Escuela Politécnica Superior de Jaén

Grado en Ingeniería Electrónica Industrial

**UNIVERSIDAD DE JAÉN**
*ESCUELA POLITÉCNICA SUPERIOR DE JAÉN*

Trabajo Fin de Grado

# SISTEMA DE ADQUISICIÓN Y REGISTRO DE INFORMACIÓN A TRAVÉS DE LA INTERFAZ OBD II EN UN AUTOMOVIL

**Alumno: Jesús Gutiérrez Hidalgo**

Tutor: Antonio Abarca Álvarez
Dpto: Ingeniería Electrónica y Automática

**Junio, 2018**

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Electrónica

Don Antonio Abarca Álvarez , tutor del Proyecto Fin de Carrera titulado: “Sistema de adquisición y registro de información a través de la interfaz OBD II en un automóvil”, que presenta Jesús Gutiérrez Hidalgo, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Junio de 2018

El alumno: El tutor:

ABARCA ALVAREZ ANTONIO - 23786769P

Firmado digitalmente por ABARCA ALVAREZ ANTONIO - 23786769P
Nombre de reconocimiento (DN): c=ES, serialNumber=23786769P, sn=ABARCA ALVAREZ, givenName=ANTONIO, cn=ABARCA ALVAREZ ANTONIO - 23786769P
Fecha: 2018.06.25 19:08:52 +02'00'

Jesús Gutiérrez Hidalgo Antonio Abarca Álvarez

## RESUMEN

Este documento presenta el desarrollo de una aplicación móvil para la interpretación y representación de los datos procedentes del vehículo a través del protocolo OBD II.

OBD (*On Board Diagnostics*) es un sistema de diagnóstico a bordo en vehículos (coches y camiones). Actualmente se emplean los estándares OBD-II (Estados Unidos), EOBD (Europa) y JOBD (Japón) que aportan un monitoreo y control completo del motor y otros dispositivos del vehículo.

OBD II es la segunda generación del sistema de diagnóstico a bordo, sucesor de OBD I. Alerta al conductor cuando el nivel de las emisiones es 1.5 mayor a las diseñadas. A diferencia de OBD I, OBD II detecta fallos eléctricos, químicos y mecánicos que pueden afectar al nivel de emisiones del vehículo.

El sistema verifica el estado de todos los sensores involucrados en las emisiones. Cuando algo falla, el sistema se encarga automáticamente de informar al conductor encendiendo una luz indicadora de fallo.

Para ofrecer la máxima información posible para el mecánico, guarda un registro del fallo y las condiciones en que ocurrió. Cada fallo tiene un código asignado. El mecánico puede leer los registros con un dispositivo que envía comandos al sistema OBD II llamados PID (Parameter ID).

Generalmente el conector OBD II suele encontrarse en la zona de los pies del conductor, consola central o debajo del asiento del copiloto. Actualmente se puede conectar con la máquina de diagnosis de diferentes maneras, mediante Bluetooth, Wifi, USB, cayendo en desuso el protocolo de conexión, el puerto serie (RS232).

# INDICE GENERAL

INTRODUCCION



DESARROLLO DE LA APLICACIÓN

# INDICE ILUSTRACIONES

# CAPITULO 1. INTRODUCCIÓN

## 1.1 Justificación del proyecto

Aproximadamente en todo el mundo hay sobre 2000 millones de smartphones de los cuales la mayoría utiliza como sistema operativo Android, o algún derivado de este. El smartphone es en la actualidad una herramienta de trabajo más, y quizás una de las más imprescindibles a la hora del día a día.

En 2010 según algunos estudios estadísticos, existían sobre unos mil millones de unidades de vehículos en el mundo, lo cual indica que en países desarrollados equivale prácticamente a un vehículo por cada dos o tres personas.

Observando estos datos es llamativo como en los últimos años escaseaban los softwares para la comunicación entre la herramienta de trabajo que cualquier persona suele tener siempre a mano, y el medio de transporte privado más utilizado en el mundo.

Aunque es cierto que en poco tiempo se han desarrollado unas cinco o seis aplicaciones importantes en este campo (Torque, Car Scanner, OBD Arny…) se pensó que sería interesante el desarrollo de una nueva App para el conocimiento de este protocolo.

Se ha elegido el sistema operativo Android, ya que es el más utilizado tanto en smartphones como en tablets.

Aunque el sistema de diagnóstico puede darnos a saber varios parámetros, para el desarrollo de nuestra App hemos escogido algunos de los más usuales para comprobar que todo va a la perfección. Estos parámetros serían ampliables siempre que el protocolo característico de cada vehículo lo permita.

Estos parámetros son:

**Velocidad:** distancia recorrida por tiempo, todo vehículo nos muestra este valor por defecto, lo cual lo hace una buena elección para ver si nuestro valor recibido es correcto.

**Revoluciones:** vueltas por minuto del motor, otro dato dado siempre por el vehículo salvo alguna excepción, interesante siempre conocer el valor de este parámetro.

**Temperatura ambiental:** este nos da el valor en grados centígrados de la temperatura exterior al vehículo, este dato es usual en vehículos modernos, aunque no tanto en otros con falta de climatizador, por ejemplo.

**Posición relativa del pedal de aceleración:** da un porcentaje de 0% a 100 % de como de accionado está el pedal del acelerador, algo interesante para ver como nuestra aplicación varia este valor en tiempo real.

**Kilometraje:** nos muestras la cantidad de kilómetros recorridos desde que el vehículo borro sus códigos de falla.

**Tiempo desde arranque del motor:** nos muestra los segundos desde que el motor se puso en marcha, valor interesante sobre todo a la hora de ampliar la aplicación.

**Nivel de entrada del tanque de combustible:** dato algo más técnico que nos permite ver el porcentaje de combustible de entrada al motor.

**Presión barométrica absoluta:** nos da a conocer en Pascales la presión del motor en todo momento.

## 1.2 Antecedentes

Los sistemas de diagnóstico a bordo están en la mayoría de los coches y de los camiones ligeros en el día de hoy. Durante los años setenta y principios los ochenta, los fabricantes empezaron a utilizar medios electrónicos para controlar las funciones del motor y diagnosticar los problemas del motor. Esto fue principalmente para cumplir con los estándares de emisiones de la EPA(Environmental Protection Agency).

A través de los años los sistemas de diagnóstico a bordo se han vuelto más sofisticados. OBD-II, un nuevo estándar introducido a mediados de los noventa, proporciona control de motor casi completo y también supervisa las partes del chasis, del cuerpo y de los dispositivos accesorios, así como la red de control de diagnóstico del coche.

No todos los vehículos utilizan el mismo estándar ISO, esto dependerá de la fecha de fabricación de este, los estándares ISO más importantes son:

-ISO 9141, International Organization for Standardization, 1989.

-ISO 11898, International Organization for Standardization, 2003.

-ISO 14230, International Organization for Standardization, 1999.

-ISO 14320.

-ISO 15031, International Organization for Standardization, 2010.

-ISO 15765, International Organization for Standardization, 2004.

Modos de operación

El estándar OBD-II SAE J1979 define diez modos de operación (modos Pids):

Modo 01: Muestra los parámetros disponibles.

Modo 02: Muestra los datos almacenados por evento.

Modo 03: Muestra los códigos de fallas de diagnóstico ( Diagnostic Trouble Codes, DTC).

Modo 04: Borra los datos almacenados, incluyendo los códigos de fallas (DTC).

Modo 05: Resultados de la prueba de monitoreo de sensores de oxígeno (solo aplica a vehículos sin comunicación Controller Area Network, CAN).

Modo 06: Resultados de la prueba de monitoreo de sensores de oxígeno en vehículos con comunicación CAN.

Modo 07: Muestra los códigos de fallas (DTC) detectados durante el último ciclo de manejo o el actual.

Modo 08: Operación de control de los componentes/sistema a bordo.

Modo 09: Solicitud de información del vehículo.

Modo 0A(10): Códigos de fallas (DTC) permanentes (borrados).

La comunicación será diferente para cada uno de estos protocolos, lo que provocó que para unificar todos estos lenguajes se creó el ELM 327.

El ELM 327 es un circuito integrado basado en la mejora de sus antecesores ELM320, ELM322, y ELM323 agregando siete protocolos más consiguiendo así el entendimiento y procesamiento de hasta 12 protocolos de comunicación.

Cabe también destacar de este chip: su capacidad para elegir automáticamente el protocolo al que se expone, su completa configuración a través de los comandos AT y su modo bajo consumo.

A parte del OBD, podemos encontrarnos con otras variantes:

-El EOBD, es su variante europea, la mayor diferencia es que no se monitorizan datos sobre las evaporaciones del depósito de combustible.

EOBD es un sistema mucho más sofisticado que OBD II ya que usa “mapas” de las entradas a los sensores basados en las condiciones de operación del motor, y los componentes se adaptan al sistema calibrándose empíricamente. Esto significa que los repuestos necesitan ser de alta calidad y específicos para el vehículo y modelo.

-El JOBD es la versión japonesa del OBD II.

## 1.3 Objetivos

El objetivo principal de este proyecto, es la realización de una aplicación para la mayoría de los dispositivos con sistema operativo Android.

Esta aplicación servirá como interfaz para mostrar al usuario algunos parámetros dados por el diagnostico a bordo del vehículo.

Como aspecto interesante, la aplicación deberá permitir que el usuario elija en todo momento los parámetros a mostrar.

La aplicación debe mostrar los parámetros actualizados elegidos, teniendo un flujo continuo de información, que permita tener al usuario al tanto de cualquier variación en alguno de ellos.

## 1.4 Alcance

Desarrollo de una aplicación en lenguaje Android, que se compatible con la mayoría de smartphones y tablets, y que mantenga una comunicación constante con el diagnostico a bordo del vehículo.

Esta aplicación debe comunicarse con el vehículo a través de una conexión bluetooth, utilizando el circuito integrado ELM 327.

## 1.5 Descripción general del proyecto

La aplicación será compatible con versiones Android posteriores a Android 4.0 Ice Cream Sandwich, lo cual alcanzará prácticamente la totalidad de los dispositivos Android en activo.

La app desarrollada utilizará una comunicación bluetooth como canal de transmisión de información.

El mediador entre el automóvil y el dispositivo móvil será un ELM 327, el cual permitirá la interpretación de los protocolos característicos de cada vehículo.

### 1.5.1 Descripción básica del hardware

El único hardware necesario será el circuito integrado ELM 327 y el dispositivo Android.

Para las pruebas se ha utilizado un **Sony Xperia M2**, con una versión API 22 y con comunicación Bluetooth.

Algunas de sus características técnicas:

- Pantalla qHD de 4.8 pulgadas

- Procesador quad-core Snapdragon 400 a 1.2GHz.

- Conexión 4G LTE.

- Cámara de 8 megapíxeles Exmor RS.

- 1GB de RAM.

-8GB de almacenamiento interno.

-Batería de 2300mAh.

-Android 4.3 Jelly Bean.

Ahora vamos a ver las especificaciones técnicas del chip ELM 327:

El ELM 327 es un circuito integrado desarrollado por ELM electronics, empresa dedicada a la producción de microcontroladores e infrarrojos, aunque su producto estrella es la producción de microchips ELM que permiten la comunicación con los diferentes protocolos ISO.

**Pines del chip:**

**MCLR (PIN 1):** aplicar un nivel bajo a esta entrada restablecerá el C.I. Si no se utiliza, este pin debe conectarse a un nivel alto (VDD).

**Vmeasure (PIN 2):** esta entrada analógica se utiliza para medir una señal de 0 a 5V. Se debe tener cuidado para evitar que la tensión no salga de los niveles de suministro o se pueden producir daños. Si no se utiliza, este pin debe conectado a VSS o VDD.

**J1850 Volts (PIN 3):** esta salida se puede usar para controlar una fuente de voltaje para la salida del J1850 bus +. El pin dará una salida lógica de nivel alto cuando se requieren 8 voltios y dará salida a un nivel bajo cuando se necesitan 5 voltios. Si no se requiere esta capacidad de conmutación para su aplicación, esta salida se puede dejar en circuito abierto.

**J1850 bus + (PIN 4):** este activo de alto rendimiento se utiliza para impulsar el J1850 Bus línea a un nivel activo.

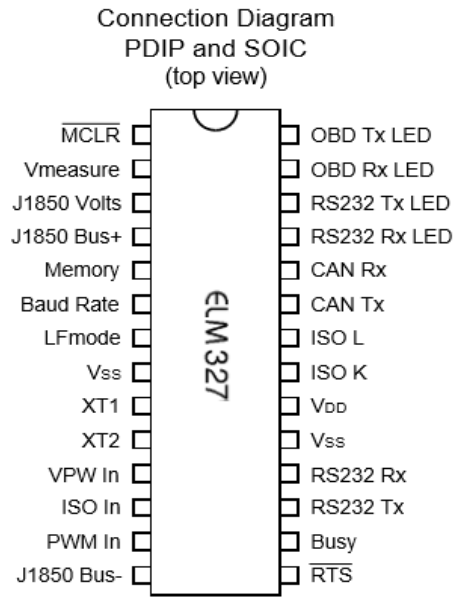
**Memoria (PIN 5):** esta entrada controla el estado predeterminado de la opción de memoria. Si este pin está en un nivel alto durante el encendido o reinicio, la función de memoria se activará por defecto. La memoria siempre puede activarse o deshabilitarse con los comandos AT M1 y M0.

**Baud Rate (PIN 6):** si está en un nivel alto durante el encendido o reinicio, la velocidad de transmisión se ajustará a 38400, si está en un nivel bajo, la velocidad en baudios será 9600.

**Lfmode (PIN 7):** esta entrada se usa para seleccionar el modo de salto por defecto que se usará después de un encendido o restablecimiento del sistema. Si está en un nivel alto, entonces por defecto los mensajes enviados por el ELM327 serán terminados con un retorno de carro y un carácter salto. Si está en un nivel bajo, las líneas serán terminadas por una vuelta del carro solamente.

**VSS (PINS 8 y 19):** el circuito común debe estar conectado a estos pines

**XT1 (PIN 9) y XT2 (PIN 10):** un oscilador de cristal de 4,000 MHz está conectado entre estos dos pines. Los condensadores de carga según lo requerido por el



1.Diagrama conexión ELM327

cristal (típicamente 27pF cada uno) también necesitarán ser conectados entre sí y el circuito común (VSS).

**VPW In (PIN 11):** ésta es la alta entrada activa para la señal de datos de J1850 VPW. Cuando está en reposo este pin debe estar a un nivel bajo de lógica. Esta entrada tiene un pulsador de forma de onda Smitch, por lo que no se requiere amplificación especial.

**ISO In (PIN 12):** esta es la entrada activa a nivel bajo para la señal de datos ISO 9141 e ISO 14230, debe de estar a un nivel lógico alto cuando está en reposo. No requiere amplificación especial.

**PWM In (PIN 13):** ésta es la entrada activa a nivel bajo para la señal de datos de J1850 PWM. Debe estar normalmente a nivel alto cuando recibe el bus. No requiere amplificación adicional.

**J1850 Bus- (PIN 14):** esta salida activa alta es usada para manejar el bus – del J1850. Sino se usa de debe de dejar en circuito abierto.

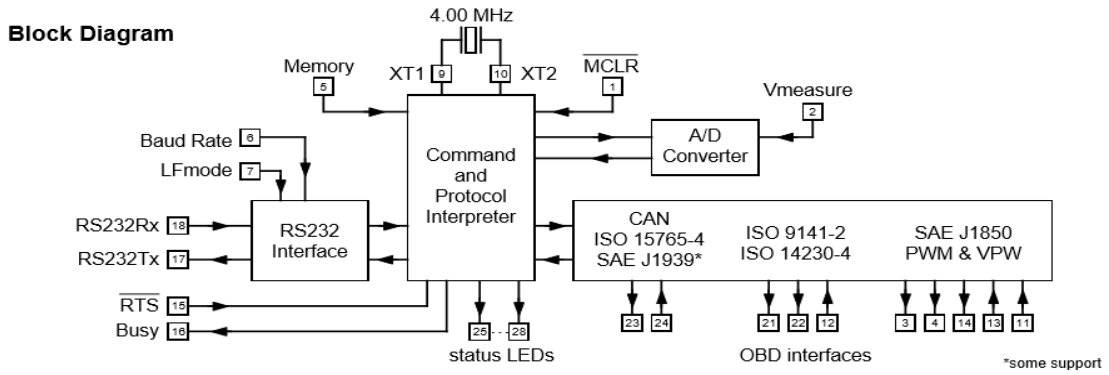
**RTS (PIN 15):** entrada activa a nivel bajo es utilizada para la interrupción del procesamiento OBD, para poder enviar un nuevo comando. Su estado natural es nivel alto, solo alcanza un nivel bajo cuando quiere interactuar con chip.

**Busy (PIN 16):** Si está en un nivel bajo, el procesador está listo para recibir comandos y caracteres ASCII, pero si está en un nivel alto, los comandos se están procesando.

**RS 232 Tx (PIN 17):** esta es la salida de transmisión de datos RS232. El nivel de señal es compatible con la mayoría de los C.I (la salida es normalmente alta), y hay suficiente unidad de corriente para permitir la interconexión utilizando sólo un transistor PNP, si se desea.

**RS 232 Rx (PIN 18):** es la entrada de datos de recepción RS232. El nivel de señal es compatible con la mayoría de los C.I (cuando está en modo reposo el nivel es normalmente alto).

**VDD (PIN 20):** Este pin es el pin de voltaje positivo, y debe ser siempre el punto más positivo del circuito.




2.Diagrama de bloques ELM327

**ISO K (PIN 21) e ISO L (PIN 22)**: estas son las señales activas de alto rendimiento que se utilizan para conducir los buses ISO 9141 e ISO 14230 a un nivel activo (dominante). Muchos vehículos nuevos no requieren la línea L – si el suyo no lo hace, simplemente puede dejar el PIN 22 en circuito abierto.

**Can TX (PIN 23) y Can RX (PIN 24)**: estas son las dos señales de interfaz CAN que deben estar conectadas al transceptor del C.I para un funcionamiento correcto.

**RS232 Rx LED (Pin 25), RS232 TX LED (PIN 26), OBD RX LED (PIN 27) y OBD TX LED (PIN 28)**: estos cuatro pines de salida son normalmente altos, y son conducidos a niveles bajos cuando el ELM327 está transmitiendo o recibiendo datos.

Elm Electronics ha estado produciendo el circuito integrado ELM327 multi-protocolo desde 2005. El ELM327 ha sido actualizado varias veces en respuesta a sus peticiones, y como resultado han producido muchas versiones del C.I a lo largo de los años. Todas las versiones del ELM327 apoyan los protocolos estándar de OBDII:

-SAE J1850 PWM

-SAE J1850 VPW

-ISO 9141-2

-ISO14230-4

-ISO15765-4

| | ELM327 v1.3a | ELM327 v2.2 | ELM327L v2.2 |
|---|---|---|---|
| Operating Voltage | 4.5V to 5.5V | 4.2V to 5.5V | 2.0V to 5.5V |
| Low Power (sleep) Mode | No | Yes | Yes |
| Settings Retained on Wake | – | Yes | Yes |
| RS232 Transmit Buffer Bytes | 256 | 512 | 2048 |
| AT Commands | 93 | 128 | 128 |
| CAN Frequency Check | No | Yes | Yes |
| Response Pending (7F) Support | No | Yes | Yes |

3.Diferentes versiones ELM327

## ELM327 V 2.2

La versión 2,2 del firmware es la versión más reciente. Este C.I requiere una alimentación de 4.2 V a 5.5 V.

ELM327L V 2.Este circuito integrado soporta todas las funciones ELM327 v 2.2, pero funciona sobre una gama más amplia de tensiones de alimentación (de 2,0 V a 5,5 V).

ELM327 V 1.3 A

El producto inicial y más vendido.

ELM327 V 1.4 A

Ya no está disponible, era básicamente un ELM327 V 1.3 A pero con modo bajo consumo para cuando el circuito no estaba en funcionamiento.

**Características eléctricas del ELM327:**

| Characteristic | | Minimum | Typical | Maximum | Units |
|---|---|---|---|---|---|
| Supply voltage, $V_{DD}$ | | 4.5 | 5.0 | 5.5 | V |
| $V_{DD}$ rate of rise | | 0.05 | | | V/ms |
| Average supply current, $I_{DD}$ | | | 9 | | mA |
| Input threshold voltage | | 1.0 | | 1.3 | V |
| Schmitt trigger input thresholds | rising | | 2.9 | 4.0 | V |
| | falling | 1.0 | 1.5 | | V |
| Output low voltage | | | 0.3 | | V |
| Output high voltage | | | 4.6 | | V |
| Brown-out reset voltage | | 4.07 | 4.2 | 4.59 | V |
| A/D conversion time | | | 7 | | msec |

4.Características electrícas ELM327

**Terminales del conector OBD II del vehículo:**

PIN 2: J1850 bus positivo

PIN 4: Tierra del vehículo

PIN 5: Tierra de la señal

PIN 6: Can high

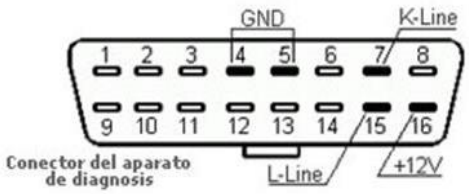
PIN 7: ISO 9141-2 línea K

PIN 10: J1850 bus negativo

PIN 14: CAN low

PIN 15: ISO 9141-2 línea L

PIN 16: Alimentación.




5.Esquema conexión ELM327

**Posibles ubicaciones del conector OBD II en el vehículo:**

-Zona pies del conductor, tanto debajo del volante como en la zona de fusibles.

-En la zona del cenicero, aunque los últimos modelos de coches ni tienen accesorios para fumadores.

-En la guantera del copiloto, configuración utilizada más en un pasado pero que podemos seguir viendo aún.

### 1.5.2 Descripción básica del software

Para el desarrollo de la app se ha utilizado **Android Studio**.

**Android Studio** es el entorno de desarrollo integrado oficial para la plataforma Android. Fue anunciado el 16 de mayo de 2013 en la conferencia Google I/O, y reemplazó a Eclipse como el IDE oficial para el desarrollo de aplicaciones para Android. La primera versión estable fue publicada en diciembre de 2014.

Está basado en el software IntelliJ IDEA de JetBrains y ha sido publicado de forma gratuita a través de la Licencia Apache 2.0. Está disponible para las plataformas Microsoft Windows, macOS y GNU/Linux. Ha sido diseñado específicamente para el desarrollo de Android.

Algunas de las **funciones** que ofrece Android Studio:

-Un sistema de compilación basado en Gradle flexible

-Un emulador rápido con varias funciones

-Un entorno unificado en el que puedes realizar desarrollos para todos los dispositivos Android

-Instant Run para aplicar cambios mientras tu app se ejecuta sin la necesidad de compilar un nuevo APK

-Integración de plantillas de código y GitHub para ayudarte a compilar funciones comunes de las apps e importar ejemplos de código

-Gran cantidad de herramientas y frameworks de prueba

-Herramientas Lint para detectar problemas de rendimiento, usabilidad, compatibilidad de versión, etc.

-Compatibilidad con C++ y NDK

-Soporte incorporado para Google Cloud Platform, lo que facilita la integración de Google Cloud Messaging y App Engine

**Estructura de un proyecto:**

Cada proyecto en Android Studio contiene uno o más módulos con archivos de código fuente y archivos de recursos. Entre los tipos de módulos se incluyen los siguientes:

- Módulos de apps para Android
- Módulos de bibliotecas
- Módulos de Google App Engine

De manera predeterminada, Android Studio muestra los archivos de tu proyecto en la vista de proyectos de Android, esta vista se organiza en módulos para proporcionar un rápido acceso a los archivos de origen clave de tu proyecto.

Cada módulo de la aplicación contiene las siguientes carpetas:

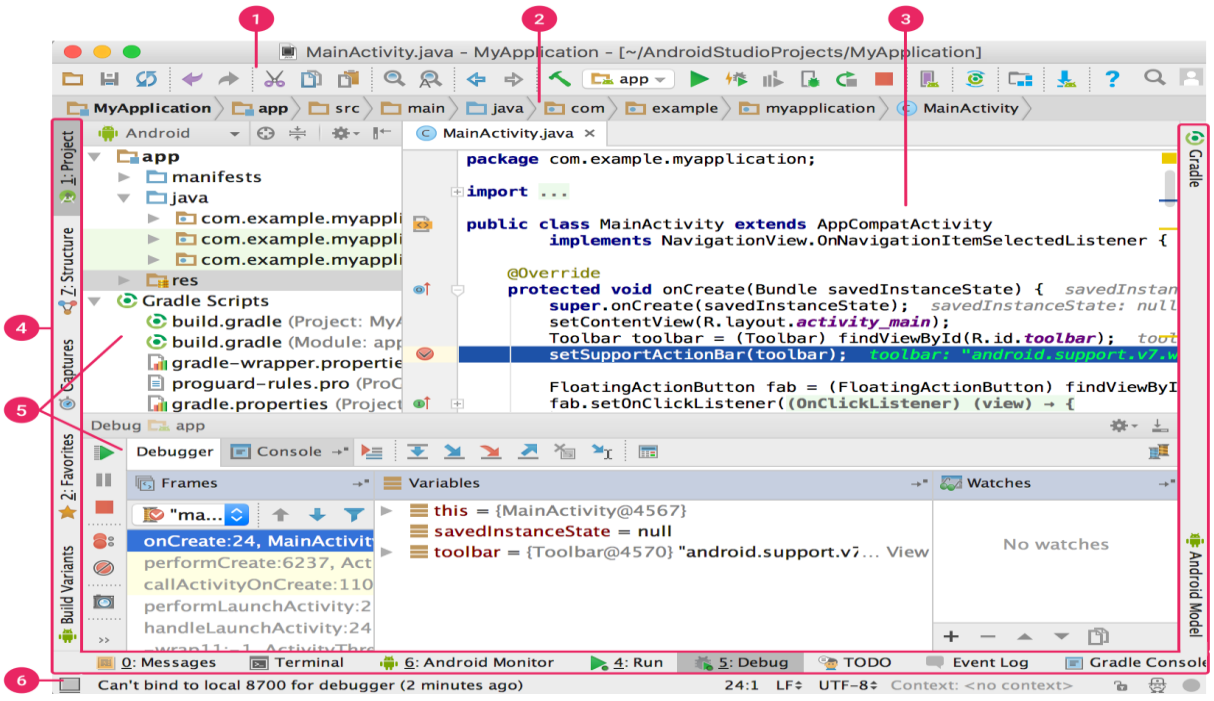
Manifests: contiene el archivo AndroidManifest.xml.

Java: contiene los archivos de código fuente de Java, incluido el código de prueba Junit.

Res: Contiene todos los recursos, como diseños XML, cadenas de IU e imágenes de mapa de bits.

**Interfaz de usuario:**

1. La **barra de herramientas** te permite realizar una gran variedad de acciones, como la ejecución de tu app y el inicio de herramientas de Android.
2. La **barra de navegación** te ayuda a explorar tu proyecto y abrir archivos para editar. Proporciona una vista más compacta de la estructura visible en la ventana **Project**.



6.Interfaz Android Studio

3. La **ventana del editor** es el área donde puedes crear y modificar código. Según el tipo de archivo actual, el editor puede cambiar. Por ejemplo, cuando se visualiza un archivo de diseño, el editor muestra el editor de diseño.
4. La **barra de la ventana de herramientas** se extiende alrededor de la parte externa de la ventana del IDE y contiene los botones que te permiten expandir o contraer ventanas de herramientas individuales.
5. Las **ventanas de herramientas** te permiten acceder a tareas específicas, como la administración de proyectos, las búsquedas, los controles de versión, etc. Puedes expandirlas y contraerlas.
6. En la **barra de estado**, se muestra el estado de tu proyecto y del IDE en sí, como también cualquier advertencia o mensaje.

## 1.6 Sistemas operativos para móviles

Un sistema operativo móvil o SO móvil es un conjunto de programas de bajo nivel que permite la abstracción de las peculiaridades del hardware específico del teléfono móvil y provee servicios a las aplicaciones móviles, que se ejecutan sobre él. Los sistemas operativos móviles están más orientados a la conectividad inalámbrica, los formatos multimedia para móviles y las diferentes maneras de introducir información en ellos.

**Estructura de un sistema operativo para móviles**

-El núcleo o **kernel** proporciona el acceso a los distintos elementos del hardware del dispositivo. Ofrece distintos servicios a las superiores como son los controladores o drivers para el hardware, la gestión de procesos, el sistema de archivos y el acceso y gestión de la memoria.

- El **middleware** es el conjunto de módulos que hacen posible la propia existencia de aplicaciones para móviles. Es totalmente transparente para el usuario y ofrece servicios claves como el motor de mensajería y comunicaciones, códecs multimedia, intérpretes de páginas web, gestión del dispositivo y seguridad.

-El **entorno de ejecución** de aplicaciones consiste en un gestor de aplicaciones y un conjunto de interfaces programables abiertas y programables por parte de los desarrolladores para la creación de software.

-Las **interfaces de usuario** facilitan la interacción con el usuario y el diseño de la presentación visual de la aplicación.

En cuanto a opciones de sistemas operativos en el mercado podemos destacar tres grandes opciones:

IOS: sistema operativo de Apple, se caracteriza por su gran fiabilidad, una sencilla y útil interfaz, y la gran optimización ante un hardware que suele ser de menor tamaño que el de un celular perteneciente a otro sistema operativo. Este SO es elegido por

millones y millones de personas, aunque suele ser menos asequible que otros SO más corrientes.

Android: es el sistema operativo más utilizado en el mundo, se caracteriza por ser un SO versátil, que da opción a una gran cantidad de variantes en cuanto la personalización y configuración del dispositivo por parte del usuario. Este sistema operativo está basado en Linux, y es utilizado tanto en smartphones de distintas marcas, como en tablets.

Variantes de Android: en la actualidad el mercado chino es cada vez más y más importante en Europa, desde el continente asiático llegan cada vez más marcas de smartphones, los cuales tienen unas prestaciones importantes a un precio bastante asequible. Estos smartphones usan como SO un derivado de Android, que no tendrá ningún inconveniente a la hora de utilizar aplicaciones desarrolladas para Android.

## 1.7 Conexión del hardware

El dispositivo ELM327 presenta la posible configuración de conexión de Wifi y Bluetooth. Para este presente proyecto se ha decidido utilizar una conexión del segundo tipo, ya que suponemos que la aplicación va a ser utilizada en carretera, donde en la actualidad, solo los coches más recientes y de alta gama poseen Wifi.

Por consiguiente, el smartphone que vayamos a utilizar para la conexión, debe poseer conexión bluetooth, la cual será utilizada siempre por nuestra app durante el funcionamiento de la misma.

## 1.8 Equipo y material necesario

- Sony Xperia M2
- C.I ELM327
- Vehículo
- Computador con Android Studio y SDK Tools de Java.

## 1.9 Presupuesto

| Equipo | Cantidad | Precio |
|---|---|---|
| Instalación de Android studio y herramienta Java | 1 | 0€ |
| Sony Xperia M2 | 1 | 0€ |
| C.I ELM327 | 1 | 2.85€ |
| Vehículo | 1 | 0€ |
| Hora Ingeniero Industrial | 200 | 10€/h*200h=2000€ |
| Presupuesto material total | | 2002.85€ |

| | |
|---|---|
| Presupuesto material total | 2002,85€ |
| 13% Beneficio Industrial | 260,37€ |
| Total | 2263,22€ |

| | |
|---|---|
| Presupuesto base de licitación sin IVA | 2263,22€ |
| 21% IVA | 475,27€ |
| Total | 2.738,49€ |

Asciende el presupuesto a la cantidad de DOS MIL SETENCIENTOS TREINTA Y OCHO con CUARENTA Y NUEVE EUROS.

# CAPITULO 2. DESARROLLO DE LA APLICACIÓN

# 2 Programación en Android Studio

Para programar con éxito en Android Studio lo primero que tenemos que tener en cuenta es la estructura de una aplicación.

Una aplicación está compuesta por diferentes activities, cada activity es básicamente una interfaz diferente con la que el usuario puede interactuar, ya que cada una tiene asociada un archivo java, pero también un archivo layout diferente(una interfaz diferente).

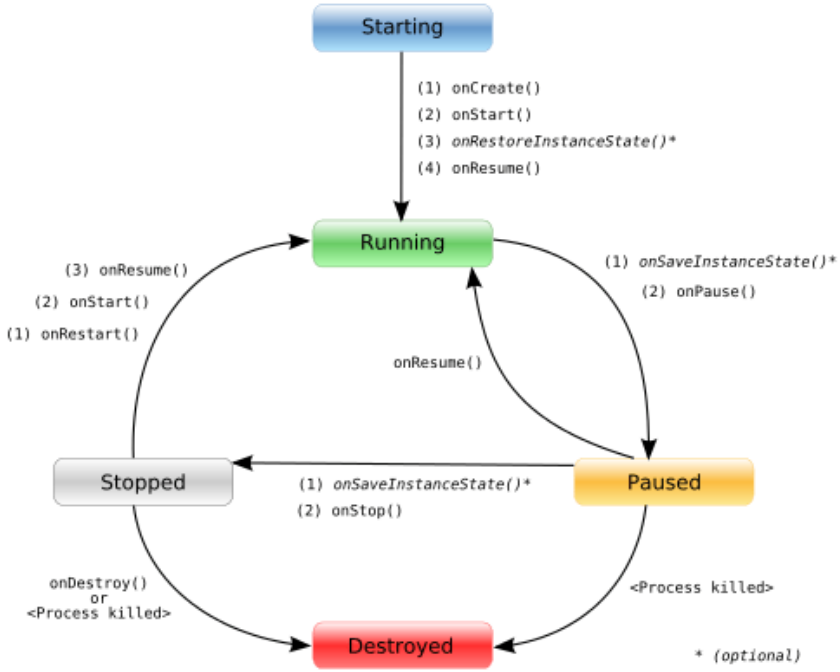
En el archivo java llamamos a su correspondiente layout, que es creado automáticamente cuando creamos la activity.

Siempre debe de haber una activity principal la cual será la inicial y de la cual derivan todas las activities que vayamos añadiendo.

Toda activity añadida debe ser configurada en el manifest de cada proyecto, el manifest es el lugar donde tenemos declaradas todas las activites diferenciando entre la activity launcher (principal) y las demás.

Cuando creamos una activity el asistente de diseño nos da la opción de crearla totalmente en blanco, o con un diseño en concreto ya impuesto, lo cual nos hará más breve el trabajo.

### Ciclo de vida de una activity




7.Ciclo de vida de una activity

Ahora vamos a definir las etapas de vida de una activity:

-Activa (Running): Está la primera en la pila de ejecución, el usuario ve la actividad y puede interactuar con ella.

-Pausada (Paused): Ha pasado a segundo plano, pero aún está visible porque otra actividad se coloca sobre ella, pero no la tapa del todo. En este caso, la actividad tapada puede ser cerrada por el sistema si necesita liberar recursos para la nueva actividad.

-Parada (Stopped): Ha pasado a segundo plano y está completamente tapada por la nueva actividad, en ese caso el sistema también puede optar por cerrarla si necesita liberar recursos.

-Destruida (Destroyed):ya no está disponible, se han liberado todos sus recursos y en caso de ser llamada, necesitaría comenzar un nuevo ciclo de vida.

Sus métodos de gestión son:

-onCreate(): Se llama al crear la actividad. Es donde se prepara la interfaz gráfica de la pantalla. Tras esta función, el proceso sobre el que se ejecuta la Actividad no puede ser destruido por el sistema. El siguiente método que se llama es onStart().

-onRestart(): Se llama cuando una actividad que se había parado vuelve a estar activa, justo antes de que comience de nuevo. Tras ella se llama a onStart() y su proceso no puede ser destruido ni durante ni tras su ejecución.

- onStart(): Se ejecuta justo antes de que la aplicación sea visible al usuario. El siguiente método de ciclo de vida llamado será onStop() u onResume(), dependiendo de la situación.

-onResume(): Se ejecuta en el momento en que la actividad se encuentra en la parte superior de la pila, justo antes de que el usuario pueda interactuar con ella. El siguiente método será onPause().

-onPause(): Se llama cuando la actividad va a ser tapada por otra, por tanto, se llama cuando se llame al onRestart() de otra. En este método debemos aprovechar para liberar todo aquello que consume recursos (para música, detener procesos…) o guardar datos de manera persistente. No podrá contener tareas lentas ya que hasta que no termine este método no podrá ejecutarse el onResume de la nueva actividad. El método siguiente será onResume() u onStop().

-onStop(): Se ejecuta cuando la actividad se hace invisible al usuario. Puede ser porque otra actividad la tape y entonces el siguiente método será onRestart() o porque la actividad haya sido destruida, llamado a continuación a onDestroy().

-onDestroy(): Se llama antes de destruir la actividad. Durante la destrucción de la actividad se perderán todos los datos asociados a ella por lo que en este método podrá ser utilizado para controlar la persistencia de datos. Se llamará cuando se ejecuta el

método de finalización finish() sobre la actividad o porque el sistema elimina la actividad para conseguir más recursos.

**Elementos de una activity**

Como podemos observar en el ciclo de vida de las activities, cuando esta se encuentra en estado activo(running), permite la interacción del usuario con la misma, ahora vamos a ver los elementos de Android que permiten están interacción.

El mismo Android studio clasifica los posibles elementos de una activity en los siguientes tipos:

-Widgets: Button, Listview, Checkbox, Progressbar…

-Layouts: Constraintlayout, LinearLayout, FrameLayout, RelativeLayout…

-Text: Textview, Plain Text, Password, Time, Date…

-Containers: RadioGroup, Listview, Gridview, Tabhost…

-Images: Imagebutton, Imageview, Videoview….

-Date: Timepicker, Datepicker, Textclock…

-Transitions: Imageswitcher, Stackview, Textswitcher…

-Advanced: NumberPicker, Include, TextureView…

-Google: Adview, Mapview….

-Design: CoordinadorLayout, AppbarLayout, FloatingActionButton….

-AppCompat: CardView, Toolbar, GridLayout

De los cuales en nuestra activities, va a predominar el uso de widgets, ya que son los que por definición son los elementos que permiten la interacción con el celular.

**Añadir widget a nuestra activity**

Para incorporar un widget a nuestra activity, primero habrá que añadir el widget físicamente en el layout de la activity, tendremos obligatoriamente que añadirle una id, para poder identificar cada widget en caso de que tengamos más de uno del mismo tipo. Por ejemplo, android:id="@+id/button"

Una vez que la layout está configurada habrá que dar vida a ese widget en la activity principal, antes del onCreate(), creamos una variable del tipo del widget que queremos dar vida. Por ejemplo, Button botón;

El siguiente paso sería asociar esta variable de tipo widget, a el widget que hemos definido anteriormente en el layout, para ello, dentro del onCreate() asociamos la id creada con el nombre de la variable con findViewbyId().Por ejemplo, botón=findViewbyId (R.id.button).

En el ejemplo que hemos ido proponiendo el botón ya tendríamos asociado el botón de nuestro layout con el botón definido en el archivo java, ahora solo habría que añadirle algún evento para permitir que el usuario pueda comunicarse con la app.

Para botones el evento más utilizado es el onClickllistener(); el cual llama a la función onClick() cada vez que hemos clickeado este widget.

Para otros widget como pueden ser switches o radiobuttons el evento onChangedListener(), suele ser muy utilizado ya que ejecutaría alguna acción impuesta por programador cuando el estado del widget en si, sufre un cambio, tras la interacción con el usuario.

## 2.1 Protocolo conexión Bluetooth

Lo primero a tener en cuenta a la hora de crear nuestra aplicación debe ser la conexión, ya que es la que nos suministra toda la información a procesar en nuestro dispositivo.

Para ello investigamos un poco en la página oficial de Android Studio, y sacamos las siguientes conclusiones.

Partiremos de un estado en el cual el dispositivo ya ha sido sincronizado con el móvil, para evitar cualquier problema esto quedará redactado en la sección de ayuda de la aplicación.

Así que para comenzar vamos a ver cómo conseguir ver la lista de dispositivos sincronizados por nuestro teléfono.

Para conseguir esto deberemos de usar la clase BluetoothAdapter, la cual Representa el adaptador local de Bluetooth (radio Bluetooth). El BuetoothAdapter es el punto de entrada de toda interacción de Bluetooth. Gracias a esto, puedes ver otros dispositivos Bluetooth, consultar una lista de los dispositivos conectados (sincronizados), crear una instancia de BluetoothDevice mediante una dirección MAC conocida y crear un BluetoothServerSocket para oír comunicaciones de otros dispositivos.

En una comunicación entre dos dispositivos siempre habrá un servidor y un cliente, el servidor escucha a través de un canal RFCOMM. Este esperará que algún dispositivo intente conectar con él, para crear una conexión segura.

El cliente es el dispositivo que crea un canal RFCOMM y solicita una conexión con el servidor.

El RFCOMM es un transporte de streaming orientado a la conexión a través de Bluetooth. También se conoce como el perfil de puerto serie.

En nuestro caso debemos realizar una conexión como cliente con el C.I, el cual tiene una dirección UUID.

Un identificador único universal (UUID) es un formato estandarizado de 128 bits para un ID de string empleado para identificar información de manera exclusiva. La ventaja de un UUID es que es suficientemente grande como para que puedas seleccionar

cualquiera al azar y no haya conflictos. En este caso, se usa para identificar de manera única el servicio Bluetooth de tu aplicación

EL UUID característico del ELM327 es el "00001101-0000-1000-8000-00805F9B34FB", tendremos que usar este ya que tanto el servidor como el cliente deben de tener la misma dirección UUID.

Vamos a ver los pasos básicos seguidos por la guía de Android studio para realizar una conexión como cliente:

1. Usando BluetoothDevice, sacado de los dispositivos sincronizados, obtén un BluetoothSocket, para ello, llama a createRfcommSocketToServiceRecord(UUID).
2. Esto inicializa un BluetoothSocket que se conectará al BluetoothDevice. El UUID pasado aquí debe coincidir con el UUID empleado por el dispositivo del servidor. Usar el mismo UUID implica simplemente codificar de manera rígida la string del UUID en tu aplicación y luego hacer referencia a este desde el código del cliente y el servidor.
3. Llama a connect() para inicializar la conexión.

Una vez que se realice esta llamada, el sistema realizará una búsqueda del SDP en el dispositivo remoto a fin de que coincida el UUID. Si la búsqueda tiene éxito y el dispositivo remoto acepta la conexión, este último compartirá el canal RFCOMM que se usará durante la conexión y se mostrará connect(). Este método es una llamada de bloqueo. Si, por algún motivo, la conexión falla o el método connect() caduca (después de unos 12 segundos), se generará una excepción.

Este proceso es el necesario para iniciar la conexión, ahora tendremos que administrar esta conexión, con el fin de tener un hilo de comunicación entre servidor y cliente.

El InputStream y OutputStream administran transmisiones a través del socket, mediante getInputStream() y getOutputStream().

Para escribir y leer datos debes inicializar un subproceso, esto es importante porque los métodos read(byte[ ]) y write(byte[ ]) son llamadas de bloqueo. read(byte[ ]) aplicará un bloqueo hasta que exista algo para leer del flujo. Por lo general, write(byte[ ]) no aplica bloqueos, pero puede hacerlo a los fines de controlar el flujo si el dispositivo remoto no llama a read(byte[ ]) con suficiente rapidez y los búferes intermedios están completos.

## 2.2 Proyecto inicial

Para comenzar a desarrollar la app se creó primeramente una app tipo chat la cual sirvió para:

- Conseguir mostrar los dispositivos sincronizados, y actuar respecto la elección del usuario.
- Realizar una conexión bluetooth con el dispositivo seleccionado.
- Conseguir ver en cada momento el estado de la conexión.

- Poder enviar diferentes tipos de órdenes y obtener la respuesta del ELM327.

### Interfaz del proyecto inicial

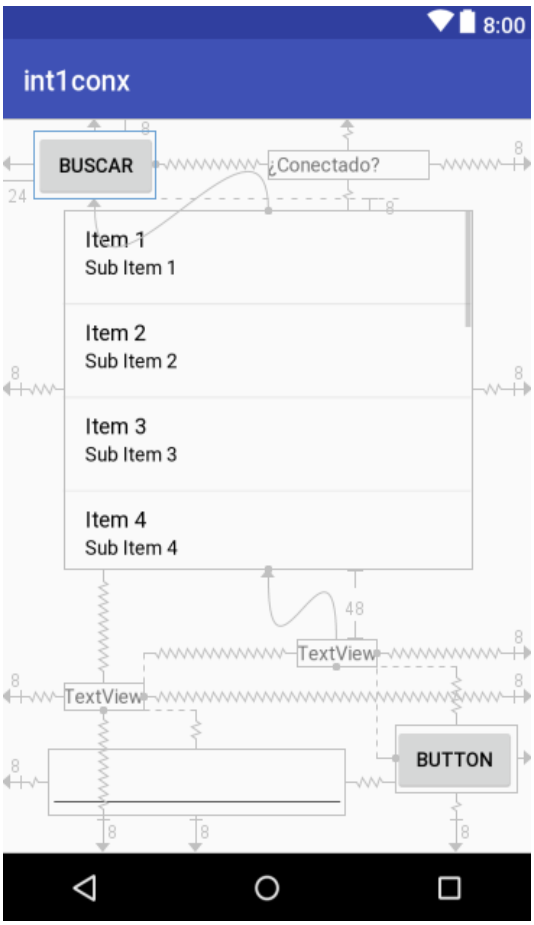
En esta interfaz podemos destacar los siguientes elementos:

Button: permiten al usuario comunicarse con la interfaz.

TextView: son capaces de mostrar texto en la interfaz, no permiten su edición.

ListView: utilizada para mostrar una lista de parámetros, si se programa puede ser interactiva con el usuario.

EditText: utilizado para la recepción por parte de la aplicación de información procedente del usuario.




8.Interfaz proyecto inicial

### Funcionamiento

Esta app está compuesta por una activity principal y su correspondiente layout, además incorporará una clase llamada Bluetooth donde se gestionará todo el proceso de conexión.

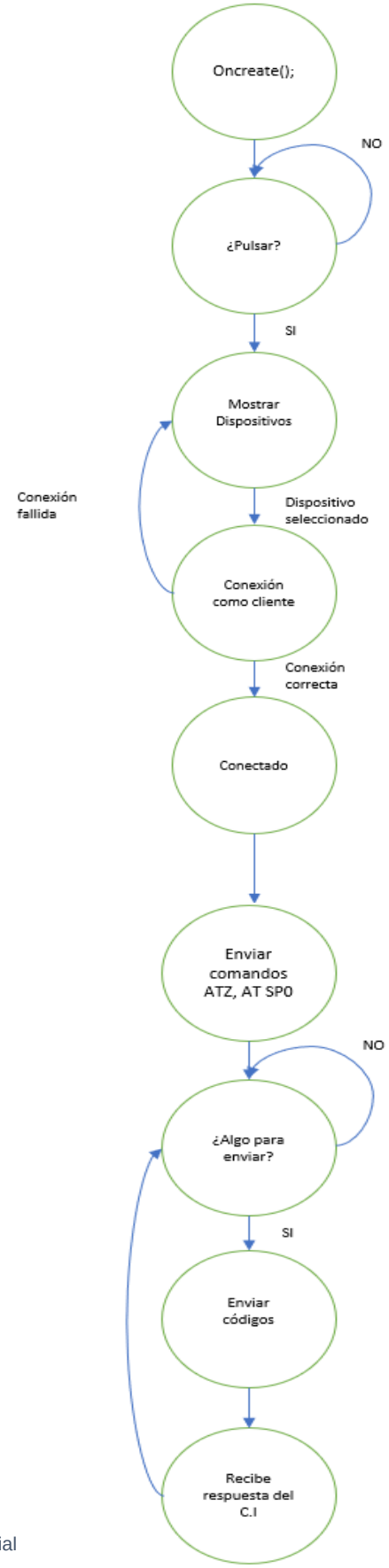
Al pulsar “BUSCAR” se despliega una listview al principio no visible para el usuario, la cual programaremos con la función Onclicklistener(); para hacer clickeable y poder seleccionar así el dispositivo con el cual queremos iniciar la conexión.

Una vez iniciada la conexión, la app enviará al chip los parámetros ATZ y AT SP 0 los cuales sirven para resetear el mismo e imponer el protocolo de forma automática respectivamente.

Una vez recibido la primera respuesta del ELM327 ya podemos comunicarnos con el vehículo.

Los datos serán enviados y recibidos siempre en bytes, modificando posteriormente este primer diseño con dos clases intermediadoras que se encargarán de darnos el número de bytes que debemos de leer y de acondicionar la señal respectivamente.

## Diagrama de bloques




9.Diagrama flujo proyecto inicial

## 2.3 Proyecto final

Tras las pruebas con el primer proyecto de prueba, se comenzó a plantear como sería la interfaz y el funcionamiento de la app definitiva.

Según lo requerido la app deberá de atender a unos parámetros, aunque estos deben ser seleccionables por el usuario.

Por lo que nuestra app tendrá en un principio unas 3 activities, la activity principal la cual se utilizará para la gestión de la conexión, una activity para la modificación de los parámetros y una tercera activity donde se mostrarán los parámetros seleccionados.

Permitiremos al usuario acceder a la activity de modificación de los parámetros desde la activity de visualización de los mismos, haciendo así más fácil y rápida su modificación.

También crearemos una cuarta y última activity la cual servirá para ayudar al usuario en caso de cualquier duda acerca la aplicación.

### 2.3.1 Diseño elegido

Primero vamos a ver el diseño de la app general, luego le echaremos un vistazo al diseño de cada activity por separado.

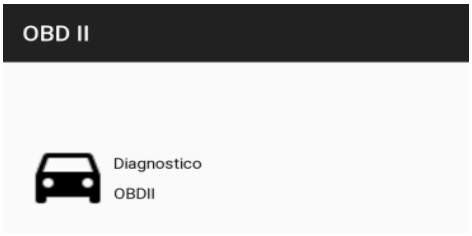
Para el diseño global se ha pensado en crear dos menús, uno desplegable en la zona izquierda superior el cual nos dará acceso a la activity de visualización de los parámetros, este menú estará disponible en la activity principal y en la activity de visualización.

El sentido de mostrar este menú desplegable en la activity de visualización, es la posible ampliación de las funciones de la app.

Para acceder a las activities de modificación de los parámetros y de ayuda se ha creado un submenú en la parte superior derecha, este menú incorpora tanto el acceso a las dos activities ya nombradas como una opción a cerrar la app.

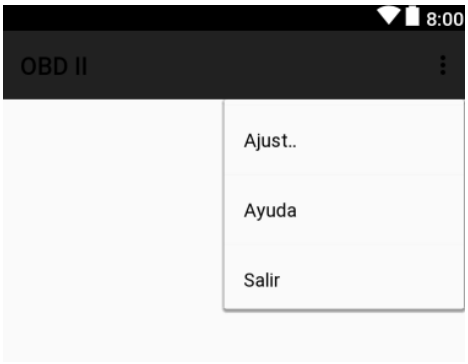
Este submenú está presente en todas las activities menos en la de ayuda al usuario, no se ha visto útil su incorporación en esta activity.

Para el diseño del primer menú despegable también se ha incorporado una cabecera.



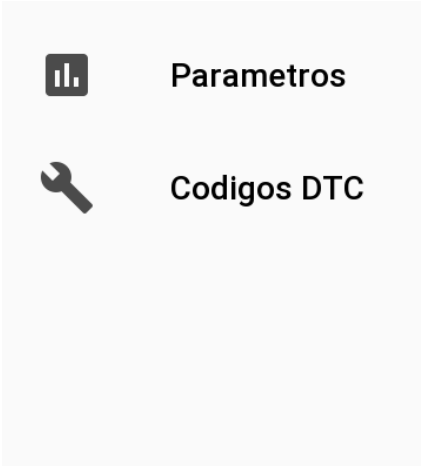

10.Menú desplegable

El diseño del submenú sería:




11.Submenú

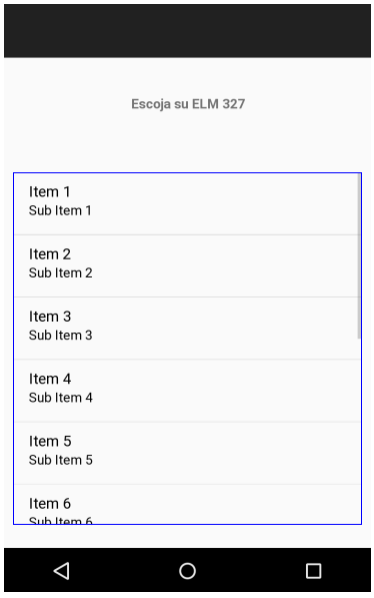
Y el diseño del menú desplegable:




12.Diseño menú desplegable

Ahora vamos a mostrar como quedarían los diseños de las tres activities nombradas anteriormente:

Esta es la **activity principal** y este sería su estado inicial, se podría decir que esta activity principal tiene tres estados.




El estado inicial: el consumidor inicia la app y esta da a elegir entre todos los dispositivos sincronizados.

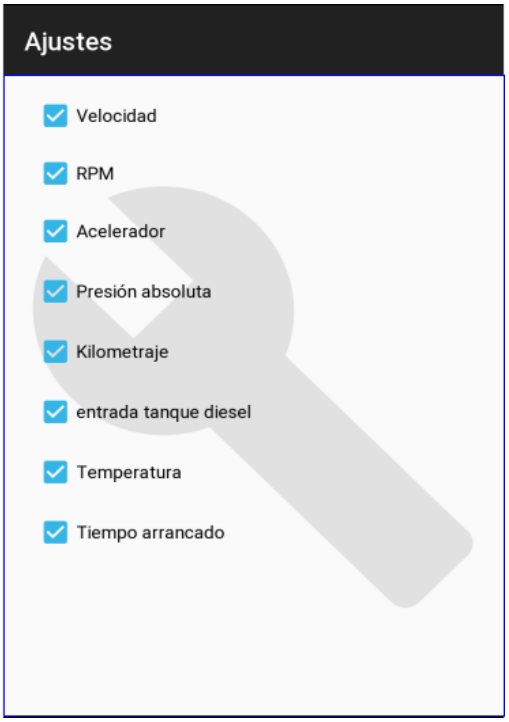
13.Estado inicial Mainactivity


14.Segundo estado Mainactivity

Segundo estado: el consumidor selecciona el dispositivo ELM327, se cambia el textview a “conectando” y se oculta la lista de dispositivos dando paso a una progessbar.


15.Ultimo estado Mainactivity

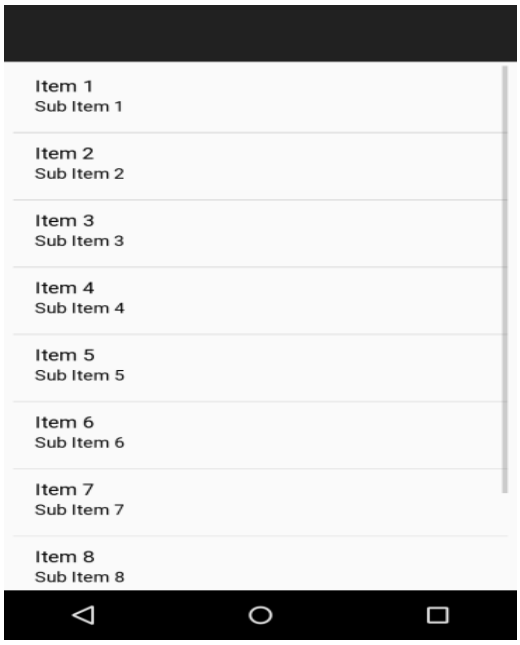
En su tercer y último estado, la conexión se ha realizado con éxito, en el textview se muestra el estado de conectado, y se oculta la progressbar dejando una imageview de tu tick, que confirma que todo está en perfecto estado




16.Diseño activity ajustes

Como podemos observar la activity ajustes incorpora ya unos Checkboxes, con los parámetros elegidos, que han sido configurados para que puedan ser clickeados por el usuario.

También incorpora una imagen de background (fondo), para tener un diseño más llamativo.

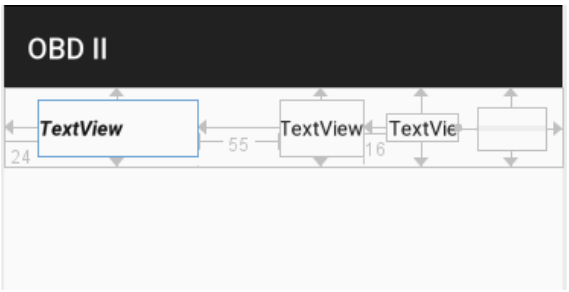



*17.Activity parametros*

Esta vista es la correspondiente a la activity de visualización de los parámetros, y su estado final no sería este ya que, aunque incorpora un listview, se ha modificado el diseño de esta listview, a un diseño el cual se adapta mejor a nuestro requisitos.

Este sería el diseño de cada parámetro, se puede observar que incorpora un primer textview el cual destaca su texto en negrita, en este textview se mostrara el nombre del parámetro a mostrar.

Después vendrá un siguiente textview el cual se utilizará para mostrar el valor recibido por el C.I, en el tercer y último textview se mostrará las unidades de medida del parámetro recibido.



18.Diseño listview personalizada

El último elemento es una progessbar, widget utilizado para ver el progreso de la variable, es el elemento más visual de esta listview y tendrá un máximo impuesto por el programador. Por ejemplo, en la velocidad se le ha impuesto un máximo de 200 km/h.

### 2.3.2 Código

Vamos a introducir las clases y las partes más interesantes de código, para ellos comenzaremos introduciendo las clases no asociadas a ninguna activity, estas clases son:

- **Bluetooth.java:** clase utilizada para la gestión de la conexión bluetooth, es prácticamente igual a la clase creada en la primera app, destacar su función ya que es seguramente la clase más importante del proyecto.
- **Param.java:** clase utilizada para gestionar la lista de parámetros, desde ella se inicializa el vector correspondiente a los parámetros, siempre y cuando es necesario. Sirve prácticamente como apoyo para las activities.
- **Frutasverduras.java:** clase creada para la creación de la listview personalizada.
- **Frutasverdurasadapter.java:** configura el adapter que toda listview necesita, de ello que use la extensión arrayadapter.

En cuanto a **Bluetooth.java** vamos a destacar algunas funciones importantes en la conexión bluetooth.

```
private class ClientClass extends Thread {
    private BluetoothSocket socket;
    private BluetoothDevice device;

    public ClientClass(BluetoothDevice device1) {
        device = device1;
        BluetoothSocket tmp = null;
        try {
            tmp = device.createRfcommSocketToServiceRecord(MY_uuid);
        } catch (IOException e) {
            error();
        }
        socket = tmp;
    }

    public void run() {

        try {
            socket.connect();

        } catch (IOException connectException) {

            try {
                socket.close();


            } catch (IOException closeException) {
                error();
            }
            return;
        }
        synchronized (this) {
            connectThread = null;

        }

        connected(socket);
    }
    public void cancel() {
        try {
            socket.close();
        } catch (IOException e) { }
    }


}
```

19.Clase ClientClass

Esta clase es la utilizada para iniciar una conexión como cliente con el C.I, recibe el BluetoothDevice de la clase connect(), que a su vez recibe este de la activity principal, luego se utiliza el bluetoothDevice para iniciar un BluetoothSocket, cuando inicia este Bluetooth Socket utiliza la UUID del ELM327, ya nombrado anteriormente.

Para llamar esta clase, hay que llamar antes a la clase connect() que también pertenece a la clase Bluetooth, la clase connect() es la que llamamos desde el flujo de la activity principal y es la que inicia el proceso de conexión como cliente.

Ahora vamos a ver la clase que gestiona la conexión, la cual es llamada por la clase connected(), a la cual se accede cuando el servidor acepta el inicio de conexión con el cliente.

```
private class ConnectedThread extends Thread{
   private final InputStream inputStream;
   private final OutputStream outputStream;
   private final BluetoothSocket bluetoothSocket;

    public ConnectedThread(BluetoothSocket bluetoothSocket1) {
        InputStream tmpint=null;
        OutputStream tmpout=null;
        bluetoothSocket=bluetoothSocket1;

        try {
            tmpint=bluetoothSocket.getInputStream();
            tmpout=bluetoothSocket.getOutputStream();
        }
        catch (IOException e){
            error();
        }
        inputStream=tmpint;
        outputStream=tmpout;
}
public void run() {
    byte[] buffer = new byte[1024];
    int bytes=0;

    while (true) {
        try {
            inputStream.read(buffer);
            bytes=read1(buffer);



            mHandler.obtainMessage(MainActivity.MENSAJE_LEIDO,bytes, arg2: -1,buffer).sendToTarget();

        } catch (IOException e) {
            break;
        }

    }
}
   public void write(byte[] bytes) {
         try {
             outputStream.write(bytes);
         } catch (IOException e) { }
    }
    public void cancel() {
         try {
             bluetoothSocket.close();
         } catch (IOException e) { }
    }
    private int read1(byte[] buffer) throws IOException {
         int bytes = 0;
         boolean escape = false;
         boolean prompt = false;
         byte[] readBuf;
         readBuf = new byte[1];
         while (!escape)
         {
             bytes += inputStream.read(readBuf);
             String s = new String(readBuf);
             if (!TextUtils.isEmpty(s))
             {
                 buffer[bytes] = readBuf[0];
                 prompt = s.contains(">");
             }

             escape = prompt;
         }


         return bytes;
    }

}
```

20.Clase Connected Thread

De esta clase de gestión de la conexión cabe destacar la incorporación de la clase privada read1 la cual nos permite saber la cantidad de bytes debemos leer del buffer que recibimos.

En esta clase aparece por primera ver el Handler, el cual es el principal intermediador entre la clase bluetooth y la activity principal. Donde vemos Mainactivity.MENSAJE_LEIDO se traduce a que en la activity principal siempre que llegue un mensaje recibirá un msg.what equivalente a MENSAJE_LEIDO.

En la clase Bluetooth.java también utilizamos el Handler para saber cuándo estamos cambiando el estado de la conexión. Tendrá la misma estructura solo que al llamarlo utilizaremos Mainactivity.ESTADO_CAMBIANDO.

```
public void Setestado(int estado1){
    ESTADO=estado1;
    mHandler.obtainMessage(MainActivity.ESTADO_CAMBIANDO,ESTADO, arg2: -1).sendToTarget();
}
```

21.Clase Setestado

Los argumentos tanto el uno como el dos, solo pueden tener valor integer, mientras que el objeto puede tener cualquier valor, si observamos el código utilizamos el objeto para enviar el buffer, mientras que los bytes a leer los enviamos a través del argumento uno.

Ahora vamos a repasar la función **param.java**:

Esta es una sencilla clase, la cual tiene dentro tres clases más,

Iniarray(): es llamada desde el oncreate() de la actividad principal, sirve para inicializar todos los checkboxes activos, lo que permite ver todos los parámetros en una primera estancia.

Escuchar(): se utiliza para actualizar el array de integers a[ ], el cual será referencia para todas las activities que necesiten conocer cuántos son los checkboxes activos o no activos.

Actualizar(): utilizado por las activities para conocer el estado del array a[]. Devuelve un array de integers.

```
public class param {
    Context context;
    private static int[] a=new int[8];


    private param() {
        context = null;

    }

    public param(Context context1) {
        context = context1;

    }

    public void iniarray(){
        for (int i=0;i<=7;i++){
            a[i]=1;
        }
    }

    public int [] actualizar(){

        return a;
    }

    public void escuchar (int [] c)
    {

        a=c;
    }

}
```

22.Clase param

Ahora toca hablar de las clases encargadas de crear la listview personalizada:

**Frutasverduras.java**

```
public class frutasverduras {
  public String valor;
  public String parametro;
  public String unidad;
  public Integer maxbar;
  public frutasverduras(){

      super();
  }
public frutasverduras(String parametro, String valor,String unidad,int maxbar){
    super();
    this.valor=valor;
    this.parametro=parametro;
    this.unidad=unidad;
    this.maxbar=maxbar;


}

}
```

23.Clase frutasverduras

Esta primera clase es la necesaria para pedir al usuario los datos necesarios para rellenar el layout correspondiente a nuestra listview editada, como podemos vimos anteriormente nuestra nueva listview editada contiene tres textview y una progressbar, por eso esta clase pide al usuario tres datos de tipo String y uno de tipo Integer que nos proporcionara el máximo de cada progressbar.

Ahora vamos con el **frutasverdurasadapter.java:**

Cuando creamos una listview con un diseño ya prediseñado y la queremos rellenar con unos datos determinados, nos bastaría con crear una nueva variable de tipo ArrayAdapter. Pero en nuestro caso que queremos una listview personalizada necesitamos un adapter propio el cual vamos a implementar en esta clase.

```
public class frutasverdurasadapter extends ArrayAdapter {
Context mycontext;
int mylayoutResource;
 frutasverduras [] mydata= null;

 public frutasverdurasadapter(Context context, int layoutresource, frutasverduras data []){
     super(context,layoutresource,data);
     this.mycontext=context;
     this.mylayoutResource=layoutresource;
     this.mydata=data;
 }
 public View getView(int position, View convertView, ViewGroup parent){

View row= convertView;
frutasverdurasHolder holder =null;

if (row==null){
    LayoutInflater inflater =((Activity)mycontext).getLayoutInflater();
        row=inflater.inflate(mylayoutResource,parent, attachToRoot: false);
        holder= new frutasverdurasHolder();
        holder.textView2=row.findViewById(R.id.textView2);
        holder.textView=row.findViewById(R.id.textView);
        holder.textView3=row.findViewById(R.id.tv);
        holder.progressBar=row.findViewById(R.id.progressBar);

        row.setTag(holder);

}else {
    holder=(frutasverdurasHolder)row.getTag();
}


 frutasverduras frutasverduras= mydata[position];
 holder.textView.setText(frutasverduras.valor);
 holder.textView2.setText(frutasverduras.parametro);
 holder.textView3.setText(frutasverduras.unidad);
 int a=Integer.parseInt(frutasverduras.valor);
 holder.progressBar.setProgress((a*100)/frutasverduras.maxbar);

 return row;
  }


 static class frutasverdurasHolder{
       TextView textView2;
       TextView textView;
       TextView textView3;
       ProgressBar progressBar;
 }

 }
```

24.Clase frutasverdurasadapter

Cuando el row es nulo, asociamos con el setTag() cada variable del frutasverdurasHolder() con cada view de la listview editada, si por el contrario ya hay algo cargado en el row, únicamente obtenemos el Tag con getTag(). Una vez cargado el holder, asociamos cada variable del holder con las variable de la clase frutasverduras(), el paso intermedio de crear el frutasverdurasHolder() es necesario para evitar la pérdida de datos y garantizar el correcto funcionamiento de este adapter. Como último paso retornamos el row y ya tenemos creado nuestro adapter modificado.

Ahora vamos a repasar algunas de las clases más importantes de las activities presentes en nuestras apps:

La clase Checkbox() es llamada en el onResume() de la activity dedicada a los ajustes, esta clase se dedica a actualizar el valor del array de integers equivalentes a cada parámetro, el estado inicial de los checkboxes es marcado, cuando el usuario pulsa alguno de ellos pasa a estar no marcado, con esta función pasamos el valor de su equivalente en el array a 0.

```
public void checkbox() {

    velocidad.setOnClickListener((view) → {
            velocidad.setChecked(velocidad.isChecked());
             if (velocidad.isChecked()) {
                b[0]=1;

             } else{
                b[0]=0;


             }



    });
```

25.Clase checbox

En caso de que estuviera ya a 0, se pasaría a 1, cambiando así el checkbox ha marcado de nuevo.

Al final de la función checkbox() se llama a la subclase escuchar perteneciente a la clase param.java para actualizar su array.

Como vimos en la estructura de una activity, durante el ciclo de vida de una activity no se produce ningún ciclo cíclico de programación, es decir cuando la activity termina el ultimo onResume(), ya pararía de correr código y solo actuaría ante la interacción del usuario.

Esto es un problema, ya que en la activity de muestra de los parámetros necesitamos que se vayan actualizando cada cierto tiempo.

Para solucionar este problema usamos este temporizador creado

```
private Runnable mrunnable =new Runnable() {
    @Override
    public void run() {

        principal();

        mhandler.postDelayed( r: this, delayMillis: 1000);

    }
};
public void repetir(){
    mrunnable.run();

}
public void pararrepetir(){

    mhandler.removeCallbacks(mrunnable);
}
```

26.Programación del temporizador

con un handler, la llamada al temporizador se realiza en cada onResume() y según lo programado permite que cada un segundo se ejecute la clase principal,

La cual utiliza los datos recogidos por la activity principal( que está en continua actualización) y llama a frutasverduras para cargar los datos en nuestra listview personalizada.

### Código activity principal

Ahora vamos a con las clases utilizadas por la activity principal, para el acondicionamiento del bucle, y para la obtención de los bytes útiles proporcionados por el ELM327.

La clase limpiarbucle(), retorna una variable de tipo string, esta variable tendrá un contenido similar a la variable de entrada. Tras su paso por la clase solo se eliminarán los huecos en blanco con string.trim, y se eliminarán posibles retornos de carro y el “>” característico que nos indica el final bucle recibido.

```
private String limpiarbucle (String text){
    text = text.trim();
    text = text.replace( target: "\t",  replacement: "");
    text = text.replace( target: " ",  replacement: "");
    text = text.replace( target: ">",  replacement: "");

    return text;
}
```

27.Clase limpiarbucle

Vamos a echar un vistazo a como se pediría y también como se trataría por ejemplo la respuesta del C.I ante la petición de conocer la velocidad:

Lo primero que tendríamos que tener en cuenta es que deberíamos de estar en modo conectado, una vez aquí enviaríamos los bytes 010D, el 01 indica que está en el primer PID y el 0D que queremos conocer la velocidad en este instante.

```
if (bluetooh.Estado() == 3) {
    if (sigue == 0) {
        String x = "010D";
        x += "\r";
        String a = x;
        byte[] bytecmd = a.getBytes();
        bluetooh.write(bytecmd);
    }
```

28.Ejmplo mandar código

Para que el chip lea nuestra orden tendríamos que añadir un retorno de carro a los bytes anteriores, el retorno de carro “\r” se añadiría a los bytes que queremos enviar como cadena de texto, y posteriormente pasar todo a bytes para enviarlo al ELM.

```
case MENSAJE_LEIDO:
    byte[] leerbuffer=(byte []) msg.obj;
    String readMessage= new String(leerbuffer, offset: 0,msg.arg1);
    readMessage=readMessage.trim();
    readMessage=readMessage.toUpperCase();
    String readMessage1= readMessage.toString();
    readMessage1=limpiarbucle(readMessage1);
```

29.Caso mensaje_leído

De esta forma cada vez que se recibe un mensaje correctamente, se crea una cadena de texto la cual tendrá como tamaño el argumento del handler, que como ya vimos

```
    if( readMessage1.contains("410D")){//velocidad
    String readMessage2=readMessage1.substring(readMessage1.indexOf("410D"));
    String MSB=readMessage2.substring(4,6);
    int A=Integer.valueOf(MSB, radix: 16);
    int C=A;
    velocidad=String.valueOf(C);
    sigue++;
    if (sigue==8){
        sigue=0;
    }
    mandarparametros();

}
```

30.Caso velocidad

anteriormente, será un entero con el número de bytes que tiene que leer.

Con el .trim eliminamos los espacios en blanco, y con el .toUpperCase conseguimos cambiar todo el string a mayúscula, el readmessage1 es solamente un apoyo para confirmar que a la clase limpiarbucle() llega un string.

Seguido a esto vendría varios condicionales, nos centramos en el referido a la velocidad:

Lo primero que destacaría es la respuesta del ELM, el C.I cambia el valor “0” de la orden “010D” por un “4” para confirmar que todo va correctamente.

Lo siguiente que hará será quedarse solo con los bytes de respuesta, con el indexOf consigue eliminar cualquier carácter que haya delante del 410D, y con el posterior substring y conociendo los bytes de respuesta que tendría la velocidad en concreto, selecciona únicamente los bytes que contienen la información sobre la velocidad, eliminando en este caso “410D”.

Para pasar el valor de hexadecimal a decimal se utilizar el integer.valueOf(bytes,16).

Ahora ya tendríamos la respuesta de la velocidad, en algunos otros parámetros sobre todo donde la respuesta tendría un tamaño de dos bytes, habría que jugar con los bytes de mayor valor y los bytes de menor valor para llegar a conseguir el valor decimal del parámetro.También debemos de nombrar la clase troncal de la activity principal, esta provoca que la activity principal se dirija en un sentido u otro dependiendo de los datos recibidos por la clase anteriormente vista setestado() perteneciente a Bluetooth.java.

```
private final Handler handler=new Handler(){
    public void handleMessage(Message msg){
        switch (msg.what) {
            case ESTADO_CAMBIANDO:
                switch (msg.arg1){
                    case Bluetooh.ESTADO_ESCUCHAR:

                        break;
                    case Bluetooh.ESTADO_CONECTANDO:
                        textView.setText("CONECTANDO");
                        listView.setVisibility(View.INVISIBLE);
                        progressBar.setVisibility(View.VISIBLE);

                        break;
                    case Bluetooh.ESTADO_CONECTADO:
                        progressBar.setVisibility(View.INVISIBLE);
                        textView.setText("CONECTADO");
                        imageView.setVisibility(View.VISIBLE);
                        mandarparametros();
                        break;
                    case Bluetooh.MAL:
                        textView.setText("INTENTELO DE NUEVO");
                        imageView.setVisibility(View.INVISIBLE);
                        progressBar.setVisibility(View.INVISIBLE);
                        listView.setVisibility(View.VISIBLE);

                    default:
                        break;
                }
                break;
case MENSAJE_LEIDO:
    byte[] leerbuffer=(byte []) msg.obj;
    String readMessage= new String(leerbuffer, offset: 0,msg.arg1);
    readMessage=readMessage.trim();
    readMessage=readMessage.toUpperCase();
    String readMessage1= readMessage.toString();
    readMessage1=limpiarbucle(readMessage1);

    if (readMessage1.contains("4131")){// distancia recorrida
        String readMessage2=readMessage1.substring(readMessage1.indexOf("4131"));

        String MSB=readMessage2.substring(4,6);
        String LSB=readMessage2.substring(6,8);
        int A=Integer.valueOf(MSB, radix: 16);
        int B=Integer.valueOf(LSB, radix: 16);
        A=A*256;
        A+=B;
        kilometros= String.valueOf(A);
        sigue++;
        if (sigue==8){
            sigue=0;
        }
        mandarparametros();
```

```
if (readMessage1.contains("4146")){// temperatura ambiental
    String readMessage2=readMessage1.substring(readMessage1.indexOf("4146"));
    String MSB=readMessage2.substring(4,6);
    int A=Integer.valueOf(MSB, radix: 16);
    A=A-40;
    temperatura=String.valueOf(A);
    sigue++;
    if (sigue==8){
        sigue=0;
    }
    mandarparametros();


}

if( readMessage1.contains("410C")){//revoluciones
    String readMessage2=readMessage1.substring(readMessage1.indexOf("410C"));
    String MSB=readMessage2.substring(4,6);
    String LSB=readMessage2.substring(6,8);
    int A=Integer.valueOf(MSB, radix: 16);
    int B=Integer.valueOf(LSB, radix: 16);
    int C=(256*A + B)/4;
    revoluciones=String.valueOf(C);
    sigue++;
    if (sigue==8){
        sigue=0;
    }
    mandarparametros();
```

```
 if( readMessage1.contains("4149")){//acelerador
     String readMessage2=readMessage1.substring(readMessage1.indexOf("4149"));
     String MSB=readMessage2.substring(4,6);
     int A=Integer.valueOf(MSB, radix: 16);
     int C=(A*100/255);
     Posaceler=String.valueOf(C);
     sigue++;
     if (sigue==8){
         sigue=0;
     }
     mandarparametros();

 }


 if( readMessage1.contains("4133")){//Presion absoluta
     String readMessage2=readMessage1.substring(readMessage1.indexOf("4133"));
     String MSB=readMessage2.substring(4,6);
     int A=Integer.valueOf(MSB, radix: 16);
     Presiónabs=String.valueOf(A);
     sigue++;
     if (sigue==8){
         sigue=0;
     }
     mandarparametros();

 }
```

```
if( readMessage1.contains("411F")){//Tiempo
    String readMessage2=readMessage1.substring(readMessage1.indexOf("411F"));
    String MSB=readMessage2.substring(4,6);
    String LSB=readMessage2.substring(6,8);
    int A=Integer.valueOf(MSB, radix: 16);
    int B=Integer.valueOf(LSB, radix: 16);
    int C=256*A+B;
    Tiempo=String.valueOf(C);
    sigue++;
    if (sigue==8){
        sigue=0;
    }
    mandarparametros();

}
if (readMessage1.contains("412F")){// entrada tanque
    String readMessage2=readMessage1.substring(readMessage1.indexOf("412F"));
    String MSB=readMessage2.substring(4,6);
    int A=Integer.valueOf(MSB, radix: 16);
    int C =((A*100/255));
    Taceite=String.valueOf(C);
    sigue++;
    if (sigue==8){
        sigue=0;
    }
    mandarparametros();


}



break;
```

31.Clase handlemessage

### Código layouts

Cada activity tiene un layout con su respectivo código asociado, a su vez estos layouts tienen incorporada la llamada a los menús los cuales queremos mostrar en cada una de estas activities.

En este apartado vamos a ver el código correspondiente a los layouts de los menús tanto desplegable como no desplegable, y el correspondiente a la cabecera que incorpora el menú desplegable.

Luego echaremos un vistazo a el código perteneciente al layout asociado a la activity principal, para ver cómo se realzaría la llamada a estos menús y a su correspondiente cabecera.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:showIn="navigation_view"
    >

    <group android:checkableBehavior="single">
        <item
            android:id="@+id/parametros"
            android:icon="@drawable/ic_assessment_black_24dp"
            android:title="Parametros" />
        <item
            android:id="@+id/DTC"
            android:icon="@drawable/ic_build_black_24dp"
            android:title="Codigos DTC" />


    </group>
```

32.Codigo layout menu desplegable

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/action_settings"
        android:checkable="false"
        android:orderInCategory="100"
        android:title="Ajustes" />
    <item
        android:id="@+id/inf"
        android:orderInCategory="100"
        android:title="Ayuda"
        app:showAsAction="never" />
    <item
        android:id="@+id/salir"
        android:orderInCategory="100"
        android:title="@string/Salir"
        app:showAsAction="never" />
```

33.Código layout submenú

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="176dp"
    android:background="@color/grisoscuro"
    >

    <TextView
        android:id="@+id/lv1"
        android:layout_width="84dp"
        android:layout_height="28dp"
        android:layout_marginBottom="8dp"
        android:layout_marginTop="8dp"
        android:text="Diagnostico"
        android:textColor="@color/negro"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toEndOf="@+id/imageView2"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.59" />

    <TextView
        android:id="@+id/lv2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginTop="8dp"
        android:text="OBDII"
        android:textColor="@color/negro"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toEndOf="@+id/imageView2"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.751" />

    <ImageView
        android:id="@+id/imageView2"
        android:layout_width="75dp"
        android:layout_height="71dp"
        android:layout_marginBottom="8dp"

        android:layout_marginStart="16dp"
        android:layout_marginLeft="16dp"
        android:layout_marginTop="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.707"
        app:srcCompat="@drawable/ic_directions_car_black_24dp" />


</android.support.constraint.ConstraintLayout>
```

34.Código layout cabecera

Ahora vamos a echar un vistazo a cómo llamar desde el layout de una activity a estos layouts complementarios. Observando el layout de la activity principal podemos ver como se realiza la llamada a el menú desplegable y a la cabecera, en este layout también se incorpora el layout app_bar_main_main.

```
<?xml version="1.0" encoding="utf-8"?>


<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/a
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">


    <include
        layout="@layout/app_bar_main_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/mi_cabezera"
        app:menu="@menu/activity_main_drawer" />


</android.support.v4.widget.DrawerLayout>
```

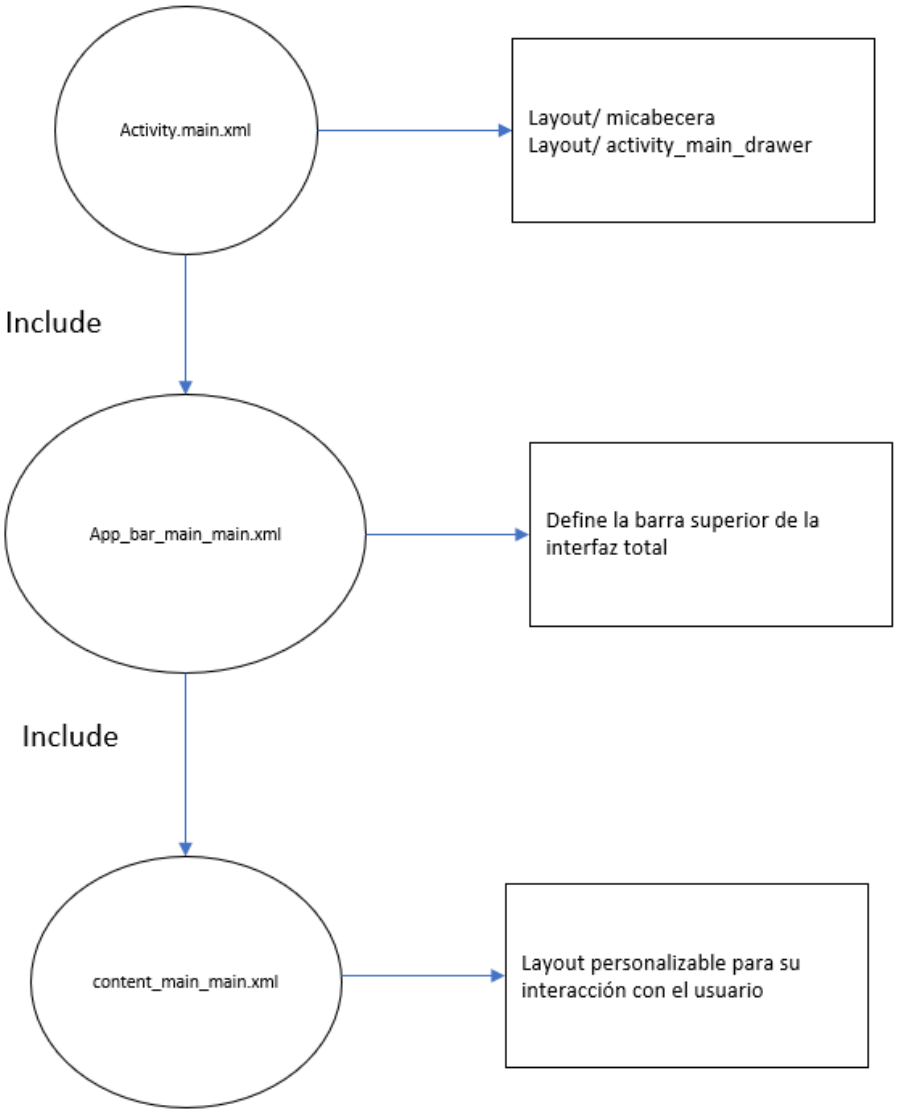
35.Código layout acitivity principal

En este layout es el encargado de definir la barra superior de la pantalla la cual percibe el usuario y a su vez es el encargado de incorporar el layout content_main_main, que finalmente es el layout que podemos editar para que interactúe con el usuario.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.jesus.menuprincipal.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main_main" />


</android.support.design.widget.CoordinatorLayout>
```

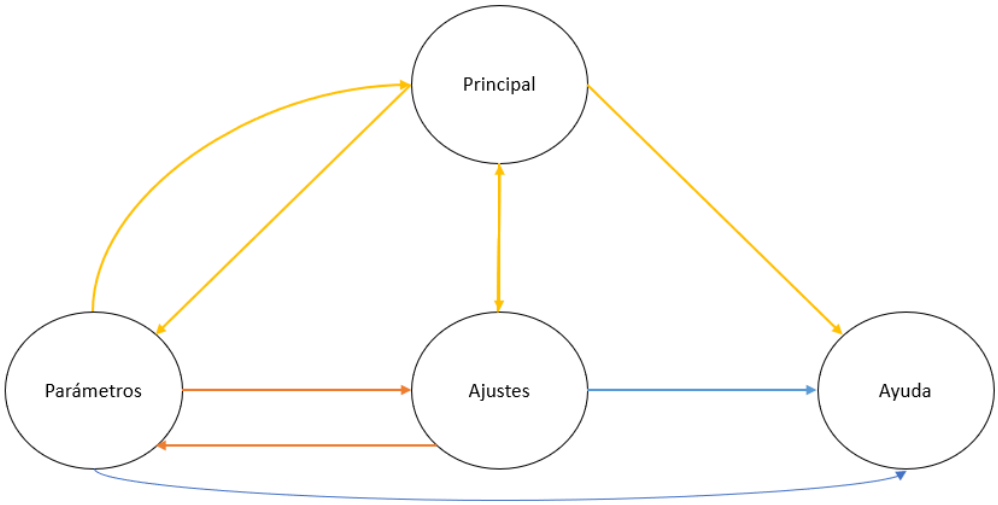
36.Código layout app_bar_main_main

Vamos a ver un pequeño diagrama de cómo quedaría la formación del layout asociado a la **MainActivity.java**




37.Diagrama flujo layouts activity principal

### 2.3.3 Diagrama de bloques de activities




38.Diagrama de flujo de activites

Principal→ Main.activity

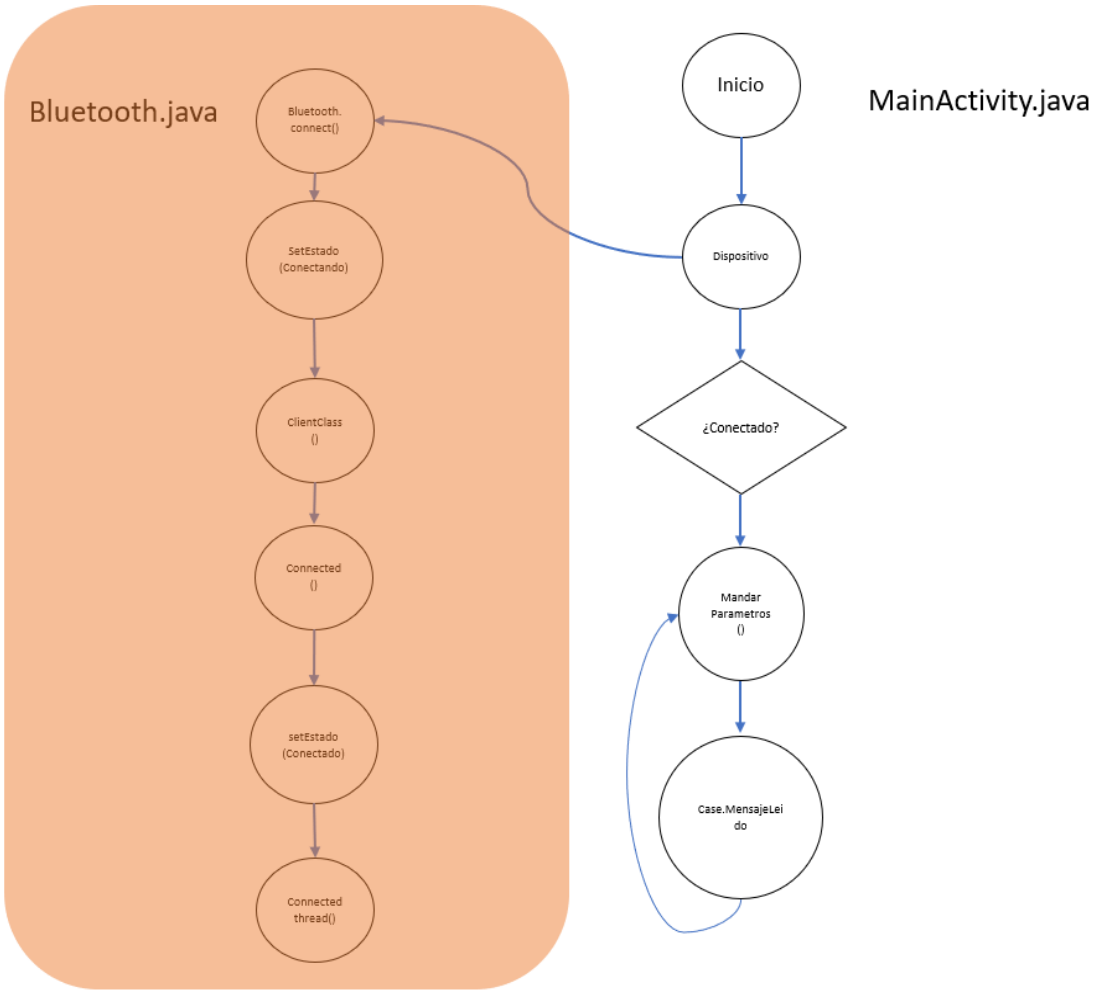
Parámetros→ Main2para

Ajustes→Main4Ajustes

Ayuda→ Main5Ayuda

### 2.3.4 Diagrama de bloques código

En este apartado vamos a ver cómo sería el funcionamiento de la aplicación, el siguiente diagrama de bloques sería un diagrama de bloques muy resumido de las principales clases que marcan el ritmo principal de nuestra app.



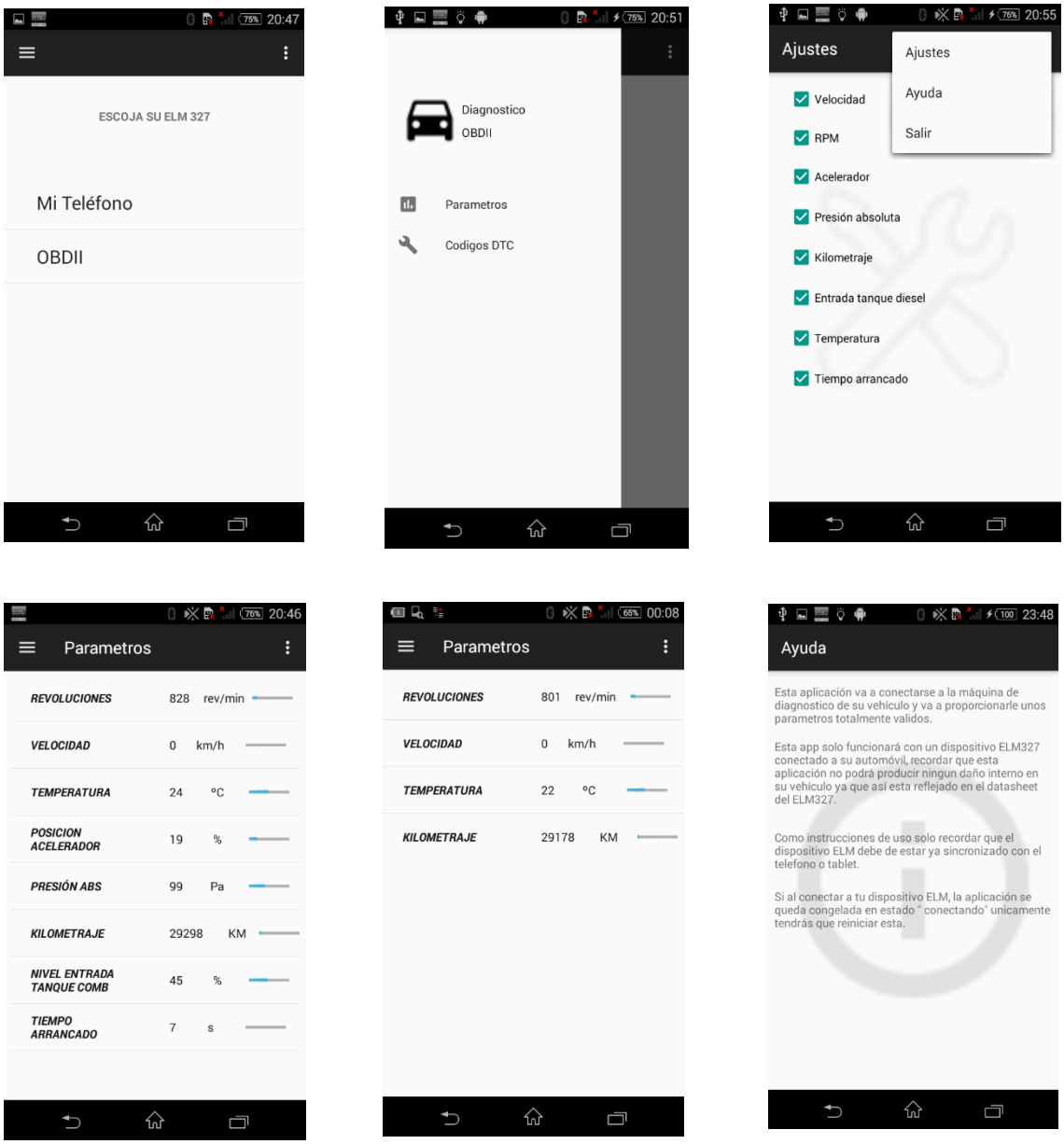

39.Diagrama de flujo del código troncal

## 2.4 Resultados

El resultado a todo este proyecto es una aplicación compatible con todos los dispositivos Android que posean una API 14 o posterior(prácticamente el 100% de los dispositivos Android que están en funcionamiento) que es capaz de conseguir recibir datos de la ECU del vehículo a través de un ELM327.

La aplicación mostrará los datos actualizados al segundo, y tras probarla en varias ocasiones no hemos observado ningún error que detenga el flujo de información o cierre la aplicación.

A continuación vamos a ver algunas capturas de pantalla de la aplicación en funcionamiento, y de cómo quedaría su interfaz finalmente.




40.Resultado final app

## 2.5 Posibles ampliaciones

Como se puede observar en las capturas de pantalla de la aplicación, en el menú desplegable hay una pestaña llamada códigos DTC, la implementación de esta pestaña sería la primera modificación sobre este proyecto.

Los códigos DTC están reflejados en el modo tres de los modos de funcionamiento de la máquina de diagnóstico, se debería de realizar algunas pruebas con coches averiados para comprobar cómo se recibirían los códigos DTC, y a partir de ahí acondicionar los bytes que llegarían a nuestra app para mostrar solamente el código de falla.

Otra posible ampliación seria incorporar en esta misma sección del menú por ejemplo, un botón que llamaría al modo cuatro de los modos Pid , el cual serviría para eliminar todos los códigos DTC.

Esta aplicación esta realizada sobre una temática interesante y muy amplia por lo que a parte de estas dos ampliaciones esenciales, habría muchas más modificaciones interesantes.

# BIBLIOGRAFÍA

https://es.wikipedia.org/wiki/OBD (Información histórica acerca del sistema de diagnóstico a bordo)

https://www.elmelectronics.com/products/ics/obd/ (Página oficial de ELM Electronics)

https://developer.android.com/studio/intro/ (Página oficial de desarrolladores de Android)

https://es.wikipedia.org/wiki/Sistema_operativo_móvil#Android (Wikipedia página oficial)

https://es.wikipedia.org/wiki/OBD-II_PID (Wikipedia página oficial)

https://developer.android.com/guide/topics/connectivity/bluetooth(Página oficial de desarrolladores de Android)

https://developer.android.com/reference/android/bluetooth/BluetoothAdapter(Página oficial de desarrolladores de Android)

https://developer.android.com/reference/android/bluetooth/BluetoothServerSocket (Página oficial de desarrolladores de Android)

https://telekita.wordpress.com/2012/02/03/ciclo-de-vida-de-una-activity/ (Página oficial Telekita)