Trabajo Fin de Grado

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Diseño de un sistema de medida de parámetros físicos en una motocicleta de competición

Autor: Francisco José Zapata Galafate

Tutor: Manuel Ángel Perales Esteve

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Carrera
Ingeniería Electrónica, Robótica y Mecatrónica

Diseño de un sistema de medida de parámetros físicos en una motocicleta de competición

Autor:
Francisco José Zapata Galafate

Tutor:
Manuel Ángel Perales Esteve
Profesor Titular

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Proyecto Fin de Carrera: Diseño de un sistema de medida de parámetros físicos en una motocicleta de competición

Autor: Francisco José Zapata Galafate

Tutor: Manuel Ángel Perales Esteve

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi madre.

A Cobi.

Gracias.

Agradecimientos

Tengo muchas gracias que dar, a mis amigos, a todos. Jesús por estar siempre, Lalo por las 2*k*pi vueltas a la urbanización hablando de los problemas de la vida, a mis compañeros del trabajo Gon y Alfon, a Raparranz y a Dani por ser mi gran valedor. A todos mis amigos de la ETSI y, en especial, a mis amigos de US-Racing: Guille por ayudarme con todo lo presente en este trabajo, Javi por ser un descubrimiento, Paco por la risa esa que se te pega e Ignacio por las salidas por la tangente que nadie espera.

Dar gracias también a toda mi familia, en especial a mi primo Juan Antonio que es mi hermano y la mejor persona que jamás conoceré. A mis primos pequeños que sacan mi mejor versión y a todas mis tías.

Gracias a mi padre porque es una persona con una capacidad de reinvertarse que me gustaría poder aprender algún día.

También a Salvador porque ha sido el gran profesor de mi vida enseñándome incluso sin verme, incluso sin saberlo.

Y los dos más especiales y por ello los últimos:

Gracias a Cobi, mi perrito, es la definición de la pureza, del querer porque sí y al máximo sin explicación, incluso sin lógica. Mi mejor elección.

Y a mi Madre. Es el único agradecimiento que, escriba lo que escriba, va a ser injusto porque no existen palabras. Siempre, para todo, contra todo. Me dio la vida y desde entonces no ha parado, hasta dármelo todo. El mejor ejemplo que he tenido de que uno solo puede todo, incluso criar a un hijo. Todos los momentos, las cenas, las series, los viajes, los paseos, las horas escuchadas… Tendría que vivir dos veces para poder devolver si quiera la mitad. Te quiero maní.

Francisco José Zapata Galafate

Sevilla, 2020

Resumen

En el presente trabajo se pretende detallar y explicar el diseño de un sistema de medida de parámetros físicos de una motocicleta de competición. Cabe destacar que, el proyecto consta de tres partes: la programación gráfica de un dispay, la implementación de un método de *datalogging* o guardado de datos y la representación de estos a través de la herramienta Matlab.

La necesidad de conocer determinados parámetros de la motocicleta en régimen de marcha y el elevado coste de alternativas ya existentes son los factores que motivan el diseño de este sistema. Siendo dicho diseño la antesala de la aplicación práctica como solución viable de ingeniería. A lo largo de este trabajo se expondrán los pasos de desarrollo y los problemas o limitaciones que podría tener esta solución en caso de aplicación práctica en una motocicleta de competición.

El prototipo de competición, para el cuál se ha diseñado este sistema de medida y adquisición de datos, es el prototipo USR18 realizado por la escudería de motociclismo de la Escuela Superior de Ingenieros: US-Racing.

Abstract

This paper aims to detail and explain the design and testing of a system for measuring the physical parameters of a racing motorcycle. The project consists of three parts: the graphic programming of a display, the implementation of a datalogging method and the representation of the data through the Matlab tool.

The need to know certain parameters of the motorcycle in while running and the high cost of the existing alternatives are the factors that motivate the design of this system. This design is the prelude to practical application as a feasible engineering solution. Throughout this work, the development steps and the problems or limitations that this solution could have in case of practical application in a competition motorcycle will be exposed.

The competition prototype, for which this data acquisition and measurement system has been designed, is the USR18 prototype made by the motorcycle racing team of the Escuela Superior de Ingenieros: US-Racing.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

Notación

A^*	Conjugado
c.t.p.	En casi todos los puntos
c.q.d.	Como queríamos demostrar
■	Como queríamos demostrar
e.o.c.	En cualquier otro caso
E	número e
IRe	Parte real
IIm	Parte imaginaria
sen	Función seno
tg	Función tangente
arctg	Función arco tangente
sen	Función seno
$\sin^x y$	Función seno de x elevado a y
$\cos^x y$	Función coseno de x elevado a y
Sa	Función sampling
sgn	Función signo
rect	Función rectángulo
Sinc	Función sinc
∂y ∂x	Derivada parcial de y respecto
$x_\circ$	Notación de grado, x grados.
Pr(A)	Probabilidad del suceso A
SNR	Signal-to-noise ratio
MSE	Minimum square error
:	Tal que
<	Menor o igual
>	Mayor o igual
\	Backslash
⇔	Si y sólo si

1 INTRODUCCIÓN

La moto no es un trozo de hierro, tiene alma, porque una cosa tan bella no puede carecer de alma.

- Valentino Rossi -

1.1 Contexto del proyecto

La motocicleta es un medio de transporte generalmente de dos ruedas impulsado por un motor que transforma energía almacenada en un medio físico, a energía cinética para el transporte de mercancías y personas. Además de por su versatilidad para uso recreativo y logístico, la motocicleta destaca por la elevada variedad y existencia de competiciones de velocidad que han sido creadas a lo largo de los años.

Huelga decir que, aunque las de velocidad son las competiciones más habituales, no son las únicas que se pueden encontrar en el mundo de las dos ruedas. Existen modalidades totalmente dispares, desde las más populares que son las carreras en circuito cerrado sobre asfalto con prototipos (entre las que destaca MotoGP) hasta las carreras campo a través en modalidad contrarreloj, como puede ser el Rally Dakar.

Pero a pesar de ser muy distintas, en unas competeciones dónde la igualdad es cada vez mayor y dónde sólo la victoria es una opción, aparece el factor común que comparten todas las competencias de motor. **La búsqueda de la mayor comprensión y mejor ajuste posible de la máquina**, en persecución de un rendimiento que permita al conjunto moto-piloto llegar donde los demás no puedan.

Figura 1: Piloto e ingenieros revisando los datos de la telemetría en el Gran Premio de MotoGP de Jerez.

Durante décadas, se comprendía y ajustaba el prototipo a través de la sensibilidad e indicaciones del ser humano que lo operaba. Aunque, con el paso de los años y el aumento de la tecnología (en especial la cada vez mayor presencia de electrónica en la gestión de la potencia) esto comenzó a ser insuficiente por si sólo.

Y es dónde confluyen los incrementos de competitividad y tecnología, el marco dónde aparecen los **sistemas de medición de parámetros físicos** aplicado a una motocicleta, o la comúnmente (y en adelante en este trabajo) llamada *telemetría*.

El concepto o definición del término ***telemetría*** en el campo de la ingeniería no es único, descomoponiendo su etimología se puede entender de manera más precisa su definición y aplicación al motociclismo de competición. El prefijo *tele-* proviene del griego antiguo y significa "lejos" aplicando el matiz "de modo remoto", "a distancia" o "desde lejos" a la palabra que compone. La terminación *–metría* proviene también del griego significando "metrón" indicando "medida o medición".

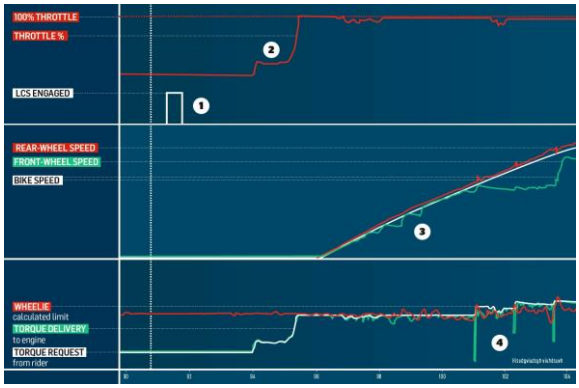


Figura 2: Gráfica de la telemetría del launch control o control de salida en una MotoGP

Aplicado a una motocicleta, el sistema de medida es llamado *telemetría* porque mide y almacena las medidas de los parámetros físicos de la moto mientras ésta está en pista, para luego más tarde y en otro lugar ver e interpretar los datos. Cabe especificar que, no está permitido en ninguna competición de dos ruedas enviar en directo dichos datos, debiendo verse en diferido de manera obligatoria.

Este trabajo surge por la necesidad de crear un primer diseño de ***telemetría*** con objeto de estudiar la viabilidad real de aplicarlo a un prototipo de competición. Dicho prototipo de motocicleta es diseñado y fabricado por el equipo US-Racing de la Universidad de Sevilla, que compite en el Campeonato Internacional de MotoStudent.

A lo largo del mismo, se analizarán las necesidades del sistema, el diseño de este, los posibles errores o problemas que pueda tener su aplicación práctica y una pequeña comparativa con los sistemas existentes en el mercado.

1.2 Resumen del Trabajo

El porqué del trabajo, la elección del diseño y su realización, como se ha especificado en el apartado anterior, responde a las necesidades y recursos del Departamento de Electrónica y Telemetría del equipo de motociclismo de competición: US-Racing. Siendo el recurso humano, el económico y sobretodo, el temporal los factores más limtantes a la hora de realizar esta tarea.

Antes de explicar el sistema completo, se van a detallar las necesidades antes referidas que el sistema ha de satisfacer.

- Primero, el tiempo que el equipo disponía desde que abordó el diseño y construcción del prototipo de telemetría hasta su fecha límite, era de apenas dos meses y medio.
- Segundo, el departamento en el que este proyecto se enmarca tiene unos recursos humanos limitados, concretamente cinco integrantes, pudiendo sólo uno de ellos dedicarse en exclusiva a esta tarea.
- Tercero, el sistema ha de ser lo más económico posible al ser el presupuesto disponible dentro de la escudería US-Racing uno de los puntos más críticos de la realización del diseño. Hay que destacar que, la mototicleta sólo va a realizar una carrera, por lo que el sistema debe ceñirse a los datos que sean interesantes de cara al ámbito competitivo ya que no se va a evolucionar la moto.

Teniendo en cuenta los tres elementos anteriores, el departamento decidió crear un primer prototipo de telemetría de la motocicleta para, una vez teniéndolo realizado, estudiar la viabilidad y problemas encontrados en pos de la realización de un segundo sistema que ya si tuviera aplicación práctica.

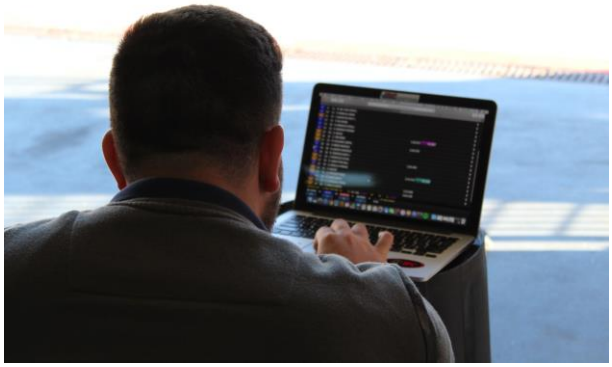


Figura 3: Miembro del departamento de electrónica de US-Racing observando los resultados obtenidos.

Es importante destacar, la importancia de la decisión de realizar un prototipo en primera instancia en lugar de un sistema completo directamente. Se tienen dos ventajas principales: en caso de obtener buenos resultados (o de solo tener que realizar determinadas variaciones) el sistema estaría prácticamente listo para su aplicación práctica. Y, en caso de obtener resultados negativos, no se habrían puesto en compromiso los escasos recursos de la escudería.

Por lo anterior, que será además más detallado en adelante, se ha realizado el *sistema prototipo de telemetría* que ocupa este documento.

El trabajo realizado se puede dividir en diferentes partes, concretamente se van a hacer dos distinciones: por un lado, el **sistema de medida** y, por otro lado, la **representación de los datos en un PC** u ordenador personal.

1.2.1. Sistema de Medida

El *sistema de medida* propiamente dicho es el dispositivo de adquisición de datos que va instalado en la motocicleta. Su función es la de leer el valor de cada sensor, almacenarlo y representar algunos de ellos a través de una pantalla que iría instalada en la tija de la dirección a modo de *dashboard*.

Para la realización de esta tarea, y teniendo en cuenta las limitaciones antes descritas, se eligió una arquitectura de varios módulos siendo:

- Un primer microcontrolador Arduino Mega 2560 que realiza la lectura de alguno de los sensores y los representa a través de la pantalla TFT.
- El segundo microcontrolador Arduino Mega 2560, se encarga de leer los datos de todos los sensores, les une los datos del GPS y se almacena todo en una tarjeta MicroSD.

Se eligió este esquema de trabajo y componentes por dos motivos: uno, obtener el mayor número de medidas posible por segundo y dos el bajo presupuesto de este sistema, además de su amplia compatibilidad. El motivo de la búsqueda de la mayor velocidad de adquisición posible no es otro que acercar la capacidad de este diseño de telemetría a la de las presentes en el mercado.

Más adelante, se tratarán dichas especificaciones y su lugar con respecto las necesidades del usuario y las opciones ya presentes en la industria (además del presupuesto necesario para éstas) y así poder enmarcar, comparar y evaluar el diseño realizado.

Hay que destacar que, otra de las partes vitales de la parte del Sistema de Medida son los sensores elegidos para medir las magnitudes físicas (y sus variaciones) con el Arduino. Todos estos sensores se especificarán, detallarán y justificarán en su apartado correspondiente más adelante en este trabajo.

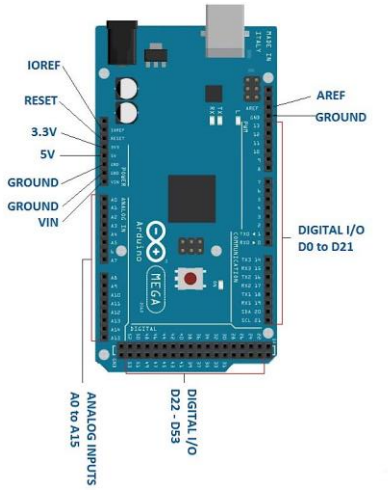


Figura 4: esquema del arduino Mega 256

1.2.2. Representación de los Datos en PC

Una vez la motocicleta ha terminado de rodar en el circuito, el piloto llega al box y es la hora de la descarga y estudio de los datos recogidos por el Sistema de Adquisición de Datos. Hay que recordar que, dichos datos los almacenaba el sistema de medida en una tarjeta SD que, una vez llegado el momento, se extrae e inserta en el ordenador de uno de los técnicos para estudiar los datos adquiridos.

Por lo que, surge una nueva necesidad que el sistema de telemetría que se está describiendo debe cumplir: un modo de representar gráficamente los datos y, a ser posible, poder verlos por canales (o sea, pudiendo elegir qué ver en cada momento). Esto es uno de los puntos más importantes, dado que, para analizar diferentes comportamientos de la motocicleta se necesitan algunos sensores en conjunto, pero no todos, por lo que poder elegir facilita y permite la tarea enormemente a los técnicos. En adición a lo anterior hay que resaltar que, un tratamiento correcto de los datos puede corregir fallos del sistema y aumentar las capacidades del mismo, si se tiene en cuenta la dinámica del sistema que se ha muestreado.

A priori no es una tarea trivial, sobretodo si la opción de la elección de canales es una máxima. Pero, al saberse de antemano que éste era uno de los requisitos del sistema, se guardaron todos los datos de los sensores, GPS y tiempo a modo de vectores, de modo que facilita esta tarea en gran medida.

Cómo se especificó antes, el requisito más limitante era el temporal, de modo que, se eligió para la programación de la interfaz gráfica una plataforma que era más que familiar: Matlab. Esta elección además tiene otras ventajas: como la flexibilidad en la programación, la potencia de la que hace gala la herramienta y, sobretodo que, al ser un programa ampliamente utilizado, cualquiera de los miembros del equipo puede utilizar el *script* realizado (que además es bastante ligero en cuanto a su tamaño) en cualquiera de sus equipos.

Al igual que en el caso anterior, todos los códigos, tareas realizadas y problemas encontrados serán detallados debidamente en el epígrafe correspondiente de este trabajo.

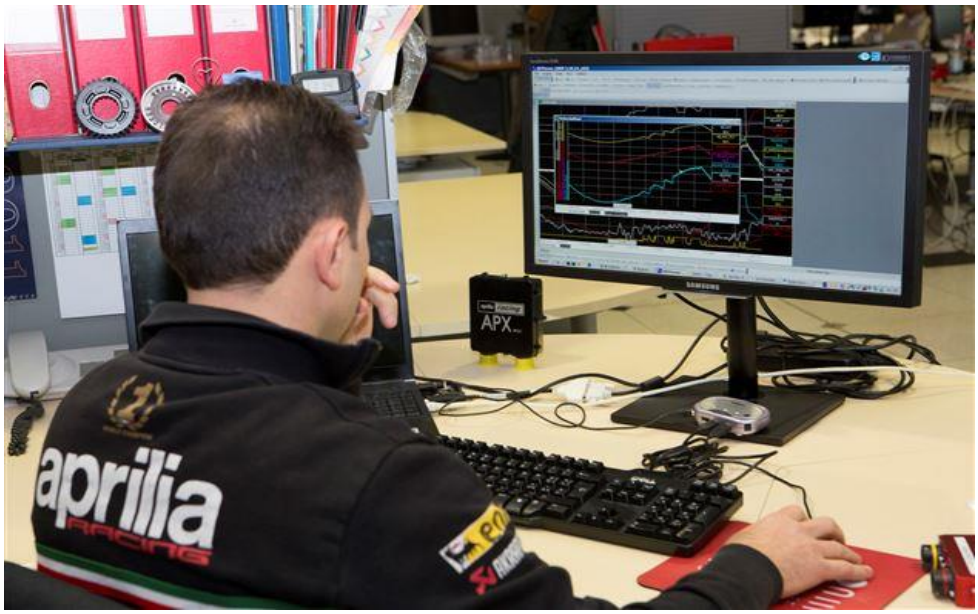


Figura 5: Ingeniero telemétrico del equipo Aprilia Racing Team MotoGP analizando los datos de la telemetría.

2 HISTORIA Y ESTADO DEL ARTE

2.1 Breve historia de la motocicleta y contexto histórico

Las motocicletas, y sobretodo las de competición son hoy en día un conjunto de disciplinas llevadas a la máxima expresión. Para la parte estructural se realizan cientos de simulaciones con diferentes materiales y formas hasta conseguir resultados punteros. El mismo procedimiento se utiliza en la parte aerodinámica y en términos de electrónica, se utilizan microprocesadores muy avanzados para la recogida y análisis de los datos. Pero para comprender esto, se va a volver hacia atrás hasta los orígenes de la motocicleta.

A lo largo de las siguientes páginas se va a explicar de manera general el progreso de esta máquina a lo largo de los años hasta llegar a la aplicación de la electrónica y sistemas de medidas en ella, que es lo que ocupa este trabajo.

El primer vehículo que puede aproximarse al concepto de motocicleta como se conoce hoy día fue creado en el año 1867 por el estadounidense Sylvester Howard Roper. Aquel modelo era impulsado por un motor de cilindros accionados por vapor generado a través de la combustión de carbón.

Sin embargo, el primer medio de transporte considerado por los historiadores como motocicleta en el más estricto de los sentidos fue diseñado en 1876 por Wilhelm Maybach y por Gottlieb Daimler montando por vez primera un motor de combustión interna de ciclo Otto de cuatro tiempos. Durante los próximos años la motocicleta no cejó en su desarrollo, surgiendo las primeras marcas de desarrollo en serie y venta al gran público.

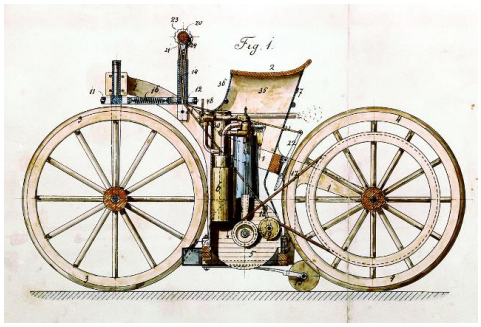


Figura 6: Motocicleta creada por Gottlieb Daimler con motor monocilíndrico de 4t.

Como cualquier elemento del mundo de la ingeniería, fue en tiempos de guerra cuando la motocicleta experimentó un gran avance en un breve lapso de tiempo.

Durante la Primera Guerra Mundial, la producción de estos vehículos aumentó de manera exponencial al sustituir al caballo como medio individual de transporte de mensajería y mercancía de pequeño volumen en el frente.

Como dato, la compañía Triumph vendió más de 30,000 unidades del modelo *Triumph Type H* (impulsado ya por un motor de 499 cc) a las Fuerzas Aliadas durante este período bélico. Fue ésta la considerada como "primera motocicleta moderna" gonzando de una fiabilidad tan grande que le hizo ganarse el apodo de "*Trusty Triumph*" que en lengua cervantina sería "*La confiable Triumph*".

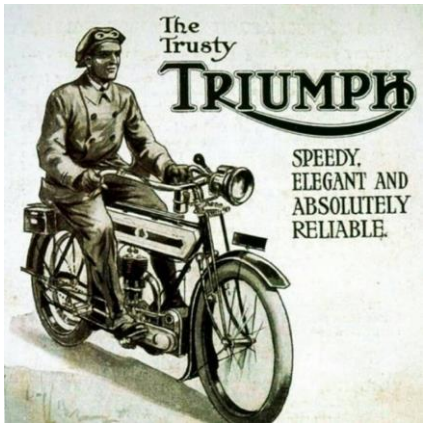


Figura 7: Póster comercial de la Triumph Type H

Ya en la segunda guerra mundial, el uso de motocicleta como medio de transporte en la guerra fue una constante en ambos bandos, modelos más ligeros, más resistentes y rápidos fueron creados.

Dentro de esta etapa histórica, fueron los siguientes modelos los más revolucionarios en términos de producción y tecnología: En el bando Aliado, la Harley Davidson WLA (con más de 70 mil unidades producidas) y la BSA M20 (con la increíble cifra para la época de 126 mil unidades producidas) fueron las dos mejores bazas del bando. Con mención especial para la Royal Enfield WD/RE un modelo especialmente interesante en términos de ingeniería dado que era tan ligero que un soldado podía cargarla en peso, al haber sido pensadas para lanzarse en paracaídas con ellas.

Pero el fabricante (porque no sólo realizó un modelo concreto) que supuso un antes y un después en términos tecnológicos durante la WWII fue BMW. Sus motos eran tan avanzadas para la época que, el ejército aliado robaba y reacondicionaba las BMW del ejército nazi al no poder igualar las prestaciones con los modelos americanos e ingleses antes listados.

Una vez terminada la Segunda Gran Guerra, la popularidad de las motos creció de manera exponencial debido a la cantidad de unidades que se produjeron y a la cantidad de soldados que volvieron enamorados de ellas del frente. Cada vez más motocicletas poblaban las calles, siendo ya un medio de transporte plenamente aceptado y muy popular dada la velocidad que eran capaces de alcanzar en comparación con los coches.

Llegados a este punto de la cronología, en el año 1949 se produce un giro vital para llegar a la aplicación de la electrónica en la moto, se crea el primer Campeonato del Mundo de Motociclismo. A partir de este momento, el avance tecnológico es exponencial hasta llegar a los modelos que se pueden observar y disfrutar en nuestros días.

Las primeras competiciones de dos ruedas se realizaron utilizando modelos derivados de las motocicletas de serie, poniendo a punto al máximo estas máquinas para poder extraer de ellas el mayor rendimiento posible. Pero, pronto comenzaron a utilizarse prototipos excluisvos fabricados y diseñados por las marcas para la competición. En pos de una mayor igualdad, las diferentes categorías comenzaron a separarse de acuerdo al tamaño de los motores, pudiendo elegirse el número de cilindros y arquitectura según gustos.

Figura 8: Mike Hailwood corriendo el campeonato del mundo en el año 1962.

Ya en la década de los 60 y 70, el motociclismo de competición es un deporte de fama y prestigio tanto para pilotos y marcas, existiendo categorías para motores de hasta 500cc y siendo algunas de las marcas presentes MV Augusta, Yamaha, Cagiva, etc.

La introducción de prototipos disparó la introducción de nuevas tecnologías y el avance de los modelos de competición. Al no tener que venir heredados de motocicletas de serie, la inversión, capacidad y posibilidad de investigar e introducir nuevas soluciones fue cada vez mayor. El nivel del campeonato del mundo fue subiendo y los prototipos de competición fueron cada vez más complejos, rápidos e iguales. Ya no bastaba con batir deportivamente a los rivales, sino que había que hacerlo también técnicamente.

Y es ya en la década de los 80, cuando surge un ingeniero que cambiaría la forma de diseñar y entender las motos: Antonio Cobas.

Este ingeniero natural de Barcelona es, considerado por muchos, el creador del motociclismo tal y como se conoce hoy. Se formó como ingniero trabajando primero en automovilismo (que en términos de técnica siempre ha ido un paso por delante del mundo de las dos ruedas) pero fue en el diseño de chasis para motos de competición dónde realizó su primera gran aportación.

Figura 9: Chasis "deltabox" o doble viga inventado por Antonio Cobas.

Fue el creador del chasis llamado "deltabox" o "doble viga de aluminio" que luego adoptaron otras marcas, incluso sigue presente en nuestros días en los que motos como la Yamaha M1 2109 llevan evoluciones de aquel primer modelo, mantiendo la estructura y la tipología de "doble viga".

Este genial ingeniero catalán siguió creciendo y mejorando en el mundo del motor, hasta que en 1989 le creó a un jovencísimo Álex Crivillé la motocicleta de 125 cc que le haría ganal el mundial, dando por completo la sorpresa.

Este hecho es muy importante, pero su nacimiento está algunos años atrás, concretamente en 1982. En este año Antonio Cobas compró su primer ordenador y, al instante y, citando sus propias palabras: "*Vi que mejoraba la productividad, la calidad de mis diseños, y la posibilidad de hacer cálculos muy precisos, así que decidí crear mis propios programas de cálculo de ingeniería... Así construí la JJ-Cobas 125 de Crivillé.*"

Figura 10: Álex Crivillé pilotando la JJ Cobas de 125 cc que le hizo campeón del mundo en 1989.

Fue en el campo de la programación dónde Cobas encontró una nueva pasión convirtiéndose en muy poco tiempo en un experto desarrollador de software. Observó que, las posibilidades de aplicación, tanto en diseño como "on board", de los ordenadores a las motocicletas parecían no tener límites.

Su siguiente innovación es herencia directa de lo anterior suponiendo un antes y un después en el motociclismo de competición. Antonio Cobas diseño, programó, creó e implementó el primer **sistema de adquisición de datos para motocicletas.** A través de la lectura de datos a través de sensores, medidas de tiempo y el tratamiento de los datos, pretendía entender los que otros solo intuían y ver lo que para los demás estaba oculto. Además, permitía un modo de poner a punto la moto infinitamente más preciso que las palabras de un piloto, que podían ser relativamente subjetivas e incluso depender de su habilidad a manos de la máquina.

Pero lo más importante para Cobas de esta aplicación no era la mera comprensión competitiva de lo que en la moto y en el piloto estaba sucediendo, sino que había conseguido lo que para él era "la cuadratura del círculo". Cobas era capaz de, a través de su propio software, calcular los parámetros de diseño de la moto, simular los comportamientos que podía tener y con sus sitema de adquisición de datos, confirmar que todo lo anterior encajaba.

Es complicado valorar cuál de sus aportaciones fue más importante, si fueron sus conceptos de chasis o su aplicación de la electrónica. Por un lado, sus ideas, conceptos de chasis e invenciones fueron adoptadas por una gran mayoría de fábricas e ingenieros, incluso su modo de trabajar, sus cálculos y conclusiones son una máxima innegociable hoy en día. Pero por otro, absolutamente todos los fabricantes, equipos y marcas de motos utilizan sistemas de adquisición de datos. Tanto en competición como para diseño, utilizando ese mismo sistema la propia motocicleta para realizar la gestión electrónica del motor.

Figura 11: Antonio Cobas revisando el software de adquisición de datos realizado por él mismo para una moto de competición de 500 cc.

El idilio de Anotnio Cobas con el mundo de la motocicleta de competición termina de la manera más trágica posible, con el genial diseñador apartado de la competición a causa de una enfermedad fatal. Aunque antes de este lamentable final, dejó un último legado, un alumno aventajado.

A principios de los años 90, un jovencísimo Ramón Aurín aceptaba colaborar con un ya experimentado Antonio Cobas que necesitaba de su ayuda para desarrollar sus sistemas telemétricos y para interpretar los datos adquiridos. Codo con codo, fueron desarrollando los sistemas que Cobas tenía prototipados y desdibujados, convirtiéndolos juntos en una realidad cada vez más avanzada.

El primer gran reto que tuvieron que afrontar juntos era una cuestión de espacio, en dos sentidos completamente diferentes (por no decir opuestos) de la palabra. Los sistemas que realizaban eran tremendamente grandes a nivel físico, pero gozaban de un espacio de memoria informática demasiado pequeño para los datos que debían almacenar. Por lo que, desde ese momento, la telemetría, cableados y centralitas comenzaron a necesitar un lugar de honor a la hora de diseñar las motocicletas de competición.

Figura 12: Ramón Aurín trabajando con Jorge Lorenzo en la temporada 2019.

Muchos avances se han realizado desde los primeros sistemas desarollados por Aurín y Cobas, para adquirir datos del comportamiento y funcionamiento de la motocileta, hasta los que existen hoy en día. Si se tuviera que marcar un punto de inflexión en el tiempo, podría decirse que fue en la segunda mitad de la primera década de los 2000 cuando los sistemas ya tenían capacidades altas similares a las actuales, muy alejadas de los sistemas primitivos de la dupla de ingenieros españoles Cobas-Aurín.

El momento actual de a *telemetría de competición* se tratará en el punto de este trabajo denominado: ***Estado del Arte***.

En paralelo a la evolución de la motocicleta que se ha expuesto en los párrafos anteriores, se ha producido la evolución de los prototipos de US-Racing. El equipo ha participado en las cinco ediciones bienales del campeonato MotoStudent, modificando y mejorando sus prototipos en cada una.

El primero de ellos era una motocicleta de 125 cc con motor de dos tiempos, que era el impuesto por la organización del evento para aquella competición. Hay que decir que, las soluciones adoptadas eran de un perfil más tradicional dado que era el primer prototipo que se realizaba. Aún así destacó por la utilización de elementos estructurales de fibra de carbono, algo no tan común hace unos diez años. Pero en el ámbito que se encuadra este trabajo hay que decir que, la moto carecía totalmente de algún sistema de medida de parámetros.

Figura 13: Quinto prototipo realizado por el equipo US-Racing para el que se diseñó el prototipo de telemtría de este trabajo.

Para el segundo prototipo el campeonato evolucionó hacia motores de 4 tiempos y 250 cc, este modelo realizado por la Escuela de Ingenieros gozaba de un gran trabajo de cálculo y análisis, pero de nuevo, no montó ningún sistema de adquisición de datos. En este caso la razón fue meramente temporal, el equipo intentó hasta el útimo momento refinar el prototipo al máximo y dado que sólo corre una vez, la telemetría se entendió como algo accesorio.

En la tercera y cuarta edición ya si se plantearon y proyectaron sendos sistemas de telemetría que por diferentes motivos no llegaron nunca a construirse. En el caso de la tercera edición, fue debido a unos problemas con el motor que lastraron el desarrollo del prototipo y absorbieron todos los recursos (humanos y económicos) del proyecto hasta el mismo día de la carrera.

Para la cuarta, los problemas fueron sobretodo organizativos, no siendo capaz el equipo de acometer correctamente la realización del trabajo proyectado. El número de personas y el margen de tiempo no fueron suficientes y esto impidió la correcta realización del conjunto.

Por todo lo anterior, para esta quinta edición se proyectó la tarea del modo que se explicó en la *Introducción,* es decir, creando primero un prototipo que diera una idea aproximada del reto que se estaba afrontando. En caso de tener éxito, ya se tendría prácticamente el sistema completo casi listo para su instalación posterior. Y si las conclusiones no eran buenas, sólo bastaría con realizar los cambios o las adaptaciones pertinentes, pero partiendo de la base del mismo sistema. Como resulta evidente, hay algunas partes que ya serían definitivas independientemente de si la adquisición de datos se realiza de una manera u otra, como, por ejemplo, la programación de toda la parte de representación de los datos.

2.2 Estado del Arte

A lo largo de este apartado se pretende, continuando de manera aproximada con el anterior, explicar y detallar como se encuentra el mundo de la telemetría en el motociclismo actualmente. Hay que señalar que, esta no es tarea fácil debida a las razones que a continuación se exponen.

La primera de ellas es que, en términos de competición de alto nivel (como el Campeonato del Mundo de MotoGP) es prácticamente imposible conocer las capacidades, características y especificaciones del sistema desde fuera.

Además, cada empresa guarda con extremo recelo su forma de adquirir los datos, las operaciones que se realizan y la manera de guardar y transmitir los valores medidos. A lo que hay que añadirle que, cada fabricante (sea de motos en general, o de telemetría en particular) suele realizar su propia telemetría y siguiendo su propia estrategia en la alta competición.

Figura 14: Parte del sistema de Adquisición de Datos de una MotoGP

Es decir, hay marcas que compran sistemas de telemetría a empresas del sector (como AIM), realizan colaboraciones creando partes la marca y subcontratando otras o incluso las más grandes realizan su propio sistema completo.

Otra de las razones, y sin salir del mundo de las carreras, es que los sistemas que se utilizan y aplican pueden variar muchísimo de un tipo de motocicleta a otra, o de un tipo de competición a otra. Por ilustrar un ejemplo, no se necesitan los mismos datos para estudiar el comportamiento de la moto si es una competición sobre asfalto en circuito, (que no presenta irregularidades, por ejemplo) o si es una competición rally como el Dakar (dónde sí interesa medir como se comporta en terreno irregular el vehículo).

Tanto la opacidad de los sistemas utilizados en competición, como la amplia diversidad de sistemas posibles de encontrar en el mercado, motivan y ratifican la decisión que tomó el equipo de crear un prototipo de ***sistema de medida de parámetros físicos*** para estudiar, resultados en mano, su viabilidad.

Figura 15: software de representación de los datos adquiridos de la telemetría AIM.

Para ubicar de la manera más precisa posible (teniendo en cuenta lo anterior) en qué punto se encuentra el mundo de la *telemetría* en el momento actual, se van a tratar dos puntos claramente diferenciados: la importancia de la telemetría actualemente, y, los sistemas que se pueden encontrar.

2.2.1. Importancia de la telemetría en la actualidad

La telemetría, como se ha especificado con anterioridad, se trata de la medida de parámetros a distancia generalmente en directo, pero, se ha extendido el concepto incluyendo entornos en los que las medidas se leen en diferido.

La sensorística, como disciplina general, está completamente extendida en nuestros días y presente en cada uno de los productos y máquinas que se utilizan. Con el progreso de la industria, y más concretamente de la electrónica, la necesidad de medir parámetros físicos y la aplicación de sensores ha sido una consecuencia directa.

Generalmente, esto ha sido en busca de dos propósitos diferentes: la mejora del rendimiento y capacidades de la máquina, por un lado, e interactuar con el entorno o el usuario, por otro.

El ejemplo más destacado de esto aparece en la telefonía móvil, en la que el dispositivo medio tiene aproximadamente 10 sensores diferentes que son:

1. Acelerómetro
2. Barómetro
3. Giroscopio

4. GPS
5. Lector de huella dactilar
6. Magnetómetro (o brújula electrónica)
7. Sensor de proximidad
8. Sensor de luz ambiental
9. Sensor infrarrojo
10. Termómetro (por seguridad de los componentes y batería)

Figura 16: sensor de huella dactilar de Apple.

Obviamente, si se retrocede hasta uno de los primeros modelos de telefonía móvil más conocidos, el Nokia 3310 lanzado en el año 2000, la lista de sensores de los que dispone es nula. Solo pudiendo considerarse como sensores el micrófono y el encargado de calcular el nivel de batería.

Y observando los modelos más avanzados, se tiene la lista anterior y a la que se pueden añadir sensores como el podómetro, lector de iris o pulsómetro, entre otros. El último avance en este sentido, y sólo como curiosidad, ha sido el sensor de huella dactilar por ultrasonidos, pudiendo colocarse debajo de la pantalla al no necesitar contacto directo con la huella.

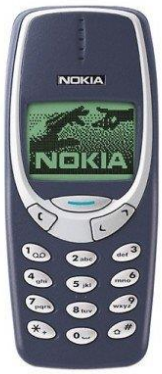


Figura 17: Nokia 3310

Este cambio exponencial en términos de sensores, lo han sufrido todos los dispositivos electrónicos que utiliza el usuario medio, televisores, planchas e incluso frigoríficos están hoy día repletos de sensores que miden los parámetros que definen el comportamiento de la máquina. Llegando, en aplicaciones como la domótica, a ser telemetría en toda su definición, dado que los valores de dichos sensores se observan y controlan a distancia.

Otro de los campos en los que se ha realizado un tremendo avance en telemetría ha sido en medicina, o más bien, en la adquisición y monitorización de parámetros vitales. Desde las aplicaciones médicas más exigentes hasta los últimos *wearables* de moda, la medida de parámetros como el pulso, la temperatura corporal, presión sanguínea e incluso la monitorización del sueño están a la orden del día.

Dentro de esta revolución sensorística no se puede ignorar ni dejar de lado el mundo del motor, en el que se va a entrar más en detalle.

Los primeros vehículos de calle en incorporar distintas clases de sensores fueron los automóviles, teniendo hoy en día hasta 24 tipos diferentes en un mismo coche. Huelga decir que, se está hablando exclusivamente de la tipología, siendo la cantidad númerica de sensores muy superior (dado que muchos de ellos están repetidos).

Al igual que en los casos anteriores, la introducción de esta gran cantidad de transductores (y de la electrónica asociada), ha sido en pos del aumento de rendimiento del vehículo y de su interacción con el usuario, en forma de confort y seguridad. Atendiendo a estas tres familias, se van a diferenciar los tipos de sensores que se pueden encontrar en cualquiera de los vehículos que ocupan hoy día las carreteras.

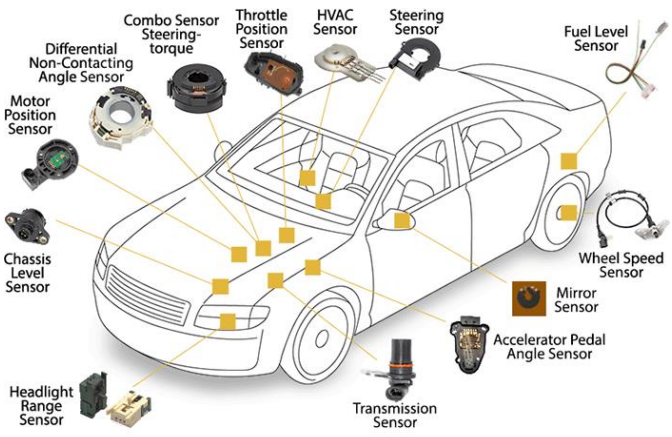


Figura 18: algunos de los sensores que tienen los coches en la actualidad.

1. ***Sensores de motor y transmisión***: encargados de evaluar, mejorar e interactuar con el rendimiento y la parte ciclo del motor, algunos ejemplos son:
    a. Sensor de Presión
    b. Sensor de Presión de la Sobrealimentación
    c. Sensor de Masa de Aire
    d. Sensor de Presión Ambiente
    e. Sensor de Alta Presión (en la inyección de combustible)
    f. Sonda Lambda (mide la pureza de los gases de escape)
    g. Sensor de Velocidad de Rotación
    h. Sensor de Presión en el Depósito
    i. Sensor de Posición de los Pedales
    j. Sensor de Ángulo de Posición del Árbol de Levas
    k. Termómetros para Aceite, Motor y Agua.
2. ***Sensores de Seguridad:*** su misión es la de evitar accidentes, adelantarse a sucesos peligrosos y mejorar la conducción en condiciones de riesgo.
    a. Radar Telemétrico (para evitar colisiones)
    b. Sensor de Inclinación (para regular correctamente faros o incluso suspensiones)
    c. Sensor de Par
    d. Sensores de Alta Presión y Ángulo de Volante de Dirección (para el ESP)
    e. Sensores de Ocupación de Asientos
    f. Sensor de Aceleración Transversal (también para el ESP)
    g. Sensor de Inclinación
    h. Sensor de Vuelco
    i. Sensor de Velocidad de Giro de Ruedas
3. ***Sensores de Confort:*** involucrados en haer de la conducción una mejor experiencia y más disfrutable al usuario.
    a. Sensor de Viraje (para la navegación)
    b. Sensor de Calidad del Aire
    c. Sensor de Lluvia
    d. Termómetros (exterior y habitáculo)
    e. Sensores de Ultrasonido (para localización de personas o aparcamiento)

Figura 19: Sensor de proximidad mediante ultrasonidos

Simplemente observando el tamaño de la lista anterior y comparándola con la que se ha redactado anteriormente sobre el mismo tema en *smartphones*, se puede ver que, esta revolución ha sido aún mayor en los vehículos.

Una vez se ha introducido el mundo del motor, encuadrados en el avance y la importancia de la telemetría, se va a caracterizar al tema que ocupa este trabajo: la telemetría en motocicletas y más concretamente las de competición.

En el caso de las dos ruedas, existió un enorme punto de inflexión cuando se introdujo la inyección electrónica en las motocicletas (tanto de competición como de calle). La gestión de la potencia siempre se ha realizado controlando la ratio de la mezcla (relación aire/combustible introducida en el cilindro) y la cantidad de esa mezcla que se introduce en el motor, pero antiguamente se realizaba a través de carburadores, sistemas mucho menos flexibles y mucho más rudimentarios que la inyección electrónica. Más tarde, se introdujeron los sistemas que, a través de una centralita electrónica que controla los inyectores de combustible, calculaban al milímetro y de manera preprogramada la mezcla y su inyección.

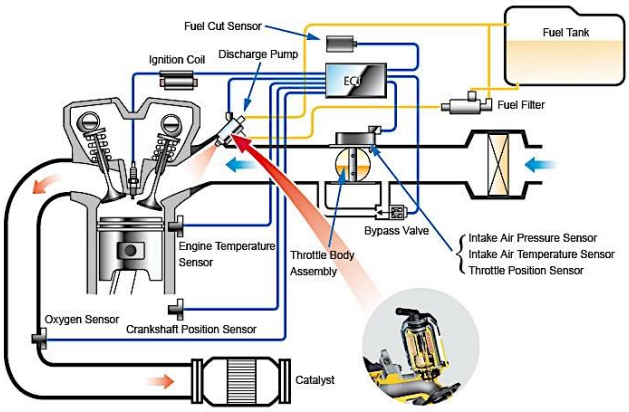


Figura 20: diagrama de la inyección electrónica en una moto, y algunos de los sensores necesarios.

Figura 21: ECU de competición de Moto3.

La centrailta de la moto o ECU (Engine Control Unit) puede ser programada por el usuario de innumerables formas diferentes, pudiendo variar parámetros como el avance de chispa, el tiempo de inyección, la pureza de la mezcla, etc. Pero, para poder realizar todo este tipo de actuaciones diferentes y ser programadas según ciertos sucesos o acciones del piloto, la ECU necesita una gran cantidad de información de la motocicleta y de como está siendo operada. Es aquí donde surge la necesidad de añadirle sensores electrónicos a la moto que "le dicen" a la centralita lo que está ocurriendo en el vehículo.

Estos sensores son además vitales para los sistemas de seguridad del vehículo, que en las motocicletas no son tan numerosos como en los coches, pero son más importantes (aún si cabe), dado que la peligrosidad es más elevada. Los avances más importantes y que han requerido explícitamente de sensorística para su implementación son: el ABS y el Control de Tracción.

El ABS es una implementación y variación directa del que se ha realizado en coches, siendo el funcionamiento exactamente el mismo: cuando la rueda se bloquea por frenada máxima, el sistema acciona y libera el freno intermitentemente para sacar la rueda de la posición de equilibrio que significa el bloqueo, haciendo que vuelva a frenar correctamente.

El Control de Tracción, sin embargo, es una revolución electrónica muy diferente a su implementación para automóviles, dado que requiere de un sistema de sensores más complejo.

Esto es porque la motocicleta se mueve en 3 dimensiones (al inclinar además de girar) mientras que un coche o camión se mueve exclusivamente en dos (realmente son 3 pero lo que bascula la suspensión es despreciable en comparación con los grados de inclinación de una motocicleta). El control de tracción observa la velocidad de ambas ruedas y el ángulo de inclinación del vehículo, permitiendo la aceleración proporcional según esté el piloto inclinando y si las ruedas están derrapando o no (velocidades diferentes). Esto ha supuesto un descenso drástico de un tipo de caídas muy frecuentes y peligrosas en el mundo de las ruedas, el denominado *highside,* que se va a mostrar y detallar.

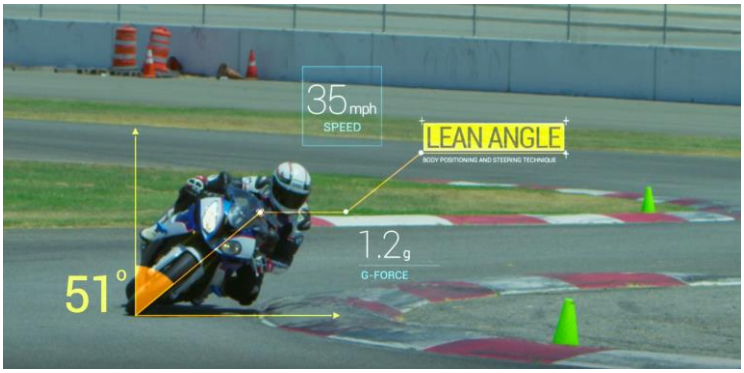


Figura 22: Representación gráfica de la dinámica de una moto.

En la secuencia de la *Figura 23* se puede ver un highside que sufrió el piloto Jorge Lorenzo en el circuito de Shangai en 2008 por fallo de los sensores y, como consecuencia, la inacción del Control de Tracción asociado. En el paso 1 la motocicleta está entrando en curva, en los pasos 2 y 3 se puede ver como la rueda trasera (la tractora), al tener mayor velocidad que la delantera, pierde adherencia. En menos de un segundo y antes de que el piloto pueda reaccionar, ocurre el paso 4, la rueda trasera recupera adherencia de golpe y al ir mucho más rápido que la delantera causa la contracción repentina del amortiguador trasero. En el paso 5, que ocurre escasas décimas de segundo más tarde, el amotiguador (por inercia) se estira de golpe, catapultando (paso 6) al piloto.

Todo esto no hubiera ocurrido de funcionar correctamente el sistema de sensores, dado que, al notar una disparidad de velocidades entre las ruedas delantera y trasera, y además un cierto grado de inclinación, el sistema hubiera cortado la potencia del motor de inmediato. Evitándose de este modo el accidente antes descrito.

Figura 23: Accidente de Jorge Lorenzo en el GP de Shangai 2008 a causa de un fallo en el sistema de sensores y telemetría.

Son ejemplos como el anterior, el motivo de la tremenda importancia de los sensores, y en concreto de la telemetría dado que evita caídas muy peligrosas en circuitos y fatales en la vía pública. Este sistema no es puramente telemetría hoy en día, pero vino heredado de ella, pues toda la información que necesita la ECU para "detectar" esta situación fue consultada y observada primero por los ingenieros con sistemas como el que (en estado prototipo) ocupa a este trabajo.

2.2.2. Sistemas Actuales de Telemetría en el Motociclismo de Competición

Una vez se ha desgranado e indicado la importancia de los sistemas de telemetría y sensores en la actualidad, se van a intentar exponer y aclarar algunos de los sistemas aplicados en el motociclismo.

Generalmente los sistemas de telemetría están creados para ser utilizados en un contexto de competición de velocidad, dado que la información que ofrecen permite (como ya se explicó) poner a punto la motocicleta de manera más precisa. Esta necesidad no la tiene un usuario medio que utiliza este vehículo a modo de medio de transporte o para realizar viajes o paseos.

Aún así se debe decir que, existen hoy día aplicaciones para los dispositivos móviles que son capaces de conectarse con la centralita (que suele leer todos los sensores de la motocicleta) y muestran por pantalla algunos de estos datos para que los usuarios más curiosos puedan disfrutar como aficionados de esta información. Son especialmente populares en las motos eléctricas que hay ya en el mercado, que se centran en aspectos algo más tecnológicos que las marcas tradicionales y desarrollan software de este tipo, como hace la marca Zero Motorcycles. Obviamente no suelen suponer una solución de ingeniería válida, dado que, tienen importantes limitaciones (como la frecuencia de refresco y adquisición de los datos) que son inadmisibles para un usuario profesional o una aplicación en términos de competición.

Figura 24: Diferentes pantallas y opciones de una de las aplicaciones móviles para motos eléctricas.

Para aplicaciones con requisitos más elevados, como por ejemplo en los campeonatos de velocidad, los diferentes equipos que forman parte de ellos pueden tomar dos caminos: comprar un sistema comercial profesional o desarrollar el suyo propio.

Los sistemas comerciales son muy variados y complejos, pero como ocurre con cualquier tipo de necesidad, existe una amplia gama tando de marcas como de productos capaces de cumplir los requisitos del usuario adaptándose a él.

Una de las marcas comerciales con una mayor presencia y reputación en el mundo de la telemetría de competición (tanto para vehículos de dos como de cuatro ruedas) es *AiM Sportline.* El gran éxito de esta marca reside en su amplia gama de productos que cubren desde los niveles más altos de competición hasta servicios casi para *amateurs*, y su elevado rendimiento calidad-precio. Además, consiguió a nivel técnico determindas innovaciones como hacer un todo en uno con el *display* y el *datalogger* y que sea completamente portátil como dispositivo. Otras marcas de telemtría comerciales son: *Beracing, Magneti Marelli, Dellorto,* etc. Para este tipo de sistemas, no es posible encontrar mucha información sobre la frecuencia de muestreo que utilizan, pero por compañeros del equipo US-Racing que trabajan con este tipo de telemetrías en su tiempo libre, se ha podido conocer que aproximadamente toman **una muestra cada 0.05 segundos**.

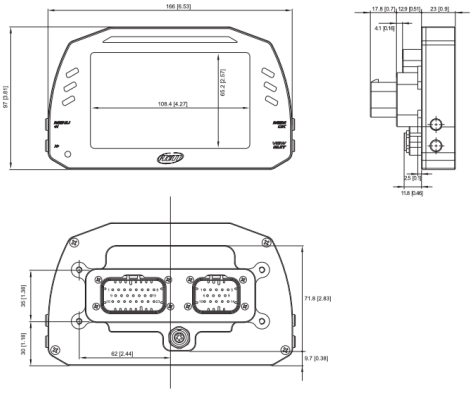


Figura 25: Uno de los modelos de Display-Data Logger de la empresa AiM.

En términos de sistemas de telemetría desarrollados íntegramente por equipos o marcas, es completamente imposible saber a ciencia cierta el número de sensores, la velocidad de adquisición, etc. En algunos textos divulgativos se habla de un número aproximado de 25-30 sensores diferentes para la competición de MotoGP (que es la que cuenta con los prototipos más complejos), pero **es inviable encontrar información en términos de frecuencia de muestreo**, resolución, etc.

Debido a todo lo anterior, la tarea de situar las especificaciones que harían de un prototipo de telemetría uno válido para su aplicación práctica, es realmente complicada.

2.2.3. Objetivos del Sistema Proyectado

Cuando el equipo de motociclismo de competición US-Racing se embarca en la tarea de querer realizar un primer prototipo de telemetría para evaluar la viabilidad del mismo, se encontró con lo que se ha desarrollado en el epígrafe anterior: es imposible saber a ciencia cierta especificaciones de otros sistemas.

Por lo que, establecer el baremo que permitiera saber si el sistema era o no viable suponía un problema. De modo que, en lugar de intentar llegar a especificaciones de diseño o rendimiento de otros sistemas comerciales y así ver la validez o no del trabajo realizado, se hizo el camino inverso.

Se tuvo en cuenta que, los datos recogidos pretendían ilustrar el comportamiento de la motocicleta en pista, las variaciones de sus parámetros, sus movimientos, etc. Llegándose a la siguiente conclusión: si el prototipo de

telemetría no era capaz de ver los sucesos ocurridos en la moto, por tardar demasiado entre muestras, este sistema no sería útil y, por tanto, no valdría para el caso de aplicación que se está tratando.

Esto tampoco es fácil de cuantificar porque dependiendo del tipo de motocicleta y de su velocidad media, los sucesos en la misma pasarán en más o menos tiempo. Como se puede apreciar en la *Figura 26*, se han utilizado los datos oficiales del proveedor de frenos más importante del Campeonato de MotoGP para analizar los tiempos de frenado (uno de los momentos críticos en los que la adquisición de datos es muy importante) y ver las especificaciones a elegir.

Se ha buscado el circuito en el que nuestro prototipo rodará para realizar una estimación algo más precisa, y se ha elegido la frenada de final de recta dado que es cuando la motocicleta va a mayor velocidad y el tiempo entre muestras es más crítico.

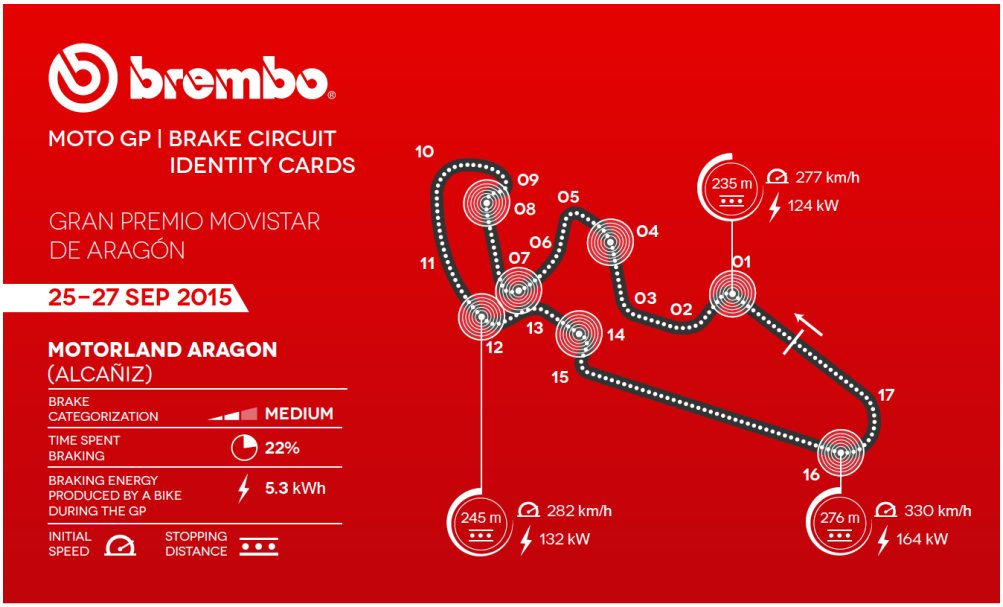


Figura 26: Datos oficiales de las frenadas del circuito de Aragon para una MotoGP.

Se quiere calcular el tiempo de frenado para estimar la frecuencia de muestreo del sistema. Primero, se calcula la velocidad inicial de frenado en metros por segundo:

$$330\frac{km}{h}\cdot\frac{1h}{60\ min}\cdot\frac{1\ min}{60\ s}\cdot\frac{1000\ m}{1km}=91.67\ m/s$$

Suponiendo que la motocicleta realiza un movimiento rectilíneo uniformemente acelerado a lo largo de su frenada (lo que simplifica bastante los cálculos) y teniendo en cuenta las dos ecuaciones que definen dicho movimiento:

Velocidad: $v=v_o+a\cdot t$

Distancia recorrida: $d=v_o\cdot t+\frac{1}{2}\cdot a\cdot t^2$

Uno de los datos que se necesitan para calcular el tiempo de frenado es la velocidad final, es decir, la velocidad a la que el prototipo toma la curva (numerada como 16 en la *Figura 26*). Observando que, la velocidad máxima en la recta es para una MotoGP y recurriendo a los datos y vídeos de la página oficial de MotoGP, se ve que la velocidad final en el cálculo que se quiere hacer es de:

$$v=145\ km/h=40.28\ m/s$$

Teniendo como datos la velocidad incial, final y distancia recorrida que son respectivamente:

$$v=40.28\ m/s\quad,\quad v_o=91.67\ m/s\quad,\quad d=276\ m$$

Y realizando un sistema de ecuaciones se puede obtener el tiempo de frenado y la desaceleración, que resultan:

$$a = -12.29\ m/s^2 \quad , \quad t = 4.18\ s$$

Es decir, para una de las frenadas más grandes del circuito, la motocicleta tarda en parar 4.18 segundos. Lo que quiere decir que, obteniendo, por ejemplo, **una muestra cada décima de segundo** el proceso completo de frenada se vería representado por aproximadamente 42 muestras.

Antes de entrar a valorar si ese número de muestras es suficiente resolución o no para el objetivo del prototipo, hay que decir que, los datos obtenidos son para una MotoGP (la motocicleta más rápida del mundo). La moto del equipo US-Racing pasaría por el mismo punto a una velocidad muy inferior. Pero en el punto de diseño del prototipo de telemetría que se encuentra el equipo, no disponía de datos suficientes para hacer el mismo cálculo conocidas todas las velocidades.

Figura 27: Comparativa de tamaños entre las diferentes motos del campeonato del mundo. La Moto3 es la que llevaría esta telemetría

Por lo que, para estimar todos los cálculos anteriores, pero referidos a la motocicleta que llevaría el sistema proyectado, se necesitan estimar ciertos valores que en el caso anterior (para una MotoGP) eran datos. Se pueden seguir dos estrategias diferentes a la hora de estimar los parámetros buscados:

- A través de la relación de los tiempos por vuelta.
- A través de la relación de velocidades máximas.

La estrategia elegida será la de la relación de velocidades máximas, dado que esa relación para un mismo punto del circuito es más fiable y extrapolable que un tiempo de vuelta completo (que puede variar por un conjunto mayor de eventualidades que una medida de velocidad máxima).

Haciendo el mismo análisis que en el caso anterior, se necesitará conocer los siguientes datos (a poder ser de manera precisa, pero pudiendo hacerse también una estimación):

- Velocidad Inicial (antes de comenzar a frenar).
- Distancia de Frenado. 150m
- Velocidad Final (al terminar el proceso de frenado y tomar la curva).

El primer dato que se debe conocer es la *Velocidad Inicial*, en el caso del análisis que se está llevando a cabo, será la máxima de la motocicleta de US-Racing: ***189 km/h***

La *Distancia de Frenado* es, la distancia que recorrida entre el punto inicial de frenado hasta el momento de ingreso en curva.

En la mayoría de los circuitos de velocidad existe una señalización que los pilotos deben usar de referencia para comenzar a frenar, en el caso del piloto de US-R en carreras anteriores, el punto de comienzo de la frenada es a ***150 metros*** de la curva. Este dato es aproximadamente común en este tipo de motocicletas de menor tamaño, dado que, las velocidades máximas y los pesos de las mismas son muy parecidos entre ella.

Para obtener la *Velocidad Final*, se establece la siguiente relación entre las *Velocidades Iniciales*.

$$r_v = \frac{189\ km/h}{330\ km/h} = \frac{52.5\ m/s}{91.67\ m/s} = 0.573$$

Por lo que, teniendo en cuenta el dato anterior y aceptando como válida esta segunda aproximación, se puede obtener la velocidad buscada:

- Velocidad Final: $v = 145\ km/h * 0.573 = 40.28\ m/s\ * 0.573\ = 23.08\ m/s$

Una vez obtenidos o estimados los datos necesarios, hay que rehacer el estudio anterior para obtener el nuevo tiempo de frenado. Es necesario recordar que, se necesita el tiempo que tarda la motocicleta en frenar (que es uno de los procesos más críticos) para estimar la resolución o cada cuanto tiempo la telemetría debería tomar muestras para ser válida.

Haciendo una pequeña recopilación para el análisis que se ha de repetir, los datos son:

- Velocidad Inicial: $v_o = 52.5\ m/s$
- Velocidad Final: $v = 23.08\ m/s$
- Distancia de Frenado: $d = 150\ m$

Y, tras aplicar los mismos cálculos, el resultado que se obtiene para los datos relativos al prototipo de Moto3 de US-Racing son:

$$a = -7.43\ m/s^2 \quad , \quad t = 3.96\ s$$

Si se observan los resultados obtenidos, el tiempo de frenado es muy parecido para ambos casos, por lo que se puede afirmar que, a pesar de ser un prototipo más lento, también tiene peor capacidad de frenado. Si (al igual que en el caso de las motos más veloces) se estima una muestra cada décima de segundo, el sistema que ocupa este estudio debería ser capaz de obtener unas 40 muestras para representar toda la dinámica de la frenada más exigente del circuito.

A modo de conclusión se puede decir que, si el sistema es capaz de capturar más de **10 muestras de todos los sensores en un segundo** (salvando quizás el GPS por tener requisitos especiales) podría catalogarse como válido en términos de resolución. Pero, si por el contrario tiene una resolución menor, éste no sería válido.

El número de sensores era también un punto crítico del diseño, pero en este caso la elección fue mucho más sencilla al intentarse sencillamente recoger los datos necesarios para el análisis en la única carrera que el prototipo de motocicleta iba a disputar. Los sensores que el prototipo de sistema de telemetría incluiría son:

- 2 sensores de elongación de las suspensiones (trasera y delantera)
- 2 sensores de temperatura (agua y aceite)
- GPS
- Sensor de apertura del puño del gas
- Pulsadores (para contar marchas)

Figura 28: Pulsador utilizado.

Una vez se tienen definidos los mínimos que el sistema de telemetría que se va a proyectar debe cumplir, se va a pasar a realizar un esquema del trabajo realizado enfocándose sobretodo en las diferentes partes, componentes, programación, etc.

3 DESARROLLO DEL SISTEMA

3.1 Introducción al trabajo realizado

Esta parte del trabajo será el grueso del mismo, dedicada a detallar el sistema que se ha proyectado, el hardware, el software, los sistemas microprocesados elegidos, la representación de los datos, etc.

Para una mejor organización de las tareas acometidas se van a dividir en los siguientes epígrafes:

1. **Hardware**: se tratará el hardware elegido, sus características, datasheets, etc.
2. **Dashboard o Display:** teóricamente es parte del *hardware* del sistema, pero, al ser una parte crítica se tratará de manera aislada y se incluirán *software y hardware.*
3. **Esquema de Montaje del Hardware:** se incluirá de manera esquemática cuál ha sido el montaje definitivo del sistema de adquisición de datos y pantalla.
4. **Software del Sistema On-Board:** en este primer apartado referido al software realizado se detallará y explicará la programación realizada para los sistemas microprocesados que iran en la motocicleta.
5. **Software del Sistema de Tratamiento de la Información:** en el segundo apartado del software se tratará la programación realizada en Matlab para el tratamiento y representación de los datos obtenidos.

Otra manera de clasificar las diferentes partes puede ser diferenciando entre los elementos (y su trabajo asociado) que irán en la motocicleta y los que no. Por un lado, todos los sistemas microprocesados y el display serían la parte que iría en el vehículo es decir ***On-Board***. Mientras que, la parte del software del PC se quedaría fuera del vehículo u ***Off-Board***.

3.2 Hardware

Como se ha refierido con anterioridad, esta parte del apartado 3 de la presente memoria es la referida al hardware utilizado. Dentro del mismo, se tienen claramente diferenciadas dos familias: los sensores y los sistemas microprocesados.

Se tratarán de manera diferenciada dado que son de naturaleza distinta, los primeros son transductores que medirán la variación de diferentes magnitudes físicas y, los otros, los sitemas encargados de leer y almacenar de manera automática dichas variaciones.

3.2.1. Microprocesador Empleado: Arduino Mega2560

La elección del sistema microprocesado que se ha utilizado para la realización de este prototipo de telemetría fue uno de los puntos clave o críticos del diseño dado que, en términos de velocidad de adquisición de datos, es posible que no hubiera un elemento más importante que éste.

Se tuvieron en cuenta otro tipo de placas[1] (y por ende otros tipos de microprocesadores) pero la programación desde cero, las pruebas necesarias tanto de dispositivos más simples como sensores analógicos hasta uno de los más complejos como el GPS y la búsqueda y creación de las bibliotecas necesarias, etc, hubiera sido demasiado costoso en términos temporales. Y como se

Figura 29: Arduino Mega2560 como el utilizado en el proyecto.

[1] Modo en el que se conoce al dispositivo completo, es decir, microprocesador y periféricos. No confundir sólo con el microprocesador que es el "cerebro" o parte computacional de la placa al completo.

especificó al principio del documento, el tiempo era junto con el dinero, el recurso más limitado en aquel momento.

Por todo lo anterior, se eligió utilizar la placa de Arduino Mega2560 que, aunque más simple, ya era conocida por varias de las personas que iban a trabajar con ella, tenía una gran compatibilidad con todo tipo de dispositivos (lo que ahorraría posibles problemas, tiempo y dinero) y además un gran soporte.

La placa **Arduino Mega2560** está basada en el microcontrolador ***ATmega2560*** que, a su vez, forma parte de la familia AVR creada por el fabricante estadounidense Atmel. Concretamente este grupo, denominado de manera general Atmega, son microcontroladores grandes que tienen desde 4 a 256 kB de memoria flash programable (256 kB para el caso del ATmega2560), encapsulados de 28 a 100 pines y con un diseño específico para la ejecución eficiente de código C compilado.

Algunas de las especificaciones más remarcables de la placa utilizada (Arduino Mega2560) son:

- 16 MHz de velocidad de reloj
- 256 kB de memoria Flash
- 4 kB de memoria EEPROM[2]
- 8kB de memoria SRAM[3] interna
- 54 pines digitales I/O (15 de los cuales dan salida PWM)

Además de los componentes ariba listados, lo interesante de Arduino es la gran cantidad de dispositivos o periféricos que admite pudiendo aumentar en gran medida la capacidad de la placa. Generalmente se presentan en forma de *Booster Packs* o *Arduino Shields*, que no son más que placas que se pueden conectar de manera directa con el Arduino (coincidiendo la mayoría de sus pines, aunque algunos no tengan función específica) y aumentando así las capacidades del mismo. Pueden ser desde módulos Bluetooth para transmisión de datos, conexiones Ethernet, o simplemente interruptores o pulsadores capacitivos.

Por ejemplo, el almacenamiento interno de Arduino Mega2560 es demasiado limitado como para hacer un *datalogging*[4] que sea realmente viable para una aplicación de telemetría sobre una motocicleta, es por ello que utilizando el ***puerto serie*** de la placa, se puede utilizar un lector de tarjetas SD para aumentar el almacenamiento del sistema completo a voluntad.

Todas las placas de Arduino tienen, entre sus componentes un puerto serie o UART, en el caso del Arduino Mega tiene hasta 4. Este tipo de pines o conexiones es de vital importancia dado que permite establecer comunicaciones con periféricos mucho más complejas que la simple lectura de un valor analógico.

En cuanto a sus dimensiones, la placa de Arduino Mega no goza de un gran tamaño. Lo cuál siempre supone una ventaja, aunque las restricciones de espacio no eran una de las peores en este proyecto. Las dimensiones de la placa se pueden ver en la *Figura 30*.

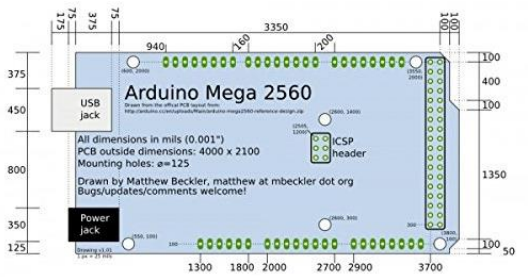


Figura 30: Plano del Arduino Mega 2560

Uno de los documentos más importantes a la hora de trabajar con una placa como ésta es el que define su *pinout* de manera completa. Esta información puede ser aportada de muchas formas diferentes, desde tabulada hasta gráfica, siendo ésta útima por la que se va a optar.

[2] Tipo de memoria ROM que peude ser eliminada y programada eléctricamente, no como la EPROM que necesita de rayos ultravioletas para ser eliminada.

[3] Memoria Estática de Acceso Aleatorio, un tipo de memoria RAM basada en semiconductores que no necesita circuito de refresco para mantener los datos siempre y cuando permanezca alimentada.

[4] Anglicismo utilizado para referirse al proceso de adquisición de datos y almacenamiento de los mismos para su posterior análisis. Palabra compuesta por los términos anglosajones "data" (datos) y "log" (anotación o registro).

La *Figura* (que se puede ver a continuación) servirá de documento de referencia cuando se trate la programación y cableado del sistema.

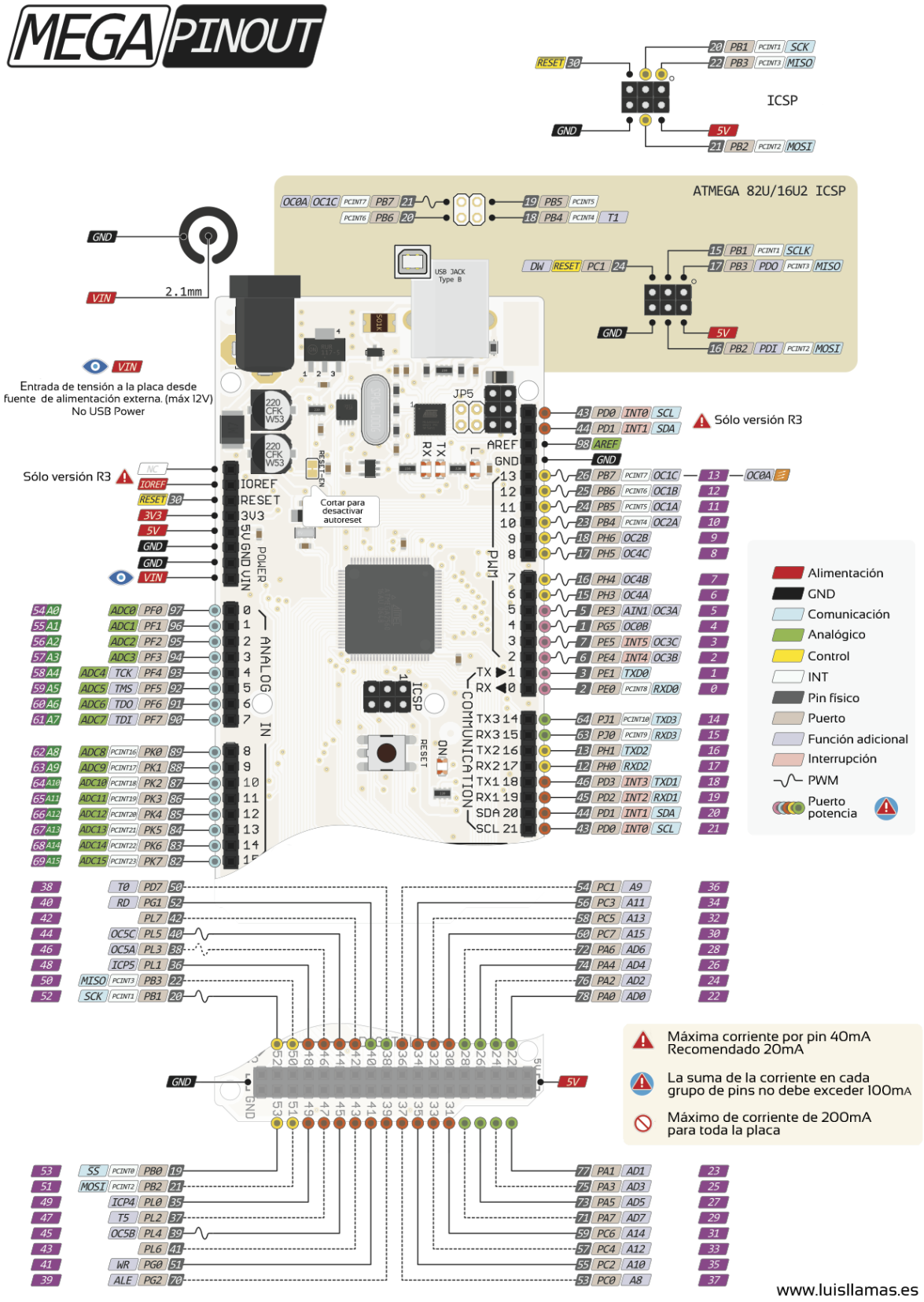


Figura 31: Pinout completo del Arduino Mega2560 que será utilizado como referencia a lo largo de este documento.

3.2.2. Sensores

Los sensores son dispositivos que tienen algún tipo de propiedad sensible a una magnitud del medio, al variar dicha magnitud, también lo hace de manera proporcional la propiedad asociada del dispositivo. En industria, esta definición se particulariza para objetos de carácter electrónico que son capaces de ver y transmitir la variación de variables físicas o químicas.

Existe un amplio abanico de posibilidades a la hora de caracterizar y dividir los diferentes sensores, según la magnitud que leen, según su rango de entrada o salida, según el tipo de salida, etc. Para el caso que ocupa este estudio, se hará hincapié en los tipos de salida que pueden existir y en la tipología de los sensores, concretando en los utilizados.

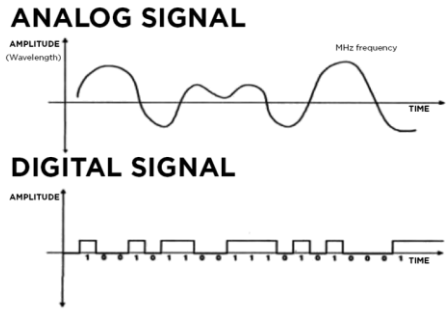


Figura 32: Diferencia entre señales digitales y analógicas

Según el tipo o formato de salida que ofrecen al leer una magnitud, los sensores se pueden clasificar en dos grupos principales: *analógicos* y *digitales.*

1. Los sensores analógicos son los que dan a su salida valores proporcionales continuos y sin codificar digitalmente, encargándose de hacer la transformación a digital el microprocesador encargado de leerlo. Por ejemplo, un *potenciómetro analógico* es capaz de convertir una posición angular variando el voltaje que cae entre sus pines, pero sin llegar a codificarlo digitalmente.
2. Los sensores digitales, al contrario que los analógicos, dan a su salida valores concretos, no continuos. Necesitando de una variación mínima en la magnitud de entrada para que la salida pase al valor siguiente. Este salto mínimo se denomina **resolución**. Por ejemplo, si la resolución de *termómetro digital* es de 0.2 ºC esto significa que, para este dispositivo variaciones de 0.1º C (de 30ºC a 30.1ºC por ejemplo) serían completamente invisibles. Mientras que, para uno analógico si tendría siempre reflejo en la salida al poder dar un rango de valores continuo.

A lo largo de los siguientes epígrafes y cuando se analice la viabilidad del proyecto, se comentarán las diferencias prácticas, ventajas e incovenientes de cada sensor y su posible aplicación o no para el caso que ocupa este documento.

Según la magnitud que física que midan la clasificación es diferente a la anterior y más amplia, siendo las tipologías más comunes las siguientes:

- Posición angular y lineal: *potenciómetro, encoder, sensor hall, etc.*
- Desplazamiento y deformación: *LVDT, galga extensiométrica, magnetoestrictivos, magnetorresistivos, etc.*
- Velocidad lineal y angular: *dinamo tacométrica, encoder, detector inductivo, etc.*
- Presión: *Membranas, pierzoeléctricos, manómetros digitales.*
- Temperatura: *termopar, RTD, termistor NTC, etc.*
- Sensores de Luz: *fotodiodo, fotorresistencia, fototransistor, célula fotoeléctrica, etc.*

Como se lleva desarrollando a lo largo de todo este documento, el sistema de adquisición de datos que se ha prototipado tiene la finalidad de ir sobre una motocicleta para analizar su comportamiento. Pero, es importante remarcar que, para este caso solo se pretende analizar el comportamiento del prototipo en carrera y, por ello, el número de sensores totales es menor.

Si se quisiera evolucionar el prototipo y estudiar parámetros como las deformaciones o flexiones del chasis, flujo del aire a través del carenado u otras propiedades interesantes para el desarrollo, el número y complejidad de sensores sería muy superior.

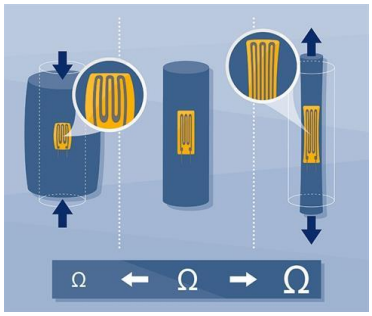


Figura 33: Esquema de funcionamiento de una galga extensiométrica.

3.2.3. Sensores utilizados en el proyecto

Los sensores utilizados se han elegido en base a diferentes factores, por su precio, por sus características, por haber trabajado con ellos en ocasiones anteriores, etc.

Para cada uno de ellos se tratará de detallar y explicar el motivo de su elección, sus características y las particularidades que se hayan podido encontrar a la hora de utilizarlos.

La lista de sensores aplicados al prototipo de telemetría realizado es:

- 2 sensores de temperatura LM35.
- 2 sensores de elongación potenciométricos para medir las extensiones de los amortiguadores.
- 1 GPS.
- 1 potenciómetro analógico.

A continuación, se pasa a caracterizar los sensores, añadir su documentación técnica y justificar su elección e inclusión en el sistema.

3.2.3.1 LM35

El LM35 es un sensor de temperatura desarrollado por Texas Instruments y destaca principalmente por ofrecer una salida que depende linealmente de la temperatura. Lo que, unido a que su calibración está realizada para grados Celsius, hace que se obtengan resultados de una elevada precisión preparados ya para su interpretación. Esto es una ventaja (para esta aplicación) frente a los sensores calibrados en Kelvin dado que, no habrá que hacer operaciones de resta de constantes que harían más dificultoso el tratamiento de los datos y podrían suponer una pérdida de precisión.

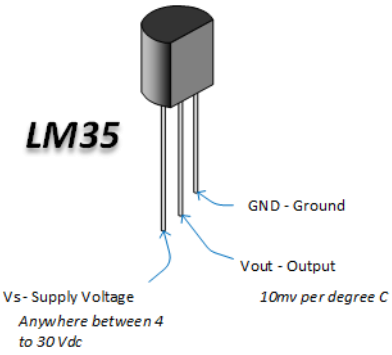


Figura 34: Encapsulado utilizado para el sensor LM35.

Otra de las múltiples ventajas que presenta este dispositivo es que, tiene una baja impedancia que hace que el dispositivo por si mismo no se caliente cuando está operando (sólo 0.1 ºC sin convección forzada).

Característica crucial dado que, un sensor de temperatura que se caliente por sí mismo podría hacer que las medidas se falsearan o perdieran precisión.

En términos de su rango de medida, puede depender de la versión del sensor que se esté utilizando, pero en el caso generalizado es capaz de medir desde -55ºC hasta 150ºC con una precisión de $\pm 1/4$ºC a temperatura ambiente y una de $\pm 3/4$ºC en su caso más extremo.

Además de todo lo anterior (y todo lo que se expondrá en adelante) es de valorar el bajo precio del sensor, lo que lo hacía muy propicio para la aplicación que se está tratando.

La relación de características técnicas completas (y por ende razones para la elección) del sensor LM35 es la siguiente:

- Calibrado directamente en Celsius
- Lineal con un escalado de +10 mV/ ºC
- Precisión garantizada de 0.5 ºC a temperatura ambiente (25 ºC)
- Rango de salida [-55 , 150] ºC
- Capaz de operar de 4V a 30V
- Absorbe una corriente de menso de 60 μA
- Bajo autocalentamiento, 0.08 ºC sin convección forzada.
- Baja impedancia de salida, 0.1Ω para una carga de 1 mA.

El encapsulado que se ha utilizado para el caso de este sensor es el que se puede apreciar en la *Figura 34*, que se relaciona con la *Figura 35* que es el esquema eléctrico interno:

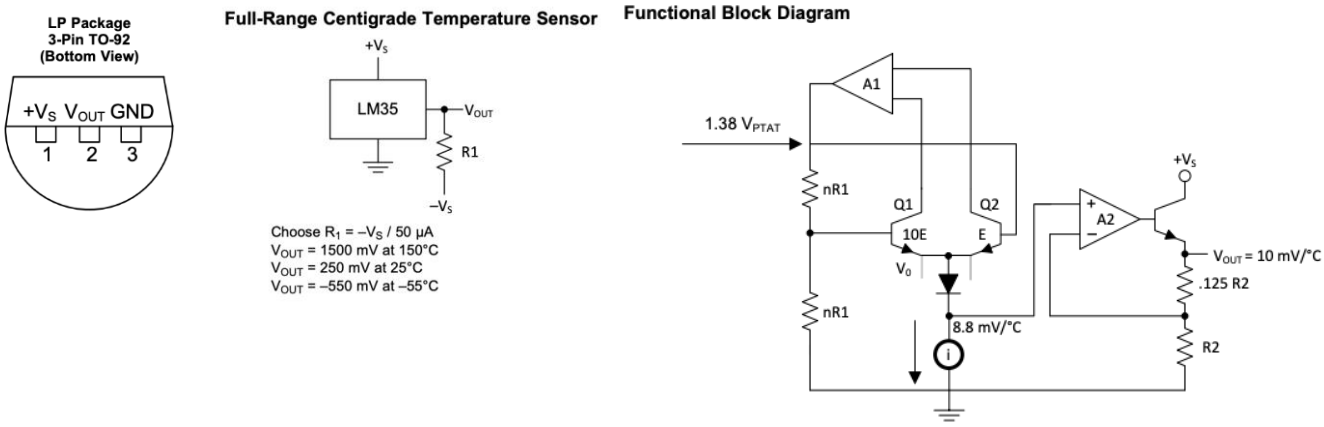


Figura 35: (de izquierda a derecha) Plano del Encapsulado del LM35, Esquema del termómetro para su rango completo, Diagramas de Bloques Funcional.

En las siguientes figuras (tratadas concretamente como *tablas*), se van a exponer algunas de las características tabuladas extraídas del *Datasheet*[5] del dispositivo. Estos datos, que además se explicarán para los casos más importantes, son los que fueron tomados como referencia a la hora de elegir el dispositivo.

En la *Figura 35* se puede apreciar el *pinout*[6] o diposición de pines del dispositivo, para el trabajo que se está realizando se ha utilizado el encapsulado que consta de 3 pines (aunque en la tabla llamada *Figura 36* venga caracterizado para la versión de encapsulado que cuenta con un mayor número de pines el comportamiento será el mismo).

Pin Functions

PIN					TYPE	DESCRIPTION
NAME	TO46	TO92	TO220	SO8		
V_{OUT}	2	2	3	1	O	Temperature Sensor Analog Output
N.C.	—	—	—	2	—	No Connection
	—	—	—	3		
GND	3	3	2	4	GROUND	Device ground pin, connect to power supply negative terminal
N.C.	—	—	—	5	—	No Connection
	—	—	—	6		
	—	—	—	7		
$+V_S$	1	1	1	8	POWER	Positive power supply pin

Figura 36: Tabla de Funciones de los Pines del LM35.

Una de las características más interesantes a la hora de elegir un dispositivo electrónico, puede ser las temperaturas límite que éste puede alcanzar. Sin embargo, hacer un estudio exhaustivo de esto no era necesario dado que las temperaturas máximas y mínimas medibles por el dispositivo (que siempre serán menores a las que soporta como se puede inferir de manera obvia) no se alcanzan en ningún caso. Siendo innecesario tener que cerciorarse de las temperaturas límite que puede soportar el sensor.

La propiedad que sí es necesario verificar es lo llamado *Application Curve* (Curva de Aplicación), en la que se puede apreciar la pérdida de precisión del sensor enfrentada a la temperatura.

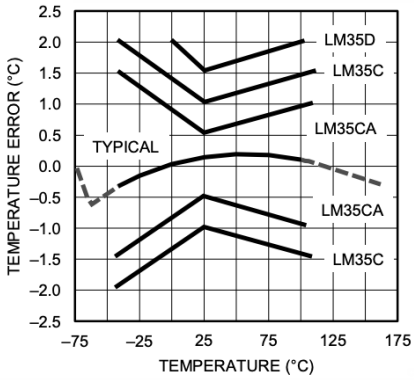


Figura 37: Curva de precisión del sensor en cuestión.

[5] Documento técnico referido a un dispositivo electrónico entregado por el fabricante, en el cual se pueden conocer todos los datos técnicos del componente en cuestión.

[6] Anglicismo utilizado para referirse a la caracterización de pines del dispositivo, es decir, qué función cumple cada pin o pata del dispositivo electrónico.

De la manera esperada, cuanto más se aleja la temperatura de ambiente o típica, mayor es la pérdida de precisión que sufre el sensor.

Justificación de la elección del sensor

Las razones que motivaron la elección de este sensor fueron de diversa índole.

En primer lugar, se observó el rango de temperaturas que debería poder registrar y, además, el posible fallo que pudiera tener en la medida. Estos dos sensores se utilizarían para medir temperatura del radiador de agua y del aceite, siendo la más crítica la del agua.

El valor típico de esta magnitud en una motocicleta en régimen de marcha puede estar entre los 85-95 ºC , comenzando a ser peligroso para el vehículo cuando alcanza valores superiores a los 100 ºC, de lo que se pueden inferir dos conclusiones: se necesita un sensor capaz de alcanzar y medir dicha temperatura y, no menos importante, que no tuviera demasiado error en dicha zona de su rango. Si se presta atención a la *Figura 38* se puede apreciar que, el sensor cumple con creces la condición de tener rango de medidas suficiente y que es la zona entre 75 ºC y 125 ºC la de mayor precisión, haciendo que el LM35 sea (desde estos dos puntos de vista) idóneo para la aplicación.

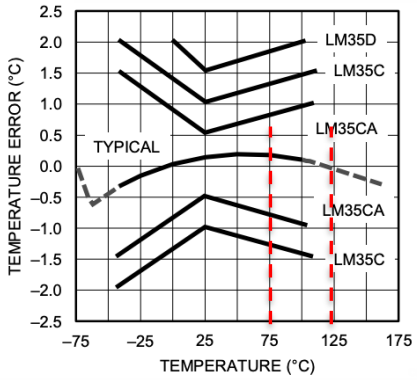


Figura 38: Curva de precisión del sensor con el rango de trabajo marcado.

En segundo lugar, se tuvieron en cuenta otras dos características del sensor, de menor importancia, pero no triviales: el precio del dispositivo que no era elevado (lo que, como se dijo al principio del documento, era un factor limitante) y que era un dispositivo conocido por haberlo usado previamente (lo que, relacionado con la falta de tiempo también especificada al principio, era también importante).

Como se dijo con anterioridad, el encapsulado que se ha elegido es el que tiene sólo 3 pines: Vcc, GND, y Vout, esto hace que su montaje en el sistema sea trivial. Se alimentó el pin Vcc con 5V, se unión el GND a la tierra general y se midió la salida Vout (utilizando un pin que tiene incorporado un convertidor A/D).

Utilizando una entrada analógica del Arduino, se ha medido la salida del sensor que, tras pasar por el convertidor analógico digital que tiene incorporado, se convierte a valores entre 0 y 1023. Tras realizar una pequeña calibración y cálculos, dicho valor numérico hay que transformarlo en una medida de grados comprensible para el ser humano, y para ello se ha usado la siguiente fórmula:

$$Temp\ (ºC) = \frac{5.0 \cdot Valor\ Leído \cdot 100}{1024}$$

Los pequeños detalles y aclaraciones que puedan surgir de su prueba real se verán en el apartado de programación de los códigos o *Software*, dado que fue al probar dichos programas cuando se detectaron pequeñas anomalías o particularidades del dispositivo.

3.2.3.2 Sensor de Elongación Potenciométrico

Figura 39: Sensor de elongación similar al aplicado en el sistema.

Este tipo de sensores (como la palabra “elongación” sugiere) están pensados para medir variaciones en la longitud de otro cuerpo.

De manera externa, cuentan con una forma similar a la de un pistón, con una varilla que entra y sale dentro de un cilindro. En sus extremos suelen contar con algún tipo de agarre o arandela que permita su fijación solidaria al cuerpo que se quiere sensorizar.

En la *Figura 39*, se puede ver el aspecto exterior del sensor, antes descrito, en uno exactamente igual al utilizado para el sistema que se está desarrollando.

Ya que se está haciendo referencia primero al aspecto físico, hay que remarcar que, la longitud máxima que puede medir este sensor es la misma que la de su émbolo. Es decir, si su tamaño completamente comprimido (o su émbolo dado que miden aproximadamente lo mismo) es de 15 cm, ésta será la máxima elongación que el sensor podrá medir.

Este es uno de los aspectos críticos y cruciales de este tipo de sensores, dado que, de comprobarse mal la medida del objeto que se quiere medir, el sensor sería completamente inútil para dicha apliación.

Las dos principales y más importantes partes externas, émbolo y cilindro, también tienen un papel claramente diferenciado (y en cierto sentido opuesto) de manera interna. El cilindro sostiene las dos resistencias de variación mecánica, que en el caso del sensor que se está usando no son bobinadas sino impresas. Este tipo de resistencias están realizadas con una pista de carbón o de *cermet*[7]. Por otro lado, la parte que hace de émbolo tiene un patín que va haciendo contacto en la pista o pistas del cilindro, variando la resistencia en función de la distancia que recorra el patín sobre la pista.

Figura 40: Sensor de suspensión de la Honda RCV213 de MotoGP, similar al utilizado en este trabajo.

Habiendo explicado ya el concepto general del tipo de transductor utilizado, se va a particularizar para el utilizado concretamente en este proyecto. El *sensor de posición lineal* que se ha utilizado es fabricado y distribuido por la empresa Ixthus, compañía de sensorística industrial y de automoción del Reino Unido. Las características concretas del dispositivo en cuestión son las siguientes:

- Longitud medible: 150 mm
- Linealidad del sensor: ±0.5 %
- Resistencia 6.0 kΩ ±20 %
- Velocidad máxima de desplazamiento: 10 m/s
- Ajuste de ± 2 mm

A diferencia del caso anterior en el que había mucha información del dispositivo y su funcionamiento, para este caso no se han podido encontrar curvas de comportamiento y linealidad. Al ser en formato numérico y listado en el que se dan las características del sensor.

El plano del sensor utilizado es el que se puede ver en la *Figura 41*.

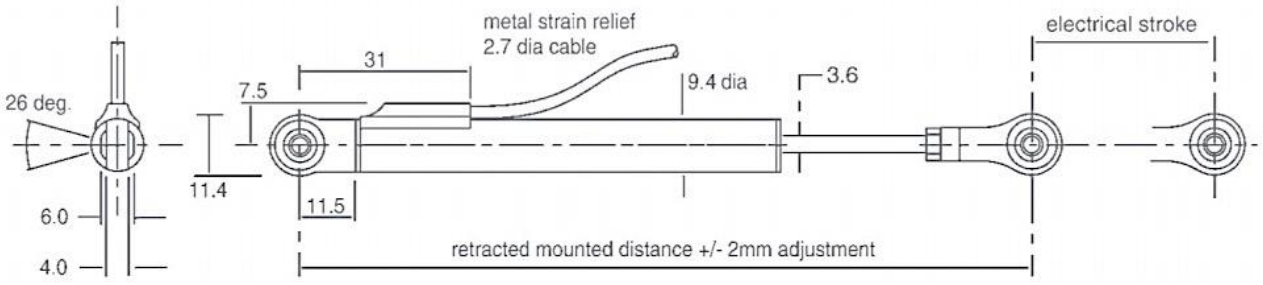


Figura 41: Plano del sensor de elongación potenciométrico que se está tratando.

[7] Material compuesto creado a partir de cerámicas y metales (de ahí su nombre) .

Justificación de la elección del sensor

La elección de este sensor no fue demasiado compleja de llevar a cabo ya que está pensado y diseñado para su aplicación en la automoción. Además de eso, su longitud, construcción y aislamiento a suciedad y polvo hacían de ésta la elección correcta a pesar de su elevado precio.

La única pega que suelen presentar este tipo de sensores, es que son de longitud fija y normalizada. Por lo que generalmente, hay que adaptar el cuerpo a medir al sensor más cercano en dimensiones de los que se puedan encontrar en el mercado. En el caso que nos ocupa, los compañeros del equipo de US-Racing, concretamente del departamento de Estructural, diseñaron unas cogidas para que este sensor fuera compatible con las suspensiones que llevaba el prototipo de motocicleta que se diseñó.

Al igual que en el caso anterior, será en el apartado de software dónde se especificará si este dispositivo en concreto al realizársele pruebas presentó curiosidades y características dignas de mención en el apartado práctico.

3.2.3.3 Global Positioning System (GPS)

Este es sin duda uno de los dispositivos (ya que estrictamente no encaja con la definición ni de GPS ni de transductor) de mayor complejidad e importancia del sistema que se está tratando de implementar. Es por ello que se le dedicará un esfuerzo especial y una mayor documentación que al resto de epígrafes de este apartado.

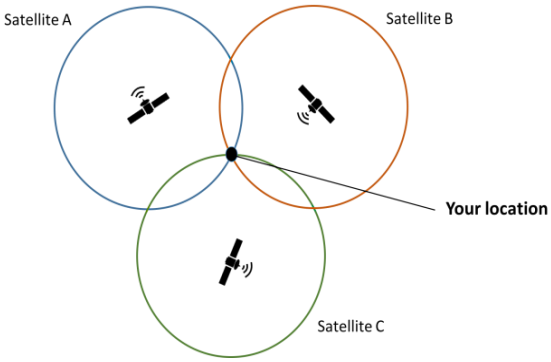


Figura 42: Diagrama del proceso de triangulación o trilateración llevado a cabo por los satélites GPS

El Sistema de Posicionamiento Global o GPS es un sistema que permite la localización de un dispositivo, persona o vehículo en la Tierra con precicisión (que puede variar desde metros hasta centímetros). Este sistema fue desarrollado e implementado por el Departamento de Defensa de los Estados Unidos, aunque hoy en día su uso excede con creces el militar, estando presente en todos los dispositivos móviles, vehículos e incluso en determinadas prendas de ropa.

El GPS funciona utilizando la trilateración[8] de 3 satélites con el dispositivo que se quiere localizar, al menos de manera teórica puesto que en la práctica se necestian generalmente 4 o más satélites. La red completa necesaria para poder triangular cualquier dispositivo en cualquier lugar del mundo y en cualquier momento, consta de un total de (cómo mínimo) 24 satélites necesarios para cubrir todo el globo terráqueo.

Existen otras versiones de este sistema, tanto de la Unión Europea (llamado Galileo) como de la Federación Rusa (llamado en este caso GLONASS).

El funcionamiento del GPS no es trivial y ocurre del siguiente modo:

1. Todos los satélites por separado indican un área circular en la que potencialmente se encuentra el objeto a detectar, siendo el centro de dicha zona la posición del satélite y el radio de la misma, la distancia del GPS al objeto.
2. Dónde aproximadamente se corten las tres circunferencias estará el objeto, como si de un sistema de tres ecuaciones con tres incógnitas se tratase o un problema de resolución gráfica. En ocasiones es posible que, las tres circunferencias no se corten en un solo punto, quedando acotada un "área

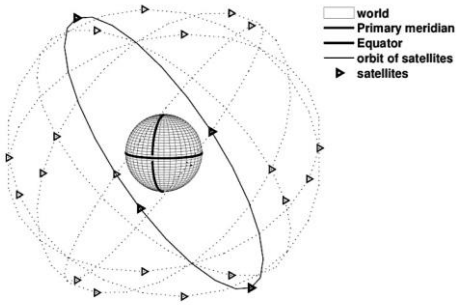


Figura 43: Esquema de la constelación de satélites encargados de la gestión del sistema GPS.

[8] Método matemático utilizado para, usando la geometría de triángulos, hallar posiciones relativas de objetos o cuerpos. Para obtener la posición relativa de un punto en un plano se necesitan, al menos, 3 puntos de referencia. 4 serán necesarias, como mínimo, para el caso tridimensional.

resultado", para resolver esto se utilizan el resto de datos (tando de los satélites como del receptor) y se comparan entre ellos. Observando similitudes y diferencias entre satélites y receptor se puede reponderar y calcular la posición exacta.

La mayoría de los sistemas de geolocalización suelen tener un error de algunos metros, motivado por los retrasos en la transferencia de datos, por falta de sincronización, climatología, etc. Pero, a pesar de que en dispositivos móviles se aprecian errores de hasta 50 metros, este sistema ha demostrado en las pruebas realizadas (mostradas luego en los epígrafes correspondientes a ello) precisiones de, en el mayor de los casos, 5 metros.

Figura 44: Visualización GPS en tiempo real en el Mundial de MotoGP

El sistema GPS en el mundo de la competición con vehículos de motor es de vital importancia dado que, permite encarar con exactitud los datos de los sensores que ilustran qué ocurre, con la zona del circuito dónde ocurren. En el pasado se intentaba interpretar la zona del circuito a través del cronómetro o de lo que ocurría con los sensores, pero era una forma un tanto imprecisa de saber qué estaba ocurriendo. Al poder encarar estos datos con la posición precisa y real del vehículo en el circuito, se puede observar mejor las anomalías o que eventos sucenden en un instante exacto (como la apertura del gas, la frenada, etc).

A medida que los sistemas electrónicos de la moticicleta fueron avanzando y siendo cada vez más complejos, la idea de poder programarlos según la zona del circuito en el que se encontrase el vehículo fue tomando forma. No pasó demasiado tiempo hasta que se llegó a los sistemas de gestión de potencia, amortiguación y demás que existen en la actualidad, que pueden ser ajustados de manera variable adoptando diferentes valores y comportamientos para cada curva de cada circuito.

Es por esto que, la función del GPS en el mundo de la competición (y concretamente en el de las motocicletas) se ha convertido en prioritaria y en uno de los sistemas más complejos que van en el vehículo.

Existen multitud de receptores GPS que pueden utilizarse con el sistema microprocesado Arduino, pero para la aplicación de una motocicleta de competición se eligió el ***Venus GPS*** de la empresa Sparkfun. Las razones de elección de este receptor (y no de otro) se tratarán más adelante.

El receptor GPS Venus de Sparkfun es el más ligero y rápido que fabrica la compañía y uno de los que más frecuencia de muestreo tiene que sea compatible con Arduino. Algunas de sus características técnicas son:

- Frecuencia de refresco de hasta 20Hz
- Tiempo de arranque en frío de 29 segundos
- Precisión de hasta 2.5 metros
- Capacidad para trabajar con antena pasiva o activa
- Soporta Datalogging a través de SPI
- Tensión de alimentación de 2.7 o 3.3 V
- Dimensiones: 2,9 x 18 cms

Figura 45: GPS Venus de Sparkfun

La manera que tiene el GPS de funcionar y de devolver los datos se verá más adelante en profundidad cuando se llegue al apartado del software que es dónde realmente se ha podido comprobar y ver el dispositivo en profundidad. Pero, a modo de introducción se podría decir que, el receptor GPS puede pasar los datos de posición, velocidad y hora (entre otros) encapsulados por el puerto serie en forma de vector con campos.

A pesar de ser éste un modo bastante eficiente de transmitir información, el ***receptor GPS Venus Sparkfun*** permite además que el dispositivo microcontrolado maestro haga una demanda de algún campo concreto, pasando sólo esta información y, por ende, haciendo el tratamiento de los datos mucho más rápido.

Justificación de la elección del sensor

Como se puede intuir al comprobar las características de este sensor, la elección de dicho módulo GPS vino motivada por su posibilidad de una frecuencia de muestreo de más de 1 Hz. Tras hacer una pequeña comparativa entre los modelos de GPS que eran compatibles con la placa microcontroladora que se iba a utilizar, se convino que, la mayoría tenían una frecuencia de muestreo demasiado lenta, tomando menos de una medida por segundo.

En términos de análisis del comportamiento de una motocicleta de competición, el GPS no es estrictamente necesario por lo que no existe un requisito de resolución tan crítico como en el caso de los demás sensores (al menos para la telemetría y caso que se está llevando a cabo).

Añadiendo que, a la hora de representar los datos se utilizan métodos de interpolación, una frecuencia de refresco o de adquisición de datos de 1 Hz (una vez por segundo) fue la que se estableció como mínimo asumible a conseguir. Por lo que la opción del ***Venus GPS de Sparkfun*** era, obviamente, la correcta desde este prisma.

Figura 46: Antena del GPS de la Honda RCV213 de MotoGP del mismo tipo que la utilizada en este proyecto.

A lo anterior hay que añadir que, el resto de características del sensor también eran adecuadas, como por ejemplo la precisión.

Si la frecuencia de refresco era quizás el requisito más importante, la precisión no se queda detrás si se tiene en cuenta la función y el uso de este receptor GPS. Generalmente, y como se ha dicho en epígrafes anteriores, la precisión de un GPS comercial normal como el de cualquier dispositivo móvil puede tener de error un par de decenas de metros. Esto es inasumible para una aplicación móvil a elevada velocidad como es el motociclismo de competición por lo que, que el receptor de Sparkfun elegido tuviera un error de solo ***2.5 metros*** lo hacía muy válido para este proyecto.

Otra de las características que lo hacían óptimo para la aplicación, es la de tener un conector SMA para la antena y ser compatible con las antenas GPS magnéticas utilizadas en automovilismo y motociclismo de competición (como se puede ver en la *Figura 46*).

3.2.3.4 Potenciómetro analógico rotatorio

Es un tipo de transductor muy similar al sensor de elongación potenciómetrico, salvo que, en lugar de medir una variación de longitud lineal, en este caso el potenciómetro mide variaciones angulares. Esta diferencia no es trivial porque, aunque sean ambos de carácter potenciométrico, la construcción y estructura de los sensores son completamente diferetentes.

El funcionamiento del potenciométro rotatorio es relativamente sencillo. El dispositivo tiene una resistencia variable mecánica con tres pines y un cursor, que actúa de divisor de tensión, teniendo siempre en el pin central un voltaje proporcional a la caída máxima fija que existe entre los pines de alimentación y tierra.

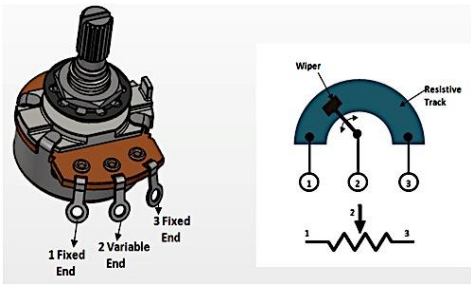


Figura 47: Potenciómetro analógico rotatorio utilizado en el proyecto.

Figura 48: Aceleración en el momento justo de puesta en marcha de la motocicleta.

En el caso del sensor de elongación potenciómetrico, las resistencias variables eran de tipo ***impresas***, con una pista construida en cermet y un patín que realizaba las funciones de pin selector. Pero para el caso del potenciómetro rotatorio, la resistencia interna que tiene es de tipo ***bobinada***, su estructura es un toroide arrollado de un hilo resistivo (en ocasiones fabricado en constatán[9]) y un patín, que hace las veces de cursor, moviéndose sobre dicho toroide.

Su uso más extendido es a modo de divisor de tensión para circuitería o como selector de audio, pero en este sistema hace la función de sensor de apertura del gas.

Para comprender de una manera más precisa el uso o aplicación que se le va a dar al potenciómetro elegido, hay que entender primero qué es la apertura del gas.

En una motocicleta, el mando del que dispone el piloto para controlar cuánta potencia quiere del motor es el puño derecho o puño del gas. Con un simple gesto, el humano que opera el vehículo puede elegir cuanta mezcla introduce la centralita o el carburador (si no hay inyección electrónica) en la cámara de admisión. El puño del gas, generalmente está operando la "palometa" o válvula de la admisión que lleva solidaria a ella un sensor que le dice a la centralita cuánto se está abriendo o, lo que es lo mismo, el porcentaje de todo el recorrido del puño del gas que ha elegido el piloto en ese momento.

Este sensor llamado TPS o *Throttle Sensor Position* es un sensor fundamental en la electrónica de la motocicleta y de tener un funcionamiento incorrecto o defectuoso el vehículo no funcionaría. Por ello, tomar la señal directamente de él e intervenir en el cableado de operación de la motocicleta no se contempló en ningún momento, haciéndose la telemetría independiente de éste. Pero, como se dijo arriba, el porcentaje de apertura del gas y de la admisión, son el mismo dato, por lo que utilizando un potenciómetro que girase solidario al puño del gas se obtendría la misma medida. Y es aquí dónde entra en función el ***potenciómetro analógico rotatorio*** que se está describiendo en este apartado.

Figura 50: Sensor TPS y su pinout.

Justificación de la elección del sensor

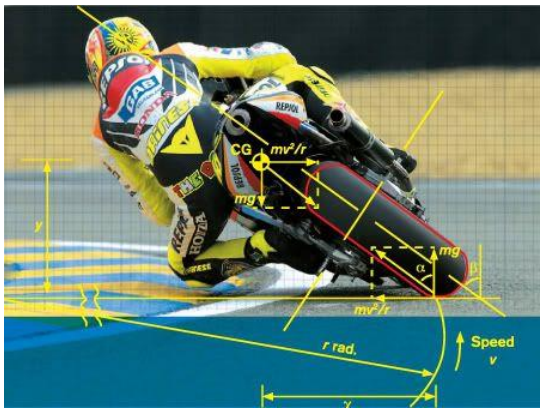


Figura 51: Inclinación con respecto a la vertical de una motocicleta cuando efectúa un giro.

Como ya se ha esbozado en el párrafo anterior, uno de los motivos de medir de manera alternativa la posición del acelerador era no intervenir el cableado de electrónica de motor, dada su criticidad.

Pero, existían diferentes estrategias y posibilidades para realizar esto girando todas en torno a la idea de medir una rotación, que es realmente la magnitud física que varía el operario para dar más velocidad a la motocicleta.

La primera opción y además la más intuitiva fue la de elegir un giroscopio para medir dicha rotación. El giroscopio mide los giros en los tres ejes del espacio. La idea era fijarlo de manera que la rotación del puño del gas se efectuase al igual que uno de sus tres ejes para así medirla de manera sencilla.

Pero lo que coincide con un eje del espacio en posición natural, es una composición de ejes cuando la motocicleta está inclinada que, añadido a que no siempre es el mismo grado de inclinación, complicaba exageradamente los cálculos.

[9] Aleación de un 55% de cobre y de un 45% de níquel caracterizado por tener, para un amplio rango de temperaturas, una resistencia térmica constante.

Por todo lo anterior, la opción de utilizar un giroscopio para medir el porcentaje de apertura del gas fue descartada.

Una vez se vió la primera opción como inviable, se decidió utilizar el potenciómetro como opción final, dada su simplicidad en la operación y en la medida. Además de eso, su precio (uno de los factores limitantes) no es elevado y funciona con los niveles de voltaje que la placa de arduino es capaz de dar a su salida para la alimentación de periféricos. Todo esto, y su elevada linealidad, hacen del sensor la opción a priori adecuada para la medida que se quería realizar.

3.3 Dashboard o Display

Figura 52: Dashboard de la Honda de MotoGP de 2003 (sin display LCD aún).

Terminado ya el apartado de los diferentes sensores que el proyecto va a llevar, llega el turno de detallar una de las partes de mayor complejidad del mismo: el display gráfico ***on board***[10] de los datos obtenidos.

La inclusión de *dashboards* o tableros de instrumentos en las motocicletas de competición es un fenómeno relativamente reciente, a diferencia de las motos de calle, en las que se torna esencial para el piloto poder disponer de manera visual de los datos para la conducción. Sin embargo, en el mundo de la máxima competición, hasta principios de los 2000 se podían ver tacómetros y cuadros de mandos analógicos, no contando aún con pantallas o displays para la visión de los datos.

En el caso que ocupa a este trabajo se decidió que, ya que se está tratando de realizar un primer prototipo de sistema de telemetría, se debería tratar de añadir un display para evaluar su viabilidad una vez construido el prototipo completo.

La elección del tipo de display o del sistema microprocesado que haría las veces de GPU[11] para gestionar losdatos, gráficos y códigos, no fue para nada trivial.

Por un lado, se tenía la posibilidad de utilizar un sistema comercial ya existente en un formato ***plug-and-play***[12] como los desarrollados por la empresa AiM (como se detalló en el apartado *Sistemas Actuales de Telemetría en el Motociclismo de Competición* de este trabajo) descartada por su elevado precio.

Pero, finalmente, se decidió acometer la realización de un sistema propio, aunque fuera de una manera más simple que en el caso del sistema comercial, su precio no sería ni de una quinta parte, lo que lo haría cuanto menos realizable.

La primera tarea para la elección y caracterización del sistema gráfico fue dimensionarlo de manera correcta y eficiente. La mayoría de motocicletas de competición de la actualidad utilizan un display que funciona con la propia centralita de la motocicleta y, generalmente, no disponen de colores. Pero, al igual que en el caso de la elección del potenciómetro, una de las prioridades era proyectar el sistema de telemetría de manera independiente del resto de cableados y dispositivos electrónicos de la motocicleta, no comprometiendo así ningún otro sistema en caso de fallo o error.

[10] Anglicismo utilizado para hacer referencia a un dispositivo o sistema especialmente pensado para funcionar exclusivamente en el vehículo. Por ejemplo: una cámara subjetiva en un coche de fórmula 1.
[11] Unidad de Procesamiento Gráfico.
[12] Término utilizado para definir aquellos sistemas que tras iniciarse, funcionan directamente sin previa preparación o instalación.

3.3.1. Elección del display y del sistema de gestión de los gráficos

En primera instancia se pensó en utilizar una pantalla de un tamaño aproximado de 5" para que el piloto tuviera una mayor claridad de visión de los parámetros representados. Pero, por temas de espacio se comprobó que, este display sería demasiado grande para las dimensiones disponibles entre la cúpula y la tija de dirección de la motocicleta por lo que se rebajó a 3.5".

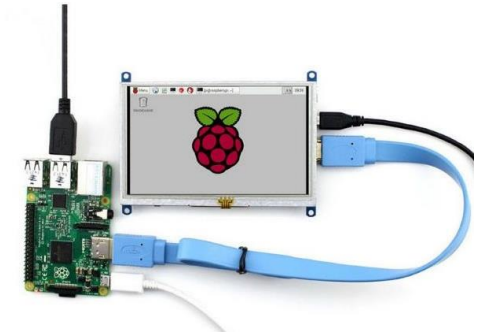


Figura 53: Display de 7" conectado a una placa Raspberry Pi.

Teniendo elegidas las dimensiones de la pantalla que se iba a incorporar al sistema, la unidad encargada de gestionar los gráficos de la misma debía estar dimensionada en consonancia. Se pensó que, para la gestión gráfica del display de 5", se utilizaría una placa microprocesada llamada Raspberry Pi que goza de soporte específico y aplicaciones para la generación de gráficos.

Sin embargo, para una pantalla de 3.2" dicha placa se antoja bastante sobredimensionada e innecesaria, teniendo en cuenta además que es más costosa que otras placas como la de Arduino, que lleva un lenguaje y una programación completamente diferentes y que tiene un consumo de potencia mucho más elevado. Todo lo anterior hizo que se valorase la opción de incorporar como GPU para esta pantalla de 3.2" un Arduino Mega, similar al que hace la aduqisición de datos, pero con la única tarea de representar los datos por pantalla.

Tras una breve investigación y pruebas (que se detallarán en el apartado de software) se concluyó que 3.5" era lo máximo que podía manejar un Arduino Mega ofreciendo un rendimiento suficiente, por lo que un display de 3.2" como el que se barajaba debía funcionar, al menos, correctamente. Al no tener soporte para gráficos, toda la programación de los gráficos por pantalla por medio de Arduino se tuvo que hacer a mano, en un formato ***píxel a píxel*** a través de bucles y utilizando formas matriciales.

Cómo se dijo al principio de este documento, uno de los factores limitantes del proyecto, era el económico. Esto, para otros componentes del mismo era meramente anecdótico, pero, para el caso de elegir un display que instalar en la motocicleta era algo crítico dado que con cierta facilidad se encontraban precios elevados.

Todos los TFT LCD que se podían encontrar disponían de entrada táctil que, a pesar de no ser útil para el proyecto que se está utilizando, no era posible encontrarlos sin dicha característica. Por lo que, el único parámetro que, si era diferencial en términos de precio, era la resolución existiendo prácticamente dos grupos (dado que las de muy baja resolución no se van a tratar):

1. TFT LCDs con resolución y gama de colores media.
2. TFT LCDs con Alta definición y gama de colores completa.

Determinar que opción era la óptima no fue tarea sencilla. Primero se sopesó el precio que, era del orden del doble para las pantallas con mejores características (alta definición, mayor gama de colores, etc) lo que era una desventaja grande dado lo limitante del factor presupuestario.

Por otro lado, que la pantalla sea HD o no, no era algo realmente necesario (por no decir casi inútil) dado que su función es mostrar datos al piloto en régimen de marcha.

Para terminar con esta comparativa entre las diferentes opciones de displays, han de tenerse en cuenta dos implicaciones remarcadas en los párrafos anteriores. Primero, el código de Arduino utiliza una lógica de bucles "dibujando" píxel a píxel. Segundo, el Arduino Mega2560 podía manejar una pantalla de aproximadamente 3.5" pero llegando, para según que acciones, a su límite de rendimiento. Por lo que, elegir una pantalla con un número mayor de píxeles (aunque tuviese el mismo tamaño físicamente hablando) podía comprometer este rendimiento.

Por lo que, como se puede extraer de la comparativa realizada, la balanza se decidió a favor de la tipología de pantalla de menor resolución, pero, dentro de esta familia existían a su vez más opciones.

Una de las pantallas encontradas que eran compatibles con el dispositivo a utilizar eran las de la empresa *Seeed Studio*, concretamente la de 3.2" TFT. Una pantalla que cuenta de serie con opciones interesantes como control gestual, reproducción de vídeo o teclado en pantalla. Opciones que no serían de utilidad en un proyecto como éste y que, además, elevaban mucho su precio.

Por lo que, una vez descartado ese display (y los que se encontraron similares) se decidió buscar uno más económico y que permitiese una programación de más bajo nivel para así poder adaptarlo y programarlo de una manera más exacta al propósito que iba a cumplir.

Tras una búsqueda algo más exhaustiva forzada por las restricciones que se han puesto sobre la pantalla a utilizar, se llegó a la conclusión de que una de las mejores opciones era la de usar la pantalla TFT 3.2" Open-Smart. Dicho display se eligió porque cumplía las restricciones de precio y de tipo de programación (bajo nivel) que se requerían y, además, podría usarse un Arduino Mega a modo de GPU (cómo se indicó más arriba en esta memoria).

3.3.2. Display elegido: Open-Smart TFT 3.2"

Figura 54: Pantalla que se ha utilizado como display del proyecto.

El display de la *Figura 54* llamado *Open-Smart TFT 3.2"* fue finalmente elegido porque, como se explicó en párrafos anteriores, era el que cumplía de una manera más completa las restricciones expresadas para este dispositivo.

Hay que destacar que, lo que se puede ver realmente en la *Figura 55* no es sólo el display, sino que, además éste viene en una placa de extensión que permite un conexionado mucho más sencillo.
Dicha placa de extensión en la que aparece integrado el display cuenta, además, con un sensor de temperatura, sensores de corriente e intensidad y ranura para tarjeta SD entre otros. En cuanto a la pantalla en sí misma, tiene una tecnología táctil (que no se va a utilizar) de carácter resistivo, un tamaño de 3.2" y una resolución de 340x400 píxeles. Este último dato es relativamente restrictivo a la hora de comprobar el rendimiento del Arduino como GPU dado que, al pintar la pantalla píxel a píxel, la resolución estará íntimamente relacionada con el número de operaciones a realizar.

Algunas otras especificaciones técnicas son:

- Dimensiones de la placa: 90 x 53 mm
- Dimensiones de la parte visible de la pantalla: 70 x 43 mm
- Interfaz: 8 pines de datos, 5 pines de control
- Dos modos de trabajo en cuanto a colores: 16 y 18 bits.
- Controlador gráfico: HX8352B

En la *Figura 55* se puede apreciar el reverso de la placa en la que se han destacado algunos de los componentes anteriormente listados.

Figura 55: Trasera de la placa en la que viene la pantalla con sus periféricos necesarios.

Controlador HX8352B

El controlador gráfico HX8352B es el implementado en la placa ya antes comentada para controlar esta pantalla. Dicho driver gráfico ha sido diseñado específicamente para ser una solución implementada en un solo chip a varias de las necesidades que puede tener el display instalado. A saber, el HX8352B hace las veces de: controlador de puerta, de fuente y de fuente de alimentación para la pantalla.

Una de las grades ventajas que tiene este controlador es que puede ser operado a bajo voltaje, contando dentro del mismo con amplificadores que son los encargados de elevar dicha tensión a la que demanda el cristal líquido del display.

Para que esto no se traduzca en un incremento de las pérdidas, este driver lleva integradas unas funciones de *software* de control para minimizar el consumo de energía. Todo lo anterior está ligado al uso que se le va a dar al display en cuestión, dado que por su tamaño serán habituales aplicaciones portátiles en las que un consumo de energía óptimo es necesario.

El dispositivo que se está desarrollando en este epígrafe del trabajo tiene como esquemático el mostrado en la *Figura 56*.

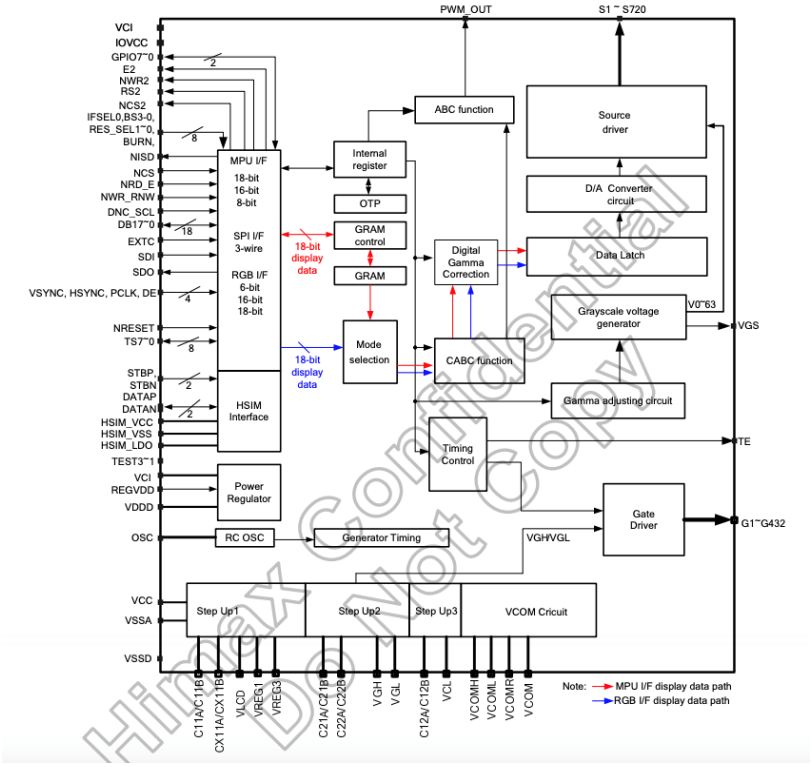


Figura 56: Esquemático del driver gráfico HX8352B.

Explicar al detalle este dispositivo excede lo que, a priori, es el objetivo de este trabajo, pero caben destacar ciertas características que son importantes a la hora de utilizar dicho componente electrónico. Por un lado, destacar la versatilidad del producto, pudiendo trabajar con displays diferentes y en modos de funcionamiento diferentes, permitiendo adaptarse a diferentes pantallas según los siguientes criterios:

- Resolución: Este *driver* gráfico soporta de manera óptima, 3 resoluciones diferentes, aunque pueda utilizarse de manera distinta.
- Modo de color del display: el driver es capaz de modificar la cantidad de colores que es capaz de representar en el display modificando el *tamaño de palabra de bits*[13] destinada a este propósito. Generalmente, la paleta cromática se suele definir en torno a tres colores rojo, verde y azul (de ahí el RGB: *Red, Green y Blue*) pudiéndose definir cualquier color solo con expresar la cantidad de cada uno de los anteriores presente en el mismo. De este modo, cuantos más bits se tenga para expresar los colores anteriores, más divisiones posibles entre el blanco y el negro y, por tanto, más colores. El driver HX8352B es capaz de trabajar con los siguientes números de colores y, por ende, tamaños de palabras:
    - Modo de color completo: para este modo se utilizan 18 bits, 6 por componente del color y se pueden representar un total de 262 mil colores diferentes.
    - Modo de color reducido: este modo tiene a su vez dos variantes, una de 12 bits (4 por

[13] Dícese del número de bits reservados para un propósito como un conjunto. En este caso, para la codificación cromática RGB se dividirá la palabra en un número de bits concreto para cada color.

componente) y que otorga una cantidad de 4096 colores diferentes. Y otra de 16 bits (6 para la componente verde y 5 para las componentes roja y azul) que permite una variedad de 65 mil colores.

Además de lo anterior, el driver permite modificar también otros tipos de valores para adaptarlos al display en cuestión que se esté usando. Por ejemplo, las tensiones de alimentación y puerta que puede dar el driver a su salida no son fijas, están en un rango acotado dentro del cual se puede elegir la deseada. Así mismo, es posible modificar también el tamaño de bits del bus de comunicaciones o el número de bits destinados a la interfaz RGB.
Una vez terminado el apartado del display, se pasa al apartado del software que será de los más extensos de este trabajo.

3.4 Software del Sistema O*n-Board*

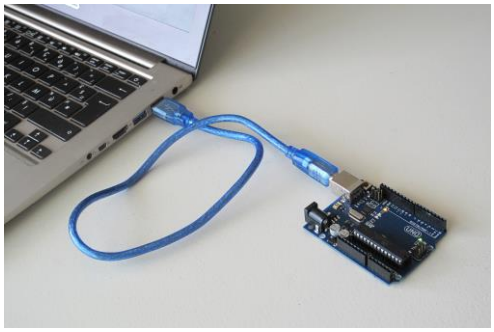


Figura 57: Arduino conectado a un PC por medio de un USB.

Lo primero que habría que distinguir antes de comenzar el desarrollo del trabajo realizado en este apartado son los dos apartados de software que tiene el proyecto. Por un lado, se puede categorizar como software la programación necesaria para la adquisición de datos de los sensores, lo anterior (como conjunto) va instalado todo en la motocicleta persé. Mientras que, por otro lado, también se considera software la programación a realizar para la representación y tratamiento de los datos una vez la motocicleta ha llegado al garaje. Ambos se tratarán de manera diferenciada a lo largo de esta memoria.

El sistema elegido, como se ha comentado en epígrafes anteriores, es uno basado en la tecnología Arduino. Este tipo de microprocesadores se pueden encontrar de muchas características y tamaños diferentes, pero todos comparten lenguaje de programación, lo cual es una gran ventaja.

El lenguaje de Arduino está basado en el lenguaje C++ que, a su vez, es una extensión del lenguaje C. Tanto es así que, a la hora de programar una de las placas de Arduino se pueden utilizar sentencias de C/C++ que el microprocesador reconocerá sin mayor problema. De hecho, para determinadas operaciones será óptimo (llegando incluso a ser la única opción a veces) utilizar directamente C o C++.

Para poner en contexto la programación que se ha llevado a cabo y que se detallará más adelante, se va a tratar brevemente el origen y las características especiales de estos lenguajes de programación.

C es un lenguaje creado entre los años 1969 y 1972 por Dennis Ritchie, como evolución de un lenguaje anterior existente llamado: B. Es un lenguaje orientado a la programación de sistemas, no tanto para crear aplicaciones, aunque pueda utilizarse para tal fin. Es considerado un lenguaje de medio nivel dado que, es posible utilizar estructuras e instancias similares a los lenguajes de más alto nivel, pero también, permite un control a muy bajo nivel llegando incluso a acceder directamente a memoria, por ejemplo.

Es el lenguaje más extendido del mundo, dado que tiene un núcleo del lenguaje muy simple pero que puede ser expandido y mejorado añadiendo funcionalidades a través del uso de bibliotecas, como puede ser el uso de funciones matemáticas, entre otros. Es un lenguaje muy eficiente dado que, puede llegarse (mediante el uso de punteros) a escribirse en las direcciones concretas de memoria deseadas, haciendo implementaciones óptimas. Concretamente, fue específicamente pensado para ser un lenguaje de programación pensado para sistemas portátiles.

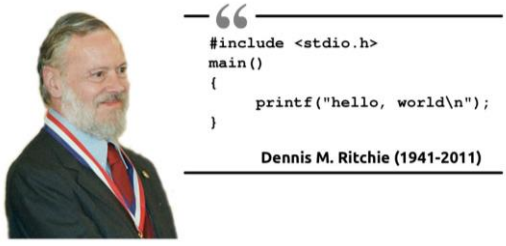


Figura 58: Denis Ritchie con la sentencia más famosa de su código.

Por todo lo anterior referido al lenguaje C, es posible entender mejor porque es el modo de programación que Arduino elige para sus sistemas microprocesados. Encaja a la perfección en todos los sentidos, desde la sencillez de implementación que permite una mayor versatilidad a la hora de crear proyectos, hasta su poca

carga computacional que lo hace óptimo para sistemas portátiles de baja potencia.

A partir de este punto, y tal como se realizó en el apartado de *Hardware*, se van a separar los diferentes softwares desarrollados según su función y creación. No sólo se va a representar y explicar el producto final, sino también algunos de los pasos intermedios con el objetivo de dejar patente el camino de desarrollo seguido.

3.4.1. Software de pantalla

3.4.1.1 Introducción al software de pantalla

El software que se encarga de representar gráficamente los datos por pantalla ha sido, probablemente, uno de los más complejos de desarrollar. Y se debe a que, para la pantalla que se está utilizando, el Arduino Mega llega al límite de sus capacidades para según que acciones.

La primera tarea a realizar para desarrollar y probar el software de pantalla fue, como en casi cualquier proyecto de esta clase, encontrar y configurar las bibliotecas y librerías que funcionasen con el display en cuestión. Esto no fue tarea fácil dado que, al ser una pantalla de ajustado precio, no era de las más ampliamente utilizadas por lo que cualquier tipo de recurso que se necesitase era de difícil acceso. A pesar de lo anterior, retocando algunas de las librerías que el fabricante proporciona y depurando algunos errores que contenían, la pantalla pasó a ser completamente operativa.

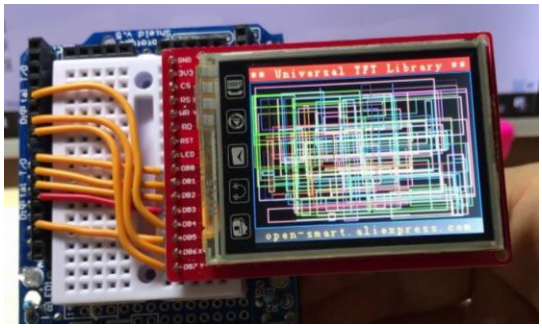


Figura 59: Otro de los modelos de pantalla creados por Open-Smart.

Conseguida la puesta en marcha de la pantalla, el objetivo inmediatamente siguiente era realizar un diseño de display que fuera funcional y que mostrase la información de manera clara en la pantalla. A pesar de parecer una tarea sencilla, fue a partir de este punto dónde comenzaron a surgir problemas en la programación y rendimiento de la pantalla. La idea original era, representar los datos numéricos en pantalla y, además, una barra que mostrase el porcentaje de apertura del acelerador (dato obtenido de la lectura de un potenciómetro) entre otros, siendo un factor vital la *frecuencia de refresco*.

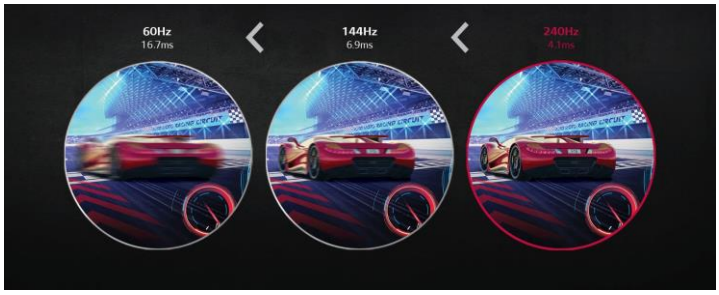


Figura 60: Ejemplo de la mejora que supone una mayor frecuencia de muestreo.

La *frecuencia de refresco* se puede definir como el número de veces por segundo que se actualiza una pantalla. Para el caso que ocupa a este estudio, la definición anterior puede ser engañosa o poco precisa porque, tal y como se verá a continuación, es posible no refrescar la pantalla completa y refrescar solo una parte. En cualquier caso, entiéndase como el número de "imágenes" válidas para el espectador que representa la pantalla por segundo. Este parámetro es directamente proporcional a la fluidez con la que el humano capta los movimientos que pueda haber en pantalla. Por lo que, se prestó la máxima atención a este factor a la hora de intentar obtener el máximo rendimiento a la pantalla utilizada.

La idea de partida era utilizar una función incluida en las bilbiotecas de la pantalla destinada a dibujar en la misma cualquier mapa de bits. Primero, se dibujaría el fondo, acto seguido se le dibujaría encima la información deseada y pasados unos instantes, se borraría entera la pantalla de nuevo y vuelta a empezar. Pero, sin embargo, pronto se pudo ver que este enfoque de representación "absoluta" no era posible y se tuvo que pasar a una estrategia "relativa", es decir, dibujar algo y borrar solamente ese algo (o una porción del mismo).

Una vez aclarado lo anterior, es una implicación directa que, para la primera estrategia de dibujado por pantalla la frecuencia de refresco era menor que para la segunda. De hecho, para el primer caso, el Arduino demoraba varios segundos en representar una imagen completa, mientras que en el segundo caso era capaz de representar mínimo un par por segundo. Pero la opción utilizada finalmente, la relativa, tenía algunas

desventajas con respecto a la primera, principalmente dos: no tener una frecuencia de refresco estable y posibles imprecisiones.

La varianza mostrada en términos de frecuencia de refresco se debía a que, al solo eliminar una parte de lo dibujado, dependía de cuánto y como se realizase dicha eliminación. Aún así fue la opción utilizada finalmente al presentar un rendimiento bastante adecuado, en lugar del rendimiento tan bajo que presentaba el modelo de implementación absoluto.

Antes de abordar la explicación del funcionamiento completo de la pantalla y su resultado, se ha de detallar el proceso completo, las diferentes pruebas realizadas y los diferentes pasos hasta llegar al punto final.

3.4.1.2 Primera iteración: Dibujar Fondo

Como se especifica en párrafos anteriores, la idea era representar un fondo sobre el que imprimir los datos leídos por diferentes sensores. Para ello, se utilizó como base una de las funciones de la biblioteca de la pantalla, aunque tuvo que pasar por un proceso de depurado, adaptación y reprogramación de algunas de sus partes que no eran compatibles con el modelo utilizado de display.

El código resultante se va a explicar a continuación, fragmentado para permitir una mejor explicación y comprensión, pero completo.

Como en cualquier programación basada en Arduino, el primer paso es la inclusión de las librerías necesarias para incluir las funciones que requiera el hardware en cuestión para su funcionamiento. Esto se realiza con la sentencia: ***#include*** cómo se puede ver a continuación.

```
#include <SPI.h>
#include <SD.h>
#include <Adafruit_GFX.h>
#include <MCUFRIEND_kbv.h>
```

Acto seguido, se presentan las declaraciones de algunas variables que serán usadas a lo largo del código y la sentencia ***#define*** se establece una regla de sustitución, es decir, el primer elemento pasará a ser el segundo en términos de programación. De este modo, se facilita la interpretación y programación pasando de datos ilegibles a datos comprensibles para el programador. En este caso, se asocian números en hexadecimal que serán colores, a su nombre en inglés. Así, cada vez que aparezca la palabra “CYAN” se sustituirá internamente por el número que corresponda, en este caso: 0x07FF. Se establecen también dos variables que definen el ancho y alto del display y se les da valor de inicio.

```
MCUFRIEND_kbv tft;
#define BLACK   0x0000
#define BLUE    0x001F
#define RED     0xF800
#define GREEN   0x07E0
#define CYAN    0x07FF
#define MAGENTA 0xF81F
#define YELLOW  0xFFE0
#define WHITE   0xFFFF
#define NAVY        0x000F
#define DARKGREEN   0x03E0
#define DARKCYAN    0x03EF
#define MAROON      0x7800
#define PURPLE      0x780F
#define OLIVE       0x7BE0
#define LIGHTGREY   0xC618
#define DARKGREY    0x7BEF
#define ORANGE      0xFD20
#define GREENYELLOW 0xAFE5
#define PINK        0xF81F
int16_t largo=400;
uint16_t ancho=240;
#define SD_CS 5
#define BUFFPIXEL 2
```

Con la función utilizada a continuación se establece el directorio del que se leerá en la tarjeta SD, en este caso, el directorio raíz. Y justo después, se declaran algunas variables más, como la cadena de caracteres que

contendrá el nombre del archivo a buscar en la propia tarjeta.

```
File root;
char namebuf[32] = "backg.bmp";
int pathlen;
uint8_t spi_save;
```

Los programas de Arduino tienen generalmente dos bucles principales, uno llamado *setup,* que se ejecuta una sola vez, y otro llamado *loop* que se ejecuta ilimitadas veces de manera cíclica. Este trozo de código es la primera parte del setup que se utilizará para dos fines. Por un lado, para configurar e inicializar los procesos necesarios para el manejo de la pantalla. Por otro, para pintar el fondo de la pantalla que, a pesar de no ser configuración de la misma, es un proceso que debe realizarse una sola vez.

Este bucle inicializa el driver de la pantalla y la pinta de negro, para comenzar. Seguidamente, se pone a 1 un *flag* que sirve para indicar el comienzo de la escritura de una imagen mapa de bits y se coloca la orientación de la pantalla con ***tft.setRotation*.** Se llama a la función ***bmpDraw*** (explicada más adelante) que imprime la imagen por pantalla, y se vuelve a 0 el mismo *flag* anterior. Por último, se vuelve a poner la misma rotación de pantalla (***tft.setRotation***) y, colocando el cursor (***tft.setCursor***) y eligiendo color y tamaño de texto (***tft.TextColor*** y ***tft.TextSize*** respectivamente), se pinta el título de la pantalla con ***tft.print***.

```
void setup() {
    tft.begin(0x65);//to enable HX8352B driver code
    tft.fillScreen(0x0000);
    bool good = SD.begin(SD_CS);
    if (!good) {
        Serial.print(F("cannot start SD"));
        while (1);
    }

   tft.flag_write_bmp = 1;
   tft.setRotation(1);
   bmpDraw("backg.bmp", 0, 0);
   tft.flag_write_bmp = 0;
   tft.setRotation(1);
   tft.setCursor(150,15);
   tft.setTextColor(LIGHTGREY);
   tft.setTextSize(3);
   tft.print("DASHBOARD");
}
```

El siguiente bucle, el denominado *loop* es característico de Arduino y se repite de manera infinita. En este caso, lo único que se representa es un texto para comprobar que el código ha alcanzado este punto después de pintar el fondo.

```
void loop() {
  tft.setCursor(150,150);
  tft.setTextColor(RED);
  tft.print("TEST OK");
}
```

La función que se va a mostrar a continuación es la que dibuja la imagen persé, al ser muy extensa y de una mayor complejidad que las anteriores, se van a ir poniendo por separado sus diferentes partes para una mayor claridad de explicación.

Este es el comienzo de la función de dibujado de mapa de bits, en esta parte del código se definen las variables que se van a utilizar más adelante.

```
void bmpDraw(char *filename, int x, int y) {
   File   bmpFile;
   int    bmpWidth, bmpHeight;   // Ancho y alto en píxeles
   uint8_t  bmpDepth;        // profundidad de bit (debe ser 24)
   uint32_t bmpImageoffset;      // Comienzo de los datos de imagen dentro del archivo
   uint32_t rowSize;
   uint8_t  sdbuffer[3*BUFFPIXEL]; // píxel en buffer (R+G+B por píxel)
   uint16_t lcdbuffer[BUFFPIXEL];  // pixel fuera de buffer (16-bit por píxel)
   uint8_t  buffidx = sizeof(sdbuffer); // Posición actual en sdbuffer
   boolean  goodBmp = false;     // Se pone a "true" cuando la cabecera del bmp es
correcta
```

```
boolean  flip  = true;       // El BMP está guardado desde el fondo al top
int    w, h, row, col;
uint8_t  r, g, b;
uint32_t pos = 0, startTime = millis();
uint8_t  lcdidx = 0;
boolean  first = true;
```

La siguiente función mira primero si se exceden las dimensiones, y que comprueba si existe o no un archivo en la tarjeta SD con el nombre especificado previamente. En caso de no existir un archivo con el nombre especificado o si se exceden los límites sale del bucle.

```
if((x >= tft.width()) || (y >= tft.height())) return;
 // Abre el archivo pedido de la SD
 SPCR = spi_save;
 if ((bmpFile = SD.open(filename)) == NULL) {
 return;
 }
```

Para poner en contexto el siguiente fragmento del código, primero ha de especificarse qué es la cabecera de un archivo mapa de bits. La cabecera de un BMP es la parte inicial del archivo que proporciona datos sobre la imagen de por sí como número de colores, orden de los datos (de arriba abajo o viceversa).

Además, se obtienen otro tipo de datos como el número de bits por píxel de imagen, si el formato de archivo es soportado, cuando empiezan los datos de imagen, etc.

A lo largo del tramo de código siguiente se analiza la cabecera del BMP y, como se ha explicado en los párrafos anteriores, las características de los datos de imagen.

```
// analiza la cabecera del BMP
   if(read16(bmpFile) == 0x4D42) {
   read32(bmpFile);
   (void)read32(bmpFile); // Lee e ignora bits de info del creador
   bmpImageoffset = read32(bmpFile); // Comienza los datos de imagen
   read32(bmpFile);
   bmpWidth  = read32(bmpFile);
   bmpHeight = read32(bmpFile);
   if(read16(bmpFile) == 1) { // # debe ser '1'
     bmpDepth = read16(bmpFile); // bits por píxel
     if((bmpDepth == 24) && (read32(bmpFile) == 0)) { // 0 = sin compresión
     goodBmp = true; // Comprueba si el formato de bmp es soportado
    rowSize = (bmpWidth * 3 + 3) & ~3;
     // Si bmpHeight es negativo, la imagen está ordenada de arriba a abajo.
     if(bmpHeight < 0) {
       bmpHeight = -bmpHeight;
       flip    = false;
     }
```

Se presenta a continuación el último trozo del código referido a la función de dibujado por pantalla. En este trozo, y tras hacer determinadas comprobaciones, es dónde se procede al dibujado píxel a píxel a través de la pantalla. Se hace utilizando un buffer que almacena el píxel leído del bmp de origen y más tarde se pasa dicha información al píxel correspondiente de la pantalla. Como se viene haciendo hasta este punto, se han añadido determinados comentarios que faciliten la compresión del código presentado.

```
// Recorta el área a cargar
     w = bmpWidth;
     h = bmpHeight;
     if((x+w-1) >= tft.width())  w = tft.width()  - x;
     if((y+h-1) >= tft.height()) h = tft.height() - y;
     // Pone las direcciones de TFT a los lugares recortados.
     SPCR = 0;
     tft.setAddrWindow(x, y, x+w-1, y+h-1);
     for (row=0; row<h; row++) {
       if(flip) // BMP está guardado de abajo a arriba. (normal)
       pos = bmpImageoffset + (bmpHeight - 1 - row) * rowSize;
       else   // BMP está guardado de arriba a abajo. (invertido)
       pos = bmpImageoffset + row * rowSize;
       SPCR = spi_save;
       if(bmpFile.position() != pos) {
```

```
      bmpFile.seek(pos);
      buffidx = sizeof(sdbuffer); // Fuerza la recarga del buffer
      }
      for (col=0; col<w; col++) {
      if (buffidx >= sizeof(sdbuffer)) {
        // Pasa el buffer del LCD al display primero.
        if(lcdidx > 0) {
        SPCR = 0;
        tft.pushColors(lcdbuffer, lcdidx, first);
        lcdidx = 0;
        first  = false;
        }
        SPCR = spi_save;
        bmpFile.read(sdbuffer, sizeof(sdbuffer));
        buffidx = 0; // Pone índice al incio.
      }
      // Convierte el píxel de formato BMP a TFT
      b = sdbuffer[buffidx++];
      g = sdbuffer[buffidx++];
      r = sdbuffer[buffidx++];
      lcdbuffer[lcdidx++] = tft.color565(r,g,b);
      } }// Termina el píxel
    // Pasa los datos que falten al LCD
    if(lcdidx > 0) {
      SPCR = 0;
      tft.pushColors(lcdbuffer, lcdidx, first);
    } }
  }}
  bmpFile.close();}
```

Para terminar con el programa de dibujado de fondo por completo, se presentan las dos últimas funciones contenidas en el mismo cuya función no es otra que leer archivos de 32 o 16 bits desde la tarjeta SD. Diviendo la información en pedazos de 8 bits.

```
// Estas dos funciones leen tipos de 16 y 32 bits de la tarjeta SD.
 uint16_t read16(File f) {
   uint16_t result;
   ((uint8_t *)&result)[0] = f.read(); // LSB
   ((uint8_t *)&result)[1] = f.read(); // MSB
   return result;
 }
 uint32_t read32(File f) {
   uint32_t result;
   ((uint8_t *)&result)[0] = f.read(); // LSB
   ((uint8_t *)&result)[1] = f.read();
   ((uint8_t *)&result)[2] = f.read();
   ((uint8_t *)&result)[3] = f.read(); // MSB
   return result;
 }
```

Una vez se ha terminado y comprobado el funcionamiento correcto de la primera iteración del software de la pantalla, se pasó a realizar la segunda iteración del código. Hay que destacar que, entre versiones, se realizaron cantidad de pequeñas pruebas (concretamente una por función nueva que se añadía al código) hasta llegar a una programación que se puede considerar como realmente diferente a la anterior. En este caso, una de las pruebas que se realizaron una vez la primera iteración funcionaba de manera estable, fue la de pintar encima del fondo para comprobar que efectivamente se veía correctamente.

3.4.1.3 Segunda iteración: Elementos Dinámicos sobre Fondo

Como punto de partida para esta segunda versión o iteración del software de pantalla se tomó la primera, cuya funcionalidad no era más que representar una imagen por pantalla. Imagen que hace las veces de fondo del display en esta segunda versión.

La funcionalidad que se añadió era la de representar un valor leído de un sensor, concretamente del potenciómetro que se especificó al principio de este documento. Pero, en lugar de hacerse de manera numérica, al ser indirectamente la potencia demandada al motor se quiso representar de manera gráfica con una barra de progreso, en este caso, de un solo color.

Para realizar esta tarea, se tuvieron que añadir ciertas variables y funciones al código anterior que se mostrarán a continuación. Es debido especificar que, a pesar de sí tener adiciones, el programa de dibujado del fondo se vio inalterado en cuanto a eliminaciones, recortes o sustituiciones de partes se refiere.

Las variables globales creadas y añadidas al principio del código son sólo las siguientes:

```
uint16_t throttle;  //A15
uint16_t throttle_1=0;
```

Son sólo dos porque, al haber utilizado una estructura de funciones, el resto se han creado de manera local dentro de cada una de las funciones que se representarán a continuación.

Las funciones que se han creado han sido principalmente tres, una que dibuja la barra acorde con el valor de sensor leído, otra cuya misión es borrar la barra antes de pintar el nuevo valor y, una tercera, encargada de coordinar ambos procesos para que el resultado sea fluido. Además, en el bucle principal o `void loop()` se han de llamar dichas funciones y leer el sensor (será mostrado más tarde).

La función de representación de la barra de progreso por pantalla, dado un valor del sensor leído es la siguiente:

```
void drawBar(uint16_t gas){
  uint16_t bar;
  uint16_t i;
  bar=gas/2.56; //conversion de valor de sensor a píxeles
    for(i=0;i<bar;i++){
    tft.drawFastVLine(0+i,55,25,OLIVE);}i=0; //Función incluida en la biblioteca
    };
```

La función solo necesita un argumento de entrada, la lectura del sensor de apertura del gas. En cuanto a la cota vertical de la barra en la pantalla y la anchura de la misma, esto se define por defecto y de manera inamovible para ordenar los elementos del display de manera correcta. En referencia al funcionamiento, se transforma el valor del sensor potenciométrico (0-1023) a un valor entero que reflejara la anchura en píxeles que debería tener la barra siendo 1023 todo el ancho de la pantalla. Una vez calculados el número de columnas de píxeles de altura ***h***, se pasa en un bucle a pintarlas del color elegido, en este caso: verde oliva.

La función de borrado de la barra es la que sigue:

```
void eraseBar (uint16_t throttle,uint16_t throttle_1)
  { uint16_t puntero;
    uint16_t ancho;
    puntero=throttle_1/2.56;
    ancho=(throttle-throttle_1)/2.56+1;
    tft.fillRect(puntero,55,ancho,25,BLACK);}
```

Para este caso, los argumentos de entrada son dos: valor actual del sensor de aceleración y valor anterior. El obejtivo es leer el nuevo valor del sensor y "repintar" el trozo de barra de progreso sobrante (desde el valor nuevo al valor anterior, que era mayor) del mismo color del fondo para eliminarlo a la vista. El valor *puntero* es el valor anterior de aceleración transformado a píxeles y, el *ancho*, la resta en píxeles (más uno extra para asegurar el borrado completo).

La tercera función necesaria es la que gestiona y coordina las dos anteriores, haciendo que el borrado y pintado ocurra cuando debe.

```
void barLoop (uint16_t throttle,uint16_t throttle_1) {
  if(throttle>throttle_1+2){
  drawBar(throttle,25);}
  else if (throttle<throttle_1-2){
    eraseBar(throttle,throttle_1);
    }
  else {}  }
```

Al igual que en el caso de la función anterior, se reciben dos argumentos de entrada: aceleración actual y anterior. Por medio de una serie de condiciones, si el valor actual es mayor que el anterior se procede a pintar la barra y, en caso contrario, se borra el pedazo sobrante. Es de observar que, en la parte dónde se evalúan las condiciones se han añadido un "+2" y un "-2" que actúan a modo de *offset,* evitándose que el ruido del sensor fuerce un continúo redibujado que se percibiría como un parpadeo en la pantalla.

Una vez se han mostrado las funciones creadas para la gestión de la barra gráfica, se van a presentar las sentencias incluidas en el bucle principal de Arduino. Dado que, sin éstas, las funciones no serían llamadas ni tendrían datos de entrada, por lo que no harían nada.

```
analogRead(A15);
  throttle = analogRead(A15);
  barLoop(throttle,throttle_1);
```

En estas líneas se lee el sensor y se le da de entrada a la función de gestión de gráficos por pantalla. En las siguientes se actualizan los valores, dándo a valor de aceleración anterior el actual como último paso del programa.

```
//Actualización de los valores anteriores
  throttle_1=throttle;
```

Como resumen de esta segunda versión del código dedicado a la pantalla puede concluirse que, se consigue correctamente leer un sensor y representarlo por pantalla no sólo de manera numérica. Para la siguiente iteración, en la que ya se explicará el resultado final, se han añadido más sensores y un sistema para contar marchas que se detallará más tarde.

3.4.1.4 Tercera iteración: Versión Final del Software de Pantalla

Como se dijo en el párrafo anterior, esta es la última versión del software de pantalla. Es una mejora de la segunda iteración añadiendo una mayor cantidad de sensores, así como de elementos a la pantalla. Los sensores que se leerán para representarse en el display son:

- Dos sensores de temperatura LM35.
- Dos pulsadores utilizados para la realización del contador de marchas.
- Un potenciómetro analógico (utilizado ya en el caso anterior).

Para entender bien el trabajo realizado y su complejidad, se tienen que explicar primero ciertas características limitantes de la placa utilizada y de como se hace en otros sistemas similares.

Generalmente, cuando se programan placas o sistemas microprocesados para propósitos de este tipo o incluso diferentes, se utilizan interrupciones. Las interrupciones son recursos esenciales de los sitemas microprocesados mediante los cuáles, un microprocesador puede (cada cierto tiempo o al ocurrir algún determinado evento) interrumpir su tarea actual para atender una de mayor prioridad. En ocasiones se puede encontrar diferentes tipos de interrupciones, de tiempo o activadas por hardware. Las interrupciones de disparo temporal son de las más útiles dado que, se puede organizar el desarrollo de la programación de manera precisa y estable, definiendo cuando sucederán los diferentes eventos.

Pero, en Arduino se tiene la limitación de que, en Arduino si se utiliza una interrupción como pueden ser las de hardware, los timer se detienen durante el tiempo de ejecución de la subrutina de interrupción. Esto, si dicho tiempo de parada no es elevado, no debería dar problemas, pero para el caso de gestión gráfica que se está abordando, los procesos no son precisamente rápidos. Por lo que, sólo se han utilizado las interrupciones (concretamente las de *hardware*) para evitar el debounce de los pulsadores al contar las marchas. Todo esto se explicará más adelante cuando se analice el código.

A diferencia del caso anterior, dónde solo se reflejaron las modificaciones, para esta última versión se va a presentar el código completo.

La primera parte del código, en la que se incluyen librerías y se definen la relación de equivalencia de los colores en formato hexadecimal y en lengua inglesa, permanece prácticamente inamovible.

```
#include <SPI.h>
```

```
#include <SD.h>
#include <Adafruit_GFX.h>
#include <MCUFRIEND_kbv.h>

MCUFRIEND_kbv tft;
#define BLACK   0x0000
#define BLUE    0x001F
#define RED     0xF800
#define GREEN   0x07E0
#define CYAN    0x07FF
#define MAGENTA 0xF81F
#define YELLOW  0xFFE0
#define WHITE   0xFFFF
#define NAVY        0x000F
#define DARKGREEN   0x03E0
#define DARKCYAN    0x03EF
#define MAROON      0x7800
#define PURPLE      0x780F
#define OLIVE       0x7BE0
#define LIGHTGREY   0xC618
#define DARKGREY    0x7BEF
#define ORANGE      0xFD20
#define GREENYELLOW 0xAFE5
#define PINK        0xF81F
```

Una vez hecho lo anterior, el siguiente paso del código no es otro que declarar las variables que se van a utilizar a lo largo del código. Se han declarado diferentes tipologías de variables dependiendo de su función, su aplicación, la resolución necesaria, el tipo de dato, etc.

```
const byte  downPin = 2;
const byte upPin = 3;
const int timegap = 100;
long startTime=0;
volatile int gear=0;
volatile int gear_1=0;
uint16_t throttle;  //A15
uint16_t throttle_1=0;
int counter;
float grados1;
float grados2;
float grados1_1;
float grados2_1;
uint16_t temp1;    //A14
uint16_t temp2;    //A13
int temp1_flag;
int temp2_flag;
uint16_t largo=400;
uint16_t ancho=240;
#define SD_CS 5
File root;
char namebuf[32] = "backg.bmp";
int pathlen;
uint16_t color;
uint32_t i=0;
uint8_t  spi_save;
```

Mirando a simple vista el código superior se puede ver que, se han establecido una serie de variables diferentes que se explican a continuación:

- *int*: Número entero codificado en 16 bits y que tiene como valores mínimo y máximo -32,768 y 32,767.
- *float:* Número real, con decimales, almacenado igualmente en 32 bits.

- *long:* Variables que almacenan un número entero, pero utilizando 32 bits para ello.
- *char:* Varable utilizada para almacenar un carácter o una cadena de caracteres. Hay que especificar que realmente se guardan internamente como números (ver la tabla ASCII) y que se puede operar con ellos.
- *uint8_t , uint_16:* Este tipo de variables es especial porque son derivadas de las anteriores pero con modificaciones. Observando cuidadosamente se puede ver que su sintaxis se compone de las siguientes partes:
    - *Prefijo (u):* hace referencia a que es una variable *"unsigned"* es decir, su valor solo puede ser mayor que cero.
    - *Parte central (int):* indica que el tipo va a ser un entero.
    - *Sufijo (8_t, 16_t):* indica el tamaño en bits que se quiere reservar para dicha variable, quedando forzado así su valor límite.

A parte de lo anterior, hay dos modificadores de variable más en el código usado:

- *volatile :* este modificador indica que la variable se almacena en la memoria RAM, no en un registro, de este modo cualquier subrutina (incluida las de interrupción) puede modificar su valor, en cualquier momento.
- *const:* modificador de variable que fija su valor, conviertiéndola en una variable de sólo lectura.

Una vez definidos los tipos de variables (los usos de cada una se verán a la hora de usarse en el código), el siguiente paso es comenzar a explicar las diferentes funciones realizadas a lo largo del código. Primero se va a mostrar la función (incluida en todos los programas de Arduino) denominada *setup*.

Cómo se ha explicado en epígrafes anteriores, esta función tiene dos contenidos principalmente, la parte de código empleada para encender y configurar determinados elementos del sistema y, todas aquellas tareas que quieran realizarse una sola vez. La función *setup* de este programa es la siguiente:

```
void setup()
{
pinMode(upPin, INPUT_PULLUP); //Pone el pin "upPin" (3) como Resistencia pullup
pinMode(downPin, INPUT_PULLUP); //Pone el pin "downPin" (2) como Resistencia pullup
//////////////INICIO DE LA DEFINICIÓN DE LAS INERRUPCIONES///////////////
attachInterrupt(digitalPinToInterrupt(upPin), gear_up, RISING);
attachInterrupt(digitalPinToInterrupt(downPin), gear_down, RISING);
//En estas dos líneas se definen las interrupciones, los pines, las funciones
//a ejecutar y la condición de disparo (RISING)
//////////////INICIO DE LA DEFINICIÓN DE LAS INERRUPCIONES///////////////
uint16_t ID;
ID = tft.readID();
if (ID == 0x0D3D3) ID = 0x9481;
tft.begin(0x65);//Habilita  los drivers de HX8352B
tft.fillScreen(0x0000);
bool good = SD.begin(SD_CS);
tft.flag_write_bmp = 1;
tft.setRotation(1);
bmpDraw("backg.bmp", 0, 0);
tft.flag_write_bmp = 0;
tft.setRotation(1);
tft.setCursor(150,25);
tft.setTextColor(WHITE);
tft.setTextSize(3);
tft.print("DASHBOARD");
tft.drawFastHLine(0,67,400,LIGHTGREY);
tft.drawFastHLine(0,68,400,LIGHTGREY);
tft.drawFastHLine(0,69,400,LIGHTGREY);
tft.drawFastHLine(0,70+25,400,LIGHTGREY);
tft.drawFastHLine(0,70+26,400,LIGHTGREY);
tft.drawFastHLine(0,70+27,400,LIGHTGREY);
tft.setCursor(0,115);
tft.setTextSize(2);
tft.setTextColor(LIGHTGREY);
```

```
tft.print("Temp. Aceite: ");
tft.setCursor(0,145);
tft.setTextSize(2);
tft.setTextColor(LIGHTGREY);
tft.print("Temp. Agua: ");
tft.setCursor(270,110);
tft.setTextSize(3);
tft.setTextColor(LIGHTGREY);
tft.print("C");
tft.setCursor(270,140);
tft.setTextSize(3);
tft.setTextColor(LIGHTGREY);
tft.print("C");
tft.drawFastVLine(310,180,100,LIGHTGREY);
tft.drawLine(310,180,310+25,180-35,LIGHTGREY);
tft.drawFastVLine(311,180,100,LIGHTGREY);
tft.drawLine(311,180,311+25,180-35,LIGHTGREY);
tft.drawFastVLine(312,180,100,LIGHTGREY);
tft.drawLine(312,180,312+25,180-35,LIGHTGREY);
tft.drawFastHLine(312+25,180-35,70,LIGHTGREY);
tft.drawFastHLine(312+25,181-35,70,LIGHTGREY);
tft.drawFastHLine(312+25,182-35,70,LIGHTGREY);
tft.setCursor(0,40);
tft.setTextSize(2);
tft.print("Gas:");
   tft.setCursor(312+27,122);
   tft.print("Gear:");
   tft.setCursor(312+32,170);
   tft.setTextSize(7);
   tft.setTextColor(WHITE);
   tft.print("N");
   }
```

A lo largo del código se han dejado determinados comentarios explicativos que ayudan a la major comprensión de partes concretas del código. Los comentarios en el lenguaje nativo de Arduino comienzan con dos barras "//" y a partir de ellas el compilador ignorará las pabras escritas entendiendo que son anotaciones para el programador.

En la parte no comentada del código se dibujan las líneas y formas geométricas, se escriben y dan formatos a los textos de la pantalla y se llama a la función que pinta el fondo (explicada en el primer código de pantalla que se explicó en el documento, no se incluirá en esta parte).

Una vez explicada una de las funciones principales, se van a presentar las pequeñas funciones creadas concretamente para este programa y que serán llamadas y ejecutadas en bucle iterativo llamado *loop,* que es la otra de dichas funciones principales.

```
//////////////FUNCIÓN PARA DIBUJAR LA BARRA//////////////
void drawBar(uint16_t gas,uint16_t h){
  uint16_t bar;
  bar=gas/2.56; //conversion de sensor a pixeles
  if (bar <= 200){
  tft.fillRect(0,70,bar,h,OLIVE);}
  if( bar>200 && bar < 350){
    tft.fillRect(0,70,bar,h,YELLOW);}
  if( bar >= 350){
    tft.fillRect(0,70,bar,h,MAROON);}
    };
///////////FUNCION PARA BORRAR LA BARRA////////////////////
void eraseBar(uint16_t gas,uint16_t h){
  uint16_t bar;
  bar=gas/2.56; //conversion de sensor a pixeles
  tft.fillRect(bar,70,(400-bar),h,BLACK);
    };
```

Estas dos funciones son similares, por no decir idénticas a la de la versión anterior del código. La única

variación es que, dependiendo de la longitud de la barra que indica la apertura del acelerador, se dibuja la misma de un color u otro. Se ha elegido el verde oliva para valores bajos (o una barra de pequeño tamaño), se torna por completo amarilla para valores intermedios y, por último, se pone roja granate para valores elevados cercanos al total.

La función siguiente también ha sido realizada de casi la misma forma que en el caso anterior y es la encargada de gestionar cuándo y como se pinta o borra la barra de progreso anterior.

```
//////////////FUNCIÓN GRÁFICOS BARRA////////////////////
void barLoop (uint16_t throttle,uint16_t throttle_1) {
  if(throttle >= throttle_1+2){
  drawBar(throttle,25);}
  if(throttle < throttle_1-2){
  eraseBar(throttle,25);
  }
  }
```

Presentadas ya las funciones de representación y gestión de una de las partes gráficas más importantes de la pantalla, la barra, se pasa a las funciones que hacen una función muy similar, pero para representar el valor numérico de un sensor leído. Concretamente se utilizarán en el bucle principal para representar las dos temperaturas leídas que, a la hora de aplicarse en una motocicleta de competición, serían la temperatura del agua y del aceite.

La manera de proceder es relativamente sencilla, primero se elimina (dibujando un cuadrado negro encima, que es el mismo color del fondo) el valor anterior que pueda haber del sensor y, acto seguido se representa el último valor leído que es precisamente argumento de entrada de la función.

```
/////////FUNCIÓN PARA REPRESENTAR VALOR NUMÉRICO DE UN SENSOR/////////////////
void sensorPrint (float sensor,uint16_t lettersize,uint16_t color,uint16_t xo,uint16_t
yo){
  tft.fillRect(xo-1,yo-1,100,24,BLACK); //Borrado del valor anterior que pueda haber
  tft.setTextSize(lettersize); //Selección del tamaño de letra
  tft.setCursor(xo,yo); //Colocación del cursor en el lugar deseado
  tft.setTextColor(color); //Se elige el color de letra (entrada de la función)
  tft.print(sensor);} // Se imprime por pantalla el valor del sensor
```

Una de las novedades que tiene esta versión de la pantalla, a parte de las ya explicadas, es la lectura de dos pulsadores que hacen las veces de cambio de marchas. Para leer dichos pulsadores y contar el número de pulsaciones (sea para subir o bajar de marchas) se han utilizado diferentes tipos de funciones.

Cómo primera prueba, se realizó primero con una programación en el bucle principal, con una sentencia *if else* que comprobaba si se estaba realizando una pulsación en ese momento o no (par ambos pulsadores). Este primer enfoque resultó ser erróneo porque, aunque realizaba de manera correcta las tareas de sumado y restado al valor de la variable de marcha engranada, requería que la pulsación se estuviera realizando en el momento exacto en el que el Arduino interpretaba esas líneas de código. Esto último hacía que el comportamiento de la pareja de pulsadores fuera altamente deficiente.

La segunda prueba realizada ya se acercaba bastante al comportamiento requerido, al utilizarse interrupciones de hardware. Las interrupciones de hardware, como se explicó con anterioridad, detenían el código principal por dónde fuera y priorizaban el código de la subrutina asociada a la interrupción, al mismo instante de ocurrir un evento concreto.
Por esto, se conectaron los pulsadores a pines que tenían asociadas dichas interrupciones de hardware y se les configuró para que, al pulsar, se ejecutase la función de subida o bajada de marcha, según el caso.

Sin embargo, al presionar alguno de los botones se podía observar que realizaba su operación asignada más de una vez. Por lo que se podía inferir de manera directa que se ejecutaba, en ocasiones, más de una interrupción por pulsación. Esto es debido a lo denominado como rebote del pulsador.

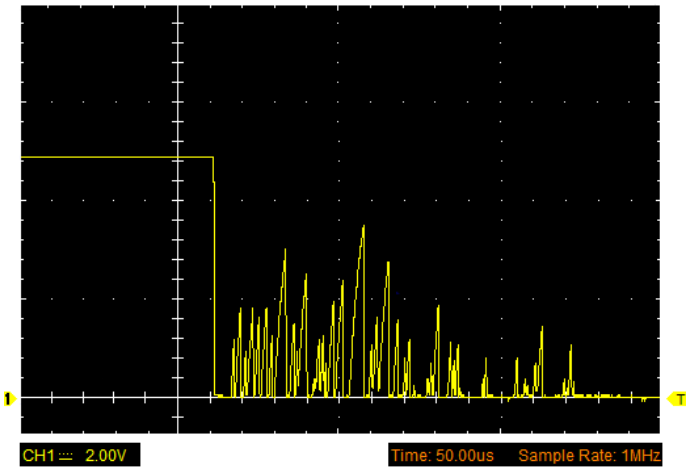


Figura 61:Señal de voltaje (con ruido) a la salida de un pulsador ejemplo.

En general la mayoría de dispositivos electrónicos generan una señal limpia pero, muchos otros generan ruidos en los flancos de subida y bajada, sobretodo en términos de tensión. De manera que, si se configura la rutina de interrupción para que salte al detectar uno de los dos tipos de flancos que el pin de Arduino puede apreciar, puede falsearse debido al ruido o rebote (fenómeno representado en la *Figura 61*).

Para poder paliar este efecto y leer una sola interrupción por pulsación se pueden afrontar dos mecanismos de *debounce* (filtrado del rebote) diferentes:

- Mediante *hardware* con la inclusión de capacidades en paralelo que asuman las variaciones de voltaje.
- Mediante *software* estableciendo un *offset* o margen temporal que haya que esperar entre interrupciones.

En este caso, al no necesitar un filtrado muy preciso porque se conoce que las pulsaciones, aunque consecutivas, no tienen una diferencia temporal demasiado pequeña, se elige hacer un filtrado mediante *software*.

En este caso, como se puede intuir por la *Figura 62* de la derecha, si durante el tiempo de *offset* sucedieran otros flancos de subida (o bajada según se configure el disparo de la interrupción) serían ignorados por el software. De este modo, se detectaría solo un flanco por pulsación que es, precisamente, lo buscado al implementar esta técnica.

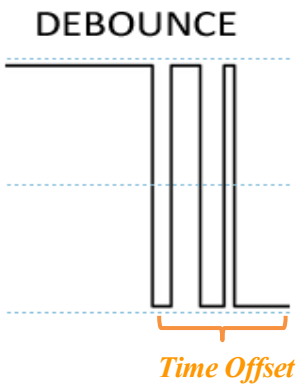


Figura 62: Explicación gráfica del rebote y del time offset impuesto.

La tercera prueba implementada para poder leer las presiones de los interruptores fue ya incluyendo este mecnismo de *debounce*. El código implementado se muestra a continuación:

```
/////////FUNCIONES PARA CONTAR MARCHAS////////////

void gear_up() {
  if (millis()-startTime > timegap){
  gear++;
  if (gear > 5){gear=5;}
  if (counter < 0) {gear=0;}
  startTime=millis();}
}

void gear_down() {
  if (millis()-startTime >timegap){
  gear--;
  if (gear > 5){gear=5;}
  if (gear < 0) {gear=0;}
  startTime=millis();}
```

Se define la variable *timegap* como constante de modo que siempre tenga el mismo valor pase lo que pase en el código, esto se hace de este modo po

rque al ser un parámetro crítico se impide alguna posible fisura en la programación que cambie su valor.

Ambas funciones anteriores están hechas siguiendo un esquema general bastante similar, por no decir idéntico, con los cambios necesarios para que, al leerse un botón se baje una marcha y al leer el otro se suba. Se le han introducido además saturaciones mínimas y máximas (0 y 5 respectivamente) que son los límites que tendría una caja de cambios real.

Cómo se puede apreciar, la primera línea que se ejecuta dentro de cada función comprueba si ha pasado un tiempo suficiente entre la última línea de la interrupción (que sería lo último que ejecute la interrupción anterior) y el siguiente flanco. Realmente y siendo completamente estrictos, la interrupción se produce todas las veces que detecte flanco por el rebote, pero sólo se ejecuta la subrutina una vez por pulsación por el *offset* de tiempo.

Una vez se ha solventado este problema, se crea una función complementaria (como en el caso de la barra de progreso) que gestiona la representación del valor por pantalla, su borrado e incluso su color si la marcha es elevada o no. La función de gestión gráfica del valor de la marcha engranada es la siguiente:

```
/////////FUNCIÓN PARA REPRESENTAR MARCHAS/////////////////
void gearprint(){
  if(gear!=gear_1){
  tft.setCursor(312+30,168);
  tft.fillRect(312+30,168,40,80,BLACK);
  tft.setCursor(312+32,170);
  tft.setTextSize(7);
  switch (gear){
    case 0:
      tft.setTextColor(WHITE);
      tft.print("N");
      break;
    case 1:
      tft.setTextColor(GREEN);
      tft.print(gear);
      break;

    case 2:
      tft.setTextColor(GREEN);
      tft.print(gear);
      break;
    case 3:
      tft.setTextColor(YELLOW);
      tft.print(gear);
      break;
    case 4:
      tft.setTextColor(ORANGE);
      tft.print(gear);
      break;
    case 5:
      tft.setTextColor(RED);
      tft.print(gear);
      break;
    }
  }
  else return;
}
```

Cómo se puede apreciar, se utiliza un bucle del tipo *switch case* que permite establecer diferentes acciones según sea el valor de la variable que se elija. Para este caso, se elige la variable *gear* en la que las subrutinas de interrupción escriben y con los diferentes casos del bucle se toman para darle un formato de color al número representado.

El motivo de haber explicado todas estas funciones previamente es que todas se llaman y ejecutan en el bucle principal del programa, llamado *loop*, que se expone a continuación:

```
void loop()
 {
  throttle = analogRead(A15);
  analogRead(A6);
  temp1= analogRead(A6);
  analogRead(A7);
  temp2=analogRead(A7);
  grados1 = temp1*5.0*100.0/1024.0;
  grados2 = temp2*5.0*100.0/1024.0;
  barLoop(throttle, throttle_1);
  if (grados1>(grados1_1+0.5) || grados1<(grados1_1-0.5)){
    temp1_flag=1;}
    else {temp1_flag=0;}
  if (grados2>(grados2_1+0.5) || grados2<(grados2_1-0.5)){
    temp2_flag=1;}
    else {temp2_flag=0;}
  if (temp1_flag!=0){
  sensorPrint(grados1,grados1_1,3,GREEN,170,110);}
  else{};
   if (temp2_flag!=0){
  sensorPrint(grados2,grados2_1,3,CYAN,170,140);
   }
   else{};
  gearprint();
  throttle_1=throttle;
  grados1_1=grados1;
  grados2_1=grados2;
  gear_1=gear;
    }
```

En las primeras líneas del código se puede apreciar como, utilizando la funció nativa de Arduino *analogRead*, se lee la entrada de voltaje analógico por un pin y se convierte a valor digital. Esto se realiza primero para todos los pines necesarios dado que así ya se tienen los valores de los sensores disponibles para su posterior representación.

En las siguientes líneas, se transforma el valor leído de 0 a 1023 proveniente de los termómetros a un valor de grados centígrados utilizando una fórmula que viene dada el Datasheet entregado por el fabricante. Una vez hecho esto, se llama a la subrutina de gestión gráfica de la barra de progreso del acelerador dóndole como entrada el valor del sensor actual, y el pasado.

La siguiente parte del programa, formada por dos bucles if en los que se comparaban el valor actual y pasado de los grados medidos por los termómetros, necesita una explicación concreta. Se extraen unas líneas del código anterior para comentarlas más en profundidad, se van a comentar por parejas en dos pequeños paquetes. El primero es el siguiente:

```
 if (grados1>(grados1_1+0.5) || grados1<(grados1_1-0.5)){
    temp1_flag=1;}
    else {temp1_flag=0;}
  if (grados2>(grados2_1+0.5) || grados2<(grados2_1-0.5)){
    temp2_flag=1;}
    else {temp2_flag=0;}
```

En esta combinación de bucles *if* se comprueba si la temperatura ha cambiado un valor mínimo determinado respecto al valor anterior, de ser así pone a valor 1 o TRUE un *flag* que se usará en las siguientes líneas. En las dos próximas líneas se establece la condición de llamada a la escritura del sensor, condición que se cumple básicamente si el *flag* está a 1. Las líneas son las siguientes:

```
if (temp1_flag!=0){
  sensorPrint(grados1,grados1_1,3,GREEN,170,110);}
  else{};
   if (temp2_flag!=0){
  sensorPrint(grados2,grados2_1,3,CYAN,170,140);
   }
```

El resumen de los dos paquetes de código anteriores sería que, si se cumple una condición de cambio mínimo de temperatura, se escribe el valor del sensor por pantalla. Esto se hace de este modo porque, de no existir dicha condición de escritura, el valor del sensor estaría cambiando continuamente y, por ende, parpadeando por pantalla.

Se establece un offset de medio grado puesto que este dato no requiere una mayor resolución al ser meramente informativo y, además, de ser el orden de magnitud de las temperaturas medidas decenas de grados (con un valor aproximado de entre 80 y 90 grados). Por lo que se puede concluir que un *offset* de medio grado, es una pérdida de resolución asumible.

Siguiendo con el código principal (función *loop*) se puede apreciar que, pasadas las líneas comentadas aparte anteriormente, se hace la llamada a la función que representa el valor de la marcha engranada y se actualizan valores de las variables. Una vez realizadas estas operaciones el código habría terminado y se ejecutaría de nuevo, al tratarse de una función, como ya se ha explicado, iterativa.

La única función que quedaría por añadir al código de esta tercera versión de software de pantalla sería el de representación de una imagen o fondo, que no ha cambiado desde los otros ejemplos.

Una vez se ha terminado de exponer y detallar la programación del código de pantalla se va a pasar a detallar el de la adquisición de datos, que forma la segunda mitad de lo que se ha denominado **sistema on-board**.

3.4.2. Software del Sistema de Adquisición de Datos

3.4.2.1 Introducción y Dispositivos Necesarios

Como se ha detallado con anterioridad, la idea principal es la de hacer un segundo subsistema que realice la lectura de los sensores y GPS para después guardar los datos en una tarjeta SD externa. Una vez que se tengan los datos en la tarjeta, ésta se insertará en un ordenador en el que se ejecutará un script que represente los datos gráficamente.

Para la realización de este proceso de guardado de datos se necesita un dispositivo de almacenamiento, en este caso una tarjeta SD como se ha especificado, cuya capacidad es vital para poder guardar una cantidad suficiente de datos. Teniendo en cuenta la cantidad de datos que se quiere almacenar, el tiempo medio por vuelta y la cantidad de vueltas al circuito que se quieren almacenar, se puede estimar un tamaño mínimo necesario para el sistema.

En las primeras fases de este estudio de viabilidad, se estimó que una muestra de todos los sensores cada 0,1 segundos sería signo de un sistema viable. Por lo que, teniendo en cuenta lo anterior, habría que elegir un sistema de almacenamiento acorde a dicha cantidad de datos.

Si se tienen 10 lecturas por segundo y una vuelta al circuito del tamaño del *Circuito de Motorland Aragón* (donde se utilizaría este sistema) para la motocicleta utilizada, es de aproximadamente 2 minutos y medio (dejando cierto margen), se necesitarían aproximadamente:

$$10\,\frac{lecturas}{s}\cdot 150\,\frac{s}{vuelta} = 1500\,\frac{lecturas}{vuelta}$$

El número de lecturas calculado es el necesario para almacenar los datos de una vuelta, si se tienen aproximadamente siempre un número menor a 30 vueltas[14]:

$$1500\,\frac{lecturas}{vuelta}\cdot 30\ vueltas = 4500\ lecturas$$

Una vez calculado el número total de lecturas de todos los sensores que hay que poder almacenar, es necesario calcula el espacio en *bytes* que se necestaría. Esto depende directamente del número de sensores y de lo que ocupa el valor de cada uno de ellos en memoria. Para estimar el tamaño necesario para almacenar una muestra de todos los sensores, se pueden tomar dos estrategias u opciones:

[14] El número de vueltas totales se calcula en cada evento para que la distancia recorrida sea siempre la misma y, dadas las longitudes de los circuitos, siempre es menor a 30.

- Analizar el tamaño de cada variable utilizada para guardar los valores y realizar los cálculos necesarios.
- Guardar una lectura de muestra y, viendo el valor que ocupa en la memoria del Arduino, realizar una estimación del tamaño total necesario.

Anteriormente en este trabajo se ha detallado algunos de los tamaños que reserva Arduino para los diferentes tipos de variables utilizadas, por lo que sería posible y bastante acertado utilizar la primera variante. Sin embargo, esto no tiene en cuenta de manera totalmente precisa los separadores utilizados entre datos, los tamaños del GPS y demás factores. Por lo que, usar la segunda opción sería lo que a nivel práctico sería más preciso.

Tomándose una muestra de la localización GPS y, por separado, una de cada uno de los sensores, el tamaño total para guardar una lectura completa sería:

- Para el caso del GPS, el sistema reserva 22 *bytes* por lectura.
- Para la lectura completa de los sensores, el sistema reserva aproximadamente *18 bytes*.

Calculando el tamaño total, se puede deducir que se necesitan reservar unos 40 *bytes* de espacio para cada lecura. Si se estableció, mediante cálculos anteriores, que se necesitan 4500 lecturas en total:

$$4500\ lecturas \cdot 40\ bytes = 180\ 000\ bytes = 180\ kilobytes$$

A priori, no es un tamaño muy grande el necesario para hacer el guaradado de datos al ser un proceso que se ha depurado al máximo, pero aún así, las memorias presentes en Arduino Mega se antojan insuficientes. Tanto la EEPROM como la SRAM son memorias de tamaño insuficiente (de poder usarse) y la memoria Flash está reservada para programas, por lo que, lo anterior confirma la tesis de querer añadir una tarjeta SD externa. Sin embargo, podría usarse prácticamente cualquiera, dado que hoy en día, todos los tamaños que se trabajan en el mercado exceden los 180 *kilobytes*.

Para usar la tarjeta SD anteriormente descrita, se necesita un dispositivo conecte la tarjeta SD con el Arduino para poder hacer la lectura y escritura de datos. Este tipo de modulos utilizan una conexión SPI y alimentación, lo que los hace realtivamente fáciles de utilizar. Un ejemplo de lector (en este caso se ha utilizado de microSD por ser más compacto) y una propuesta de conexionado se puede ver en la *Figura 63*.

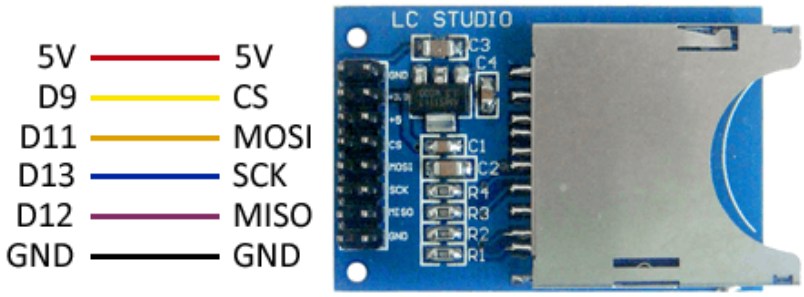


Figura 63: Módulo de tarjeta microSD utilizado.

A lo largo del código de *datalogging* se detallarán los pines reales utilizados, pero no diferirán demasiado de los mostrados en la imagen, de hecho, algunos como la alimentación o la tierra (GND) serán los mismos.

3.4.2.2 Incompatibilidades encontradas

Siguiendo el mismo *modus operandi* que con la pantalla, se han realizado primero diferentes programaciones y pruebas antes de llegar a un diseño final. Sin embargo, a diferencia de para el caso de la pantalla en el que cada prueba confirmaba la dirección tomada en cuanto a desarrollo, para el caso del *datalogging* se han encontrado ciertos impedimentos.

El primero de ellos, más que un impedimento es una limitación que viene dada por la lógica interna de Arduino.

En este sistema, cuando se realiza una interrupción de hardware (cómo la utilizada en el contador de marchas) los *timers* que se han utilizado para guardar el tiempo en el que se lee un sensor se detienen, por lo que el contador de marchas sería perfectamente válido para la pantalla, pero falsearía los resultados del guardado del resto de sensores. Teniendo en cuenta esto, el contador de marchas se queda sólo en la pantalla, pudiéndose salvar el resto de sensores y el GPS en la tarjeta SD.

La segunda y más importante incompatibilidad se va a expresar a continuación y, dado el nivel de detalle que necesita (además de las consecuencias que tiene), se va a dedicar un pequeño epígrafe a la misma.

Incompatibilidad entre sensores y GPS

De inicio, la dirección que se tomó al principio de este trabajo de estudio de viabilidad del sistema fue la de dividir el sistema de adquisición de datos en dos, por un lado, un sistema de adquisición de datos de sensores y un GPS y, por otro, la representación de algunos sensores por pantalla. Esta dirección establecía la necesidad de tener dos módulos en el sistema, uno para pantalla y otro para la lectura y guardado de datos.

Para estudiar correctamente el segundo de los módulos, se divide el trabajo, a su vez en dos. Primero, se realizan unas primeras pruebas de adquisición y guardado de los datos de los sensores utilizados (que son todos de una tipología similar) y, en segundo lugar, se realizan unas pruebas similares para el sistema de posicionamiento global o GPS.

El código de adquisición de sensores y guardado de los mismos (con una marca de tiempo de cuando se toman las medidas) es el siguiente:

```
#include <SPI.h>
#include <SD.h>

const int chipSelect = 4;
int sensor;

void setup() {
  // Abre las comunicaciones de puerto serie
  Serial.begin(9600);
  while (!Serial) {
    ; // espera la conexión del puerto serie
  }


  Serial.print("Initializing SD card...");

  //comprueba si la tarjeta está presente:
  if (!SD.begin(chipSelect)) {
    Serial.println("Card failed, or not present");
    //si no, no hacer nada más:
    while (1);
  }
  Serial.println("card initialized.");
}

void loop() {
  // crea un string para concatenar los datos leídos:
  String dataString = "";
  // lee los sensores y los añade:
  for (int analogPin = 11; analogPin < 16; analogPin++) {
      sensor=analogRead(analogPin);
      if(analogPin>=11 && analogPin<=14) {
      sensor = analogRead(analogPin);
      dataString += String(sensor);
      dataString += String(",");}
      if(analogPin > 14 )
      {sensor = analogRead(analogPin);
      dataString += String(sensor);
      dataString += String(",");
```

```
      dataString += String(millis());
      dataString += String(";");}
    }
  Serial.println(dataString);
  // abre el archivo y activa la opción de escritura:
  File dataFile = SD.open("datalog.txt", FILE_WRITE);
  // si el archivo está disponible, escribe en él:
  if (dataFile) {
    dataFile.println(dataString);
    dataFile.close();
  }
  // si el archivo no está presente, muestra errorx:
  else {
    Serial.println("error opening datalog.txt");
  }
```

En el código que se ha expuesto, simplemente se leen los sensores cableados a los pines utilizados y se concatenan en un vector de datos con una marca de tiempo al final del mismo. Se han añadido comentarios para una mejor interpretación de los resultados a los que solamente quedaría añadir que, todas las líneas que contienen funciones del puerto serie a PC, es decir `Serial`, se utilizan para comunicarse con el ordenador para dar información del programa y depurarlo, en la última versión del programa no estarían presentes.

Una vez se han recogido los datos de los sensores con su marca de tiempo respectiva se representan en *Matlab* para poder interpretar los datos. Esta representación es provisional dado que, más adelante se presentará el software para la representación de datos que se ha realizado. Para analizar los datos obtenidos, se presentan de manera gráfica:

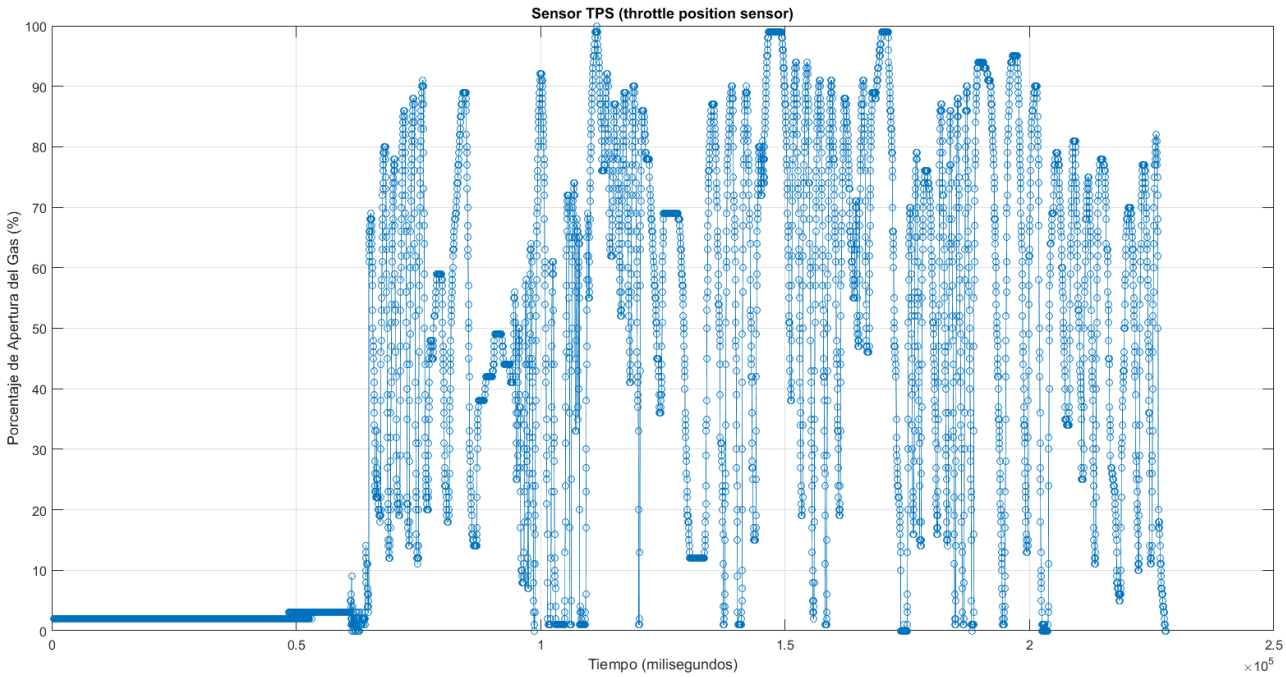


Figura 64: datos leídos y almacenados del sensor TPS.

Al estar el eje x de la gráfica mostrada en la *Figura 64* medido en milisegundos, no se permite apreciar bien cada cuanto tiempo se ha conseguido una muestra. Haciendo zoom en dicho eje se obtiene una mejor precisión como se puede ver en la *Figura 65*.

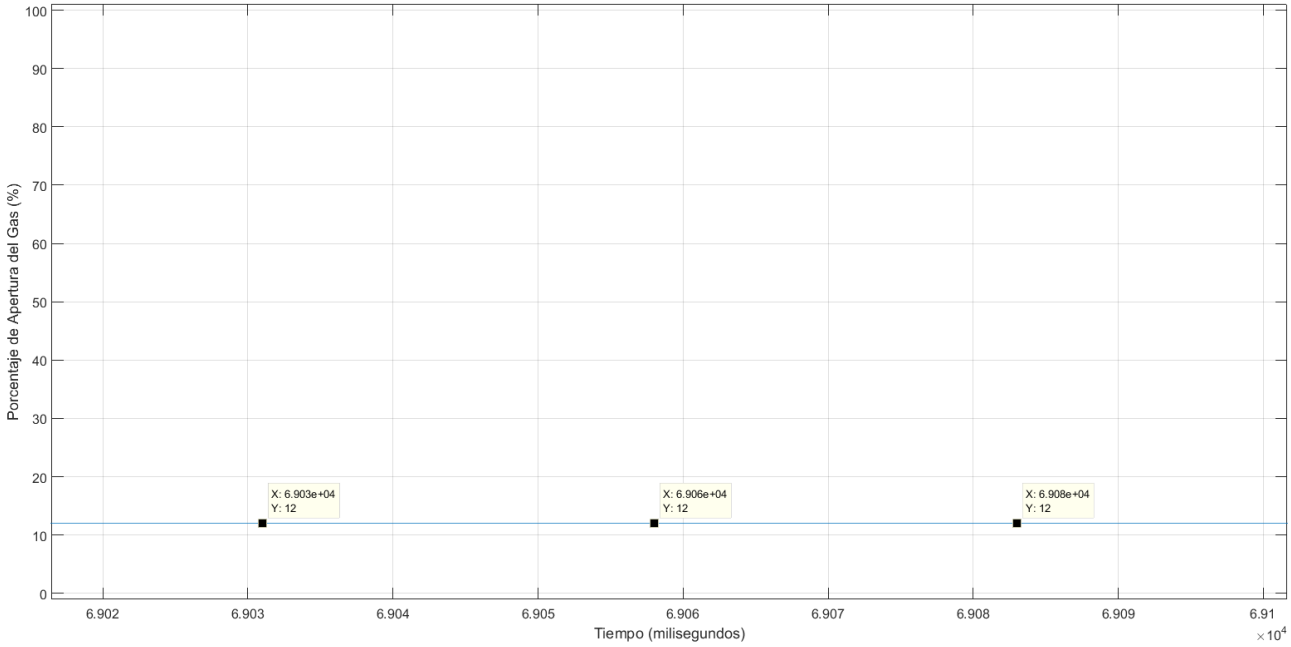


Figura 65: Detalle de los datos representados en la gráfica de la Figura 64.

Teniéndose las marcas de tiempo se puede calcular cuanto tiempo en milisegundos han pasado entre medidas que, para la lectura de todas las medidas oscila entre los 24 y 26 milisegundos. Este tiempo cumple con creces, en un principio, el objetivo de una muestra cada décima de segundo.

Pero cuando se trata el logado de datos del GPS, los resultados son muy diferentes. El GPS, realiza un logado de datos mucho más lento, con una diferencia de tiempo entre muestras de entre medio y un segundo. Esto realmente era un comportamiento esperado y, en cierta manera suficiente, pero al realizar un código conjunto se descubre que no es posible hacer el logado de los sensores y GPS al mismo tiempo.

Esto sucede, como se verá cuando se presente el código completo de GPS, porque el Arduino debe mantener la comunicación serie con el GPS en todo momento dado que desde que se socilita el dato de posición hasta que se obtiene pasa ese tiempo que lo hace incompatible con la lectura rápida de sensores.

3.4.2.3 Solución propuesta a este problema

La solución propuesta a este problema no es otra que la adoptada para el problema de incompatibilidad entre la pantalla y el datalogging, es decir, la de dividir las dos acciones en dos subsistemas. De este modo, se haría un *datalogging* de los sensores en una SD con un Arduino y, lo mismo, con el GPS.

El primer subsistema es el que se ha presentado para hacer pruebas, es decir: los sensores y la tarjeta SD que, con el código que se ha presentado, se guardarían en formato *string* en un archivo.*txt.* Hace falta especificar que, se guardan todos los datos en formato entero para luego tratarlos en el *software* de ordenador que interpreta y representa los mismos. Un ejemplo de uno de los archivos de datos generados en formato de texto es el mostrado en la *Figura 66* a continuación:

```
29,0,0,2,1023,751;
29,0,0,2,1023,776;
29,0,0,2,1023,799;
29,0,0,2,1023,823;
29,0,0,2,1023,847;
29,0,0,2,1023,871;
29,0,0,2,1023,896;
29,0,0,2,1023,919;
```

Figura 66: ejemplo de como se almacenan los datos.

El segundo subsistema que se va a tratar es el que realiza el *dataloggin* del GPS, en este caso es una programación ligeramente más compleja que para los cinco sensores y necesitará de una pequeña explicación. El objetivo del código que se ha utilizado es el de adquirir datos del GPS y guardarlos en la tarjeta microSD con el mismo formato que los sensores para tratar todos los datos de una manera más homogénea.

La primera parte del código desarrollado es en la que se incluyen librerías y definen variables. Cómo en otros casos, el código se ha realizado siguiendo un enfoque incremental, comenzando por versiones más simples hasta llegar a la actual. Es por esto que, puede que algunas de las variables que se hayan en el código, en la versión presentada, estén en desuso. La pimera parte del programa es la siguiente:

```
#include <SPI.h>
#include <SD.h>
#include <SoftwareSerial.h>
const int chipSelect = 4;
int contador=0;
float valor=1;
String dataString;
// Utiliza la función gpsSerial (de la biblioteca SoftwareSerial para
//establecer los pines de la comunicación serial que va a usar el GPS.
SoftwareSerial gpsSerial(11,10); // RX, TX (TX no se usa)
const int sentenceSize = 80;
char sentence[sentenceSize];
```

Una vez se han definido las variables y librerías a utilizar, se pasa al bucle llamado *setup* que se ejecuta una sola vez y que, en este caso, incializa la tarjeta SD. En este caso, si falla la tarjeta el programa se detiene y no realiza ninguna acción más. El bucle de configuración es el siguiente:

```
void setup() {

  Serial.begin(9600);
  while (!Serial) {
    ; // espera que conecte el puerto serie
  }
  Serial.print("Initializing SD card...");

  // comprueba que la tarjeta esté presente y se pueda utilizar
  if (!SD.begin(chipSelect)) {
    Serial.println("Card failed, or not present");
    while (1);
  }
  Serial.println("card initialized.");
  gpsSerial.begin(9600);
}
```

El otro bucle principal de la programación realizada por Arduino es el que se ejecuta de manera iterativa, será en este cuando se realice la lectura del GPS y su almacenamiento en la tarjeta SD. Para poder explicar de mnera precisa cada parte del código, se van a utilizar comentarios dentro del mismo código. El bucle que se repite de manera periódica es el mostrado a continuación:

```
void loop() {
static int i = 0;

//Comprueba si la comunicación serie del GPS está disponible y de ser así,
//calcula el largo de la "frase" de datos que da el GPS.

  if (gpsSerial.available())
  { char ch = gpsSerial.read();
    if (ch != '\n' && i < sentenceSize)
    {
```

```
      sentence[i] = ch;
      i++;    }
    else
    {

//Si se llega al final de la frase de datos, llama a la función que
//representa y guarda los datos del GPS.

sentence[i] = '\0';
     i = 0;
     displayGPS();
     }
  }
    }
```

Una vez ha terminado el bucle *loop,* en el que se realiza la llamada recurrente a funciones para la adquisición de datos, se presenta la función *displayGPS* utilizada para representar los datos obtenidos por el puerto serie (para comprobar el funcionamiento) y, además, los alamacena en la tarjeta externa.

```
 void displayGPS()
{
  char field[20];
  dataString="";
  getField(field, 0);
  if (strcmp(field, "$GPRMC") == 0)
  {
// llama a la función getfield que isla el campo concreto de la frase
//completa que da el GPS a la salida.
    Serial.print("Lat: ");
    getField(field, 3);
// concatena el campo obtenido al final de un string de datos que,
//finalmente, será el que se guarde.
    dataString += String(field);
    dataString += String(",");
    Serial.print(field);
    getField(field, 4); // N/S
    Serial.print(" ");
    Serial.print(field);
    Serial.print(" Long: ");
    getField(field, 5);  // number
    dataString += String(field);
    dataString += String(";");
    Serial.print(field);
    getField(field, 6);  // E/W
    Serial.print(" ");
    Serial.print(field);
    Serial.println();
    Serial.println(dataString);
// Una vez se tiene el string completo con todos los datos se abre el archivo
//txt de la microSD y se escribe en una línea nueva.
    File dataFile1 = SD.open("gps.txt", FILE_WRITE);
       if (dataFile1) {
        dataFile1.println(dataString);
        dataFile1.close();
        }
        else {
         Serial.println("error opening datalog.txt");
          }
  }
}
```

La última función que se va a comentar para esta programación es la que recorta la tira de datos en forma de frase (que da el GPS) en diferentes campos. Esta función es muy importante para el código desarrollado, puesto que gracias a ella se pueden consultar y guardar datos que sean interesantes por separado, en lugar de tener que trabajar con todos los datos (habiendo algunos que carezcan de interés como la hora o la fecha).

Su funcionamiento a grandes rasgos es el de encontrar las diferentes "," que tiene la tira de datos, dividiéndola, sabiendo que hay un dato antes y otro después de cada ",". La función resulta de la siguiente manera:

```
void getField(char* buffer, int index)
{
  int sentencePos = 0;
  int fieldPos = 0;
  int commaCount = 0;
  while (sentencePos < sentenceSize)
  {
    if (sentence[sentencePos] == ',')
    {
      commaCount ++;
      sentencePos ++;
    }
    if (commaCount == index)
    {
      buffer[fieldPos] = sentence[sentencePos];
      fieldPos ++;
    }
    sentencePos ++;
  }
  buffer[fieldPos] = '\0';
}
```

Para el caso que ocupa a este trabajo, los únicos datos que se han almacenado del GPS son la latitud y la longitud, añadiéndole las modificaciones necesarias (si es que se requiere alguna) para que se puedan representar correctamente. Una vez se tienen en el *txt* y se guardan en el ordenador, se representar utilizando el programa *Matlab.* Para representar los datos, la técnica utilizada habitualmente para series de dos datos suele ser la función *plot,* si se utiliza se obtienen la representación gráfica de la *Figura 67* de justo debajo.

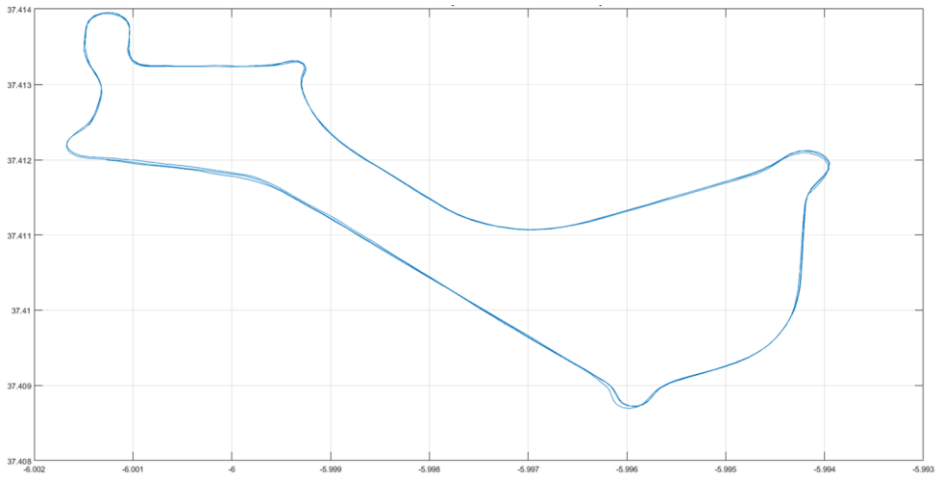


Figura 67: Representación de los datos GPS con la función "plot".

Como se puede intuir y apreciar, esta representación no es del todo correcta dado que, por la naturaleza de los datos obtenidos (y del modo en el que se obtienen), el tamaño unitario de las coordenadas verticales y horizontales no es el mismo. Esto resulta en una representación poco precisa de las distancias, quedando unas relaciones de aspecto enrarecidas.

Para solucionar esto, dentro del programa *Matlab* existe una función para representar coordenadas geográficas denominada *geoshow*. Utilizando los mismos datos, pero represetnándolos esta vez con la función adecuada la figura resultante es la *Figura 68*:

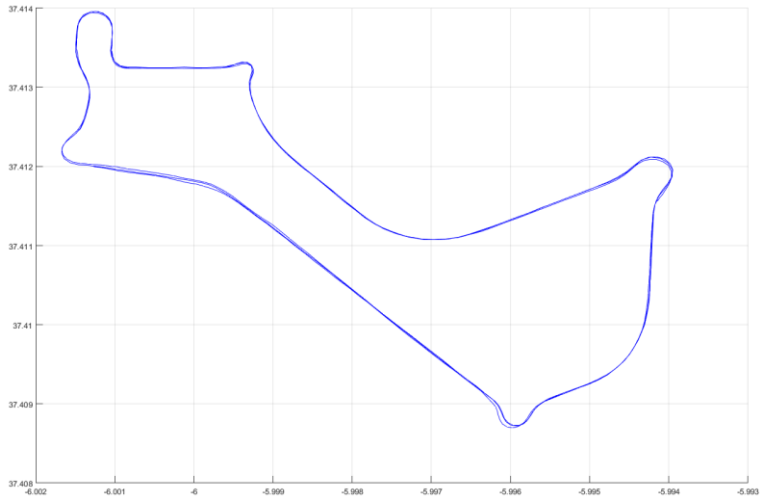


Figura 68: representación de los datos del GPS con la función geoshow

Para esta última representación, la relación de aspecto y de dimensiones entre sí es bastante mejor y más precisa que en el caso anterior. Para confirmar esto, además de valorar otros aspectos como la precisión, se van a representar los datos de posicionamiento obtenidos en *Google Maps* utilizando una página web de dominio público.

Los datos adquiridos y mostrados en la *Figura 68*, se tomaron utilizando el sistema pleanteado y un coche en las inmediaciones de la *Escuela Ténica Superior de Ingenieros*, realizando un recorrido cíclico para asemejar lo máximo la toma de datos a la que se haría en un circuito. Además, de este modo, haciendo variaciones voluntarias entre vueltas se podría ver la precisión real del GPS.

Los datos, representados en el mapa utilizando la herramienta antes mencionada, resultan de la manera mostrada en la *Figura 69*.

Figura 69: Representación de los datos de GPS utilizando una aplicación web libre denominada: "www.gpsvisualizer.com".

A simple vista es posible apreciar que, la representación que se realizó utiliando el comando *geoshow* es prácticamente exacta a la del mapa web. Por lo que se confirma que esta herramienta es más adecuada que el comando *plot*.

Por otro lado, realizando zoom dentro de la propia representación web puede hacerse una estimación de la precisión del GPS ante variaciones. En términos de geolocalización, la posición obtenida es precisa siendo capaz de localizar el vehículo en el que se probó en todo momento, cuadrándolo incluso dentro de los carriles viales en los que se movía.

Este último hecho ya implica un error de menos del ancho de la vía, puesto que, de ser mayor el coche sería localizado fuera de la misma y esto no sucede. Teniendo en cuenta el tamaño nominal de un carril, el número de carriles y el arcén, para que el vehículo no quede localizado fuera de la carretera, en esta prueba, se requiere un error de menos de aproximadamente 6 metros.

Haciendo zoom (*Figura 70*) y observando con mayor detenimiento los resultados obtenidos, se puede tener una mayor aproximación del error que puede tener este sistema GPS utilizado.

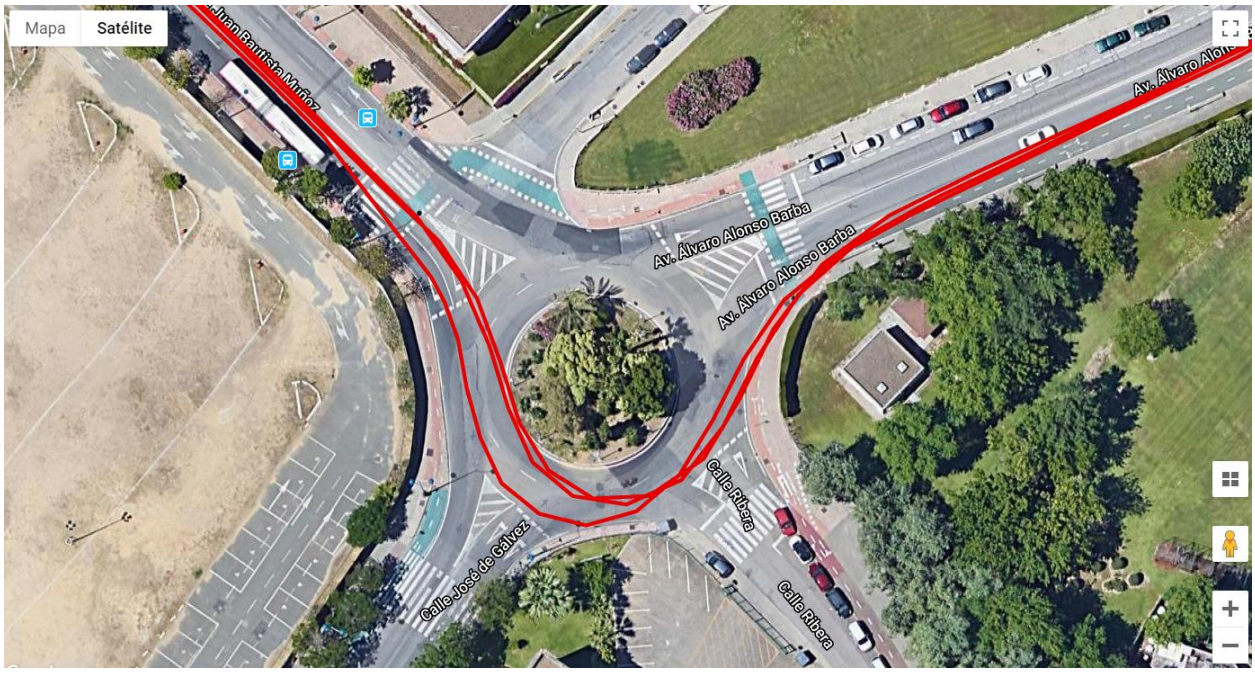


Figura 70: Detalle del GPS representado en la herramienta web antes mencionada.

Observando las diferentes líneas que el GPS ha conseguido diferenciar, se puede inferir que el sistema utilizado tiene una precisión elevada, con un margen de error de quizás un metro. De hecho, algunos de los errores que se pueden apreciar son producto de la linealización entre muestras, no realmente de lectura.

Una vez se ha conseguido realizar una adquisición de datos que se podría considerar efectiva (a falta del análisis final de los resultados obtenidos que se hará más adelante) se va presentar los montajes del hardware utilizado.

3.5 Esquema de Montaje del Hardware

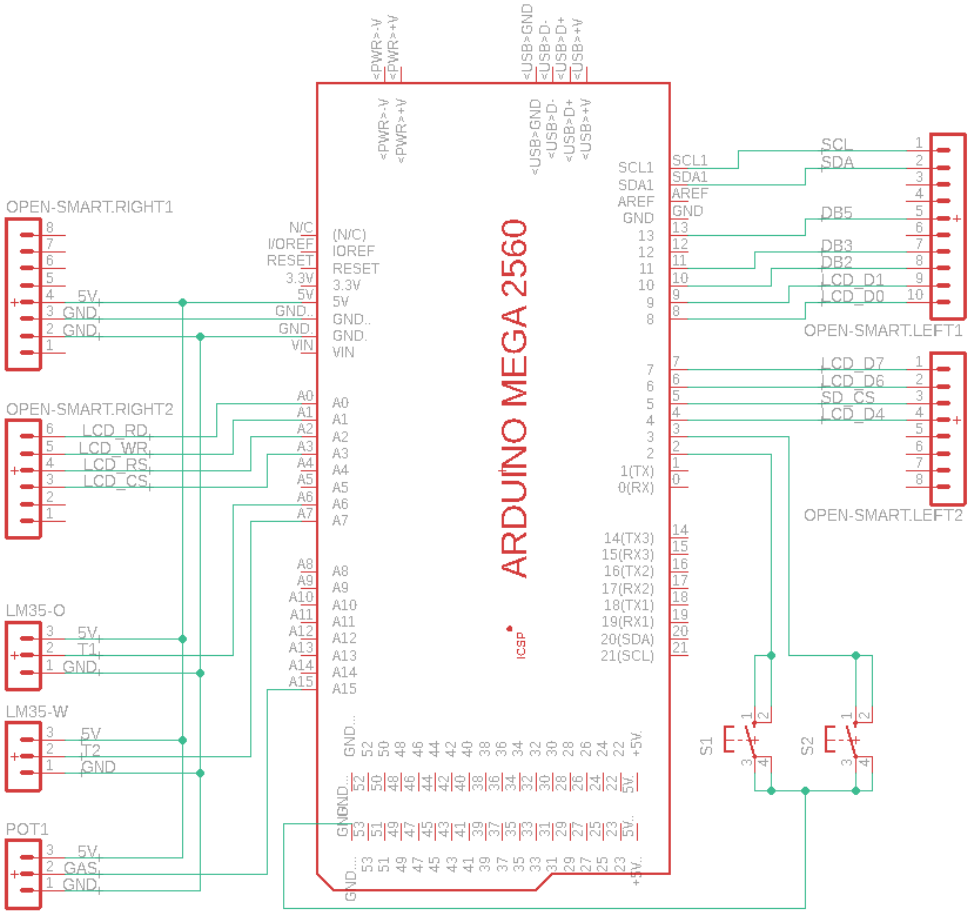
A lo largo de este apartado se van a presentar los diferentes esquemas de montaje para lo que han sido los tres subsistemas o módulos diferentes que se han desarrollado en el apartado anterior referido al *software.* Como ya se ha expresado en este trabajo, para los sistemas de pantalla y adquisición de datos se han creado tres módulos diferentes, cada uno con sus dispositivos, software y montaje propios.

Los diferentes montajes que se han realizado han sido los que se van a presentar a continuación en los que se ha tratado de que el conexionado coincida al máximo con los códigos presentados.

Sin embargo, es posible que la coincidencia no sea al 100% exacta dada la gran cantidad de pruebas que se han realizado hasta llegar a las versiones finales presentadas.

3.5.1. Esquema de montaje de la pantalla

Como su propio nombre indica, en este subapartado se va a presentar el esquema de conexionado que se ha utilizado para probar, depurar y diseñar todo el submódulo de pantalla. La pantalla se ha comprador en formato *boosterpack*, es decir, como un módulo que se conecta encima del Arduino tapándolo por completo, aunque no usa todos los pines del mismo (se incluyen más de la cuenta para dar rigidez a la estructura). Los pines de dicho *boosterpack* se han denominado OPEN-SMART (nombre de la pantalla) y se han etiquetado por el lugar que ocupan (derecha, izquierda, etc).



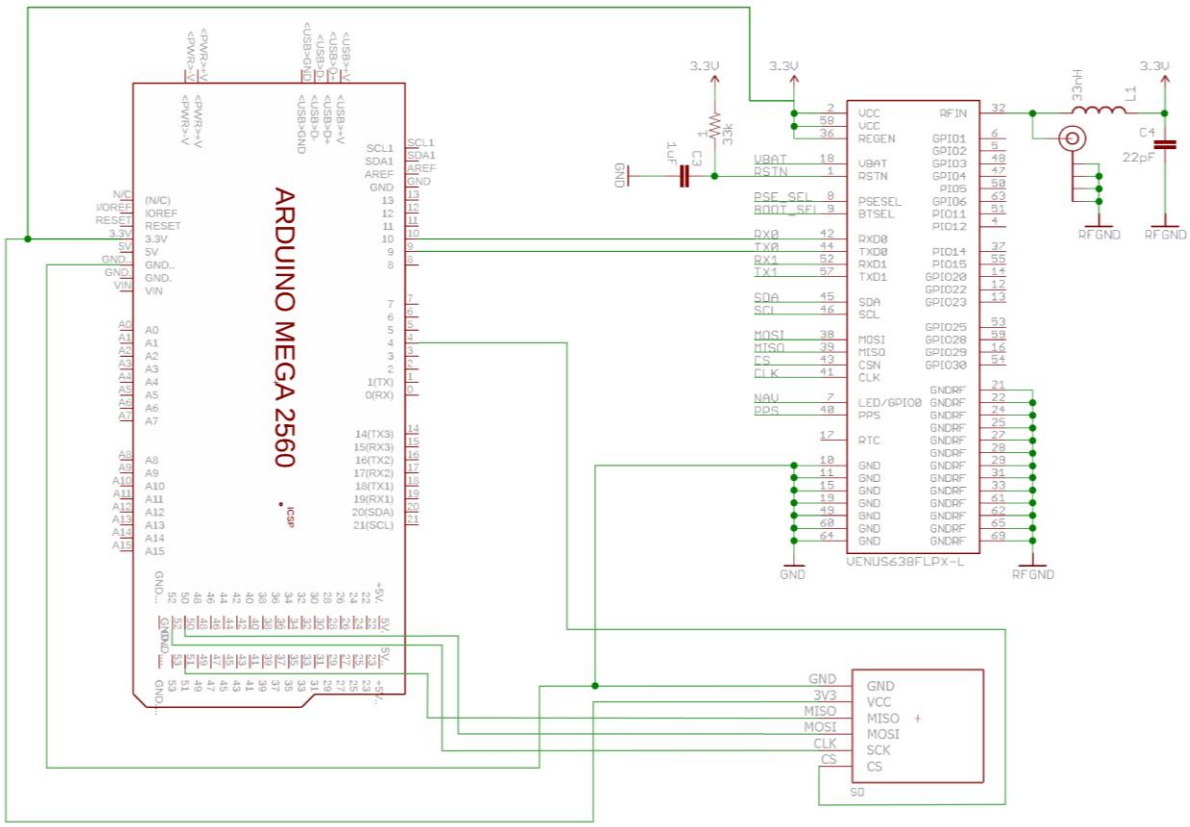
Montaje 1: Esquema de conexionado de la pantalla, sensores mostrados por la misma y botones para la selección de marchas.

Como se ha indicado en el pie de foto, se han incluido los sensores de temperatura mostrados LM35-O (temperatura de aceite), LM35-W (temperatura de agua) y el sensor del gas (POT1). Además, se han añadido los dos pulsadores que invocaban las interrupciones del Arduino para el conteo de marchas.

3.5.2. Esquema de montaje del GPS

Este es un montaje con menos elementos que el anterior, en el que priman los elementos GPS y la tarjeta SD para el guardado de los datos de posición. El Arduino se vuelve a postular como elemento central en torno al que gira todo el montaje.

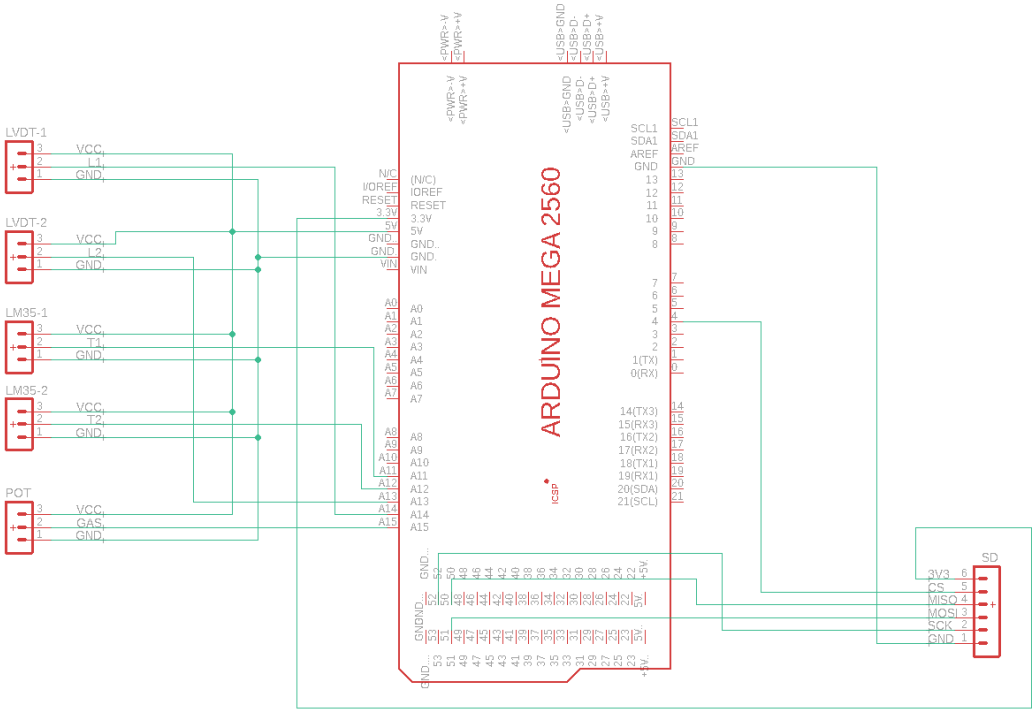
Como curiosidad hay que añadir que, el GPS tiene muchos más pines de los utilizados dado que permite ser conectado a varios sistemas a la vez si fuera necesario.



Montaje 2: Esquema de conexionado del sistema de GPS Venus y del lector de tarjetas microSD.

3.5.3. Esquema de montaje del *datalogger* de los sensores

Este último esquema pertenece al tercer módulo encargado de leer los valores de los sensores y guardarlos en la tarjeta SD para su posterior presentación en un *Software* de PC realizado también en este trabajo, detallado más adelante.



Montaje 3: Esquema de conexionado de los sensores y el lector de tarjetas microSD.

3.6 Software Creado para la Representación de Datos en PC

Cómo se ha expresado con anterioridad, una de las necesidades del proyecto, que nace inherentemente dada su tipología, es la de poder representar de manera sencilla y clara todos los datos obtenidos. Para ello, se recurre al programa informático *Matlab* para hacer dicha representación.

Teniendo en cuenta el útimo rumbo que ha tomado la programación de la adquisición de datos una vez solventados los problemas de compatibilidad, era necesario hacer un programa o *script* de *Matlab* que leyera ambos archivos de dato (en formato txt) y representase los datos, sin más que ejecutarlo.

Dentro de todas las opciones que se podán utilizar, se ha de admitir que, a priori, *Matlab* no es una opción ligera en cuanto a términos de programa base, ni muy versátil para el usuario medio de un ordenador personal. Pero, observándolo en el contexto del uso que se le iba a dar, al ser una escudería de motociclismo dentro de la Universidad de Sevilla y, más concretamente, de la Escuela Técnica Superior de Ingenería, era un programa conocido y utilizado por todos. Llegando al punto de que, se tomara el ordenador que se tomara, cualquier miembro del proyecto tendría dicho *software* instalado.

Teniendo en cuenta lo anterior, y en este contexto concreto, el programa realizado tendría unos escasos *kylobytes* de tamaño siendo altamente portable de un ordenador a otro (dado que el requisito de tener *Matlab* puede ser sacado de la ecuación).

Una razón añadida para el uso de este programa como base de la representación de los datos, es la gran potencia para el tratamiento de datos y la gran variedad de opciones que tiene *Matlab*. De hecho, es tan potente que podría llegar a utilizarse recursos como interpolaciones (del orden que se requiera) para mejorar las prestaciones del sistema.

3.6.1. Explicación del Código Realizado

Como en ocasiones anteriores en las que se ha detallado alguno de los *softwares* utilizados, se va a detallar parte por parte, en lugar de todo de golpe, al ser una programación no trivial. El comienzo del código de *Matlab* que se ha desarrollado es el siguiente:

```
%% Titulo :Script Telemetria
clear; clc;
%% Introduce nombre de archivos
name_tel='DATALOG_3.txt';
name_gps = 'gps.txt';
```

En estas primeras líneas simplemente se limpian las variables y el espacio de trabajo de *Matlab* y se definen los nombres de los archivos txt que se quieren abrir.

```
%% Abre el txt de los datos y los guarda por columnas en una
%%matriz llamada A
fileID = fopen(name_tel,'r');
format = '%d,%d,%d,%d,%d,%d;\n';
sizeA=[6 Inf];
A = fscanf(fileID,format,sizeA);
fclose all;
A = A';
%% Separa por columnas los valores enteros guardados
Temp1_abs = A(:,1);
Temp2_abs = A(:,2);
Front_abs = A(:,3);
Rear_abs = A(:,4);
TPS_abs = A(:,5);
time_abs = A(:,6);
```

Las operaciones anteriores realizadas son la apertura del archivo de los datos de telemetría para, dándole a *Maltlab* el formato que tienen esos datos, copiarlos en una matriz.

Una vez hecho esto, se divide esa matriz en vectores, dado que cada fila (a partir de ahora vector) son las lecturas de cada sensor. Estos datos que se han exportado a vectores están en valores entre 0 y 1023 que son como se guardaron, en este caso, de Arduino.

```
%% Varía los valores para ponerlos en el SI
Temp1_SI = (A(:,1)*5.0*100.0)/1023.0;  % grados Centigrados
Temp2_SI = (A(:,2)*5.0*100.0)/1023.0; % grados Centigrados
Front_SI = (A(:,3)/1024.0)*150.0; % milimetros
Rear_SI = (A(:,4)/1024.0)*150.0; % milimetros
TPS_SI = (A(:,5)/1024.0)*100.0; %porcentaje de gas
time_SI = A(:,6)/1000.0; %tiempo en segundos
```

En la sección de código anterior se pasan los valores leídos al sistema internacional de unidades, según la magnitud físicas que lea cada sensor en cada caso. En las siguientes líneas de código que se van a analizar, se han utilizado dos subfunciones de Matlab, una llamada *corregir_tiempo* y otra llamada *graf_telemetría,* que se detallarán y explicarán de manera más extensa una vez se termine la función principal.

```
%% Se corrige el tiempo leido del arduino
corregir_tiempo;
%% Representacion con el tiempo corregido
Datos1=[Front_SI, Rear_SI,TPS_SI];
Datos2=[Temp1_SI, Temp2_SI];
graf_telemetria(t_corregido/1000.0,Datos1,Datos2);
```

En la función *graf_telemetria* se establece una figura de dos ejes para organizar los datos de la manera más conveniente, por lo anterior, en la sección de código mostrada justo arriba se organizan en dos grupos llamados *Datos 1* y *Datos 2* siendo los primeros los sensores de gas y elongaciones y los segundos las temperaturas.

Una vez hecho esto y habiéndose representado los datos de sensores con la función ya nombrada, se pasa a la representación de los datos de GPS, utilizándose las líneas siguientes:

```
%% Representacion del GPS
%Abre el txt de los datos y los guarda por columnas en una
%matriz llamada A

fileID = fopen('gps.txt','r');
format = 'Lat:%2d%7fN Long:%3d%7fW\n';
sizeB=[4 Inf];
B = fscanf(fileID,format,sizeB);
B = B';
lat_min = B(:,2);
lat_deg = B(:,1);
long_min = B(:,4);
long_deg = B(:,3);

lat_dec=lat_min/60;
long_dec=long_min/60;

lat=lat_deg+lat_dec;
long=long_deg+long_dec;
long=-long;

figure();geoshow(lat,long);grid;
```

Este código sigue la misma estructura que el que tomaba los datos de los sensores y los organizaba y representaba, incluido la adaptación a las unidades de medida correctas y su posterior representación. Para representar los datos se ha utilizado el comando *geoshow* por las razones expuestas anteriormente cuando se explicó el GPS (para obtener resultados mucho más fidedignos que con el comando básico *figure*).

Una vez se ha programado la representación de datos, se va a pasar a un apartado del código un tanto innovador. El siguiente código genera un archivo *txt* con el formato necesario para subirlo a una página web de uso libre que permite ver los objetos dentro de *Google Maps* y, además, abre dicha página dejándolo todo listo para la subida del nuevo archivo. Dicho código es el siguiente:

```
%% Generacion archivo para la pagina web
 fopen all;
 n=length(lat)
 fileID = fopen('gps_web.txt','w');
 fprintf(fileID,'latitude,longitude \n');
 for i=1:n
 fprintf(fileID,'%f,%f \r\n',lat(i),long(i));
 end
 fclose all;
 web('http://www.gpsvisualizer.com/');
```

Una vez explicado el código principal, se van a detallar las dos funciones auxiliares que se han usado, siendo éstas: *corregir_tiempo* y *graf_telemetria.*

La función *corregir_tiempo*, como su propio nombre indica, se encarga de corregir la marca de tiempo que Arduino guardaba en la tarjeta SD junto a cada lectura de los sensores. Esta función nace de la necesidad de solventar un problema de serie que traía la placa de Arduino utilizada.

Tras realizar muchas pruebas diferentes de *datalogging* se observó que, el contador de tiempo (se usase la función *micros()* o *millis()*) se reiniciaba antes de llegar a su límite teórico. Para mayor complejidad, este reinicio del valor de contador de tiempo se producía de manera aleatoria, sin seguir un patrón concreto y, a veces, incluso en más de una ocasión cada par de minutos.

En la siguiente figura (*Figura 71*) se puede apreciar el valor del acumulador de la marca de tiempo, frente al número de muestra y, cómo se puede observar, en un punto concreto (inferior a los dos minutos) se reinicia yéndose a cero.

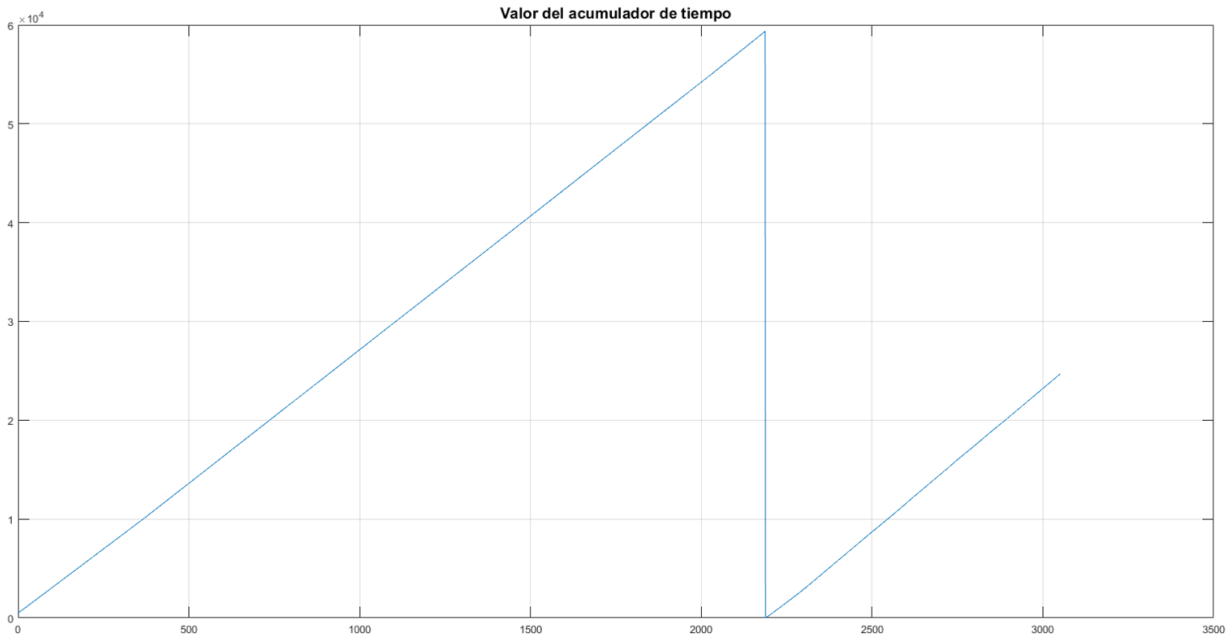


Figura 71: Representación gráfica del valor del tiempo obtenido del Arduino.

La función realizada toma el vector de tiempo guardado en la tarjeta SD y lo corrije para que no ocurra este fenómeno dado que, una vez que se reinicia habría dos instantes de tiempo con el mismo valor, resultándo en dos medidas distintas para el mismo tiempo, lo cuál no sería real.

El código realizado para dicha función es el siguiente:

```
%% correcion de tiempo
n = length(time_SI);
x=[0:1:n-1];
t_cortes = (0);
time_lapse=(0);
tramos=1;

%% Buscamos fallos en el reloj.
for N=2:n
if time_SI(N)<time_SI(N-1)
    t_cortes(tramos) = [N-1];%vector de ultimos instantes de cada tramo
    tramos=tramos+1; %contador de tramos
end
end
limite=length(t_cortes);

%% codigo de prubea
t_corregido=time_SI(1:t_cortes(1)); %primer tramo

%tramos intermedios
for Ntramos=1:length(t_cortes)-1
for N3=t_cortes(Ntramos)+1:t_cortes(Ntramos+1)
    t_corregido(N3)=time_SI(N3)+t_corregido(t_cortes(Ntramos));
end
end
%ultimo tramo
for N4=t_cortes(limite)+1:n
    t_corregido(N4)=time_SI(N4)+t_corregido(t_cortes(limite));
end
```

El código realizado lo único que hace realmente es, detectar el número de tramos diferentes de rampa que hay en el acumulador de tiempo. Lo detecta cuando un instante de tiempo tiene un valor mayor a uno posterior. Una vez tomados los diferentes valores de tiempo se van sumando a la primera rampa original para ponerlos justo a continuación, resultando una única rampa. En la *Figura 72*, se ha representado el valor del vector corregido frente al que se tenía originalmente, pero de manera gráfica.

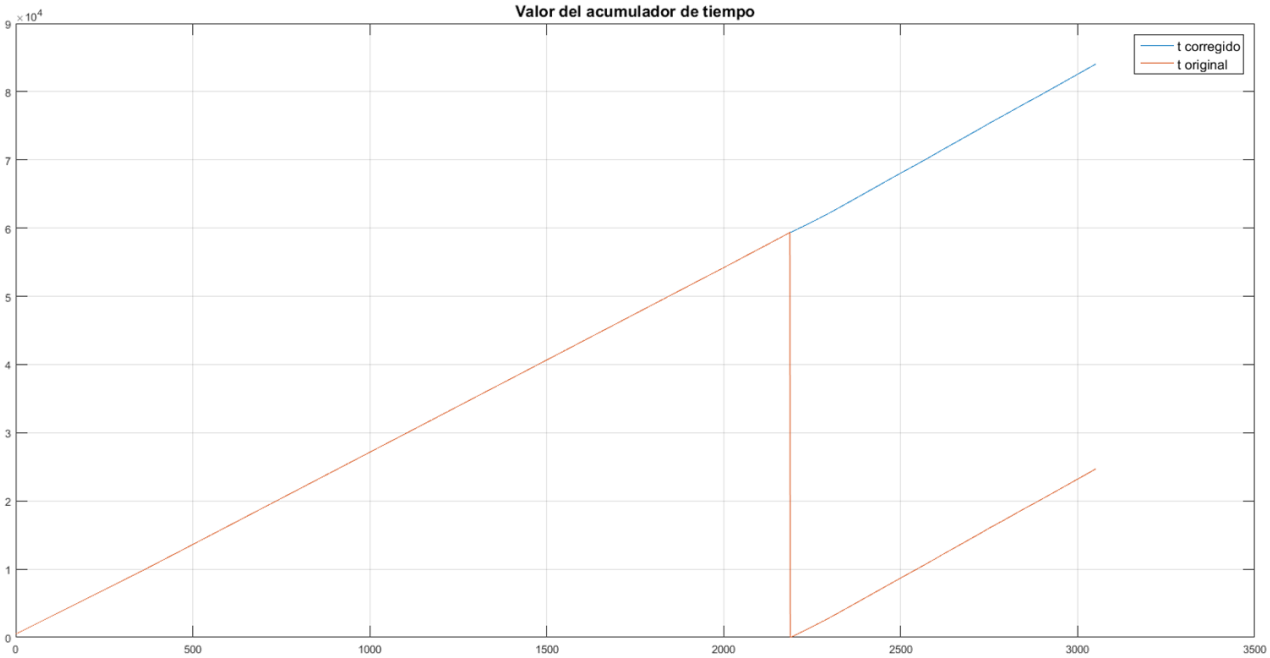


Figura 72: Representación gráfica del valor del tiempo obtenido del Arduino y su valor corregido.

La función *graf_telemetria* simplemente obtiene dos vectores de datos de entrada y un vector de tiempo y representa los datos en unos ejes determinados con la configuración de color y ejes deseados. No se ha explicado el código de manera extensa puesto que la configuración es meramente estética y con un fin de automatizar el proceso al ejecutar el *script*. Dicha programación es la siguiente:

```
function createfigure(X1, YMatrix1, YMatrix2)
%CREATEFIGURE(X1, YMATRIX1, YMATRIX2)
figure1 = figure('NumberTitle','off','Name','Telemetria','Color',[0 0 0]);
plotbrowser('on');

% Create axes
axes1 = axes('Parent',figure1,...
'Position',[0.0298672566371681 0.262008733624454 0.956305309734513
0.708717261691705]);
hold(axes1,'on');
% Create multiple lines using matrix input to plot
plot1 = plot(X1,YMatrix1,'Parent',axes1,'LineWidth',0.8);
set(plot1(1),'DisplayName','Front','Color',[1 1 0]);
set(plot1(2),'DisplayName','Rear','Color',[0 1 1]);
set(plot1(3),'DisplayName','TPS','Color',[1 0 1]);
box(axes1,'on');
grid(axes1,'on');
% Set the remaining axes properties
set(axes1,'Color',[0 0 0],'FontSize',12,'XColor',...
    [0.831372559070587 0.815686285495758 0.7843137383461],'XTick',...
    [0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
210 220 230 240 250],...
    'YColor',[0.831372559070587 0.815686285495758
0.7843137383461],'YTick',...
    [0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150],'ZColor',...
    [0.831372559070587 0.815686285495758 0.7843137383461]);
% Create legend
legend1 = legend(axes1,'show');
set(legend1,'TextColor',[1 1 1],...
    'Position',[0.909074106797461 0.855609275497467 0.0558628311166457
0.0891557471959527],...
    'FontSize',14,...
    'EdgeColor',[1 1 1]);
% Create axes
axes2 = axes('Parent',figure1,...
    'Position',[0.0282079646017699 0.0513100436681223 0.956858407079646
0.174672489082969]);
hold(axes2,'on');
% Create multiple lines using matrix input to plot
plot2 = plot(X1,YMatrix2,'Parent',axes2);
set(plot2(1),'DisplayName','Temp1','Color',[1 0 0]);
set(plot2(2),'DisplayName','Temp2','Color',[0 0 1]);

box(axes2,'on');
grid(axes2,'on');
% Set the remaining axes properties
set(axes2,'Color',[0 0 0],'FontSize',12,'XColor',...
    [0.941176474094391 0.941176474094391 0.941176474094391],'XTick',...
    [0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
210 220 230 240 250],...
    'YColor',[0.941176474094391 0.941176474094391
0.941176474094391],'YTick',...
    [28 29 30 31 32],'ZColor',...
    [0.941176474094391 0.941176474094391 0.941176474094391]);
```

```
% Create legend
legend2 = legend(axes2,'show');
set(legend2,'TextColor',[1 1 1],...
    'Position',[0.919387950144984 0.1587226313413 0.0553097337879965
0.0480349332614757],...
    'EdgeColor',[1 1 1]);
```

Este último apartado, es el responsable de representar los datos obtenidos por lo que, para evaluar su éxito o fracaso, habría que evaluar el del sistema completo. Por esto, se va a representar la gráfica que resulta de esta programación en el próximo apartado, destinado a la representación y analásis de los datos obtenidos.

4 Representación y Análisis de los Resultados Obtenidos

4.1 Introducción

En este apartado, que quizás se pueda antojar relativamente breve en comparación con los anteriores, pero no por ello menos importante, se van a representar y analizar los datos obtenidos y ponerlos en contraste con los objetivos iniciales.

Ya se han presentado resultados parciales en alguno de los apartados, pero en este caso se van a representar lo que se obtendría de utilizar el sistema completo. Primero, se ha de remarcar que el sistema utlizado debe ser aquel que solvente o esquive las incompatibilidades y problemas encontrados.

El único enfoque compatible con utilizar el sistema completo es utilizar la estrategia modular: un Arduino por tarea, es decir, uno para la pantalla, uno para el sistema GPS y un último para el guardado de los datos. Esto encarecería el sistema ligeramente pero, a priori podría hacer el sistema viable, si no completamente, parcialmente.

Tiene la ventaja, por otro lado, de que, si uno de los módulos funciona correctamente, toda la parte del proyecto entorno al mismo sería completamente válido. Teniendo en cuenta esto, le da al enfoque utilizado mucha cohesión con el trabajo realizado que no era otro que el de estudiar la viabilidad del sistema y sus partes.

Resumiento de nuevo el sistema completo del que se van a presentar los resultados obtenidos son:

1. Un módulo que se encarga de la gestión y dibujado de los gráficos de la pantalla.
2. Un módulo que realiza la lectura de los sensores y los almacena en una tarjeta microSD.
3. Un módulo que realiza la lectura del GPS y su almacenamiento en una tarjeta microSD.

Una vez obtenidos y recogidos los datos de manera correcta en un archivo *txt,* (como se ha expresado anteriormente en el documento) se pasa a la ejecución del *script* de Matlab que se ha descrito con anterioridad.

4.2 Representación de los datos obtenidos

Al ejecutar el *script* antes detallado y explicado, se abren varias ventanas, que se van a detallar una a una junto con sus funcionalidades. La primera de ellas, representada en la Figura 73 es una ventana de representación de los datos, con varios ejes y una columna a la derecha del todo que permite la selección de canal.

El *layout* o configuración gráfica elegida se ha realizado a imagen y semejanza de los programas de telemetría de referencia del mercado. Los datos se han dividido en dos, dadas las diferencias de magnitud tan elevadas, pero, para el caso de datos que tienen valores límite similares se ha usado a propósito el mismo eje.

Esto es así para facilitar la interpretación de los datos, pudiendo ver el máximo número de sensores superpuestos se puede entender mejor los eventos que suceden en la moto en cada instante. Éste es el caso del acelerador que está íntimamente relacionado con la compresión o extensión de las suspensiones en momentos críticos.

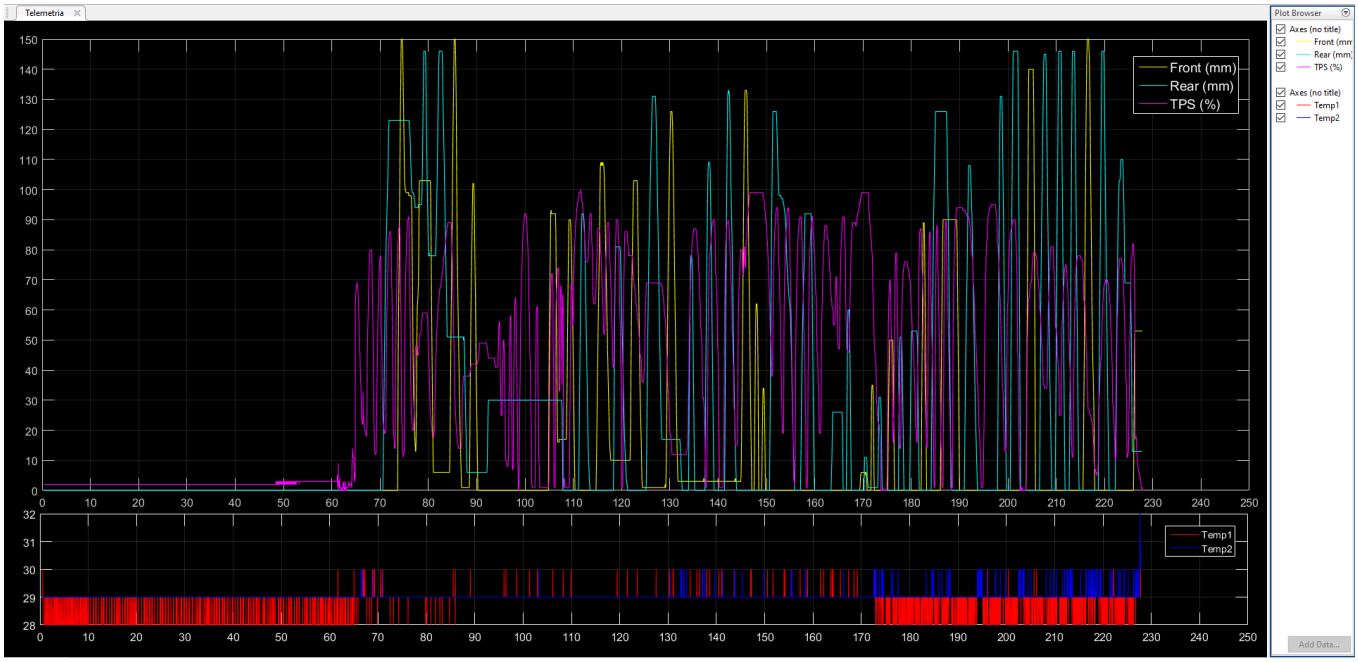


Figura 73: Representación de los datos obtenidos por medio de la herramienta Matlab.

Huelga decir que se ha representado en negro porque es así como se hace en los programas profesionales y se quería mostrar como es la ventana emergente que aparece en el ordenador, no cómo gráfica entregable de este trabajo.

De querer analizarse sensores por separado, se ha habilitado la vista por canales, de este modo pueden activarse y/o desactivarse ciertas curvas en el momento deseado. Un ejemplo de gráfica con algún canal desactivado es la mostrada en la Figura 74.

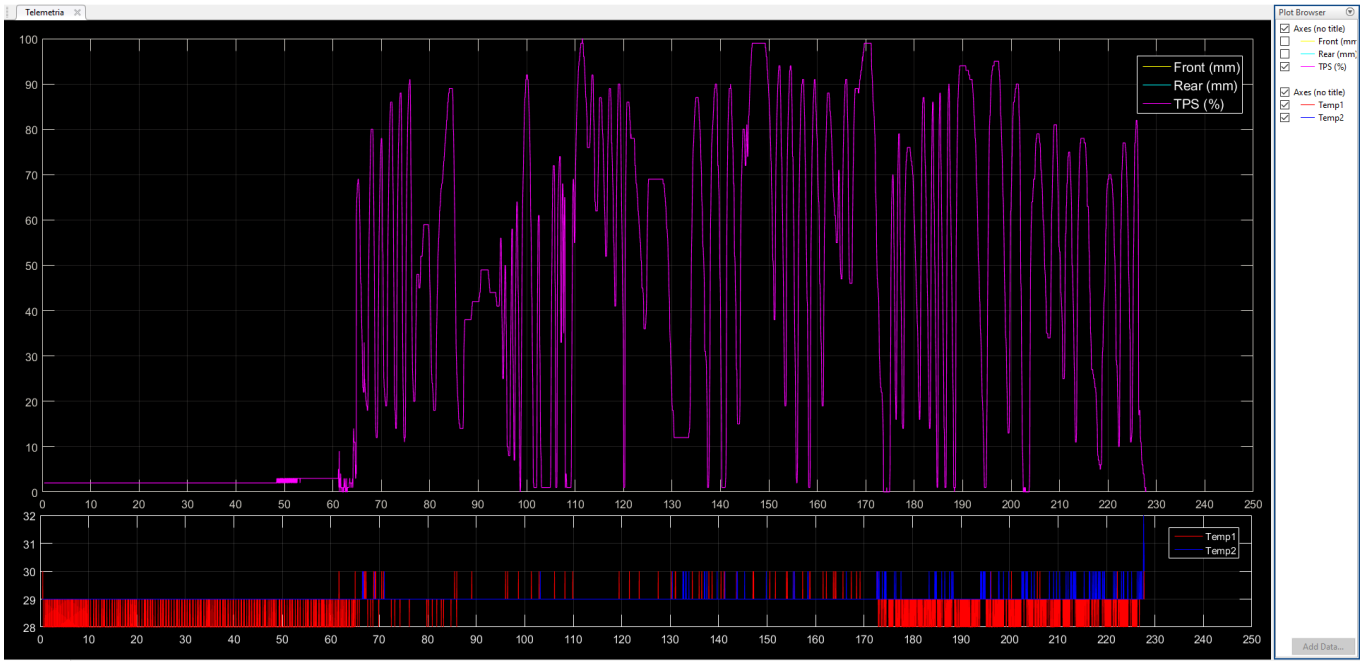


Figura 74: Representación de los datos de obtenidos, pero con un solo canal activado.

Como se puede ver además en la propia Figura 74, el elegir un solo canal o señal el sistema ofrece una ventaja más al hacer zoom de manera automática adaptándose a los límites del canal activo en cada momento.

Además de la pantalla que se genera, mostrada en la Figura 73, se genera además una pequeña ventana que da una ligera idea del recorrido que ha registrado el GPS. Como se expresó con anterioridad, se ha realizado dicha impresión por pantalla con la función *geoshow*. En este caso la gráfica al ser meramente informativa, sí se ha mantenido con un color blanco genérico, en lugar del negro que va más enfocado al análisis.

La ventana del GPS va a acompañada de una segunda ventana emergente que abre un navegador web embebido en *Matlab* para poder subir el archivo *txt* generado por el propio *script* que rectifica los datos de GPS para que sean legibles por la página.

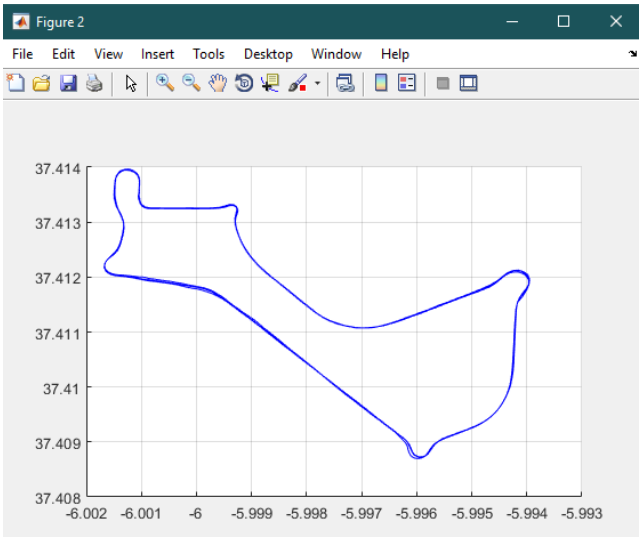


Figura 75: Representación de los datos GPS con el comando geoshow disponible en Matlab.

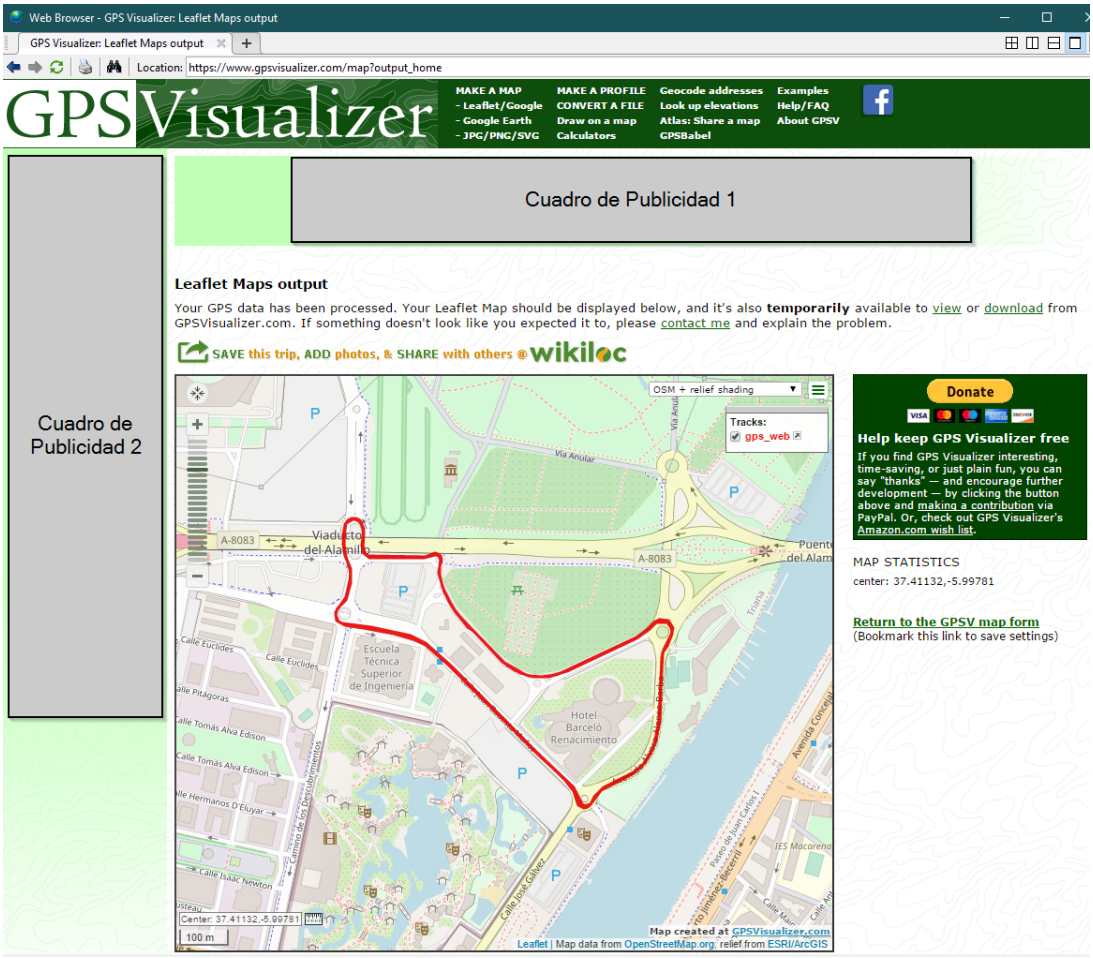


Figura 76: Representación de los datos GPS sobre un mapa para estudiar los datos con una mayor precisión al poder ver el lugar exacto en contexto que ha registrado el GPS.

La Figura 75 es la que da una ligera idea del recorrido del GPS y la Figura 76 es la ventana del programa con el mapa real y el recorrido sobre impreso en el mismo.

Hay que añadir que, enla web de uso libre www.gpsvisualizer.com que se puede ver en la Figura 76, se puede configurar el tipo de mapa que se quiere ver. Es posible utilizar la versión satélite del mapa para una mayor precisión, es éste el usado en epígrafes anteriores en los que se evaluó la precisión del GPS en cuestión.

Ya se hizo la comparación en su momento, pero, como se puede ver en ambas Figuras (75 y 76) se puede observar perfectamente que los datos gozan de una gran precisión al encajar a la perfección con la posición real del vehículo en el que se probó el sistema.

Una vez presentados los datos y resultados obtenidos se va a pasar al último de los epígrafes en el que se va a estudiar la viabilidad del proyecto teniendo en cuenta estos resultados y su análisis frente a os objetivos que se tenían inicialmente.

5 CONCLUSIÓN: VIABILIDAD DEL PROYECTO

Una vez se han mostrado los resultados obtenidos, llega el punto más importante de un estudio como el que se está realizando, que es la conclusión de si el proyecto persé es viable.

Para ello, es necesario volver a hacer un vistazo general del sistema que se pretendía evaluar. El sistema que se ha trabajado tiene como objetivo la adquisición de datos a bordo de una motocicleta de competición. Se pretendía además realizar una pantalla o *dashboard* que representara ciertos datos que fueran de interés para el piloto y el almacenamiento y representación de los datos mediante un PC.

Hay que añadir a lo anterior que, se establecieron unas prestaciones mínimas que de no alcanzarse el sistema sería catalogado directamente de inválido.

Los requisitos que se establecieron como umbral fueron los siguientes:

1. Pantalla: un refresco fluido de los elementos y gráficos.
2. Datos de sensores: 10 muestras de cada sensor por segundo.
3. GPS: una muestra por segundo y sobretodo una precisión con un error menor de un par de metros.
4. Software de PC: permitir una representación de datos correcta.

Para los sensores se ha de aclarar que, es virtualmente lo mismo decir 10 muestras de cada sensor por segundo que una muestra de todos los sensores cada décima de segundo por como se han muestreado los datos. Por lo que, de tomar un sensor de los datos obtenidos y probar que tiene una muestra por cada décima de segundo sería suficiente.

Se van a analizar por separado la viabilidad de cada uno de los puntos listados justo arriba, que además se van a tratar en orden cronológico de realización y exposición en este trabajo, dado que representan módulos o partes diferentes del proyecto que podrían funcionar por separado de ser viables.

5.1 Conclusión de la viabilidad del módulo: Pantalla

El caso de la pantalla es uno de los más sencillos dado que primero se va a adjuntar en este trabajo un video demostrativo del funcionamiento de la misma y, segundo, el modo en el que se realizó el desarollo.

Es interesante recordar que, la pantalla se programó siguiendo una estructura de versiones incrementales en la que cada versión contenía, prácticamente al completo a la anterior. Y, además, este paso se daba una vez que se había comprobado, mediante contínuas pruebas, que el resultado cumplía un mínimo de rendimiento.

Además de lo anterior, ha de recordarse que, para un mejor rendimiento y desempeño de la pantalla se programó utilizando una técnica “incremental”. En otras palabras, se decidió solo borrar y dibujar los elementos que modificaban su valor, en lugar de hacerlo con toda la pantalla.

Cuando se explicó esta técnica y cuando se aplicó se especificó concretamente que, de hacerlo así sí podía ser utilizable, pero de elegir borrar y dibujar toda la pantalla, no.

Por lo que, además de poder verse en el vídeo adjunto del funcionamiento de la misma, por la naturaleza del desarrollo y las técnicas utilizadas el uso de esta pantalla y su programación se puede entender como viable.

5.2 Conclusión de la viabilidad del módulo: Sensores

La conclusion a la que se ha llegado en el análisis de este modulo no es tan directa y absoluta como en el caso anterior. Para que el trabajo realizado sea considerado como viable se debería tener, como máximo una muestra cada décima de segundo, de tener un tiempo mayor entre muestras, directamente sería catalogado de inviable.

Si se observa con detenimiento la Figura 65, se puede observar que (como se especifica en el pie de página) las lecturas están separadas en términos temporales por entre 24 y 26 milésimas de segundo. Se puede añadir que, en la Figura 66 se puede er que todas las medidas se toman con la misma marca de tiempo, por lo que se confirma que es equivalente comparar una medida con todas para este análisis.

Teniendo en cuenta el aspecto meramente temporal se puede afirmar que, se ha cumplido con creces la restricción temporal que se estableció como umbral, pero no es el único aspecto a tener en cuenta.

Hay que recordar que, los sensores utilizados son de carácter analógico, siendo el conversor analógico-digital parte del Arduino o microcontrolador no del sensor mismo. El problema que se presenta para el uso de este tipo de sensores es que son altamente sensibles a ruidos de carácter electromgnético, vibraciones y demás.

Este estudio se realiza en un contexto de laboratorio en el que, cuando se obtienen los datos, no existen ruidos o perturbaciones de este tipo por lo que en las lecturas obtenidas no se puede apreciar esta limitación. Pero al montarse en un vehículo, si harían acto de presencia.

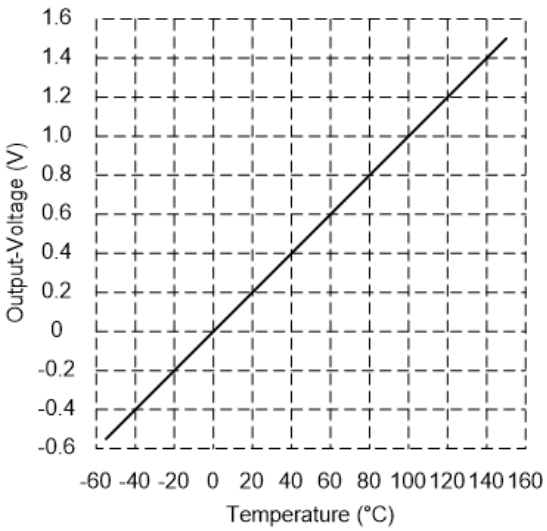


Figura 77: Recta voltaje temperatura característica del LM35.

Para el caso que ocupa a este estudio, todos los sensores se han alimentado con la salida de 5 V que proporciona la fuente de voltaje incluida en Arduino, aunque para el caso real se alimentarían con el mismo tipo de salida que tiene el cableado general de la propia motocicleta. A grandes rasgos, los sensores tienen un pin de alimentación y otro de tierra que tienen un voltaje fijo y los demás, variable, según el valor de la medida que se esté tomando.

En la mayoría de los casos los sensores no aprovechan o utilizan todo el rango de voltaje que hay entre los pines de alimentación y tierra para su pin de salida. Como ejemplo, una salida de 3V corresponde a un valor concreto de magnitud física (como puede ser la temperatura) y uno de 3.2V puede ser otro muchísimo mayor (o menor), por lo que no se guarda una relación lineal en casi ningún caso.

Teniendo en cuenta todo lo anterior, puede ser que las intereferencias electromagéticas induzcan a errores de medida en el pin de salida muy muy elevados por lo que el valor de dichas perturbaciones es crucial. Es complejo cuantificar este tipo de interferencias, pero como mínimo, se pueden estimar en medio voltio, en el menor de los casos.

Este ruido es crítico en el diseño y elecciones tomadas puesto que, si atendemos por ejemplo a la gráfica de voltaje de salia frente a la temperatura medida, o también llamada característica lineal del LM35 (termómetro utilizado) se puede apreciar que un error de 0.5 V a la salida haría que la medida fuera completamente falseada.

Si se atiende a la Figura 77, se puede apreciar que un voltaje de salida de 0.4 V correspondería a una temperatura de 40º C mientras que si se le suma (podría restársele también) un error de 0.5 V se tendría que para 0.9 V se tendrían entre 80 y 100º C.

Esto hace que el ruido y por ende el uso del termómetro elegido sea inútil puesto que se necesita una precisión de aproximadamente 1 o 2ºC dado que, si el agua de la motocicleta (por ejemplo) supera los 95ºC ya supondría un problema y de superar los 100ºC esta herviría y el motor quedaría completamente destruido por falta de refrigeración.

Para el resto de los sensores ocurre algo similar, pero en ninguno de los casos es tan crítico como en el caso de la temperatura de agua y aceite, encargados de refrigerar y lubricar el motor.

5.2.1. Solución propuesta: CAN Bus

La solución propuesta cambia radicalmente el proyecto, al menos en la parte de sensores dado que habría que cambiar tanto los tipos de sensores como el protocolo de comunicaciones. Aunque, es posible que no sea necesario cambiar la unidad de procesamiento utilizada puesto que existen proyectos de CAN-Bus basados en Arduino, aunque es cierto que existe la posibilidad de que el modelo utilizado no tuviera velocidad suficiente para hacer el *datalogging* que se pretende en este proyecto.

El protocolo CAN-Bus o también llamado ***Controller Area Network*** fue creado por la empresa alemana Bosch y está basado, como su propio nombre indica en la topología Bus para el envío y recepción de datos. Este protocolo tiene multitud de ventajas que lo hacen muy interesante para la aplicación que se estudia, de hecho, es el protocolo estandarizado y más utilizado tanto en el sector automovilístico como en el motociclístico. En este protocolo, una unidad central se comunica con diferentes sub-unidades que le facilitan la información cuando la unidad central la demanda (pudiéndose demandar solo a una fuente en cada caso).

Es un protocolo que tiene una larga historia dado que se lanza de manera oficial en el año 1986, desde entonces solo ha hecho más que evolucionar en todas sus capas (desde la física hasta la de enlace) y además han surgido continúas versiones y revisiones. Ha ido mejorándose con el tiempo y adaptándose a las circunstancias y tipos de proyectos, esto unido a que es un protocolo de comunicaciones además muy distribuido lo hace sólido y fácil de aplicar.

Algunas de las ventajas que ofrece esta solución de comunicaciones son las siguientes:

- Protocolo normalizado: facilita la implementación y búsqueda de subsistemas y dispositivos compatibles con el mismo.
- Red multiplexada: reduce la cantidad de cableado y complejidad necesarias.
- Alta inmunidad a interferencias: éste es el punto más interesante y determinante para el trabajo que se está realizando.

Por su interés para con el contexto de este trabajo, se va a detallar específicamente la última de las ventajas arriba expuestas para poder presentar mejor como soluciona el problema del ruido electromagnético que se tenía.

La resistencia a las interferencias que tiene este protocolo es debida a lo que se podría resumir en dos factores: el tipo de cables utilizados y los niveles de tensión.

Los cables utilizados son de la topología de par trenzado, configuración que hace que dismunyan las interferencias electromagnéticas notablemente. La pareja de cables utilizado tiene una notación concreta de cara al protocolo: CAN_H (*CAN High*) y CAN_L (*CAN Low*). Este tipo de configuración de cables es ampliamente utilizada, de hecho, una técnica similar de cable de par trenzado de cobre es la utilizada en las redes ADSL convencionales.

En cuanto a los niveles de voltaje, este protocolo utiliza principalmente dos estados: dominante y recesivo. En el estado recesivo, ambos cables tienen el mismo nivel de voltaje, mientras que en el estado dominante los diferentes niveles de voltaje que tienen CAN_H y CAN_L están separados como mínimo 1.5 V (se puede ver una representación de lo anterior en la Figura 78).

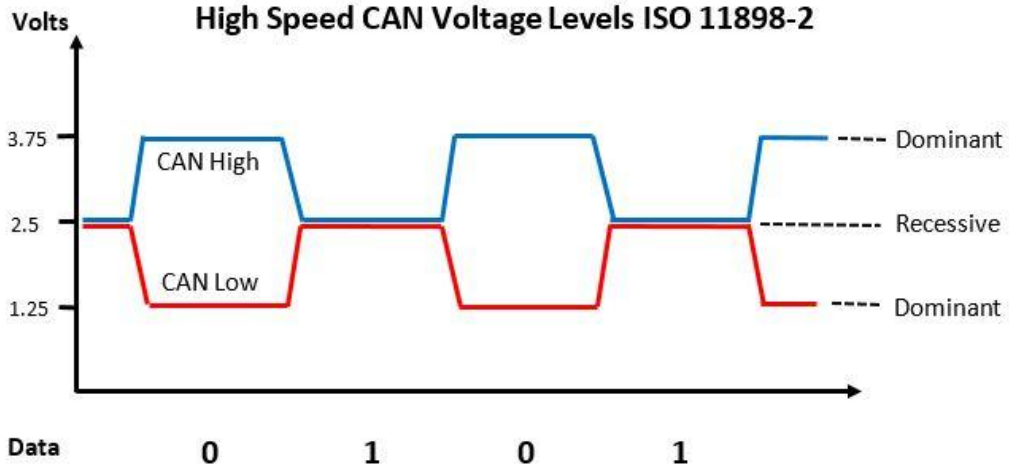


Figura 78: Niveles de tensión utilizados en el protocolo CAN-Bus.

Además, como se puede ver en la misma Figura 78, cuando ambos cables están en el mismo nivel de voltaje representa un 1 y cuando están con esa diferencia de voltaje antes descrita, un 0. De modo que, teniendo en cuenta que para interpretar un valor deben tener el mismo nivel de voltaje y para interpretar otro tienen que estar los niveles a una distancia mínima, aunque entrase alguna interferencia electromagnética esas dos premisas se seguirían manteniendo. Por lo que la medida no se vería afectada a menos que fuera una interferencia muy muy grande la que existiese.

Teniendo en cuenta todo lo anterior, y aunque no se puede asegurar que la solución propuesta sea totalmente efectiva si no se prueba, parece que al *CAN-Bus* podría ser una buena opción para la aplicación estudiada. Aunque es importante destacar que, cada sensor o dispositivo de medida debería poder comunicarse dentro de una red CAN-Bus y ser en realidad un subsistema microprocesado en sí mismo que le permita entrar en la red maestro-esclavo formada. Por lo que, los sensores analógicos (al menos los utilizados) no tendrían cabida en esta red CAN.

5.3 Conclusión de la viabilidad del módulo: GPS

Este es uno de los módulos que major ha cumplido las especificaciones trazadas al principio del trabajo, por lo que, a priori se puede establecer directamente que es uno de los considerados como uno de los módulos viables.

Al comienzo del trabajo se establecía que, para dar como válido este modulo se requería un tiempo de muestreo de aproximadamente un segundo además de un error precisión de, como máximo, unos metros. El requisito de precision podría llegar, en realidad, a ser incluso más laxo dado que en un circuito tres o cuatro metros no supondrían un gran error.

De este modo y teniendo en cuenta las restricciones, se realizó la programación del modulo completo con lectura del GPS y su guardado en tarjeta SD. Alimentando el conjunto con un ordenador portátil, se realizó una prueba real en un vehículo por lo que, desde el punto de vista del ruido que afectaba al otro tipo de sensores, este módulo no se veía afectado.

Se hicieron dos tipos de pruebas, en recorridos cíclicos simulando un circuito y en recorridos lineales de un punto a otro. Para ambos los resultados fueron, como se ha mostrado ya, satisfactorios. En la Figura 79 se puede observar uno de los recorridos registrados como lineales que tiene varios kilómetros de distancia.

Figura 79: Recorrido completo realizado en la toma de datos experimental del GPS

Para este recorrido se ha escogido un formato de presentación de los datos diferentes dado que, al ser un viaje lineal y no cíclico, interesaba una herramienta que mostrase el inicio y el fin del recorrido y, además, una representación satélite en la que se pueden apreciar las vías y carreteras tomadas en el camino.

Como ya se pudo describir y explicar en el epígrafe dedicado a la programación del módulo GPS, este modulo si cumple las expectativas que tenían del mismo, como se puede ver en la Figura 70.

Como conclusión y, teniendo en cuenta tanto las especificaciones cumplidas como las pruebas realizadas en un ámbito real de aplicación, se puede decir que el módulo es perfectamente viable. Además, puede utilizarse por separado en cualquier aplicación de la misma índole.

5.4 Conclusión de la viabilidad del módulo: Software de PC

En este caso no se trata de un modulo físico dentro del Proyecto y trabajo que se está llevando a cabo pero se ha considerado como tal al ser una parte que se puede considerar estanca de las demás. De funcionar correctamente, al igual que el modulo GPS, podría ser utilizado para cualquier aplicación similar solo con introducirle los datos en el formato que los requiere.

Como requisitos para este módulo se estableció que, debía ser un *script* de *Matlab* que no ocupase demasiado espacio de manera que esto lo hiciera portable para poder tenerlo en cualquier PC con dicho programa base necesario.

Además, debía ser fácil de usar, con darle los datos de entrada y pulsar sobre “ejecutar” el programa debía presentar directamente todas las gráficas pedidas.

Y como último requisito, debía poder discernir la información representada por canales para poder elegir que sensores se querían ver y en que momento.

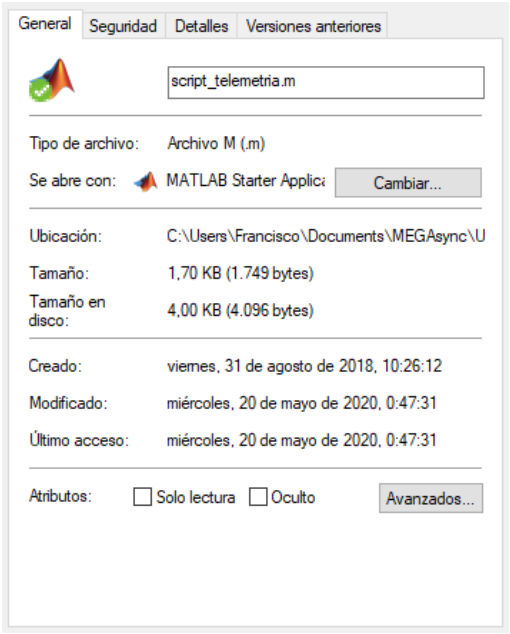


Figura 80: Detalles del archivo de Matlab.

La mayoría de resquisitos ya se han mostrado cumplidos a lo largo del propio apartado de explicación del trabajo realizado en torno a este módulo, el único que puede ser que no quede correctamente definido es el del tamaño de archivo y, por ende, su portabilidad de un PC a otro.

Observando las propiedades del archivo creado, que al final es el tamaño de datos que se necesitan transferir de un programa a otro, podemos encontrar si realmente es un programa fácil de transmitir o no entre equipos.

Viendo la Figura 80, se puede ver que el tamaño en disco del archivo (se ha tomado este por ser el mayor de los dos que se especifican) es de tan solo 4kB.

Dicho tamaño es minúsculo en la tecnología actual, por no decir despreciable. Podría llevarse en cualquier dispositivo *pendrive* del mercado, enviarse por correo o, como curiosidad, almacenarse en el chip que lleva el Documento Nacional de Identidad.

Por lo que, a pesar de poder sonar redudante, hay que concluir que este módulo es perfectamente viable y sólo habría que darle los datos de entrada en el formato que los necesita para poder representarlos. Cómo se hayan tomado dichos datos queda completamente aguas arriba de este programa para representación de datos.

5.5 Conclusión General de las Soluciones Adoptadas

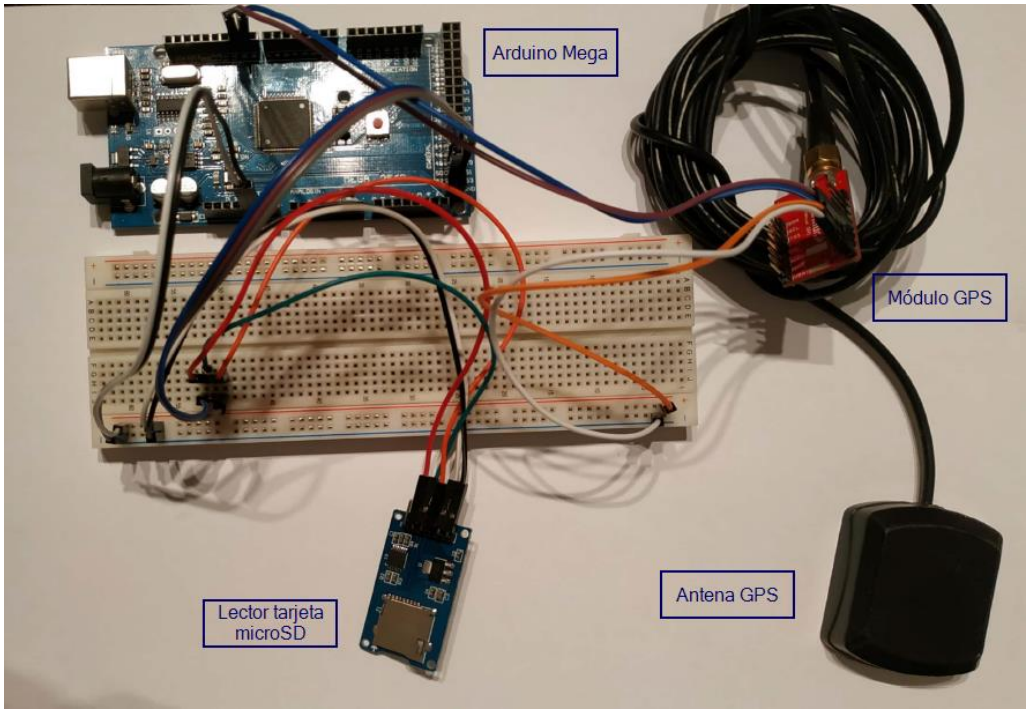
Una vez se han estudiado los módulos por separado, se debe hacer un acopio de toda la información para poder hacer una conclusión global de todo el trabajo que se hizo para poder estudiar la viabilidad o no del *Sistema de Adquisición de Datos para una Motocicleta de Competición* y de las soluciones que se adoptaron o proyectaron en su realización.

Observando módulo a módulo, las conclusions son las siguientes:

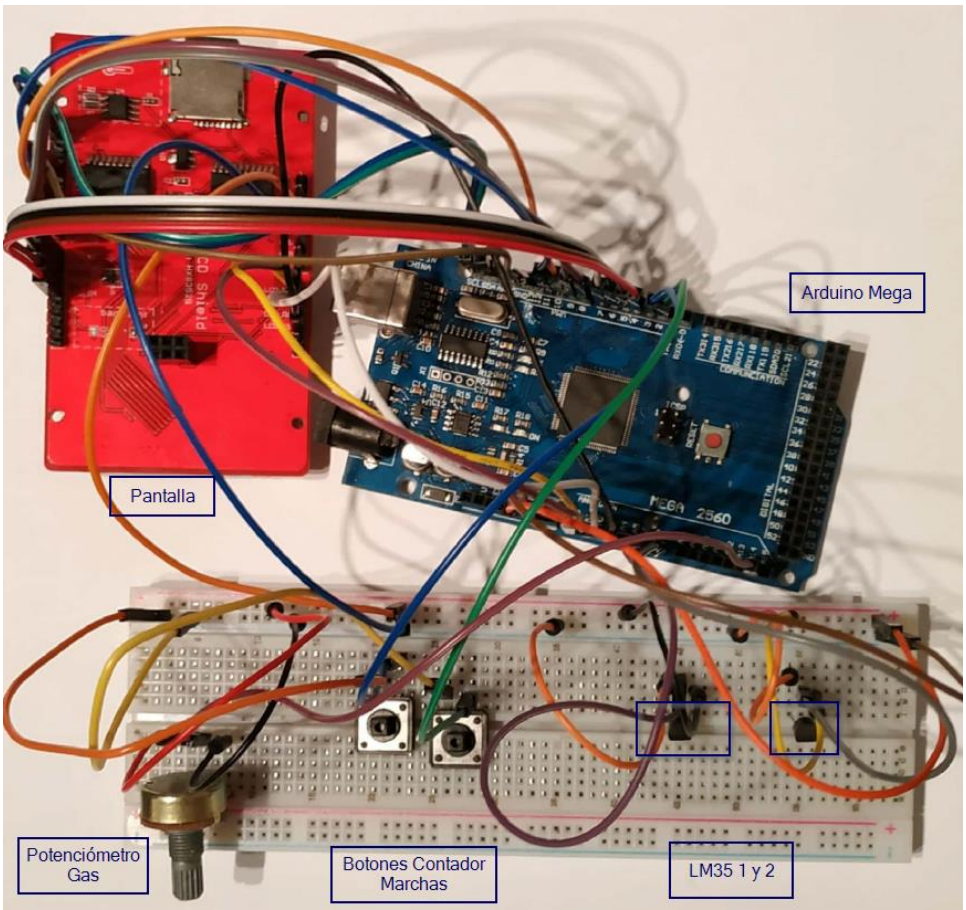
1. El módulo de los sensores no es aplicable a una aplicación real puesto que el tipo de sensores y el método de transmisión de datos no es adecuado. Se ha propuesto utilizar CAN-Bus y otro tipo de sensores.
2. Está directamente realcionado con el anterior el módulo de pantalla, teniendo en cuenta los sensores originales el módulo es perfectamente válido, pero, si éstos no valen, habría que rehacerlo con los nuevos y su nuevo protocolo de datos para valorar si realmente es viable o no.
3. El módulo GPS es perfectamente viable y válido para el propósito propuesto. Es además aplicable y extensivo a cualquier posicionamiento GPS necesario con tan solo alimentar el módulo mediante, por ejemplo, una batería.
4. El módulo ficticio formado por la programación PC y la representación de los datos se ha probado también como válido y perfectamente utilizable con solo darle los datos que sean en su formato de entrada.

Por todo lo anterior, la conclusión a la que se llega tras la realización del trabajo es, que el estudio en sí mismo (que era el objetivo principal de esta memoria) ha sido realizado con éxito pudiéndose discernir qué soluciones son correctas y cuáles no. Y, en cuanto a las soluciones tomadas, solo dos de las cuatro son buenas direcciones a seguir y se podrían aplicar en el ámbito práctico directamente, mientras las otras dos necesitarían una nueva dirección de desarrollo para poder ser realmente aplicables.

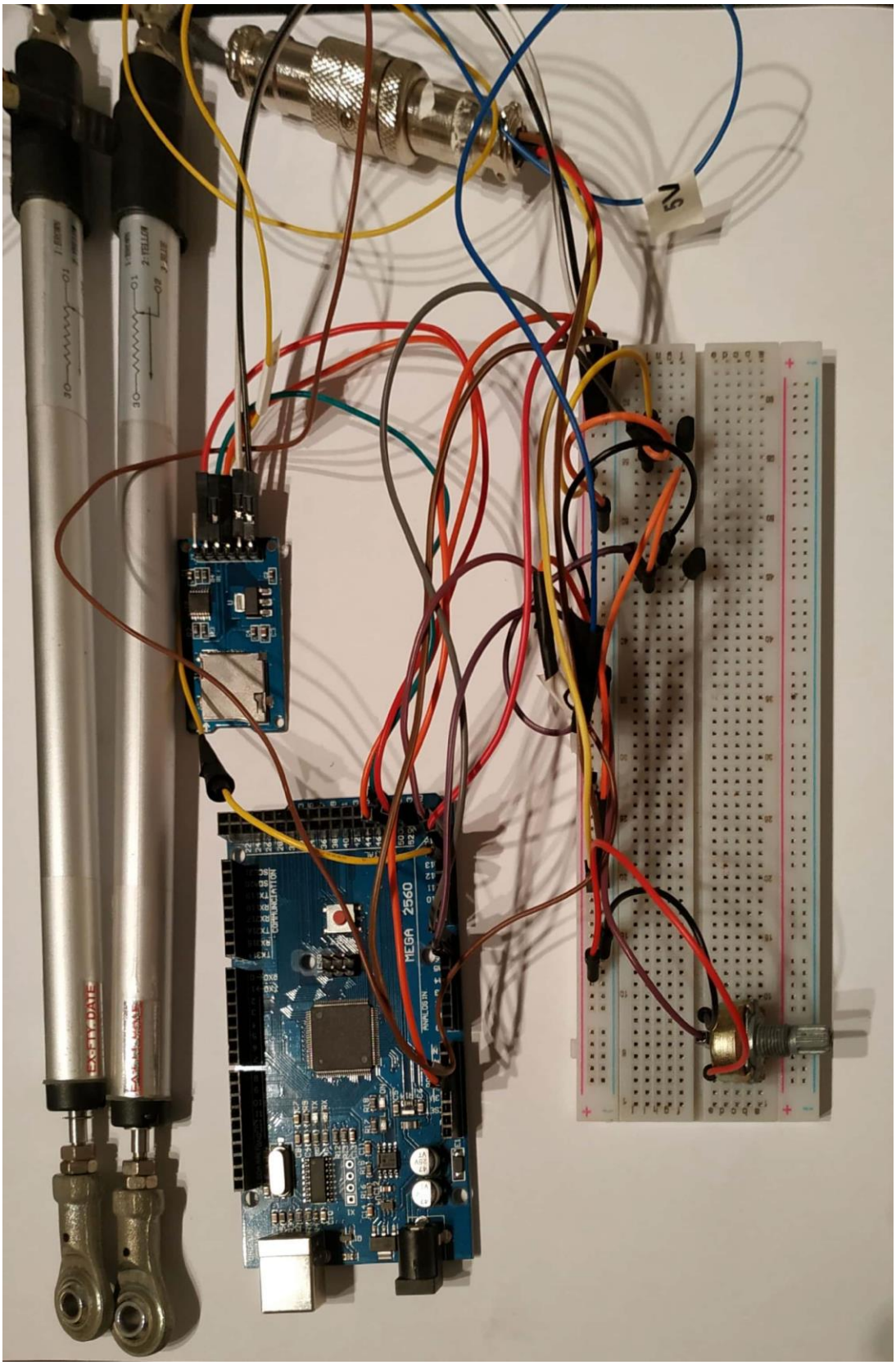
6 ANEXO: FOTOS DE MONTAJES Y PANTALLA



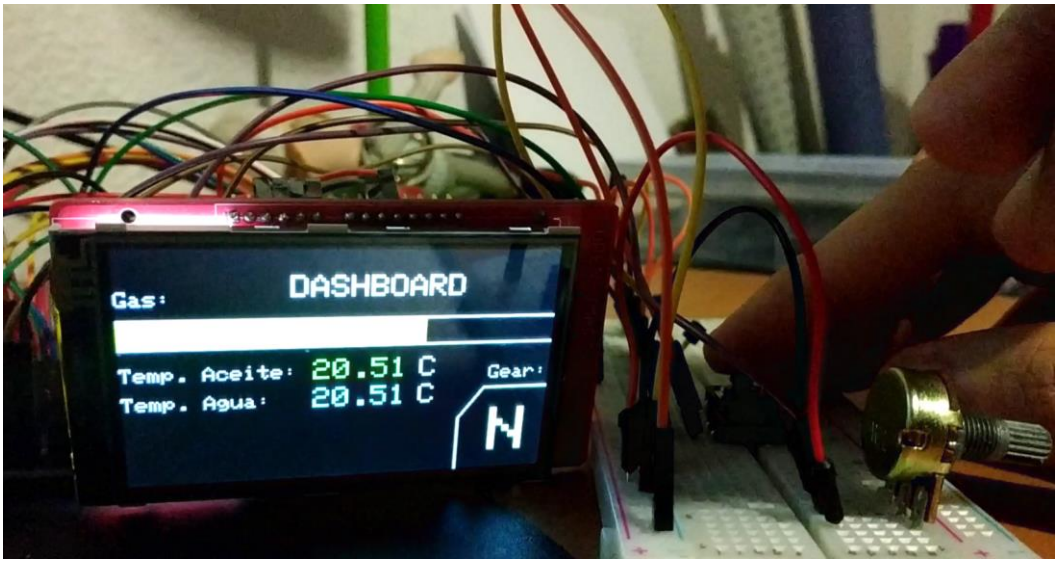
Montaje del Módulo de datalogging del GPS.

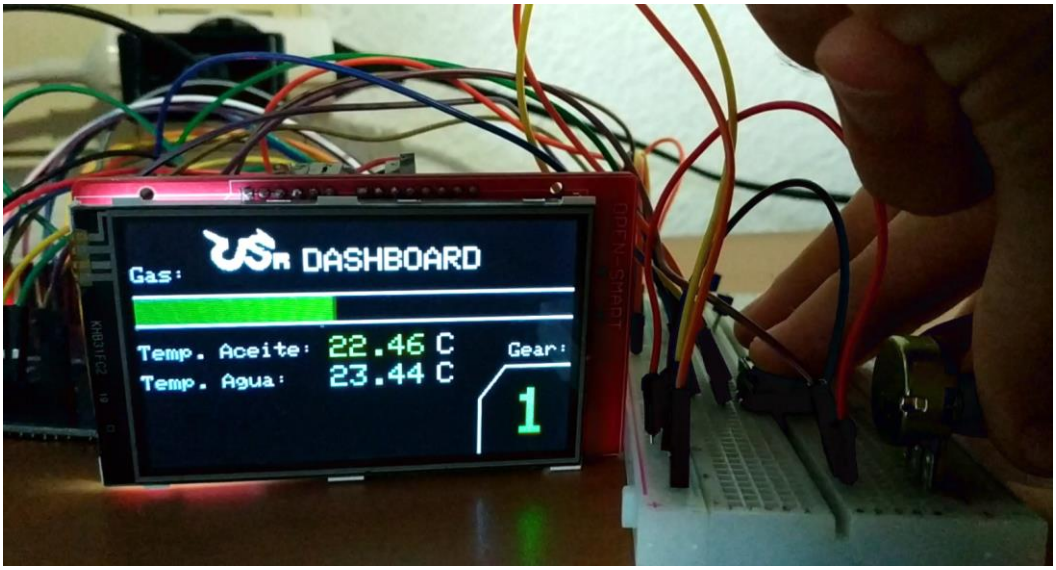


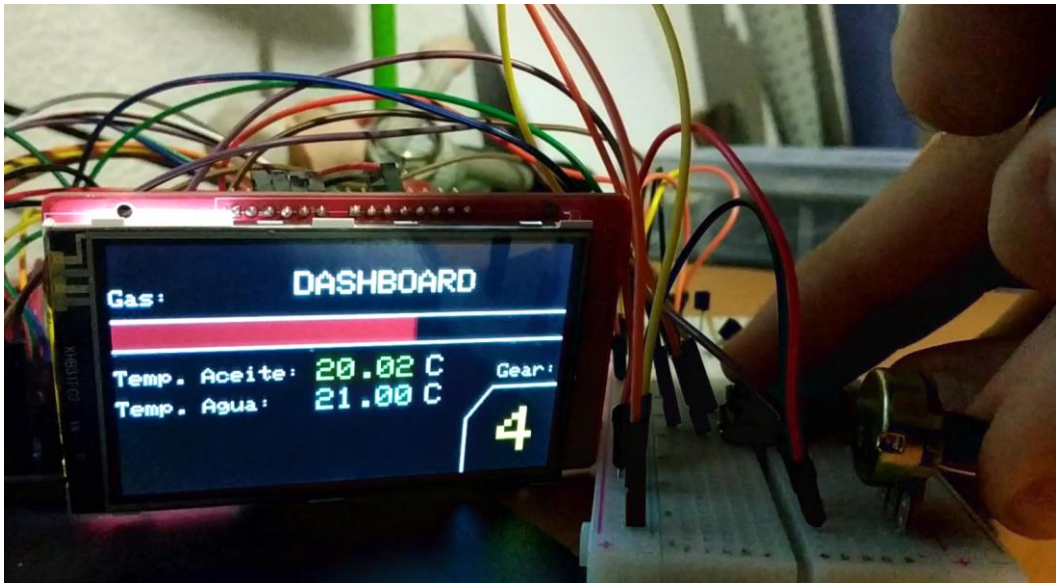
Montaje del Módulo de Pantalla

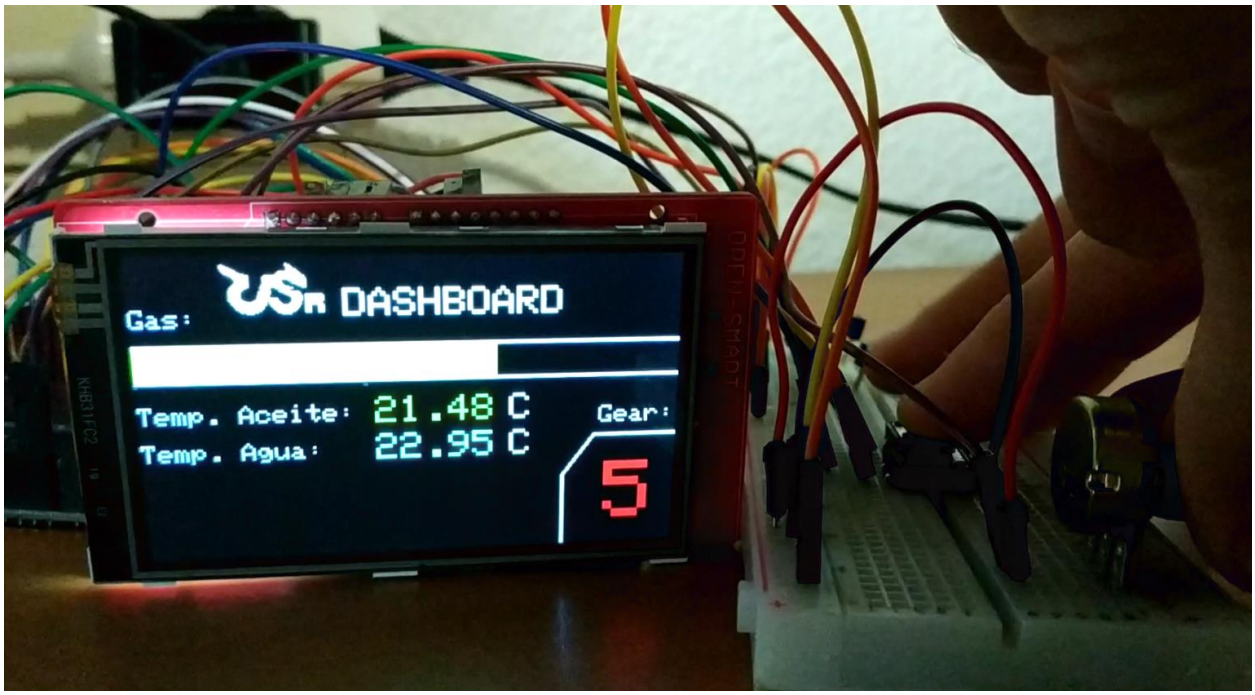


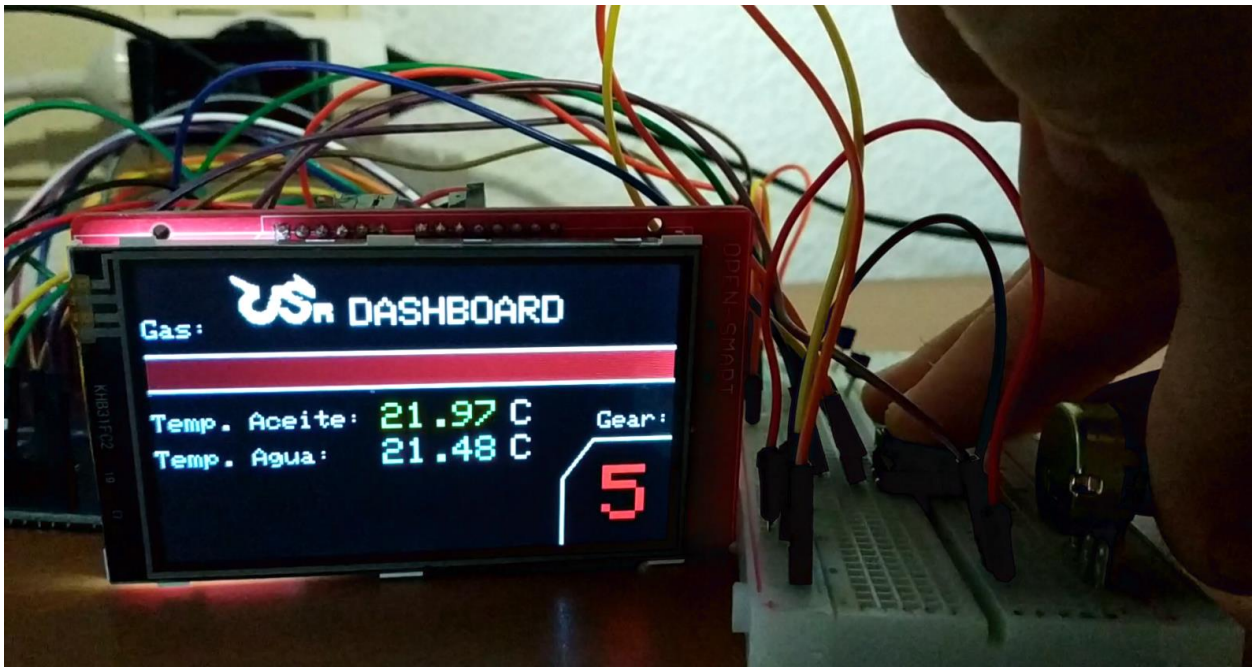
Montaje del Módulo de datalogging de los Sensores.


DASHBOARD
Gas:
Temp. Aceite: 20.51 C
Temp. Agua: 20.51 C
Gear:
N


DASHBOARD
Gas:
Temp. Aceite: 22.46 C
Temp. Agua: 23.44 C
Gear:
1


DASHBOARD
Gas:
Temp. Aceite: 20.02 C
Temp. Agua: 21.00 C
Gear:
4


DASHBOARD
Gas:
Temp. Aceite: 21.48 C
Temp. Agua: 22.95 C
Gear:
5


DASHBOARD
Gas:
Temp. Aceite: 21.97 C
Temp. Agua: 21.48 C
Gear:
5

7 BIBLIOGRAFÍA

https://en.wikipedia.org/wiki/Grand_Prix_motorcycle_racing

https://motogp.hondaracingcorporation.com/history/

https://www.motofan.com/noticias/antonio-cobas-el-genio/23587

https://www.w3schools.com/html/tryit.asp?filename=tryhtml_formatting

https://datasheetspdf.com/pdf-file/1099409/Himax/HX8352-B00/1

https://naylampmechatronics.com/arduino-shields/445-shield-display-lcd-tft-32-tactil-open-smart.html

https://es.wikipedia.org/wiki/Motocicleta

http://www.autosclasicosehistoricos.com/archivo/vehiculos-militares/235-las-motocicletas-de-la-segunda-guerra-mundial.html

https://www.thrillist.com/cars/history-of-u-s-military-motorcycles

https://www.xataka.com/basics/12-sensores-que-encontraras-tu-movil-sirven

https://en.wikipedia.org/wiki/Nokia_3310#Features

http://www.aficionadosalamecanica.net/sensores.htm

https://www.roadracingworld.com/news/motogp-aprilia-racing-and-manpowergroup-collaborate-on-new-motorcycle-race-engineering-program/

https://www.insella.it/motogp/news/motogp-2016-intervista-matteo-flamigni-valentino-rossi-e-piu-forte-scorso-anno-159753

https://elpais.com/deportes/2013/04/19/actualidad/1366399860_156446.html

https://www.aim-racingstore.com/es/

https://es.wikipedia.org/wiki/Sensor

http://www.ti.com/lit/ds/symlink/lm35.pdf

https://es.wikipedia.org/wiki/AVR

http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf

https://www.arduino.cc/reference/en/language/functions/communication/serial/

https://www.luisllamas.es/esquema-de-patillaje-de-arduino-pinout/

https://www.ixthus.co.uk/test-and-measurement-sensors/sensors-detail.php?mid=8&did=ELPM

https://es.wikipedia.org/wiki/Potenci%C3%B3metro

https://es.wikipedia.org/wiki/GPS

https://es.wikipedia.org/wiki/Trilateraci%C3%B3n

https://es.wikipedia.org/wiki/Constant%C3%A1n

https://naylampmechatronics.com/arduino-shields/445-shield-display-lcd-tft-32-tactil-open-smart.html

https://logictechno.com/wp-content/uploads/2016/11/LTTD240400030-L3-TF-V1.2-driver-IC_HX8352B.pdf

https://www.luisllamas.es/debounce-interrupciones-arduino/

https://es.wikipedia.org/wiki/Bus_CAN