This is the first lesson in a series that will teach you to program ActivityBot to travel through a maze. It is the same maze used in a contest at the National Robotics Challenge.

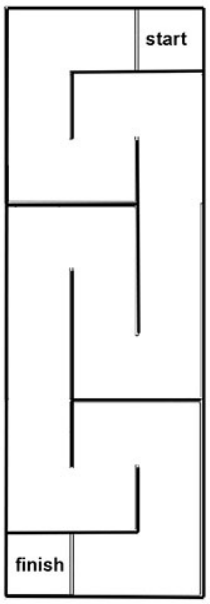

Figure 1 National Robotics Challenge maze - elementary and middle school age contestants

Figure 1 is a plan drawing of the maze. The robot must travel from the start position to the finish position without touching any wall. There are seven 90-degree right turns and seven 90-degree left turns in the maze that the robot must negotiate. The width of the maze corridor is 15 inches or 38 cm (spacing between walls on the right and left sides of the robot). Your ultimate objective will be to program your robot to travel the maze in the fastest time possible. At the National Robotics Challenge, the winning robot is the one that travels the fastest from the start to finish without touching a wall.

Various strategies apply to solve a maze with a robot. A common technique included in many strategies is **wall following**. The program causes the robot to travel parallel to a wall, using a sensor to maintain a specific distance from the wall.

The maze in Figure 1 is solved by following either the right wall or left wall. You will write a code solution for the maze utilizing a right-wall following technique.

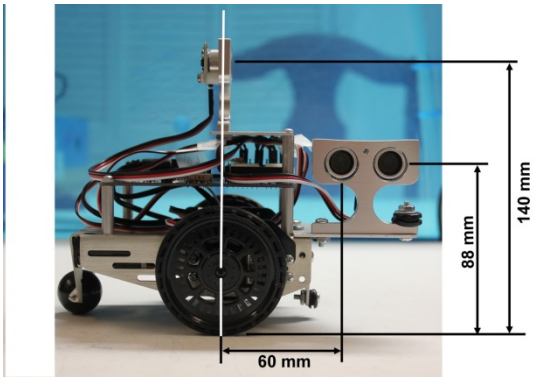

Figure 2 some important dimensions for Ping sensors attached to robot

Figure 2 is a side view of the ActivityBot robot, configured with two Ping sensors for measuring distances to walls. The sensor pointing to the right side of the robot (facing you in this photo) is centered 88 millimeters above the floor. The receiving port of this sensor is centered 60 millimeters forward of the center of the drive wheels.

The positions of the sensors on the robot are engineered to enable proper performance in the maze. Why do you suppose the right-facing sensor is placed well ahead of the drive wheels? Two important advantages are: **1)** placing the sensor ahead of the drive wheels allows the robot to detect the end of a wall early, in time to execute a proper turn and **2)** the sensor makes significant swings toward or away from the wall as the robot makes corrective steering moves. The second factor allows the robot to better control the wall-to-robot distance in the wall following procedure. Poorer performance of the robot would result if the right-facing sensor were placed directly over the drive wheels (I know because I tested it). It would be more difficult to program the robot to turn properly at the end of a wall. In addition, the robot would not follow a wall very well because the wall-to-sensor distance does not change fast at the position of the wheel when the robot makes corrective steering moves.

The Ping sensor facing forward is centered 140 mm above the floor with the front face of the sensor approximately aligned with the center of the drive wheels (see Figure 2). By placing this sensor higher than the right facing sensor, we avoid possible interference in distance measurements between the two

sensors. Additionally, placing this second sensor over the drive wheels is convenient for designing some of the turning code for the robot. A third factor is that of weight distribution. The robot is already somewhat unbalanced forward due to the overhanging weight of the right-facing sensor. When the robot stops quickly, the robot tips forward. If we also placed the front-facing sensor forward, the change in balance point would cause the robot to tip forward while standing still, which must be avoided.

A long wall will be set in place on the floor of our meeting room. 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 P17 of the microcontroller board. The second sensor points forward and is connected to port P16 (Figure 3).

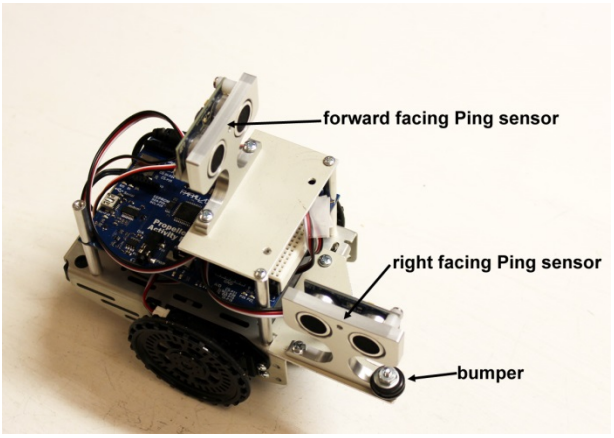

Figure 3 robot with Ping sensors installed

Calibrating your Ping sensor

Before you begin writing your maze program you should calibrate your Ping sensors. In order to do that you can copy the code on the next page (alternatively I may provide the code in a file for you).

```
/*
Test Ping_mm.c
Test the PING))) sensor before using it to navigate with the
ActivityBot.
*/

#include "simpletools.h"
#include "ping.h"

int ping_mm(int pin);
int dist;

int main()
   {
      while(1)
         {
            dist = ping_mm(17);  //make sure sensor is connected to
                                 // port P17, or change port number
            print("%c distance = %d%c mm", HOME, dist, CLREOL);
            pause(200);
         }
   }

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

Open the SimpleIDE software and click the new project button  . Save the project file with this name: **test ping mm** in the **My Projects** folder. With the code above, the robot will operate the sensor connected to port P17 (the sensor facing to the right). After you calibrate this sensor, you will need to edit the port number in **ping_mm()** by changing 17 to 16. Then calibrate the forward-facing sensor.

While you still have the **test ping mm** file open in SimpleIDE, connect your robot to the computer using the USB cable. Select the proper **COM** port in SimpleIDE (ask for help if you don't know how). Slide the robot power switch to position 1. Then click the **Run with Terminal** button  in the software toolbar. A terminal window should appear on your computer screen. Then slide the power switch to position 2, which will provide power to the sensor. Place an object with a flat, vertical face near the sensor. Move the object closer and then farther away from the sensor. The distance measurement seen in the terminal window should change. Set the robot at a distance of about 150 mm from the object and then be careful not to move the robot or object further during your measurement. Record the distance reported by the sensor (in terminal window) in the table on the next page. Using a metric ruler, measure the distance between the object and the front edge of the sensor and record it in the table.

It is likely that the measurement displayed on the computer screen will not match the measurement done with a ruler. Subtract the measured distance from the distance reported by the robot.

distance reported – distance measured = correction factor

Enter the correction factor in the table (the value may be positive or negative, make sure to indicate if negative with a minus sign).

You may use the correction factor for improving the accuracy of your code. Let's calculate the corrections for two examples. In both examples let's say you want a distance of 100 mm between the front edge of the sensor and a wall.

If the correction is +10 mm:

100 mm + 10 mm = 110 mm

If your correction is +10 mm, then you must add 10 mm to the **desired distance** and use that value in your code. Your code should contain a value of 110 when you want the true distance to be 100.

If the correction is -8 mm:

100 mm - 8 mm = 92 mm

If your correction is -8 mm, then you must subtract 8 mm from the desired distance and use that value in your code. Your code should contain a value of 92 when you want the true distance to be 100.

Now edit the ping test code by changing the port number 17 to 16. Repeat the calibration procedure for the forward-facing sensor connected to port P16 and add the values in the table below.

Correction Factors for sensors

	Distance reported by sensor	Distance measured by metric ruler	Correction factor
Right facing sensor			
Forward facing sensor			

The Ping sensor utilizes ultrasound, which is sound at frequencies above the hearing range of humans (humans can't hear sound above a frequency of about 20 kHz and the sensor produces sound at a frequency of 40 kHz). Shortly after the microprocessor sends an electrical "Start Pulse" to the sensor, the sensor emits a "click" of ultrasound (figure 4). When the sound leaves the sensor, the sensor also starts sending a high electrical signal back to the microprocessor. When the sensor detects the returning sound (echo), it stops the high signal output to the microcontroller. The time-length of the high signal (Echo Time Pulse) represents the amount of time for the sound to travel to the object and return. The distance to the object can be calculated by using the velocity of sound.

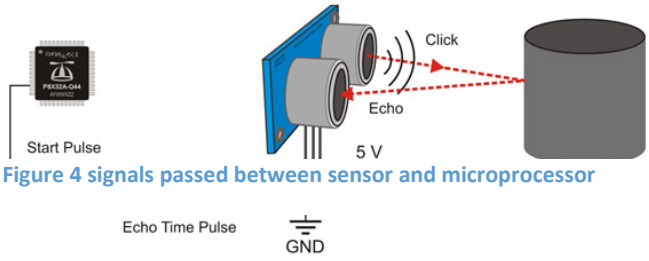

Figure 4 signals passed between sensor and microprocessor

The velocity of sound in dry air at 20 degrees Celsius is about 343.2 meters per second. Temperature has some effect on the velocity and at 30 degrees Celsius the velocity is about 349.5 meters per second,

about 2% greater than at 20 degrees. For our needs we can ignore this small difference in the velocity of sound and just use the velocity at 20 degrees Celsius.

The SimpleIDE software for programming the robot includes a ping library (ping.h) that has functions useful for working with the Ping sensor. There are two functions used to directly determine distance, one for distance in inches and one to determine distance in centimeters. In the case of our maze, with a small corridor width of 15 inches, measuring the distance in inches or centimeters may not provide the fine control necessary to prevent the robot from crashing into a wall. It would be better if the robot could measure distance in units of millimeters. The ping library does not have a function for measuring in units of millimeters. To solve this problem, you need some code that will create a custom function to measure distance in millimeters. That code will be provided to you in this lesson (it is the same code that is included in the Ping test file).

It would be most useful to convert the velocity of sound to units of millimeters per microsecond. First, multiply the distance in meters by 1,000 to convert to millimeters. Sound travels at a velocity of 343,200 mm per second. The ActivityBot microcontroller measures the time of high signal pulses from the Ping sensor in units of microseconds (µsec). There are one million microseconds in one second. Divide by one million to get the distance traveled in one microsecond. Sound travels at a velocity of **0.343 mm/µsec**.

Suppose the Ping sensor sends a high signal pulse to the microcontroller lasting 5.8 microseconds. Multiply **0.343 X 5.8 = 2.0 mm** . That is a convenient figure because it is equivalent to a distance of 1.0 mm to the object. In other words, if the object is one millimeter from the sensor, the sound will return to the sensor in 5.8 µsec. **If the time in microseconds is divided by 5.8, the time measurement is converted to a distance measurement in millimeters.**

Now let's take a look at some code for making a custom ping function.

```
int ping_mm(int pin)       //create a custom function named ping_mm()
   {
      return ping(pin) * 10 / 58;   //measure the time, multiply by
                                    //10, then divide by 58, return value
   }
```

The function **ping(pin)** measures the number of microseconds of the high signal sent by the Ping sensor. The number is multiplied by 10 and then divided by 58 in the code above. Why not just divide by 5.8?

The value returned by the **ping(pin)** function is an integer (whole number). If an integer is divided by a floating point number (like 5.8), then the result will be a floating point number (that is a rule of C language). Notice that our custom function is created with this line: **int ping_mm(int pin)**. The **int** part before **ping_mm** means that this function will return an integer. But if we divide by 5.8, the function won't be returning an integer. That is a problem. Since the number must be returned as an integer, our calculations must use whole numbers. Thus, multiply by 10 then divide by 58. That is the same as dividing by 5.8, but avoids using a float data type number.

There is no **ping_mm()** function in the ping library of SimpleIDE, which is why I have provided this custom function. However, there is a function named **ping_cm()** in the library. This function measures the distance in centimeters.

Here is how the **ping_cm()** function is defined in the ping.h library:

```
int ping_cm(int pin)
   {
      return ping(pin)  / 58;
   }
```

This code is very similar to the code for measurement in millimeters. The only difference is that the time value measured by **ping(pin)** is not multiplied by 10. Therefore, the values calculated will be one tenth that of values calculated by **ping_mm()**, which is correct since there are 10 millimeters in one centimeter. Now let's take a look at a code structure using the **ping_mm()** custom function.

```
#include "simpletools.h"
#include "ping.h"           // function ping() is in this library
#include "abdrive.h"        //function drive_ramp() is in this library

int ping_mm(int pin);       //notification of a custom function named
ping_mm()

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

//definition of function named ping_mm() is given in code below

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

The above is not a complete code solution. I could give you a complete code solution, but I believe you will become a proficient programmer sooner if you try to write part of the code yourself. I will give you some guidance to help you with the process.

First, let us go over the code I have provided for you. At the top are the **include statements** that name the libraries **simpletools.h**, **ping.h** and **abdrive.h**. In your code you will need to use functions contained in these libraries, so it is necessary to name them at the top of your code, exactly as written above (excluding the comments if you wish – the start of a comment is marked by two slashes: //).

The next line is:

```
int ping_mm(int pin);
```

This is the notification that a custom function named **ping_mm()** will be used.

Then take a look at this line near the bottom of the code:

```
int ping_mm(int pin)
```

This is the same as the notification, but without a semicolon (**;**) at the end. It is the beginning of the definition of the custom function. Notice that it is located below the end brace of the **main()** function. A definition of a custom function should be located outside the **main()** function.

The first line of the definition of **ping_mm()** indicates that the function will return an integer (**int**) and the name of the function will be **ping_mm().** The function includes one integer (**int**) variable named **pin** as the argument inside the parentheses. The value stored in the variable named **pin** will be the number placed inside the parentheses of the function when it is called within the **main()** function code block. That value should be the port number of the Ping sensor connection.

I have already discussed the purpose of this line: **return ping(pin) * 10 / 58;**

This lesson is continued in part 2