CDA 4630 – Embedded Systems

Final Report

Group 4: Camilo Moreno, Ahmed Awada

------------------------------------------------------------------------------------------------------------------------------------------

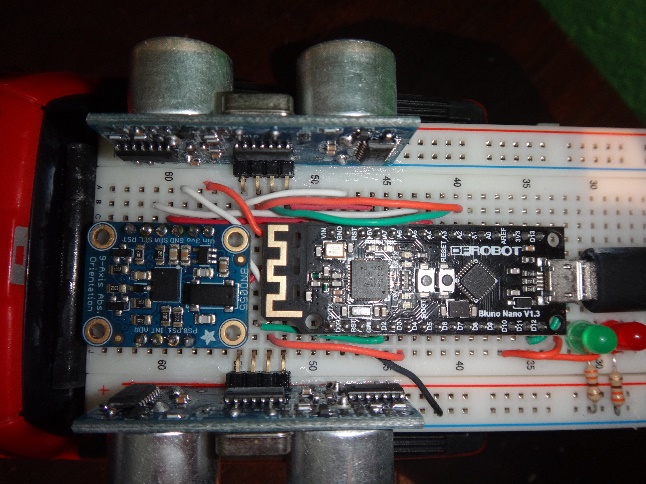
**2D Floor-Mapping Car**

**Summary**

The idea for this project is to outfit a remote control (RC) toy car with ultrasonic sensors and a Bluetooth enabled microcontroller to create a two-dimensional map of its immediate surrounding by plotting points.

The sensors will be used to first scan for obstacles beside the car as it drives; second, the microcontroller (besides being in control of the sensors and their timing) will relay the distances it reads from all scans via Bluetooth to an android application currently paired and connected to the microcontroller; finally, all of the distance scans will be plotted as points on a bitmap image and compiled to create a rough map of the car’s surroundings

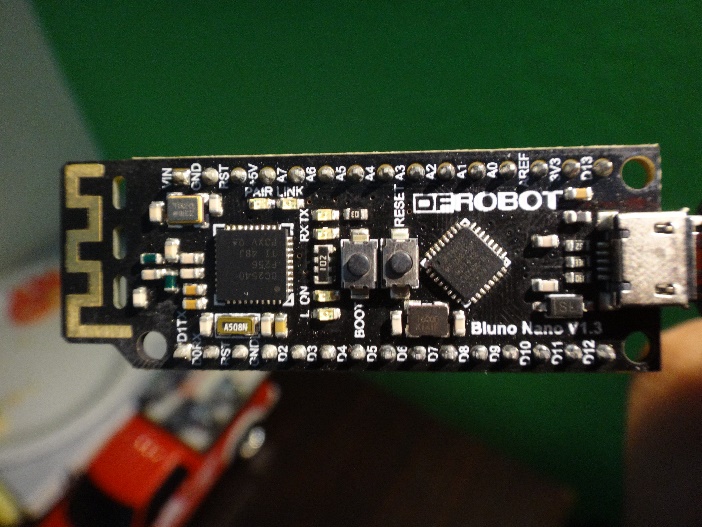
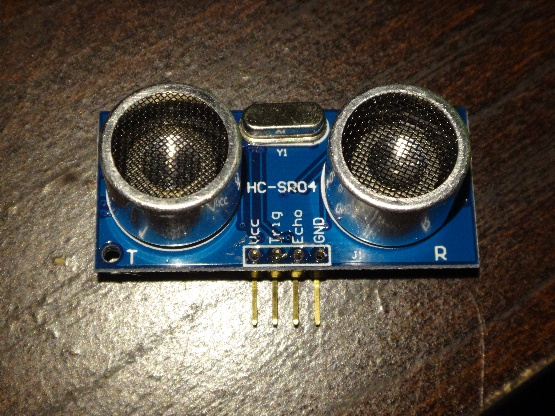
The final design and implementation is shown in the following pictures:

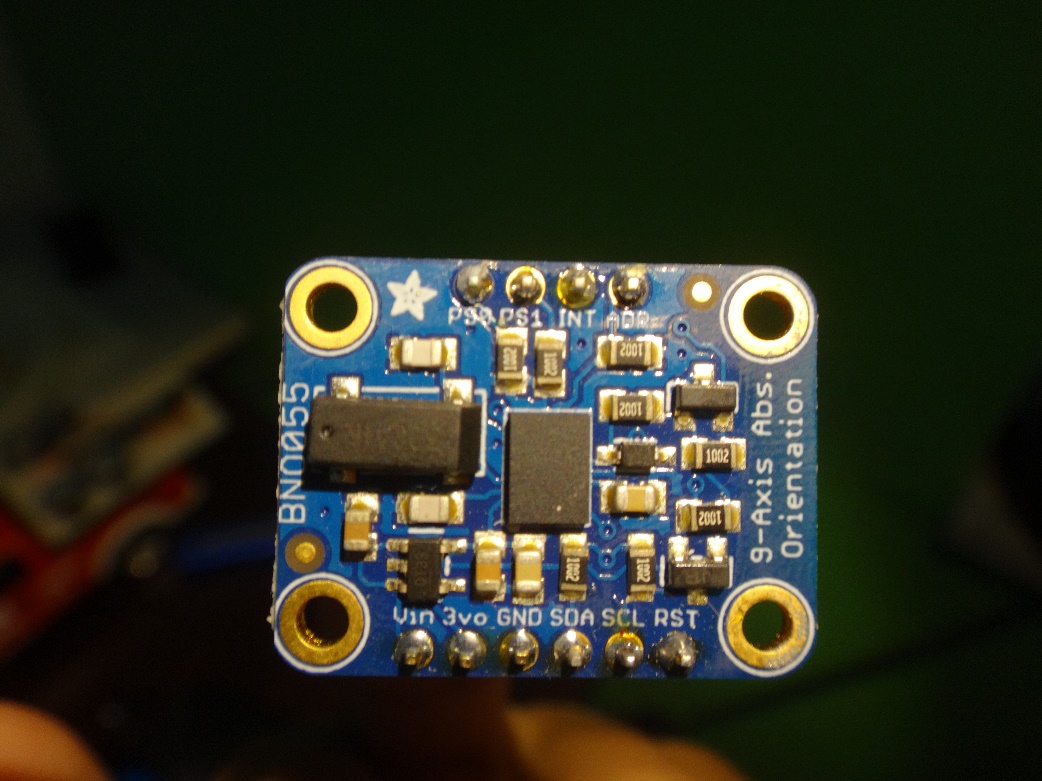
**Parts Used**

The microcontroller is the Atmega328P and it is found on our DFRobot Bluno Nano. It has Bluetooth 4.0 directly accessible and is powered via micro USB to a mobile power bank.

The ultrasonic sensors are the HC-SR04. Their pins are VCC (5v), ground, trigger, and echo. These last two pins will be seen in the software implementation section.

The third chip seen is not used in the implementation. It is the BNO055, also known as a 9-degreed-of-freedom sensor. It was being used to read the car’s linear acceleration and orientation (via Euler angles), but time ran out to actually implement it in the whole scope of the project. A little more shall be discussed in the implementation section.



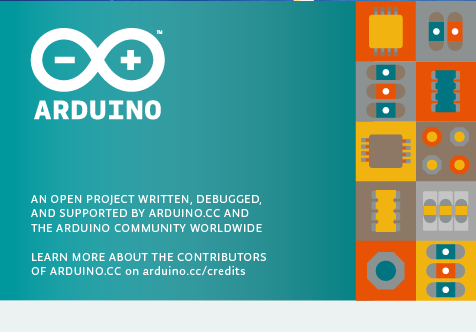
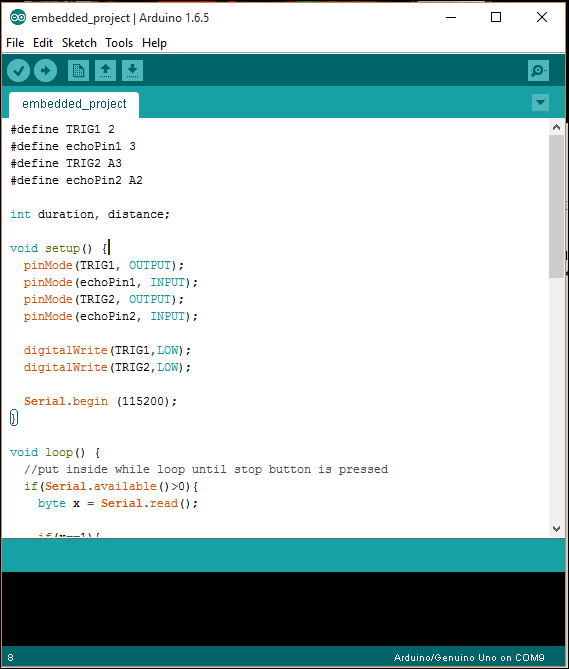
Last but not least, the battery and the RC car itself. Both are common items as they can be found at a local Wal-Mart. The specific battery was a giveaway at a campus event (Graduate College Fair). The car runs on two AA batteries and its remote control on one AAA battery. The mobile power bank is rechargeable via USB, and it also powers the entire project via micro USB, excluding the car and remote controller. The power bank outputs 5v at a current of 1A and has over 2 amp-hours in capacity.

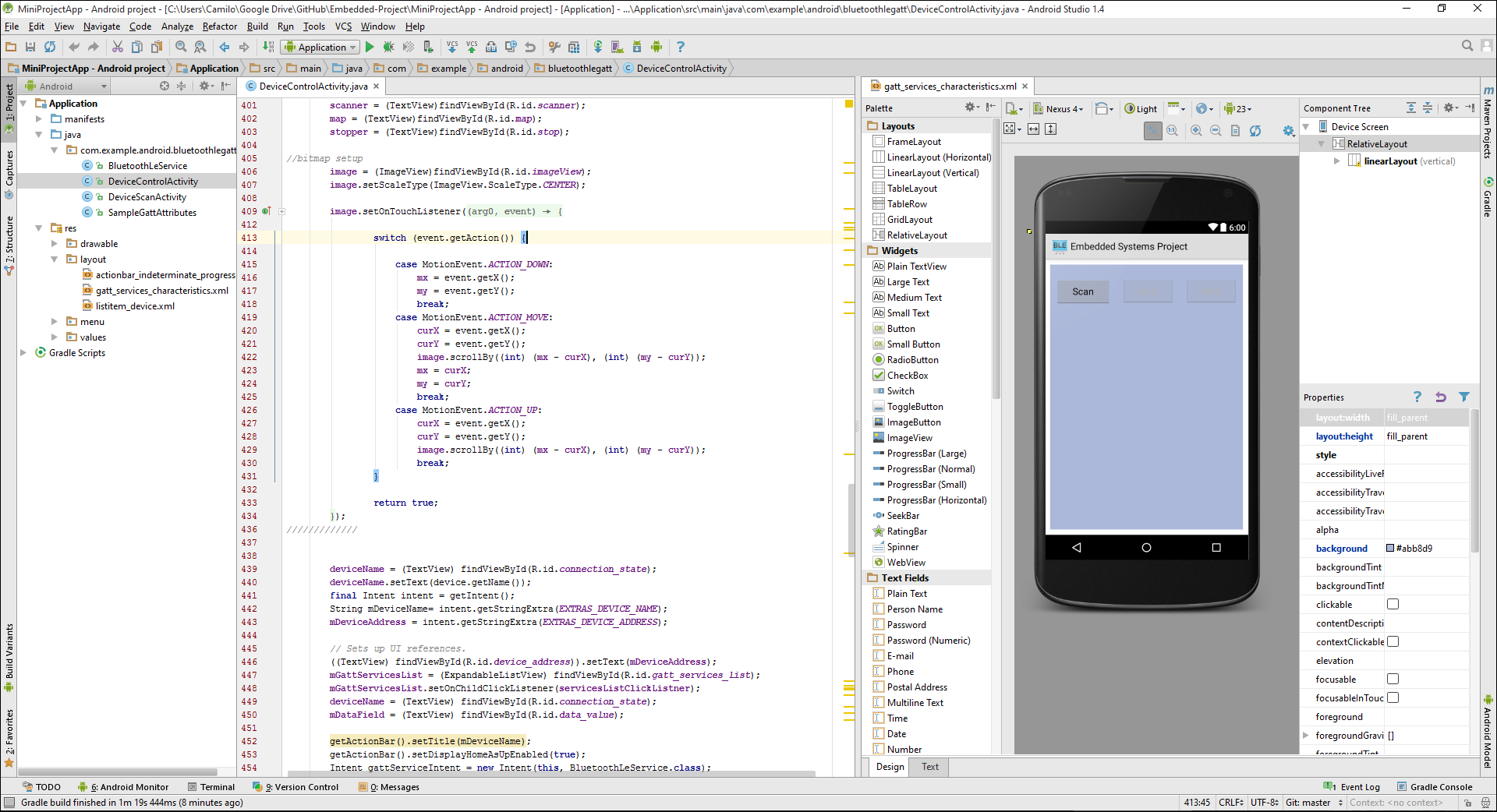


**Software Used**

The Bluno Nano has Arduino Uno’s bootloader, so it was programmed in the Arduino IDE using C/C++. The Android application was programmed using Android Studios, which is based on the IntelliJ IDEA platform, community edition. All programming was done in Java; layouts for the screens use XML.

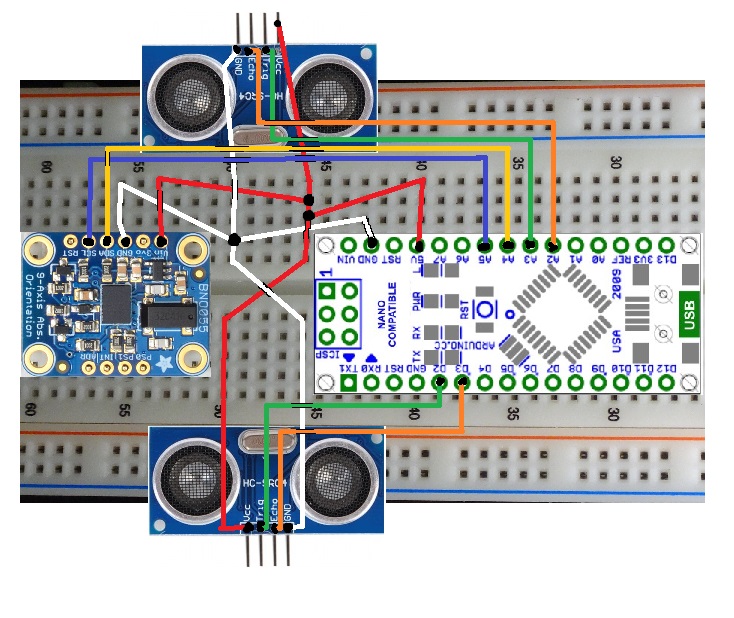
Also, it is necessary to mention that the Bluetooth setup for the android application was taken from the open source BluetoothLeGatt project on the Android-developers website. It is a sample that demonstrates how to use the Bluetooth LE GATT to transmit data between devices. All programming for this project was done in addition to this application.



**Design**

The ultrasonic sensors are connected to a small breadboard facing opposite directions away from the center of the car. The microcontroller is then connected to the breadboard; the sensor’s implementation would also require it to be plugged in here. All connections are depicted in the diagram below:



Any remote control toy car will work for the design, but we chose a pickup truck model for its flat bed in the back. Here we placed the bread board with all of its components and superglued a bent piece of tape to the toy car’s rear window and one end of the bread board.

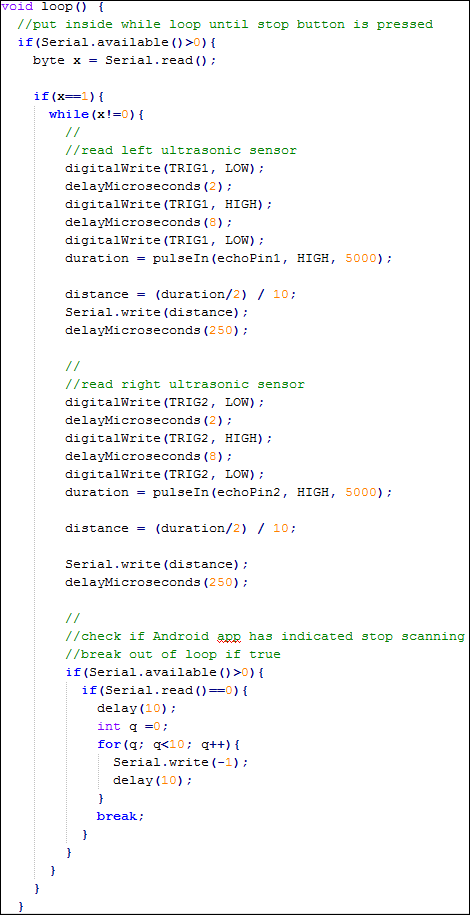
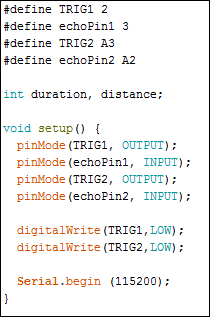
Finally, the battery pack had to be placed on the car as well, but placing it in the front would cause trouble when driving the RC car. A set of stackable pin headers that I found lying around were superglued to the base of the battery, creating a place where it could be attached to the unused rows of the breadboard.



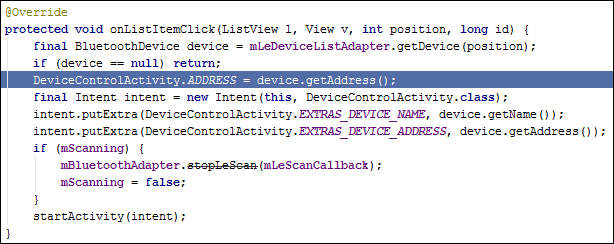
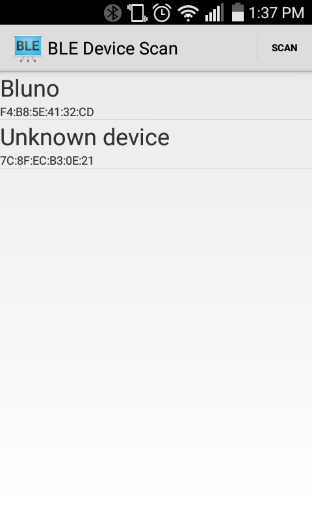
**Implementation (AKA Programming)**

All of the code used in this project is available in the zip file for this project. That being said, the Arduino code will be shown here since it is relatively short; the Java code, however, will be partially shown as snippets since there is so much of it and most of it deals with setting up the BLE connection and other functionality that is not essential or even related to this project’s implementation.

**On the microcontroller’s side**, we setup the appropriate pins for input (the ultrasonic pin ECHO) and output (TRIGGER), as well as enabling serial communication at 115200 baud rate. The Bluno will constantly check if any bytes are available from the serial receive buffer. Whenever something is sent, and it is equal to 1, we will start the ultrasonic sensors and scan the surroundings.

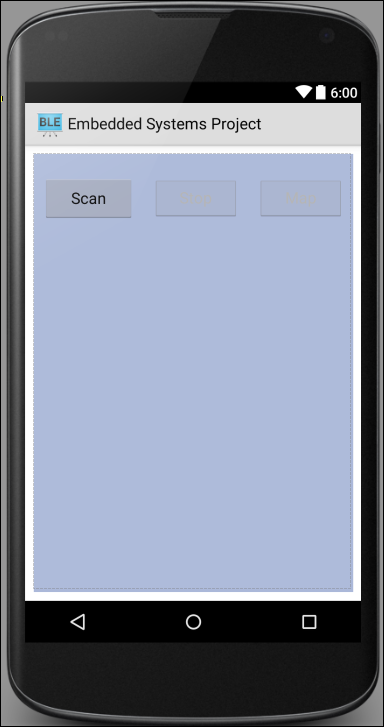
Scanning happens in a loop that continues for as long as the user (via the Android app) decides. It consists of triggering one ultrasonic sensor, measuring its delay, shortening it (through division) for the purpose used in the Android app, and then writing this “distance” (an integer) as a byte over the serial port; this process is repeated for the second sensor. This concludes a “pair” of scans; the last thing we do is check if the Android app has sent a zero over the serial connection, indicating that scanning should stop and breaking out of the loop if so. Before we break out, we write -1 a few times to the serial port so the android app knows where the sequence of the distances sent ends.

**On the Android app’s side**, there are three files with which we worked on mostly. The first is a simple one-line-of-code addition. In the file DeviceScanActivity.java, we add the following line in the onListItemClick() method:

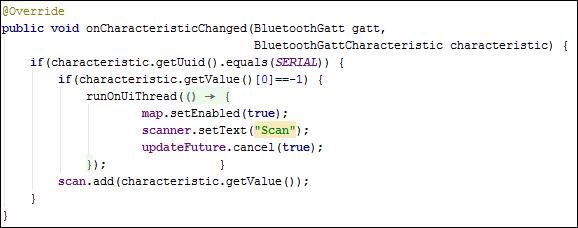
 

When we select the Bluno to select to in the first screen of the app (the righthand picture), this will update a static variable in DeviceControlActivity.java, which we will need. That’s all for that file

Next, we must make our layout with which we will interact with once we select the Bluetooth device (the Bluno). In gatt\_services\_characterisitcs.xml, we will disable all the previous content that comes from the open source app we are building on top of. In place of it, we add a vertical layout to hold our buttons and an image view to display the map once we render it. Each button will call a function, which we will discuss in the following file.



The DeviceControlActivity.java file is where the bulk of the processing occurs. The onCharacteristicChanged() method of a BluetoothGattCallback field receives all the information that the Bluno will send.



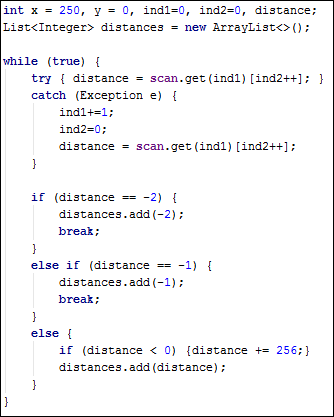
Here, we check if the Bluno sent a -1, indicating that it has finished scanning and we should now allow the user to map the completed array of distances. Regardless, every time the app receives something from the Bluno, it adds it to a byte array called *scan.*

The next method, *scan()*, is called when you press the scan button on the screen. It disables the scan button so you can’t rescan while it is scanning, enables a stop button, and also disables the map button so you don’t try to map an incomplete array of distances. Finally, we write to the Bluno a 1 to start the scanning.

The *map()* method is called from either the stop or map buttons on the screen. When called from the former, it disables the stop button and enables the scan button for rescanning. It also tells the user to wait (while the array finishes being populated) and sends the Bluno a 0 so it stops scanning.

When the array finishes populating, the map button is enabled. Pressing it will disable it once more and call our last function: *generateBMP()*.

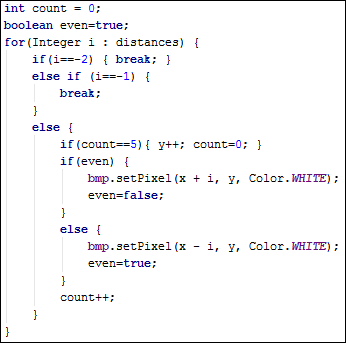
The *generateBMP()* method takes the *scan* array that contains all the distances sent by the Bluno and renders a map of the surroundings by plotting the distances as points on an image. The first part of this function makes some preliminary variables, notably, the *distances* array of Integers.



The *scan* array is actually 2 dimensional, because the way android receives information (characteristics) from Bluetooth LE is via byte arrays. Thus, the while loop iterates through the *scan* array, taking every element in each byte array and adding it to our new *distances* array. Additionally, since the Bluno sends these numbers in the range from 1 to 250, we have to add 256 to the number if it is negative. The reason behind this is that the twos-complement operation is performed on number greater than 127, since a byte is only 8 bits long and is considered signed. Also, the 1 to 250 range is not arbitrary: in the microcontroller’s code, we set a maximum delay of 5000 microseconds (5ms) for the ultrasonic to read a return signal; dividing this by 2 (since it’s a round-trip measurement) and then by 10 gives us the range (0,250).

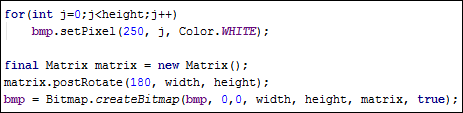
The reason for making a new array of distances is simplicity. In this next step, we plot five distances at a time (a.k.a. at the same y-coordinate value) as a sort of averaging. If we had to do the same with a 2D array, it would complicate things a little more, so we chose to recreate the contents of *scan* in the continuous array *distances*. Also, we can easily determine the length of this array before traversing it, which is what we need to set the bitmap’s height, shown here:

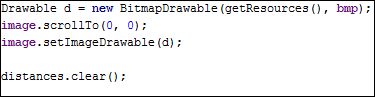


In this block of code, we traverse the *distances* array and plot each element on the bit map based on its value (which represents the distance from the ultrasonic sensor to the object it detected.

Since there are two sensors facing opposite directions, and the left sensor fire first, we set up some conditions to sequentially map the distance to the right and then to the left of a reference point.

The x-position of our reference point is fixed at the center of our 500-wide bitmap: at 250. Since the distances are all positive values, a distance on the right side of the car is added from the x-position of reference point, and vice versa.

This may seem confusing, since adding to the x value will actually make the point appear on the right half of the bitmap not the left. We have to keep in mind, however, that the y values increase downwards for the bitmap. To correct both of these issues and render an orientation-correct image (in relation to the cars orientation) on the screen, we use a rotation method on the bitmap to rotate it 180 degrees. The first couple lines also draw a reference line along the center of the bitmap to visualize the car’s path.



Finally, we set the finished bitmap as the resource for our screen image, center the image onto the screen, and clear the *distance* array to be ready for the next scan.

**Conclusion**

The ideal outcome of this project would be having the ability to drive the car around until something runs out of battery or the android app runs out of memory, then map everything out nicely to have corners, tables, chairs, shoes, backpacks, etc. depicted in 2D fashion. However, without implementing the 9 degrees-of-freedom sensor, there is no way to track turning, orientation, or even position and distance traveled.

As a result, the outcome is that the car must stay as close to a straight path as possible while being driven, otherwise the truth about the position of objects being shown in the map is shattered and you can’t tell a ceiling from a wall (which is an exaggeration). The image is also pretty rough, due to there being no active filtering algorithm implemented to remove excess points.

That being said, however, the final result does speak some extent of the truth. On the next page, you will see a little obstacle course I set up for the 2D floor-mapping car; I have labeled the obstacles 1-10 for comparison and also added a white line for the approximate path the car traveled on. After that page, the results of the car scanning through the obstacles is shown with the same labeling, 1-10.

