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

int ping_mm(int pin)
{
    return ping(pin) * 10 / 58;
}
```

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 P17 of the microcontroller board. Here is a proper code statement:

```
int wallDistR = ping_mm(17);
```

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 P17 send out a burst of sound and then determine the amount of time in microseconds for the sound to return. Multiply the time by 10 and then divide by 58 and store the result in the variable named **wallDistR**.”*

The distance to the right wall in millimeters 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 107 to 117 mm.** The

midpoint of this range is 112 mm. The front face of the right facing sensor is about 15 mm inboard of the outside edge of the right wheel. Therefore, if your code keeps the robot sensor spaced an average of 112 mm from the wall, the robot wheel will be about  $112 - 15 = 98$  mm from the wall. How can you write the code to make that happen?

First, use the function `drive_ramp()` for driving the robot. Remember that `drive_ramp()` 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_ramp(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_ramp(_____, 64);       //turn slightly left, slow wheel speed not provided
```

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

```
drive_ramp(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 (107 to 117 mm).

Let us suppose that you set your robot down next to the wall carefully, making sure it is 112 mm 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. If 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:

```
if(put your condition here)
{
  drive_ramp(put wheel speeds here);
}

else if(put your condition here)
{
  drive_ramp(put wheel speeds here);
}

else          // no condition needed here, if above two conditions not
              // met, then this will happen
{
  drive_ramp(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 107 to 117 mm from the wall. In my tests, I found that the correction factor for the sensor was 11 mm, so the sensor distance range to code for my robot is 118 to 128 mm. 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 118 mm?

The code below poses the question properly in C language:

```
if(wallDistR < 118)
```

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 118?” If this condition evaluates to true, then code placed directly below the **if()** 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 128).

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\_mm(17);** 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,

December 13, 2015

[updated 3/1/18]

Making ActivityBot Follow a Wall – part 2

J. La Favre

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.