# Cell Phone Robot

Submitted to

George T.-C.Chiu

Professor of Mechanical Engineering

E-mail: gchiu@purdue.edu

Submitted by

Ahmed Khalil & Chukwuebuka Emmanuel Efobi

Undergraduate Research Assistant

E-mail: khalil@purdue.edu

E-mail: cefobi@purdue.edu

Purdue University

School of Mechanical Engineering

Summarizing progress for:

January 2016 to May 2016

# Table of Contents


**Executive Summary:** ..... 7

**Introduction:** ..... 8

**Related Work:** ..... 8

**Hardware Setup:** ..... 10

**Robot:** ..... 10

**Assembly:** ..... 12

**Connections:** ..... 15

**Code Logic:** ..... 16

**Code Setup:** ..... 18

Dependencies: ..... 18

Android Manifest: ..... 18

**TaskActivity:** ..... 20

Extensions & Implementations: ..... 21

Public variables: ..... 21

Methods & Classes: ..... 22

**NavigationService:** ..... 28

Extensions & Implementations: ..... 29

Public Variables: ..... 29

Methods & Classes: ..... 29

**LocationHelper:** ..... 40

Extensions & Implementations: ..... 40

Public Variables: ..... 40

Methods: ..... 40

**IOIOClass:** ..... 41

Extensions & Implementations: ..... 41

Public Variables: ..... 41

Constructor: ..... 42

Methods: ..... 43

**IncomingSms:** ..... 46

Extensions & Implementations: ..... 46

Public Variables: ..... 46

Constructor: ..... 46

Methods: ..... 47

**DataParser:** ..... 48

Extensions & Implementations: ..... 48

Public Variables: ..... 48

Methods: ..... 49

**Prior to Using the App:** ..... 51

**Bill of Materials:** ..... 53

**Future and Improvements:** ..... **53**

**Conclusion:** ..... **54**

**References:** ..... **54**

**Appendix:** ..... **55**

I. **Connections Diagram:** ..... **55**

# Table of Figures

Figure 1: Front view of robot assembly ..... 10
Figure 2: Back view of robot assembly ..... 11
Figure 3: Robot platform setup ..... 11
Figure 4: View of the second platform ..... 12
Figure 5: Makeblock instructions page 1 ..... 12
Figure 6: Makeblock instructions page 2 ..... 13
Figure 7: Makeblock instructions page 3 ..... 13
Figure 8: Makeblock instructions page 4 ..... 13
Figure 9: Makeblock instructions page 5 ..... 14
Figure 10: Limiting charge current ..... 15
Figure 11: Code Flowchart ..... 17
Figure 12: App dependencies ..... 18
Figure 13: App's permissions ..... 19
Figure 14: App's manifest ..... 20
Figure 15: TaskActivity flowchart ..... 21
Figure 16: TaskActivity class definition ..... 21
Figure 17: TaskActivity public variables ..... 22
Figure 18: onCreate() code snippet ..... 22
Figure 19: BroadcastReciever() code snippet ..... 23
Figure 20: Lifecycle methods code snippet ..... 24
Figure 21: updateUI() code snippet ..... 25
Figure 22: onVideoRecorded() code snippet ..... 25
Figure 23: onPhotoTaken() code snippet ..... 26
Figure 24: onRecieverEvent code snippet ..... 26
Figure 25: onSmsRecieved() code snippet ..... 27
Figure 26: callNavigationService() code snippet ..... 27
Figure 27: NavigationService lifecycle ..... 28
Figure 28: NavigationService extensions and implementations ..... 29

Figure 29: NavigationService public variables ..... 29
Figure 30: onCreate() code snippet ..... 30
Figure 31: onStartCommand() code snippet ..... 30
Figure 32: startNavigation() code snippet ..... 31
Figure 33: startNavigation() code snippet continuation ..... 32
Figure 34: calculateDistance() code snippet ..... 32
Figure 35:positionArray() code snippet ..... 33
Figure 36: getUrl() code snippet ..... 33
Figure 37: downloadUrl() code snippet ..... 34
Figure 38: onBind() code snippet ..... 34
Figure 39: FetchUrl code snippet ..... 35
Figure 40: ParserTask code snippet ..... 35
Figure 41: ParserTask code snippet continuation ..... 36
Figure 42: buildGoogleApiClient() code snippet ..... 36
Figure 43: onConnected() code snippet ..... 36
Figure 44: onLocationChanged() code snippet ..... 37
Figure 45: onConnectionSuspended() code snippet ..... 37
Figure 46: onConnectionFailed() code snippet ..... 37
Figure 47: onDestroy() code snippet ..... 37
Figure 48: startCompass() code snippet ..... 37
Figure 49: onSensorChanged() code snippet ..... 38
Figure 50: bearing() code snippet ..... 39
Figure 51: onAccuracyChanged() code snippet ..... 39
Figure 52: LocationHelper flowchart ..... 40
Figure 53: LocationHelper definition ..... 40
Figure 54: initocation() code snippet ..... 40
Figure 55: IOIOClass flowchart ..... 41
Figure 56: IOIOClass definition ..... 41
Figure 57: IOIOClass public definitions ..... 42

Figure 58: IOIOClass constructor ..... 42
Figure 59: setup() code snippet ..... 43
Figure 60: loop() code snippet ..... 44
Figure 61: setMotion() code snippet ..... 45
Figure 62: createIOIOLooper() code snippet ..... 45
Figure 63: getIOIOAndroidApplicationHelper() code snippet ..... 45
Figure 64: IncomingSms flowchart ..... 46
Figure 65: IncomingSms definition ..... 46
Figure 66: IncomingSms public variables ..... 46
Figure 67: IncomingClass constructor ..... 46
Figure 68: onRecieve() code snippet ..... 47
Figure 69: DataParser flowchart ..... 48
Figure 70: parse() code snippet ..... 49
Figure 71: decodePoly() code snippet ..... 50
Figure 72: Permissions screenshot ..... 51
Figure 73: USB Debugging off ..... 51
Figure 74: First prompt ..... 52
Figure 75: Second prompt ..... 52
Figure 76: Bill of Materials ..... 53
Figure 77: Connections Diagram ..... 55

## Executive Summary:

The world of robotics is quickly expanding. Research into robotics has led to applications in many industries and situations inside and outside the world. This research paper aims to further said research by investigating the possibility of having robots interact with physical world and record its experience on social media, just as humans do today. This research project attempts to answer the question, 'Is it is possible to have a robot interact so smoothly with people on social media that they would assume they were interacting with another human?' This paper details an investigation into robot interaction with physical world and with social media upon command.

To undergo this exploration, the robot was built using a robot kit and an android phone. The aim is to have the robot accept a location command via text message, plot a course to its destination, reach that destination, and take a picture of said destination. The robot would then upload that image onto a social media website. The robot built could carry out its task, but there are many possible improvements.

One of such improvements could be in the size, sturdiness and agility of the wheels and motors of the robot. This would make movement over various surfaces smoother and faster. Another improvement could be in the location code for the GPS. The current code can be improved to get a more accurate location and plot a path to its destination faster. Also, the robot's logic could be enhanced to follow the path much faster. Other improvements may include the ability to navigate indoors, interpret complex instructions, go to multiple locations and take multiple pictures depending on the situation and on prior instructions. Additionally, the code structure of the code could become more modular to allow for a more flexible design. That is to allow for future expansion and improvements of the code easily. Moreover, a future goal for the app written for the purposes of this project is to be able to identify the type of robot it is attached to and control it autonomously. It is also desired for the robot to have obstacle avoidance to allow for better navigation.

## Introduction:

Apart from regular online computer programmed robots and robots used in special functions such as for reconnaissance of dangerous or hard to reach areas, there are very few cases of robots sending information online, especially via social media. Yet smartphones, on which many robot programming projects have been based in recent years, are used daily by people to access and interact with social media sites. It seems only natural that robots can use smartphones for this purpose as well.

One major goal of this project is to explore the possibility of having a robot that can interact autonomously with the world around it and with social media websites. The driving belief is that it is possible to create a robot that can share its experience online. This research project consists of building a robot and programming it to receive a command to move to a certain location over text message, take a picture of its destination, and upload it online to a social media website such as Facebook or Twitter. This creates a method whereby the possibility of owning a personal robot that can interact with social media perhaps in the stead of the actual human user, can be assessed.

The robot was built using a premade robot kit. Most of its functionality was implemented using android code via a IOIO board. The code was broken down into different parts each with a different function as follows; reception and interpretation of SMS or text messages, movement for the motors, reception and interpretation of SMS or text messages, GPS location and path creation ability, camera access and picture taking ability, and online access with picture and text post creation ability.

## Related Work:

Exploration into similar technology led to the discovery of various interesting projects on robotics. The most relevant of these projects are included in this section. Although many share parts of the functionality required for this project, none perfectly matched the requirements.

Android based robotics (ABR) is a part of a much larger research project run by the Cognitive Anteater Robotics Laboratory. The ABR program is aiming to use Android phones in combination with robotic vehicles for search and rescue purposes. As one of their sub-projects, they built a leader robot and an autonomous following robot. The leader is controlled remotely using an android phone, while the follower uses visual based coding to follow a green balloon mounted on the leader. Both robots use IOIO boards as an interface between the phones and the robotic sensors and actuators. Their website provides many documents and links related to hardware and software configuration.

The ABR project is related the project detailed in this paper in certain terms, but it has some missing functionality. In terms of it sturdiness and ability to move around on terrain that is imperfect, this is a good project to emulate as the movement functionality will give the robot the ability to move around easily without getting stuck between gaps in outdoor tiles or getting stuck in ditches of indentations in the ground for example. Also, the ABR robot can track objects; this ability only comes close to this paper's robot's need for obstacle avoidance but is not exact. The mounted android phone can swivel and tilt this means that the robot can scan a view of its entire surrounding for a desired object, a useful ability to have when sending the robot to a certain location.

This robot differs in the sense that its movement is not fully autonomous, without an object to follow it cannot function. This paper's robot must be GPS capable and follow a path it has determined on its own without needing to maintain any communication with the commanding phone.

Another project that was evaluated as part of the preliminary research was posted on hackaaday.IO by Jason Bowling. In this project, a user sends commands from a computer or tablet device to an android phone connected to an IOIO board, which interfaces with a robot. The idea is to receive sensor information from the robot and then send back inputs. The phone had a simple socket server running on it to send and receive signals over the internet.

In this case, the robot was remotely controlled, but their design had PWM throttles to the motors allowing for better control of the robot, a useful addition to the design of the robot discussed in this paper. Another useful and very attractive quality of this robot is its ability to send and receive data over the internet; this kind of functionality was of paramount importance to the completion of their project. Since their robot is remotely controlled, the user must always remain in contact with the robot, this lack of autonomy makes their project much different from what this current project aims to achieve.

Yet another research project encountered was a smartphone controlled autonomous robot. The goal of the researchers here is develop a better and cheaper robot for students to experiment with. Therefore, they decided to build a robot that is controlled by android phones as they are very accessible and have many useful capabilities. The robot is a car that can travel between two points while avoiding obstacles on its course. The android phone is mounted on the robot itself and is used as the microcontroller. A IOIO board is used as the hardware to software interface by connecting the phone to the robot's motor actuators.

# Hardware Setup:

## Robot:

The robot must be able to drive around between buildings with ease. It must be stable, durable, and able to travel across multiple terrain. The robot was built using a Makeblock robot kit and a few additional parts. The kit includes multiple beams of various sizes. The beams are very sturdy and include multiple threaded and unthreaded holes for flexible assembly of the chassis. Furthermore, the kit includes two 25mm DC motors, so an additional two had to be purchased. A full assembly of the robot chassis is shown in Figures 1 and 2. The four-wheel drive provides the robot with enough torque to traverse the streets with ease.

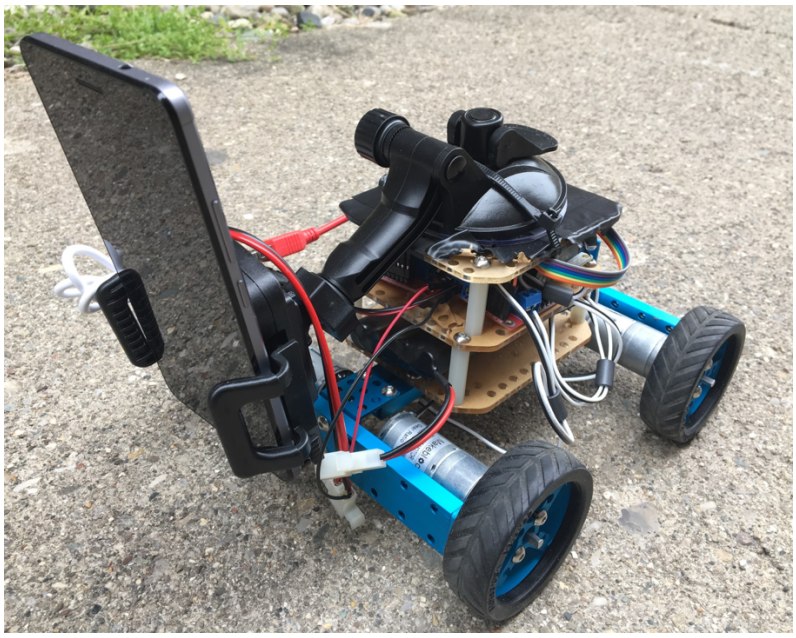

*Figure 1: Front view of robot assembly*

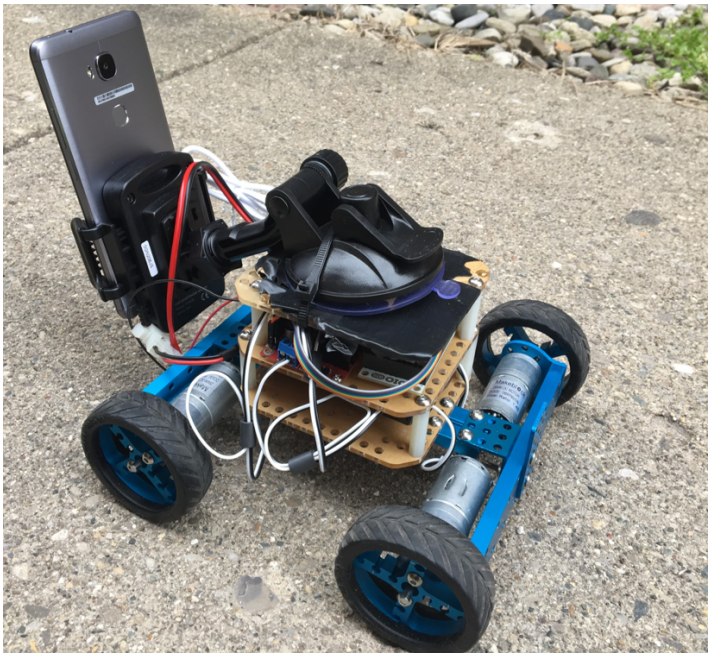

*Figure 2: Back view of robot assembly*

After the chassis was built, three platforms were added as shown in Figure 3. The first platform is for the 9.6V battery, the second is for the IOIO OTG board and two L298N drivers, and the third is for the Android phone holder.

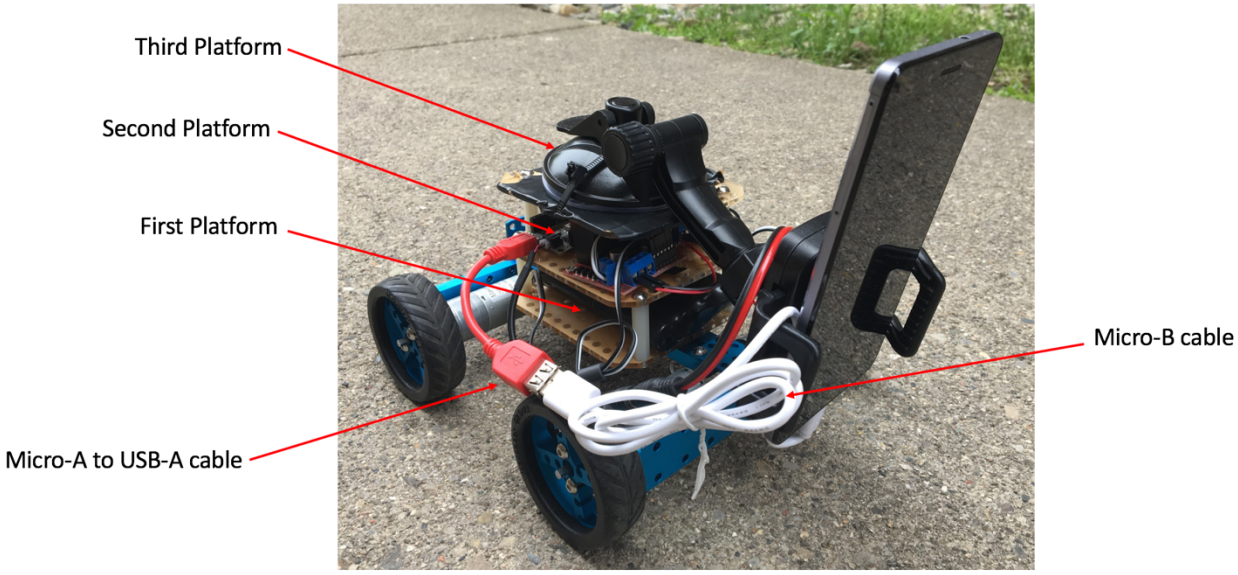


*Figure 3: Robot platform setup*

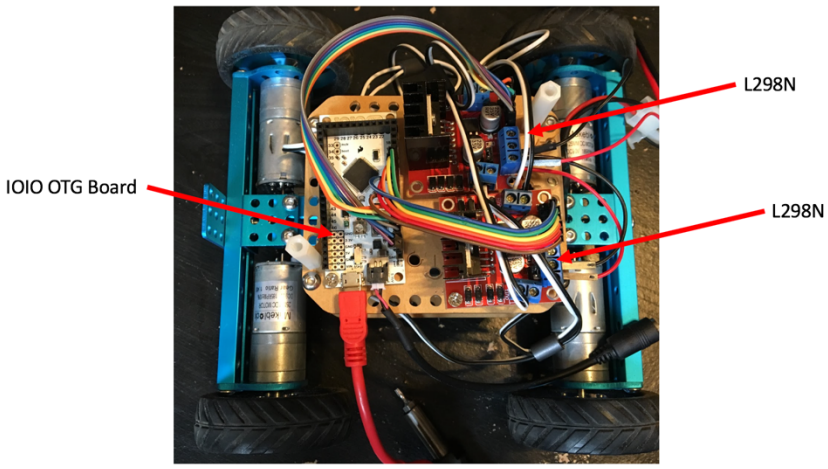


*Figure 4: View of the second platform*

## Assembly:

The assembly instructions are from the Makeblock robot kit. There is a slight addition to their set of instructions, which will be mentioned at the end. The instructions can be found under the link below; however, they will be included in this report as well.

https://github.com/Makeblockofficial/AssemblyInstructions/blob/master/Starter_Robot_Kit_Bluetooth_Instruction.pdf

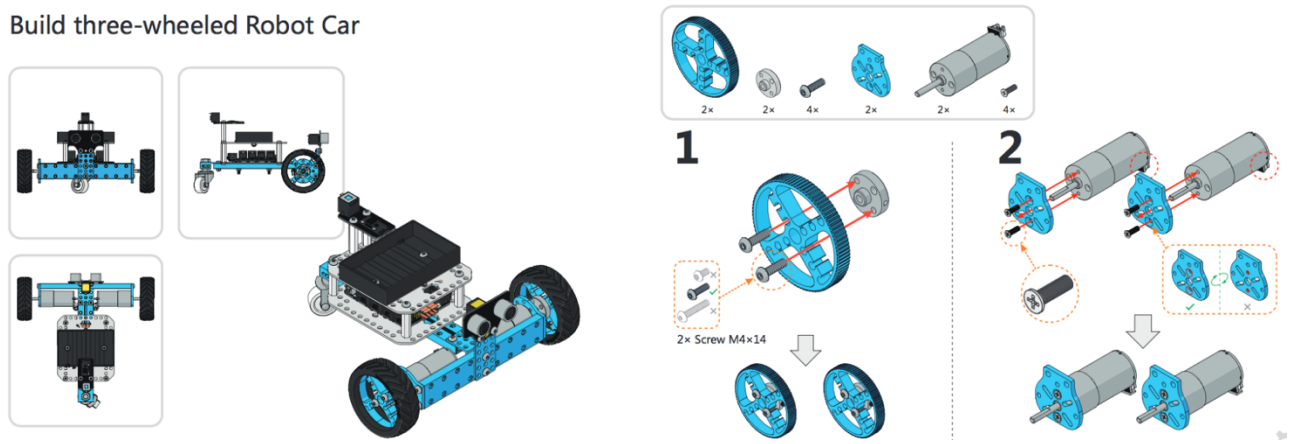


*Figure 5: Makeblock instructions page 1*

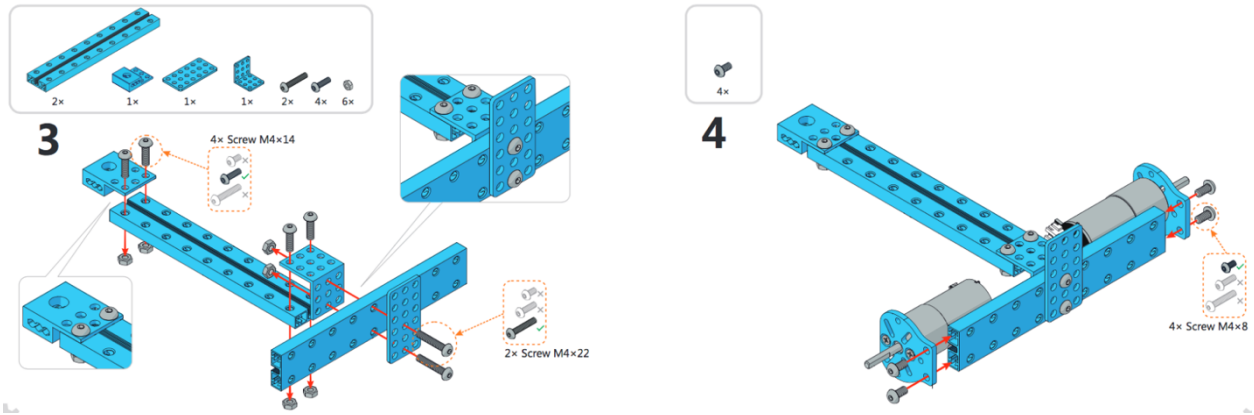


*Figure 6: Makeblock instructions page 2*

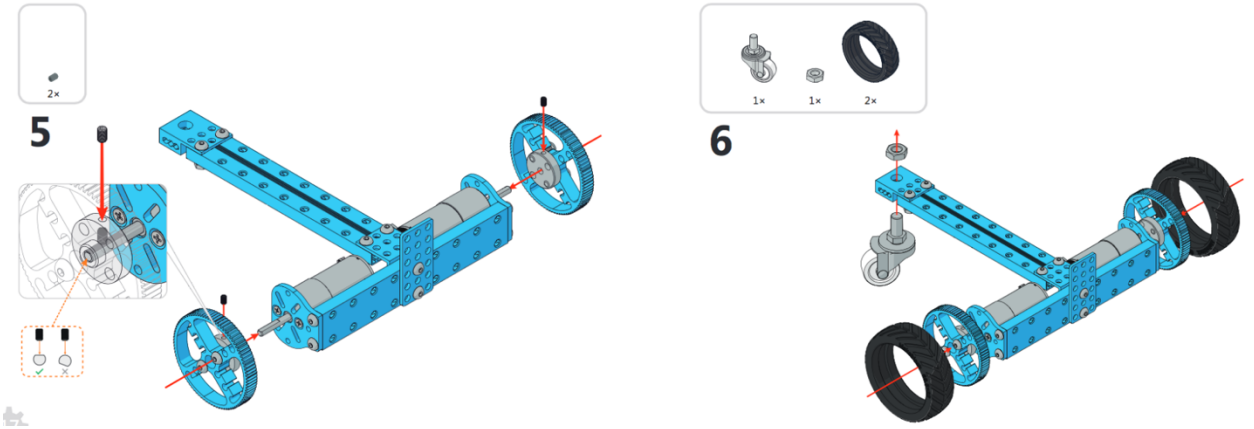


*Figure 7: Makeblock instructions page 3*

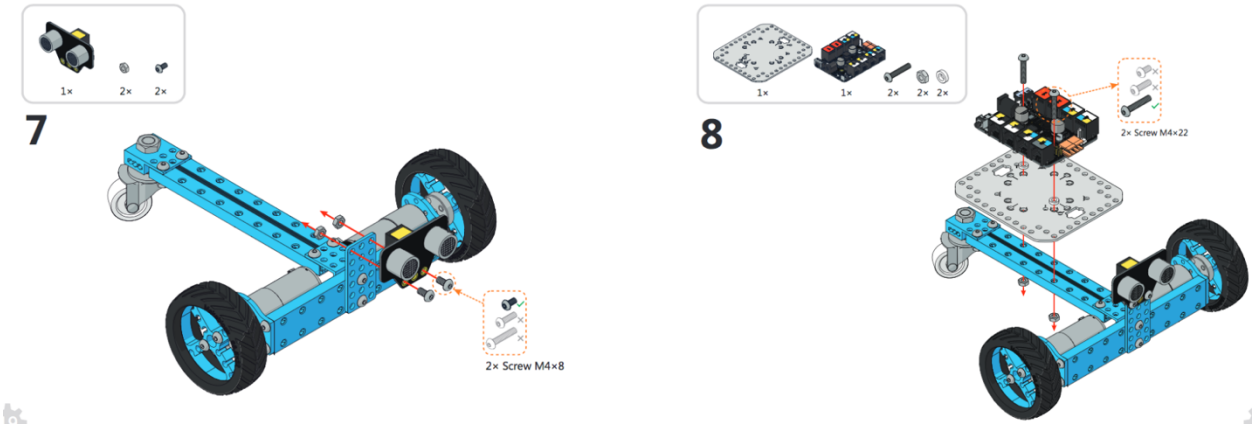


*Figure 8: Makeblock instructions page 4*

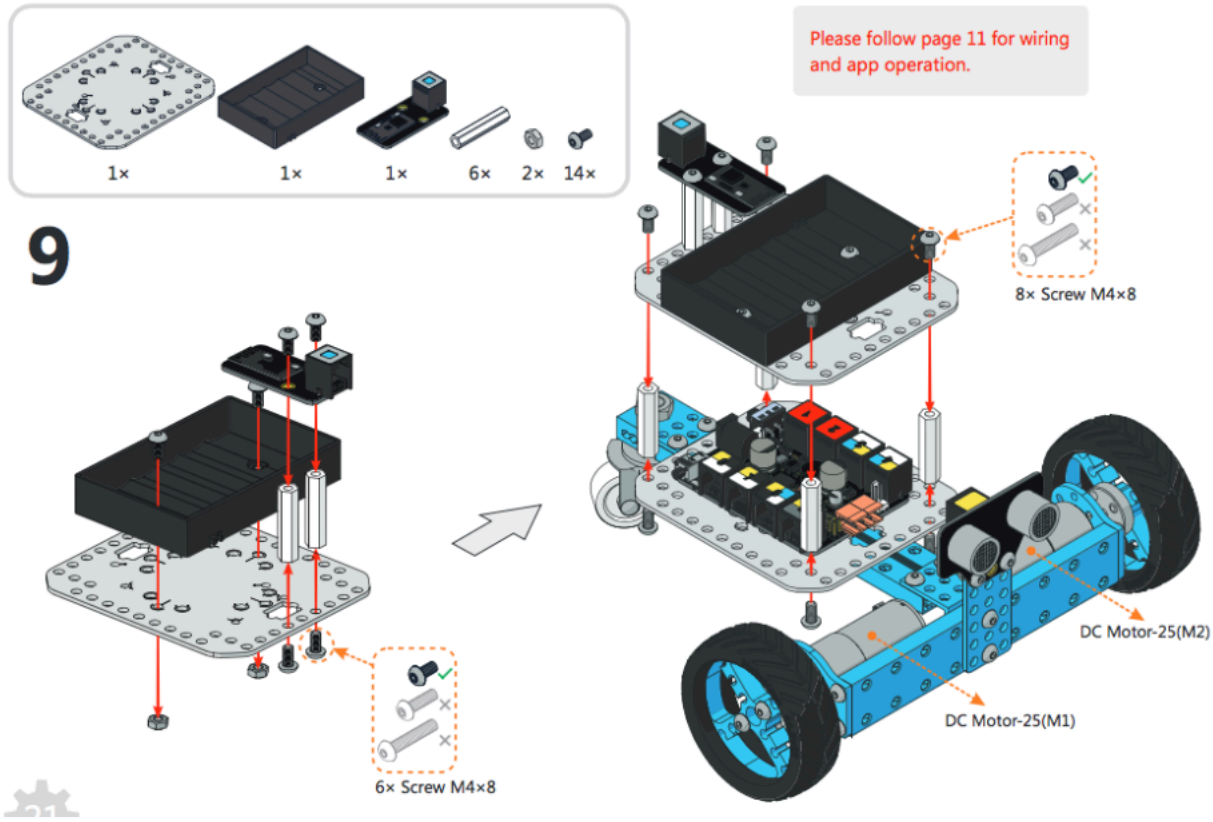


*Figure 9: Makeblock instructions page 5*

The differences between this version of the robot and the one built are:

- The one built has four wheels instead of three. The back two wheels are identical to the front two.
- The one built has three platforms rather than just two.

## Connections:

The schematic shown in Appendix I represents the circuitry connections. There are two L298N motor drivers for the four robots, two motors per driver. The drivers are then connected to the IOIO board, which is connected to the Android phone. The schematic shows a different version from the IOIO board used. Figure 3 shows the connection between the IOIO OTG board and the android phone. A micro-A to USB-A cable and a micro-B cable were used in tandem to achieve the connection.

There are a few points that must be kept in mind to achieve a viable connection between the IOIO and its respective phone. The "USB Debugging" must be off in the Developer Options on the phone. Additionally, the limiting charge current must be clockwise, as shown in Figure 4, to allow USB connection to fall through.

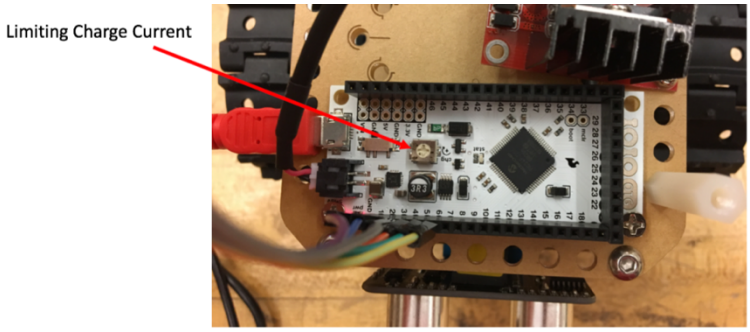


*Figure 10: Limiting charge current*

## Code Logic:

The code is broken down into six classes: *TaskActivity*, *NavigationService*, *LocationHelper*, *IOIOClass*, *IncomingSms*, and *DataParser*. The classes work together in tandem to achieve full functionality for the app. The flow chart shown in Figure 11 provides an overview of how the classes interact with one another and the lifecycle of the app.

When an SMS is received, the code is triggered to start navigating. The SMS should include a location's name, which could be a building, a street, etc. The robot will use this SMS to file a request to Google Directions API obtaining a navigation route from the robot's current GPS location to the destination. The route returned by Google is converted to an array of Location points. The robot takes the next point from its location, finds its bearing to that point, and then reacts accordingly. Based on the bearing, the robot would rotate clockwise, anticlockwise, or move forwards. To allow the robot to obtain its location and provide accurate results, this entire cycle occurs every four seconds. In the future, this cycle will be optimized and the *NavigationService* will be called more frequently to allow the robot to move seamlessly.

In this segment of the report, each class will be described as well as the methods within. Code snippets will be shown and explained; however, although these snippets are enough to guide the reader into replicating the app from scratch, it is recommended to also look at the original code itself. Furthermore, before proceeding any further with the report it is highly recommended to look at the code flow chart in Figure 11, because even though each class is individually explained, the overall interaction between the classes is only fully understood through the flowchart.

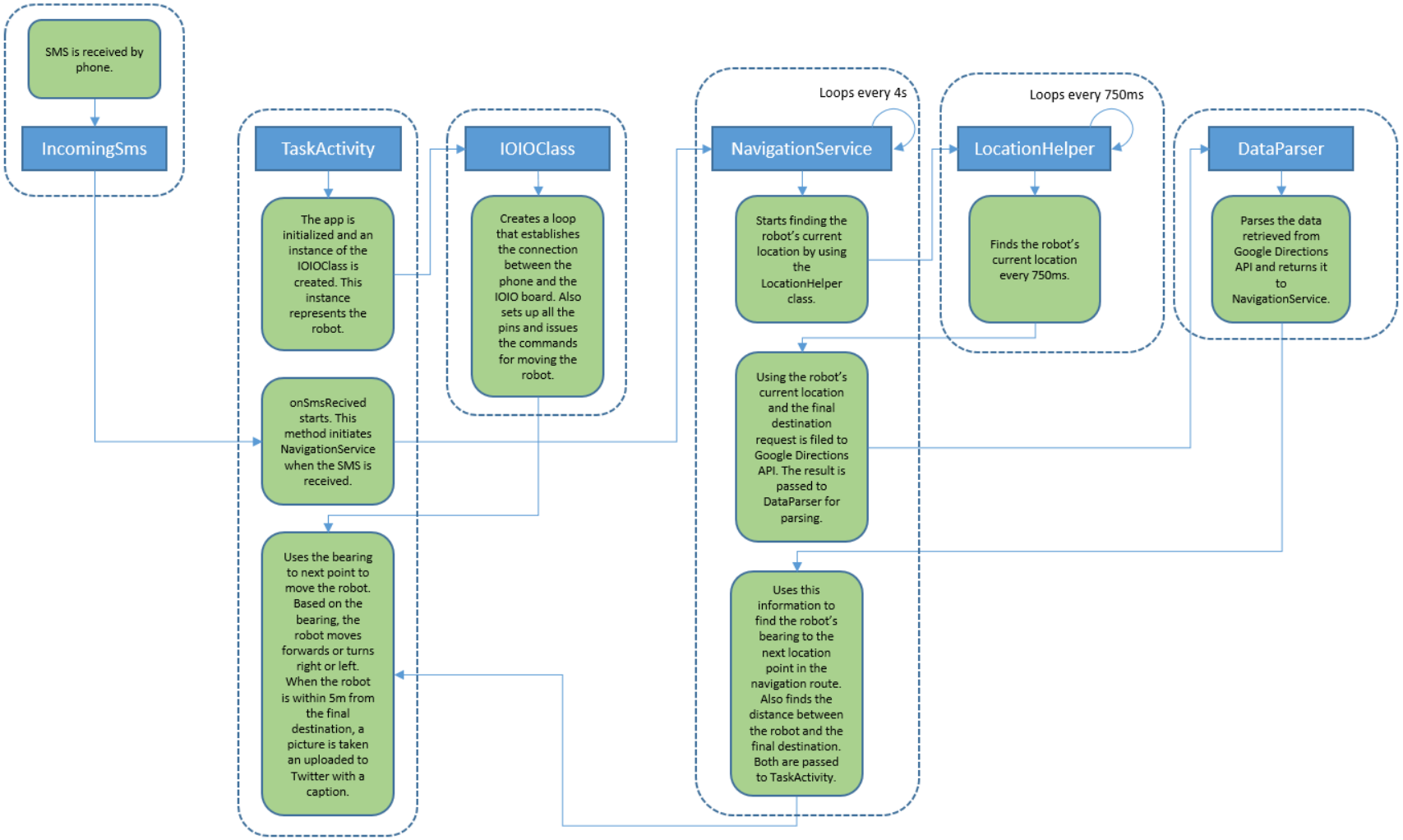


*Figure 11: Code Flowchart*

## Code Setup:

This section includes the different tasks that had to be done prior to creating any of the classes used in this app. Without these preparations, none of these classes will function.

## Dependencies:

Many of the functionality and methods used in this app are externally imported. These imports are responsible for most of the app's functionality such as:

- Google Play Services, which allow the Google API interactions
- The IOIO board libraries, which allow the app to communicate with the IOIO OTG board
- EventBus library, which facilitates class to class transaction of data
- Twitter library, which allows the app to access Twitter and post an image
- Camera fragment library, which allows the app to access the camera
- Basic android libraries, which allow the app to perform basic tasks

The code snippet below is extracted from the *build.gradle* file, it shows all the dependencies used in the construction of this app.

```
22  dependencies {
23      compile fileTree(include: ['*.jar'], dir: 'libs')
24      androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
25          exclude group: 'com.android.support', module: 'support-annotations'
26      })
27      compile 'com.android.support:appcompat-v7:25.3.1'
28      compile 'com.android.support.constraint:constraint-layout:1.0.0-beta5'
29      compile 'com.google.android.gms:play-services-location:10.2.1'
30      compile 'com.google.firebase:firebase-messaging:10.2.1'
31      compile 'com.google.android.gms:play-services-maps:10.2.1'
32      compile 'com.google.maps.android:android-maps-utils:0.5+'
33      compile 'org.greenrobot:eventbus:3.0.0'
34      testCompile 'junit:junit:4.12'
35      compile 'com.github.ytai.ioio:IOIOLibAndroidDevice:5.07'
36      compile 'com.github.ytai.ioio:IOIOLibAndroidBluetooth:5.07'
37      compile 'com.github.ytai.ioio:IOIOLibAndroidAccessory:5.07'
38      compile 'com.github.ytai.ioio:IOIOLibAndroid:5.07'
39      compile 'com.github.ytai.ioio:IOIOLibCore:5.07'
40      compile 'com.yayandroid:LocationManager:2.0.1'
41      compile 'com.github.florent37:camerafragment:1.0.4'
42      compile 'org.twitter4j:twitter4j-core:4.0.6'
43  }
```

*Figure 12: App dependencies*

## Android Manifest:

This file is responsible for ensuring all the necessary permissions for the app are present. These permissions are necessary for all the GPS and Google API interactions, as well as the Camera and internet access. The snipped below shows the all the necessary permissions.

```
    <uses-permission android:name="android.Manifest.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.FLASHLIGHT" />
    <uses-permission android:name="android.permission.RECORD_AUDIO" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

*Figure 13: App's permissions*

Android manifest is also responsible for storing important values regarding the app such as its name, icon, theme, and google API key. The google API key is what allows the app to request data from the Google Directions API. This key can be created by going to: https://console.developers.google.com/apis/dashboard?project=i-like-buffalo-wings&duration=PT1H

Once the key is created, it is important to provide it with the necessary access. Google has multiple tutorials on achieving this and it is a very simple step.

Furthermore, in the manifest file the main activity is defined, as well as any classes that are services, file intents, or receive broadcasts. The code snippet below shows all of what has been mentioned.

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">

    <service android:enabled="true" android:name=".NavigationService" />
    <activity android:name=".TaskActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <!--
         The API key for Google Maps-based APIs is defined as a string resource.
         (See the file "res/values/google_maps_api.xml").
         Note that the API key is linked to the encryption key used to sign the APK.
         You need a different API key for each encryption key, including the release key that is used to
         sign the APK for publishing.
         You can define the keys for the debug and release targets in src/debug/ and src/release/.
    -->
    <meta-data
        android:name="com.google.android.geo.API_KEY"
        android:value="@string/google_maps_key" />

    <receiver
        android:name=".IncomingSms"
        android:enabled="true"
        android:exported="true">
        <intent-filter>
            <action android:name="android.provider.Telephony.SMS_RECEIVED"></action>
        </intent-filter>
    </receiver>
</application>
```

*Figure 14: App's manifest*

## TaskActivity:

*TaskActivity* is the app's Main Activity. Here the app's view is set up, an instance of *IOIOClass* is created and controlled, *NavigationService* is called, the picture is taken, and is uploaded to Twitter. This is the app's main hub. The instance of the *IOIOClass* represents the robot itself, by controlling this instance, the activity controls the actual robot's movements. *NavigationService* is the service class responsible for obtaining all the GPS coordinates and routes needed to move the robot.

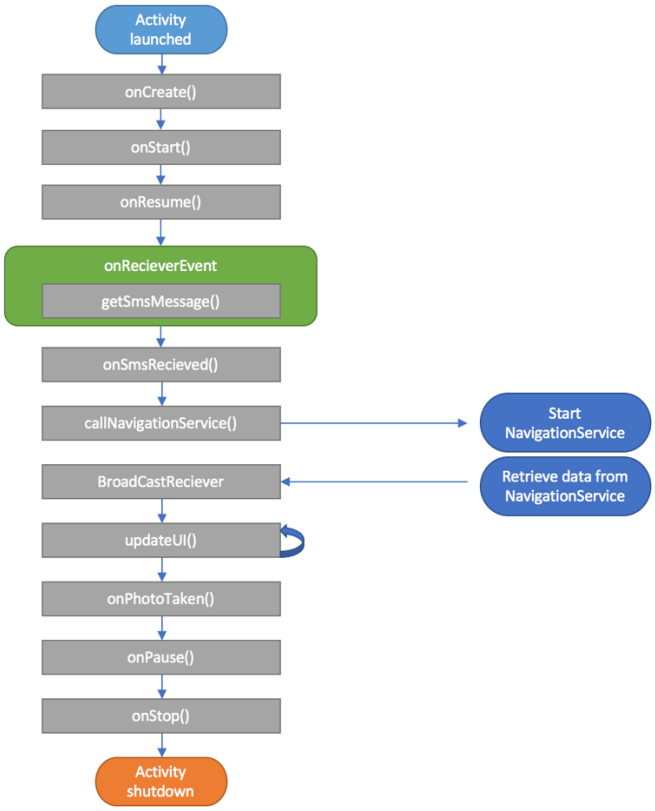


*Figure 15: TaskActivity flowchart*

## Extensions & Implementations:

Given that this is the Main Activity, it necessary that it extends *AppCompatActivity*, which allows the app to use the Support Library. This library allows the app to utilize functionality from earlier or newer versions of the Android platform.

*TaskActivity* has only one implementation, *CameraFragmentResultListener*. To use this implementation some methods must be included in the code and overridden; these are: *onVideoRecorded()* and *onPhotoTaken().* The snippet below shows the extensions and implementations used for this class.

```
40  public class TaskActivity extends AppCompatActivity implements CameraFragmentResultListener {
```

*Figure 16: TaskActivity class definition*

## Public variables:

Below is a list of the public variables used in *TaskActivity*. It is best they are defined in this section as they will be mentioned repeatedly across the code.

```
41      private Intent mIntent; // The intent that starts the NavigationService.
42      public float direction; // The robot's bearing to the next location point.
43      public String message;  // The SMS message passed in through the EventBus.
44      IOIOClass myRobot;  // An instance of the robot. A setter method will be used on this instance
45              // to move the robot.
46      public static final String FORWARDS = "forwards";   // A string command to move forwards.
47      public static final String RIGHT = "right"; // A string command to turn right.
48      public static final String LEFT = "left";   // A string command to turn left.
49      public static final String STOP = "stop";   // A string command to stop.
50      public String motionDirection = "not moving";  // An initialization of the direction the robot
51              // is moving in.
52      public final CameraFragment cameraFragment =
53              CameraFragment.newInstance(new Configuration.Builder().build()); // A camera fragment
54              // used to take the picture.
```

*Figure 17: TaskActivity public variables*

## Methods & Classes:

### *onCreate():*

This method sets up the entire app, from the different intents that will be issued throughout its lifecycle to the main XML file that is issued to the user.

```
60      @Override
61      protected void onCreate(Bundle savedInstanceState) {
62          super.onCreate(savedInstanceState);
63          setContentView(R.layout.activity_task); // Sets the XML view file that appears to the user.
64          mIntent = new Intent(this, NavigationService.class); // Associates mIntent with
65                  // NavigationService.
66          myRobot = new IOIOClass(this); // Creates the robot instance from IOIOClass.
67          myRobot.getIOIOAndroidApplicationHelper().create(); // Retrieves the IOIO helper, which is
68                  // responsible for starting the IOIO loop from another class, and creates it.
69                  // This allows TaskActivity to access the IOIOClass instance.
70
71
72          // This IF block insures the camera permission is granted.
73          if (ActivityCompat.checkSelfPermission(this, Manifest.permission.CAMERA) != PackageManager.
74                          PERMISSION_GRANTED) {
75              // TODO: Consider calling
76              //    ActivityCompat#requestPermissions
77              // here to request the missing permissions, and then overriding
78              //   public void onRequestPermissionsResult(int requestCode, String[] permissions,
79              //                                          int[] grantResults)
80              // to handle the case where the user grants the permission. See the documentation
81              // for ActivityCompat#requestPermissions for more details.
82
83              Toast.makeText(this, "Grant camera permission please.", Toast.LENGTH_LONG).show();
84              return;
85          }
86
87          // Attaches the camera fragment to the XML file so the user can see it.
88          getSupportFragmentManager().beginTransaction()
89                  .replace(R.id.content, cameraFragment, "TheCameraThing")
90                  .commit();
91      }
```

*Figure 18: onCreate() code snippet*

*BroadcastReciever():*

This *BroadcastReceiver* starts when an intent is sent from *NavigationActivity* to *TaskActivity*. It is responsible for receiving the data sent from *NavigationActivity*. As shown in the code snippet below, the robot stops after 150ms when the robot is turning and after 3s when the robot is going forward

```
96      private BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {
97          @Override
98          // onReceive receives the intent from NavigationService
99          public void onReceive(Context context, Intent intent) {
100             updateUI(intent); // The updateUI method is started. It needs the received intent to be
101                     // passed to it.
102
103             // This IF block stops the robot after 150ms after a command is issued for it to turn
104                     // left or right. Else if the robot is moving forwards, it will stop the robot
105                     // after 3 seconds.
106             if (motionDirection.equals(RIGHT) || motionDirection.equals(LEFT)) {
107                 // This handler block runs a certain code after a period of time, in this case it
108                         // stops the robot after 150ms and updates the TextViews on the main screen.
109                 final Handler handler = new Handler();
110                 handler.postDelayed(new Runnable() {
111                     @Override
112                     public void run() {
113                         myRobot.setMotion(STOP);
114                         motionDirection = STOP;
115                         TextView directionTextView = (TextView) findViewById(R.id.bearingToDis);
116                         directionTextView.setText("Direction: " + direction);
117
118                         TextView motionTextView = (TextView) findViewById(R.id.motion);
119                         motionTextView.setText("I am moving: " + motionDirection);
120                     }
121                 }, 150);
122             } else {
123                 // This handler block runs a certain code after a period of time, in this case it
124                         // stops the robot after 3s and updates the TextViews on the main screen.
125                 final Handler handler = new Handler();
126                 handler.postDelayed(new Runnable() {
127                     @Override
128                     public void run() {
129                         myRobot.setMotion(STOP);
130                         motionDirection = STOP;
131                         TextView directionTextView = (TextView) findViewById(R.id.bearingToDis);
132                         directionTextView.setText("Direction: " + direction);
133
134                         TextView motionTextView = (TextView) findViewById(R.id.motion);
135                         motionTextView.setText("I am moving: " + motionDirection);
136                     }
137                 }, 3000);
138             }
139         }
140     };
```

*Figure 19: BroadcastReciever() code snippet*

*Lifecycle Methods:*

These methods ensure that the app's lifecycle doesn't get messed up due to interruptions from the phone. They ensure that if the user leaves the app and returns, or if the user receives an email or a phone call, the app won't crash.

```
142     /**
143      * This method insures the app's lifecycle doesn't get messed up if the app is restarted due to
144      * a global event occurring, i.e. the phone's orientation is flipped, an SMS or an email
145      * is received, a call is incoming, etc.
146      */
147     @Override
148     public void onStart() {
149         super.onStart();
150         EventBus.getDefault().register(this);
151         myRobot.getIOIOAndroidApplicationHelper().start();
152     }
153
154     /**
155      * This method insures the app's lifecycle doesn't get messed up if the app is stopped due to
156      * a global event occurring, i.e. the phone's orientation is flipped, an SMS or an email
157      * is received, a call is incoming, etc.
158      */
159     @Override
160     public void onStop() {
161         super.onStop();
162         EventBus.getDefault().unregister(this);
163         myRobot.getIOIOAndroidApplicationHelper().stop();
164     }
165
166     /**
167      * This method insures the app's lifecycle doesn't get messed up if the app is paused due to
168      * a global event occurring, i.e. the phone's orientation is flipped, an SMS or an email
169      * is received, a call is incoming, etc.
170      */
171     @Override
172     public void onPause() {
173         super.onPause();
174         unregisterReceiver(broadcastReceiver);
175         stopService(mIntent);
176     }
177
178     /**
179      * This method insures the app's lifecycle doesn't get messed up if the app is returned to after
180      * a global event occurs, i.e. the phone's orientation is flipped, an SMS or an email is
181      * received, a call is incoming, etc.
182      */
183     @Override
184     public void onResume() {
185         super.onResume();
186         startService(mIntent);
187         registerReceiver(broadcastReceiver, new IntentFilter(NavigationService.BROADCAST_ACTION));
188     }
```

*Figure 20: Lifecycle methods code snippet*

*updateUI():*

This method is where the commands are issued to move the robot. The intent passed to this method is from *NavigationService*, which contains two parameters, direction and distance. Direction is used to orient the robot and move it in the correct direction. Distance is the distance between the robot and its destination, it is used to stop the robot when it is 5m away from the target and take a picture.

```
197     private void updateUI(Intent intent) {
198         direction = intent.getFloatExtra("direction", 0.0f); // obtains the direction parameter from
199             // the intent passed by NavigationService.
200         double distance = intent.getDoubleExtra("distance", 0.0f); // obtains the distance parameter
201         // from the intent passed by NavigationService.
202
203         // This IF block sets the robot's motion based on the next location's bearing with respect
204                 // to the robot. If the robot is within 5m from the destination, the robot will
205                 // stop and take a picture.
206         if (!(distance < 5 && distance > 0)) {
207             if (direction > 350 || direction < 10) {
208                 myRobot.setMotion(FORWARDS); // A setter method that sets the robot's motion.
209                 motionDirection = FORWARDS;
210             } else if (direction < 350 && direction > 180) {
211                 myRobot.setMotion(RIGHT);
212                 motionDirection = RIGHT;
213             } else if (direction < 180 && direction > 10) {
214                 myRobot.setMotion(LEFT);
215                 motionDirection = LEFT;
216             }
217         } else {
218             myRobot.setMotion(STOP);
219             motionDirection = "I have arrived to destination.";
220             new Handler().postDelayed(() → {
223                     cameraFragment.takePhotoOrCaptureVideo(TaskActivity.this,
224                             "/storage/self/primary", "thePicture001");
225             }, 5000);
227         }
228
229         Log.d("OUTSIDE", String.valueOf(direction));  // Used for debugging.
230
231         // This block updates the user with the phone's direction to the next point. Used for
232                 // debugging.
233         TextView directionTextView = (TextView) findViewById(R.id.bearingToDis);
234         directionTextView.setText("Direction: " + direction);
235
236         // This block updates the user with the phone's motion to the next point. Used for
237         // debugging.
238         TextView motionTextView = (TextView) findViewById(R.id.motion);
239         motionTextView.setText("I am moving: " + motionDirection);
240
241         // This block updates the user with the phone's distance to the final destination. Used for
242         // debugging.
243         TextView distanceTextView = (TextView) findViewById(R.id.distanceToFinal);
244         distanceTextView.setText("I am: " + distance + "m away.");
245     }
```

*Figure 21: updateUI() code snippet*

*onVideoRecorded():*

This method must be included for *CameraFragmentResultListener* to be implemented. This method would be used if the goal is to record a video.

```
251     @Override
252     public void onVideoRecorded(String filePath) {
253         Toast.makeText(this, "Video", Toast.LENGTH_SHORT).show();
254     }
```

*Figure 22: onVideoRecorded() code snippet*

*onPhotoTaken():*

This method is triggered when a picture is taken. Here the picture is uploaded to Twitter.

```
259     @Override
260     public void onPhotoTaken(byte[] bytes, String filePath) {
261         Toast.makeText(this, "Photo: " + bytes.length + filePath, Toast.LENGTH_SHORT).show();
262
263         // This is the background task in charge of accessing Twitter and uploading the picture to
264                 // it.
265         (AsyncTask) (params) → {
269                 ConfigurationBuilder twitterConfigBuilder = new ConfigurationBuilder();
270                 twitterConfigBuilder.setDebugEnabled(true);
271                 twitterConfigBuilder.setOAuthConsumerKey("lxRCnjL6HaMUg7HjxAJC1k6IH");
272                 twitterConfigBuilder.setOAuthConsumerSecret(
273                         "6E3oLs4kln9p4oMkBRi2LceOkXuDYKASlXIm53UEq1wDNC4FxI");
274                 twitterConfigBuilder.setOAuthAccessToken(
275                         "854512192879820800-zcc88HtCEEcHyXO0JjgZJEFKmLP2HUi");
276                 twitterConfigBuilder.setOAuthAccessTokenSecret(
277                         "bUPpgmB6ipYkVb2kQ0LgAOeUPQtzZ78qBRB2iSrHQdJAe");
278
279                 Twitter twitter = new TwitterFactory(twitterConfigBuilder.build()).getInstance();
280                 File file = new File(params[0]);
281
282                 StatusUpdate status = new StatusUpdate("This is my view from " + message + "!");
283                 status.setMedia(file); // set the image to be uploaded here.
284                 try {
285                     twitter.updateStatus(status);
286                 } catch (TwitterException e) {
287                     e.printStackTrace();
288                 }
289                 return null;
290         }.execute(filePath);
292     }
```

*Figure 23: onPhotoTaken() code snippet*

*onRecieverEvent:*

This class receives the SMS sent by the EventBus triggered by IncomingSms. It is important to note that this is not a method, but a class.

```
297     public static class OnReceiverEvent {
298         private String smsMessage;
299
300         public OnReceiverEvent(String sms) { this.smsMessage = sms; }
303
304         public String getSmsMessage() { return smsMessage; }
307     }
```

*Figure 24: onRecieverEvent code snippet*

### *onSmsRecieved():*

This method is triggered when *TaskActivity* receives the SMS. Here the *NavigationService* is started.

```
313     @Subscribe
314     public void onSmsReceived(OnReceiverEvent event) {
315         message = event.getSmsMessage();
316
317         // This IF block ensures the SMS is not null. If not, the service starts.
318         if (message != null) {
319             callNavigationService(message);
320         }
321     }
```

*Figure 25: onSmsRecieved() code snippet*

### *callNavigationService():*

This is the method that starts the *NavigationService* by filing the *mIntent* when called.

```
326     private void callNavigationService(String message) {
327         mIntent.putExtra("message", message); // The SMS, which is the destination name, is passed
328                 // to NavigationService, because it needs it to file the API request.
329         this.startService(mIntent);  // This line is what actually starts the NavigationService.
330     }
```

*Figure 26: callNavigationService() code snippet*

## NavigationService:

The main purpose of this service class is to interact with Google Directions API and retrieve the necessary data for the robot's navigation. To achieve this, *NavigationService* find's the robot's current location, which is the starting point, and it uses the SMS destination message, which is the destination point, to construct a web request to Google Directions API. Once the request is filed, the API responds with an array list of locations. The *DataParser* helper class takes care of parsing the information retrieved from the API. *NavigationService* then rewrites the data into an array of location points. The location points represent the route the robot must travel through to reach its destination.

*NavigationService* uses the second point in the locations array as the point the robot needs to head to. It finds the robot's bearing to that point and passes that information to *TaskActivity*, which based on, will move the robot accordingly. *NavigationService* also finds the distance between the robot's current location and destination and returns it to the *TaskActivity*.

It is necessary to note that *NavigationService* creates its own thread and it keeps repeating infinitely every four seconds, until the robot reaches its destination.

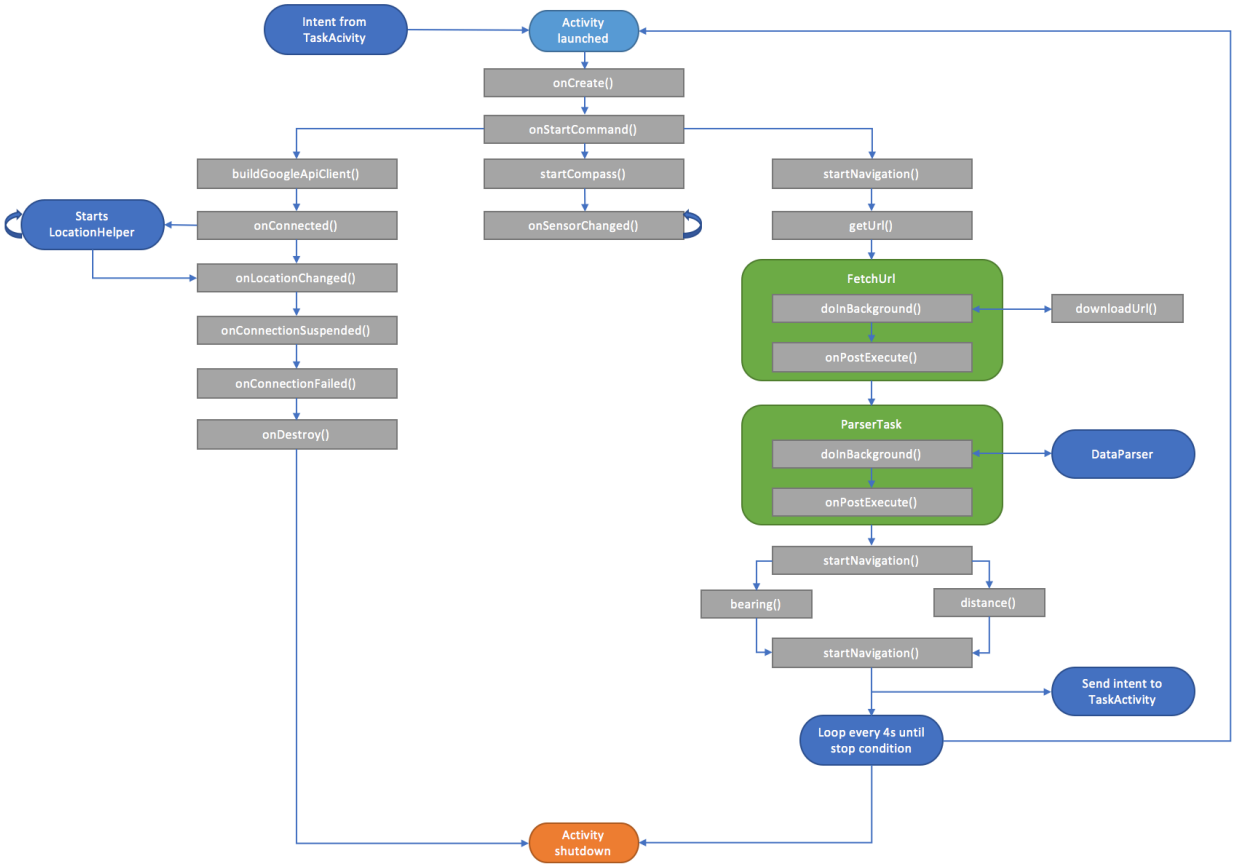


*Figure 27: NavigationService lifecycle*

## Extensions & Implementations:

*NavigationService* extends Service, which allows it to be a service class. A service class is a class that once started can run independently from the main activity performing tasks that take a long time, such as downloading content, playing music, etc. In this case, the app is using this service to download and build a navigation route. This class has many implementations: *GoogleApiClient*.*OnConnectionFailedListener* and *GoogleApiClient*.*ConnectionCallbacks* are both for interactions with Google API, *LocationListener* is for obtaining the phone's current location, and *SensorEventListener* allows the app to access the inner phone sensors.

```
59  public class NavigationService extends Service implements
60          GoogleApiClient.OnConnectionFailedListener,
61          GoogleApiClient.ConnectionCallbacks,
62          LocationListener,
63          SensorEventListener {
```

*Figure 28: NavigationService extensions and implementations*

## Public Variables:

These public variables are used throughout the code; hence they are defined here.

```
65      GoogleApiClient mGoogleApiClient;  // This is the API client that interacts with the Google API.
66      Location mCurrentLocation;  // This variable stores the robot's current location.
67      public LatLng currentLocationLatLng;  // This variable stores the current location as a LatLng.
68      public double currentLocationLat;  // This is the current location latitude.
69      public double currentLocationLng;  // This is the current location longitude.
70      public double nextDestinationLat;  // This is the final destination latitude.
71      public double nextDestinationLng;  // This is the final destination longitude.
72      public String routesUrl;  // This is the url that gets filed to the API.
73      private static final String TAG = "BroadcastService";  // The TAG used for the logcat for
74              // debugging.
75      // This is the string used for broadcasting the results to TaskActivity.
76      public static final String BROADCAST_ACTION = "com.websmithing.broadcasttest.displayevent";
77      Intent intent; // The intent filed back to TaskActivity.
78      public String mMessage;  // This stores the SMS message retrieved from TaskActivity.
79      public Location nextDestinationObj;  // This stores the next destination point in the location
80              // array.
81      public LatLng destination;  // This stores the next destination point in the location
82              // array as a LatLng.
83      private SensorManager sensorManager;  // This manager allows access to the phone's sensors.
84      private Sensor gsensor;  // This is the phone's gravity sensor.
85      private Sensor msensor;  // This is the phone's magnetic sensor.
86
87      // These variables are necessary for finding the phone's current bearing due compass north and
88              // the phone's bearing to a custom location point, which, in this case, is the next
89              // location point in the routes array.
90      private float[] mGravity = new float[3];  //
91      private float[] mGeomagnetic = new float[3];
92      private float azimuth = 0f;
93      private static final int earthRadius = 6371;
94
95      public double distance;  // This stores the distance between the robot and the final
96              // destination.
```

*Figure 29: NavigationService public variables*

## Methods & Classes:

### *onCreate():*

This method sets up the Google API client, the sensors, and the intent that gets sent back to *TaskActivity*.

```
@Override
public void onCreate() {

    // This IF block checks for permission and then builds the Google API client.
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_FINE_LOCATION)
                == PackageManager.PERMISSION_GRANTED) {
            buildGoogleApiClient();
        }
    } else {
        buildGoogleApiClient();
    }

    intent = new Intent(BROADCAST_ACTION);  // This creates the broadcast intent.

    // This block initiates the sensors and start them.
    sensorManager = (SensorManager) getBaseContext()
            .getSystemService(Context.SENSOR_SERVICE);
    gsensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    msensor = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
    startCompass();

    super.onCreate();
}
```

*Figure 30: onCreate() code snippet*

### *onStartCommand():*

This method is called when the *NavigationService* is called from *TaskActivity*. It finds the intent *TaskActivity* has filed, obtains the SMS message, and calls *startNavigation*().

```
public int onStartCommand(Intent intent, int flags, int startId) {
    mMessage = intent.getStringExtra("message");  // Obtains the SMS.

    // Creates a new thread upon which NavigationService runs.
    Thread t = new Thread(new Runnable() {
        @Override

        // Checks if the message is null, if not the service starts.
        public void run() {
            if (mMessage != null) {
                startNavigation(mMessage);
            }
        }
    });

    t.start();  // The thread starts.
    return START_STICKY;
}
```

*Figure 31: onStartCommand() code snippet*

### *startNavigation():*

This method is what gets everything going. It takes in the destination SMS message as an input parameter. It also has the current location which is stored as a global variable due to *LocationHelper*. *startNavigation* then constructs the url address using *getUrl*(). Afterwards, it uses *FetchUrl* which takes care of downloading the url and sending it to *DataParser* for parsing. What is returned is a *List<List<HashMap<String, String>>>* and stored as *routesList* by the use of *get*() from *ParserTask*. Then *routesUrl* is converted to an array using *positionArray*(). The second point in this array is extracted and that is the point that the robot will move to. Afterwards, the compass code block is used to find the bearing to this point and passes it as an intent to *TaskActivity*. Furthermore, since we know the final

destination, *calculateDistance*() is used to find the distance between the robot and the final destination and it is then passed to *TaskActivity* a broadcast intent.

```
165     public void startNavigation(String destinationSms) {
166         ArrayList<Location> pointsArray = null;
167
168         if (currentLocationLatLng != null) {
169
170             LatLng origin = currentLocationLatLng;  // Retrieves current location.
171             String dest = destinationSms;  // Retrieves final destination.
172
173             // Getting URL to the Google Directions API.
174             String url = getUrl(origin, dest);
175             Log.d("onMapClick", url.toString());
176
177             // Start downloading json data from Google Directions API.
178             try {
179                 routesUrl = new FetchUrl().execute(url).get();
180             } catch (InterruptedException e) {
181                 e.printStackTrace();
182             } catch (ExecutionException e) {
183                 e.printStackTrace();
184             }
185
186             // Invokes the thread for parsing the JSON data.
187             List<List<HashMap<String, String>>> routesList = null;
188             try {
189                 routesList = new ParserTask().execute(routesUrl).get();
190             } catch (InterruptedException e) {
191                 e.printStackTrace();
192             } catch (ExecutionException e) {
193                 e.printStackTrace();
194             }
195
196             pointsArray = positionArray(routesList);  // converts routesList to an array of
197                     // locations.
198
199             // This IF block retreives the second point in the array unless the array's size is 1.
200             if (pointsArray.size() == 1) {
201                 nextDestinationObj = pointsArray.get(0);
202             } else {
203                 nextDestinationObj = pointsArray.get(1);
204             }
205
206             nextDestinationLat = nextDestinationObj.getLatitude();
207             nextDestinationLng = nextDestinationObj.getLongitude();
208             destination = new LatLng(nextDestinationObj.getLatitude(),
209                     nextDestinationObj.getLongitude());
210             Location finalDestination = pointsArray.get(pointsArray.size() - 1);
211             double finalDestinationLat = finalDestination.getLatitude();
212             double finalDestinationLng = finalDestination.getLongitude();
213             distance = calculateDistance(currentLocationLat, currentLocationLng,
214                     finalDestinationLat, finalDestinationLng);
```

*Figure 32: startNavigation() code snippet*

```
215
216             intent.putExtra("distance", distance);
217             sendBroadcast(intent);  // Intent is broadcast.
218         }
219         // This IF block puts the thread to sleep for four seconds when it is done. However, if
220                 // the distnace is below 5m, it will stop the service.
221         if (distance > 5 && distance > 0) {
222             //job completed. Rest for 4 second before doing another one
223             try {
224                 Thread.sleep(4000);
225             } catch (InterruptedException e) {
226                 e.printStackTrace();
227             }
228             //do job again
229             startNavigation(destinationSms);
230         } else {
231             stopSelf();
232         }
233     }
```

*Figure 33: startNavigation() code snippet continuation*

### *calculateDistance():*

This method calculates the distance between two arbitrary locations.

```
238     public double calculateDistance(double lat1, double lon1, double lat2, double lon2) {
239         double dLat = Math.toRadians(lat2 - lat1);
240         double dLon = Math.toRadians(lon2 - lon1);
241         double a =
242                 (Math.sin(dLat / 2) * Math.sin(dLat / 2) + Math.cos(Math.toRadians(lat1))
243                         * Math.cos(Math.toRadians(lat2)) * Math.sin(dLon / 2) * Math.sin(dLon / 2));
244         double c = 2 * Math.asin(Math.sqrt(a));
245         double d = earthRadius * c * 1000;
246         return d;
247     }
```

*Figure 34: calculateDistance() code snippet*

### *positionArray():*

This method converts a *List<List<HashMap<String, String>>>* into an *ArrayList<Location>.* The main purpose of this method is to convert the result obtained from *DataParser* into a simple format of an array with *Location* variables.

```
252     public ArrayList<Location> positionArray(List<List<HashMap<String, String>>> routesList) {
253         ArrayList<Location> points = null;
254
255         // Traversing through all the routes
256         for (int i = 0; i < routesList.size(); i++) {
257             points = new ArrayList<>();
258
259             // Fetching i-th route
260             List<HashMap<String, String>> path = routesList.get(i);
261
262             // Fetching all the points in i-th route
263             for (int j = 0; j < path.size(); j++) {
264                 HashMap<String, String> point = path.get(j);
265
266                 double lat = Double.parseDouble(point.get("lat"));
267                 double lng = Double.parseDouble(point.get("lng"));
268                 Location position = new Location(LocationManager.GPS_PROVIDER);
269                 position.setLatitude(lat);
270                 position.setLongitude(lng);
271
272                 points.add(position);
273             }
274         }
275         return points;
276     }
```

*Figure 35:positionArray() code snippet*

### *getUrl():*

This method builds the url that gets filed to the Google API. It takes in two parameters, the first is the *LatLng* of the robot's current location, the second is the string destination, which is retrieved from the SMS passed to this service.

```
283     public String getUrl(LatLng origin, String dest) {
284         // Origin of route
285         String str_origin = "origin=" + origin.latitude + "," + origin.longitude;
286
287         // If the SMS destination is composed of multiple words like "Engineering Fountain", this
288                 // block of code breaks this phrase into separate words and add a "+" sign between
289                 // each word resulting in "Engineering+Fountain". This is done because the url filed
290                 // to Google has to be in a certain format.
291         String[] encodeDest = dest.split("\\P{L}+");
292         StringBuilder str_dest = new StringBuilder("destination=");
293         for (int encodeLength = 0; encodeLength < encodeDest.length; encodeLength++) {
294             str_dest.append(encodeDest[encodeLength] + "+");
295         }
296
297         // This stores the destination as a variable.
298         String destination = str_dest.toString();
299         Log.i("SmsReceiver", "Destination: " + destination + "; message: " + message);
300
301         // Sensor enabled.
302         String sensor = "sensor=false";
303
304         // Building the parameters to the web service.
305         String parameters = str_origin + "&" + str_dest + "&" + sensor;
306
307         // Output format.
308         String output = "json";
309
310         // Building the url to the web service.
311         String url = "https://maps.googleapis.com/maps/api/directions/" + output + "?" + parameters
312                 + "&mode=walking";  // The mode is walking.
313
314         return url;
315     }
```

*Figure 36: getUrl() code snippet*

*downloadUrl():*

This method downloads the data returned from the API as a string.

```
320     private String downloadUrl(String strUrl) throws IOException {
321         String data = "";
322         InputStream iStream = null;
323         HttpURLConnection urlConnection = null;
324         try {
325             URL url = new URL(strUrl);
326
327             // Creating an http connection to communicate with url.
328             urlConnection = (HttpURLConnection) url.openConnection();
329
330             // Connecting to url.
331             urlConnection.connect();
332
333             // Reading data from url.
334             iStream = urlConnection.getInputStream();
335
336             BufferedReader br = new BufferedReader(new InputStreamReader(iStream));
337
338             StringBuffer sb = new StringBuffer();
339
340             String line = "";
341             while ((line = br.readLine()) != null) {
342                 sb.append(line);
343             }
344
345             data = sb.toString();
346             Log.d("downloadUrl", data.toString());
347             br.close();
348
349         } catch (Exception e) {
350             Log.d("Exception", e.toString());
351         } finally {
352             iStream.close();
353             urlConnection.disconnect();
354         }
355         return data;
356     }
```

*Figure 37: downloadUrl() code snippet*

*onBind():*

This method is necessary for any service class. It is not being used however.

```
361     @Nullable
362     @Override
363     public IBinder onBind(Intent intent) { return null; }
```

*Figure 38: onBind() code snippet*

*FetchUrl:*

This class Fetches data from url passed. It runs in the background as an *AsyncTask*. After the data is retrieved, this class starts another background class called ParserTask, which parses the data.

```
370     public class FetchUrl extends AsyncTask<String, Void, String> {
371
372         @Override
373         protected String doInBackground(String... url) {
374
375             // For storing data from web service.
376             String data = "";
377
378             try {
379                 // Fetching the data from web service.
380                 data = downloadUrl(url[0]);
381                 Log.d("Background Task data", data.toString());
382             } catch (Exception e) {
383                 Log.d("Background Task", e.toString());
384             }
385             return data;
386         }
387
388         @Override
389         protected void onPostExecute(String result) {
390             super.onPostExecute(result);
391
392             ParserTask parserTask = new ParserTask();
393
394             // Invokes the thread for parsing the JSON data.
395             parserTask.execute(result);
396
397         }
398     }
```

*Figure 39: FetchUrl code snippet*

### *ParserTask:*

A class to parse the Google Directions in JSON format. This occurs in the background. Here, *DataParser* is called.

```
406     private class ParserTask extends AsyncTask<String, Integer,
407             List<List<HashMap<String, String>>>> {
408
409         // Parsing the data in non-ui thread.
410         @Override
411         protected List<List<HashMap<String, String>>> doInBackground(String... jsonData) {
412
413             JSONObject jObject;
414             List<List<HashMap<String, String>>> routes = null;
415
416             try {
417                 jObject = new JSONObject(jsonData[0]);
418                 Log.d("ParserTask", jsonData[0].toString());
419                 DataParser parser = new DataParser();
420                 Log.d("ParserTask", parser.toString());
421
422                 // Starts parsing data
423                 routes = parser.parse(jObject);
424                 Log.d("ParserTask", "Executing routes");
425                 Log.d("ParserTask", routes.toString());
426
427             } catch (Exception e) {
428                 Log.d("ParserTask", e.toString());
429                 e.printStackTrace();
430             }
431             return routes;
432         }
```

*Figure 40: ParserTask code snippet*

```
433
434         // Executes in UI thread, after the parsing process.
435         @Override
436         protected void onPostExecute(List<List<HashMap<String, String>>> result) {
437             ArrayList<LatLng> points;
438
439             // Traversing through all the routes.
440             for (int i = 0; i < result.size(); i++) {
441                 points = new ArrayList<>();
442
443                 // Fetching i-th route.
444                 List<HashMap<String, String>> path = result.get(i);
445
446                 // Fetching all the points in i-th route.
447                 for (int j = 0; j < path.size(); j++) {
448                     HashMap<String, String> point = path.get(j);
449
450                     double lat = Double.parseDouble(point.get("lat"));
451                     double lng = Double.parseDouble(point.get("lng"));
452                     LatLng position = new LatLng(lat, lng);
453
454                     points.add(position);
455                 }
456
457                 Log.d("onPostExecute", "onPostExecute lineoptions decoded");
458
459             }
460         }
461     }
```

*Figure 41: ParserTask code snippet continuation*

### *buildGoogleApiClient():*

This method builds the Google Api Client and establishes the listener.

```
466     protected synchronized void buildGoogleApiClient() {
467         mGoogleApiClient = new GoogleApiClient.Builder(this)
468                 .addConnectionCallbacks(this)
469                 .addOnConnectionFailedListener(this)
470                 .addApi(LocationServices.API)
471                 .build();
472         mGoogleApiClient.connect();
473     }
```

*Figure 42: buildGoogleApiClient() code snippet*

### *onConnected():*

This method is called when a connection to the Google API is established. The *LocationHelper* is called to start the current location requests.

```
479     @Override
480     public void onConnected(Bundle bundle) {
481         LocationHelper.initLocation(this.getBaseContext(), (LocationListener) this,
482                 mGoogleApiClient);
483     }
```

*Figure 43: onConnected() code snippet*

### *onLocationChanged():*

This method is called when the robot's location changes. This updates all the current location variables.

```
489     public void onLocationChanged(Location location) {
490         mCurrentLocation = location;
491         currentLocationLat = location.getLatitude();
492         currentLocationLng = location.getLongitude();
493         currentLocationLatLng = new LatLng(currentLocationLat, currentLocationLng);
494     }
```

*Figure 44: onLocationChanged() code snippet*

### *onConnectionSuspended():*

This method is called when a connection to the Google API is suspended.

```
499     @Override
500     public void onConnectionSuspended(int i) {
501     }
```

*Figure 45: onConnectionSuspended() code snippet*

### *onConnectionFailed():*

This method is called when a connection to the Google API has failed.

```
506     @Override
507     public void onConnectionFailed(ConnectionResult connectionResult) {
508     }
```

*Figure 46: onConnectionFailed() code snippet*

### *onConnectionFailed():*

This method is called when the service stops. It disconnects the API client.

```
514     public void onDestroy() {
515         mGoogleApiClient.disconnect();
516         super.onDestroy();
517     }
```

*Figure 47: onDestroy() code snippet*

### *startCompass():*

This method starts the compass sensors.

```
526     public void startCompass() {
527         sensorManager.registerListener(this, gsensor,
528                 SensorManager.SENSOR_DELAY_GAME);
529         sensorManager.registerListener(this, msensor,
530                 SensorManager.SENSOR_DELAY_GAME);
531     }
```

*Figure 48: startCompass() code snippet*

*onSensorChanged():*

This method is triggered whenever the compass readings are changed. This method returns the phone's bearing to the next location point in the routes array.

```
@Override
public void onSensorChanged(SensorEvent event) {
    final float alpha = 0.97f;

    synchronized (this) {
        if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {

            mGravity[0] = alpha * mGravity[0] + (1 - alpha)
                    * event.values[0];
            mGravity[1] = alpha * mGravity[1] + (1 - alpha)
                    * event.values[1];
            mGravity[2] = alpha * mGravity[2] + (1 - alpha)
                    * event.values[2];
        }

        if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {

            mGeomagnetic[0] = alpha * mGeomagnetic[0] + (1 - alpha)
                    * event.values[0];
            mGeomagnetic[1] = alpha * mGeomagnetic[1] + (1 - alpha)
                    * event.values[1];
            mGeomagnetic[2] = alpha * mGeomagnetic[2] + (1 - alpha)
                    * event.values[2];
        }

        float R[] = new float[9];
        float I[] = new float[9];
        boolean success = SensorManager.getRotationMatrix(R, I, mGravity,
                mGeomagnetic);
        if (success) {
            float orientation[] = new float[3];
            SensorManager.getOrientation(R, orientation);
            azimuth = (float) Math.toDegrees(orientation[0]); // orientation
            azimuth = (azimuth + 360) % 360;
            if (destination != null) {
                Toast.makeText(this, "I've been called", Toast.LENGTH_SHORT).show();
                azimuth -= bearing(currentLocationLat, currentLocationLng, nextDestinationLat,
                        nextDestinationLng);
                if (azimuth < 0) {
                    azimuth = azimuth + 360;
                }
                intent.putExtra("direction", azimuth);
                Log.d(TAG, "azimuth (deg): " + azimuth);
            }
        }
    }
}
```

*Figure 49: onSensorChanged() code snippet*

### *bearing():*

This method is what calculates the phone's bearing to the next location point. This method is called inside *onSensorChanged*().

```
589     protected double bearing(double startLat, double startLng, double endLat, double endLng) {
590         double longitude1 = startLng;
591         double longitude2 = endLng;
592         double latitude1 = Math.toRadians(startLat);
593         double latitude2 = Math.toRadians(endLat);
594         double longDiff = Math.toRadians(longitude2 - longitude1);
595         double y = Math.sin(longDiff) * Math.cos(latitude2);
596         double x = Math.cos(latitude1) * Math.sin(latitude2) - Math.sin(latitude1) *
597                 Math.cos(latitude2) * Math.cos(longDiff);
598 
599         return (Math.toDegrees(Math.atan2(y, x)) + 360) % 360;
600     }
```

*Figure 50: bearing() code snippet*

### *onAccuracyChanged():*

This method is necessary for the *SensorEventListener* implementation. It is not used however.

```
605     @Override
606     public void onAccuracyChanged(Sensor sensor, int accuracy) {
607 
608     }
```

*Figure 51: onAccuracyChanged() code snippet*

## LocationHelper:

This helper class finds the robot's current location every 750ms. The location found is returned to the *NavigationService* for further processing.

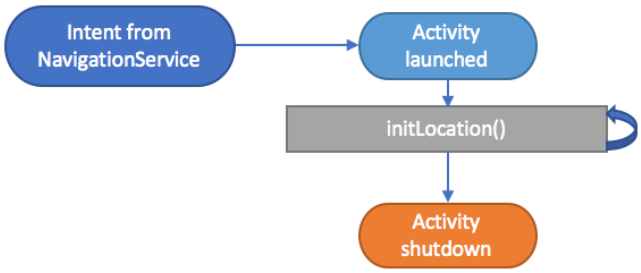


*Figure 52: LocationHelper flowchart*

## Extensions & Implementations:

This class has neither.

```
20    public class LocationHelper {
```

*Figure 53: LocationHelper definition*

## Public Variables:

This class is very short and doesn't have any public variables.

## Methods:

This method files for a location request. It specifies the priority as high accuracy and uses ACCESS_FINE_LOCATION for best results. It also sets the time interval after which another location request will be filed.

```
26      static void initLocation(Context c, LocationListener listener, GoogleApiClient apiClient) {
27          LocationRequest mLocationRequest = LocationRequest.create();
28          mLocationRequest.setInterval(750);
29          mLocationRequest.setFastestInterval(750);
30          mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
31          if (ContextCompat.checkSelfPermission(c,
32                  Manifest.permission.ACCESS_FINE_LOCATION)
33                  == PackageManager.PERMISSION_GRANTED) {
34              LocationServices.FusedLocationApi.requestLocationUpdates(apiClient, mLocationRequest,
35                      listener);
36          }
37      }
```

*Figure 54: initocation() code snippet*

## IOIOClass:

The class is responsible for setting up the communication between the phone and the IOIO board. It establishes a loop that keeps sending out information to the board. In this class, all the pins required to control the motors are defined. There is a public setter method in this class that allows an external class to change the motors directions. This method is primarily used by *TaskActivity* to move the robot.

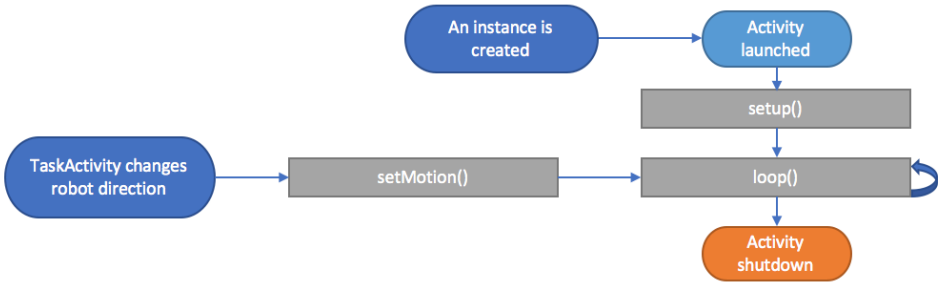


*Figure 55: IOIOClass flowchart*

## Extensions & Implementations:

This class extends *BaseIOIOLooper*, which allows for the basic IOIO functionality. Furthermore, this class implements *IOIOLooperProvider*, which allows an external class to create and control an instance of *IOIOClass*.

```
19    public class IOIOClass extends BaseIOIOLooper implements IOIOLooperProvider {
```

*Figure 56: IOIOClass definition*

## Public Variables:

Below is a list of the public variables used in *IOIOClass*. It is best they are defined in this section as they will be mentioned repeatedly across the code.

```
21      // Motor DC : Right Forward.
22      private DigitalOutput INA1F; // L298n In 1
23      private DigitalOutput INA2F; // L298n In 2
24      private PwmOutput ENAF; // L298n Enable 1
25
26      // Motor DC : Right Rear.
27      private DigitalOutput INA1R; // L298n In 1
28      private DigitalOutput INA2R; // L298n In 2
29      private PwmOutput ENAR; // L298n Enable 1
30
31      // Motor DC : Left Forward.
32      private DigitalOutput INA3F; // L298n In 3
33      private DigitalOutput INA4F; // L298n In 4
34      private PwmOutput ENBF; // L298n Enable 2
35
36      // Motor DC : Left Rear.
37      private DigitalOutput INA3R; // L298n In 3
38      private DigitalOutput INA4R; // L298n In 4
39      private PwmOutput ENBR; // L298n Enable 2
40
41      // All motors are initialized to stop.
42      private boolean FMotorLeft = false;
43      private boolean FMotorRight = false;
44      private float FRightSpeed = 0;
45      private float FLeftSpeed = 0;
46      private boolean RMotorLeft = false;
47      private boolean RMotorRight = false;
48      private float RRightSpeed = 0;
49      private float RLeftSpeed = 0;
50
51      // These strings are used as commands.
52      private static final String FORWARDS = "forwards";
53      private static final String RIGHT = "right";
54      private static final String LEFT = "left";
55      private static final String STOP = "stop";
56
57      private IOIOAndroidApplicationHelper helper;  // This helper is necessary to start the IOIOClass
58              // loop from another class, TaskActivity.
```

*Figure 57: IOIOClass public definitions*

## Constructor:

This class has a constructor to allow for an instance of it to be created using the helper.

```
63      public IOIOClass(TaskActivity mTheGui) {
64          helper = new IOIOAndroidApplicationHelper(mTheGui, this);
65      }
```

*Figure 58: IOIOClass constructor*

## Methods:

### *setup():*

This method is used to set up all the IOIO pins.

```
70      @Override
71      protected void setup() throws ConnectionLostException, InterruptedException {
72          try {
73              // Motor DC : Right Forward.
74              INA1F = ioio_.openDigitalOutput(1);
75              INA2F = ioio_.openDigitalOutput(2);
76              ENAF = ioio_.openPwmOutput(3, 100);
77
78              // Motor DC : Right Rear.
79              INA1R = ioio_.openDigitalOutput(11);
80              INA2R = ioio_.openDigitalOutput(12);
81              ENAR = ioio_.openPwmOutput(7, 100);
82
83              // Motor DC : Left Forward.
84              INA3F = ioio_.openDigitalOutput(4);
85              INA4F = ioio_.openDigitalOutput(5);
86              ENBF = ioio_.openPwmOutput(6, 100);
87
88              // Motor DC : Left Rear.
89              INA3R = ioio_.openDigitalOutput(13);
90              INA4R = ioio_.openDigitalOutput(14);
91              ENBR = ioio_.openPwmOutput(10, 100);
92          } catch (ConnectionLostException e) {
93              throw e;
94          }
95      }
```

*Figure 59: setup() code snippet*

*loop():*

This method is the loop that keeps sending out commands to the IOIO pins.

```
100     @Override
101     public void loop() throws ConnectionLostException, InterruptedException {
102         ioio_.beginBatch();  // Used when multiple pins are commanded at once.
103         try {
104             // Right motor forward.
105             ENBF.setDutyCycle(FRightSpeed);
106             INA4F.write(FMotorRight);
107             INA3F.write(!FMotorRight);
108
109             // Left motor rear.
110             ENBR.setDutyCycle(RLeftSpeed);
111             INA4R.write(RMotorLeft);
112             INA3R.write(!RMotorLeft);
113
114             // Left motor forward.
115             ENAF.setDutyCycle(FLeftSpeed);
116             INA2F.write(FMotorLeft);
117             INA1F.write(!FMotorLeft);
118
119             // Right motor rear.
120             ENAR.setDutyCycle(RRightSpeed);
121             INA2R.write(RMotorRight);
122             INA1R.write(!RMotorRight);
123
124             Thread.sleep(10);
125         } catch (InterruptedException e) {
126             ioio_.disconnect();
127         } catch (ConnectionLostException e) {
128             throw e;
129         } finally {
130             ioio_.endBatch();
131         }
132
133     }
```

*Figure 60: loop() code snippet*

### *setMotion():*

This method is used to change the motors' directions and speeds based on a string command. This method is publicly accessible by other classes, such as *TaskActivity*.

```
139     public void setMotion(String direction) {
140         if (direction.equals(FORWARDS)) {
141             FLeftSpeed = (float) 0.6;
142             FRightSpeed = (float) 0.6;
143             FMotorLeft = true;
144             FMotorRight = false;
145
146             RLeftSpeed = (float) 0.6;
147             RRightSpeed = (float) 0.6;
148             RMotorLeft = true;
149             RMotorRight = false;
150         } else if (direction.equals(RIGHT)) {
151             FLeftSpeed = (float) 0.4;
152             FRightSpeed = (float) 0.4;
153             FMotorLeft = true;
154             FMotorRight = true;
155
156             RLeftSpeed = (float) 0.4;
157             RRightSpeed = (float) 0.4;
158             RMotorLeft = true;
159             RMotorRight = true;
160         } else if (direction.equals(LEFT)) {
161             FLeftSpeed = (float) 0.4;
162             FRightSpeed = (float) 0.4;
163             FMotorLeft = false;
164             FMotorRight = false;
165
166             RLeftSpeed = (float) 0.4;
167             RRightSpeed = (float) 0.4;
168             RMotorRight = false;
169             RMotorLeft = false;
170         } else if (direction.equals(STOP)) {
171             FLeftSpeed = 0;
172             FRightSpeed = 0;
173
174             RLeftSpeed = 0;
175             RRightSpeed = 0;
176         }
177     }
```

*Figure 61: setMotion() code snippet*

### *createIOIOLooper():*

This method is necessary for the *IOIOLooperProvider* implementation.

```
182     @Override
183     public IOIOLooper createIOIOLooper(String connectionType, Object extra) { return this; }
```

*Figure 62: createIOIOLooper() code snippet*

### *getIOIOAndroidApplicationHelper():*

This method is used to allow an external class to access the helper to be able to create an instance of the *IOIOClass*.

```
191     public IOIOAndroidApplicationHelper getIOIOAndroidApplicationHelper() { return helper; }
194 }
```

*Figure 63: getIOIOAndroidApplicationHelper() code snippet*

## IncomingSms:

This class is on the lookout for any incoming SMS messages when the app is on. Once it receives an SMS, it sends the message to *TaskActivity* through an *EventBus*.

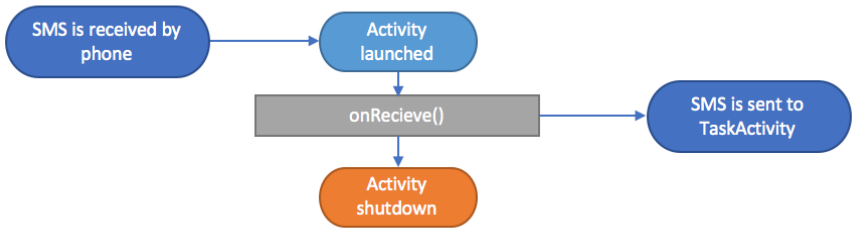


*Figure 64: IncomingSms flowchart*

## Extensions & Implementations:

This class only extends BroadcastReciever, which allow it to receive SMS messages sent to the phone.

```
21  public class IncomingSms extends BroadcastReceiver {
```

*Figure 65: IncomingSms definition*

## Public Variables:

This class contains only one public variable.

```
23      public String message;  // The SMS variable.
```

*Figure 66: IncomingSms public variables*

## Constructor:

This class has a very basic constructor to allow an instance of it to be created.

```
28      public IncomingSms () {
29      }
```

*Figure 67: IncomingClass constructor*

## Methods:

### *onRecieve():*

This method receives the SMS and sends it to *TaskActivity* as an *EventBus*.

```
34  public void onReceive(Context context, Intent intent) {
35      // Retrieves a map of extended data from the intent.
36      final Bundle bundle = intent.getExtras();
37      try {
38          if (bundle != null) {
39              final Object[] pdusObj = (Object[]) bundle.get("pdus");
40
41              for (int j = 0; j < pdusObj.length; j++) {
42
43                  SmsMessage currentMessage = SmsMessage.createFromPdu((byte[]) pdusObj[j]);
44                  String phoneNumber = currentMessage.getDisplayOriginatingAddress();
45
46                  String senderNum = phoneNumber;
47                  message = currentMessage.getDisplayMessageBody();
48
49                  Log.i("SmsReceiver", "senderNum: " + senderNum + "; message: " + message);
50
51                  // This block takes care of sending the SMS to TaskActivity.
52                  EventBus.getDefault().post(new TaskActivity.OnReceiverEvent(message));
53                  Toast toast = Toast.makeText(context, message, Toast.LENGTH_SHORT);
54                  toast.show();
55              }
56          }
57      } catch (Exception e) {
58          Log.e("SmsReceiver", "Exception smsReceiver" + e);
59      }
60  }
```

*Figure 68: onRecieve() code snippet*

## DataParser:

This helper class is used to parse the data retrieved from the Google API and returns it to *NavigationService* as an array list.

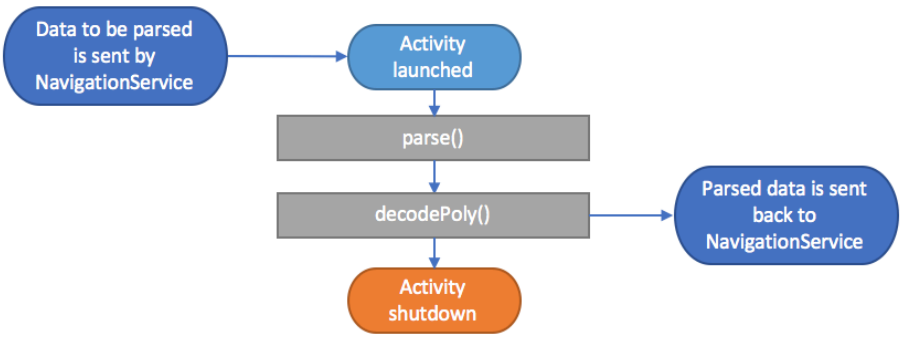


*Figure 69: DataParser flowchart*

## Extensions & Implementations:

This class has no extensions or implementations.

## Public Variables:

This class has no public variables.

## Methods:

### *parse():*

Receives a *JSONObject* and returns a list of lists containing latitude and longitude.

```
22      public List<List<HashMap<String,String>>> parse(JSONObject jObject){
23
24          List<List<HashMap<String, String>>> routes = new ArrayList<>() ;
25          JSONArray jRoutes;
26          JSONArray jLegs;
27          JSONArray jSteps;
28
29          try {
30
31              jRoutes = jObject.getJSONArray("routes");
32
33              /** Traversing all routes */
34              for(int i=0;i<jRoutes.length();i++){
35                  jLegs = ( (JSONObject)jRoutes.get(i)).getJSONArray("legs");
36                  List path = new ArrayList<>();
37
38                  /** Traversing all legs */
39                  for(int j=0;j<jLegs.length();j++){
40                      jSteps = ( (JSONObject)jLegs.get(j)).getJSONArray("steps");
41
42                      /** Traversing all steps */
43                      for(int k=0;k<jSteps.length();k++){
44                          String polyline = "";
45                          polyline = (String)((JSONObject)((JSONObject)jSteps.get(k)).get("polyline"))
46                                  .get("points");
47                          List<LatLng> list = decodePoly(polyline);
48
49                          /** Traversing all points */
50                          for(int l=0;l<list.size();l++){
51                              HashMap<String, String> hm = new HashMap<>();
52                              hm.put("lat", Double.toString((list.get(l)).latitude) );
53                              hm.put("lng", Double.toString((list.get(l)).longitude) );
54                              path.add(hm);
55                          }
56                      }
57                      routes.add(path);
58                  }
59              }
60          } catch (JSONException e) {
61              e.printStackTrace();
62          }catch (Exception e){
63          }
64          return routes;
65      }
```

*Figure 70: parse() code snippet*

*decodePoly():*

Method to decode polyline points.

```
73      private List<LatLng> decodePoly(String encoded) {
74
75          List<LatLng> poly = new ArrayList<>();
76          int index = 0, len = encoded.length();
77          int lat = 0, lng = 0;
78
79          while (index < len) {
80              int b, shift = 0, result = 0;
81              do {
82                  b = encoded.charAt(index++) - 63;
83                  result |= (b & 0x1f) << shift;
84                  shift += 5;
85              } while (b >= 0x20);
86              int dlat = ((result & 1) != 0 ? ~(result >> 1) : (result >> 1));
87              lat += dlat;
88
89              shift = 0;
90              result = 0;
91              do {
92                  b = encoded.charAt(index++) - 63;
93                  result |= (b & 0x1f) << shift;
94                  shift += 5;
95              } while (b >= 0x20);
96              int dlng = ((result & 1) != 0 ? ~(result >> 1) : (result >> 1));
97              lng += dlng;
98
99              LatLng p = new LatLng((((double) lat / 1E5)),
100                     (((double) lng / 1E5)));
101             poly.add(p);
102         }
103
104         return poly;
105     }
106 }
```

*Figure 71: decodePoly() code snippet*

## Prior to Using the App:

All the permissions the app requires must be allowed prior to using the app; otherwise, it will crash.

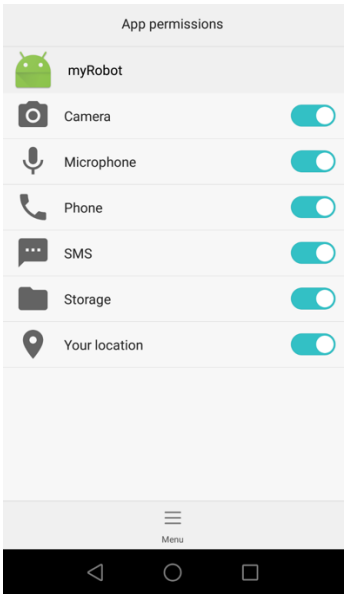


*Figure 72: Permissions screenshot*

Moreover, USB debugging must be turned off prior to connecting the phone to IOIO as shown in Figure 19. When the IOIO is connected to the phone a prompt as in Figure 20 will appear, the user should exit that prompt. Then as the user opens the app another prompt will appear closing the app, after the user hits OK, they should open the app again. This is an issue that needs to be solved in the future.

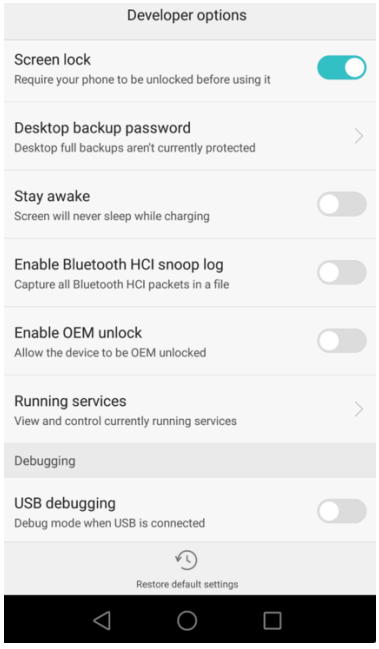


*Figure 73: USB Debugging off*

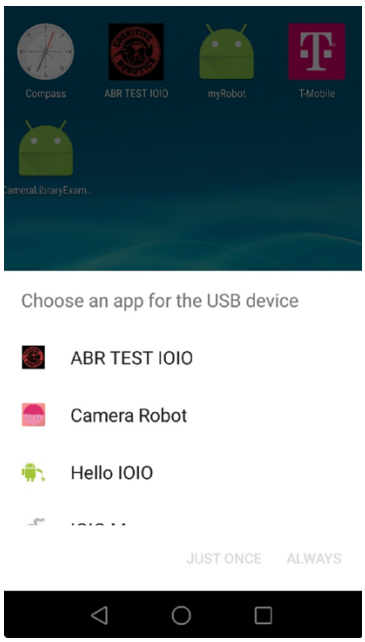


*Figure 74: First prompt*

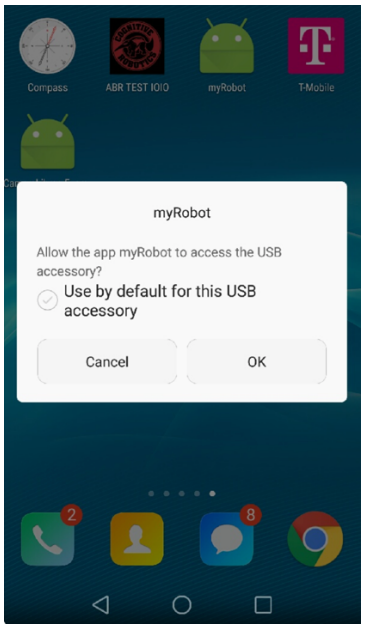


*Figure 75: Second prompt*

## Bill of Materials:

The BOM below is not fully representative of the project since some of the materials used to build the robot was already there and didn't need to be bought. A simple example of that is the two back wheels; to obtain those a secondary kit must be purchased, which is what included in the BOM. However, in truth the wheels used were from a different project lying around. It is also necessary to note that any phone can be used for the purposes of this project if it has Android 4 or newer.

| Item | Quantity | Use | Price | Total | Link |
|---|---|---|---|---|---|
| Sparkfun (PID 13964) IOIO-OTG - V2 (with Headers) | 1 | Interface board (between android and robot). | $34.95 | $34.95 | https://www.amazon.com/Sparkfun-PID-13964-IOIO-OTG-Headers/dp/B01M7O5Z9N/ref=sr_1_4?s=electronics&ie=UTF8&qid=1485529441&sr=1-4&keywords=ioio+board |
| Starter Robot Kit (Bluetooth Version) | 2 | Robot. | $149.99 | $299.98 | http://www.makeblock.com/starter-robot-kit |
| Honor 5X - 16 GB - Gray - Unlocked | 1 | 2 phones, one to be mounted on the robot and the other for communicating and controlling the phone on the robot. | $165.00 | $165.00 | http://www.bestbuy.com/site/honor-refurbished-5x-4g-lte-with-16gb-memory-cell-phone-unlocked-gray/5489200.p?skuId=5489200&ref=212&loc=1&ksid=320ee66f-2b4b-4f64-84d0-e67b6f3afe08&ksprof_id=10&ksaffcode=pg129284&ksdevice=c&lsft=ref:212,loc:2 |
| T-Mobile Prepaid Complete SIM Starter Kit - No Contract Network Connection (Universal: Standard, Mirco, Nano SIM) | 1 | Prepaid SIM cards for 4G communication between both phones. | $9.39 | $9.39 | https://www.amazon.com/T-Mobile-Prepaid-Complete-SIM-Kit/dp/B00LPPHHFK/ref=sr_1_2?s=wireless&ie=UTF8&qid=1485530797&sr=1-2 |
| DC Stepper Motor Drive Controller Board Module L298N for arduino Dual H Bridge | 2 | Control the motors | $7.99 | $15.98 | https://www.amazon.com/gp/product/B00M4348Z4/ref=oh_aui_detailpage_o06_s00?ie=UTF8&psc=1 |
| | | | Total | $509.32 | |

*Figure 76: Bill of Materials*

The links in the image are not clickable; therefore, they can be found below:

- First link: https://www.amazon.com/Sparkfun-PID-13964-IOIO-OTG-Headers/dp/B01M7O5Z9N/ref=sr_1_4?s=electronics&ie=UTF8&qid=1485529441&sr=1-4&keywords=ioio+board
- Second link: http://www.makeblock.com/starter-robot-kit
- Third link: http://www.bestbuy.com/site/honor-refurbished-5x-4g-lte-with-16gb-memory-cell-phone-unlocked-gray/5489200.p?skuId=5489200&ref=212&loc=1&ksid=320ee66f-2b4b-4f64-84d0-e67b6f3afe08&ksprof_id=10&ksaffcode=pg129284&ksdevice=c&lsft=ref:212,loc:2
- Fourth link: https://www.amazon.com/T-Mobile-Prepaid-Complete-SIM-Kit/dp/B00LPPHHFK/ref=sr_1_2?s=wireless&ie=UTF8&qid=1485530797&sr=1-2
- Fifth link: https://www.amazon.com/gp/product/B00M4348Z4/ref=oh_aui_detailpage_o06_s00?ie=UTF8&psc=1

## Future and Improvements:

There are three aspects to this project; the first is the autonomous navigation of the robot, the second is the code structure itself, and the third is the interaction between the robot and social media. The aim is to grow in all aspects with this project. Currently the robot can move from its current location to a destination; however, this is still very primitive. The robot moves very slowly because it is programmed to stop after three seconds when going straight, and after one second when rotating. The reason for doing so is to prevent the robot from going off course quickly, and to allow it some time to reassess its location. Furthermore, the robot has no obstacle avoidance whatsoever, it merely follows a

path. Therefore, regarding the autonomous navigation aspect of the robot, in the future the robot will be able to travel in a seamless way without stopping every few seconds. Furthermore, the robot will be able to avoid any obstacles in its path.

The code structure for the app can be broken down into more helper classes. Currently, the TaskActivity class preforms more than one job. It is the main hub for all the interactions between the different classes; however, it is also where the robot is being navigated as well as the location for the interaction with social media. This class must be broken down into more subclasses to allow for a better code structure and a more flexible design. Moreover, one of the future aims of this project is to have the app independent from the robot upon which it runs. The app will be downloadable by any android phone and the phone will be attached to a robot, which can be an RC car, a drone, etc. If the robot has an IOIO board, the phone will be able to connect. Once the connection is established, the app will identify what type of robot it is tethered to, connect to its actuators and apply PID control, and then it will autonomously navigate to whichever location the user texts it to. A key point to achieving this goal is to insure the code structure is fully modular, meaning any segment of the code can be modified or replaced without affecting the interactions between each segment.

The robot's interaction with social media is currently very simple. The robot takes a picture and uploads it to Twitter with a caption including the final destination's name. The goal here is to have the robot behaving as a person. In the future, the robot will be able to interact with people on social media as opposed to just uploading a picture.

There is a small issue in the code that must be fixed; if the GPS connection fails, the app crashes. The way to fix that problem is through modifying *onConnectionSuspended*() and *onConnectionFailed*() in *NavigationService*.

## Conclusion:

The robot has performed all the tasks it set out to. This project proved that if a robot has a phone it can be treated as a person. All that needs to be done is to send it a destination and it will travel to it, take a picture, and upload it to social media. There are still many improvements to be done to this project. The navigation of the robot must be fully autonomous with obstacle avoidance in the future. Furthermore, the app's code structure needs to be made fully modular and flexible for future modifications. Moreover, this app will be adaptable to any robot is it tethered to in the future. Additionally, the robot will have more advanced interactions with people and social media.

## References:


Cognitive Anteater Robotics Laboratory (27 July 2015). Android Based Robotics: Powerful, Flexible and Inexpensive Robots for Hobbyists, Educators, Students and Researchers. Retrieved from http://www.socsci.uci.edu/~jkrichma/ABR/index.html#tutorials

Jason Bowling (07/03/2015).WIFI Operated Rover w/ Android and IOIO. Retrieved from https://hackaday.io/post/20395

Christian Bodenstein, Michael Tremer, Jonathan Overhoff, Rolf P. Würtz (2015). A Smartphone-controlled Autonomous Robot. Retrieved from https://www.ini.rub.de/upload/file/1470692860_4ccfceb655d65637fab8/android-robot.pdf

# Appendix:

## I. Connections Diagram:

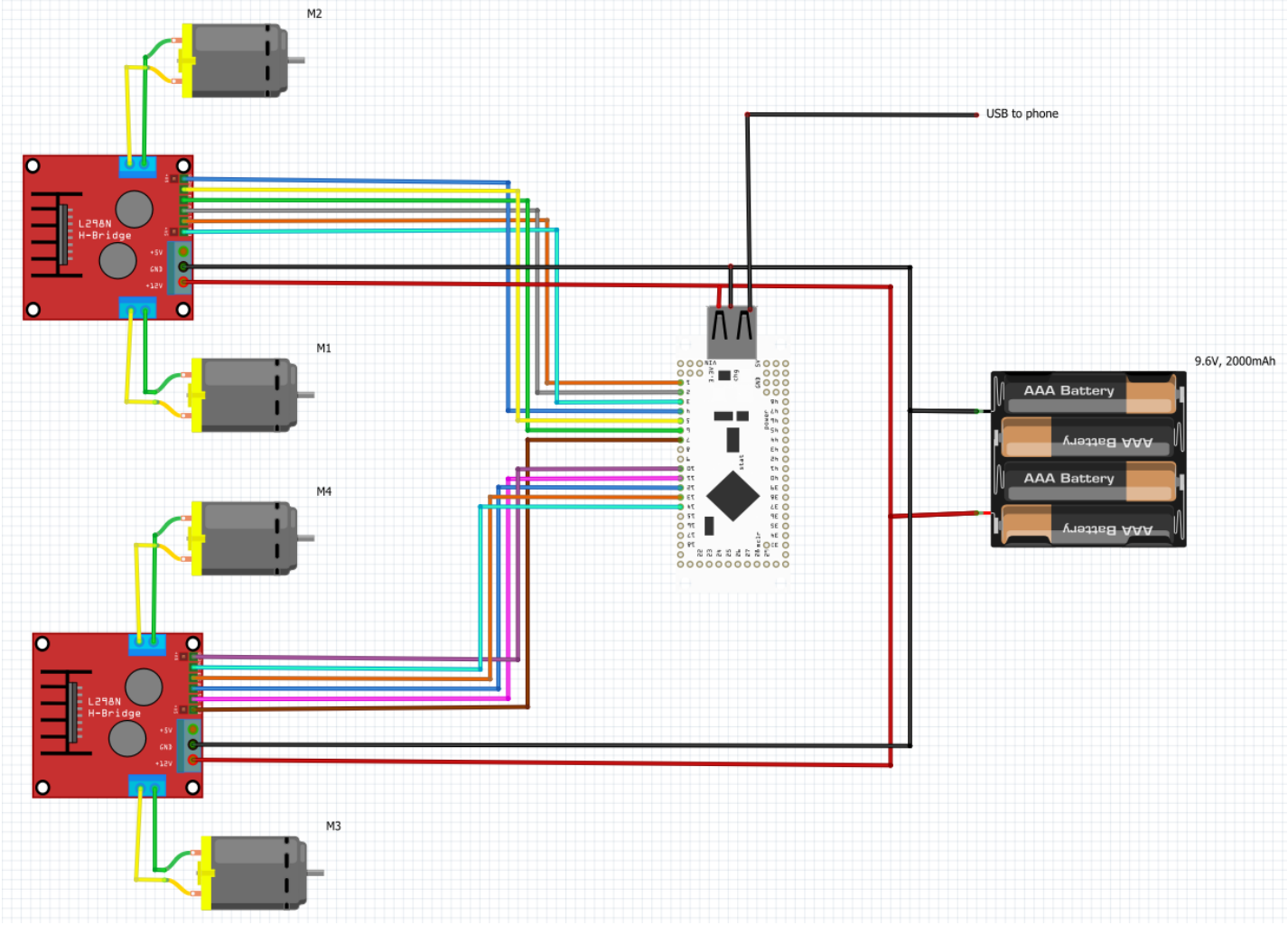


*Figure 77: Connections Diagram*