

**Infrastructure for BMW Test**

**Car Display System**

Part B – Spring 2019

**Project report**

Developed by

Daher Daher daher928@campus.technion.ac.il

Vivian Lawin vivlawin@campus.technion.ac.il

Supervisor

Boaz Mizrachi



Infrastructure for BMW Test Car Display System

Table of Contents

[1 Preface 2](#_Toc35163616)

[2 Introduction 3](#_Toc35163617)

[2.1 Is it necessary? 3](#_Toc35163618)

[2.2 Project Goals 4](#_Toc35163619)

[3 Technology 5](#_Toc35163620)

[3.1 Display screen 5](#_Toc35163621)

[3.2 Software and IDEs 6](#_Toc35163622)

[3.3 Graphics and UI components 7](#_Toc35163623)

[4 Part A 8](#_Toc35163624)

[4.1 Recap 8](#_Toc35163625)

[4.2 Data Transfer Solution 9](#_Toc35163626)

[4.3 High Level Design 10](#_Toc35163627)

[4.4 Features 11](#_Toc35163628)

[5 Part B 12](#_Toc35163629)

[5.1 Overview and Incentive 12](#_Toc35163630)

[5.2 Features 13](#_Toc35163631)

[5.3 High Level Design 14](#_Toc35163632)

[5.4 Development Process 14](#_Toc35163633)

[5.5 Implementation 14](#_Toc35163634)

[5.5.1 User Interface 14](#_Toc35163635)

[5.6 Back-end 24](#_Toc35163636)

[6.2.1 Transmitter 24](#_Toc35163637)

[6.2.2 Receiver 25](#_Toc35163638)

[7 Appendix 28](#_Toc35163639)

[7.1 Stream File 28](#_Toc35163640)

[7.2 Sensors Configurations File 30](#_Toc35163641)

[7.3 GraphView Library 31](#_Toc35163642)

# 1 Preface

In order to enhance and improve the driving experience in BMW cars in general (and BMW’s Autonomous cars in particular), drivers and car testers must understand and feel not only the condition of the environment and the traffic, but also the condition of the road, e.g. fraction of the road, humidity, temperature and angle.

Feeling and experiencing the interaction between the car and road is important for optimizing car’s safety and efficiency both for human drivers and autonomous driving.

Our project aims to visualize the conditions of the road and the real-time data collected by the car’s sensors and provide a comfortable and elegant user interface to graphically display the results on the vehicle’s dashboard screen display.

# 2 Introduction

## 2.1 Is it necessary?

Car testing is very essential, and the more the testing procedure is efficient the more the results are reliable, and time is saved. both advantages are highly desired in the era of new technology, and especially in the emerging field of Artificial Intelligence and autonomous cars, enabling a transformation and technological rise in the mobility industry.

When mentioning AI and autonomous cars, the term “Safety” pops up as a key goal. Car testing – and especially when intended to be driven autonomously – is a necessary process for assuring and guaranteeing the safety of the passengers and the vehicle’s surrounding environment and pedestrians. 

## 2.2 Project Goals

The main goal of this project is to provide car drivers and testers with a friendly and easy-to-use application with the ability to graphically display the real-time data received by the car’s sensor.

Each of the following features of the application will add to the users experience much convenience, starting with choosing the configurations of each graph, to getting previous results, which are very useful for comparing results and investigating the changes in the results according to changes in the surrounding circumstances.

**Full features list:**

* Support multiple graphs (Up to 3 graphs at a time)
* Support logging and data recording
* Cloud storage of test results
* User authentication and user space
* Choose different configurations (Resolution, Max value, Color…)
* Time axis scaling
* Different appearance theme
* Caching previous configurations set for reuse



# 3 Technology

The following chapter describes the different technologies used for implementing the project, such like: display screen, data transfer solutions, development environments and developing tools.

## 3.1 Display screen

The device that is used for running the application and displaying the Realtime results and graphs is GINI Tab V7. Following are the specifications and features of the device:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Native Platform | Android Nougat 7 | | Release Year | 2017 | | Device RAM | 2 GB | | Battery Capacity | 2800 mAh | | Max Internal Storage | 16 GB | | Expansion Slot Type | MicroSD | | Expansion Slot Max Size | 32 GB | | Supported I/O | 3.5mm Jack, Micro-USB | | Supported Charger Types | Wire | | Supported Bearers | Bluetooth, WiFi | | Has NFC | true | | Screen Type | IPS | | Screen Inches Diagonal | 7.0 inch | | Screen Pixels Width | 1280 pixels | | Screen Pixels Height | 800 pixels | | CPU Cores | 4 | | CPU Maximum Frequency | 1.3 GHz | | Screen Inches Square | 22 inch | | Screen Inches Width | 5.94 inch | | Screen Inches Height | 3.71 inch | |  |

## 3.2 Software and IDEs

The transmitter is a Java program developed as a Java application, which runs on windows laptop. The IDE used for developing the transmitter is Eclipse, using common Java libraries for TCP networking and sockets.

The receiver is built-in the Android Application for the display system, developed in Android Studio IDE in Java language for android.

* Android Studio IDE 3.3



* + Java over Windows OS
* Eclipse Oxygen 4.7
  + Java over Windows OS
* Arduino 1.8.12

## 3.3 Graphics and UI components

Most of the graphic components in the Android application are designed using Adobe Illustrator and Adobe Photoshop:

* Adobe Photoshop CC 2015
* Adobe Illustrator CC 2015

# 4 Part A

## **4.1 Recap**

# 

In part A, we had two independent parts communicate one another, in order to transfer data (figure 1). The roles are a *Transmitter* and a *Receiver* (client-server). The Transmitter is a laptop which receives file of sensors data streams, parsing it and transmitting the data to the receiver. The receiver is a part of the application installed in the car’s dashboard display system. On receiving data, the receiver parses, filters and enqueues the data for further processing by the application (e.g. graph updates, logging, etc.…). Each of the above roles are independent and can be replaces and improved independently.

File parsing and data extraction

Receive data and

enqueue for processing

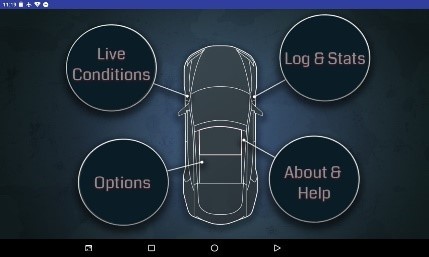
Transfer

Produces data stream file

(

fi

gure 1)



## **4.2 Data Transfer Solution**

(Part A)

As mentioned, there are two independent parts in part A which aim to transfer data between them (figure 2); the transmitter (laptop), and the receiver (display system application). In this project, we achieve this type of

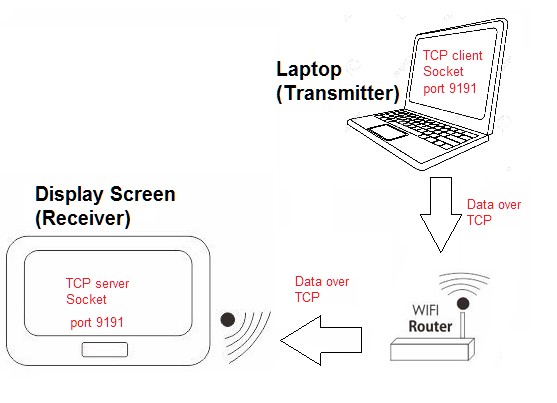
communication using **TCP networking protocol** and **TCP Sockets**. Both the transmitter and receiver must be connected to the same WI-FI network, then open a TCP socket on the same port allowing them to communicate. The receiver acts as a server and listens to incoming connections from the transmitter (client).

(

figure

2

)



## **4.3 High Level Design**

(Part A)

As mentioned earlier, the main components are a transmitter, and a receiver.

The transmitter (Java application), reads the streams file which the vehicle system (multiple sensors) produce, and parses each streamline according to the agreed structure. Afterwards, the application transmits data streams over TCP to the receiver application.

The receiver (Android application) contains a process for receiving the data sent over TCP, and enqueues relevant parts of the stream to a ProducerConsumer queue. The receiving thread also logs the data received into a log file located in the tablets file system, allowing the screen to display the loggings of the data. Another process in the receiver is responsible for consuming the data from the queue, filtering relevant data according to the user’s desire, and updating the real-time graphs.

(

fi

g

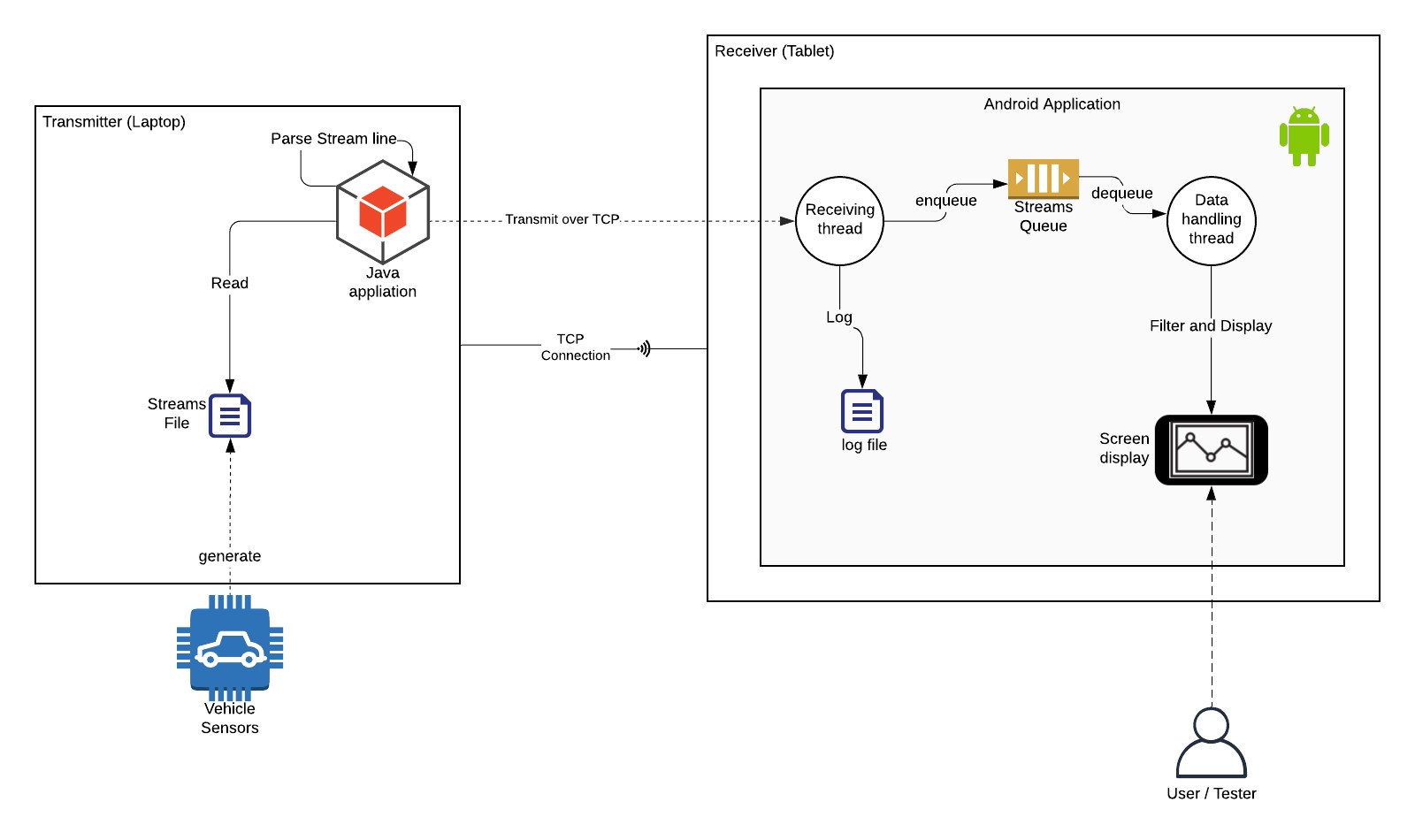
u

r

e

3

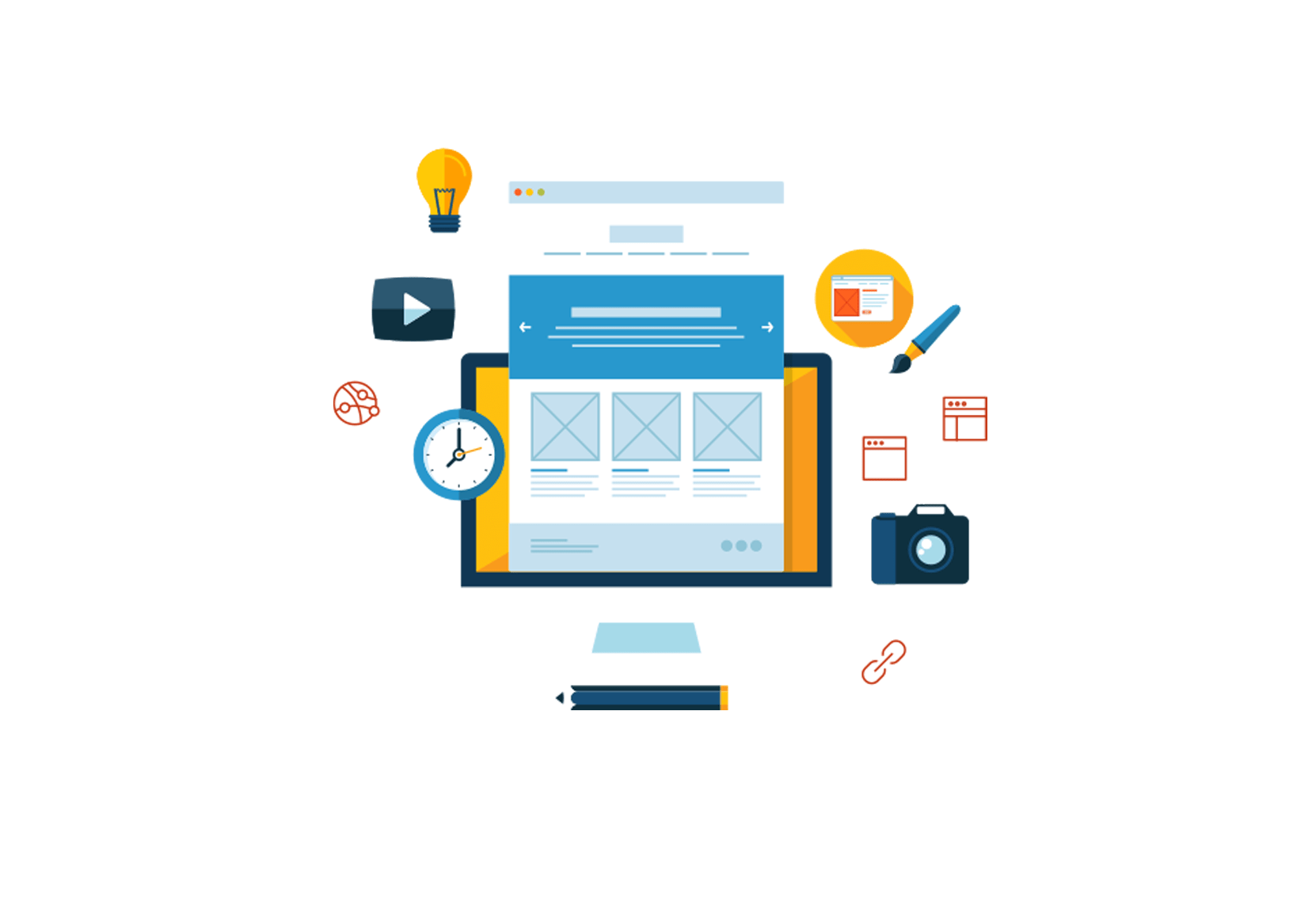
)



## **4.4 Features**

(Part A)

**Supported features of Part A**

* Support multiple graphs (Up to 3 graphs at a time)
* Support logging and data recording (Internal Storage)
* Choose different configurations (Resolution, Max value, Color…)
* Time axis scaling
* Different appearance theme

# 5 Part B

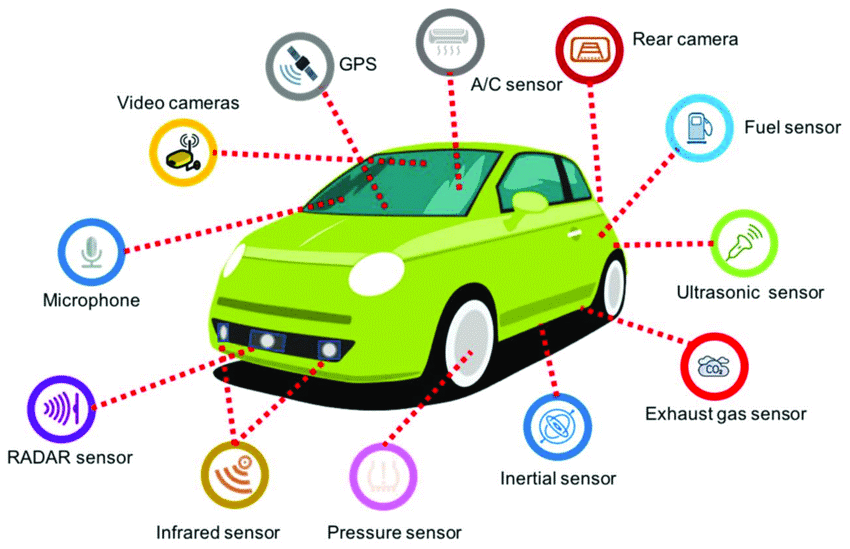
## **5.1 Overview and Incentive**

While in part A we imitated the vehicle’s operation of transferring sensors data via a Java PC application communicating over TCP connection with the Android application, in Part B we aim to receive sensors data collected by the vehicle, directly from the vehicle without any middle-man.

Such goal requires connecting the Android device directly to the vehicle’s computer using USB connection. In order to do that, the Android device must act as a serial host, and to achieve this we use an **OTG** cable.

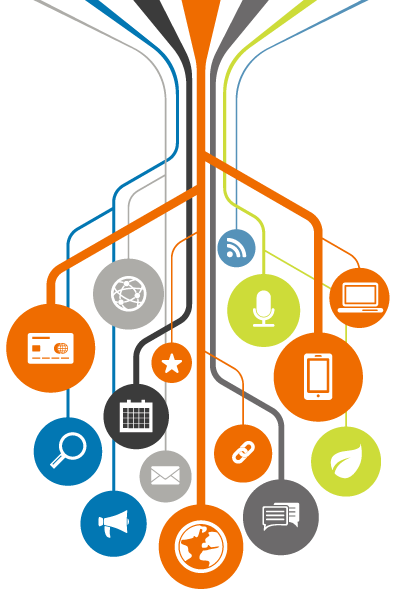
**OTG** (on-the-go) cable allows USB devices, such as tablets or smartphones, to act as a host, allowing other USB peripheral devices to be attached to them. Use of USB OTG allows devices to switch back and forth between the roles of host and device.

In addition, in part B we aim to migrate logs and data logging from the device’s internal storage into cloud storage/database in order to support larger data and an ability to inspect the logs not only from the Android screen, but also in the cloud storage UI.



## **5.2 Features**

**Additional features supported in Part B**

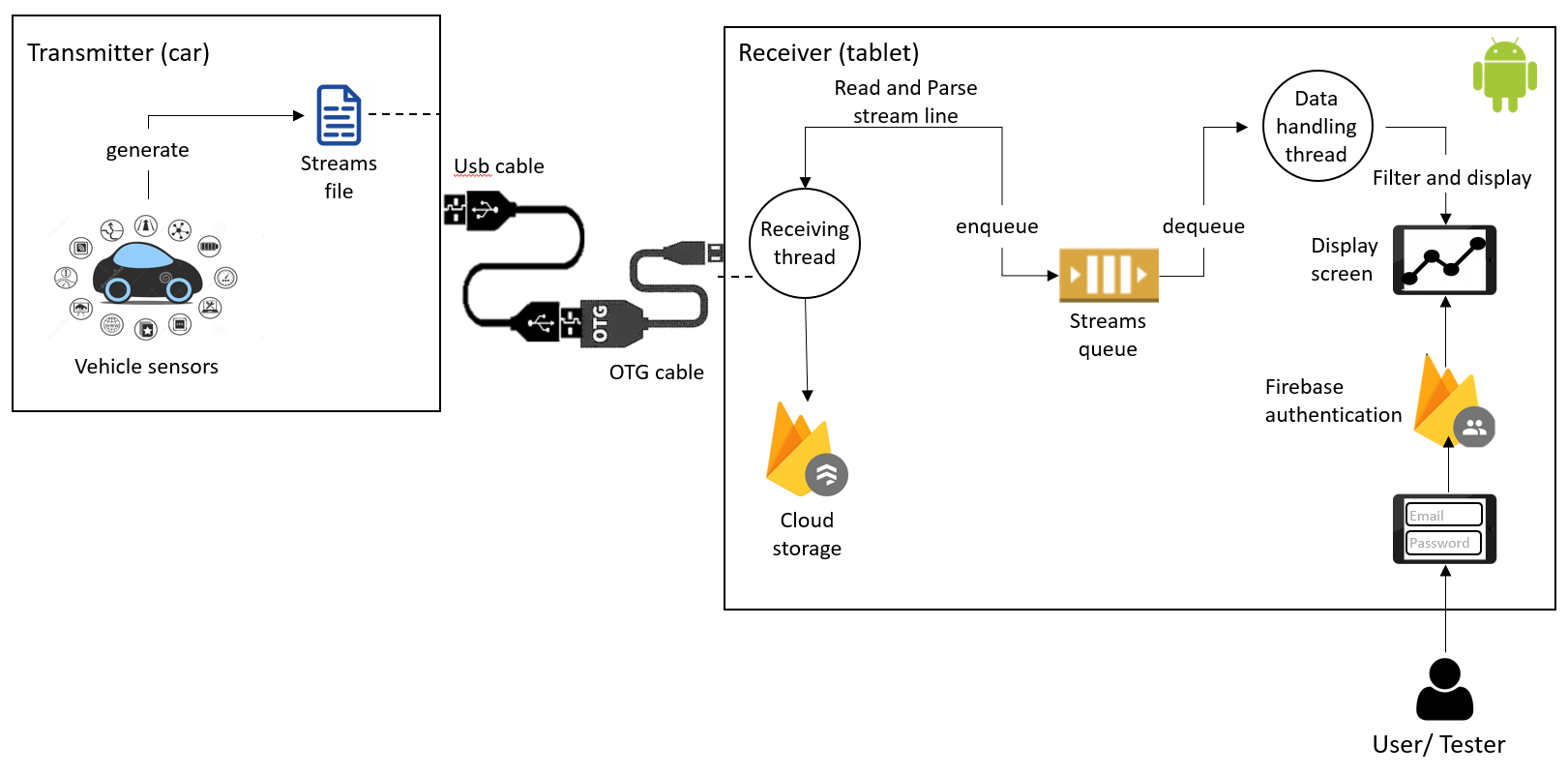
* Replacing TCP connection (Between PC & Android device) with USB connection, enabling data receiving from vehicle's sensors.
* Cloud storage logging of test results (Using Firebase Database and Functions)
* Log-in and User registration ability. (Using Firebase Authentication)
* UI improvements

## **5.3 High Level Design**

As described in [5.1](#_5.1_Overview_and), we are replacing the Java application and TCP connection with direct USB connection, using 1 USB cable and 1 OTG cable which enables the Android device to act as host.

Upon establishing the serial communication, the receiver (Android device) handles the data by parsing the stream line received and enqueues the parsed object producer-consumer queue. This thread also logs the data received into a cloud-based database, allowing the screen to display the loggings of the data.

The flow proceeds with another process responsible for consuming the data from the queue, filtering relevant data according to the user’s desire, and updating the real-time graphs.



## **5.4 Development Process**

## **5.5 Implementation**

### 5.5.1 User Interface

#### 5.5.1.1 Flowchart

#### 5.5.1.2 Screens and Usage

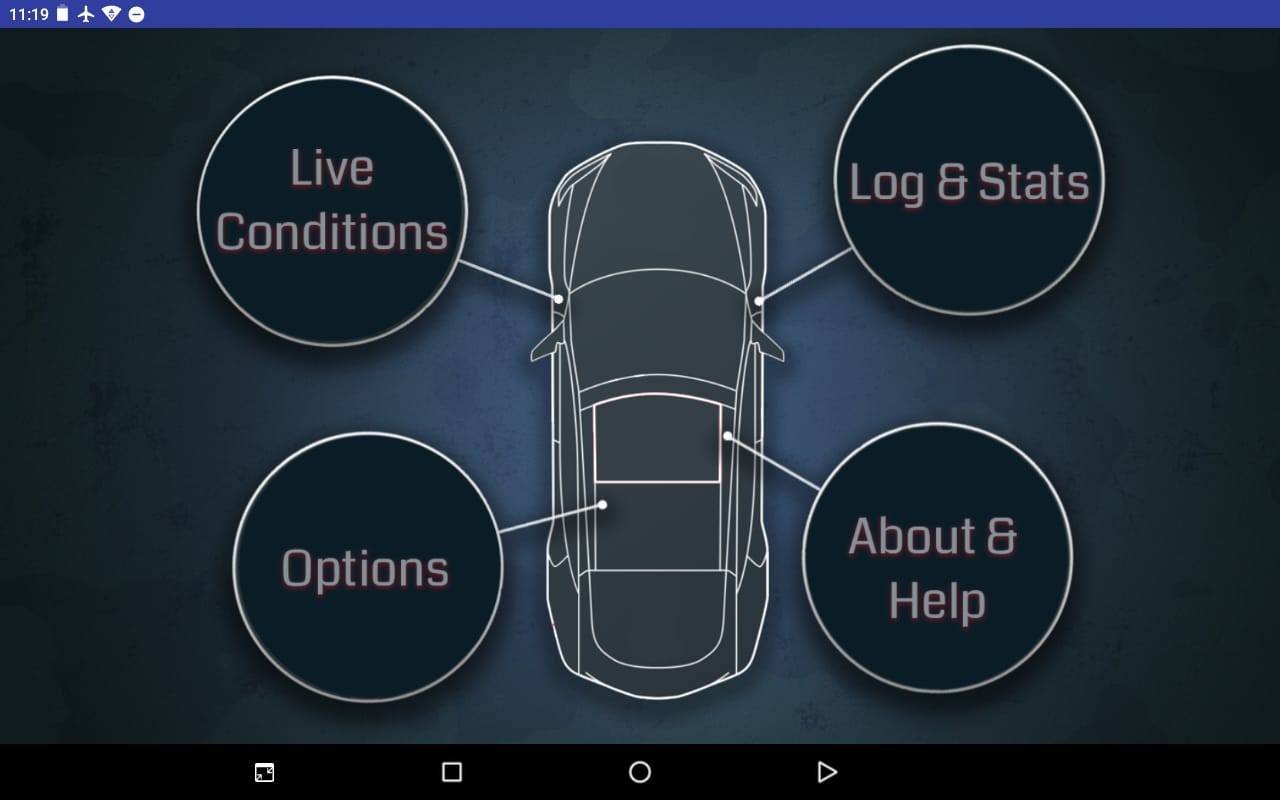


(figure 4

##### 6.1.2.1 Main page

This screen is the homepage of the application and the first screen that the user sees within entering the application.

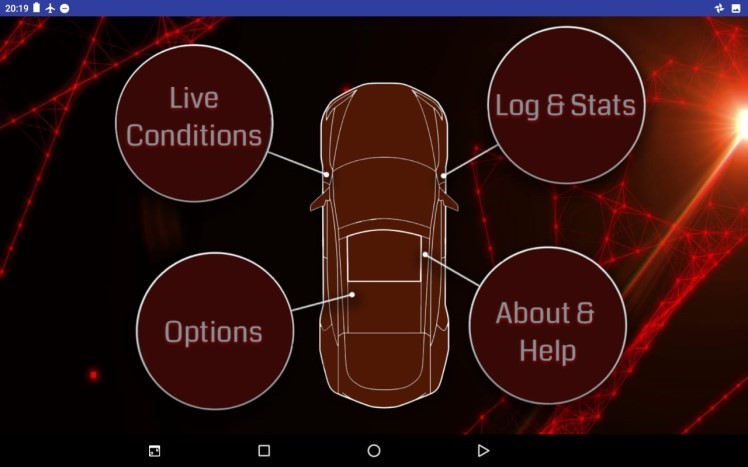
In the main page (Figure 1) the user can choose one of the following:



*Figure 1*

*Main Page*

User can in addition change the theme of the application (see Options) according to their preferences. This application supports two themes: blue and red).



*Fig*

*ure*

*2*

*-*

*Main Page*

*-*

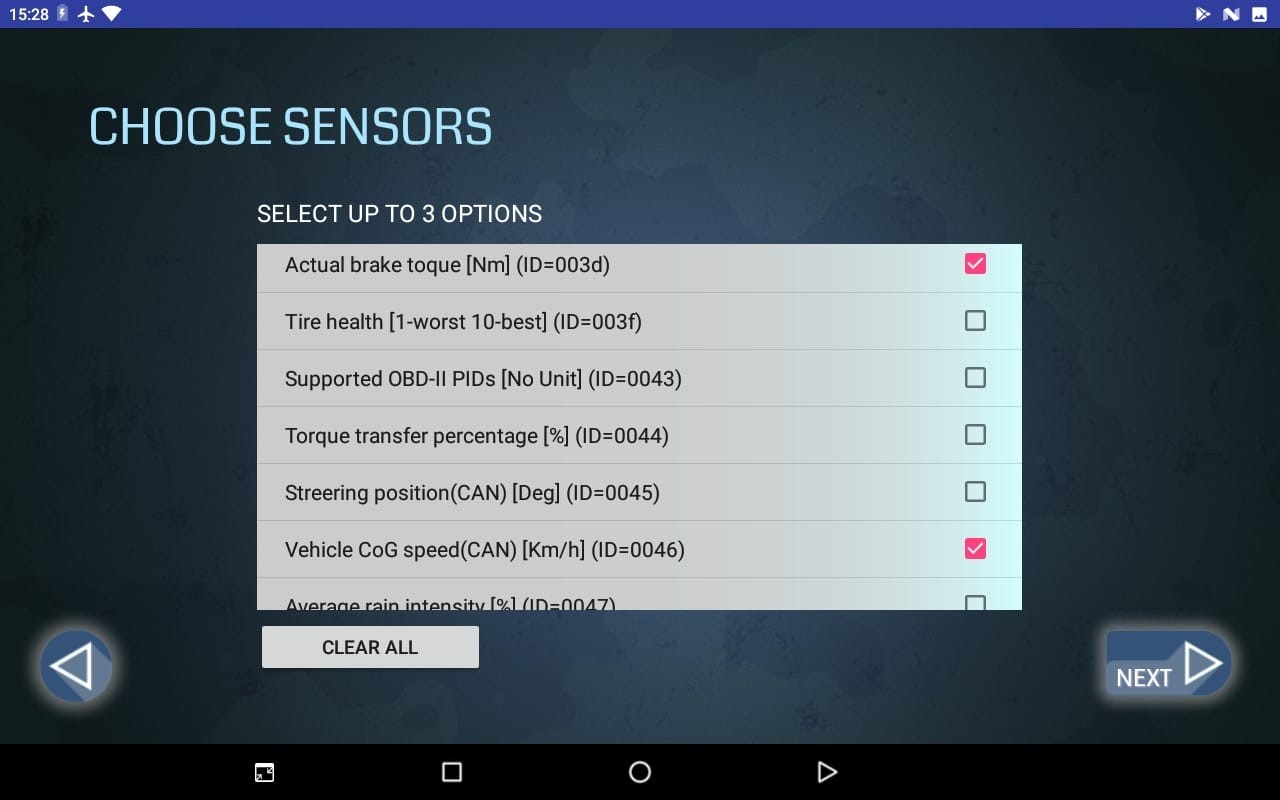
*red theme*

##### 6.1.2.2 Live Conditions – Choose Sensors

In this page **Error! Reference source not found.**the user will be asked to choose up to 3 of the sensors to be shown in the graphs of real time measurements.

The user can choose to clear all their selections by pressing CLEAR ALL button as seen below. Users can also go back to the previous page or the next one, after they have chosen all the options they desire to be shown, by clicking the appropriate button.

Users cannot proceed to the next screen (using the next button) unless at least one option was chosen.

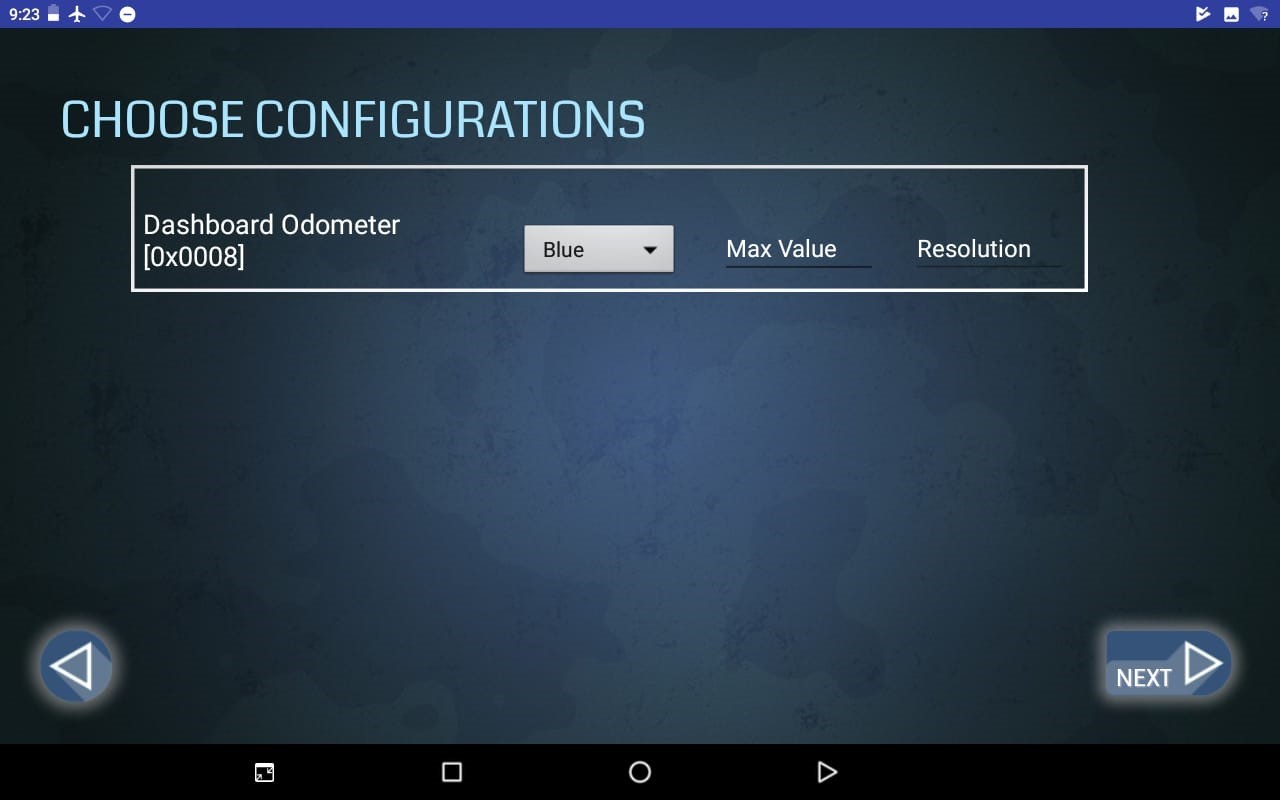


*Figure 3*

##### 6.1.2.3 Live Conditions – Choose Configurations

After choosing the sensors the user desires to be shown, the next step is to choose the configurations. When clicking the NEXT button, the user is redirected to the next page:

If no specific configurations were chosen, the default configurations (read by sensors configurations file) is used.

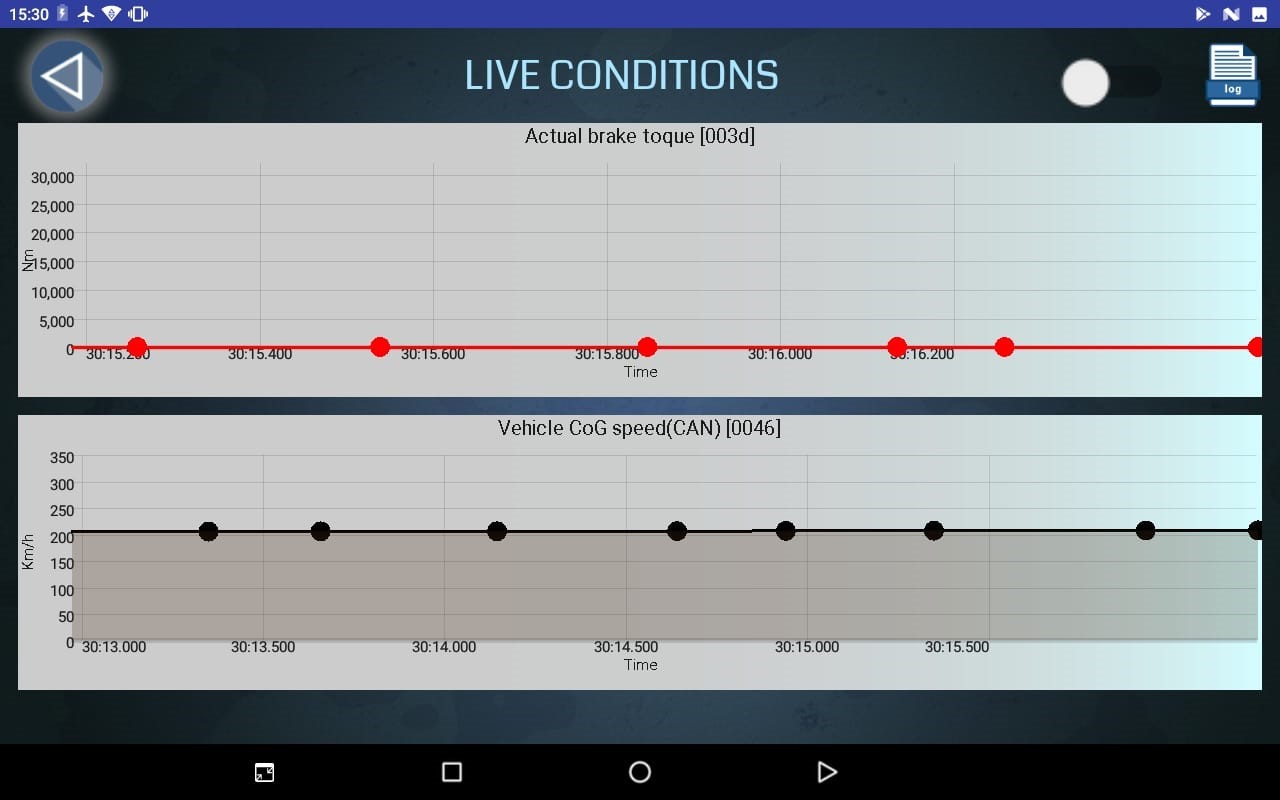
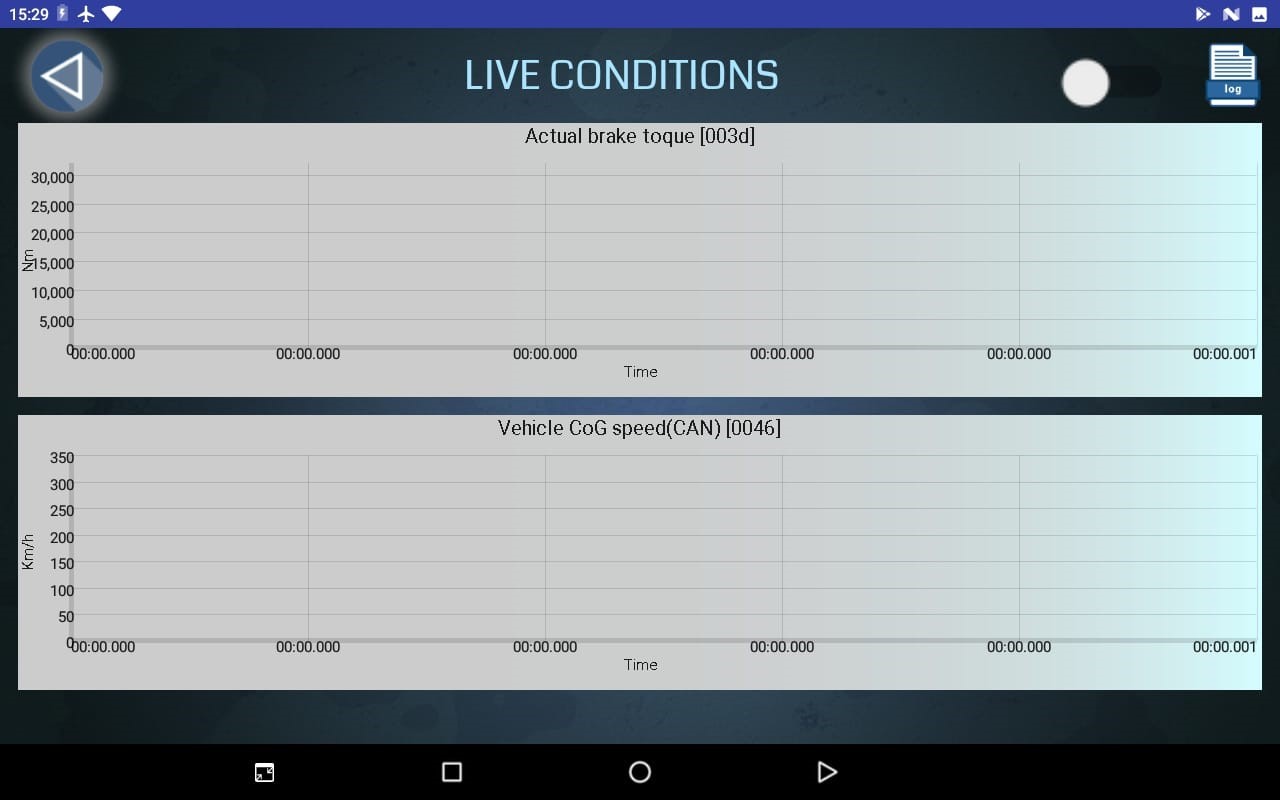


*Figure 4, 5*

##### 6.1.2.4 Live Conditions

The most significant page – the actual live conditions and graphs, according to the configurations chosen by the user. (Figure 4 and Figure 4 *,Figure 5*).

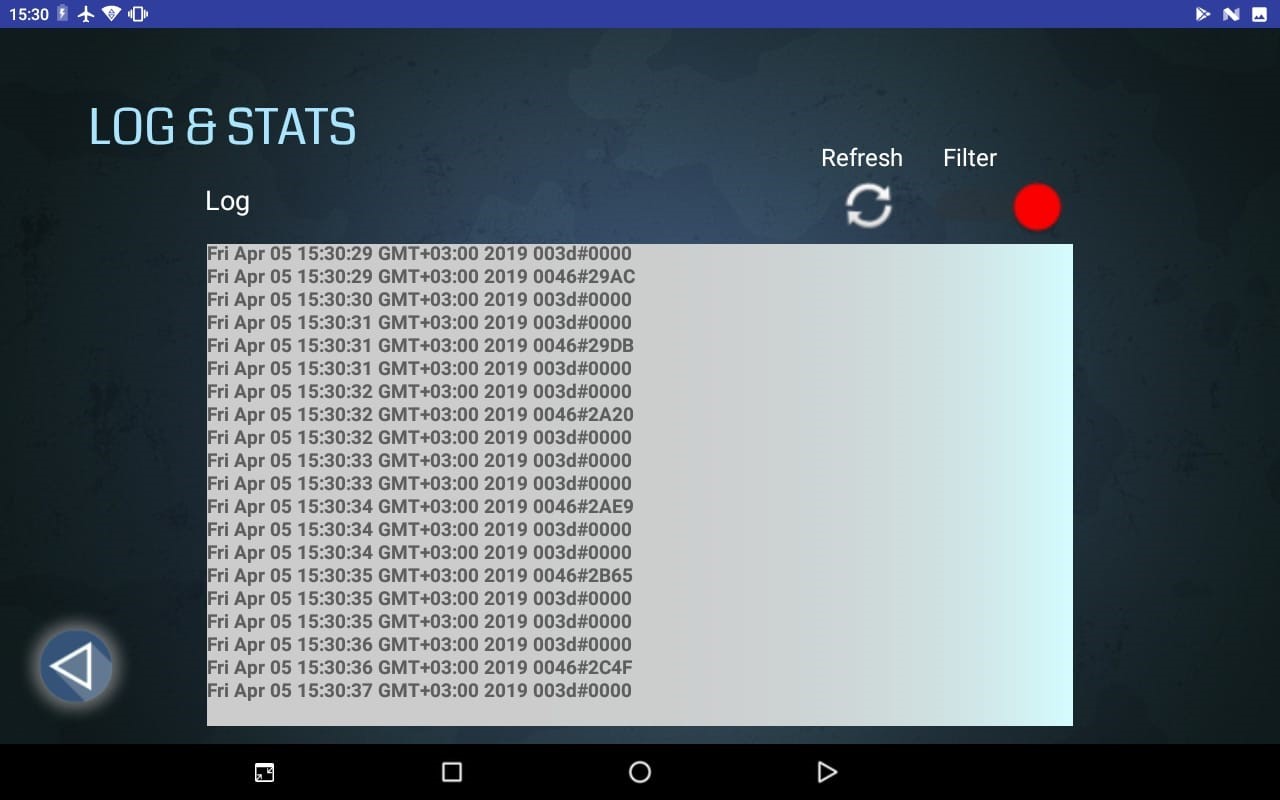
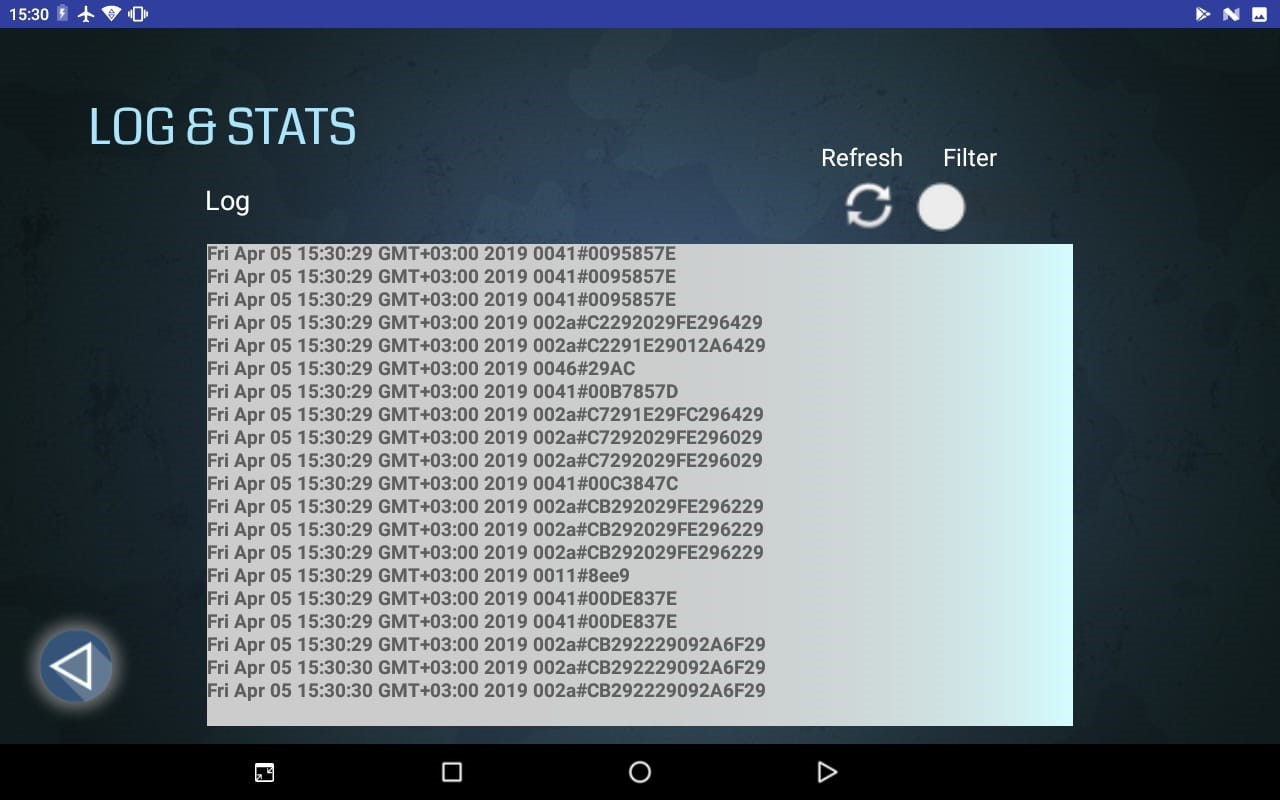
On the top right-hand corner, the user can switch the logging option on/off. When the option is On, the log file is displayed in the Log & Stats page.



*Figure 4 ,Figure 5*

##### 6.1.2.5 Log & Stats

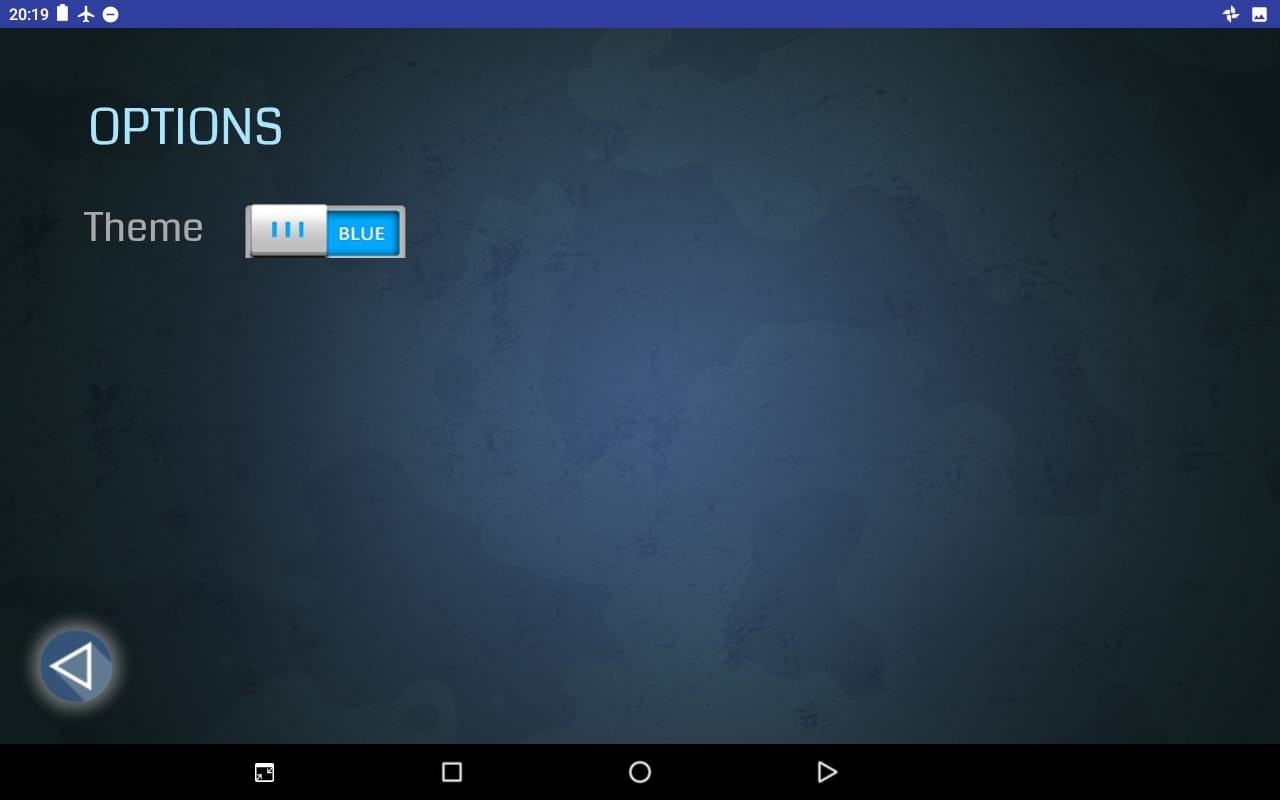
In Logs & Stats the user can find log that were recorded. It helps the user compare between results and save previous results which makes this application more efficient. This application provides logs with and without filtering according to the configurations (sensors) chosen in the Choose Sensors. (Figure 6 and 8).



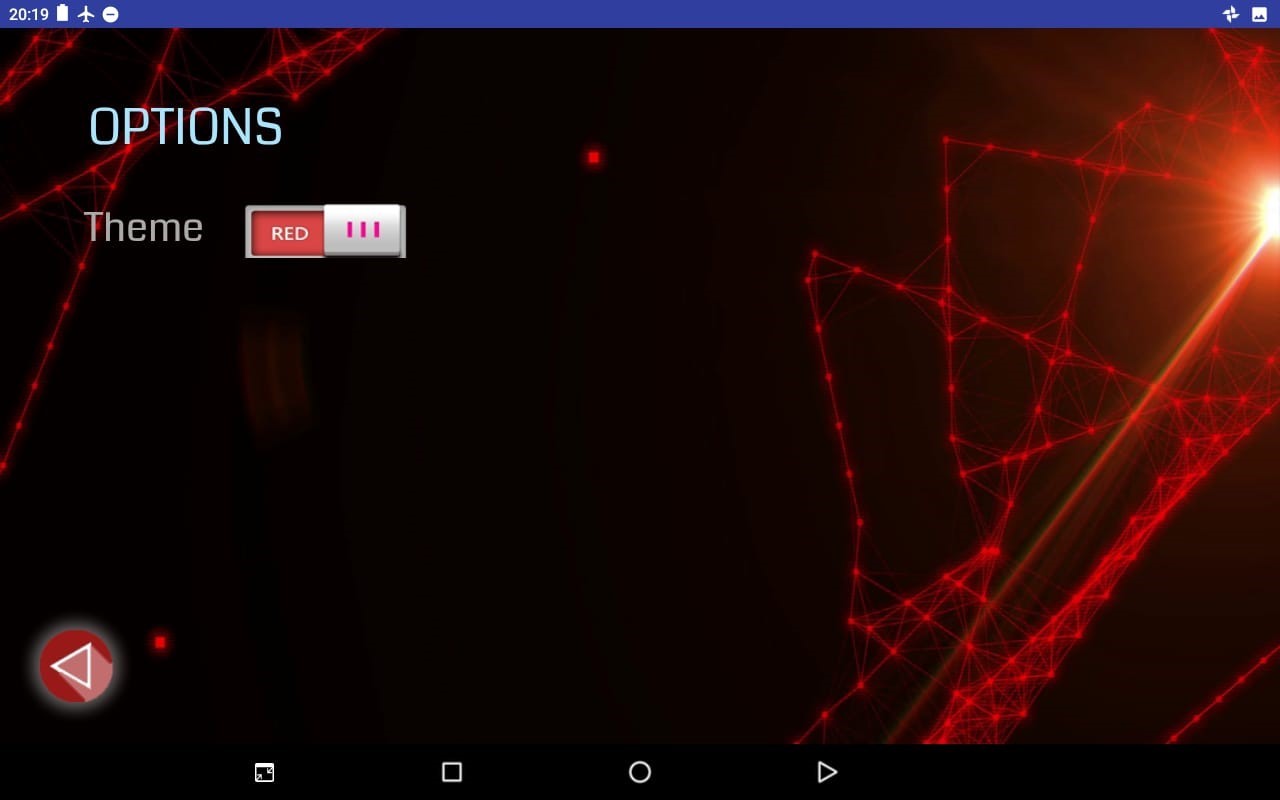
*Figure 6, Figure 7*

##### 6.1.2.6 Options

In options the user can change the theme of the application (Figure 8 and Figure 9). The application supports two themes: red and blue.



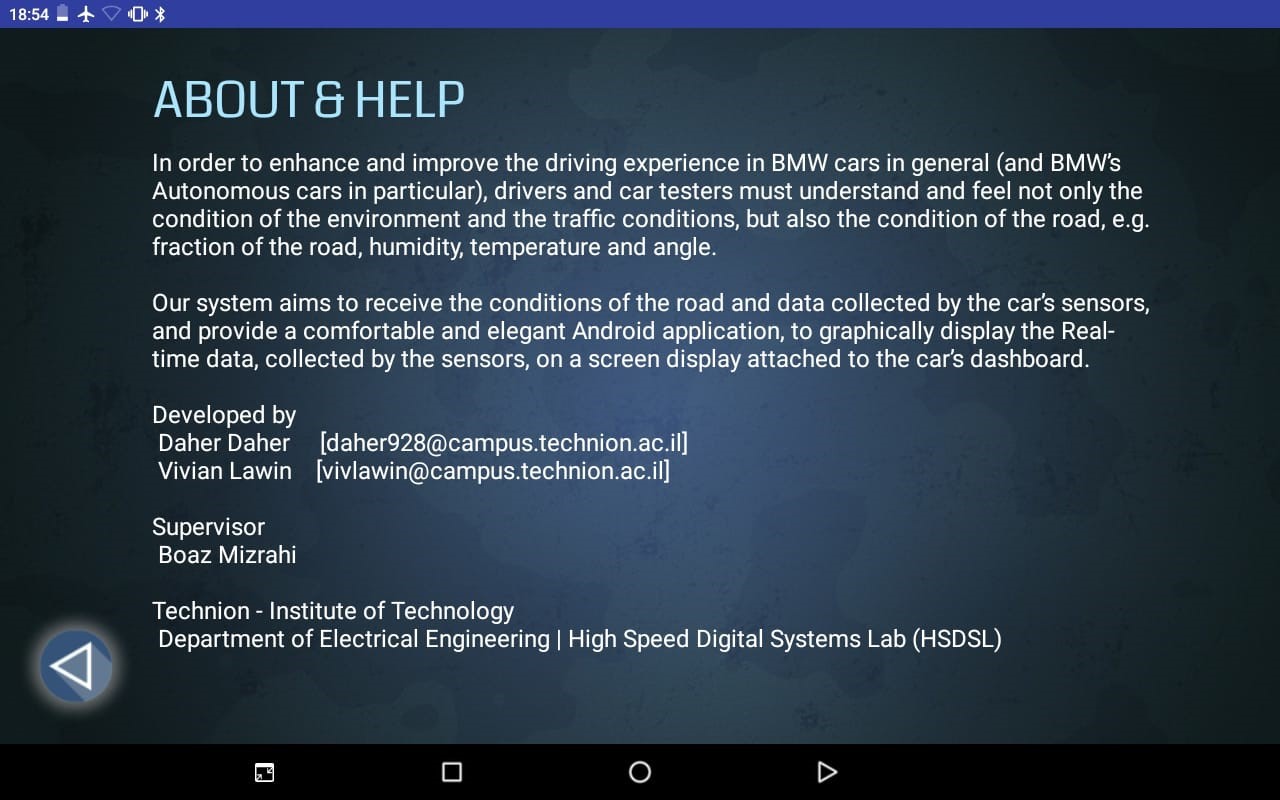
*Figure 8*



*Figure 9*

##### 6.1.2.7 About & Help

In this page you can find a brief summary about the application.



## **5.6 Back-end**

In this section, you will find a description of both the transmitter and receiver at a more low-level and parts of the implementation code of the operations they perform, as well as a description of the library used for managing graphs.

### 6.2.1 Transmitter

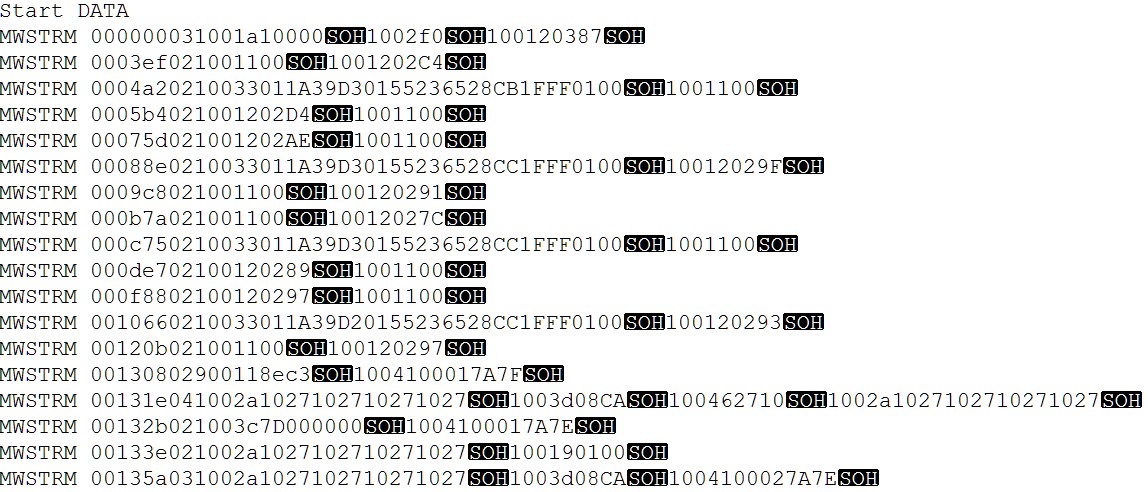
The transmitter, as mentioned earlier, is a Java application responsible for collecting the relevant data and transmitting them to the android application for further processing and displaying.

In order to do so, the Java Application sets up a connection over TCP

(see chapter 4.2 – Data Transfer Solution)

|  |
| --- |
| **static** *final* **String** androidId **=** "192.168.43.156"**;** **static** *final* **int** port **=** 9191**;**  **static** **Socket** socket; **static** **PrintWriter** printwriter; public static **void** ***setUpConnection*(**String Ip**,** int port**) {** socket **=** **new** Socket**(**Ip**,** port**);** // connect to server  printwriter **=** **new** PrintWriter**(**socket**.**getOutputStream**(),** **true);**  **}** |

Afterwards, the application proceeds to read the streams file written by the vehicle’s system. The following is an example of the streams file’s structure:



(See Appendix for full structure and parsing of the streams file).

For each streamline read, the application parses it into a DataSample object,

|  |
| --- |
| **public** **class** **DataSample** **{** public **int** timeStamp**;** public **int** samplesCount**;** public **List<String>** sampleIds**;**  public **List<String>** samplesData**;**  **}** |

and sends over the socket every sample data in the DataSample.

**DataSample** ds = ***parseLine***(line);

for(int i = 0; i < ds.samplesCount; i++) {

String sampleId = ds.sampleIds.get(i); String sampleVal = ds.samplesData.get(i);

***send***(sampleId + "#" + sampleVal);

}

The send method:

**public** **static** **void** ***send***(String **message**) { printwriter.write(**message** + "\n"); printwriter.flush();

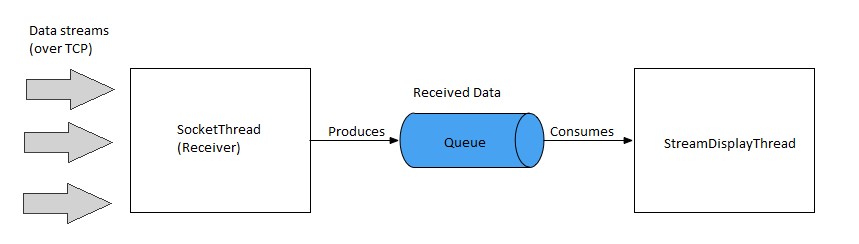
}

As seen in the code above, the transmitter reads a complicated line, parses it and transmits a simple stream of <SampleId#SampleVal>.

This operation is done for every streamline in the streams file, and the transmitter assumes there’s a receiver process listening to its streams.

### 6.2.2 Receiver

The receiver (as mentioned in chapter 5 – High Level Design) is composed of two main threads.



First, *SocketThread*, the thread responsible for the actual receiving of the data by setting up the other endpoint of the connection set by the transmitter. It listens on port 9191 to incoming data in the format <SensorId#Value>. On receiving data message, in case the logging option is set ON, it logs it into a log file (bmwLog) and then enqueues the data into the Producer-Consumer queue.

|  |
| --- |
| isr = new InputStreamReader(socket.getInputStream()); br = new BufferedReader(isr); while((message = br.readLine()) != null){ h.post(new Runnable() {  @Override  public void run() {  Timestamp ts = new Timestamp(System.currentTimeMillis());  Date d1 = new Date(ts.getTime());  String **logText** = d1 + " " + message + "\n"; if (AppState.isLogActive){  FileOutputStream **fileOutputStream** = context.openFileOutput("**bmwLog**",  Context.MODE\_APPEND);  **fileOutputStream**.**write**(**logText**.getBytes());  }    }  });  AppState.**queue**.**add**(message);  } |

The Producer-Consumer queue is shared with the second thread, *StreamDisplayThread*, which is responsible for handling the data. It dequeues the data and handles the filtering and updating the graphs.

|  |  |
| --- | --- |
| while (!AppState.queue.isEmpty() && running) {  **final String s = AppState.queue.poll();**    h.post(new Runnable() {  @Override  public void run() {  Timestamp ts = new Timestamp(System.currentTimeMillis()); Date d1 = new Date(ts.getTime()); final String id = s.split("#")[0]; final String val = s.split("#")[1];     |  | | --- | | **FILTERING** |   if (!AppState.selectedIds.contains(id)){ //  return;  } else {  final double double\_val = Integer.valueOf(val, 16); int **graph\_idx** = AppState.selectedIds.indexOf(id); int sid = AppState.selectedIds.get(graph\_idx); Sensor **currSensor** = AppState.getSensorFromId(sid); double resolution = currSensor.getConfig().getResolution();  double offset = currSensor.getOffset();  double **final\_val** = double\_val \* resolution + offset;    switch(graph\_idx){ case 0:  series1.**appendData**(new DataPoint(d1,final\_val),true,10,false);  break; case 1:  series2.**appendData**(new DataPoint(d1,final\_val),true,10,false);  break; case 2:  series3.**appendData**(new DataPoint(d1,final\_val),true,10,false);  break;  }  }  }  }); |

In the code above, while the queue is not empty and contains data to be parsed and viewed graphically, the thread enqueues the data, parses it into an id and a value, filters the data by the id (checking if the id was chosen earlier by the user). If the data is relevant, the corresponding graph shall be updated, otherwise ignore the data.

The ability to display the data graphically and in real-time graphs is obtained by the GraphView library.

See Appendix for GraphView library description and further information.

Also, see Appendix for sensors configurations file – the file that the application reads on startup to learn about the available sensors, and their default configurations.

# 7 Appendix

## 7.1 Stream File

The stream file as mentioned in section 6.2.1 - Transmitter, is a file produced by the vehicles system, and contains real-time information about the conditions and data collected by the sensors.

The stream file content is all Hexadecimal and is as follows:

**File name: MW\_samples\_original\_0001613\_20181114\_1754**



Each streams file must contain a line indicating the start of the data samples (“Start Data”). In the example above, the first line indicates that the following lines are data samples.

Parsing of a stream file line is done according to the following structure:

Streamline:

**MWSTRM AAAAAA BB C DDDD HHHH…HHH [SOH] DDDD HHHH…HHH [SOH]…**

Parsing:

**AAAAAA –** 6 Hexadecimal digits representing the timestamp.

**BB –** 2 Hexadecimal digits representing the number of samples in the streamline.

**C –** 1 Hexadecimal digit representing sample type (if not equal to 1, then we ignore the line).

**DDDD –** 4 Hexadecimal digits representing the sensor ID.

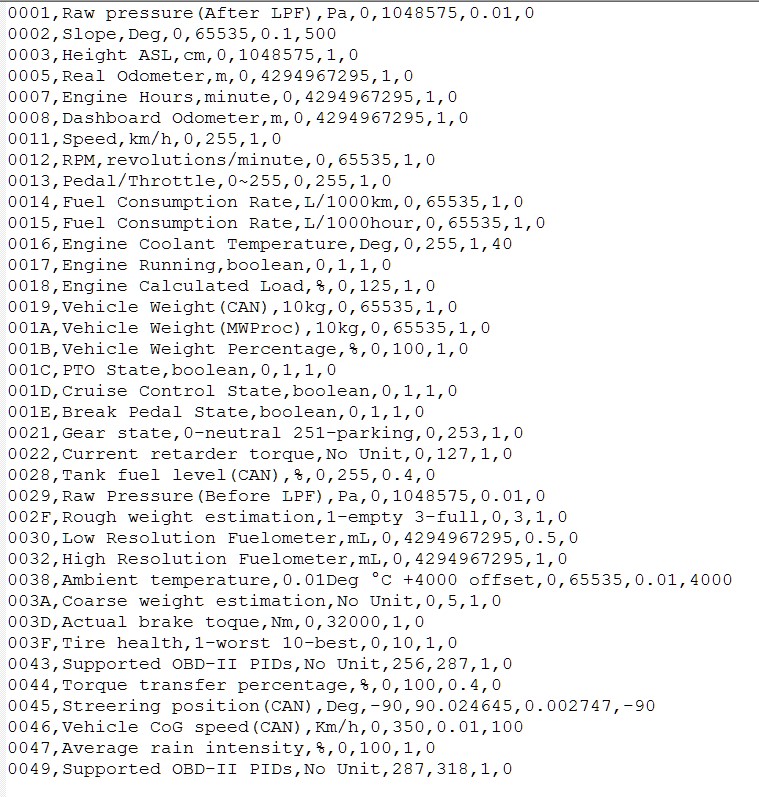
**H…HHH –** Hexadecimal digits representing the sample’s actual data.

**SOH –** Special character indicates an end of sample.

## 7.2 Sensors Configurations File

Sensors configurations file is a CSV file that the Android application (receiver) reads on startup to learn and get information about the available sensors, and their default configurations.

File name: sensors.csv Content:



Each line is parsed as follows:

**<SensorId>,<SensorName>,<Units>,<minVal>,<maxVal>,<Resolution>,<Offset>**

## 7.3 GraphView Library

GraphView is a library for Android to programmatically create graphs and diagrams.

The GraphView library enables programmers to create Line Graphs, Bar Graphs, Point Graphs or create their custom graph.

#### **Key Features**

* Different plotting types: Line Chart, Bar Chart and Points Chart and they can be plotted together as a combination.
* Draw multiple series of data
* Let the diagram show more than one series in a graph.
* Realtime / Live Chart - Append new data live or reset the whole data.
* Secondary Scale.
* Tap Listener
* Handle tap events on specific data points.
* Show legend.
* Custom label formatter
* Handle incomplete data
* Viewport
* Scrolling and Scaling / Zooming
* XML Integration
* Optional Axis Titles
* Set vertical and horizontal axis titles.
* Customizable - color and thickness, label font size/color and more

XML Layout file:

<com.jjoe64.graphview.GraphView android:layout\_width="match\_parent" android:layout\_height="200dip" android:id="@+id/graph" />

Java code:

|  |
| --- |
| GraphView graph = (GraphView) findViewById(R.id.graph);  LineGraphSeries<DataPoint> series = new LineGraphSeries<>(new DataPoint[] {  new DataPoint(0, 1), new DataPoint(1, 5), new DataPoint(2, 3)  });  graph.addSeries(series); |

Reference: <https://github.com/jjoe64/GraphView>