C Language Reference for ActivityBot

Jeffrey La Favre - November 4, 2015

Code for Driving ActivityBot

There are two types of functions for driving the robot. The first kind specifies a specific distance to travel and the second kind specifies a specific speed. These functions are part of the **abdrive.h** library so make sure to include this line at top of your code: **#include "abdrive.h"**

Driving the robot a specific distance or a specific amount of turn

The drive_goto() Function

The function **drive_goto()** is used to specify a distance for the robot to travel. It can be used for travel in a straight line or for sharp turns of the **pivot** or **zero radius** type. There are two arguments placed inside the parentheses of the function, the first for the **left wheel** and the second for the **right wheel**. The distance traveled is in units of wheel encoder ticks. There are 64 ticks per revolution of the wheel. The wheel has a circumference of 208 mm. Therefore, each tick is equal to a distance of 3.25 mm traveled (208/64 = 3.25).

Driving in a Straight Line with drive_goto()

Suppose you want the robot to move forward 100 mm. Then divide that amount by 3.25 to get the number of ticks required (100/3.25 = 30.8 ticks). If the answer is not a whole number of ticks, round off to the nearest whole number. You can't specify fractions of a tick.

The code for driving the robot forward 100 mm would be:

drive_goto(31, 31);

If you want the robot to move backward a distance of 100 mm, then the code would be:

drive_goto(-31, -31);

Specifying a negative number in the argument makes the wheel rotate backwards.

Making Sharp Turns of the Pivot and Zero Radius Type with drive_goto()

To make turns, the amount of rotation for each wheel must be different. For **pivot turns**, the wheel serving as the center of the pivot does not rotate at all (distance = 0). For **zero radius turns**, one wheel rotates forward while the other wheel rotates the same amount backwards.

In order to calculate the amount of distance to move in a turn, we need know the track of the robot (distance between the driving wheels). The track for ActivityBot is published as 105.8 mm. My robot is closer to 103 mm but we will stick with the published specification. If the track is 105.8 mm, then the radius of a pivot turn is also 105.8 mm. The circumference of a circle of 105.8 mm radius is:

$$c = 2\pi r$$

$$c = 2 \text{ X } 3.14 \text{ X } 105.8 = 664 \text{ mm}$$

If we divide the circumference by 360 degrees, we have the amount of travel along the circumference required to turn one degree:

664/360 = 1.84 mm per degree

Suppose we want to make a 90 degree turn. Then multiply 1.84 X 90 = 166 mm. The outboard wheel of the robot must travel 166 mm to make a 90 degree pivot turn. Then divide 166 mm by 3.25 = 51 encoder ticks.

Pivot Turns

To make a 90 degree right pivot turn this is the code:

drive_goto(51, 0);

Why does this cause the robot to turn 90 degrees right? The left wheel turns forward 51 ticks while the right wheel does not move (0 ticks).

To make a 90 degree pivot turn left, use this code:

drive_goto(0, 51);

Zero Radius Turns

To make **zero radius turns**, divide the number of ticks by two and assign one wheel half the ticks as a positive value and one wheel half the ticks as a negative value. Then one wheel moves forward while the other wheel moves backward. This causes the robot to rotate around the center point between the two drive wheels (which is the definition of a zero radius turn).

To make a zero radius turn of 90 degrees to the right use this code:

drive_goto(26, -25);

To zero radius turn 90 degrees left:

drive_goto(-25, 26);

The total number of ticks required is 51. Dividing by 2, we get 25.5 ticks. We can't use fractions of a tick so one wheel gets 26 ticks while the other gets 25.

A nice feature of **drive_goto()** is the built-in ramp up and ramp down. A ramp is a programmed rate of acceleration or deceleration. The robot starts out at a slow speed and accelerates at a controlled rate to full speed (if there is enough distance to get to full speed). As the robot approaches the specified distance, it begins to slow down at a controlled rate. This prevents wear and tear and jerky movements of the robot. If the robot accelerates or brakes at too great a rate, the wheels can slip, causing errors in the calculated distance traveled.

It can be difficult to get your robot to travel an exact distance and direction using the **drive_goto()** function. Keep in mind that there will be small errors in the mechanical alignment of the drive wheels. Also, the robot wheels can slip, causing errors in accounting for a distance traveled or amount of a turn. If the path your robot must travel is a long distance with several turns, it will be very difficult to program it to arrive at an exact final destination. You can make small adjustments by trial and error to your code, but you will still have variability between multiple runs of a course using the exact same code. If your objective is to arrive at a specific destination, then you may need to use another approach. For example, suppose you want your robot to travel in a maze without hitting the walls. If you do your code using only **drive_goto()**, it is likely your robot will crash into a wall before it gets to the end of the maze. A better approach would be to use sensors to assist with navigation. In this approach you give the robot various drive commands according to feedback from robot sensors. And you should use drive functions that specify speed, not distance.

Driving the robot at a specific speed

The drive_speed() and drive_ramp() Functions

There are two functions in this category: **drive_ramp()** and **drive_speed()**. The speed is specified in encoder ticks per second. The maximum speed is 128 ticks per second. This is equivalent to a wheel rotation rate of 120 RPM (revolutions per minute). The **drive_speed()** function does not include ramping (controlled acceleration and deceleration). For example, if the function **drive_speed(128, 128);** is applied, full power will immediately be applied to both wheels and the robot will accelerate at the maximum possible for the motors. On the other hand, **drive_ramp()** does apply controlled acceleration and deceleration. The default rate of control is a ramp step of 4 ticks per second every 20 milliseconds. In other words, if the robot starts from standing still, in 20 milliseconds it will be running at 4 ticks per second and at 40 milliseconds it will run at 8 ticks per second. The same is true in reverse when the robot is slowing down. You can change the ramp step using the **drive_setRampStep()** function. Suppose you want the robot to accelerate and brake at twice the default rate. Then you could add this code:

drive_setRampStep(8);

Driving an approximate distance at a specific speed

It is possible to drive an approximate distance using **drive_speed()** or **drive_ramp()**. At a speed of 128 ticks per second, maximum speed, the robot wheels are turning two times per second. The circumference of the wheel is 208 mm. Therefore, at a speed of 128 on both wheels, the robot is moving forward in a straight line at a rate of 416 mm per second. Let's say we want to move a short distance and use **drive_speed()** as the function. We want to move 400 mm forward. We know the robot moves 3.25 mm per tick. Therefore, we want to move forward 400/3.25 = 123 ticks. We only want to go half speed, or 64 ticks per second. Therefore, divide 123 ticks by 64 ticks/sec = 1.922 seconds or 1,922 milliseconds. If the robot is moving forward at a speed of 64 ticks per second, then in 1,922 milliseconds it will move 400 mm. We can code the time using the **pause()** function, which takes its argument in units of milliseconds. Here is how we could code for driving at a speed of 64 ticks per second for a distance of 400 mm:

```
drive_speed(64, 64);
pause(1922);
```

You may not want to use **drive_speed()** because it results in fast acceleration and braking (jerky movements). You could use **drive_ramp()** instead. However, then the distance traveled will be subject to more error unless an accounting for the acceleration and deceleration is included.

Driving a specific distance using drive_speed() or drive_ramp()

The function **drive_getTicks()** can be utilized with **drive_speed()** or **drive_ramp()** to make the robot travel a specified amount of distance. The procedure is more complicated than using **drive_goto()**, but does have some advantages.

When an ActivityBot robot program begins to run, the microprocessor uses a routine that tracks the number of wheel ticks moved for each wheel. This is a running total. You can use **drive_getTicks()** at any point in a program to get the number of ticks each wheel has moved up to that point. When you use **drive_getTicks()**, the numbers are stored in two variables which you create. Suppose you create four variables named leftTicks1, leftTicks2, rightTicks1 and rightTicks2. And you want to drive forward 300 mm. Then divide 300 mm by 3.25 = 92 ticks. You can create some code like this to drive 300 mm forward in a straight line:

```
#include "simpletools.h"
#include "abdrive.h"
//program to drive forward 300 mm

int ticksLeft1, ticksLeft2, ticksRight1, ticksRight2;           //declare variables

main()
{
ticksLeft2 = 0;                                   //make sure ticksLeft2 is low number before while()
drive_getTicks(&ticksLeft1, &ticksRight1);            //get number of ticks for each wheel

while(ticksLeft1 + 92 > ticksLeft2)  //stop driving after 92 wheel ticks on left wheel
{
drive_ramp(128, 128);                             //drive forward at speed 128
pause(20);                                //wait 20 milliseconds and then repeat while loop
drive_getTicks(&ticksLeft2, &ticksRight2); //get number of ticks but store in different variables
}
drive_ramp(0,0);                          //stop the robot
}
```

Analysis of above code

```
drive_getTicks(&ticksLeft1, &ticksRight1);
```

The above line gets the number of wheel ticks each wheel has turned since the program started and stores the number for the left wheel in the variable we have named ticksLeft1 and stores the number for the right wheel in the variable we have named ticksRight1. Notice that we use a pointer, the ampersand (&), directly in front of the variable name. We are pointing to that variable, telling the program to store the retrieved value of wheel ticks in that variable. If you don’t use the ampersand (&), the program won’t work.

```
while(ticksLeft1 + 92 > ticksLeft2)
```

The while() loop above will continue to run as long as the value of ticksLeft1 plus 92 is greater than the value of ticksLeft2. You will notice that further down in the code **drive_getTicks()** is used again, but this time storing the values in **ticksLeft2** and **ticksRight2**. Therefore, as the loop runs, the values stored in **ticksLeft2** and **ticksRight2** continue to increase. The **drive_getTicks()** function call, where values were stored in **ticksLeft1** and **ticksRight1**, comes before the while() loop, and therefore those values stay the same because their function call is outside the while loop. After the robot has moved 92 ticks forward, then the values stored in ticksLeft2 and ticksRight2 will be 92 greater in value than ticksLeft1 and ticksRight1. At that time the argument inside the while loop (ticksLeft1 + 92 > ticksLeft2) is no longer true and the program leaves the loop.

Obviously this is more complicated than using **drive_goto()**. What is the advantage? In general, using **drive_speed()** or **drive_ramp()** can result in a faster time for a robot moving through a course. The key factor is in making turns, where the **drive_goto()** approach requires the robot to come to a stop before making the turn. In contrast, using **drive_speed()** or **drive_ramp()**, you can use arc turns, where the robot does not need to even slow down (just the inboard wheel slows down, not the outboard wheel).

Driving in a curved path

You can't use **drive_goto()** for driving in curved paths, only straight lines or the pivot and zero radius turns. You may find the need for your robot to turn in a gentle manner, along a curved path. In order to do this you should use **drive_speed()** or **drive_ramp()**. If a different speed is specified for left and right wheels, then the robot will follow a curved path. You will need to do some calculations to code for these kinds of turns, which I call arc turns.

Making an Arc Turn

Unlike the pivot and zero radius turns, the arc turn can be a more gradual turn. If both wheels are moving forward or backward, but at different rates, then the robot will execute an arc turn. If the difference between the rates is small, then the turn will be very gradual. If the difference between the rates is large, then the robot will turn sharply.

Figure 1 provides a diagram of a sharp arc turn. To make calculations for the turn, we need to decide on the turn radius. There are different ways to specify the turn radius, but we will select a turn radius for the outboard wheel, which is labeled **ro** in the figure. We also need to know the radius of the inboard wheel, labeled **ri**. The **ri** radius can be calculated if we know the distance between the two drive wheels, which is about 106 mm. Therefore ri = ro – 106 mm.

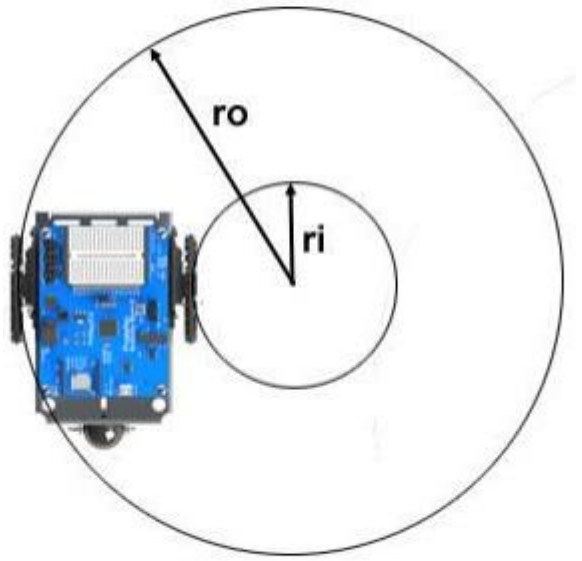

Figure 1 arc turn

Calculating the Arc Turn

If we know the radii of the circles for the outboard and inboard wheels, then we can calculate the speeds required for the wheels. First we need to select the speed desired for the outboard wheel (left wheel for a right turn). The speed unit is wheel ticks per second. The default maximum speed is 128 ticks per second so we will use that. Then we need to calculate the speed required for the inboard wheel. But first, let us consider the geometry of this type of turn.

Let us say we want the robot to make a complete 360 degree arc turn. In order to accomplish this, the outboard wheel must travel a distance equal to the circumference of the circle with the radius **ro**. In addition, the inboard wheel must travel a distance equal to the circumference of the circle with the radius **ri**. The math to be used here involves a ratio of the circumferences. But since the circumference of a circle is calculated by using the radius, multiplied by 2π, then we can just use the ratio of the two radii instead. Here is the formula for calculating the speed of the inboard wheel given the speed of the outboard wheel:

speed of inboard wheel = speed of outboard wheel X (ri/ro)

Let's do an example calculation. Suppose we want the robot to make a right hand arc turn of an outside radius of 190 mm. Furthermore, we want the left wheel to run at a speed of 128 ticks per second. Then what should the speed be for the right wheel? First we need to subtract 106 mm from the outside radius to find the inside radius: ri = 190 – 106 = 84 mm. The inboard wheel radius would be 84 mm if the outboard wheel radius is 190 mm. Let SIW be the speed of the inboard wheel and SOW be the speed of the outboard wheel.

SIW = SOW X (ri/ro)

SIW = 128 X (84/190) = 57

To make a right hand arc turn of an outside radius of 190 mm, we could code like this:

drive_ramp(128, 57);

In other words, the left wheel will run at a speed of 128 ticks per second, and the right wheel will run at a speed of 57 ticks per second

.

Making an Arc Turn of a Specified Number of Degrees

If we could use **drive_goto()** for arc turns, it would be relatively simple to code for a turn of a certain number of degrees. Unfortunately, we just can't do that with **drive_goto()** due to the way it functions. It works for pivot and zero radius turns, but not for arc turns. We must use a function that specifies the speed of wheel rotation. But we can calculate distance traveled using two factors: **1)** speed of wheel rotation **2)** amount of time wheel rotates.

We already have covered the calculations for determining the speeds required for the wheels. Now if we can specify the amount of time the wheels must rotate, then we can make the robot turn a desired number of degrees. The time can be controlled with the **pause()** function, placed on the line after the

drive_ramp() function. We will see how that is done shortly. After making calculations for the required wheel speeds, we need to do some additional calculations.

1. Calculate the circumference of the circle for the outboard wheel
2. Divide this circumference by 360 to get the distance to travel along the circumference for a one degree turn
3. Multiply the degrees of the desired turn by the distance to travel per degree to get the distance to travel
4. Multiply the speed of the outboard wheel given in ticks per second by 0.00325, which converts the speed into millimeters per millisecond
5. Divide the distance to travel by the speed of the wheel (mm/ms) to get the time in milliseconds required to make the turn

In order to accomplish a turn close to the desired number of degrees, it is necessary to add some additional lines of code. Remember that the **drive_ramp()** function accelerates and decelerates the wheels to the specified speeds. To simplify the calculations, the method of calculation I have presented does not account for acceleration or deceleration. We would need to use calculus for a more accurate treatment, which is too advanced to cover here.

The errors in degrees turned will be the greatest if you try to make the robot turn starting from a standstill and then trying to stop exactly at the end of the turn. To improve the accuracy of the turn, the robot should be moving in a straight line at the speed required for the outboard wheel during the turn and then after the turn is completed, should continue to move in a straight line for a short period while it comes to a stop (or continues). Even with these adjustments, you will still need to make small adjustments in the **pause()** time to get the exact amount of turn you want. Here is what an example code block might look like for an arc turn of 300 mm outside radius, 180 degrees:

```
drive_ramp(128, 128);   //accelerate to a speed of 128 while driving in a straight line forward

pause(1000);             //continue to drive straight ahead for one second

drive_ramp(128, 83);    //now change to drive speeds required to make the turn, right wheel slows

pause(2266);            //continue to drive in a circle for 2266 milliseconds, about 180 degree turn

drive_ramp(128, 128);   //now drive in a straight line

pause(1000);            //drive in a straight line for one second

drive_ramp(0, 0);       //now slow down to a stop
```

The **drive_getTicks()** function can be used to help reduce errors in an arc turn when using **drive_speed()** or **drive_ramp()**. Suppose you want to track the number of wheel ticks for the outboard wheel in an arc turn. You could track that with two **array** variables. Let's name them ticksLeft[] and ticksRight[]. Let's say you want to make a right arc turn of 180 degrees with a 300 mm radius. And you want to use **drive_getTicks()** to improve the accuracy of the turn. Then you could use the code found on the next page.

```
#include "simpletools.h"
#include "abdrive.h"
//program to arc turn 180 degrees right with 300 mm outside radius – code by Jeff La Favre

int ticksLeft[2], ticksRight[2]; //declare array variables, each array holds 2 variables, 0 to 1


main()
{
ticksLeft[1] = 0;                       //make sure ticksLeft[1] is low number before while()
drive_ramp(128,128);                    //accelerate to speed driving straight forward
pause(1000);                            //enough time to get up to full speed
drive_getTicks(&ticksLeft[0], &ticksRight[0]);      //get number of ticks for each wheel

while(ticksLeft[0] + 290 > ticksLeft[1])  //stop driving in circle after 290 wheel ticks left wheel
{
drive_ramp(128, 83);                    //drive in a circle of outside radius 300 mm
pause(20);                              //wait 20 milliseconds and then repeat while loop
drive_getTicks(&ticksLeft[1], &ticksRight[1]);  //store ticks in ticksLeft[1] & ticksRight[1]
}                                       //end of while loop

drive_ramp(128, 128);                   //now drive straight for 1 second
pause(1000);
drive_ramp(0, 0);                       //decelerate to speed 0
pause(1000);
}
```

Instead of naming four variables, as was done previously, in this example I have used **array variables**, so we only need two array names. Remember that array variables can be thought of as multiple variables with nearly the same names. Here is the declaration line for the array variables:

```
int ticksLeft[2], ticksRight[2];
```

This means that there will be two elements in each array, which can be referred to by the following names: ticksLeft[0], ticksLeft[1], and ticksRight[0], ticksRight[1].

Now let's jump down to this line:

```
drive_getTicks(&ticksLeft[0], &ticksRight[0]);
```

Here we are storing the number of wheel ticks traveled up to this point in the variable array elements ticksLeft[0] and ticksRight[0]. Remember to use the ampersand (&) in front of a variable name to point to it when using this function. Keep in mind that as the code is written above, there will be no further call of **drive_getTicks()** to update the values stored in ticksLeft[0] and ticksRight[0]. These represent the number of wheel ticks traveled just as the robot enters the turn.

```
while(ticksLeft[0] + 290 > ticksLeft[1])
```

The **while()** code line above is perhaps the most important line of code. We must concentrate on the argument inside the parentheses: ticksLeft[0] + 290 > ticksLeft[1]. As long as this argument evaluates to true, the while loop will continue to run. When will the loop stop? It will stop after the left (outboard) wheel has traveled 290 ticks along the circumference of the outboard circle. Suppose our robot has traveled 100 ticks for each wheel at the time it enters the turn. Then the value stored in ticksLeft[0] will be 100. To that value we must add 290, as that is what is given in the argument. So the total value would be 390. In the argument we are asking if 390 is greater than the value stored in ticksLeft[1]. The value of ticksLeft[1] will be 390 after the left wheel has moved 290 ticks in the turn because it had already moved 100 ticks before the turn. This can be better understood by considering this line Inside the while loop:

```
drive_getTicks(&ticksLeft[1], &ticksRight[1]);
```

In this case, the number of wheel ticks is being stored in ticksLeft[1] and ticksRight[1]. And each time the loop runs, that number is updated. After the left wheel has run 290 ticks (well close to that number), then the value stored in ticksLeft[1] will have a value of 390 (100 before the turn and 290 during the turn). But since were are adding 290 to ticksLeft[0] in the while argument, then we can say ticksLeft[0] + 290 equals ticksLeft[1] after 290 ticks of turn and at that point it is no longer true that ticksLeft[0] + 290 is greater than ticksLeft[1]. Since the argument inside the parentheses of while() is now false, the program exits the loop and continues on with the next line below the end of the loop.

Coding for Robot Sensors

Imagine what your world would be like without any senses. If you could not see, you would have great difficulty in navigating around your house, let alone outside your house. If you could not see or hear, then you would have great difficulty in communicating with others.

The addition of sensors to robots greatly enhances their capabilities. There are a number of sensors available for the ActivityBot robot. You will be using distance sensors in a number of your robot projects. We can divide the distance sensors into two categories: **1)** those that involve detection of an object by touch and **2)** those that detect objects remotely, without touch.

The standard ActivityBot kit includes a pair of “whiskers” that can be used to detect objects by touch. The kit also includes an ultrasound ping sensor for determining the distance to an object by bouncing sound off of it. In addition, the kit contains sensors that can detect objects using infrared light or visible light. This section of the code reference will cover the coding for some of these devices.

In using sensors, it is important to have a basic understanding of the input/output ports available on the robot microcontroller board. The ports provide a way for sensors to feed information to the microprocessor so that decisions can be made regarding the actions the robot should take. The actions take the form of some kind of output, for example, a signal sent to a continuous rotation servo motor, specifying a speed of rotation or a signal to a standard servo, connected to a robotic arm, specifying a certain amount of movement.

Think about the way you do a simple task, like pick up a glass of water. Your eyes serve as sensors to locate the position of the glass. That information is sent to your brain where it is processed. Your optic nerves serve as the input connection from eyes to brain. Your brain sends signals through nerves (output) to your arm and hand muscles to pick up the glass. All during this process, continual input from your eyes assists the brain in directing your arm and hand in picking up the glass. In a very real sense, there are similarities between how you would pick up a glass and how a robot would complete a task using sensors.

Figure 2 shows part of the microcontroller board of ActivityBot. Just to the left of the white breadboard, there is a series of input/output sockets labeled P0 to P15. You use these sockets to make wired connections to sensors, servo motors and other devices used with the robot. There are additional input/output ports along the top of the board, labeled P12 through P17. Therefore, ports P12 through P15 have two connection points for each port (one is a socket on the left, and one is a pin at the top). There is a difference between the connections at the top of the board and those on the left. The ports on the left have a direct connection to the microprocessor. The ports on top have a resistor between the port and the microprocessor. In most applications, it is necessary to have a resistor between a sensor or other device and the microprocessor to limit current flow. For the ports on the top, you don't need to worry about adding a resistor. However, such is not the case for the ports on the left.

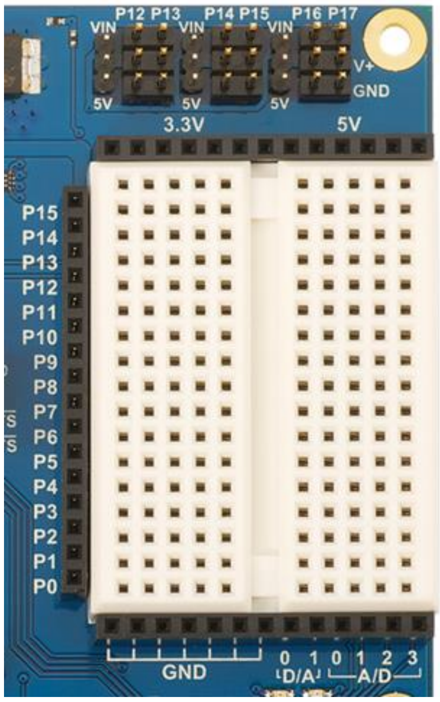

Figure 2

Notice that the ports along the top each have two additional pins below them, making a vertical column of three pins. The pin on top is the input/output port. The pin just below the port is a connection to positive power and the bottom pin is a connection to negative power (GND). Just to the left of top connections P12, P14 and P16, there is an additional set of three pins to be used with a jumper (not shown). Depending on the position of the jumper, the power delivered to the positive power pin will be 5 volts regulated or a direct connection to the positive side of the power supply (7.5 volt battery pack).

The sets of three pins with ports are used with connectors to wire the continuous rotation servos (wheel motors) and the wheel encoders. The left wheel motor is connected to P12, the right motor to P13, the left encoder to P14 and the right encoder to P15. That leaves P16 and P17 on the top of the board available for connecting other devices, such as a standard servo, which could be used on a robotic arm. Or a sensor that requires power to operate.

Some of the robot sensors, like the ultrasonic sensor, are complete modules requiring only power and a connection to an input/output port. In other cases, you can build a sensor circuit on the breadboard. In the latter case you may need to add power to the circuit by using the positive power sockets at the top side of the breadboard and the negative power sockets (GND) at the bottom of the breadboard. There are 6 sockets for positive 5 volts and 7 sockets for positive 3.3 volts.

Lastly, there are ports for digital to analog conversion (D/A) and analog to digital conversion (A/D) below the breadboard.

You may be wondering why I have given you all this information about the robot board. When you are writing your robot program, you will need to specify the port numbers in your code for the various devices you connect to the robot.

A touch sensor – coding for input

Figure 3 is an illustration of the breadboard layout for a set of whisker touch sensors (I have shortened the length of the whiskers to make a smaller illustration). While not shown in the illustration, the metal whiskers are connected to the negative side of the power supply (GND).

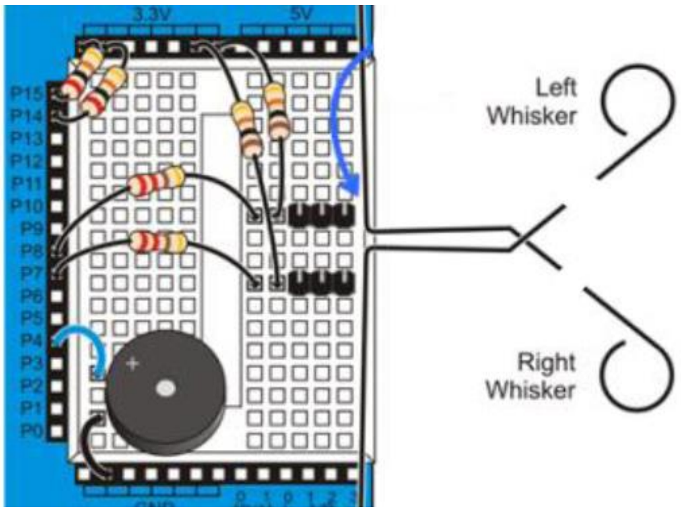

Figure 3

When one of the whiskers touches an object, its wire is pushed backwards until it makes electrical contact with a pin on the breadboard. That pin is connected to a resistor which in turn is connected to a socket supplying 3.3 volts. Another resistor is connected to the same pin and that resistor is connected to an input/output port on the left. A schematic of the wiring for the left whisker is given in Figure 4.

When the whisker is not in contact with an object, then the voltage supplied to port 7 (P7) will be 3.3 volts. You might wonder why the 10K and 220 ohm resistors in series between the positive voltage supply and P7 don't drop the voltage. However, the input at P7 is high impedance, which means there is very little voltage drop across the 10K and 220 resistors. In other words, when the whisker switch is open, the P7 port will see a high volt signal, which we usually just say is "high." Now when the whisker touches an object, the switch closes. Now P7 sees a "low" input because most of the voltage is dropped over the 10K resistor between the 3.3 V socket and ground. To reiterate, when the switch is open (no touch) there will be a HIGH signal input to P7 and when the switch is closed, there will be a LOW input signal to P7.

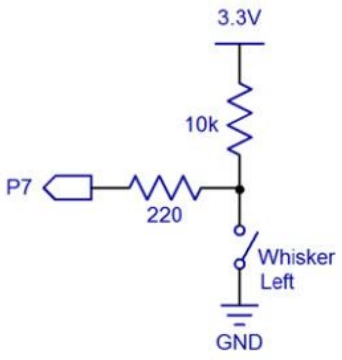

Figure 4

Now let's take a look at a function we can use in coding a program to use a touch sensor.

The input() function

The **input()** function is part of the **simpletools.h** library. Suppose we write some code like this:

input (7);

That means this: set an I/O pin to input and return 1 if pin detects a high signal, or 0 if it detects low.

Let's see that in a section of code for the touch sensor:

```
 while(1)
 {
   int wL = input(7);
   int wR = input(8);

   if((wL == 0) || (wR == 0))
   {
     drive_speed(-64, -64);
   }
   else
   {
     drive_speed(0, 0);
   }
 }
```

Let's look at this line:

int wL = input(7);

The above line declares a variable of the integer type named wL. Furthermore, the code uses the **input()** function to determine if there is a high or low input at P7. If high, then 1 is stored in wL and if low, 0 is stored in wL. If the touch sensor connected to P7 has detected an object, a value of 0 is stored in wL, otherwise a value of 1 is stored in wL. The same pattern holds for the variable named wR which stores the condition of the right touch sensor, which is connected to P8.

Now let's look at this line:

if((wL == 0) || (wR == 0))

Keep in mind that if the left whisker has detected an object, then wL will equal 0 and if the right whisker had detected an object, then wR will equal 0. The above code line is asking this question: does wL equal 0 OR does wR equal 0. This code: || means OR and == means equal, when you are asking a question, not assigning a value. Therefore, since this: **(wL == 0) || (wR == 0)** is inside the parentheses for **if()**, we are asking if either wL or wR equals 0. The question will evaluate to TRUE if either or both are equal to 0.

Then the next line:

drive_speed(-64, -64);

In other words, back up if either detector has detected an object.

The line under else is:

drive_speed(0, 0);

In other words, if no object has been detected, do not move

This programs results in the following behavior for the robot. If either whisker is touching an object, then the robot backs up until it does not detect the object. Suppose you put a book on the floor next to the robot and then push the book toward the robot until one or both of the whiskers detect the book. Then the robot backs up until the whiskers no longer detect the book.

Making a Light Blink – coding for output

Now let's take a look at coding that causes the microprocessor to do some output to a port. The ActivityBot has two built-in LEDs on the board below the breadboard as seen in Figure 5. One is connected to port 26 and the other to port 27 (these ports don't have sockets on the board). We can do a coding exercise to demonstrate output to port 26 (P26). In this case the output at P26 applies a positive voltage to the connected LED, causing it to glow. Let's take a look at a code snippet on the next page.

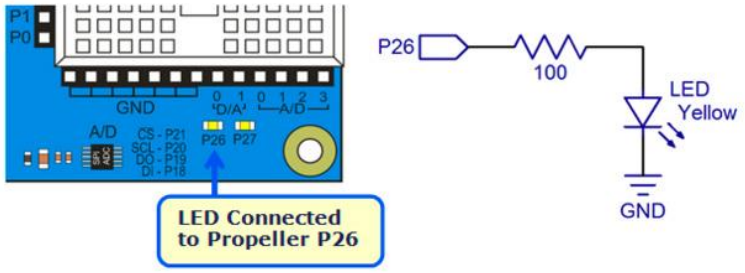

Figure 5

```
while(1)                    // Endless loop
  {
    high(26);                   // Set P26 I/O pin high
    pause(100);                   // Wait 1/10 second
    low(26);                    // Set P26 I/O pin low
    pause(100);                   // Wait another 1/10 second
  }
```

The above code causes the LED connected to P26 to blink on and off. It is on for 100 milliseconds and then off for 100 milliseconds.

high() and low() functions

The **high()** and **low()** functions are part of the **simpletools.h** library. The **high()** function causes the microcontroller board to connect the specified port to the positive 3.3 volt supply. The total output the board is capable of sending to all ports is 40 milliamps so keep that in mind when you are connecting many devices to ports on the robot. That current limit does not include ports 12 to 17 when their power jumper is set to VIN, which is 7.5 V. So for example, the drive motors for the robot are not included in the current limit. The **low()** function causes the microcontroller board to connect the specified port to the ground side of the power supply (0 volts).

Looking above at the code then, **high(26);** causes 3.3 volts to be applied to port 26, at which time the LED will start to glow. It will stay in that condition for 100 milliseconds because the next line of code is **pause(100);**. The code **low(26);** causes 0 volts to be applied to port 26, at which time the LED will turn off.

In the next section on the Ping Sensor, we will see how the functions **high()** and **low()** are used to send control signals to the sensor.

The Ultrasound Ping Sensor

Figure 6 is a schematic of the wiring of a Ping sensor. The key here is that the SIG (signal) pin of this module must be connected to an input/output port on the microcontroller board. In the schematic the sensor's SIG pin is connected to a 2.2 kΩ resistor, which in turn is connected to P8 (remember, we need to use a resistor to connect a device to the port).

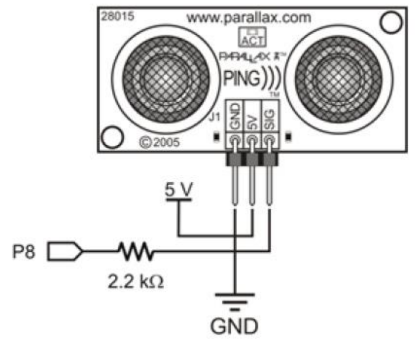

Figure 6

The **ping.h** library contains functions that can be used with this sensor to code a program. Let's start with the **ping()** function, which is the most basic of functions available in this library. This is a custom function provided by the robot manufacturer, so that coding will be simplified for you. The code developed by the manufacturer is on the next page.

```
int ping(int pin)            //this line is the definition of the function, which will be named ping()
{
low(pin);                    //set the port number (pin) to output 0 volts
pulse_out(pin, 10);          //cause the port to go to 3.3 volts for 10 microseconds
return pulse_in(pin, 1);     //measure the length (time) of a pulse input to the port
}
```

The first line is:

int ping(int pin)

which gives the function the name **ping** and specifies that one argument should be included inside the parentheses (the port number). The whole code is included in the **ping.h** library, which means that all you have to do is use **ping(**port number here**)** to take advantage of the code lines included with this function. The first line inside the braces for this function is:

low(pin);

and therefore, the port that the sensor is connected to will be set to output 0 volts. Immediately after that we have a call to a function that is part of the **simpletools.h** library:

pulse_out(pin, 10);

This causes the voltage at the specified port to go to 3.3 volts and stay at that voltage for 10 microseconds. Then it goes back to 0 volts. What has just happened is that the microcontroller has sent a 10 microsecond pulse to the sensor. If we look in the data sheet for the sensor, we find that a 5 microsecond pulse is the suggested minimum signal input, which will cause the sensor to send out a burst of ultrasonic sound.

The next line of code is:

return pulse_in(pin, 1);

pulse_in() is another function in the **simpletools.h** library. This function causes the microcontroller to measure the pulse length of a pulse input in units of microseconds (there are one million microseconds in one second). The number 1 in (pin, 1) is specifying that the microcontroller should measure the length of a positive voltage pulse being input (from the sensor in this case).

At the time the sensor sends out an ultrasonic sound pulse, it starts to send a positive voltage signal back to the microcontroller port (the port is switched from output to input). The voltage will remain high until the sensor detects the echo from the sound pulse. Therefore, the high voltage pulse sent back to the microcontroller is a measure of the time it takes for sound to leave the sensor and then return. The function **pulse_in()** is used to measure the time. Then with some additional math, we can calculate the distance from the sensor to the object that caused the sound to bounce back to the sensor. In other words, we can measure the distance to an object, which is very useful in navigation.

Well, how do we convert the time pulse into a distance measurement? The calculation is taken care of with another function in the **ping.h** library, but let's do the calculations as an exercise.

We have just measured the amount of time it takes for a burst of sound to travel to an object, bounce off, and return. If we know the velocity of sound, then we can convert the time measurement into a distance measurement.

In dry air at 20 °C, the speed of sound is 34,320 centimeters per second. Then how long does it take sound to travel a centimeter? Divide one second by 34,320 = 0.0000291 seconds to travel one centimeter. Multiply by 1,000,000 to get it in units of microseconds: sound travels one centimeter in 29.1 µs (microseconds). We have our distance measurement in units of microseconds, so divide by 29.1 to convert to centimeters. Let's take this example. We use **ping()** to make a measurement and it returns a value of 400 µs. Then divide 400 µs by 29.1 µs/cm = 13.74 cm. The sound traveled a distance of 13.74 cm. But remember this is the distance to the object and back. So divide 13.74 by 2 = 6.87 cm. The distance to the object is 6.87 cm. We could simplify the procedure by just dividing once by 58. Let's try that: 400 µs / 58 = 6.90 cm, the distance to the object. That number, 58, is exactly what is used in the function **ping_cm()**. Let's look at the code for the function **ping_cm()**.

```
int ping_cm(int pin)   //statement defining the function
{
  long tEcho = ping(pin);          //store time for sound bounce into a long integer variable named tEcho
  int cmDist = tEcho / 58;         //divide the value stored in tEcho by 58 and store value in cmDist
  return cmDist;                   //function returns the value stored in cmDist
}
```

Let's look at this line:

long tEcho = ping(pin);

We are using the **ping()** function to measure the amount of microseconds that it takes for sound to travel to and back from an object. We have already done the calculations to reveal that we can divide the number of microseconds by 58 to find the distance to the object in centimeters. So let's look at the next line:

int cmDist = tEcho / 58;

The number of microseconds was stored in the variable named tEcho. Now we are going to divide that number by 58 and store the answer in a variable named cmDist. The number stored in cmDist is the number returned by the **ping_cm()** function.

Suppose we use the **ping_cm()** function in our program. And we want the distance determined by **ping_cm()** to be stored in a variable named **distance**. Furthermore, we want to print the distance determined. The code snippet on the next page accomplished these tasks;

```
int distance;

int main()

{

distance = ping_cm(8);

print("The distance to the object is %d centimeters", distance);

}
```

The above code specifies that the ping sensor is connected to port 8: **ping_cm(8)** The distance measured in centimeters is stored in the variable named **distance**. Then the value stored in distance is displayed on the computer screen. Suppose the distance determined was 10 cm. Then the display would be "The distance to the object is 10 centimeters."