Diseño y programación del sistema embebido de un prototipo Hyperloop

Por:

Guzmán Martín García

Máster Universitario en Ingeniería Mecatrónica

Director:

José Luís Poza Luján

Codirector:

José Alberto Conejero Casares

Septiembre de 2018

A toda mi familia de Hyperloop UPV

A la UPV, por habernos permitido embarcarnos en esta aventura

A Borrell S.A., y a José Borrell, por toda la ayuda que nos ha prestado, tanto aquí como en California

Gracias.

Contenido

0. Introducción 5
1. Entorno de realización 6
1.1. El equipo 6
1.2. La competición 7
1.3. La competición previa 8
1.4. Sistemas similares 9
1.5. Software 9
1.5.1. Software embarcado 9
1.5.2. Software no embarcado 10
2. Especificación de requisitos 11
2.0. Sistema: 11
2.1. Placa maestra: 11
2.2. Placa de navegación: 12
2.3. Placa de motores: 12
2.4. Placa de frenos: 12
2.5. Placa de energía: 12
3. Diseño del sistema 14
3.0. Diseño general 14
3.1. El hardware 16
3.1.1. Fundamentos del sistema de hardware 16
3.1.2. Arquitectura de hardware 18
3.1.2.1. Sensorización 19
3.1.2.2. Controlador máster: 24
3.1.3.2. Controlador de motores: 25
3.1.3.4. Controlador de navegación: 27
3.1.3.5. Controlador de frenos: 27
3.1.3.6. Controlador de energía: 27
3.2. Software 27
3.2.1. El protocolo de comunicaciones 28
3.2.2. Comunicaciones con tierra 31
3.2.3. Software embebido. 33
3.2.3.1. Placa Máster 33
3.2.3.2. Placa de motores 42
4. Prueba del sistema 47
4.1. Pruebas de hardware 47

4.1.1. Pruebas comunes 47
4.1.2. Placa de navegación 47
4.1.3. Placa de energía 48
4.1.4. Placa de frenos 50
4.1.5. Placa de motores 51
4.1.6. Placa maestra 51
4.2. Pruebas de software 51
4.2.1. Pruebas comunes 51
4.2.2. Placa de navegación 52
4.2.3. Placa de energía 52
4.2.4. Placa de motores 52
4.2.5. Placa de frenos 53
4.2.6. Placa maestra 53
4.2.7 Pruebas de integración 54
5. Conclusión 57
6. Referencias 59
Anexo 1: Documento con la estructura de todas las tramas CAN empleadas 60
Anexo 2: Mensaje JSON 64

0. Introducción

Desde la generalización del avión como medio de transporte, la industria del transporte masivo de personas no ha tenido ninguna innovación importante. Si bien se han generalizado las líneas ferroviarias de alta velocidad y existen ciertas líneas de trenes *bala*, que permiten alcanzar velocidades del orden de magnitud de los aviones empleando levitación magnética, su uso no se ha generalizado debido al alto gasto energético que suponen, y a la costosa infraestructura que requieren, al emplear vías de superconductores que necesitan ser refrigerados a temperaturas cercanas al cero absoluto para funcionar. Desde hace 50 años, y con la fugaz excepción del Concorde, la velocidad a la que nos desplazamos en viajes largos ha permanecido prácticamente constante, en las cercanías de la velocidad del sonido.

Pensando exactamente en esto mismo, el emprendedor Elon Musk propuso, en 2012, el primer concepto del medio de transporte del futuro, el *Hyperloop*. El diseño consistía en un vehículo moviéndose en el interior de un tubo al que se le ha extraído la mayor parte del aire. El vehículo emplearía un compresor en la parte frontal para mover el poco aire restante en el tubo hacia la parte inferior del vehículo, creando un cojinete de aire sobre el que deslizarse sin apenas rozamiento. Al succionar parte del aire en la parte delantera, y asumiendo existe muy poco espacio entre el vehículo y el tubo, se crea una sobrepresión en la parte trasera del vehículo que lo propulsa hacia adelante.

Este diseño acaba de golpe con dos problemas de los trenes actuales: la resistencia a la rodadura y la resistencia del aire, permitiendo alcanzar velocidades del orden de la velocidad del sonido con facilidad, con la comodidad del tren convencional. Además, transfiere la tecnología de la infraestructura al vehículo, ya que la infraestructura es únicamente un tubo de acero sellado con bombas de vacío, mientras que es el vehículo el que tiene toda la tecnología necesaria para funcionar.

Dada la cantidad de proyectos en los que se encontraba trabajando Musk en el momento de la concepción de la idea (*Tesla Motors, SpaceX*), Musk decidió “externalizar” el desarrollo de esta idea. Para ello, convocó una competición para todas las universidades del mundo. La competición consistiría en diversas etapas, en las que se irían requiriendo diferentes características. La primera consistía en un diseño conceptual, la segunda (denominada H2 y realizada en agosto de 2017) una prueba de integración y levitación de un vehículo capaz de funcionar de forma autónoma dentro de un tubo de una milla de largo en Hawthorne, California. La tercera competición (denominada H3 y realizada en julio de 2018) requería alcanzar la máxima velocidad posible dentro del tubo, empleando los métodos propulsivos que los equipos considerasen adecuados.

Es en el contexto de esta competición en el que se desarrolla este trabajo de fin de Máster. El equipo *Hyperloop UPV*, tras ganar varios premios (mejor diseño en conjunto y mejor sistema propulsivo) en la H1 y quedar en el top 10 en la H2, se propuso continuar y mejorar la marca para el año 2018 con un *Pod* (como se denominan los vehículos de forma informal) capaz de alcanzar más de 400 kilómetros por hora en el tubo de vacío construido por SpaceX en sus instalaciones en Hawthorne, California.

1. Entorno de realización

El objetivo de este punto es describir el entorno bajo el cual se realiza el proyecto. Al ser el proyecto una pequeña porción de un proyecto aún mayor, las relaciones con el resto son de especial importancia, puesto que el sistema empotrado no tiene sentido por si mismo, sino junto al sistema que se debe controlar (la planta). Además, existen muchas partes diferentes que influyen en el diseño del sistema embebido. Por una parte, los requisitos de sensorización, control, etcétera, del resto del proyecto, y, por otra parte, los requisitos de la competición. Detallar el esquema organizativo del equipo y la base sobre la que parte (la competición del año pasado) simplifica la posterior explicación de los procedimientos y soluciones empleadas.

1.1. El equipo

Hyperloop UPV es un equipo de cerca de 30 personas, todas estudiantes de distintas ramas de la UPV, cuyo objetivo es competir en la *Hyperloop Pod Competition.* El esquema organizativo es un esquema típico en equipos de estudiantes, organizando el proyecto en distintos equipos, encargados a su vez de los distintos subsistemas que componen un proyecto de este estilo. Pertenecer a un equipo sin embargo no implica ser exclusivamente parte de ese equipo, ya que existe bastante heterogeneidad. Equipos con gran relación (estructuras y propulsión, o aviónica y energía), tienen miembros en común que hacen de puente.

Los distintos equipos son:

-**Propulsion**: equipo encargado de la motorización del vehículo, y de cualquier sistema que requiera el motor para funcionar.

-**Structures**: equipo encargado del sistema estructural, el chasis, el fuselaje y los distintos elementos que unen cada uno de los componentes que forma el vehículo.

-**Energy**: equipo encargado del sistema de alimentación, tanto de la parte de alto voltaje, destinada a alimentar el motor, como la parte de bajo voltaje que alimenta la electrónica y el resto de sistemas auxiliares.

-**Avionics**: equipo encargado del sistema electrónico, tanto hardware como software, de control del *Pod*, de su sensorización, y de la telemetría.

-**Pneumatics:** equipo encargado del sistema neumático de actuación de los frenos.

-**Dynamics**: equipo destinado a realizar simulaciones por software tanto de piezas estructurales, como de mecánica de fluidos, etcétera

-**Creative**: equipo destinado a dar visibilidad al trabajo y a relaciones con medios de comunicación.

-**Partners and Economy**: equipo destinado a relaciones con empresas e instituciones con el objetivo de financiar el proyecto.

-**Direction**: equipo directivo, se encarga de tomar las decisiones de alto nivel relacionadas con el equipo, su gestión, y la relación con las instituciones o empresas más importantes.

Durante el transcurso del año, todos los equipos se reúnen cada viernes con el objetivo de poner en común los avances realizados y problemas encontrados, para facilitar encontrar soluciones en común.

El equipo dentro del cual se desarrolla este proyecto es el de *Avionics.* Este equipo se divide a su vez en 2 subequipos íntimamente conectados pero formados por personas con diferentes funciones, *Hardware y Software.* El equipo de Hardware diseña y fabrica el sistema electrónico, y el equipo de Software realiza toda la programación. Obviamente, existe feedback constante entre ambos equipos, y si bien los miembros están definidos, la mayoría de los miembros del equipo de *Avionics* realizaron tareas tanto de hardware como de software. Aunque este trabajo se centra sobre todo en el apartado de Software, también se tendrá en cuenta el trabajo realizado por el equipo de Hardware

1.2. La competición

La *Hyperloop Pod Competition*, en su edición del año 2018, consta de 3 entregables diferentes que marcan hitos en el diseño de cada vehículo. Cada entregable sirve a su vez de filtro, eliminando *SpaceX* a los equipos que no cumplen con las expectativas de la competición. Los 3 entregables son los siguientes:

- -El *Preliminary Design Briefing:* Este entregable es una presentación de aproximadamente 20 diapositivas en las que el equipo realiza una declaración de intenciones. Se explica de forma general el diseño del prototipo, las características técnicas principales (potencia, estimación de peso, sistema de control, sistema energético) y las soluciones a los principales problemas (transmisión de potencia al raíl, seguridad, etcétera). No hace falta entrar en detalles técnicos con mucha profundidad, simplemente dar una imagen de lo que se pretende llevar a cabo. La entrega se realizó en noviembre de 2016
- -El *First Design Package:* Este entregable es el principal hito de la competición. En este documento técnico, se detallan con precisión todos los aspectos del vehículo: el sistema de propulsión, energía, estructuras, etcétera. Todo debe ir acompañado de sus simulaciones o justificaciones, ya sean realizadas por el equipo o por los diferentes proveedores. Es uno de los principales filtros de la competición, y es donde *SpaceX* elimina a la mayor cantidad de equipos. La entrega se realizó en mayo de 2018 e implica cerrar el diseño, puesto que una vez entregado y aceptado el FDP, pocas alteraciones sobre el diseño ahí descrito se permiten. Únicamente 22 universidades del mundo consiguieron que su FDP fuera aprobado por *SpaceX*.
- -El *Safety Package:* Este último documento se entrega una vez se ha asegurado la entrada del equipo en la competición, es decir, no sirve de filtro. En el documento, se explican todos los procedimientos que se deberán llevar a cabo en la competición para las diferentes tareas que se deberán ejecutar: desplazamiento del vehículo, montaje, desmontaje, precauciones a la hora de operar, etcétera. Se describen también todas las medidas de seguridad que se llevarán a cabo cuando se operen con sistemas peligrosos, como el sistema neumático o el sistema de alto voltaje. Una vez en la competición, los ingenieros de *SpaceX* comprueban y aprueban los contenidos del *Safety Package*, o dan su opinión y puntos que el equipo deberá mejorar si quiere avanzar en la competición. Una vez aprobado el *Safety* el equipo es libre de operar con su vehículo, siguiendo los procedimientos descritos.

Una vez en la competición, que se desarrolla en un párking en la fábrica de *SpaceX* en Hawthorne, California, el objetivo de la competición es someterse a distintas pruebas en las que los ingenieros de *SpaceX* comprueban la validez del diseño. Las pruebas son varias, y pueden ser desde inspecciones visuales o de los procedimientos hasta pruebas funcionales. Por ejemplo:

-Vacuum test: se introduce el prototipo en una cámara de vacío y se comprueba tanto la telemetría como el funcionamiento de los distintos sistemas

-Open run test: se coloca el prototipo en un tramo de vía de aproximadamente 100 metros y se simula un *run,* una carrera, estableciendo un límite de distancia y potencia de los motores. Este test sirve para comprobar el funcionamiento de la máquina de estados. También se prueba a desconectar distintas partes para comprobar el comportamiento del prototipo a distintos fallos.

Una vez un equipo ha pasado toda la batería de tests e inspecciones, recibe el visto bueno para entrar en el tubo de una milla de largo para probar sus sistemas por última vez al aire libre.

El último día de la competición, aquellos equipos que hayan demostrado su fiabilidad, entran en el tubo de metal al que se le hace el vacío, y compiten para conseguir la mayor velocidad.

1.3. La competición previa

En la competición H2 (2017), uno de los principales problemas a los que se tuvo que enfrentar el equipo fue el sistema de aviónica. Debido a varios problemas organizativos, el equipo de aviónica no tuvo el tiempo suficiente para programar, probar e integrar todos los distintos sistemas que conformaban el *Pod*, suponiendo un quebradero de cabeza enorme para el equipo. Con el objetivo de mejorar en este aspecto para la competición H3, se rediseñó el sistema completamente. Si bien se emplearán los mismos microprocesadores (aunque en mayor número) y sensores similares, dado que su comportamiento ya era conocido, la arquitectura y metodología de programación cambia radicalmente.

El sistema de la competición H2 estaba basado en 3 microprocesadores ARM M4+, en una placa comercial integrada llamada *Teensy 3.6.* Las tres placas estaban situadas en una misma placa madre, y comunicadas por CAN entre si mismas. Una de ellas se encargaba de la interfaz con los drivers de los motores, otra era la placa maestra que llevaba toda la máquina de estados y leía los distintos sensores, y otra se encargaba únicamente de la comunicación con la estación base.

Los problemas a los que se enfrentó el equipo de aviónica fueron variados:

-Poco tiempo disponible para el prototipado, ocasionando que fallos de diseño en las placas electrónicas tuvieran que ser arreglados sobre ellas mismas, en lugar de *Pod*er solicitar otras placas con el fallo arreglado. Este tipo de arreglos son muy frágiles y si bien sirven para comprobar el correcto funcionamiento, no deberían emplearse en una placa destinada a competir.

-Bloqueos de los microprocesadores bajo determinadas circunstancias aparentemente aleatorias.

-Problemas en la comunicación vía Ethernet. En concreto, se solapaban mensajes entre sí, haciendo imposible separarlos y diferenciarlos.

-Complejidad a la hora de programar el protocolo de comunicación. Al programarse al vuelo, no existía un registro con las diferentes tramas y su contenido, dificultando su implementación en código.

-Y, sobre todo, el debugging era complicado y requería mucho más tiempo del disponible.

Todos estos problemas se tuvieron en cuenta a la hora de diseñar el sistema para esta competición. Usando tanto los diagramas eléctricos como el software del año previo como base, y el consejo de los miembros del equipo del año pasado que participaron en el sistema, se intentó que, en el nuevo diseño, los problemas anteriormente mencionados o bien se eliminaran, o bien fueran más fáciles de arreglar. Durante el documento se irán detallando las diferentes soluciones técnicas alcanzadas a los distintos problemas.

1.4. Sistemas similares

Dado que el campo en el que se desarrolla este proyecto (el de las competiciones estudiantiles) es bastante limitado de por si, encontrar sistemas similares y que sean comparables es difícil. Si realizamos la comparación con, por ejemplo, modos de transporte comerciales, sale a la luz la enorme diferencia de recursos y tecnología existente entre la industria y un equipo de estudiantes.

Además, por lo especial de la competición, es complicado comparar con otros equipos pertenecientes a otras competiciones. *Fórmula Student*, por ejemplo, también muy prevalente en la UPV, tiene complejos sistemas electrónicos para el control y la telemetría del vehículo, pero salvo tareas sencillas de control, el vehículo no realiza nada de forma autónoma, y sólo responde a entradas del piloto.

No obstante, dada la gran cantidad de equipos que participan en la competición, es factible comparar los sistemas con los equipos de las diferentes universidades que se presentan a la competición. Una vez detallado el sistema, existe un pequeño resumen de los enfoques adoptados por los diferentes equipos.

1.5. Software

Todo el software empleado en esta competición es una evolución lógica de la competición pasada. El sistema empleado en el software no embarcado en la competición previa recibió elogios de *SpaceX* al ser único entre todos los equipos. El software embarcado sin embargo fue más problemático, motivo por el cual se ha empezado de cero a la hora de diseñar el sistema.

1.5.1. Software embarcado

La programación de los microcontroladores se realizará en C++ empleando varias librerías del proyecto *Arduino*. El IDE empleado será Visual Studio, junto con un plugin que permite programar el microcontrolador de forma sencilla, y hacer debugging empleando las interfaces existentes. Debido a la existencia de varios programas diferentes que deberán ser integrados en el proyecto final, se emplea el sistema de gestión de proyectos de programación *GitHub* para sincronizar los distintos códigos creados por los distintos programadores del equipo.

Los puntos clave del sistema son los siguientes:

- Tener un protocolo de comunicaciones que transmita la información necesaria en el menor tiempo posible desde cualquier microcontrolador esclavo al microcontrolador maestro.
- Tener un sistema de gestión de emergencias que, en caso de detectar una emergencia, minimice el tiempo existente entre la detección y la actuación de los sistemas mitigativos (en esencia, apagar los drivers para minimizar el riesgo de las baterías y activar los frenos en caso de emergencia durante la carrera).
- Tener una máquina de estados que se ejecute de forma sencilla y cumpla todos los requisitos de *SpaceX*.

- Transmitir los datos a la estación base con de forma rápida.

1.5.2. Software no embarcado

Además del software existente en los microcontroladores, existe software en las estaciones base que recibirán los datos del *Pod* para poder monitorizar su estado. Si bien este software no entra dentro del objetivo de este TFM, su uso es indispensable, por lo que se menciona en esta sección de forma resumida.

Para hacerlo lo más eficiente posible, el software no embarcado está dividido en capas. La primera de ellas es *Mule*, un software encargado de hacer de interfaz entre los protocolos empleados para la comunicación con el software no embarcado, y el software que correrá en los distintos ordenadores. La siguiente es *Apache Kafka*, un gestor de colas que facilita mucho el trabajo con fuentes de datos en tiempo real. En este caso, se encargará de repartir la información recibida por el *Pod* entre los distintos ordenadores que tendrán los jefes de cada subsistema. Por último, una GUI desarrollada por uno de los miembros mostrará la información recibida de forma eficaz, y permitirá realizar todas las acciones de control necesarias, entre ellas, una parada de emergencia en caso necesario. Hasta 5 personas pueden conectarse al *Pod* simultáneamente para dar órdenes, y muchas más para recibir los datos en tiempo real, gracias al sistema existente.

Imagen 1, Graphical User Interface del vehículo

2. Especificación de requisitos

Una vez detalladas las condiciones en las que se va a desarrollar el proyecto, el siguiente paso es enumerar los objetivos y requisitos que debe de tener el sistema final. En las tablas a continuación se nombran y asignan un código alfanumérico a los distintos requisitos que deberá tener tanto el sistema completo como los distintos subsistemas que lo componen. Los requisitos son una combinación tanto de hardware como de software, ya que ambos elementos están muy entrelazados.

2.0. Sistema:

Requisito	Descripción	Código
Ejecutar la misión	El sistema debe de ser capaz de ejecutar la misión para la que ha sido diseñado: iniciar los motores, ir a máxima potencia hasta alcanzar la máxima velocidad esperable y parar de forma segura.	SYS01
Comunicaciones	El sistema debe de comunicarse de forma bidireccional con la estación base para enviar datos de sensores y recibir órdenes, y enviar datos de telemetría al ordenador de SpaceX	COM01
Emergencias	El sistema debe de ser capaz de detectar de forma rápida emergencias, y en caso de producirse, actuar de la forma más rápida y eficaz posible para minimizar el riesgo para el equipo y las personas	EMER01

Tabla 1, requisitos del sistema

2.1. Placa maestra:

Requisito	Descripción	Código
Enviar datos de telemetría a la estación base	La placa deberá ser capaz de, bajo petición, enviar via ETHERNET un paquete de datos con la información de todos los sensores y variables que indican el estado de funcionamiento de los distintos programas	DATA01
Recuperar datos de todos los sensores requeridos de las placas esclavas	La placa deberá ser capaz de solicitar los datos via bus CAN de cada una de las variables de interés, ya sean sensorizadas o internas del programa y recibir las respuestas guardando la información en la memoria de ejecución para *Pod*er hacer un paquete de datos para el requisito DATA01	DATA02
Recibir información sobre el estado deseado de los motores y actuadores	La placa deberá ser capaz de recibir información via ETHERNET sobre el estado deseado por parte del equipo de los diferentes actuadores que existen en el sistema.	DATA03
Enviar el estado de los motores y actuadores a las placas esclavas	En relación con el requisito DATA03, La placa deberá ser capaz de una vez recibida la información del estado de los actuadores, redirigirla a la placa esclava correspondiente para que ejecute	DATA04
Recibir emergencias	En caso de que se reciba una emergencia, la placa debe valorar el tipo de emergencia lo antes posible y elegir una forma de actuar, y comunicárselo al resto de placas	EMER02
Ejecutar la misión	La placa maestra lleva un registro de todos los sensores y es la que determina cuándo deben de	SYS02

	arrancar los motores y cuándo deben de actuar los frenos.	

Tabla 2, requisitos de la placa maestra

2.2. Placa de navegación:

Requisito	Descripción	Código
Leer los datos de los lectores de cintas	La placa deberá leer la información de los lectores de cintas para poder contar el número de cintas por las que ha pasado el *Pod*, para poder determinar con relativa exactitud la posición de éste dentro del tubo.	NAV01
Leer los datos de las IMUs	La placa deberá recibir información de las 2 IMUs incorporadas, tanto de actitud como de aceleraciones	NAV02
Envío de información al máster	Una vez recogida la información, la placa debe bajo petición de la placa maestra, enviarle la información más actualizada	NAV03

Tabla 3, requisitos de la placa de navegación

2.3. Placa de motores:

Requisito	Descripción	Código
Comunicación con los drivers	La placa deberá ser la interfaz del sistema con los drivers. Deberá tener una lista de comandos para enviar órdenes y recibir información de los drivers de los motores	MOT01
Envío de información al máster	Una vez recogida la información, la placa debe bajo petición de la placa maestra, enviarle la información más actualizada	MOT02
Recepción de información	La placa debe recibir órdenes para reenviar a los drivers de los motores	MOT03
Comportamiento en caso de emergencia	Si la placa detecta que no hay comunicación con la placa maestra, deberá apagar los motores automáticamente asumiendo que ha ocurrido una emergencia.	EMER02

Tabla 4, requisitos de la placa de motores

2.4. Placa de frenos:

Requisito	Descripción	Código
Actuación de los frenos	La placa debe ser capaz de actuar los dos sistemas de frenos tras la recepción de una orden.	BRK01
Envío de información al máster	La placa debe bajo petición de la placa maestra, enviarle la información más actualizada de los sensores a bordo	BRK02
Recepción de información	La placa debe recibir órdenes de la maestra	BRK03
Comportamiento en caso de emergencia	Si la placa detecta que no hay comunicación con la placa maestra, deberá actuar los frenos asumiendo que ha ocurrido una emergencia.	EMER03

Tabla 5, requisitos de la placa de frenos

2.5. Placa de energía:

Requisito	Descripción	Código

Actuación de los contactores	La placa deberá ser capaz de, a petición de la placa maestra, activar los contactores que energizan a los drivers.	ENE01
Envío de información al máster	La placa debe bajo petición de la placa maestra, enviarle la información más actualizada del estado de las baterías.	ENE02
Recepción de información	La placa debe recibir órdenes de la maestra	ENE03
Comportamiento en caso de emergencia	Si la placa detecta que no hay comunicación con la placa maestra, deberá desactivar las baterías para desenergizar el sistema propulsivo.	EMER04

Tabla 6, requisitos de la placa de energía

3. Diseño del sistema

Este capítulo desarrolla el diseño final del sistema, tal y como compitió en agosto de 2018. Se empieza con el sistema de hardware, y a continuación se detalla la estructura de software y los distintos algoritmos que la componen.

3.0. Diseño general

El diseño general del sistema concierne a la visión general del sistema final. Este diseño es el producto de la combinación de todos los diferentes diseños de todos los equipos. Aunque en este capítulo se desarrolla un orden pseudo cronológico de los diferentes hitos en el diseño, prácticamente todo se centra en el diseño final:

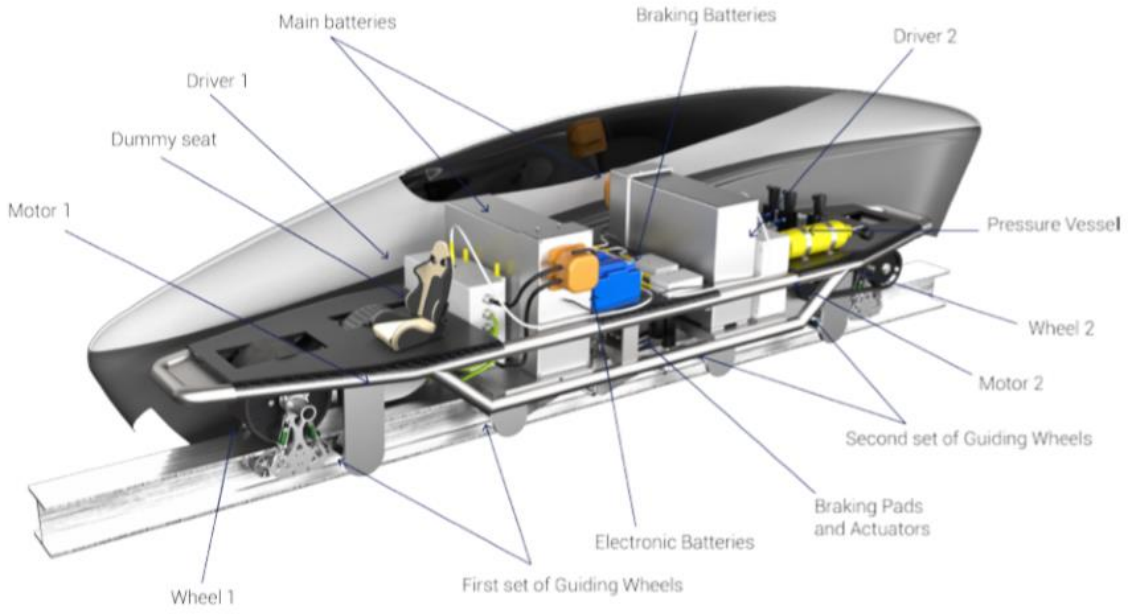

Imagen 2, modelo 3D del prototipo

Éste diseño se basa en una estructura de aluminio con planchas de fibra de carbono, dos ruedas motrices y varias ruedas de control de la estabilidad. Los motores elegidos son 2 EMRAX 228. Estos motores eléctricos de imanes permanentes y flujo axial se caracterizan por una elevada potencia (100kW de pico) en un factor de forma y peso muy reducidos.

Imagen 3, motor EMRAX 228

Para controlar los motores, se necesita un driver adecuado capaz de inyectar la señal SPWM en el motor para hacerlo girar. Por recomendación del fabricante, el driver elegido son dos unidades del modelo UNITEK BAMOCAR D3 como el mostrado en la imagen:

Imagen 4, driver BAMOCAR D3

Para frenar existen dos sistemas redundantes, eléctrico y neumático. El sistema eléctrico se basa en 4 actuadores lineales de 24V, y el sistema neumático de 4 cilindros actuados por 2 electroválvulas, una para los 2 cilindros superiores y otra para los 2 cilindros inferiores.

3.1. El hardware

Antes de poder crear un programa, necesitas saber qué tienes que programar exactamente. Por ello, los primeros meses de trabajo estuvieron orientados casi en exclusiva a definir el diseño de hardware, de arriba a abajo. Una vez el resto de equipos hubieran establecido sus requisitos de sensorización y control, ya se podría concretar más el diseño, hasta acabar con la fabricación y soldado a mano de las placas integradas (PCBs) que componen el hardware.

3.1.1. Fundamentos del sistema de hardware

El proceso de diseño del sistema comenzó con el inicio del equipo, en Septiembre de 2016. Los primeros meses fueron formalidades y decisiones sobre el comportamiento a alto nivel del sistema, ya que existía muy poca información sobre el resto del vehículo como para poder hacer diseños concretos. No obstante, en las primeras semanas, ya se tenían claras varias cosas:

-El sistema estaría sometido a una gran cantidad de EMIs (*Electromagnetic Interference)* debido a la naturaleza de los motores que se iban a emplear.

-La cantidad de datos respecto al año anterior aumentaría un orden de magnitud. La potencia de cálculo debería aumentar de forma acorde

-El sistema de gestión de las baterías requeriría un trabajo muy importante, dadas las características preliminares de las baterías (alto voltaje e intensidad)

A partir de ahí, se tomaron las primeras decisiones:

-El sistema sería modular, para poder reemplazar módulos en caso de problemas con facilidad, y para simplificar las labores de montaje y cableado.

-Se emplearían sobre todo buses digitales, estando los microcontroladores lo más cerca posible de los sensores, para evitar tener manojos de cables en la estructura, que complica el montaje y el mantenimiento. El bus empleado deberá ser diferencial, para minimizar la influencia de las interferencias electromagnéticas en las comunicaciones.

La primera decisión importante que tomar es el microcontrolador en el que se basaría el sistema. El microcontrolador determina el lenguaje de programación y las características más fundamentales del sistema, velocidad, buses de comunicación, etcétera. Dada su elevada penetración en el mercado de los sistemas embebidos, en prácticamente cualquier campo de la ingeniería, la arquitectura se basaría en un procesador ARM.

El microprocesador debería de ser adquirido en una placa con toda la electrónica de potencia y adaptadores para periféricos, entrada y salida, con el objetivo de simplificar el desarrollo de hardware.

Las características principales que se buscaban en el microprocesador eran:

-Compatibilidad con los sensores más habituales en la industria

-Facilidad de programación

-La mayor cantidad posible de interfaces de comunicación, para poder elegir la más adecuada

-Existencia de un ADC, del mayor rendimiento posible

Tras una primera revisión del estado del arte de microprocesadores dedicados a proyectos de relativamente baja complejidad existían 3 alternativas distintas para las placas que contendrían el microprocesador:

-STMicroelectronics STM32F4

-Arduino Due

-Teensy 3.6

Los tres sistemas tenían características muy similares en cuanto a velocidad de reloj del sistema (siendo el *Arduino Due* el más lento) y resto de requisitos: la diferencia era obviable en la mayoría de los casos. Si bien la placa basada en el STM32F4 era la que tenía la mayor cantidad de puertos de entrada/salida/lectura analógica, lo hacía a costa de bastante más tamaño que la compacta Teensy 3.6. Aunque tan pronto en el diseño el tamaño no era una limitación, era de esperar que en el futuro pudiera serlo.

Se acabó decidiendo por la Teensy 3.6 por un motivo bastante básico: era el sistema empleado el año pasado, y ninguna de las alternativas encontradas ofrecían nada que inclinara la balanza.

La Teensy 3.6 ofrecía todos los puntos básicos junto con el añadido de que ya se conocía el comportamiento, y en un tamaño muy reducido (apenas 6x0.8cm). La programación se podía realizar empleando el conjunto de librerías del proyecto *Arduino*. Esto libera de mucho trabajo y potenciales problemas al equipo de software, ya que facilita mucho la interacción con los periféricos del microprocesador (puertos GPIO, buses de comunicación, ADCs…), y gracias a la enorme comunidad de usuarios del proyecto, existen librerías que facilitan la interacción con infinidad de sensores: están los protocolos ya implementados en librerías testeadas por la comunidad. Prácticamente todas las librerías están publicadas como código abierto, lo que permite usarlas para cualquier proyecto sin problemas de licencias, y además, permite modificar el código a voluntad en caso de alguna necesidad específica.

Una vez elegido el microprocesador y teniendo en cuenta los buses de comunicación del que dispone, el siguiente paso era elegir el bus de comunicación. El requisito de ser diferencial reducía bastante el ámbito: buses como I2C, RS232, no son diferenciales y por ende serían vulnerables al ruido electromagnético, por lo que quedarían automáticamente descartados. Los candidatos principales eran RS422, RS485 y CAN.

Aunque RS422 y RS485 permitían velocidades un orden de magnitud superiores a CAN (10Mbits/s contra 1Mbit/s), tenían una desventaja importante: Mientras que el bus CAN ya tiene un protocolo de comunicaciones que se asegura que no existan colisiones entre mensajes al priorizarlos, y establece cómo es la trama, RS422 y RS485 son simplemente estándares de comunicaciones. Elegir CAN permitía ahorrarse el trabajo de tener que crear de cero un protocolo de comunicaciones y todo el trabajo que ello conlleva: elegir el tamaño de cada trama, la estructura, el protocolo para evitar colisiones, etcétera. Por simplicidad y por el hecho de que es un bus muy extendido en la industria (que permitía, por ejemplo, comunicarnos con sensores industriales con mayor facilidad) y con la previsión de que la capacidad del bus sería suficiente (1Mbit/s), la elección del bus estaba hecha, siendo CAN el elegido.

Si bien la Teensy 3.6 es compatible con el protocolo CAN, no es compatible a nivel eléctrico, ya que el bus CAN es un bus diferencial que funciona a 5 V, y la Teensy funciona a 3.3V. Por ello, es necesario el uso de un transceptor que adapte los niveles (y, además, sea capaz de crear la señal diferencial, que la electrónica incorporada en la Teensy 3.6 es incapaz). La elección del chip

transceptor se limita a básicamente comprobar en comunidades de internet el funcionamiento de varios, para asegurar la compatibilidad y evitar la aparición de problemas. El elegido en este caso es el SN65HVD230 de Texas Instruments.

El siguiente punto por determinar era la cantidad de microprocesadores distribuidos y la arquitectura y protocolo a seguir en las comunicaciones. Tras varios avances en el diseño por parte de los diferentes subequipos, quedó claro que habría cuatro diferentes grupos de sensores y actuadores.

-Motores
-Sistema energético
-Navegación
-Frenos

Por lo tanto, era lógico dedicar un microcontrolador con un programa específico para cada uno de los grupos de sensores. Además de esos cuatro microcontroladores, habría un microcontrolador maestro encargado de gestionar las comunicaciones con la estación base y el sistema en global: es decir, sería el que daría órdenes al resto de microcontroladores, ya sea para actuar los frenos, para encender los motores, etcétera, y además recibiría toda la información pertinente de su parte para poder actuar en consecuencia, en caso de emergencia.

Para las comunicaciones y debido a la arquitectura del sistema (con cuatro microprocesadores esclavos y un maestro) se optó por una arquitectura de red maestro-esclavo, en la que el microprocesador maestro se encargaría de solicitar información o ordenar acciones a los microprocesadores esclavos, no existiendo ninguna comunicación entre ellos: todo pasaría por el maestro.

3.1.2. Arquitectura de hardware

Una vez establecidos los principios básicos sobre los cuales fundamentar el diseño, el hardware empleado por el sistema de aviónica quedó definido en el *Final Design Package*.

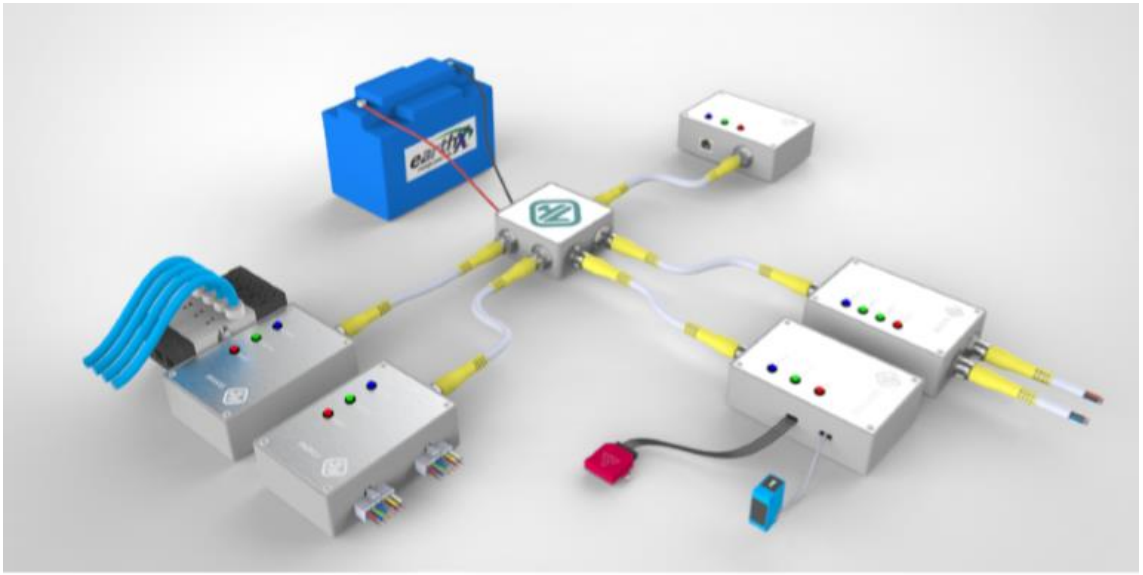

Imagen 5, modelado 3D del sistema de aviónica

El sistema descrito consiste en los 5 microcontroladores *Teensy 3* mencionados previamente basados en un ARM m4 a 180MHz. Los motivos principales por los que se emplea este microcontrolador son los siguientes:

-Compatibilidad con hardware y software diseñados para Arduino
-Posibilidad de programación con el IDE Visual Studio
-2 Puertos CAN
-Precio
-Enorme cantidad de puertos de entrada y salida tanto analógicos (24) como digitales (57) e interfaces de comunicación (I2C, SPI, Serie).
-2 ADC de 16bits, de los cuales 13 bits son usables por los requisitos de precisión.

Todos los sistemas estarán conectados entre si vía bus CAN, y el controlador Máster estará conectado con la estación base vía Ethernet. La alimentación de los distintos módulos se realiza también mediante el bus CAN.

3.1.2.1. Sensorización

Uno de los apartados fundamentales del sistema electrónico son los sensores usados, que determinan tanto el rango de lectura como el método para leer la información disponible. La distribución de sensores a lo largo del sistema es la siguiente:

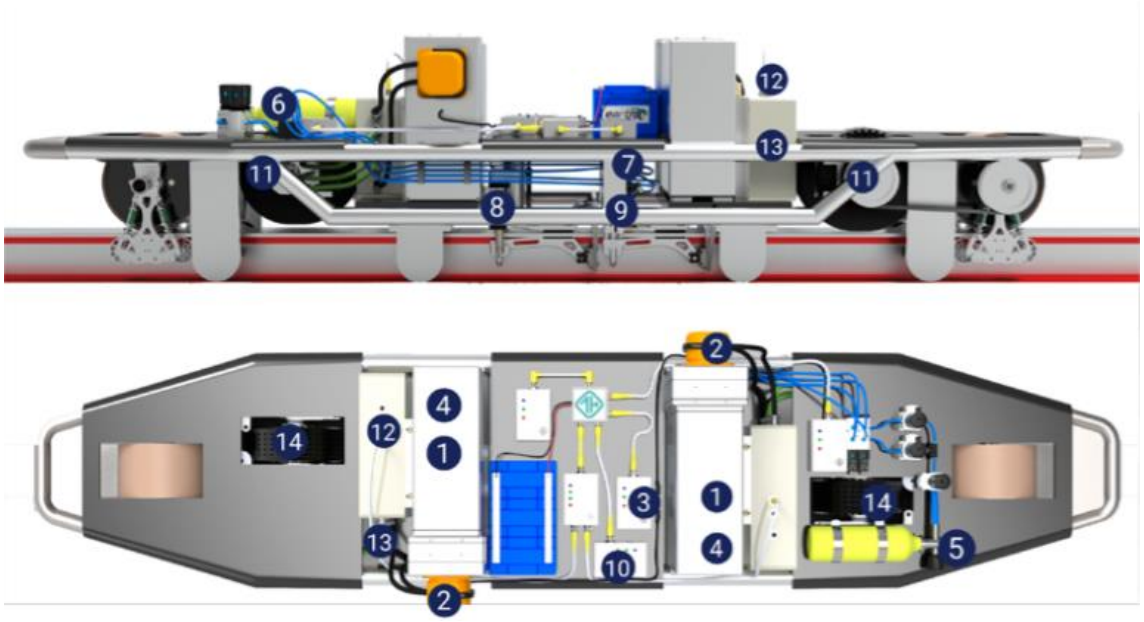

Imagen 6, ubicación de los diferentes sensores

Número	Sensor	Notas
1	Temperatura de las baterías	
2	Corriente de salida de las baterías	
3	Sensores de voltaje de las baterías individuales	Si bien se indica en la posición del microprocesador de energía, su posición física real es el interior de las cajas de las baterías, de forma distribuida.
4	Presión de las cajas de las baterías	Dado que las cajas de baterías van presurizadas, es necesario monitorizar la presión
5	Sensores de presión del sistema neumático de alta presión	El sistema neumático parte de una botella a 3000psi y debe de bajar la presión por etapas hasta los cilindros neumáticos, por ello, existen diferentes sensores para las diferentes etapas.

6	Sensores de presión del sistema neumático de media presión	
7	Sensores de presión del sistema neumático de baja presión	
8	Sensores de posición de los frenos neumáticos	
9	Sensores de posición de los frenos eléctricos	
10	Sensor de presión ambiente	
11	Resolvers de los motores	Los resolvers están situados coaxiales al motor, integrados por el fabricante
12	Sensores de temperatura de los drivers	El driver dispone de sensores de temperatura para los semiconductores encargados de generar la señal de potencia
13	Sensores de corriente y tensión de los motores	El driver dispone también de sensores de la tensión y corriente que está introduciendo al motor
14	Sensores de temperatura de los motores	El motor dispone de un sensor de temperatura en el estator, capaz de ser leído por el driver del motor.

Tabla 7, posición de los principales sensores

Los modelos y rangos de operación de los de los diferentes sensores son los siguientes:

Sensor	*Modelo*	*Magnitud*	*Ubicación*	*Cantidad*	*Rango de operación*	*Salida*
Sensor de temperatura digital	DS18B20	Temperatura	Baterías	2	-55º a 125º	12 bits, por interfaz 1Wire
Sensor de corriente	L06P400S05	Corriente	Líneas de alta tensión	2	0-400A	0-5V
Sensor de presión	SSCSANN001BAAA3	Presión	Baterías Ambiente	2 1	0-1bar	0.33-2.97V
Sensor de presión	PX3AN1BH100PSAAX	Presión	Refrigerante	1	1-7bar	0.5-4.5V
Sensor de presión	PX3AN1BH010BSAAX	Presión	Baja presión	2	1-10bar	0.5-4.5V
Sensor de presión	PX3AN2BS250PAAAX	Presión	Media presión	1	1-17bar	0.5-4.5V
Sensor de presión	M7139-05KPN-500000	Presión	Alta presión	1	1-350bar	0.5-4.5V
Sensor de presión	4003K11	Presión	Alta presión	1	1-350bar	0.5-4.5V
Sensor de temperatura analógico	GE-2102	Temperatura	Refrigerante	1	-40º-120º	PTC 33.277 ~ 338.2 Ω
Sensor fotoeléctrico	QMIC-0N-0A	Reflectividad	2 en cada lado del *Pod*	4	10 a 70cm de distancia, hasta 200m/s	10-30V
IMU	VectorNav VN-100	Aceleración	Cerca del centro de masas	1	+-16g	RS-2323
Resolvers	TS2620N21E11	Posición angular	Motores	2	0-10.000rpm	7Vrms @ 10kHz

Tabla 8, lista de sensores utilizados

Sensorización de las baterías:

El BMS empleado está basado en un diseño propio y uno de los sistemas con más trabajo detrás de todo el vehículo. El sistema permite medir el voltaje de cada pack de batería de forma individual, pero no permite el balanceo de las celdas del pack (tampoco era necesario debido a la metodología de carga).

El BMS debe de ser capaz de trabajar con tensiones de modo común de 700 V, la tensión máxima del pack de baterías. Para poder trabajar con tensiones de modo común tan elevadas, sistemas de aislamiento galvánico se hacen necesarios. El bus de comunicaciones empleado es I2C, que de por si no dispone de ningún tipo de aislamiento galvánico. Sin embargo, existen chips comerciales capaces de proporcionar ese aislamiento. El chip empleado en nuestro caso es el Si8600. Si bien este chip no es un optoacoplador al uso, en lugar de utilizar luz para aislar galvánicamente, emplea radiofrecuencia. El efecto es similar, y permite aislar hasta 5000V durante periodos de operación de 1 minuto.

Una vez con aislamiento galvánico, es posible utilizar el bus I2C sin problemas de seguridad debido a las altas tensiones empleadas. Para monitorizar el voltaje de las baterías, se emplea un ADC I2C conectado entre la tierra relativa de las baterías y la última celda, empleando el conector de balanceo. El modelo elegido es el MCP3221 de Microchip technology. Sin embargo, de este microchip existen únicamente 8 direcciones diferentes, de las cuales únicamente 6 estaban en stock. Como se debían de medir 30 celdas diferentes, era necesaria la incorporación de un multiplexor I2C. Este multiplexor permitía la elección de hasta 8 canales diferentes. Por último, como se ha comentado anteriormente, los buses de comunicaciones están sometidos a una elevada cantidad de ruido electromagnético. Para mejorar la tolerancia del sistema a ruido, se emplean conversores de I2C a I2C diferencial, empleando 2 chips PCA9615, uno cerca del multiplexor, y otro en la placa base de energía.

El esquema queda así:

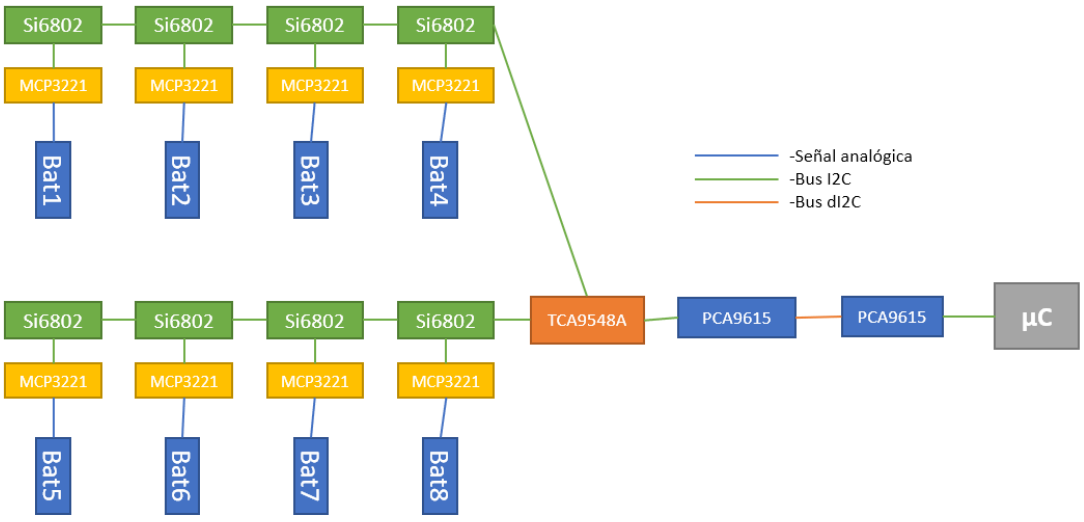

Ilustración 1, esquema del Battery Monitoring System empleado

Pero extendido a lo largo de 5 canales diferentes, con 6 celdas por canal, y en 2 cajas de baterías. En total, una medición de 60 valores de voltaje diferentes.

A esto se le une la medición del voltaje entero del pack que se realiza desde el Driver, y la monitorización de la intensidad que circula por el pack, con el sensor mencionado previamente. En total, 32 mediciones por pack para permitir hacer una valoración precisa del estado de las baterías.

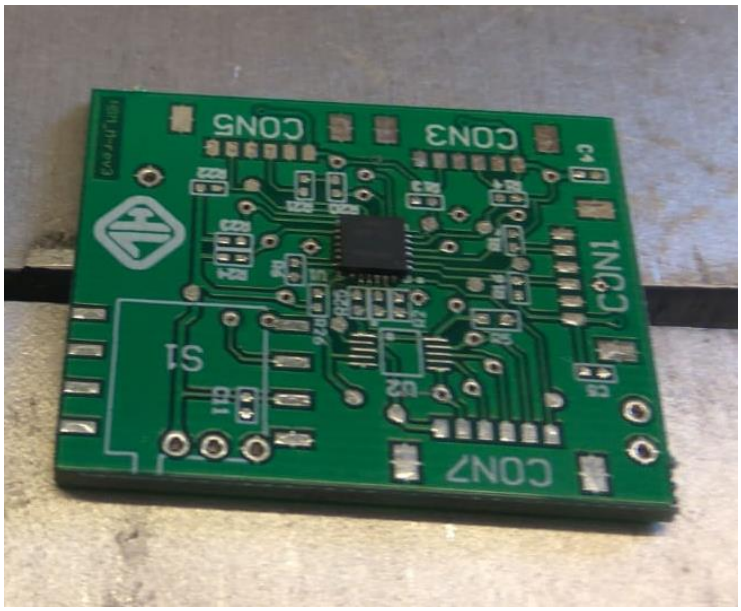

Imagen 7, PCB con el adaptador a I2C diferencial (no soldado todavía, en la posición U2) y el multiplexor (el chip QFN instalado).

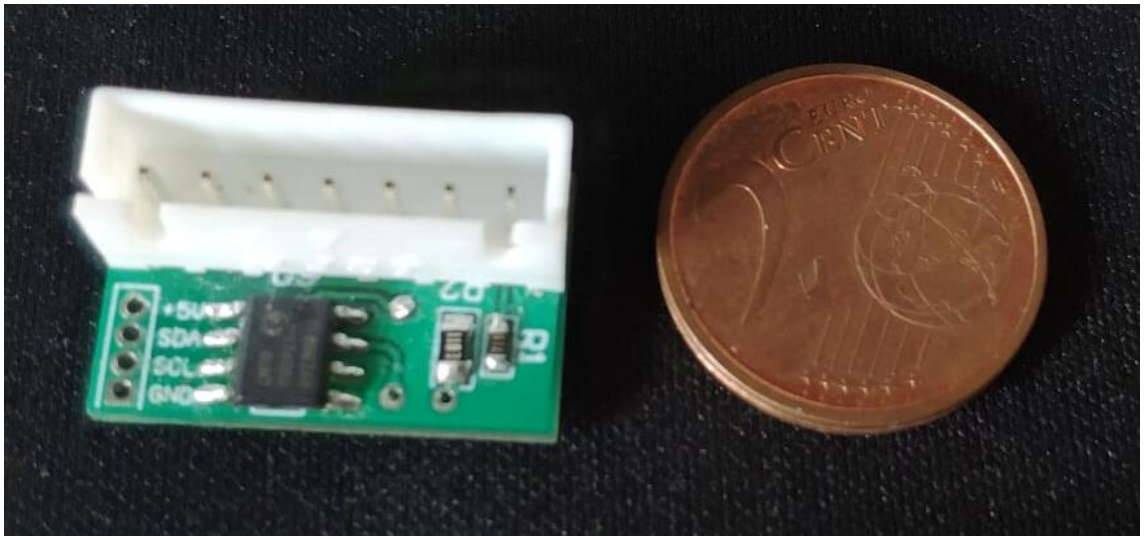

Imagen 8, placa encargada de la lectura de un único pack de baterías. Céntimo como referencia de tamaño. El chip que se ve es el aislador I2C Si6800

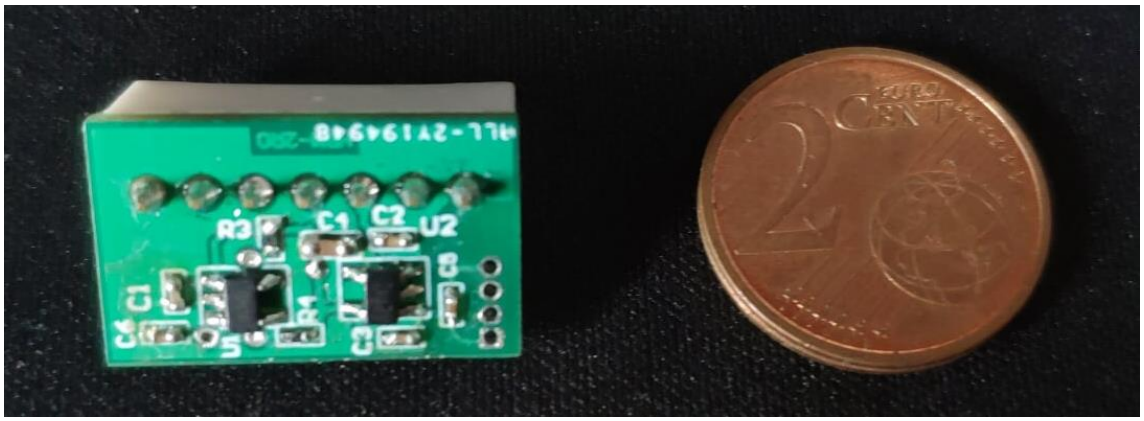

Imagen 9, parte trasera de la placa de lectura. Se aprecia el LDO (chip encargado de alimentar la placa) y el ADC I2C MCP3221

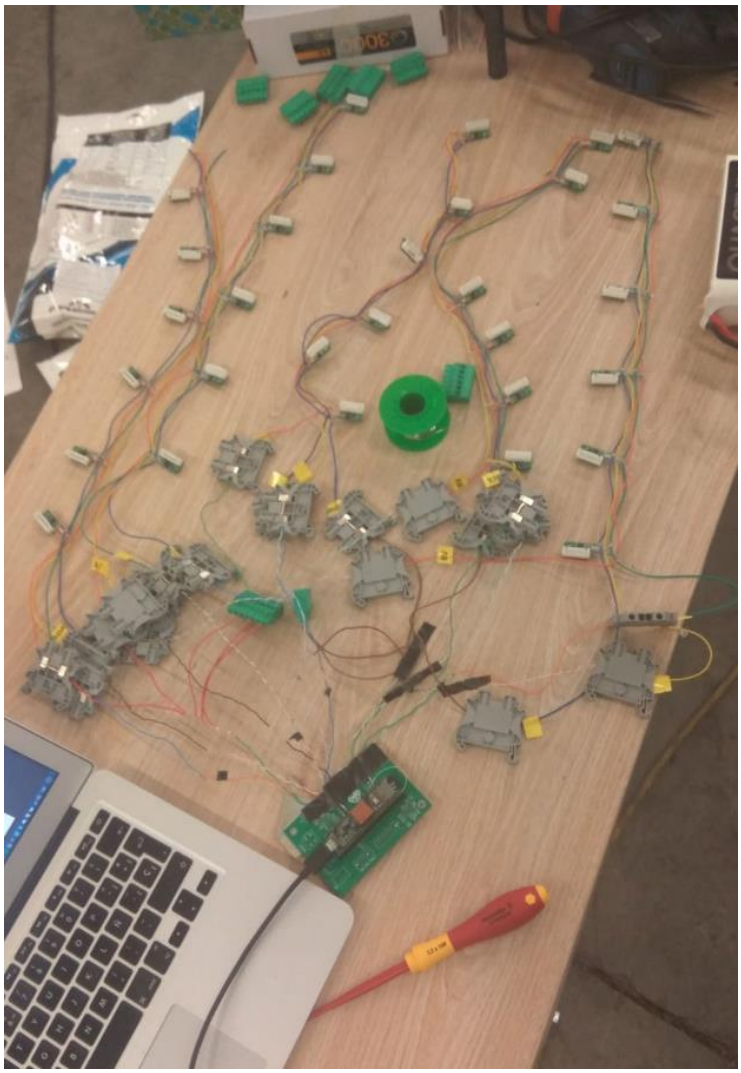

Imagen 10, ensamblaje del sistema de lectura. Se aprecian las 5 ramas diferentes del bus I2C con las 6 unidades de lectura. No se aprecia la placa multiplexora puesto que se estaba realizando una prueba para comprobar la posibiliad de multiplexar por software.

	V1	V2	V3	V4	V5	V6
Wire0 1819:	3402mV	3392mV	3435mV	3331mV	3440mV	0mV
Wire0 3334:	3364mV	3400mV	3359mV	3391mV	3447mV	3392mV
Wire0 78:	3391mV	3383mV	3359mV	3367mV	3402mV	3392mV
Wire1 3738:	3400mV	3381mV	3375mV	3383mV	3400mV	3392mV
Wire2 34:	3402mV	3391mV	3375mV	3400mV	3398mV	3407mV

Imagen 11, salida del software de lectura del voltaje de los diferentes módulos. El voltaje está sin escalar, por eso mide tan poco.

3.1.2.2. Controlador máster:

El controlador máster tiene como funciones las siguientes:

- Controlar el funcionamiento general del *Pod*. Este módulo se encarga de ordenar al resto de módulos qué deben hacer, y recibe información de éstos en tiempo real para decidir qué hacer a continuación. En resumen, llevar la máquina de estados del sistema.
- Recibir datos de los sensores del resto de placas mediante bus CAN.
- Comunicarse con la estación base, enviando datos en tiempo real del funcionamiento del *Pod* empleando todos los sensores disponibles para tal efecto.
- Valorar en caso de emergencia cual es la mejor vía para asegurar la seguridad del *Pod* y de las personas de su entorno.

Además, el controlador incluye sensores de temperatura y presión ambiente. El esquema básico del módulo maestro es el siguiente:

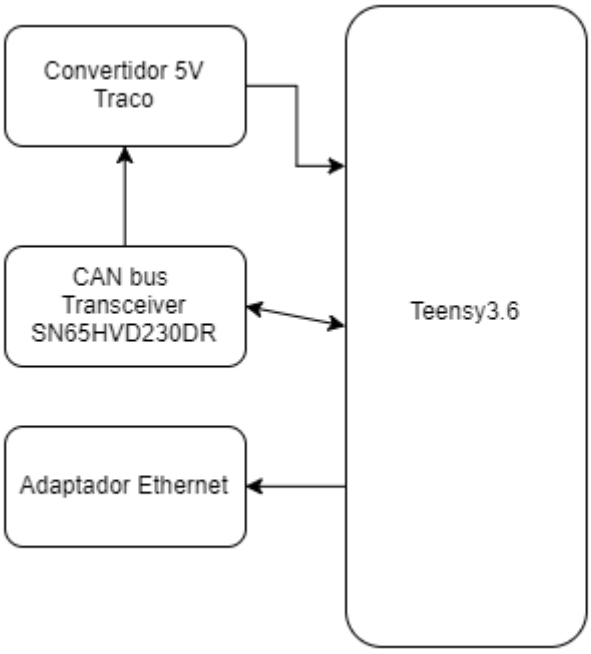

Ilustración 2, arquitectura de hardware de la placa maestra

La PCB necesaria por lo tanto es relativamente sencilla. El adaptador Ethernet elegido es uno estándar basado en el chip Wiz820io, compatible con la Teensy y con un amplio soporte detrás. La comunicación con el chip encargado de Ethernet se realiza por SPI, un bus que permite altas

velocidades. El conversor DCDC de 24V a 5V de Traco power provee de alimentación a la Teensy a partir de los 24V que se distribuyen por la línea CAN. El modelo 3D de la placa es el siguiente:

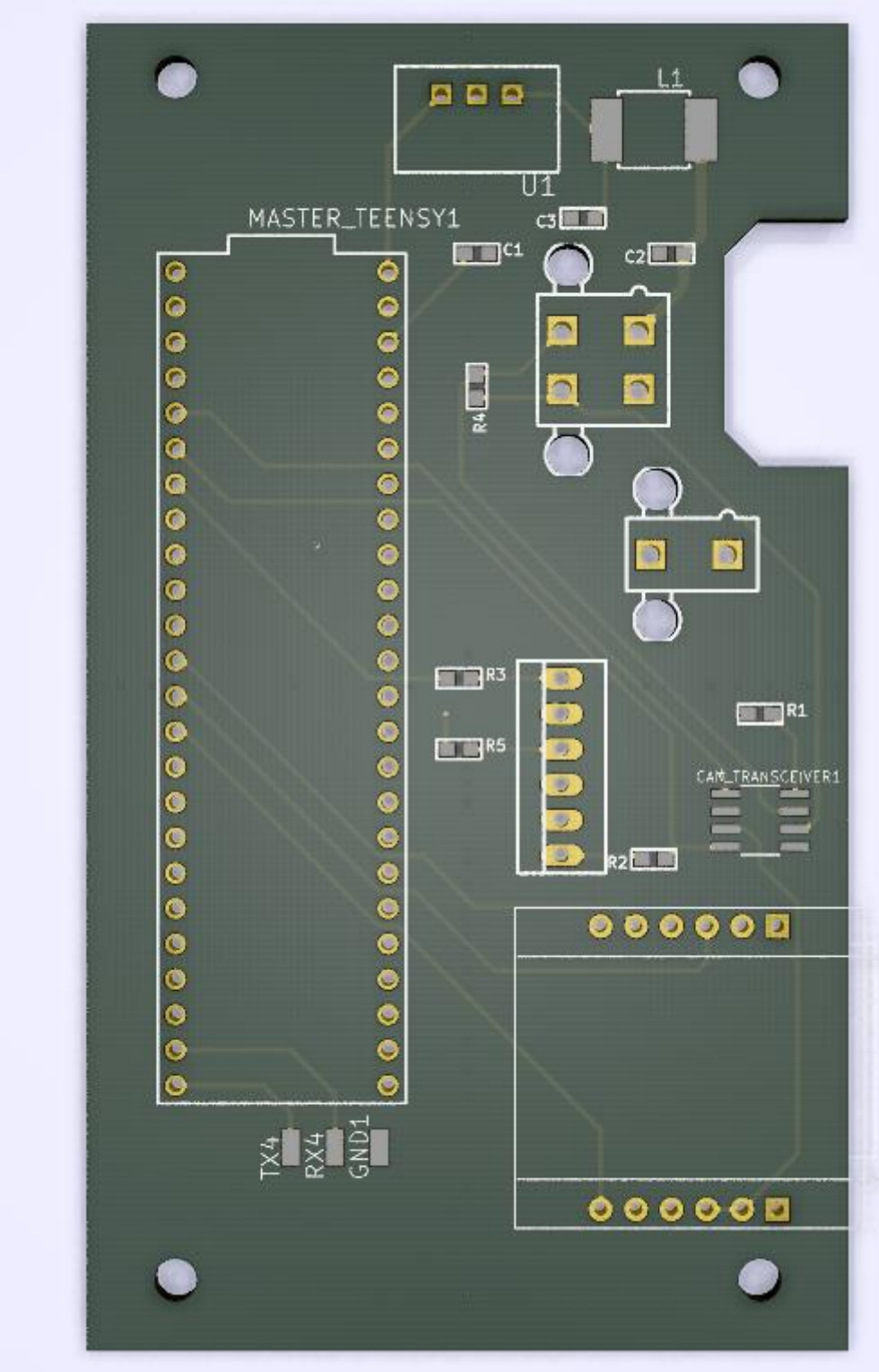

Imagen 12, PCB de la placa maestra

3.1.3.2. Controlador de motores:

El controlador de los motores tiene como función ser la interfaz entre el controlador máster y los drivers que controlarán el funcionamiento de los motores. Los drivers se actúan mediante bus CAN, y en este controlador el empleo de las Teensys 3.6, con sus 2 buses CAN, se hace

necesario, puesto que un bus será para la comunicación con el resto de placas, y el otro para la comunicación con los motores. Si bien es perfectamente posible conectar los drivers de los motores al mismo bus que el resto de microcontroladores, simplifica mucho la tarea y ahorra a los microcontroladores descartar mensajes que no van destinados a ellos separar los buses en dos diferentes.

El driver permite la comunicación a través de bus CAN y de RS232, es capaz de trabajar con hasta 800V de corriente continua en la entrada y de transferir al motor 100kW de potencia. Además, dispone de entradas digitales y analógicas para controlar distintos aspectos de funcionamiento del motor: frenada regenerativa, el *throttle* o estado de velocidad del motor, parada de emergencia, etcétera. El protocolo de comunicación mediante bus CAN se establece en un documento por parte del fabricante.

Las funciones del controlador de motores por lo tanto se resumen en las siguientes:

- Controlar el funcionamiento de los motores a partir de los drivers. Esto implica tanto enviar órdenes a los drivers, como leer los datos que mandan de vuelta con información sobre los distintos sensores que disponen.
- Comunicar con la estación base todos los datos relevantes sobre el estado de funcionamiento de los motores
- Leer la información de los encoders existentes en los motores, e interpretarla para obtener tanto las velocidades como aceleraciones de los motores.
- Elevar una emergencia en caso de que los valores de funcionamiento de los motores o los drivers se salgan de los valores nominales.
- Actuar sobre la bomba de refrigerante del circuito de refrigeración de los drivers.

La PCB necesaria por lo tanto tendrá 2 transceptores CAN conectados a 2 puertos diferentes de la Teensy, y un MOSFET para controlar la velocidad de la bomba de refrigerante usando PWM. El esquema de la PCB es el siguiente:

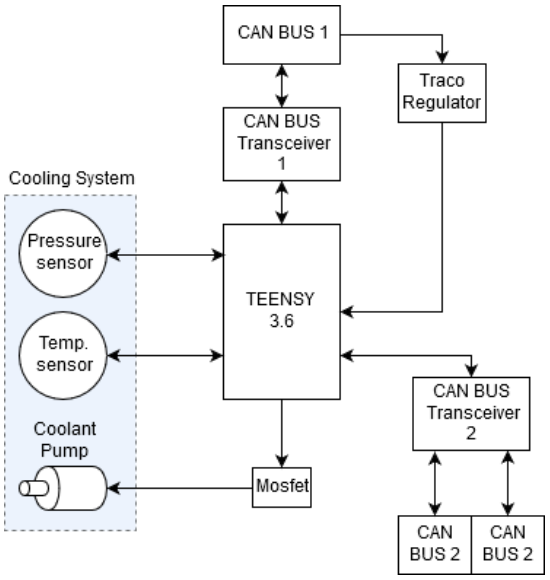

Ilustración 3, arquitectura de hardware de la placa de motores

3.1.3.4. Controlador de navegación:

El controlador de navegación recibe información sobre la mayoría de sensores destinados a posicionar el *Pod* dentro del tubo. Estos sensores son lectores de cintas, que permiten contar el número de cintas reflectantes a radiación infrarroja dispuestas a intervalos regulares por el interior del tubo, y distintas IMUs (Unidades de medición inercial). Las IMUs emplean bus CAN, por lo que tener 2 buses diferentes también se hace necesario. Las funciones por lo tanto son las siguientes:

- Estimar en todo momento con la mayor exactitud posible la posición del *Pod* dentro del tubo y transmitir la información al controlador máster con la mayor celeridad posible.
- Elevar una emergencia en caso de que la aceleración o el frenado no esté dentro de los márgenes nominales.

3.1.3.5. Controlador de frenos:

El controlador de frenos combina los dos tipos de frenado disponibles: frenado neumático y frenado eléctrico, mediante servomotores. Tener 2 tipos de frenos diferentes da bastante resiliencia del sistema a fallos, puesto que tienen que fallar completamente 2 sistemas para que el sistema se quede sin medios de frenar. Las tareas de este controlador son las siguientes:

- Controlar el estado del sistema neumático mediante lecturas de presión en las diferentes secciones del circuito.
- Accionar los frenos al recibir la orden con el mínimo retardo posible.
- En caso de detectar una pérdida de conexión con la placa maestra, activar los frenos de inmediato asumiendo un fallo catastrófico del sistema.

3.1.3.6. Controlador de energía:

Las baterías son una parte crítica del vehículo, puesto que van a almacenar una cantidad considerable de energía. Es por ello por lo que requieren una monitorización bastante exhaustiva para tener la seguridad de que no existe riesgo alguno. Las baterías irán en 2 cajas selladas y separadas, para minimizar su exposición al vacío y asegurar la integridad del vehículo en caso de incendio. Además, habrá multitud de sensores de tensión, intensidad y temperatura repartidos por las cajas. Éstos sensores monitorizarán las 3 variables fundamentales para la salud de las baterías. Las funciones de esta placa serán las siguientes:

- Monitorizar los valores de todos los sensores de temperatura, tensión e intensidad a lo largo de todas las baterías
- Elevar una emergencia en caso de que un valor se salga de los valores nominales, o una advertencia en caso de que esté cerca.

3.2. Software

El siguiente capítulo describe los programas creados para el control del sistema. Debido a que existen 5 microprocesadores, existen a su vez 5 programas diferentes, de los cuales 2 se explicarán en detalle. Como uno de los elementos básicos de los 5 programas es el protocolo de comunicación empleado, el primer capítulo se dedicará a describir en detalle el protocolo de comunicaciones creado.

3.2.1. El protocolo de comunicaciones

El bus CAN determina el formato de cada trama. Sin embargo, qué hay dentro de cada trama se configura por el usuario. Si bien existen protocolos estándar, como CANOpen, debido a los requisitos tan concretos de nuestro sistema, se decidió crear un protocolo desde cero.

Los dos puntos básicos sobre los que se basa el protocolo son los siguientes:

-La placa maestra solicitará información a las placas esclavas en paquetes de solicitud, y las placas esclavas deberán responder de forma acorde

-La placa maestra es capaz de modificar variables al vuelo de las placas esclavas con el objetivo de, o bien configurar parámetros de funcionamiento del *Pod*, o bien para hacer pruebas de software.

El formato de una trama CAN es el siguiente:

-11 o 29 bits de ID, según se usen IDs estándar o IDs extendidos. Las IDs se pueden emplear o bien para identificar el tipo de mensaje, el emisor, el receptor, o cualquier cosa. En nuestro caso, existiendo únicamente 5 placas, y aproximadamente 40 tipos de mensajes diferentes, con 11 bits es suficiente.

-1 bit RTR (Remote Transmisión Request), que se encuentra en desuso por el estándar, y se mantiene por fines de retrocompatibilidad. En nuestro caso, no se empleará.

-3 bits DLC (Data Length), que determinan el número de bytes que contiene la trama

-De 0 a 8 bytes de datos, según lo determinado en el bit DLC.

Para el protocolo, los 11 bits del ID se dividen de la siguiente forma:

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit 10	Bit 11
ID de la placa de destino			Bit de solicitud	ID del mensaje						

Tabla 9, estructura del ID CAN

Tener un ID de la placa de destino permite a las placas rápidamente descartar el mensaje en caso de que no se dirija hacia ellas. Esto ahorra tiempo de ejecución en las placas esclavas. El bit de solicitud funciona de la siguiente forma. Si el bit es 1, significa que el mensaje es una solicitud de información. La información solicitada va acorde al ID del mensaje: cada ID implica que se solicita una trama diferente. Si el bit de solicitud es 0, significa que no es una solicitud, sino una orden de establecer las variables como se indica en el ID de mensaje. Por ejemplo: si la placa máster manda un mensaje de solicitud (bit de solicitud a 1) al bus, con la placa de destino siendo Navegación, y el ID del mensaje se corresponde a un mensaje que contiene todas las aceleraciones en el siguiente formato:

ID:

Decimal	Binary	Hex
11	0001011	0B

Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Acceleration Z	Acceleration Z	Acceleration Y	Acceleration Y	Acceleration X	Acceleration X
Integer X100	Integer X100	Integer X100	Integer X100	Integer X100	Integer X100

Tabla 10, ejemplo de trama CAN

Una vez lo reciba la placa de navegación, mandará lo antes posible el mensaje con la información organizada como se indica en la tabla superior. Si, sin embargo, el bit de solicitud está a 0, la placa de navegación sobrescribirá los valores que tiene de aceleración en memoria con los que le indica el mensaje en la trama.

El uso del bit de solicitud es programar los valores al vuelo. Empleando un dispositivo *sniffer* del bus CAN como el PCAN-USB, es posible inyectar paquetes al vuelo en el bus desde un ordenador. Esto permite por ejemplo realizar pruebas en el sistema en tiempo real, conectando un ordenador al BUS y cambiando variables para ver cómo se comporta el sistema. Inyectando por ejemplo una variable que se va fuera de los rangos nominales, se puede comprobar el tiempo que tarda el sistema en entrar en modo de emergencia, en actuar los frenos etcétera.

Imagen 13, Dispositivo PCAN-USB que permite interfacear un ordenador con un bus CAN

Los IDs de las placas son los siguientes:

Máster	001
ID (receiving)	
Nav	010
ID (receiving)	
Motors	011
ID (receiving)	
Brakes	100
ID (receiving)	
Energy	101
ID (receiving)	

Tabla 11, IDs de las diferentes placas

Y todas las tramas existentes se encuentran en el Anexo 1

El siguiente paso consistía en determinar cómo se iba a implementar a nivel de software el protocolo de comunicaciones. Comenzando por la recepción, en la librería empleada existen 2 alternativas diferentes para gestionar la recepción de mensajes. Éstas 2 alternativas son muy típicas en varios campos de la informática: *Polling* o *Interrupts*.

Utilizando el sistema *polling*, lo que hace el microprocesador es, cada ejecución del bucle principal, pregunta al periférico encargado de gestionar el protocolo de comunicaciones si han

llegado mensajes, y en caso positivo, los lee y despacha. El principal problema de este método es que el periférico tiene una capacidad limitada de almacenar mensajes hasta que son despachados. Pongamos por ejemplo que el periférico del protocolo tiene un búfer capaz de almacenar 10 mensajes, y llegan aproximadamente 100 mensajes por segundo. En este caso, si el bucle principal se ejecuta a 10Hz, será capaz de leer todos los mensajes que le lleguen al dispositivo. Sin embargo, cualquier velocidad de ejecución inferior a 10Hz implicará que el periférico no podrá almacenar suficientes mensajes, que se perderán y dejarán de ser leídos sin notificación alguna.

La alternativa es emplear interrupciones. Cuando se configura una interrupción en un microcontrolador, introduces una pequeña norma en el programa: Cuando pase X, para todo lo que estés haciendo, y haz Y. En este caso, la interrupción se configura de tal forma que, cuando se reciba un mensaje, se interrumpa el flujo de ejecución del bucle principal y se ejecute lo que se denomina habitualmente como una *Interrupt Service Routine (ISR)*. Estas rutinas son funciones que se ejecutan automáticamente en cuanto salta una interrupción. En este caso, la ISR debería decodificar el mensaje, comprobar si se trata de un mensaje para esa placa en concreto, en caso afirmativo ignorarlo y en caso positivo, interpretar el tipo de mensaje.

Sin embargo, las interrupciones plantean otro tipo de problemas. Por ejemplo, al interrumpir la ejecución del bucle principal, si la interrupción sucede en un momento que requiere un control estricto del tiempo de ejecución (por ejemplo, en una lectura analógica existe cierto timing para asegurar que el condensador encargado de mantener la tensión a leer se ha cargado correctamente, o la escritura de un mensaje en un bus de comunicaciones requiere un estricto control del tiempo, acorde a la velocidad del bus), este control preciso del tiempo se pierde al no tener un control exacto del tiempo de ejecución de la interrupción. Para solucionar este problema, es buena práctica que las *ISR* sean funciones lo más sencillas posibles. Además, existe la posibilidad de desactivar las interrupciones durante momentos críticos. Sin embargo, esto implica la posibilidad de que se pierda algún mensaje si durante el periodo en el que las interrupciones están desactivadas se recibe.

La solución empleada es una combinación de ambas técnicas: el principal medio para gestionar los mensajes serán las interrupciones. Durante la ISR, según la prioridad del mensaje, hay dos alternativas:

-Si el mensaje es prioritario, se ejecutará el código necesario en la propia interrupción. Si bien esto aumentará el tiempo de ejecución de la interrupción notablemente, los mensajes prioritarios serán escasos y principalmente cobrarán importancia en caso de emergencia. Por ejemplo: existe una trama de mensaje que implica una frenada de emergencia. Al recibir este mensaje, los motores deben desactivarse y los frenos activarse lo antes posible. Por ello, en caso de que las placas de *Motors* y la de *Brakes* reciban esta trama, la desactivación de los drivers y la activación de los frenos se ejecutará en la propia interrupción. En condiciones normales, esto es un desperdicio de ciclos de reloj: desactivar los drivers implica enviar una trama por otro periférico CAN y activar los frenos, accionar una señal digital, pero en condiciones de emergencia se ignora.

-Si el mensaje no es prioritario, se guarda toda la información relevante del mensaje en variables adecuadas y se activa una flag concreta para el tipo de mensaje. En la próxima ejecución del bucle principal, al comienzo de éste, de acuerdo con las flags activadas se realizarán las acciones correspondientes a cada mensaje. Tras ser despachadas, cada

flag se reinicia a su valor original, preparadas para volver a ser activadas en caso de que se reciba otro mensaje.

Después de leer todas las flags, además, el sistema comprueba que no hay ningún mensaje en el búfer, y de haber uno, lo trata. De esta forma, si por ejemplo se ha recibido un mensaje mientras las interrupciones estaban desactivadas, el mensaje no se pierde.

3.2.2. Comunicaciones con tierra

Para las comunicaciones con tierra, SpaceX aporta un sistema de comunicaciones inalámbrico, basado en un hilo radiante que recorre toda la longitud del tubo. A nivel usuario, este sistema es básicamente un puente ethernet. Mediante un controlador que aportan ellos a cada equipo (y que por consiguiente tiene que pasar distintas pruebas de integración), establecen un puente ethernet entre el vehículo y el equipo. Para simular estas condiciones, únicamente es necesario un enrutador o un switch Ethernet para poner al vehículo y el ordenador de control en la misma subred.

Para las comunicaciones con nuestro *Pod*, empleamos dos protocolos distintos: TCP y UDP. Se emplean estos dos protocolos por sus diferencias:

-TCP es un protocolo orientado a la conexión. Esto quiere decir que antes de enviar cualquier tipo de información, existe un proceso entre el cual servidor y cliente establecen una conexión, y continuamente se comprueba que esta conexión sigue en funcionamiento, y se han recibido todos los datos en el orden adecuado. Con TCP, puedes saber exactamente si la otra parte ha recibido el mensaje correctamente, o no.

-UDP es un protocolo no orientado a conexión. Esto quiere decir que una de las partes manda los datos a la dirección establecida, y no importa, ni sabe, si hay alguien recibiéndolos o no.

Debido a todo el proceso que implica crear una conexión y asegurarse de que sigue funcionando, TCP es considerablemente más lento que UDP. Sin embargo, te permite con facilidad diagnosticar problemas en la conexión.

En nuestro sistema, las comunicaciones estación base-*Pod*, tienen 3 objetivos principales:

-Comprobar que existe una conexión, en cuyo caso el software embebido que se ejecuta en la placa Máster sigue funcionando correctamente.

-Recibir datos de telemetría de todos los sensores importantes.

-Enviar parámetros de configuración.

Cabe decir de antemano, que, si bien el microcontrolador de la placa Máster es relativamente potente, la comunicación Ethernet es bastante intensiva en el uso de recursos computacionales. Es decir, mandar paquetes de 1.5 kilobits con frecuencias de 1000Hz por ejemplo consume bastante tiempo de trabajo del procesador, Por ello, el controlador no manda constantemente datos a la estación base: lo hace bajo solicitud.

En nuestro caso, las comunicaciones se realizan de la siguiente forma:

-El *Pod* crea un servidor TCP para que la estación base se conecte. En ese servidor TCP, el *Pod* está esperando constantemente mensajes *request*. A cada mensaje *request* recibido, el microcontrolador prepara, por UDP, un paquete de datos con la información

de los sensores, y lo envía al siguiente ciclo de ejecución del bucle principal. La frecuencia de cada request se puede configurar desde la estación de tierra.

-El *Pod*, además, escucha en UDP en un determinado puerto para los mensajes de la estación base. Estos mensajes se mandan desde tierra y pueden contener desde parámetros de configuración hasta órdenes, como actuar los frenos, encender o apagar los motores, etcétera.

El mensaje de *request* que se manda desde la estación base al *Pod* es únicamente "a\". La contrabarra se emplea como carácter de escape: el parseador por software del microcontrolador está configurado de tal forma que lee hasta que encuentra una contrabarra.

Los mensajes de datos que manda el *Pod* a la estación base tienen dos tipos de formato:

-El formato reducido manda los datos que varían con mayor velocidad: distancia, velocidad, voltaje, corriente, etcétera.

-El formato completo manda todos los datos: los que varían rápido y los que tienen una dinámica lenta (temperaturas, presiones, etcétera)

Esto se debe al coste computacional de crear el paquete de datos para mandar. Crear un paquete de datos implica trabajar con *Strings*, un tipo de variable para el que los microprocesadores por lo general no están optimizados para trabajar. Si bien con *Strings* cortas esta falta de optimización es prácticamente inapreciable, cuando se trabaja con *Strings* del tamaño de un paquete Ethernet (del orden de 1 kilobyte), la pérdida de rendimiento es notable.

Por ello, de cada 10 solicitudes de datos, 9 se responden con un mensaje reducido y 10 con un mensaje completo, aumentando considerablemente la velocidad de ejecución del código sin afectar a la telemetría.

El formato de los mensajes es *JSON (JavaScript Object Notation)*. Este formato permite el intercambio de datos entre distintas plataformas, y tiene la ventaja de que es mucho más fácil de parsear (interpretar) que varios formatos. Permite, además, varios tipos de variables, números, cadenas, matrices, o diccionarios (conjuntos clave-valor).

En nuestro caso, el mensaje tiene un formato similar al existente en el Anexo 2, donde básicamente el mensaje es del tipo: "nombre_sensor": valor_sensor, todo dentro de campos que permiten organizar los distintos sensores.

En el caso del mensaje hacia el *Pod*, el tipo de mensaje queda establecido en esta tabla:

Software				
Int/Bool	Range	Module	Tag GUI	notes
Int	--	Motors	aotorSpeed	sets speed for the motors
Int	0/100	Motors	motrs_CoolantPump	sets pwm duty cycle to the coolant pump
Bool	0,1	Brakes	brakes_3and4Set	0 means: 28->0 and 27->1 then open actuators 3 and 4 1 means: 28->1 and 27->0 then close actuators 3 and 4

Bool	0,1	Brakes	brakes_1and2Set	0 means: 30->0 and 29->1 then open actuators 1 and 2 1 means: 30->1 and 29->0 then close actuators 1 and 2
Int	0-100	Brakes	brakes_ActuatorsSpeed	sets the pwm value for all 4 pins to the drivers maybe a vector to change it individually?
Bool	0 , 1	Brakes	brakes_valve1State	opens/close pneumatic valve 1
Bool	0 , 1	Brakes	brakes_valve2State	opens/close pneumatic valve 2
Bool	0 , 1	Brakes	brakes_valve3State	opens/close pneumatic valve 3
Bool	0 , 1	Brakes	brakes_valveAuxState	opens/close pneumatic valve aux
Bool	0 , 1	Energy	energy_ReadingBEnable	enable/disable battery reading in B connector
Bool	0 , 1	Energy	energy_ReadingAEnable	enable/disable battery reading in B connector
Bool	0 , 1	Energy	energy_BatteryBSwitch	conects/disconects battery B to the driver
Bool	0 , 1	Energy	energy_BatteryASwitch	conects/disconects battery A to the driver

Tabla 12, esquemático del mensaje de actuación

Además, existe un parámetro llamado “Order” que, si lee un código, ejecuta una orden compleja (como, por ejemplo, iniciar la competición, desactivando las limitaciones y encendiendo el motor al 100%)

3.2.3. Software embebido.

El software embebido se divide en el software dedicado a cada una de las cinco placas. Mientras que el software de las placas esclavas suele ser relativamente sencillo (básicamente implica leer datos de los sensores, guardarlos en memoria, y enviarlos en el formato adecuado cuando se manda una solicitud desde el bus de comunicaciones), el software de la placa Máster controla prácticamente todo el sistema, el inicio, las comprobaciones iniciales de los sensores, las órdenes de arranque, y las comunicaciones con tierra.

3.2.3.1. Placa Máster

El código encargado de ejecutarse en la placa Máster tiene 3 tareas fundamentales:

-Gestionar el vehículo, mediante la máquina de estados que gobierna el funcionamiento de éste

-Gestionar las comunicaciones con tierra

-Gestionar las comunicaciones con el resto de placas.

A la hora de higienizar el código, este está distribuido de forma que las diferentes funcionalidades queden distribuidas en diferentes archivos (*librerías*) que contienen las diferentes funciones que se requiere para cada apartado. La distribución en el IDE (*Integrated Desktop Environment,* el programa encargado de gestionar, compilar y debuggear el código), es la siguiente:

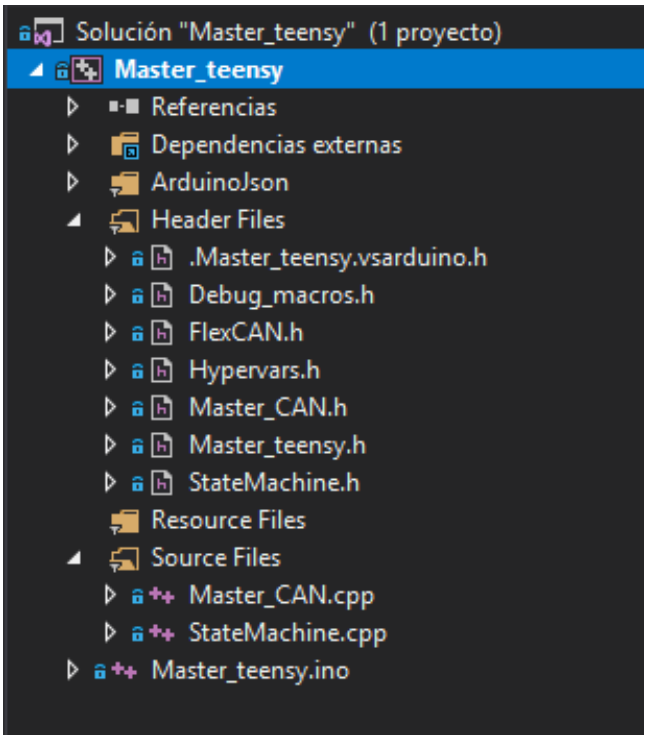

Imagen 14, Lista de archivos del código fuente de la placa maestra

Cada archivo contiene diferentes partes fundamentales para el funcionamiento del sistema. Los archivos de *header* no contienen, por norma general, código. Contienen fundamentalmente *macros* o pre-declaraciones, que permiten al compilador funcionar con normalidad cuando se trabaja con varios archivos de código fuente. Los archivos .cpp contienen todo el código fuente del programa.

3.2.3.1.1. Macros

Empezando desde arriba, el archivo *Debug_vars.h* contiene Macros que activan o desactivan en el código funcionalidad de debug. La funcionalidad de debug existente en el código son comunicaciones por USB hacia el ordenador en las que se enumeran estados por los que pasa el código, el estado de las variables, etcétera. Existen diferentes macros que informan sobre diferentes partes del código, por ejemplo, la recepción de paquetes, el envío de paquetes, etcétera. Por ejemplo, la macro `#define CAN_INPUT_DEBUG 1` informa de todos los mensajes recibidos por el bus CAN, y los muestra por completo por puerto Serie en el ordenador.

El header *FlexCAN.h* incorpora la librería de código abierto empleada para hacer uso del periférico CAN del microprocesador.

El header *Hypervars.h* contiene una pre-declaración de todas las variables en las que se almacenan los datos de los diferentes sensores del *Pod*. Esto se debe a que, aunque estas variables se declaran en el código principal (*Master_teensy.ino*), las distintas librerías también hacen uso de ellas (por ejemplo, la librería *Master_CAN.cpp* actualiza sus valores en memoria). Para el compilador, estas librerías deben de tener constancia de la existencia de esas variables, y en este archivo está toda la información necesaria para ello.

3.2.3.1.2. Comunicaciones CAN

El archivo *Master_CAN.h* contiene las pre-declaraciones de todas las variables y funciones relacionadas con la librería *Master_CAN.cpp*. Al igual que con las variables, cuando accedes desde un archivo de código fuente a una función incorporada en otro archivo de código fuente,

el compilador debe de tener constancia de dónde está esa función, y para ello se indica en el *header* correspondiente.

Con el resto de headers sucede algo similar a los dos anteriores.

El archivo *Master_CAN.cpp* contiene todas las funciones relacionadas con la lectura y escritura de datos empleando el bus CAN. Existen 2 funciones principales y varias funciones auxiliares. Las 2 funciones principales son:

```
-void HyperCAN::decodeCANmsg(CAN_message_t &CANmsg_input) {
-void sendCANmsg(uint8_t target_id, uint8_t msg_code, uint64_t payload,
bool request)
```

La primera, como su nombre indica, decodifica un mensaje recibido por CAN. Esta función realiza todo el proceso indicado previamente en el punto 3.2.1, y cuando recibe un mensaje con información, actualiza las variables en memoria con la información. Por ejemplo:

```
                else if (msg_id.data_id == CANmsg_id_accelerations) {
                        stateMachine::last_comm_nav = 0;

                        int16_t accel_x = CANmsg_input.buf[0] | CANmsg_input.buf[1] << 8;
                        int16_t accel_y = CANmsg_input.buf[2] | CANmsg_input.buf[3] << 8;
                        int16_t accel_z = CANmsg_input.buf[4] | CANmsg_input.buf[5] << 8;
                        hyperVars::accel_x = accel_x / 100.0 / 9.81;
                        hyperVars::accel_y = accel_y / 100.0 / 9.81;
                        hyperVars::accel_z = accel_z / 100.0 / 9.81;

#if CAN_INPUT_DEBUG
                        Serial.println("Aceleraciones actualizadas");
                        Serial.print(hyperVars::accel_x);
                        Serial.print("  ");
                        Serial.print(hyperVars::accel_y);
                        Serial.print("  ");
                        Serial.println(hyperVars::accel_z);
#endif
```

Este fragmento de código se ejecuta si el ID del mensaje se corresponde con una trama en la que se han almacenado las aceleraciones medidas por la IMU. En caso de ejecutarse, actualiza las variables en las que se guarda la aceleración, y si está activada la macro `CAN_INPUT_DEBUG`, muestra la información por puerto serie.

La segunda, también con un nombre bastante autodescriptivo, envía un mensaje por el puerto CAN. Como primer argumento, acepta el código de 3 bits que identifica a la placa de destino. El segundo argumento es el código que identifica a la trama. El tercer argumento contiene la carga de pago del mensaje y el cuarto indica si es una *request* de datos. Empleando estos argumentos, la función crea el mensaje y lo manda por CAN.

El archivo *State_machine.cpp* contiene toda la lógica que implementa en el código la máquina de estados del *POD*.

3.2.3.1.3. La máquina de estados

La máquina de estados del *Pod* consta de dos etapas, una superior y otra inferior: la primera, superior, es la encargada de detectar emergencias y en caso afirmativo suspender la operación normal del *Pod*, y la segunda es la que regula el comportamiento del *Pod* cuando está operando de forma nominal.

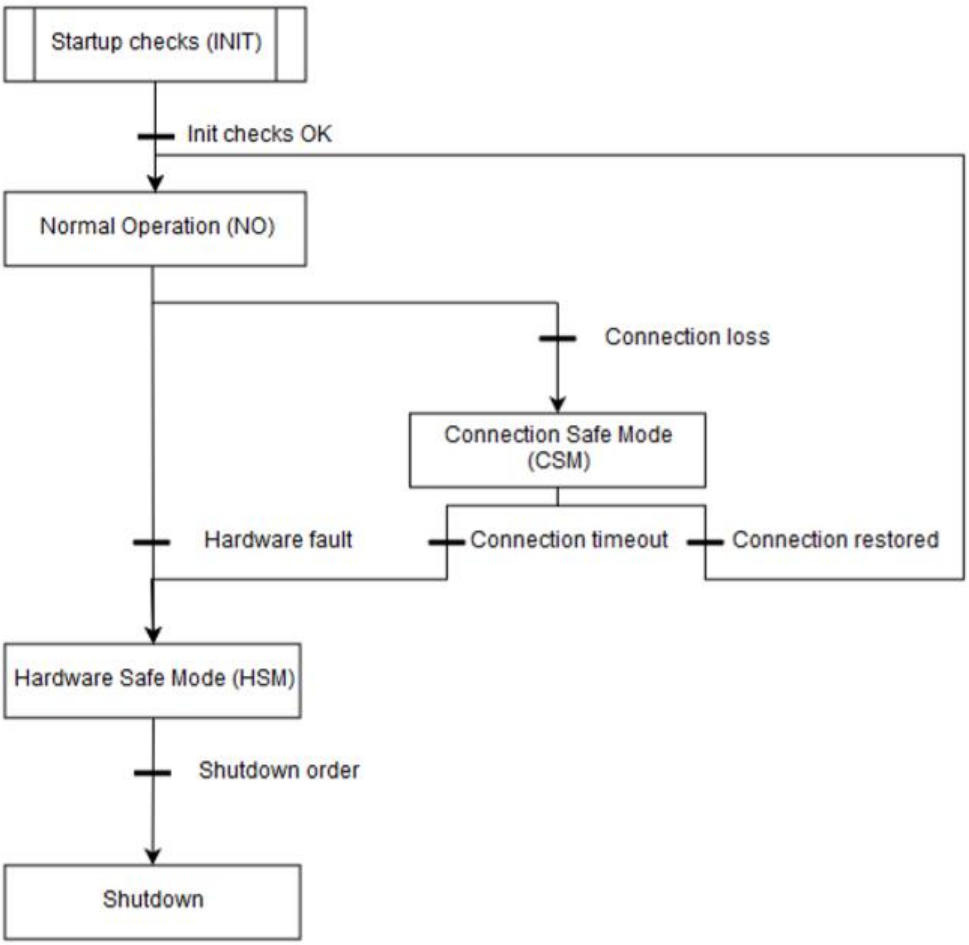

Ilustración 4, diagrama de estados de la máquina de estados superior

Esta máquina consta de 5 estados, llamados en el código superestados, 4 de los cuales son estados de software. El primer estado, *INIT*, se ejecuta nada más encender el *Pod*. Para pasar al siguiente estado, *Normal Operation (NO)*, el *Pod* comprueba 2 cosas:

-Todas las placas han respondido a un mensaje, por lo que están encendidas y funcionando normalmente.

-Existe conexión con la estación base.

-Todos los sensores están dentro de su rango nominal.

Una vez realizadas estas comprobaciones iniciales, el *Pod* entra en el estado *Normal Operation*. Dentro de este estado se ejecuta otra máquina de estados, la inferior, que se explicará a continuación.

Si durante el estado *Normal Operation* se pierde la conexión con la estación base, el *Pod* entra en estado *Connection Safe Mode*. En este estado, el *Pod* ejecuta sus funciones normalmente tal y como estaba haciendo en el estado anterior, pero inicia un contador. Si no se ha reestablecido la conexión en un intervalo de 0.5 segundos, se considera que se ha perdido y entra en modo seguro. Si se restaura la conexión, vuelve al estado normal.

El estado *Hardware Safe Mode* busca proteger al vehículo de cualquier forma. A este estado se entra sea cual sea la emergencia que se haya detectado: un sensor fuera de rango nominal, la

comunicación con una placa perdida (se considera pérdida de comunicación cuando una placa no ha contestad a una solicitud de datos en menos de 50ms), etcétera. Por ello, la respuesta que hace es genérica y abarca la mayor cantidad posible de alternativas. En este estado (y si está activada una *flag* que indica que el *Pod* ha iniciado el movimiento), la placa Máster manda inmediatamente 3 órdenes a 3 placas distintas:

-A la placa Brakes para que active ambos frenos, neumáticos y eléctricos.

-A la placa Energy para que abra los contactores que cierran el circuito de las baterías. Esta acción deja sin alimentación a los motores y desenergiza el sistema en caso de emergencia eléctrica

-A la placa Motors para que desactive los motores inmediatamente, dejando el rotor libre.

A partir de aquí, existe una gran casuística para la que se puede entrar en detalle:

-Si la placa Motors pierde la alimentación, se desactivarán los contactores de las baterías, dejando a los motores sin energía y por lo tanto libres, y se actuarán los frenos, parando de forma segura el *Pod*.

-Si se pierde la alimentación de la placa de Energy, la placa Motors desactiva los motores y la placa de Brakes activa los frenos, parando de forma segura el *Pod*.

-Si se pierde la alimentación de la placa Navigation, aunque no se pierda información indispensable para el funcionamiento del *Pod* (puesto que la medida de velocidad y de distancia se obtiene directamente desde el encoder del motor, a través de la placa Motors), puede indicar un problema eléctrico, y por lo tanto se activará el procedimiento de emergencia.

-Si se pierde la alimentación de la placa Máster, todas las placas tienen integrado también una mini máquina de estados similar a la anterior. Básicamente, si pasan más de 50ms sin recibir un mensaje de la placa Máster, la placa entrará en modo de emergencia, y realizará la acción correspondiente (abrir contactores, desactivar motores, activar frenos).

El caso más crítico es perder el contacto con la placa de Brakes. En este caso, no se podrán activar los frenos eléctricos. Sin embargo, los frenos neumáticos están configurados de tal forma que requieren que la placa de Brakes tenga alimentación para estar retraídos (es decir, requieren una señal digital en Alto para estar retraídos), por lo que se actuarían inmediatamente (y tienen la capacidad de frenar el *Pod* a velocidad máxima en el margen requerido). Además, existe un feedback entre los frenos y el driver del motor: la señal digital que la operación del campo magnético de los drivers está conectada a la señal de actuación de los frenos neumáticos. Si se accionan los frenos neumáticos, ya sea de forma manual, automática o por emergencia, los motores eléctricos dejarán de funcionar automáticamente, evitando el potencial caso problemático de estar frenando y acelerando a la vez.

El último estado es *Shutdown.* En este estado, todos los contactores están abiertos y el *Pod* considera que está apagado, aunque la aviónica reciba potencia. No realizará ninguna acción hasta que no se reinicie manualmente.

La máquina de estados inferior se ejecuta únicamente cuando el *Pod* está en el superestado *Normal Operation*. El diagrama de estados es el siguiente:

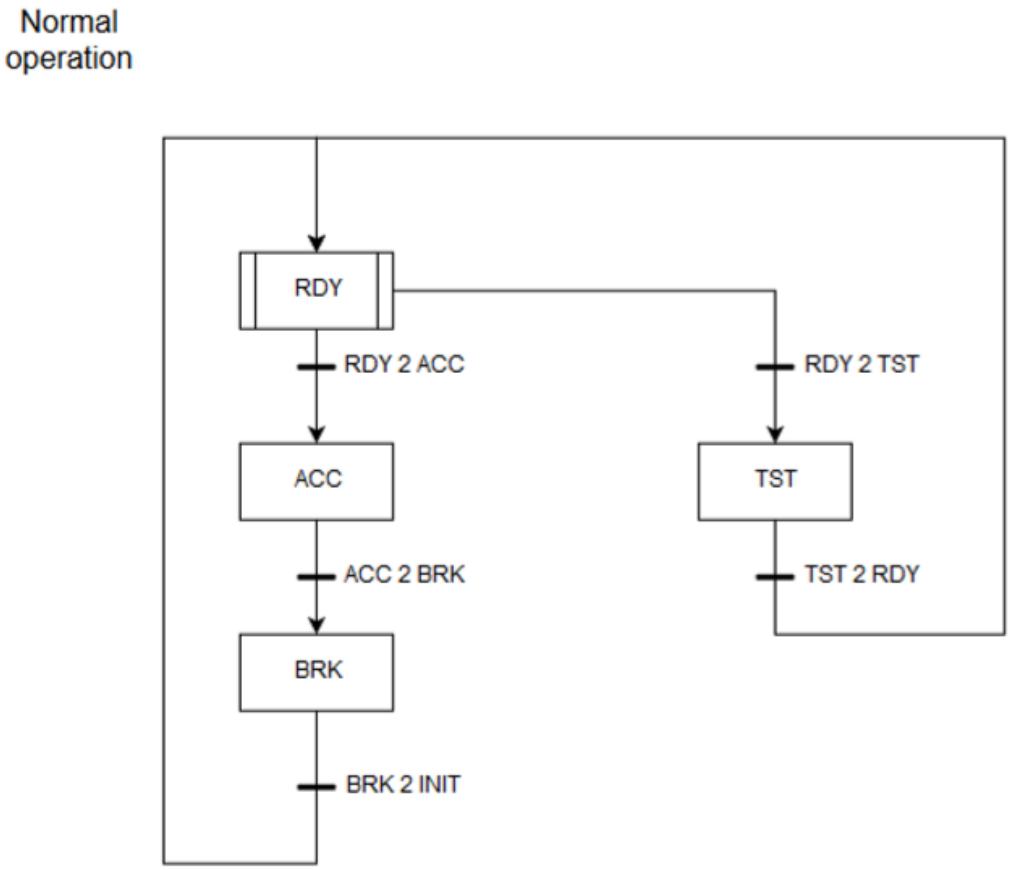

Ilustración 5, diagrama de estados de la máquina de estados inferior

La máquina consta de cuatro estados: Ready (RDY), Acceleration (ACC), Braking (BRK) y Test (TST).

ESTADO	EXPLICACIÓN
READY	Estado por defecto. En este estado el *Pod* está esperando la orden para proceder al estado de Test, o de iniciar el run. En este estado los sensores se comprueban periódicamente (ya que eso es una tarea de la máquina de estados superior. Los contactores están cerrados por lo que los motores tienen alimentación, salvo que se mande la orden de apertura desde la estación base
ACCELERATION	Cuando se da la orden de arrancar, la placa Máster mandará la orden por el bus de empezar el Run. La placa de motores desactivará cualquier limitación. pondrá los motores al máximo, y empezará a integrar la velocidad recibida desde el driver para calcular la distancia recorrida.
BRAKING	Para empezar a frenar se debe de cumplir una de dos condiciones: O se ha recorrido la distancia que se ha configurado como límite, o se ha sobrepasado el tiempo que, de acuerdo con las simulaciones, debería durar el run. Una vez se llega a este estado, la placa Máster manda la orden de

	frenar a la placa Brakes. Esta orden también llega a la placa Motors, que desactiva los motores, y a la placa de Energy, que abre los contactores de las baterías.
TEST	Este estado sirve para comprobar y modificar las variables importantes del sistema (velocidad máxima, distancia máxima, timeout antes de empezar la frenada) y para accionar manualmente los distintos actuadores del vehículo. Existe para evitar que por error se actúen los frenos en modo normal, o que se activen los motores.

Tabla 13, Lista y explicación de los distintos estados de la máquina de estados inferior

La tabla resumen de las transiciones empleadas para la máquina de estados es la siguiente:

De	A	Requisitos (condición Y)	Mecanismo para determinar la transición
INIT	NO	Conexión con la estación base establecida Conexión con todas las placas esclavas establecida Todos los sensores en valores nominales	Paquetes *Keep-alive* enviados desde la estación base al *Pod*, y mensajes enviados por el bus CAN.
RDY	TST	Solicitud desde la estación base	La estación base enviará un paquete solicitando entrar en modo Test
RDY	ACC	Solicitud desde la estación base Velocidad medida < 1rpm Aceleración medida < 0.1Gs	La estación base solicitará al *Pod* iniciar el run.
ADD	BRK	Distancia medida>distancia límite Tiempo medido>tiempo límite	La distancia se mide integrando la velocidad obtenida desde los resolvers del motor y empleando los lectores de cintas reflectantes. El tiempo se calcula empleando el reloj interno del microprocesador de la placa Máster
BRK	INIT	Velocidad medida < 1rpm Aceleración medida < 0.02Gs	La velocidad se mide empleando los resolvers del motor, y la aceleración, mediante la IMU
TST	INIT	Solicitud desde la estación base	La estación base enviará un paquete solicitando salir del modo Test
NO	HSM	Pérdida de conexión con alguna placa, pérdida de potencia, o algún sensor fuera de los valores nominales	Paquetes *Keep-alive* permitirán mantener el contacto entre las diferentes placas
NO	CSM	Pérdida de conexión con la estación base	Paquetes *Keep-alive* entre la estación base y el *Pod*
CSM	NO	Conexión con la estación base reestablecida	Paquetes *Keep-alive* entre la estación base y el *Pod*
CSM	HSM	Tiempo sin reestablecer la conexión > 1s	Contador de tiempo en el microprocesador
HSM	OFF	Orden desde la estación base o interruptor físico en el sistema.	Interruptores software o hardware

Tabla 14, Listado de las transiciones y mecanismos empleados para detectarlas

El código para la máquina de estados contiene 2 variables del tipo *Enum*. Estas variables permiten asignar una *String* a lo que, internamente, es un valor numérico, lo que facilita la lectura del código. En este caso, estas 2 variables representan el superestado y el subestado:

```
enum super_State {
      Initialization,
      NormalOperation,
```

```
        ConnectionSafeMode,
        HardwareSafeMode,
        Shutdown
    };

    enum sub_State {
        Ready,
        PreAcceleration,
        Acceleration,
        Braking,
        Test
                                    };
```

Además de las dos variables, existe una función para cada transición, que comprueba si se cumple o no la condición para la transición, y en caso afirmativo, actualiza el valor del estado al nuevo. Por ejemplo:

```
    void check_init_transition() {
        bool healthCheck = performHealthCheck();
        if (healthCheck) {
#if STATE_MACHINE_DEBUG
                Serial.println("Initialization complete. System entering
Normal Operation");
#endif
                superState = NormalOperation;
        }
    }
```

Esta función en concreto comprueba la condición de transición del estado *INIT*. Además de una función para cada transición, existen dos funciones generales que, básicamente, en función del estado actual, comprueban las transiciones adecuadas. Por ejemplo, si estás en el estado *TST*, es una pérdida de tiempo comprobar las transiciones que cambiarían de *ACC* a *BRK*. Estas funciones se encargan de ahorrar tiempo de ejecución comprobando solo las transiciones importantes, y son las que se llaman desde el código principal a la hora de comprobar la máquina de estados.

3.2.3.1.4. Código principal

El código principal de la placa Máster contiene la lógica principal por la que se rige esta placa. El bucle principal tiene la siguiente estructura:

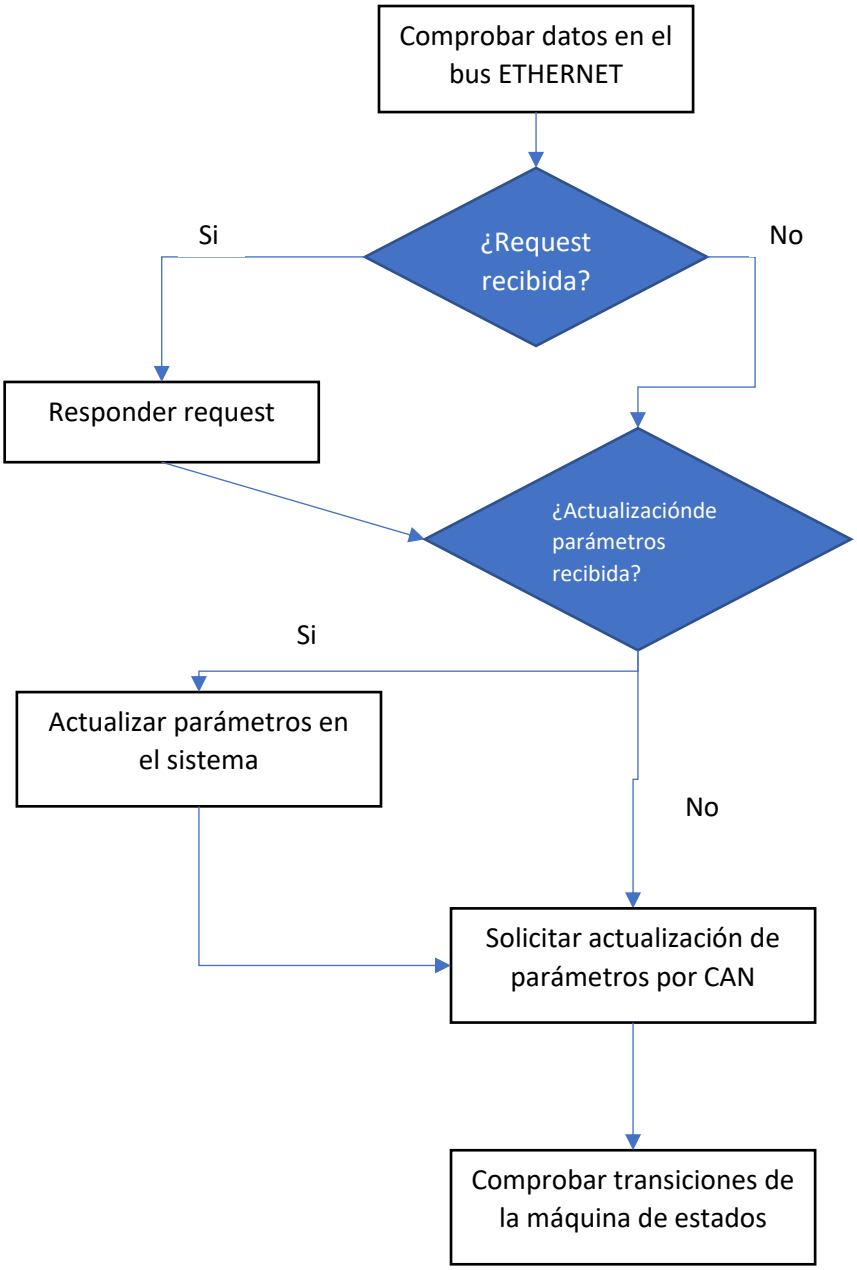

Ilustración 6, diagrama de flujo del código de la placa maestra

Salvo toda la parte de comunicaciones por Ethernet, el resto del código está implementado en las librerías descritas anteriormente. Esto simplifica en gran medida el código en el bucle principal, que se limita prácticamente a ejecutar funciones. Por ejemplo, en relación con la máquina de estados, la función principal tiene esta estructura:

```
switch (stateMachine::subState)
{
case stateMachine::Ready:
       requestSensorUpdates();
       break;
case stateMachine::PreAcceleration:
       static bool preAccelStarted = false;
       if (preAccelStarted == false) {
```

```
                    stateMachine::preAccelerationTimeout = 0;
                    sendCANorder_motors(CANmsg_id_set_motor_speed,
stateMachine::settings::maxSpeed);
                    preAccelStarted = true;
             }
             requestSensorUpdates();
             break;
      case stateMachine::Acceleration:
             requestSensorUpdates();
             break;
      case stateMachine::Braking:
             if (brake_order_counter < 3) {
                    sendCANorder_motors(CANmsg_id_stop_motors,
CANmsg_stop_motors_payload);
             }
             sendCANorder_brakes(CANmsg_id_neumaticBrakes_action,
CANmsg_payload_neumaticBrakes_activate);
             sendCANorder_brakes(CANmsg_id_electricBrakes_action,
CANmsg_payload_electricBrakes_activate);
             requestSensorUpdates();
             break;
      case stateMachine::Test:
             requestSensorUpdates();
             break;
      default:
             break;
      }
```

Si bien no se incluye el código por simplicidad, dentro de cada *case* estaría el código específico que se ejecuta en cada estado de la máquina superior. La actualización de las transiciones se ejecuta a posteriori.

3.2.3.2. Placa de motores

El código encargado de ejecutarse en la placa de motores sigue una estructura similar al anterior:

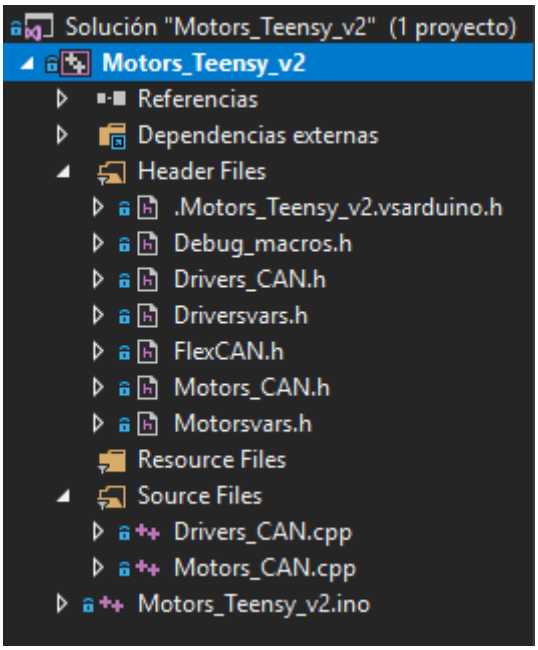

Imagen 15, estructura del código fuente de la placa de motores.

Los headers cumplen exactamente la misma tarea que en el caso anterior, y los dos archivos de código fuente interaccionan con los dos puertos CAN existentes en esta placa: el que se

comunica con el resto de placas (*Motors_CAN.cpp*), y el que se comunica con los drivers de los motores (*Drivers_CAN.cpp*)

3.2.3.2.1. Comunicación con los drivers

La estructura de la comunicación con los drivers sigue una filosofía similar, a nivel de programación, que la comunicación con el resto de placas. Obviamente, el protocolo es diferente: el protocolo de comunicación con el driver está establecido en un documento del fabricante. El protocolo se resume en la siguiente imagen:

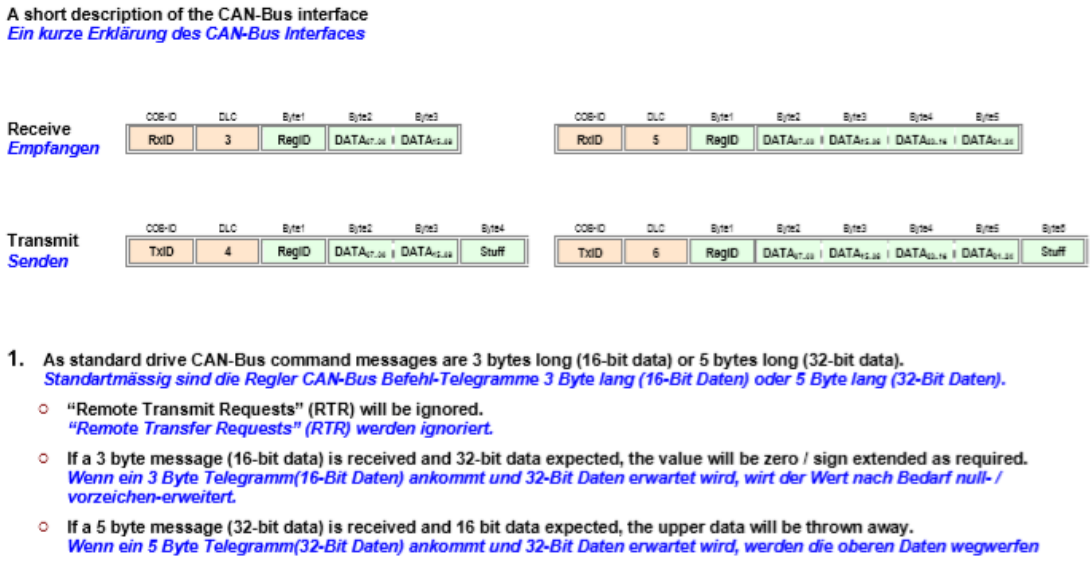

Ilustración 7, estructura del protocolo CAN de comunicación con los drivers

La ilustración está desde el punto de vista del receptor. Básicamente, a la hora de transmitir, hay que mandar mensajes de 4 o 6 byte al ID de recepción del driver, con el registro que se desea escribir en el primer byte de la trama, a continuación, los datos que se desea escribir a ese registro (ya sean 2 o 4 bytes), y el último byte, un byte vacío. Si se desea solicitar el valor de una variable, se escribe en el primer byte el registro 0x3D, en el segundo byte se escribe el registro que se desea leer, y en el tercer byte, se escribe un valor distinto de 0 si se desea que el driver envíe el valor periódicamente, con un intervalo en milisegundos igual al valor del tercer byte del mensaje enviado.

A la hora de recibir datos desde el driver, se reciben mensajes de 3 o 5 bytes (esta diferencia permite saber si el mensaje se ha originado desde un driver, o si es una orden desde algún otro participante de la red hacia un driver), siendo el primer byte, al igual que antes, el registro al que se corresponden los datos, los siguientes 2 o 4 bytes, los valores del registro.

Debido a la necesidad de integrar los valores de velocidad desde los drivers, para obtener la mayor precisión posible, el retardo tiene que ser mínimo. Además, en caso de que algún parámetro de los motores se salga de los valores nominales, es mejor saberlo cuanto antes para evitar daños al motor. Por ello, todos los mensajes provenientes desde los drivers se leen mediante interrupciones.

Por ejemplo, la parte del código que se ejecuta inmediatamente al recibir un mensaje de velocidad:

```
if (frame.id == FRONT_DRIVER_TX_ID) {
```

```
            if (frame.buf[0] == CANmsg_id_driver_filtered_speed) {

                  driversVars::front_motor_speed_rts = true;
                  driversVars::front_motor_speed_adc = frame.buf[1] |
frame.buf[2] << 8;
                  driversVars::front_motor_speed_rpm =
(float)(driversVars::front_motor_speed_adc * driversVars::max_driver_speed_rpm /
32767.0);
                  update_front_driver_encoder_distance();
#if DRIVERS_CAN_INPUT_DEBUG
                  Serial.print("Recibida velocidad del motor frontal: ");
                  Serial.println(driversVars::front_motor_speed_rpm);
#endif
            }
```

En este caso, se llama a la función `update_front_driver_encoder_distance();` que es la encargada de integrar la velocidad y actualizar el valor de la distancia. Esta función es la siguiente:

```
void DriversCAN::update_front_driver_encoder_distance() {

      if (!firstSpeedMsgRecvFrontDriver) {
            driversVars::front_motor_encoder_distance = 0;
            firstSpeedMsgRecvFrontDriver = true;
            lastSpeedMsgFrontDriver = 0;
      }
      else {
#if INTEGRATION_DEBUG
            Serial.println("Interrupción para integrar la velocidad delantera
recibida");
            Serial.print("deltaT : ");
            Serial.print(lastSpeedMsgFrontDriver);
#endif
            uint16_t deltaT = lastSpeedMsgFrontDriver;
            lastSpeedMsgFrontDriver = 0;
            driversVars::front_motor_encoder_distance =
driversVars::front_motor_encoder_distance +
rotational2linearSpeed(driversVars::front_motor_speed_rpm)*(deltaT / 1000000.0);
#if INTEGRATION_DEBUG
            Serial.print("  New distance:");
            Serial.println(driversVars::front_motor_encoder_distance);
#endif
      }
}
```

Siendo exactamente igual para el motor trasero. Ambas distancias deberían coincidir en un margen muy estrecho, y, de hecho, un criterio de fallo es una distancia medida divergiendo mucho de otra, puesto que implica que o una rueda está deslizando, o se ha roto el vínculo mecánico entre un motor y una rueda, por lo que el motor gira libre.

Para solicitar datos a los drivers, existen también funciones independientes que encapsulan el mensaje, y lo mandan automáticamente a los dos drivers. La función para mandar un mensaje a los dos drivers tiene un pequeño retardo: si se envía el mismo mensaje a los dos drivers a la vez, los dos drivers responderán en un intervalo de tiempo similar. Esto implica que pueden saltar dos interrupciones de recepción a la vez, lo que puede producir resultados inesperados. Por ello, existe un pequeño retardo de 10 microsegundos entre 2 envíos consecutivos.

```
void DriversCAN::voltageRequest() {
      CAN_message_t msg;
```

```
        msg.flags.extended = 0;
        msg.flags.remote = 0;
        msg.len = 3;
        msg.buf[0] = 0x3D;   //READ
        msg.buf[1] = 0xEB;   //RegID
        msg.buf[2] = 0x00;

        sendMsgToBothDrivers(msg);
}
```

3.2.3.2.2. Comunicación con el resto de placas

Este apartado muestra un ejemplo de la programación empleada para la comunicación de los esclavos. En primer lugar, tenemos el envío de información, que es básicamente enviar un paquete con la información tal y como se describe en el protocolo. Por ejemplo, para enviar las distancias medidas por la integración de los encoders:

```
void MotorsCAN::sendEncoderDistances(int16_t frontEncoderDist, int16_t
rearEncoderDist) {
        CAN_message_t msg;
        msg.flags.remote = 0;
        msg.flags.extended = 0;
        msg.flags.overrun = 0;
        msg.flags.reserved = 0;
        msg.id = encodeCANid(CAN_BOARDID_MASTER, CANmsg_id_encoder_distance, 0);

        msg.buf[0] = (rearEncoderDist & 0x00FF);
        msg.buf[1] = (rearEncoderDist & 0xFF00) >> 8;
        msg.buf[2] = (frontEncoderDist & 0x00FF);
        msg.buf[3] = (frontEncoderDist & 0xFF00) >> 8;

#if MOTORS_CAN_DEBUG
        Serial.print("Enviando distancia de los drivers por CAN a MASTER: ");
        printCANmsg(msg);
        Serial.println(Can0.write(msg));
#else
        Can0.write(msg);
#endif
}
```

La primera sección crea el mensaje y llena todas las flags necesarias para su correcto envío. La función `encodeCANid();` convierte el ID de una placa, el ID de un mensaje, y si es un request o no, en un ID can válido (básicamente, concatenando los tres elementos). A continuación, se llena el buffer del mensaje con los datos, y se envía.

Para la recepción de datos, está la función `MotorsCAN::decodeCANmsg(CAN_message_t &frame)` que es similar a la empleada en la placa máster, pero en este caso también tiene que responder a requests de datos empleando el protocolo híbrido interrupción/polling. Por ejemplo, cuando se recibe un mensaje que no es crítico:

```
            else if (msg_id.data_id == CANmsg_id_frontMotor_temp) {
                if (msg_id.request == 1) {
#if MOTORS_CAN_INPUT_DEBUG
                    Serial.println("Front motor temperature request");
#endif
                    driversVars::front_motor_temperature_request = true;
                }
            }
```

En este caso, se ha solicitado la temperatura del motor delantero. Para enviarla, lo que hace la función es activar la flag `driversVars::front_motor_temperature_request`. Esta flag será leída en el bucle principal en el siguiente bucle de ejecución, y en ese momento, será cuando el programa responda al mensaje. El retardo máximo será el tiempo de ejecución de un bucle, que no debería superar 1ms.

Sin embargo, cuando se recibe un mensaje crítico, como el de parar los motores:

```
	else if (msg_id.data_id == CANmsg_id_stop_motors) {

#if MOTORS_CAN_INPUT_DEBUG
		Serial.println("STOP MOTORS");
#endif
		DriversCAN::stopMotors();
	}
```

En este caso, una vez se recibe el mensaje se ejecuta directamente la función dentro de la interrupción que envía el mensaje de parada a los dos drivers. Esto implica que existe un retardo mínimo (del orden de microsegundos) entre que se recibe el mensaje de parada y se envía a los drivers la orden de parada.

Esta filosofía de marcar con flags el envío de variables no críticas y realizar acciones críticas dentro de la interrupción se exporta al resto de placas en exactamente la misma mecánica.

3.2.3.2.3. Bucle principal

El bucle principal del código es por lo tanto bastante sencillo, puesto que lo único que hace es, después de programar los drivers al inicio del programa, responder a órdenes que va recibiendo, de esta forma:

```
	if (driversVars::encoder_distance_request) {
		driversVars::encoder_distance_request = false;

	motorsCAN.sendEncoderDistances(driversVars::front_motor_encoder_distance,
driversVars::rear_motor_encoder_distance);
	}
```

Si la flag está activada, se ejecuta la función que envía el mensaje de respuesta, y la flag se limpia, para poder esperar el siguiente mensaje de solicitud.

4. Prueba del sistema

Las pruebas del sistema fueron varias y distribuidas a lo largo del tiempo, si bien la mayor parte se concentró durante la semana de la competición, presentarse sin haber pasado tests previos era bastante optimista.

Las pruebas se dividieron en pruebas de hardware y prueba de software. En las pruebas de hardware, el objetivo era verificar que las PCBs diseñadas funcionaban correctamente, que los cables realizaban contacto, que los conectores funcionaban, etcétera. En definitiva, probar a nivel de hardware todas las conexiones.

A nivel de software, las pruebas eran básicamente comprobar que se cumplían los objetivos establecidos, primero a nivel individual de cada placa, y posteriormente en conjunto como sistema, y realizar pruebas de estrés al sistema para comprobar con qué velocidad era capaz de escupir datos.

4.1. Pruebas de hardware

La mayoría de las pruebas de hardware eran realizadas cuando se recibía por primera vez el elemento, para comprobar que venía bien de fábrica, cuando se ensamblaba o soldaban los componentes por primera vez, o bien cuando se encontraba un fallo o un comportamiento inesperado del sistema. Las herramientas básicas para la realización de las pruebas eran las herramientas típicas de un laboratorio de electrónica: multímetro, osciloscopio con sus sondas, y soldador de estaño para poder soldar/desoldar elementos.

4.1.1. Pruebas comunes

Las pruebas comunes son las pruebas que se ejecutaban en todas las placas una vez recibidas y ensambladas.

PRUEBA	RESULTADO
INSPECCIÓN VISUAL	Correcto. Ninguna placa fue recibida con algún defecto de fabricación apreciable.
COMPATIBILIDAD MECÁNICA CON LOS COMPONENTES SMD	Correcto. Los componentes se soldaron sin problemas por un miembro del equipo.
ALIMENTACIÓN	Correcto. El transformador DC/DC funcionaba correctamente con una entrada de 24V
COMUNICACIONES CAN	Correcto. El hardware de comunicaciones CAN es exactamente el mismo en todas las placas, y en todas funcionaba perfectamente

4.1.2. Placa de navegación

PRUEBA	RESULTADO
COMPATIBILIDAD MECÁNICA CON LOS COMPONENTES SMD	Correcto. Los componentes se soldaron sin problemas por un miembro del equipo.
ALIMENTACIÓN	Correcto. El transformador DC/DC funcionaba correctamente con una entrada de 24V
SENSOR DE PRESIÓN	Correcto: La lectura del sensor de presión incorporado daba un resultado válido

LECTORES DE CINTAS	Versión 1: Incorrecto En la primera revisión de la placa y por un error en el esquemático, la salida de los lectores de cintas no pasaba por los comparadores. Este elemento era necesario puesto que la salida de los lectores es de 12V, y debía ser adaptada a los 3.3V de entrada al microprocesador. Versión 2: El fallo fue solucionado y la placa era capaz de detectar cintas sin problemas
COMUNICACIÓN CAN	Correcto: la placa comunicaba correctamente con el bus CAN
COMUNICACIÓN SERIE CON LA IMU	Resultados varios. La placa comunicaba correctamente con la IMU principal que teníamos, la Vectornav VN1000, pero era incapaz de comunicarse con otra IMU de otro fabricante, bastante menos fiable y barata. La solución fue descartar la IMU barata y tomar únicamente las medidas de la IMU principal

4.1.3. Placa de energía

La placa de energía pasó por varias iteraciones. En la primera iteración, el objetivo era tomar 3 mediciones de voltaje únicamente, englobando cada medida de voltaje en total 10 baterías. La idea era conectar el comienzo de una tira de 10 baterías a la tierra de la placa empleando IGBTs, y a continuación medir con un divisor resistivo de muy alta impedancia la tensión en el otro extremo de la tira de las baterías. Sin embargo, durante las pruebas, este enfoque demostró ser impreciso y peligroso. Debido a un fallo que no se logró identificar, la primera medición (la tira que iba de 0 a 240V) se realizaba correctamente, pero los valores leídos en las tiras segunda y tercera (de 240V a 480V y de 480V a 720V) eran muy inferiores al nivel de ruido, lo que impedía realizar una medición precisa. Además, existía el peligro de conectar un circuito a voltajes tan altos en la parte de aviónica, que en teoría iba completamente aislada del circuito de alto voltaje. La segunda revisión es la descrita en este documento.

4.1.3.1. Módulos de lectura individual

Los módulos de lectura individual eran los encargados de leer el voltaje de cada uno de los packs de baterías, como el de la Imagen 8. Se necesitaban en total 60 de estos componentes, uno para cada batería, por lo que se fabricaron 65, asumiendo que existía la posibilidad de generar problemas con los componentes a la hora de soldarlos a mano.

PRUEBA	RESULTADO
LECTURA DE LA TENSIÓN DE LA BATERÍA	Correcto. Los componentes fueron soldados a lo largo de 2 días por 3 miembros del equipo. De las 65 placas soldadas, 12 mostraron fallos de distinto tipo, de las que 8 pudieron ser arregladas. En total 61 placas pudieron funcionar.
COMUNICACIONES I2C	Correcto. El ADC funcionaba en la dirección especificada y a la velocidad especificada. Esta prueba también comprobaba el funcionamiento del optoacoplador I2C

4.1.3.2. Módulo multiplexor

Los módulos multiplexores se encargaban de distribuir el bus I2C en las 6 ramas diferentes de medida, como el de la Imagen 7.

PRUEBA	RESULTADO
MULTIPLEXADO I2C	Correcto por separado. Empleando varias placas Arduino conectadas a los distintos canales I2C del multiplexor, el funcionamiento era correcto.

I2C DIFERENCIAL	Correcto. El funcionamiento del bus I2C diferencial era correcto y como las especificaciones: era completamente transparente al usuario (no había que realizar ningún tipo de configuración por parte del usuario)

4.1.3.3. Integración entre los dos sistemas

Las pruebas de integración consistían en medir, a través del multiplexor, 2 ramas diferentes de módulos de lectura con las mismas direcciones. Era, básicamente, medir el sistema a escala. Sin embargo, no salió correctamente.

Debido a incompatibilidades entre los niveles lógicos del multiplexor, y del optoacoplador I2C, era imposible comunicarse con los módulos si estaban conectados al optoacoplador. Por separado, ambos componentes funcionaban a la perfección, pero una vez se juntaban, era imposible comunicarse con los módulos.

Este problema se descubrió 10 días antes de la competición, y el margen de maniobra para arreglarlo era bastante reducido. La solución al multiplexor no funcionando fue realizar multiplexado *por software*. El microcontrolador de la *Teensy* tiene 4 puertos diferentes I2C, varios de los cuales se pueden acceder en sets de pins distintos. Esto quiere decir que se puede configurar un bus I2C para leer en dos pines, y al vuelo, cambiar estos dos pines, dando como resultado una multiplexación a nivel de software. El prototipo de la placa fue ensamblado con los materiales disponibles:

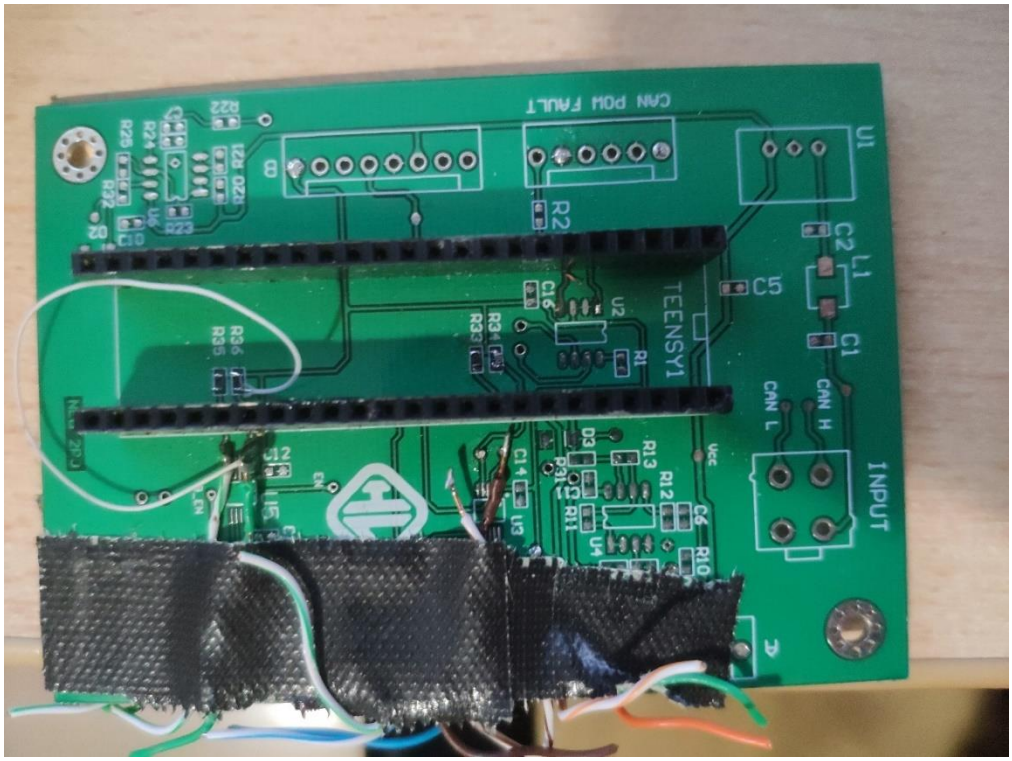

Imagen 16, parte superior de la placa adaptada in situ para probar la estrategia de medición de multiplexado por software.

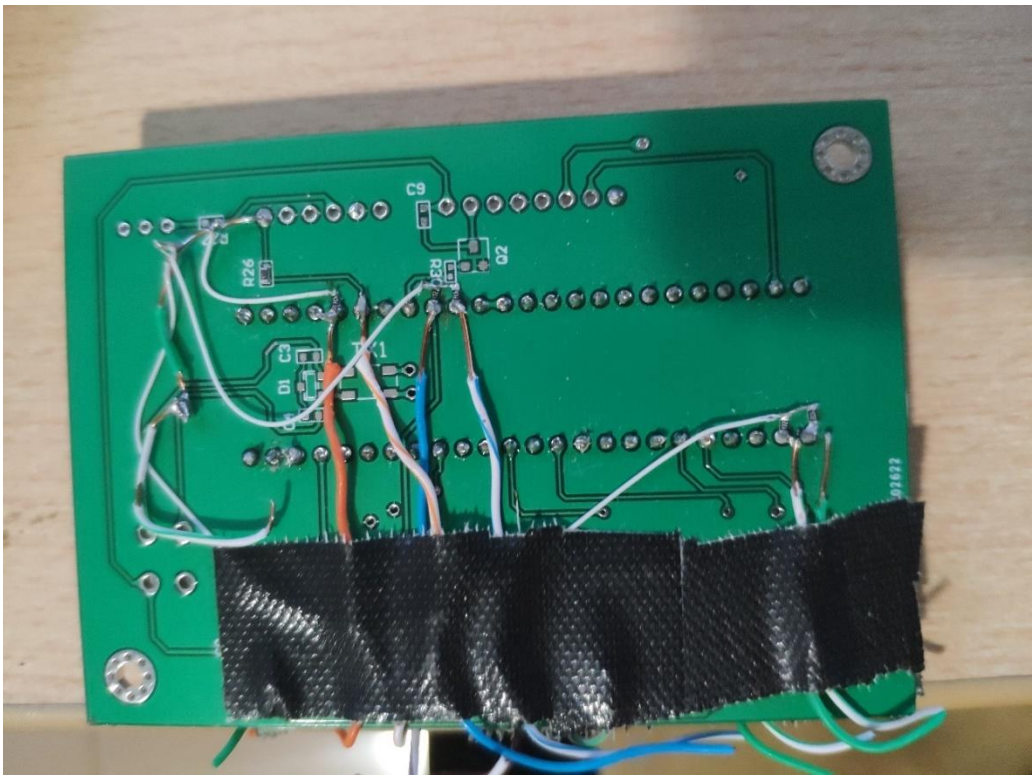

Imagen 17, parte inferior de la misma placa. Se puede apreciar en las dos conexiones de la esquina inferior izquierda las resistencias de terminación del bus I2C, conectadas en línea con el cable.

Una vez se pudo comprobar con este prototipo que la técnica de multiplexado era viable, se diseñó una placa rápida que se mandó fabricar en Silicon Valley, yendo un par de miembros del equipo desde la ciudad de Merced, donde se encontraba el taller en el que una empresa de Denia permitía trabajar al equipo, hasta San Francisco, a recoger las placas.

Esta fue, efectivamente, la tercera versión de la placa de energía, y la primera en funcionar correctamente.

PRUEBA	RESULTADO
LECTURA DE 30 MÓDULOS ADC	Correcto: La placa era capaz de leer 30 voltajes distintos con 30 baterías conectadas en serie, sin transmitir voltajes peligrosos a la placa.

4.1.4. Placa de frenos

La placa de frenos pese a su complejidad (debía controlar 8 actuadores eléctricos capaces de consumir hasta 20 amperios de pico, por lo que debía emplear 4 drivers de alta potencia para los actuadores) no supuso problemas graves, más allá de la complejidad de soldar a mano los 4 drivers y comprobar que las conexiones estaban bien realizadas.

PRUEBA	RESULTADO
ACTUACIÓN DE LOS FRENOS ELÉCTRICOS	Correcto: La placa era capaz de activar los 4 actuadores eléctricos a la vez
ACTUACIÓN DE LOS FRENOS NEUMÁTICOS	Correcto: La actuación de los frenos neumáticos era sencilla ya que no requería de excesiva potencia.
LECTURA DE LOS SENSORES DE PRESIÓN DEL SISTEMA NEUMÁTICO	Correcto: la placa era capaz de leer todos los sensores del sistema neumático.

4.1.5. Placa de motores

La placa de motores era relativamente sencilla: era, fundamentalmente, 2 buses CAN, y un MOSFET para controlar la alimentación de la bomba de refrigerante.

PRUEBA	RESULTADO
ACTUACIÓN DE LA BOMBA DE REFRIGERANTE	Correcto: La placa era capaz de activar y controlar la velocidad de la bomba de refrigerante a través de una PWM a la entrada de un MOSFET
CONTROL DE LOS MOTORES POR CAN	Correcto: La comunicación con los drivers funcionaba a la perfección en los dos sentidos.

4.1.6. Placa maestra

La placa maestra tampoco tenía mucha complejidad. A parte del bus CAN, únicamente necesitaba un controlador Ethernet externo, conectado por bus SPI al microcontrolador. El controlador Ethernet empleado estaba recomendado por varias fuentes en internet por su elevada compatibilidad y demostró funcionar sin problemas.

PRUEBA	RESULTADO
COMUNICACIONES ETHERNET	Correcto: Las comunicaciones ethernet eran capaces de alcanzar tasas de 100kilobits por segundo, más de lo necesario para nuestras necesidades
SENSORES DE A BORDO	Correcto: la placa era capaz de leer la temperatura y presión ambiental.

4.2. Pruebas de software

Una vez comprobado que todo el hardware funcionaba correctamente, el siguiente paso era comprobar el software. Durante la programación se realizó gran parte del código empleando la metodología *Test Driven Programming*. La idea detrás de esta filosofía es hacer el código mínimo para pasar un test, y los test van aumentando en complejidad. Por ejemplo, el primer test es "Enviar un paquete por CAN". El siguiente test es "Enviar un paquete CAN con la trama adecuada". El siguiente es "Enviar periódicamente un paquete CAN con la trama adecuada", etcétera. La idea detrás de esta metodología era simplificar la programación al máximo posible.

La prueba final, obviamente, era el funcionamiento correcto del sistema, y era la combinación de todos los test pasados anteriormente. Sin embargo, a medida que se acercaba la fecha de la competición, la metodología se dejó de lado por falta de tiempo, y fue más *ad-hoc*, programar al lado del *Pod* hasta que la funcionalidad estuviera implementada y pasar pruebas básicas de funcionamiento.

El primer paso a nivel de software era comprobar el funcionamiento de cada placa individual: cada placa debía de ser capaz de leer los sensores de los que disponía e interactuar con los actuadores.

4.2.1. Pruebas comunes

La prueba común a todas las placas era la implementación del protocolo de comunicaciones por CAN. Para esta prueba, se conectaba la placa a una interfaz CAN para ordenador, y desde el ordenador se mandaban todas las tramas que podía recibir esa placa en concreto. Empleando los debuggers integrados en el IDE se comprobaba el funcionamiento correcto del software: interpretaba correctamente la trama y ejecutaba lo que se le ordenaba, ya fuera mandar datos, o realizar alguna acción.

PRUEBA	RESULTADO	REQUISITOS
IMPLEMENTACIÓN DEL PROTOCOLO DE COMUNICACIONES POR CAN	Correcto: el protocolo de comunicaciones se implementó correctamente en todas las placas, empleando una estructura de software similar.	NAV03, MOT02, BRK02, ENE02

4.2.2. Placa de navegación

PRUEBA	RESULTADO	REQUISITO
ACTUALIZACIÓN DE LOS PARÁMETROS DE ACELERACIONES EMPLEANDO LA ENTRADA DE LA IMU	Correcto: El software era capaz de interpretar la información recibida desde la IMU por puerto serie y extraer los datos de las aceleraciones, que empleaba para actualizar variables en memoria para luego ser enviada por CAN.	NAV02
CONTEO DE CINTAS	Correcto: El software leía con gran precisión el número de cintas que pasaban por los lectores. El sistema de hecho fue empelado en una bancada de pruebas para el motor, pegando cinta en un disco giratorio para *Pod*er medir la velocidad de giro y la aceleración del motor con precisión.	NAV01

4.2.3. Placa de energía

PRUEBA	RESULTADO	REQUISITO
MEDICIÓN DE LAS BATERÍAS	Correcto: El software era capaz de leer los 30 valores de los 30 módulos diferentes, cambiando al vuelo el puerto I2C, en un tiempo de ciclo de aproximadamente 100ms, suficiente teniendo en cuenta la lenta dinámica de la química de las baterías.	ENE02
MEDICIÓN DE TEMPERATURA	Correcto: empleando 5 sensores DS18B20 la placa era capaz de tomar 5 medidas de temperatura del conjunto de las baterías. Sin embargo, este sensor era mucho más lento de lo esperado a la hora de leer los datos, por lo que se tuvo que emplear una técnica de lectura asíncrona para *Pod*er leer los datos de los 5 sensores en un tiempo razonable.	ENE02
ACTUACIÓN DE LOS CONTACTORES	Correcto: La placa abría y cerraba los contactores bajo petición manual o desde el bus de comunicaciones.	ENE01

4.2.4. Placa de motores

PRUEBA	RESULTADO	REQUISITO
PROGRAMACIÓN INICIAL DE LOS DRIVERS	Correcto: El software era capaz de programar los ajustes importantes en los drivers al arrancar. Si bien los drivers disponían de una EEPROM interna en la que guardar los ajustes, por precaución, se programaban antes de cada ejecución. Entre estos ajustes por ejemplo estaba el de mandar los datos de velocidad cada 1ms, limitar la velocidad máxima del motor, etcétera.	MOT1

INTEGRACIÓN DE LOS DATOS DE VELOCIDAD	Correcto: tras la programación de los drivers, el software empezaba a recibir tramas desde los drivers con los valores de velocidad, en intervalos de 1ms. El software integraba los valores recibidos para dar una medida muy precisa de la distancia recorrida por el *Pod*, asumiendo que no existía deslizamiento entre las ruedas y el suelo.	MOT03
ENCENDIDO/ APAGADO DE LOS MOTORES	Correcto: el software era perfectamente capaz de encender y apagar los motores, y de controlar la velocidad	MOT03
LECTURA DE SENSORES	Correcto: el software era capaz de pedir valores de los distintos sensores de los que disponía el driver (Voltaje, tensión, temperatura…) e interpretar los resultados de vuelta.	MOT03
CONTROL DE LA BOMBA DE REFRIGERANTE	N/A. Debido a la complejidad del montaje, se decidió descartar la bomba de refrigerante, dejando el circuito de refrigerante como una masa térmica sin refrigeración forzada.	-

4.2.5. Placa de frenos

PRUEBA	RESULTADO	REQUISITO
ACTUACIÓN DE LOS FRENOS	Correcto: La placa era capaz de actuar los frenos en varios modos: en modo normal, la placa actuaba todos los frenos a la vez (eléctricos y neumáticos, los 8 actuadores a la vez). Sin embargo, a petición de la placa maestra, era capaz de activar distintos actuadores de forma independiente, para comprobar su funcionamiento individual, o en grupos, para comprobar el funcionamiento del sistema de frenos. Estas peticiones se recibían desde la placa maestra.	BRK01
LECTURA DE LOS VALORES DE PRESIÓN	Correcto: la placa leía periódicamente los valores de presión del circuito neumático y actualizaba las variables correspondientes	BRK02

4.2.6. Placa maestra

PRUEBA	RESULTADO	REQUISITO
COMUNICACIONES ETHERNET	Correcto: Una de las primeras pruebas realizadas fue la comunicación estación base-placa maestra. La placa genera datos aleatorios con los que llenaba un *JSON* que posteriormente enviaba a la estación base. Esto nos permitía comprobar toda la línea intermedia entre placa maestra->estación base. Después de varias iteraciones y optimizaciones de código, se consiguió que la placa pudiera responder a peticiones a velocidades superiores a 1000Hz, que, tomando un tamaño de paquete medio de 1000 bytes, implicaba una velocidad del bus de aproximadamente 1Megabyte por segundo. Las comunicaciones en sentido opuesto (estación base->placa maestra), también funcionaron sin ningún problema después de depurar ambos softwares. El	DATA01, DATA03

	principal problema encontrado eran fallos de formato en el *JSON*	
COMUNICACIONES CAN	Correcto: la funcionalidad completa del protocolo CAN se implementó con éxito en la placa maestra. La placa era capaz de leer e interpretar hasta 300.000 paquetes por segundo (obviamente en el bus final, la cantidad sería mucho menor, pero se realizaban pruebas de estrés para comprobar la fiabilidad del sistema)	DATA02, DATA04
MÁQUINA DE ESTADOS	Parcialmente correcto. Aislada la placa maestra y mandando las condiciones para las transiciones por separado, la máquina de estados funcionaba perfectamente.	SYS02

4.2.7 Pruebas de integración

Dada la naturaleza centralizada del sistema, en el que las placas esclavas no realizan comunicaciones entre sí, y todas las comunicaciones se realizan entre esclava-maestra, realizar las pruebas de integración se simplificaba notablemente. Para probar el funcionamiento correcto de una placa, la transmisión de datos, etcétera, se conectaban únicamente esa placa, la placa maestra, y a continuación, la telemetría. En la placa maestra se ejecutaba una versión reducida del software que limitaba las transiciones de estados a aquellas que fueran directamente afectadas por la placa esclava de pruebas.

Una de las primeras pruebas ejecutadas así fue el control manual de la bomba de refrigerante.

Imagen 18, bomba de refrigerante con el circuito de refrigeración.

Conectando la placa de motores a la bomba de refrigerante, y al sistema, la primera variable que se pudo controlar en remoto fue la velocidad de la bomba de refrigerante. A partir de ahí, se fue añadiendo funcionalidad.

El siguiente paso con la placa de motores fue controlar el encendido, apagado y control de la velocidad de los motores, empleando telemetría. El test fue pasado con éxito y se empleó el sistema para controlar la velocidad de los motores en diversas pruebas que fueron realizadas

con ayuda de un servomotor actuando como freno del DIE (Departamento de ingeniería eléctrica)

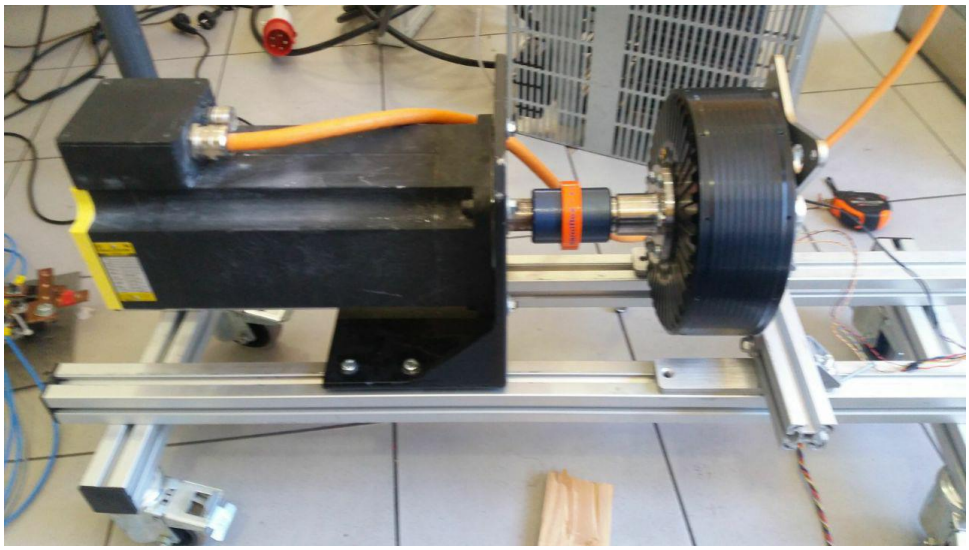

Imagen 19, bancada de pruebas con el servodriver actuando como freno a la izquierda, y nuestro motor a la derecha. El motor de la derecha tiene aproximadamente 5 veces más potencia.

El sistema de frenos, el segundo sistema que *Pod*ía ser controlado por telemetría, también fue probado con éxito. Estos tests, sin embargo, al requerir presión en el sistema neumático y que el sistema de frenos estuviera ensamblado, se realizaron bastante tarde: días antes de la competición.

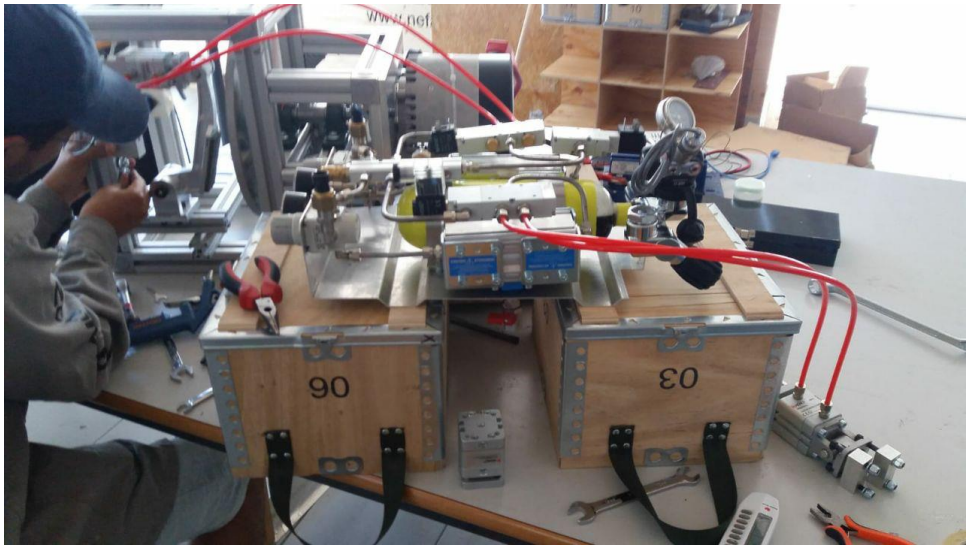

Imagen 20, sistema de neumática, con un actuador en la parte inferior derecha

Los tests de integración final sin embargo tuvieron que esperar a la competición. *Pod*er conectar todas las placas entre si implicaba tener todo el cableado hecho y crimpado a las distancias correctas, ya que los conectores estaban en cantidades justas, y comprar de más era inviable. Sin embargo, durante la competición, un fallo técnico en las baterías provocó la descalificación del equipo, a 2 días del final de la competición. El fallo técnico sucedió cuando se estaban ensamblando las baterías para el test de integración completo: los ingenieros de SpaceX iban a comprobar el funcionamiento del sistema de telemetría y sensórica en estacionario para dar el visto bueno a introducir el vehículo en una pista de prueba y comprobar el funcionamiento del

sistema general. A partir de aquí, y tras una difícil decisión el equipo decidió dejar de intentar superar los tests que todavía estábamos habilitados, puesto que el impacto en la moral del equipo había sido muy elevado.

Este problema técnico dejó sin probar el sistema de aviónica en su conjunto. Sin embargo, a pesar de ello, se habían conseguido superar varias pruebas:

PRUEBA	RESULTADO	REQUISITO
CONTROL DE ACTUADORES POR TELEMETRÍA	Correcto: Prácticamente todos los actuadores disponibles se *Podí*an controlar desde la interfaz gráfica en un ordenador en la estación base. Debido a los cambios de diseño a última hora de la placa de energía, el único actuador que no fue probado de forma remota (fue probado in situ empleando software en una placa únicamente) fueron los contactores de las baterías. Sin embargo, su actuación no suponía ningún problema y no era de esperar que existieran problemas	N/A
CONTROL DEL MOTOR POR TELEMETRÍA	Correcto: se *Podí*an establecer límites de velocidad, curvas de aceleración, y demás parámetros al motor, que los seguía sin problemas	
RECEPCIÓN DE DATOS POR TELEMETRÍA	Correcto: gran parte de los sensores que se consiguieron integrar en el sistema final mostraban sus valores en la interfaz gráfica.	
MÁQUINA DE ESTADOS: TRANSICIONES DE OPERACIÓN NORMAL	Correcto: Se realizaron simulacros de una carrera en la que se inicializaban los motores, se pasaba una cantidad de cintas por delante de los sensores para indicar que se había superado cierta distancia, y la máquina de estados se comportaba correctamente. Este test debería haberse pasado al día siguiente en presencia de los ingenieros de SpaceX	
EMERGENCIA: PARADA AUTOMÁTICA EN CASO DE DESCONEXIÓN DE LA ESTACIÓN BASE	Correcto: En caso de perder la comunicación con la estación de tierra, el sistema mandaba orden de paro a los motores y de frenada automáticamente.	

Sin embargo, también quedaron gran cantidad de tests sin pasar. La gran mayoría de estas pruebas requerían tener todo el sistema ya integrado y realizar pruebas sobre el comportamiento de éste bajo circunstancias adversas: sobre todo, situaciones de emergencia.

PRUEBA	RESULTADO	REQUISITO
EMERGENCIA: PARADA EN CASO DE PÉRDIDA DE UNA PLACA	El código estaba implementado a nivel superficial, a falta de comprobaciones y optimización, pero fue imposible probar exactamente el comportamiento del sistema.	N/A

EMERGENCIA: COMPORTAMIENTO DE LA MÁQUINA DE ESTADOS ANTE PÉRDIDA DE DETERMINADOS SENSORES	N/A	N/A
EMERGENCIA: PRUEBA DE ESTRÉS	N/A	N/A
EMERGENCIA: REESTABLECER LA COMUNICACIÓN	N/A	N/A
EMERGENCIA: APAGADO SEGURO	N/A	N/A
MÁQUINA DE ESTADOS: FUNCIONAMIENTO COMPLETO	N/A	N/A
BUS DE COMUNICACIONES: FUNCIONAMIENTO COMPLETO	N/A	N/A

5. Conclusión

A pesar de no haber cumplido el objetivo final, la prueba (y éxito de funcionamiento) del sistema completo, considero que el proyecto ha sido un éxito. Se ha conseguido diseñar, en un periodo de tiempo muy limitado, un sistema electrónico, tanto a nivel de software como de hardware, capaz de competir a nivel mundial contra universidades de todo el mundo. Durante la semana de la competición, la mayoría de los equipos preguntaban con admiración cómo realizábamos el sistema de control.

La mayoría de los equipos, por ejemplo, se sorprendía por el hecho de emplear un sistema distribuido, pero cuando veían los resultados y las capacidades de nuestro sistema, entendían las ventajas. La mayoría de los equipos se limitaban a un sistema industrial, con sensores industriales, tipo PLC, para realizar el control de sus prototipos. Otro punto que destacaba era el BMS: prácticamente todos los equipos llevaban uno comercial (en concreto, la mayoría del fabricante ORION BMS). Únicamente 2 equipos llevaban un BMS personalizado: Hyperloop UPV y Badgerloop. Y de esos 2, únicamente el nuestro era capaz de monitorizar hasta 700V de baterías (y expandible mucho más).

Si bien es cierto que no era el óptimo (los costes de desarrollo y prototipado superaron los costes de un BMS comercial), el objetivo al final no era la optimización: era aprender, y desarrollar tecnología.

A nivel de software, el software embebido no tuvo mucho protagonismo, y es normal: la tarea principal del software embebido es funcionar bien, sin dar problemas, y sin llamar la atención. Sólo se presta atención al software cuando no funciona, y se veían a equipos con varias personas en ordenadores alrededor de su prototipo. Sin embargo, a nivel personal, estoy orgulloso del trabajo del equipo de software. Los 3 miembros del equipo programamos en apenas meses un sistema que bien podría pasar por comercial. No haberlo probado en su conjunto deja una espina clavada, pero es fácil ignorarla cuando compruebas todo lo que se había conseguido hasta entonces: controlar un motor de 100kW y leer todos sus parámetros empleando tu software es un hito importante, como lo es leer y enviar los voltajes de 30 baterías conectadas en serie.

A nivel personal, si bien siempre queda ese sentimiento de “Podría haber hecho más”, se esfuma rápidamente al mirar atrás. Estuvimos a nada de quedar en el top 5, o incluso mejor, y eso en sí ya es un logro. Posicionarnos como equipo y superar a universidades enormes, habiendo tenido

muchos más problemas de logística y financiación, dice mucho del equipo. Me llevo de este año y de esta competición muchas experiencias, profesionales y personales, buenas y malas. Me llevo un equipo, tanto de excelentes profesionales como de geniales compañeros y amigos. Me llevo también un puesto de trabajo que ni me habría imaginado capaz de desempeñar antes de entrar en el equipo. Habremos quedado octavos, pero yo he salido ganando.

Muchas gracias.

6. Referencias

-First Design Package, Hyperloop UPV, 2018[1]

-Safety Package, Hyperloop UPV, 2018[2]

-UNITEK BAMOCAR D3 User manual, UNITEK Industrie Elektronik GmbH

-Teensy 3.6 Datasheets and user manuals

-VisualMicro user guide

-Programming Embedded Systems: With C and GNU Development Tools, Michael Barr y Anthony Massa, O'Reilly

-Arduino language reference

[1] Disponible bajo petición al equipo
[2] Disponible bajo petición al equipo

Anexo 1: Documento con la estructura de todas las tramas CAN empleadas

FRAME ID		PAYLOAD								FORMAT
		MSB							LSB	
Decimal	Hex	Position 7	Position 6	Position 5	Position 4	Position 3	Position 2	Position 1	Position 0	
1	01							% MOTOR SPEED	% MOTOR SPEED	Adimensional
								Integer Y in XXX.Y%	Integer X in XXX.Y%	INT
2	02								STOP MOTORS	Signal
									Full 1s if stop motors order issued	
3	03								START BRAKING	
									11111111	
4	04							Energy Health Check	Energy Health Check	health check => 1 = ERROR, 0 = OK
								XXXXXXXX	XXXXXXXX	Every bit corresponds to a different health check
5	05								Navigation Health Check	
									XXXXXXXX	"1 byte": XXXXXXXX = pressure | temp | angularz | angulary | angularx | accelz | accely | accelx
6	06						Brakes Health Check	Brakes Health Check	Brakes Health Check	
							XXXXXXXX	XXXXXXXX	XXXXXXXX	"3 bytes (only 2 for the moment)": XXXXXXXX = "byte0":temp4| temp3 | temp2| temp1| highpressure | medianpressure| lowpressure2| lowpressure1; "byte1": reed8 | reed7 | reed6 | reed5 | reed4 | reed3 | reed2 | reed1; /*"byte2": 0|0|0|0| current4 | current3 | current2| current1*/
7	07								Motors Health Check	**1 second after receiveing the request, the current health state is sent. If more data is received after that second, the health state is sent again**
									XXXXXXXX	"1 byte": rearspeed | frontspeed| rearspeed |frontspeed |cool_temp | .cool_pressure
11	0B			Acceleration Z	Acceleration Z	Acceleration Y	Acceleration Y	Acceleration X	Acceleration X	m/s^2

				Integer X100	Integer X100	Integer X100	Integer X100	Integer X100	Integer X100	INT
12	0C			Angularz	Angularz	Angulary	Angulary	Angularx	Angularx	º
				Integer X100	Integer X100	Integer X100	Integer X100	Integer X100	Integer X100	INT
13	0D					Distance	Distance	Distance	Distance	m
						Integer X100	Integer X100	Integer X100	Integer X100	INT
14	0E					Stripe_count_reader4	Stripe_count_reader3	Stripe_count_reader2	Stripe_count_reader1	-
						Integer 8bits	Integer 8bits	Integer 8bits	Integer 8bits	INT
15	0F							*Pod* pressure	*Pod* pressure	atm
								Integer x1000	Integer x1000	INT X1000
16	10							Temperature	Temperature	
								Integer X10	Integer X10	
21	15					Coolant temp	Coolant temp	Coolant pressure	Coolant pressure	º, atm
						Integer X10	Integer X10	Integer X1000	Integer X1000	
22	16						Driver_speed_front	Driver_speed_front	Driver_speed_front	
							UNDEFINED	UNDEFINED	UNDEFINED	
23	17						Driver_speed_rear	Driver_speed_rear	Driver_speed_rear	
							UNDEFINED	UNDEFINED	UNDEFINED	
24	18								Coolant_pump	%
									Integer	INT
25	19							Rear_Motor_Temperature	Rear_Motor_Temperature	
								Integer X10	Integer X10	
26	1A							Front_Motor_Temperature	Front_Motor_Temperature	
								Integer X10	Integer X10	
27	1B					Front_motor_encoder_distance	Front_motor_encoder_distance	Rear_motor_encoder_distance	Rear_motor_encoder_distance	
						Integer X10	IntegerX10	IntegerX10	IntegerX10	
28	1C	Front_driver_current	Front_driver_current	Rear_driver_current	Rear_driver_current	Front_driver_voltage	Front_driver_voltage	Rear_driver_voltage	Rear_driver_voltage	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
29	1D								Driver order	
									XXXXXXXX	1. Restart drivers

31	1F	high_pressure	high_pressure	median_pressure	median_pressure	low_pressure2	low_pressure2	low_pressure1	low_pressure1	
		Integer x10	Integer x10	Integer x1000	Integer x1000	Integer x1000	Integer x1000	Integer x1000	Integer x1000	
34	22	Brakes_Temperature 4	Brakes_Temperature 4	Brakes_Temperature 3	Brakes_Temperature 3	Brakes_Temperature2	Brakes_Temperature2	Brakes_Temperature1 MSB	Brakes_Temperature1 LSB	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
35	23								Electric brakes action	
									Integer	0: stopMotors1 1: stopMotors2 2: startMotorsOppositeSense 3: startMotorsOppositeSenseUpsideDown 4: startMotorsSameSense1 5: startMotorsSameSense2 6: startOnlyBrake1 7: startOnlyBrake2 8: startOnlyBrake3 9: startOnlyBrake4 10: stopOnlyBrake1 11: stopOnlyBrake2 12: stopOnlyBrake3 13: stopOnlyBrake4
36	24					Driver4 PWM	Driver3 PWM	Driver2 PWM	Driver1 PWM	
						Integer	Integer	Integer	Integer	
37	25	Driver4 current	Driver4 current	Driver3 current	Driver3 current	Driver2 current	Driver2 current	Driver1 current	Driver1 current	
		Integer x1000	Integer x1000	Integer x1000	Integer x1000	Integer x1000	Integer x1000	Integer x1000	Integer x1000	
38	26								Neumatic brakes action	
									Integer	0: Idle 1: noBrakes 2: bothBrakes 3: oneBrake1 4: oneBrake2 5: purge_temp 6: purge_def
39	27	REED8	REED7	REED6	REED5	REED4	REED3	REED2	REED1	
		Integer	Integer	Integer	Integer	Integer	Integer	Integer	Integer	
42	2A	batt8voltage	batt7voltage	batt6voltage	batt5voltage	batt4voltage	batt3voltage	batt2voltage	batt1voltage	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
43	2B	batt16voltage	batt15voltage	batt14voltage	batt13voltage	batt12voltage	batt11voltage	batt10voltage	batt9voltage	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	

44	2C	batt24voltage	batt23voltage	batt22voltage	batt21voltage	batt20voltage	batt19voltage	batt18voltage	batt17voltage	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
45	2D			batt30voltage	batt29voltage	batt28voltage	batt27voltage	batt26voltage	batt25voltage	
				Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
46	2E	batt38voltage	batt37voltage	batt36voltage	batt35voltage	batt34voltage	batt33voltage	batt32voltage	batt31voltage	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
47	2F	batt46voltage	batt45voltage	batt44voltage	batt43voltage	batt42voltage	batt41voltage	batt40voltage	batt39voltage	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
48	30	batt54voltage	batt53voltage	batt52voltage	batt51voltage	batt50voltage	batt49voltage	batt48voltage	batt47voltage	
		Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
49	31			batt60voltage	batt59voltage	batt58voltage	batt57voltage	batt56voltage	batt55voltage	
				Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	Integer X10	
50	32						frontBattPressure	frontBattCurrent	frontBattTemp	
							integer x100	integer	integer	
51	33						rearBattPressure	rearBattCurrent	rearBattTemp	
							integer x100	integer	integer	
52	34								ENABLE BATT CONTACTORS	
53	35								DISABLE BATT CONTACTORS	
63	3F								KEEPALIVE	
									11111111	

Anexo 2: Mensaje JSON


```
{
  "tripData" : {
    "imuData" : {
      "position" : {
        "x" : 0,
        "y" : 0,
        "z" : 0
      },
      "velocity" : {
        "x" : 0,
        "y" : 0,
        "z" : 0
      },
      "acceleration" : {
        "x" : 0,
        "y" : 0,
        "z" : -1
      },
        "attitude" : {
            "roll" : 1,
            "pitch" : 1,
            "yaw" : 1
        }
    },
    "motorData" : {
      "front_motor_encoder_distance" : 1,
      "rear_motor_encoder_distance" : 1,
      "front_motor_encoder_speed" : 1,
      "rear_motor_encoder_speed" : 1
    },
    "tapereaderData" : {
      "tapereader_frontright" : 1,
      "tapereader_frontleft" : 1,
      "tapereader_rearright" : 1,
      "tapereader_rearleft" : 1
    }
  },
  "sensorData" : {
    "batteries" : {
      "front_pack_temperature" : 1,
      "rear_pack_temperature" : 1,
      "front_pack_current" : 1,
      "rear_pack_current" : 1,
      "front_pack_pressure" : 1,
      "rear_pack_pressure" : 1,
      "front_pack_mean_voltage" : 1,
      "rear_pack_mean_voltage" : 1,
      "front_pack_lowest_voltage" : 1,
      "rear_pack_lowest_voltage" : 1,
      "front_pack_highest_voltage" : 1,
      "rear_pack_highest_voltage" : 1
    },
    "motors" : {
      "front_motor_status" : 1,
      "rear_motor_status" : 1,
      "front_motor_temperature" : 1,
      "rear_motor_temperature" : 1,
      "front_motor_current" : 1,
      "rear_motor_current" : 1,
      "front_motor_voltage" : 1,
      "rear_motor_voltage" : 1
```

```
    },
    "pressure": {
      "ambient_pressure" : 1,
      "coolant_pressure" : 1,
      "pnematic_circuit_pressure1" : 1,
      "pnematic_circuit_pressure2" : 1,
      "pnematic_circuit_pressure3" : 1,
      "pnematic_circuit_pressure4" : 1
    },
    "actuators" : {
      "aotorSpeed" : 0,
      "motrs_CoolantPump" : 0,
      "brakes_3and4Set" : 1,
      "brakes_1and2Set" : 1,
      "brakes_ActuatorsSpeed" : 0,
      "brakes_valve1State" : 1,
      "brakes_valve2State" : 1,
      "brakes_valve3State" : 1,
      "brakes_valveAuxState" : 1,
      "energy_ReadingBEnable" : 1,
      "energy_ReadingAEnable" : 1,
      "energy_BatteryBSwitch" : 1,
      "energy_BatteryASwitch" : 1
    },
    "electric_brakes" : {
      "position" : 1
    },
    "other":{
      "ambient_pressure" : 1,
      "ambient_temperature" : 1
    },
    "config":{
      "stripe_count" : 0,
      "motor_speed" : 0,
      "timeout" : 0,
      "max_speed" : 0
    }
  },
  "stateMachine" : {
    "status" : "IDLE",
    "last_status" : 1
  },
  "messageData" : {
    "timestamp" : 1
  }
}
```