Custom Arc Turn Function

Jeffrey La Favre

November 6, 2015

If you completed all the lessons designed for Dev-C++, then you have already learned about custom functions in the custom pause function lesson. When we are writing code for ActivityBot, there is no need for a custom pause function because it is already included in the abdrive library of the SimpleIDE software. However, there is not a function in the SimpleIDE libraries for an arc turn. This lesson will cover the coding that could be used to create a custom arc turn function named **drive_arcTurn()**. This custom function eliminates the need to make calculations on your own for making an arc turn. You just add the specifications for the turn as arguments of **drive_arcTurn()** and the calculations are done for you by the program. The code for the custom function is below and on the next page.

```
/* custom arc turn function by Jeffrey La Favre
function format:  drive_arcTurn(int radius, int speed, int degrees, int turn)
radius is radius of outboard wheel in mm, speed is in ticks per second (max 128), degrees is number of
degrees to turn, and for last argument named turn, enter 1 to turn right and 2 to turn left.    */

#include "simpletools.h"                      // Include simple tools library
#include "abdrive.h"                          //Include abdrive library

void drive_arcTurn(int radius, int speed, int degrees, int turn);             //notification of custom function

int main()                                    // Main function
{
drive_ramp(128, 128);                         //drive straight for 1 second so robot is up to speed at start of turn
pause(1000);
drive_arcTurn(300, 128, 180, 1); // 300 mm radius, outboard speed 128, turn 180 degrees, right turn
drive_ramp(128, 128);                         //drive straight for 1 second after turning
pause(1000);
drive_ramp(0, 0);                             //robot comes to a stop
}                                             //end of main() function

//all lines below define the custom function named drive_arcTurn()

void drive_arcTurn(int radius, int speed, int degrees, int turn)              //declares the function
{
//variables
float c;                                      //variable for outside circumference
int ro;                                       //variable for turn radius of outboard wheel
int ri;                                       //variable for turn radius of inboard wheel
int inside_speed;                             //variable to store inboard wheel speed
float ticks_per_degree;                       //number of ticks required on outboard wheel to move one degree
int ticks_turn;                               // number of ticks on outboard wheel to turn specified number of degrees
int ticks_left[2], ticks_right[2];            //array variables to track number of wheel ticks traveled


if(speed > 128) speed = 128;                  //prevents a speed exceeding 128 for outboard wheel
ro = radius;                                  //radius is specified in mm, so ro will be in mm
```

```
ri = ro - 106;                          //106 mm is the track of the robot (distance between drive wheels)
inside_speed = speed * ri/ro;       //calculate speed of inboard wheel
c = radius * 6.283;                     //calculate circumference of circle for outboard wheel
ticks_per_degree = c / 1170;  // 360 degrees x 3.25 mm/tick = 1170, divide by 1170 to get ticks per degree
ticks_turn = ticks_per_degree * degrees;   //calculate number of outboard wheel ticks required for turn

if (turn == 1)                           //if true execute a right turn
{
  ticks_left[1] = 0;
  drive_getTicks(&ticks_left[0], &ticks_right[0]);  //get number of wheel ticks traveled for each wheel
  while(ticks_left[0] + ticks_turn > ticks_left[1])    //outboard wheel turns the number ticks in ticks_turn
  {
    drive_ramp(speed, inside_speed);  //left wheel speed is speed, right wheel speed is inside_speed
    pause(20);
    drive_getTicks(&ticks_left[1], &ticks_right[1]);  //get number of wheel ticks total traveled
  }
}
else                                     //if turn does not equal 1, then execute a left turn
{
  ticks_right[1] = 0;
  drive_getTicks(&ticks_left[0], &ticks_right[0]);
  while(ticks_right[0] + ticks_turn > ticks_right[1])
  {
    drive_ramp(inside_speed, speed);
    pause(20);
    drive_getTicks(&ticks_left[1], &ticks_right[1]);
 }       //end of while
}        //end of else
}        //end of code block defining the custom function
```

Let's take a look at specific lines of the code. Let's start with this line inside the **main()** function:

drive_arcTurn(300, 128, 180, 1);

If the lines of code defining the **drive_arcTurn()** function are already in place (for example if you just copied my code), then this is the only line of code required to use the custom function. You need to understand that this custom function takes four arguments in this order: radius of turn, speed of outboard wheel, number of degrees to turn and direction of turn. Note that the arguments are given in the code line above as 300, 128, 180, 1. In other words, the programmer wants the robot to make a turn with an outside radius of 300 mm, with an outboard wheel speed of 128, a turn of 180 degrees and the turn will be to the right, designated by the value of 1. As long as the programmer understands how to write the required arguments for the function, there is no need to understand the code below the **main()** block, that defines what the function will do with the arguments specified. In fact, all of that code could be placed in a custom library and then would not need to be added to the code below the **main()** block. The only requirement then would be to mention the custom library name in an #include statement.

Now let's look at a line of code near the top:

void drive_arcTurn(int radius, int speed, int degrees, int turn);

When we define a custom function below the **main()** code block, it is necessary to let the computer know we intend to use a custom function in a statement placed before the **main()** block. Therefore, this line becomes a notification of the use of a custom function. The first word in the notification is **void**, which means this custom function will not return any value upon execution of the function. You don't need to worry about that if you don't understand. The next word is **drive_arcTurn**, which is the name the programmer has selected to represent this custom function. A set of parentheses with arguments inside follow directly after the name of the function. Here is where we learn that four arguments must be included inside the parentheses. Furthermore, the first number we place there will be stored in a variable named **radius**, the second number in **speed**, the third number in **degrees** and the fourth number in **turn**. When the **drive_arcTurn()** function executes, the values stored in those four variables are used in the code inside the function definition.

Now let's take a look at the code defining the function. The first line is exactly the same as the notification line above, except there is not semicolon at the end (**;**):

void drive_arcTurn(int radius, int speed, int degrees, int turn)

This may seem redundant, but this is how we need to start the definition of the function. Next we have 7 lines of code that declare 8 variables. We need these variables to carry out the calculations required to determine the values needed for the turn. Take a look at this line:

ro = radius;

What we have done here is store the value contained in the variable **radius** in the variable named **ro**. That value will be the first value specified in the line **drive_arcTurn(300, 128, 180, 1);** In other words, the value stored in the variable would be 300 and therefore that value will also be stored in **ro**.

The next line is:

ri = ro - 106;

The track of the ActivityBot robot is 106 mm, which is the distance between the drive wheels. If the outboard wheel radius (**ro**) is 300 mm, then we must subtract 106 mm from it to find the radius of the inboard wheel (**ri**). Whatever the outboard radius might be, the inboard radius will be 106 mm less. That is why we have subtracted 106 from the outboard radius and stored the result in the variable named **ri**.

We need to know three values to specify an arc turn of a set number of degrees: speed of outboard wheel, speed of inboard wheel and number of wheel ticks for outboard wheel. The speed of the outboard wheel is decided by the programmer. It is not calculated. That leaves two values to be calculated. The next line of code calculates the speed of the inboard wheel:

inside_speed = speed * ri/ro;

If we divide the radius of turn for the inboard wheel (ri) by the radius of turn of the outboard wheel (ro) and multiply the result by the speed of the outboard wheel, then that result will be the speed required

for the inboard wheel. That is what this line of code does. Now we have the speed of the inboard wheel stored in the variable named **inside_speed**. The only value remaining to calculate is the number of ticks for the outboard wheel to turn for the specified number of degrees. That will take three lines of code:

c = radius * 6.283;

With the above line of code we calculate the circumference of the circle that the outboard wheel travels on during the turn (6.283 is equivalent to 2π). That value is stored in the variable named **c**.

ticks_per_degree = c / 1170;

Here we have divided the value in **c** by 1170 and stored it in the variable named **ticks_per_degree**. Where did I get 1170? Well, we want to find the number of wheel ticks required for the outboard wheel to travel along the circumference to make a turn of one degree. If we divide the circumference by 360 we get the number of millimeters along the circumference that equal one degree. And then if we were to divide that by 3.25, we would get the number of wheel ticks that equal one degree. So where did I get 1170? Here is the answer: 360 X 3.25 = 1170. Now we have stored in the variable **ticks_per_degree** the number of wheel ticks for the outboard wheel to move one degree. We are almost there:

ticks_turn = ticks_per_degree * degrees;

We multiply the value stored in **ticks_per_degree** by the value stored in **degrees** and store that in **ticks_turn**. Now we have the third value we need. The remainder of the code utilizes these values with the **drive_ramp()** function to make the robot turn the specified number of degrees. So let's take a look at how that is coded.

if (turn == 1)

Here we have an **if()** which is used to specify either the code for a right turn (when the variable **turn** equals 1) or a left turn (when the variable **turn** does not equal 1). The code for a right turn is directly under the **if()** line and the code for the left turn is under the **else** line. We will just take a look at the right turn since the left turn is nearly the same code.

drive_getTicks(&ticks_left[0], &ticks_right[0]);

Here we are using the **drive_getTicks()** function to find out how many wheel ticks the robot has traveled since the start of the program. The numbers are stored in the array variables **ticks_left[0]** and **ticks_right[0]**.

while(ticks_right[0] + ticks_turn > ticks_right[1])

The code above is very important. This **while()** line is what determines the amount of turn. It is the argument inside the parentheses of **while()** that determines the turn. A greater than (**>**) symbol is used. On the left side of the argument we have **ticks_right[0] + ticks_turn** and on the right **ticks_right[1]**. The value stored in **ticks_right[0]** is the number of wheel ticks the robot has moved up to the point where it starts to turn. To that value we are adding the number stored in **ticks_turn**, which is the number of wheel ticks we want the right wheel to turn while it is executing the arc turn. The argument is asking this question: has the right wheel NOT turned the number of ticks stored in **ticks_turn** since the robot started turning. If the answer is TRUE, then the while loop continues to run and the robot continues to turn. But when the robot has completed the specified number of degrees of turn, then the number stored in **ticks_right[1]** will be greater than the number stored in **ticks_right[0]** by a value equal to the number stored in **ticks_turn**. When that happens, **ticks_right[0] + ticks_turn** EQUALS **ticks_right[1]** and the **while()** loop condition is no longer true. Then execution of the turn stops.

Well, I am guessing your brain might be hurting a little at this point. Mine is after writing it. If this is just too confusing, you should not worry. As you do more coding, it will be easier to follow the logic required to program in C language.

The key here is that you can copy the code I have developed here and use it in a custom function. I can help you do that. Then when you are programming your robot to do various arc turns, your coding job will be easier and you will be able to make changes in the turn parameters quickly, without the need of doing calculations. That is the advantage of using custom functions. In fact, there are many custom functions provided in SimpleIDE that make your coding task much easier. For example, the code behind the **drive_goto()** function is quite complex and would require much time to develop on your own.

Here are the directions for using the custom function named **drive_arcTurn()**.

1. Copy all lines of code below the line: "//all lines below define the custom function named **drive_arcTurn()"**

2. Paste that code at the bottom of your code (below the close brace (**}**) of the **main()** function.

3. The following lines must be at the top of your code:
   ```
   #include "simpletools.h"
   #include "abdrive.h"
   void drive_arcTurn(int radius, int speed, int degrees, int turn);
   ```

4. You should probably include this comment above the line #include "simpletools.h"
   ```
   /* custom arc turn function by Jeffrey La Favre
   function format:  drive_arcTurn(int radius, int speed, int degrees, int turn)
   radius is radius of outboard wheel in mm, speed is in ticks per second (max 128), degrees is
   number of degrees to turn, and for last argument named turn, enter 1 to turn right and 2 to turn
   left.   */
   ```