GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

TELEMETRÍA PARA MOTOSTUDENT

Autor: Alberto Sánchez Cuadrado

Director: José Daniel Muñoz Frías

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Telemetría para Motostudent

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2020/21 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Alberto Sánchez Cuadrado Fecha: 06 / 26 / 2021

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado por MUÑOZ FRIAS JOSE DANIEL - 52573841G el día 08/07/2021 con un certificado emitido por AC FNMT Usuarios

Fdo.: José Daniel Muñoz Frías Fecha: ……/ ……/ ……

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

TELEMETRÍA PARA MOTOSTUDENT

Autor: Alberto Sánchez Cuadrado

Director: José Daniel Muñoz Frías

Madrid

TELEMETRÍA PARA MOTOSTUDENT

Autor: Sánchez Cuadrado, Alberto.
Director: Muñoz Frías, Jose Daniel.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

El proyecto aborda el diseño de un sistema de gestión de baterías y un sistema de gestión de datos, ambos integrados en el chasis de una moto de competición eléctrica. Este trabajo cubre desde las primeras fases de diseño hasta la implementación de los sistemas y su uso en las primeras pruebas del vehículo.

Palabras clave: CAN, BMS, Telemetría

1. Introducción

El proyecto se desarrolla en el contexto de la competición Motostudent Electric. El objetivo de esta competición es involucrar a estudiantes de todas partes del mundo en el diseño de una moto eléctrica de competición para aportar a estos experiencia en el sector de la movilidad eléctrica. Los sistemas desarrollados en este trabajo se utilizarán en el prototipo de moto eléctrica del ICAI Speed Club.

Este proyecto se centra en el diseño del dos sistemas electrónicos de gran importancia: el sistema de gestión de baterías, o BMS (Battery Management System) y el sistema de adquisición de datos. Ambos sistemas se encuentran dentro de la categoría de sistemas de baja tensión según el reglamento de la competición. No obstante, el sistema BMS actúa sobre el sistema de alta tensión y, por lo tanto, deberá cumplir con las medidas de seguridad establecidas para la competición [1].

2. Definición del proyecto

Los dos sistemas a diseñar trabajarán de forma conjunta para conseguir un vehículo eléctrico seguro y funcional. No obstante, sus funciones están claramente separadas y se buscará que cada sistema pueda operar de forma independiente. Esta modularidad permitirá reemplazar o actualizar cada sistema de manera individual mucho más fácilmente.

El sistema de gestión de baterías se encarga de realizar las siguientes 3 funciones: En primer lugar, proporciona seguridad al vehículo al impedir que las celdas del pack de baterías operen fuera de los límites de tensión y temperatura establecidos. En segundo lugar, proporciona

información acerca del estado de carga, o State of Charge (SoC) de la batería para que el piloto pueda planificar su conducción. Por último, el sistema BMS aplica balanceo sobre las distintas celdas para todas tengan la misma carga, prolongando así la vida útil de la batería y asegurando un buen comportamiento de esta a lo largo de su ciclo de descarga.

Por otro lado, el sistema de adquisición de datos tiene 2 funciones: Primero, muestra datos sobre el estado de la moto al piloto en tiempo real a través de un pequeño display y segundo, guarda estos datos para que sean posteriormente analizados por el equipo de ingeniería, permitiendo optimizar los sistemas de la moto para futuras ediciones.

En el proyecto, el proceso de diseño de cada uno de los sistemas se ha dividido en los siguientes 4 apartados. En primer lugar, se estudia el estado de la cuestión y posibles soluciones comerciales ya existentes. En segundo lugar, se fijan los objetivos y requerimientos de los sistemas y, por último, se realiza el desarrollo del hardware y software.

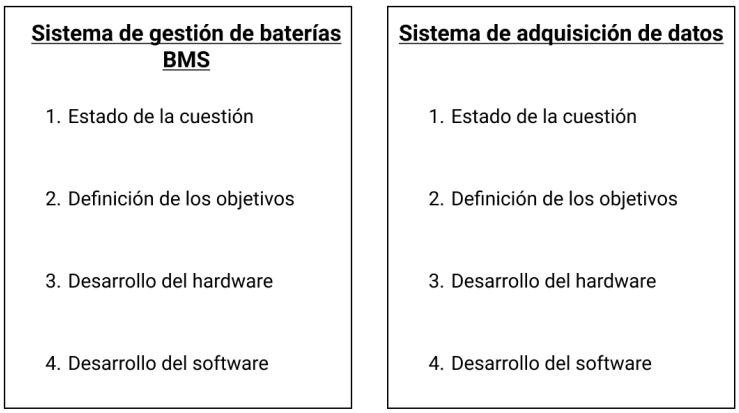


Figura 1. Fases de diseño de los sistemas.

3. Descripción de los sistemas

El sistema de gestión de baterías se basará en una topología modular, tal y como se muestra en el esquema a continuación. Cada uno de los 3 módulos distribuidos se encarga de realizar lecturas de tensión y temperatura para módulos de hasta 14 celdas y enviar los datos a través de bus CAN a un microprocesador central donde se ejecuta la lógica que determina si hay que aplicar balanceo o si hay que activar las protecciones y donde se calcula el estado de carga de la batería.

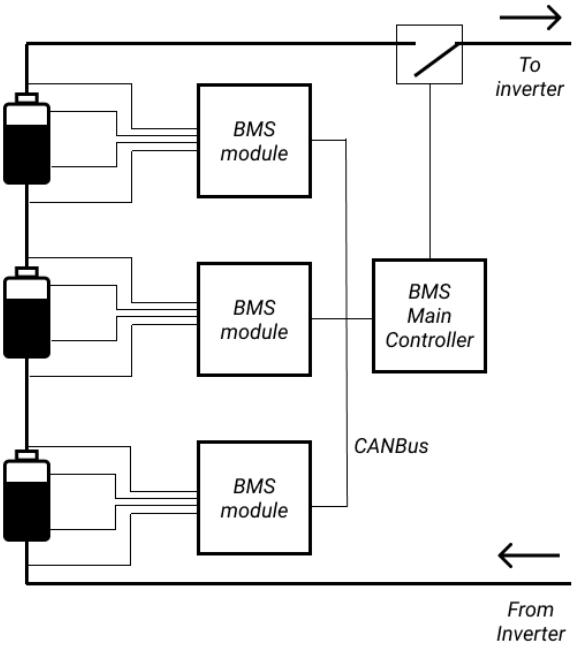


Figura 2. Esquema de la topología del sistema BMS.

El sistema de adquisición de datos aprovecha la topología distribuida del BMS y de las funcionalidades del bus CAN para tomar lecturas de las tensiones y temperaturas. Además, toma lecturas periódicas provenientes del cargador y del inversor. Con todos estos datos, el procesador central controlará un display en tiempo real y generará una base de datos que posteriormente se subirá a la nube para realizar un análisis del comportamiento de la moto.

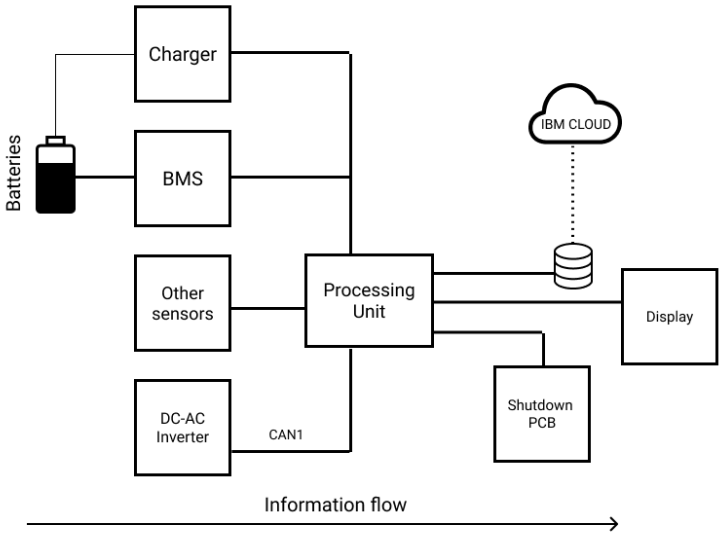


Figura 3. Esquema del sistema de adquisición de datos.

4. Resultados

Los primeros ensayos en el banco de pruebas han demostrado que la implementación llevada a cabo en el ICAI Speed Club funciona correctamente y que su integración en el sistema eléctrico de la moto eléctrica es prometedora.

5. Conclusiones

El proyecto ha proporcionado un diseño robusto y modular de un sistema de gestión de baterías y un sistema de adquisición de datos. Gracias a las características del diseño, será fácil realizar modificaciones o adaptaciones para ediciones futuras de la competición de Motostudent.

6. Referencias

[1] «MotoStudent | Rules & Regulations». http://www.motostudent.com/rules.html (accedido jul. 01, 2021).

TELEMETRY SYSTEMS IN MOTOSTUDENT

Author: Sánchez Cuadrado, Alberto.
Supervisor: Muñoz Frías, Jose Daniel.
Collaborating Entity: ICAI – Universidad Pontificia Comillas.

ABSTRACT

This aim of this project is to design a Battery Management System and a data acquisition system, both of which are to be integrated in the chassis of an electric motorcycle. The project will cover the whole design process, from the first phases of design to the implementation of the systems and their use in the first dynamic tests of the vehicle.

Keywords: CAN, BMS, Telemetry. Telemetría

1. Introduction

This project aims to develop two important electronic systems for an electric motorcycle that will be involved in the sixth edition of Motostudent Electric. The objective of this competition is to have students from universities all around the world participate in the engineering design of a competition-ready electric motorcycle with the aim of providing these students with experience in the Electric Vehicle sector. The systems designed in this project will be integrated into the prototype being developed by the ICAI Speed Club, a student association at Comillas University in Madrid.

The two electronic systems that this project will focus on developing are the Battery Management System (BMS) and the data acquisition system. Both systems are considered low voltage according to the rules and regulations of the competition. Nevertheless, the BMS system will be connected to the high-voltage system of the EV and will therefore be subject to the security measures that Motostudent requires.

2. Project definition

The battery management system and the data acquisition system will have to work together in order to make the electric vehicle both safe and functional. Nevertheless, their functions are clearly different, and this project will aim to keep these two systems working independently. This modularity will make it easier in the future to replace or upgrade certain parts of the system.

The Battery Management System is responsible for the following three tasks: Firstly, it provides security to the EV by avoiding its operation if the cells in the battery pack are found outside the voltage or temperature limits that have been considered safe. Secondly, it provides information regarding the State of Charge of the battery so that the pilot can adapt their driving. Lastly, the BMS system applies balancing among the cells so that they all have the same level of charge, prolonging the battery pack's life and ensuring a correct behavior throughout its discharge cycle.

The data acquisition system is responsible for the following two tasks: Firstly, it shows the pilot real-time data regarding the state of the motorcycle through a small built-in display and secondly, saves this real-time data in a database so that it can be analyzed by an engineering team later, allowing them to optimize the systems for future prototypes.

In this project, the design process followed for each of the systems has been divided into the following 4 stages. First, the state-of-the-art technologies are studied in order to find potential commercial solutions that are readily available. Second, the objectives and requirements of the systems are set and lastly, the development of the hardware is carried out, followed by the development of the software.

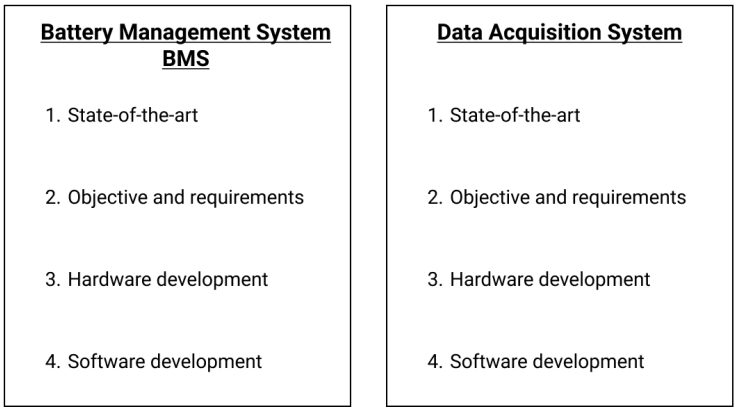


Figure 1. Stages of the development of the systems.

3. Description of the systems

The battery management system is based on a distributed topology, just like is shown in the figure below. Each of the 3 distributed modules is tasked with taking voltage and temperature measurements for each of the modules, which can be made of up to 14 cells. These modules then use a CAN bus to send the data back to a central microprocessor, where logic is executed

in order to determine whether if any balancing needs to be carried out or if the system needs to be shut down due to security concerns and where the state of charge is calculated.

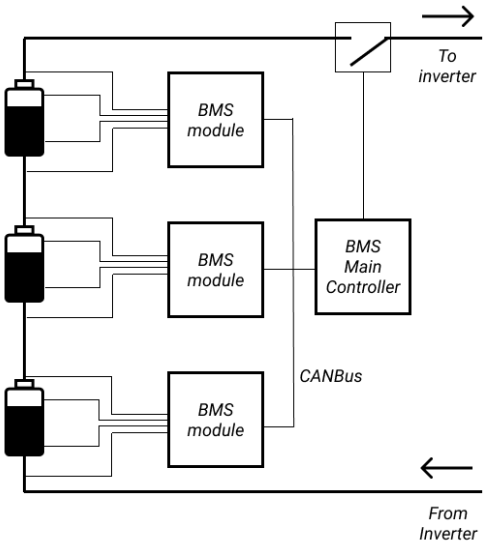


Figure 2. Diagram of the topology of the BMS.

The data acquisition system takes advantage of this distributed BMS topology and CAN bus functionality to read the voltages and temperatures of the cells. On top of that, it takes periodic readings from the charger and inverter. With all this data being collected, the central processor controls a display in real time and generates a database that will eventually be uploaded to a cloud service where the data can be analyzed more in depth.

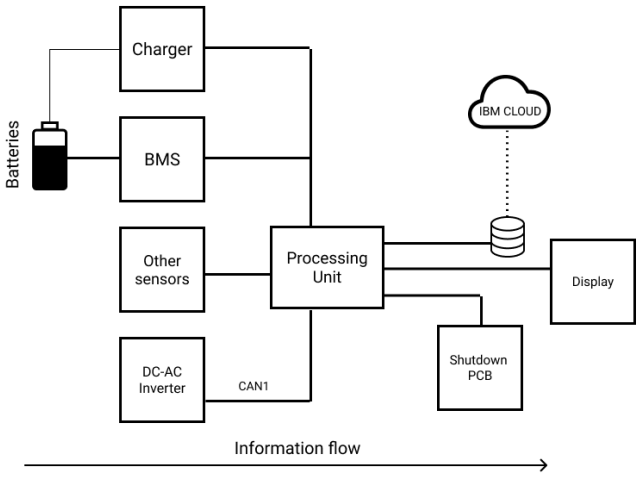


Figure 3. Diagram of the data acquisition system.

4. Results

The first tests carried out in a test bench have shown that the implementation carried out in the ICAI Speed Club is fully functional and its integration in the electric motorcycle prototype is very promising.

5. Conclusions

This project has provided a robust and modular design for a battery management system and a data acquisition system. Thanks to the characteristics of this design, it will be relatively simple to make modifications and adaptations to the systems in order to add or improve its functionality for future editions of the Motostudent competition.

6. Referencias

[1] «MotoStudent | Rules & Regulations». http://www.motostudent.com/rules.html (accessed jul. 01, 2021).

Índice de la memoria

Índice de figuras

Índice de tablas

Capítulo 1. INTRODUCCIÓN

En los últimos años los vehículos eléctricos han experimentado numerosas mejoras en sus tecnologías, especialmente en cuanto a la densidad de carga y eficiencia de los sistemas de almacenamiento de energía eléctrica. Gracias a estos avances los vehículos eléctricos están demostrando ser una alternativa más eficiente y menos contaminante a los vehículos de combustión interna.

En este contexto de avances en vehículos propulsados por energía eléctrica surge la competición MotoStudent Electric. El objetivo de esta competición es involucrar a estudiantes de todo el mundo en el proceso de diseño de una moto eléctrica con el fin de desarrollar tecnologías que avancen el sector de la movilidad eléctrica. El ICAI Speed Club (ISC) es una asociación de estudiantes que participa en esta competición.

En un vehículo eléctrico se denomina powertrain al conjunto de subsistemas encargados de propulsar al mismo. El powertrain por tanto abarca sistemas de alto voltaje (HV – High Voltage) y de bajo voltaje (LV – Low Voltage). En la competición de Motostudent se denomina HV a cualquier sistema que trabaje con más de 40 VDC. El sistema HV se encarga de la acumulación, conversión y control de la energía eléctrica que permite propulsar al vehículo. Por otro lado, el sistema LV se encarga de monitorizar el estado de los sistemas HV, recopilar datos y proporcionar elementos de protección. Aunque este trabajo se va a centrar en el sistema de baja tensión es importante no perder de vista que actúa sobre una serie de elementos de alta tensión determinados y que por lo tanto deberá cumplir con los requerimientos que el sistema HV necesite. En concreto, este proyecto se centrará en el desarrollo del sistema de gestión de baterías y del sistema de adquisición de datos.

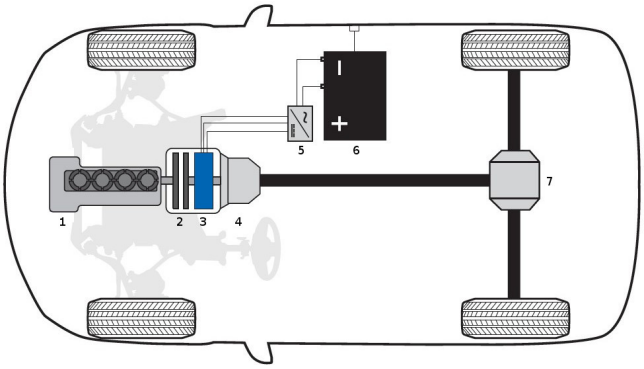


Figura 4. Esquema simplificado del sistema HV de un vehículo eléctrico. [1]

El sistema de gestión de baterías es un elemento crítico de cualquier vehículo eléctrico ya que es de gran importancia la monitorización del estado de todas y cada una de las celdas para evitar cualquier problema eléctrico. Este sistema tiene varias funciones. En primer lugar, debe proporcionar seguridad (evitando que las baterías operen fuera de sus límites de tensión o temperatura). En segundo lugar, un BMS debe proporcionar información acerca del estado de carga (SoC: State of Charge) y de salud (SoH: State of Health) de la batería. Por último, un buen sistema BMS aplica balanceo sobre las diferentes celdas de la batería. Este último paso es especialmente necesario para asegurar un correcto comportamiento de las baterías durante toda su vida útil.

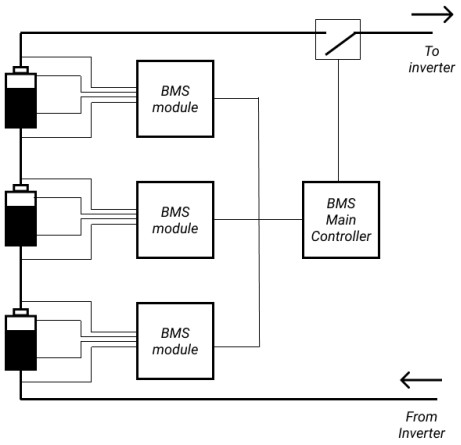


Figura 5. Esquema del sistema BMS distribuido de este proyecto.

El sistema de adquisición de datos tiene dos objetivos. Por un lado, debe ser capaz de mostrar en tiempo real al piloto de la moto eléctrica varios parámetros de interés sobre el estado de los distintos sistemas de la moto para que el mismo pueda tomar decisiones sobre su estilo de pilotaje en medio de la carrera. Por otro lado, debe permitir el guardado de una serie de datos que describen el comportamiento de la moto para su posterior descarga y análisis. Este paso es de gran utilidad para el análisis ingenieril y posterior optimización de los sistemas de la moto.

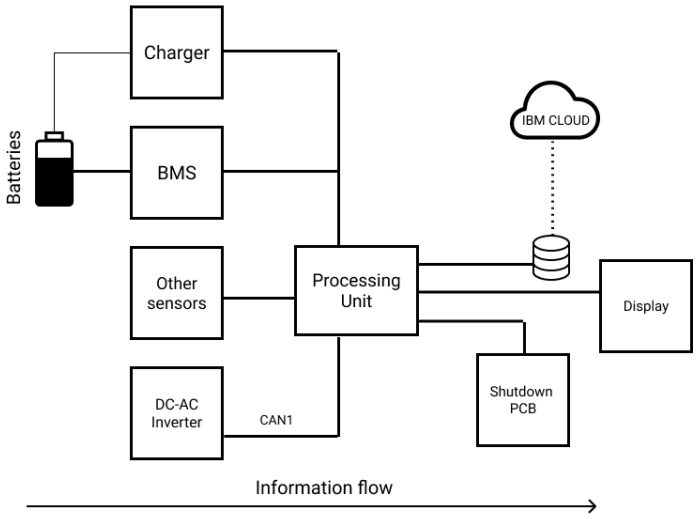


Figura 6. Esquema del sistema de adquisición de datos del proyecto.

Dado que el desarrollo de los sistemas se da en el contexto de la competición de Motostudent, muchas de las decisiones de diseño se verán afectadas por la normativa y especificaciones marcadas por los organizadores de la competición. Esta normativa está redactada principalmente teniendo en mente la seguridad de los pilotos, ingenieros y asistentes a la competición.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

Los dos sistemas desarrollados en este trabajo se apoyan en una serie de tecnologías y protocolos. En este capítulo se describirá el funcionamiento de cada tecnología y su función dentro de los sistemas para tener una visión más completa del proyecto.

2.1 CAN BUS

El protocolo CAN (Controller Area Network) es un protocolo de comunicación por cable desarrollado por Robert Bosch Gmbh. Y que es actualmente usado a menudo en la industria automovilística por permitir comunicación entre microcontroladores sin necesitar una unidad central y por su buen comportamiento frente a interferencias electromagnéticas.

Un bus CAN está compuesto por nodos conectados entre sí por dos cables entrelazados, CAN-H y CAN-L, separados normalmente por una resistencia de 120 Ω para evitar reflexiones en el bus. Cada nodo está compuesto por 3 elementos separados: Una unidad de procesamiento (normalmente un microcontrolador), un controlador CAN (encargado de aplicar el protocolo CAN en su capa enlace de datos o "Data Link") y el transceptor CAN (encargado de aplicar la capa física del protocolo CAN). De esta manera, en una configuración típica, un microcontrolador leería datos de un sensor del vehículo y determinaría el mensaje CAN correspondiente. El controlador CAN aplica las reglas del protocolo CAN y envía este mensaje al transceptor, donde la señal se ajusta a los niveles de tensión establecidos por la capa física. El siguiente esquema muestra con claridad estos 3 elementos que componen un nodo CAN.

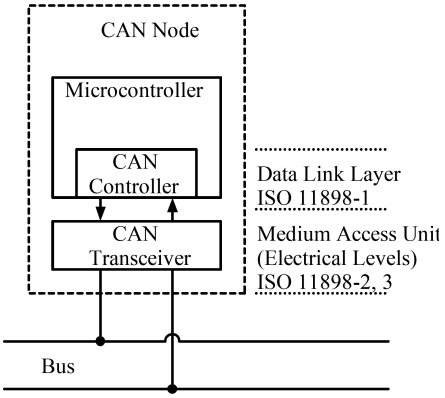


Figura 7. Esquema de los 3 elementos de un nodo CAN. [2]

Cada nodo CAN tiene la capacidad tanto de transmitir como de leer datos del bus CAN. Para entender cómo se envía la información a través de este bus ayuda fijarse en la siguiente figura, en la que se muestra la estructura de un mensaje CAN y su representación en niveles lógicos en cada una de las líneas (H y L) del bus. Antes de entrar en detalles de cómo se transmiten los mensajes por el bus es importante recalcar que cualquier comunicación CAN se basa en enviar un mensaje de hasta 8 bytes acompañado de un ID. El ID tiene una longitud de 11 bits en el formato CAN 2.0 A (o estándar) y una longitud de 29 bits en formato CAN 2.0 B (o CAN extended). Aunque ambos formatos puedan coexistir en el mismo bus, siempre tendrá prioridad un mensaje con ID estándar frente a un ID extended. [3]

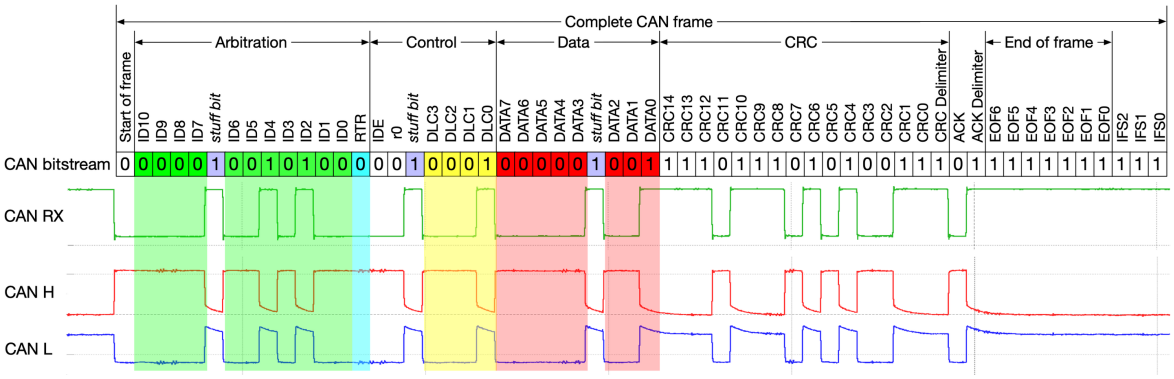


Figura 8. Representación de un mensaje CAN estándar. [4]

La transmisión de un mensaje CAN está compuesta de 3 fases: Una primera fase de arbitración, una segunda fase de transmisión de datos y una tercera fase de arbitraje y gestión

de errores. Aunque no sea de especial interés para este proyecto entrar en los detalles exactos de un mensaje CAN, sí que es relevante explicar que el ID desde el que se envía un mensaje es clave en la resolución de conflictos en el bus: Si dos nodos tratan de establecer comunicación al mismo tiempo, aquel con el menor ID tendrá preferencia frente al otro. Esta parte de la comunicación que evita que varios controladores "hablen" al mismo tiempo tiene lugar en la primera fase de arbitraje.

Aunque los detalles específicos de transmisión de datos y control de errores puedan hacer complicado implementar el protocolo CAN, existen controladores comerciales que llevan a cabo estas tareas, dejando al ingeniero únicamente la tarea de programar el envío y recepción de mensajes CAN. Existen además protocolos de abstracción superiores, como el CANopen, utilizado comúnmente en sistemas automáticos. CANopen estandariza las direcciones a utilizar y el formato de los mensajes para añadir funcionalidad al protocolo CAN y permitir comunicación entre dispositivos de fabricantes diferentes. [5]

Capítulo 3. ESTADO DE LA CUESTIÓN

A continuación, se analizará cómo han abordado el diseño de estos sistemas varios equipos de ingeniería. Primero se estudiarán las soluciones a las que han llegado empresas de vehículos eléctricos, posteriormente se analizarán las distintas opciones disponibles en el mercado para desarrollar los sistemas de acuerdo con nuestros requerimientos.

3.1 SOLUCIONES PROFESIONALES

Actualmente hay muchas empresas produciendo vehículos eléctricos. Es muy buena idea fijarse en cómo han solventado estas empresas los problemas que pretende abordar este proyecto para tener un punto de partida de donde empezar a diseñar. Aunque estas empresas operan con otros requerimientos, presupuestos o con distintos tipos de vehículos, el estudio de los distintos enfoques que han usado para diseñar los sistemas BMS es de gran utilidad para el proyecto.

Para este apartado se estudiará el enfoque de Tesla y el de Play and Drive. La primera empresa es una productora de coches eléctricos californiana conocida por ser una de las más avanzadas en cuanto a la tecnología de sus baterías y powertrain (sistema de propulsión). La segunda es una empresa que desarrolló un prototipo de moto eléctrica para demostrar la viabilidad de la competición Motostudent en 2014.

3.1.1 PACK DE BATERÍAS

El último modelo de coche de esta empresa californiana (Tesla Model 3) se ha diseñado con un pack de baterías compuesto por 4,416 celdas de ion de litio. La configuración de estas es cilíndrica 21700, lo que significa que tienen un diámetro de 21mm y una altura de 700mm. El uso de estas celdas de pequeño tamaño permite su agrupamiento en grupos en paralelo y en serie de forma que se alcance la tensión de salida especificada y una forma que se amolde bien a los requerimientos del resto de departamentos. En el caso del Model 3 las celdas se

disponen en 96 paralelos de 46 celdas lo que da una tensión nominal del pack completo de 346 V. En la imagen se puede apreciar cómo es la disposición de las celdas dentro del coche en sí.

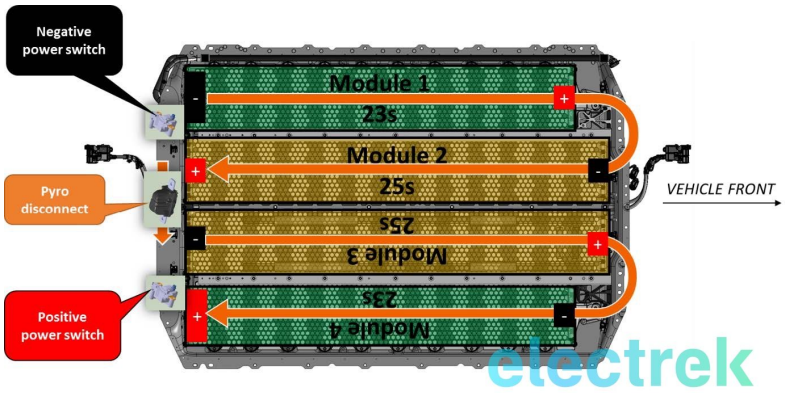


Figura 9. Disposición de las celdas en un Tesla Model 3. [6]

Figura 10. Formato de las celdas de los coches Tesla. [7]

Las características del sistema acumulador de energía del Model 3 quedan resumidas en la siguiente tabla:

Acumulador Tesla Model 3	
Tensión nominal	346 v
Nº de celdas	4,416

Nº paralelos	46
Nº de series	96
Química	Li-Ion (Lithium Ion)

Tabla 1. Resumen de las características del acumulador del Tesla Model 3.

También se tendrán en cuenta las especificaciones de la moto diseñada por Play and Drive ya que el tipo de vehículo se adapta mucho más a lo que se pretende diseñar en este trabajo. [8]

Acumulador Play & Drive	
Tensión nominal	96.2 v
Capacidad	3.8 kWh
Química	Li-Po (Lithium Polymer)

Tabla 2. Resumen de las características del acumulador de la moto Play & Drive.

3.1.2 BATTERY MANAGEMENT SYSTEM (BMS)

El estudio del sistema BMS de Tesla ha sido de gran utilidad para decidir cómo afrontar la gestión de las celdas. Dentro de las distintas topologías posibles para un BMS, el Model S de Tesla utiliza un sistema modular. Esto significa que cada módulo está acompañado por un controlador BMS que lo monitoriza y envía la información relevante a través de algún protocolo de comunicación (cada módulo está formado por varios grupos de celdas en paralelo, todos a su vez conectados en serie). En este caso el protocolo es CAN bus. Como ya se ha visto anteriormente, este protocolo es muy usado en la industria automovilística por su robustez frente a interferencias electromagnéticas y por su carácter modular. [9]

3.1.3 GESTIÓN DE DATOS

Los coches diseñados por Tesla cuentan con un sistema de procesamiento de datos muy potente formado por una CPU (Central Processing Unit), una GPU (Graphics Processing Unit) y varias NPU (Neural Processing Unit) [10, p. 3]. Si bien es verdad que no se pretende en este proyecto abordar problemas de Machine Learning que requieran tanta capacidad de procesamiento, este análisis no deja de ser de gran utilidad ya que pues se puede ver qué tipo de procesador, sistema operativo o lenguajes de programación han utilizado en la empresa californiana. Gracias al artículo de Patrick Kiley se ha podido conocer que el software de los BMS del Model S de Tesla se ejecuta en un entorno Linux y que ejecuta principalmente código en C, como en la mayoría de sistemas de producción [9]. Como nuestro diseño no deja de ser un prototipo, se tratará de utilizar lenguajes de programación donde el desarrollo sea más rápido, aunque la ejecución se vea ralentizada, como en Python.

3.2 SOLUCIONES COMERCIALES

Una decisión que deben tomar muchos equipos de ingeniería en el proceso de diseño es la de Make-or-buy. Esto se refiere a decidir si es más conveniente desarrollar una solución de acuerdo con los requerimientos del proyecto o si es mejor comprar una solución ya desarrollada por otra empresa. Las ventajas de comprar sistemas ya hechos normalmente incluyen una mayor rapidez en el proceso de desarrollo, pero suelen ser más caras en el largo plazo y no permiten a los ingenieros diseñar los sistemas con unas especificaciones exactas, sino que se tienen que amoldar a lo que existe en el mercado.

3.2.1 BMS

En la actualidad existen numerosos proveedores de sistemas BMS comerciales con distintos grados de complejidad. Se empezará estudiando soluciones como las de LithiumBalance o ZEVA (Zero Emission Vehicles Australia). Ambas son empresas que comercializan soluciones BMS.

LithiumBalance ofrece varias soluciones pensadas para distintos usos concretos. La que más nos interesa es el producto s-BMS ya que está pensado para vehículos eléctricos y cumple nuestros requisitos. El sistema tendría una topología modular que consistiría en conectar una unidad central llamada BMCU (Battery Management Control Unit) y varias LMU (Local Management Unit). En las tablas a continuación quedan resumidas las características de este sistema para poder compararlo con otras soluciones:

Características BMCU		**Características LMUs**	
Max pack Voltage	1000 V	**Max Numero LMUs**	32
Max pack Current	2000 A	**Max num celdas**	8
Comunicaciones	SI – CAN Bus	**Consumo eléctrico**	< 20 mA
Control cargador	SI	**Corriente balanceo**	800 mA/celda
Control contactor	SI		

Tabla 3. Características del LithiumBalance s-BMS. [11]

El coste estimado del sistema BMS de LithiumBalance es de aproximadamente 300 € para la unidad central (BMCU) y de 4 x 80 € para los módulos LMU, llegando a un coste total de aproximadamente 620 €.

Por otro lado, ZEVA ofrece algunas soluciones que son de gran interés para el proyecto. Entre estas soluciones destacan los módulos BMS de 12 y 24 celdas. Estos módulos BMS utilizan el protocolo de comunicaciones CAN para transmitir información sobre la tensión de cada celda y permiten aplicar balanceo. A continuación, se muestran las características de estos módulos resumidas para poder comparar con otras alternativas.

Características BMSv3 (ZEVA)	
Max pack voltage	700 V

Max pack current	2000 A
Comunicaciones	CAN bus
Control cargador	NO
Control contactor	NO

Tabla 4. Características del módulo BMSv3 de ZEVA. [12]

El coste estimado para este sistema BMS es de 3 x 115 €, aproximadamente 345€. No obstante, habrá que tener en cuenta que se tendrían que desarrollar elementos aparte para gestionar la carga de las baterías y el control del contactor.

3.2.2 SISTEMA DE ADQUISICIÓN DE DATOS

Encontrar una solución comercial para el sistema de adquisición de datos es más complicado ya que para este sistema se busca obtener la mayor flexibilidad y que se amolde a los requerimientos del proyecto de la mejor forma posible. No obstante, existen una serie de soluciones comerciales que pueden ser de gran utilidad.

Un ejemplo de un producto que podría ser de utilidad es el DL1 PRO. Se trata de un datalogger profesional que permite guardar datos de todo tipo. Cuenta con un módulo GPS, acelerómetros y varios canales de comunicación CAN. A pesar de ser un excelente dispositivo para recopilar los datos de la moto, cuenta con algunas desventajas, como un tamaño poco manejable para un vehículo tan pequeño, falta de conectividad a internet y falta de flexibilidad para el tratado de datos.

Figura 11. Módulo recopilador de datos DL1 PRO.

A continuación, se muestra un resumen de las especificaciones técnicas del DL1 Pro.

Características DL1 Pro	
Fuentes de datos	1x CAN, GPS, Giroscopio, Acelerómetro, 12x Inputs Analógicos
Peso	660 g
Guardado de datos	Memoria SD (max. 32 GB)

Tabla 5. Resumen características del data logger DL1 Pro. [13]

Otra opción comercial que se ha tenido en cuenta es el uCAN. Este dispositivo es mucho más manejable en cuanto a tamaño que el DL1 Pro, pero sigue sin proporcionar la conectividad a internet que tan útil sería para desarrollar el proyecto de innovación.

Figura 12. Módulo recopilador de datos uCAN

A continuación, se resumen las principales especificaciones técnicas de esta posible solución comercial.

Características uCAN	
Fuentes de datos	2x CAN, GPS, Acelerómetro, 8x Inputs Analógicos
Peso	150 g
Guardado de datos	Memoria integrada (2 GB)

Tabla 6. Resumen características del data logger uCAN. [14]

Una vez analizadas estas soluciones comerciales y viendo qué características se demandan de estos dispositivos en el mundo de la competición, el proyecto pasará a la fase de definición del trabajo.

Capítulo 4. DEFINICIÓN DEL TRABAJO

En el capítulo anterior se han analizado varias soluciones comerciales para los problemas planteados en el trabajo. No obstante, se ha podido comprobar que para conseguir unos sistemas que se ajusten perfectamente a los requerimientos de la competición de Motostudent habrá que desarrollar algunas partes. Además, los sistemas que se elijan deberán configurarse e instalarse en la moto.

4.1 OBJETIVOS

El principal objetivo del proyecto es desarrollar y construir dos de los sistemas de una moto eléctrica. El primer sistema es el sistema de gestión de baterías y el segundo el de adquisición de datos. Puesto que este objetivo es demasiado poco concreto para determinar las características de los sistemas, se han definido 3 características que serán claves en el desarrollo de los sistemas:

4.1.1 SEGURIDAD

Este es uno de los pilares más importantes de los sistemas, especialmente del BMS. El proyecto trata de diseñar sistemas de un vehículo eléctrico que pilotará una persona por lo que es esencial tener en todo momento presente cualquier riesgo de seguridad que se presente. Se tendrá en cuenta en todo momento que se cumplen los niveles de aislamiento adecuados y se asegurará que el software es suficientemente robusto.

4.1.2 MODULARIDAD

Los elementos de cada sistema deben ser fácilmente reemplazables. Se utilizarán conectores siempre que sea posible (en lugar de soldaduras) y se emplearán métodos de fijación que permitan realizar cambios con facilidad (tornillos en vez de pegamentos o remaches). Se ha decidido incluir esta característica como una de las más importantes porque la moto eléctrica

que se va a diseñar se utilizará en el futuro para probar versiones mejoradas de los sistemas desarrollados en esta edición.

4.1.3 FLEXIBILIDAD

Esta característica está especialmente enfocada al desarrollo del software. Siempre que se pueda, será preferible elegir un diseño en el que se puedan cambiar parámetros para modificar el comportamiento de la moto. La idea es que se pueda personalizar el control de la moto a las distintas situaciones en las que se va a encontrar. No será igual el funcionamiento en la competición que en un día de entrenamientos o de pruebas técnicas. Además, esta flexibilidad permitirá adaptar a la moto a futuras ediciones de la competición en las que es posible que cambien ciertos aspectos del reglamento.

4.2 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

El trabajo abarca desde las primeras fases de diseño hasta la construcción de los propios sistemas, por lo que una planificación temporal del trabajo es esencial. Además, estos sistemas se tienen que integrar con varios otros sistemas, lo que hace que la planificación y la coordinación con el resto de los departamentos del ISC sea fundamental. A continuación, se muestran las fechas más significativas para el trabajo desde el punto de vista del desarrollo de la parte eléctrica de la moto.

31-10-2019. MSE Entrega 1 – Desarrollo conceptual.

31-11-2019. MSE Milestone 1 – Esquema eléctrico

29-02-2020. MSE Milestone 2 – Especificaciones batería

28-02-2021. MSE Milestone 3 – Ensamblaje batería

31-03-2021. MSE Special Milestone 4 – Pruebas del powertrain eléctrico

31-04-2021. Primeras pruebas en circuito

15-07-2021. Evento Final

Teniendo en cuenta estas fechas se ha diseñado un cronograma que proporcione una planificación temporal del proyecto.

	2019					2020											2021						
	Ago	Sep	Oct	Nov	Dic	Ene	Feb	Mar	Abr	May	Jun	Jul	Ago	Sep	Oct	Nov	Dic	Ene	Feb	Mar	Abr	May	Jun	Jul
Definición conceptual																								
Esquemas eléctricos																								
Selección componentes																								
Lectura de BMS																								
Actuación del BMS																								
Guardado de datos BMS																								
Visualizado en tiempo real																								
Construcción pack baterías																								
Pruebas de los 2 sistemas juntos																								
Pruebas actuación BMS																								
Diseño final en PCB																								
Pruebas finales PCB																								

Tabla 7. Planificación temporal del proyecto.

Es importante resaltar que el desarrollo del proyecto se ha visto afectado por la pandemia del covid-19. Es por esto por lo que entre marzo de 2020 y septiembre de 2020 el progreso se ha visto ralentizado significativamente. Las fechas propuestas inicialmente por la organización también se han visto retrasadas por lo que el cronograma mostrado anteriormente no se ajusta a la planificación ajustada sino a la planificación a febrero de 2021.

Además de una planificación temporal del trabajo, es de vital importancia establecer un presupuesto para los distintos sistemas. Este presupuesto se debe adecuar a la capacidad financiera del ICAI Speed Club para que el proyecto sea viable. Con el objetivo de asegurar financiación de los distintos patrocinadores se realiza una estimación inicial del coste del proyecto desglosado por departamentos. En concreto, vamos a analizar el coste de los componentes del departamento de powertrain, encargado de el diseño y construcción de los

sistemas de alta tensión (acumulador, inversor, motor) y de baja tensión (BMS, sistema de adquisición de datos, sensores y protecciones).

Para decidir el presupuesto necesario para el departamento se ha utilizado una aproximación del coste de los componentes y se ha tenido en cuenta que se pedirán más de los estrictamente necesarios para tener repuestos en caso de que alguno deje de funcionar. Además, es importante tener en cuenta que ciertos componentes no tienen que ser comprados, como el motor (proporcionado por los organizadores de la competición) o el inversor (ya comprado en la edición anterior).

Descripción	Coste/ud.	Cantidad	Total
Celdas	3 €	1.100	3.300 €
Módulos BMS	120 €	4	480 €
Microcontroladores	20 €	4	80 €
Componentes electrónicos	200 €	1	200 €
Materiales caja	500 €	1	500 €
Soldadores	250 €	2	500 €
Cableado	200 €	1	200 €
PCB	30 €	4	120 €
Refrigeración	20 €	6	120 €
Total		5.500 €	

Tabla 8. Planificación económica del proyecto.

En esta primera planificación del proyecto se ha obtenido un coste total de 5.500 €. Para establecer el presupuesto estimado del departamento se ampliará este coste estimado en un 20 %. Este incremento en el coste esperado del proyecto se justifica en concepto de costes

de envío, componentes extra y cambios en las decisiones de diseño. Tras aplicar este 20 % extra se ha determinado que el departamento necesitará aproximadamente 6.600 € para llevar a cabo el proyecto.

En la siguiente figura se puede ver de manera visual cómo se reparten los costes del proyecto en los distintos sistemas o componentes. Claramente el grueso del coste se encuentra en la compra de celdas para construir el acumulador.

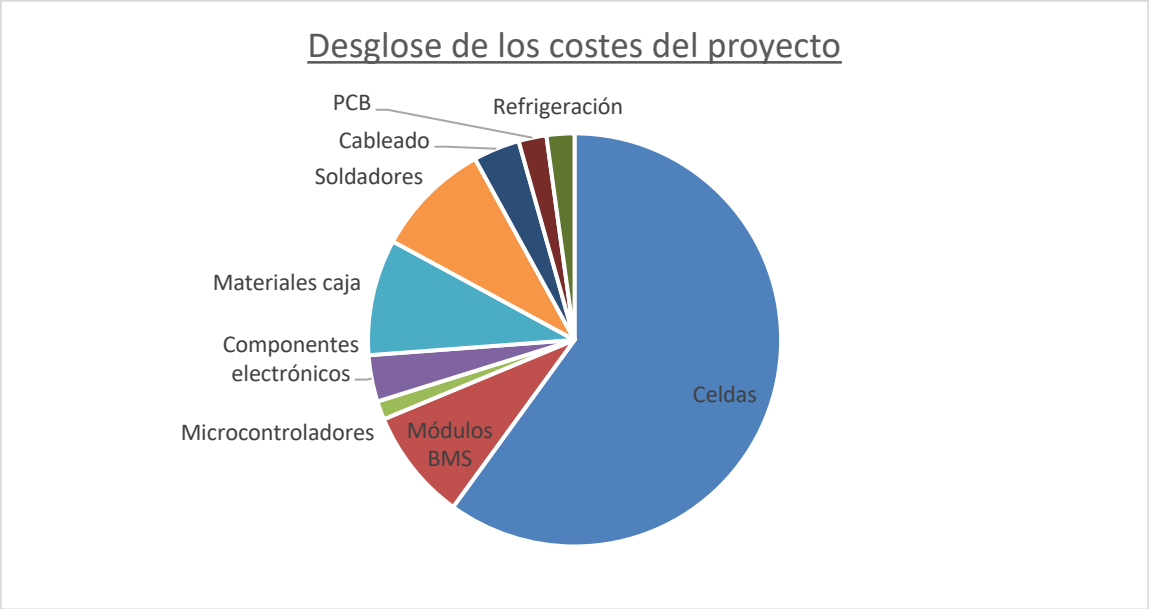


Figura 13. Desglose de los costes del proyecto.

4.3 ALINEACIÓN CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE (ODS)

Aunque el proyecto tiene como principal objetivo participar en una competición de motos eléctricas y obtener la mejor marca posible, el desarrollo de este está también alineado con una serie de Objetivos de Desarrollo Sostenible, o ODS. Estos objetivos, fijados desde la Organización de Naciones Unidas, buscan concretar una serie de metas que aseguren la prosperidad y sostenibilidad del planeta. De estos 17 objetivos, hay 3 que son especialmente importantes para el proyecto:

- **Objetivo 7. Energía asequible y no contaminante.** En el proyecto se desarrollarán tecnologías que permitan gestionar la carga de un pack de baterías de forma segura y fiable. Los avances en tecnologías de gestión de carga son de gran importancia para crear acumuladores asequibles que hagan viable el uso de fuentes renovables y aseguren la estabilidad en la red eléctrica.
- **Objetivo 11. Ciudades y comunidades sostenibles.** Este es el objetivo de desarrollo sostenible más importante para el proyecto. Las soluciones de movilidad eléctrica son claves para asegurar la sostenibilidad en las grandes ciudades ya que otras alternativas como los motores de combustión interna son causantes de los altos niveles de contaminación que tanto afectan a ciudades como Madrid. El proyecto se centrará en demostrar la viabilidad de los vehículos eléctricos como una alternativa a los de combustión interna.

Capítulo 5. DESARROLLO DEL HARDWARE

La parte del hardware de los dos sistemas a desarrollar en el proyecto es la parte física e incluye desde los componentes electrónicos que permiten la lectura de las tensiones de las celdas a los microcontroladores y procesadores donde se ejecutará la lógica que monitoriza y guarda los datos recopilados. Para hacerse una idea de los componentes que habrá que diseñar o comprar, es de ayuda el siguiente esquema en el que se muestra una visión global de los dos sistemas y cómo están interconectados.

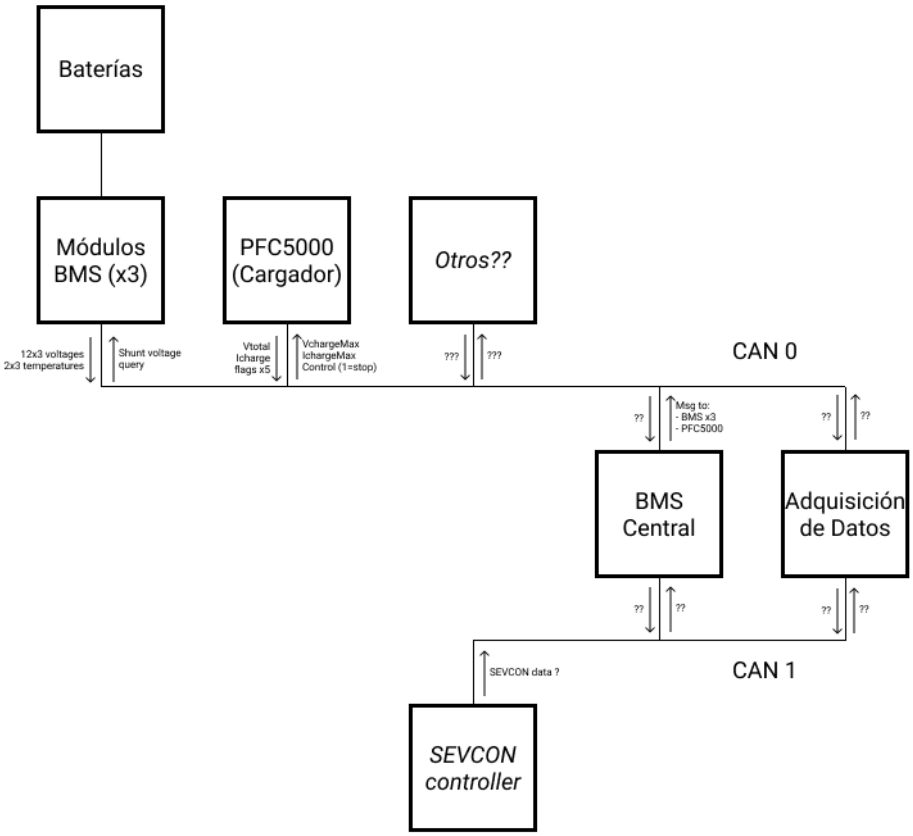


Figura 14. Vista general del sistema BMS y del sistema de adquisición de datos.

En este esquema se muestra cómo en la moto se utilizarán dos bus CAN en paralelo. Esto es porque algunos subsistemas requieren una velocidad de transmisión CAN de 500 kbps (BMS y cargador) mientras que otros operan a una velocidad de transmisión de 1 Mbps (controlador SEVCON). Además, esta separación nos permite separar estos dos sistemas por completo, evitando así interacciones inesperadas entre el controlador y el sistema BMS. Tanto el controlador central del BMS como el sistema de adquisición de datos tendrán acceso a ambos buses y la información que se comparte a través de ellos.

5.1 HARDWARE DEL BMS

Para construir el sistema de gestión de baterías se ha recurrido finalmente a una mezcla de soluciones comerciales (módulos ZEVA BMSv3) y de soluciones diseñadas a medida (PCB de comunicaciones CAN).

Se muestra un esquema de todos los componentes de alta tensión a continuación. La caja de baterías irá conectada a otros elementos del sistema de alta tensión como el contactor, el controlador (o inversor) y el motor. Recordemos que el principal objetivo del sistema BMS es monitorizar la salud de las baterías y actuar sobre el contactor en caso de detectar algún problema. Para hacerse una idea de cómo está dispuesto el sistema de alta tensión del vehículo se muestra el siguiente esquema.

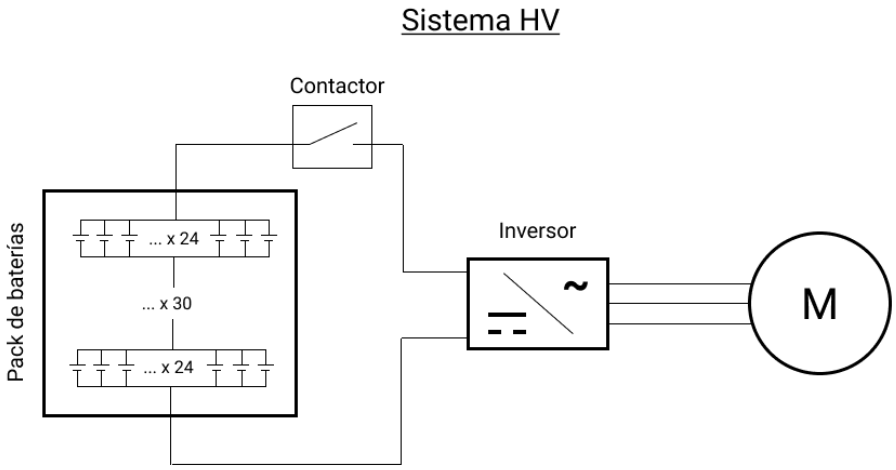


Figura 15. Esquema del sistema de alto voltaje de la moto

5.1.1 ACUMULADOR

Antes de empezar a diseñar un BMS hay que tener muy en cuenta sobre qué tipo de baterías va a actuar y cómo están dispuestas. Para esta edición de Motostudent se ha optado por utilizar celdas de ion de litio en configuración cilíndrica 18650. La razón por la que se ha elegido este tipo de celda y en esta disposición se puede ver con claridad tras analizar las siguientes tablas:

Comparación de distintas químicas de baterías			
Tipo de química	**Li-ion (Lithium-Ion)**	**Ni-MH (Nickel Metal Hydride)**	**Pb-A (Lead-Acid)**
Densidad energética	100 – 180 Wh/kg	40 – 120 Wh/kg	30 – 40 Wh/kg
Durabilidad	500 – 15 000 ciclos	500 – 1 000 ciclos	500 – 800 ciclos
Eficiencia típica	95 – 99 %	65 – 80 %	70 – 92 %
Coste	$ 0.50 – $ 2.50 /Wh	$ 0.30 – $ 0.60 /Wh	$ 0.15 – $ 0.30 /Wh

Tabla 9. Resumen de las distintas químicas de batería consideradas. [15]

Aunque las baterías de ion de litio tienen un coste superior a los otros dos tipos, su mayor eficiencia y densidad energética las hacen idóneas para el uso que se les va a dar en un vehículo eléctrico. La durabilidad de estas es algo a tener en cuenta en el desarrollo de un vehículo eléctrico comercial, pero al estar diseñando una moto de competición, 500 ciclos es más que suficiente. La siguiente decisión a tomar será la disposición física de las mismas. Las opciones que se barajan son configuración prismática, pouch o cilíndrica.

Comparación de las distintas configuraciones de Li-ion		
Prismática	**Pouch**	**Cilíndrica**
		
Uniones fáciles Pack seguro No estandarizado Poco flexible	Eficiente uso del espacio No estandarizado Peligrosas	Tamaño estándar Flexibilidad de disposición Relativamente seguras

Tabla 10. Resumen de los distintos formatos de batería Li-ion considerados

La elección final fue usar celdas cilíndricas por su bajo coste y gran grado de estandarización. Además, permiten diseñar un pack de baterías con prácticamente cualquier forma. El principal inconveniente de esta disposición es el trabajo que requiere construir el pack de baterías final ya que hay que soldar un gran numero de uniones. En concreto el modelo elegido fue el Sony VTC6 con configuración 18650 [16].

Las características más relevantes de esta celda en concreto se resumen a continuación. Es muy importante conocer los límites de operación de las baterías porque el BMS se tiene que encargar de mantener la tensión y temperatura dentro de los rangos adecuados.

Celdas individuales

Temperatura de operación	Carga	0°C ~ 45°C
	Descarga	-20°C ~ 60°C
Capacidad	Mínima	3 000 mAh
	Nominal	3 120 mAh
Tensiones	Mínima - Máxima	2.5 V – 4.2 V
	Nominal	3.6 V
Intensidades	Carga (recomendada)	0.2 C = 0.6 A
	Descarga máxima (continua)	10 C = 30 A
	Descarga máxima (19s)	18.3 C = 55 A

Tabla 11. Características de la celda usada (SONY US18650 VTC6)

Estas celdas se dispondrán en módulos más pequeños para facilitar su manejo. Se ha optado por hacer módulos dispuestos en paralelo que a su vez se conectarán en serie hasta alcanzar la tensión nominal del pack completo. Esto hace que cada módulo individual sea más seguro de manejar ya que no se alcanzarán tensiones elevadas. Cada paralelo contiene 24 celdas de tamaño 18650. En total se conectarán 30 de estos paralelos en serie, dando como resultado una tensión nominal de 30 * 3.6 = 108 V. De esta manera la caja de baterías una vez ensamblada tendrá las siguientes características.

Caja de baterías		
Temperatura de operación	Carga	0°C ~ 45°C
	Descarga	-20°C ~ 60°C
Capacidad	Nominal	72 Ah (= 3 x 24)
Tensiones	Mínima - Máxima	75 V – 126 V

	Nominal	108 V (= 3.6 x 30)
Intensidades	Carga (recomendada)	0.2 C = 14.4 A
	Descarga máxima (continua)	10 C = 720 A
	Descarga máxima (19s)	18.3 C = 1320 A

Tabla 12. Características de la caja de baterías completa

5.1.2 MÓDULOS BMS

Una vez se conoce cómo va a ser el acumulador se puede pasar a diseñar el sistema de BMS. Primero hay que analizar las distintas topologías de sistemas BMS entre las que se puede elegir. Las opciones barajadas se muestran en la siguiente tabla.

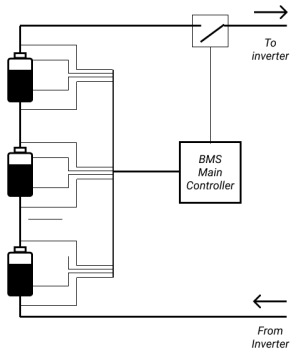
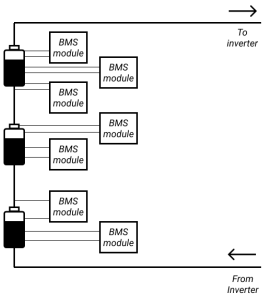
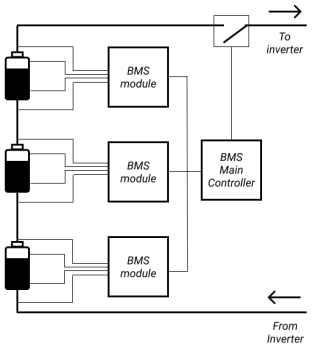
BMS centralizado	BMS distribuido	BMS modular
		
Sólo un controlador Muchos cables Software es sencillo (no hay comunicación) Hardware es complejo (muchos cables)	Un controlador por cada celda Difícil de coordinar en conjunto Montaje sencillo	Controladores en cada módulo Requiere de un protocolo de comunicación Montaje sencillo

Figura 16. Resumen de las distintas topologías de BMS

Tras analizar las distintas opciones se ha decidido que la solución modular es la que mejor se ajusta al proyecto. Por un lado, ofrece bastante comodidad en el montaje, evitando los problemas de un sistema centralizado, y a la vez permite coordinar y obtener información de todas las celdas de una forma centralizada.

Tras analizar las distintas opciones comerciales se ha optado por comprar los módulos BMSv3 de la empresa ZEVA. La principal ventaja de estos módulos es la flexibilidad que aporta al sistema ya que permiten monitorizar cajas de hasta 192 baterías. Son relativamente fáciles de instalar y establecer una comunicación a través de CAN bus con los mismos no supone ningún problema. Además, vienen con un sistema de balanceo integrado. Como ventaja adicional, permiten la lectura de temperatura en 2 puntos por cada módulo, algo que es obligatorio realizar según el reglamento de la competición.

Figura 17. Módulo BMSv3 de ZEVA. [12]

Los módulos BMS tienen un montaje bastante sencillo, ilustrado muy bien en la siguiente imagen del manual. Además de las conexiones con las celdas, cada módulo necesita una alimentación a 12 V y una conexión a un bus CAN. Cada módulo ofrece además una conexión "shield" de apantallamiento para evitar que se induzcan interferencias electromagnéticas en el cable usado para la comunicación CAN.

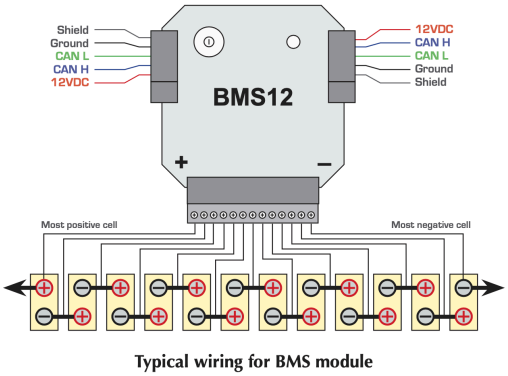


Figura 18. Esquema del conexionado de un único módulo BMS. [12]

La caja de baterías que pretende monitorizar este BMS tiene 30 celdas en serie. Por esta razón se necesitarán 3 módulos BMS de 12 conexiones, de los cuales 6 conexiones no serán útiles. Estas posiciones se cortocircuitarán y por lo tanto se obtendrá una lectura de 0 V. A continuación, se muestra cómo quedan las conexiones de los módulos BMS entre ellos y a las celdas. Como se puede apreciar, se han utilizado conectores siempre que fuera posible para permitir desmontar la caja o cambiar componentes fácilmente.

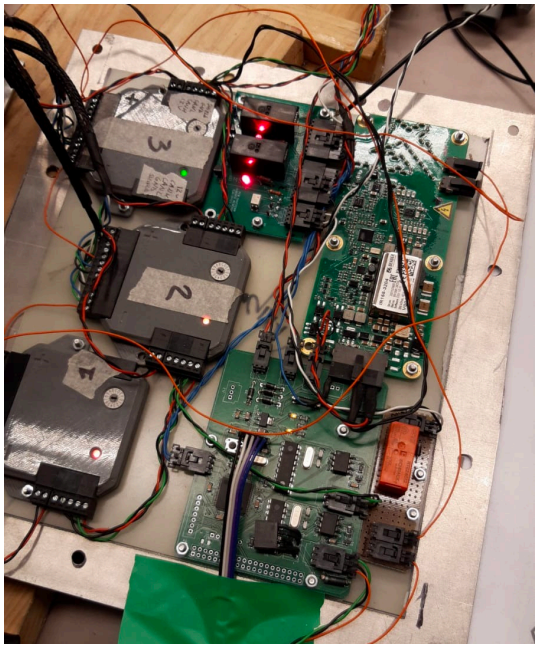


Figura 19. Disposición de los módulos BMS y el conexionado.

En la imagen anterior se pueden ver los tres módulos BMS distribuidos, numerados del 1 al 3. Además, se ven las conexiones de los módulos a las celdas y al bus CAN que los une. Las PCB de la derecha son la PCB de comunicaciones desarrollada en este trabajo y la PCB de protecciones.

5.1.3 CARGADOR

Aunque el cargador no es estrictamente una parte del sistema de gestión de baterías, en este proyecto se ha tomado la decisión de controlar el cargador desde el sistema BMS. La razón es que así se podrá monitorizar la salud de las baterías mientras se carga, evitando que haya alguna celda fuera de sus valores nominales. Si el sistema BMS detectara cualquier problema sería capaz de parar la carga de forma automática.

Con estas consideraciones en mente se ha elegido el cargador PFC 5000 de ElCon. Este cargador tiene una serie de características que lo hacen una opción muy interesante para el proyecto. En primer lugar, tiene una eficiencia superior a la mayoría de los cargadores (>93 %) lo que disminuye las pérdidas por calor y por lo tanto disminuye riesgos de incendios eléctricos. En segundo lugar, está sellado y cuenta con una certificación IP46, lo que lo hace idóneo para trabajar en el box. En tercer lugar, permite cargar baterías con un amplio rango de tensiones. Por último, admite comunicación CAN para ajustar las curvas de carga. Esta última característica es muy interesante ya que nos permite controlarlo desde el propio sistema BMS.

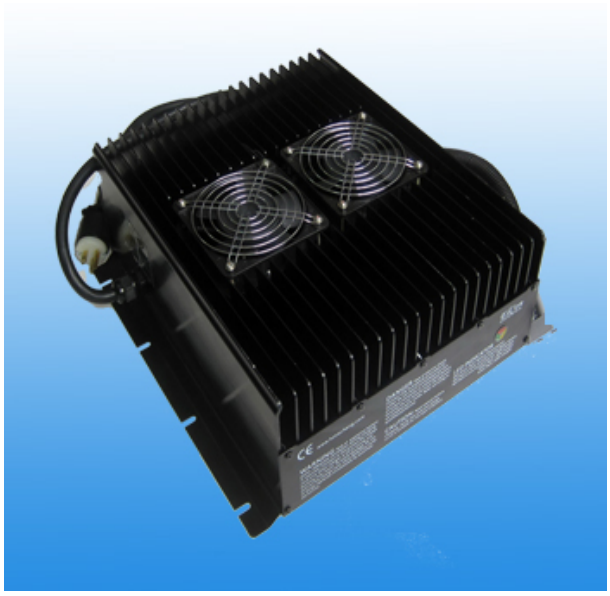


Figura 20. Cargador PFC 5000. [17]

El conexionado del cargador es muy sencillo ya que basta con conectar los terminales positivo y negativo correctamente a la batería y conectar los 2 cables de comunicación CAN (CAN H y CAN L). No obstante, para los terminales positivo y negativo de la batería hay que tener en cuenta los niveles de aislamiento necesarios para evitar cualquier derivación, ya que se trabajará con tensiones superiores a 100 voltios. Por eso las conexiones se han realizado con un conector industrial que proporciona el aislamiento necesario. En concreto, el modelo elegido soporta tensiones nominales de 1kV y corrientes de hasta 400 A [18]. Este sobredimensionamiento de los conectores nos asegura que los ingenieros trabajando en la moto estén siempre seguros.

Figura 21. Conector utilizado para conectar el cargador a la caja de baterías.

5.1.4 INTEGRACIÓN CON EL CONTROLADOR

Como se ha visto en el esquema del sistema de alta tensión, un elemento de gran importancia es el controlador. Este componente se encarga de accionar el motor y, para ello, transforma la energía eléctrica de las baterías (corriente continua), en una serie de corrientes trifásicas que mueven el motor PMAC que la competición proporciona a todos los equipos. El funcionamiento de este elemento en detalle queda fuera de los objetivos de este trabajo, pero aun así es conveniente conocer sus funciones y modo de trabajo ya que el BMS tendrá que coordinar con el mismo la activación del contactor.

El modelo elegido para la competición es el SEVCON Gen4 (Size 6). La principal función de este controlador será tomar la lectura analógica del puño acelerador y actuar sobre el motor de acuerdo con los parámetros con los que esté configurado. Es el encargado de transformar la energía eléctrica DC proveniente de la batería a la energía trifásica con la que opera el motor brushless utilizado. Además de esta función, el controlador transmitirá información de gran utilidad en tiempo real a través del bus CAN. Algunos de los datos de mayor interés que se esperan recopilar incluyen: Tensión de la batería, intensidad de entrada, corrientes de magnetización, tensión del condensador de entrada, temperatura del radiador y revoluciones del motor.

Figura 22. Controlador SEVCON Gen4 Size 6 usado en el proyecto. [19]

El controlador SEVCON cuenta con 2 entradas de alimentación, 1 para la electrónica y otra para la potencia. El manual de operación establece que primero se debe alimentar por la entrada de alimentación para electrónica. Esto es porque la entrada de potencia cuenta con un condensador de gran capacidad en paralelo. Para evitar grandes corrientes de pico, este condensador se carga a través de la alimentación para la electrónica y una vez está al 90% de su tensión final se activa un contactor, permitiendo la entrada de potencia al controlador. Aunque se podrían colocar 2 contactores en serie (uno controlado por el sistema BMS y otro por el controlador), se ha preferido utilizar 1 solo contactor y coordinar la actuación de ambos sistemas sobre el mismo. De esta manera se ahorra una gran cantidad de espacio y peso que es de gran valor en una moto de competición.

Para activar el contactor, el BMS central estará recibiendo continuamente información del SEVCON sobre si éste está preparado para activar el contactor (si la tensión del condensador de precarga supera el 90% de la final y si no se detecta ningún fallo en el puño acelerador). La lógica detrás de este algoritmo se tratará con más detalle en el capítulo siguiente.

5.1.5 BMS CENTRAL

El BMS central es el encargado de interpretar las lecturas de tensión y temperatura de cada uno de los módulos BMS, de controlar el cargador en caso de que este este conectado y de activar el contactor (solamente si todos los valores están dentro de los límites de operación).

Para realizar las tareas descritas anteriormente se ha decidido usar un microcontrolador. Dentro de la multitud de microcontroladores disponibles se utilizará un Arduino Mega Pro.

Las razones por las que se ha elegido este modelo frente a otros microcontroladores más profesionales son por la gran cantidad de librerías y documentación que se puede encontrar en internet. De hecho, como se verá más adelante, el poder contar con una librería de código abierto para comunicarse con el módulo MCP2515 ha permitido avanzar en el proyecto mucho más rápido que si se hubiese tenido que desarrollar esta librería. Además, la plataforma de desarrollo de Arduino, aunque menos profesional, es rápida y fácil de usar. Características muy interesantes para cuando el proyecto lo tenga que retomar otro miembro del ICAI Speed Club en ediciones futuras.

Se ha elaborado esta tabla para resumir las principales características de la placa de desarrollo junto con el microcontrolador usado.

Arduino MEGA 2560 Pro		
Microcontrolador	Modelo	ATmega2560
	Memoria	256 KB
	Frecuencia	16 MHz
Periféricos	SPI	x 1
	Entradas analógicas	x 16 (a 5V)
	Regulador tensión	6V a 9V
	USB-TTL	CH340

Tabla 13. Especificaciones de interés de la placa de desarrollo utilizada. [20]

Puesto que el contactor se activa con 12V de corriente continua y el microcontrolador sólo tiene salidas a 5 V, hará falta usar un relé o transistor para activarlo. Para minimizar las pérdidas eléctricas y optimizar el espacio utilizado se utilizará un transistor MOSFET BS170. El esquema eléctrico del circuito de activación del contactor será como se muestra a continuación.

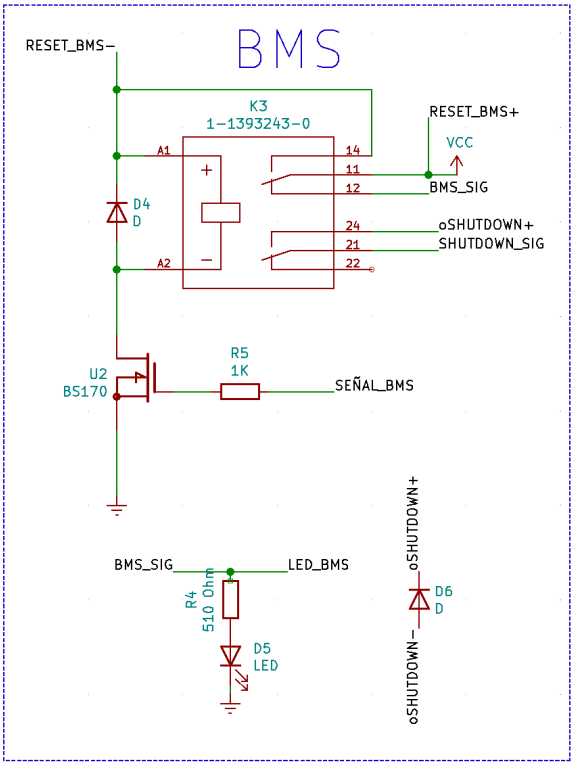


Figura 23. Esquema eléctrico de activación del contactor.

En el esquema se ve cómo la señal proveniente del microcontrolador (SEÑAL_BMS) controla la activación del transistor. Para evitar quemar el pin del microcontrolador se coloca una resistencia de 1 kΩ. Además, se coloca un diodo en paralelo con la bobina del contactor para evitar quemar el transistor cuando se abre el circuito.

Otro elemento de hardware importante para que el BMS central funcione correctamente es la PCB de comunicaciones. Puesto que el microcontrolador empleado no cuenta con un módulo CAN integrado hace falta utilizar un módulo conversor. Los más utilizados son unos módulos conversores de CAN bus a comunicación SPI basados en el chip MCP2515. En un primer momento se probaron las comunicaciones CAN bus usando unos módulos CAN-SPI comerciales. No obstante, estos módulos demostraron tener algunas limitaciones. Por un lado, el conexionado (originalmente usando conectores DuPont) era muy inseguro y acababa dando problemas. Además, no permite aislar las comunicaciones sin usar componentes adicionales.

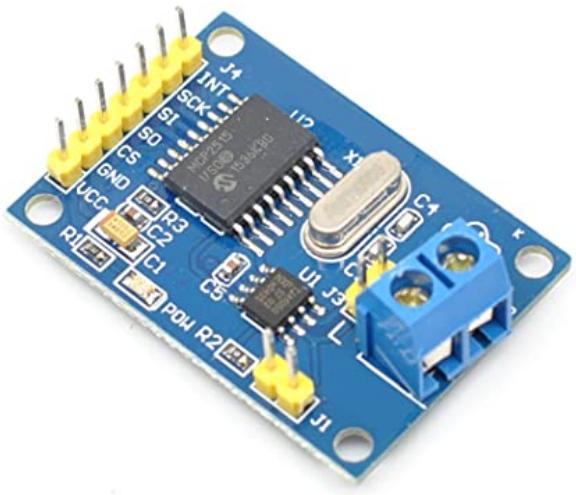


Figura 24. Módulo CAN-SPI basado en el chip MCP2515. [21]

Tras analizar estas limitaciones, se ha decidido diseñar una PCB (Printed Circuit Board) inspirada en el módulo CAN-SPI de la imagen, pero con las especificaciones que requiere el proyecto: aislamiento del bus y conectores seguros. También se ha decidido integrar dos módulos en paralelo en la propia PCB para así poder acceder a los dos buses CAN del proyecto.

La conexión de dos módulos en paralelo se hace usando el mismo bus SPI, pero conectando distintos pines CS a cada módulo. La mejor forma de ver este conexionado es con el esquema mostrado a continuación. Todo lo que se encuentra dentro de las líneas discontinuas se integrará en una única PCB.

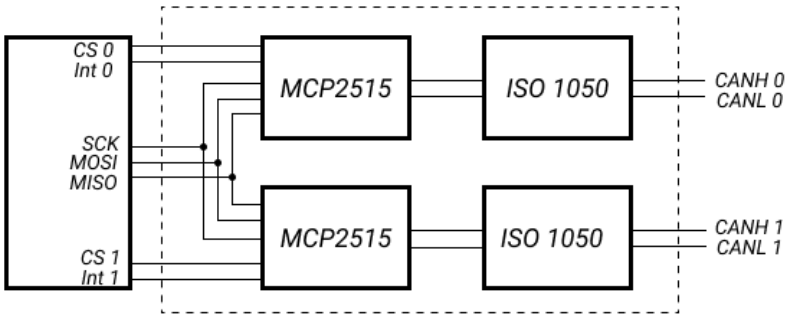


Figura 25. Esquema de los componentes de la PCB de comunicaciones CAN.

En el esquema se puede ver cómo se conectan los dos chips MCP2515 (controladores CAN) a través de una conexión SPI al microcontrolador. Esta configuración del bus SPI para controlar los dos bus CAN a la vez impide obtener mensajes de los módulos MCP2515 de manera simultánea. No obstante, eso no es un problema ya que el bus SPI trabajará a una velocidad muy superior al CAN (10 Mbit/s frente a 500 KBit/s y 250 KBit/s) y además los módulos MCP2515 cuentan con 2 buffers de recepción de mensajes cada uno.

Un componente que aparece nuevo en este esquema es el ISO 1050. Se trata de un transceptor CAN y su función es convertir los pulsos lógicos que recibe del controlador en las tensiones diferenciales que caracterizan la capa física de un bus CAN. La elección del modelo ISO 1050 permite aislar la comunicación CAN. Esto es necesario porque el controlador SEVCON se comunica vía CAN referenciado al borne negativo de la batería y según el reglamento de Motostudent se debe aislar galvánicamente el bus para evitar problemas de seguridad.

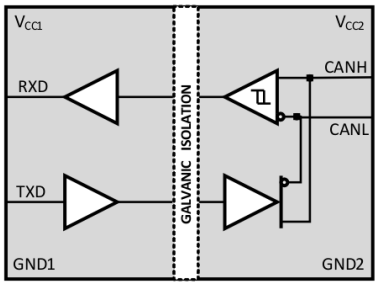


Figura 26. Aislamiento galvánico del transceptor CAN (ISO 1050). [22]

El desarrollo del circuito impreso se ha realizado a través de KiCAD y su fabricación se ha realizado a través de la empresa JLCPCB. Esta empresa china ofrece servicios desde fabricación de PCB hasta montaje de los componentes en la propia PCB. Puesto que los componentes ya habían sido comprados para hacer pruebas, se ha optado por únicamente mandar a fabricar la PCB y realizar el montaje de los componentes en el propio taller del Speed Club. A continuación se muestra una renderización de esta PCB.

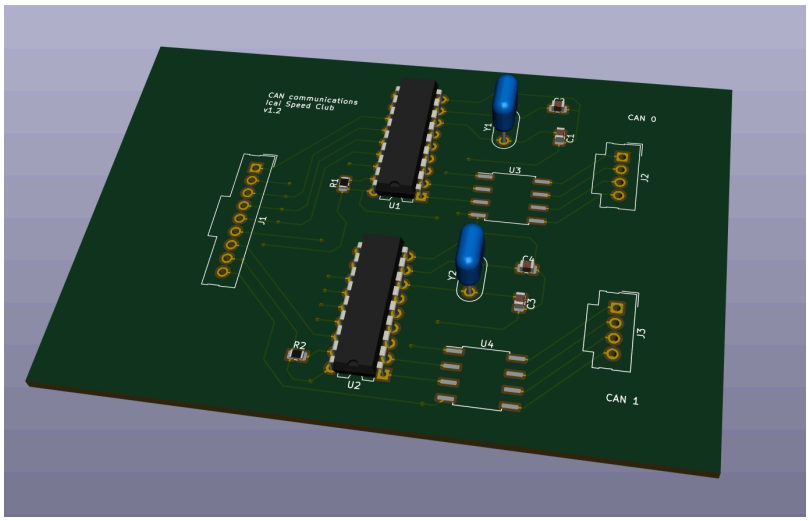


Figura 27. Renderización de la PCB de comunicaciones en KiCAD

5.1.6 ALIMENTACIÓN Y CONEXIONADO

La alimentación de todos estos componentes se realizará a través de otro circuito impreso. A este circuito, diseñado en el propio ISC, nos referiremos como placa de shutdown. Esta placa, cuenta con un conversor DC-DC y un relé para la activación del contactor. Como se ha explicado en los objetivos del trabajo, se harán por conectores todas las conexiones posibles para facilitar el recambio de componentes en caso de fallo.

En la imagen que se muestra a continuación se ve cómo se ha realizado el cableado del sistema BMS a las celdas. Todos los cables están aislados por encima de los requerimientos de la competición y trenzados y apantallados para minimizar las interferencias electromagnéticas que pueden causar las grandes corrientes saliendo de la batería.

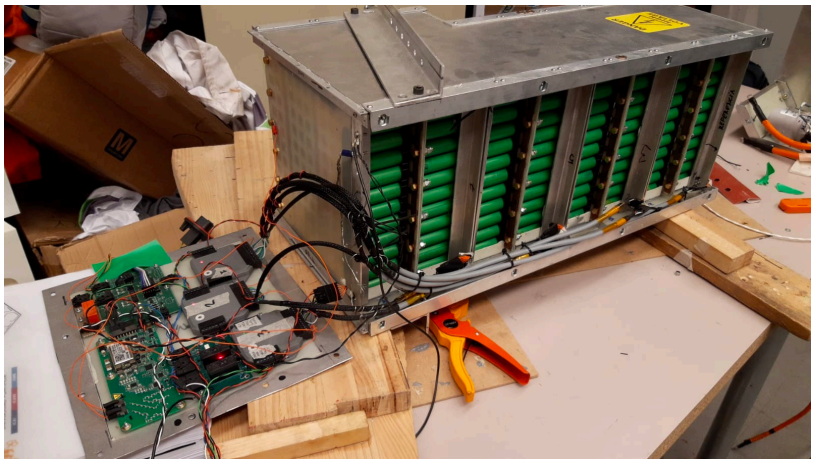


Figura 28. Vista general del conexionado en la caja de baterías.

5.2 HARDWARE DEL SISTEMA DE ADQUISICIÓN DE DATOS

El sistema de adquisición de datos se integrará en paralelo al sistema BMS. Este sistema deberá funcionar de manera independiente ya que se pretende que sea totalmente opcional su integración en el día de la competición. Para conseguir esto, se han tenido en cuenta 2 opciones para la adquisición de datos.

La primera opción consiste en que el sistema BMS vuelque los datos recibidos tanto del controlador SEVCON como de los módulos BMS a través de una conexión UART o similar. Esta opción es interesante ya que se podrían enviar únicamente los datos que sean útiles y con la frecuencia de muestreo que se desee. Sin embargo, esta tarea de volcado añade cierta complejidad al microcontrolador del BMS central, haciendo que el programa a ejecutar sea más complejo. Además, una conexión UART es más vulnerable al ruido electromagnético que un bus CAN y dado que el sistema se va a encontrar en un vehículo eléctrico, esto es algo que hay que tratar de minimizar.

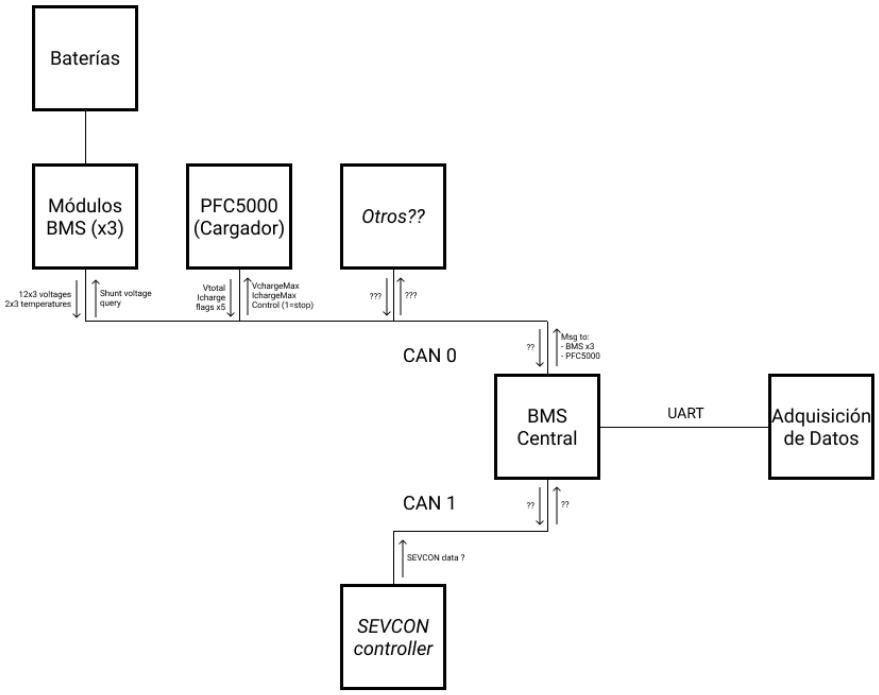


Figura 29. Primera topología considerada para el sistema de adquisición de datos

La segunda opción consiste en instalar el sistema de adquisición de datos de forma completamente paralela al BMS, es decir, leyendo los datos de los 2 bus CAN. Aunque procesar estos mensajes CAN añada complejidad a la recogida de datos, se consiguen 2 sistemas completamente independientes y queda un diseño muy limpio y sencillo del BMS.

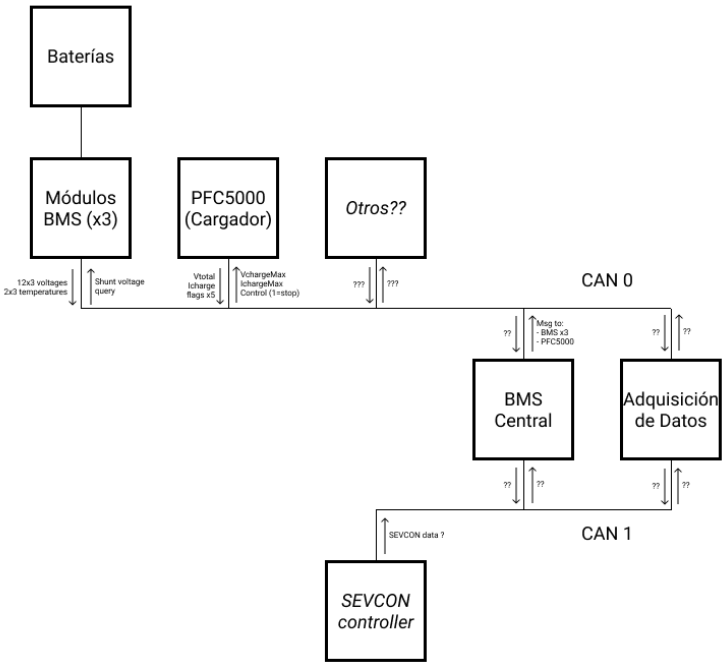


Figura 30. Topología utilizada finalmente para el sistema de adquisición de datos

Otra ventaja de la segunda opción es que las conexiones, tanto del sistema BMS como del sistema de adquisición de datos, a los dos buses CAN se realizarán de la misma manera, a través del circuito impreso diseñado. De esta manera se evita tener que diseñar 2 módulos de comunicación distintos y se simplifica el desarrollo.

Aunque a primera vista el sistema BMS y el sistema de adquisición de datos, parezca que tienen que realizar tareas muy parecidas (leer mensajes CAN e interpretarlos), su función es completamente diferente. Por un lado, el sistema BMS está encargado de encontrar cualquier fallo en el sistema y apagar la moto en caso de emergencia y por lo tanto requiere de un diseño sencillo, rápido y robusto. Por otro lado, el sistema de adquisición de datos es totalmente accesorio y está pensado para recopilar información, mostrarla en tiempo real y guardarla para un análisis posterior.

Para cumplir con estas funciones se ha determinado que el mejor sistema es una Raspberry Pi. Este modelo de ordenador es pequeño, barato, eficiente y cuenta con todas las

salidas/entradas que se necesitan en el proyecto: Un bus SPI, una salida HDMI y conectividad wifi (para facilitar el traspaso de datos). El único inconveniente de este sistema es que habrá que buscar o desarrollar unas nuevas librerías para la comunicación por bus CAN.

Capítulo 6. DESARROLLO DEL SOFTWARE

Tanto el sistema de BMS como el de adquisición de datos requieren de una parte de desarrollo de software. En este capítulo se tratará la recepción de los mensajes, su procesamiento y su guardado para su posterior análisis.

6.1 SOFTWARE DEL BMS

6.1.1 COMUNICACIÓN CON MÓDULOS BMS

Antes de desarrollar el software del sistema BMS hay que recordar la disposición de los elementos de este y su función. En este esquema se muestran los elementos del sistema junto con la información que envía de cada uno.

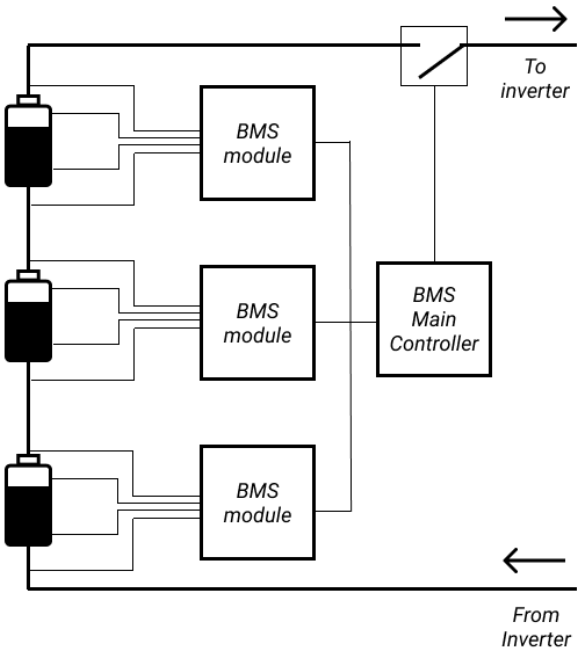


Figura 31. Diseño del sistema BMS

Los módulos BMS comprados se comunican a través un bus CAN a 250 kbps. La especificación del protocolo CAN en concreto es CAN 2.0B o CAN extended, el cual usa direcciones de 29 bits. En la operación de los módulos intervienen 5 paquetes de datos. El primer paquete se envía desde el controlador principal, en nuestro caso, el Arduino Mega Pro. Los otros 4 paquetes contienen la información que han recopilado los BMS y se envía a modo de respuesta al primer paquete. Los detalles de estos paquetes se pueden ver en la siguiente tabla.

Protocolo BMSv3 CAN			
Nº	Dirección	ID	Información
0)	Controlador → Módulo	Base ID	2 bytes → Tensión de balanceo
1)	Módulo → Controlador	Base ID + 1	8 bytes → Tensiones de 4 celdas
2)	Módulo → Controlador	Base ID + 2	8 bytes → Tensiones de 4 celdas
3)	Módulo → Controlador	Base ID + 3	8 bytes → Tensiones de 4 celdas
4)	Módulo → Controlador	Base ID + 4	2 bytes → 2 medidas de temperatura

Tabla 14. Mensajes intercambiados entre módulos BMS y BMS central.

En la tabla se muestra el intercambio de mensajes entre un módulo BMS cualquiera y el controlador principal. Para distinguir entre los distintos módulos se selecciona una dirección base distinta para cada uno. Los módulos BMS disponen de una rueda de selección que permite dar una dirección base distinta a cada módulo empezando en 300 y en incrementos de 10 (hasta 16 módulos). Utilizando este sistema se ha determinado que para el primer módulo BMS, la dirección base será igual a 300 (0x12C). Para el segundo, será 310 (0x136), y para el tercero será 320 (0x140).

La razón por la que hay que recibir varios paquetes de cada módulo radica en que el bus CAN sólo admite paquetes de hasta 8 bytes. Esto no es suficiente para enviar toda la

información de las celdas con la resolución necesaria por lo que hay que usar varios mensajes.

La interpretación de los paquetes recibidos es relativamente sencilla y solo requiere de unos cálculos rápidos en el microcontrolador. Los datos de tensión se obtienen a partir de 2 bytes expresados en formato "big endian" y tienen como unidades milivoltios. Este formato especifica que los bytes más significativos se guardan en la posición de memoria más pequeña. A continuación, se muestra un ejemplo de cómo funciona este formato y cómo se preparan los datos para su envío o recepción.

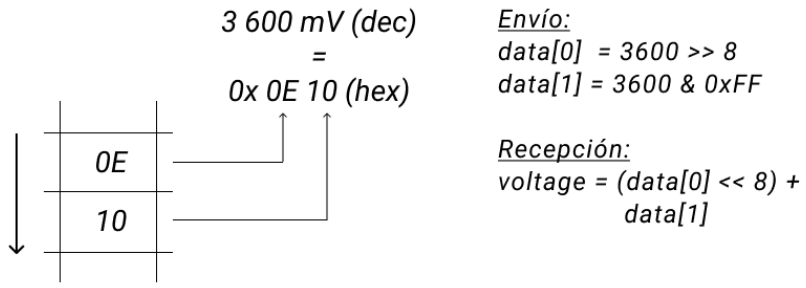


Figura 32. Explicación del formato "big endian" y cómo tratar los datos de tensión en la comunicación CAN.

Puesto que cada dato de temperatura viene dado por un único byte, directamente en grados Celsius, las operaciones a realizar para su interpretación son aún más sencillas. La única operación que tiene que realizar el microcontrolador es deshacer el offset de 40 grados incluido por los módulos BMS. La justificación de este offset es que al incluirlo se pueden expresar temperaturas de hasta 40 grados bajo cero (data es un byte sin signo).

$$Temperatura = data - 40$$

Ecuación 1. Interpretación de los datos de temperatura en la comunicación CAN.

Una vez se conoce cómo se envían y cómo se interpreta cada uno de los datos recibidos de los módulos BMS, su implantación en código no será complicada.

6.1.2 COMUNICACIÓN CON CARGADOR PFC 5 000

El sistema BMS no sólo se encarga de monitorizar la tensión de las baterías, sino que tendrá que coordinar la carga del pack de baterías con el cargador. Con el objetivo de eliminar sistemas innecesarios se ha elegido un cargador producido por ElCon que utiliza comunicación CAN a 250 kbps para recibir instrucciones del BMS. Gracias a esto, se puede conectar al bus CAN ya existente, simplificando mucho el diseño de los buses de comunicación.

La comunicación con el cargador es relativamente sencilla: El sistema BMS deberá enviar un mensaje CAN de manera periódica, cada 1 segundo, con un determinado ID. En este mensaje el BMS comunicará la máxima tensión e intensidad de carga y si el cargador puede empezar a cargar inmediatamente. El mensaje se estructura de la siguiente manera:

Protocolo PFC 5000 CAN		
BMS => Cargador (ID: 0x1806E7F4)		
Posición	**Información**	
Byte 1	Max. Tensión de carga (high byte)	0.1 V/byte Eg. 1201 => 120.1 V
Byte 2	Max. Tensión de carga (low byte)	
Byte 3	Max. Intensidad de carga (high byte)	0.1 A/byte Eg. 253 => 25.3 A
Byte 4	Max. Intensidad de carga (low byte)	
Byte 5	Control	0: Iniciar carga 1: Parar carga

Tabla 15. Estructura del mensaje CAN enviado desde el BMS al cargador.

Como se puede observar en la tabla anterior, se vuelve a utilizar un formato “big endian”, como con los módulos BMS de ZEVA. El cargador también envía de manera periódica, cada 1 segundo, un mensaje CAN para que el BMS conozca su estado de operación. El mensaje

viene estructurado de manera muy similar al que envía el BMS, pero con pequeñas diferencias:

Protocolo PFC 5000 CAN		
Cargador => BMS (ID: 0x1806E7F4)		
Posición	**Información**	
Byte 1	Tensión de carga medida (high byte)	0.1 V/byte Eg. 1201 => 120.1 V
Byte 2	Tensión de carga medida (low byte)	
Byte 3	Intensidad de carga medida (high byte)	0.1 A/byte Eg. 253 => 25.3 A
Byte 4	Intensidad de carga medida (low byte)	
Byte 5	Flags	5 flags explicadas más adelante

Tabla 16. Estructura del mensaje CAN enviado desde el cargador al BMS.

Se puede observar cómo el formato de las tensiones e intensidades leídas es idéntico a como se envían desde el BMS, por lo que se puede seguir utilizando el código para interpretar mensajes en formato "big endian". Además, el cargador proporciona un byte en el que vienen codificadas 5 flags que avisan al BMS si hay algún parámetro mal que impida al cargador iniciar la carga. Estos flags se muestran en más detalle a continuación.

Flags PFC 5000		
Posición	**Información**	**Descripción**
Bit 0	Estado del hardware	0: normal 1: fallo de hardware
Bit 1	Temperatura del cargador	0: normal 1: protección activada por temperatura

Bit 2	Tensión de entrada	0: normal 1: tensión de entrada fuera de límites
Bit 3	Estado de carga	0: todo bien, iniciar carga 1: tensión de batería mal, no cargar
Bit 4	Estado de comunicación	0: normal 1: time-out de comunicación

Tabla 17. Flags enviados por el cargador para comunicar posibles errores.

El software instalado en el BMS deberá entonces realizar dos funciones para gestionar la carga de las baterías: Deberá enviar una vez por segundo el mensaje al cargador con la información de carga y deberá interpretar la información recibida del cargador. Si hubiera alguna discrepancia entre lo que recibe del cargador y lo que está leyendo desde los módulos BMS, deberá parar la carga ya que podría significar que alguna celda está mal conectada o que ha habido una derivación.

6.1.3 COMUNICACIÓN CON CONTROLADOR SEVCON

El controlador utilizado para accionar el motor utiliza una comunicación CANopen aplicado sobre una capa física CAN 2.0B. CANopen, como se ha explicado en el capítulo 2, define el funcionamiento de sistemas empotrados en aplicaciones industriales. La especificación CANopen se divide en 3 partes: el protocolo CANopen para comunicaciones, el diccionario de objetos y la capa de aplicación. Para este trabajo es de especial interés conocer el diccionario de objetos del controlador SEVCON y cómo establecer una comunicación para leer ciertos parámetros.

El diccionario de objetos del controlador contiene 1 889 entradas. Muchas de estas entradas sólo se utilizarían en aplicaciones muy determinadas (por ejemplo, para coordinar el funcionamiento entre 2 controladores que trabajan en un mismo vehículo) o son parámetros internos y no son de interés. El protocolo de comunicaciones CANopen describe cómo leer cada una de estas entradas, pero su implementación en un microcontrolador como el Arduino

Mega Pro es demasiado tediosa y no aporta grandes ventajas al proyecto. La opción preferida en estas situaciones es configurar unos TPDOs (Transmission Processing Data Objects) que envíen de manera asíncrona los datos que sea necesario recopilar.

Para poder configurar el controlador se ha recurrido a la solución comercial recomendada: El módulo USB-to-CAN de IXXAT junto con el programa de configuración SEVCON-DVT para PC.

Figura 33. USB-to-CAN de IXXAT utilizado para configurar el controlador SEVCON. [23]

El programa de servicio SEVCON-DVT permite configurar infinidad de parámetros del controlador, desde las curvas de aceleración y par a los parámetros límites del sistema. Se utilizará este programa, junto con el conversor USB-CAN para ajustar el comportamiento de la moto en distintas situaciones: pruebas en el taller, pruebas en circuito y por último en la competición. También servirá para definir los TPDOs que utilizaremos para recopilar datos en tiempo real de la moto.

Los Transmission Processing Data Objects son un tipo de mensaje definido en la especificación CANopen que están pensados para poder enviar información de manera asíncrona a otros nodos del sistema como puede ser un display, o un sistema de adquisición de datos. Cada TPDO define un mensaje CAN transmitido en un intervalo fijo y que contiene como máximo 8 bytes. Los TPDOs envían una serie de campos del objeto de diccionarios que podemos definir a través de la herramienta SEVCON-DVT. [24]

Para realizar cualquier cambio de la moto es necesario entrar en modo pre-operacional. Una vez allí, se seleccionan los campos del diccionario de objetos que se quieran volcar a través de TPDOs, se guardan los cambios y se vuelve a modo operacional. Todo este proceso se lleva a cabo a través del protocolo CANopen pero gracias a la interfaz gráfica del programa SEVCON-DVT no es necesario programar todos estos cambios desde nuestro sistema de adquisición de datos. Una vez se ha cambiado la configuración del SEVCON, esta queda guardada en memoria no volátil por lo que no será necesario volver a realizar esta configuración a no ser que se quieran cambiar los parámetros que el controlador envía.

Los TPDOs definidos se han seleccionado tras consultar con el departamento de powertrain. Se ha intentado seleccionar aquellos datos que sean de mayor utilidad para poder analizar la eficiencia del sistema después de realizar pruebas en circuito y datos relevantes para mostrar en tiempo real en el display. En la tabla se muestran los datos seleccionados finalmente.

Elección de TPDOs				
	Información	Bits/bytes	Scaling	Unidades
TPDO 1 (ID:0x101)	Target Id	16/2	0.0625	A
	Id	16/2	0.0625	A
	Target Iq	16/2	0.0625	A
	Iq	16/2	0.0625	A
TPDO 2 (ID:0x102)	Battery voltage	16/2	0.0625	V
	Battery current	16/2	0.0625	A
	Line contactor	16/2	0.0625	V
	Capacitor voltage	16/2	0.0625	V
TPDO 3	Throttle Value	16/2	1/32 767	0-1

(ID:0x103)	Target torque	16/2	E-3	% of max
	Torque	16/2	E-3	% of max
TPDO 4 (ID:0x104)	Heatsink temp	8/1	1	ºC
TPDO 5 (ID:0x105)	Max. Motor speed	32/4	1	Rpm
	Motor velocity	32/4	1	Rpm

Tabla 18. Datos a recibir a través de TPDOs del controlador SEVCON.

6.1.4 LIBRERÍA CAN

El chip utilizado para establecer comunicación CAN desde el Arduino Mega Pro es el MCP2515. Este chip tiene una serie de registros donde se guarda la información a enviar por el bus CAN o la información recibida de este. Para leer o escribir en estos registros se utiliza el protocolo de comunicación SPI. Afortunadamente existe una librería de funciones pública que facilita esta comunicación con el chip MCP2515. [25] Gracias a esta librería, en vez de tener que estudiar las transacciones SPI que hay que realizar para modificar cada registro, se ejecutarán unas funciones de más alto nivel y la propia librería se encargará de modificar o leer los registros necesarios.

6.1.5 DATOS RECOGIDOS

Cada vez que se envía un mensaje por el bus CAN, este es detectado por el chip MCP2515. Si el mensaje cumple todos los requisitos de los filtros del chip, pasa a ser guardado en uno de los 2 buffers y el chip lanza la interrupción (0 lógico en el pin de interrupciones). Esta bajada en el pin de interrupciones es detectada por el Arduino, el cual rompe la ejecución del bucle main y ejecuta la función de lectura de datos. Una vez se haya leído el mensaje, el Arduino volverá al mismo punto del bucle main.

La gestión de interrupciones en la plataforma Arduino es muy sencilla, basta con ejecutar la siguiente línea al principio del programa y la función CAN0Interrupt () se llamará cada vez que el pin CAN0IntPin lea un 0 lógico.

```
attachInterrupt (digitalPinToInterrupt (CAN0IntPin), CAN0Interrupt, LOW);
```

Una vez leído el mensaje se ha optado por usar una cola circular para guardarlo hasta su procesamiento. La idea de usar esta cola es hacer que la interrupción sea lo más corta posible ya que los mensajes de los BMS y del SEVCON llegan muy rápidamente uno tras otro y pueden llegar a saturar los 2 buffers del chip MCP2515, especialmente en el bus CAN1 que es más rápido (500 kbps).

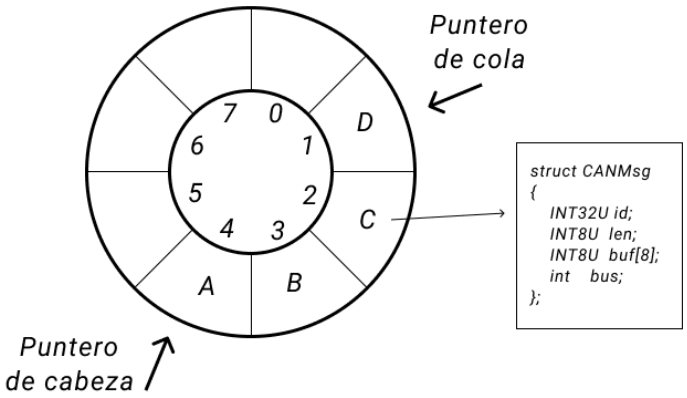


Figura 34. Esquema de una cola circular.

Cuando se usa una cola circular hay que tener en cuenta qué pasaría en el caso límite de que se llenara la cola y el puntero de cabeza alcanzara al de cola. Como esto en nuestra aplicación no va a ocurrir nunca ya que los mensajes van a llegar de manera muy seguida en grupos, pero con un gran periodo de tiempo entre grupos, se utilizará la implementación más sencilla en la que si el puntero de cabeza alcanza al de cola, simplemente se sigue sobrescribiendo la cola, aunque esto implicaría la pérdida de algunos paquetes.

La lectura de mensajes con la librería CAN utilizada se realiza a través de la función readMsgBuf y se guarda en la cola circular. A continuación se muestra la implementación en código de esta recogida de datos.

```
void CAN0Interrupt() // New msg on CAN0 -> read and into buffer
{
    if (!CAN0.readMsgBuf(&MSGBuffer[headCounter].id, &MSGBuffer[headCounter].len,
&MSGBuffer[headCounter].buf[0]))
    {
        MSGBuffer[headCounter].bus = 0;
        headCounter++;
        headCounter %= N;
    }
}
```

El procesamiento de los mensajes se hará teniendo en cuenta la dirección de la que viene cada mensaje (ID). Conociendo el ID se puede saber a qué módulo corresponde el mensaje y la información que contiene. Esta información se tiene que interpretar (teniendo en cuenta si los datos están en formato Little o big Endian) y se guardan en una estructura de datos. A continuación, se puede ver cómo está organizada esta estructura en formato JSON.

```
{
    "allOK": 0,
    "BMS": [{
        "cellVoltagemV": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        "temperatures": [0, 0]
    }, {
        "cellVoltagemV": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        "temperatures": [0, 0]
    }, {
        "cellVoltagemV": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        "temperatures": [0, 0]
    }],
    "SEVCON": {
        "TPDO1_1": 1

    },
    "CHARGER": {
        "Vtotal": 0,
        "Icharge": 0,
        "flags": [0, 0, 0, 0, 0]
    }
}
```

Para comprobar si existe algún problema con cualquiera de los módulos será necesario crear una función que reciba esta estructura y compruebe si todas las medidas están dentro de los valores de operación. Para ello es importante recordar que hay una serie de conexiones que no están conectadas a ninguna celda y que por lo tanto lo lógico es que midan una tensión nula. Esta función de comprobación se ejecutará de manera periódica.

Un último aspecto de la estructura de datos usada que hay que tener en cuenta es cómo hacer para detectar si alguno de los sistemas no está conectado. Para ello la solución propuesta ha sido reiniciar todos los valores a -1 después de ejecutar la función de comprobación. Si la siguiente vez que se ejecuta esta función, hay algún valor que sigue valiendo -1, significa que no ha llegado el mensaje del módulo correspondiente y que por lo tanto hay algún problema en el sistema. En este caso la moto deberá dejar de funcionar. Es importante entonces elegir un periodo de muestreo de los datos inferior al periodo de comprobación de estos.

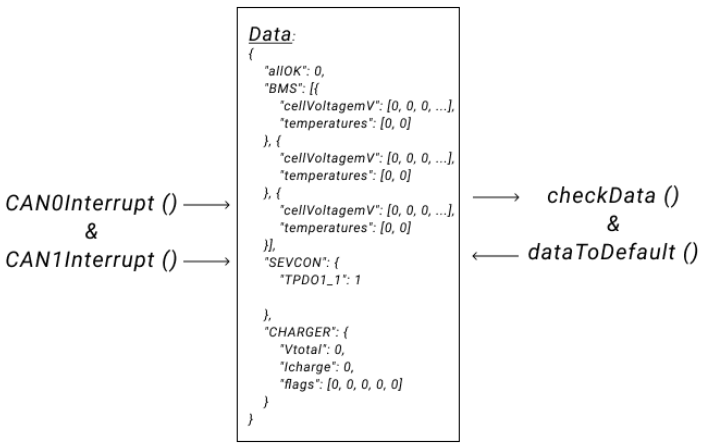


Figura 35. Flujo de datos dentro del Arduino Mega Pro.

6.1.6 TIMERS

El programa que se ejecuta en el Arduino Mega Pro necesita ejecutar varias funciones de manera periódica: Enviar "queries" (mensajes) a los módulos BMS y al cargador y

comprobar que todos los datos recibidos se encuentran dentro de los límites establecidos. Para realizar estas tareas se han valorado 3 opciones distintas:

1. Instalar un RTOS como *FreeRTOS*

Esta opción es la más profesional de las 3 ya que un Sistema Operativo en Tiempo Real (RTOS) está diseñado específicamente para tratar con este tipo de problemas de multitasking. No obstante, el Arduino Mega utilizado sólo contiene 8 KB de RAM y FreeRTOS consumiría gran parte de esta (7086 B). [26] Si se quisiera usar un RTOS la mejor opción sería cambiar a un microcontrolador con más RAM, pero se perdería la ventaja de poder utilizar la librería CAN específicamente desarrollada para Arduino.

2. Utilizar los Timers por hardware del Arduino Mega

La segunda opción valorada es utilizar los Timers por hardware del propio Arduino Mega para hacer saltar una interrupción. Las desventajas de esta opción son que se tienen únicamente 2 Timers libres (en principio se necesitarían 3). Además, usar interrupciones para las tareas de comprobación de los datos bloquearía las interrupciones que saltan al recibir datos por los bus CAN, lo que podría hacer que saturen los buffers de los chips MCP2515 y como consecuencia, perder paquetes CAN. Se verá que es más sencillo crear unos Timers usando la función millis () de Arduino, aunque no sea tan eficiente como las otras opciones.

3. Crear unos Timers propios usando la función millis ()

Gracias a la función millis () de Arduino se puede saber en todo momento cuánto tiempo ha pasado desde el inicio de la ejecución del programa. Esto permite crear, mediante el uso de variables de forma inteligente y unas cuantas comparaciones, unos Timers muy sencillos.

```
void loop ()
{
    unsigned long currentMillis = millis ();

    if ((unsigned long)(currentMillis - startMillis0) >= period0)
    {
        timer0();
```

```
        startMillis0 = currentMillis;
    }
}
```

Para poder usar esta opción sólo hay que asegurarse de que no hay ningún problema de "integer overflow". Este problema es común en sistemas empotrados que están funcionando mucho tiempo. Cuando un número entero va creciendo indefinidamente, llega un momento en el que es más grande que lo que puede almacenar la variable. Para la función millis(), la cual devuelve un número en formato unsigned long, esto ocurre después de unos 50 días (~2^32 milisegundos).

Una solución a este problema pasa por forzar que la comparación se haga en formato unsigned long. Cuando la función millis() vuelve a 0, la resta (currentMillis - startMillis0) da un resultado negativo. Al pasar este resultado de manera forzosa a unsigned long se obtiene un número muy grande (~2^32). Este número es claramente superior al periodo especificado y se ejecuta el timer. [27]

6.1.7 COMPROBACIÓN DE DATOS

La comprobación de los datos se hará a través de una función llamada periódicamente. Esta función marcará la variable allOK como 0 y desactivará el contactor si detecta cualquiera de los siguientes casos.

Situaciones que causarán la parada de la moto	**Cómo se detecta**
Desconexión de algún módulo BMS o del controlador SEVCON	Algún campo de datos valdrá -1
Tensiones extremas	Cualquier tensión por encima de 4.21 V o por debajo de 2.79 V
Temperaturas extremas	Cualquier temperatura por encima de 80 ºC o por debajo de 0 ºC

Tabla 19. Descripción de las situaciones que causarán una parada de la moto.

6.1.8 MÁQUINA DE ESTADOS

Para asegurarse de que el software del BMS es totalmente predecible se ha decidido diseñar una máquina de estados que no es más que un modelo que describe el comportamiento del sistema en función del estado en el que se encuentre y la transición entre estados.

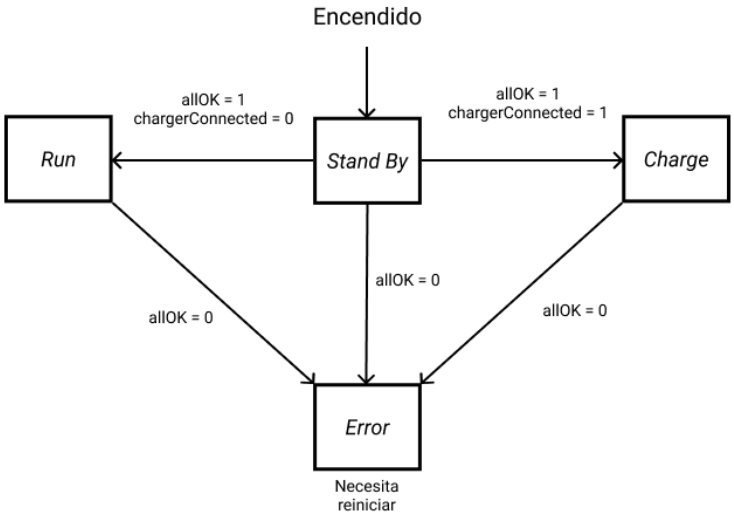


Figura 36. Máquina de estados del software del BMS.

La máquina de estados diseñada está pensada para priorizar la seguridad de todo el sistema. Una vez la moto detecta algún problema para a estado de error y requiere de un reinicio. De esta manera, si algún parámetro se sale de su rango de operación en cualquier momento, la moto parará y no podrá volver a arrancar hasta que el piloto haya apagado por completo y vuelto a encender. Lo mismo ocurre si la moto entra en modo de carga, hace falta un reinicio completo para que la moto pueda moverse. Así se evita que fallos humanos causen accidentes mayores. Además, el estado de la moto se mandará por bus CAN al sistema de adquisición de datos, donde se mostrará en el display para que tanto el piloto como los ingenieros puedan ver en qué estado se encuentra la moto.

De estos 4 estados posibles en los que se puede encontrar el sistema BMS, sólo en 2 de ellos se cierra el contactor que permite cargar o descargar la batería: Run y Charge.

6.2 SOFTWARE DEL SISTEMA DE ADQUISICIÓN DE DATOS

El sistema de adquisición de datos tendrá que realizar una labor muy similar al sistema BMS en cuanto al procesamiento de los mensajes recibidos por bus CAN. No obstante, su función es totalmente diferente ya que busca recopilar información y mostrar algunos datos en tiempo real en un display. A diferencia del BMS, no tendrá que determinar si existe algún parámetro fuera de los límites establecidos ni deberá actuar sobre el sistema. Por estos motivos, las elecciones de hardware y software para este sistema han sido también muy diferentes. Este sistema se ejecutará desde una Raspberry Pi en un entorno Linux (Raspberry Pi OS) ya que proporciona el mejor balance entre soporte online y facilidad de instalación.

6.2.1 ENTORNO LINUX

A diferencia del desarrollo en Arduino, donde únicamente hay que escribir un programa en C++ y subirlo al microcontrolador, el entorno Linux es mucho más amplio y permite muchas más opciones. En este entorno es posible escribir programas en distintos lenguajes de programación (para este proyecto se han considerado Python o C++), se pueden ejecutar varios procesos en paralelo (recoger datos a la vez que se actualiza el display) y, además cuenta con mucho soporte y documentación online.

Todos los sistemas operativos de Linux se basan en un Kernel (o núcleo) común que gestiona la interacción de las aplicaciones con la parte física del dispositivo. El desarrollo de este proyecto se limita a activar ciertas configuraciones del Kernel y a diseñar aplicaciones que interactúan con los dispositivos a través del Kernel.

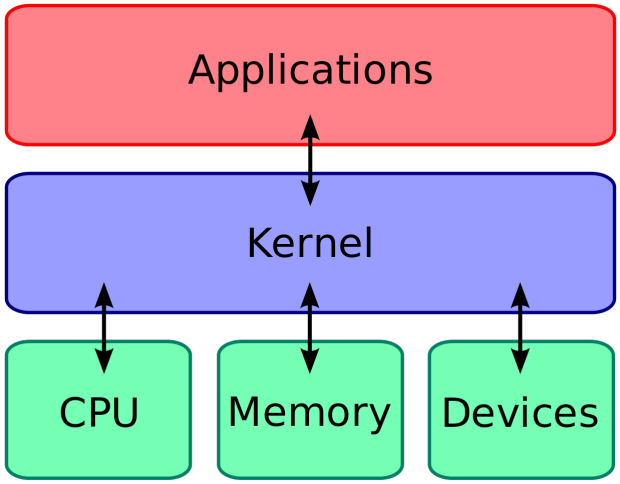


Figura 37. Esquema del funcionamiento de un sistema operativo basado en Linux.

Para establecer la comunicación CAN se han utilizado unos drivers que vienen integrados por defecto en el sistema operativo de la Raspberry Pi. Para que el kernel de Linux (la parte central de todo sistema operativo basado en Linux) reconozca que debe cargar estos drivers habrá que añadir unas líneas en el archivo /boot/config.txt. La configuración de estos drivers se trata con más detalle en el Anexo III de este trabajo.

Una vez cargados los drivers habrá que iniciar la comunicación CAN y crear un programa para procesar los mensajes recibidos. Se ha decidido crear un primer prototipo del sistema en el lenguaje Python pues es el más sencillo de usar y tiene librerías predefinidas para comunicación CAN, gestión de base de datos y comunicación por sockets.

6.2.2 BASE DE DATOS

Una de las funciones del sistema de adquisición de datos es guardar los parámetros del sistema para su posterior análisis. Para realizar esta función se han valorado 2 opciones distintas:

1. Guardar los datos en archivos de texto con formato JSON
2. Usar una base de datos relacional

Guardar los datos en un archivo JSON es una opción sencilla pero poco eficiente ya que en cada archivo hay partes que siempre van a ser iguales (que se podrían omitir) y se están usando caracteres para representar números (utilizando más memoria y CPU).

Utilizar una base de datos relacional es mucho más eficiente ya que no se repiten los encabezados y se optimiza el espacio mejor que con archivos de texto. El problema de una base de datos relacional es que no permitiría guardar las relaciones jerárquicas entre los sistemas. Para solventar este problema se han creado varias tablas dentro de la misma base de datos donde se representan cada uno de los sistemas. Una base de datos no relacional no es necesaria ni recomendada ya que cada entrada en la base de datos va a tener siempre la misma estructura.

El sistema de gestión de bases de datos utilizado será SQLite3 ya que cuenta con una librería para Raspberry Pi y está instalado por defecto en el sistema operativo. A continuación, se muestra cómo se han organizado las tablas y las entradas en la base de datos.

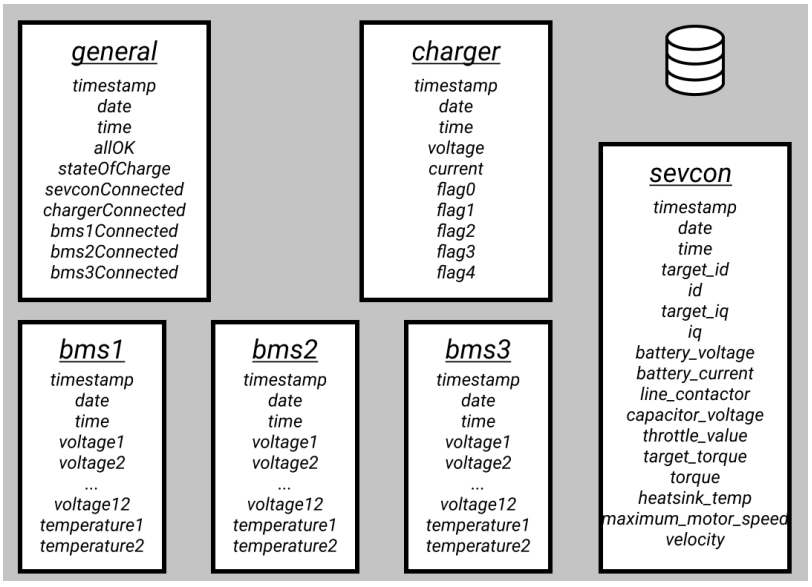


Figura 38. Tablas usadas en la base de datos sqlite.

Una vez se han diseñado las tablas de la base de datos y las entradas correspondientes se crea un proceso que los guarda de manera periódica. Para ello es necesario recurrir a la librería threading de Python.

6.2.3 PYTHON THREADS

Para gestionar varios procesos a la vez en Python lo más fácil es crear distintos "threads" o hilos de ejecución. Gracias a su capacidad multitarea, el kernel del sistema operativo determinará cómo y cuando se ejecuta cada uno de los threads. A través de la librería "threading" de Python se crearán 5 threads, cada uno con una función determinada.

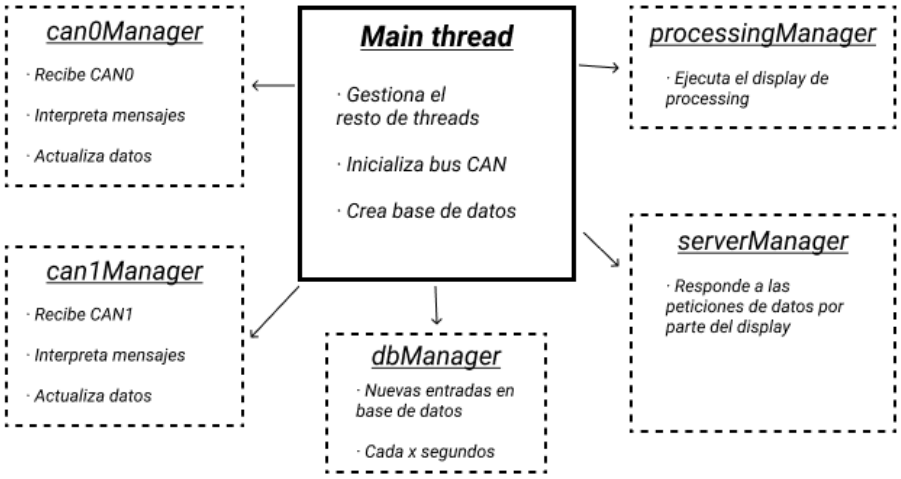


Figura 39. Esquema de los 5 threads creados en la Raspberry Pi.

Las funciones de los threads: *can0Manager* y *can1Manager* están bastante claras, se encargan de recibir los mensajes de cada bus CAN, interpretarlos y actualizar la estructura de datos del programa. El thread *dbManager* se ejecuta de manera periódica y se encarga de crear una nueva entrada en la base de datos. El thread *processingManager* se encarga de ejecutar el programa que gestionar el display y *serverManager,* de enviar datos actualizados al display. Estos dos últimos threads se analizan en más detalle en el próximo apartado.

6.2.4 PROCESSING & WEB SERVER

Para diseñar un display en Raspberry Pi existen multitud de opciones: implementarlo en HTML5 como página web; desarrollar una aplicación en Qt (C++); utilizando el módulo *guizero* de Python; o usando la plataforma de Processing (Java).

Para este proyecto se ha considerado que la mejor opción es a través de la plataforma de Processing ya que permite diseñar y probar el display en otros sistemas operativos (Windows o MacOS) y posteriormente exportar un sketch (aplicación) a la Raspberry. Además, es fácil de aprender a usar y permite explorar varias opciones rápidamente.

El problema de usar Processing es la integración con lo ya diseñado en Python. Para poder compartir datos en tiempo real entre estos 2 módulos, la opción recomendada es usar sockets internos. En Python se crea un thread que trabaja como servidor, esperando a recibir una petición de algún programa. Por otro lado, en Processing se crea un cliente que solicita los datos al servidor cada vez que lo necesite.

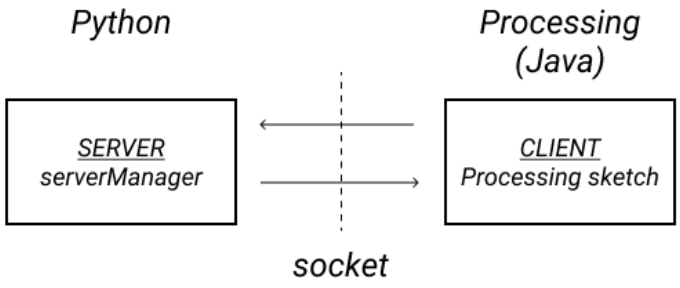


Figura 40. Relación Cliente-Servidor entre el thread de Python y el sketch de Processing.

Una vez el sketch de Processing tiene los datos actualizados del funcionamiento de la moto, se refresca el display mostrando los nuevos valores de interés. Debido al reglamento de MotoStudent algunos de estos campos son obligatorios. Por ejemplo, la tensión de las baterías junto con sus límites es un campo que se ha de mostrar obligatoriamente. Se ha tenido la precaución de cumplir con toda la normativa de esta edición, pero es importante tener en cuenta que la normativa puede cambiar para futuras ediciones.

Figura 41. Diseño del display.

El display muestra información acerca del estado actual de la moto como la velocidad, la tensión del pack de baterías, el modo de operación y una alerta si hubiera algo mal con el sistema. Además, muestra dos indicadores visuales: El de la izquierda muestra el estado de carga de la batería y el de la derecha muestra el par que está entregando el motor.

El display se ha diseñado de manera que se pueda ejecutar en dos modos distintos: Modo pruebas y modo carrera. En el modo pruebas se muestra información de especial interés para el equipo de ingeniería, como las tensiones máximas y mínimas de las celdas, la tensión del condensador de entrada y las revoluciones del motor. Para el modo carrera, esta información se considera innecesaria y por tanto no se muestra, simplificando el display y mostrando únicamente la información necesaria.

6.2.5 CALCULAR STATE OF CHARGE

Calcular el estado de carga de un sistema acumulador de carga se puede realizar a través de la medida de la tensión de la batería, contando la corriente que sale de la batería (Coulomb Counting) o con una combinación de ambos métodos con un filtro de Kalman o redes neuronales [28]. Para este trabajo se ha optado por primero hacer una estimación usando la tensión del pack de baterías y posteriormente una mejor estimación usando Coulomb Counting gracias a las medidas de intensidad que el controlador aporta.

Para la primera estimación del estado de carga se ha usado una estimación lineal entre la máxima tensión y la mínima tensión admitida. El método de cálculo se puede resumir en la siguiente ecuación:

$$SoC\ (\%) = \frac{V_{actual} - V_{min}}{V_{max} - V_{min}} \cdot 100$$

Ecuación 2. Cálculo del estado de carga mediante estimación lineal.

Este método proporciona una estimación del estado de carga de la batería razonable, pero tiene sus desventajas. En primer lugar, como la curva de descarga de una batería de ion de litio no es lineal, el estado de carga estará distorsionado especialmente alrededor del 40 %. En esta zona, parecerá que la batería no se está descargando, ya que el perfil de descarga es muy llano, mientras que por debajo del 15 % parecerá que la batería se está descargando de manera extremadamente rápida. En la siguiente figura se puede ver la diferencia entre el verdadero estado de carga (en azul) y la estimación realizada siguiendo este método (rojo).

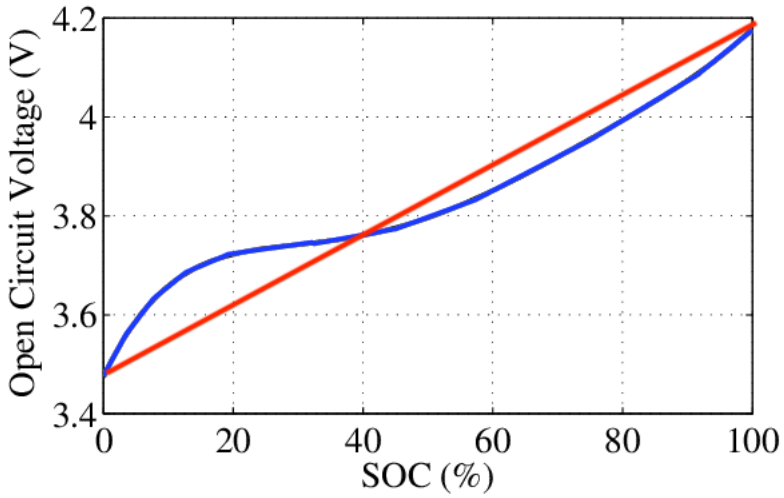


Figura 42. Cálculo del estado de carga midiendo la tensión en bornes.

Para solventar este problema se puede recurrir a la técnica de Coulomb Counting. Este método de estimación del estado de carga se basa en integrar la corriente de descarga para proporcionar una medida más fiable de la carga que ha salido de la batería y, por tanto,

cuanta carga queda. La ecuación que describe este método es como se muestra a continuación.

$$SoC\ (\%) = SoC_0(\%) - \frac{100}{C_{rate}} \int i(t) \cdot dt$$

Ecuación 3. Cálculo del estado de carga mediante Coulomb counting.

Implementar este algoritmo en un microcontrolador requiere de una pequeña modificación ya que se trabaja con lecturas discretas de intensidad y por lo tanto hay que estimar el valor de la integral.

$$SoC\ (\%) = SoC_0(\%) - \frac{100}{C_{rate}} \sum I_i \cdot \Delta t$$

Ecuación 4. Cálculo del estado de carga mediante Coulomb counting

Este segundo método de cálculo del estado de carga proporciona resultados mucho más precisos que la estimación lineal realizada al principio siempre y cuando el muestreo de la intensidad sea suficientemente preciso. El principal problema de este método es que puede acumular error a lo largo de varios ciclos de carga ya que existen corrientes parásitas dentro de las celda y por lo tanto es imposible medir con total precisión la corriente que sale de las mismas. Es por esto por lo que se han desarrollado métodos más complejos que combinan los dos explicados anteriormente en un filtro de Kalman o redes neuronales que se ajustan mejor a las curvas de descarga.

Para este proyecto, ya que la moto va a estar cargada al 100% cada vez antes de una carrera o pruebas en circuito, estos métodos no van a aportar precisión adicional. Por lo tanto, se utilizará el método de Coulomb counting inicializado cuando la carga de la batería está al máximo.

Capítulo 7. MONTAJE Y CONEXIONADO

Una parte de gran importancia en este proyecto es el montaje y conexionado de todos los elementos que conforman el sistema de gestión de baterías y el sistema de adquisición de datos. En este apartado de la memoria se muestran algunas imágenes del montaje de los sistemas con el objetivo de que sean de utilidad como referencia para futuras ediciones de la competición Motostudent.

En esta primera imagen se muestra el conexionado de los 3 módulos BMS junto con un par de PCBs. Estas placas se encargan de activar o desactivar el contactor, de detectar fallos en el aislamiento de la caja de baterías y el chasis (fuera del objetivo de este trabajo), y de permitir la comunicación CAN con el BMS central.

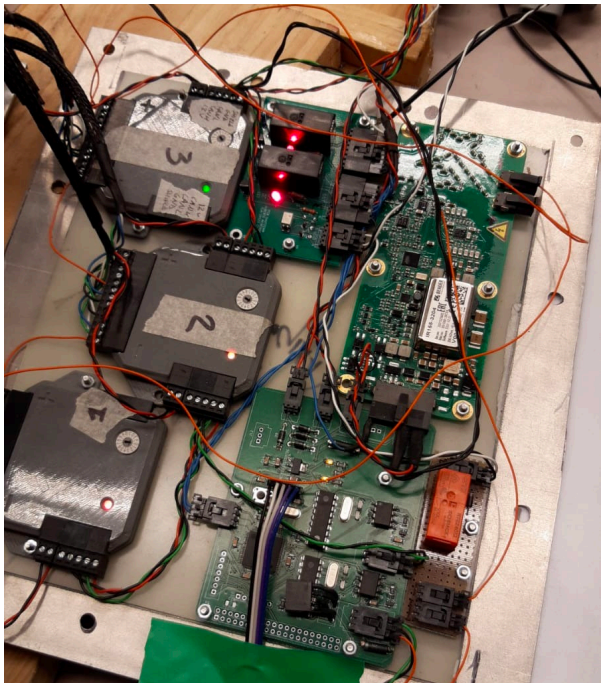


Figura 43. Conexionado de los 3 módulos BMS.

En la segunda imagen se muestra cómo se ha realizado el cableado de los módulos BMS a las celdas del pack de baterías. Alineado con los objetivos del proyecto, se han usado

conectores en todo este cableado para poder desmontar los sistemas y reemplazar algunos módulos de celdas si fuera necesario.

En esta segunda imagen se ve con más claridad el montaje de todo el sistema BMS, incluyendo la conexión de los módulos BMS a las celdas del pack de baterías.

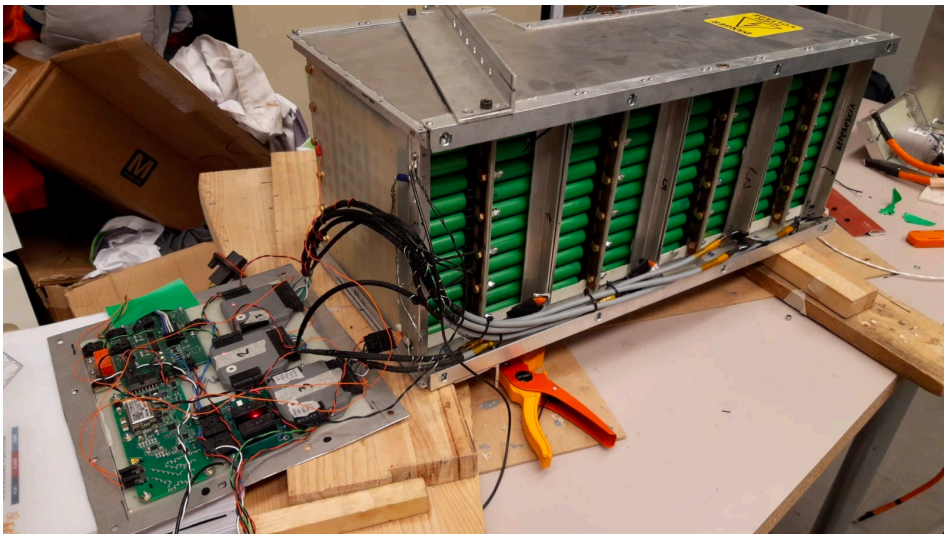


Figura 44. Conexión de los módulos BMS a las celdas.

Capítulo 8. ANÁLISIS DE RESULTADOS

En este trabajo fin de grado se ha desarrollado un sistema de gestión de baterías modular, se ha implantado en una moto eléctrica de competición y se ha probado su funcionamiento en un banco de ensayos. Al mismo tiempo, se ha diseñado un sistema de adquisición de datos y se ha instalado en paralelo con el sistema BMS. Su funcionamiento también se ha probado en banco de ensayos, pero no en circuito en el momento de redacción de esta memoria.

Ambos sistemas han demostrado ser fiables y robustos, sus conexiones están aisladas y se han probado frente a pequeños tirones. No obstante, deben ser probados en un circuito con la moto en movimiento para poder decir con certeza que aguantan todos los esfuerzos a los que está sometida una moto de competición.

Aunque los objetivos del proyecto se han cumplido, estos sistemas tienen algunas pequeñas limitaciones. El sistema de gestión de baterías, aunque utiliza conectores seguros, no es fácil de desmontar y manipular. Hace falta conocer el sistema a fondo para hacerlo con seguridad y para evitar dañar el cableado. El equipo que retome este proyecto en la siguiente edición tendrá que familiarizarse con estos sistemas si quieren realizar cualquier modificación. Por otro lado, una de las principales limitaciones del sistema de adquisición de datos es la tasa de refresco del display. El procesador de la Raspberry Pi no es suficientemente potente como para ejecutar todo el procesamiento de los mensajes CAN y actualizar el display con la fluidez que se querría. El resultado es que el display se refresca unas 18 veces por segundo, un ritmo algo inferior al que estamos acostumbrados a ver en la mayoría de los dispositivos electrónicos hoy en día.

Capítulo 9. CONCLUSIONES Y TRABAJOS FUTUROS

Este proyecto ha cubierto los objetivos fijados: Se ha diseñado e implantado un sistema de gestión de baterías y un sistema de adquisición de datos. Si bien estos sistemas vienen con algunas limitaciones, como se ha explicado en el apartado anterior, realizan las tareas que se les ha asignado muy satisfactoriamente.

De cara a ediciones futuras, este proyecto asienta las bases para que estos sistemas se optimicen, se actualicen, o se adapten a las necesidades del equipo. Para el sistema BMS el siguiente paso sea seguramente simplificar el cableado para hacer más fácil reemplazar módulos de baterías. Por otro lado, el sistema de adquisición de datos tiene mucho potencial por alcanzar. En primer lugar, se sugiere actualizar el procesador central (Raspberry Pi 3 B+) a el último modelo, logrando mejorar los cuadros por segundo a los que se refresca el display. En segundo lugar, se podría plantear portar la ejecución del display a una plataforma más optimizada que Processing, como podría ser Qt. Por último, el sistema de adquisición de datos puede incorporar nuevos sensores que ayuden a optimizar otros sistemas de la moto, como medidores de caudal de aire, galgas extensiométricas, acelerómetro, etc.

Capítulo 10. BIBLIOGRAFÍA

[1] x-engineer.org, «BMW's iPerformance plug-in hybrid electric vehicle (PHEV) powertrain architecture – x-engineer.org». https://x-engineer.org/automotive-engineering/vehicle/hybrid/bmw-iperformance-plug-in-hybrid-electric-vehicle-phev-powertrain-architecture/ (accedido jun. 14, 2021).

[2] E. JRW, *English: CAN bus Node*. 2014. Accedido: jun. 14, 2021. [En línea]. Disponible en: https://commons.wikimedia.org/wiki/File:CAN_Node.png

[3] «(8) Automotive CAN Bus System Explained | LinkedIn». https://www.linkedin.com/pulse/automotive-can-bus-system-explained-kiril-mucevski/ (accedido jun. 15, 2021).

[4] «File:CAN-bus-frame-with-stuff-bit-and-correct-CRC.png», *Wikipedia*. jun. 06, 2021. Accedido: jun. 14, 2021. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=File:CAN-bus-frame-with-stuff-bit-and-correct-CRC.png&oldid=1027154253

[5] «CAN in Automation (CiA): CANopen». https://www.can-cia.org/canopen/ (accedido jun. 15, 2021).

[6] «Tesla Model 3: Exclusive first look at Tesla's new battery pack architecture», *Electrek*, ago. 24, 2017. https://electrek.co/2017/08/24/tesla-model-3-exclusive-battery-pack-architecture/ (accedido jun. 15, 2021).

[7] «Tesla's new 2170 battery cell packs more power», *EVANNEX Aftermarket Tesla Accessories*. https://evannex.com/blogs/news/tesla-s-new-2170-cell-packs-more-power (accedido jun. 15, 2021).

[8] «Motostudent Electric», *Play and Drive*. https://www.playanddrive.com/prototypes/motostudent-electric/ (accedido jun. 15, 2021).

[9] P. Kiley, «Reverse Engineering the Tesla Battery Management System to Increase Power Available», p. 28.

[10] C. Bos, «Tesla's New HW3 Self-Driving Computer — It's A Beast (CleanTechnica Deep Dive)», *CleanTechnica*, jun. 15, 2019. https://cleantechnica.com/2019/06/15/teslas-new-hw3-self-driving-computer-its-a-beast-cleantechnica-deep-dive/ (accedido jun. 15, 2021).

[11] «s-BMS™ Battery Management System (BMS)», *LiTHIUM BALANCE*. https://lithiumbalance.com/products/s-bms/ (accedido jun. 15, 2021).

[12] «Zero Emission Vehicles Australia». https://www.zeva.com.au/index.php?product=133 (accedido jun. 15, 2021).

[13] «Race Technology Knowledge Base | DL1PRO-WP / TechSpecification». https://www.race-technology.com/wiki/index.php/DL1PRO-WP/TechSpecification (accedido jun. 16, 2021).

[14] «uCAN», *2D Debus & Diebold Meßsysteme GmbH*. http://2d-datarecording.com/en/produkte/hardware/logger/analog-logger/ucan-09/ (accedido jun. 16, 2021).

[15] «Zero Emission Vehicles Australia». https://www.zeva.com.au/Tech/Batteries/ (accedido jun. 16, 2021).

[16] «Test of Sony US18650VTC6 3000mAh (Green)». https://lygte-info.dk/review/batteries2012/Sony%20US18650VTC6%203000mAh%20(Green)%20UK.html (accedido jun. 16, 2021).

[17] «PFC 5000 Battery Charger», *elconchargers.com*. http://www.elconchargers.com/catalog/item/7344653/7638130.htm (accedido jun. 17, 2021).

[18] «UPC R 012A LS1 - Conector de Alta Potencia, UPC Series, Receptáculo, 1 kV, 250

A, Montaje en Panel, Soldable». https://es.farnell.com/amphenol-industrial/upc-r-012a-ls1/conec-alimentaci-n-hembra-250a/dp/2580198 (accedido jun. 17, 2021).

[19] «Gen4 AC». http://www.sevcon.com/products/low-voltage-controllers/gen4-ac/ (accedido jun. 16, 2021).

[20] «Arduino Mega 2560 Rev3 | Arduino Official Store». https://store.arduino.cc/usa/mega-2560-r3 (accedido jun. 16, 2021).

[21] «CAN Module MCP2515», *Makerfabs*. http://www.makerfabs.com/can-module-mcp2515.html (accedido jun. 16, 2021).

[22] «iso1050.pdf». Accedido: jun. 16, 2021. [En línea]. Disponible en: https://www.ti.com/lit/ds/symlink/iso1050.pdf?ts=1609842082084&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FISO1050

[23] «CAN and CAN FD interfaces for USB connection». https://www.ixxat.com/products/products-industrial/can-interfaces/usb-can-interfaces/usb-to-can-v2-professional?ordercode=1.01.0281.12001 (accedido jun. 16, 2021).

[24] «CAN in Automation (CiA): PDO protocol». https://www.can-cia.org/can-knowledge/canopen/pdo-protocol/ (accedido jun. 16, 2021).

[25] Cory, *coryjfowler/MCP_CAN_lib*. 2021. Accedido: jun. 16, 2021. [En línea]. Disponible en: https://github.com/coryjfowler/MCP_CAN_lib

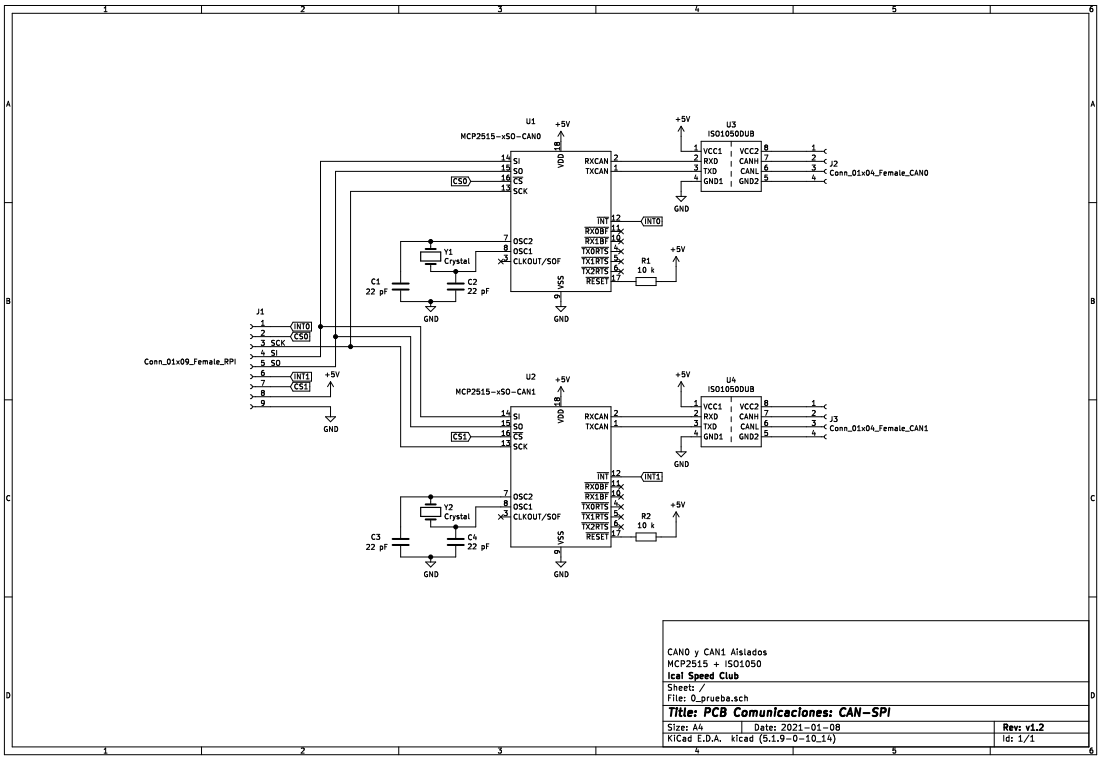
[26] «Using FreeRTOS multi-tasking in Arduino - Arduino Project Hub». https://create.arduino.cc/projecthub/feilipu/using-freertos-multi-tasking-in-arduino-ebc3cc (accedido jun. 16, 2021).

[27] «Arduino Tutorial: Avoiding the Overflow Issue When Using millis() and micros()». https://www.norwegiancreations.com/2018/10/arduino-tutorial-avoiding-the-overflow-issue-when-using-millis-and-micros/ (accedido jun. 16, 2021).

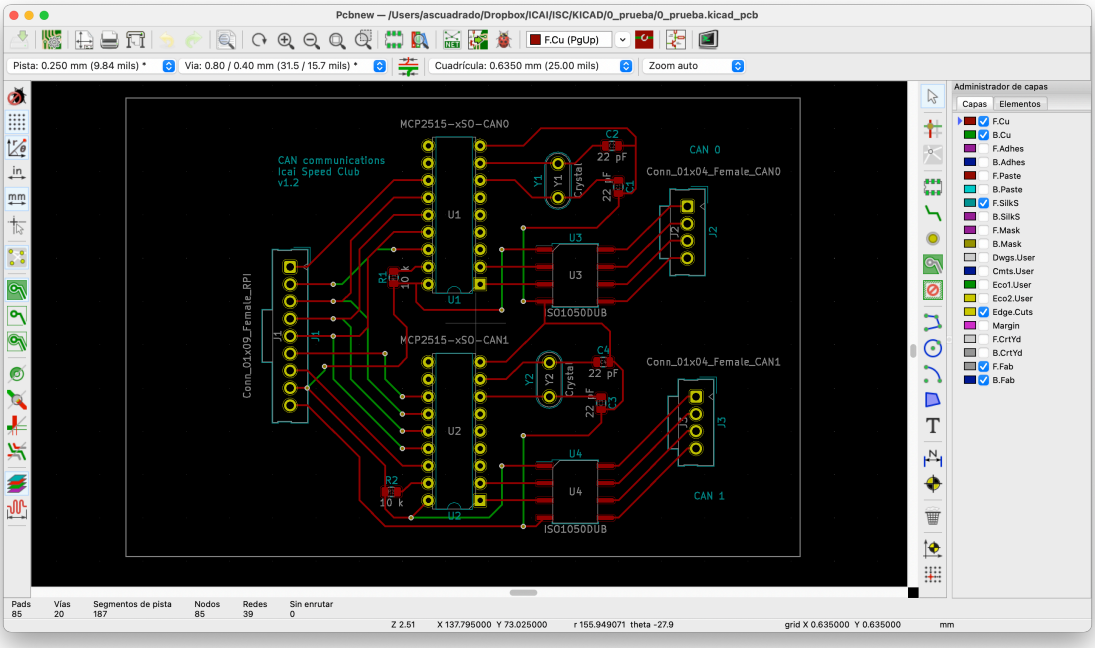
[28] W.-Y. Chang, «The State of Charge Estimating Methods for Battery: A Review», *ISRN Appl. Math.*, vol. 2013, pp. 1-7, jul. 2013, doi: 10.1155/2013/953792.

ANEXO I. DISEÑO PCB COMUNICACIONES

Primero diseñar diagrama electrónico:



Luego diseñar la PCB:


Pcbnew — /Users/ascuadrado/Dropbox/ICAI/ISC/KICAD/0_prueba/0_prueba.kicad_pcb
MCP2515-xSO-CAN0
CAN communications
Icai Speed Club
v1.2
CAN 0
Conn_01x04_Female_CAN0
Conn_01x09_Female_RPI
MCP2515-xSO-CAN1
Conn_01x04_Female_CAN1
CAN 1
ISO1050DUB
Administrador de capas
Capas
Elementos

ANEXO II. CÓDIGO DE BMS CENTRAL (ARDUINO)

Estructura de archivos para el código del BMS Central:

```
.
├── README.md
├── src
│   ├── main.cpp
│   ├── mcp_can.cpp
│   ├── mcp_can.h
│   ├── mcp_can_dfs.h
│   └── setup_config.h
```

main.cpp:

```
// Libraries
#include <Arduino.h> // For platformio development
#include "mcp_can.h" // mcp2515 library
#include <SPI.h>

#include "setup_config.h" // Contains all CAN configuration(pins, ids, etc)

// CAN instances
MCP_CAN CAN0(CAN0CS);
MCP_CAN CAN1(CAN1CS);

// Timer help
unsigned long startMillis0    = 0;
unsigned long startMillis1    = 0;
unsigned long startMillis2    = 0;
int           period0         = 1000;
int           period1         = 1000;
int           period2         = 3000;
int           BMSQueryCounter = 0;

// Functions
void dataToDefault();
void parseMessage(INT32U id, INT8U len, INT8U *buf, INT8U busN);
int checkData();
void CAN0Interrupt();
void CAN1Interrupt();
void timer0(); // Query BMS (one at a time)
void timer1(); // Query charger
void timer2(); // Check data and debug
void setup();
void loop();
```

```
// Data structures
struct Data data;

// CAN MSG Buffer (circular buffer -> if full, overwrites messages)
#define N    20
int           tailCounter = 0;
int           headCounter = 0;
struct CANMsg MSGBuffer[N];


/*
 * Main function
 */
int main()
{
    setup();
    while (true)
    {
        loop();
    }
}


/*
 * Setup function
 */
void setup()
{
    Serial.begin(115200);

    Serial.println("Arduino MEGA. 2 CAN buses, same SPI");

    attachInterrupt(digitalPinToInterrupt(CAN0IntPin), CAN0Interrupt, LOW);
    attachInterrupt(digitalPinToInterrupt(CAN1IntPin), CAN1Interrupt, LOW);

    while (CAN0.begin(MCP_ANY, CAN0Speed, MCP_8MHZ))
    {
        Serial.print("Failed starting CAN 0");
        delay(1000);
    }

    while (CAN1.begin(MCP_ANY, CAN1Speed, MCP_8MHZ))
    {
        Serial.print("Failed starting CAN 1");
        delay(1000);
    }

    CAN0.setMode(MCP_NORMAL);
    CAN1.setMode(MCP_NORMAL);

    Serial.print("CAN ready: CAN0 - ");
    Serial.print(CAN0Speed);
```

```
    Serial.print(" , CAN1 - ");
    Serial.print(CAN1Speed);
    Serial.println();
}


/*
 * Loop function (runs continuously)
 */
void loop()
{
    unsigned long currentMillis = millis();

    if ((unsigned long)(currentMillis - startMillis0) >= period0)    // Timer0
    {
        timer0();
        startMillis0 = currentMillis;
    }

    if (currentMillis - startMillis1 >= period1)    // Timer1
    {
        timer1();
        startMillis1 = currentMillis;
    }

    if (currentMillis - startMillis2 >= period2)    // Timer2
    {
        timer2();
        startMillis2 = currentMillis;
    }

    if (headCounter != tailCounter) // Process new msg in queue
    {
        parseMessage(MSGBuffer[tailCounter].id, MSGBuffer[tailCounter].len,
                     &MSGBuffer[tailCounter].buf[0], MSGBuffer[tailCounter].bus);
        tailCounter++;
        tailCounter %= N;
    }
}


void timer0() // BMS queries
{
    INT32U id       = BMS0ID;
    INT8U  ext      = 1;
    INT8U  len      = 2;
    INT8U  buf[len] = { (shuntVoltagemV >> 8) & 0xFF, shuntVoltagemV & 0xFF };

    // Query one BMS at a time
    if (BMSQueryCounter == 0)
    {
        CAN0.sendMsgBuf(id, ext, len, buf);
    }
```

```
    else if (BMSQueryCounter == 1)
    {
        CAN0.sendMsgBuf(id + 10, ext, len, buf);
    }
    else if (BMSQueryCounter == 2)
    {
        CAN0.sendMsgBuf(id + 20, ext, len, buf);
    }

    BMSQueryCounter++;
    BMSQueryCounter = BMSQueryCounter % 3;
}


void timer1() // Query charger (every 1s)
{
    long id     = chargerIDSend;
    int  v      = (maxChargeVoltage * 10);
    int  i      = (maxChargeCurrent * 10);
    int  charge = chargeIfPossible;

    uint8_t messageCharger[5] = { (v >> 8) & 0xFF,
                                  v & 0xFF,
                                  (i >> 8) & 0xFF,
                                  i & 0xFF,
                                  (1 - charge) };

    CAN0.sendMsgBuf(id, 1, 5, messageCharger);
    Serial.println("Query charger");
}


void timer2() // Check data and output to serial
{
    writeData(data);
    Serial.print("Check data: ");
    Serial.println(checkData());
}


void CAN0Interrupt() // New msg on CAN0 -> read and into buffer
{
    if (!CAN0.readMsgBuf(&MSGBuffer[headCounter].id, &MSGBuffer[headCounter].len,
&MSGBuffer[headCounter].buf[0]))
    {
        MSGBuffer[headCounter].bus = 0;
        headCounter++;
        headCounter %= N;
    }
}


void CAN1Interrupt()  // New msg on CAN0 -> read and into buffer
```

```
{
    //int a = micros(); // Count time of interrupt (for debugging only)
    if (!CAN1.readMsgBuf(&MSGBuffer[headCounter].id, &MSGBuffer[headCounter].len,
&MSGBuffer[headCounter].buf[0]))
    {
        MSGBuffer[headCounter].bus = 1;
        headCounter++;
        headCounter %= N;
    }
    //int b = micros();
    //Serial.print("t =  ");
    //Serial.println(b-a);
}


void parseMessage(INT32U id, INT8U len, INT8U *buf, INT8U busN)
{
    id = id & 0x1FFFFFFF;

    // PRINT Message for debugging
    char message[128];

    sprintf(message, "Message on Bus %d, id: 0x%lx, len: %d", busN, id, len);
    //Serial.println(message);
    for (int i = 0; i < len; i++)
    {
        sprintf(message, " 0x%.2X", buf[i]);
        //Serial.print(message);
    }
    //Serial.println();

    // ----

    // BUS 0: BMS and charger, 250kbps
    if (busN == 0)
    {
        // Charger
        if ((id == chargerIDRecv))
        {
            // Voltage measured by charger
            data.CHARGER.Vtotal = ((buf[0] << 8) + buf[1]);
            // Instant charging current
            data.CHARGER.Icharge = ((buf[2] << 8) + buf[3]);
            // Charger flags
            int flags = buf[4];
            data.CHARGER.flags[0] = (flags >> 0) & 0x1;     // Flag 0
            data.CHARGER.flags[1] = (flags >> 1) & 0x1;     // Flag 1
            data.CHARGER.flags[2] = (flags >> 2) & 0x1;     // Flag 2
            data.CHARGER.flags[3] = (flags >> 3) & 0x1;     // Flag 3
            data.CHARGER.flags[4] = (flags >> 4) & 0x1;     // Flag 4
        }

        // BMS Modules
```

```
        if ((id > 300) && (id < 300 + 3 * 10))
        {
            // BMS number (1-16)
            int n = (id - 300) / 10;
            // Message number (0-3)
            int m = (id - 300 - n * 10 - 1);

            // Voltage frame
            if (m < 3)
            {
                for (int i = 0; i < 4; i++)
                {
                    // i = number of cell within message
                    data.BMS[n].cellVoltagemV[m * 4 + i] = (buf[2 * i] << 8) +
buf[2 * i + 1];
                }
            }
            // Temperature frame
            else if (m == 3)
            {
                for (int i = 0; i < 2; i++)
                {
                    data.BMS[n].temperatures[i] = buf[i] - 40;
                }
            }
        }
    }

    // BUS 1: SEVCON controller
    if (busN == 1)
    {
        switch (id)
        {
        case 0x274:
            // TPDO1
            data.SEVCON.TPDO1_1 = (buf[0] << 8) + buf[1];
            data.SEVCON.TPDO1_2 = (buf[2] << 8) + buf[3];
            data.SEVCON.TPDO1_3 = ((int16_t)buf[4] << 8) + buf[5];
            data.SEVCON.TPDO1_4 = (buf[6] << 8) + buf[7];
            break;

        case 0x195:
            // TPDO2
            data.SEVCON.TPDO2_1 = (buf[0] << 8) + buf[1];
            data.SEVCON.TPDO2_2 = buf[2];
            data.SEVCON.TPDO2_3 = (buf[3] << 8) + buf[4];
            break;

        case 0x146:
            // TPDO3
            data.SEVCON.TPDO3_1 = ((int16_t)buf[0] << 8) + buf[1];
            data.SEVCON.TPDO3_2 = (buf[2] << 8) + buf[3];
            data.SEVCON.TPDO3_3 = (buf[4] << 8) + buf[5];
```

```
            data.SEVCON.TPDO3_4 = (buf[6] << 8) + buf[7];
            break;

        case 0x168:
            // TPDO4
            data.SEVCON.TPDO4_1 = (buf[0] << 8) + buf[1];
            data.SEVCON.TPDO4_2 = (buf[2] << 8) + buf[3];
            data.SEVCON.TPDO4_3 = (buf[4] << 8) + buf[5];
            data.SEVCON.TPDO4_4 = (buf[6] << 8) + buf[7];
            break;

        case 0x370:
            // TPDO5
            data.SEVCON.TPDO5_1 = (buf[0] << 24) + (buf[1] << 16) + (buf[2] << 8)
+ buf[3];
            data.SEVCON.TPDO5_2 = (buf[4] << 24) + (buf[5] << 16) + (buf[6] << 8)
+ buf[7];
            break;
        }
    }
}


int checkData()
{
    // allOK = 0 means something is wrong
    int allOK = 1;
    // intended to debug missing systems: {bms1,bms2,bms3,charger,sevcon}
    int sysFlags[5] = { 1, 1, 1, 1, 1 };

    int  minV = data.BMS[0].cellVoltagemV[0];
    int  maxV = minV;
    int  minT = data.BMS[0].temperatures[0];
    int  maxT = minT;
    long sumV = 0;

    // Check if are BMS connected & calculate maxV, minV, maxT, minT, sumV
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 12; j++)
        {
            int v = data.BMS[i].cellVoltagemV[j];
            if (v == -1)
            {
                allOK       = 0;
                sysFlags[i] = 0;
            }
            else if (((i == 1) && (j >= 8)) || ((i == 2) && (j >= 10)))
            {
                continue;
            }
            sumV += v;
            if (v < minV)
```

```
            {
                minV = v;
            }
            else if (v > maxV)
            {
                maxV = v;
            }
        }
        for (int j = 0; j < 2; j++)
        {
            int t = data.BMS[i].temperatures[j];
            if (t == -1)
            {
                allOK       = 0;
                sysFlags[i] = 0;
            }
            if (t < minT)
            {
                minT = t;
            }
            else if (t > maxT)
            {
                maxT = t;
            }
        }
    }

    if ((minV < 3200) || (maxV > 4000) || (maxT > 35))
    {
        allOK = 0;
    }

    // Check if charger connected
    if ((data.CHARGER.Vtotal == -1) || (data.CHARGER.Icharge == -1))
    {
        sysFlags[3] = 0;
    }
    for (int i = 0; i < 5; i++)
    {
        if (data.CHARGER.flags[i] == 0)
        {
            allOK = 0;
        }
    }

    // Check if sevcon connected
    if (data.SEVCON.TPDO1_1 == -1)
    {
        sysFlags[4] = 0;
        allOK       = 0;
    }

    // Print summary stats
```

```
    char buffer[64];

    sprintf(buffer, "MaxV: %d, MinV: %d, TotalV: %ld, maxT: %d \n", maxV, minV,
sumV, maxT);
    Serial.print(buffer);
    Serial.println("-----------------------------");

    if (!allOK)
    {
        Serial.println("-----------------------------");
        Serial.println("SOMETHING's WRONG!!!");
        Serial.println("-----------------------------");
    }
    dataToDefault();
    return allOK;
}


void dataToDefault() // Set all values to -1 to check if any system is
disconnected
{
    int d = -1;

    data.allOK = d;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 12; j++)
        {
            data.BMS[i].cellVoltagemV[j] = d;
        }
        data.BMS[i].temperatures[0] = d;
        data.BMS[i].temperatures[1] = d;
    }
    data.CHARGER.Vtotal  = d;
    data.CHARGER.Icharge = d;
    for (int i = 0; i < 5; i++)
    {
        data.CHARGER.flags[i] = d;
    }
    data.SEVCON.TPDO1_1 = d;
}
```

setup_config.h:

```
#include "mcp_can_dfs.h"

// CAN config
#define CAN0Speed           CAN_250KBPS
#define CAN0IntPin          2
#define CAN0CS              53
```

```
#define CAN1Speed           CAN_500KBPS
#define CAN1IntPin          3
#define CAN1CS              48

// Setup parameters
#define BMS0ID              0x12C
#define chargerIDSend       0x1806E7F4
#define chargerIDRecv       0x18FF50E7
#define shuntVoltagemV      0
#define maxChargeCurrent    10  // 1 amp
#define maxChargeVoltage    105 // 120 volts
#define chargeIfPossible    1   // 0 = don't charge

// CANMsg structure for buffer in main program
struct CANMsg
{
    INT32U id;
    INT8U  len;
    INT8U  buf[8];
    int    bus;
};

// Data structures (see datos.json to understand how they are organized)
struct BMSData
{
    int cellVoltagemV[12];
    int temperatures[2];
};

struct SEVCONData
{
    // TPDO 1
    int  TPDO1_1;
    int  TPDO1_2;
    int  TPDO1_3;
    int  TPDO1_4;

    // TPDO 2
    int  TPDO2_1;
    int  TPDO2_2;
    int  TPDO2_3;

    // TPDO 3
    int  TPDO3_1;
    int  TPDO3_2;
    int  TPDO3_3;
    int  TPDO3_4;

    // TPDO 4
    int  TPDO4_1;
    int  TPDO4_2;
    int  TPDO4_3;
```

```
    int  TPDO4_4;

    // TPDO 5
    long TPDO5_1;
    long TPDO5_2;
};

struct CHARGERData
{
    int Vtotal;
    int Icharge;
    int flags[5];
};

struct Data
{
    int                allOK;
    struct BMSData     BMS[3];
    struct SEVCONData  SEVCON;
    struct CHARGERData CHARGER;
};

/*
 * writeData: Output all data for debugging
 */
void writeData(struct Data data)
{
    Serial.println("Data:");
    char buffer[256];

    sprintf(buffer, "{\n");
    Serial.print(buffer);
    sprintf(buffer, "\"data\": {\n");
//    sprintf(buffer, "\"timeStamp\": %ld,\n", timestamp);
    sprintf(buffer, "\"allOK\": %d,\n", data.allOK);
    Serial.print(buffer);
    sprintf(buffer, "\"BMS\": [{\n");
    Serial.print(buffer);
    sprintf(buffer, "\"cellVoltagemV\":
[%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d],\n", data.BMS[0].cellVoltagemV[0],
data.BMS[0].cellVoltagemV[1], data.BMS[0].cellVoltagemV[2],
data.BMS[0].cellVoltagemV[3], data.BMS[0].cellVoltagemV[4],
data.BMS[0].cellVoltagemV[5], data.BMS[0].cellVoltagemV[6],
data.BMS[0].cellVoltagemV[7], data.BMS[0].cellVoltagemV[8],
data.BMS[0].cellVoltagemV[9], data.BMS[0].cellVoltagemV[10],
data.BMS[0].cellVoltagemV[11]);
    Serial.print(buffer);
    sprintf(buffer, "\"temperatures\": [%d,%d]\n", data.BMS[0].temperatures[0],
data.BMS[0].temperatures[1]);
    Serial.print(buffer);
    sprintf(buffer, "},{\n");
    Serial.print(buffer);
```

```
    sprintf(buffer, "\"cellVoltagemV\":
[%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d],\n", data.BMS[1].cellVoltagemV[0],
data.BMS[1].cellVoltagemV[1], data.BMS[1].cellVoltagemV[2],
data.BMS[1].cellVoltagemV[3], data.BMS[1].cellVoltagemV[4],
data.BMS[1].cellVoltagemV[5], data.BMS[1].cellVoltagemV[6],
data.BMS[1].cellVoltagemV[7], data.BMS[1].cellVoltagemV[8],
data.BMS[1].cellVoltagemV[9], data.BMS[1].cellVoltagemV[10],
data.BMS[1].cellVoltagemV[11]);
    Serial.print(buffer);
    sprintf(buffer, "\"temperatures\": [%d,%d]\n", data.BMS[1].temperatures[0],
data.BMS[1].temperatures[1]);
    Serial.print(buffer);
    sprintf(buffer, "},{\n");
    Serial.print(buffer);
    sprintf(buffer, "\"cellVoltagemV\":
[%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d],\n", data.BMS[2].cellVoltagemV[0],
data.BMS[2].cellVoltagemV[1], data.BMS[2].cellVoltagemV[2],
data.BMS[2].cellVoltagemV[3], data.BMS[2].cellVoltagemV[4],
data.BMS[2].cellVoltagemV[5], data.BMS[2].cellVoltagemV[6],
data.BMS[2].cellVoltagemV[7], data.BMS[2].cellVoltagemV[8],
data.BMS[2].cellVoltagemV[9], data.BMS[2].cellVoltagemV[10],
data.BMS[2].cellVoltagemV[11]);
    Serial.print(buffer);
    sprintf(buffer, "\"temperatures\": [%d,%d]\n", data.BMS[2].temperatures[0],
data.BMS[2].temperatures[1]);
    Serial.print(buffer);
    sprintf(buffer, "}],\n");
    Serial.print(buffer);
    sprintf(buffer, "\"SEVCON\": {\n");
    Serial.print(buffer);
    sprintf(buffer, "\"TPDO1_1\": %d\n", data.SEVCON.TPDO1_1);
    Serial.print(buffer);
    sprintf(buffer, "\n");
    Serial.print(buffer);
    sprintf(buffer, "},\n");
    Serial.print(buffer);
    sprintf(buffer, "\"CHARGER\": {\n");
    Serial.print(buffer);
    sprintf(buffer, "\"Vtotal\": %d,\n", data.CHARGER.Vtotal);
    Serial.print(buffer);
    sprintf(buffer, "\"Icharge\": %d,\n", data.CHARGER.Icharge);
    Serial.print(buffer);
    sprintf(buffer, "\"flags\": [%d, %d, %d, %d, %d]\n", data.CHARGER.flags[0],
data.CHARGER.flags[1], data.CHARGER.flags[2], data.CHARGER.flags[3],
data.CHARGER.flags[4]);
    Serial.print(buffer);
    sprintf(buffer, "}\n");
    Serial.print(buffer);
    sprintf(buffer, "}\n");
    Serial.print(buffer);
    sprintf(buffer, "}\n");
    Serial.print(buffer);
    sprintf(buffer, "\n");
```

```
    Serial.print(buffer);
}
```

ANEXO III. CONFIGURACIÓN DE RASPBERRY PI

setup keyboard:

```
echo XKBLAYOUT="es" | sudo tee -a /etc/default/keyboard
sudo reboot
```

Install can-utils

```
sudo apt-get install can-utils
```

Install python-can

```
pip3 install python-can
```

Install sqlite3

```
sudo apt-get install sqlite3
```

Install java (for processing)

```
sudo apt install openjdk-8-jdk
```

Avoid screen sleep

Add on: `/etc/xdg/lxsession/LXDE-pi/autostart`

```
@xset s off
@xset -dpms
```

Setup of HDMI Screen

Add to `/boot/config.txt:`

```
# HDMI Screen
hdmi_group=2
hdmi_mode=87
hdmi_cvt=800 480 60 6 0 0 0
hdmi_drive=1
max_usb_current=1
```

setup CAN:

Add to `/boot/config.txt:`

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=25
dtoverlay=mcp2515-can1,oscillator=8000000,interrupt=24
```

Start CAN

manual:

```
sudo ip link set can0 up type can bitrate 250000
sudo ip link set can1 up type can bitrate 500000
```

Send files over ssh

```
scp -r /Users/ascuadrado/Dropbox/ICAI/ISC/ISC/Processing/sketch_0_test
pi@betoberry.local:Desktop
scp -r /Users/ascuadrado/Dropbox/ICAI/ISC/ISC/Rasp-main
pi@betoberry.local:Desktop
```

Receive files over ssh

```
scp -r pi@betoberry.local:Desktop /Users/ascuadrado/Desktop
```

Remove rpi files

```
rm -r Desktop/sketch_0_test
```

Execute python main

```
python3 Desktop/Rasp-main/main.py
```

Execute processing

```
export DISPLAY=:0.0
Desktop/sketch_0_test/application.linux32/sketch_0_test
```

Automatic run on reboot

`crontab -e` -> Add to file:

```
@reboot python3 Desktop/Rasp-main/main.py
```

ANEXO IV. CÓDIGO PYTHON EN RASPBERRY PI

Estructura de archivos para Raspberry Pi

```
.
├── README.md
└── Rasp-main
    ├── CollectFromDB.py
    ├── Database.db
    ├── PruebasDB.py
    ├── configFile.py
    ├── createTables.sql
    ├── isc.txt
    └── main.py
```

Main.py:

```
# External Imports
import can
import threading
import socket
import os
import time

# Internal imports
import configFile

# Data
data = configFile.ISCData()

# Daemons
def can0Manager():
    print("Can0 manager online")
    bus0 = can.interface.Bus('can0', bustype='socketcan_native')
    while True:
        msg = bus0.recv()
        data.interpret_can_msg(0, msg.arbitration_id, msg.data)

def can1Manager():
    print("Can1 manager online")
    bus1 = can.interface.Bus('can1', bustype='socketcan_native')
    while True:
        msg = bus1.recv()
        data.interpret_can_msg(1, msg.arbitration_id, msg.data)

def dbManager():
```

```
    print("Database manager online")
    while True:
        time.sleep(configFile.DBDelay)
        data.save_to_db()
        data.connected_systems_to_default()

def serverManager():
    print("Server manager online")
    HOST = ''
    PORT = 50007
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((HOST, PORT))
    s.listen(1)
    while True:
        try:
            conn, addr = s.accept()
            print('Connected by', addr)
            while True:
                reply = data.fetch_processing_data(conn.recv(1024))
                conn.send(reply)
            conn.close()
        except:
            print("Connection closed")

def processingManager():
    print("Processing starting")
    os.system("export DISPLAY=:0.0;
/home/pi/Desktop/sketch_0_test/application.linux32/sketch_0_test")
    time.sleep(1)

# Main execution
if __name__ == "__main__":
    # Create database if it does not exist
    os.chdir('Desktop/Rasp-main')
    os.system('sudo ip link set can0 up type can bitrate 250000')
    os.system('sudo ip link set can1 up type can bitrate 500000')
    if not os.path.exists(configFile.dbFile):
        print("No database, creating one")
        os.system('sqlite3 "Database.db" < "createTables.sql"')
    else:
        pass
    # Create threads for CAN, Processing Server and DB management
    can0M = threading.Thread(target=can0Manager, daemon=True)
    can1M = threading.Thread(target=can1Manager, daemon=True)
    dbM = threading.Thread(target=dbManager, daemon=True)
    serverM = threading.Thread(target=serverManager, daemon=True)
    processingM = threading.Thread(target=processingManager, daemon=True)
    # And start those threads
    can0M.start()
    can1M.start()
    dbM.start()
    serverM.start()
    processingM.start()
```

```
    while True:
        # Just wait while daemons do everything
        time.sleep(1)
```

ConfigFile.py:

```
import sqlite3
import json
import time

dbFile = 'Database.db'
DBDelay = 2 # seconds
chargerID = 0x1806E7F4
rpmToSpeedMultiplier = 1
maxVoltage = 4.2*30
minVoltage = 2.8*30 #
soc_method = 0 # 0=battery voltage, 1=integrate current loss
battery_capacity = 70*1000 # Milliamps-hour

class ISCData:
    general = dict()
    charger = dict()
    sevcon = dict()
    bms1 = dict()
    bms2 = dict()
    bms3 = dict()
    bms = []

    dischargeIntegrator = []

    def __init__(self):
        self.bms.append(self.bms1)
        self.bms.append(self.bms2)
        self.bms.append(self.bms3)
        self.init_dict(0)

    def init_dict(self, d):
        '''
        Initializes internal dictionaries (general, sevcon, charger,  bms1, bms2,
bms3)

        Args:
            d: value to default all dictionaries

        '''
        self.bms1['voltages'] = [d,d,d,d,d,d,d,d,d,d,d,d]
        self.bms2['voltages'] = [d,d,d,d,d,d,d,d,d,d,d,d]
        self.bms3['voltages'] = [d,d,d,d,d,d,d,d,d,d,d,d]
```

```
        self.bms1['temperatures'] = [d,d]
        self.bms2['temperatures'] = [d,d]
        self.bms3['temperatures'] = [d,d]

        self.charger['voltage'] = d
        self.charger['current'] = d
        self.charger['flags'] = [d,d,d,d,d]

        self.sevcon['target_id'] = d
        self.sevcon['id'] = d
        self.sevcon['target_iq'] = d
        self.sevcon['iq'] = d

        self.sevcon['battery_voltage'] = d
        self.sevcon['battery_current'] = d
        self.sevcon['line_contactor'] = d
        self.sevcon['capacitor_voltage'] = d

        self.sevcon['throttle_value'] = d
        self.sevcon['target_torque'] = d
        self.sevcon['torque'] = d

        self.sevcon['heatsink_temp'] = d

        self.sevcon['maximum_motor_speed'] = d
        self.sevcon['velocity'] = d

        self.general['allOK'] = d
        self.general['stateOfCharge'] = 1
        self.general['opMode'] = "Testing"
        self.general['sevconConnected'] = 0
        self.general['chargerConnected'] = 0
        self.general['bms1Connected'] = 0
        self.general['bms2Connected'] = 0
        self.general['bms3Connected'] = 0

    def save_to_db(self, dataBaseName="Database.db"):
        '''
        Saves the data collected to database

        Args:
            dataBaseName: Name of the file to sqlite3 database
        '''
        conn = sqlite3.connect(dataBaseName)
        c = conn.cursor()

        self.general['timestamp'] = time.time()
        self.general['date'] = time.strftime("%Y-%m-%d")
        self.general['time'] = time.strftime("%H:%M:%S")
        self.calculate_new_soc(method=soc_method)

        general = self.general
        charger = self.charger
```

```
        sevcon = self.sevcon
        bms1 = self.bms1
        bms2 = self.bms2
        bms3 = self.bms3

        c.execute("INSERT INTO general (timestamp, date, time, allOK,
stateOfCharge,"
                  "sevconConnected, chargerConnected, bms1Connected,"
                  "bms2Connected, bms3Connected) VALUES (?,?,?,?,?,?,?,?,?,?)",
                  (general['timestamp'], general['date'], general['time'],
                   general['allOK'], general['stateOfCharge'],
general['sevconConnected'],
                   general['chargerConnected'], general['bms1Connected'],
                   general['bms2Connected'], general['bms3Connected']))

        c.execute("INSERT INTO charger (timestamp, date, time, voltage, current,"
                  " flag0, flag1, flag2, flag3, flag4) VALUES
(?,?,?,?,?,?,?,?,?,?)", (
                      general['timestamp'], general['date'], general['time'],
                      charger['voltage'], charger['current'],
                      charger['flags'][0], charger['flags'][1],
charger['flags'][2],
                      charger['flags'][3], charger['flags'][4]))

        c.execute("INSERT INTO sevcon (timestamp, date, time, target_id, id, "
                  "target_iq, iq, battery_voltage, battery_current,
line_contactor, "
                  "capacitor_voltage, throttle_value, target_torque, torque, "
                  "heatsink_temp, maximum_motor_speed, velocity "
                  ") VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)",
                  (general['timestamp'], general['date'], general['time'],
                  sevcon['target_id'], sevcon['id'], sevcon['target_iq'],
                  sevcon['iq'], sevcon['battery_voltage'],
sevcon['battery_current'],
                  sevcon['line_contactor'], sevcon['capacitor_voltage'],
                  sevcon['throttle_value'], sevcon['target_torque'],
                  sevcon['torque'], sevcon['heatsink_temp'],
                  sevcon['maximum_motor_speed'], sevcon['velocity']))

        c.execute("INSERT INTO bms1 (timestamp, date, time, voltage1, voltage2, "
                  "voltage3, voltage4, voltage5, voltage6, voltage7, voltage8, "
                  "voltage9, voltage10, voltage11, voltage12, temperature1, "
                  "temperature2) VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)", (
                      general['timestamp'], general['date'], general['time'],
                      bms1['voltages'][0], bms1['voltages'][1],
bms1['voltages'][2],
                      bms1['voltages'][3], bms1['voltages'][4],
bms1['voltages'][5],
                      bms1['voltages'][6], bms1['voltages'][7],
bms1['voltages'][8],
                      bms1['voltages'][9], bms1['voltages'][10],
bms1['voltages'][11],
                      bms1['temperatures'][0], bms1['temperatures'][1]))
```

```
        c.execute("INSERT INTO bms2 (timestamp, date, time, voltage1, voltage2, "
                  "voltage3, voltage4, voltage5, voltage6, voltage7, voltage8, "
                  "voltage9, voltage10, voltage11, voltage12, temperature1, "
                  "temperature2) VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)", (
                      general['timestamp'], general['date'], general['time'],
                      bms2['voltages'][0], bms2['voltages'][1],
bms2['voltages'][2],
                      bms2['voltages'][3], bms2['voltages'][4],
bms2['voltages'][5],
                      bms2['voltages'][6], bms2['voltages'][7],
bms2['voltages'][8],
                      bms2['voltages'][9], bms2['voltages'][10],
bms2['voltages'][11],
                      bms2['temperatures'][0], bms2['temperatures'][1]))

        c.execute("INSERT INTO bms3 (timestamp, date, time, voltage1, voltage2, "
                  "voltage3, voltage4, voltage5, voltage6, voltage7, voltage8, "
                  "voltage9, voltage10, voltage11, voltage12, temperature1, "
                  "temperature2) VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)", (
                      general['timestamp'], general['date'], general['time'],
                      bms3['voltages'][0], bms3['voltages'][1],
bms3['voltages'][2],
                      bms3['voltages'][3], bms3['voltages'][4],
bms3['voltages'][5],
                      bms3['voltages'][6], bms3['voltages'][7],
bms3['voltages'][8],
                      bms3['voltages'][9], bms3['voltages'][10],
bms3['voltages'][11],
                      bms3['temperatures'][0], bms3['temperatures'][1]))

        conn.commit()
        conn.close()

    def connected_systems_to_default(self):
        self.general['sevconConnected'] = 0
        self.general['chargerConnected'] = 0
        self.general['bms1Connected'] = 0
        self.general['bms2Connected'] = 0
        self.general['bms3Connected'] = 0

    def interpret_can_msg(self, bus, id, msg):
        if bus == 0: # CAN0 bus (BMS & Charger)
            if (id > 300) and (id < 400): # BMS
                n = int((id-300)/10) # BMS number (0-2) -> 3 bms modules
                m = (id-300-n*10-1) # Message number (0-3) -> 4 different
messages

                # Show that a certain bms is connected
                self.general['bms'+str(n)+'Connected'] = 1

                if m>=0 and m<3: # voltage frame
                    for i in range(4): # we read 4 cell voltages
```

```
                        self.bms[n]['voltages'][4*m+i] = ((msg[2*i]<<8) +
msg[2*i+1])*1.0/1000
                else: # temperature frame
                    for i in range(2): # we read 2 temperatures
                        self.bms[n]['temperatures'][i] = msg[i]-40

            elif id == chargerID: # CHARGER
                self.general['chargerConnected'] = 1

                self.charger['voltage'] = ((msg[0]<<8) + msg[1])/10
                self.charger['current'] = ((msg[2]<<8) + msg[3])/10
                self.charger['flags'][0] = msg[4]>>7 & 0x01
                self.charger['flags'][1] = msg[4]>>6 & 0x01
                self.charger['flags'][2] = msg[4]>>5 & 0x01
                self.charger['flags'][3] = msg[4]>>4 & 0x01
                self.charger['flags'][4] = msg[4]>>3 & 0x01

        elif bus == 1: # CAN1 bus (SEVCON)
            self.general['sevconConnected'] = 1
            if id == 0x101:
                self.sevcon['target_id'] = ((msg[1]<<8)+msg[0])*0.0625
                self.sevcon['id'] = ((msg[3]<<8)+msg[2])*0.0625
                self.sevcon['target_iq'] = ((msg[5]<<8)+msg[4])*0.0625
                self.sevcon['iq'] = ((msg[7]<<8)+msg[6])*0.0625
            elif id == 0x102:
                self.sevcon['battery_voltage'] = ((msg[1]<<8)+msg[0])*0.0625
                self.sevcon['battery_current'] = ((msg[3]<<8)+msg[2])*0.0625
                self.dischargeIntegrator.append(self.sevcon['battery_current'])
                self.sevcon['line_contactor'] = ((msg[5]<<8)+msg[4])*1.0
                self.sevcon['capacitor_voltage'] = ((msg[7]<<8)+msg[6])*0.0625
            elif id == 0x103:
                self.sevcon['throttle_value'] = ((msg[1]<<8)+msg[0])*1.0/32767
                self.sevcon['target_torque'] = ((msg[3]<<8)+msg[2])*0.1/100
                self.sevcon['torque'] = ((msg[5]<<8)+msg[4])*0.1/100
            elif id == 0x104:
                self.sevcon['heatsink_temp'] = msg[0]
            elif id == 0x105:
                self.sevcon['maximum_motor_speed'] =
(msg[3]<<24)+(msg[2]<<16)+(msg[1]<<8)+msg[0]
                self.sevcon['velocity']
=(msg[7]<<24)+(msg[6]<<16)+(msg[5]<<8)+msg[4]

    def fetch_processing_data(self, msg):
        replyDict = dict()

        # Calculate values
        v = []
        t = []
        for b in self.bms:
            v+=(b.get('voltages'))
            t+=(b.get('temperatures'))
        maxCellV = max(v)
        minCellV = min(v)
```

```
        maxTemp = max(t)

        # Save values
        replyDict['vTotal'] = sum(v)
        replyDict['speed'] = self.sevcon.get("velocity")*rpmToSpeedMultiplier
        replyDict['torque'] = self.sevcon.get("torque")
        replyDict['throttle'] = self.sevcon.get("throttle_value")
        replyDict['stateOfCharge'] = self.general.get("stateOfCharge")
        replyDict['opMode'] = self.general.get("opMode")

        replyDict['maxCellV'] = maxCellV
        replyDict['minCellV'] = minCellV
        replyDict['maxTemp'] = maxTemp
        replyDict['capacitorV'] = self.sevcon.get("capacitor_voltage")
        replyDict['velocity'] = self.sevcon.get("velocity")
        replyDict['current'] = self.sevcon.get("battery_current")

        reply = json.dumps(replyDict) + '\n'
        return reply.encode()

    def calculate_new_soc(self, method=0):
        if method == 0:
            v = []
            for b in self.bms:
                v+=(b.get('voltages'))
            print(sum(v))
            self.general['stateOfCharge'] = 1.0*(float(sum(v))-
minVoltage)/(maxVoltage-minVoltage)
        elif method == 1:
            new_discharge = sum(dischargeIntegrator)
            self.general['stateOfCharge'] -= new_discharge/battery_capacity
```

ANEXO V. CÓDIGO PROCESSING EN RASPBERRY PI

Main sketch:

```
import processing.net.*;

Client c;
JSONObject json;

PFont digital;
PFont roboto;

Data        data;
FullDisplay display;

void setup()
{

    try {
      digital = createFont("Digital.ttf", 150);
      roboto  = createFont("Roboto-Regular.ttf", 24);
      c = new Client(this, "127.0.0.1", 50007);  // Connect to server on port 80
    } catch (Exception e){
      println(e);
    }

    json = new JSONObject();
    data    = new Data();
    display = new FullDisplay();

    //size(800, 480);
    noCursor();
    fullScreen();

    // Output more information
    data.testing = false;

    println("Setup Done");
}


void draw()
{
    background(0);

    data.update();

    display.drawBattery(data.stateOfCharge);
```

```
    display.drawAlert(true);
    display.drawSpeed((int)data.speed);
    display.drawPower(data.torque, data.throttle);
    display.drawBatteryV(data.vTotal);
    display.drawOpMode(data.opMode);
    if(data.testing){
      display.drawTestingData();
    }

    //display.drawBattery(1-1.0*mouseX/700);
    println(frameRate);
}
```

Data:

```
class Data {
    boolean testing = false;

    // Important data
    float vTotal = 119;
    float speed        = 225;
    float torque = 0.5;
    float throttle     = 0.2;
    float stateOfCharge = 0.45;
    String opMode = "Testing";

    // Testing data
    float maxCellV = 3.251;
    float minCellV = 3.345;
    float maxTemp = 21;
    float capacitorV = 85;
    float velocity = 3150;
    float current = 0;

    void update()
    {
        c.write("request Data\n");

        while(c.available()==0){
          delay(1);
        }

        String data = c.readString();
        data = data.substring(0, data.indexOf("\n"));
        print(data);

        try{
          json = parseJSONObject(data);

          // Important data
```

```
            vTotal = json.getFloat("vTotal");
            speed = json.getFloat("speed");
            torque = json.getFloat("torque");
            throttle = json.getFloat("throttle");
            stateOfCharge = json.getFloat("stateOfCharge");
            opMode = json.getString("opMode");

            // Testing data
            maxCellV = json.getFloat("maxCellV");
            minCellV = json.getFloat("minCellV");
            maxTemp = json.getFloat("maxTemp");
            capacitorV = json.getFloat("capacitorV");
            velocity = json.getFloat("velocity");
            current = json.getFloat("current");

          } catch (Exception e){
            println(e);
          }
      }
}
```

Display:

```
class FullDisplay {
  int margen = 15;

  void drawBattery(float percentage){
    int batWidth = 50;

    // Green bar
    fill(0,255,71);
    noStroke();
    rect(margen, height-margen, batWidth, -(height-2*margen), batWidth/2);
    fill(0);
    rect(margen, margen, batWidth, (height-2*margen)*(1-percentage));
    // Edge of shape
    stroke(255);
    strokeWeight(4);
    noFill();
    rect(margen, height-margen, batWidth, -(height-2*margen), batWidth/2);

    // Percentage indicators
    strokeWeight(1);
    fill(255);
    textFont(roboto);
    textSize(24);
    textAlign(RIGHT);
    for (int i = 0; i <= 10; i++) {
      int y = margen+batWidth/2 + i*39;
      line(margen+7, y, margen+batWidth-7, y);
```

```
    text(String.format("%d",(10-i)*10)+" %", margen+batWidth+12+65, y+10);
  }
}


void drawAlert(boolean on){
  int x = margen+170;
  int y = margen+5;
  int w = 125;
  int h = 125;

  if(!on){
    fill(10);
    noStroke();
    rect(x,y,w,h);
  }else{
    fill(100);
    noStroke();
    //rect(x,y,w,h);
    PShape s;
    s = loadShape("Alert.svg");
    shape(s, x, y, w, h);
  }
}

void drawSpeed(int speed){
  textFont(digital);
  textAlign(RIGHT, BOTTOM);
  fill(255);
  text(String.valueOf(speed), 100, 191, 420, 250);
  textSize(50);
  textAlign(LEFT, BOTTOM);
  text("KM/H", 520, 333, 200, 100);
}

void drawPower(float torque, float throttle){
  int h = 250;
  int powerW = 50;

  // POWER
  textFont(digital);
  textSize(24);
  textAlign(CENTER, CENTER);
  text("POWER", 670, 130, 130, 60);

  // Torque meter
  fill(0,255,240);
  noStroke();
  rect(710, 190+h, powerW, -h, powerW/2);
  fill(0);
  rect(710, 190, powerW, h*(1-torque));
  stroke(255);
  strokeWeight(4);
```

```
    noFill();
    rect(710, 190, powerW, h, powerW/2);

    // Throttle indicator
    stroke(255);
    strokeWeight(2);
    line(710+powerW/2-40, 190+h*(1-throttle), 710+powerW/2+40, 190+h*(1-
throttle));
  }

  void drawBatteryV(float voltage){
    textFont(digital);
    fill(0,255,71);
    textAlign(LEFT, CENTER);
    textSize(50);
    text(String.format("%.1f",voltage)+"V", 320, 20, 250, 80);
    textSize(22);
    fill(255);
    text(String.format("MAX: %.0fV", maxVoltage), 590, 20, 200, 40);
    text(String.format("MIN: %.0fV", minVoltage), 590, 60, 200, 40);
  }


  void drawOpMode(String mode){
    textFont(digital);
    fill(255);
    textAlign(CENTER, CENTER);
    textSize(50);
    text(mode, 149, 141, 323, 96);
    textSize(40);
  }

  void drawTestingData(){
    textFont(roboto);
    fill(255);
    textAlign(LEFT, TOP);
    textSize(18);
    String fullText = "";
    fullText += "Testing data:\n\n";
    fullText += "maxCellV: " + String.format("%.3f", data.maxCellV)+" v\n";
    fullText += "minCellV: " + String.format("%.3f", data.minCellV)+" v\n";
    fullText += "maxTemp: " + String.format("%.3f", data.maxTemp)+" °C\n";
    fullText += "capacitorV: " + String.format("%.1f", data.capacitorV)+" v\n";
    fullText += "velocity: " + String.format("%.0f", data.velocity)+" rpm\n";
    text(fullText, 507, 124, 184, 224);
  }

}
```

Settings:

```
static float maxVoltage = 4.2*30;
static float minVoltage = 2.8*30;
static int margen = 15;
```