UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Especialidad de Computación

Desarrollo de aplicaciones Human-Machine Interface para motos eléctricas

Pablo Ruiz Martínez

Director: Daniel Gratacós Lincoln (Silence)
Ponente: María José Serna Iglesias (CS)

16 de enero de 2020

Resumen

El presente proyecto consiste en el diseño y la implementación del software de escritorio de la moto electrica S01 de la marca Silence. Este software interacciona con los distintos elementos de la moto y permite, por una parte, la actualización del *firmware* de los distintos elementos que conforman el hardware de la moto y, por otra parte, la monitorización de un conjunto de variables con el fin de facilitar al servicio oficial de mantenimiento la prevención y detección de averías.

Aunque existen programas estándares para interaccionar con el CAN Bus de vehículos, Silence estaba interesada en crear una interfaz a medida para que pudieran usarla todos los actores implicados en el diseño y fabricación de la moto, así como los servicios oficiales.

Aunque inicialmente se iba a desarrollar una única aplicación, basada en otra ya existente para otro modelo anterior escrita Java, se decidió dividir la aplicación en dos aplicaciones que llegaron a denominarse: *Monitor* y *Bootloader*. Asímismo se decidió migrar del lenguaje Java a Kotlin.

El programa *Monitor* debe mostrar una serie de información, valores a tiempo real y parámetros de la batería y de la ECU (dos elementos del hardware de la moto) de una forma que resulte útil para el operario que lo use, es decir, usando una interfaz gráfica y sencilla que oculte las dificultades del protocolo CAN.

El programa *Bootloader* sirve para actualizar el *firmware* de elemento de hardware seleccionado. Todo ello debe hacerse seleccionando la velocidad de transmisión adecuada y con las medidas de seguridad necesarias para abortar el proceso si se produce algún error o la moto no responde.

Abstract

The present project consists in the design and implementation of desktop software for the company Silence's S01 electric scooter. This software interacts with the different elements of the scooter and allows, on the one hand, the firmware update of the different elements that make up the hardware of the scooter and, on the other hand, the monitoring of a set of variables in order to facilitate the prevention and detection of failures to the official maintenance service.

Although there are standard programs to interact with CAN Bus, a protocol extensively used in vehicle manufacturing, Silence was interested in creating a custom interface so that all the actors involved in the design and manufacture of the motorcycle could use it, as well as the official services.

Although initially it a single application was going to be developed, based on the one that was being used for the previous scooter model and that was written in java, it was decided to divide the application into two applications that came to be called: *Monitor* and *Bootloader*. Likewise, it was decided to migrate from the Java language to Kotlin.

The *Monitor* program must show a series of information, real-time values and parameters from the battery and from the ECU (two hardware elements of the scooter) in a way that is useful for the operator who uses it, that is, by using a simple and graphical interface that conceals the difficulties of the CAN protocol.

The *Bootloader* program updates the firmware of a selected hardware element. All this must be done by selecting the appropriate transmission speed and with the necessary security measures to abort the process if an error occurs or the motorcycle does not respond.

Sumario

1. Introducción **2**

1.1. Contexto 2

1.2. Justificación 3

1.3. Metodología y rigor 4

2. Planificación temporal **5**

2.1. Definición de tareas 5

2.2. Estimaciones temporales iniciales 5

2.3. Planificación final 6

3. Presupuesto **8**

3.1. Identificación de costes 8

3.2. Estimación de costes 10

3.3. Control de gestión 11

4. La moto S01 **12**

5. Protocolos de comunicación **24**

5.1. CAN Bus 24

5.2. Kinetis ROM Bootloader 27

6. Análisis de requisitos **32**

6.1. Requisitos del programa *Monitor* 32

6.2. Requisitos del programa *Bootloader* 34

7. Diseño del software **35**

7.1. Diseño del programa *Monitor* 35

7.2. Diseño del programa *Bootloader* 41

8. Implementación **46**

8.1. Implementación del programa *Monitor* 46

8.2. Implementación del programa *Bootloader* 49

9. Vistas **54**

9.1. Vistas del programa *Monitor* 54

9.2. Vista del prograna *Bootloader* 58

10. Informe de sostenibilidad **59**

10.1. Dimensión ambiental 59

10.2. Dimensión económica 61

10.3. Dimensión social 61

10.4. Vida útil y riesgos 62

11. Conclusiones y trabajo futuro **63**

12. Bibliografía **64**

Glosario

- **Aplicaciones Human-Machine Interface (HMI):** aplicaciones focalizadas en la interacción con el usuario, es decir, en cómo se percibe la información por pantalla y en cómo se aporta información al sistema.
- **Sistema *embedded*:** del inglés *Embedded System* (incrustado). Es un controlador con una función dedicada en un sistema mecánico o eléctrico, normalmente se trata de un sistema de computación a tiempo real, es decir, que es un sistema informático que interacciona con su entorno físico y responde a unos estímulos en un periodo de tiempo determinado.
- **ECU:** acrónimo de *Electronic Control Unit* (Unidad de control electrónico). Es cualquier sistema *embedded* en el sistema electrónico de un automóvil que controla uno o más elementos electrónicos del vehículo. En terminología de Silence, la empresa donde se realiza este proyecto, se refiere únicamente a un sistema *embedded* que controla los frenos, las luces, el acelerador, el caballete y la apertura del baúl, pero no el que controla la batería (BMS) ni el de la conectividad (Astra).
- **BMS:** acrónimo de *Battery Management System* (Sistema de gestión de batería). Sistema *embedded* que controla la batería de la moto.
- **Astra:** sistema *embedded* que controla la conectividad de la moto con internet y bluetooth.
- **CAN Bus:** acrónimo *Controller Area Network Bus* (Bus de red de área de controladores). Es un protocolo de comunicaciones desarrollado por la firma alemana Robert Bosch GmbH, basado en una topología bus para la transmisión de mensajes en entornos distribuidos. Es usado comúnmente para la comunicación entre los distintos procesadores de los vehículos.
- **Ixxat:** empresa que comercializa componentes de CAN para desarrollo y producción. En adelante en este documento, nos referiremos como Ixxat al conversor *USB-to-CAN* desarrollado por esta empresa.
- **Kotlin:** lenguaje de programación de tipado estático que corre sobre la máquina virtual de Java. Es el lenguaje en el que se ha programado este proyecto.
- **Flag:** información de 1 solo bit, es decir, que solo puede valer «cierto» o «falso». En la comunicación de la moto mediante CAN se utiliza para indicar, por ejemplo, si las luces están encendidas o apagadas, si se está accionando un freno o si hay algún error o advertencia.
- **Maestro/esclavo:** modelo de comunicación en el que un proceso o dispositivo (maestro) controla a otro proceso o dispositivo (esclavo).
- **Bootloader:** proceso que ejecutan los ordenadores al arrancar que carga el software a ejecutar en la memoria principal de la CPU. En los microcontroladores Kinetis empleados en los sistemas *embedded* de la moto, este proceso se encuentra en una ROM y permite la interacción mediante CAN Bus con un periférico para poder cargar una imagen del *firmware*.
- **Ciclado de batería:** proceso mediante el cual se descarga completamente y se vuelve a cargar del todo una batería para probar su buen estado.

1. Introducción

1.1. Contexto

En este proyecto se han desarrollado dos aplicaciones para la empresa de motos eléctricas Silence. Esta empresa lleva desde el año 2011 diseñando y fabricando motos eléctricas. Uno de los aspectos de los que se enorgullecen es que este diseño y esta fabricación son propios. Esto se extiende al software que utilizan.

Uno de los tipos de software que se emplea a lo largo de la vida de estas motos son los programas de escritorio que actualizan el software interno de los componentes y monitorizan y parametrizan el vehículo desde un ordenador. Para llevarlo a cabo para el último modelo de moto llamado S01, desde Silence se publicó una oferta de prácticas en la bolsa de trabajo de la FIB, la cual yo solicité. Tras dos entrevistas, me convertí en el desarrollador de este software y estas aplicaciones de escritorio objeto de mi Trabajo de Fin de Grado.

Los desarrolladores de los sistemas *embedded* de los distintos componentes de la moto, los operarios de fábrica, los responsables del departamento de Calidad y los talleres oficiales necesitan conocer el estado de dichos sistemas internos (como temperaturas, voltajes, estado de la carga de la batería...), recolectar historiales de datos y eventos (por ejemplo, para identificar el origen de una avería), configurar parámetros de la moto y actualizar el software interno. Con estas aplicaciones se pretende conseguir la máxima sencillez posible en la realización de estas tareas, con una interacción con el programa lo más intuitiva posible y que no requiera conocimientos del protocolo de comunicación que hay detrás, CAN Bus, empleado extensamente en la industria automovilística.

Las funcionalidades de estas aplicaciones de escritorio responden a las necesidades de los actores que las usarán:

- Los desarrolladores de los sistemas *embedded* prueban sus sistemas.
- Los operarios de fábrica comprueban que el montaje ha sido correcto, que no hay piezas defectuosas y configuran la moto.
- Los responsables del departamento de Calidad obtienen los historiales de los datos de funcionamiento de la moto para verificar que cumple con los estándares de calidad.
- Los talleres oficiales pueden obtener el historial de eventos para detectar el origen de una avería o configurar la moto con unos parámetros distintos.

Como estudiante que finaliza el grado en Ingeniería Informática, el desarrollo de estas aplicaciones me ha servido para poner a prueba los conocimientos obtenidos a lo largo de mis estudios, así como para profundizar en un caso real concreto.

1.2. Justificación

Aunque ya existen programas para comunicarse mediante CAN Bus, para usarlos es necesario un conocimiento extenso del protocolo. Las aplicaciones que se requieren en Silence deben ser sumamente específicas, con unas funcionalidades muy concretas que además pueden ser distintas dependiendo del actor que use la aplicación. También se requiere una interfaz muy visual e intuitiva que permita un uso sencillo y rápido. Es por ello por lo que no era conveniente usar una aplicación ya existente.

Pero ¿era posible adaptar una aplicación anterior? Las aplicaciones que Silence ya usaba para su otro modelo, el S02, desarrolladas también por anteriores informáticos de la propia empresa, compartían muchas especificaciones con el nuevo software. No obstante, no era posible simplemente modificar el programa del S02 porque el protocolo de comunicación que se usaba era distinto. Además, se ha cambiado el lenguaje de programación de Java a Kotlin.

Java es un lenguaje de programación en el que es posible crear cualquier tipo de aplicación. También puede usarse prácticamente en cualquier tipo de máquina. Está en los sistemas operativos Windows, Linux y también en Android. Es totalmente gratuito y es uno de los más importantes en el mundo de la informática y programación. Por ello, cuenta con una gran comunidad de usuarios.

Su principal ventaja es que puede usarse desde el lado de cliente y también desde el back end. Es también un lenguaje independiente, por lo tanto, también se puede usar en cualquier ordenador en local.

En cuanto a sus desventajas, la principal es que no es un lenguaje muy moderno. Java 8 sí ha dado un paso importante en cuando a nuevas prestaciones, pero no es compatible con programación para Android que es una de las plataformas más extendidas hoy en día. Kotlin es un lenguaje de programación creado por JetBrains. Fue anunciado en la Google I/O como lenguaje oficial para programar aplicaciones Android, junto con C++ y el propio Java. Se trata de un lenguaje Open Source, que se encuentra bajo licencia Apache 2.0 y una de sus grandes ventajas es que se puede hacer llamadas a Java y viceversa. Es común encontrar aplicaciones escritas en los dos lenguajes: Java y Kotlin, y ambos pueden coexistir casi sin problemas.

Otra ventaja interesante es que se trata de un lenguaje más moderno. Por lo tanto ahora puede que no haya demasiada diferencia, pero poco a poco se irá viendo como Kotlin va incorporando nuevas posibilidades que no se encuentran en Java actualmente. Fundamentalmente Kotlin es mejor que Java en temas de seguridad, sintaxis, compatibilidad y programación funcional. En definitiva y a modo de resumen:

Ventajas de Kotlin frente a Java

- Permutabilidad Con Java

  Total interoperabilidad entre ambos lenguajes, teniendo perfectamente un proyecto con código de Kotlin y Java a la vez sin problemas de compilación.

- Curva de Aprendizaje suave

  Es un lenguaje de programación con una curva de aprendizaje suave que hace que programadores de Java se sientan cómodos con el lenguaje y la sintaxis les resulte familiar.

- Programación Funcional y Procedural

  Existen muchos paradigmas de programación, y cada uno tiene sus pros y sus contras. En el caso de Kotlin, éste combina las fortalezas del lenguaje funcional y a su vez las del lenguaje procedural.

- Código más conciso

  En comparación con el código en una clase de Java y otra de Kotlin, la de Kotlin usa un código más conciso, lo que mejora la legibilidad.

Por estos motivos se decidió desarrollar las aplicaciones a partir de cero, si bien basándose extensamente en aplicaciones usadas para el modelo anterior.

1.3. Metodología y rigor

Las primeras semanas del proyecto fueron dedicadas al estudio y análisis de las aplicaciones que funcionaban en los modelos anteriores. A su vez, hubo un periodo de aprendizaje del nuevo lenguaje de programación decidido así como de aspectos de la comunicación con la moto.

La siguiente fase fue la determinación de los requisitos de la aplicación. Para ello, se llevaron a cabo reuniones con los diferentes responsables de las partes implicadas en el uso de la aplicación, así como con el responsable de las anteriores aplicaciones.

Una vez recopilada toda la información sobre cómo tenía que funcionar el programa, se procedió al diseño, siempre riguroso con los métodos de diseño aprendidos a lo largo del Grado.

Con el diseño concebido, se empezó a programar. Primero se programó la estructura básica de la aplicación, tanto la lógica interna como la interfaz gráfica. Una vez acabado esto, se fueron añadiendo funcionalidades y testándolas a medida que surgían nuevas necesidades en las reuniones.

Ya se previó que a medida que avanzara el proyecto se irían modificando los requisitos y, por consiguiente, el diseño y el programa en sí. El cambio más significativo en este aspecto tuvo lugar a mediados de mayo de 2019, después de 5 meses de proyecto. En ese momento se decidió separar las funcionalidades en dos aplicaciones distintas y hubo que realizar una formación adicional sobre el protocolo que iba a emplearse en el segundo programa y diseñarlo, implementarlo y testarlo, todo ello en pleno proceso de implementación del programa principal.

Para seguir los avances del proyecto, se han realizado reuniones semanales con el equipo de I+D así como con el director del proyecto. También se empezó a utilizar Wrike, un software de seguimiento de proyectos para estar al día del estado en el que nos encontrábamos.

2. Planificación temporal

2.1. Definición de tareas

Se han definido las siguientes tareas:

- **Formación:**
  Los inicios fueron dedicados al conocimiento exhaustivo de las herramientas que se iban a utilizar: el lenguaje de programación (Kotlin) y el protocolo de comunicación (CanBUS); así como comprender el tipo de aplicación que se requiere estudiando las anteriores.

- **Análisis de requisitos:**
  Debido a la dinámica de este proyecto hubo que proceder con el diseño y la implementación antes de acabar el análisis de requisitos. Por ello, se dividió esta tarea en dos subtareas: análisis de requisitos inicial y análisis continuo de requisitos. La primera recogía toda la información que se tenía inicialmente sobre qué se quiere de la aplicación. La segunda ha consistido en recopilar los cambios y sugerencias que iban proponiendo los usuarios para implementarlos posteriormente.

- **Diseño del software:**
  Esta fase requería haber finalizado el análisis de requisitos inicial. Cuando se hubo definido qué debía hacer el programa (o al menos un subconjunto inicial de ello) se procedió al diseño de dicho programa. Para ello, se han empleado los métodos y lenguajes de modelización aprendidos en las asignaturas de Introducción a la Ingeniería del Software (IES) y de Proyectos de Programación (PROP). A medida que los requisitos fueron cambiando, el diseño se iba haciendo a la par. Por ello, igual que en la tarea anterior, hemos diferenciado dos subtareas: diseño inicial y modificaciones en función de nuevas especificaciones.

- **Implementación y testing:**
  Debido a la urgencia con la que se necesitaban estas aplicaciones, las tareas de programación y testeo del programa se fusionarion en una. Esto se debe a que se requería que el programa estuviese siempre a punto para liberar una versión provisional. Para ello, se estableció un orden de prioridad de las funcionalidades consensuado con los responsables y, una vez programada la estructura básica del programa, se fueron añadiendo y testando estas funcionalidades una a una, siguiendo dicho orden.

2.2. Estimaciones temporales iniciales

Antes de empezar el proyecto, se estimó el tiempo requerido para la realización de cada una de las tareas definidas. Para ello, se estableció una jornada diaria de 8 horas. Esas estimaciones fueron las siguientes:

- **Formación**
  - Aprendizaje y familiarización con Kotlin:
    5 días x 8 horas = **40 horas**
  - Aprendizaje del protocolo CanBUS y familiarización con aplicaciones anteriores:
    12 días x 8 horas = **96 horas**

- **Análisis de requisitos**
  - Análisis inicial de requisitos:
    7 días x 8 horas = **56 horas**
- **Diseño del software**
  - Diseño inicial (+ análisis continuo de requisitos):
    15 días x 8 horas = **120 horas**
- **Implementación y testing**
  - Implementación y testing (+ análisis continuo de requisitos + modificaciones del diseño):
    42 días x 8 horas = **336 horas**

En el siguiente diagrama de Gantt podemos observar las dependencias de estas tareas:

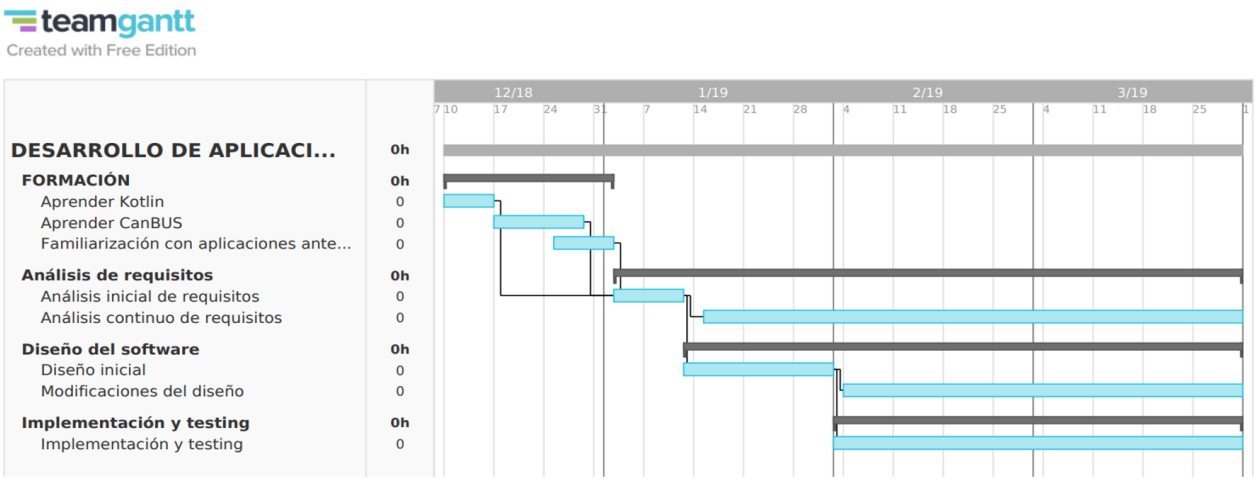


Figura 1: Diagrama de Gantt con la planificación inicial.

Estimación temporal total = 40 + 96 + 56 + 120 + 336 = **648 horas**

2.3. Planificación final

Tal y como se estimó, después de la formación sobre las herramientas que se iban a utilizar, se procedió al análisis de requisitos, el diseño del software y la implementación y testing. Estas tareas eran susceptibles de retomarse una vez empezada la siguiente, lo cual se tuvo en cuenta en aquella planificación y no ha supuesto ninguna desviación en el tiempo empleado finalmente.

Se estableció un margen de dos meses y medio adicionales para absorber imprevistos temporales. En marzo y abril de 2019, no se avanzó en el proyecto ya que fue necesario hacer modificaciones en el software para el modelo anterior. No obstante, esto no impidió que el proyecto se acabara a tiempo.

A mediados de mayo de 2019, se decidió que las funcionalidades acordadas debían separarse en dos aplicaciones distintas: una que sirviera únicamente para la carga de software en los sistemas *embedded* de la moto mediante un protocolo de Bootloader y otra para el resto de

funcionalidades. Además, el primero urgía por necesidades de producción así que toda la semana siguiente fue dedicada a aprender el protocolo y a diseñar, implementar y probar este programa, al que se le puso el nombre de *Bootloader*. Al otro, *Monitor*.

Finalmente, se han añadido dos tareas relacionadas directamente con la parte académica del Trabajo de Fin de grado: la redacción de la memoria y la preparación de la defensa.

El diagrama de Gantt final es el siguiente:

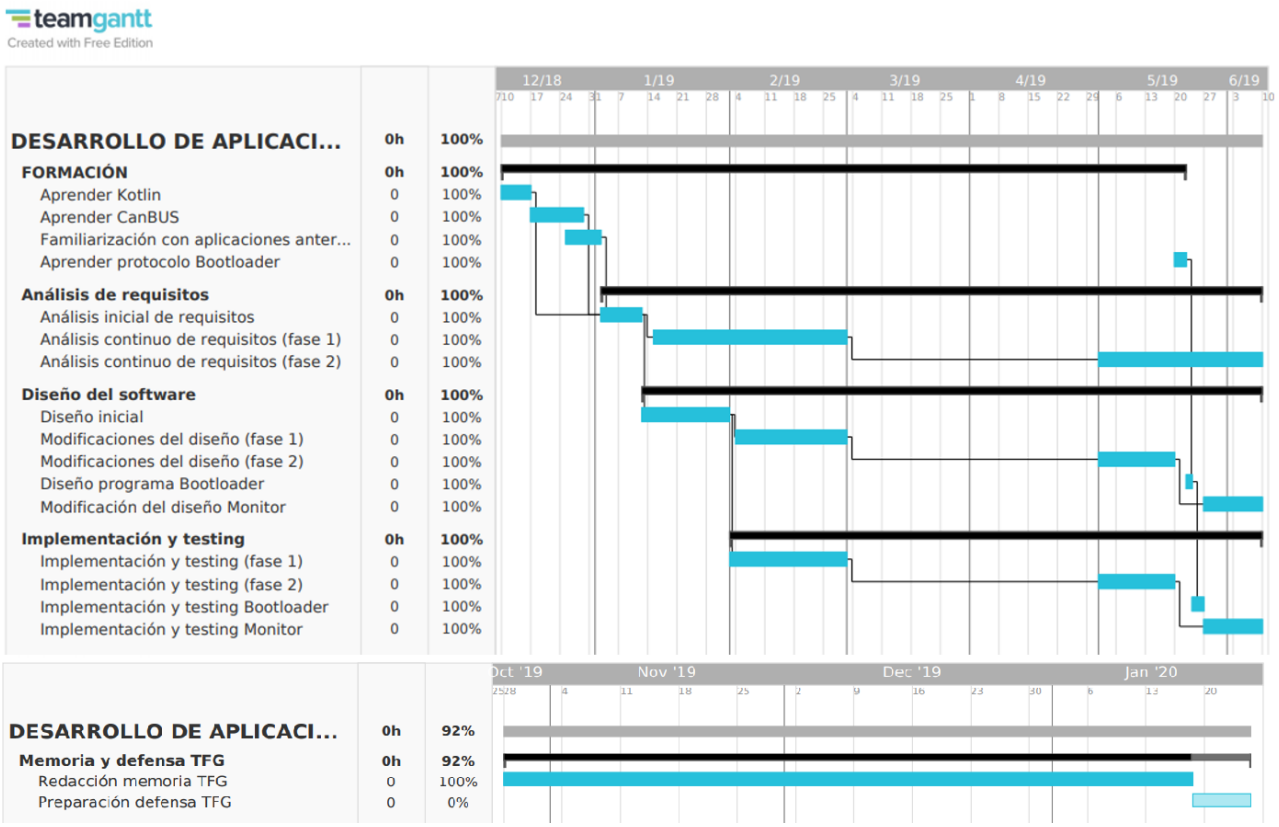


Figura 2: Diagrama de Gantt con la planificación definitiva.

3. Presupuesto

3.1. Identificación de costes

Para el desglose de costes de este proyecto, se identificaron las siguientes categorías de coste:

- Recursos humanos
- Hardware
- Software
- Costes adicionales

Recursos humanos

Se dividieron los recursos humanos en dos subgrupos: aquellos que han participado directamente en su desarrollo y aquellos que han hecho de forma indirecta.

En el primer subgrupo encontramos los participantes directos en el desarrollo:

- **Analista de requisitos:** se encarga de reunirse con los actores implicados y recabar información sobre las funcionalidades que se requieren de las aplicaciones, así como analizar las aplicaciones para la versión anterior.
- **Diseñador del software:** con los datos recopilados, diseña las aplicaciones siguiendo unos rigurosos métodos de diseño. Este diseño debía satisfacer todas las funcionalidades requeridas, así como cumplir otras características que se le suponen a un buen software: modularidad, fácil mantenimiento, inteligibilidad, etc.
- **Desarrollador del software:** siguiendo el diseño establecido, se encarga de la programación *per se*.
- **Testador principal:** a medida que se iban liberando versiones, se encarga de probar que el programa funcione y de detectar posibles errores. Aunque otros actores implicados también podían hacer pruebas con el programa, este es el responsable de que la aplicación funcione correctamente.

Y en el segundo los participantes indirectos:

- **CTO/Director de I+D:** como director del departamento de I+D es el máximo supervisor de todos los proyectos de dicho departamento. Ha estado presente en todas las reuniones semanales y se ha encargado de la realización del proyecto.
- **Responsables de batería y ECU:** como máximos responsables del desarrollo de las partes de la moto con las que se debe comunicar la aplicación, han participado en las reuniones de análisis de requisitos y de seguimiento del proyecto. Además, han participado en las pruebas del programa.

- **Antiguo responsable de estas aplicaciones:** el informático que hasta mi llegada se encargaba de las aplicaciones de monitorización y parametrización y que estaba disponible para consultas.
- **Otros actores implicados:** todos los actores que utilizan la aplicación y que aún no se han mencionado en esta lista han aportado sus sugerencias siempre a través del CTO/-Director de I+D y, por norma general, no han estado presentes en las reuniones.

No se consideró que los recursos humanos de este segundo subgrupo formasen parte directamente del proyecto puesto que su participación en las tareas definidas ha sido mayoritariamente tangencial, es decir, solo consistía en la comunicación de la información de sus respectivos proyectos que podía ser de interés para este. Las reuniones de seguimiento no eran exclusivas de este proyecto. Es más, el tiempo que se le ha dedicado en las mismas era proporcionalmente pequeño. Por ello, estos recursos humanos no se han incluido en el presupuesto del proyecto.

Por lo que al primer grupo concierne, todas las funciones han sido realizadas por mí. Estas tareas son las especificadas en el diagrama de Gantt del capítulo anterior. Como estudiante en prácticas, el sueldo mínimo viene definido por la FIB: 8 euros brutos por hora para prácticas realizadas en el curso 2018/2019. La empresa se ha ceñido a este mínimo.

Hardware

Para el desarrollo del proyecto se utilizará el siguiente hardware:

- Portátil Medion AKOYA S3409 (30021746) - 13.3”Laptop Intel Core i5-7200U / 2.5 GHz (3.1 GHz Turbo) Processor, 8GB RAM, 256GB SSD, Full HD Display, HDMI, USB 3.1, Windows 10 Home 64-bit.
- Monitor AOC e2270swhn 21.5”Widescreen TN LED Black.
- Ratón y teclado Logitech MK270 Wireless Keyboard and Mouse Combo for Windows, Long Range Wireless Connection, 2.4 GHz Wireless, Compact Mouse, Full Sized Keyboard, QWERTY UK Layout, Black.

Software

Para el desarrollo del proyecto se utilizará software gratuito: la versión gratuita (Community) de IntelliJ como IDE y el JDK, que hasta 2018 fue gratuito

. También se estableció un software de gestión de proyectos en el departamento: Wrike.

Costes adicionales

Al presupuesto se le añadió un 10 % adicional por posibles contingencias.

Además, en cualquier momento, el único desarrollador del proyecto podía ser requerido para otras funciones en la empresa, lo cual retrasaría la lista de tareas definidas en el diagrama

de Gantt, circunstancia que ha acabado pasando. Para cualquier imprevisto de este tipo, se contempló en el presupuesto el salario del estudiante en prácticas y el software de gestión de proyectos desde el supuesto final del proyecto, el 29 de marzo de 2019, hasta el final de sus prácticas en la empresa, el 7 de junio de 2019.

3.2. Estimación de costes

Para estimar los costes en recursos humanos se utilizó la previsión temporal y el salario del estudiante en prácticas. Puesto que todas las tareas han sido realizadas por el mismo estudiante, se hizo el cálculo con las horas totales:

- 648 horas x 8 euros por hora = **5184 euros**

A esto hay que añadirle el 15.7 % que la empresa ha de abonar a la UPC "para cubrir el mantenimiento del servicio y sus costes de gestión".

- 15.7 % de 5184 = **813.89 euros**

El hardware se ha comprado en Amazon, de donde podemos obtener los precios exactos:

- Portátil: **490 euros**
- Monitor: **79 euros**
- Teclado y ratón: **17 euros**

En la página de Wrike encontramos los precios del software:

- 24.80 USD al mes por usuario x 1 usuario x 4 meses = 99.2 USD

Los precios están en dólares estadounidenses. Para la estimación en euros consultamos xe.com:

- 99.2 USD = **89.32 euros**

Para los imprevistos calculamos 51 días más de salario (del 1 de abril al 7 de junio de 2019 quitando festivos), con la correspondiente aportación a la UPC, y tres meses más de Wrike (abril, mayo y junio).

- 51 días * 8 horas * 8 euros brutos la hora = **3264 euros**
- 15.7 % de 3264 = **512.45 euros**
- 24.80 USD * 1 usuario * 3 meses = 74.4 USD = **66.99 euros**

Nos quedó la siguiente tabla de gastos:

PRESUPUESTO	
Recursos humanos	
-Salario estudiante en prácticas	**5184 euros**
-Aportación UPC	**813.89 euros**
Hardware	
-Portátil	**490 euros**
-Monitor	**79 euros**
-Teclado y ratón	**17 euros**
Software	
-Gestor de proyectos online (Wrike)	**89.32 euros**
Imprevistos	
-Salario	**3264 euros**
-Aportación UPC	**512.45 euros**
-Wrike	**66.99 euros**
SUBTOTAL	10516.65 euros
CONTINGENCIAS (10 %)	1051.67 euros
TOTAL	**11568.32 euros**

Cuadro 1: Resumen del presupuesto.

3.3. Control de gestión

Este presupuesto era muy robusto. Las desviaciones que surgieron fueron absorbidas con facilidad por la parte destinada a contingencias e imprevistos, anteriormente detallada.

El margen de dos meses entre la duración del proyecto y la duración de las prácticas fue suficiente para absorber los retrasos. La empresa no aumentó los recursos humanos.

También se consideraba muy bajo el riesgo de que el hardware sufra algún deterioro aunque en el improbable caso de que hubiese hecho falta cambiar o reparar algún elemento de trabajo, esto habría sido cubierto como contingencia. Finalmente, no ha hecho falta recurrir a esto.

4. La moto S01

La moto de Silence para la que se han desarrollado las aplicaciones de este proyecto es el modelo S01. Se trata de una moto eléctrica equivalente a una 125cc diseñada para uso urbano y totalmente ensamblada en Barcelona. Tiene una velocidad máxima de 100 km/h y una autonomía de la batería que oscila sobre los 115 Km. Se ha estimado que supone un ahorro del 85 % comparado con una moto de combustión.

- **Componentes generales**

Los componentes generales que forman esta moto son los siguientes:

Figura 3: Componentes generales de la moto S01 de Silence.

a. **Motor**
En la rueda trasera, se aloja un motor 100 % eléctrico con tecnología *Brushless HUB*, que proporciona transmisión directa y refrigeración por aire. Su potencia nominal es de 7000W (homologación de la categoría L3e) con una potencia máxima que puede alcanzar los 11000 W. La velocidad máxima es de 100km/h. y permite una aceleración de 0 a 50 Km/h en 3.8 segundos.

Los motores *brushless HUB* son los que se emplean en patinetes eléctricos y, cada vez más, en motos eléctricas. Como van instalados en la rueda, no necesitan cadenas ni piezas que puedan producir roces. Son motores mucho más ligeros que apenas requieren mantenimiento.

Dispone de freno regenerativo, freno de motor y marcha atrás.

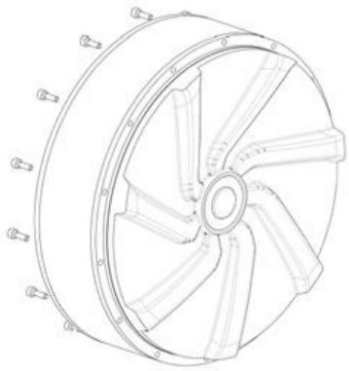


Figura 4: Detalle esquemático del motor *brushless HUB* .

b. **Pack de baterías**
El modelo S01 de Silence dispone de un innovador pack de baterías extraíble compuestas por celdas de ión Litio. Dispone de asa y ruedas para poder ser transportado a modo de trolley. De este modo es posible llevarlo de la propia moto a donde se desee, como por ejemplo, a una toma de corriente. Este pack de baterías, además, puede ser intercambiado entre motos del mismo modelo e incluso ser usado para alimentar otros aparatos.

b.a. **Energy IN**
Se denomina Energy IN a las distintas modalidades de carga de la batería. Puede cargarse en uno de los armarios de carga rápida que se prevé que se distribuyan en varios puntos de la ciudad, cargarlas en una toma de corriente normal o a través de energía solar mediante el denominado árbol solar que se puede adquirir como accesorio para generar un círculo de energía limpia.

Figura 5: Árbol solar. Dispositivo de carga diseñado por Silence.

Por último, está previsto que puedan usarse las *Battery Station SILENCE*, un sistema de intercambio de baterías. Se podrá reservar una batería cargada y disponible a través de la APP de Silence e intercambiarla por la batería descargada sin perder tiempo.

La conexión se realiza mediante una clavija IEC macho en la cual se conecta el cable de alimentación.

Figura 6: Detalle del dispositivo del cargador incorporado en el pack de baterías.

El cargador de 600W va incorporado en el propio pack de baterías. Durante la carga se refrigera el cargador por convección. Por seguridad, sólo se realizará la carga si la temperatura está entre 0 y 55 ºC (estas temperaturas límite son ejemplo de los parámetros que se pueden configurar con los programas desarrollados). Si se conecta estando a baja temperatura, se enciende un calentador interno que eleva la temperatura hasta los 15ºC aproximadamente. Este calentador funciona mientras

esté enchufada a la corriente y consigue que el paso de corriente se produzca en condiciones de temperatura adecuadas.

b.b. **Energy OUT**
Por otro lado, está la modalidad Energy OUT, que es cuando la batería tiene conectado algún elemento al que le proporciona energía. Este elemento puede ser la moto, obviamente, pero, debido al sistema extraíble y a un inversor que convierte los 60V de corriente contínua a 220V de corriente alterna, también puede enchufarse cualquier aparato que requiera conectarse a una fuente de 220V.

Figura 7: Ejemplo de uso del inversor incorporado en el pack de baterías.

b.c. **Sistema de control de la batería (BMS)**
La batería dispone de un sistema de monitorización denominado BMS (*Battery Magagement System*) que se encarga de controlar los parámetros tales como la temperatura y la tensión. Ha sido desarrollado por el equipo de I+D de Silence. El sistema es el encargado de asegurar un balance de la carga individual y en conjunto de cada serie de celdas, permitiendo un funcionamiento óptimo de la batería. Este sistema también es el responsable de comunicar el estado de carga y fijar las consignas de corriente de carga y descarga. También dispone de un plan de acción en caso de un estado anómalo de la batería, haciendo saltar protecciones a modo preventivo en caso de exceso de corriente, tensión o temperatura.

Este es uno de los dos sistemas *embedded* con los que se comunican los programas desarrollados.

b.d. **Sensores de temperatura**
La moto S01 cuenta con un sistema de control y estabilización de tensión y temperatura de las celdas. Para evitar situaciones críticas, los sistemas de seguridad limitan el uso de la batería si la temperatura de la celda supera los límites de seguridad.

El rango de funcionamiento de la batería es entre -10º y 65ºC. Dependiendo de la temperatura, el rendimiento de las celdas de litio puede ser variable.

El cargador no carga la batería si la temperatura de las celdas es inferior a 0ºC o superior a 55ºC.

La temperatura actual de la batería puede consultarse en la pantalla de la moto. En caso de superarse alguno de los límites, se indicará mediante el correspondiente led del tablero luminoso.

c. **Cuadro de instrumentos**

El cuadro de instrumenos permite conocer toda la información relativa a la moto necesaria para la conducción. Dispone de una pantalla LCD, 2 botones ("SET" e "INFO") y 10 testigos luminosos.

Figura 8: Detalle del cuadro de instrumentos de la S01.

c.a. **Indicador de carga**

Marca el estado de carga de la batería o SOC (State of Charge).

Figura 9: Detalle de la posición del indicador de carga en el cuadro de instrumentos.

c.b. **Indicador de corriente**

Muestra el valor de la corriente instantánea en un arco que se encuentra en la parte superior. Este valor puede ser positivo, si la corriente está saliendo de la batería al ser consumida, o negativo, si la corriente está entrando (al ser regenerada mediante el freno regenerativo del motor).

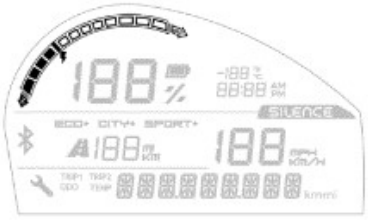


Figura 10: Detalle de la posición del indicador de corriente en el cuadro de instrumentos.

c.c. **Temperatura ambiente**
Muestra el valor de la temperatura ambiente en la parte superior derecha. Este valor se puede mostrar tanto en grados Celsius como en Fahrenheit, según la configuración que se haya establecido.

Figura 11: Detalle de la posición del indicador de temperatura ambiente en el cuadro de instrumentos.

c.d. **Hora actual**
La hora actual se muestra justo debajo de la temperatura. Puede mostrarse tanto en modo 12 horas como en modo 24 horas.

Figura 12: Detalle de la posición del indicador de Hora actual en el cuadro de instrumentos.

c.e. **Conexión bluetooth**
En la parte izquierda del cuadro se muetra el icono de Bluetooth que indica el emparejamiento entre el smartphone del conductor y la ECU de la moto. Este icono parpadea cuando se produce el emparejamiento y se queda fijo de forma indefinida cuando este se ha completado. Cuando el usuario desconecta su smartphone, deja de estar activo.

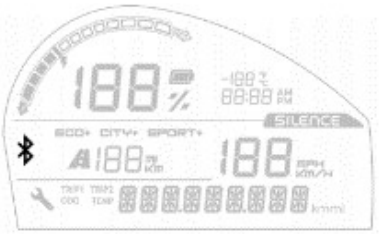


Figura 13: Detalle de la posición del indicador de conexión bluetooth en el cuadro de instrumentos.

c.f. **Modo de conducción**
Bajo el porcentaje de carga, se muestra el modo de conducción actualmente activo. Existen tres modos:

- ECO
- CITY
- SPORT

Figura 14: Detalle de la posición del indicador de modo de conducción en el cuadro de instrumentos.

c.g. **Autonomía restante estimada**
Bajo el modo de conducción se encuentra disponible la información de la autonomía restante, en kilómetros o millas. Este dato es aproximado y depende del modo de conducción y de la descarga que se esté produciendo em ese momento.

Figura 15: Detalle de la posición del indicador de autonomía restante estimada en el cuadro de instrumentos.

c.h. **Indicador de velocidad**
A la derecha, se indica la velocidad actual de la moto. Puede leerse en kilómetros por hora o en millas por hora, según la configuración establecida.

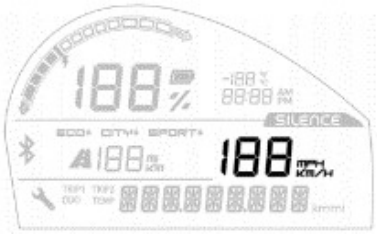


Figura 16: Detalle de la posición del indicador de velocidad en el cuadro de instrumentos.

c.i. **Indicador de servicio**
Este indicador con forma de llave fija situado en la parte inferior izquierda del indicador se iluminará cuando sea necesario realizar una revisión periódica de la moto (según el kilometraje). Los talleres autorizados de la marca podrán desactivar este icono mediante la aplicación *Monitor* desarrollada en este proyecto y volverá a activarse automáticamente cuanto se requiera una nueva revisión. No obstante, también puede desactivarse este aviso periódico con los botones SET e INFO.

Figura 17: Detalle de la posición del indicador de servicio en el cuadro de instrumentos.

c.j. **Odómotro**
El odómetro o cuentakilómetros indica el total de kilómetros (o millas, según la configuración establecida) recorridos desde que el vehículo salió de la fábrica.

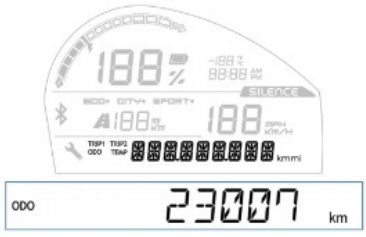


Figura 18: Detalle de la posición del indicador del cuentakilómetros.

c.k. **Temperaturas de algunos componentes**
El botón INFO permite mostrar las temperaturas de algunos componentes de la moto:

- TEMP BAT: temperatura del pack de baterías
- TEMP ENG: temperatura del motor
- TEMP INV: temperatura del controlador o inverter

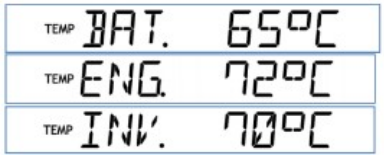


Figura 19: Indicador de las distintas temperaturas.

c.l. **Cuentakilómetros parciales**
El indicador también pemite mostrar la distancia en kilómetros o millas recorridos desde que se reinició. Hay dos cuentaquilómetros parciales (TRIP1 y TRIP2). En cada uno de ellos se puede mostrar tanto la distancia recorrida como la velocidad media a la que se ha recorrido dicha distancia.

Estos cuentaquilómetros de alternan entre ellos y con el cuentakilómetros total (ODO) mediante el botón INFO.

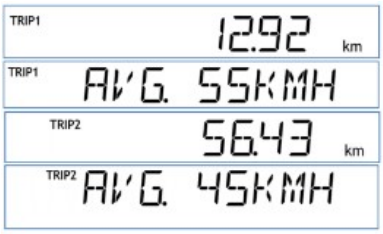


Figura 20: Indicador de cuentakilómetros y velocidades medias parciales.

d. **Botones INFO y SET**
Para seleccionar las distintas opciones del cuadro de instrumentos, la moto dispone de dos botones: INFO (derecha y duplicado en los controles del lado derecho del manillar) y SET (izquierda).

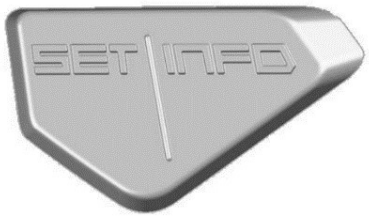


Figura 21: Detalle de los botones INFO y SET.

e. **Testigos luminosos**

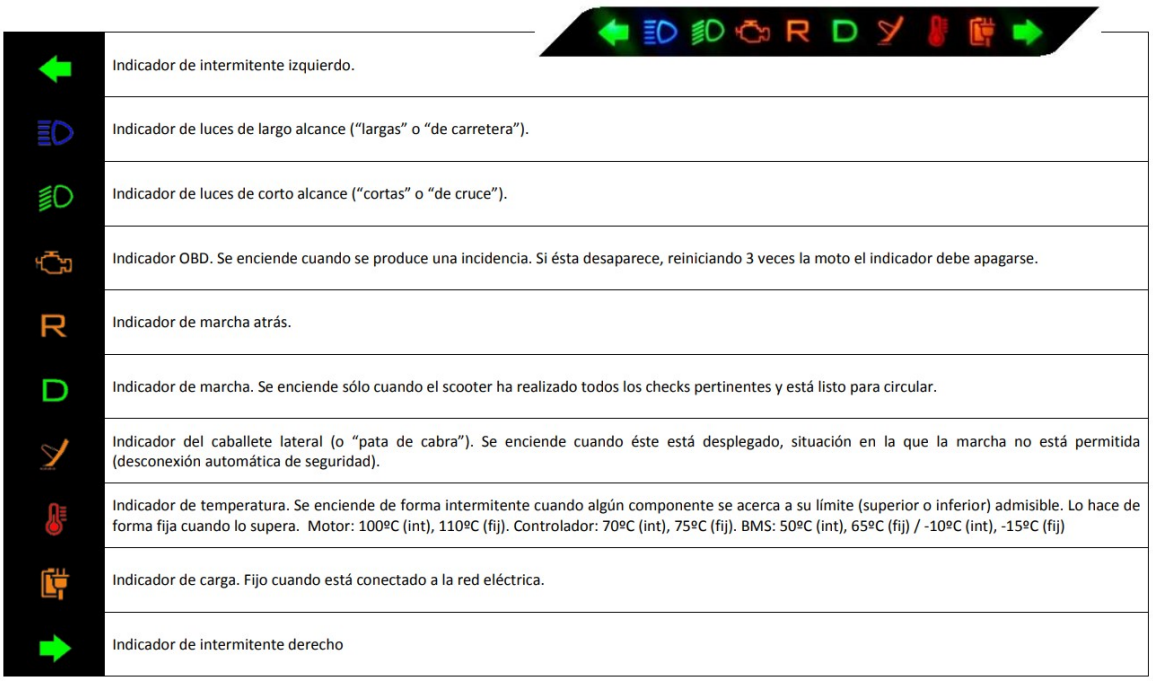


Figura 22: Conjunto de los testigos luminosos del cuadro de instrumentos.

f. **Controles y manejo del vehículo**

Todos los controles están en la parte delantera de la moto. La siguiente figura muestra todos estos controles:

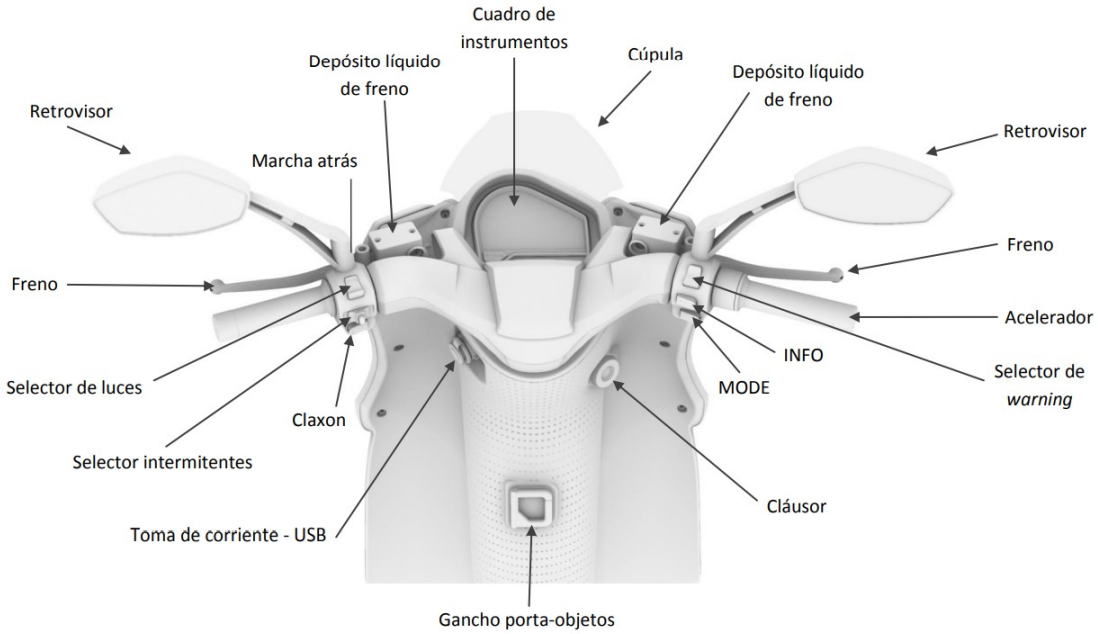


Figura 23: Controles de la parte delantera de la moto.

En el lado izquierdo del manillar se encuentra el selector de luces, intermitentes, claxon y botón de marcha atrás. En el lado derecho el selector de *warning* y los botones INFO y MODE. Este lado derecho contiene también el acelerador.

f.a. **Frenos combinado y regenerativo**
Esta moto está equipada con un sistema de freno combinado que funciona de la siguiente manera: el freno derecho frena la rueda delantera mecánicamente (mediante freno de disco) y activa el freno regenerativo (electrónicamente) de la rueda trasera, mientras que el freno izquierdo frena mecánicamente tanto la rueda delantera como la trasera (aplicando fuerza sobre ambos discos, gracias a un distribuidor de frenada). Ambas manetas son regulables mediante la aplicación *Monitor*. El freno regenerativo aprovecha la energía de frenada para cargar la batería.

f.b. **Modos de funcionamiento**
El botón MODE situado en el manillar permite cambiar el modo de conducción. Hay tres modos de conducción:

- **CITY 'C':** Es el modo por defecto, el que tiene mejores prestaciones y un consumo más equilibrado. Dispone de freno regenerativo limitado. La velocidad máxima en este modo es de 85 Km/h.
- **SPORT 'S':** Es el modo de conducción que permite disponer de una mayor potencia y velocidad en situaciones puntuales. El uso frecuente del modo SPORT disminuye la autonomía de la moto y puede llegar a aumentar en exceso la temperatura del motor y la batería, lo que provocaría la bajada de rendimiento o desconexión del vehículo. Dispone de freno regenerativo total. La velocidad máxima en este modo es de 100 Km/h. El modo SPORT solo está disponible si el estado de carga de la batería supera el 20 %, la temperatura de la batería no supera los 45ºC, la temperatura del motor no supera los 105ºC y la del inversor los 70ºC.
- **ECO 'E':** Permite una conducción más relajada, donde la velocidad y la aceleración están limitadas. Todo ello permite una mayor autonomía al vehículo. No dispone de freno regenerativo. La velocidad máxima en este modo es de 67 Km/h. No es posible cambiar a este modo si la moto va a más de 55 Km/h.

f.c. **Iluminación**
Toda la iluminación de la moto utiliza LEDs, incluidos los intermitentes, luces de posición, de freno, de cruce y largas. Los distintos grupos ópticos son:

- Faro delantero (luces largas, cortas y semiaros perimetrales de posición)
- Luces de posición e intermitentes delanteros
- Grupo óptico posterior (posición, freno e intermitentes traseros)

- **APP para smartphone**

Una característica importante de la S01 es su aplicación para móvil que permite conocer el estado de la moto desde cualquier lugar. Desde la aplicación es posible:

- Encender y apagar la moto

- Abrir el asiento
- Compartir la moto mediante un código sin necesidad de llave
- Encontrar la moto mediante geolocalización
- Conocer el estado de la batería
- Conocer la autonomía restante (en modo CITY)
- Recibir alertas por caídas o robo
- Recibir alertas sobre la temperatura del vehículo
- Planificar la ruta
- Conocer la huella de carbono (estadísticas del CO2 dejado de emitir)

5. Protocolos de comunicación

5.1. CAN Bus

CAN Bus es la columna que vertebra toda la electrónica de los automóviles hoy en día y, por supuesto, también la electrónica de la S01.

En un origen, los dispositivos electrónicos en un automóvil eran muy pocos y se conectaban unos con otros mediante cables directos (punto a punto). A medida que la tecnología iba avanzando, se iban añadiendo más y más dispositivos, sensores y actuadores que complicaban sustancialmente el cableado. Fue entonces cuando se decidió definir un protocolo de comunicaciones para la automoción y es así como en 1982 nace el CAN bus.

CAN es el acrónimo de *Controller Area Network* y es un protocolo definido sobre una topología de bus, es decir, un solo cable recorre el vehículo conectando los diferentes dispositivos electrónicos que deben comunicarse. Esta topología representa un enorme ahorro de cable respecto a la conexión punto a punto.

CAN es un protocolo orientado a mensajes, es decir, la información que se va a intercambiar se descompone en mensajes de un tamaño máximo de 8 bytes. Cualquier dispositivo conectado al bus puede mandar mensajes y el resto le escuchan. Cada tipo de mensaje lleva un identificador con el cual los nodos deciden si les es relevante o no.

El primer modelo de automóvil en utilizarlo fue el Mercedes-Benz clase E de 1992. Desde entonces, el CAN bus se ha convertido en un estándar de facto en el sector del automóvil ya que proporciona alta inmunidad a las interferencias, herramientas para el autodiagnóstico y reparación de errores de datos. Aunque están apareciendo nuevos competidores como el moderno FlexRay u otros protocolos con interfaces de fibra óptica, todavía CAN es el sistema de comunicación líder en la producción automovilística actualmente.

Al ser los automóviles entornos agresivos (calor, frío, vibraciones, ruido electromagnético, etc.), se requiere que el sistema sea robusto y fiable. La transmisión de señales en un bus CAN se lleva a cabo a través de dos cables trenzados. Las señales de estos cables se denominan CAN_H (CAN high) y CAN_L (CAN low). El bus tiene dos estados definidos: estado dominante (estado 0) y estado recesivo (estado 1). En estado recesivo, los dos cables del bus se encuentran al mismo nivel de tensión (*common-mode voltage*), mientras que en estado dominante hay una diferencia de tensión entre CAN_H y CAN_L de al menos 1,5 V. La transmisión de señales en forma de tensión diferencial, en comparación con la transmisión en forma de tensiones absolutas, proporciona protección frente a interferencias electromagnéticas.

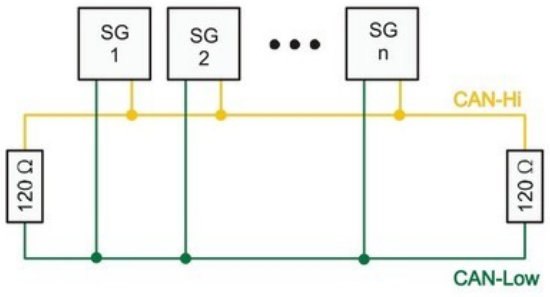


Figura 24: Arquitectura básica del bus CAN.

La velocidad del CAN bus va desde los 40Kbps en distancias de un kilómetro hasta 1Mbps, siempre que el cable no mida más de 40 metros. En la práctica, en automoción se emplean velocidades entre los 125Kbps hasta los 500Kbps. El aumento de velocidad incide negativamente en la fiabilidad de la comunicación.

Una trama de CAN contiene la siguiente información:

- Identificador
- *Payload*
- *Timestamp*
- DLC (*Data Length Code*): es el tamaño en bytes del *payload* que vale entre 0 y 8
- RTR (*Remote Transmission Request*): flag que vale 1 si la trama es remota, 0 en cualquier otro caso. Una trama remota es aquella que se emplea para solicitar datos al receptor. En las comunicaciones con la S01, este tipo de trama solo se utiliza para solicitar algunos datos. Para la mayoría de datos que se le solicitan a la moto se emplean dos tramas no remotas que tienen identificadores distintos, uno para la trama que solicita y el otro para la que devuelve los datos.

El equipo de I+D de Silence ha desarrollado un extenso *datasheet* que detalla qué trama es la correspondiente a cada identificador, qué datos contiene y si es una trama remota o no.

ID hex	DLC	Description	Period (ms	0	1	2	3	4	5	6	7		Node	Tx/Rx	MODE
181h	8	Status frame	300	counter	SOC [%]	owet temp. [ºC	hest temp. [º	Pack voltage [0.1V]		Pack current [0.1A]		ELIMINAR		TX	0
182h	7	Error frame	-	State				MSB Warnings LSB				NO	ECU	TX	NO
183h	8	BPID	async	BPID				YY	MM	DD			BMS	TX & RTR	1
184h	8	BMScutum	300	_BMScutum_status									BMS	TX	0
185h	8	CellV 1-4	300	mV cell 1		mV cell 2		mV cell 3		mV cell 4			BMS	TX	0
186h	8	CellV 5-8	300	mV cell 5		mV cell 6		mV cell 7		mV cell 8			BMS	TX	0
187h	8	CellV 9-12	300	mV cell 9		mV cell 10		mV cell 11		mV cell 12			BMS	TX	0
188h	6	CellV 13-15	300	mV cell 13		mV cell 14		mV cell 15					BMS	TX	0
189h	8	Current + Temps	300	Current (10 mA)		TempA (0.01 ºC)		TempB (0.01 ºC)		TempC (0.01 ºC)			BMS	TX	0
18Ah	8	ADC0 RAW	300	Channel 3		Channel 2		Channel 1		Channel 0				TX	10
18Bh	8	ADC1 RAW	300	Channel 2		Channel 6		Channel 5		Channel 13				TX	10
18Ch	6	ADC1 RAW	300	Channel 4		Channel 12		Channel 10						TX	10
18Dh	8	ADC2 RAW	300	Channel 3		Channel 5		Channel 4		Channel 2				TX	10
18Eh		balancing		balance flags active		flags status	age_delta_ma	milis_balance						TX	0
18Fh	8	VC RAW SUM debug	300	BAT RAW		VC RAW SUM				CC_RAW				TX	10
190h	5	SoCs	300	SoC (1/1000)		Remaining capacity (mAh)		VSOC	SOH					TX	0
0x191	8	BMStatus	300	_BMScutum_flags										TX	0
192	4	Ah	300	Actual Capacity										TX	10
193	8	GPIOA&B	300	GPIOA PIR				GPIOB PIR						TX	11
194	8	GPIOC&D	300	GPIOC PIR				GPIOD PIR						TX	11
195	4	GPIOE	300	GPIOE PIR										TX	11
196	8	Ah p	300	Previous Capacity Chg				Previous Capacity Dsch						TX	10

Cuadro 2: Panorámica del *datasheet* que detalla las tramas asociadas a cada identificador.

Por ejemplo, la trama con identificador 0x183 es remota. Cuando el PC lo envía con el flag RTR a 1, el BMS devuelve una trama con el mismo identificador y cuyo *payload* contiene el BPID de la batería en los cuatro primeros bytes y la fecha de fabricación de la moto en los tres siguientes.

El BMS y la ECU están enviando mensajes de CAN constantemente con datos a tiempo real. Una de las funciones principales del programa *Monitor* es recogerlos, identificarlos y mostrarlos de forma visual en pantalla.

La segunda función principal de este programa es la parametrización. En el *datasheet* se puede encontrar un apartado dedicado a esto.

ID hex	DLC	Description	Period (ms	0	1	2	3	4	5	6	7
233h		Params Write		parameter 0	value 1	value 2	value 3	value 4	value 5	value 6	value 7
234h		read param request		parameter 0							
235h		saved param		parameter 0	value 1	value 2	value 3	value 4	value 5	value 6	value 7

Cuadro 3: Panorámica del *datasheet* asociado a la parametrización.

Tanto el BMS como la ECU tienen tres identificadores dedicados a los parámetros:

- Trama para escribir parámetros en el BMS o la ECU. En el *payload* se especifica el identificador de los parámetros que se quieren escribir y se envían los valores con los que se quiere configurar el dispositivo.
- Trama de solicitud de lectura de parámetros actuales que tienen configurados el BMS o la ECU. El *payload* solo especifica el identificador de los parámetros que se quieren leer.
- Trama respuesta a la solicitud de lectura. Es enviada por el BMS o la ECU cuando se le solicitan sus parámetros actuales. El *payload* contiene el identificador de los parámetros que se han solicitado y sus valores.

Los identificadores de parámetros también van definidos en el *datasheet*:

parameter 0	value 1	value 2	value 3	value 4	value 5	value 6
00	RESERVED (parameters received)					
01	capacity (mAh)					
02	charge current (mA)				VCT current (mA)	
03	precharge current (mA)				revive current (mA)	
04	ki_cc_1A/x*10		ki_cv_1A/x*10		ext chg factor 10/x	
05	Current_dsch_max (mA)				Current_dsch_high (mA)	
06	Current_reg_max (mA)				Current_reg_high (mA)	
07	Current_chg_max (mA)					
08	Current_chg_ob_high (mA)				Current_chg_ex_high (mA)	
09	cell voltage max (mV)		cell voltage high (mV)		cell voltage max chg (mV)	
0A	cell voltage min (mV)		cell voltage low (mV)		cell voltage min chg (mV)	
0B	cell voltage harakiri (mV)		cell voltage harakiri delta (mV)		harakiri delay (sec)	
0C	cell voltage power down (mV)		cell voltage power down delta (mV)		power down delay (sec)	
0D	Temp max dsch (signed, ºC)	Temp high dsch (signed, ºC)	Temp min dsch (signed, ºC)	Temp low dsch (signed, ºC)	Temp min chg (signed, ºC)	Temp low chg (signed, ºC)
0E	Temp Heater OFF (signed, ºC)	Temp Fan OFF (signed, ºC)	Time Fan OFF (sec)		Temp max chg (signed, ºC)	Temp high chg (signed, ºC)
0F	Temp Enable	Temp Detected				
10	OverTemp Warning ΔT (dºC)	UnderTemp Warning ΔT (dºC)	UnderVoltage Warning ΔV (mV)	Low Soc %		
11	WOV Time (0,1s)		WUV Time (0,1s)		WOT Time (0,1s)	
12	WOCD Time (0,1s)		WOCR Time (0,1s)		WOCC Time (0,1s)	
13	EOV Time (0,1s)		EUV Time (0,1s)		EOT Time (0,1s)	
14	EOCD Time (0,1s)		EOCR Time (0,1s)		EOCC Time (0,1s)	
15	WUT Time (0,1s)		EUT Time (0,1s)			

Cuadro 4: Panorámica del *datasheet* asociado a los identificadores de parámetros.

Por ejemplo, el grupo de parámetros 0x10 contiene los valores límites a partir de los cuales saltan las advertencias de sobrecalentamiento, temperatura demasiado baja, voltaje demasiado bajo y estado de carga de la batería demasiado bajo. Todos los valores que ocupan más de un byte están ordenados en *little endian*.

Para la comunicación entre los programas que se han desarrollado y los drivers de Ixxat se ha utilizado una librería desarrollada por el anterior informático de la empresa. No obstante, ha sido necesario modificarla para que se pudiese configurar adecuadamente el DLC y el RTR.

5.2. Kinetis ROM Bootloader

Los sitemas *embedded* de la moto (BMS y ECU) funcionan con microcontroladores Kinetis de la subfamilia KE1xF. Estos microcotrontroladores contienen el *bootloader* en una ROM. Al encenderse, el hardware comprueba si hay una imagen de *firmware* en la memoria flash. Si lo hay, lo ejecuta. Si no, ejecuta el *bootloader* para poder cargar una. También se puede forzar el *bootloader* mediante una trama de CAN.

Para cargar una imagen de firmware en la memoria flash, el *bootloader* hace las veces de proceso esclavo y escucha al periférico conectado por CAN (también soporta otras interfaces como I2C, SPI o UART) que ejerce de maestro. Este último es el que se encarga de enviar los distintos comandos (escribir en memoria, borrar memoria, configurar las opciones del *bootloader*, etc.) y los datos. Para ello, se emplea un protocolo específico que se explicará a continuación y al que denominaremos «protocolo *bootloader*».

Al iniciarse, el *bootloader* lee una serie de parámetros de configuración (p.ej. qué periféricos están habilitados o en qué posición de las tramas se encuentra el CRC) de una zona fija de la memoria flash denominada *Bootloader Configuration Area* (BCA) que empieza en la dirección de memoria 0x3C0 y ocupa 64 bytes. Para cambiar estos parámetros, se debe escribir en esa zona de la memoria flash. Si todos los bits de la zona valen 1, se utilizan los parámetros por defecto.

Una de las cualidades del *bootloader* es la capacidad de detectar la velocidad de las comunicaciones de CAN Bus del periférico automáticamente. Tanto para iniciar las comunicaciones, una vez el microcontrolador entra en modo *bootloader*, como para detectar la velocidad automáticamente y el periférico adecuado, es necesario enviar una trama de ping. Una vez iniciadas las comunicaciones, se pueden enviar el resto de comandos. Todas las tramas que se envíen deben ser contestadas por una trama de *acknowledge* (ack). No se enviará la siguiente trama hasta no haber recibido esta respuesta.

Todas las tramas tienen una cabecera de la siguiente forma: byte inicial con valor 0x5A, segundo byte de código de trama. Para las tramas distintas a ping, ping response y ack, la cabecera también contiene: dos bytes que indican el tamaño en bytes del payload de la trama (el contenido de la trama sin contar la cabecera) y un CRC de dos bytes. Estos dos valores están en orden *little endian*. En la siguiente tabla se muestra esta estructura. Las tramas ping y ack solo contienen los dos primero bytes. La trama ping response se trata más adelante.

Byte #	Value	Parameter	
0	0x5A	start byte	
1		packetType	
2		length_low	Length is a 16-bit field that specifies the entire command or data packet size in bytes.
3		length_high	
4		crc16_low	This is a 16-bit field. The CRC16 value covers entire framing packet, including the start byte and command or data packets, but does not include the CRC bytes. See the CRC16 algorithm after this table.
5		crc16_high	
6 . . .n		Command or Data packet payload	

Cuadro 5: Estructura de las tramas del bootloader.

Byte #	Value	Name
0	0x5A	start byte
1	0xA6	ping

Cuadro 7: Tipos de tramas con sus respectivos códigos.

El CRC, acrónimo de *cyclic redundancy check*, es un código de detección de errores que se calcula a partir de la trama y se utiliza para comprobar si la trama recibida es la misma que se ha enviado.

En la siguiente tabla podemos encontrar todos los tipos de trama posibles con sus respectivos códigos:

Código	Nombre	Descripción
0xA1	`kFarmingPacketType_Ack`	El paquete previo a esta trama ha sido recibido correctamente.
0xA2	`kFarmingPacketType_Nak`	El paquete previo a esta trama está corrompido y debe ser reenviado.
0xA3	`kFarmingPacketType_AckAbort`	El envío de datos ha sido abortado.
0xA4	`kFarmingPacketType_Command`	El payload de esta trama es un comando.
0xA5	`kFarmingPacketType_Data`	El payload de esta trama son datos.
0xA6	`kFarmingPacketType_Ping`	Trama previa a la comunicación.
0xA7	`kFarmingPacketType_PingResponse`	Respuesta a la trama de ping.

Cuadro 6: Todas las tramas posibles y sus respectivos códigos.

- **Trama ping**

Esta trama debe ser la primera que se envíe y se envía de periférico a *bootloader*. Sirve para establecer la conexión con velocidad y periférico adecuado. La trama se compone de 2 bytes, 5A el start byte y A6 que es el comando de PING.

- **Trama pingResponse**

Esta trama es la respuesta a una trama ping.

- **Trama ack**

Byte #	Value	Parameter
0	0x5A	start byte
1	0xA7	Ping response code
2		Protocol bugfix
3		Protocol minor
4		Protocol major
5		Protocol name = 'P' (0x50)
6		Options low
7		Options high
8		CRC16 low
9		CRC16 high

Cuadro 8: Estructura de la trama ping.

La trama de *Acknowledge* es la trama más corta (junto a la de Ping) y la más utilizada. Es una trama que se envía después de cada mensaje (excepto ping, que tiene su propio mensaje de respuesta) sea en el sentido de la comunicación que sea. La finalidad de esta trama es la de hacer saber al transmisor del mensaje anterior que el receptor lo ha recibido y entendido correctamente.

Si el mensaje recibido requiere una respuesta adicional (p.ej. cuando se solicita leer la memoria), esta se envía inmediatamente después del ack y se esperará otro ack conforme ha sido recibida.

Por otra parte, tenemos el *Not Acknowledge* (nak), que se enviará cuando el mensaje recibido no sea correcto. Esto puede deberse a errores en los comandos o en la comprobación del CRC.

- **Trama de comando**

El *payload* de una trama de comando tiene esta forma:

Command Packet Format (32 bytes)										
Command Header (4 bytes)				28 bytes for Parameters (Max 7 parameters)						
Tag	Flags	Rsvd	Param Count	Param1 (32-bit)	Param2 (32-bit)	Param3 (32-bit)	Param4 (32-bit)	Param5 (32-bit)	Param6 (32-bit)	Param7 (32-bit)
byte 0	byte 1	byte 2	byte 3							

Cuadro 9: Estructura del payload de una trama de comando.

Los primeros 4 bytes forman una cabecera específica de la trama comando y los 28 restantes son parámetros que algunos comandos requieren. La cabecera contiene los siguientes elementos:

El Tag indica el código del comando que se quiere enviar. Flags vale 1 si este comando va seguido de tramas de datos, 0 en cualquier otro caso. El tercer byte está reservado y debe valer 0x00. El cuarto es el número de parámetros del comando.

Byte #	Command Header Field	
0	Command or Response tag	The command header is 4 bytes long, with these fields.
1	Flags	
2	Reserved. Should be 0x00.	
3	ParameterCount	

Cuadro 10: Estructura de los cuatro primeros bytes.

Command	Name
0x01	FlashEraseAll
0x02	FlashEraseRegion
0x03	ReadMemory
0x04	WriteMemory
0x05	Reserved
0x06	FlashSecurityDisable
0x07	GetProperty
0x08	Reserved
0x09	Execute
0x0A	Reserved
0x0B	Reset
0x0C	SetProperty
0x0D	FlashEraseAllUnsecure
0x0E	Reserved
0x0F	Reserved
0x10	Reserved
0x11	Reserved
0x12	Reserved

Cuadro 11: Relación dels códigos de comando.

Los códigos de comando son:

- **Trama de datos**

El *payload* de una trama de datos es únicamente los datos a enviar. De esta manera, se pueden enviar un máximo de 32 bytes de datos por trama (el máximo que permite el tamaño de las tramas). La cabecera debe estar ya que permite verificar que los datos se han enviado correctamente. Si se quiere enviar más de 32 bytes, la información debe dividirse en paquetes de este tamaño y ser enviados sucesivamente.

- **Trama de respuesta**

Las tramas de respuesta se envían justo después de haberse procesado un comando. El código de frame y la cabecera son los mismos que los de comando. El byte de código de comando puede valer 0xA0, 0xA7 o 0xA3 dependiendo de si se trata de una respuesta genérica (que responde a todos los comandos menos a los dos que tienen una respuesta específica), una respuesta al comando getProperty o al comando readMemory, respectivamente. En el protocolo que implementa el software de este proyecto solo se consideran las dos primeras.

Igual que las tramas de comando, estas tramas pueden tener parámetros a continuación de la cabecera de comando. En el caso de la respuesta genérica, los parámetros son el código de estado, que indica si ha habido algún error durante la ejecución del comando, y el código del comando al que responde. En cuanto a la respuesta al getProperty, los parámetros son las propiedades que se han solicitado.

Aunque en el manual del microcontrolador nombra CAN Bus entre las interfaces con las que el *bootloader* puede comunicarse, no se especifica cómo formular las tramas de hasta 38 bytes con el protocolo de CANbus que permite un *payload* máximo de 8 bytes.

A través de pruebas que se han realizado, se ha demostrado que basta con dividir las tramas del *bootloader* en paquetes de 8 bytes (salvo el último, que puede tener menos) y enviarlos uno detrás del otro sin necesidad de especificar cúantos de estos paquetes se van a enviar. De esta manera se respeta el protocolo.

6. Análisis de requisitos

Para llevar a cabo el análisis de los requisitos del software de actualización, monitorización y parametrización de la moto, antes de empezar con el diseño, tuvo lugar una serie de reuniones con los actores que iban a usar estas aplicaciones y con el desarrollador de las aplicaciones equivalentes para el modelo anterior. En estas reuniones, se detalló para qué debían servir, qué aspectos de las del modelo anterior debían permanecer iguales, cuáles debían modificarse o mejorarse y qué funcionalidades completamente nuevas debían añadirse.

También se estableció que, debido a la dinámica con la que se estaban abordando otros aspectos de la producción del vehículo, el análisis de requisitos resultante de estas reuniones iniciales era susceptible de modificarse. Para ello, se llevaron a cabo reuniones ordinarias semanales y algunas extraordinarias para discutir la eliminación, modificación o adición de funcionalidades en función del estado de la producción en ese momento.

En una de estas reuniones que tuvo lugar a mediados de mayo de 2019, se decidió que la actualización del software interno de las motos debía hacerse con una aplicación distinta que la monitorización y parametrización. Al primer programa se le llamó *Bootloader*; al segundo, *Monitor*.

A continuación, encontramos el resultado definitivo del proceso de análisis de requisitos, después de todos los cambios y ya dividido en dos aplicaciones.

6.1. Requisitos del programa *Monitor*

- **Selección y conexión de dispositivo:**
  a. Se deben reconocer todos los dispositivos Ixxat conectados al ordenador y mostrarlos en una lista de selección.
  b. Si la lista de selección no está vacía, el programa puede conectarse con uno solo de los dispositivos disponibles como máximo.
  c. Si hay un dispositivo conectado, debe poder desconectarse.
  d. Si no se detectan *drivers* de Ixxat, debe mostrarse un mensaje de error.
- **Monitorización:**
  a. Debe mostrarse por separado la monitorización de la batería y la de la ECU.
  b. **Batería:**
    b.a. Siempre que el programa se encuentre en una pantalla de batería, debe haber una cabecera que muestre el ID de la batería (BPID), la hora actual de la batería y la fecha de manufactura de la misma.
    b.b. En una pantalla principal deben mostrarse en forma numérica y en forma gráfica los valores a tiempo real del voltaje, la corriente, el estado de carga y las temperaturas registradas por los tres sensores de temperatura.
    b.c. En una pantalla deben mostrarse en forma numérica y en forma gráfica los valores a tiempo real de los voltajes individuales de las 14 celdas que componen la batería.
    b.d. El voltaje más pequeño y el más grande deben destacarse a la vista.
    b.e. Debe mostrarse la media de los 14 voltajes.

b.f. Debe mostrarse la diferencia entre el voltaje más grande y el más pequeño.

b.g. Debe mostrarse otro gráfico de barras de los 14 voltajes que sea a tiempo real hasta que estos sean mínimos. Cuando ningún voltaje es más pequeño que el anterior mostrado, este gráfico debe quedarse estático mostrando los valores más pequeños para cada celda. El gráfico tiene que poder volverse a mostrar a tiempo real cuando el usuario lo desee.

b.h. En una pantalla debe mostrarse un gráfico de líneas que muestre a tiempo real y superpuestos los valores de los voltajes de las 14 celdas, el estado de carga, la corriente y las tres temperaturas.

b.i. Estas magnitudes deben estar claramente diferenciadas en el gráfico mediante colores y estilos de línea distintos.

b.j. Cada magnitud debe ir claramente asociada a un eje Y de coordenadas con la escala que le corresponda.

b.k. Debe poder seleccionarse en todo momento cuáles de estas magnitudes se muestran.

b.l. El gráfico tiene que poder resetearse.

b.m. El gráfico tiene que poder guardarse como imagen.

b.n. En una pantalla debe mostrarse a tiempo real la información de los *flags* que envía la batería.

c. **ECU:**

c.a. Debe mostrarse a tiempo real la información de los *flags* que indican qué entradas de la ECU están activadas.

c.b. Debe poderse activar y desactivar los *flags* de salida de la ECU.

- **Escritura y lectura de parámetros:**

a. Debe mostrarse por separado la parametrización de la batería y la de la ECU.

b. Para cada dispositivo (batería y ECU) debe aparecer una lista con todos los nombres de los parámetros de la moto.

c. Esta lista se debe poder modificar mediante un archivo externo sin tener que cambiar el código del programa y junto a cada nombre de parámetro debe aparecer un campo de texto.

d. El usuario debe poder activar y desactivar la edición de estos campos de texto.

e. Debe poder haber parámetros de solo lectura que nunca permitan la edición del campo de texto correspondiente.

f. Debe mostrarse, si hubiese, las unidades del valor del parámetro, el valor mínimo permitido y el máximo.

g. En el caso de *flags*, se debe indicar si se expresa en binario o en hexadecimal.

h. Debe resaltarse un campo de texto si el valor introducido no es un valor válido.

i. Deben poderse cargar los valores de los parámetros desde un archivo.

j. Deben poderse guardar los valores que aparecen en los campos de texto en un archivo.

k. Deben leerse los parámetros que hay guardados en la moto y mostrarlos en la pantalla.

l. Si todos los valores que hay en los campos de texto son válidos, estos deben poderse escribir en la moto.

m. Debe leerse de la batería el ID de la batería (BPID) y, si es válido, mostrarlo.

n. Si el BPID leído no es válido, debe poderse escribir en la batería uno nuevo junto a la fecha actual como valor del parámetro *Manufacturing Date* (fecha de manufactura).

ñ. Pulsando cinco veces seguidas sobre un BPID ya válido, debe permitirse la escritura en la batería de uno nuevo.

o. Debe poderse escribir en la batería el parámetro *Current Time* (hora actual) con la fecha y hora actual del ordenador.

p. Debe poderse resetear los parámetros de la batería.

6.2. Requisitos del programa *Bootloader*

- **Selección y conexión de dispositivo:**

  a. Se deben reconocer todos los dispositivos Ixxat conectados al ordenador y mostrarlos en una lista de selección.

  b. Si la lista de selección no está vacía, el programa puede conectarse con uno solo de los dispositivos disponibles como máximo.

  c. Si hay un dispositivo conectado, debe poder desconectarse.

  d. Si no se detectan *drivers* de Ixxat, debe mostrarse un mensaje de error.

- **Escritura del archivo binario en memoria:**

  a. El usuario debe poder seleccionar entre escribir en la batería o escribir en la ECU.

  b. El usuario debe poder elegir la velocidad de transmisión de datos por CAN Bus.

  c. Se debe poder seleccionar un archivo .bin que contenga el programa que se quiere cargar en la moto.

  d. Al seleccionar un archivo .bin, se debe comprobar que tenga el BCA (Bootloader Configuration Area) configurado.

  e. Si el BCA no está configurado debe avisarse al usuario y preguntarle si desea continuar de todos modos.

  f. Cuando se ha seleccionado el dispositivo, la velocidad y el archivo, puede procederse con la escritura del archivo en la moto.

  g. Si hay algún error o la moto no responde, debe comunicarse al usuario y abortarse el proceso.

  h. Debe mostrarse el progreso a tiempo real de la escritura.

  i. Debe haber un campo de texto que muestre a tiempo real el log de eventos del proceso de escritura.

  j. El contenido de este campo de texto debe poder guardarse.

  k. Cuando finaliza con éxito la escritura y cuando se aborta el proceso, debe poder repetirse desde el principio.

7. Diseño del software

Para diseñar estas aplicaciones se ha empleado una arquitectura Modelo-Vista-Controlador, que permite separar la lógica de las comunicaciones y el tratamiento de los datos que hay que mostrar y enviar (modelo), la interfaz gráfica que permite que el usuario interaccione con el programa (vista) y el controlador, que responde a los eventos de la interacción del usuario para ejecutar alguna función de la lógica del modelo y actualiza la vista en función de lo que se ejecuta en el modelo.

7.1. Diseño del programa *Monitor*

El programa *Monitor* consta de las siguientes clases:

- **MainController**
  Es el controlador al que hace referencia el nombre de la arquitectura empleada. Encargada de la comunicación entre la interacción del usuario y la lógica de comunicaciones. Implementa la interfaz CommunicationStatusListener que proporciona la libería de CAN que permite escuchar el estado de las comunicaciones con los periféricos conectados por CAN.
  Métodos que implementa:
  - ***+ onCommAlive(): Unit***
    Ejecuta todo lo necesario cuando se establece comunicación con CAN.
  - ***+ onCommAsleep(): Unit***
    Ejecuta todo lo necesario cuando se deja de recibir mensajes por el bus.
  - ***+ initialize(): Unit***
    Inicializa el *thread* sobre el que se ejecutan las comunicaciones de CAN.
  - ***+ connectButtonClicked(): Unit***
    Ejecuta todo lo necesario cuando se pulsa el botón de conectar al dispositivo CAN.
  - ***- setCanControlsStatus(enable: Boolean): Unit***
    Actualiza el estado de la conexión con CAN.
  - ***- setCanDevices(): Unit***
    Envía la lista de dispositivos conectados por CAN a la vista para que los muestre.
  - ***- connectToDevice(deviceInfo: DeviceInfo): Unit***
    Se conecta al dispositivo CAN que tiene como parámetro.
  - ***- disconnect(): Unit***
    Se desconecta del dispositivo CAN al que esté conectado.
  - ***- selectFirstCanDeviceAvailable(): Unit***
    Si la lista de dispositivos disponibles no está vacía, selecciona el primero.
  - ***- setECUDebug(debug: Boolean): Unit***
    Manda al controlador de la ECU que le active el modo Debug.
  - ***+ getBPID(): Unit***
    Pide el BPID al controlador del BMS.
  - ***+ setSummaryPanel(value: String, item: Int): Unit***
    Pasa los datos obtenidos del BMS a la pantalla de gráficos.

- + ***setCellVoltage(vol: String, cell: Int)***
  Pasa los datos de voltajes del BMS a la pantalla de gráficos.

- **MainViewController**
  Es la clase que gestiona la ventana principal de la aplicación. Sobre ella se sitúan el panel de conexión, la cabecera y las distintas subventanas.
  Métodos que implementa:

  - + ***init***
    Inicializador. Añade el resto de vistas sobre la principal.
  - **-** ***addTabs(): Unit***
    Añade las pestañas para navegar por las subventanas.
  - + ***onDock(): Unit***
    Se ejecuta cuando se acopla la ventana al *Stage*.
  - **-** ***setCanControlsStatus(enable: Boolean): Unit***
    Se ejecuta cuando se desacopla la ventana del *Stage*.

- **ChartsPanelController**
  Es la clase que gestiona la ventana que contiene el gráfico de SOC, voltajes, corriente y temperaturas a tiempo real.
  Métodos que implementa:

  - + ***init***
    Inicializador. Añade el gráfico a la pantalla.
  - + ***newSoc(soc: String): Unit***
    Añade un nuevo valor de SOC al gráfico.
  - + ***newCurrent(current: String): Unit***
    Añade un nuevo valor de corriente al gráfico.
  - + ***newTemp(deg: String, temp: Int): Unit***
    Añade un nuevo valor de temperartura al gráfico.
  - + ***newCell(vol: String, cell: Int): Unit***
    Añade un nuevo valor de voltaje al gráfico.

- **FlagsPanelController**
  Es la clase que gestiona las ventanas que muestran flags y botones para probar las salidas de la ECU.
  Métodos que implementa:

  - + ***Init***
    Inicializador.
  - **-** ***addSections(): Unit***
    Añade las distintas secciones de información a la pantalla.
  - **-** ***readCML(): Unit***
    Lee el archivo XML que contiene los elementos que se deben mostrar.
  - + ***reset(): Unit***
    Resetea la información mostrada.

- **SummaryPanelCotroller**
  Es la clase que gestiona la ventana *Summary*, que es la que muestra la información principal del BMS.
  Métodos que implementa:

- + ***init***
  Inicializador. Añade los gráficos a la pantalla.
- + ***writePackVoltage(val: String): Unit***
  Muestra en el gráfico y por escrito el voltaje del pack de baterías actual.
- + ***writeSoc(val: String): Unit***
  Muestra en el gráfico y por escrito el SOC actual.
- + ***writeCurrent(val: String): Unit***
  Muestra en el gráfico y por escrito la corriente actual.
- + ***writeTempA(val: String): Unit***
  Muestra en el gráfico y por escrito la temperatura del sensor A actual.
- + ***writeTempB(val: String): Unit***
  Muestra en el gráfico y por escrito la temperatura del sensor B actual.
- + ***writeTempC(val: String): Unit***
  Muestra en el gráfico y por escrito la temperatura del sensor C actual.

- **CellVoltagesPanelController**
  Es la clase que gestiona la ventana que muestra los voltajes de cada celda por escrito y en dos gráficos, uno a tiempo real y otro que se congela cuando los voltajes son mínimos.
  Métodos que implementa:

  - + ***init***
    Inicializador. Añade los gráficos a la pantalla.
  - + ***writeCellVoltage(vol: String, cell: Int): Unit***
    Recibe el voltaje actual de una celda concreta y lo gestiona: lo muestra en los gráficos y por escrito y se recalcula la media y la desviación de los voltajes para mostrarlos también.
  - + ***copySeries(from: Series, to: Series): Unit***
    Copia el contenido de un gráfico en otro.

- **FactorySettingsPanelController**
  Es la clase que gestiona la ventana de parametrización.
  Métodos que implementa:

  - + ***init***
    Inicializador.
  - **- *prepareParametersForTransmission(read: Boolean): Unit***
    Se prepara el contenedor *parametersForTransmission* para ser usado para la transmisión de parámetros. Se diferencia si el contenedor va a ser usado para leer o escribir datos, lo cual se especifica con el parámetro booleano de la función.
  - **- *addSections(): Unit***
    Añade los parámetros disponibles divididos en secciones.
  - **- *readXML(dile: File): Unit***
    Lee el archivo XML que contiene los parámetros disponibles.
  - **- *enableEditing(): Unit***
    Activa la posiblidad de editar los cuadros de texto de los parámetros.
  - **- *disableEditing(): Unit***
    Desactiva la posiblidad de editar los cuadros de texto de los parámetros.

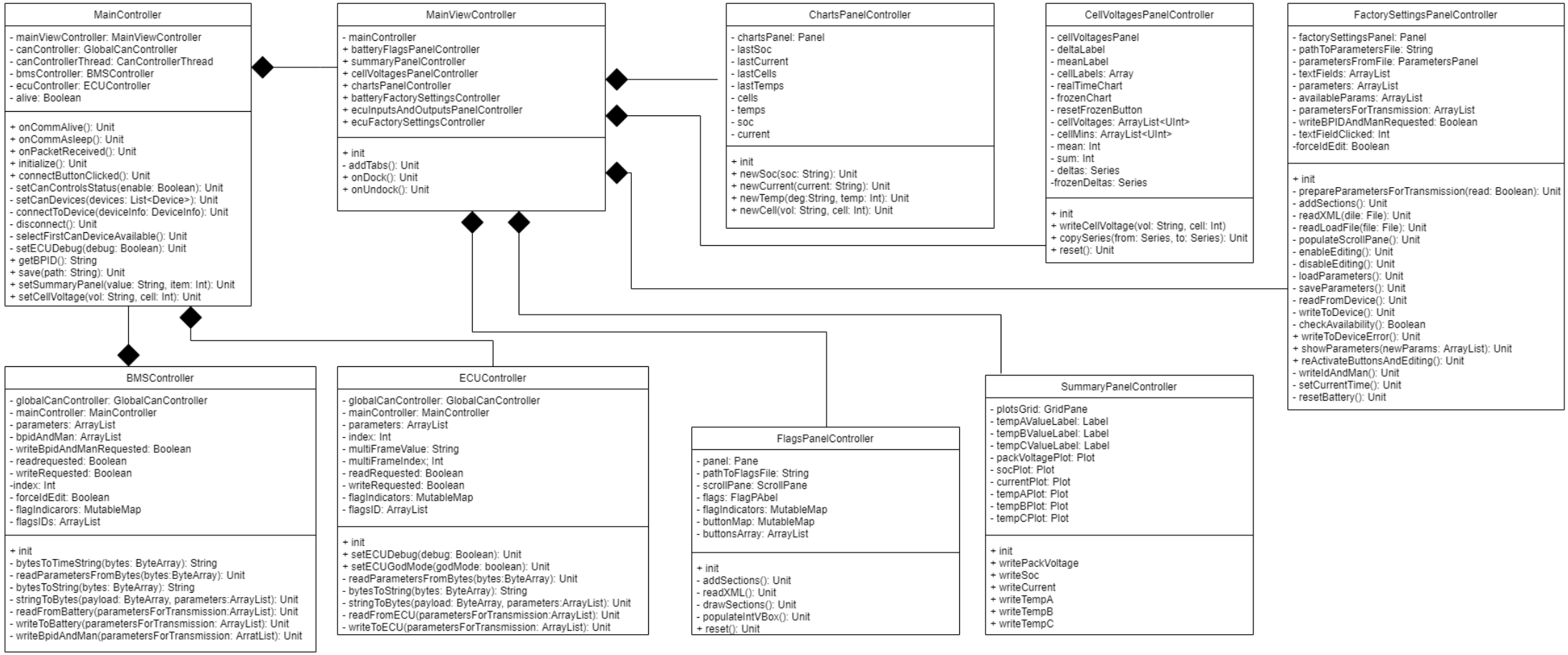
- ***- loadParameters(): Unit***
  Carga un fichero con valores de los parámetros.
- ***- saveParameters(): Unit***
  Guarda en un fichero los valores de los parámetros que hay en los cuadros de texto.
- ***- readFromDevice(): Unit***
  Pide leer los valores de los parámetros que hay guardados en el BMS o la ECU.
- ***- writeToDevice(): Unit***
  Manda a escribir los parámetros que hay en los cuadros de texto al BMS o la ECU.
- ***+ showParameters(newParams: ArrayList): Unit***
  Muestra en los cuadros de texto los valores de los parámetros que hay en *newParams*.
- ***- writeIdAndMan(): Unit***
  Manda a escribir el BPID y la fecha de fabricación al BMS.
- ***- setCurrentTime(): Unit***
  Manda a configurar la fecha y la hora actual al BMS.
- ***- resetBattery(): Unit***
  Manda a enviar un comando de *reset* al BMS.

- **BMSController**
  Esta clase gestiona todos los envíos y recepciones de paquetes con identificador perteneciente al rango del BMS.
  Métodos que implementa:

  - ***+ init***
    Inicializador. Define la función *listen* que recibirá todos los paquetes que envíe el BMS y los gestionará.
  - ***- bytesToTimeString(bytes: ByteArray): String***
    Devuelve el String que contiene la fecha y la hora que hay en el ByteArray *bytes*.
  - ***- readParametersFromBytes(bytes:ByteArray): Unit***
    Lee y guarda en el contenedor de transmisión de parámetros los parámetros que hay en el ByteArray *bytes*.
  - ***- bytesToString(bytes: ByteArray): String***
    Devuelve el String que contiene la fecha y la hora que hay en el ByteArray bytes.
  - ***- stringToBytes(payload: ByteArray, parameters:ArrayList): Unit***
    Coloca los valores de los parámetros que hay en *parameters* en el ByteArray *payload*.
  - ***- readFromBattery(parametersForTransmission:ArrayList): Unit***
    Solicita la lectura de los parámetros guardados en el BMS.
  - ***- writeToBattery(parametersForTransmission: ArrayList): Unit***
    Envía los valores de los parámetros que hay en *parametersForTransmission* para que se escriban en el BMS.
  - ***- writeBpidAndMan(parametersForTransmission: ArratList): Unit): String***
    Envía los valores de los parámetros BPID y fecha de fabricación que hay en *parametersForTransmission* para que se escriban en el BMS.

- **ECUController**
  Esta clase gestiona todos los envíos y recepciones de paquetes con identificador perteneciente al rango de la ECU.
  Métodos que implementa:

- **+ *init***
  Inicializador. Define la función *listen* que recibirá todos los paquetes que envíe la ECU y los gestionará.
- **+ *setECUDebug(debug: Boolean): Unit***
  Activa y desactiva el modo debug de la ECU según diga el parámetro booleano *debug*.
- **+ *setECUGodMode(godMode: boolean): Unit***
  Activa y desactiva el modo God de la ECU según diga el parámetro booleano *godMode*.
- **- *readParametersFromBytes(bytes:ByteArray): Unit***
  Lee y guarda en el contenedor de transmisión de parámetros los parámetros que hay en el ByteArray *bytes*.
- **- *bytesToString(bytes: ByteArray): String***
  Devuelve el String que contiene la fecha y la hora que hay en el ByteArray bytes.
- **- *stringToBytes(payload: ByteArray, parameters:ArrayList): Unit***
  Coloca los valores de los parámetros que hay en *parameters* en el ByteArray *payload*.
- **- *readFromECU(parametersForTransmission:ArrayList): Unit***
  Solicita la lectura de los parámetros guardados en la ECU.
- **- *writeToECU(parametersForTransmission: ArrayList): Unit***
  Envía los valores de los parámetros que hay en *parametersForTransmission* para que se escriban en la ECU.


MainController
- mainViewController: MainViewController
- canController: GlobalCanController
- canControllerThread: CanControllerThread
- bmsController: BMSController
- ecuController: ECUController
- alive: Boolean
+ onCommAlive(): Unit
+ onCommAsleep(): Unit
+ onPacketReceived(): Unit
+ initialize(): Unit
+ connectButtonClicked(): Unit
- setCanControlsStatus(enable: Boolean): Unit
- setCanDevices(devices: List<Device>): Unit
- connectToDevice(deviceInfo: DeviceInfo): Unit
- disconnect(): Unit
- selectFirstCanDeviceAvailable(): Unit
- setECUDebug(debug: Boolean): Unit
+ getBPID(): String
+ save(path: String): Unit
+ setSummaryPanel(value: String, item: Int): Unit
+ setCellVoltage(vol: String, cell: Int): Unit
MainViewController
- mainController
+ batteryFlagsPanelController
+ summaryPanelController
+ cellVoltagesPanelController
+ chartsPanelController
+ batteryFactorySettingsController
+ ecuInputsAndOutputsPanelController
+ ecuFactorySettingsController
+ init
- addTabs(): Unit
+ onDock(): Unit
+ onUndock(): Unit
ChartsPanelController
- chartsPanel: Panel
- lastSoc
- lastCurrent
- lastCells
- lastTemps
- cells
- temps
- soc
- current
+ init
+ newSoc(soc: String): Unit
+ newCurrent(current: String): Unit
+ newTemp(deg:String, temp: Int): Unit
+ newCell(vol: String, cell: Int): Unit
CellVoltagesPanelController
- cellVoltagesPanel
- deltaLabel
- meanLabel
- cellLabels: Array
- realTimeChart
- frozenChart
- resetFrozenButton
- cellVoltages: ArrayList<UInt>
- cellMins: ArrayList<UInt>
- mean: Int
- sum: Int
- deltas: Series
-frozenDeltas: Series
+ init
+ writeCellVoltage(vol: String, cell: Int)
+ copySeries(from: Series, to: Series): Unit
+ reset(): Unit
FactorySettingsPanelController
- factorySettingsPanel: Panel
- pathToParametersFile: String
- parametersFromFile: ParametersPanel
- textFields: ArrayList
- parameters: ArrayList
- availableParams: ArrayList
- parametersForTransmission: ArrayList
- writeBPIDAndManRequested: Boolean
- textFieldClicked: Int
-forceIdEdit: Boolean
+ init
- prepareParametersForTransmission(read: Boolean): Unit
- addSections(): Unit
- readXML(dile: File): Unit
- readLoadFile(file: File): Unit
- populateScrollPane(): Unit
- enableEditing(): Unit
- disableEditing(): Unit
- loadParameters(): Unit
- saveParameters(): Unit
- readFromDevice(): Unit
- writeToDevice(): Unit
- checkAvailability(): Boolean
+ writeToDeviceError(): Unit
+ showParameters(newParams: ArrayList): Unit
+ reActivateButtonsAndEditing(): Unit
- writeIdAndMan(): Unit
- setCurrentTime(): Unit
- resetBattery(): Unit
BMSController
- globalCanController: GlobalCanController
- mainController: MainController
- parameters: ArrayList
- bpidAndMan: ArrayList
- writeBpidAndManRequested: Boolean
- readrequested: Boolean
- writeRequested: Boolean
-index: Int
- forceIdEdit: Boolean
- flagIndicarors: MutableMap
- flagsIDs: ArrayList
+ init
- bytesToTimeString(bytes: ByteArray): String
- readParametersFromBytes(bytes:ByteArray): Unit
- bytesToString(bytes: ByteArray): String
- stringToBytes(payload: ByteArray, parameters:ArrayList): Unit
- readFromBattery(parametersForTransmission:ArrayList): Unit
- writeToBattery(parametersForTransmission: ArrayList): Unit
- writeBpidAndMan(parametersForTransmission: ArratList): Unit
ECUController
- globalCanController: GlobalCanController
- mainController: MainController
- parameters: ArrayList
- index: Int
- multiFrameValue: String
- multiFrameIndex; Int
- readRequested: Boolean
- writeRequested: Boolean
- flagIndicators: MutableMap
- flagsID: ArrayList
+ init
+ setECUDebug(debug: Boolean): Unit
+ setECUGodMode(godMode: boolean): Unit
- readParametersFromBytes(bytes:ByteArray): Unit
- bytesToString(bytes: ByteArray): String
- stringToBytes(payload: ByteArray, parameters:ArrayList): Unit
- readFromECU(parametersForTransmission:ArrayList): Unit
- writeToECU(parametersForTransmission: ArrayList): Unit
FlagsPanelController
- panel: Pane
- pathToFlagsFile: String
- scrollPane: ScrollPane
- flags: FlagPAbel
- flagIndicators: MutableMap
- buttonMap: MutableMap
- buttonsArray: ArrayList
+ init
- addSections(): Unit
- readXML(): Unit
- drawSections(): Unit
- populateIntVBox(): Unit
+ reset(): Unit
SummaryPanelController
- plotsGrid: GridPane
- tempAValueLabel: Label
- tempBValueLabel: Label
- tempCValueLabel: Label
- packVoltagePlot: Plot
- socPlot: Plot
- currentPlot: Plot
- tempAPlot: Plot
- tempBPlot: Plot
- tempCPlot: Plot
+ init
+ writePackVoltage
+ writeSoc
+ writeCurrent
+ writeTempA
+ writeTempB
+ writeTempC

7.2. Diseño del programa *Bootloader*

El programa *Bootloader* consta de las siguientes clases:

- **MainController**
  Es el controlador al que hace referencia el nombre de la arquitectura empleada. Es la clase que se encarga de la comunicación entre la interacción del usuario y la lógica de comunicaciones.
  Métodos que implementa:
  - ***+ initialize(): Unit***
    Inicializa el *thread* sobre el que se ejecutan las comunicaciones de CAN.
  - ***+ connectButtonClicked(): Unit***
    Ejecuta todo lo necesario cuando se pulsa el botón de conectar al dispositivo CAN.
  - ***- setCanControlsStatus(enable: Boolean): Unit***
    Actualiza el estado de la conexión con CAN.
  - ***- setCanDevices(): Unit***
    Envía la lista de dispositivos conectados por CAN a la vista para que los muestre.
  - ***- connectToDevice(deviceInfo: DeviceInfo): Unit***
    Se conecta al dispositivo CAN que tiene como parámetro.
  - ***- disconnect(): Unit***
    Se desconecta del dispositivo CAN al que esté conectado.
  - ***- selectFirstCanDeviceAvailable(): Unit***
    Si la lista de dispositivos disponibles no está vacía, selecciona el primero.
  - ***+ setFile(file: File): Unit***
    Establece el fichero *file* como el fichero .bin que se quiere escribir en el BMS o la ECU.
  - ***+ start(device: String): Unit***
    Empieza el protocolo de escritura del archivo.
  - ***+ error(type:String): Unit***
    Muestra el error que ha ocurrido y detiene el protocolo de escritura.
  - ***+ checkItem(item:String): Unit***
    Manda a la vista que muestre que la tarea *item* ya se ha completado.
  - ***+ loadingItem(item: String): Unit***
    Manda a la vista que muestre que la tarea *item* ya está en proceso.
  - ***+ end(): Unit***
    Ejecuta todo lo necesario al finalizar el protocolo de escritura.
  - ***+ writeInfo(text: String): Unit***
    Manda a la vista que muestre la información *info* en el cuadro de texto.
  - ***+ updateProgress(progress: Double): Unit***
    Manda a la vista que actualice la barra de progreso con el nuevo valor *progress*.
  - ***+ writeLogs(): Unit***
    Guarda en ficheros los logs de eventos y posibles errores producidos durante la ejecución del protocolo de escritura.

- **MainViewController**
  Es la clase que gestiona la ventana principal de la aplicación. Sobre ella se sitúan el panel de conexión, la cabecera y el panel que muestra el progreso del protocolo de escritura.
  Métodos que implementa:
  - + ***init***
    Inicializador. Añade el resto de paneles sobre la ventana principal.
  - + ***onDock(): Unit***
    Se ejecuta cuando se acopla la ventana al *Stage*.
  - **- *setCanControlsStatus(enable: Boolean): Unit***
    Se ejecuta cuando se desacopla la ventana del *Stage*.
- **BootloaderPanelController**
  Es la clase que gestiona la ventana que contiene el gráfico de SOC, voltajes, corriente y temperaturas a tiempo real.
  Métodos que implementa:
  - + ***init***
    Inicializador.
  - + ***writeInfo(str: String): Unit***
    Escribe el string *str* en el cuadro de texto.
  - + ***updateStatus(status: String, item: String): Unit***
    Pone el icono *status* al lado del paso del protocolo *item* en la *checklist* de progreso. Status puede ser *completado*, *error* o *en progreso*.
  - + ***resetInfoText(): Unit***
    Borra el texto que hay en el cuadro de texto.
  - + ***resetCheckList(resetBCA: Boolean):Unit***
    Quita los iconos de al lado de los pasos del protocolo en la *checklist* de progreso. El del paso *BCA Check* solo lo quita si el parámetro *resetBCA* es cierto.
- **HeaderGridController**
  Es la clase que gestiona la cabecera de la ventana en la que se selecciona el archivo .bin que se quiere enviar.
  Métodos que implementa:
  - + ***Init***
    Inicializador.
  - **- *selectFile(): Unit***
    Abre el explorador de archivos y asigna al atributo *binFile* el archivo que se haya seleccionado.
  - **- *startButtonPessed(): Unit***
    Se ejecuta todo lo necesario para empezar el protocolo de escritura.
- **ConnectionPanelController**
  Es la clase que gestiona el panel de conexión con CAN. Es donde el usuario selecciona el dispostivo que quiere utilizar de entre los disponibles.
  - + ***init***
    Inicializador.

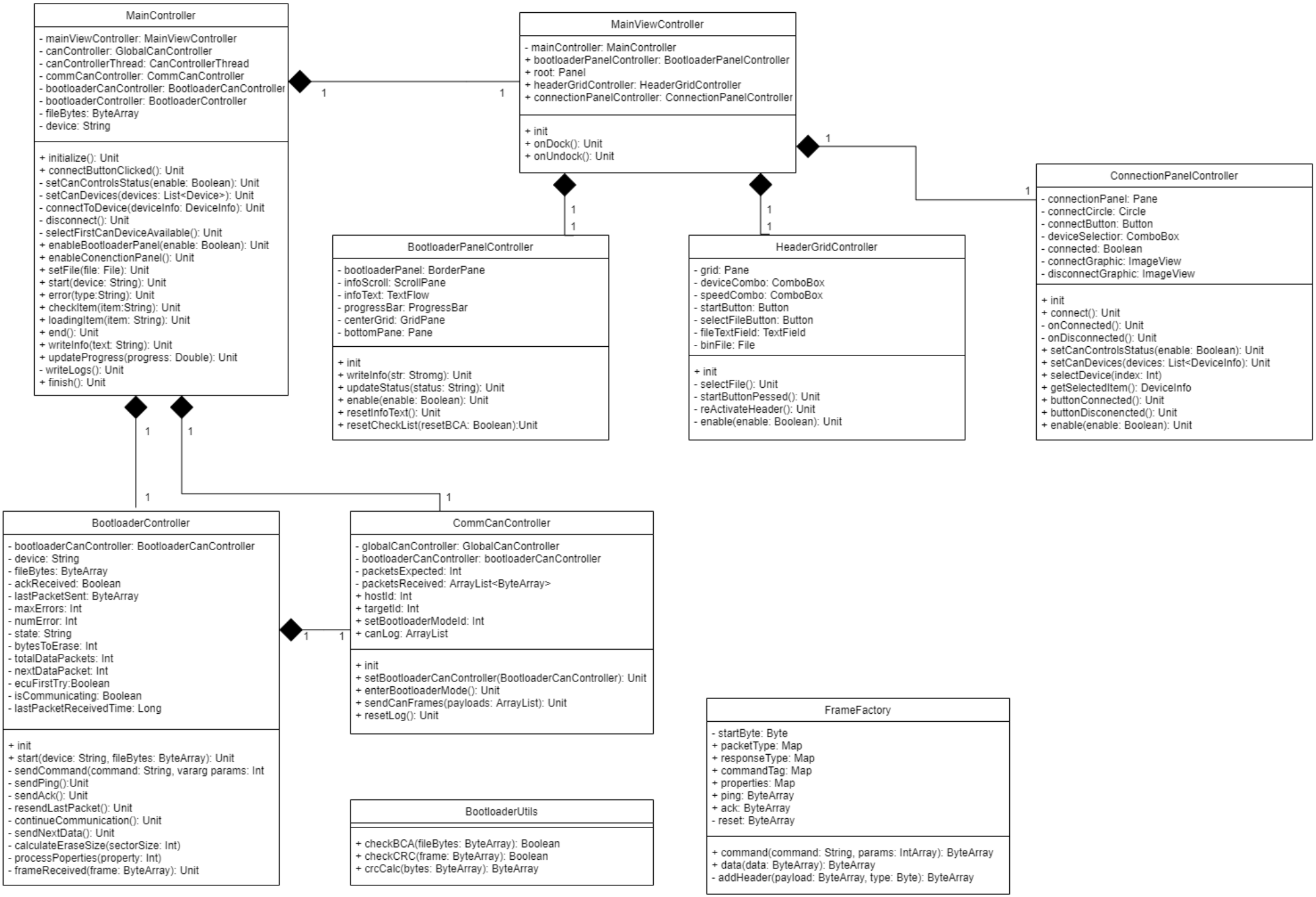
- ***+ connect(): Unit***
  Conecta el dispositivo seleccionado.
- ***+ onConnected(): Unit***
  Ejecuta todo lo que es necesario cuando se ha conectado un dispositivo.
- ***+ onDisconnected(): Unit***
  Ejecuta todo lo que es necesario cuando se ha desconectado un dispositivo.
- ***+ setCanControlsStatus(enable: Boolean): Unit***
  Pinta el indicador de estado de conexión de verde o de gris dependiendo de si el parámetro *enable* es cierto o falso, respectivamente.
- ***+ setCanDevices(devices: List¡DeviceInfo): Unit***
  Pone todos los dispositivos de la lista *devices* en el selector desplegable de dispositivo.
- ***+ getSelectedItem(): DeviceInfo***
  Devuelve el dispositivo que hay seleccionado en el selector desplegable.

- **BootloaderController**
  Es la clase que implementa toda la lógica del protocolo de escritura de archivos .bin en el BMS y en la ECU mediante el *bootloader* de sus microcontroladores.

  Métodos que implementa:

  - ***+ init***
    Inicializador. Configura el temporizador de *timeout*.
  - ***+ start(device: String, fileBytes: ByteArray): Unit): Unit***
    Empieza el protocolo de la escritura del archivo *fileBytes* en el dispositivo *device* (BMS o ECU).
  - ***- sendCommand(command: String, vararg params: Int***
    Envía un comando de tipo *command* y los parámetros *params*.
  - ***- sendPing(): Unit***
    Envía una trama de tipo ping.
  - ***- sendAck(): Unit***
    Envía una trama de tipo ack.
  - ***- resendLastPacket(): Unit***
    Reenvía el último paquete enviado.
  - ***- continueCommunication(): Unit***
    Avanza al siguiente paso del protocolo.
  - ***- sendNextData(): Unit***
    Envía el siguiente paquete de datos.
  - ***- calculateEraseSize(sectorSize: Int): Unit***
    Calcula cuántos sectores de memoria flash hay que borrar para que quepa el nuevo archivo .bin teniendo en cuenta que cada sector mide *sectorSize*.
  - ***- processPoperties(property: Int)***
    Procesa la propiedad *property* en función delestado del protocolo en que se encuentre la ejecución.
  - ***- frameReceived(frame: ByteArray): Unit***
    Procesa la trama *frame* que ha enviado el BMS o la ECU.

- **CommCanController**
  Es la clase que gestiona la clase que gestiona todos los envíos y recepciones de paquetes de CAN con identificador correspondiente a *bootloader* de BMS o ECU.
  Métodos que implementa:
  - **+ *init***
    Inicializador. Define la función *listen* que recibirá todos los paquetes que envíe el BMS o la ECU y los gestionará.
  - **- *enterBootloaderMode(): Unit***
    Envía al BMS o a la ECU la instrucción que les hace ejecutar el *bootloader* de la ROM de su microcontrolador.
  - **- *sendCanFrames(payloads: ArrayList): Unit***
    Envía al dispositivo los paquetes de CAN formados por los payloads de la lista.
- **FrameFactory**
  Clase estática que genera las tramas de *bootloader*.
  Métodos que implementa:
  - **+ *command(command: String, params: IntArray): ByteArray***
    Devuelve una trama de comando de tipo *command* y parámetros *params*.
  - **+ *data(data: ByteArray): ByteArray***
    Devuelve una trama de datos con los datos *data*.
  - **+ *addHeader(payload: ByteArray, type: Byte): ByteArray***
    Calcula la cabecera correspondiente a una trama de tipo *type* y payload *payload* y la devuelve entera.
- **BootloaderUtils**
  Clase estática que implementa métodos que son de utilidad para el resto de clases que gestionan el protocolo *bootloader*.
  Métodos que implementa:
  - **+ *checkBCA(fileBytes: ByteArray): Boolean***
    Devuelve falso si la zona del ByteArray *fileBytes* reservada al BCA tiene todos sus bits a 1 (es decir, no está configurado), cierto en cualquier otro caso.
  - **+ *checkCRC(frame: ByteArray): Boolean***
    Devuelve cierto si el CRC que contiene la cabecera la trama *frame* es el que se calcula de esa trama, falso si no lo es.
  - **+ *crcCalc(bytes: ByteArray): ByteArray***
    Devuelve el CRC que se calcula de la trama representada por *bytes*.


MainController
- mainViewController: MainViewController
- canController: GlobalCanController
- canControllerThread: CanControllerThread
- commCanController: CommCanController
- bootloaderCanController: BootloaderCanController
- bootloaderController: BootloaderController
- fileBytes: ByteArray
- device: String
+ initialize(): Unit
+ connectButtonClicked(): Unit
- setCanControlsStatus(enable: Boolean): Unit
- setCanDevices(devices: List<Device>): Unit
- connectToDevice(deviceInfo: DeviceInfo): Unit
- disconnect(): Unit
- selectFirstCanDeviceAvailable(): Unit
+ enableBootloaderPanel(enable: Boolean): Unit
+ enableConenctionPanel(): Unit
+ setFile(file: File): Unit
+ start(device: String): Unit
+ error(type:String): Unit
+ checkItem(item:String): Unit
+ loadingItem(item: String): Unit
+ end(): Unit
+ writeInfo(text: String): Unit
+ updateProgress(progress: Double): Unit
- writeLogs(): Unit
+ finish(): Unit
MainViewController
- mainController: MainController
+ bootloaderPanelController: BootloaderPanelController
+ root: Panel
+ headerGridController: HeaderGridController
+ connectionPanelController: ConnectionPanelController
+ init
+ onDock(): Unit
+ onUndock(): Unit
BootloaderPanelController
- bootloaderPanel: BorderPane
- infoScroll: ScrollPane
- infoText: TextFlow
- progressBar: ProgressBar
- centerGrid: GridPane
- bottomPane: Pane
+ init
+ writeInfo(str: Stromg): Unit
+ updateStatus(status: String): Unit
+ enable(enable: Boolean): Unit
+ resetInfoText(): Unit
+ resetCheckList(resetBCA: Boolean):Unit
HeaderGridController
- grid: Pane
- deviceCombo: ComboBox
- speedCombo: ComboBox
- startButton: Button
- selectFileButton: Button
- fileTextField: TextField
- binFile: File
+ init
- selectFile(): Unit
- startButtonPessed(): Unit
- reActivateHeader(): Unit
- enable(enable: Boolean): Unit
ConnectionPanelController
- connectionPanel: Pane
- connectCircle: Circle
- connectButton: Button
- deviceSelectior: ComboBox
- connected: Boolean
- connectGraphic: ImageView
- disconnectGraphic: ImageView
+ init
+ connect(): Unit
- onConnected(): Unit
- onDisconnected(): Unit
+ setCanControlsStatus(enable: Boolean): Unit
+ setCanDevices(devices: List<DeviceInfo): Unit
+ selectDevice(index: Int)
+ getSelectedItem(): DeviceInfo
+ buttonConnected(): Unit
+ buttonDisconencted(): Unit
+ enable(enable: Boolean): Unit
BootloaderController
- bootloaderCanController: BootloaderCanController
- device: String
- fileBytes: ByteArray
- ackReceived: Boolean
- lastPacketSent: ByteArray
- maxErrors: Int
- numError: Int
- state: String
- bytesToErase: Int
- totalDataPackets: Int
- nextDataPacket: Int
- ecuFirstTry:Boolean
- isCommunicating: Boolean
- lastPacketReceivedTime: Long
+ init
+ start(device: String, fileBytes: ByteArray): Unit
- sendCommand(command: String, vararg params: Int
- sendPing():Unit
- sendAck(): Unit
- resendLastPacket(): Unit
- continueCommunication(): Unit
- sendNextData(): Unit
- calculateEraseSize(sectorSize: Int)
- processPoperties(property: Int)
- frameReceived(frame: ByteArray): Unit
CommCanController
- globalCanController: GlobalCanController
- bootloaderCanController: bootloaderCanController
- packetsExpected: Int
- packetsReceived: ArrayList<ByteArray>
+ hostId: Int
+ targetId: Int
+ setBootloaderModeId: Int
+ canLog: ArrayList
+ init
+ setBootloaderCanController(BootloaderCanController): Unit
+ enterBootloaderMode(): Unit
+ sendCanFrames(payloads: ArrayList): Unit
+ resetLog(): Unit
BootloaderUtils
+ checkBCA(fileBytes: ByteArray): Boolean
+ checkCRC(frame: ByteArray): Boolean
+ crcCalc(bytes: ByteArray): ByteArray
FrameFactory
- startByte: Byte
+ packetType: Map
+ responseType: Map
+ commandTag: Map
+ properties: Map
+ ping: ByteArray
+ ack: ByteArray
- reset: ByteArray
+ command(command: String, params: IntArray): ByteArray
+ data(data: ByteArray): ByteArray
- addHeader(payload: ByteArray, type: Byte): ByteArray

8. Implementación

En este capítulo veremos cómo se han implementado algunas de las funciones de las descritas en el capítulo anterior. Para realizar esta selección, se ha tenido en cuenta la complejidad de las funciones y que tengan relación con los conocimientos específicos estudiados para este proyecto: los dos protocolos de comunicación y el componente Human-Machine Interfece.

8.1. Implementación del programa *Monitor*

Este programa gestiona muchos datos a tiempo real y los muestra de forma visual por la pantalla mediante gráficos. Dos de esos gráficos son los que aparecen en la pestaña *Cell voltages*, que muestran la desviación del voltaje de cada celda de la batería respecto a la media del pack completo. Uno de ellos muestra los valores a tiempo real. El otro lo hace también a tiempo real hasta que el siguiente valor que llega es superior al actual (es decir, se ha llegado al mínimo); enntonces la barra correspondiente a esa celda no se modifica más. Este segundo gráfico tiene un botón para resetear la congelación y volver a mostrar datos a tiempo real hasta que se encuentren nuevos mínimos.

Veamos como gestiona un nuevo valor de voltaje la función *writeCellVoltage(vol: String, cell: Int)* del controlador de la vista de esa pestaña:

```
fun writeCellVoltage(vol: String, cell: Int){
    cellLabels[cell-1].text = vol        //Se escribe por pantall el nuevo voltaje
    sum -= cellVoltages[cell-1]          //Se resta el anterior voltaje de la suma total
    cellVoltages[cell-1] = vol.toUInt()  //Se guarda el nuevo voltaje
    sum += cellVoltages[cell-1]          //Se suma el nuevo voltaje a la suma total
    mean = (sum.toInt().toFloat() / 14 + 0.5).toInt()   //Se calcula la nueva media
    minValue = 10000.toUInt()
    maxValue = 0.toUInt()
    cellVoltages.forEach{ cellVol ->          //Se busca el mínimo y el máximo de los voltajes
        if(cellVol < minValue) minValue = cellVol
        if(cellVol > maxValue) maxValue = cellVol
    }
    cellVoltages.forEachIndexed{ i, cellVol ->      //Cambia el color del texto en función de si el
        when (cellVol) {                            //nuevo voltaje es mínimo, máximo o ninguna de las dos
            maxValue -> cellLabels[i].textFill = Paint.valueOf("green")
            minValue -> cellLabels[i].textFill = Paint.valueOf("red")
            else -> cellLabels[i].textFill = Paint.valueOf("black")
        }
    }
    deltaLabel.text = (maxValue - minValue).toString()
    meanLabel.text = mean.toString()
    deltas.data.clear()
    cellVoltages.forEachIndexed{i, cellVol ->          //Actualiza el gráfico a tiempo real
        deltas.data.add(XYChart.Data<String, Number>((i+1).toString(), cellVol.toInt() - mean))
    }
    if(vol.toUInt() <= cellMins[cell-1]){              // Si no es el mínimo, actualiza el segundo gráfico
        cellMins[cell-1] = vol.toUInt()
        copySeries(deltas, frozenDeltas)
    }
}
```

Figura 25: Código de la función *writeCellVoltage(vol: String, cell: Int)*.

Cuando el controlador principal recibe un nuevo valor voltaje por parte del controlador de comunicación con el BMS, llama a la función con los parámetros *vol* (valor del nuevo voltaje) y *cell* (celda a la que pertenece, va de 1 a 14). El método hace lo siguiente:

- Escribe por pantalla el nuevo valor.
- Se resta el anterior voltaje de esa celda de la suma total.
- Se guarda el nuevo voltaje.
- Se suma el nuevo voltaje a la suma total.
- Se calcula la nueva media.
- Se busca el mínimo y el máximo de los voltajes
- Se compara el mínimo y el máximo con el nuevo voltaje. Si este pasa a ser el nuevo mínimo o el nuevo máximo, se resalta con el color adecuado.
- Se escribe por pantalla la nueva diferencia entre mínimo y máximo y la nueva media
- Se actualiza el primer gráfico y, si el valor es inferior al mínimo de esa celda, el segundo y, en el vector de valores mínimos, el correspondiente a esa celda esa celda.

Ese nuevo voltaje, así como muchos datos más, son enviados por la moto constantemente mediante el bus y la librería de CAN se encarga de recogerlos, decidir si son relevantes y, si lo son, para el controlador de qué dispositivo. Para ello, utiliza un objeto de la clase *PublishSubject* de la librería *RxJava*. A este objeto se suscriben los controladores de comunicación con el BMS y con la ECU, los cuales deben implementar su propia función *listen* que gestiona los mensajes que van llegando.

Esta función, en el caso del controlador de la ECU, implementa dos tipos de comportamiento:

- Cuando el identificador pertenece al rango de identificadores de los *flags* de *inputs*:
  Primero, se clona el payload en una variable local para evitar que se interfiera con los datos. Después mediante una máscara que se va desplazando, se comprueba para cada bit si vale 1 o 0. Si vale 1, se activa el indicador de *flag*, si vale 0, se desactiva.
- Cuando el identificador corresponde a la respuesta a una demanda de lectura de parámetros:
  Primero, se clona el payload en una variable local para evitar que se interfiera con los datos. Después se comprueba si realmente se ha enviado una demanda de lectura de parámetros. Si es así, los parámetros del payload se convierten a string y se almacenan en el contenedor *parametersForTransmission* para ser enviados al controlador principal. Después se solicita el siguiente paquete de parámetros si lo hubiese.

```
globalCanController.listen(ECUCanPacket::class.java).subscribe{
    when(it.id){
        in flagsIDs -> {
            val bytes = it.payload.clone()
            val packetId = it.id
            flagIndicators[packetId]?.forEach { (id, byteMap) ->
                var mask = 0x01.toByte()
                for (j in 0.until(8)) {
                    byteMap[j].setActive(bytes[id].and(mask) == mask)
                    mask = mask.toInt().shl( bitCount: 1).toByte()
                }
            }
        }
        0x31C -> {
            val bytes = it.payload.clone()
            if((writeRequested || readRequested) && ((index+multiFrameIndex-1).toByte() == bytes[0])){
                readParametersFromBytes(bytes)
                val numFrames = parameters[index][0].numFrames
                val payload = ByteArray( size: 8)
                if (parameters[index][0].type == "ASCII") {
                    if (multiFrameIndex < numFrames) {
                        payload[0] = (index + multiFrameIndex).toByte()
                        ++multiFrameIndex
                    } else {
                        multiFrameIndex = 1
                        parameters[index][0].value = multiFrameValue
                        multiFrameValue = ""
                        ++index
                        while (index < 0xFF && parameters[index].isEmpty()) ++index
                        payload[0] = index.toByte()
                    }
                } else {
                    ++index
                    while (index < 0xFF && parameters[index].isEmpty()) ++index
                    payload[0] = index.toByte()
                }

                if (index <  0xFF) {
                    var id = 0x31B
                    if(writeRequested){
                        stringsToBytes(payload, parameters[index],  auxIndex: multiFrameIndex-1)
                        id = 0x31A
                    }
                    val packet = BMSCanPacket(id, payload)
                    globalCanController.transmitPacket(packet)
                } else{
                    readRequested = false
                    writeRequested = false
                    Platform.runLater {
                        mainController.showParameters(parameters, "ECU")
                         ^runLater mainController.reActivateButtonsAndEditing("ECU")
                    }
                }
            }
        }
    }
}
```

Figura 26: Código de la implementación de la función *listen* de *ECUController*.

8.2. Implementación del programa *Bootloader*

En el programa *Bootloader*, toda la comunicación con CAN funciona exactamente igual que en el *Monitor* y la única vista es sumamente sencilla; así que en este apartado nos centraremos únicamente en la implementación del protocolo de escritura de archivos .bin en el BMS y la ECU mediante el *bootloader* de la ROM de sus microcontroladores.

Veamos detalladamente el funcionamiento del protocolo:

- **Ping**
  En primer lugar, el *Host* (PC) manda un ping con el identificador correspondiente al dispositivo (BMS o ECU). En el caso de la ECU, dependiendo de qué versión del *firmware* tengan, este identificador puede ser el 0x666 o el 0x321. El ping se utiliza también para determinar cuál de estos identificadores se debe usar en el resto de la comunicación. Si no se obtiene pingResponse con el primero, debe ser el segundo.

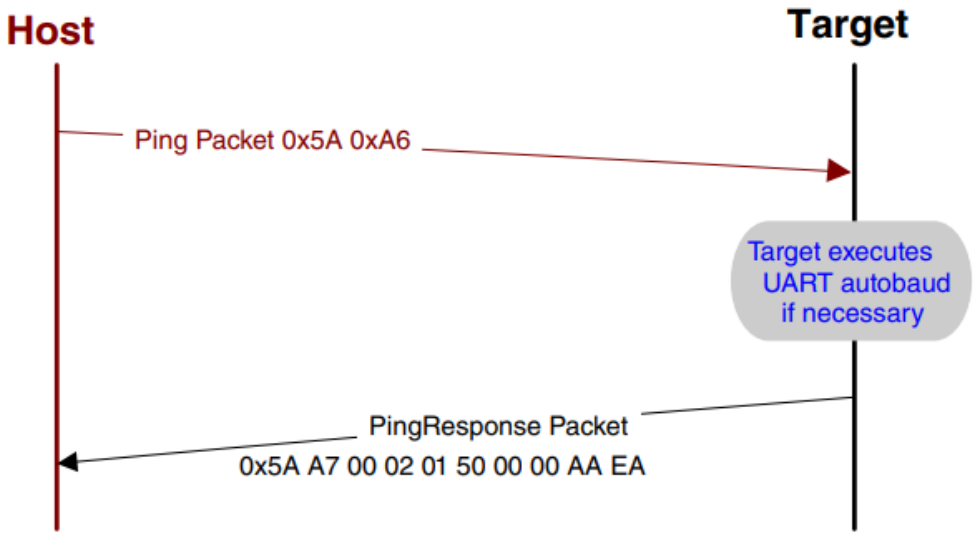


Figura 27: Secuencia de ping.

- **Secured**
  Una vez establecida la comunicación entre el *Host* y el *Target* BMS o ECU), hay que comprobar si la memoria flash tiene activada la seguridad o no. De tenerla activada, no podríamos borrar ni escribir. Para hacer esta comprobación, mediante un comando de tipo getPropery, pedimos la propiedad *flashSecurityState*.

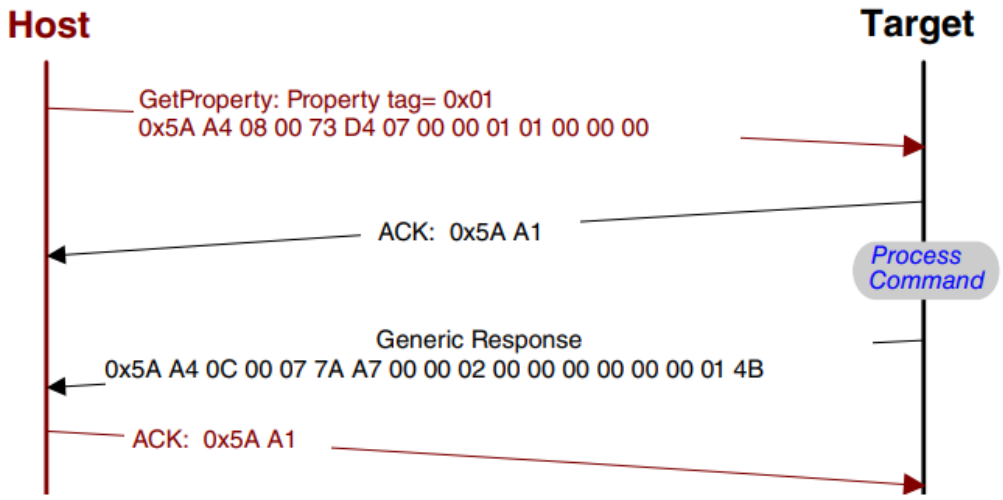


Figura 28: Secuencia del comando getProperty.

- **Size**
  Si la seguridad de la flash está desactivada, se procede con la comprobación de que el archivo que queremos cargar cabe en la memoria. Para ello se pide la propiedad *flashSizeInBytes* y se compara con el tamaño del archivo.

- **Sector size**
  A continuación, se calcula cuántos sectores de la flash hará falta borrar ya que no se puede borrar a nivel de bytes. Primero, se pide la propiedad *flashSectorSize*.

- **Erase**
  Después, se calcula el mínimo número de sectores que hace falta borrar para que quepa todo el archivo. Este mínimo es el entero más próximo por exceso al resultado de *tamañoFichero*/*tamañoSector*. Se envía un comando de borrado del mínimo de sectores de flash desde la dirección de memoria 0.

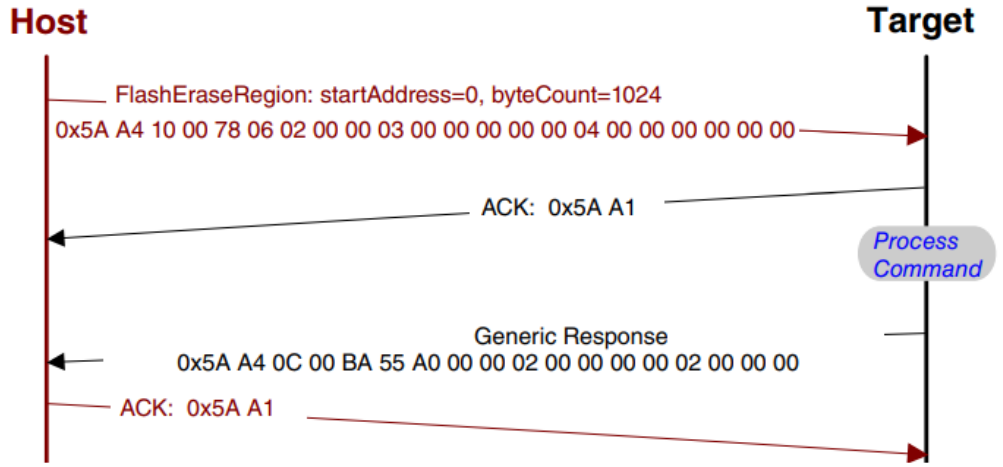


Figura 29: Secuencia del comando eraseRegion.

- **Write Memory Command / Write**
  Ahora ya está todo listo para escribir. Se divide el archivo .bin en paquetes de 32 bytes y se envían todos siguiendo esta secuencia:

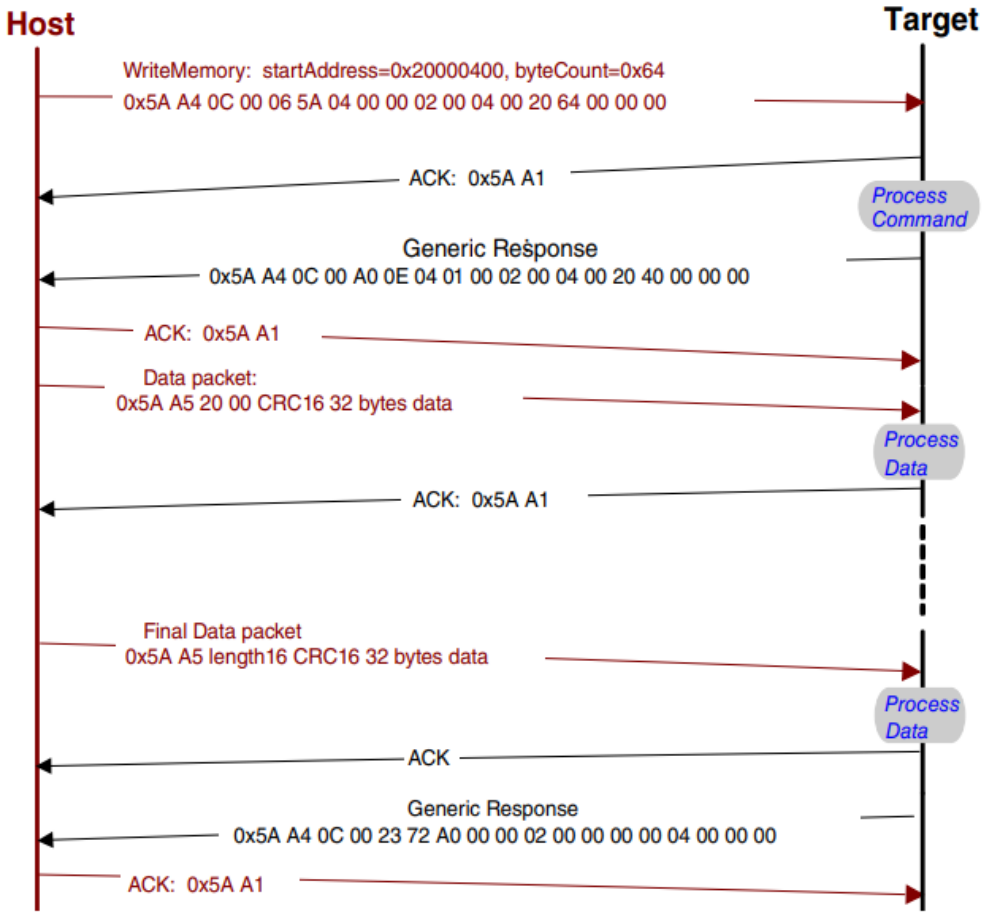


Figura 30: Secuencia de escritura de datos en la memoria.

- **Reset and End**
  Cuando ya se han enviado todos los datos, hay que enviar un comando de tipo reset para avisar al *Target* que el proceso ha acabado y puede reiniciarse con el nuevo *firmware*.

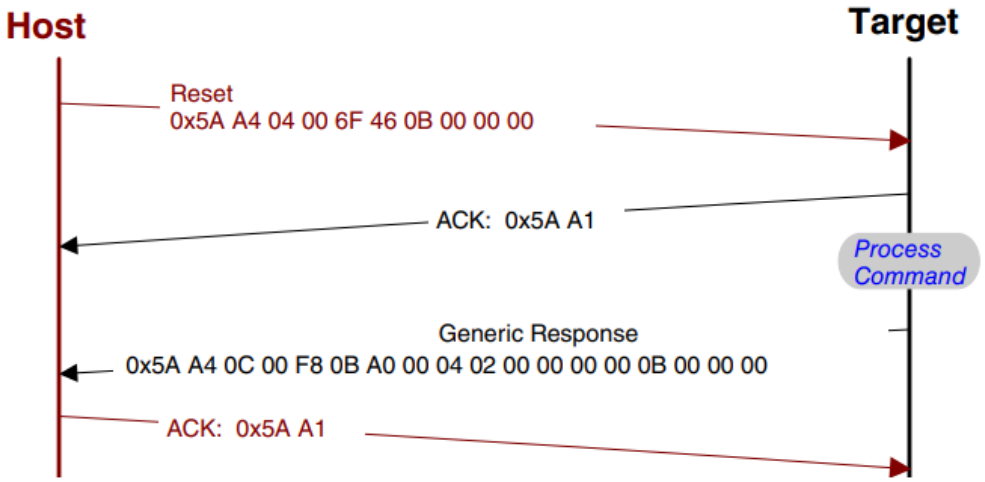


Figura 31: Secuencia del comando de reset.

Para implementar los pasos de este protocolo (*Write Memory Command* y *Write* son dos

pasos distintos), se ha considerado que cada paso es un estado en el que puede estar la ejecución. Cuando el *Host* reciba un paquete, este se procesará distinto en función de en qué estado nos encontremos. Una vez se ha procesado este paquete, se ejecuta la función *continueCommunication()*.

```
private fun continueCommunication(){
    if(state != "sector size" && state != "resetAndEnd" && state != "write memory command") mainController.checkItem(state)
    state = when(state){
        "ping"              -> {
            sendCommand( command: "getProperty", FrameFactory.properties["flashSecurityState"]!!)
            "secured"
        }
        "secured"           -> {
            sendCommand( command: "getProperty", FrameFactory.properties["flashSizeInBytes"]!!)
            "size"
        }
        "size"              -> {
            sendCommand( command: "getProperty", FrameFactory.properties["flashSectorSize"]!!)
            "sector size"
        }
        "sector size"   -> {
            sendCommand( command: "eraseRegion",  ...params: 0, bytesToErase)
            "erase"
            //sendCommand("reset")
            //"reset"
        }
        "erase"             -> {
            sendCommand( command: "write",  ...params: 0, fileBytes.size)
            "write memory command"
        }
        "write memory command"  -> {
            totalDataPackets = fileBytes.size / 32
            if(fileBytes.size % 32 != 0) ++totalDataPackets
            nextDataPacket = 0
            Platform.runLater{
                mainController.updateProgress( progress: 0.0)
                mainController.loadingItem( item: "write")
            }
            sendNextData()
            "write"
        }
        "write"             -> {
            sendCommand( command: "reset")
            "reset"
        }
        "resetAndEnd"    -> {
            sendCommand( command: "reset")
            "reset"
        }
        else                -> {
            isCommunicating = false
            mainController.end()
            ""
        }
    }
}
```

Figura 32: Código de la función *continueCommunication()*.

Esta función es la que guía la ejecución por los pasos ordenadamente. Se ejecuta cada vez que se ha procesado una respuesta del *Target*. Esta función realiza tres tareas:

- Manda a actualizar el *checklist* de progreso.
- Envía la siguiente trama o finaliza el protocolo en el caso del último estado.
- Asigna al atributo *state* el nuevo estado al que avanzar.

En el paso del estado *write memory command* al *write*, adicionalmente, se calcula cuántos paquetes de datos se enviarán y se asigna ese valor al atributo *totalDataPaquets* para que la función *sendNextData()* sepa cuándo acabar.

Antes de llamar a *continueCommunication()* se debe procesar la respuesta al anterior mensaje enviado. En los casos en los que se recibe una respuesta a un comando de tipo getProperty, esta se procesa con el método *processProperties(property: Int)*.

```
private fun processProperties(property: Int){
    when(state){
        "secured" -> {
            if(property == 0){
                mainController.checkItem( item: "secured")
                Platform.runLater { mainController.writeInfo( text: "Flash security is disabled.", type: "info") }
            }
            else{
                Platform.runLater { mainController.writeInfo( text: "Cannot write. Flash security is enabled", type: "error") }
                isCommunicating = false
                mainController.error(state)
                state = "resetAndEnd"
            }
        }
        "size" -> {
            Platform.runLater {
                mainController.writeInfo( text: "Flash size: $property bytes", type: "info")
                mainController.writeInfo( text: "File size: ${fileBytes.size} bytes", type: "info")
            }
            if(property >= fileBytes.size){
                Platform.runLater { mainController.checkItem( item: "size") }
            }
            else{
                Platform.runLater { mainController.writeInfo( text: "Cannot write. File is bigger than flash memory size", type: "error") }
                isCommunicating = false
                mainController.error(state)
                state = "resetAndEnd"
            }
        }
        "sector size" -> {
            calculateEraseSize(property)
            Platform.runLater {
                mainController.writeInfo( text: "Flash sector size: $property bytes", type: "info")
                mainController.writeInfo( text: "Memory to erase: $bytesToErase bytes", type: "info")
            }
        }
    }
}
```

Figura 33: Código de la función *processProperties(property: Int)*.

Como se puede apreciar en el código, esta función recibe una propiedad representada por un entero (recuérdese que las propiedades ocupaban 4 bytes en las tramas, igual que los enteros) como parámetro. Al recibirse la trama de tipo *getPropertyResponse*, ya se ha comprobado que la propiedad recibida es la que se había pedido, de forma que esta función no tiene que volverlo a hacer. En cambio, recibe como parámetro; directamente la propiedad. Para saber de qué propiedad se trata y cómo tiene que tratarla, basta con saber en qué estado de la ejecución se encuentra:

- Si el estado es *secured*, la propiedad es *flashSecurityState*. Si la seguridad está desconectada (property = 0) se procede con la ejecución; si no, se detiene.
- Si el estado es *size*, la propiedad es *flashSizeInBytes*. Si el fichero cabe en la memoria, se sigue; si no, se aborta la ejecución.
- Si el estado es *sector size*, la propiedad es *flashSectorSize*. Se calcula cuántos sectores se van a borrar y se avanza al siguiente paso.

9. Vistas

Tal y como dice el título de este proyecto, las aplicaciones desarrolladas son aplicaciones Human-Machine Interface. Era de vital importancia dedicar especial atención a las ventanas con las que iba a interactuar el usuario. Es muchísima la información que circula por el bus de CAN y estos programas deben seleccionar la que es relevante y mostrarla de forma comprensible. También hacía falta abstraer la escritura de parámetros y la actualización del *firmware* de los protocolos que hay detrás, proporcionando una interfaz intuitiva que permita realizar de forma sencilla tareas que requieren una compleja lógica.

En este capítulo, se muestra el resultado de la implementación de esta característica.

9.1. Vistas del programa *Monitor*

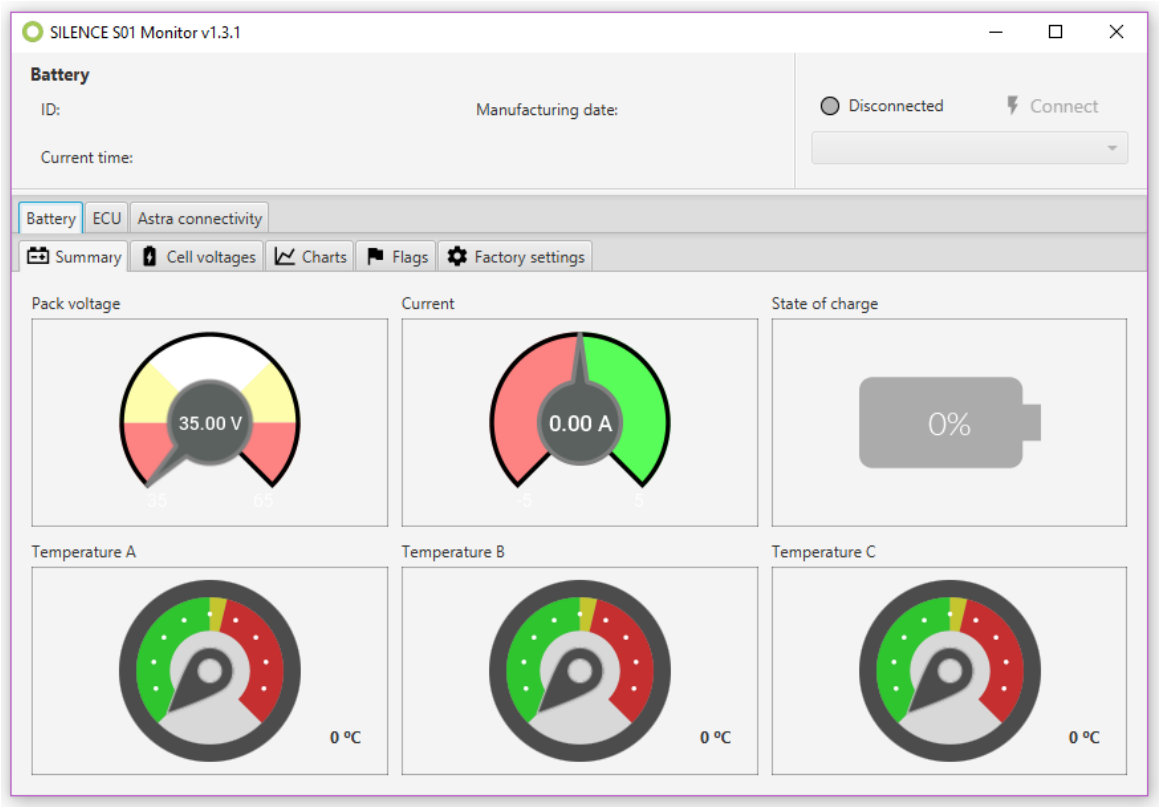


Figura 34: Vista del panel *Summary*

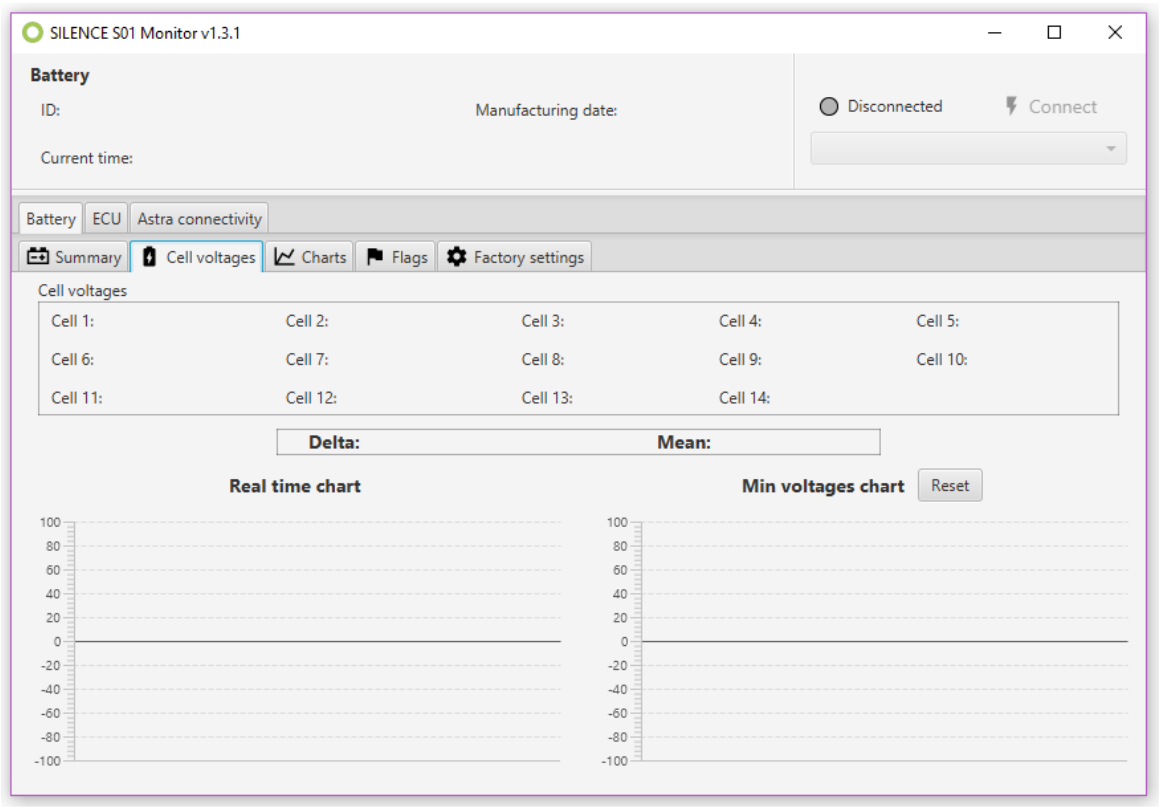


Figura 35: Vista del panel *Cell voltages*

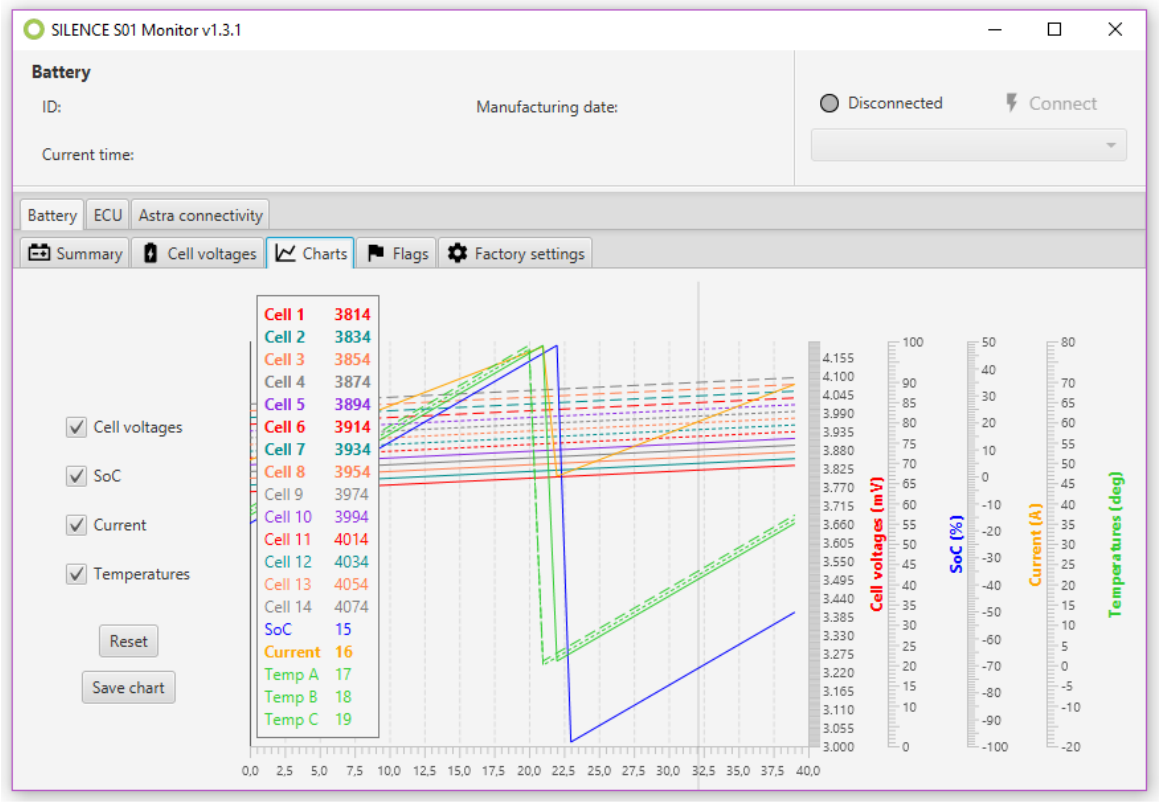


Figura 36: Vista del panel *Charts*

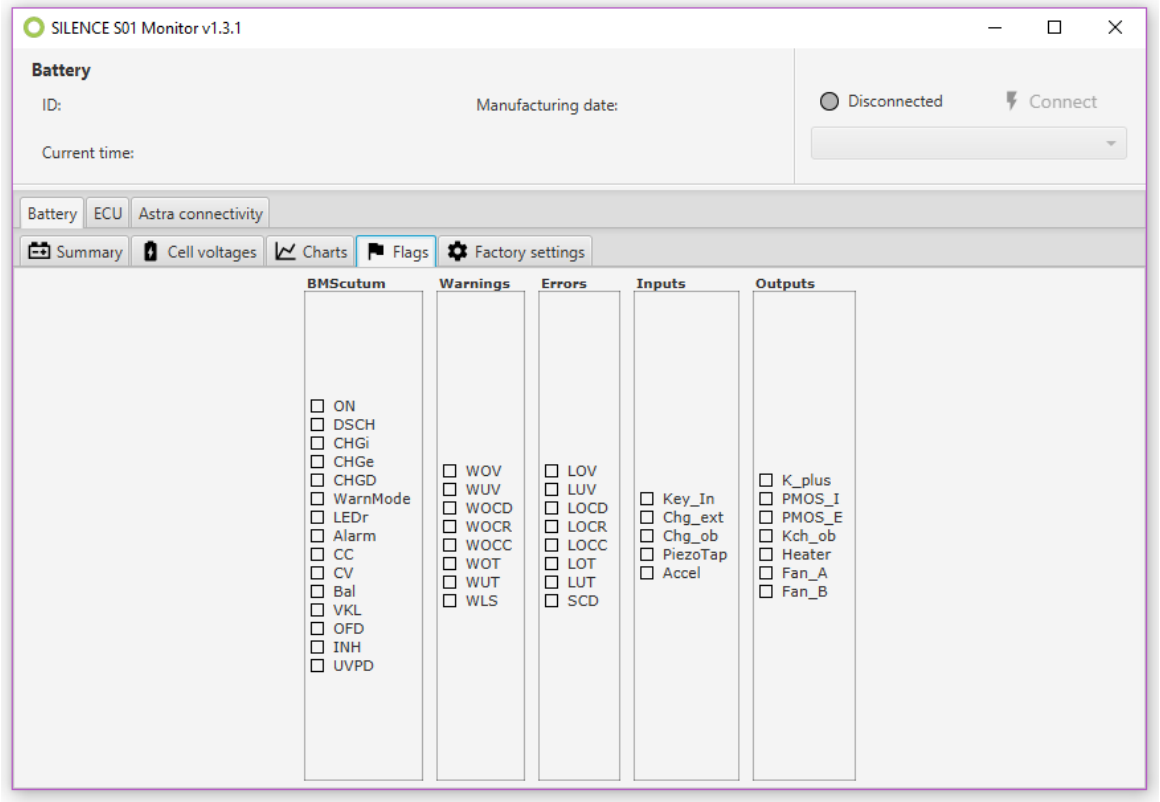


Figura 37: Vista del panel *Flags*

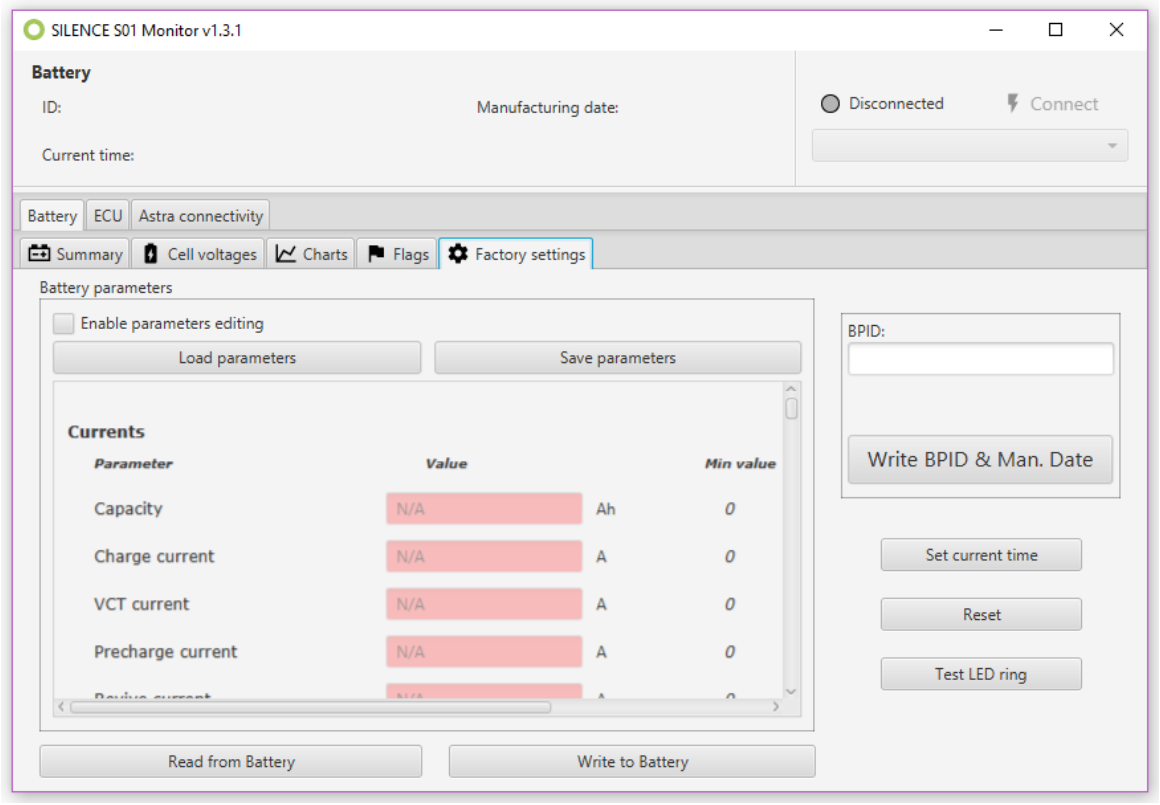


Figura 38: Vista del panel *Factory settings* del BMS

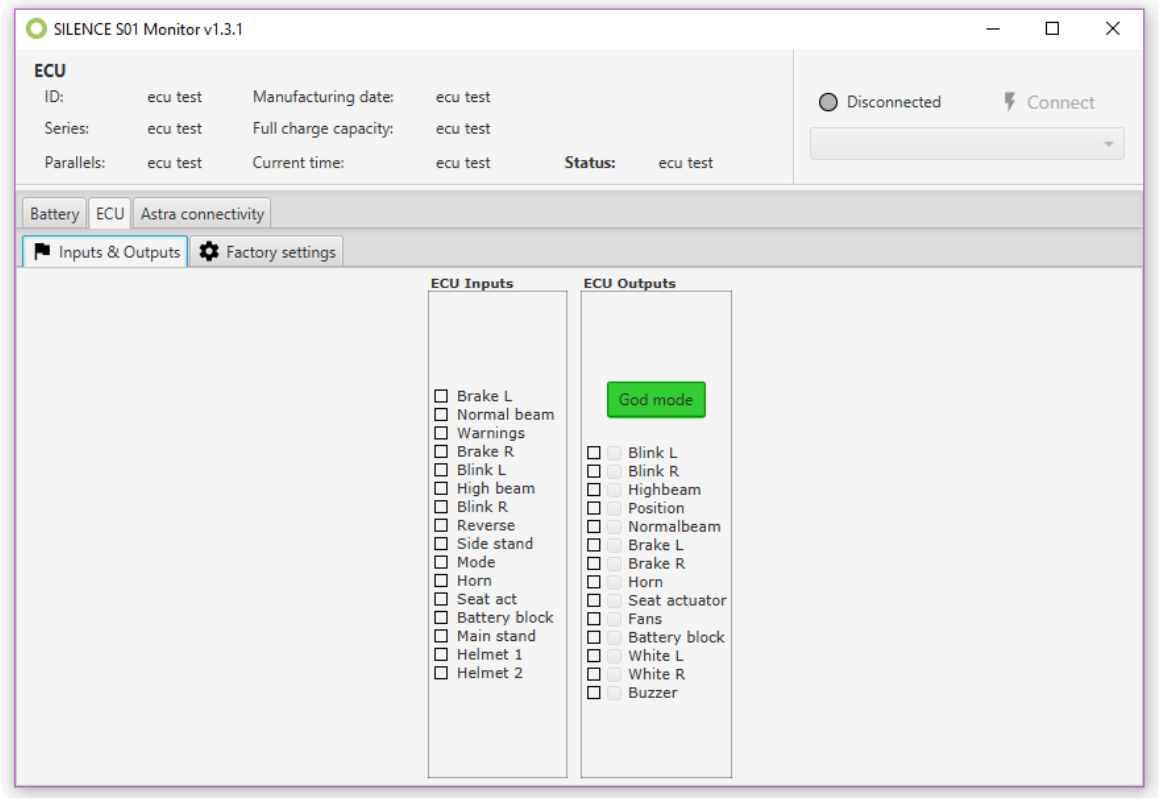


Figura 39: Vista del panel *Inputs & outputs*

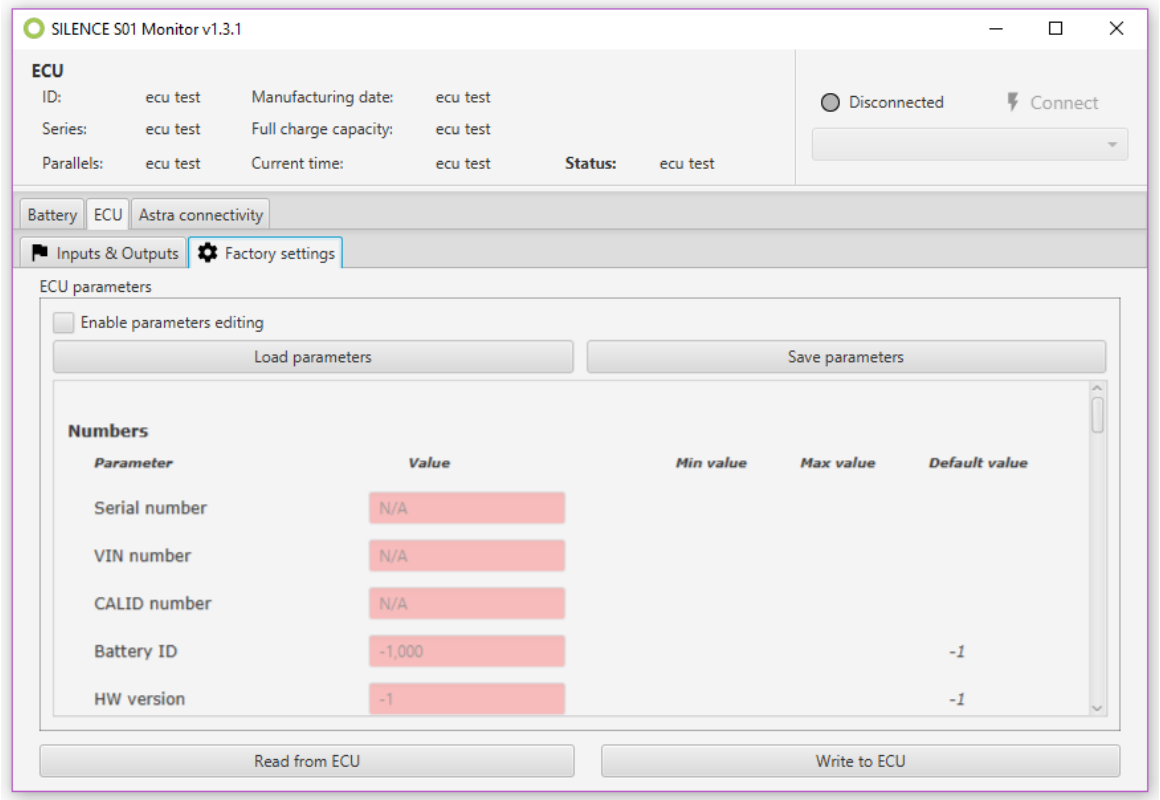


Figura 40: Vista del panel *Factory settings* de la ECU

9.2. Vista del prograna *Bootloader*

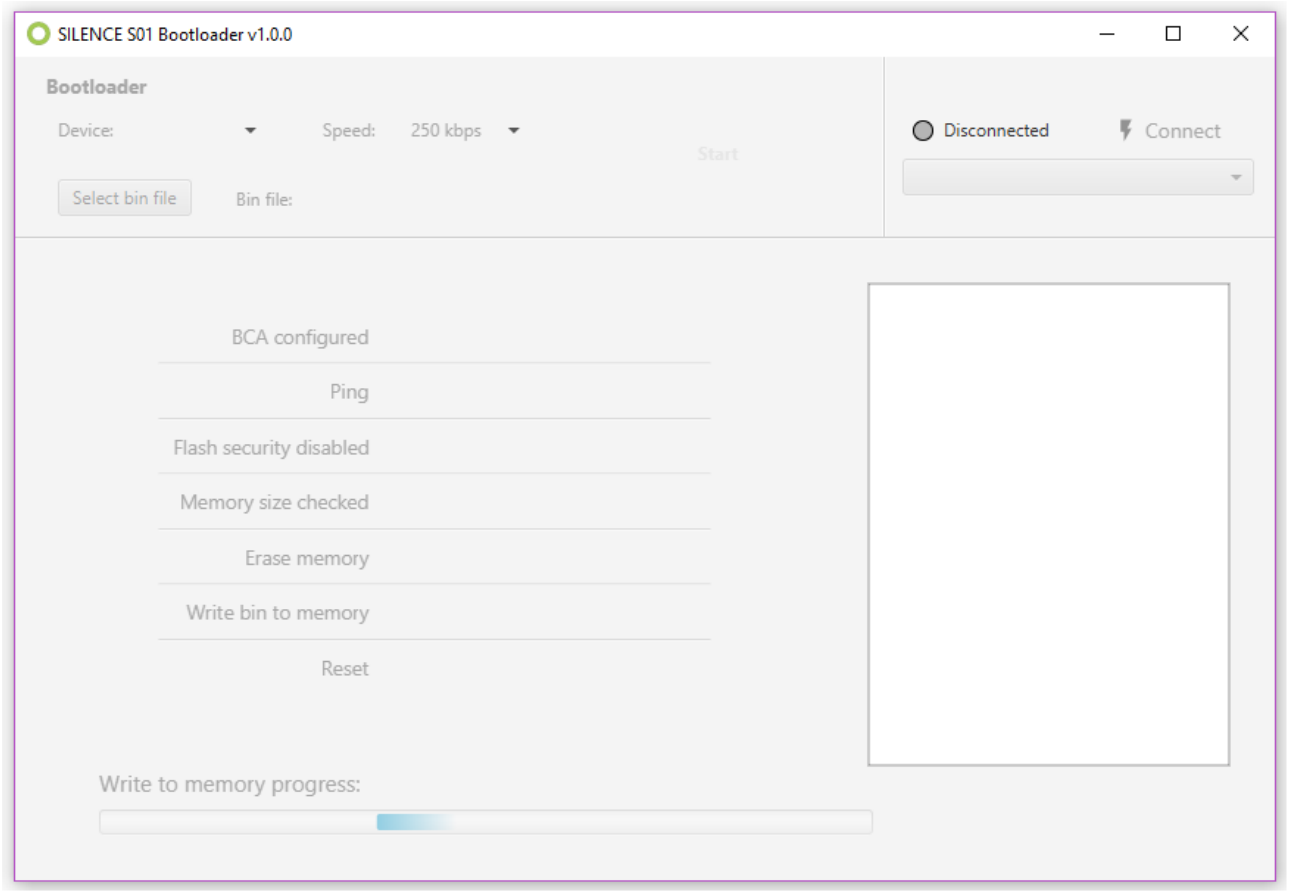


Figura 41: Vista del programa *Bootloader*

10. Informe de sostenibilidad

Una vez desarrollados los aspectos teóricos y tecnológicos del proyecto, en este capítulo se procede al estudio de los indicadores de sostenibilidad, es decir, medioambientales, económicos y sociales que se desprenden de este.

Este estudio se basa en la matriz de sostenibilidad propuesta en la guía publicada por la Facultad de Informática de Barcelona.

En esta matriz se propone el estudio de la sostenibilidad en el proyecto en tres etapas (proyecto puesto en producción, vida útil y riesgos) y en tres dimensiones (ambiental, económica y social). Obviamente, este informe solo puede analizar con más profundidad los indicadores de sostenibilidad de la primera etapa, es decir, mientras se ha desarrollado el proyecto. No obstante, dada la naturaleza del proyecto, también se realizarán reflexiones de las dos últimas etapas.

El análisis de la sostenibilidad es especialmente adecuado en este proyecto ya que el propio concepto de un vehículo eléctrico ya está concebido para contribuir a ciudades más sostenibles, menos contaminadas, que no pongan en peligro el medio ambiente ni agoten los recursos existentes.

10.1. Dimensión ambiental

Entre las medidas ambientales generales que se contemplan en Silence, cabe destacar las siguientes:

- Se está desarrollando el llamado *Solar Be Tree*, una placa fotovoltaica en forma de árbol que carga la batería gracias a la energía solar, sin necesidad de estar conectada a la red eléctrica.

Figura 42: Árbol solar desarrollado por Silence.

Figura 43: Ejemplo de uso del inversor del pack de batería.

- La batería dispone de un inversor para poder aprovechar el dispositivo como fuente de alimentación alterna para distintos electrodomésticos. Juntamente con la posibilidad de cargar la batería con fuentes alternativas como la energía solar, poder utilizar la batería como fuente de alimentación de pequeños electrodomésticos permite potenciar el uso de estas fuentes alternativas de energía con el beneficio que ello supone para el medio ambiente.
- Silence ha apostado por el uso compartido de motos eléctricas con un servicio de *sharing* integral que gestiona plataformas de *sharing* como Acciona. El vehículo compartido es una apuesta por el uso más sostenible de la movilidad urbana ya que esta modalidad permitirá en un futuro la disminución del número total de vehículos y la reducción de los trayectos.

El proyecto de software que se presenta en este Trabajo Final de Grado también colabora con buena parte en estas medidas medio-ambientales. El objetivo es sacar la máxima eficiencia de la moto en cuanto consumo y potencia para alargar la vida útil de la moto. Los cambios de firmware en los distintos elementos de la moto son necesarios para que estos puedan mejorar todos estos aspectos.

Por otra parte, para motivar un uso racional de la moto, la APP proporciona lo que se conoce como *huella de carbono* en la que se muestra estadísticas del CO_2 dejado de emitir en todos los recorridos.

En lo que al desarrollo de este proyecto se refiere, el impacto ambiental es mínimo ya que se trata únicamente del desarrollo de un software. El único impacto que se puede considerar es el de tener que recurrir a comprar hardware nuevo en vez de reutilizar; sin embargo, esto era necesario porque la empresa no disponía de ordenadores que no se estuvieran usando.

10.2. Dimensión económica

En la dimensión económica, la eliminación del combustible fósil y su sustitución por la energía eléctrica permite un ahorro de casi el 85 % del coste por kilómetro.

No solo el combustible, la iluminación de la moto basada en LEDS (incluidos los intermitentes, luces de posición, de freno, de cruce y largas) hace que, aunque la inversión inicial de una moto eléctrica sea importante, esta inversión se ve compensada con el bajo mantenimiento que supone.

Este ahorro se ve potenciado gracias a los sistemas de control que en buena parte se implementan gracias al presente proyecto de software. Por otra parte, la flexibilidad que permite poder modificar fácilmente el comportamiento electrónico de los distintos elementos de la moto facilitará que se realicen más frecuentemente estas mejoras. Esto contribuirá a retrasar la obsolescencia de la moto y aumentar la rentabilidad de la inversión inicial.

El software de monitorización diseñado permitirá reducir los tiempos de diagnóstico en caso de averías. En principio, esto contribuirá a servicios de mantenimiento más cortos y eficientes.

Por lo que al proyecto en sí respecta, el gasto económico es óptimo. Solo se ha tenido que contratar a un trabajador y solo se ha comprado el hardware necesario. Sí que se puede pensar en alternativas a tener un trabajador en prácticas en la empresa para la realización de este proyecto, como, por ejemplo, subcontratar una empresa de software. Probablemente, esto se ha descartado por el afán de la empresa en producir localmente todo lo que sea posible.

10.3. Dimensión social

La dimensión social también se ha tenido muy en cuenta en los proyectos de Silence. Por ejemplo, la apuesta por las plataformas de *sharing*, que se han mencionado anteriormente, es una contribución a nuevas relaciones sociales y relación de la sociedad con la movilidad urbana. La potenciación de las energías renovables y la facilitación de poder usar la batería recargada con dichas energías para su uso cotidiano contribuirá a una modificación del comportamiento social hacia un gasto de energía más sostenible.

En Silence se ha querido que sus manuales hagan hincapié a sus usuarios en las recomendaciones de vestimenta, elementos de protección, consejos de conducción y mantenimiento de la moto. Eso favorece la concienciación de que conducir una moto es una acción que contribuye a mejorar la movilidad urbana de manera palpable.

El presente proyecto ha tenido muy en cuenta estos aspectos sociales ya que buena parte del software desarrollado ha tenido como objetivos, no solo sacar la máxima eficiencia de la moto en cuanto consumo y potencia, sino que, respetando los distintos modos de conducción, anima a sustituir ciertas conductas de conducción por otras más respetuosas con los demás conductores y viandantes.

El propio software de la moto influye un uso más eficiente y sostenible, por ejemplo, al proponer la secuencia de modos de conducción (CITY-SPORT-CITY-ECO-CITY-SPORT-CITY...) o forzando a que, antes de pasar a modo ECO, se deba reducir la velocidad por debajo de 55 Km/h, o por el uso prioritario del freno electrónico antes del freno mecánico.

Para el desarrollo de este proyecto necesario (resuelve una necesidad muy específica de una empresa que no puede solventar con otro producto del mercado) no se ha considerado subcontratar una empresa de software de modo que la empresa se asegura que el desarrollador

recibe un salario que considera justo y promueve la producción local. A mí, personalmente, este proyecto me ha permitido desarrollar un software que se usa en un caso real y saber que he formado parte de un proyecto social muy importante: el avance a unas ciudades más limpias mediante la revolución del vehículo eléctrico que, además en este caso, es de producción local.

10.4. Vida útil y riesgos

Aunque el concepto de moto eléctrica por sí solo ya tiene en cuenta muchos de los aspectos de sostenibilidad, hay que tener en cuenta que estos vehículos se van a usar juntamente con el parque actual de vehículos de combustible y en una infraestructura viaria que no tiene en cuenta esta diversidad de vehículos. No obstante, la infraestructura viaria empieza a evolucionar hacia:

- Facilitar la convivencia de todos estos nuevos vehículos (incluyendo también patinetes y bicicletas).
- Adaptación de las vías públicas para reducir el riesgo de la convivencia de vehículos que van a distinta velocidad.
- Adaptación de espacios de aparcamiento para motos eléctricas con puntos de carga rápida.

También debe promocionarse el uso de vehículos compartidos para reducir el número total de estos y disminuir el número de trayectos.

Al software desarrollado se le espera una larga vida útil. Al haber sido diseñado siguiendo los métodos adecuados para la orientación a objetos como la modularidad, es fácilmente mantenible y adaptable para futuros modelos. Además, no parece conllevar ningún riesgo ya que el constante testing y mantenimiento lo hacen fiable y, dada su gran utilidad en la producción de la empresa, no hay que temer que caiga en desuso.

11. Conclusiones y trabajo futuro

En este proyecto se ha diseñado e implementado el software de escritorio de la moto eléctrica S01 de Silence, empresa en la que he realizado las prácticas curriculares correspondientes al grado de Ingeniería Informática desde diciembre de 2018 a junio de 2019.

Cuando comencé el período tuve que acabar de dar servicio a la aplicación anterior, creada fundamentalmente para la versión anterior de la moto eléctrica, la S02. De la misma forma que, después de mi salida de la empresa por haber acabado el período de prácticas, aún quedan aspectos que el futuro informático de la empresa debe acabar de implementar. Uno de los aprendizajes que me llevo de esta fase de mi formación es que los proyectos no se finalizan, están vivos, evolucionando hasta que otro proyecto lo sustituye.

En el momento de finalizar estas prácticas en Silence, las aplicaciones desarrolladas ya estaban siendo utilizadas por los distintos trabajadores de la empresa. Los encargados del BMS y la ECU ya lo empleaban para probar las nuevas versiones del *firmware* de estos sistemas que estaban desarrollando y en la linia de montaje ya se estaban ciclando las primeras baterías y parametrizando las primeras motos.

No obstante, el trabajo con estas aplicaciones no se ha acabado. De la misma manera que yo tuve que dedicar parte de mi tiempo al mantenimiento de las aplicaciones de la S02, el desarrollador que siga en la empresa deberá hacer lo propio con las de la S01. Esto me ha obligado a documentar exhaustivamente todas las clases, funciones y programas realizados.

Algunas de las funcionalidades que habrá que incorporar en un futuro son:

- Integración del *Monitor* con el sistema Astra que controla la conectividad de la moto con Internet y Bluetooth.
- Adaptación del programa *Monitor* para que se pueda ver correctamente desde las pequeñas pantallas táctiles que tienen los cicladores de batería.
- Identificación de usuario.
- Distintas funcionalidades disponibles dependiendo de qué tipo de usuario utilice la aplicación.
- Extracción del histórico de eventos de la moto para ser mostrado en el *Monitor*.

12. Bibliografía

1. Blanco Curieses, Francisco J. (2019) Estudio del bus de comunicaciones CAN. Universitat Oberta de Catalunya. Licencia Creative Commons.

   http://openaccess.uoc.edu/webapps/o2/bitstream/10609/88505/8/fblancocuTFM0119memoria.pdf

2. Climent, J., Cabré, J., Sánchez, F., Martín, C., Vidal, E., López, D. (2018). El informe de sostenibilidad del Trabajo de Fin de Grado del área de las ingenierías. *REDU. Revista de Docencia Universitaria*, 16(2), 75-86. https://doi.org/10.4995/redu.2018.10092

3. Comunidad de Madrid. Guía del vehículo eléctrico. *Fundación de la Energía de la Comunidad de Madrid* (consultado 2019) https://www.icmm.csic.es/es/divulgacion/documentos/Guia-del-Vehiculo-Electrico-2009-fenercom.pdf?id=127

4. Convenio de Cooperación educativa para la realización de prácticas académicas externas (consulatdo 2020)

   https://www.fib.upc.edu/sites/fib/files/images/catala.pdf

5. FIB: información sobre prácticas en empresa (consultado 2020).

   https://www.fib.upc.edu/ca/empresa/practiques-en-empresa

6. Kinetis KE1xF Sub-Family Reference Manual (consultado 2019)

   https://www.nxp.com/docs/en/reference-manual/KE1xFP100M168SF0RM.pdf

   Concretamente el capítulo 23: Kinetic ROM Bootloader

7. IDE IntelliJ IDEA. Entorno de desarrollo (consultado 2020).

   https://www.jetbrains.com/idea/download/#section=windows

8. Informe de sostenibilidad en el TFG (consultado 2019)

   https://www.fib.upc.edu/sites/fib/files/documents/estudis/tfg-informe-sostenibilitat-2018.pdf

9. Oracle Java SE Support Roadmap (consultado 2020).

   https://www.oracle.com/technetwork/java/java-se-support-roadmap.html

10. Silence S01. Manual del propietario (consultado 2019)

    https://www.silence.eco/wp-content/uploads/2019/09/S01-MANUAL-DE-USUARIO.pdf

11. Silence Urban ecomobility Página oficial (consultado 2020).

    https://www.silence.eco/

12. Teamgantt. Pàgina de descripción y descarga (consultado 2020).. https://www.teamgantt.com/

13. Wrike. Software de gestión de proyectos (Consultado 2020). https://www.wrike.com/price/