myOBC

Report on the final project
Mobile Development

Marco Zucca

September 2023

Contents

1 Description

`myOBC` (**my On Board Computer**) can be used to visualize information about your car, such as the current speed or RPM of the engine. It is possible to connect your Smartphone to a compatible OBD-II reader based on ELM327 via Bluetooth(classic). After the connection, some selected parameters are displayed on the screen, with the ability to log this data to a CSV file. The data logging feature is meant to be used for short tracks to monitor changes and do some post-processing later, like plotting a graph or doing some statistics.
It is also possible to save some information about the vehicle.

2 Usage

In this application, there are three Activities: The Main one, the Bluetooth device list, and a car information activity:

2.1 Main

In the main are displayed all the "gauges" and display (5) where the information about the car will be displayed, a string where the user can read the status of the connection (2), a button to stop the connection (4), a toolbar with two buttons one for connecting to an OBD interface (1) and another one to open a menu(3) where the user can start to record data in a CSV file or save information about their veichle

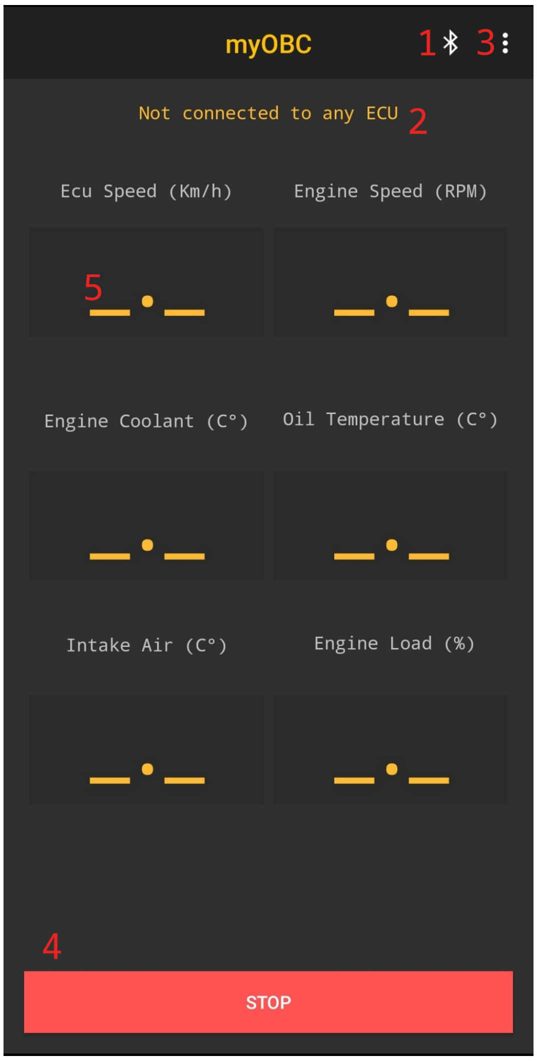

Figure 1: Screenshoot of the main activity of the app

2.2 Bluetooth list

In the Bluetooth Activity, when you start it, it starts scanning for nearby Bluetooth devices and then shows in a list all the devices(3) including the one already saved (2). The user can also stop the discovery of the devices (5) or start a new scan (4).
When the app is scanning for devices, the progress bar (1) is activated

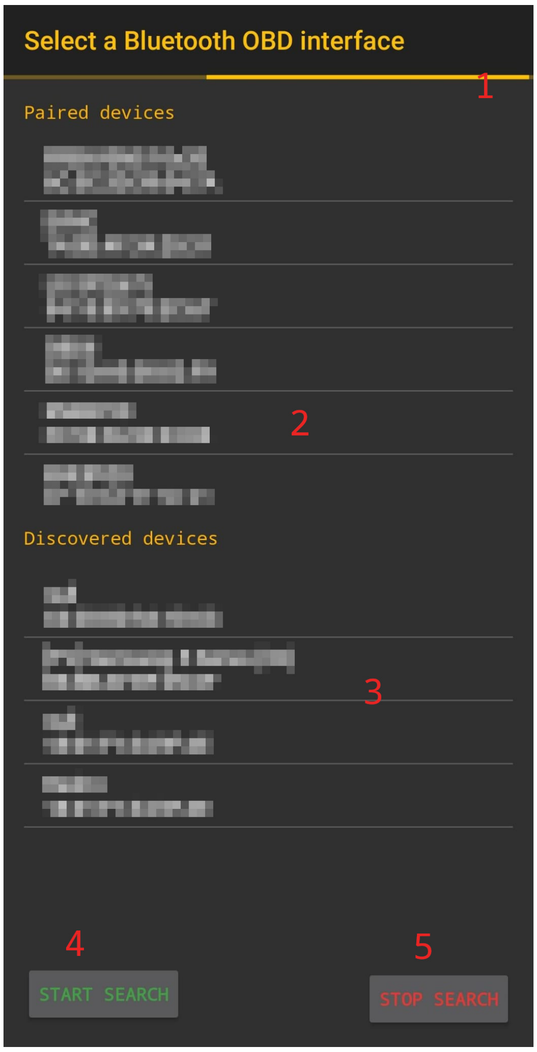

Figure 2: Screenshoot of the Bluetooth activity of the app

2.3 Car info

The activity about the vehicle information is quite simple: the user can insert some information about their vehicle and save it. The information will be kept in memory by clicking on the *Save* and reset with the *Reset*.

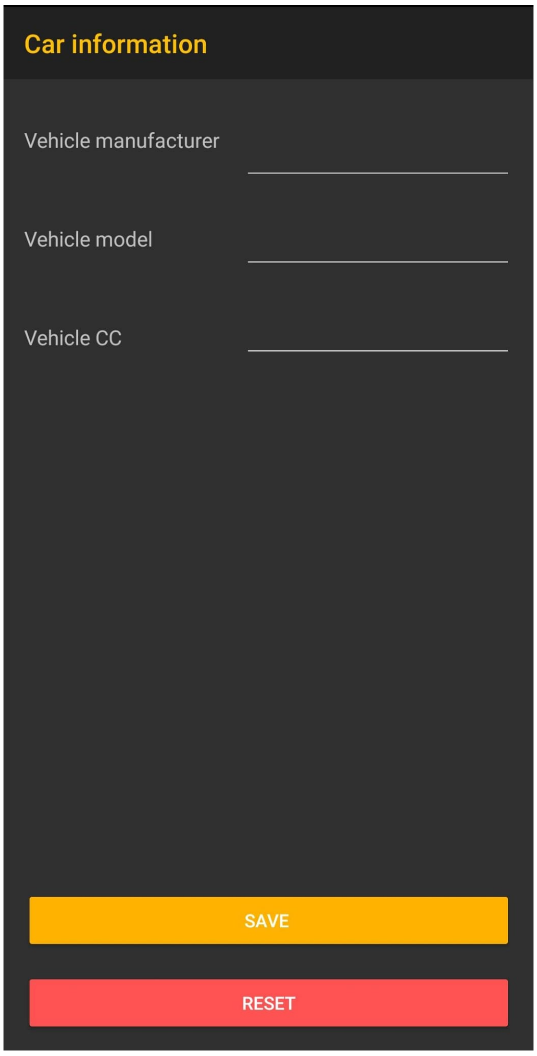

Figure 3: Screenshoot of the car info activity of the app

3 What is OBD?

OBD (On Board Diagnostics) is an interface present in modern cars (likely newer than 2000) used by mechanics or technicians to diagnose problems with a vehicle by gathering information about the car's ECU (Electronic Control Unit), like error code or car statistics to investigate and repair car's problems.
It is possible to use this interface to gather "live" data about the vehicle, like current speed or current RPM, to give the driver information about some data not shown by the car onboard computer, for example, some car does not show coolant temperature on the car cluster or like some data that are not relevant to all drivers (turbocharger pressure, air-fuel ratio, ...).
So, this interface is also used by car enthusiasts to create virtual custom gauges and displays using external devices to connect to OBD, including smartphones.

3.1 The protocol

There are five main OBD protocols, but car manufacturers only implement one of these, It is possible to know which protocol it is implemented by looking at the pinout of the OBD connector. But for the scope of this report, I will describe only the basic communication protocol.
There are many parameters that a user can request for the car's ECU, using parameter identification numbers (PIDs), These PIDs are defined in the SAE J1979 standard, but car manufacturers are not required to implement all and they also can implement them in other way or adding non-standard PIDs.
So the user can request a PID for specific information and the ECU will answer with that information and how to read that data.
Commands are organized in 10 "Service / Modes", each with different meanings, for example, the mode `01` indicates that the command is used to get current data.
After the service/mode, there is the actual PID, for example `0C` it is for RPM value.
So, a typical OBD command can be `01 0C`, to get information about the RPM of the engine, After receiving that command, the ECU replies with the information requested with a variable number of bytes depending on the information requested, for example, the previous command should be replied with 2 bytes. A list of some standard PIDs and the returned value can be found in this page

3.2 Send OBD command via an Android app

As described before, the principle of this protocol is quite simple, you just need to send a sequence of values and ECU will reply with what you asked. Let's ignore for now how to create a channel between the smartphones (described in detail in Implementation) and the ECU, and that we have created a socket between the two parts.

3.2.1 Kotlin OBD API

To simplify the process, I have used a Kotlin library kotlin-obd-api to send OBD commands throughout a Kotlin inputStream and receive ECU replies in an outputStream. This simplifies a lot the process because it abstracts the communication to only create an object giving the streams and invoke the desired method to obtain the requested information.
It is possible to create custom commands to extend the list of commands already implemented in the library, in this project, custom commands can be found in `OBDPids.kt`. In this class, I also wrote a "patch" for the RPM command, since it should return 2 bytes but in the library implementation process only 1 byte.

3.2.2 Initialization

The first thing to do before retrieving ECU information is to initialize the OBD adapter by sending specific commands. Since I used an OBD adapter based on the ELM327, you should initialize the communication by sending "AT" commands, which are specific commands used to send information to the adapter. For this project, I used the following initialization routine also used by "Torque", a famous Android app to interfacing to OBD:[1]

```
ATZ             ;reset adapter
ATE0            ;set ECHO off
ATE0            ;set ECHO off
ATM0            ;set AT memory off
ATL0            ;set AT linefeed off (no linefeed)
ATS0            ;set AT spaces off (no spaces)
AT@1            ;print AT device descriptor
ATI             ;print AT device information
ATH0            ;set AT header off
ATAT1           ;set adaptive timing mode
ATDPN           ;print protocol number
ATSP0           ;set protocol to 0

0100            ;show available PIDs, used as a check
```

[1]note that ';' is used like a comment character

4 Development and Testing

This application was developed with Android Studio in Kotlin. Since it is heavily based on Bluetooth connectivity, the application was mostly tested on my personal smartphone, since it was easier to use Bluetooth. So the target Android version was Android 11 with API 31, but I managed to run the application even on a newer device (Android 13).

The application was tested using an ECU Emulator, which emulates a car ECU when a device queries OBD command to it ELM327-emulator. This script is written in Python 3.x and it is possible to run on Windows and *nix, (but I only managed to run it on Windows) and requires a machine with a Bluetooth adapter. I ran the script by executing the following command:

```
python3 -m elm -p COM3 -s default -v -1 -a 38400
```

and opening a Bluetooth COM port via Windows connectivity settings.
This script was crucial for the implementation of the application, first of all, I could test the application without being in a car, and since it has a debug mode where it prints every command sent from the device, I was able to see if I was sending the correct command and if it was well formatted. This mode was also useful to learn how to initialize the OBD device. The application was also tested in an actual car, showing coherent results and data.

5 Implementation

The interesting part of the implementation can be split like so:

5.1 User interaction

The user has the ability to stop the connection by clicking on the *Stop* button only if the device is connected to the ECU, and when the user is connected to an ECU and want to connect to another one, first it need to click stop and then selecting a new device by clicking on the Bluetooth symbol. When the device is connected, that button is disabled.

5.2 Displays and exception

It is possible that the ECU can not reply to the requested PID, because the requested operation is not valid or because the ECU does not have that feature, If this happens a `NoDataException` is thrown. Since this error could happen in a normal environment, I decided to catch this exception and display the string `!DATA` to let the user know that the ECU replied with an empty response. It is even possible that the ECU replies with a value that is not a number, throwing the `NonNumericResponseException` exception and showing in the corresponding display the string `?NaN`.
I decided to handle this exception like this so the app can continue to work even if some parts are not working, and the driver is informed about that.
I tried to minimize the UI updating, since it is an expensive operation, by checking if the new value to be displayed, is the same as before, if it is true, I don't update the display, else I will update it with the new value.

5.3 Bluetooth

For the connectivity part, I used Bluetooth Classic, implementing the connection between the two parts using Android Bluetooth (classic) API and by following the Android official Manual and examples.
The first thing to do is create a Bluetooth adapter, then search for devices and display them to the user.
Before this, I add a separate list of all the already coupled devices, then start discovery for new ones using the methods exposed by the adapter. When the user clicks on the desired device, the Bluetooth address is sent with the intent mechanisms back to the caller, the main Activity, and then this address is used to create a `BluetoothClient` object and then call its method `connect()` where we connect with the Bluetooth device and start the OBD communication part.
(If the user starts a new discovery, the old one will be stopped to free resources.) The application needs Bluetooth connectivity and localization to be activated to be able to scan for Bluetooth devices and connect to them, so if the user does not have these features enabled, an alert dialog pops up telling the user to activate Bluetooth and/or localization.
To use these features, the user needs to consent to give some permission. It needs to be asked at runtime and not only declared in the Android manifest, so I decided to check if all permissions are granted immediately when the Bluetooth activity is created since it is considered a best practice. If the user dismisses the alert without

turning on the services, it will be prompted to the main activity, since the app could not work without those services enabled

5.4 OBD and Coroutines

After a Bluetooth socket is opened, it is possible to query commands to the ECU using the OBD interface. The app should display different values on the screen, so I had to use some kind of concurrent system. In this project, I used the Kotlin coroutine, a mechanism for asynchronous and concurrent programming, that can help in implementing responsive and efficient applications. In a nutshell, coroutines are used to execute part of codes in the background or in separate areas and help to keep the UI thread free from CPU-intensive works.
I have used coroutines to keep the UI thread "safe" by doing most of the communication in the IO thread and using the UI thread only to draw and write the value gathered.
For every parameter displayed I have created a method that asks for that specific information, It must be run in an IO thread. These methods are invoked (first in the intermediate method with the name of the data) in a method (`updateUI`) of the main activity to retrieve all the data needed in a sequential way, and then display it. The operation of display text must be executed in the Main thread. This method is then called in a loop (`display()`), where we launch that in another thread, so the Main thread is kept free from this operation, but the UI updates are still made in the Main thread. This operation is executed in a loop (break only if the user presses *stop*) so the displays are always kept updated and every OBD command sent to the ECU is delayed by 100 milliseconds, so the interface has some time to process the request and reply. Adding a delay is recommended when dealing with this type of communication, for this, the OBD API method has an option to specify a delay directly on the command to execute.
Generally, coroutines are automatically destroyed and they don't require the developer to join or destroy them, but since this loop becomes a pretty heavy task, when the user exits from the app or presses the *stop* button.
I chose to create different methods to increase the modularity of the code, so it will be easier in the future to add more values to monitor and manage better the concurrency.

5.5 Log file

In the toolbar in the main Activity, the user can find a menu where it is possible to start recording ECU data in a .CSV file. This task has been implemented by creating a Kotlin class where,by passing the destination file, will create a file with the given name in the external memory(only if the memory is available). This class exposes the path of the file so it can be used to tell the user where the file has been saved. It has three methods to initialize and manipulate CSV files: `addColumn`,`addRow`,`makeHeader`. The first two are quite self-explanatory: add a column or row in the CSV file, and the other one creates the header of the CSV file, by adding a column for every type of data that will be logged. The file is created only when the user clicks on the *Start log* menu entry only if there is no logging ongoing, and after calling the makeHeader method, the file is ready. The data to insert into the CSV file is gathered in `updateUI()` method (again, only if the user decides to log data), after updating the UI. By doing this the data is only requested one time and is the same as the one shown on the displays. To update the file, I used the `addRow` method by giving as a parameter the list of the values returned by the previous methods called to retrieve data from ECU plus the current timestamp, so it will be possible to plot a graph in the function of the time.

5.6 Car info

As said before, the user can save some information about their car. Those data are kept in memory using the *Shared Preference* mechanism, so data persist even if the user closes the app.

6 Problems encountered

There were some problems in the implementation of this application:

- **Bluetooth permissions**: This was one of the major problems that I had encountered, since Bluetooth Android API changes between versions. The most notable changes are how the permissions are managed and which permission a developer has to ask the OS. For example, there are new permissions to ask to scan for nearby devices that are not present in the older version, and other permissions deprecated in a new version that must be used in the older version (≤ 11). I managed that by requesting all required permission in the `onCreate()` method, so i didn't have to ask every time.
- **Concurrency**: For this project, I decided to manage concurrency with Kotlin coroutines, because they seem a good way to achieve great performance. However, the paradigm was quite different compared to Java AsyncTask so I needed to figure out how to use it.

7 Future projects and limitation

The biggest limitation of this app is probably the static display, where the user is stuck with the six initial displays and can not change them. The displays that I have chosen, are quite standard values that should work on most cars. But maybe some user has the need to monitor different values for their particular vehicle, for example, Diesel engines have a different parameter to monitor in comparison to a petrol one.