Extra Lesson 2 – Code for Driving the ActivityBot Robot

Jeffrey La Favre

November 2, 2015

The ActivityBot robot has left and right drive wheels and an unpowered rear tail wheel. Each drive wheel has a separate motor. If both wheels rotate at the same speed and direction, the robot moves in a straight line. If the wheels don't rotate at the same speed and direction, then the robot will execute a turn. This arrangement is known as **differential drive steering**. In this lesson you will learn how to write code for driving the robot in straight lines and turns.

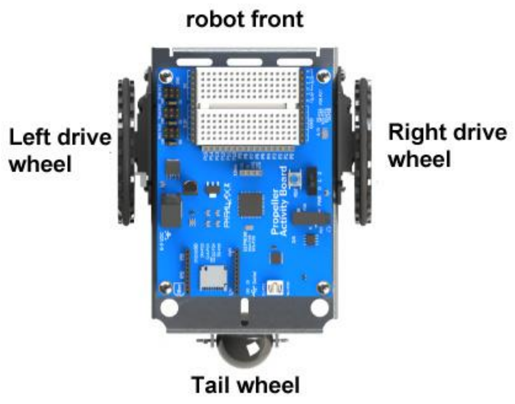

Figure 1 wheels of ActivityBot robot

The IDE for the ActivityBot robot contains a number of functions that can be used to make the robot wheels rotate. These functions are not available in **Dev-C++** because it is a general IDE, not intended to develop a program for a robot. In any case, we can use **Dev-C++** to write programs that will do some calculations useful in developing a robot program for ActivityBot.

In the IDE libraries for ActivityBot there is a function named **drive_goto()**, that can be used to cause the wheels of the robot to spin. There are two arguments required inside the parentheses of **drive_goto(***number of ticks for left wheel, number of ticks for right wheel***)**. This is an example statement:

drive_goto(256, 256);

In pseudocode this would be something like this: *turn the left wheel 256 ticks forward and the right wheel 256 ticks forward*. A tick is approximately 3.25 mm (you will see shortly how this was calculated). Therefore, the line of code above instructs the robot to move forward in a straight line a distance of 256 ticks X 3.25 mm/tick = 832 mm.

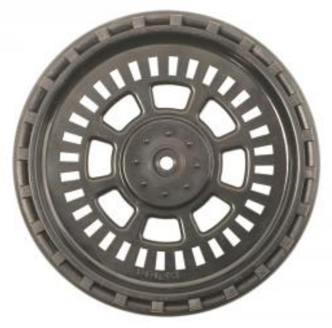

Figure 2 ActivityBot drive wheel

What is a tick? We are not referring to a clock tick here, but a wheel encoder tick. A wheel encoder tick is a pulse sent to the robot **microcontroller** from an **encoder** attached on the frame next to the wheel. The encoder consists of an IR emitter and receiver, similar to the circuit you completed for the IR module of the line following robot. The IR emitter shines light at the wheel, which contains an outer ring of 32 perforations. When the IR light shines on a spoke, it is reflected back to the IR receiver. When the light shines through a perforation, the light is not reflected back to the receiver. Therefore, with each complete rotation of the wheel, the IR receiver will pulse HIGH 32 times and LOW 32 times. The robot microprocessor counts the high and low pulses, which we call ticks. Therefore, there are 64 ticks for each revolution of the wheel.

We would like to convert a tick into a distance traveled by the robot. To do that, we need to measure the diameter of the wheel, which is 66.2 mm. Then we need to calculate the circumference of the wheel. The circumference of the wheel is also the distance the wheel will travel upon one revolution. Now we are getting somewhere, but how do we measure the circumference? That would not be very easy, but we can use a formula to calculate the circumference:

$$\text{circumference} = \pi \times \text{diameter}$$

The value of π is approximately 3.14. So let's calculate the distance the wheel will move with one revolution:

$$\text{circumference} = 3.14 \times 66.2 \text{ mm} = 208 \text{ mm}$$

How far does the wheel move for each tick of the encoder?

$$208 \text{ mm} / 64 \text{ ticks} = 3.25 \text{ mm/tick}$$

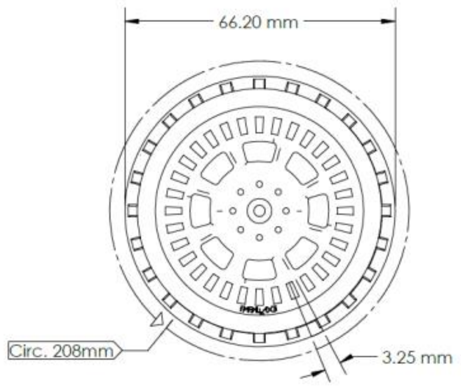

Figure 3 ActivityBot drive wheel dimensions

We could do the math by hand to calculate the number of ticks required to move a specific distance, but since we are learning programming, let's write a C program to do the calculations.

We want our program to calculate the number of ticks required to travel a specified distance. We could describe the process like this:

1. Create two variables of the floating point type and name them **distance** and **ticks**. Actually, we could name the variables almost anything we like, but it is a good idea to use a name that relates to what the variable stores.
2. Display text in the terminal window that asks the user to use the keyboard to input the distance the robot should travel.
3. Store the value input in the variable named **distance**
4. Divide the value stored in **distance** by 3.25 and assign the answer to the variable named **ticks**.
5. Use the **printf()** function to output the value contained in the variable named **ticks**.

An example code is on the next page.

```
#include <stdio.h>
// calculate the number of ticks to move ActivityBot forward or backward a specified distance in mm

float distance, ticks;      //create two variables of the floating point type named distance and ticks

int main()
{                               // start of main() block of code
        printf("  This program calculates the number of ticks for moving ActivityBot \n");
        printf("a specific distance in millimeters.\n\n");
        printf("  The drive_goto() function is used to make the wheels turn a specified amount.\n");
        printf("The encoder for each wheel sends 64 ticks to the microprocessor for each \n");
        printf("revolution of the wheel. The circumference of the wheel is 208 mm and dividing \n");
        printf("this by 64 ticks gives 3.25 mm per tick. In other words, the robot wheel will \n");
        printf("move a distance equivalent to the number of ticks multiplied by 3.25. \n");
        printf("The format for the drive_goto() function is:\n\n");
        printf(" drive_goto(number of ticks for left wheel, number of ticks for right wheel)\n\n");
        printf("For example, drive_goto(234, 234) will cause both wheels to rotate forward 234 ticks.\n\n");

        while(1)            // this is an indefinite loop, so program will run indefinitely until it is closed
        {                   //  start of while() block of code
        printf("Enter the distance to travel in millimeters and then press the Enter key\n\n");
        scanf("%f", &distance);             //get input from keyboard and store it in variable named distance
        ticks = distance / 3.25;            // divide value stored in distance by 3.25 and store answer in variable named ticks
        printf("\n To travel %.0f mm forward, use %.0f ticks for each wheel\n", distance, ticks);   //display number of ticks to move desired mm
        printf("The code would be like this: drive_goto(%.0f, %.0f);\n\n", ticks, ticks);      //display code to move forward desired mm
        printf("\nTo travel %.0f mm backward, use -%.0f ticks for each wheel\n", distance, ticks);
        printf("The code would be like this: drive_goto(-%.0f, -%.0f);\n\n", ticks, ticks);   //display code to move backward desired mm
        }                                   // end of while() block of code
}                                           // end of main() block of code
```

Copy all of the code above (including code on page 3) into your text window in **Dev-C++** and save with the file name of **calculate_robot_distance**. Use caution when copying code that you do not include the page number at the bottom of page 3. The best way to do this is to first copy all the code on page 3 and paste it in Dev-C++. Then copy the code on this page and paste it directly below the code of page 3. If you can't remember how to paste code into Dev-C++ and run it, check instructions on page 2 of lesson 1. After you save the program code in Dev-C++, compile and run it. Hopefully the program will work.

Now let's go through some of the lines of the program. Most of the code should look familiar. Let's start with this line of code:

float distance, ticks;

This line names the two variables we want to use: **distance** and **ticks**. We have assigned the data type of **float** (*i.e.,* floating point number) to these variables. This was done because in the calculation the value for distance is divided by a floating point number, 3.25.

The first 10 statement inside the **main()** function are **printf()** functions which display information about the program and how to use the **drive_goto()** function. The first two statements are:

printf(" This program calculates the number of ticks for moving ActivityBot \n");

printf("a specific distance in millimeters.\n\n");

The code at the bottom of page 4 causes the text inside the parentheses to be displayed on the computer screen like this:

This program calculates the number of ticks for moving ActivityBot
a specific distance in millimeters.

All of the above text could be placed in one **printf()** function, but it is divided into two because that results in a better display of the text in the small terminal window of the program. The amount of text in the first statement is about the maximum amount that can be displayed on one line on my computer. If you try to put too much text in one **printf()**, then a word might be cut into two parts, with one part on each line. Also, remember that the invisible character **\n** means new line. When the program reads **\n**, it moves the cursor on the computer screen to the next line.

After the first 10 **printf()** statements, we have the **while(1)** loop, which allows the program to continue to run, so that we can do more than one calculation if we wish.

The first statement inside the **while(1)** loop is:

printf("Enter the distance to travel in millimeters and then press the Enter key\n\n");

This line of code prints instructions for using the program.

Next we have:

scanf("%f", &distance);

The **scanf()** function collects input from the computer keyboard. The **%f** inside the double quotes is the placeholder for a floating point variable and that variable is identified by including the second argument: **&distance**. In other words, the number entered using the keyboard will be stored in the variable named **distance**. The ampersand character **&** is called a pointer and it must be placed just before the name of the variable for the **scanf()** function. The pointer is used to point to the address location in RAM memory where the value of the named variable is stored.

Once we have collected the desired distance to move and store it in the variable named **distance**, it is time to do the math:

ticks = distance / 3.25;

This line of code divides the number stored in the variable named **distance** by 3.25 and then assigns the result of the calculation (the answer) to the variable named **ticks**. Now we have our answer. The value stored in the variable named **ticks** is the number of ticks we must specify in our robot code in order to make the robot move the desired distance. But we now need to display this information on the computer screen using this code:

printf("\n To travel %.0f mm forward, use %.0f ticks for each wheel\n", distance, ticks);

Notice that we have two place holders (**%.0f**) in the first argument of the **printf()** function. The number stored in the variable named **distance** will appear in the first placeholder listed and the number stored in the variable named **ticks** will appear in the second placeholder listed. That is because the variables are named in that order at the end of the arguments inside the **printf()**. The numbers displayed will not have any places to the right of the decimal point because **.0** is included before the **f** in the placeholder (%.**0**f). The simple form of the placeholder would be **%f**, which makes no specification about the number of places after the decimal point. If we used **%f**, then the number displayed would have many places to the right of the decimal point, which we don't want for this program.

Suppose we used the keyboard to enter a value of 100. Then the number stored in the variable named **distance** would be 100. After the calculation is executed (distance / 3.25), a value of about 30.77 would be stored in the variable named **ticks**. The code would result in the following text displayed on the screen: "**To travel 100 mm forward, use 31 ticks for each wheel**." Since we have specified that the numbers to be displayed will have no places to the right of the decimal place, the number stored in ticks (about 30.77) is rounded to the nearest whole number. There cannot be a fraction of a tick in the robot system, and this makes the best choice for displaying the number.

In order to help the person writing a robot program, next we will display text on the screen for the proper way to code the function:

printf("The code would be like this: drive_goto(%.0f, %.0f);\n\n", ticks, ticks);

In our example, the rounded value of the floating point number stored in the variable ticks is 31. Therefore, the above code would result in the following display on the computer screen: "**The code would be like this: drive_goto(31, 31);**".

If we use negative numbers for ticks in the **drive_goto()** function, then the robot will move backwards. The next line of code prints out this information:

printf("\nTo travel %.0f mm backward, use -%.0f ticks for each wheel\n", distance, ticks);

Notice that there is a minus sign before the second placeholder (**-%.0f**) so that the number displayed will have a negative value, as it needs to be for moving backward.

And then the next line gives the code example for moving backwards:

printf("The code would be like this: drive_goto(-%.0f, -%.0f);\n\n", ticks, ticks);

The above code would result in the following display on the computer screen: "**The code would be like this: drive_goto(-31,- 31);**".

Making the ActivityBot robot turn

Up to this point you have learned some coding that will make the robot move forward or backward in a straight path. The ActivityBot can turn right or left by using differential drive steering, just like the line following robot you have already completed. If the two wheels of the robot turn different amounts, then the robot will execute a turn. In this exercise you will learn how to make the robot turn a specified amount and will create a program to calculate the ticks needed for each wheel to make a turn.

The ActivityBot has a track of 105.8 millimeters (the distance between the drive wheels). Suppose we used the **drive_goto()** function to make one wheel turn a certain number of ticks while the other wheel turned zero ticks. Then the robot would pivot in a circle around the wheel that did not move. Take the example in figure 4 where the robot pivots around its right wheel. The point where the right wheel touches the floor marks the center of the circle. The radius of the circle is the distance between the right and left wheels (the track), which is 105.8 mm. The left wheel will travel around the circumference of the circle. If the amount the left wheel turns is equivalent to one fourth of the circle circumference, then the robot will execute a right turn of 90 degrees. If the amount the left wheel turns is equivalent to the circumference of the circle, the robot will turn a full circle, 360 degrees, to the right.

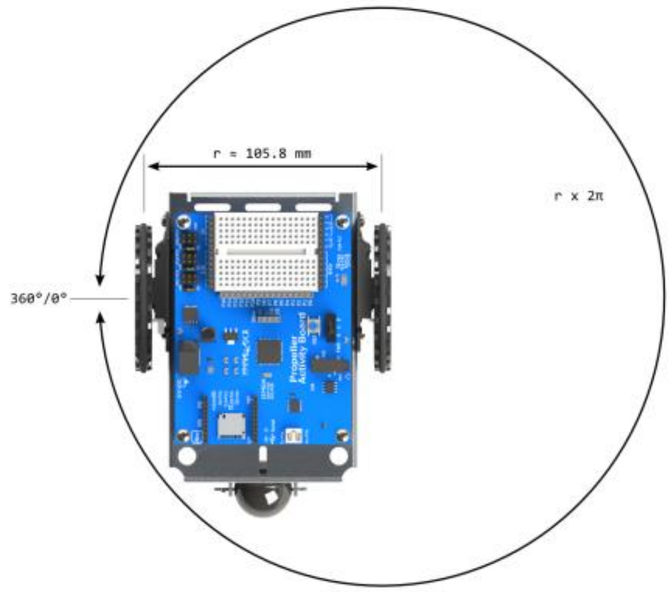

Figure 4 pivot of robot around the right wheel

Suppose we want the robot to turn 90 degrees to the right. Then we need to code for a left wheel that runs a distance equivalent to 1/4th of the circumference. First we need to calculate the circumference:

circumference = 2 X radius X π

circumference = 2 X 105.8 mm X 3.14 = 664 mm

The circumference of a complete 360 degree turn is 664 mm. We divide 664 by 4 to find the distance for a 90 degree turn: 664 / 4 = 166 mm. The left wheel must travel 166 mm to make a 90 degree right turn. Now we need to know the number of ticks for the left wheel to move. Divide 166 mm by 3.25 ticks/mm = 51 ticks. If the left wheel moves forward 51 ticks while the right wheel does not move, the robot will turn 90 degrees to the right. This code: **drive_goto(51, 0);** will make the robot turn 90 degrees to the right with a pivot around the right wheel. The code for turning 90 degrees left, pivoting around the left wheel, would be: **drive_goto(0, 51);**

We could of course specify turns that are more or less than 90 degrees. A full 360 degree turn takes 664 mm / 3.25 mm/tick = 204.3 ticks and therefore a one degree turn takes 204.2/360 = 0.5672 ticks. To make a turn of 45 degrees then, we need to move one wheel 0.5672 X 45 = 26 ticks.

Making a zero radius turn

There is another way to make the robot turn by making both wheels move the same amount, but in different directions. This makes what is called a zero radius turn. The robot turns around a center point located half way between the two wheels. In this case we divide the number of ticks required for the turn by 2 and assign that number to each wheel, but with opposite signs. For example, to make a zero radius turn to the right 90 degrees, the code would be: **drive_goto(25, -26)**. This makes the left wheel move forward 25 ticks and the right wheel move in reverse 26 ticks. The total of the two is 51 ticks, the amount required for a 90 degree turn. We can't do this: **drive_goto(25.5, -25.5)** because we can only specify whole numbers for the ticks. Therefore, if the total ticks is an odd number, we must assign one wheel to move one more tick than the other wheel.

Now we have enough information to make a program to calculate the number of ticks required to make a turn. We must multiply the number of degrees of a desired turn by 0.5672 to get the number of ticks required.

The code on the next page calculates the number of ticks required to make a turn.

```
#include <stdio.h>
// calculate the number of ticks to make ActivityBot turn a specified number of degrees.  code by Jeffrey La Favre

float degrees, ticks;                  //create two variables named degrees and ticks of the floating point data type

int main()
{
        printf("  This program calculates the number of ticks for making ActivityBot turn \n");
        printf("a specified number of degrees.\n\n");
        printf("The drive_goto() function is used to make the wheels rotate a specified amount.\n");
        printf("The encoder for each wheel sends 64 ticks to the microprocessor for each \n");
        printf("revolution of the wheel. The circumference of the wheel is 208 mm and dividing \n");
        printf("this by 64 ticks gives 3.25 mm per tick. In other words, the robot wheel will \n");
        printf("move a distance equivalent to the number of ticks multiplied by 3.25. \n");
        printf("The format for the drive_goto() function is:\n\n");
        printf(" drive_goto(number of ticks for left wheel, number of ticks for right wheel)\n\n");
        printf("The robot makes a pivot turn by rotating one wheel only.\n");
        printf("The robot makes a zero radius turn by rotating both wheels, but in opposite\n directions. " );
        printf("To execute a full 360 degree turn, the total ticks required is\n 204.3 ticks ");
        printf("or 0.5672 ticks per degree.\n");
        printf("For example, drive_goto(51, 0) will cause the robot to pivot turn 90 degrees right.\n\n");
        printf("drive_goto(26, -25) will cause the robot to zero radius turn 90 degrees right.\n\n");


        while(1)                    // indefinite while loop allows program to run until closed

        {

        printf("Enter the degrees to turn and then press the Enter key\n\n");

        scanf("%f", &degrees);  //get input from keyboard and store in the variable named degrees

        ticks = degrees * 0.5672;          //calculate the number of ticks required to turn specified degrees

        printf("\n To turn %.0f degrees with a pivot, use %.0f ticks for one wheel and 0 ticks\n", degrees, ticks);

        printf("for the other wheel.\n\n");

        printf("The code to pivot right would be like this: drive_goto(%.0f, 0);\n\n", ticks);

        printf("The code to pivot left would be like this: drive_goto(0, %.0f);\n\n", ticks);

        printf("\nTo zero radius turn, assign half the ticks to one wheel as a positive value\n ");

        printf("and the other half to the other wheel as a negative value. ");

        printf("Let us take the\n example of a 90 degree turn, which takes 51 ticks.\n");

        printf("Half of 51 is 25.5 ticks.  Therefore, one wheel gets 26 ticks the other 25 ticks\n");

        printf("Code for zero radius turn 90 degrees right would be:\n");

        printf("drive_goto(26, -25)\n");

        printf("Code for zero radius turn 90 degrees left would be:\n");

        printf("drive_goto(-25, 26)\n");

        }

}
```

Copy all of the code on pages 9 and 10 into your text window in **Dev-C++** and save with the file name of **calculate_robot_turn**. If you can't remember how to do this, check instructions on page 2 of lesson 1. After you save it, compile and run it. Hopefully the program will work.

Now let's go through some of the lines of the program. Most of the code should look familiar. Let's start with this line of code:

```
float degrees, ticks;
```

This creates two variables of the float type named **degrees** and **ticks**. Then there are 15 **printf()** lines that display information on the computer screen. After that there is the **while(1)** line to make this program run indefinitely.

This line of code:

```
scanf("%f", &degrees);
```

takes the input from the keyboard and stores the number of desired degrees of turn into the variable named degrees.

Then this line:

```
ticks = degrees * 0.5672;
```

multiplies the number of degrees for the turn by 0.562 to convert to the number of wheel ticks required to make the turn. Then there are 12 lines of **printf()** to display the information for coding the turn. The information displayed includes coding for a pivot turn and coding for a zero radius turn.

Making an Arc Turn

There is a third type of turn that ActivityBot can do, called 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.

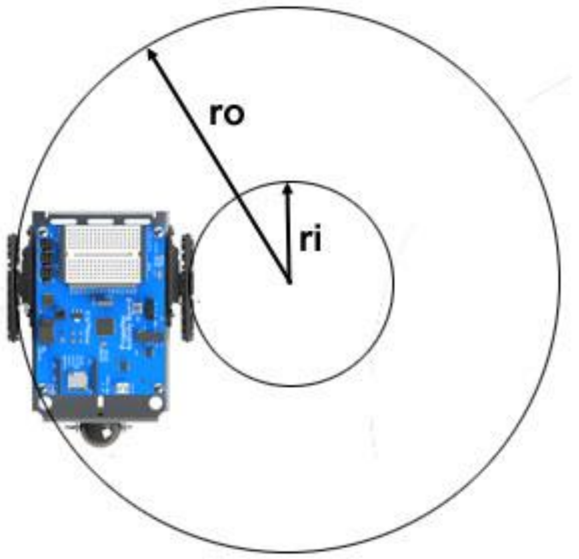

Figure 5 arc turn

Figure 5 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.

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 must run at a speed of 57. You may have noticed that I have used a new function for driving the wheels: **drive_ramp()**. We can't use **drive_goto()** for arc turns because it will not yield the desired results. In contrast, **drive_ramp()** works well with arc turns.

drive_ramp() causes the wheels to rotate at the specified speeds, but the robot ramps up to those speeds. In other words, the robot will accelerate to a specified speed at a certain rate. The default rate is an increase of 4 ticks per second every 20 milliseconds. Therefore, it will take the robot a little more than half a second to reach full speed from a standstill. If no other drive function occurs after this code line, the robot will continue to run in a circle indefinitely. There are various way to limit the time that the robot will run in the circle, and we will cover one method in the next section.

Copy all of the code on pages 14 and 15 into your text window in **Dev-C++** and save with the file name of **calculate_arc_turn**. If you can't remember how to do this, check instructions on page 2 of lesson 1. After you save it, compile and run it. Hopefully the program will work.

```c
#include <stdio.h>

/* program to calculate wheel speeds for a robot arc turn

code by Jeffrey La Favre */


int ro, ri, spo, spi;

/*  ro = radius of outboard wheel in mm, ri = radius of inboard wheel

spo = speed of outboard wheel, spi = speed of inboard wheel    */


main()

{

        printf("This program calculates the speeds for wheels of ActivityBot\n");

        printf("for arc turns using the function drive_ramp \n");

        printf("Keep in mind that the  maximum wheel speed is 128 unless specified otherwise in code.\n\n");


        while(1)                                                //infinite loop

        {

        printf("Enter the outside radius of the turn in millimeters and then press Enter key \n\n");

        scanf("%d", &ro);                                       //get input from keyboard and store in variable named ro

        ri = ro - 106;                                          // subtract 106 mm from ro to get ri

        printf("Enter the desired speed of outboard wheel and then press Enter key \n\n");

        scanf("%d", &spo);      //get input from keyboard and store in variable named spo
        spi = spo * ri / ro;          //speed of inboard wheel will be ratio of inboard and outboard radii multiplied by speed of outboard wheel
        printf("For a speed of %d for outboard wheel use a speed of %d for inboard wheel\n", spo, spi);
        printf("for an arc turn with an outside radius of %d millimeters\n", ro);
        printf("For a right turn write the code like this: drive_ramp(%d, %d)\n", spo, spi);
        printf("For a left turn write the code like this: drive_ramp(%d, %d)\n\n", spi, spo);
        }
}
```

Now let's take a look at the code for this program.

```
int ro, ri, spo, spi;
```

The code line above creates 4 variables of the int type. The variable ro will store the radius of the outboard wheel circle, the variable ri will store the radius of the inboard wheel circle, the variable spo will store the speed of the outboard wheel and the variable spi will store the speed of the inboard wheel.

```
scanf("%d", &ro);
```

The above code line retrieves the value input by the user for the outside radius and stores it in the variable ro.

```
ri = ro - 106;
```

The above code subtracts 106 from the value stored in ro and assigns the answer to the variable ri.

```
scanf("%d", &spo);
```

The above code retrieves the value input by the user for the desired speed of the outboard wheel and stores it in variable spo.

```
spi = spo * ri / ro;
```

The above code line calculates the speed for the inboard wheel and stores it in the variable spi. The calculation is done like this. First multiply the speed of the outboard wheel (spo) by the radius of the inboard wheel (ri). Then divide the answer by the radius of the outboard wheel (ro). Then store that answer in the variable named spi.

This is followed by four **printf()** lines which displays the speeds for the wheels for the desired radius and gives the code required to code for right and left turns.

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 rather than the number of wheel rotations. 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 (spo) of the outboard wheel given in ticks per second by 0.00325, which converts the speed into millimeters per millisecond (we will use a variable named sow for this)
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

The code for the program is on the next two pages.

```
#include <stdio.h>

/* program to calculate wheel speeds for a robot arc turn of specified number of degrees

code by Jeffrey La Favre */


float ro, ri, spo, spi, tp;

float co, c1d, degrees, sow, doc;

/*  ro = radius of outboard wheel in mm, ri = radius of inboard wheel, spo = speed of outboard wheel in ticks per sec, spi = speed of inboard wheel

co = circumference of outboard circle in mm, c1d = co/360, degrees = degrees to turn, sow = speed of outboard wheel in mm/ms,

tp = time for pause in ms, doc = distance to travel on outboard circle in mm*/


main()

{

        printf("This program calculates the speeds for wheels of ActivityBot\n");

        printf("for arc turns of specified degrees using the function drive_ramp \n");

        printf("Keep in mind that the  maximum wheel speed is 128.\n\n");


        while(1)                                              //infinite loop

        {

        printf("Enter the outside radius of the turn in millimeters (no less than 106) and then press Enter key \n\n");

        scanf("%f", &ro);                                     //get input from keyboard and store in variable named ro

        ri = ro - 106;                                        // subtract 106 mm from ro to get ri
        printf("Enter the desired speed of outboard wheel (maximum value is 128) and then press Enter key \n\n");
        scanf("%f", &spo);        //get input from keyboard and store in variable named spo
        printf("Enter the desired degrees for the turn and then press Enter key \n\n");
        scanf("%f", &degrees);  //get input from keyboard and store in variable named degrees
        spi = spo * ri / ro;       //speed of inboard wheel (spi) will be ratio of inboard and outboard radii multiplied by speed of outboard wheel
        co = 6.283 * ro;           // calculate circumference of outboard circle and store in variable named co
        c1d = co / 360;            //calculate c1d, distance in mm along circumference of outboard circle per degree
        doc = c1d * degrees;       // calculate distance to travel in mm (doc) for specified number of degrees
        sow = spo * 0.00325;       // converts outboard wheel speed (spo) from ticks per second to mm per millisecond (sow)
        tp = doc / sow;            // calculates time in milliseconds to run (td value goes in the pause() function after the drive function)
        printf("You specified a %.0f degree arc turn with an outside radius of %.0f mm \n", degrees, ro);
        printf("and a speed of %.0f for the outboard wheel. \nUse a speed of %.0f for inboard wheel\n", spo, spi);
        printf("The pause after drive_ramp() should be %.0f milliseconds\n", tp);
        printf("For a right turn write the code like this: drive_ramp(%.0f, %.0f);\n", spo, spi);
        printf("The next line should be this: pause(%.0f);\n", tp);
        printf("For a left turn write the code like this: drive_ramp(%.0f, %.0f);\n", spi, spo);
        printf("The next code line should be this: pause(%.0f);\n", tp);
        printf("Keep in mind that additional code will be needed before and after the turn\n to start and stop the robot\n\n" );


        }
}
```

Copy all of the code on pages 17 and 18 into your text window in **Dev-C++** and save with the file name of **calculate_arc_turn_degrees**. If you can't remember how to do this, check instructions on page 2 of lesson 1. After you save it, compile and run it. Hopefully the program will work.

Let's take a look at selected lines of code for this program, which has many of the same lines as the previous program. Here is the first line to take a look at:

```
co = 6.283 * ro;
```

Here we are calculating the circumference of the outboard circle and storing it in the variable named co. The circumference is calculated by multiplying the radius (ro) by 2π, which is approximately 6.283.

The next line,

```
c1d = co / 360;
```

divides the circumference stored in co by 360 and stores the answer in the variable named c1d. c1d is the distance that the outboard wheel must travel along the outside circle to make a one degree turn.

The next line,

```
doc = c1d * degrees;
```

multiplies the value stored in c1d by the degrees desired for the turn and stores the answer in the variable named doc. doc is the distance that the outboard wheel must travel along the outside circle to make a turn of the specified number of degrees

The next line,

```
sow = spo * 0.00325;
```

multiplies the speed of the wheel in ticks per second (spo) by 0.00325 and stores the answer in the variable named sow. Remember that the wheel turns 3.25 mm per wheel tick. So if we multiplied spo by 3.25, then we would have the speed in millimeters per second. But we need the speed in millimeters per millisecond. If we multiply spo by 0.00325, then we have the speed of the outboard wheel in millimeters per millisecond.

Finally we have this last line of calculation,

```
tp = doc / sow;
```

which divides the distance to travel along the outboard circle (doc) by the speed of the outboard wheel (sow). The answer is stored in the variable tp, which is the milliseconds required for the wheel to run along the outboard circle so that the robot turns the specified number of degrees. We need to know three values to make an arc turn of a specified number of degrees: **1)** speed in ticks per second of outboard wheel, **2)** speed in ticks per second of inboard wheel and **3)** number of milliseconds the wheel should run along the outboard circle to achieve a turn of the specified number of degrees. We covered how to calculate the wheel speeds previously. All of the new calculations in this section deal with finding the amount of time the outboard wheel must run.

In order to accomplish a turn very 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.

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. 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…in less than one second robot will be moving full speed

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

pause(2266);                   //continue to drive in a circle for 2266 milliseconds, which is approximately a 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
```

I tested the code on the previous page with my ActivityBot and found that it turned a little more than 180 degrees. When I changed the pause(2266) to pause(2200), then the robot turned 180 degrees. In my case I think the major problem was that my robot has a track of about 103 mm instead of the 106 mm used for calculation. Also of course, there are some small errors that develop as the robot enters and exits a turn due to deceleration and acceleration of the inboard wheel. In any case, it is likely that you will need to "tweak" your code if you want accurate turns.

Fine Tuning the Arc Turn

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. The code on the next page is not for Dev-C++ but rather for the robot. Nevertheless, let's take a look at it to understand how we can use the **drive_getTicks()** function to improve the accuracy of an arc turn.

```
#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);
}
```

Until you have studied the coding for the robot, you may not be able to follow this program. You can just skip the explanation if it is too difficult to understand. Let me just point out one important factor. In order to make the above code work, we need know the number of wheel ticks for the outboard wheel required to make the desired turn. For a 180 degree turn with an outside radius of 300 mm, the number of wheel ticks for the outboard wheel must be 290. You can see that number in the line of the **while()** loop. In a nutshell, the above program counts the number of wheel ticks of the left wheel during the arc turn and stops the turn when the wheel has turned 290 ticks. That is how we can improve the accuracy of an arc turn. You can read the explanation of the code below or skip it. I have marked the start and end of the explanation. After that comes the program for calculating the number of ticks required for making the arc turn.

Start of Code explanation

The above code utilizes two array variables. 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.

End of Code Explanation

After we have determined the wheel speeds required for an arc turn of a specified outside radius, then we need to calculate the number of wheel ticks required by the outboard wheel to make a turn of the desired number of degrees. Here are the steps:

1. Calculate the circumference of the circle of the outside radius
2. Divide the circumference by 360 to get the distance to travel along the outside circumference for one degree of a turn
3. Multiply the value of the distance per degree by the desired number of degrees of turn
4. Divide the distance to travel for the turn by 3.25 mm/tick to get the number of ticks required.

Some of the above steps are the same as those outlined in the previous section. The difference is that now we are calculating the number of wheel ticks required instead of the time. The program to calculate the number of ticks will be similar to the previous program.

```
#include <stdio.h>

/* program to calculate wheel speeds for a robot arc turn of specified number of degrees

code by Jeffrey La Favre */


float ro, ri, spo, spi;

float co, c1d, degrees, doc, ticks;

/*  ro = radius of outboard wheel in mm, ri = radius of inboard wheel, spo = speed of outboard wheel in ticks per sec, spi = speed of inboard wheel

co = circumference of outboard circle in mm, c1d = co/360, degrees = degrees to turn, ticks = number of ticks for outboard wheel

doc = distance to travel on outboard circle in mm*/
```

```
main()
{
        printf("Program 4 calculates the speeds for wheels of ActivityBot\n");
        printf("for arc turns of specified degrees using the functions drive_ramp \n and drive_getTicks()\n ");
        printf("Keep in mind that the  maximum wheel speed is 128.\n\n");

        while(1)                                          //infinite loop
        {
        printf("Enter the outside radius of the turn in millimeters (no less than 106) and then press Enter key \n\n");
        scanf("%f", &ro);                                 //get input from keyboard and store in variable named ro
        ri = ro - 106;                                    // subtract 106 mm from ro to get ri
        printf("Enter the desired speed of outboard wheel (maximum value is 128) and then press Enter key \n\n");
        scanf("%f", &spo);          //get input from keyboard and store in variable named spo
        printf("Enter the desired degrees for the turn and then press Enter key \n\n");
        scanf("%f", &degrees);  //get input from keyboard and store in variable named degrees
        spi = spo * ri / ro;        //speed of inboard wheel (spi) will be ratio of inboard and outboard radii multiplied by speed of outboard wheel
        co = 6.283 * ro;            // calculate circumference of outboard circle and store in variable named co
        c1d = co / 360;             //calculate c1d, distance in mm along circumference of outboard circle per degree
        doc = c1d * degrees;        // calculate distance to travel in mm (doc) for specified number of degrees
        ticks = doc / 3.25;         // convert distance to travel in mm to number of wheel ticks
        printf("You specified a %.0f degree arc turn with an outside radius of %.0f mm \n", degrees, ro);
        printf("and a speed of %.0f for the outboard wheel. \nUse a speed of %.0f for inboard wheel\n", spo, spi);

        printf("The number of ticks for the outboard wheel is %.0f ticks\n", ticks);

        printf("For a right turn write the code like this: drive_ramp(%.0f, %.0f);\n", spo, spi);

        printf("For a left turn write the code like this: drive_ramp(%.0f, %.0f);\n", spi, spo);

        printf("Consult paper on using the function drive_getTicks() for complete \ncoding guidance\n ");


        }

    }
```

You have now learned how to make calculations for driving the robot straight for a specified distance and how to make calculations for three types of turns: pivot, zero radius and arc. You have also studied the code that can be used to create programs to make the calculations. If we combine all of the programs together, then we would have a nice program to use when developing code to run the robot. I can supply the program for you.