Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Electrónica y Automática Industrial (GIEAI)

**Trabajo Fin de Grado**

# OBTENCIÓN Y CLASIFICACIÓN DE LA INFORMACIÓN DE DIAGNÓSTICO DE UN COCHE A TRAVÉS DEL K-BUS



**Autor:** Daniel Lázaro Gutiérrez

**Tutor/es:** Eliseo García García

2015-2016

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

**Grado en ingeniería electrónica y automática industrial (GIEAI)**

Trabajo Fin de Grado

# OBTENCIÓN Y CLASIFICACIÓN DE LA INFORMACIÓN DE DIAGNÓSTICO DE UN COCHE A TRAVÉS DEL K-BUS

**Autor:** Daniel Lázaro Gutiérrez

**Tutor/es:** Eliseo García García

**TRIBUNAL:**

**Presidente:** Javier de Pedro Carracedo

**Vocal 1º:** Miguel Ángel López Carmona

**Vocal 2º:** Eliseo García García

**FECHA**: 28 de Septiembre de 2016

# AGRADECIMIENTOS

Por supuesto, en este punto deben aparecer mis padres, por prestar el coche para la experimentación y ayudar en la recogida de datos una vez se tuvo el programa en funcionamiento.

Mención especial a mi madre, que me ayudó en la recopilación de ejemplos e información real de la casa BMW explicando las limitaciones y motivos por los que tuve que tomar un camino u otro.

Agradecer también a mi tutor, quien dedicó conmigo mucho tiempo en intentos fallidos de conexión al bus y a la página de Rolf Resler: http://www.reslers.de/IBUS/index.html, cuyo interfaz se adaptaba perfectamente al diseño previsto y que ahorró el costoso proceso de realizar la PCB.

También agradecer a las amistades que me han apoyado y se han preocupado por mi progreso y estado anímico (sobre todo aquellos que han soportado frustraciones en todos los intentos fallidos) durante éste último año de carrera.

# CONTENIDO GENERAL



**Nota: Aprovechando la versión digital, en lugar de páginas numeradas se han introducido hipervínculos.*

## RESUMEN DEL DOCUMENTO:

Este proyecto consiste en un estudio de los diferentes métodos y posibilidades de conexión de los buses del coche con algún medio (Pc o Microcontrolador) que permita leer los mensajes que allí aparecen y trabajar con ellos.

Se explicarán eléctrica y funcionalmente tanto el bus CAN como el bus K específico de BMW y el módulo ZKE V

Se realizará un diseño válido de un interfaz que nos permita obtener los datos del bus K del coche. Conseguiremos en la Raspberry Pi 2 una máquina de estados que nos deja monitorizar los mensajes relacionados con el módulo ZKE.

## PALABRAS CLAVE:

1. Raspberry Pi
2. Comunicación Puerto Serie
3. K-Bus
4. ECU (Electronics Control Unit)
5. Módulo ZKE V

## ABSTRACT:

In this project we've released a study of the different methods and possibilities to connect a car's bus to any device (PC or Microcontroller) that allow us to read the messages that appear on those busses and work with them

The ZKE module and K specific BMW and Can busses will be electrically and functionally explained. A design will be proposed for an interface that allow us to obtain data from the Kbus in the car.

A state machine will be developed in the Raspberry Pi 2 that will allow us to monitor every message related to the ZKE module.

## KEYWORDS:

1. Raspberry Pi
2. Serial Port communication
3. K-Bus
4. ECU (Electronics Control Unit)
5. ZKE V Module

# LISTA DE FIGURAS:

# 1- Introducción

El campo de la automovilística es uno de los temas que más interés ha suscitado en las últimas dos décadas en la población. Con la progresiva incorporación de mayor tecnología en los automóviles, también se ha ido perdiendo el total control de todas sus partes como se tenía antaño.

Hemos llegado a un punto en el que, algunos coches ni siquiera disponen de varilla del aceite para comprobar su nivel, siendo la única opción confiar en unos sensores. En muchas ocasiones, fallos de la electrónica pueden ocasionar caras visitas a las casas de la marca del automóvil para una puesta a punto.

Este problema es extrapolable a cualquier tipo de casa de automóviles.

Generalmente, la única forma de intentar localizar algún fallo en la electrónica del coche que pueda estar generando un "falso positivo" de avería es, o bien una visita al taller de la fábrica en cuestión, o el uso de algún tipo de consola de diagnóstico.

Podemos encontrar consolas de diagnóstico genéricas basadas en conexiones vía OBD-II que consisten en unos pequeños aparatos que a través del puerto de diagnóstico genérico de 16 pines (OBD-II) permiten realizar una diagnosis del estado el coche. Sin embargo, según el modelo del coche, es posible que una consola genérica no sea suficiente. En esos casos, es necesario recurrir a las consolas de diagnóstico oficiales, que además de servir de herramienta de diagnóstico, se pueden utilizar para controlar ciertos pilotos y electrónica del coche.

El problema es que ese tipo de consolas tienen un precio muy elevado y una única aplicación, comprobar errores y apagar o encender pilotos.

Hoy en día, se disponen de Microcontroladores y dispositivos baratos que, a través de su correcta manipulación, gestión y conexión, permiten conectarse a muchos dispositivos con relativa facilidad, siendo por supuesto la conexión a uno de los buses de datos de los coches una de esas posibilidades de conexión. Sin embargo, no es un tema que se haya explotado aun teniendo muchas posibilidades, lo que convierte a éste en un campo que requiere un mayor estudio y trabajo.

A lo largo de éste documento, se explicarán todos los pasos que se han realizado para la adopción de ésta filosofía en el ejemplo de un coche **BMW E46 330d del año 1998**, que es el que se encuentra a disposición experimental.

Como se detallará a lo largo del documento, el año, características técnicas, modernidad y recursos tecnológicos de cada coche será un factor muy importante en la determinación del camino a tomar para conseguir la máquina de estados que se propone.

Si bien este proyecto está guiado hacia la explotación de éste coche, se tratará la posible implementación genérica, e incluso otras posibilidades de conexión ajenas a éste proyecto que, en éste caso, por motivo del año de fabricación del coche, aún no estaban disponibles.

## 2- Objetivo, campo de aplicación y alcance

Podemos considerar el proyecto presentado como uno de estudio e investigación sobre los buses de comunicación internos de los coches o como un proyecto de desarrollo de un interfaz de comunicación con éstos.

El objetivo principal de este proyecto es la conexión al bus K de un coche BMW e46 (1998) para la obtención y posterior clasificación de la información de diagnóstico que circula en el mismo.

Esto nos abrirá un amplio abanico de posibilidades que no se llegarán a tratar debido a la extensión de las mismas, aunque sin embargo, si quedarán planteadas para futuras ampliaciones. El proyecto se va a centrar en la detección e identificación de los mensajes de diagnóstico o **estado** que el coche envía a la centralita de confort cuando sucede un evento.

El campo de aplicación es claramente automovilístico, pero no tiene por qué limitarse a un único modelo de coche, pues si bien en este proyecto se pretende demostrar las posibilidades que un coche concreto ofrece mediante este tipo de conexión, cada casa de automovilismo tiene sus propios códigos para las distintas tareas con el mismo tipo de conexión vía K-line (K-Bus).

Hoy en día, la mayoría de los coches tienen conexiones mediante una red multiplexada basada en el protocolo de Bus-Can. Sin embargo, siendo el Bus-K un predecesor de éste, los contenidos de este proyecto son fácilmente extrapolables al uso del Can (mediante el uso de los comandos adecuados, que se comentarán brevemente en el apartado correspondiente).

Por tanto, nuestro objetivo, una vez conseguida la conexión vía K-line (K-Bus) al coche, será identificar los códigos de ciertas partes del coche y conseguir un display de los datos en un formato legible.

Este proyecto se puede dividir en varias partes:

En una primera parte, habremos de conseguir la conexión vía k-bus al coche. Para ello se expondrán las distintas formas de conseguir una conexión a la red interna de un coche, para después particularizar en las posibilidades del modelo de experimentación y la justificación del uso de, en este caso, el cargador de CD.

El sistema operativo preferente con el que trabajaremos a lo largo de este proyecto será Linux, por su versatilidad y por ser un software de código fuente abierto. Como microprocesador, se utilizará la Rapsberry Pi, que trabaja en este entorno.

En el caso de conseguir con éxito la conexión, comenzará la segunda parte del proyecto, que consistirá en la observación de los códigos que circulan por el bus y se realizara una pequeña "Base de datos" identificando el uso de los mismos y a que partes del coche se refieren.

Finalmente, se plantearán posibles mejoras o proyectos que podrían derivar de la información que se ha obtenido en éste proyecto.

# 3- Conceptos previos

## 3.1- Bus de comunicaciones en los coches. OBD-Diagnostic

Los circuitos eléctricos tradicionales en el automóvil (los utilizados antes del 1993) permiten que por cada cable solo circule una información en las distintas formas de variación eléctricas (tensión, intensidad, potencia, resistencia, etc). No eran más que cableados físicos entre un punto de recepción y un generador de una variable eléctrica que lo comandaba. La evolución en seguridad, ecología y confort de los vehículos hace necesarias cada vez más informaciones con lo cual la cantidad, complejidad y coste de cableados eléctricos haría unas instalaciones imposibles.

Es muy importante que tengamos en cuenta **que cada fabricante tiene una técnica propia de interconexión**, pero hoy en día existen algunos PROTOCOLOS que son estándar.

En un vehículo de gama media se pueden encontrar hoy en día entre 4 y 10 módulos (considerando un módulo como una pequeña centralita de conexiones electrónicas) y en vehículos de alta gama se pueden encontrar hasta 40 módulos. Cada uno de ellos cuenta con una gestión electrónica independiente pero generalmente hay muchas informaciones que comparten, esto hace viable la incorporación de los sistemas de confort, chasis y emisiones, si no fuera por el sistema multiplexado, sería imposible poder tener todos estos módulos, instalados.

En los gráficos se puede ver la evolución de los cableados en un coche de gama media.


*Figura 1. Longitud de cable / Número de conexiones*

La revolución electrónica de los coches comenzó por tanto en los años 90, cuando Bosch desarrolló el protocolo de comunicaciones CAN (Control Area Network) que hace que a través de dos cables que interconectan todas las centrales electrónicas del automóvil circulen multitud de informaciones que pueden ser utilizadas por todos los elementos.
Es el más utilizado por las marcas de automóviles, pero hay otros (VAN usado por PSA, J1850 por Ford, GM y Chrysler, BEAM por Toyota, etc). Tienen algunas diferencias, pero todos se basan en los mismos principios de funcionamiento.

Otro uso de este protocolo estandarizado por Bosh y que hizo muy deseable su estandarización son los servicios de diagnosis y la toma de datos del vehículo. Los coches

comenzaron, de la mano a la introducción del CAN-bus en su estructura, a incorporar un conector especial llamado OBD que solía encontrarse debajo del volante o bajo el capó. Este conector nos permite generalmente acceder a ciertas partes electrónicas del automóvil. Hoy en día podemos, mediante un adaptador pertinente, conectar un ordenador, smartphone o similar y así nos enteraremos de todo lo que se cuece en el interior de nuestro coche.

**Desarrollaremos toda ésta información poco a poco:**

Pese a que hoy en día en todos los coches, independientemente de su gama, tienen como obligado cumplimiento la instalación de la red multiplexada del bus CAN y el protocolo de CAN en su OBD, sólo fue a partir de 2006. Por tanto, **¿cómo funcionaba la electrónica de los coches antes de utilizar éste estándar? ¿Qué es exactamente el OBD? ¿Sólo se usaba con el protocolo de bus CAN?**

3.1.1- Definiciones y conceptos de los buses de información

Pues bien, como hemos dicho, cada casa de automóviles tenía su propia solución a la hora de integrar los circuitos electrónicos pertinentes para comandar los aparatos que creyese pertinentes.

Como no podemos hablar de todos, nos vamos a centrar en como la casa BMW lo hacía en los primeros años de electrónica de automóvil. Para ello tenemos que tener claros una serie de conceptos a lo largo de todo el documento:

**Línea de bus:**

Una línea de bus es una línea de señales grupales que transmite datos en serie en ambas direcciones. Puede consistir en un hilo o dos. Todas las unidades de control están conectadas en paralelo en los buses actuales, esto significa que la información enviada puede ser escuchada por todas las unidades conectadas.

**Gateway:**

Un módulo de gateway proporciona la unión entre las diferentes líneas de bus para proporcionar un medio de transmisión de la información de un "suscriptor" o módulo que está conectado a la línea a otro "abonado" de la misma sin relación aparente. El módulo de gateway reconoce a partir de la dirección del receptor si un mensaje que transcurre por una línea ha de encaminarse a través de la puerta de enlace o no.

En una primera instancia, se pensó en realizar un proyecto que no solo realizara diagnosis, sino que también controlara ciertas partes del coche. No es algo imposible, pero sí que trae muchas complicaciones debido a éste concepto.

En el caso que estamos tratando, **BMW hace que sus módulos de Gateway presentes en cada extremo de una línea impida la propagación de cualquier mensaje que no debiese estar presente en ese extremo.** Es decir, si nosotros nos conectamos a la red electrónica interna a través de algún cable del cuadro de climatización, cualquier mensaje que tenga que ver con, por ejemplo, la iluminación, será ignorado y bloqueado por el Gateway.

En algunos coches modernos, **se puede eliminar el bloqueo del Gateway a través del código que el fabricante establece para ello. Sin embargo, esto es información clasificada y no válida para un usuario estándar.**

Otra forma de evitar el Gateway es "pinchar" directamente alguna línea del bus que interese manejar en un lugar que no sea un extremo. Así se evita el Gateway de entrada. Sin embargo, complica mucho la conexión, y puede poner en riesgo la integridad del coche.

**Master controller:**

El controlador maestro de un sistema de bus proporciona la tensión de servicio y las señales de "wake-up" o de inicio a los módulos conectados en la línea que controla. Esta tarea también puede ser realizada por varios "Standby Masters" dentro de un sistema de bus.

**Los datos en serie:**

Serial significa un evento a la vez. En la transmisión de datos, la técnica de división de tiempo (time division) se utiliza para separar los bits de datos enviados. Los mensajes enviados a través de un bus están se compuestos de:

1. Dirección del transmisor
2. Longitud de los datos
3. Dirección del Receptor
4. Comandos o información útil
5. Descripción detallada del mensaje (datos)
6. Resumen de la información transmitida (suma de comprobación ó check-sum)

3.1.2- Sistemas de buses en coches previos al estándar de 2006 (BMW)

Ahora que sabemos los conceptos básicos, hablaremos de cómo comenzó a moverse todo centrándonos en el modelo que trabajamos (e46 1998).

El bus CAN se introdujo por primera vez en BMW en el año 1993 en el modelo e38 740i. En éste momento el CAN solo se utilizó como conector entre los controladores maestros DME y AGS.

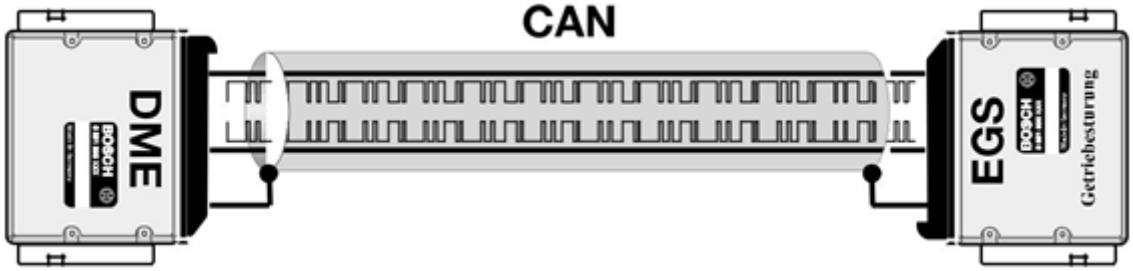



*Figura 2. Primera conexión CAN BMW*

BMW incorporaba hasta la estandarización una serie de buses o líneas diferentes al CAN para comandar la electrónica del coche. Si bien es cierto que desde el primer paso del CAN en BMW, con el tiempo, se fueron conectando más controladores al bus, tendremos que destacar las siguientes.

## Peripheral Bus (P-Bus)

El bus-P es un bus de comunicaciones de cable único que sólo se utiliza en vehículos equipados con el módulo ZKE III, que consiste en un controlador maestro que reúne en un solo módulo todas las funciones electrónicas ajenas al funcionamiento del coche consideradas como "body electronics", es decir, posiciones de espejos retrovisores, asientos de conductor o pasajero…

Se introdujo para reducir la cantidad de cableado presente en los coches. Los módulos de control comandados por esta unidad central se colocan próximos a los sensores o actuadores eléctricos.

El bus P solo se usa en éste área, y su funcionamiento y protocolo es muy similar al I/K bus que será explicado en un apartado independiente. El bus no ésta diseñado para un intercambio rápido de la información, si no cortos mensajes de control.

## Diagnosis Bus (D-bus)

El Bus de diagnóstico es un bus de comunicación serie que permite la transmisión de datos entre las unidades de control del coche y los sistemas de diagnóstico oficiales de BMW DIS o MoDiC.

La unidad de control sometida al diagnóstico se selecciona al enviar un telegrama de diagnosis a una unidad. El aparato de diagnosis puede entonces pedir la información de la memoria de fallos de la unidad o activar un control remoto de la unidad para testeo.

El bus permanece apagado hasta que el DIS o MoDiC se conecta y envía la señal de "wake-up".

Éste bus es el bus más antiguo de BMW, introducido en 1987.

Topología:

La topología del D-bus (o también llamado TXD) queda conectado a las unidades de control y a los sistemas oficiales de diagnóstico una vez hecho el montaje. Sin embargo, es mediante una segunda línea de diagnosis (denominada RXD), que no es una línea de bus, sino un cable de una sola dirección, como se envía el mensaje de "wake-up" al bus-D. Por ello generamente se considera que el bus está formado por ambas líneas.

## Information and Body Bus (I y K -bus)

Los buses I y K son buses de comunicación serie en los que cualquiera de las unidades de control que están conectadas pueden enviar y recibir información en un solo cable. **Podríamos decir que es el predecesor al CAN. Ambos buses (I y K) son eléctricamente idénticos**, la única diferencia que hay entre ellos es el uso que se le da según el modelo de coche que se trabaje.

**Debido a que son iguales, podremos referirnos a cualquiera de estos buses como I/K bus, aunque en nuestro coche (E46 1998) sólo se usa el K.**

Éste bus se introdujo por primera vez en el BMW E31 para conseguir la estandarización, fiabilidad y reducción de coste necesarias para la comunicación de la electrónica del coche al estar en gran crecimiento.

Fue a partir del modelo E38 donde se introdujeron cada vez más buses y unidades de control a la red, siendo ésta la primera aparición de los buses P y K.

**A diferencia del bus de diagnóstico, el bus I/K está activo siempre que el terminal R esté conectado.** Por tanto, esto hace del bus un candidato perfecto para el desarrollo de éste proyecto, pues no requiere de los elementos de testeo oficiales de BMW para iniciarlo. En caso de que el bus este parado más de 60 segundos, todos los módulos entran en modo sleep.

Se dedicará un apartado específico en el documento a éste bus más adelante.

### 3.1.3- OBD en BMW y ECU

Ahora que hemos tratado los conceptos básicos de los buses y un poco de historia sobre su inserción, hablaremos sobre el OBD y el concepto de ECU (Electronics Control Unit)

En la industria automotriz una unidad de control electrónico (ECU) es un dispositivo electrónico embebido, básicamente una PC digital, que lee señales provenientes de sensores ubicados en varias partes y en diferentes componentes del automóvil y dependiendo de esta información controla varias unidades importantes por ejemplo el motor y operaciones automatizadas en el auto y también verifica el rendimiento de algunos componentes clave usados en el automóvil.

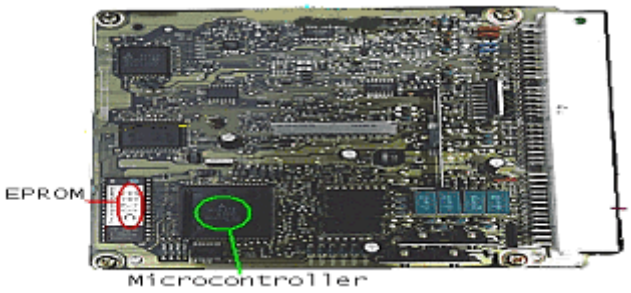



*Figura 3. ECU BMW e46*

Un ECU está hecho básicamente de hardware y software (firmware). El hardware está hecho de varios componentes electrónicos en un PCB. El componente más importante es un chip microcontrolador junto con un EPROM o un chip de memoria Flash. El software (firmware) es un juego de códigos de menor nivel que se ejecuta en el microcontrolador.

El ECU se caracteriza por:
· Varias líneas de E/S analógica y digital (alta y baja potencia)
· Dispositivo de interfaz/control de potencia
· Diferentes protocolos de comunicación (CAN, KWP-2000, ISO-9141-2, etc.).
· Grandes matrices de conmutación para señales de alta y baja potencia
· Pruebas de alto voltaje
· Adaptadores inteligentes de interfaz de comunicación (estándares o personalizados)
· Reconocimiento automático de equipo y habilitar secuencia de software
· Simulación de dispositivo de potencia

Sin embargo el término ECU puede ser ambiguo, pues actualmente los ECU's son nombrados y diferenciados dependiendo del uso o destino que tengan:

ECM – Módulo de Control de Motores. (Con frecuencia en la industria, los ECM son llamados ECU - Unidad de Control de Motores, sin embargo, no hay que confundirlo con el término de ECU al que nosotros tomaremos).

El ECM también conocido como EMS (sistema de administración de motores). Es un ECU en un motor de combustión interna que controla varias funciones del motor como por ejemplo la inyección de combustible, el sistema de control de tiempo de inyección y de distribución de válvulas, etc… Todo este control se realiza basándose en datos (como por ejemplo la temperatura del anticongelante del motor, el flujo de aire, la posición de palanca…) recibidos desde varios sensores.

El ECM también aprende sobre el motor conforme manejamos nuestro automóvil. El ECM almacena las posiciones de ciertos actuadores (posiciones de luces, espejos o asientos) "aprendidas" en RAM y así no tiene que iniciar desde cero la próxima vez que el motor es encendido. Con la aplicación de las Regulaciones de Emisión Federal el 1981, que sería el desencadenante de los sistemas OBD, los ECUs se han usado en la mayoría de los vehículos.

EBCM – Módulo de Control de Frenos Electrónicos.

Es el ECU que es usado en el módulo ABS (sistema de freno antibloqueo) de un automóvil. Se introdujeron a principios de 1970 para mejorar el frenado del vehículo sin importar las condiciones del camino o clima. Aunque es muy reciente, ya ha obtenido popularidad. El EBCM regula los sistemas de frenado en las cinco entradas que recibe.

**Entradas/Salidas Típicas de un ECU**

Un ECU consiste en un número de bloques funcionales:

1. Fuente de Alimentación - digital y analógica (potencia para sensores analógicos)
2. MPU – microprocesador y memoria (generalmente Flash y RAM)
3. Enlace de Comunicación – (ej. bus CAN)
4. Entradas Discretas – entradas tipo interruptor On/Off
5. Entradas de Frecuencia – señales tipo codificador (ej. palanca o velocidad de vehículo)
6. Entradas Analógicas - señales de retroalimentación desde sensores
7. Salidas de Conmutador - salidas tipo interruptor On/Off
8. Salidas PWM - frecuencia variable y periodo (ej. inyector o encendido)
9. Salidas de Frecuencia - periodo constante (ej. motor de pasos - control de tiempo de inyección)

Y generalmente en una Unidad de Control de Motores existen varios tipos de sensores y actuadores conectados y es importante saber el tipo de E/S que requieren.

¿Qué es el OBD que ha aparecido varias veces hasta aquí en este documento?

OBD o “On Board Diagnostic” es un sistema de ordenador creado y estandarizado en el 1996 y que surgió de las modificaciones de la Ley de Aire Limpio de 1990.

Los sistemas de OBD fueron diseñados para monitorear el desempeño de algunos de los componentes principales de un motor incluidos los responsables de controlar las emisiones.

Sin embargo, el OBD tuvo muchas más aplicaciones:

Para los técnicos de reparación:

El OBD supone una valiosa herramienta que ayuda en el servicio y reparación de vehículos, proporcionando una manera sencilla, rápida y efectiva para detectar problemas mediante la información de diagnóstico que el coche proporciona.

Para las agencias estatales:

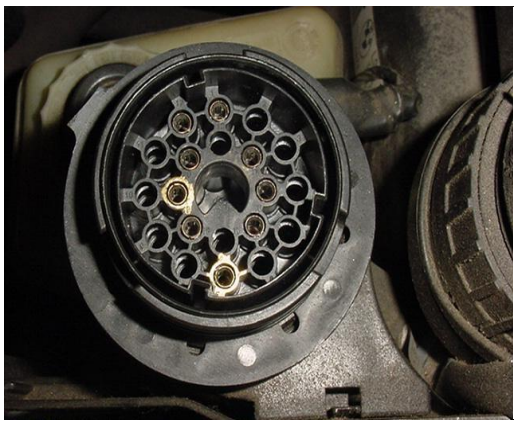
OBD juega un papel importante cuando se requieren programas de inspección y mantenimiento de vehículos.

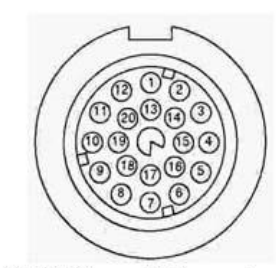
Para propietarios de vehículos:

El OBD sirve como un sistema de alerta temprana que avisa de la posible necesidad de reparación de vehículos a través de los pilotos de aviso en el salpicadero del vehículo. En el momento en que un piloto se enciende, el módulo del motor (ECU) guarda el código de error para suministrarlo a través de una herramienta de diagnosis posteriormente.

Centrándonos en los usos de reparación y diagnosis de usuario, buscaremos el caso de BMW.

Tras la normativa, BMW diseñó un sistema de OBD de 20 pines que está localizado bajo el capó.



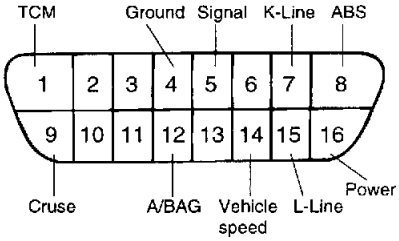


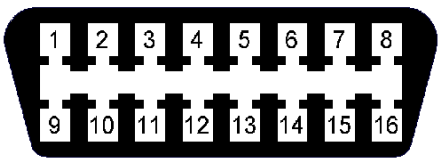
20 Pin Diagnostic Connector

Pin 1 Engine speed signal (Ti)
Pin 7 Service light reset
Pin 12 D+ Alternator
Pin 14 Full time power
Pin 15 RXD data link to all control units
Pin 16 Power in run and start
Pin 18 Direct link to DME for program selection.
Pin 19 Ground
Pin 20 TXD data link to all control units

*Figuras 4 y 5. OBD 1ºGeneración en BMW/Pinout*

En 1996, la EPA estableció un nuevo estándar denominado OBD-II que contenía más elementos de diagnosis y control, además del control de emisiones. Consistía en un sistema de 16 pines como se muestra en la figura.




*Figuras 6 y 7. OBD-2 /Pinout*

A partir de ese año, todos los coches debían llevar el OBD-II instalado. El nuevo OBD-2 sería un puerto de 16 pines localizado bajo el asiento del conductor (generalmente) que permitiría un mejor control de emisiones a la vez que diagnosis. Sin embargo, algunos coches ya cumplían con las condiciones del OBD-II en sus respectivos OBD, siendo un cambio opcional.

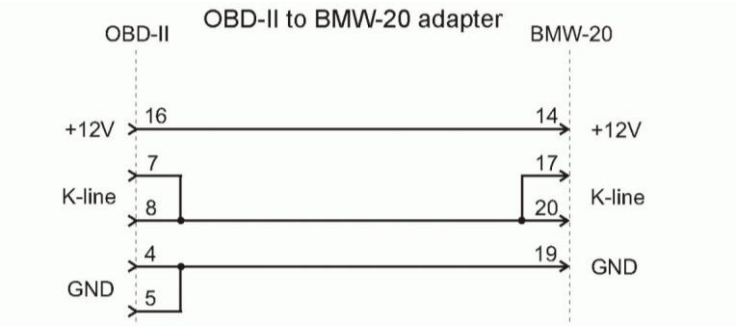
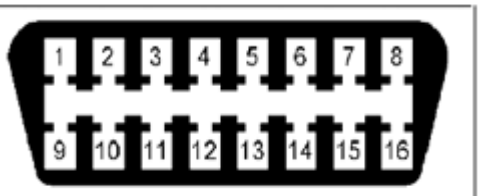



*Figura 8. Adaptación del OBD 1 de BMW al estándar OBD2*

A diferencia de muchos casos, el OBD de primera generación de BMW ofrecía algunas posibilidades que el estándar de 16 pines no. Por ello, en los primeros años de aplicación (y en nuestro caso, en el modelo E46), BMW optó por adaptar su OBD al estándar de OBD2 y mantener un único OBD bajo el capó.

Sin embargo, en los modelos de los años 2000, se comenzaron a incorporar ambas posibilidades de diagnóstico en BMW (16 pines y 20 pines). Finalmente, con la estandarización del Bus CAN, se modificaron los pines del OBD2 y quedó éste como el único método de control en todas las casas de automóviles.

**¿Por qué tratar el tema del OBD2 entonces? Porque el OBD está conectado a todos los módulos de la electrónica del coche.**




| PIN | DESCRIPTION | PIN | DESCRIPTION |
|---|---|---|---|
| 1 | Vendor Option | 9 | Vendor Option |
| 2 | J1850 Bus + | 10 | J1850 Bus – |
| 3 | Vendor Option | 11 | Vendor Option |
| 4 | Chassis Ground | 12 | Vendor Option |
| 5 | Signal Ground | 13 | Vendor Option |
| 6 | CAN (J-2234) High | 14 | CAN (J-2234) Low |
| 7 | ISO 9141-2 K-Line | 15 | ISO 9141-2 L-Line |
| 8 | Vendor Option | 16 | Battery Power |

*Figura 9. Estándar OBD2*

A partir de su estandarización, el OBD2 tiene la siguiente disposición:

Como vemos, tiene algunos pines disponibles para la elección de la casa de automóviles.

Sin embargo, nos tendremos que fijar en los pines de CAN y K o L line.

**Con el cableado pertinente, podrían suponer nuevos puntos de acceso al bus interno del coche que intentamos explotar (para la línea K).**

**En el caso del Bus CAN, la protección habitual de los coches hace que no sea posible el comando del bus a través del puerto OBD.**

**Sin embargo, algunos modelos no tienen ésta limitación. Se estudiará en algún capítulo posterior.**

## 3.2- CAN-Bus

¿En qué consiste?

CAN es el acrónimo de Controller Area Network y Bus significa una topología en forma de bus (valga la redundancia). Esto quiere decir que hay un solo cable que recorre el vehículo al que se van conectando los diferentes aparatos electrónicos que necesiten comunicarse. De esta forma reducimos la cantidad de cables que se necesitan en el coche.

¿Y qué aparatos son los que necesitan comunicarse en un automóvil moderno? Multitud. Pensemos en los elevalunas eléctricos, el climatizador, el cierre centralizado, el techo solar, los asientos eléctricos, la centralita de la inyección y todos sus sensores, el cuadro de instrumentos, los mandos en el volante, los sistemas multimedia...

De hecho, tal es la cantidad de dispositivos que en la actualidad, para garantizar la rapidez y robustez de las comunicaciones, no suele haber un solo bus CAN si no que hay varios sub-buses en el vehículo. Un bus para la gestión electrónica del motor, otro para climatización y entretenimiento, otro para temas de seguridad (alarmas, cierre centralizado, ABS) etc...

Cualquier dispositivo electrónico conectado al bus puede mandar mensajes y el resto le escucha. Cada tipo de mensaje lleva un identificador. Los oyentes deciden qué mensajes les interesan y cuáles no. Para conseguir un funcionamiento correcto, los dispositivos eléctricos se van turnando para "hablar" de uno en uno.

El protocolo CAN fue una idea de Bosch en 1982 y el primer modelo de producción en montarlo fue el Mercedes-Benz Clase E de 1992. El CAN bus se ha convertido en un estándar de facto y en la actualidad se emplea en la inmensa mayoría de automóviles que se fabrican y también comienza a introducirse en el sector de las motocicletas.

El bus CAN es un protocolo serie asíncrono del tipo CSMA/CD (“Carrier Sense Multiple Access with Collison Detection”), en el que todos los nodos (aparatos conectados al bus) tienen la posibilidad de mandar, recibir y pedir mensajes a los otros nodos tal como hemos indicado. Sin embargo, ¿cómo se turnan para hablar?
La detección de colisión hace que si dos nodos de la red transmiten un mensaje al mismo tiempo un método de resolución basado en prioridades resuelva el conflicto.

A nivel físico, este bus utiliza dos cables trenzados (bus diferencial) para ser inmune a interferencias electromagnéticas. Para que el bus deje de funcionar, es necesario cortar ambos cables que lo componen, lo que hace que este bus sea muy fiable. También se utilizan resistencias terminadoras (típicamente de 120Ω) al principio y al final del bus que evitan reflexiones y ruidos en el bus.

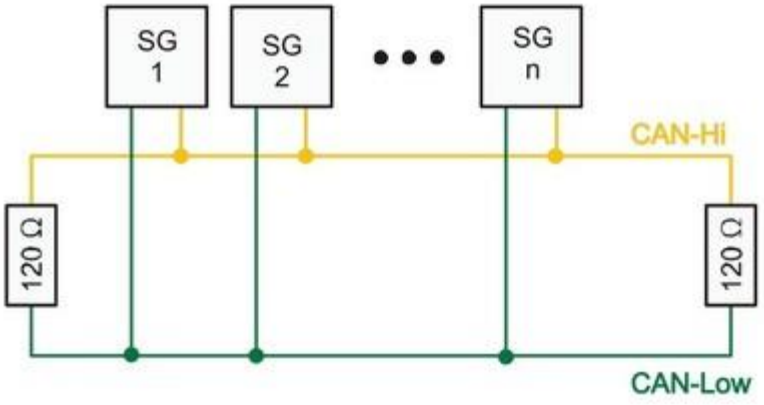



*Figura 10. Esquema básico constructivo del bus-CAN*

La velocidad máxima del bus es de 1Mbps y la longitud máxima que puede llegar a tener dependerá tanto de la velocidad como de la cantidad de ruido electromagnético que haya en el ambiente donde se encuentra el sistema. Es una práctica general que en los coches se utilice a 125kbps para el sistema de confort (elevalunas, climatizador, etc.) y a 500Kbps en el sistema de tracción (gestión del motor, ABS, etc.).

Es interesante resaltar de nuevo que en los vehículos existen puertas de enlace o "Gateway" que permiten el traspaso de información entre los buses, así como la comunicación de éstos con el exterior.

En automovilística, es el Bus CAN el encargado por estandarización de servir como bus de comunicaciones para los elementos más importantes del coche, que van a requerir una seguridad y velocidad mayores. Actualmente, se reparte parte del pastel con un nuevo bus denominado LIN, de software abierto y que se encarga de los temas menos prioritarios (como por ejemplo, elevalunas eléctricos), sin embargo, no lo trataremos al estar lejos de la fecha de nuestro coche.

Ya sabemos qué es el bus CAN, ahora, ¿cómo funciona?

El protocolo CAN está basado en mensajes, no en direcciones. El nodo emisor transmite el mensaje a todos los nodos de la red sin especificar un destino y todos ellos escuchan el mensaje para luego filtrarlo según le interese o no. En este caso, extraeremos la información del protocolo de CAN que utiliza el proveedor Texas Instruments.

Existen distintos tipos de tramas o mensajes predefinidas por CAN para la gestión de la transferencia de mensajes:
- Trama de datos: Se utiliza normalmente para poner información en el bus y la pueden recibir algunos o todos los nodos.
- Trama de información remota: Puede ser utilizada por un nodo para solicitar la transmisión de una trama de datos con la información asociada a un identificador dado. El nodo que disponga de la información definida por el identificador la transmitirá en una trama de datos.
- Trama de error: Se generan cuando algún nodo detecta algún error definido.
- Trama de sobrecarga: Se generan cuando algún nodo necesita más tiempo para procesar los mensajes recibidos.

- Espaciado entre tramas: Las tramas de datos (y de interrogación remota) se separan entre sí por una secuencia predefinida que se denomina espaciado inter-trama.
- Bus en reposo: En los intervalos de inactividad se mantiene constantemente el nivel recesivo del bus.

Lo que es realmente interesante del bus CAN es el modelo de uso real: cada paquete de datos tiene una dirección única definida por 11 o 29 bits (que recibe el nombre de trama extendida) que a su vez asigna una prioridad, y una carga útil máxima de 8 bytes. Así podemos distinguir entre la trama estándar y la extendida:

Trama estándar:

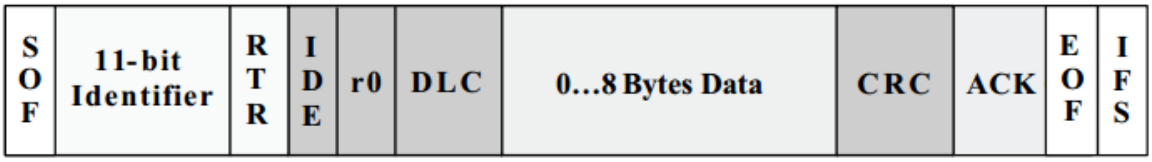



*Figura 11. Trama estándar de datos de bus CAN*

SOF- Start of Frame es un bit que marca el comienzo de un mensaje, y se usa para sincronizar los nodos en un bus después de estar inactivos.

Identificador- El identificador de 11 bits del estándar CAN establece la prioridad del mensaje. Cuanto menor sea el valor binario, mayor es su prioridad. Esto significa que habrá algunos receptores con mayor prioridad que otros, y por tanto se transmitirá primero éste mensaje que otro.

RTR – Remote Transmission Request es un bit que se pone a 0 cuando se requiere información de otro nodo, aunque tiene la menor prioridad. Indica que no son datos lo que se está enviando, sino una petición por parte de un nodo de los datos que alguien ya tiene.

IDE- Identifier extensión bit es un bit que vale 0 en caso de que el mensaje sea estándar (11 bit) o 1 si es extendido (29).

DLC- Data lenght code son 4 bits que representarán cuantos bytes de datos habrá (0-8).

CRC- Cyclic redundancy check son 15 bits que se usan en la detección de errores.

ACK- es el bit de acknowledgement, en principio es un 1, y se cambia a valor 0 cuando el mensaje es recibido por su receptor.

EOF- End of Frame son 7 bits utilizados para establecer el fin de la información útil.

IFS- Inter Frame Separator, que son otros 7 bits que determinan el tiempo que necesita el controlador para mover una trama correctamente recibida a su posición en el buffer.

Trama extendida: Su funcionamiento es igual que el de una trama estándar pero con las siguientes adiciones:

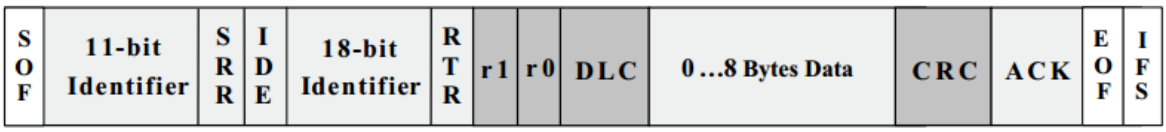



*Figura 12.Trama extendida de CAN*

SRR-Substitute Remote Request es el bit que siempre se coloca a 1 y asegura la prioridad de una trama estándar frente a una extendida.
IDE – Identifier Bit Extension un bit que en este caso, marcará 1.

Las claras ventajas del uso del CAN se reflejan en que el bus puede tener una gran cantidad de tráfico de datos en él, pero la mayoría de los nodos sólo se preocuparán por algunos identificadores de específicos y no manejan el tráfico total, cada controlador de bus CAN implementa una especie de " filtro de aceptación" que por lo general es una máscara de bits que establece que rango de direcciones de trama debe ser recibido por qué nodo, mientras que todos los demás se descartan en silencio.

Esto hace que el bus sea adecuado en el entorno del automóvil y de automatización, donde algunos sensores pueden poner los datos en el bus que serán utilizados por muchos procesadores o actuadores.

**Por tanto, y para lo que a nosotros nos interesa, el bus CAN parece un caramelo. Si nos conectamos a él podremos ver el tráfico de todos los mensajes que circulan por el interior de un coche con sus respectivos identificadores.**

Sin embargo, el acceso a un bus CAN no es para nada sencillo, pues requiere el desmontaje de algún elemento estructural del coche y una serie de acciones para poder “pincharlo” y conectar algún cable a él.

La opción más lógica es por tanto, ya que hemos explicado el sistema de OBD, y hemos visto que tiene dos pines reservados para el bus CAN H y L, **¿por qué no conectar desde ahí?**
**Como ya se ha comentado, no todos los coches permiten la conexión al bus a través del puerto de diagnóstico. BMW es uno de aquellos que no lo permite.**
**Por ello, debemos buscar otro método para poder conectarnos al interior del coche, y de aquí surge la necesidad de utilizar el bus K.**

Sin embargo, al final del proyecto se expondrá la metodología de conexión a uno de esos coches que no presentan ésta limitación.

## 3.3- K/I-Bus de BMW

El K/I Bus de BMW está basado en el estándar ISO 9141 (El mismo protocolo que utiliza el sistema de OBD y que se tratará posteriormente). Es un único cable al que se conectan varias cosas, que suele ser de color blanco/rojo/amarillo y que puede localizarse con facilidad en varios lugares, como por ejemplo, maletero o conector del teléfono.

El bus K/I es un bus que, en el año que data nuestro modelo de coche (1998) maneja mayormente información de los periféricos, pues la red CAN es la encargada de tratar con la

ECU y temas del motor. El bus I se conoce como "Instrumentation-Bus" y el bus K como "Karosserie-Bus" – Karosserie significa "cuerpo" en alemán.

Los modelos de coche que no tengan Sistema de Navegación o los más antiguos solo utilizan el bus K (como es nuestro caso).

Como ya hemos dicho, son buses de comunicación serie que funcionan a 9600 bps, con 8 bits de datos, paridad y 1 stop bit.

Eléctricamente, se trata de una única línea de 12V con varios dispositivos o receptores conectados a ella. Cuando la línea está en reposo o ha acabado de transmitir un dato, el propio bus establece el nivel H con 12 V a la línea, y ese voltaje será L cuando cualquiera de los dispositivos hable. Cualquier dispositivo puede empezar a enviar datos cuando el bus está "inactivo", es decir, a nivel H.


*Figura 13. Trama de datos de K/I Bus.*

La estructura de la trama se divide en:

| | |
|---|---|
| Source Device ID | Es el identificador del dispositivo que necesita mandar un mensaje. |
| Length | El tamaño o longitud de los datos que se van a enviar sin tener en cuenta el identificador de fuente. |
| Destination Device ID | Identificador de quien debe recibir el mensaje |
| Data | El mensaje a enviar<br><br>El primer Byte es, bien la identificación del mensaje o el comando, y el resto características o atributos de ese comando o información. |
| XOR CheckSum | El byte de checksum hace la comprobación del mensaje, igual que hemos explicado anteriormente. |

Como vemos, **la gran diferencia que hay con el bus CAN es la forma de identificar el receptor. Si bien en el CAN, una trama tiene establecida quien lo recibirá en su identificación y circula por todo el bus siendo visible a todos los dispositivos conectados, el bus K tiene ya establecido un receptor al momento de generar el mensaje**, lo que quiere decir que no todos los instrumentos lo recibirán o serán conscientes de que ese mensaje está circulando por el bus, solamente lo será el receptor.

Así el sistema de prioridad queda anulado, pues cuando se genera un mensaje, se ha de esperar a que el bus vuelva a su posición de 12V (H) para que un nuevo mensaje transite el cable.

Este sistema de H cuando el bus está inactivo y L cuando se está transmitiendo algo, nos recuerda al protocolo serie RS232, por tanto es fácil pensar que el tratamiento de datos del protocolo serie a un microcontrolador no debería ser imposible.

## 3.4- ISO-9141-2

Como hemos dicho previamente, el sistema de OBD, y el bus K que pretendemos explotar se basan en el protocolo ISO-9141-2.

Es una norma UNE, que requiere su compra online para poder disponer de toda la información que permite. Un link a una página para su compra se encuentra en la bibliografía.

Sin embargo, alguna información básica del estándar es pública:

Es una Norma Internacional que se estableció con el fin de especificar las siguientes características deseables para el diagnóstico de los sistemas de a bordo de control electrónico:

1) Determina los requisitos eléctricos del sistema de diagnóstico del vehículo de manera que un equipo de diagnóstico con, al menos, la capacidad funcional mínima especificada en el documento que recoge la norma será compatible con cualquier sistema de diagnóstico a bordo diseñado de acuerdo con estas especificaciones.

2) Limita el número de contactos en los sistemas de control electrónico para las comunicaciones de diagnóstico unidireccionales y bidireccionales.

3) Determina que al menos se dará la transmisión de información de identificación, así como la información de estado de funcionamiento, incluidos los valores reales de los parámetros y los valores requeridos.

Esta normativa se llevó a cabo para cumplir los siguientes objetivos:

a) Determinar si un sistema está funcionando correctamente.

b) Llevar a cabo una inspección.

c) Para localizar las desviaciones de las especificaciones y lograr una reparación económica y rápida.

d) Para confirmar que un sistema ha sido restaurado para corregir el mal funcionamiento correctamente.

e) Para restablecer o ajustar los valores de funcionamiento del sistema en una ECU en estricta conformidad con las instrucciones del fabricante del vehículo.

f) Supone una fuente de información ya registrada.

Esta norma específica los requisitos para configurar el intercambio de información digital entre unidades a bordo del control electrónico (ECU) de los vehículos de carretera y los probadores de diagnóstico adecuados.

Esta comunicación se establece con el fin de facilitar la inspección, el diagnóstico de prueba y ajuste de los vehículos, sistemas y centralitas.

## 3.5- ¿De dónde obtendremos los datos que buscamos? ¿Por qué allí?

En este punto, ya tenemos toda la información que necesitamos para plantearnos estas preguntas. Sabemos los protocolos que tenemos y como funciona cada uno, hemos expuesto todos los elementos del coche que necesitamos conocer para extraer datos y hemos concluido que el método menos invasivo y más sencillo sería poder extraer los datos del **Bus K**.

Bien, habíamos hablado entre uno de estos puntos de las ECU, y de los diferentes tipos y nombres que se dan según su función.

Lo que nosotros pretendemos es conseguir hacer una máquina de estados que nos informe de los sucesos en el coche. Para ello, tenemos que acceder a la ECU que se encargue de la electrónica en el modelo de coche que vamos a trabajar (E46 1998).

¿A qué ECU debemos acceder? ¿A qué ECU podemos acceder?

En éste modelo de coche, detrás de la guantera podemos encontrar el módulo ZKE. En este caso, es modelo ZKE V, que se encarga de casi todo el control de la electrónica de accesorios del coche.

El módulo básico de la electrónica centralizada de la carrocería ZKE V controla las siguientes funciones:

- Limpiado y lavado del parabrisas
- Limpiado de los faros
- Elevalunas eléctricos
- Cierre centralizado
- Instalación antirrobo
- Alumbrado del habitáculo
- Desconexión de consumidores
- Telemando

Adicionalmente, a partir del 03/2003, también se incluirá aquí la memoria de los espejos retrovisores, su retracción y su calefacción. Sin embargo, estas funciones quedan fuera de nuestro estudio al ser posteriores a nuestro modelo de coche.

Esto se debe a que existen dos versiones de modulo básico ZKE:

La conocida como “Variante High” (equipamiento completo) y “Variante Low” (sin elevalunas eléctricos en las ventanas traseras. En nuestro caso, nos encontramos con la variante LOW.

Ambas versiones son muy similares, aunque la High, como la variante de rediseño, incluye el bus LIN para las funciones de espejos y para un bloque de interruptores central.

¿Cómo nos interesa éste módulo?

**El ZKE V está directamente conectado con el bus K del coche.** El módulo básico comunica con otras unidades de mando a través del bus K. El módulo básico recibe y transmite datos a través del bus K, entre otros, los siguientes datos.

- Posición de la cerradura de encendido (estado de los bornes R y 15)
- Duración de intervalo para el limpiaparabrisas del sensor de lluvia
- Luz de población (borne 58) conectada
- Detección de colisión de la unidad de mando de airbag
- Estado del EWS (llave insertada, llave válida, etc.)

Siendo esto así, tenemos una base probada de donde partir nuestra investigación de obtención de datos, pues **si conseguimos un acceso con el bus K, visto ya su protocolo de funcionamiento, al menos deberíamos ser capaces de detectar los estados de cerraduras, limpiaparabrisas, luces, y detectar la conexión de la llave.**

## 3.6- Raspberry Pi 2 y PCB necesaria para la comunicación

En la introducción, hablábamos de la existencia de múltiples microcontroladores con muchas posibilidades en el mundo de la electrónica de confort, o a nivel usuario.

Pues bien, para la consecución de este proyecto, se planteó el uso de una **Raspberry Pi 2,** un microcontrolador que se podría considerar un ordenador de placa reducida, ordenador de placa única u ordenador de placa simple (SBC) de bajo coste desarrollado en Reino Unido por la Fundación Raspberry Pi.

Aunque no se indica expresamente si es hardware libre o con derechos de marca, en su sección de preguntas y respuestas frecuentes (FAQs) explican que disponen de contratos de distribución y venta con empresas pero al mismo tiempo cualquiera puede convertirse en revendedor o redistribuidor de los equipos, por lo que se entiende que es un producto con propiedad registrada pero de uso libre.

Tampoco deja claro si es posible utilizarlo a nivel empresarial u obtener beneficios con su uso, asunto que se debe consultar con la fundación.

Sin embargo, fue un microcontrolador hecho con objeto de servir de herramienta educativa en la enseñanza de ciencias de computación, por lo que nosotros utilizaremos éste equipo con ese fin, educativo.

En cambio el software sí es open source, siendo su sistema operativo oficial una versión adaptada de Debian, denominada RaspBian. Aunque permite otros sistemas operativos, incluido una versión de Windows 10.

El diseño incluye un System-on-a-chip Broadcom BCM2835, que contiene un procesador central (CPU) 900 MHz quad-core ARM Cortex A7, además, el firmware incluye unos modos "Turbo" para que el usuario pueda hacerle overclock de hasta 1 GHz sin perder la garantía.

Un procesador gráfico (GPU) VideoCore IV, y 1 GB de memoria RAM.

El diseño no incluye un disco duro ni unidad de estado sólido, ya que usa una tarjeta SD para el almacenamiento permanente; tampoco incluye fuente de alimentación ni carcasa.

Originalmente, la primera Raspberry fue una revolución, pero en comparación con la Raspberry Pi 2 que se utilizará en este proyecto, se pueden observar grandes diferencias.

A continuación se presenta una tabla resumen de las características técnicas de ambas:

| | RASPBERRY PI MODEL B+ | RASPBERRY PI 2 MODEL B |
|---|---|---|
| **SoC** | Broadcom BCM2835 | Broadcom BCM2836 |
| **CPU** | ARM11 ARMv6 700 MHz | ARM11 ARMv7 ARM Cortex-A7 4 núcleos @ 900 MHz |
| **Overclocking** | Sí, hasta velocidad Turbo; 1000 MHz ARM, 500 MHz core, 600 MHz SDRAM, 6 overvolt. de forma segura | Sí, hasta arm_freq=1000 sdram_freq=500 core_freq=500 over_voltage=2 de forma segura |
| **GPU** | Broadcom VideoCore IV 250 MHz. OpenGL ES 2.0 | Broadcom VideoCore IV 250 MHz. OpenGL ES 2.0 |
| **RAM** | 512 MB LPDDR SDRAM 400 MHz | 1 GB LPDDR2 SDRAM 450 MHz |
| **USB 2.0** | 4 | 4 |
| **Salidas de vídeo** | HDMI 1.4 @ 1920x1200 píxeles | HDMI 1.4 @ 1920x1200 píxeles |
| **Almacenamiento** | microSD | microSD |
| **Ethernet** | Sí, 10/100 Mbps | Sí, 10/100 Mbps |
| **Tamaño** | 85,60x56,5 mm | 85,60x56,5 mm |
| **Peso** | 45 g | 45 g |
| **Consumo** | 5v, 600mA | 5v, 900mA, aunque depende de la carga de trabajo de los 4 cores |
| **Precio** | 35 dólares | 35 dólares |

*Figura 14. Características técnicas Raspberry Pi / 2.*

Para poder trabajar con la Raspberry Pi 2, se requiere de una serie de pasos previos, entre los que se incluyen la instalación del S.O., posibles periféricos deseados, configuración del Software, etc…

Como hay multitud de tutoriales en existencia de la configuración de la raspberry Pi 2, omitiremos las primeras explicaciones en este documento, y haremos referencia a la **página oficial de Raspberry Pi** donde podemos encontrar tutoriales para su configuración: https://www.raspberrypi.org/help/quick-start-guide/

En este caso, el sistema operativo que utilizaremos será RASPBIAN JESSIE, que podemos encontrar en la misma página.

En el momento de desarrollo del proyecto, se disponía de la versión de Kernel 4.17, sin embargo, en versiones posteriores puede realizarse este proyecto sin problemas de compatibilidad.

Raspbian es un sistema operativo de base Linux. En la introducción de este documento ya se justificó el uso de éste entorno, por lo que no entraremos en más detalle.

Llegados a este punto, hemos definido el dispositivo sobre el que trabajaremos, pero **¿cómo conseguiremos comunicar el Bus K del coche con el hardware que utilicemos?**

En puntos previos, hemos determinado que el Bus K trabaja en el protocolo ISO-9141-2. También fijamos que la forma de actuar del bus tiene gran similitud con el protocolo serie RS232.

Por tanto, el primer paso es encontrar los medios necesarios para traducir desde el bus al protocolo serie, para después conseguir conectar el bus a través de un USB o algún otro sistema de conexión a un ordenador o la propia Raspberry.

Bien, comenzando desde el lado del bus del coche, genéricamente encontraríamos en cualquier conexión al menos tres puntas: 12V, el bus en sí y tierra.

Para poder trabajar con ello, lo primero que necesitaremos es un transceiver del bus al estándar de la comunicación por puerto serie.

Tras la investigación en diversas webs, se establece que un elemento perfectamente compatible, de características deseadas y un precio no muy elevado es el microchip TH3122 de Melexis

Consiste en un regulador 5V/100mA de baja caída de tensión integrado con el transceiver del K-bus. La combinación del regulador de voltaje y transceiver junto a las funciones de monitorización permiten trabajar el bus de forma barata y óptima.

Como los propios desarrolladores indican, la amplia área de corrientes de salida y el tiempo de reset y voltaje de reset permiten funcionar junto a muchos diferentes microcontroladores.

En la siguiente página aparecen los diagramas funcionales y el bloque del transceiver del bus:

**Functional Diagram**

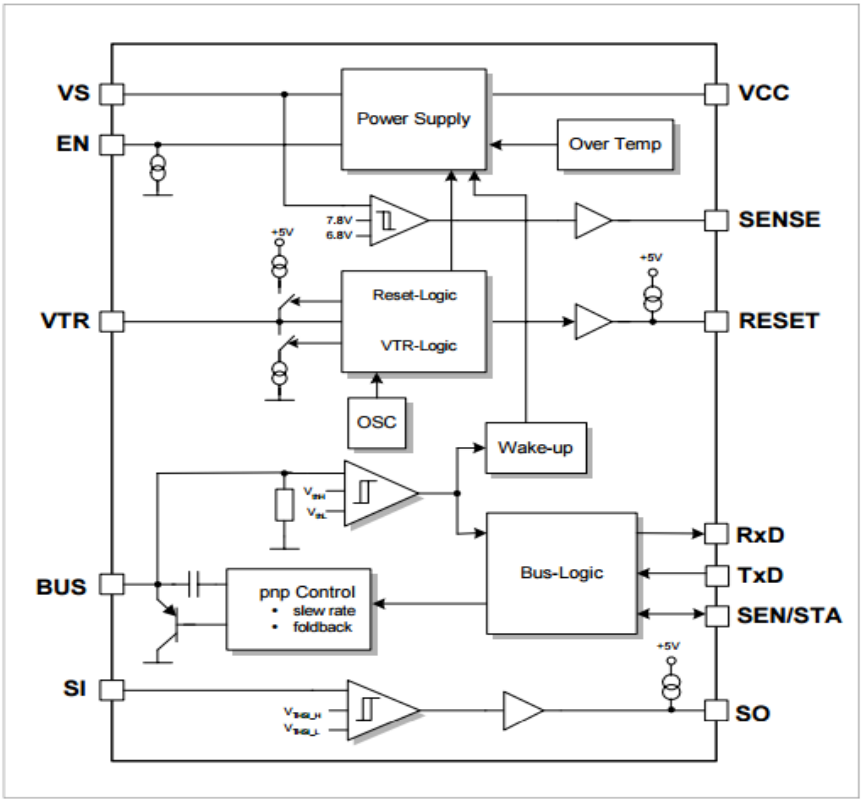



*Figura 15. Diagrama funcional TH3122.*

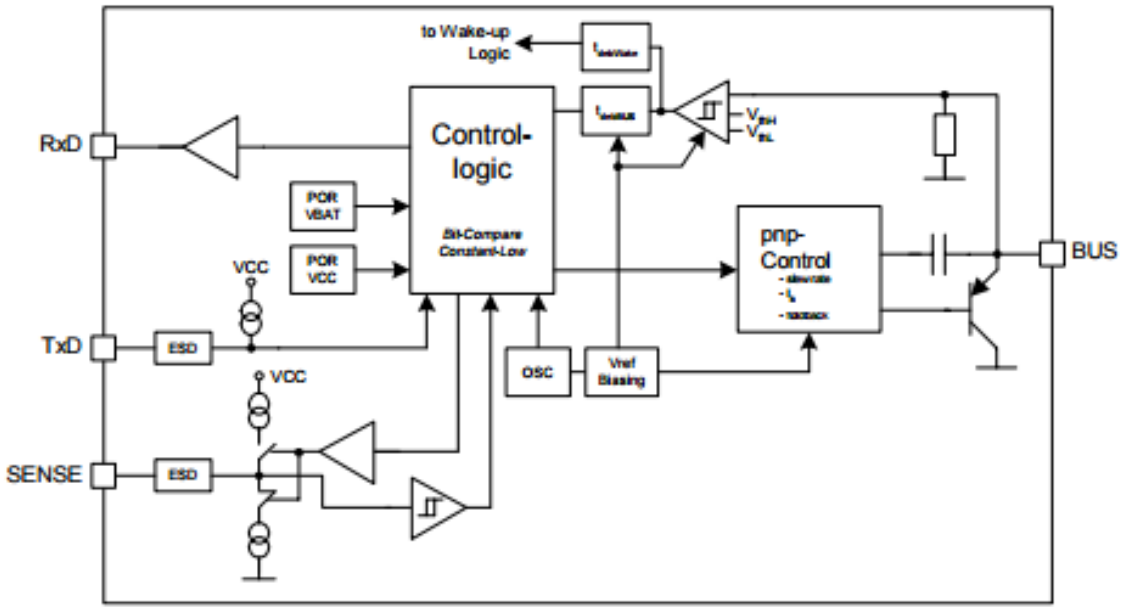



*Figura 16. Diagrama k-bus Transceiver.*

¿Cómo funcionará el interfaz del K-Bus?

El interfaz del BUS permitirá la conexión entre los 5V del protocolo serie y los 12 voltios de la línea del K-Bus.

El transceiver consiste en un transistor pnp con control del slew rate y barreras a la sobre carga que también contiene en el receptor un comparador de alto voltaje seguido una unidad anti-rebotes (debouncing).

Algunos de los puntos interesantes a explicar son: el modo de recepción, de transmisión, el bit Compare y la señal SEN/STA

Durante la transmisión de los datos, el pin TxD se transferirá al pin BUS. A continuación se presenta una imagen de la transmisión de un pulso:

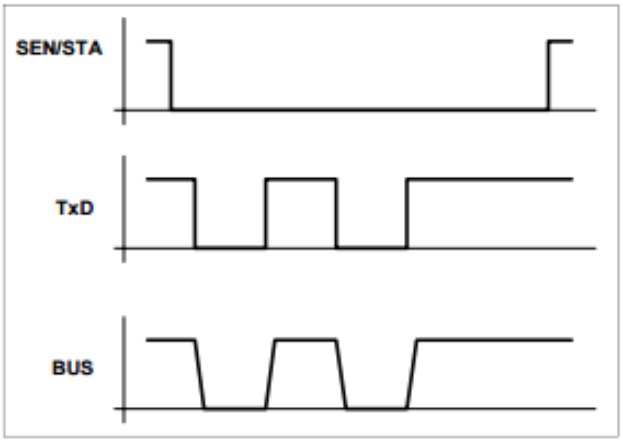



*Figura 17. Diagrama de un pulso en modo transmisión.*

Sin embargo, durante la recepción, los datos del pin BUS se transmitirán al pin RxD tal como se muestra en la imagen a continuación, donde se ve como se eliminan los rebotes:

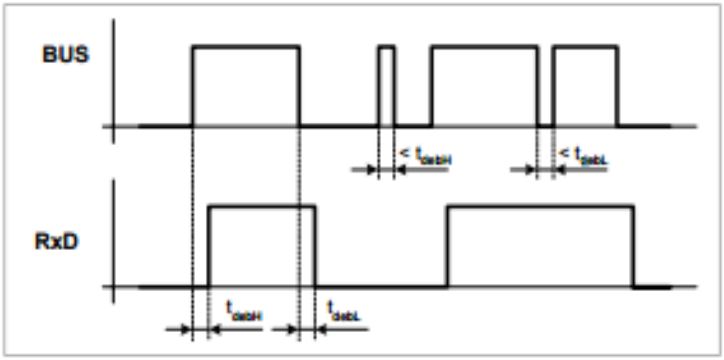



*Figura 18. Diagrama de un pulso en modo recepción.*

Si las señales en el pin TxD y el pin BUS no son idénticas dentro de un tiempo establecido, la transmisión se interrumpirá. Si sendas señales tienen nivel alto durante el tiempo establecido de enable, la transmisión se iniciará. La función del bit compare esta activa cuando SEN/STA se usa como salida:

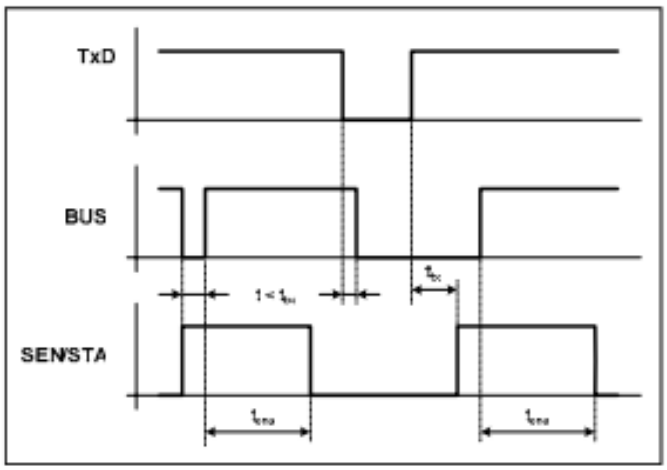



*Figura 19. Diagrama bit compare.*

El pin SEN/STA es bidireccional. Usándolo como salida, indica cuando el camino de la transmisión está activado o no:

SEN/STA =”0” transmission path is enabled SEN/STA =”1” transmission path is disabled

Como entrada indica si el camino de transmisión puede ser sobreescrito (Independientemente del bit compare).

SEN/STA=”0” forcing the transmission path free SEN/STA=”1” disable the transmission path

En esencia, estas son algunas de las características que debemos tener en cuenta al empezar a pensar en un diseño, aunque más información se puede encontrar en su datasheet.

Una vez que tenemos claro que usaremos el TH3122, debemos pensar en un circuito para adaptarlo al bus.

De nuevo, con muchos ejemplos de web e intentos por trabajar con este bus, podemos usar el siguiente esquema, teniendo en cuenta el uso del filtro RC para reducir el ruido que recomienda el datasheet, y las capacidades C1 Y C2 para evitar posibles caídas de voltaje:

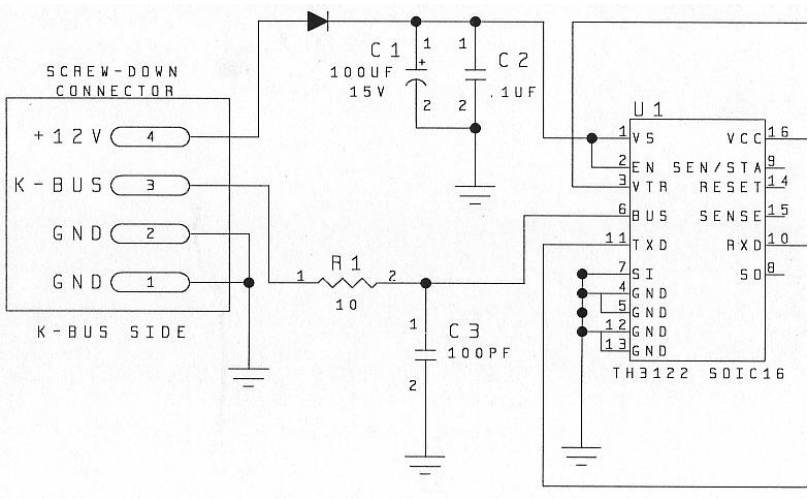



*Figura 20. Ejemplo de conexión del bus al TH3122.*

Llegados a este punto, el paso lógico a seguir es traducir al protocolo de transmisión serie RS232.

Para ello, un equipo muy comúnmente utilizado es el MAX232, que genéricamente se usa para llegar al conector de 9 pines del protocolo serie.

En alguno de los links del final del documento se puede ver como se utilizó esta solución para proyectos similares. El protocolo de comunicación serie es un mundo muy explotado y que tendrá en el punto a continuación una explicación mayor.

Una de las vías inmediatas para llegar a esta meta es la que se presenta a continuación, que pese a que no será la utilizada, es cuanto menos curiosa:

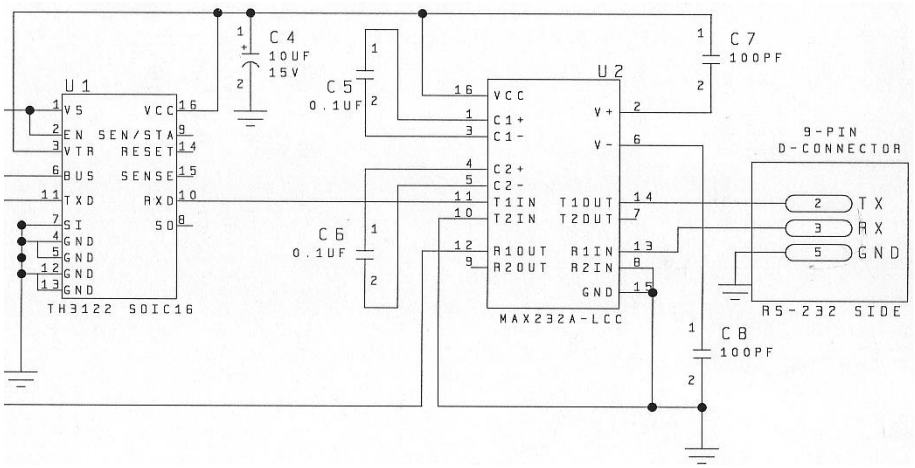



*Figura 21. Ejemplo de conexión del TH3122 al Conector serie RS232.*

Pese a que lo mostrado en la página superior es un modelo funcional, nosotros buscamos en éste proyecto una posibilidad de conexión más común. USB al ser posible como se ha comentado anteriormente.

Para ello, encontramos el chip FT232BM.

Un interface de USB a UART serie que permite transmisión de datos asíncrona, el protocolo USB totalmente integrado en el chip sin necesidad de programación adicional, una memoria EEPROM de 1024 bit. Compatible con RS485 Y RS232 con ratios de transferencias desde 300 baudios a 3Mbaudios, FIFO para los bufferes de transferencia y recepción, 3.3V integrados para las I/O USB, conversor integrado de UART, etc…

Mencionadas algunas de sus características, adjunto en los links del principio se encuentra el enlace a su datasheet, donde podemos observar todas las características deseables del chip.

Con el esquemático de aplicación del datasheet a continuación, podremos concluir que la conexión será realmente sencilla:

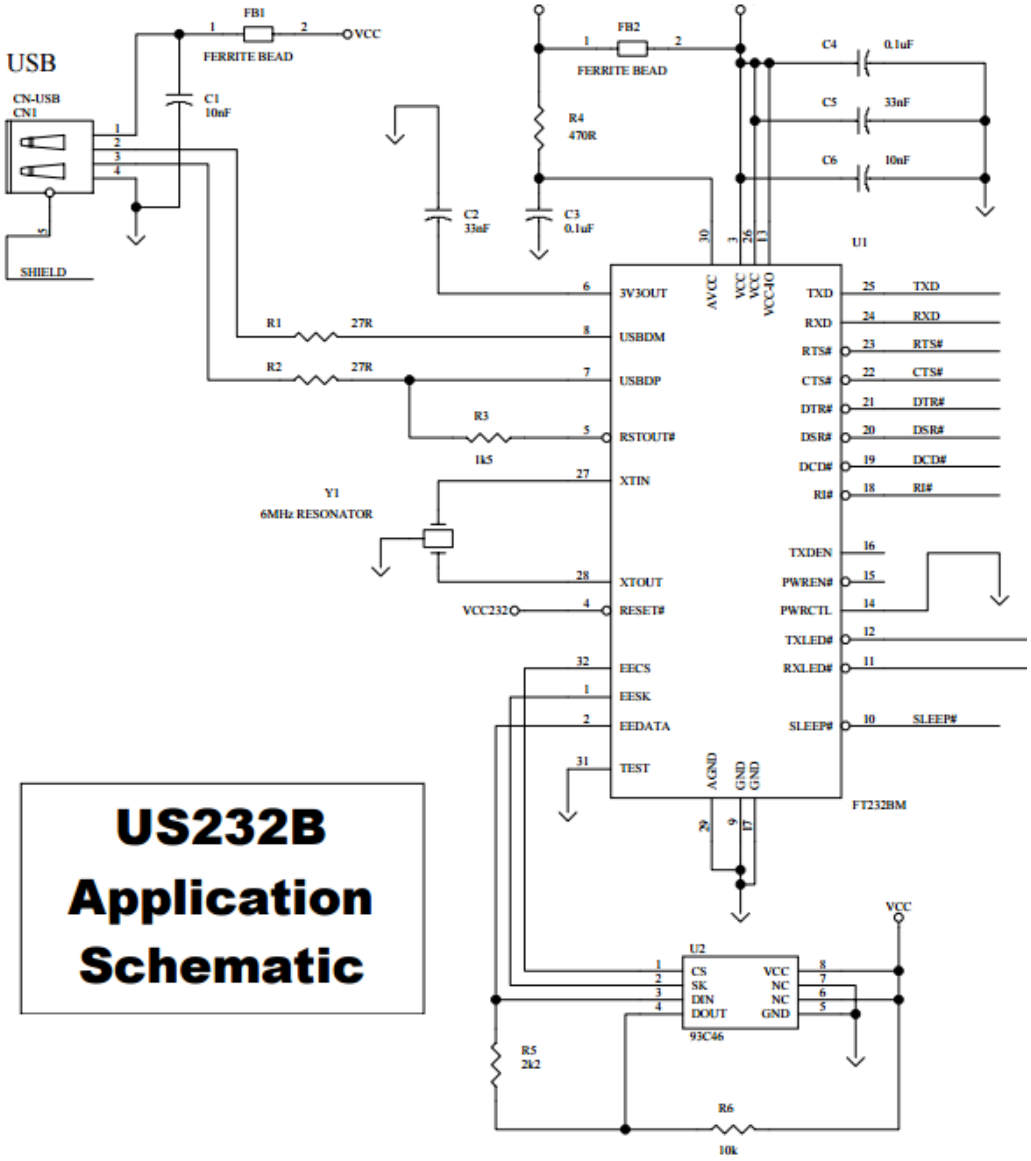



*Figura 22. Ejemplo de aplicación del FT232BM en la conexión USB.*

Bastará conectar las señales TXD Y RXD que obtenemos del TH3122 presentado anteriormente para hacer funcionar el circuito.

En ambas imágenes superiores se incluyen los componentes típicos de los circuitos de electrónica (resistencias, condensadores para desacoplos…). Debido a su uso tan estándar, no entraremos en detalles.

Una vez que tenemos claros los componentes, sólo necesitamos realizar la PCB a través de un programa como Eagle y enviarla a alguna página de diseño de PCB, al no ser posible realizarla uno mismo con facilidad.

Como alternativa, buscando por la red la combinación de los elementos seleccionados en un interfaz ya montado, encontramos la página de Rolf Resler (adjunto link en el final del documento).

En ésta página, además de encontrar más información sobre los buses K e I de los automóviles, se ofrecen esquemas y soluciones para la conexión a esos buses a precios razonables, tanto para el protocolo serie como USB, siendo esta última basada en los componentes descritos anteriormente.

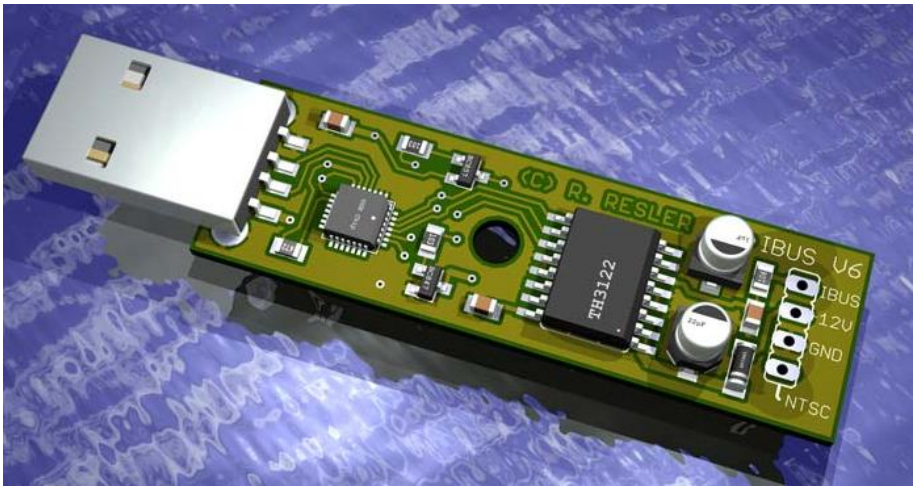



*Figura 23. Interfaz Resler de comunicación USB-KBUS.*

Por tanto, en caso de disponer de la posibilidad de montar una PCB propia, el precio de los componentes utilizados es realmente bajo y la opción preferida. Sin embargo, como vemos en la imagen el diseño Resler cuenta con todos los componentes que hemos incluido en el diseño.

El único cambio notable es la sustitución del chip FT232BM que hemos usado (que no requiere de ningún software adicional) por el chip CP2102 (La otra alternativa más común en cuanto a conversores UART-USB).

Nosotros para éste proyecto adquiriremos uno de éstos interfaces, por tanto, el chip que usaremos es el CP2102.

En el caso de decidir usar un CP2102 como lo hace éste interfaz, las conexiones serían como sigue:

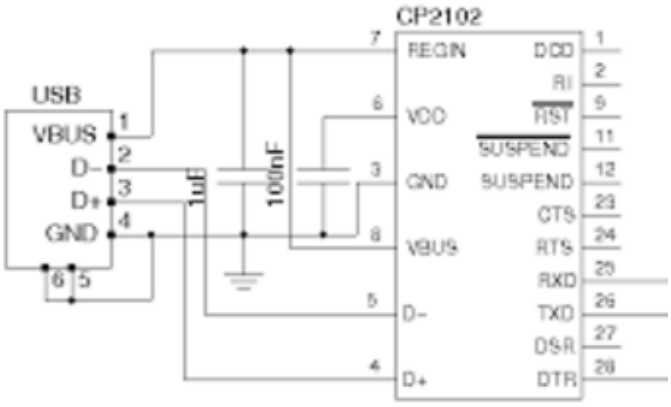



*Figura 24. Ejemplo de aplicación del CP2102 en la conexión USB.*

Así hemos conseguido comunicar el bus con el USB y queda concluido este punto.

## 3.7- Protocolo comunicación serie

A lo largo del proyecto habíamos hablado con mucha frecuencia del Protocolo de Comunicación puerto serie y hecho referencia al protocolo RS-232, pero no habíamos entrado en detalles. En este apartado trataremos todo lo importante sobre ello.

La comunicación RS-232 es de tipo serial. Se llama serial, porque los bits se reciben uno detrás de otro o "en serie".

El puerto serie o puerto serial, es un dispositivo muy extendido y ya sea uno o dos puertos, con conector grande o pequeño, casi todos los equipos PC lo incorporan actualmente.

Debido a que el estándar del puerto serial se mantiene desde hace muchos años, la institución de normalización americana (EIA) ha escrito la norma RS-232-C que regula el protocolo de la transmisión de datos, el cableado, las señales eléctricas y los conectores en los que debe basarse una conexión RS-232.

La comunicación realizada con el puerto serial es una comunicación asíncrona. Para la sincronización de una comunicación se precisa siempre de un bit adicional a través del cual el emisor y el receptor intercambian la señal del pulso.

Sin embargo, en la transmisión serial a través de un cable de dos líneas esto no es posible ya que ambas están ocupadas por los datos y la tierra.

Cuando se transmite información a través de una línea serie es necesario utilizar un sistema de codificación que permita resolver los siguientes problemas:

**Sincronización de bits:** El receptor necesita saber dónde comienza y donde termina cada bit en la señal recibida para efectuar el muestreo de la misma en el centro del intervalo de cada símbolo (bit para señales binarias).

Por este motivo se intercalan bit con información de actuación antes y después de los datos de información según el protocolo RS-232 que leerán emisor o receptor para evitar las colisiones. Los siguientes Bit son los que aseguran el funcionamiento correcto en este caso:

- **Bit de inicio.** Cuando el receptor detecta el bit de inicio sabe que la transmisión ha comenzado y es a partir de entonces cuando debe leer la transmisión, leyendo las señales de la línea a distancias concretas de tiempo, en función de la velocidad determinada.
  Durante el intervalo de tiempo en que no son transferidos caracteres, el canal debe poseer un "1" lógico. Al bit de parada se le asigna también un "1". Al bit de inicio del carácter a transmitir se le asigna un "0". Por todo lo anterior, un cambio de nivel de "1" a "0" lógico le indicará al receptor que un nuevo carácter será transmitido.

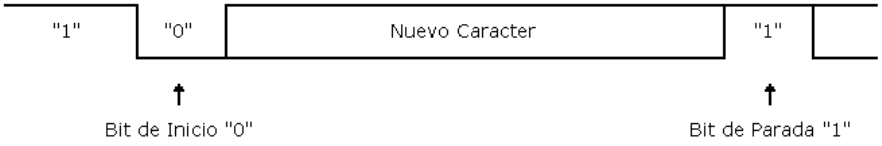



*Figura 25. Formato de transmisión asíncrona.*

- **Bit de parada.** Indica la finalización de la transmisión de una palabra de datos. El protocolo de transmisión de datos permite 1, 1.5 y 2 bits de parada.
- **Bit de paridad.** Con este bit se pueden descubrir errores en la transmisión.

Se puede dar paridad par o impar. En la paridad par, por ejemplo, la palabra de datos a transmitir se completa con el bit de paridad de manera que el número de bits "1" enviados es par.

**Sincronización del carácter:** La información serie se transmite por definición bit a bit, pero la misma tiene sentido en palabras o bytes.

**Sincronización del mensaje:** Es necesario conocer el inicio y fin de una cadena de caracteres por parte del receptor para, por ejemplo, detectar algún error en la comunicación de un mensaje.

Consideraciones sobre la transmisión asíncrona:

La transmisión asíncrona que vamos a ver es la definida por la norma RS232 que se basa en las siguientes reglas:

Cuando no se envían datos por la línea, ésta se mantiene en estado alto (1).

Cuando se desea transmitir un carácter, se envía primero un bit de inicio que pone la línea a estado bajo (0) durante el tiempo de un bit.

Durante la transmisión, si la línea está a nivel bajo, se envía un 0 y si está a nivel alto se envía un 1.

A continuación, se envían todos los bits del mensaje a transmitir con los intervalos que marca el reloj de transmisión. Por convenio se transmiten entre 5 y 8 bits.

Se envía primero el bit menos significativo, siendo el más significativo el último en enviarse.

A continuación del último bit del mensaje se envía el bit (o los bits) del final que hace que la línea se ponga a 1 por lo menos durante el tiempo mínimo de un bit. Estos bits pueden ser un bit de paridad para detectar errores y el bit o bits de stop, que indican el fin de la transmisión de un carácter.

Los datos codificados por esta regla, pueden ser recibidos siguiendo los pasos siguientes:

- Esperar la transición 1 a 0 en la señal recibida.
- Activar el reloj con una frecuencia igual a la del transmisor.
- Muestrear la señal recibida al ritmo de ese reloj para formar el mensaje.
- Leer un bit más de la línea y comprobar si es 1 para confirmar que no ha habido error en la sincronización.

En la transmisión asíncrona por cada carácter se envía al menos 1 bit de inicio y 1 bit de parada así como opcionalmente 1 bit de paridad. Esta es la razón de que los baudios no se correspondan con el número de bits de datos que son transmitidos.

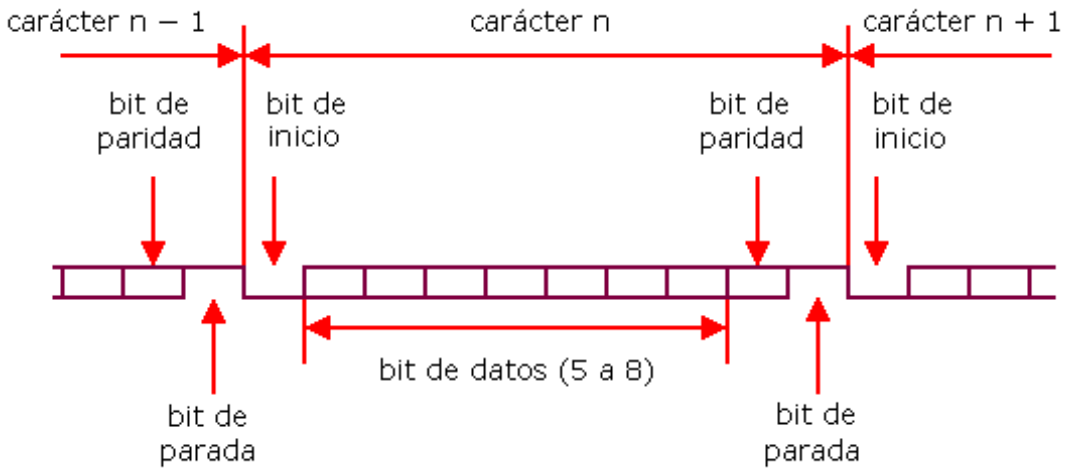



*Figura 26. Ejemplo transmisión asíncrona.*

Cuando se escriben o se envían datos, pueden producirse errores, entre otras cosas, por ruidos inducidos en las líneas de transmisión de datos. Es por tanto necesario comprobar la integridad de los datos transmitidos mediante algún método que permita determina si se ha producido un error.

En un caso típico, si al transmitirse un mensaje se determina que se ha producido un error, el receptor solicita de nuevo el mensaje al emisor.

Se pueden detectar errores de acuerdo a la forma de transmisión, en el caso de la transmisión asíncrona hay errores de: Paridad, sobre escritura y el error de encuadre (framing).

**Finalmente, trataremos el Método checksum**

Puede existir el caso en que, por ejemplo, se alteren dos bits en un caracter transmitido y si se ha implementado la comprobación de paridad, el error no será detectado.

Existen otros métodos de detección de errores como son la comprobación de redundancia cíclica (CRC) y la comprobación de suma (checksum). Por su simplicidad, será abordado el método checksum.

El método checksum puede ser utilizado tanto en la transmisión síncrona como en la asíncrona. Se basa en la transmisión, al final del mensaje, de un byte (o bytes) cuyo valor sea el complemento a dos de la suma de todos los carácteres que han sido transmitidos en el mensaje. El receptor implementará una rutina que suma todos los bytes de datos recibidos y al resultado se le sumará el último byte (que posee la información en complemento a dos de la suma de los caracteres transmitidos) y si la recepción del mensaje ha sido correcta, el resultado debe ser cero.

## 3.8- Entorno de trabajo (Windows-Linux-Putty)

Tal como se había comentado anteriormente en éste documento, el interfaz que hemos desarrollado tenía previsión de poderse usar tanto en un ordenador convencional como en un microcontrolador (nuestro objetivo).

El primer paso será conectarnos a un ordenador portátil para poder empezar a trabajar.

El ordenador disponible tiene el S.O. Windows 10.

Una vez que se haya conectado el interfaz diseñado, tendremos que utilizar algún programa que nos permita leer la información del bus que estamos obteniendo vía USB.

Al ser PuTTy un cliente SSH y TCP raw de libre licencia, primero se intentó conectar por ésta vía sin obtener resultados exitosos (véase sección de pruebas más adelante en el proyecto)

De todos modos, usaremos PuTTy en la conexión de la Raspberry pi 2 con el ordenador.

Para conseguir una lectura directa de los datos del BUS-K utilizaremos el programa HyperTerminal, versión reducida de HyperACCESS muchas veces incluido en Windows.

"Hterm" (Abreviatura frecuente para HyperTerminal) nos permite leer en muchos formatos la información recibida por el puerto USB. El interfaz que hemos desarrollado nos proporcionará por defecto los caracteres en ASCII, pero Hterm nos permitiría convertirlos en tiempo real a Hexadecimal u otros formatos más adecuados para la identificación y clasificación de los mismos en tiempo real. Sin embargo, al no ser un entorno donde se pueda programar nada, no podremos avanzar más allá de la lectura, por ello, pasaremos a la Raspberry Pi.

Partiendo de que se ha seguido el tutorial especificado en el punto 3.6 del documento, asumimos que se ha logrado la instalación de Raspbian JESSIE en la Raspberry.

Hay muchos métodos posibles de trabajar con la Raspberry, pero nosotros trabajaremos con ella desde el propio ordenador.

Por ello necesitábamos un emulador de terminal para conseguir control sobre la Raspberry, que conseguimos a través de PuTTy por el ya mencionado SSH (Protocolo y programa que lo implementa que permite manejar una computadora mediante un intérprete de comandos).

SSH nos permite copiar datos de forma segura y trabaja con técnicas de cifrado que hacen que la información que viaja por medio de la comunicación no sea legible.

De éste modo, conseguiremos entrar en la Raspberry a través del terminal.

Éste paso podría omitirse en caso de trabajar con una pantalla con conexión HDMI, pues dentro de la Raspberry se trabaja en un entorno Linux, por lo que se podrían conectar todos los periféricos a la propia Raspberry y desde ahí conectarnos al terminal de Linux.

# 4- Colocación del interfaz y pruebas en el entorno de Windows

Hasta este punto llega el estudio teórico y diseño de los componentes a utilizar.

A partir de aquí, se exponen las diferentes pruebas que resultaron en éxitos o información importante para el desarrollo del proyecto, así como justificaciones de conexiones y desarrollo de software.

## 4.1- Punto de conexión con el K-Bus del coche

Todos los modelos de BMW siguen una estructura similar en cuanto al mapeo de conexiones del bus K. Sin embargo, dependiendo del modelo, el año y la serie de coche que se trate, podemos encontrar varias diferencias.

Nuestro caso es uno de los más limitantes. Es un coche del año 1998, de los primeros de la serie. Muchas funciones que se añadieron a partir de 1999 o 2000 no están disponibles al no tener conexión al bus.

Generalmente, hay tres puntos de conexión de relativamente fácil acceso al bus en todos los modelos de BMW:

- La conexión al cargador de CDs en el maletero.

- La conexión a la radio (requiere desmontaje complejo).

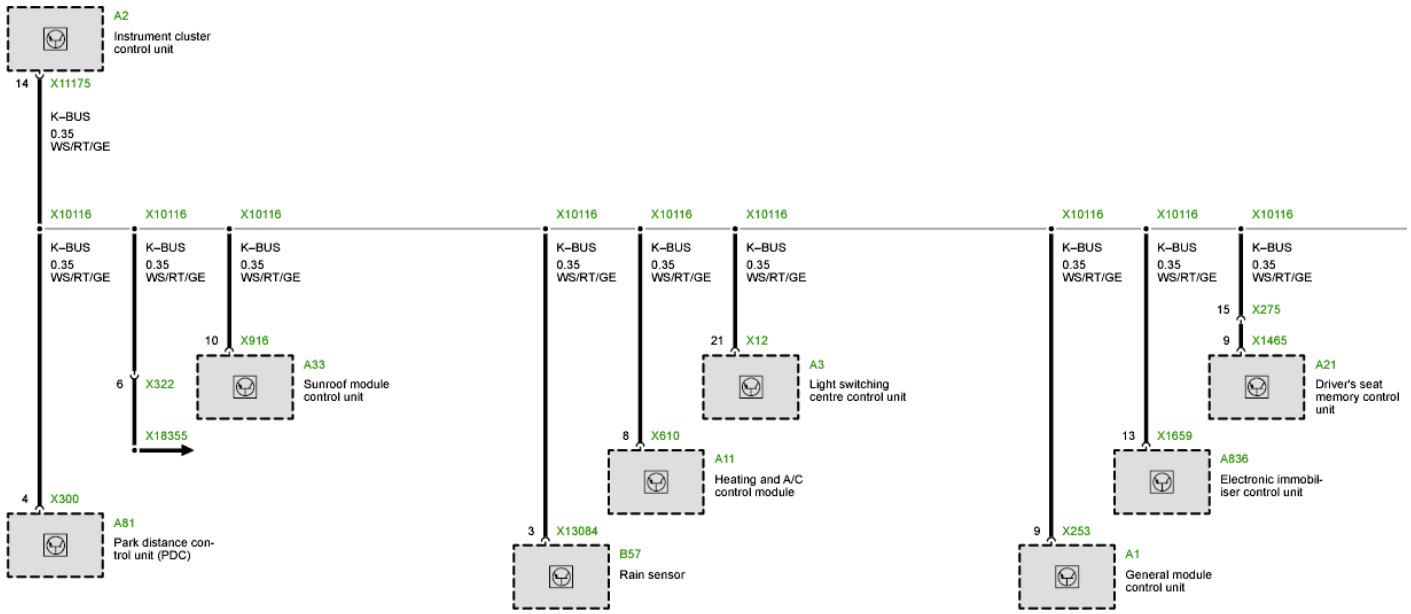
- El terminal del K-bus en la parte superior de la caja de fusibles. (Conexión complicada).

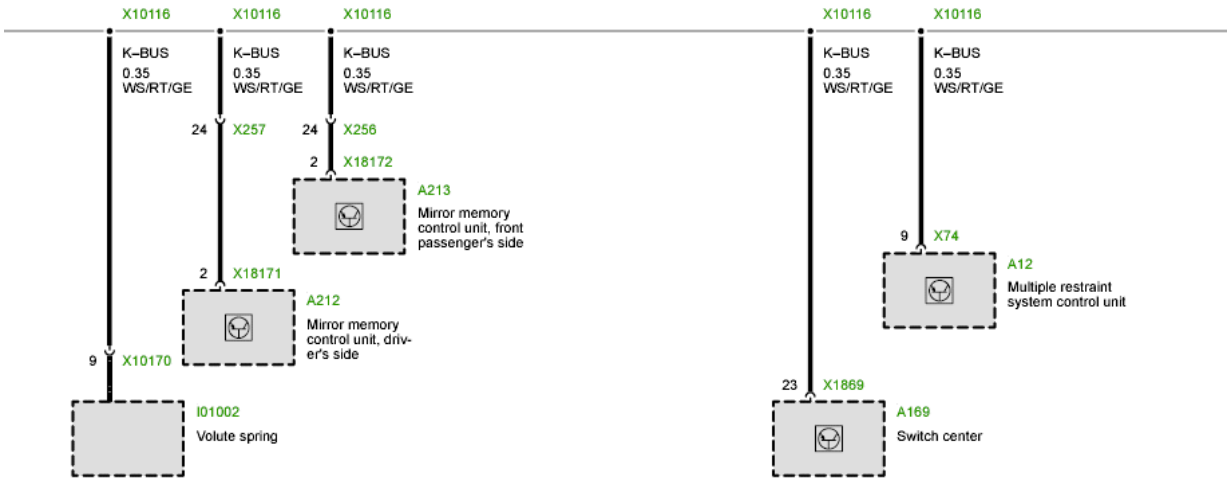
-Cualquier punto del cable a lo largo del coche. Esto es, por ejemplo, desmontar los embellecedores de una puerta para ver todos los cables que circulan por ahí, localizar el correspondiente al bus y a través de algún elemento adicional, por ejemplo un alfiler conductor, pinchar el bus.

Tal como hemos explicado en la parte teórica del proyecto, la existencia de las **Gateway** podría limitar nuestro alcance en cuanto a actuación, pues es posible que no todos los mensajes se reciban según en qué punto nos conectemos.

En nuestro caso, el coche es totalmente funcional y se le da uso diario, por lo que debemos buscar el método menos invasivo posible. Por ello, nos conectaremos a través del cargador de CD del maletero.

Hemos dicho que según los modelos de BMW el bus K puede variar en cuanto a conexiones, sin embargo, con la siguiente imagen, podemos hacernos una idea de a qué conecta el K-Bus de BMW:






*Figuras 27 y 28. Ejemplo de conexiones del K-BUS BMW.*

Nótese que en nuestro caso, no tendremos ni las memorias de los asientos, ni el techo solar, ni ninguno de los extras posteriores a 1998.

Con la conexión que pretendemos, accederemos al módulo ZKE V **que se explicó en el apartado 3.5 del proyecto.**

Recordemos que entre otros, esto sí que nos dará acceso a cierre centralizado, limpiaparabrisas, elevalunas eléctricos…

Para acceder con libertad al cargador de CDs en los BMW de serie 3, debemos abrir el maletero y quitar los embellecedores de plástico de los extremos de la izquierda:

*Figura 29. Localización cargador de CDs maletero BMW serie 3.*

Una vez dentro del maletero, debemos desmontar el cargador de CDs, dejando a la vista dos cables tal y como aparece en la siguiente imagen:

*Figura 30. Cargador de CDs en el maletero*

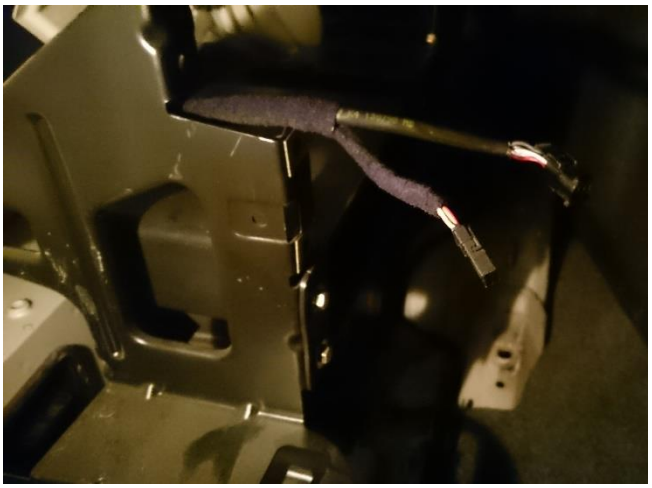


*Figura 31. Desmontado el cargador, cables*

Debemos localizar el conjunto correspondiente al K-Bus. En el modelo de coche que trabajamos, dado su año de fabricación, sabemos que hay tres cables que conectarán al cargador: El K-Bus, la alimentación y tierra.

Generalmente, todos los cables de buses de datos tienen 3 colores (dos colores y un rayado).

En nuestro caso, localizamos aquí el bus:

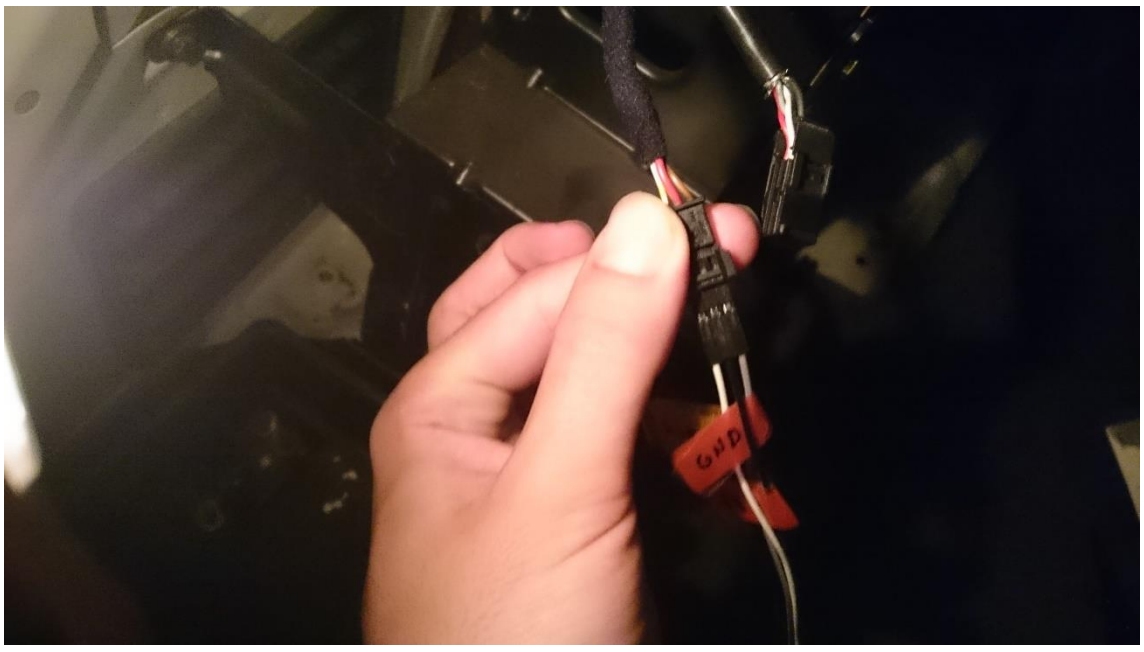


*Figura 32. Conjunto cables K-Bus*

Como se puede apreciar en la imagen, distinguimos 3 cables, uno marrón que es la tierra, uno rojo y verde, que es la alimentación de 12V de la que dispone el coche y el último, blanco con línea roja y rayas horizontales amarillas, que corresponde con el K-Bus.

Para mayor facilidad de conexión, se usaron los cables de la imagen superior.

Con el interfaz diseñado, debemos colocar cada cable en la posición correspondiente. (En caso de haber adquirido un interfaz Resler, el cable de NTSC quedaría simplemente al aire sin conectar).

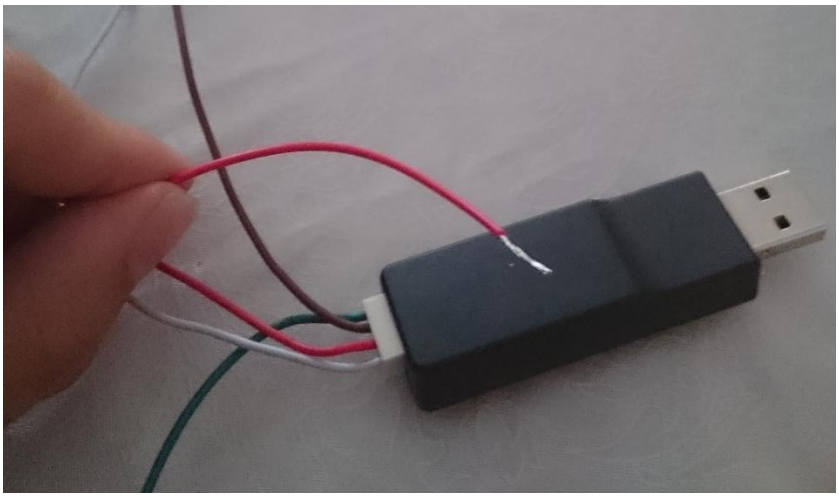


*Figura 33. Interfaz Resler v6*

Buscando mayor comodidad, al interfaz se le han acoplado cables que permitirán una conexión más sencilla al bus por la forma de la clavija:

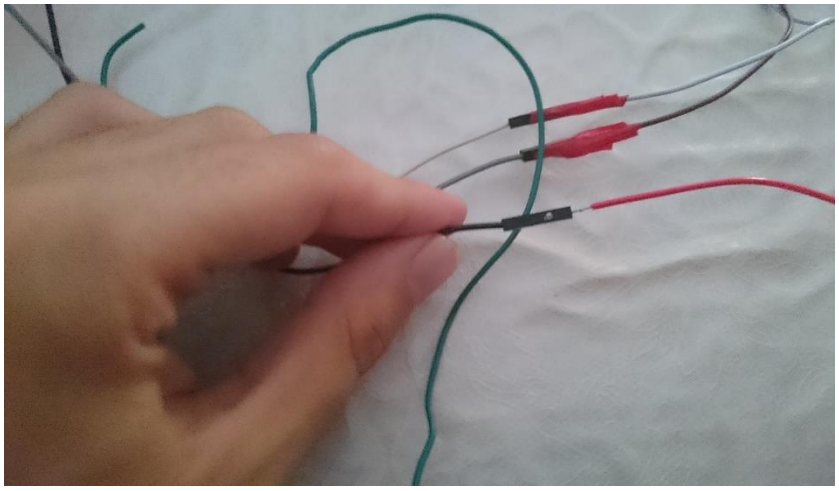


*Figura 34. Acoplo de cables al interfaz*

De este modo, usando la cinta aislante evitamos cualquier problema y queda una conexión más robusta tal como se puede observar en la siguiente página:


*Figura 35. Interfaz de conexión al bus final.*

Con esto, y tras conectar el interfaz al USB del ordenador, habremos conseguido la conexión.

## 4.2- Identificación de la velocidad del Bus

Una vez realizada la conexión, necesitamos los drivers para el controlador.

Para nuestro diseño, deberemos entrar a la página de Silabs y descargar los drivers adecuados para nuestra versión de Windows. El link se encuentra en la sección de enlaces del principio del documento.

Una vez instalados los drivers, tenemos la conexión funcional.

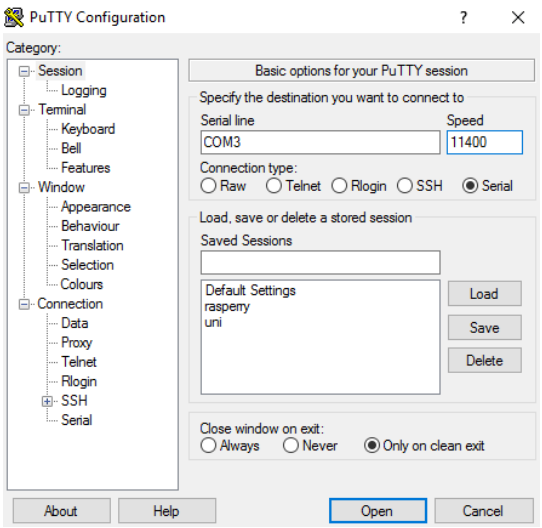



*Figura 36. Configuración PuTTY.*

Ahora requerimos de algún programa que nos permita leer lo que esté pasando por el bus al que acabamos de conectar.

Probaremos con PuTTy en primera instancia. En la conexión de PuTTy, se requiere establecer una velocidad con la pestaña serial seleccionado:

Para determinar la velocidad del bus, se hicieron las siguientes pruebas. Los resultados se muestran en la siguiente página.

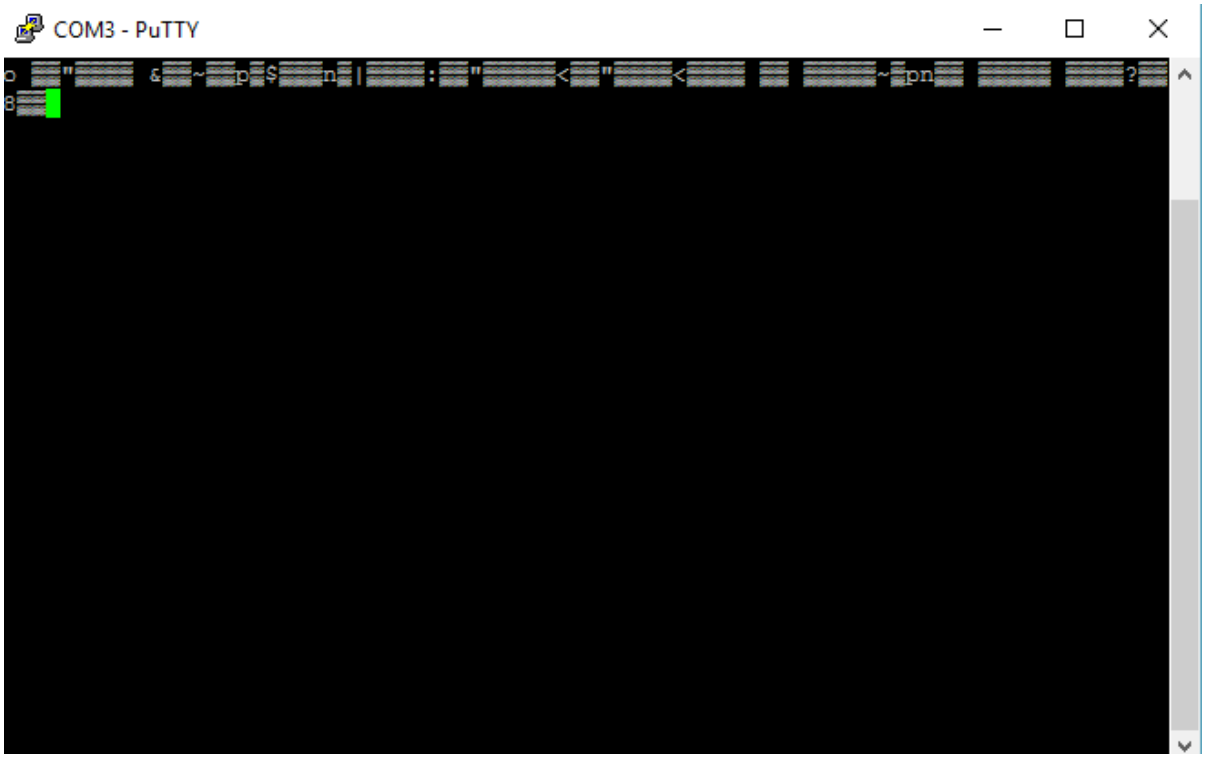



*Figura 37. Obtención de datos en PuTTY a 11400 baudios.*

En la imagen superior se ha podido observar que la comunicación ha sido un éxito, pues al abrir una puerta salieron varios caracteres.

Sin embargo, no podemos saber si es ruido o hay algo correcto o incorrecto.

Para avanzar, partimos de la premisa de que hemos colocado la velocidad mucho más alta de la que realmente es. Generalmente las comunicaciones serie trabajan en algunas de las velocidades comunes, 4800 baudios, 9600 baudios, 11400 baudios. Siguiendo con las pruebas, observamos lo siguiente:

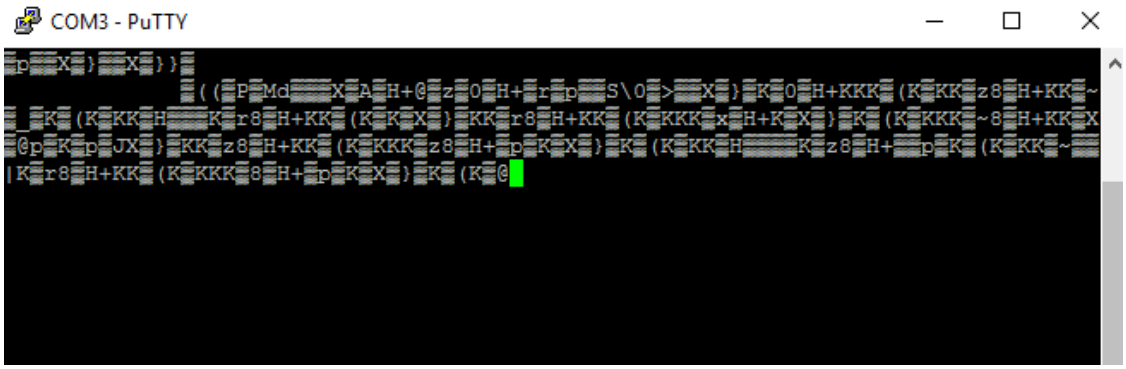



*Figura 38. Obtención de datos en PuTTY a 4800 baudios.*

Probablemente, la velocidad a la que trabaje el bus sea 9600, pues vemos que en la primera imagen teníamos menos datos que en la segunda, y que la segunda no tiene mucho sentido.

Al hacer las pruebas con 9600 baudios, obteníamos aún información que no podíamos entender, sin embargo, cada minuto aparecía un resumen del estado del coche en pantalla como se ve en la imagen a continuación:

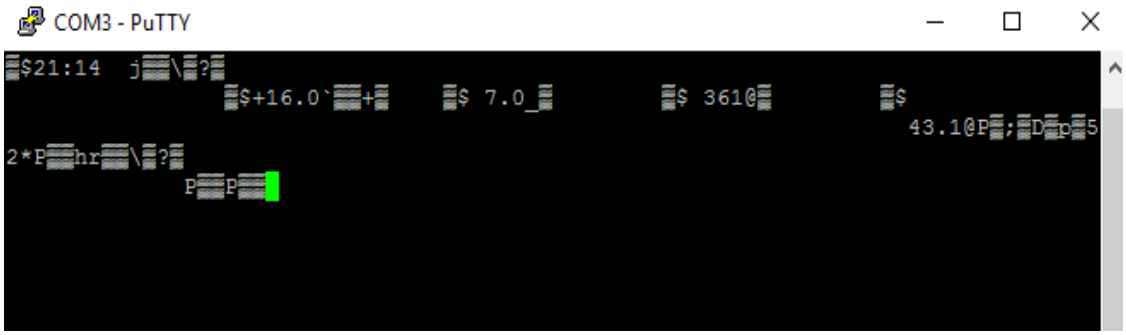



*Figura 39. Obtención de datos en PuTTY a 9600 baudios.*

Podemos ver que al dar el contacto con la llave se puso la hora en primer lugar, y el resto de números (16.0, 7.0 o 43.1) son números que están en el ordenador de a bordo del coche. (Por ejemplo, 7.0 corresponde al consumo medio)

Suponiendo que esta fuese la velocidad correcta, aunque no seamos capaces de interpretarla, probaremos algún ensayo que nos permita saber si sale siempre un mismo código para la misma acción.

Probaremos subiendo y bajando el volumen y la ventanilla derecha del coche, obteniendo los siguientes resultados:

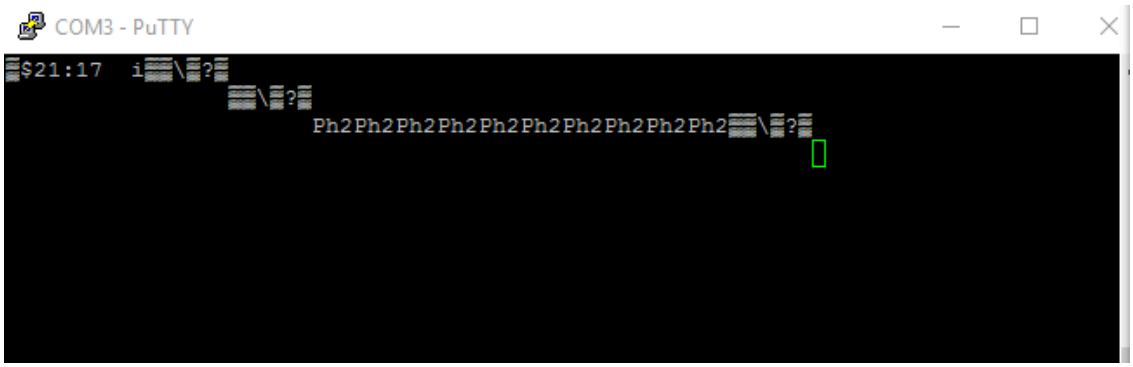



*Figura 40. Resultado de subir el volumen a 9600 baudios.*

Como observamos, siempre aparece el código Ph2 cuando subimos el volumen desde el mando habilitado para ello en el volante. Si se hace desde la propia radio, no aparece ningún mensaje.

A continuación experimentaremos con las ventanillas:

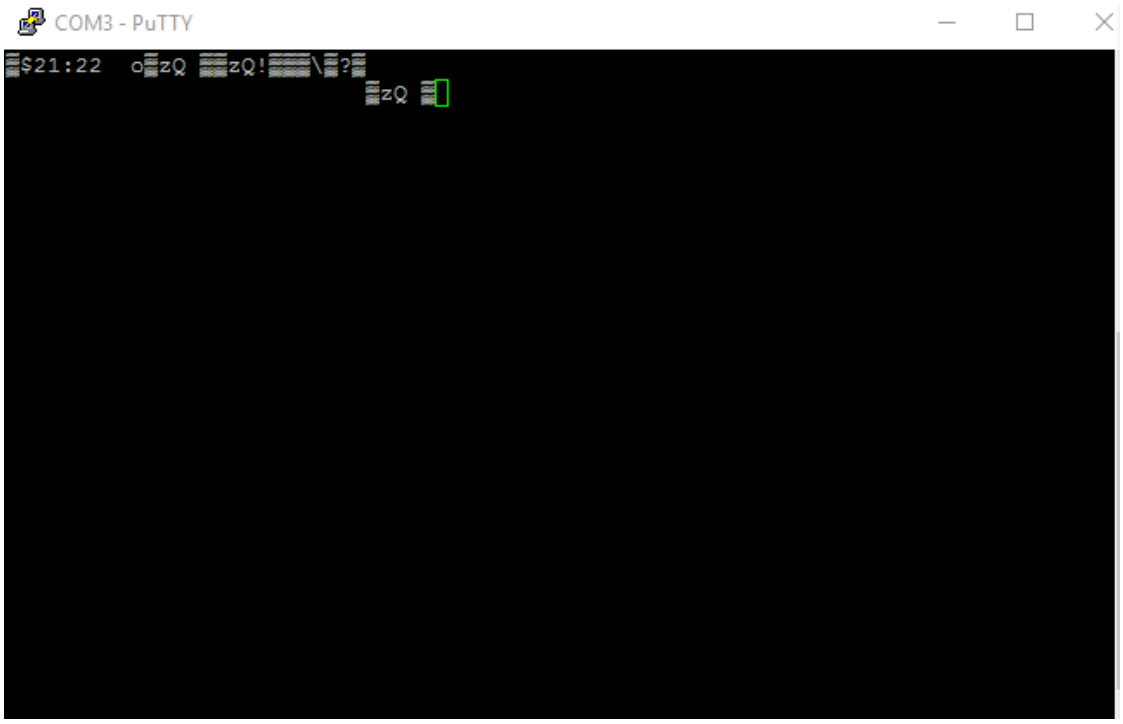



*Figura 41. Resultado de subir o bajar la ventanilla derecha a 9600 baudios.*

Trabajando a 9600 baudios, la hora cada vez que cambia se actualiza. Siguen apareciendo mensajes que no somos capaces de interpretar, pero si observamos la secuencia, de datos que aparecen, podemos concluir que sí que estamos recibiendo información del estado del coche, al igual que en la figura 40.

En este caso, cuando subíamos la ventanilla, obteníamos datos que no podíamos identificar seguido de zQ y un espacio. Al bajar la ventanilla, obteníamos el mismo formato, datos seguidos de zQ! Un espacio y más datos.

La diferencia estaba en la exclamación, por lo que es posible que los datos sean correctos pero PuTTy no sea el entorno adecuado para trabajarlos. Parece ser que la comunicación está en formato ASCII.

Por ello y tal como se ha presentado anteriormente en el documento, probaremos a usar otro programa, en este caso HyperTerminal, que permite trabajar datos del puerto serie y su traducción instantánea a otro lenguaje que sea más sencillo identificar. Asumiendo que la velocidad de 9600 baudios es la correcta, avanzamos al siguiente punto:

## 4.3- Identificación del formato de la información del bus y primeras pruebas de repetitividad

De la misma forma que conectamos el ordenador a través de PuTTy, lo haremos con el Hterm. En la siguiente imagen podemos ver la pantalla del programa:

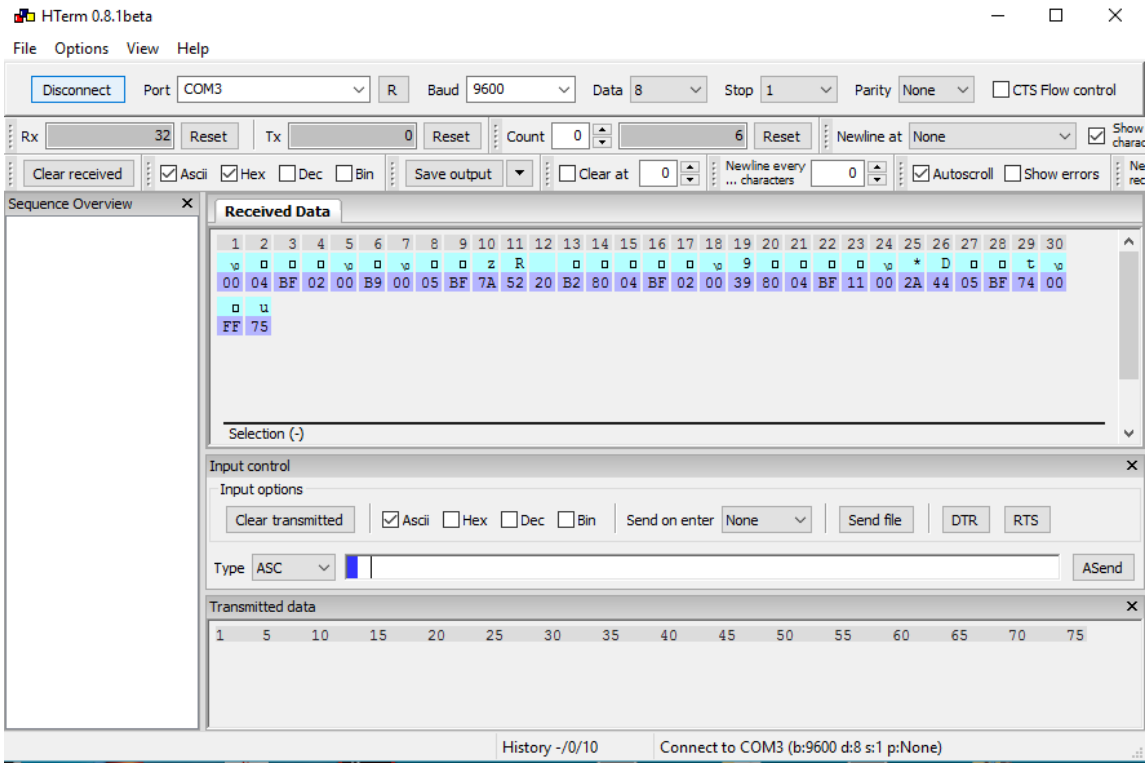



*Figura 42. Conexión y configuración de Hterm a 9600 baudios.*

Configurando los datos, bit de stop y paridad con la información del protocolo que se supone utiliza el K-Bus y se ha explicado en puntos anteriores del proyecto, conectaremos el interfaz y lo monitorizaremos con el Hterm.

Efectivamente, recibimos extraños caracteres que son ASCII. Al seleccionar la pestaña de Hex, podremos verlos traducidos a hexadecimal.

Vemos que todos los caracteres que PuTTy no podía entender y aparecían como borrones eran ASCII, y fácilmente identificables en hexadecimal.

Para intentar confirmar el funcionamiento correcto del sistema, y si nuestras hipótesis eran correctas, haremos “Clear received” a la izquierda, y abriremos la puerta del copiloto. Obteniendo lo mostrado en la siguiente página:

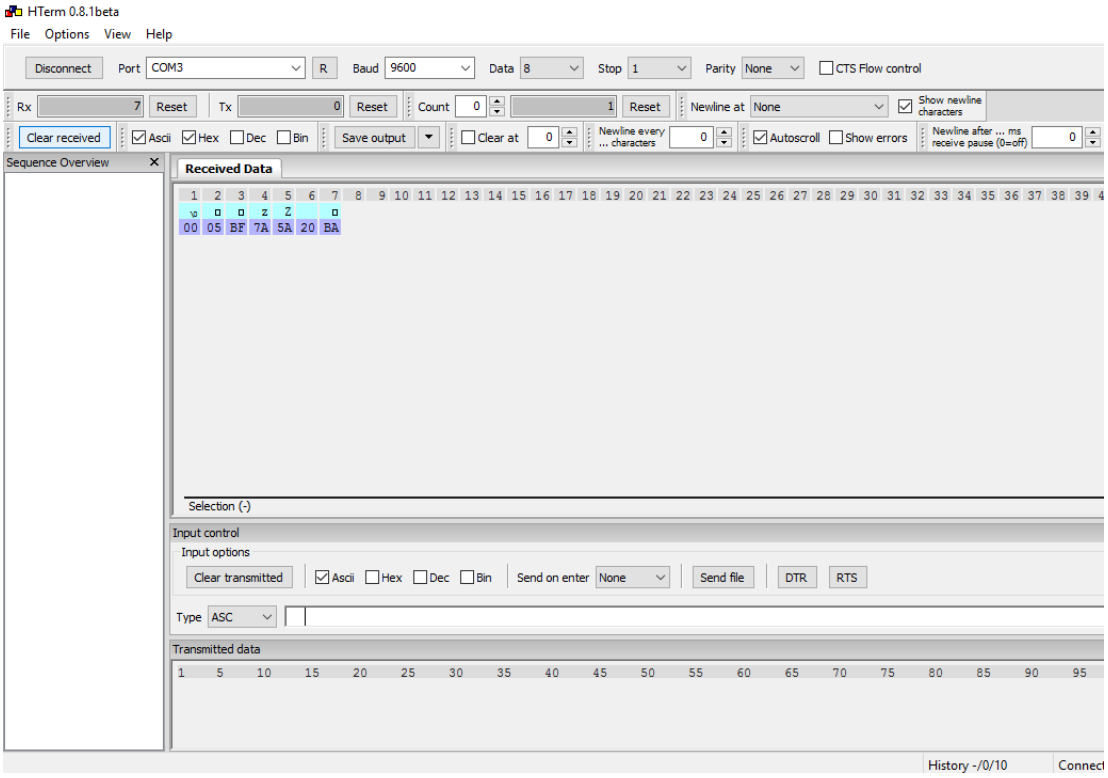



*Figura 43. Apertura puerta copiloto en Hterm a 9600 baudios.*

Al recibir solo un mensaje, podríamos concluir que efectivamente el bus está enviando el mensaje de "la puerta ha sido abierta".

El propio mensaje coincide con el formato que debiese tener un mensaje normal del bus, siendo en hexadecimal 00 el identificador de la fuente, 05 el tamaño del mensaje sin contar identificador, BF el destinatario, 7A5A20 el mensaje y BA el checksum.

Efectivamente, el mensaje está correcto.

Probando a cerrar la puerta:

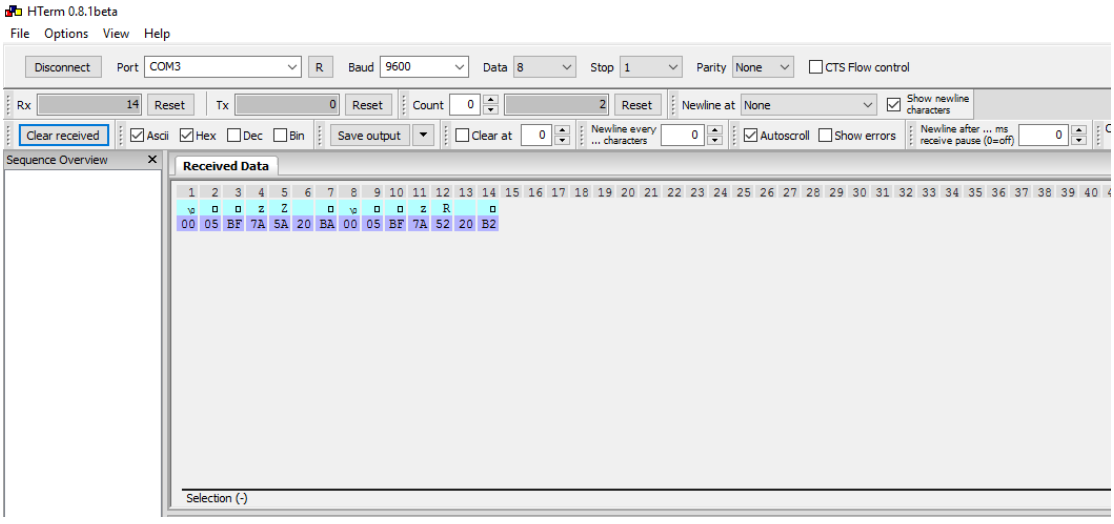



*Figura 44. Cerrar la puerta copiloto en Hterm a 9600 baudios.*

Al cerrar inmediatamente la puerta, aparece un nuevo mensaje muy parecido que nos hace pensar que lo hemos conseguido:

00 será la fuente, que coincide con la que envió el mensaje de que la puerta había sido abierta, 05 el tamaño, de nuevo BF el destinatario, mismo destinatario que el mensaje de abrir la puerta. El mensaje es muy similar: 7A5220, pero diferente en el 5A. El checksum también es correcto.

Podemos concluir que efectivamente al conectarnos al K-Bus a través del cargador de CDs, estamos recibiendo los mensajes de estado, en este caso, del cierre centralizado del coche.

Lo que quiere decir que tanto el diseño como toda la información recopilada e hipótesis son correctas.

En las pruebas con PuTTy, observamos que al encender el contacto del coche, mucha más información fluía por el bus:

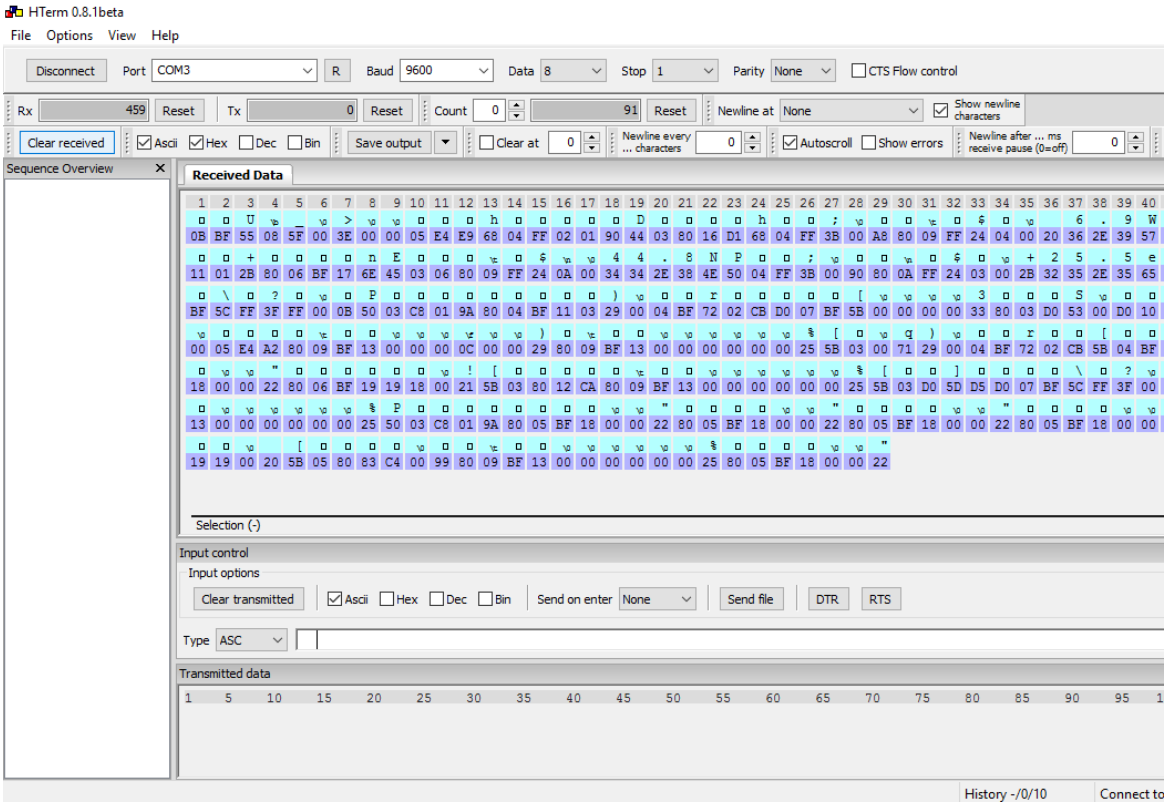



*Figura 45. Contacto de la llave en posición ON en Hterm a 9600 baudios.*

Tal como pasó con PuTTy, podemos ver mucha información fluir.

El K-Bus actuará como un receptor del estado del coche en muchas áreas diferentes, muy difíciles de identificar.

El proyecto podría convertirse por tanto en algo infinito.

Aquí se determinó que lo más lógico sería intentar identificar algunos de los mensajes que pueden ser provocados (ventanillas, puertas, intermitentes, limpia-parabrisas, etc…). En resumen, algunos de los que están contemplados en la ZKE V que se explicó.

Una vez concluido el correcto funcionamiento, nos damos cuenta que desde Hterm no podemos trabajar con la información.

No buscaremos métodos alternativos para trabajar en el entorno del PC en Windows, pues nuestro objetivo es conseguir recibir los mensajes en la Raspberry Pi y poder reflejar los mensajes de estado que fluyen de alguna forma entendible.

Así el siguiente paso será llegar aquí usando la Raspberry.

# 5- Preparación de la Raspberry pi 2

Partiendo de que se ha conseguido la instalación del sistema Operativo de Raspbian-JESSIE que se comentó en el proyecto anteriormente, trataremos ahora una de las múltiples formas de comunicación de la Raspberry Pi 2 con el ordenador sin necesidad de conexiones a internet.

La idea es conseguir un montaje portátil, que pese a usar el ordenador como monitor, se desarrolle todo en la raspberry.

## 5.1- Conexionado final.

Hay muchas formas de conseguir controlar la raspberry pi2 desde un ordenador. En este proyecto sólo trataremos una de las múltiples posibilidades.

Para llevarla a cabo bastará un cable de alimentación USB y un cable Ethernet para conectar la raspberry Pi al ordenador.

Para la primera configuración, se utilizará un lector de tarjetas micro USB para poder modificar uno de los archivos que tras grabar Raspbian aparecen en la Micro SD de la Raspberry.

Es un proceso muy sencillo. Partiendo de que ya se ha instalado el sistema Raspbian los pasos a seguir son los siguientes:

- En el panel de control, buscaremos “redes e internet” o el equivalente según la versión de Windows del pc, y en “conexiones de red” buscaremos la conexión de área local.
- En propiedades, seleccionaremos “protocolo de internet versión 4 (TCP/IPv4)” y clicaremos de nuevo en propiedades.
- Nos aseguraremos que esté seleccionada la opción de obtener la dirección IP automáticamente.

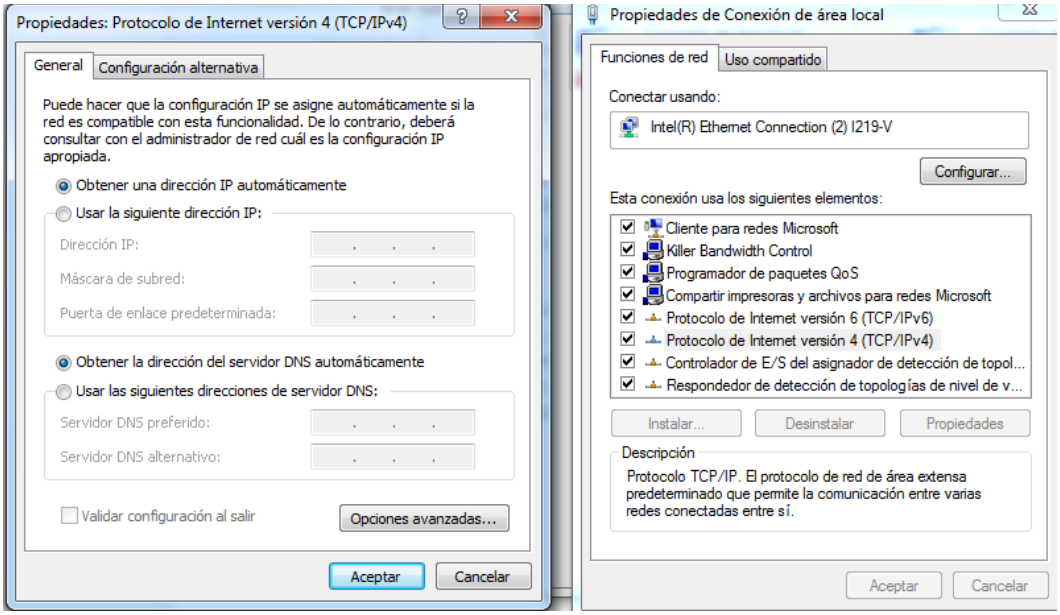



*Figura 46. Configuración conexión Ethernet en PC.*

- Conectaremos la Raspberry Pi 2 mediante el cable Ethernet y la alimentaremos. Esperaremos unos instantes a que se complete el inico.
- Ahora abriremos el CMD mediante el comando "Windows+R" y en ejecutar escribiremos cmd.exe
- Allí, ejecutaremos el comando "ipconfig" sin las comillas.
- Buscaremos la parte en la que diga Ethernet y conexión de área local. Anotaremos la Dirección IP del campo autoconfiguración de IPv4. En nuestro caso: 169.254.204.147

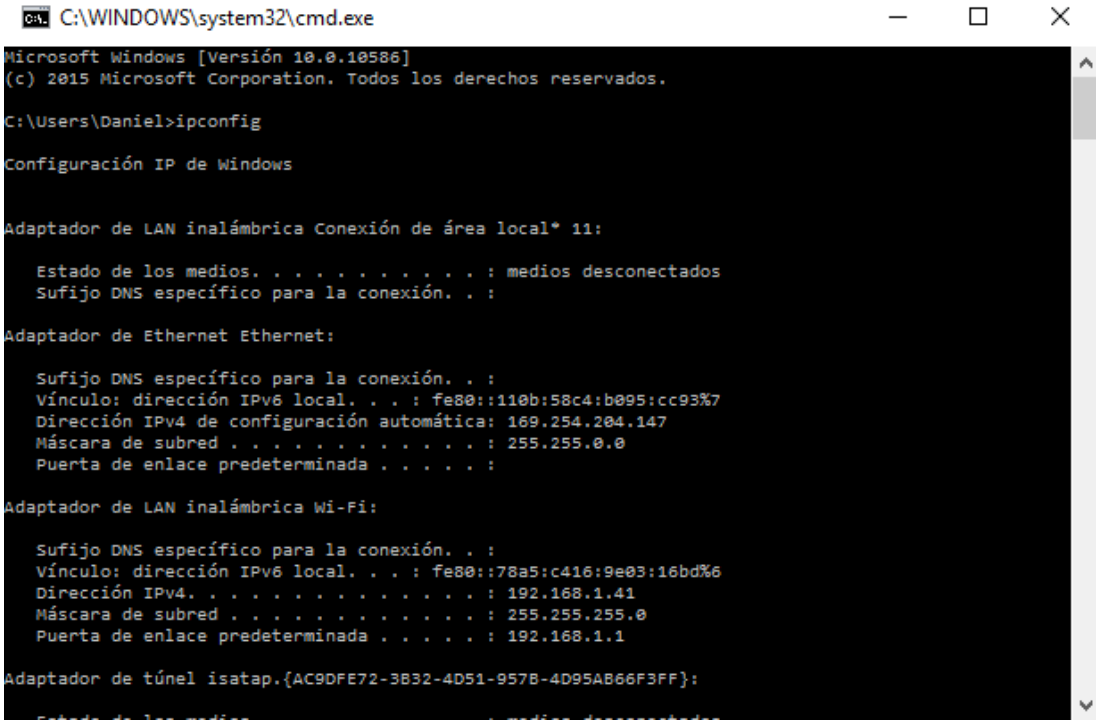



*Figura 47. Comando ipconfig y detección de la IP del ethernet.*

- Tras esto, desconectaremos la Raspberry, tomaremos la Micro SD y la conectaremos al PC.
- Encontraremos en la raíz un archivo llamado “cmdline.txt”

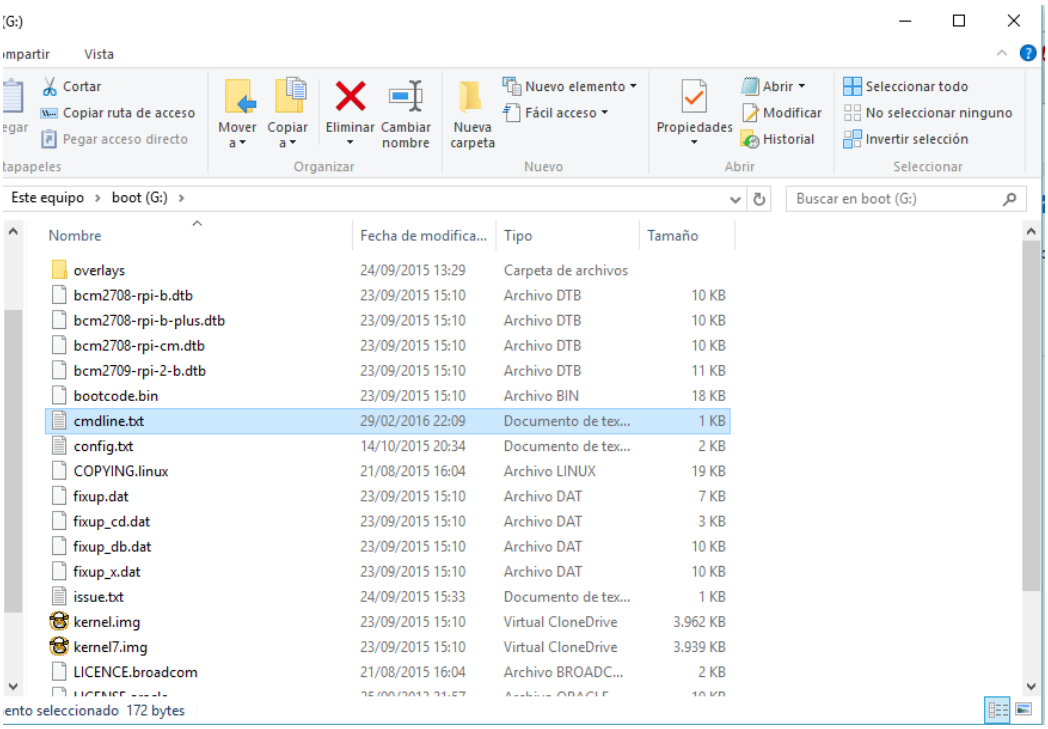



*Figura 48. Raiz de la Raspberry Pi. Localización de cmdline.*

- Abriremos el archivo e iremos al final de la línea de texto. Tras la palabra “rootwait” añadiremos un espacio y lo siguiente “ip=169.254.204.133::169.254.204.147” sin las comillas en nuestro caso. *Nota: Puede ser cualquier rango de IP siempre que esté en la misma clase.

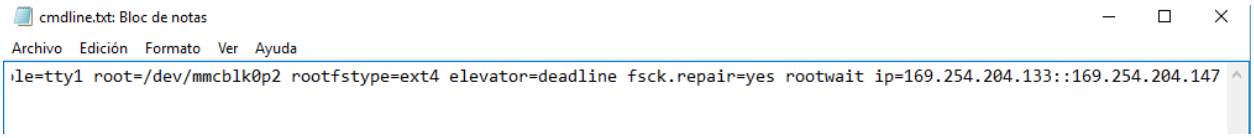



*Figura 48b. Edición archivo cmdline.*

- Cerraremos guardando y devolveremos la Micro SD a la Raspberry.
- Abriremos PuTTy y antes de anotar la dirección IP que apuntamos nos aseguraremos de que en la pestaña SSH, X11 está habilitado.

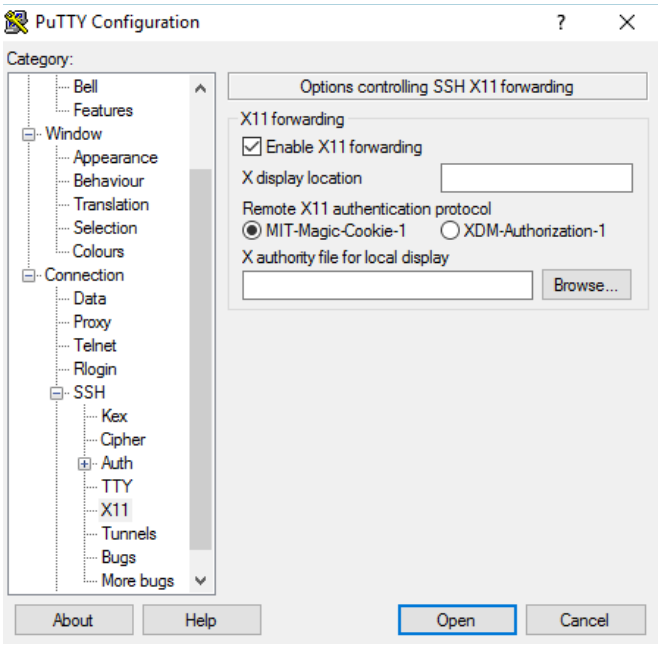



*Figura 49. Enable X11 dentro de PuTTy.*

- Finalmente escribiremos la dirección del Ethernet y guardaremos esta sesión. Así, estaremos conectados a la Raspberry. Introduciendo el usuario como "pi" sin comillas y la contraseña por defecto de "raspberry" sin comillas. Habremos accedido a nuestro microcontrolador

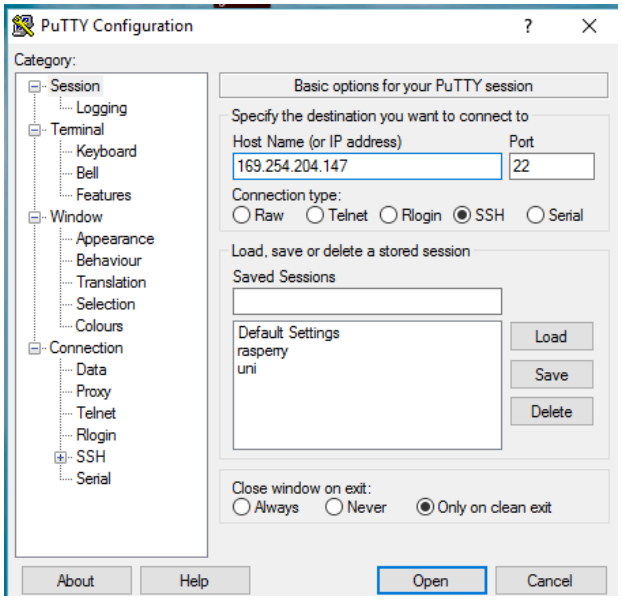



*Figura 50. Conexión Raspberry PuTTy.*

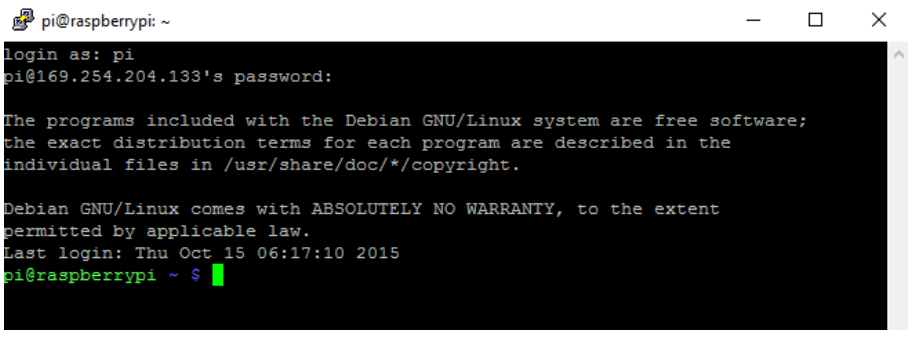



*Figura 51. Primera pantalla conexión con éxito via SSH a la Raspberry Pi.*

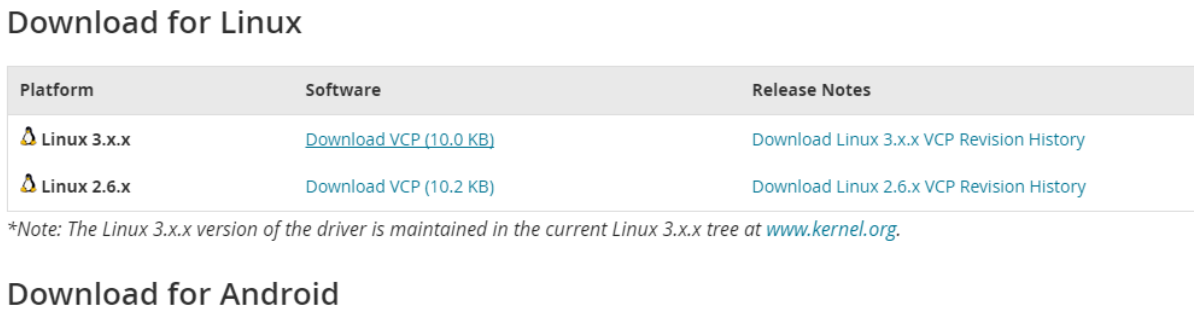
Ahora que hemos conseguido una conexión totalmente funcional entre Raspberry y ordenador, conectaremos el interfaz diseñado a la propia Raspberry y desde el interfaz al bus del coche.

Para obtener los datos, ahora necesitamos instalar los drivers necesarios en la Raspberry.

## 5.2- Drivers necesarios implementados

De forma similar a como lo hicimos en el entorno de Windows, la Raspberry Pi trabaja en el entorno Linux, y aunque generalmente los drivers al igual que nos ocurrió anteriormente suelen venir pre instalados o de descarga automática, vamos a cubrir su instalación en este apartado.

De nuevo, de la página de silabs descargaremos los drivers, en esta ocasión de Linux: http://www.silabs.com/products/mcu/Pages/USBtoUARTBridgeVCPDrivers.aspx



Download for Linux

| Platform | Software | Release Notes |
|---|---|---|
| Linux 3.x.x | Download VCP (10.0 KB) | Download Linux 3.x.x VCP Revision History |
| Linux 2.6.x | Download VCP (10.2 KB) | Download Linux 2.6.x VCP Revision History |

*Note: The Linux 3.x.x version of the driver is maintained in the current Linux 3.x.x tree at www.kernel.org.

Download for Android

*Figura 52. Descarga de los drivers de Linux.*

Dentro del archivo comprimido, encontramos un par de ejemplos de uso de los drivers y el archivo importante Makefile.c:

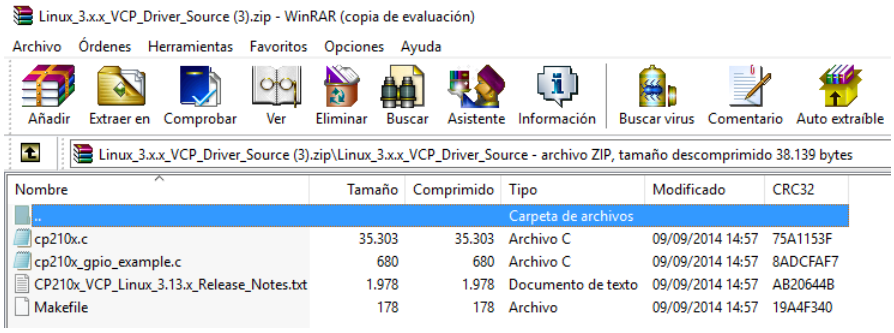



*Figura 53. Localización del Makefile.c.*

En este punto trabajaremos ya desde la propia raspberry.

Es recomendable haber configurado el entorno de video de la raspberry para copiar los archivos descargados a la raspberry mediante uno de sus puertos USB. Si no, podemos copiarlos directamente a través del propio terminal con la conexión SSH que explicamos anteriormente.

Lo primero, es asegurarnos de tener los archivos cabecera básicos. Generalmente se incluyen, pero si no es el caso, escribiremos la línea de código siguiente:

```
apt-get install build-essential linux-headers-$(uname -r)
```

Ahora tendremos que convertir el módulo de código fuente en un módulo de kernel.

Para ello ejecutaremos el comando Make al archivo Makefile.c del que disponemos, la línea de código genérica es la siguiente:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

Tras esto habremos generado un archivo de extensión .ko

Debemos ahora instalar (insertar) el modulo en el kernel, para ello nos serviremos del comando "insmod":

```
insmod Makefile.ko
```

Gracias a esto, tendremos todas las cabeceras necesarias instaladas y cuando en nuestro código de programa principal se requiera alguna de las funciones que incluyan los drivers que se describían en el archivo .c, dispondremos de ellos.

# 6- Desarrollo del programa de comunicación

Una vez tenemos todo preparado, es el momento de comenzar a recibir la información.

Al no disponer de una plataforma como PuTTy o Hterm como hacíamos en Windows, tendremos que crear un programa desde 0, dando uso a las librerías del apartado anterior y que resulte en un ejecutable que utilizaremos desde la raspberry.

El programa completo se encuentra en un adjunto con los comentarios apropiados.

Basándonos en ejemplos de internet sobre el uso de la comunicación puerto serie, incluiremos en el código todo lo siguiente:

En primer lugar, los archivos cabecera:

```
#include <errno.h>
```

Un archivo de cabecera en la biblioteca estándar del lenguaje de programación C. En ella se definen las macros que presentan un informe de error a través de códigos de error.

La macro errno se expande a un lvalue con tipo int, que contiene el último código de error generado en cualquiera de las funciones utilizando la instalación de errno.

```
#include <termios.h>
```

Un archivo de cabecera en la biblioteca estándar del lenguaje de programación C. Contiene las definiciones que utilizan las entradas o salidas del terminal.

```
#include <unistd.h>
```

Un archivo de cabecera en la biblioteca estándar del lenguaje de programación C. define Constantes simbólicas misceláneas así como declarar algunas funciones misceláneas

```
#include <stdio.h>
```

Es el archivo de cabecera que contiene las definiciones de las macros, las constantes, las declaraciones de funciones de la biblioteca estándar del lenguaje de programación C para hacer operaciones, estándar, de entrada y salida, así como la definición de tipos necesarias para dichas operaciones. Sus funciones son muy populares.

```
#include <fcntl.h>
```

Es el archivo de cabecera de la biblioteca estándar de propósito general del lenguaje de programación C. Contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras.

```
#include <pthread.h>
```

Archivo de cabecera de definición para uso de hilos.

La mayoría del código se explica en el anexo mediante comentarios. El programa desarrollado usó de base información de la página raspberry.org en cuanto a la comunicación por puerto serie. Algunas de las funciones que contempla son:

```
int set_interface_attribs (int fd, int speed, int parity)
```

En ella definiremos las características del Puerto serie.

```
cfsetospeed (&tty, speed);

cfsetispeed (&tty, speed);
```

Donde el argumento que pasamos es la dirección del puerto serie y nos permitirá establecer la velocidad deseada.

```
void set_blocking (int fd, int should_block)
```

Al ser la comunicación asíncrona, podría bloquearse el progreso de un programa mientras que la comunicación está en curso, dejando a los recursos del sistema muertos "idle".

Ya dentro del Main, tendremos funciones totalmente genéricas además de variables locales para conseguir el funcionamiento deseado, como es por ejemplo:

```
char *portname = "/dev/ttyUSB0";

fd = open (portname, O_RDWR | O_NOCTTY | O_SYNC);
```

Además de muchas otras funciones genéricas, el programa consiste en un while(1) y una serie de printf que nos permitirán visualizar lo que está pasando por el bus.

# 7- Tratamiento de datos de diagnóstico

Finalmente, tras haber desarrollado el programa solo nos queda convertirlo en un ejecutable para lanzarlo desde la terminal de la Raspberry y comenzar la extracción de datos.

Por defecto los ficheros .c no son ejecutables así que debemos usar el comando gcc –o

De éste modo e introduciendo la línea siguiente en el terminal de la Raspberry, tendremos el ejecutable funcional:

```
sudo gcc "ejecutable.c" –o "ejecutable"
```

Esto generará un archivo ejecutable de nombre "ejecutable".

## 7.1- Obtención de datos

Con el programa ya preparado en la Raspberry, solo nos queda hacer una conexión adecuada y comenzar a obtener todos los datos que se nos puedan ocurrir.

De forma similar a la conexión que se llevó a cabo durante las pruebas en Windows, el montaje final queda como sigue:

*Figura 54. Conexión final K-Bus - Raspberry.*

Una vez que estamos conectados, entrando en la terminal ejecutaremos el ejecutable para poder comenzar a extraer datos:

En un primer momento, lo único que encontraremos serán muchos mensajes que no seremos capaz de entender, al igual que nos ocurrió al conectarnos en Windows:

Pondremos a recibir datos al sistema y probaremos a encender las luces u abrir las puertas:

El resultado fue el siguiente:

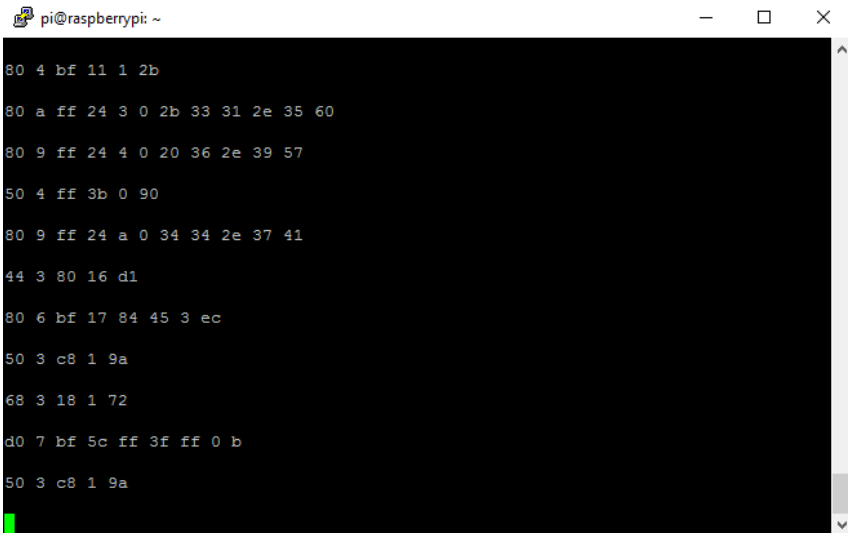



*Figura 55. Recepción de datos en la Raspberry.*

Tal como nos esperábamos nuestro ejecutable nos permite recibir en formato hexadecimal (u en cualquier otro lenguaje según lo que coloquemos en el printf, pero el más visual es éste) todos los datos que circulan por el bus.

Debemos realizar un proceso de ensayo y error para conseguir identificar alguno de los códigos que circulan por ahí.

En primer lugar, intentaremos aislar el código de abrir la puerta del conductor. Bajaremos unas cuantas líneas para ver claramente el mensaje:


*Figura 56. Código de abrir la puerta del conductor aislado.*

Parece claro que el segundo código que aparece "05bf7a5120b1" es el correspondiente a abrir la puerta, pues el primero apareció antes de abrirla.

Una vez que lo tenemos identificado, lo apuntaremos y modificaremos el código para que cada vez que aparezca ese código obtengamos el mensaje "la puerta del conductor está abierta".

**Para ello, traduciremos este mensaje hexadecimal a decimal y compararemos cada una de las posiciones del bus con el valor decimal correspondiente al valor hexadecimal que debería darse cuando aparece ese mensaje.**

A continuación, tendremos que localizar también el código de cerrar la puerta. Durante otras pruebas, resultó que el código que resulta de cerrar una puerta es diferente en cada caso. Sin embargo, común cada vez que cualquier puerta se cierra quedando todas cerradas.

Tal como lo teníamos planteado, parece que estamos ante una máquina de estados. Con una puerta abierta es un estado Q1, las dos de la parte de delante puede llamarse Q2, si abriésemos tres puertas Q3, si abriésemos las puertas en diagonal y cerrásemos la del copiloto, podría ser un estado Q4…

Algunos ejemplos, una vez localizadas algunas puertas:


*Figura 57. Código de abrir la puerta del conductor y detrás.*

Efectivamente nos damos cuenta que el código acabado en b1 o b5 correspondía tal como habíamos supuesto a algún de los estados de aperturas de puertas.

Continuando con la investigación, modificación del código y suponiendo nuevos códigos, conseguimos identificar la apertura de las puertas:

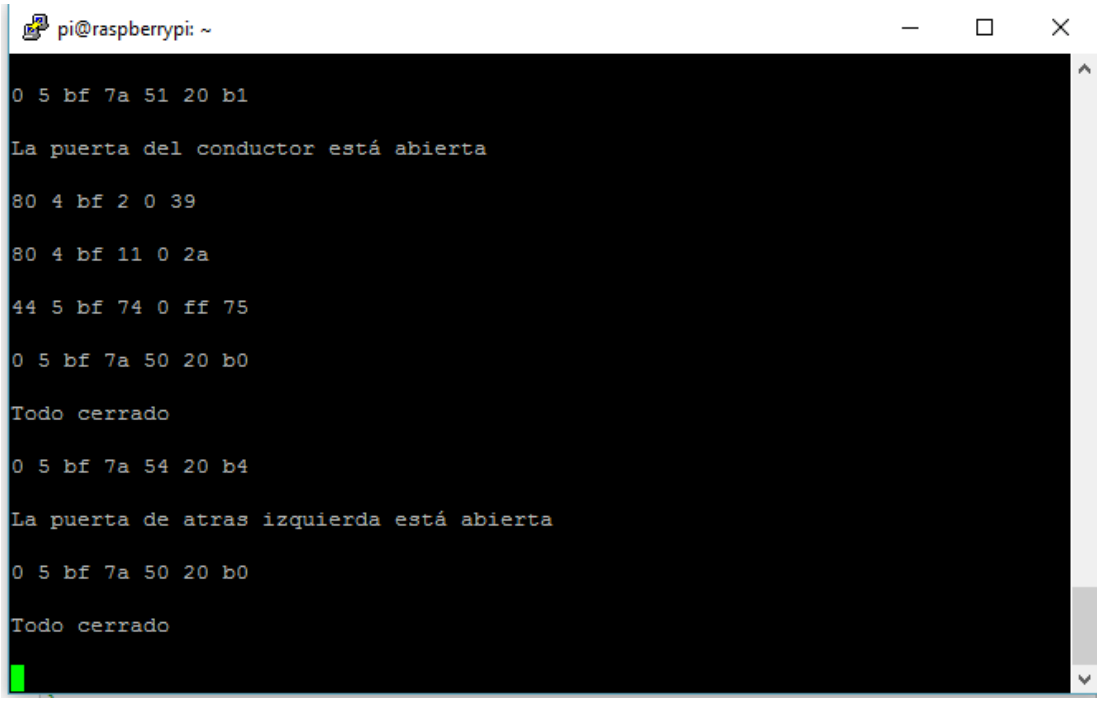



*Figura 58. Identificación de varios de los estados de cierre centralizado.*

Ahora, probaremos a mover las ventanillas para conseguir los códigos de las mismas:


*Figura 59. Código de estado de cerrar las ventanillas.*

Tal como suponíamos, el mover una ventanilla en posición cerrada o abierta daría resultado a uno de los múltiples estados que podríamos detectar.

En éste caso, la figura 59 nos demuestra que teniendo una ventanilla abierta, al cerrarla y teniendo ya identificado el código de tener todas las puertas cerradas, vemos que el código restante es “05bf7a1020f0”, que asignaremos a cerrar todas las ventanillas.

Como es de suponer, cada combinación de ventanillas abiertas o cerradas junto con la de apertura de puertas generará un estado distinto.

Conseguimos identificar así el código de abrir la ventanilla del conductor y abrir la puerta del conductor a la vez:

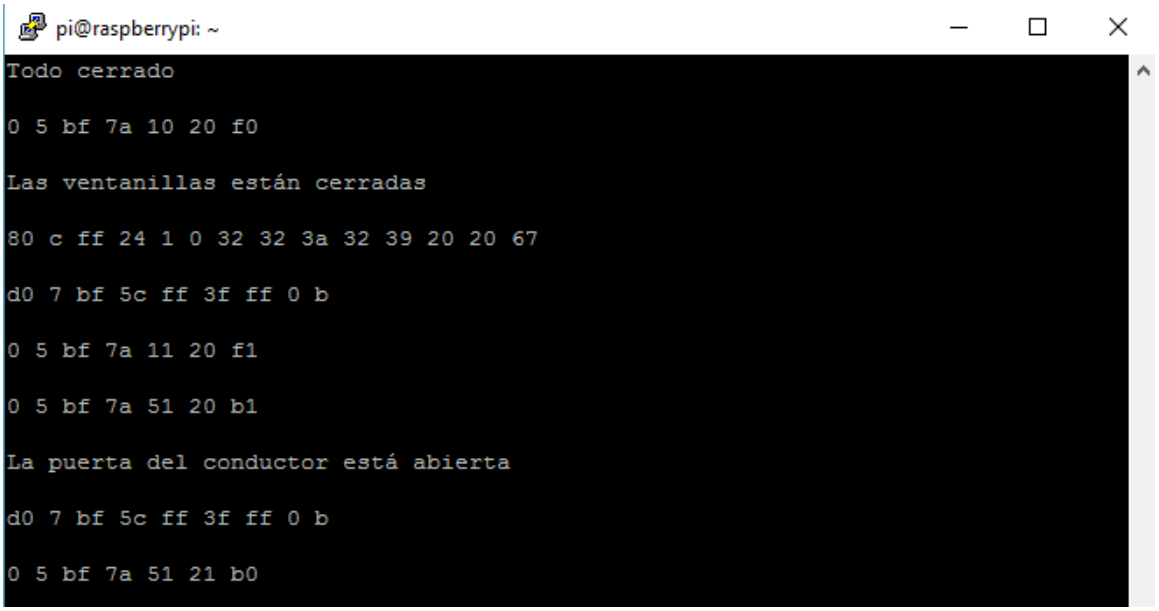



*Figura 60. Código de estado de abrir ventanilla y puerta de conductor.*

Efectivamente el código que combina ambos hechos es el “05bf7a5121b0”.

Estamos comenzando a ver un patrón en el que todos los códigos que estamos obteniendo tienen muchos dígitos iguales.

Ahora ya somos capaces de identificar la apertura y cierre de la ventanilla junto a tener la puerta abierta.

En la siguiente imagen podremos confirmar que efectivamente todo lo que estamos tratando son estados, pues cuando pasamos de tener la ventanilla abierta y la puerta abierta sólo a tener la puerta del conductor abierta, volvemos al estado que habíamos identificado como "puerta de conductor abierta".

Al igual que hemos detectado el uso de la ventanilla del conductor, también hemos podido detectar el uso de la del copiloto.

De nuevo y tras la adición de éste código comprobaremos que la asunción de códigos que hemos hecho es correcta mediante una prueba en la siguiente página.

Como podremos ver, comprobamos el funcionamiento de lo que hasta ahora hemos conseguido detectar sin ningún problema.

En la imagen, se decidió colocar la llave en la posición ON (donde toda la electrónica de confort está accionada). Así podemos ver muchos códigos que anteriormente no aparecían y que aún no sabemos relacionar.

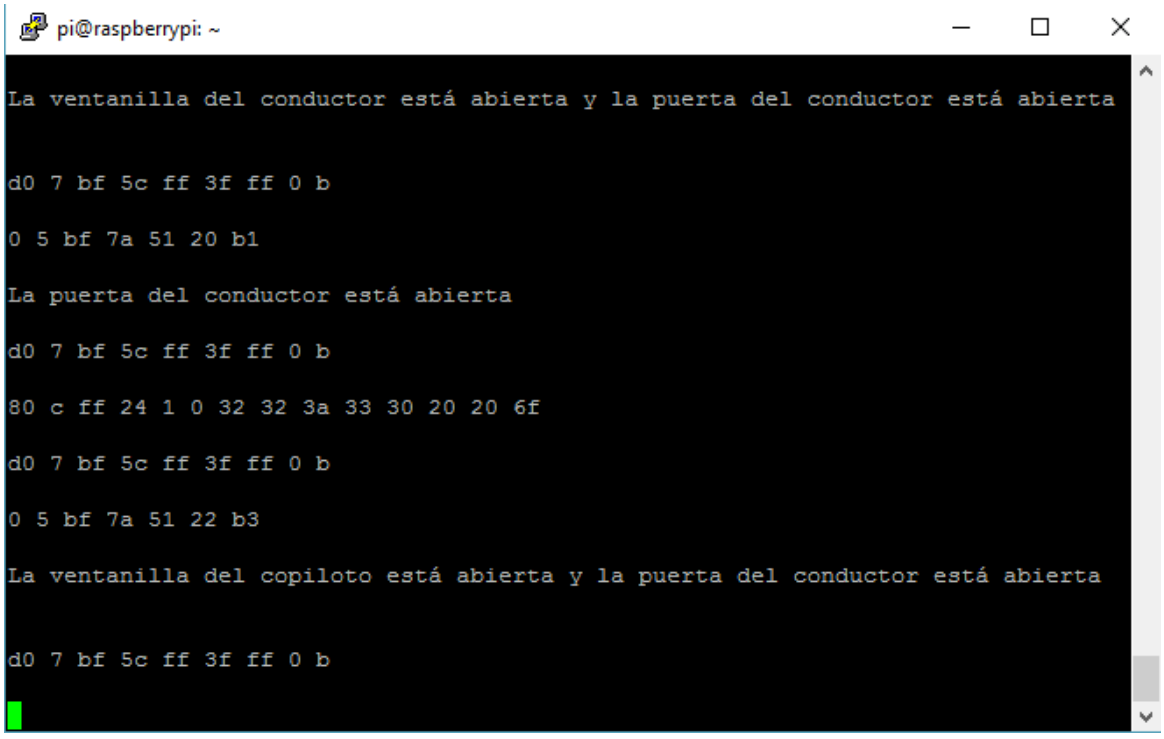



*Figura 61. Códigos de estado de ventanilla y puerta de copiloto.*

Para seguir avanzando, ahora trataremos las luces.

Según la posición de la llave (contacto, ON y en marcha) hemos visto que nos aparecen más códigos. La posición de la llave nos afecta a las luces (pues sin el contacto en ON, solo pueden encenderse las de posición.

De nuevo se tratarán de estados, por lo que debería haber un estado que aún no hayamos detectado que corresponda a tener todo apagado, y diferentes combinaciones con las posibilidades de encendido.

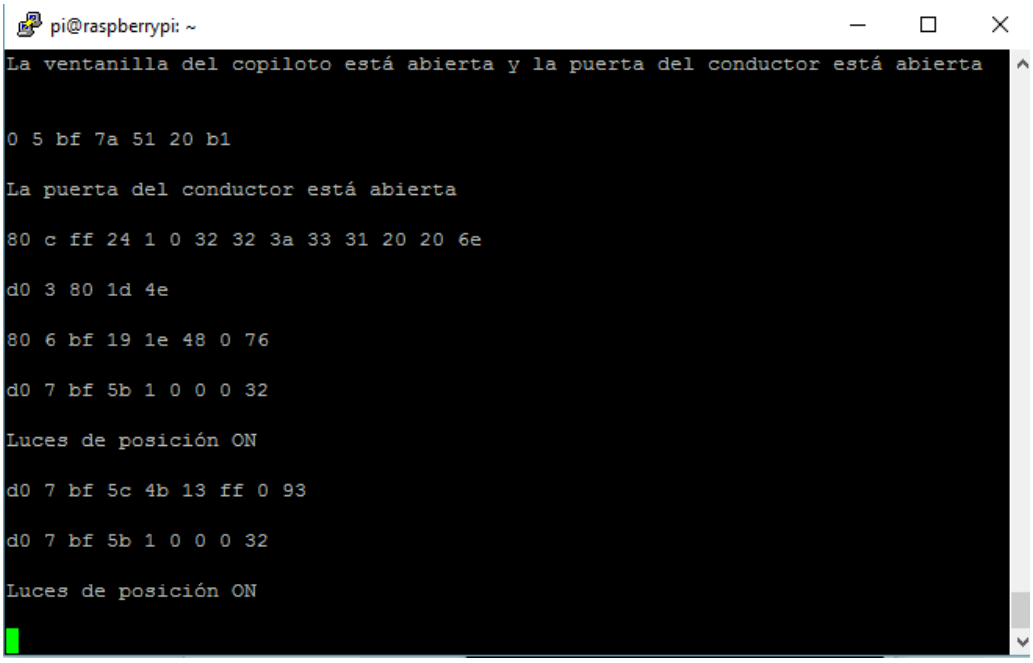



*Figura 62. Códigos de estado de luces de posición detectados.*

Durante la experimentación con las luces, se notó que los códigos de las luces y los códigos de las puertas (cierre centralizado) parecían independientes unos de otros.

Para confirmarlo, y creyendo haber conseguido detectar cual es el código de apagado de luces (que en teoría corresponderá con el estado de apagar cualquier luz, intermitente o blanca), experimentaremos encendiendo las luces cuando la puerta del conductor estaba abierta, para después abrir la puerta de atrás izquierda y a continuación apagar las luces.

Teniendo ya todos los códigos del cierre centralizado identificados y los comentados de las luces, éste fue el resultado:

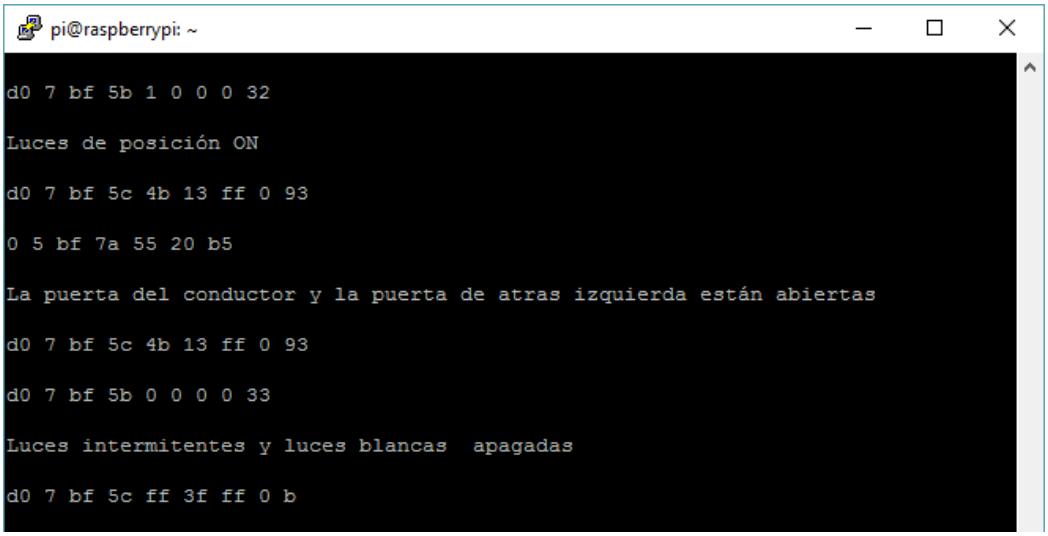



*Figura 63. Códigos de estado de luces independientes de cierre.*

Efectivamente, las luces son independientes del cierre.

Para seguir avanzando con el estudio de las luces e intermitentes, tendremos que colocar la llave en la posición de ON que nos permitirá poner las luces de cruce. Quedará en esta posición durante el estudio lumínico.

En la investigación por los intermitentes, podemos percatarnos que su funcionamiento es distinto al del resto de las luces o el cierre centralizado.

Si bien siguen siendo estados, en cada una de las intermitencias aparece el mismo mensaje.

Por tanto, los intermitentes parecen ser un código prioritario que aparece una y otra vez mientras las luces sigan encendiéndose.

Con el código depurado habrá que buscar la forma de hacer que sólo los detecte una vez al activarlos y con normalidad, acabe en el código de todas las luces apagadas (o el que corresponda).

Además, el intermitente emite un estado distinto según las luces de posición o cruce estén encendidas o no. Por tanto, las combinaciones de éste mensaje prioritario son tres.

En la imagen puede verse el ejemplo una vez identificado el intermitente izquierdo:

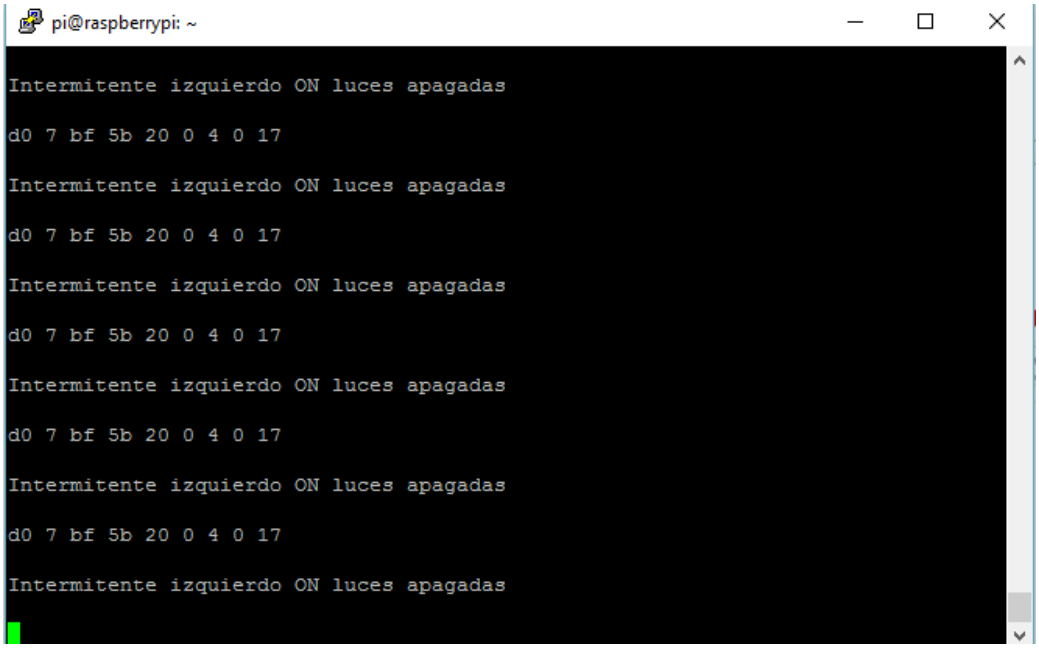



*Figura 64. Funcionamiento de los códigos de estado de intermitencia. Luces apagadas.*

Como podemos ver, es también fácil de identificar el cambio que se produce cuando encendemos las luces de posición.

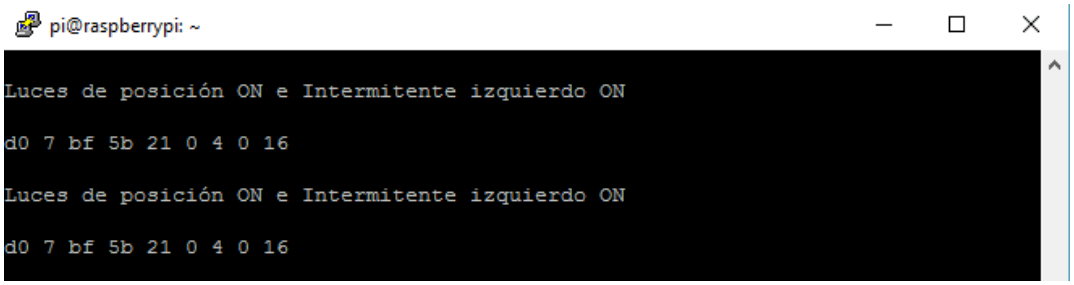



*Figura 65. Cambio del código del intermitente por efecto de las luces.*

Sin embargo, detectar los códigos para las luces de cruce se torna una tarea mucho más complicada. El contacto de la llave en ON implica muchos códigos circulando a la vez:

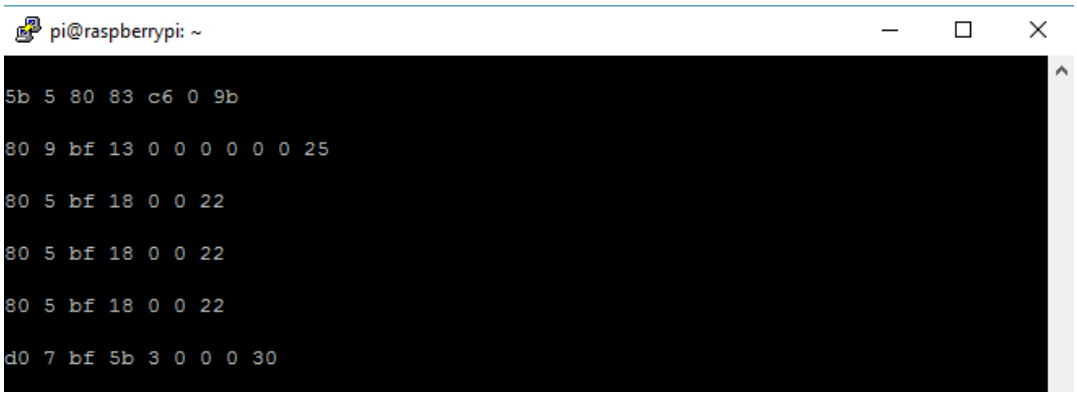



*Figura 66. Códigos que surgen del encendido de las luces de cruce.*

No somos aún capaces de saber qué código se genera dónde, ni si es producido por el encendido de luces o simplemente uno de los múltiples mensajes que circulan por el bus habitualmente.

Por similitud, al igual que nos ocurrió con el cierre centralizado, podremos asumir que el código "d07bf5b300030" es el correspondiente a las luces.

En el próximo apartado intentaremos clasificar los códigos que tan similares nos parecen. De momento continuaremos con la extracción de datos.

Para no sobre cargar el documento con imágenes muy similares, no se colocan aquí las capturas de las diferentes pruebas para conseguir todas las combinaciones. Finalmente obtuvimos todas las combinaciones de intermitencias con las diferentes posiciones lumínicas.

Ahora trataremos las luces de emergencia. Fue un código muy sencillo de extraer al ser del mismo tipo que los intermitentes. Es decir, un código prioritario sincronizado con el encendido periódico de los intermitentes.

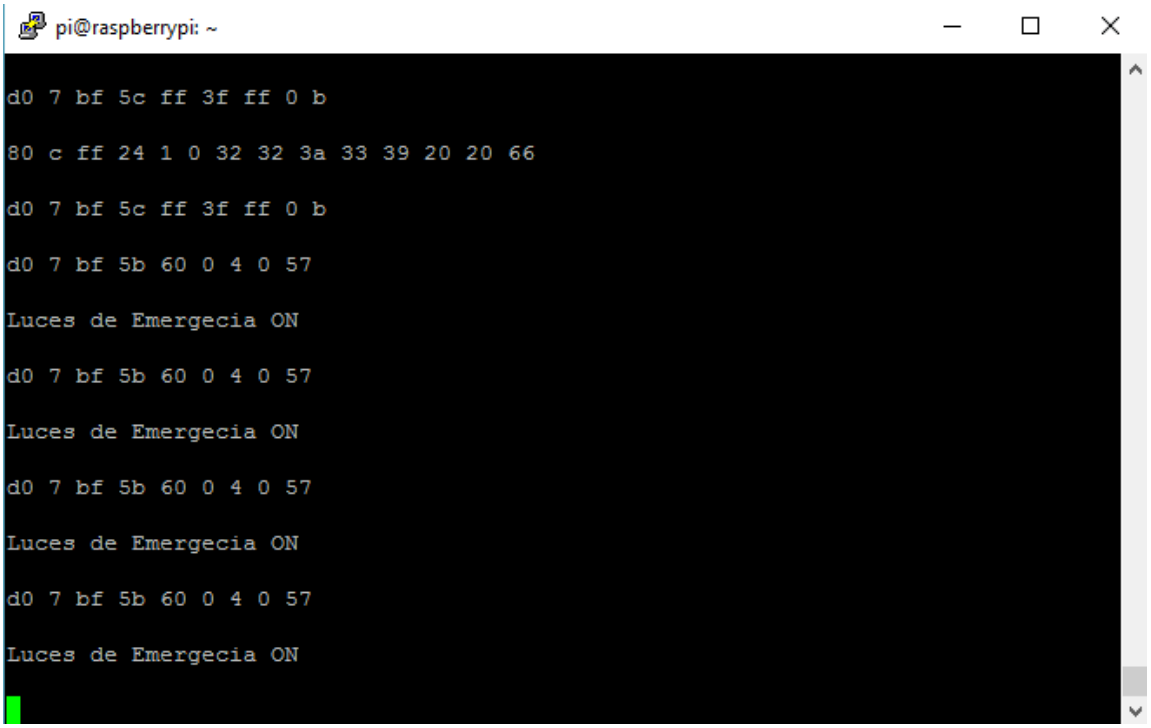



*Figura 67. Detectado código de las luces de emergencia.*

Independientemente de que un intermitente estuviese accionado, el activar las luces de emergencia tenía prioridad y hacía que el código de estado del intermitente dejase de aparecer en el bus.

A continuación, probaremos los parabrisas:

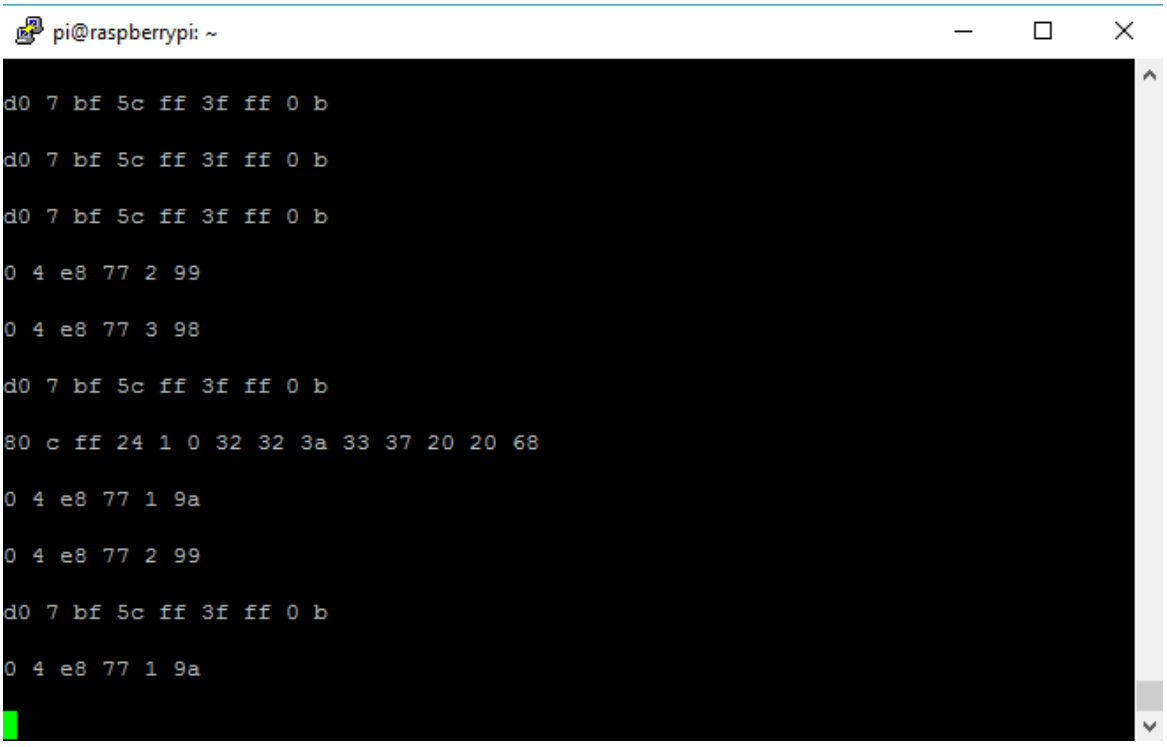



*Figura 68. Detección de códigos de barrido del parabrisas.*

Los parabrisas cuentan con varias posiciones, desde la opción de un único barrido manual a una velocidad lenta, un barrido lento en intervalos de tiempo espaciados, velocidad media y barridos rápidos constantes.

En la figura superior, están los distintos mensajes que fueron apareciendo.

Deducimos entonces que los mensajes del tipo "04e877xxx" hacen referencia a la velocidad con la que se está produciendo el barrido del limpiaparabrisas, no es como los intermitentes que genera un mensaje de estado en cada uno de los barridos.

Es decir, el código "04e877 2 99" es el correspondiente a la mayor velocidad de barrido. EL "04e877 3 98" a la velocidad intermedia y el "04e877 1 9a" al barrido lento.

El problema fue detectar qué código correspondería a la opción de un único barrido. Sin embargo, no hay un estado para esa acción y el código no varía:

Al accionar el barrido único de los limpiaparabrisas, se hace un barrido a velocidad lenta, lo que genera el código de estado de barrido lento "04e8771 9a", pero además, al interrumpirse ese estado aparece un mensaje más:

"04e87709b" aparece inmediatamente después que el barrido se ejecute a velocidad lenta, como puede verse en la imagen posterior.

Por tanto vamos a asumir que ese código es el estado de detención. Sin embargo para mayor comodidad, lo nombraremos en el futuro como "barrido único" para seguir el mismo formato que hemos dado a lo largo de los experimentos:

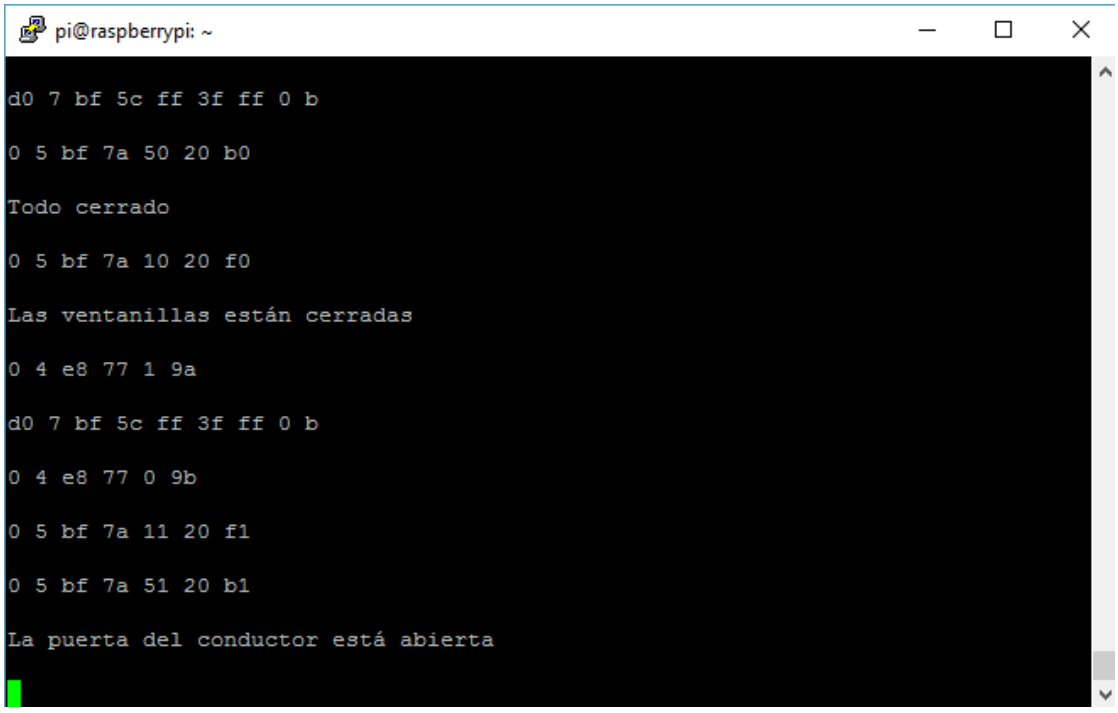



*Figura 69. Detección de código de barrido único y pruebas de dependencia.*

Además de comprobar que estábamos en lo cierto, probamos la independencia de los limpiaparabrisas con las ventanillas y puertas que ya habíamos detectado, y efectivamente lo son.

También podemos detectar los códigos de subida y bajada de volumen desde el volante. Sin embargo, al modificar el volumen a través de la propia radio, no se vuelca ningún mensaje en el bus.

Lo mismo ocurre si modificamos la temperatura del aire acondicionado.

Pese a que en un primer momento pueda parecer extraño, recordemos que hay varias entradas y salidas del Bus a lo largo del coche, y en muchas ocasiones hasta buses distintos para ciertas partes del confort del coche.

Podríamos intentar localizar alguna combinación de estados más. Al fin y al cabo en éste punto no es más que un proceso reiterativo.

Sin embargo, consideramos que los mensajes detectados hasta el momento son suficientes para probar la factibilidad del proyecto así como su correcto funcionamiento. Por tanto, en éste punto acaba la extracción de datos.

A continuación hablaremos de las similitudes que nos hemos ido encontrando y de la forma final que se le dará al código de extracción y muestra de los mensajes del bus.

## 7.2- Clasificación de datos

Tal como vimos en el apartado anterior, muchos códigos tenían una estructura muy parecida y valores casi exactos cuando se trataban de acciones similares.

Los ejemplos más inmediatos los tenemos en los códigos obtenidos mediante la apertura de puertas y ventanillas. Es decir, en el cierre centralizado.

Por ejemplo, el código que vamos a tomar como “todo cerrado” en decimal es el -05 191 122 80 32 176- mientras que el de abrir la puerta de atrás izquierda es -0 5 191 122 84 32 180-

Tal como habíamos explicado en el apartado 3.3, la figura 13 mostraba el formato de todos los mensajes del bus:

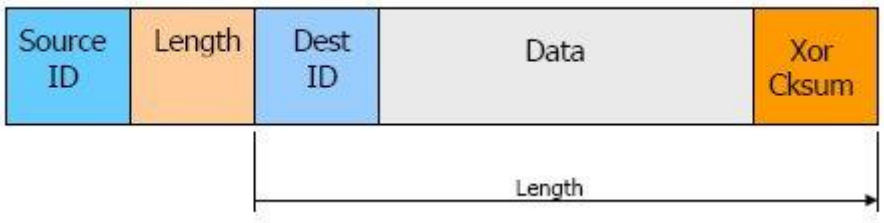



*Figura 13. Trama de datos de K/l Bus.*

Por tanto y como parece lógico, varios campos son comunes:

La fuente es "0", la longitud es correcta con "5", el destinatario es el que tenga el identificador de "191", tres datos más componen el mensaje, "122 84 y 32" en el caso que pretenda comunicar que la puerta de atrás izquierda está abierta o "122 80 32" en el caso que comunique que todas las puertas están cerradas.

Finalmente el Checksum, correcto en ambos casos.

Continuando con el estudio de los datos que hemos ido obteniendo, vemos que los códigos de las ventanillas son muy similares. Abrir la ventanilla del conductor mientras la puerta anterior estaba abierta genera el código: -0 5 191 122 84 33 181-

El código es casi exacto al de tener la puerta de atrás izquierda abierta, solo cambia un valor dentro del mensaje que es la diferenciación de tener la ventanilla bajada o subida.

Así, podemos suponer ya que todas las puertas y movimientos de las ventanillas corresponden al cierre centralizado.

"0" probablemente se trate del cierre centralizado, y "191" del identificador del módulo ZKE V que tratamos en apartados anteriores.

Éste mismo tratamiento se puede estudiar con las luces:

-208 7 191 91 0 0 0 0 51- es el código que hemos detectado para el estado de tener todas las luces, tanto intermitentes como blancas apagadas.

-208 7 191 91 1 0 0 0 50- es el código que hemos detectado para encender las luces de posición. -208 7 191 91 64 0 4 0 119- es el código para con las luces apagadas, encender el intermitente derecho.

En este caso, pese a la diferencia de longitud, tenemos "208" como el control de las luces, vemos que es el "191" el destinatario, mismo módulo ZKE. Empezamos a asumir que a través del punto por el que conectamos con el bus, podemos leer son los que se dirigen hacia allá.

De nuevo, es correcto el campo de checksum y longitud.

En cuanto al volumen de la radio, tenemos el código de -80 4 104 50 17 31- para subir el volumen.

Los códigos de subida y bajada del volumen no son tan fácilmente diagnosticables, puesto que al no tener el bluetooth conectado y no recibir códigos del cambio de emisora en la radio desde el volante, no somos capaces de saber si el "80" se refiere al volante o la radio, ni si el "104" es la propia radio, a la que no tenemos acceso desde el punto de conexión.

Finalmente, en cuanto al limpiaparabrisas, obtuvimos los códigos -0 4 e8 77 2 99- correspondiente a la mayor velocidad de barrido o el -0 4 e8 77 3 98- para la velocidad media.

Con esto confirmaremos que el "0" es el módulo ZKE. Tal como habíamos expuesto antes, 3l módulo básico de la electrónica centralizada de la carrocería ZKE V controla las siguientes funciones entre otras:

- Limpiado y lavado del parabrisas
- Elevalunas eléctricos

- Cierre centralizado
- Alumbrado del habitáculo

Hemos sido capaces de detectar todos los mensajes de estado que provienen de las fuentes anteriormente citadas a través del módulo ZKE V.

Ahora que tenemos más o menos clara la procedencia de los mensajes, vamos a pulir un poco el código que utilizamos para mostrar los mensajes en pantalla y facilitar el procesamiento de la Raspberry, agrupando las diferentes posibilidades dentro de los orígenes comunes.

Esto se tratará en el siguiente punto.

## 7.3- Traducción de datos

Tal como hemos visto, recibíamos los códigos en hexadecimal a la vez que conseguíamos que en pantalla se detectase el estado, esto se debe a que el programa original se basaba en lo siguiente:

```
void main()

{

char *portname = "/dev/ttyUSB0";

int fd;

int n,i,Gotit;

int fd2;

int count;

int kk,l;

fd = open (portname, O_RDWR | O_NOCTTY | O_SYNC);

if (fd < 0)

{
```

```
printf ("error %d opening %s: %s", errno, portname, strerror (errno));

        return;

}

set_interface_attribs (fd, B9600, 0);  // set speed to 9600 bps, 8n1 (no
parity)

set_blocking (fd, 1);

k=0;                // set no blocking

i=0;

while(1){

      n= read (fd,&a, 1); //sizeof buf);  // read up to 100 characters if
ready to read

        buf[k]=a;

        if(i==1)kk=i+a+1;

        i++;

      k++;

      if(k==1024)k=0;

      if(i==kk)

{
```

```
        i=0;

        printf("\n\n");

        buf[k+1]='\0';

        k=0;

        for(l=0;l<kk;l++)

printf("%x    ",buf[l],buf[l]);

}

}

close(fd);

}
```

Según detectábamos algún código que pudiésemos relacionar, simplemente lo añadíamos de la siguiente forma, dando uso a la variable Gotit que hemos declarado para, en un futuro, separar los mensajes detectados los unos de los otros:

```
        if((buf[0]==0) && (buf[1]==5) && (buf[2]==191)  && (buf[3]==122) &&
(buf[4]==80) && (buf[5]==32) && (buf[6]==176))

                {printf("Todo cerrado          ");Gotit=1;}

        if((buf[0]==0) && (buf[1]==5) && (buf[2]==191)  && (buf[3]==122) &&
(buf[4]==84) && (buf[5]==32) && (buf[6]==180))

        {       printf("La puerta de atras izquierda está abierta
");Gotit=1;}
```

```
        if((buf[0]==0) && (buf[1]==5) && (buf[2]==191)  && (buf[3]==122) &&
(buf[4]==88) && (buf[5]==32) && (buf[6]==184))

        {        printf("La puerta de atrás Derecha está abierta
");Gotit=1;}

        if(Gotit==1){   printf("\n\n"); Gotit=0;}
```

De esta forma, una vez que tuvimos todos los códigos detectados, bastó con convertirlos a decimal (que podría hacerse mediante código con %d), hacer muchos printf como el que se muestra arriba y comentar el primer printf que no pertenecía a ningún if.

Después, con el análisis que se hizo de la procedencia de los mensajes, se decidió ordenar un poco el código. Muchos mensajes comienzan con los mismos 2-3 números, por lo que podíamos acortar un poco los mensajes permitiendo que no se compare dígito a dígito por todos los mensajes.

Detectamos que cuando iniciábamos el motor del coche, había algunos mensajes que aparecían duplicados. Para evitarlo añadimos las variables contact y contacto.

El código final que se utilizó se encuentra en el anexo.

# 8- Otras posibilidades de conexión

Otras conexiones dentro del K-Bus:

Tal como se explicó, hay diferentes puntos donde podríamos conectarnos al bus-K para obtener la totalidad de los mensajes o mensajes no de estado, si no de acciones.

Por ejemplo, si desmontásemos el cuadro del climatizador, encontraríamos otra toma del K-bus donde podríamos ver los mensajes correspondientes a la subida o bajada de temperatura, ventilación etc.

Lo más interesante sería conseguir acceder al K-Bus en alguno de los puntos por los que fluye dentro del coche.

Como se dijo, detrás de los embellecedores de las puertas podría localizarse el bus (entre muchos otros cables, se requeriría primero hacer un estudio para ver cuál es el correspondiente con un osciloscopio). Ésta opción requiere mucho desmontaje y cierta peligrosidad al “pinchar” el bus mediante un elemento conductor como es por ejemplo una aguja, pero a cambio nos permitiría ver todos los mensajes que circulan por el bus en esa dirección, no solo los controlados por los estados del módulo ZKE V, sino también las órdenes de subida/bajada de ventanilla, subida y bajada de volumen en lugar de “se ha subido/bajado el volumen”, etc.

Otras conexiones dentro del Can-Bus:

En pocos coches, se puede acceder al Can-Bus a través del OBD sin mayor problema (si éste fuese el caso, en el próximo apartado se tratará). En el caso de BMW, hay seguridad para impedir que se pueda acceder por ésta vía.

Sin embargo, ya hay proyectos que mediante la reprogramación de la ECU correspondiente, permiten el acceso total a todos los mensajes del bus y control de todos los instrumentos/equipos conectados al mismo.

## 9- Aplicación mediante CAN-Bus

Hay muchos proyectos que tratan de comunicarse con el coche mediante el bus Can y un Arduino, Pc o la propia Raspberry Pi.

En éste proyecto hemos desarrollado una comunicación a través del K-Bus, sin embargo, vamos a cubrir también la posibilidad de conectarse al bus del coche a través del OBD y el CAN que se explicaron en la parte teórica del proyecto.

Desarrollamos la shield necesaria para comunicar la Raspberry pi con el Can, pero al llegar a conectarnos al coche, descubrimos el problema de las Gateway que tienen muchos de los coches para impedir el acceso al Can a través del OBD.

Explicaremos el conexionado, librerías y acciones que deberíamos llevar a cabo en caso de disponer de un coche que no tenga la limitación de acceso por el OBD (Por ejemplo, los Toyota de los años 2006-2011).

En primer lugar, utilizaremos un conector OBDII típico, y cortaremos uno de sus extremos.

Recordando el Pinout de la figura 9 en el apartado 3.1, debemos localizar los pines 6, 14 y 4 para el gnd:

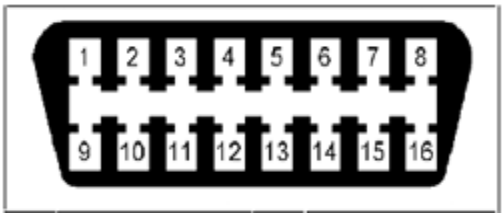



*Figura 9. Estándar OBD2*

| PIN | DESCRIPTION | PIN | DESCRIPTION |
|---|---|---|---|
| 1 | Vendor Option | 9 | Vendor Option |
| 2 | J1850 Bus + | 10 | J1850 Bus – |
| 3 | Vendor Option | 11 | Vendor Option |
| 4 | Chassis Ground | 12 | Vendor Option |
| 5 | Signal Ground | 13 | Vendor Option |
| 6 | CAN (J-2234) High | 14 | CAN (J-2234) Low |
| 7 | ISO 9141-2 K-Line | 15 | ISO 9141-2 L-Line |
| 8 | Vendor Option | 16 | Battery Power |

Para ello, recurriremos a un polímetro, poniendo un extremo en el pin y uno por uno chequeando los cables con el otro extremo hasta, dar con el correspondiente. Para ello, el polímetro debe ser colocado tal como se muestra en la imagen a continuación. Una vez detectados los pines y cables, aislaremos el resto de cables quedándonos solo con esos tres.



*Figuras 70 y 71. Selección de cables y pines necesarios para la conexión CAN Bus*

Ahora debemos comenzar con el estudio y diseño de la shield para la Raspberry que nos permita la conexión al bus CAN.

Hay muchas opciones en el mercado, desde conexiones que tienen el típico conector de 9 pines para el puerto serie hasta conexiones directas.

Sin embargo, casi todos los diseños se basan en los mismos chips siendo el más importante el MCP2515:

El Microchip Technology MCP2515-I / SO es un controlador independiente de CAN (Controller Area Network). Es capaz de transmitir y recibir tanto los datos de las tramas estándar y extendidas como de las remotas.

El MCP2515-I / SO tiene dos máscaras de aceptación y seis filtros de aceptación que se utilizan para filtrar los mensajes no deseados, reduciendo de esta manera el overclock del MCU.

El MCP2515-I / SO se comunica con los microcontroladores (MCUs) a través de un interfaz periférico serie estándar industrial (SPI).

Algunas de sus características son:

- 0-8 longitud de bytes en el campo de datos
- Tratamiento de tramas estándar y ampliadas y tramas remotas
- Dos búferes de recepción con almacenamiento de mensajes priorizada
- Seis filtros 29 bits
- Dos máscaras de bits 29
- Tres buffers de transmisión con el establecimiento de prioridades y funciones de aborto
- Interfaz SPI de alta velocidad (10 MHz)
- Modo de un impulso único que asegura que la transmisión del mensaje se intenta sólo una vez

- Pin del reloj con prescaler programable y que puede ser usado como una fuente de reloj para otros dispositivos
- Señal de inicio de trama disponible para el control de la señal SOF
- Pin de salida que actúa como una interrupción con enables seleccionables
- Tecnología CMOS de baja potencia

En esta ocasión, hay un diseño que encaja mucho con nuestra idea: El de Industrialberry.

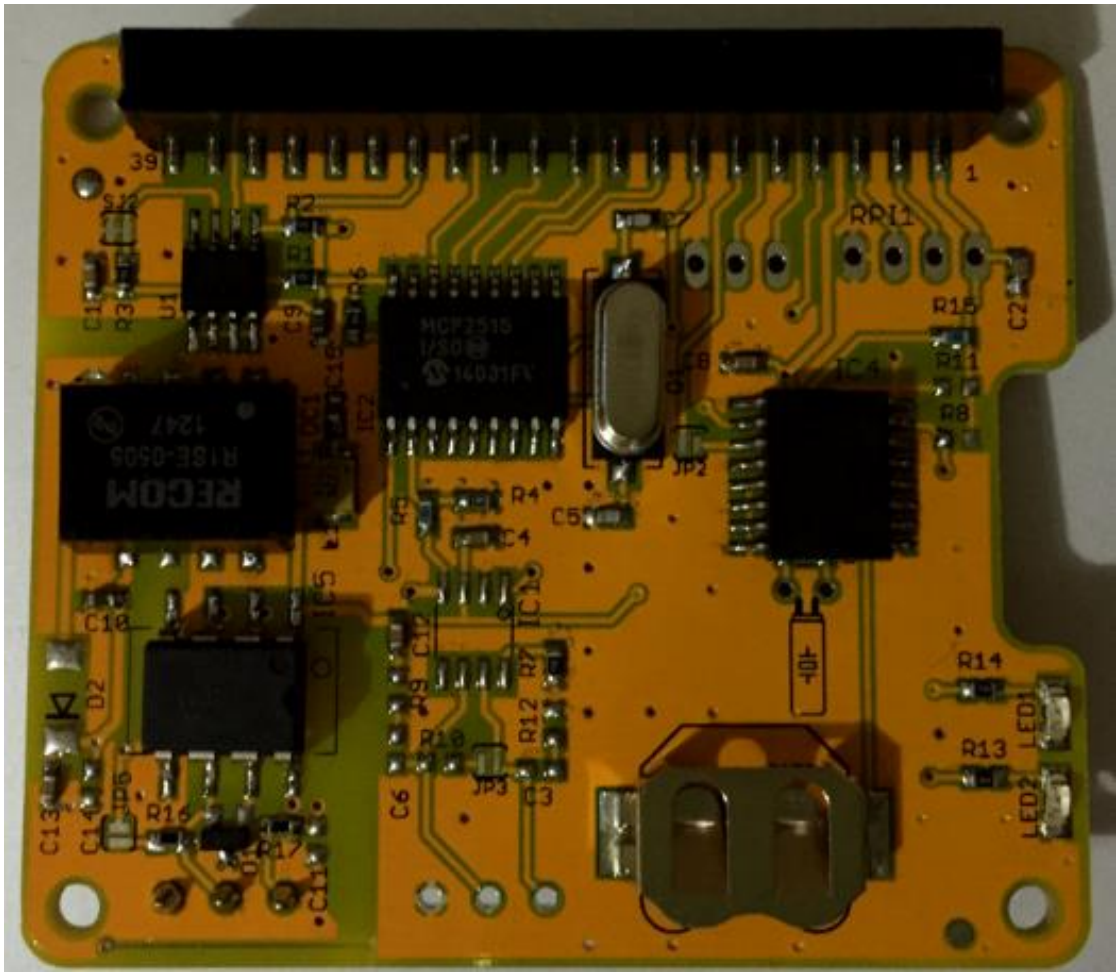



*Figura 72. Canberry V2.1 Vista superior.*

En el datasheet adjunto, podemos ver sus características más detalladamente, pero en resumen:

Canberry Pi V 2.1 es una tarjeta de ampliación (también comúnmente conocido como shield) de Raspberry Pi. Se trata de un diseño de hardware abierto. Tiene dos funcionalidades: un módulo de bus CAN y un reloj de tiempo real a bordo alimentado por una batería de 12 mm.

Todas las funcionalidades totalmente integradas en el kernel estándar de Linux, por lo que, pueden estar preparadas para correr o como mínimo, sencillas de agregar al último kernel de Linux añadiendo funcionalidades canbus.

El reloj de tiempo real se basa en el DS3231 con el controlador del oscilador I2C interno.

Totalmente compatible con Linux también. Utilizando el módulo kernel del I2C y las funciones estándar del kernel, la fecha y hora se pueden establecer u obtener mediante comandos simples.

En la parte inferior se encuentra una batería en la placa para garantizar una autonomía de datos de más de 20 años.

En el capítulo de hardware que hay en el datasheet está toda la información sobre los componentes principales y esquemas para reconstruir y modificar placa para la Raspberry Pi.

Algunos de los esquemas más importantes son los siguientes:

La conexión eléctrica del chip MCP2515 con el bloque de transmisión de Can:

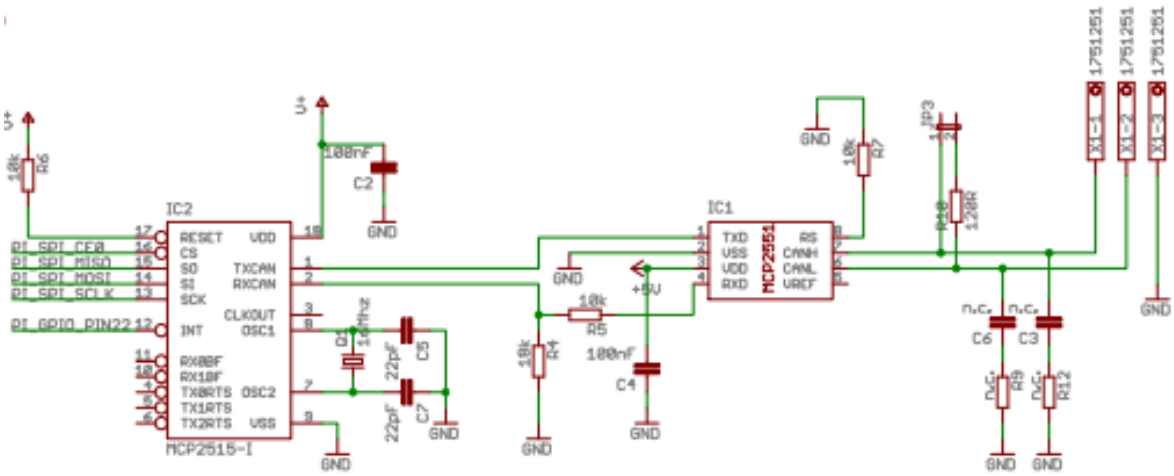



*Figura 73. Diagrama eléctrico típico del bus Can con MCP2515.*

El esquema del reloj RTC interno:

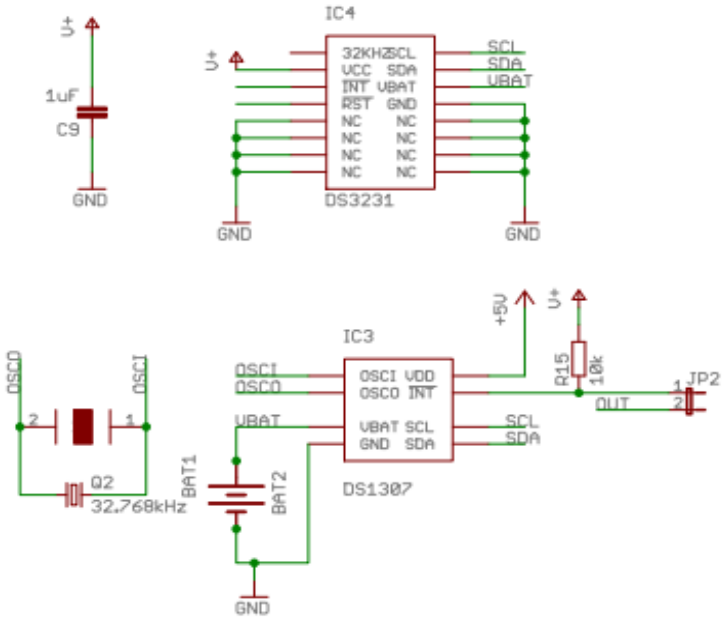



*Figura 74. Diagrama eléctrico del bloque RTC.*

Del datasheet, se clarifica que JP2 se debe usar para conectar el pin RTC Int al GPIO7 de la Raspberry.

El MCP2515 está alimentado de 3.3V a través del conector de la raspberry tal como marca el siguiente esquema:

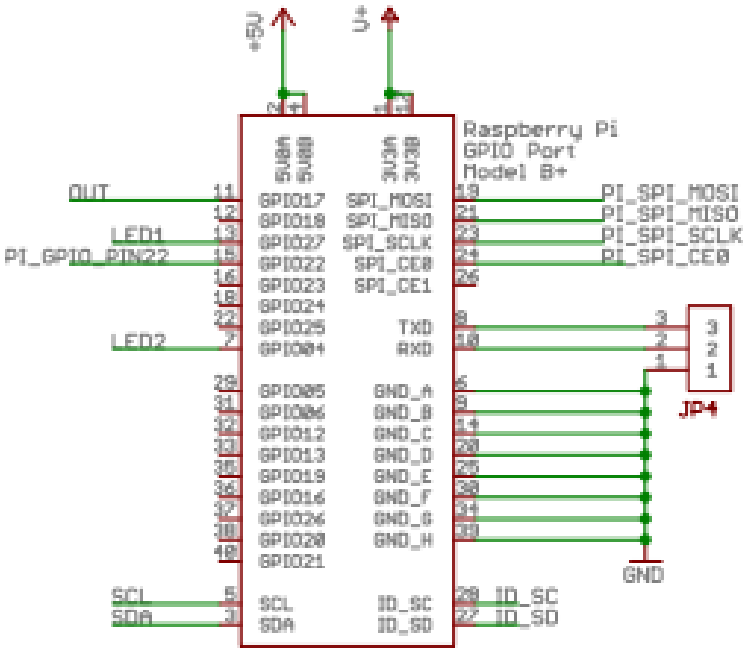



*Figura 75. Diagrama eléctrico del conector CanBerry.*

El controlador del bus se alimenta a 3V para igualar el voltaje del MCP2551. Éste transceiver usa el JP3 como la terminación de 120Ω necesarios en la comunicación CAN.

Para mayor información de hardware, consúltese el datasheet.

Ahora vamos a hablar del software, como hemos dicho, todo el shield está preparado para funcionar fácilmente en el kernel de Linux:

Vamos a seguir los pasos para la instalación de las librerías necesarias via internet en la propia raspberry, tal como se hace en uno de los enlaces de la bibliografía. Sin embargo, siempre se podrían descargar los archivos en un pc, y copiarlos en la raspberry.

Primero obtenemos los archivos necesarios:

```
cd /tmp; wget http://lnxpps.de/rpie/rpi-can-3.12.28+.tar.bz2
```

Si no teníamos conexión a internet, una vez copiados a la Raspberry, seguiremos el punto de descomprimir:

```
cd /; tar jxvf /tmp/rpi-can-3.12.28+.tar.bz2
```

Tras esto, debemos registrar los nuevos módulos a través del comando "depmod" y reiniciar la raspberry.

```
depmod -a
reboot
```

Ahora haremos los cambios de configuración en el SPI y los cambios necesarios en el módulo mcp2515x debido a que la interrupción que aparece en el código por defecto no es la que el fabricante, en este caso Industrialberry, utiliza otro GPIO. Tal como nos marca en su página web:

*"There is an important difference with V1.1, the MCP2515 interrupt is connected on GPIO22-PIN15 instead of GPIO25-PIN22. If your ISO works with boards like Canberry 1.1 or PiCan, you must redefine the interrupt pin, to use candump."*

Para ello, con un editor de texto abriremos el módulo recientemente instalado:

```
nano /etc/modules
```

El archivo debe quedar de la siguiente forma:

```
# /etc/modules: kernel modules to load at boot time.
#
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.
# Parameters can be specified after the module name.

snd-bcm2835
spi_bcm2708

# MCP2515 configuration for PICAN module
spi-config devices=\
bus=0:cs=0:modalias=mcp2515:speed=10000000:gpioirq=22:pd=20:pds32-
0=16000000:pdu32-4=0x2002:force_release

# load the module
mcp251x
```

Tras esto, configuraremos la lista negra de la raspberry para asegurarnos de que los driver del spi-bcm2708 y del mcp251x estan habilitados:

```
nano /etc/modprobe.d/raspi-blacklist.conf
```

Debe quedar como sigue:

```
# blacklist spi and i2c by default (muchos usuarios nunca los usan)

#blacklist spi-bcm2708
blacklist i2c-bcm2708
#blacklist mcp251x
```

Con esto, tendremos el equipo preparado para funcionar. Bastará con conectar los cables que habíamos preparado anteriormente a la shield de la raspberry, conectar ésta a la propia Raspberry y conectar el extremo del OBDII al coche. Recordemos que la conexión de la señal de tierra es opcional, y en este caso, no se conectará:

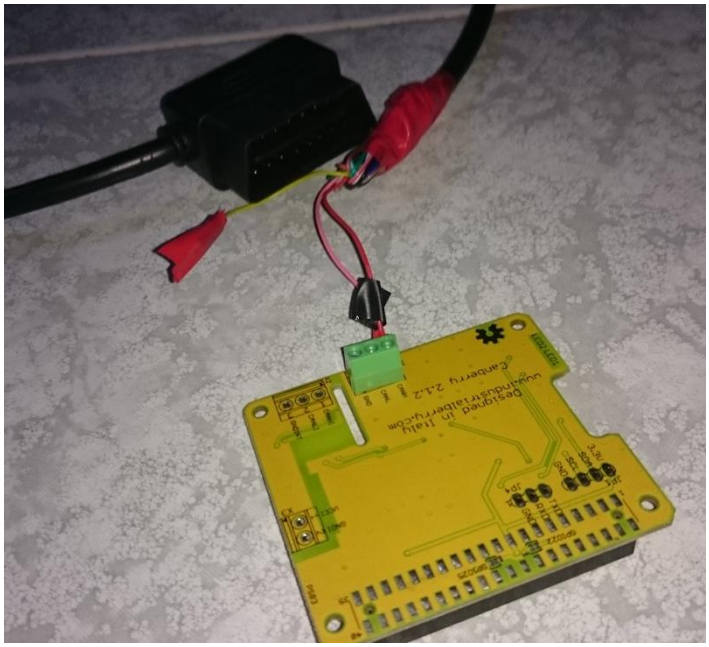


*Figura 76. Conexión extremo OBDII con shield CanBus.*

Hemos conseguido un equipo totalmente funcional, sin embargo, en los BMW no se puede extraer nada a través del OBD como ya comentamos anteriormente sin la modificación de la programación interna del coche.

Pese a ello, si tuviésemos un coche compatible, ahora bastaría con ejecutar en el terminal de la raspberry los comandos del Can Bus preparados:

Primero configuraremos la velocidad de transmisión de datos. Generalmente en la mayoría de los coches es 500000kBauds para el Can Bus. Pondremos en principio la opción "listen-only" para no tener ningún problema en caso de errores:

```
ip link set can0 type can bitrate 500000 listen-only on
```

Después habilitaremos el bus y comenzaremos la extracción con el comando "candump"

```
ip link set can0 up
sudo candump
```

En éstas condiciones, y con un coche compatible, esto es lo que se obtendría:

```
pi@raspberrypi:~$ candump
can0 120 [8] 00 00 00 00 10 10 04 4D
can0 038 [8] C0 00 08 00 00 00 00 07
can0 244 [8] 10 00 00 00 00 00 00 5E
can0 3C9 [8] 03 FF 21 02 A7 03 31 D4
can0 4C7 [8] 08 00 01 00 00 00 00 00
can0 442 [8] 00 2D 00 34 77 77 77 F3
can0 321 [8] 01 F8 01 F8 00 00 00 1C
can0 453 [8] 42 01 00 00 00 00 00 00
can0 433 [8] 20 00 30 00 00 40 00 80
can0 448 [8] 30 02 40 00 00 00 00 00
```

Como se puede observar, es el mismo concepto que lo que hemos logrado con el K-Bus, pues al fin y al cabo el K-Bus es el predecesor del bus CAN.

# 10- Posibilidades de aplicación avanzadas y conclusiones

Éste proyecto ha acabado abarcando el análisis de los buses CAN y K de un coche así como el desarrollo de los elementos necesarios para conectarse y extraer datos de estos para trabajar con ellos en un microcontrolador como es una Raspberry pi.

En el caso del K bus del BMW serie 3 de 1998 (del que dispone el coche sujeto a experimentación a lo largo del proyecto) se han descubierto los mensajes que envía el módulo ZKE V a los diferentes puntos como mensajes de estado y se han traducido e identificado para mostrarlos en la Raspberry pi en un lenguaje entendible.

Siendo esto así, este proyecto sirve como una importante fuente de información y documentación sobre las normas y funcionamiento de la transmisión puerto serie, OBD, CAN y buses K e I de los coches así como la prueba de que, pese a las limitaciones que nos encontramos al no hacer modificaciones en la estrucutura física del coche (desmontar puertas para poder pinchar bus K en otro punto) ni modificar la codificación de las ECU somos capaces de acceder de una forma no invasiva al bus de datos del coche y entender los mensajes de estado que circulan por él.

Una ampliación de este proyecto que sería una continuación inmediata, sería aprovechar el código presentado en el anexo para que con las modificaciones oportunas hiciésemos que hubiese tensión en alguno de los GPIO externos de la Raspberry a través de algún tipo de interrupción cada vez que detectamos uno de los mensajes que hemos identificado para traspasar esa tensión a una protoboard con una serie de leds colocados en disposición de los diferentes puntos del coche que simularían encendiéndose si una puerta está abierta, la ventanilla, la velocidad de los limpiaparabrisas, subidas o bajadas del volumen de la radio, etc…

Además, si el coche lo permite a través del OBD y se puede acceder al CAN, si se pincha en un punto de a través del coche el bus K o si se dispone de la codificación necesaria para la ECU podríamos llegar a recibir los mensajes de órdenes que envía el coche tales como subida y bajada de ventanillas, de temperatura, etc y comandarlos remotamente desde el ordenador o la raspberry pi enviándolos directamente al bus.

Bibliografía:

Manuales técnicos de BMW, modelos E31, E38 y E46

[1]http://www.euskalnet.net/jinfante/Multiplexado.html

[2]http://www.motorpasionfuturo.com/industria/can-bus-como-gestionar-toda-la-electronica-del-automovil

[3]http://www.ti.com/lit/an/sloa101a/sloa101a.pdf

[4]http://web.comhem.se/bengt-olof.swing/index.htm

[5]http://www.reslers.de/IBUS/index.html

[6]http://www.bmwfaq.com/threads/no-me-funciona-el-plegado-de-espejos-ni-regulacion-ni-calefaccion-de-ellos.522314/

[7]https://en.wikipedia.org/wiki/Electronic_control_unit

[8]https://www.iso.org/obp/ui/#iso:std:iso:9141:-2:ed-1:v1:en

[9]http://www.yeint.ru/suppliers/melexis/pdf/TH3122_004.pdf

[10]http://web.archive.org/web/20040920173855/http://neiland.com/images/Kbus_proj/kbus2.bmp

[11]http://pdf1.alldatasheet.com/datasheet-pdf/view/202272/FTDI/FT232BM.html

[12]http://perso.wanadoo.es/pictob/comserie.htm#la_comunicacion_serie

[13]https://www.bmwgm5.com/E46_K-Bus.htm

[14]http://www.silabs.com/products/mcu/Pages/USBtoUARTBridgeVCPDrivers.aspx

[15]https://es.wikipedia.org/wiki/Errno.h

[16]https://es.wikipedia.org/wiki/Stdio.h

[17]https://es.wikipedia.org/wiki/Stdlib.h

[18]https://www.raspberrypi.org/forums/viewtopic.php?f=33&t=153385&p=1028087

[19]http://es.farnell.com/microchip/mcp2515-i-so/can-controller-spi-1mbps-soic18/dp/1292239

[20]http://www.industrialberry.com/canberry-v-2-1-isolated/

[21]http://www.industrialberry.com/wp-content/uploads/2015/05/CanBerry-Pi-V2.1.pdf

[22]http://www.cowfishstudios.com/blog/canned-pi-part1