UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN, MENCIÓN EN TELEMÁTICA

Diseño y testeo de un sniffer-datalogger para CAN usando Arduino

Autor:
D. Alberto Domínguez García
Tutor:
D. Juan Carlos Aguado Manzano
Valladolid, 11 de Septiembre de 2017

TÍTULO: **Diseño y testeo de un sniffer-datalogger para CAN usando Arduino**
AUTOR: **D. Alberto Domínguez García**
TUTOR: **D. Juan Carlos Aguado Manzano**
DEPARTAMENTO: **Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE: Ignacio de Miguel Jiménez
VOCAL: Juan Carlos Aguado Manzano
SECRETARIO: Ramón José Durán Barroso
SUPLENTE: Noemí Merayo Álvarez
SUPLENTE: Rubén Mateo Lorenzo Toledo

FECHA: **11 DE SEPTIEMBRE DE 2017**
CALIFICACIÓN:

Resumen de TFG

El trabajo consiste en el diseño de una placa electrónica basada en el microcontrolador Arduino Nano. Debe de ser capaz de hacer la captura de tramas que viajan por un bus CAN para su posterior almacenamiento en una tarjeta SD de manera transparente a los nodos que conforman el bus.

La metodología desarrollada en el trabajo ha sido dividida en tres fases: estudio, desarrollo y testeo. En la primera fase se ha analizado las características del medio CAN de un vehículo, tales como las velocidades, frecuencias, ráfagas con las que suceden los mensajes CAN. En la segunda fase, la más amplia, se han desarrollado dos prototipos para el datalogger basados en uno y en dos Arduinos. Se han diseñado tanto con hardware como con software teniéndose en cuenta las características de la primera fase. Por último, en la tercera fase se han puesto a prueba los dos prototipos analizando sus pros y contras.

Los mejores resultados se alcanzan con el primer prototipo basado en un Arduinos Nano. Aún con esto, no se alcanzan las prestaciones totales que se pretendían, teniéndose que utilizar otro modelo de Arduino o un microcontrolar diferente para alcanzar los objetivos.

Palabras clave

Arduino, datalogger, CAN, placa electrónica, SD.

Abstract

This project is based on the design of an electronic board under Arduino microcontroller. It must be able of capturing frames traveling on a CAN bus for later storage them on an SD card transparently to the nodes that make up bus.

The methodology developed in this project has been splitted into three phases: study, development and testing. In the first phase it has been analyzed the characteristics of a vehicle's CAN means, such as, speeds, frequencies, and bursts of messages… In the second phase, it has been developed two prototypes for the datalogger based on one and two Arduino. It has been designed both hardware and software considering characteristics of the first phase. Finally, in the third phase two prototypes have been tested by analyzing their advantages and disadvantages.

Best results are achieved with first prototype based on an Arduino Nano. Even with this, the prescriptions that are intended have not been reached, it is necessary to use anhoter model of Arduino to achieve the objectives.

Agradecimientos

“En primer lugar deseo agradecer a Juan Carlos, mi tutor, su colaboración y apoyo en este período de aprendizaje a nivel científico y personal.

También deseo agradecer muy especialmente a mi hermana y a mis padres por toda la atención y apoyo mostrado durante estos años para poder concluir exitosamente el grado.”

ÍNDICE DE CONTENIDOS

Índice de figuras

Índice de tablas

CAPÍTULO I: INTRODUCCIÓN

1.1. INTRODUCCIÓN

Con la elaboración de este trabajo de fin de grado (TFG) se pretende diseñar una placa electrónica basada en Arduino que sea capaz de hacer tanto de sniffer como de datalogger del bus CAN (Controler Area Network) del vehículo Renault modelo Captur, conectándose a la interfaz OBD (On – Board Diagnosis). Por lo tanto, este dispositivo realiza de manera totalmente transparente al bus una captura de los datos que viajan en éste, para posteriormente realizar una copia de los datos en una tarjeta SD con el fin de poder analizarlos en cualquier momento y lo pone a disposición de cualquier otro dispositivo a través de otra interfaz OBD, siendo un elemento totalmente transparente tanto para el automóvil como para el dispositivo que se conecte a dicha interfaz.

Existen muchos dispositivos sniffer del bus CAN compatibles con la mayoría de marcas automovilísticas, siendo por excelencia el dispositivo *CANcaseXL*[1] de la compañía Vector uno de los más usados. Sin embargo, estos dispositivos son muy caros, especialmente para la aplicación principal que deseamos darle a nuestro dispositivo, a saber, capturar una sesión completa de diagnosis a través de la interfaz OBD que se suele realizar en todas las fábricas cuando un automóvil presenta algún problema en su fabricación. Estas sesiones de diagnosis presentan al usuario solo resultados finales y en ciertas ocasiones ocultan una parte importante de la sesión que puede ser relevante para comprender el origen del problema. De aquí surge el interés de desarrollar nuestro dispositivo que además de funcionar como sniffer como los demás también logra guardar los datos para su futuro análisis en cualquier momento o simplemente para tenerlo en modo de historial.

En el presente TFG se usa como cerebro del sistema un Arduino Nano[2], siendo éste una placa de desarrollo de hardware compuesta por un microcontrolador, memoria SRAM, capacidad de almacenamiento flash y elementos pasivos y activos. Su programación puede hacerse a través de un entorno de desarrollo (IDE) propio, el cuál compila el código desarrollado a la placa Arduino.

Este modelo Nano de Arduino se encuentra entre los que menos prestaciones poseen entre los diferentes modelos que la empresa facilita para su compra. Debido a estas bajas prestaciones y las especificaciones especialmente exigentes a las que se enfrenta el proyecto que se pretende desarrollar se presentan muchos problemas para cumplir totalmente con los objetivos. A pesar de ello se presenta un desarrollo optimizado al máximo tanto en el hardware como en el software del prototipo para estudiar si es posible desarrollar un proyecto de bajo coste o por el contrario será necesario utilizar materiales de mejores prestaciones.

1.2. OBJETIVOS Y ETAPAS

El objetivo general de este TFG es el diseño del hardware y del software de un sniffer-datalogger para llevar a cabo una captura y almacenamiento en una SD de los datos que viajan en el bus CAN del modelo Captur de Renault. Para alcanzar este objetivo general se han seguido una serie de etapas con objetivos específicos en cada una de ellas, que se Los objetivos específicos para alcanzar el objetivo general se han desarrollado en las siguientes etapas:

- Entender y analizar Arduino Nano. Específicamente sus prestaciones, funcionamiento y tipo de programación que utiliza.
- Estudiar y entender los conceptos del protocolo SPI, I2C y CAN.
- Analizar la maqueta del automóvil Captur de Renault presente en uno de los laboratorios de la Escuela Técnica Superior de Ingenieros de Telecomunicaciones.
- Diseñar el hardware del prototipo con un Arduino Nano.
- Diseño el software del prototipo.
- Evaluar los resultados obtenidos del prototipo.
- Estudiar las posibles soluciones para solventar los retardos temporales del primer prototipo.
- Diseñar el hardware del segundo prototipo con dos Arduinos Nano trabajando en paralelo.
- Diseñar el software del segundo prototipo.
- Evaluar los resultados obtenidos de este segundo prototipo.
- Extraer conclusiones del trabajo desarrollado y posibles mejoras.

1.3. MATERIALES DISPONIBLES

Para el desarrollo del trabajo final de grado se ha dispuesto de los siguientes materiales:

- Ordenador portátil DELL PP35L.
- Arduino Nano con especificaciones: procesador ATmega328, memoria flash 32 KB, SRAM 2KB, EEPROM 1KB y velocidad de reloj 16 MHz.
- MCP2515: controlador autónomo CAN capaz de transmitir y recibir tramas de datos a través de una interfaz SPI que posee a una velocidad máxima de 1 Mbps.

- MCP2551: transceptor CAN que sirve de interfaz entre el controlador CAN y el bus. Adapta la señal al medio físico.
- Módulo SD: permite introducir una tarjeta SD y realizar una comunicación vía SPI entre el Arduino y la tarjeta SD con la finalidad de escribir y leer en ficheros almacenados en la SD.
- Módulo RTC: reloj del sistema que se comunica con Arduino por bus SPI.
- Conversor de voltaje o regulador LM317t: permite convertir un voltaje de entrada de hasta 40V a una salida de 1,2 a 37V.
- Osciladores de cristal de 20MHz: necesario para el funcionamiento del controlador CAN.
- Resistencias de diferentes valores.
- Condensadores de diferentes capacidades.
- LEDs.
- Diodos 1N4001.
- Multiplexores ADG774BRZ.
- Conectores DB9
- Cables.

1.4. ESTRUCTURA DEL DOCUMENTO

La memoria de este TFG ha sido estructurada en capítulos de la siguiente manera.

En este primer capítulo del TFG se ha presentado el proyecto que se va a abordar detallando las motivaciones que han dado lugar a su propuesta y los objetivos que se persiguen.

En el *Capítulo II* se describen los conocimientos teóricos y prácticos que se deben conocer para llevar a cabo el desarrollo del prototipo. En primer lugar se habla sobre Arduino y más concretamente sobre el modelo Nano. A continuación se explican los conceptos de los protocolos SPI, I2C y CAN.

En el *Capítulo III* se estudia la arquitectura eléctrica y electrónica del modelo Captur de Renault a partir de una maqueta real. A continuación se realizan simulaciones en ella para llegar a concluir con las prestaciones que el prototipo a diseñar debe de tener.

En el *Capítulo IV* se realiza el primer prototipo del sniffer-datalogger. Se describe el hardware y el software detalladamente. Además se presentan los resultados de los test que llevaron al desarrollo de un segundo prototipo.

En el *Capítulo V* se realiza el segundo prototipo del sniffer-datalogger. Se describe el hardware y el software detalladamente. También se presentan los resultados de los tests de este segundo prototipo.

En el Capítulo VI se presentan las conclusiones y líneas futuras.

CAPÍTULO II: DIPOSITIVOS HARDWARE Y PROTOCOLOS DEL PROYECTO

2.1. INTRODUCCIÓN

En el curso 2015-2016 se logró una colaboración con la empresa automovilística Renault con el objetivo de hacer un montaje y testeo de una mesa de simulación de la arquitectura eléctrica del modelo Renault Captur. Dicha empresa donó la parte eléctrica de vehículo. Para tal objetivo se desarrolló un proyecto de fin de grado siendo el tutor responsable Juan Carlos Aguado Manzano y el estudiante que realizó el trabajo Sergio Sánchez Romero [3].

Las fases de aquel proyecto fueron los siguientes: la búsqueda de información sobre el modelo Captur, así como de los elementos que componen su arquitectura eléctrica. Con esta información se continuo el trabajo a la maqueta, previamente montada en un laboratorio presente en la ETSIT (Escuela Técnica Superior de Ingenieros de Telecomunicación), pudiendo identificar los diferentes elementos, así como su funcionamiento. Por último, se diseñó un datalogger basado en Arduino con el fin de adquirir los datos que circulan por el bus CAN en tiempo real para su posterior análisis.

El diseño de este datalogger fue el último paso de este trabajo, comprobándose que era capaz de almacenar mensajes de uno de los dos buses CAN que tiene el modelo Captur. Además, se comprobó que el prototipo funcionaba grabando en una tarjeta SD a razón de un mensaje por segundo, quedando muy lejos de las prestaciones reales que debía de tener el datalogger. Indicar que entre los dos buses CAN del modelo Captur se generan una media de 600 mensajes en un segundo.

Debido a estos problemas del prototipo que se deseaba lograr se abrió un nuevo trabajo de fin de grado para el curso 2016-2017. La intención de este nuevo trabajo sería el diseño, testeo y fabricación del datalogger basado en Arduino.

El trabajo que aquí se presenta para completar el trabajo anterior abarca una cantidad importante de dispositivos que se comunican entre sí a través de diferentes protocolos. Un conocimiento exhaustivo de todas las partes es esencial para comprender el funcionamiento final de sistema y las dificultades que entraña. Por ello, en este capítulo se introducen y describen los principales dispositivos hardware que se van a utilizar en este trabajo, así como los protocolos a través de los cuales se van a comunicar.

2.2. ARDUINO

¿Qué es Arduino?

Arduino es una plataforma de hardware libre basada en una placa con un microcontrolador y un entorno de desarrollo diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

El hardware consiste en una placa con un microcontrolador, usualmente Atmel AVR, y puertos de entrada y salida. Para poder programarlo se dispone de un IDE que facilita en gran medida el proceso de programación del código. Este entorno se puede descargar de manera gratuita a través de la página oficial de Arduino [4].

Arduino se presenta en diferentes modelos según especificaciones como el microcontrolador que presenta, frecuencia de reloj, memorias flash, SRAM (Static Random Access Memory) o EEPROM (Erasable Programmable Read-Only Memory).

En este proyecto se trabaja con el modelo Arduino Nano, ya que en el anterior proyecto[3] se decidió diseñar el prototipo con este modelo.

Origen y evolución

Arduino fue desarrollado en el año 2005 por el entonces estudiante del instituto IVRAE Massimo Banzi, quien, en un principio, pensó en hacer Arduino para facilitar el aprendizaje de los estudiantes de computación y electrónica del mismo instituto, ya que en ese entonces, adquirir una placa de microcontroladores eran bastante caro y no ofrecían el soporte adecuado. A lo largo de los años se han ido añadiendo funcionalidades y mejoras a las placas Arduino.

Especificaciones Arduino Nano

Esta versión es la más pequeña del Arduino Uno y está pensada para usarse en protoboard. Seguidamente se nombran las características más destacables:

Microcontrolador: ATmega 328 con cargador de inicio preprogramado:

- Tensión de entrada (límites): +6V a +20V
- Tensión de entrada (Recomendado): +7 a +12V
- Pines Digitales I/O: 14 (6 están provistos de salida PWM)
- Pines analógicos de entrada: 6
- DC para Pines I/O: 40mA
- Memoria Flash: 32KB (2 KB para cargador de inicio (bootloader))
- SRAM: 2KB
- EEPROM: 1KB

- Frecuencia de reloj: 16MHz
- Admite comunicación serie IC
- Dimensiones: 0,73” x 1,7”

En la Figura 2.1 se presenta una imagen de la parte frontal de Arduino Nano con las especificaciones de cada puerto PIN.

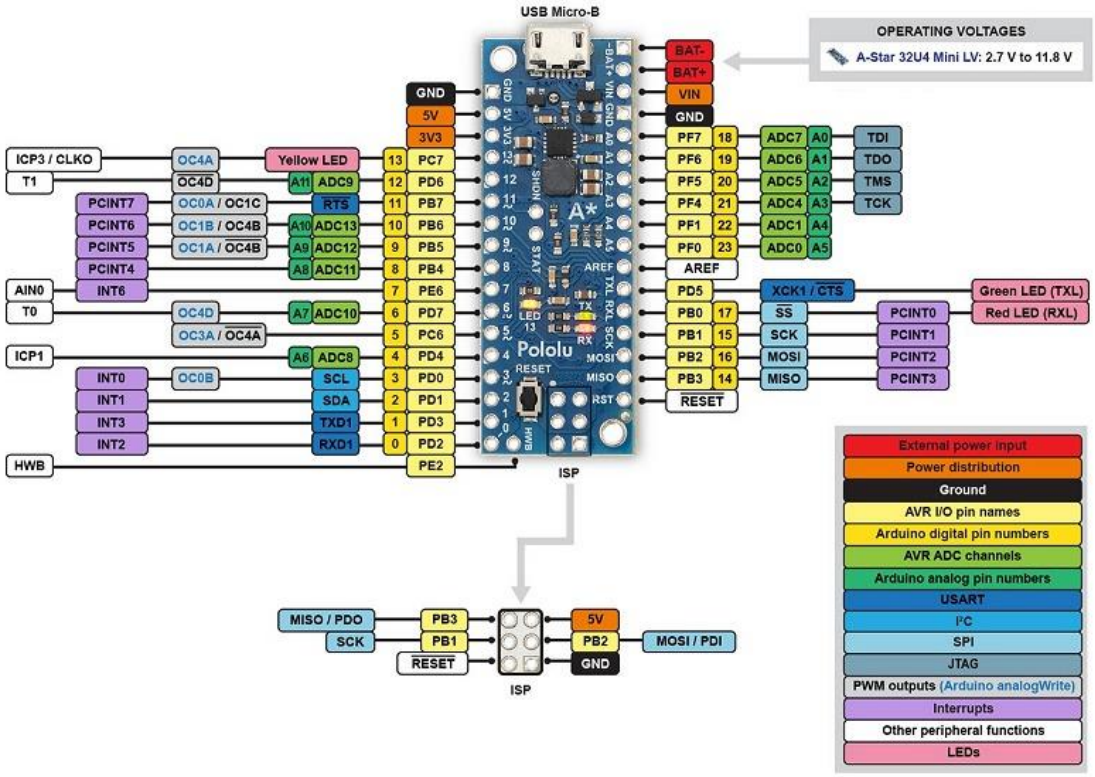


Figura 2.1. Esquema Arduino Nano

El Arduino Nano tiene varios métodos para la comunicación con un PC, otro Arduino u otros microcontroladores. Concretamente el microcontrolador de Arduino integra los siguientes protocolos de comunicación serie:

- **UART** (recepción-transmisión asíncrona universal) es uno de los protocolos serie más utilizados. Usa una línea de datos unidireccional para transmitir y otra para recibir datos. Lo que hace muy útil a este puerto es que trabaja con niveles lógicos TTL (transistor-transistor logic) lo cual hace que dos micros se puedan comunicar sin necesidad de convertir los voltajes con transceptores/drivers. Comúnmente se transmiten 8 bits de datos de la siguiente forma: un bit de inicio (a nivel bajo), 8 bits de datos y un bit de parada (a nivel alto). UART se diferencia de SPI e I2C en que es asíncrono, mientras que SPI e I2C se sincronizan con una señal de reloj. La velocidad de datos UART está limitada a 2Mbps.

- **SPI** es otro protocolo serie muy simple. Un maestro envía la señal de reloj, y tras cada pulso de reloj envía un bit al esclavo y recibe un bit de éste. Los nombres de las señales son por tanto SCK para el reloj, MOSI para el Maestro Out Esclavo In, y MISO para Maestro In Esclavo Out. Para controlar más de un esclavo es preciso utilizar SS (selección de esclavo).

- **I2C** es un protocolo síncrono. I2C usa solo 2 cables, uno para el reloj (SCL) y otro para el dato (SDA). Esto significa que el maestro y el esclavo envían datos por el mismo cable, el cual es controlado por el maestro, que crea la señal de reloj. I2C no utiliza selección de esclavo, sino direccionamiento.

En las siguientes secciones se continúa describiendo los protocolos SPI e I2C más a fondo, ya que su uso y comprensión es primordial para el diseño del prototipo. [5]

2.3. SPI

SPI (Serial Peripheral Interface) se ha ganado la aceptación en la industria como sistema de comunicación de muy corta distancia, normalmente dentro de una placa de circuito impreso, ya que permite controlar casi cualquier dispositivo electrónico digital que acepte un flujo de bits serie regulado por un reloj (comunicación sincrónica). Alcanza velocidades muy altas y además funciona en modo full dúplex.[6]

El bus SPI como se muestra en la Figura 2.2 incluye una línea de reloj, dato entrante, dato saliente y un pin de chip select, que conecta o desconecta la operación del dispositivo con el que uno desea comunicarse. De esta forma, este estándar permite multiplexar las líneas de reloj.

El bus SPI se define mediante 4 pines:

- **SCLK o SCK:** Señal de reloj del bus. Esta señal rige la velocidad a la que se transmite cada bit.

- **MISO (Master Input Slave Output):** Es la señal de entrada a nuestro maestro que será el microcontrolador, por aquí se reciben los datos del integrado esclavo.

- **MOSI (Master Output Slave Input):** Transmisión de datos desde el maestro hacia el esclavo.

- **SS o CS:** Chip Select o Slave Select, habilita el integrado hacia el que se envían los datos. Esta señal es opcional y en algunos casos no se usa. Vale la pena señalar que se necesitan tantos pines como esclavos deseemos multiplexar.

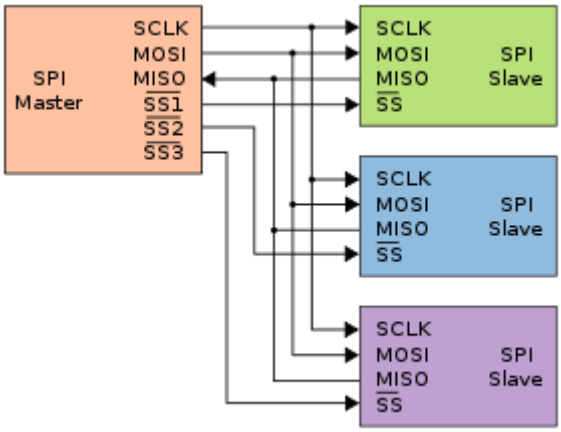


Figura 2.2. Arquitectura SPI

La comunicación siempre es iniciada por el maestro y las fases para establecer dicha comunicación son las siguientes:

- Se habilita el chip al que hay que enviar la información mediante el CS (Opcional).
- Se carga en el buffer de salida el byte a enviar.
- La línea de CLOCK empieza a generar la señal cuadrada donde normalmente por cada flanco de bajada se pone un bit en MOSI.
- El receptor normalmente en cada flanco de subida captura el bit de la línea MISO y lo incorpora en el buffer.

Si se ha terminado de transmitir se vuelve a poner la línea CS en reposo. Hay que tener en cuenta que a la vez que el maestro está enviando un dato también puede estar recibiendo datos de alguno de los esclavos, cuenta por lo tanto con comunicación full-duplex.

La señal de reloj es generada por el maestro y la línea SS normalmente se mantiene HIGH y se activa con LOW, lo que despierta al esclavo seleccionado. Cuando se termina la transferencia la línea se levanta a HIGH y el esclavo se desactiva.

En la Figura 2.3 se muestra un ejemplo de una comunicación maestro-esclavo y otra esclavo-maestro.

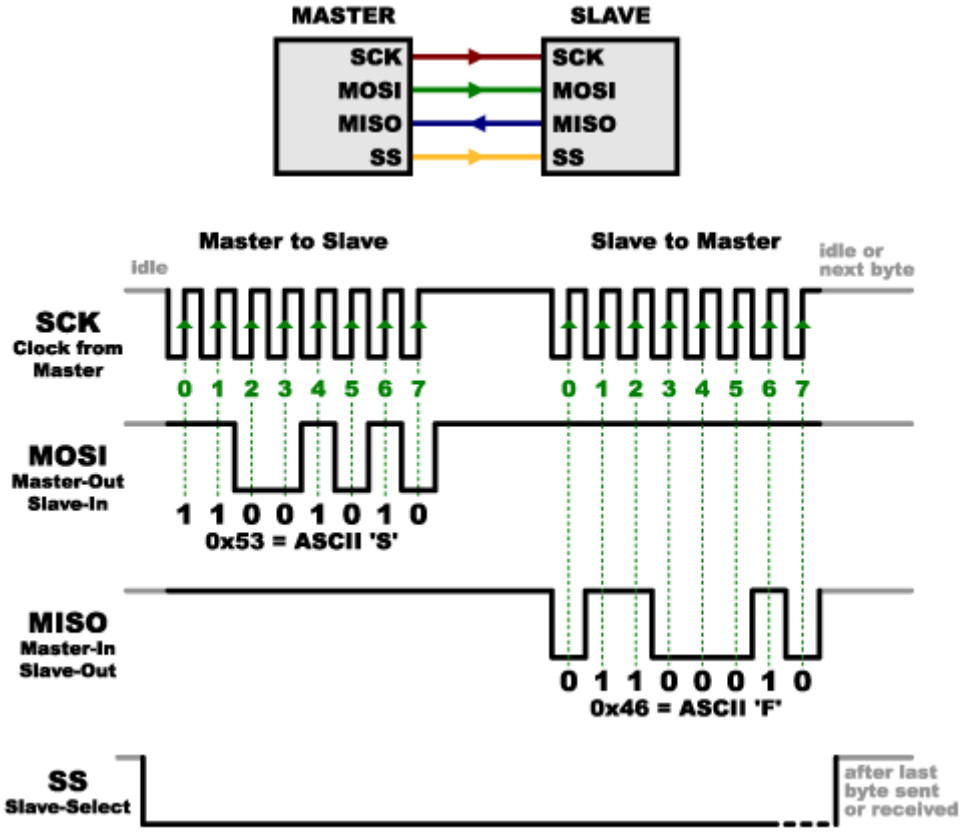


Figura 2.3. Ejemplo de comunicación SPI

Referente a Arduino, éste soporta de serie el bus SPI, con una librería estándar llamada "SPI.h" que está incluida en el IDE de Arduino y que gestiona todas las complicaciones y el arbitraje del protocolo.

Métodos librería SPI de Arduino:

- begin() — Inicializa el bus SPI
- end() — Deshabilita el bus SPI.
- *setBitOrder()* — Configura el orden de los bits enviados como el menos significativo primero o el más significativo primero.
- *setClockDivider()* — Configura el divisor de reloj en el bus SPI. Es decir, configura la velocidad del bus.
- *setDataMode()* — Configura el modo en el cual se envía la información en el bus SPI. Existen cuatro modos dependiendo de dos parámetros basados en la señal de reloj. El primero de ellos es la polaridad y el segundo es la fase. La polaridad determina si el estado Idle de la línea de reloj está en bajo o si se encuentra en un estado alto. La fase determina en que flanco se empieza a transmitir el nuevo dato sobre el bus.
- *transfer()* — Transfiere un byte sobre el bus SPI, tanto de envío como de recepción.

Se puede encontrar más información sobre los parámetros SPI en el apartado *settings* de la referencia [7].

Los pines asociados a SPI en Arduino Nano son los siguientes:

- Select Slave (SS): 10
- Master Output Slave Input (MOSI): 11
- Master Input Slave Output (MISO): 12
- Serial Clock (SCK): 13

Se puede encontrar más información de la librería en la referencia [8].

En la Figura 2.4 se analiza un rastro de bits presentes en una comunicación SPI del proyecto (señal A1 en la parte superior de la pantalla) para calcular la tasa de bits de la comunicación, dando en esta captura una tasa de 8.33 Mbps.

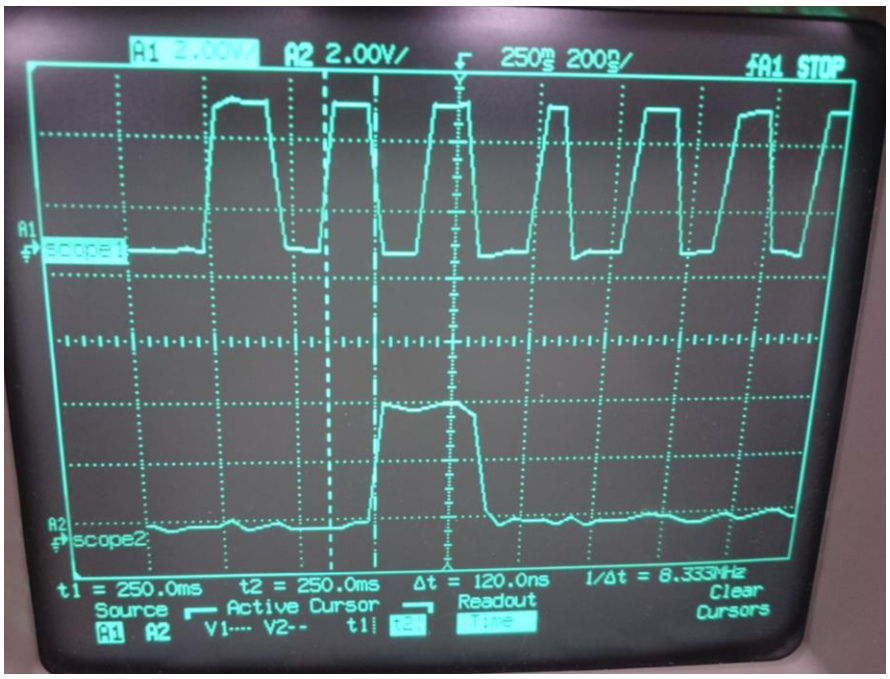


Figura 2.4. Captura osciloscopio comunicación SPI en Arduino

2.4. I2C

Es otro bus de comunicaciones en serie muy usado en la industria, principalmente para comunicar microcontroladores y sus periféricos en sistemas integrados (Embedded Systems). Es capaz de soportar velocidades de entre 100 kbit/s en el modo estándar, hasta 3.4 Mbit/s. [9]

La principal característica de I²C es que utiliza dos líneas para transmitir la información: una para los datos y otra para la señal de reloj.

El bus I2C se define mediante 2 pines:

- SDA: datos
- SCL: reloj

Estas líneas SDA y SCL son del tipo drenador abierto, por lo que necesitan resistencias de pull-up. Dos o más señales a través del mismo cable pueden causar conflicto, y ocurrirían problemas si un dispositivo envía un 1 lógico al mismo tiempo que otro envía un 0. Por tanto, el bus es "cableado" con dos resistencias para poner el bus a nivel alto, y los dispositivos envían niveles bajos. Si quieren enviar un nivel alto simplemente dejan el bus en alta impedancia.

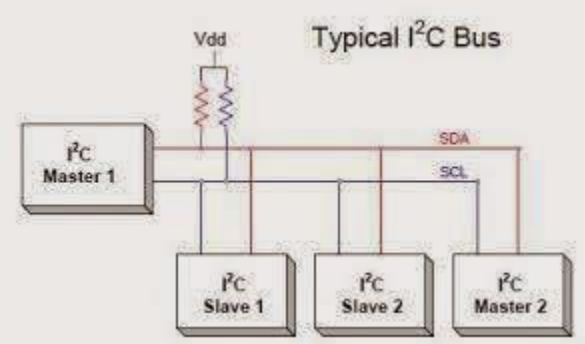


Figura 2.5. Esquema I2C

Los dispositivos conectados al bus I2C tienen una dirección única para cada uno. Este bus tiene la característica particular de que puede haber más de un maestro en él. El dispositivo maestro inicia la transferencia de datos y además genera la señal de reloj, pero no es necesario que el maestro sea siempre el mismo dispositivo, este rol se lo pueden ir pasando los dispositivos que tengan esa capacidad. Esta característica hace que al bus I2C se le denomine bus multimaestro.

El proceso de comunicación en el bus I2C es el siguiente:

- El maestro comienza la comunicación enviando un patrón llamado "*start condition*". Esto alerta a los dispositivos esclavos, poniéndolos a la espera de una comunicación.
- El maestro se dirige al dispositivo con el que quiere hablar, enviando un byte que contiene los siete bits (A7-A1) que componen la dirección del dispositivo esclavo con el que se quiere comunicar, y el octavo bit (A0) de menor peso se corresponde con la operación deseada (L/E), lectura=1 (recibir del esclavo) y escritura=0 (enviar al esclavo).
- La dirección enviada es comparada por cada esclavo del bus con su propia dirección, si ambas coinciden, el esclavo se considera direccionado como esclavo-transmisor o esclavo-receptor dependiendo del bit R/W.
- Cada byte leído/escrito por el maestro debe ser obligatoriamente reconocido por un bit de ACK por el dispositivo maestro/esclavo.

- Cuando la comunicación finaliza, el maestro transmite una "*stop condition*" para dejar libre el bus.

Referente a Arduino, éste soporta de serie I2C con una librería estándar llamada "*Wire.h*" que está incluida en el IDE de Arduino y que gestiona todas las complicaciones del protocolo.

Métodos librería *Wire.h* de Arduino:

- *begin()* – Inicia la librería Wire y especifica si es el Arduino hace de maestro o esclavo.
- *requestFrom()* – Usado por el maestro para solicitar datos del esclavo.
- *beginTransmission()* – Comenzar transmisión con esclavo.
- *endTransmission()* – Finaliza la transmisión que comenzó con un esclavo y transmite los bytes en cola.
- *write()* – Escribe datos desde un esclavo como respuesta a una petición del maestro o pone en cola la transmisión de un maestro.
- *available()* – Devuelve el número de bytes para leer
- *read()* – Lee un byte transmitido desde un esclavo a un maestro o viceversa
- *onReceive()* – Llama a una función cuando un esclavo recibe una transmisión de un maestro. Registra una función de callback.
- *onRequest()* – Llama a una función cuando un maestro solicita datos de un maestro. Registra una función de callback.

Los pines asociados a I2C en Arduino Nano son los siguientes:

- Serial Data Line (SDA): A4
- Serial Clock Line (SCL): A5

Se puede ver más información de la librería la referencia [10].

2.5. BUS CAN

Origen de CAN

El bus CAN (Controller Area Network) surge de la necesidad en los años ochenta de encontrar una forma de interconectar los distintos dispositivos de un vehículo de una manera sencilla y reduciendo significativamente las conexiones. De esta forma, la empresa Robert Bosch GmbH logra desarrollar el bus CAN, que posteriormente se estandariza en la norma ISO 11898-1. [11]

Características

CAN es un protocolo de comunicación en serie de bus compartido desarrollado para el intercambio de información entre unidades de control electrónicas del vehículo (ECUs).

Esto provoca una reducción importante tanto del número de sensores utilizados como de la cantidad de cables que componen la instalación eléctrica, pudiendo aumentar las funciones en los sistemas del vehículo donde se emplea el CAN-Bus sin aumentar los costes.

La comunicación por el bus es de tipo broadcast, es decir, todas las ECUs reciben los mensajes que se transmiten por el bus, siendo el máximo de nodos conectados al bus de hasta 110. Cabe destacar que todos los nodos tienen la misma importancia, careciendo del mecanismo maestro-esclavo, por lo que en el caso de que un nodo fallara el sistema seguiría funcionando.

El protocolo define una estructura para las tramas transmitidas.

Además, el protocolo garantiza varias velocidades de transmisión según el tipo de red CAN: red de alta velocidad (de hasta 1 Mbps) definida por la ISO 11898-2 y una red de baja velocidad tolerante a fallos (menor o igual a 125 Kbps) definida por la ISO 11898-3.

Existen diferentes versiones del protocolo, la última es CAN 2.0. Esta especificación tiene a la vez dos versiones: CAN 2.0A para el formato estándar con un identificador de 11 bits y CAN 2.0B para el formato extendido con un identificador de 29 bits.

Referente a los posibles errores producidos en las comunicaciones, CAN tiene hasta cinco mecanismos para detectarlos: tres a nivel de mensaje (chequeo de la trama, error de formato de trama y error en ACK) y dos a nivel de bit (error polaridad de bit y stuff error), pudiendo ser señalizados estos errores.

Capas de la pila OSI

Dentro de la pila OSI (Open System Interconection), CAN bus cubre la capa de Enlace de datos y la Física.

Capa de enlace de datos

El protocolo de acceso al medio utilizado es el CSMA/CD + AMP (Carrier Sense Multiple Access/ Collision Detection + Arbitration on Message Priority). En dicho protocolo, los distintos nodos conectados realizan una escucha del medio, de forma que conocen cuándo un mensaje está siendo transmitido. De esta forma, los nodos evitarán mandar una trama de datos cuando la red está ocupada.
En caso de colisión, dos nodos quieren transmitir a la vez, el mensaje con mayor prioridad es el que accederá primero al bus, siendo el más prioritario el nodo con el

identificador más bajo.[11][12] En la Figura 2.6 se muestra un ejemplo de cómo se soluciona el caso en el que tres nodos que quieren transmitir a la vez.

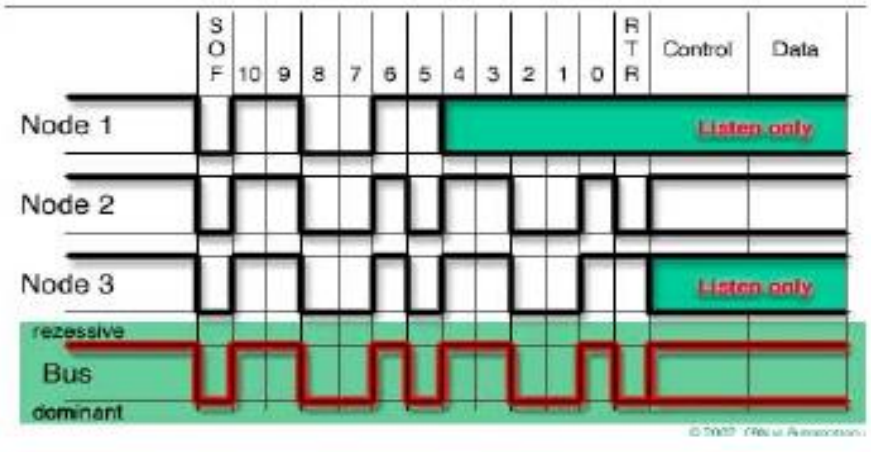


Figura 2.6. Ejemplo competición por el medio CAN

Dependiendo de cómo esté implementado el dispositivo CAN este puede tener uno o varios búferes de transmisión y/o recepción. Dichos búferes permiten controlar el tráfico de mensajes mediante filtros y máscaras.

Capa física

La capa física es la responsable de la transferencia de bits entre los distintos nodos que componen la red. Define aspectos tales como niveles de señal, codificación, sincronización y tiempos de bit.

En CAN destacan principalmente dos configuraciones de la capa física: high-speed CAN y low-speed CAN. Ambas se basan en un par trenzado de cables, siendo la única diferencia que high-speed CAN ofrece una tasa de bits de 125 kbps a 1 Mbps y en cambio low-speed CAN de 10 kbps a 125 kbps. Ya sea en el caso de high-speed CAN o low-speed CAN los dos cables que utilizan reciben el nombre de CAN_H y CAN_L. El objetivo de usar cables de par trenzado es el filtrado de las interferencias que pueden ocurrir en el bus.

Para el envío de datos es necesario codificar estos mismos, CAN para esto utiliza una codificación NRZ (Non Return to Zero). A continuación, se explicarán las características tanto del bus High-speed CAN como del bus Low-speed CAN. El bus de High-speed CAN es un bus pasivo ya que al contrario que Low-speed CAN, el cual será activo, no tiene ninguna fuente externa de alimentación. Tanto high-speed CAN como low-speed CAN presenta una red cableada, en el caso de CAN H con una resistencia de 120 Ω en sus extremos cerrando el circuito, como se puede observar en la Figura 2.7.

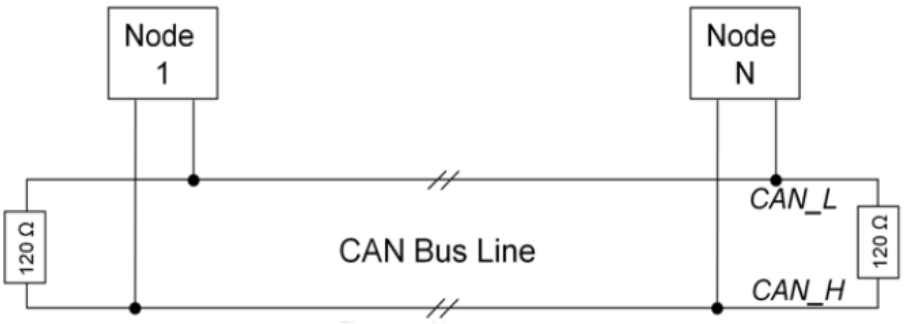


Figura 2.7. Esquema CAN

Trama CAN bus

La trama CAN-Bus está basada 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 interese o no [11][12].

Las tramas utilizan *bit stuffing*, es decir, cuando se suceden 5 bits iguales, se introduce un bit extra de valor contrario para evitar la desincronización. Este mecanismo también sirve para señalar errores, dado que una secuencia de más de 5 bits iguales supondría algún tipo de error u otro tipo de señalización como final de trama.

Existen dos formatos de trama, estándar y extendida, en base al número de bits del identificador.

CAN predefine unos tipos de trama para la gestión de la transferencia de mensajes:

- Trama de datos: Como su propio nombre indica, dichas tramas se utilizan para enviar datos a través de la red. Los datos se incluirán en el campo de datos y pueden tener una extensión de 0 a 8 bytes.
- Trama remota: Un nodo tiene la capacidad de solicitar un mensaje de otro nodo usando tramas remotas. El identificador de la trama debe ser el mismo que el del nodo del cual se quiere recibir el mensaje. Además, el campo de datos será 0. Una vez que el nodo receptor reciba el mensaje, éste enviará sus datos.
- Trama de error: Se genera al detectar un error en la red por parte de un nodo. Está formada por un campo indicador de error y un campo delimitador.
- Trama de saturación: A diferencia de la trama de error, la trama de saturación sólo se da entre tramas. Es generada al detectar un bit dominante en el espacio entre tramas o al no ser posible el envío de un mensaje por problemas internos.

Figura 2.8. Trama CAN

En la Figura 2.8 se presenta el formato de trama, la cual consiste en celdas que envían datos y añaden información definida por las especificaciones CAN. Los campos de la trama son:

- SOF (Start of Frame bit): Indica el comienzo del mensaje y permite la sincronización de todos los nodos conectados a la red. Este bit tiene estado dominante (0 lógico).
- Campo de arbitrio: Está formado por 12 bits o 32 bits dependiendo del tipo de trama. Dentro del campo se encuentra el identificador, el cual indica la prioridad del nodo. El nodo con mayor prioridad es aquel que tiene el identificador más bajo. El bit RTR se utiliza para distinguir entre una trama remota o una trama de datos.
- Campo de control: Formado por 6 bits. El bit IDE indica con un estado dominante que la trama enviada es estándar. El bit RB0, está reservado y se establece en estado dominante por el protocolo CAN. El resto de bits, el Data Length Code (DLC) indica el número de bytes de datos que contiene el mensaje. Una trama extendida tiene un bit adicional RB1.
- Campo de datos: Está formado por hasta 8 bytes, dependiendo de lo que especifiquemos en el DLC. En este campo están los datos del mensaje.
- Campo de verificación por redundancia cíclica: Este campo de 15 bits, detecta errores en la transmisión del mensaje. Se delimita con un bit final en estado recesivo.
- Campo de reconocimiento: El último campo de la trama, está formado por 2 bits. El nodo transmisor manda una trama con el bit de ACK (Acknowledge) en estado recesivo, mientras que los receptores, si han recibido el mensaje correctamente, mandarán un mensaje en estado dominante. Contiene un bit delimitador.
- Campo de fin de trama: Una serie de 7 bits recesivos indican el fin de la trama.

2.6. OTROS MÓDULOS HARDWARE DISPONIBLES

- MCP2515: Controlador autónomo CAN fabricado por Microchip capaz de transmitir y recibir tramas de datos a través de una interfaz SPI. Implementa CAN V2.0B y CAN V2.0A con una velocidad máxima de 1 Mbps.

- MCP2551: Transceptor CAN, también fabricado por Microchip, que hace de interfaz entre el controlador CAN y el bus.
- Módulo SD: Permite a un dispositivo electrónico realizar una comunicación con una tarjeta SD vía SPI con la finalidad de leer y escribir ficheros.
- Módulo RTC: Un RTC (Real Time Clock, o reloj de tiempo real) es un dispositivo electrónico que gracias a una pila de litio es capaz de llevar la cuenta de segundos, minutos y horas, además de día, mes y año automáticamente. Se comunica mediante el protocolo I2C
- Conversor de voltaje o regulador LM317t: Permite convertir un voltaje de entrada de hasta 40V a una salida configurable de 1,2 a 37V.

CAPÍTULO III: MESA DE SIMULACIÓN DEL MODELO CAPTUR

3.1. INTRODUCCIÓN

El prototipo a diseñar debe de ser capaz de capturar de manera transparente los datos que viajan en los buses CAN del modelo Renault Captur para almacenarlos en una tarjeta SD con el fin de su posterior análisis. Ya que el desarrollo del datalogger se tiene que centrar en este modelo de vehículo se desarrolla este capítulo para poder conocer en detalle la arquitectura y funcionamiento de la parte eléctrica del Captur.

3.2. CARACTERÍSTICAS DE LA MAQUETA

La maqueta, que se encuentra en un laboratorio de la ETSIT (Escuela Técnica Superior de Ingenieros de Telecomunicación), posee todos los elementos eléctricos y electrónicos del vehículo menos el motor. Es una reproducción de la presente en la fábrica Renault de Valladolid, que utilizan con el fin de detectar errores que surgen en la electrónica del modelo Captur.

En la Figura 3.1 y la Figura 3.2 se ilustra la parte frontal y trasera.[3]

Figura 3.1. Maqueta Renaul Captur - Frontal

Figura 3.2. Maqueta Renaul Captur - Trasera

Los componentes que conforman el bus CAN del Renault Captur se pueden dividir en 7 secciones:

- Cableado del coche: lo componen cada parte del vehículo, como es el motor, las puertas, el maletero, …
- Calculadores (ECUs): tablero de abordo, unidad de control, gestor de energía de los diferentes dispositivos, ayuda y asistencia al aparcamiento, control aire acondicionado, controles de la radio, accionamiento del airbag si hay impacto, …
- Iluminación y señalización: cada una de las luces del vehículo se encuentran en esta sección, además de la bocina e indicador sonoro de asistencia en el aparcamiento del vehículo.
- Botones y mandos: todos los controles manuales del vehículo, como pueden ser el subir y bajar ventanillas, habilitar/inhabilitar funciones, sistemas de control en el volante, …
- Sensores y captadores: en esta sección nos encontramos sensores como el de lluvia, temperatura, ayuda en el aparcamiento, …, y captadores como los de airbag, de pedales, de velocidades, …
- Motores, cerraduras y retrovisores: todos los motores internos como los elevalunas, limpiaparabrisas, …
- Resto de elementos: encendedor y altavoces.

La topología que presenta el bus CAN en la maqueta es una topología en bus en la que todas las ECUs están conectadas al bus, es decir, no hay ECUs que estén conectadas a otra ECU y a su vez esta ECU al bus. Como se ha indicado anteriormente, este modelo Captur tiene dos buses CAN diferentes, uno para los datos genéricos y otro para los datos multimedia. Se ilustra en una misma imagen una parte de cada bus, la superior se corresponde al bus CAN de datos y el inferior al bus CAN multimedia.

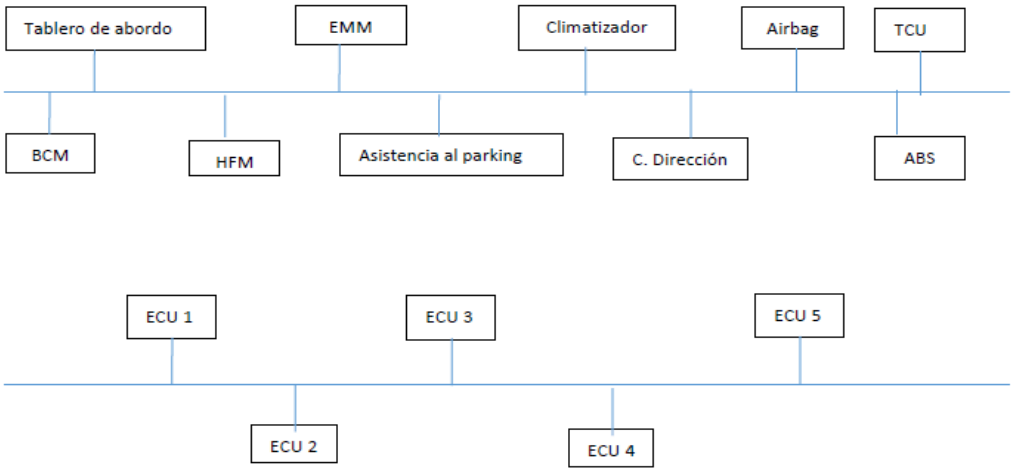


Figura 3.3. Esquema Bus CAN del Renault Captur

La velocidad de transmisión de los mensajes CAN en ambos buses es de 500 Kbps.

3.3. CAPTURA DE DATOS

Con el fin de obtener una aproximación de las características de los mensajes que viajan por los buses CAN del Captur se procede a una captura, con la herramienta BUSMASTER, que sirva de ayuda para describir periodicidades de los mensajes, media de mensajes en el canal en un segundo y tipos de tramas.

BUSMASTER es una herramienta de software libre, diseñada e implementada por la empresa Robert Bosch, que nos permite simular, adquirir y analizar los datos que se envían por un bus de comunicación utilizando protocolos como pueden ser CAN, FlexRay o LIN. En este caso, se centrará la adquisición en CAN, que es el único protocolo que se utiliza dentro de la maqueta. En esencia este programa realiza las mismas funciones que la herramienta CANoe, el más utilizado dentro de la industria del motor. Quedan algunos aspectos del mismo por desarrollar para alcanzar la potencia de esta última herramienta, pero para los objetivos que se plantean en este proyecto, BUSMASTER será una herramienta muy útil.

Además del programa BUSMASTER es necesario utilizar un dispositivo electrónico que nos permita capturar los mensajes que se envían por el bus. En este caso se ha utilizado CANcaseXL de la marca Vector que se puede ver en la Figura 3.4. CANcaseXL es una interfaz USB que incorpora un microcontrolador ARM7 core de 32 bit a 64MHz de la marca ATMEL, además de dos controladores CAN SJA1000 de Philips, permitiéndonos por lo tanto leer mensaje de dos canales CAN.

Figura 3.4. Dispositivo CANcaseXL

Para poder separar los dos canales CAN presentes en el Captur se ha utilizado un cable diseñado y elaborado por el anterior compañero de proyecto[3]. Este cable consta de un conector OBD-II hembra en uno de sus extremos y dos conectores DB9 macho en el otro extremo. En la Figura 3.5 se ilustra el OBD-II disponible en el Renault Captur y el cable anteriormente descrito.

Figura 3.5. OBD-II y Conector particular para Renault Captur

Con esto y con algunas configuraciones en BUSMASTER se logra capturar los mensajes en tiempo real presentes en el Renault Captur. Es interesante guardar un fichero en modo log para poder comparar más adelante los resultados reales con los obtenidos con el datalogger que se pretende alcanzar.

```
***BUSMASTER Ver 3.1.0***
***PROTOCOL CAN***
***NOTE: PLEASE DO NOT EDIT THIS DOCUMENT***
***[START LOGGING SESSION]***
***START DATE AND TIME 23:2:2017 12:56:51:267***
***HEX***
***SYSTEM MODE***
***START CHANNEL BAUD RATE***
***CHANNEL 1 - Vector - CANcaseXL Channel 1 SN - 23653 - 500000 bps***
***CHANNEL 2 - Vector - CANcaseXL Channel 2 SN - 23653 - 500000 bps***
***END CHANNEL BAUD RATE***
***START DATABASE FILES***
***END DATABASE FILES***
***<Time><Tx/Rx><Channel><CAN ID><Type><DLC><DataBytes>***
12:57:02:9637 Rx 2 0x58D s 8 FF FF 00 00 00 00 00 00
12:57:02:9643 Rx 2 0x58C s 8 C7 80 F4 FF FF 7F D0 00
12:57:02:9646 Rx 2 0x423 s 8 04 00 00 00 00 00 00 00
12:57:02:9649 Rx 2 0x548 s 8 00 00 0F 00 00 00 00 00
12:57:02:9662 Rx 2 0x1A1 s 8 00 00 00 00 00 00 00 00
12:57:02:9666 Rx 2 0x578 s 8 00 00 0E E0 00 00 00 00
12:57:02:9669 Rx 2 0x58B s 8 5C F8 FF FF 0B 70 00 00
12:57:02:9671 Rx 2 0x590 s 8 00 FF FE 3C 00 00 00 00
12:57:02:9674 Rx 2 0x58E s 8 FF FF FF FF FF C0 00 00
12:57:02:9678 Rx 2 0x592 s 8 FF 00 FF FF FF FF C6 00
12:57:02:9681 Rx 2 0x5A8 s 8 03 FF FF FF FF 02 00 00
12:57:02:9684 Rx 2 0x58F s 8 80 00 40 00 00 BB BB BB
12:57:02:9687 Rx 2 0x58D s 8 FF FF 00 00 00 00 00 00
12:57:02:9693 Rx 2 0x58C s 8 C7 80 F4 FF FF 7F D0 00
12:57:02:9696 Rx 2 0x423 s 8 04 00 00 00 00 00 00 00
12:57:02:9696 Rx 1 0x1B0 s 4 42 2C 1E 00
12:57:02:9699 Rx 2 0x4DE s 8 00 00 00 BB BB BB BB BB
12:57:02:9703 Rx 2 0x1A1 s 8 00 00 00 00 00 00 00 00
12:57:02:9706 Rx 2 0x58E s 8 FF FF FF FF FF C0 00 00
12:57:02:9709 Rx 2 0x592 s 8 FF 00 FF FF FF FF C6 00
12:57:02:9743 Rx 2 0x316 s 8 FE 7F 3F E0 00 00 03 00
12:57:02:9788 Rx 1 0x204 s 3 C0 00 00
```

Figura 3.6. Archivo log de la maqueta con BUSMASTER

Las estadísticas que servirán de especificaciones para nuestro sistema son las siguientes:

- Media de mensajes: alrededor de 600 tramas CAN en un segundo o lo que es lo mismo 0,6 tramas en 1 milisegundo.
- Tiempo mínimo entre dos mensajes: 200 microsegundos.
- Ráfagas: cuatro tramas en un milisegundo.

3.4. REQUESITOS DEL DATALOGGER

Nuestro prototipo datalogger debe de ser capaz de alcanzar los siguientes objetivos:

- Alimentarse eléctricamente por medio del OBD-II del automóvil.
- Sincronizarse y transferir/recibir datos por el bus compartido SPI con dos módulos MCP2515, uno por cada canal CAN, y con la tarjeta de memoria flash SD.
- Sincronizarse y transferir/recibir datos por el bus I2C con el módulo RTC (reloj del sistema).
- Recoger tramas CAN y almacenar estas en un fichero, en formato de los log habituales, creado por el datalogger en la tarjeta SD en el tiempo mínimo para poder alcanzar o acercarse a los requisitos de tiempos descritos en la sección anterior.
- Hacer todas las tareas de manera transparente.
- Dimensión de la placa reducida, de tal forma que su manejo en la fábrica sea lo más fácil posible.

CAPÍTULO IV: PRIMER PROTOTIPO CON UN ARDUINO

4.1. FUNCIONAMIENTO

En este primer prototipo se pretende diseñar el hardware y el software, además de montar la placa y su posterior testeo, de un datalogger con las prestaciones detalladas del capítulo anterior.

Se va a utilizar como cerebro del sistema un Arduino Nano, por el hecho de que uno de los requisitos es que la dimensión de la placa sea reducida y este modelo es de las versiones más pequeñas de Arduino.

El funcionamiento es el siguiente: Arduino accederá a los módulos CAN, vía SPI, para leer tan rápido como sea posible las tramas que el MCP2515 captura (controlador CAN). Por cada trama, el Arduino recogerá la hora, minutos y segundos en tiempo real del módulo RTC (Real Time Clock) vía I2C con el fin de etiquetar la trama con ese instante. Seguidamente el software del programa formará un mensaje con la etiqueta de tiempo y los datos de la trama CAN para transmitírsela vía SPI a la tarjeta SD, quedando este mensaje grabado en un fichero creado en el inicio del programa.

Para cada ejecución del sistema (conexión/desconexión o reseteo del programa) Arduino creará un nuevo fichero siguiendo el formato de fichero que usan programas como BUSMASTER y CANoe, con el objetivo de que posteriormente se pueda realizar una reproducción de nuestros ficheros en estos programas (la opción en BUSMASTER se indica con *Replay*). Para conseguirlo, Arduino escribe una cabecera específica en el fichero de la SD y a continuación los mensajes formados por el etiquetado de tiempos y tramas CAN.

4.2. DISEÑO HARDWARE

En este capítulo se intenta seguir el desarrollo temporal del datalogger, es decir, por pasos tal y como se ha hecho personalmente, comprobándose que cada módulo funcionaba de manera correcta y de manera independiente a los demás, antes de llevarse a cabo la unión de todos ellos en la placa.

Alimentación del sistema

Para dar una tensión suficiente que garantice que el Arduino y sus módulos exteriores se ponen en funcionamiento se aprovecha el conector OBD-II de un vehículo, ya que incluye una línea con la tensión de la batería. La tensión que

proporciona un vehículo de forma genérica es de 12V, por lo que se debe de diseñar un circuito integrando el regulador LM317t [13].

Arduino Nano marca en sus especificaciones que trabaja en un rango de tensiones de 2.7V a 11.8V (este proporciona a su vez la tensión necesaria para los demás módulos).

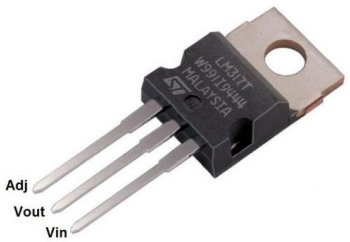


Figura 4.1. LM317t

Se escoge una tensión de alimentación de 7V para evitar que el Arduino pueda verse dañado por una sobretensión. Con este valor y con ayuda del datasheet del regulador LM317t [13] se han ajustado las resistencias oportunas para obtener la salida deseada, siendo estas R1 igual a 470 Ω y R2 igual a 2,1 KΩ. Además, se incluye protección ante cortocircuitos añadiéndose el diodo D1. De este modo se ilustra en la Figura 4.2 y en la Figura 4.3 el circuito resultante.

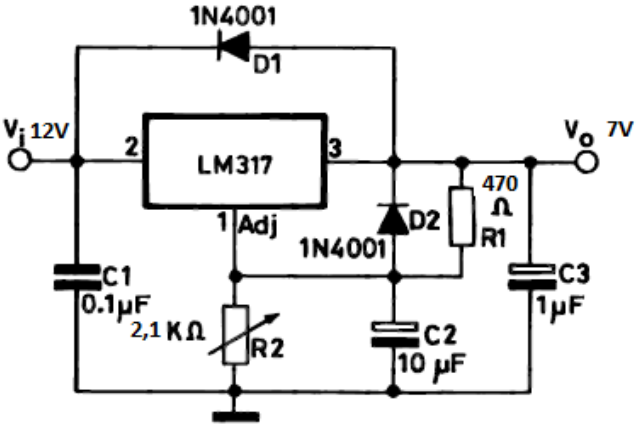


Figura 4.2. Esquema conversor de voltaje a 7V

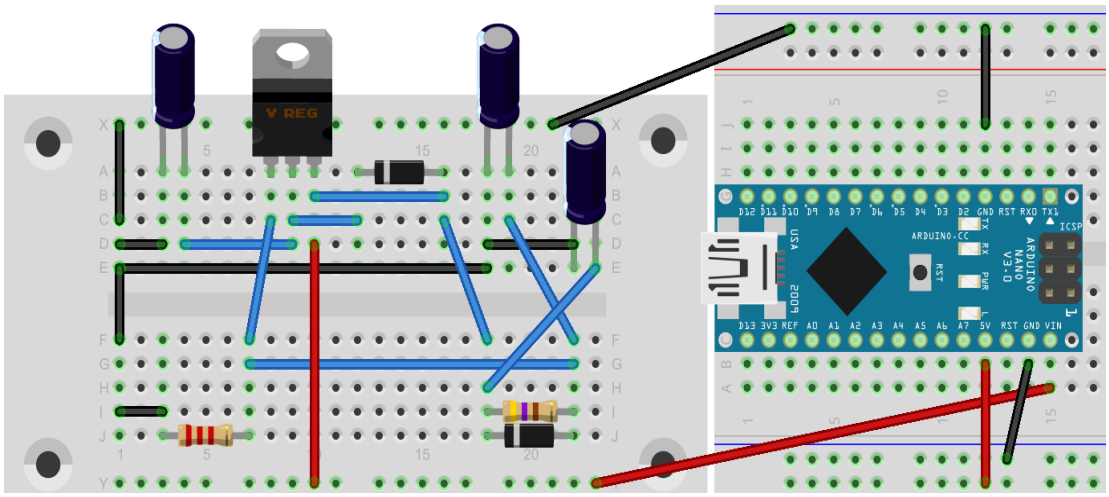


Figura 4.3. Esquema Fritzing conversor de voltaje a 7V - Arduino

RTC (Real Time Clock)

Este dispositivo proporciona un reloj al sistema y en nuestro datalogger será necesario tanto para el etiquetado de la cabecera del archivo log (fecha, hora, minutos y segundos en el que es creado) como para el etiquetado individual de cada trama CAN (hora, minutos y segundos de la captura).

Además, se debe saber que Arduino incorpora sus propios temporizadores (uno de ellos es *millis()*). Su función es la de llevar un contador para indicar el tiempo que Arduino lleva encendido. Ya que el valor de este temporizador se pierde una vez el Arduino deja de estar alimentado no nos sirve como reloj para el datalogger. Sin embargo, dado que el RTC únicamente nos aporta una precisión de segundos y para la captura de los mensajes es necesario una precisión de milisegundos, se aprovecha dicha función *millis()* en la captura de los mensajes.

Finalmente, se utiliza un RTC click de mikroBUS [14]. Este dispositivo se puede alimentar con 3.3V o con 5.5V (sólo se usa para los niveles de tensión necesarios para las comunicaciones entre Arduino-I2C). Se ilustra el módulo tanto por su cara frontal como posterior en la Figura 4.4.

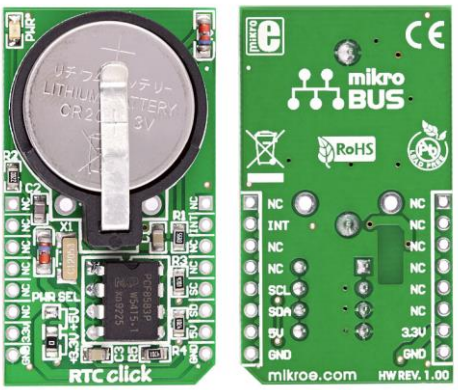


Figura 4.4. RTC mikroBUS

La comunicación del RTC es bajo el protocolo I2C, por lo que los pines de datos (SDA) y de señal de reloj (SCL) se conectan a los pines A4 y A5, respectivamente, del Arduino Nano.

A continuación se muestra el esquema de conexión:

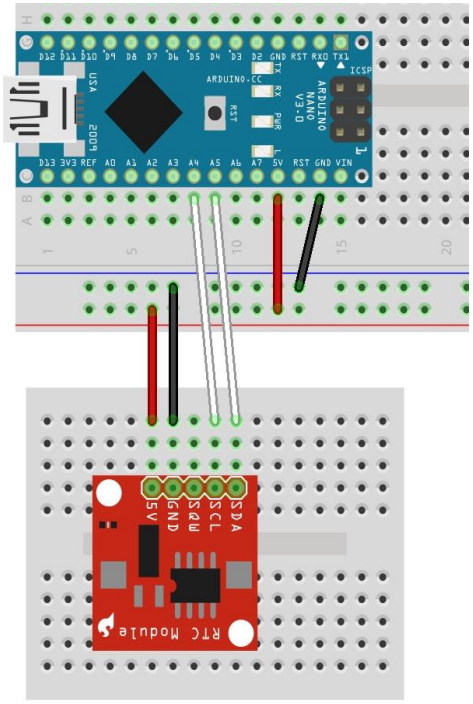


Figura 4.5. Esquema Fritzing RTC - Arduino

MCP2515 y MCP2551

Como ya se ha descrito en el capítulo de materiales disponibles, estos módulos trabajan bajo el protocolo CAN. Ambos dispositivos en conjunto nos permitirán adaptar la señal del bus físico para establecer una comunicación SPI donde Arduino recibirá las tramas que viajan en los buses CAN del Captur.

El MCP2551 [16] es un transceptor CAN de alta velocidad resistente a fallos que sirve de interfaz entre un controlador de protocolo CAN y el bus físico. Proporciona capacidad de transmisión y recepción diferencial al controlador de protocolo CAN y es totalmente compatible con el estándar ISO-11898. Sus principales características son:

- Entrada de control de pendiente
- Soporta funcionamiento de 1 Mb/s
- Implementa los requisitos de capa física estándar según ISO-11898
- Apto para sistemas de 12V y 24V
- Funcionamiento en suspensión de baja corriente
- Alta inmunidad al ruido debido a implementación de bus diferencial

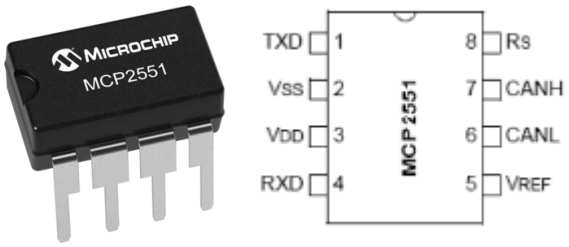


Figura 4.6. MCP2551

Número de pin	Nombre del pin	Función del pin
1	TxD	Entrada de transmisión de datos
2	Vss	Tensión de masa
3	VDD	Tensión de alimentación
4	RxD	Salida de recepción de datos
5	Vref	Voltaje de referencia
6	CANL	Nivel bajo de la señal CAN
7	CANH	Nivel alto de la señal CAN
8	Rm	Resistencia de control de modo

Tabla 4.1. Definición pines MCP2551

Por su parte, el MCP2515 [16] es un controlador autónomo CAN capaz de transmitir y recibir tramas de datos en formato estándar y extendido y tramas remotas. Cuenta con dos máscaras de aceptación y seis filtros de aceptación que se usan para descartar mensajes no deseados. El MCP2515 se conecta con los microcontroladores (MCUs) mediante una interfaz de periféricos serie (SPI) de estándar industrial. Sus principales características son:

- Longitud de 0-8 bytes en el campo de datos.
- Tramas de datos de formato estándar y extendido y tramas remotas.
- Dos búferes de recepción con almacenamiento de mensajes prioritarios.
- Seis filtros de 29 bits.
- Dos máscaras de 29 bits.
- Tres búferes de transmisión con funciones de priorización e interrupción.
- Interfaz SPI de alta velocidad (10 MHz).
- Modo de disparo único que garantiza que la transmisión de mensajes se intente una sola vez.
- Pin de salida de reloj con prescaler programable que se puede utilizar con fuente de reloj para otros dispositivos.
- Pin de salida de interrupción con habilitaciones seleccionables.
- Tecnología CMOS de baja potencia.

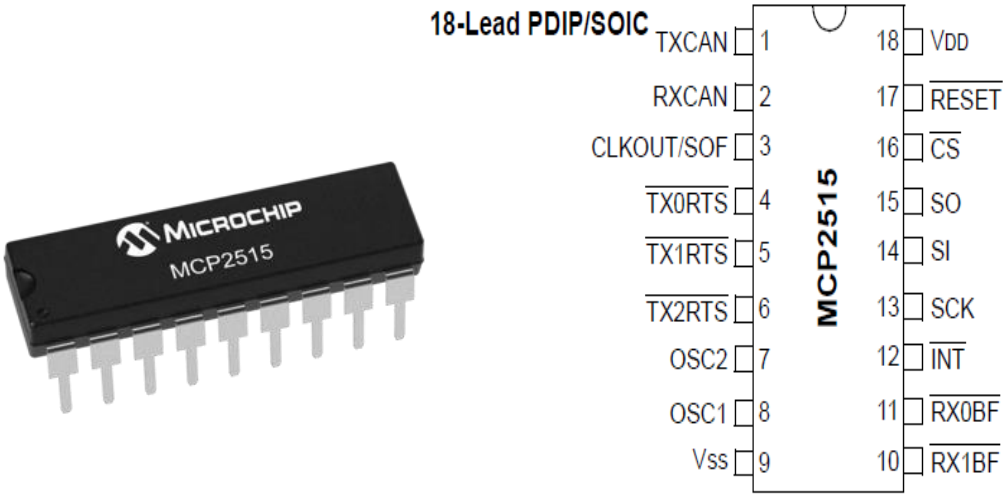


Figura 4.7. MCP2515

Name	PDIP/ SOIC Pin #	TSSOP Pin #	QFN Pin #	I/O/P Type	Description	Alternate Pin Function
TXCAN	1	1	19	O	Transmit output pin to CAN bus	—
RXCAN	2	2	20	I	Receive input pin from CAN bus	—
CLKOUT	3	3	1	O	Clock output pin with programmable prescaler	Start-of-Frame signal
$\overline{TX0RTS}$	4	4	2	I	Transmit buffer TXB0 request-to-send. 100 kΩ internal pull-up to V_{DD}	General purpose digital input. 100 kΩ internal pull-up to V_{DD}
$\overline{TX1RTS}$	5	5	3	I	Transmit buffer TXB1 request-to-send. 100 kΩ internal pull-up to V_{DD}	General purpose digital input. 100 kΩ internal pull-up to V_{DD}
$\overline{TX2RTS}$	6	7	5	I	Transmit buffer TXB2 request-to-send. 100 kΩ internal pull-up to V_{DD}	General purpose digital input. 100 kΩ internal pull-up to V_{DD}
OSC2	7	8	6	O	Oscillator output	—
OSC1	8	9	7	I	Oscillator input	External clock input
Vss	9	10	8	P	Ground reference for logic and I/O pins	—
$\overline{RX1BF}$	10	11	9	O	Receive buffer RXB1 interrupt pin or general purpose digital output	General purpose digital output
$\overline{RX0BF}$	11	12	10	O	Receive buffer RXB0 interrupt pin or general purpose digital output	General purpose digital output
$\overline{INT}$	12	13	11	O	Interrupt output pin	—
SCK	13	14	12	I	Clock input pin for SPI interface	—
SI	14	16	14	I	Data input pin for SPI interface	—
SO	15	17	15	O	Data output pin for SPI interface	—
$\overline{CS}$	16	18	16	I	Chip select input pin for SPI interface	—
$\overline{RESET}$	17	19	17	I	Active-low device Reset input	—
VDD	18	20	18	P	Positive supply for logic and I/O pins	—
NC	—	6,15	4,13	—	No internal connection	—

Note: Type Identification: I = Input; O = Output; P = Power

Tabla 4.2. Definición pines MCP2515

El MCP2515 utiliza una comunicación basada en el protocolo SPI, por lo que la conexión de pines entre este módulo y Arduino Nano será la siguiente:

- Chip Select (CS): 10
- Slave Input (SI): 11
- Slave Output (SO): 12
- Serial Clock (SCK): 13

En la Figura 4.8 se muestra el esquema de conexión.

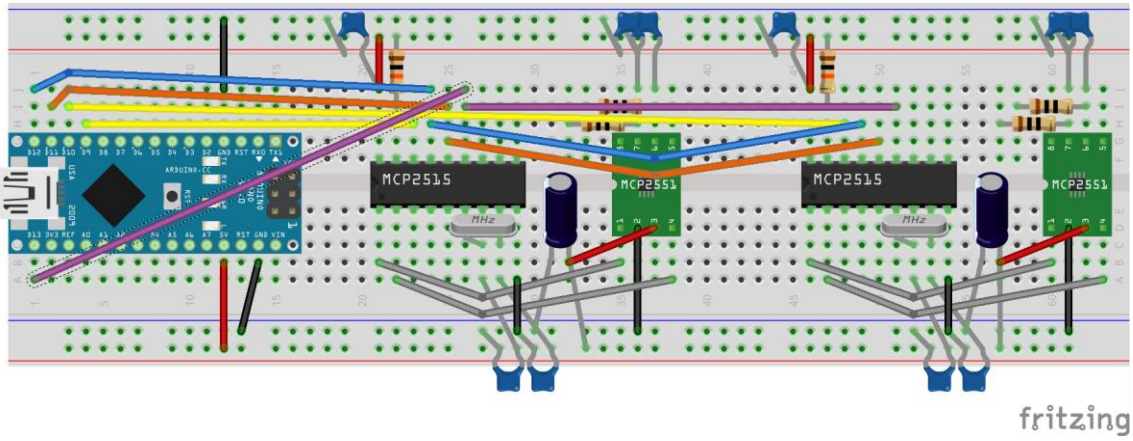


Figura 4.8. Esquema Fritzing MCPs - Arduino

Módulo SD

Nos va a permitir introducir una tarjeta de memoria flash SD para llevar a cabo una comunicación entre Arduino y SD. El propósito es el siguiente, una vez la trama CAN es capturada y etiquetada temporalmente se hará una transferencia de la información a un fichero de la SD.

Para el diseño del datalogger se utiliza el módulo Digilent Pmod SD [17], el cual presenta las siguientes características:

- No tiene limitación para el tamaño de la SD.
- Puerto Pmod de 12 pines con interfaz SPI.

Pmod SD utiliza una comunicación basada en el protocolo SPI, por lo que la conexión de pines entre este módulo y Arduino Nano será la siguiente:

- Chip Select (CS): 10
- Master Output Slave Input (MOSI): 11
- Master Input Slave Output (MISO): 12
- Serial Clock (SCK): 13

En la Figura 4.11 se muestra el esquema de conexión.

Figura 4.9. Módulo SD Pmod

Pin	Signal	Description
1	~CS	Chip Select / Data3
2	MOSI	MOSI / Command
3	MISO	MISO / Data0
4	SCK	Serial Clock
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V)
7	DAT1	Data1
8	DAT2	Data 2
9	CD	Card Detect
10	WP	Write Protect
11	GND	Power Supply Ground
12	VCC	Power Supply (3.3V)

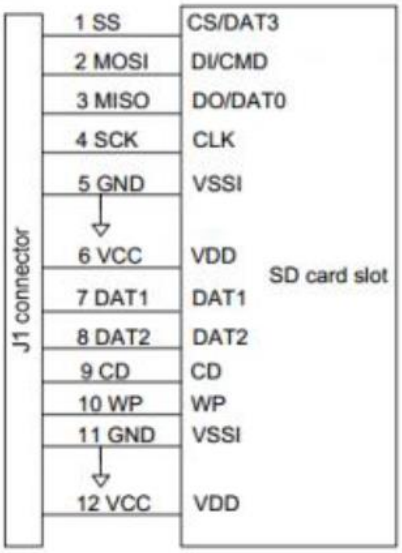


Figura 4.10. Definición pines módulo SD

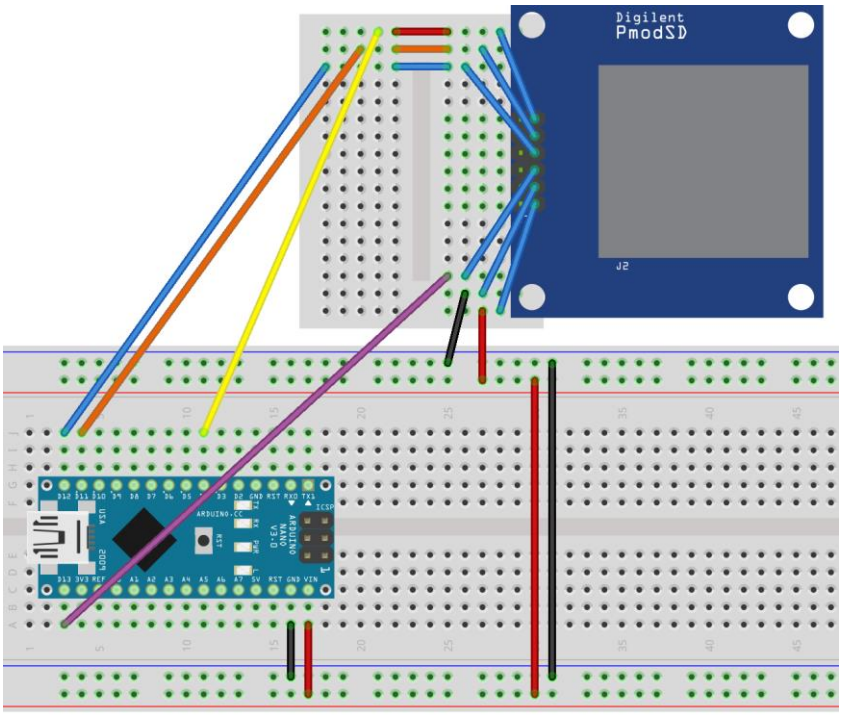


Figura 4.11. Esquema Fritzing SD - Arduino

Hardware final

Además de los módulos anteriormente descritos se añaden componentes electrónicos básicos para el correcto funcionamiento de la placa. Se describen los más significativos.

Se añaden un conjunto de resistencias y condensadores. Primeramente, para la reducción del ruido observado en las líneas CANH y CANL del bus CAN del Captur. Además, se añade una resistencia de pull-down a tierra en las líneas MISO y MOSI de los MCP2515 ya que se observaba por medio del osciloscopio que cuando los MCP transmitían 0V en las líneas de datos no era capaz de llegar a ese valor. De esta manera se evita posibles fallos de interpretación en la comunicación.

Además, MCP2515 necesita un oscilador de cristal para su reloj interno, el cual admite varias frecuencias de funcionamiento. En nuestro caso se ha escogido una frecuencia de 20 MHz, ya que es suficiente para soportar la tasa de transferencias del bus (500kbps).

También es necesario un par de conectores DB9 (uno por cada canal CAN del Captur).

Con estos agregados y los módulos anteriormente comentados se completa el diseño de la placa.

En la Figura 4.12 se muestra el diseño finalizado. Además, en los anexos de este trabajo se incluye el diseño electrónico en el programa Proteus.

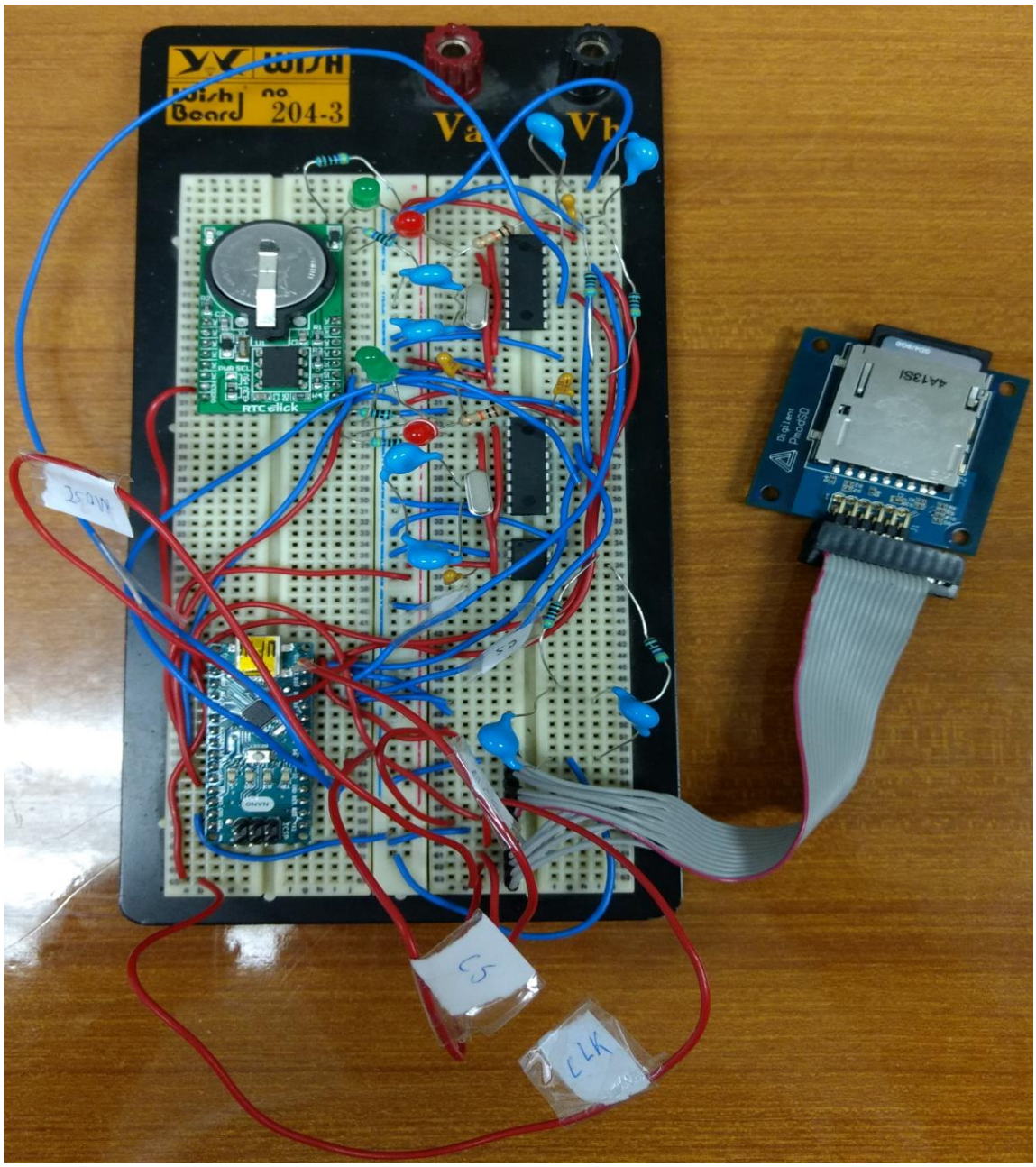


Figura 4.12. Aspecto final hardware Datalogger con un arduino

4.3. DISEÑO SOFTWARE

Este capítulo se ha separado en subsecciones para poder explicar en detalle el código software que se ha pensado que debía ser comentado brevemente, ya sean de las librerías utilizadas, de los mecanismos que se han seguido o algunas de las funciones que hace el código. Se comenzará comentando como se ha logrado desarrollar en código la creación/nombramiento del fichero SD y el formato de éste. Después, se hablará de las interrupciones, un mecanismo muy conocido que implementan algunos microcontroladores y que será de gran uso a lo hora de hacer las capturas de tramas. Para finalizar el capítulo de diseño software se nombran y en algunos casos se detallan las librerías software del proyecto.

Creación y nombramiento del fichero inicial en la SD

Nuestro sistema debe de ser capaz de crear un nuevo fichero en la SD cada vez que se quiera hacer una nueva captura de datos. Para ello, no nos basta con que se dé un nombre aleatorio al fichero porque a la hora de encontrarlo en la SD sería muy incómodo estar buscando la fecha de creación en las cabeceras de los diferentes ficheros. Por lo que se decide que se debe de seguir algún patrón de nombramiento para el fichero.

En nuestro datalogger el nombre de los ficheros tendrá la siguiente estructura o patrón: AAMMDDXX.LOG

Donde AA corresponde al año, MM al mes y DD al día actuales. XX será un número en modo de secuencia que irá de 00 hasta 99 para el caso de que se hagan varias capturas en el mismo día.

A continuación, se muestra como se ha desarrollado el código para implementarlo:

```
char nameFile[]="00000000.LOG"; // Array para almacenar el nombre del fichero, se debe de inicializar.

/*
 * Función que obtiene el nombre del fichero que se va a crear para el datalogger
 * nameFile: se pasa la dirección de memoria de nameFile para que sea modificado su valor
 */
void getFilename(char *nameFile){

  p.get_time();

  nameFile[0]=(p.year-2000)/10 + '0';
  nameFile[1]=p.year%10 + '0';
  nameFile[2]=p.month/10 + '0';
  nameFile[3]=p.month%10 + '0';
  nameFile[4]=p.day/10 + '0';
  nameFile[5]=p.day%10 + '0';
  nameFile[6]=0+ '0';
  nameFile[7]=0+ '0';
  nameFile[8]='.';
  nameFile[9]='L';
  nameFile[10]='O';
  nameFile[11]='G';

  //Se comprueba si existe ya el fichero (mismo dia) y si es así se modifica el nombre(00-99)
  unsigned int i = 0;
  while(SD.exists(nameFile))
      i++;
      nameFile[6] = i/10 + '0';

      nameFile[7] = i%10 + '0';
      nameFile[8]='.';
      nameFile[9]='L';
      nameFile[10]='O';
      nameFile[11]='G';
  }
  return;
}
```

Formato del fichero datalogger

Como ya se comentó en la descripción del funcionamiento del *datalogger* de un arduino, se va utilizar el mismo formato de fichero que usan programas como BUSMASTER y CANoe. De esta manera, una vez se tenga el fichero resultante podremos hacer una reproducción *offline* en estos programas.

Para ello únicamente basta con fijarse la cabecera que introducen los programas en sus ficheros de log y copiarlos para nuestro *datalogger*. Se muestra el código que genera y escribe la cabecera del fichero.

```
myFile = SD.open(nameFile, FILE_WRITE);

// Si el fichero se ha abierto correctamente se escribe en el.
if (myFile) {

  p.get_time();
  char time1[50];
  sprintf(time1, "***START DATE AND TIME %02d:%02d:%02d %02d:%02d:%02d:%03lu***", p.day, p.month, p.year, p.hour, p.minute, p.second,millis());
  Serial.print(F("Writing to logger.log..."));
  myFile.println(F("***BUSMASTER Ver 3.1.0***"));
  myFile.println(F("***PROTOCOL CAN***"));
  myFile.println(F("***NOTE: PLEASE DO NOT EDIT THIS DOCUMENT***"));
  myFile.println(F("***[START LOGGING SESSION]***"));
  myFile.println(time1);
  myFile.println(F("***HEX***"));
  myFile.println(F("***SYSTEM MODE***"));
  myFile.println(F("***START CHANNEL BAUD RATE***"));
  myFile.println(F("***CHANNEL 1 - Vector - CANcaseXL Channel 1 SN - 23653 - 500000 bps***"));
  myFile.println(F("***CHANNEL 2 - Vector - CANcaseXL Channel 2 SN - 23653 - 500000 bps***"));
  myFile.println(F("***END CHANNEL BAUD RATE***"));
  myFile.println(F("***START DATABASE FILES***"));
  myFile.println(F("***END DATABASE FILES***"));
  myFile.println(F("***<Time><Tx/Rx><Channel><CAN ID><Type><DLC><DataBytes>***"));
  // cerramos el fichero
  myFile.close();
  Serial.println("Cabecera escrita");


} else {
  // Si el fichero no se ha abierto notificamos el error.
  Serial.println(F("error opening logger.txt"));
}
```

En el fichero aparece tal y como se ve en la Figura 4.13.

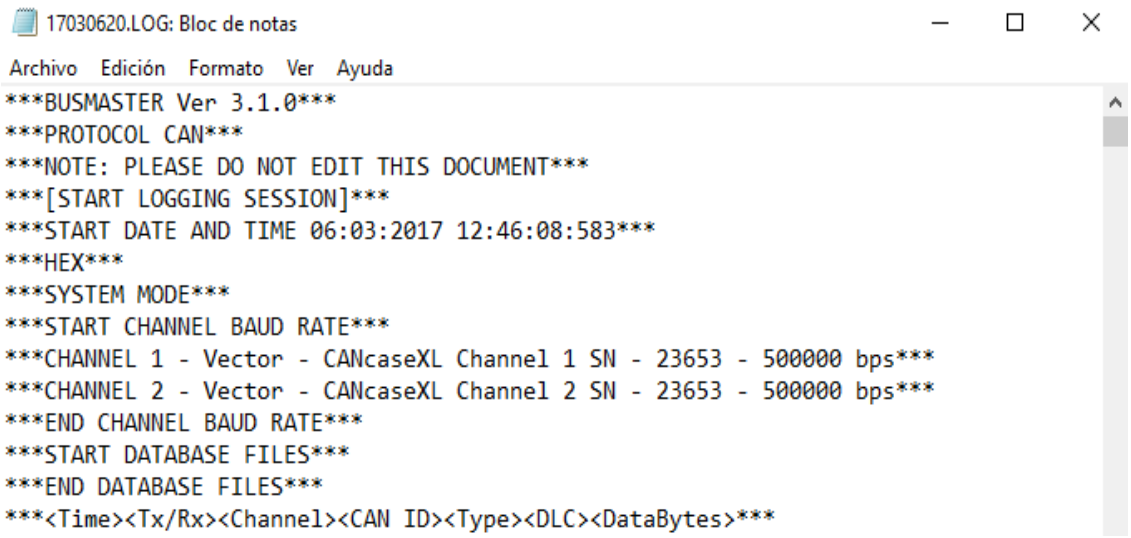


Figura 4.13. Cabecera de nuestro archivo en la SD

Interrupciones

Las interrupciones son un mecanismo muy potente y valioso en procesadores y autómatas. Pueden definirse como una señal que interrumpe la actividad normal de nuestro microprocesador y salta a atenderla, ejecutándose así una función especial llamada ISR (Interruption Service Rutine), para después de finalizada la interrupción volver de forma ordenada al lugar donde se produjo esta interrupción. [18]

Hay tres eventos que pueden disparar una interrupción:

- Un evento hardware, previamente definido.
- Un evento programado, o *Timer*.
- Una llamada por software.

Interrupciones en Arduino

Arduino dispone de dos tipos de eventos en los que definir interrupciones. Por un lado, se tiene las interrupciones de *timers*. Por otro lado, tenemos las interrupciones de hardware, que responden a eventos ocurridos en ciertos pines físicos [18].

Dentro de las interrupciones de hardware Arduino es capaz de detectar los siguientes eventos:

- RISING, ocurre en el flanco de bajada de LOW a HIGH.
- FALLING, ocurre en el flanco de subida de HIGH a LOW.
- CHANGING, ocurre cuando el pin cambia de estado (rising + falling).
- LOW, se ejecuta continuamente mientras está en estado LOW.

Los pines susceptibles de generar interrupciones varían en función del modelo de Arduino. En Arduino Nano se dispone de dos interrupciones, 0 y 1, asociados a los pines digitales 2 y 3.

La función software a la que salta Arduino tiene que ser una función que ni recibe y ni devuelve nada por definición. Al diseñar una ISR se debe mantener como objetivo que el código tenga el menor tiempo de ejecución posible, dado que mientras se esté ejecutando el bucle principal y todo el resto de funciones se encuentran detenidas.

Para definir una interrupción en Arduino se añade dentro de la estructura setup() la siguiente sentencia:

```
attachInterrupt(digitalPinToInterrupt(2), ISR, mode)
```

donde en *digitalPinToInterrupt* se indica el pin que se usa para capturar la interrupción, ISR será la función a la que salta la interrupción y *mode* el tipo de evento que se quiere capturar.

Interrupciones en MCP2515

MCP2515 dispone de un pin de salida llamado INT que se activa cuando una trama CAN es cargada en uno de sus dos búferes. A su vez el registro CANINTF.RXnIF del controlador se pone a uno. Cuando el microcontrolador destinatario lee del MCP2515 la información este registro vuelve a su valor por defecto cero.[15]

¿Por qué es interesante usar las interrupciones en nuestro programa? De esta manera se evita estar continuamente consultando el registro del MCP2515 para comprobar si ha llegado una nueva trama CAN o no. Además el retardo de una interrupción es mucho menor que el retardo en consultar los registros del controlador.

El funcionamiento de la interrupción en nuestro programa es el siguiente: El MCP2515 recibe una trama procedente del bus CAN del Captur. Una vez es almacenada en uno de sus búferes el MCP2515 activa su pin INT. Como consecuencia de esta bajada de tensión (el pin INT es inverso, se mantiene en alta tensión si no sucede nada), la gestión de interrupciones de Arduino lo captura en su pin de interrupción y pasa a ejecutar la función *miFuncion()* (función ISR), la cual únicamente modifica el valor de una constante/bandera llamada *myFlag* a 1. Lo que sucede es que en la siguiente iteración del bucle *loop()* nuestro programa detectará que el flag o constante está activado, procediendo a hacer la captura de la trama, etiquetado de tiempo para formar el mensaje y escritura en el fichero de la SD.

Librerías

Librería SD.h

Permite leer y escribir en tarjetas SD. Para ello la librería se encarga de las comunicaciones con la SD utilizando SPI. Admite los sistemas de archivos FAT16 y FAT32 en tarjetas estándar SD y tarjetas SDHC.

Esta librería está incluida en el propio Arduino por defecto. Más información se puede encontrar en la referencia [19]

Librería SDFat.h

Librería no oficial que proporciona acceso de lectura/escritura a las tarjetas SD. Se trata de una evolución de la librería anterior. Se han optimizado sus funciones y además es más configurable que la anterior. Por ejemplo, para la comunicación SPI con la SD se pueden escoger entre tres velocidades: *SPI_FULL_SPEED*, *SPI_HALF_SPEED* y *SPI_LOW_SPEED*. Más información se puede encontrar en la referencia [20]

Librería SPI.h

Le permite a Arduino comunicarse con dispositivos SPI, con éste como dispositivo maestro. Son posibles las siguientes configuraciones por medio del parámetro *SPISettings*:

- Velocidad máxima.
- Bit más significativo primero o bit menos significativo. MSBFIRST o LSBFIRST.
- Polaridad y fase de la señal de reloj.

Más información se puede encontrar en la referencia [8]

Librería Wire.h

Le permite a Arduino comunicarse con dispositivos I2C, con éste como dispositivo maestro. Más información se puede encontrar en la referencia [10].

Librerías mcp_can.h y mcp_can_dfs.h

Lleva toda la configuración del MCP2515 y aporta muchas funciones de lectura/escritura del bus CAN.

Permite las siguientes configuraciones y/o aplicaciones:

- Lectura y escritura de tramas CAN. Además, recoge los campos de la trama para devolver el ID de la trama, longitud y campo de datos.
- Especificar pin CS de SPI llamando al objeto MCP_CAN.
- En la inicialización del MCP2515 se puede especificar la velocidad de reloj de funcionamiento (en nuestro caso al tener un oscilador de cristal de 20MHz elegiremos esa frecuencia como la de reloj) y la velocidad del bus CAN (en nuestro caso 500Kbps).
- Resetear a nivel software el MCP2515.
- Leer y modificar registros del MCP2515 a partir de una dirección.
- Leer el estado en el que se encuentra el MCP2515.

- Escribir máscaras y filtros.
- Detección y gestión de errores.
- Escoger el modo de funcionamiento del MCP2515. Estos modos son [15]:
    - MCP_NORMAL: modo de funcionamiento normal en el que el MCP2515 envía confirmaciones de mensajes (ACK) para los datos recibidos, errores de trama, etc. Es el único modo en el cual MCP2515 envía datos al bus CAN.
    - MCP_SLEEP: minimiza el consumo de corriente del dispositivo. La interfaz SPI permanece activa para la lectura de mensajes.
    - MCP_LOOPBACK: se usa para sistemas de desarrollo o pruebas. Permite la transmisión interna de mensajes desde el bufer de transmisión al de recepción
    - MCP_LISTENONLY: modo de sólo escucha para recibir todos los mensajes (incluidos mensajes con errores). El uso de este modo puede utilizarse para medir velocidades.

Los modos de operaciones se solicitan poniendo los bits del registro (REQOP<2:0>) en el registro de control CAN (CiCON<26:24). El módulo CAN reconoce la entrada en el modo solicitado por los bits (CiCON<23:21>) del modo de operación (OPMOD<2:0>).

Más información se puede encontrar en la referencia [21]

Librería PCF8583.h

Implementa una interfaz simple para facilitar al Arduino los datos del RTC. Más información se puede encontrar en la referencia [22].

4.4. TESTEO Y LIMITACIONES

Análisis de retardos del programa inicial

En la primera versión de código y observando los resultados del fichero se ve el siguiente problema: aparecen retardos muy altos entre mensajes, en torno a 500 ms entre cada uno de ellos.

Para conocer más sobre el problema se recurre al osciloscopio, visualizando las diferentes líneas de comunicaciones que tiene Arduino con los módulos electrónicos. Se detecta que el problema sucede en la comunicación con la SD.

Gestión de la SD

Se decide buscar información acerca de las comunicaciones entre Arduino y SD, también sobre la biblioteca *SD.h*, además de revisar el código interno de la librería para comprender su funcionamiento [19][20]. El estudio de las diferentes referencias, [23][25], nos lleva a las siguientes conclusiones:

- Se deben usar los formatos de tabla de asignación de archivos FAT16 o FAT32.
- Asegurarse de cerrar el archivo en el que se está escribiendo ya que si no pueden ocurrir pérdidas de datos.
- Cada lectura o escritura física de datos en la SD supone hacerlo con un bloque de 512 bytes. Tanto la librería *SD.h* como *SDFat.h* disponen de un buffer interno (pila) de ese tamaño. Cuando escribimos en la SD realmente se está enviando información al buffer y actualizando el objeto File. Este proceso, al hacerse exclusivamente en la RAM, apenas consume tiempo. La transferencia real de datos físicamente se hace cuando la escritura excede el ámbito del bloque actual, la instrucción lleva implícito una instrucción *flush()* o se hace un *flush()* manual o un *close()*. Todo esto se hace de forma transparente al usuario. En definitiva, esta librería para Arduino Nano resulta poco eficaz, ya que este modelo de Arduino sólo dispone de una SRAM de 2KB (memoria donde el programa almacena y manipular variables), por lo que con la reserva de 512 bytes se está consumiendo un cuarto de memoria, dejándonos poco margen de maniobra en lo que refiere al código. En la Figura 4.14 se puede ver la parte del código de la función *write()* de la librería *SD.h* donde se hace la reserva de los 512 bytes.

```
1170      // max space in block
1171      uint16_t n = 512 - blockOffset;
1172
1173      // lesser of space and amount to write
1174      if (n > nToWrite) n = nToWrite;
1175
1176      // block for data write
1177      uint32_t block = vol_->clusterStartBlock(curCluster_) + blockOfCluster;
1178      if (n == 512) {
1179        // full block - don't need to use cache
1180        // invalidate cache if block is in cache
1181        if (SdVolume::cacheBlockNumber_ == block) {
1182          SdVolume::cacheBlockNumber_ = 0XFFFFFFFF;
1183        }
1184        if (!vol_->writeBlock(block, src)) goto writeErrorReturn;
1185        src += 512;
1186      } else {
1187        if (blockOffset == 0 && curPosition_ >= fileSize_) {
1188          // start of new block don't need to read into cache
1189          if (!SdVolume::cacheFlush()) goto writeErrorReturn;
1190          SdVolume::cacheBlockNumber_ = block;
1191          SdVolume::cacheSetDirty();
1192        } else {
1193          // rewrite part of block
1194          if (!SdVolume::cacheRawBlock(block, SdVolume::CACHE_FOR_WRITE)) {
1195            goto writeErrorReturn;
1196          }
1197        }
```

Figura 4.14. Código de la librería SD.h donde se efectúa la reserva de los 512 bytes.

- Es posible forzar la escritura de datos aunque no se tengan 512 bytes preparados para la escritura. Se podrá forzar con las funciones de cierre de fichero (*close()*) o de *flush()*.
- Los cierres de ficheros (*close()*) consumen mucho tiempo.

Con esta información se reescribe el código del programa para optimizar los tiempos de retardo. Lo primero es eliminar la llamada a la función *close()* que se hacía en cada bucle (*loop()*). Sin embargo, cuando se quiera acabar el programa el fichero debe de ser cerrado. Para ello se añade un pulsador hardware al sistema para cuando se quiera finalizar la captura. Una vez pulsado se realiza el cierre de fichero.

Con este cambio se ve una gran mejora, teniendo ahora un tiempo entre mensajes de 2 a 4 milisegundos. También se puede observar en el fichero que aparecen 9 mensajes consecutivos con tiempos entre ellos similares, pero el siguiente mensaje aparece con un retardo variable (de media 5 milisegundos). Esto ocurre porque los 9 mensajes corresponden a la capacidad de la pila de la librería (512 bytes) que una vez

completada se transfiere a la SD. Estos bytes son transmitidos por SPI a una velocidad de 820Kbps (512 bytes en 5ms), siendo esta velocidad variable. Nos encontramos entonces con el siguiente problema, el Arduino mientras sucede la comunicación con la tarjeta SD no puede ocupar el bus SPI para recoger las tramas del MCP.

En la Figura 4.15 se ve la captura en el osciloscopio de la transmisión de estos bytes a la SD, teniendo en la parte de arriba de la pantalla la línea MOSI de SPI y en la parte inferior la línea CS (Chip Select) correspondiente al esclavo SD, se observa claramente en el inferior de la pantalla con los cursores de tiempo que la transmisión completa a la SD ha ocupado 4.4 ms.

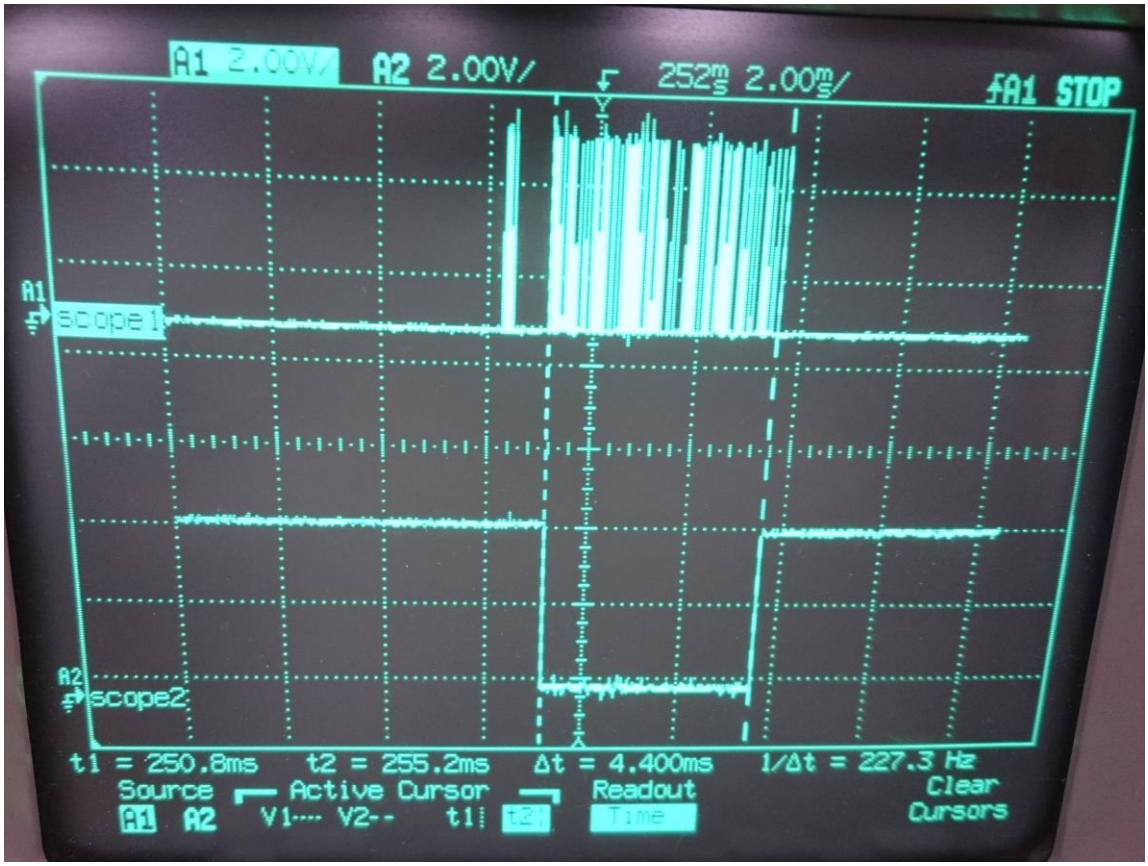


Figura 4.15. Captura osciloscopio transmisión SPI entre Arduino y SD

Para subsanar en la medida de lo posible este retardo en el que el *datalogger* pierde tramas se cambia la librería *SD.h* que Arduino tiene por defecto por la librería optimizada *SdFat.h*. Además, se configura SPI a la máxima velocidad. Como resultado se obtiene que la transferencia del bloque de 512 bytes a la SD sucede en una media de 3-4ms.

Análisis de los retardos del programa final

En modo de ejemplo en la Figura 4.16 se muestra el inicio del fichero de salida de nuestro datalogger. Se ha capturado directamente sobre la maqueta Renault.

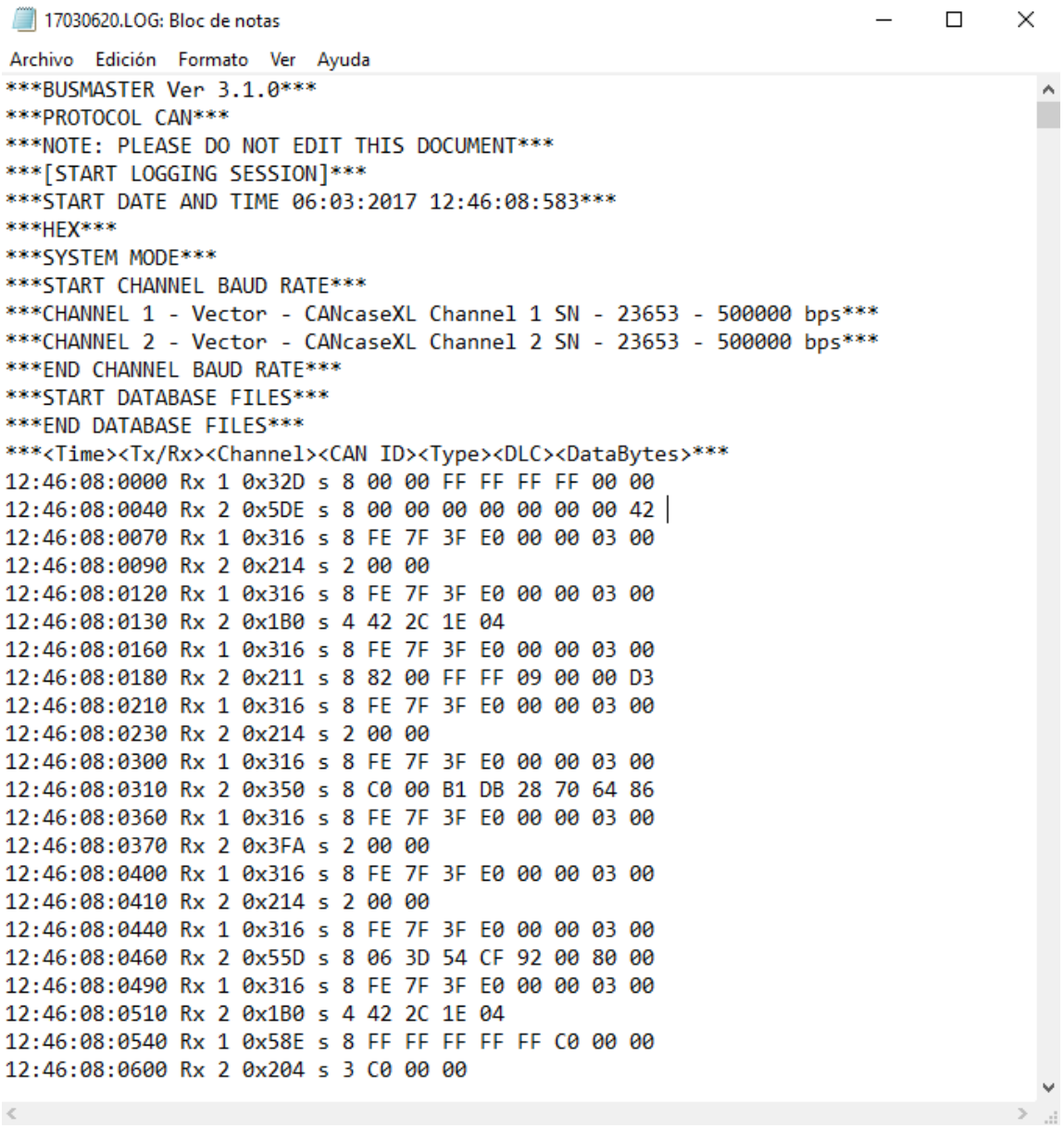


Figura 4.16. Archivo de salida de nuestro datalogger con un arduino

Se calculan los tiempos de retardos sucedidos en cada parte de nuestro programa para poder analizar si se puede optimizar aún más cada proceso. Estos son los tiempos que dedica cada función destacable del programa:

- 0,2ms<t<1ms: Arduino Nano solicita y recibe por I2C al RTC la hora, minutos y segundos actuales.
- 1ms<t<2ms: Arduino Nano lee por SPI el bufer del MCP2515 la trama CAN y la almacena en una variable del programa.
- 2ms<t<5ms: Arduino Nano transmite por SPI 512 bytes a la SD.

CAPÍTULO V: SEGUNDO PROTOTIPO CON DOS ARDUINOS

5.1. FUNCIONAMIENTO

Como se ha visto en el capítulo anterior, el prototipo datalogger con un arduino se ve limitado con las transmisiones SPI a la tarjeta SD, haciendo que éste quede bloqueado durante alrededor de 4 ms, tiempo en el que varias tramas viajan por el bus CAN sin ser capturadas por nuestro sistema.

Para intentar solventar este problema, sin tener que cambiar el material del que se dispone, se necesita de alguna manera separar los esclavos (MCP2515 y SD) dentro del bus SPI pare evitar la circunstancia en la que si alguno desea transmitir no se encuentre el bus ocupado. Para ello se decide probar un nuevo prototipo que hará trabajar dos unidades Arduino en vez de una sola para así poder disponer de dos buses SPI y poder separar las comunicaciones. Se propusieron dos ideas para el prototipo, que se describen a continuación.

Primera propuesta: Dos Arduinos Nano en serie

Al disponer cada Arduino de una interfaz SPI se puede lograr que el MCP2515 esté en un primer bus y la SD en un segundo bus.

Llamemos al primer Arduino A y al segundo B. De esta manera, A estará conectado físicamente con el MCP2515, encargándose de capturar las tramas CAN. Por otro lado, B estará conectado con la SD, siendo su función realizar las escrituras de datos.

Para que este modelo sea realizable es necesario que A transfiera los datos de las tramas a B por una interfaz que no sea SPI para así evitar colisiones. Además, para que no se produzca un cuello de botella en el enlace de A hasta B, la tasa de transferencia de este enlace debe de ser igual o mayor a la tasa de SPI.

Las diferentes posibilidades para la transferencia de A a B son:

- UART: Puerto serie con velocidad máxima de 2Mbps.
- I2C: Su velocidad máxima en Arduino está limitada a 100Kbps.
- SPI artificial: Se trata de diseñar por software el protocolo SPI en otros pines de Arduino Nano. Nunca logrará ser igual de rápido que SPI por hardware.

El protocolo SPI alcanza en Arduino velocidades de hasta 8Mbps teóricos. Para el caso práctico se vieron varias transferencias de datos con ayuda del osciloscopio, alcanzándose hasta 5 Mbps. Por lo que, desgraciadamente este prototipo de dos arduinos en serie no nos sirve por darse el problema del cuello de botella en el enlace entre ambos.

Segunda propuesta: dos Arduinos Nano en paralelo

De nuevo se dispone de dos interfaces SPI para poder separar a cada esclavo SPI en un bus individual.

El funcionamiento es el siguiente: MCP2515 se inicia en el bus SPI 1. Por su parte se inicia la SD en el bus SPI 2. Cada Arduino Nano (A y B) tendrán asignadas alternativamente la tarea de capturar las tramas CAN y almacenarlas en un buffer interno (conexión con bus 1) y la tarea de hacer una transferencia física de datos a la SD (conexión con bus 2). Entiéndase que mientras uno de los Arduino ocupa el bus SPI 1 el otro Arduino ocupará el bus SPI 2 y así intercambiándose los buses, evitándose así que alguno de los módulos SPI se encuentre el bus SPI ocupado.

Nótese que para que este prototipo cumpla con lo explicado, el tiempo que se dedica a capturar las tramas debe de ser mayor al tiempo que se dedica a la escritura en la SD, si no habrá pérdida de tramas. Si se cumple, el Arduino que ha escrito en la SD está un tiempo esperando hasta que recoge el testigo de captura de tramas.

Se continúa el trabajo diseñando este datalogger de dos arduinos trabajando de forma paralela.

5.2. DISEÑO HARDWARE

Para implementar este nuevo prototipo es necesario cambiar el hardware del que se dispone hasta ahora, ya que en esta versión se encuentra un nuevo Arduino Nano, teniéndose los mismos módulos (MCPs, SD y RTC) que antes.

Para lograr que existan dos buses SPI independientes y que ambos sean accesibles por los dos maestros SPI (Arduino A y Arduino B) es necesario añadir al esquema hardware un conjunto de multiplexores. De esta manera uno de los dos Arduino puede manejar las entradas/salidas de los multiplexores para decidir en qué bus debe de estar conectado cada uno de los maestros en cualquier momento. Además, se tendrá que establecer un mecanismo de arbitrio entre los Arduinos para indicar que función tiene que realizar en cada momento cada uno de ellos. Esto implica que habrá un Arduino maestro y un Arduino esclavo.

Las líneas o señales que se deben multiplexar son las correspondientes a las líneas que posee cada bus SPI (SPI1 y SPI2) además de las interfaces SPI de cada Arduino (A1 y A2). En la Figura 5.1 se muestra de forma global la interconexión que debe seguir el multiplexor.

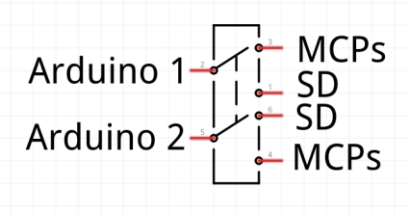


Figura 5.1 Esquema resumen multiplexor

En la Figura 5.2 se muestra como debe ser multiplexada cada línea del bus SPI. Se observa que hay cuatro multiplexores que permiten controlar todas las señales que se deben multiplexar, que son las de comunicación SPI: MISO, MOSI, CS y CLK.

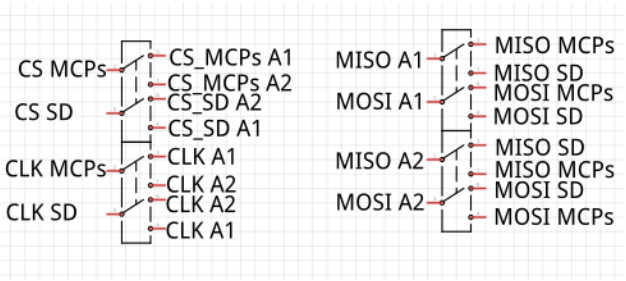


Figura 5.2 Esquema multiplexores en datalogger de dos arduinos

De esta manera se decide añadir tres multiplexores Quad 2:1 con referencia ADG774BRZ [24].

MUX Quad 2:1

Este elemento nos permite multiplexar parejas de entradas en una salida, hasta cuatro parejas. Para escoger que entrada será seleccionada únicamente utiliza una línea de direccionamiento.

En la Figura 5.3 y en la **¡Error! No se encuentra el origen de la referencia.** se muestran las especificaciones del multiplexor.

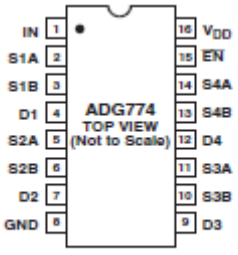


Figura 5.3 MUX ADG774

EN	IN	D1	D2	D3	D4	Function
1	X	Hi-Z	Hi-Z	Hi-Z	Hi-Z	DISABLE
0	0	S1A	S2A	S3A	S4A	IN = 0
0	1	S1B	S2B	S3B	S4B	IN = 1

Tabla 5.1. Tabla de la verdad MUX ADG774

Finalmente se muestra en la Figura 5.4 el diseño hardware finalizado.

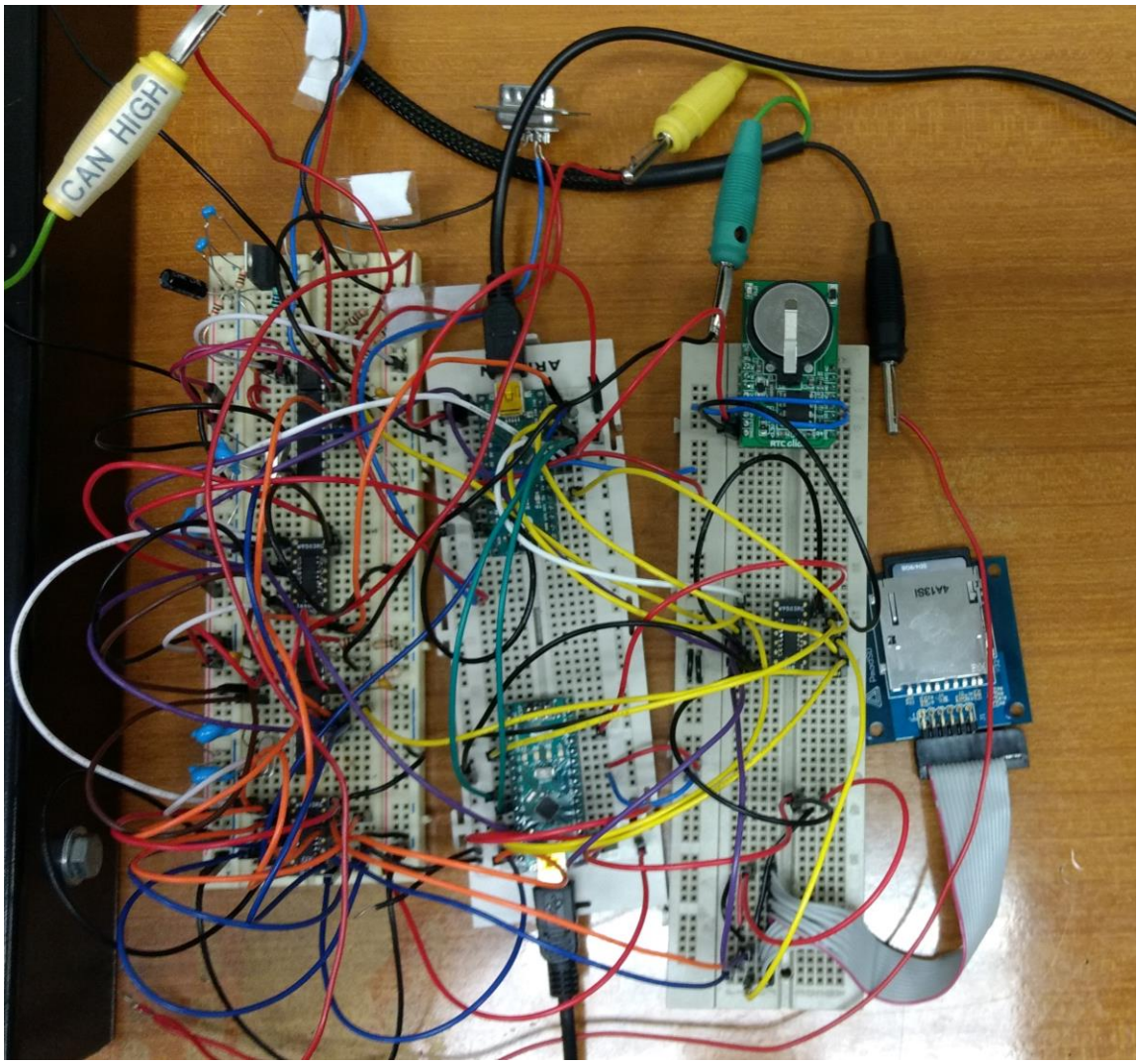


Figura 5.4. Aspecto final hardware Datalogger con dos arduinos

5.3. DISEÑO SOFTWARE

En lo que se refiere al código software también debe de ser adaptado al nuevo prototipo, teniéndose ahora dos programas que desarrollar (uno para cada Arduino, uno haciendo de maestro y otro de esclavo). Este nuevo software debe de realizar las siguientes tareas:

- Llevar a cabo la gestión de turnos para controlar en cada momento que bus SPI debe de estar conectado a cada Arduino.
- Almacenar en una pila interna de cada Arduino los mensajes que se van a grabar en la SD (tramas etiquetadas temporalmente).

Gestión de turnos

Para llevar a cabo el intercambio de buses SPI entre los Arduinos se han definido cuatro estados en nuestro programa. Para ello se han añadido dos cables de comunicación entre los Arduinos para notificar en el estado que se encuentra cada uno de ellos. El cambio entre estados se realizará pues con dos bits (Cm = estado maestro y Ce = estado esclavo), los cuales físicamente son los dos cables ya nombrados.

Se muestra en la Figura 5.5 se muestra la máquina de estados del programa.

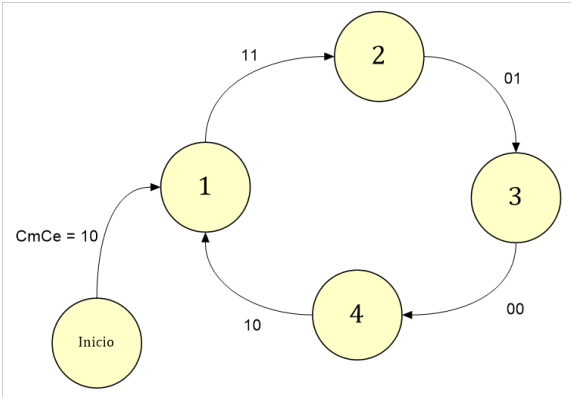


Figura 5.5 Máquina de estados del programa

Las funciones que se realizan en cada estado son las siguientes:

- Estado 1: Maestro recoge las tramas CAN y llena su pila. Esclavo escribe en la SD. Se pasa al Estado 2 cuando el esclavo finaliza la escritura (Ce = 1).

- Estado 2: Maestro cambia el multiplexor SPI. El esclavo se mantiene en espera. Se pasa al Estado 3 cuando el maestro ha cambiado el multiplexor (Cm = 0).

- Estado 3: Maestro escribe en la SD. Esclavo recoge las tramas CAN y llena su pila. Se pasa al Estado 4 cuando el maestro finaliza la escritura y a recibido la notificación del esclavo (Ce = 0).

- Estado 4: Maestro cambia el multiplexor SPI. El esclavo se mantiene en espera. Se pasa al Estado 1 cuando el maestro ha cambiado el multiplexor (Cm = 1).

Además se muestra en la Figura 5.6 el diagrama de flujo que se ha seguido en el Arduino nombrado como Master, el cual será el encargado de cambiar las entradas

de los multiplexores. Por su parte el Arduino nombrado como Esclavo se ha programado siguiendo el diagrama de flujo de la Figura 5.7.

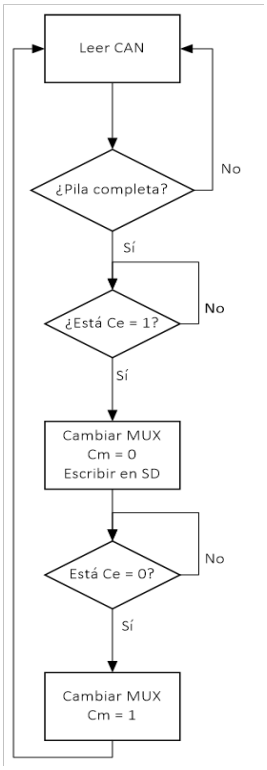


Figura 5.6 Diagrama de flujo para el Arduino Master

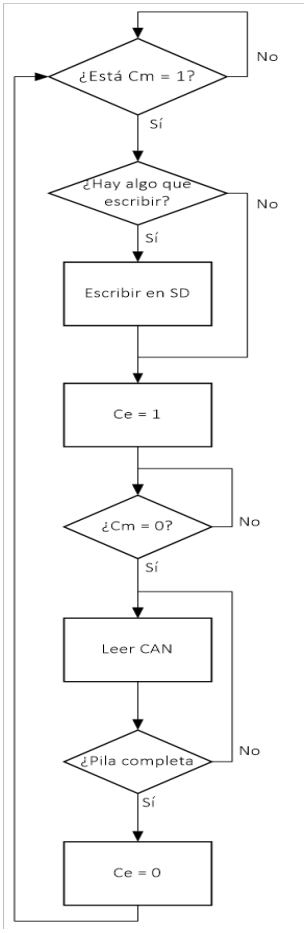


Figura 5.7 Diagrama de flujo para el Arduino Esclavo

Almacenamiento de los mensajes en la pila

A la hora de capturar las tramas CAN del bus SPI donde se encuentran los MCPs nos encontramos con que cada vez que el Arduino ha formado el mensaje a grabar éste no puede ser transmitido a la SD porque no está presente en su bus. La solución se encuentra en almacenar en una pila interna del Arduino un número de mensajes formados para su posterior transmisión cuando sea necesario. Ahora se sucede la siguiente pregunta, ¿Cuántos mensajes tienen que ser guardados en la pila?

Idealmente se tiene que construir un número de mensajes que haga que su total de bytes de información alcance los 512 bytes, para así a la hora de realizar la transferencia a la SD se llene al completo la pila interna de la biblioteca *SD.h*. Este método sería la forma más eficaz de realizar este problema, sin embargo, estamos limitados por la capacidad de la memoria SRAM del Arduino (2KB). Se recuerda que Arduino utiliza esta memoria para almacenar y manipular las variables del programa. La biblioteca mencionada ya utiliza 512 bytes y nosotros queremos reservar otros 512 bytes, además de necesitarse más espacio para otras variables del programa, por lo que se tiene que disminuir el número de mensajes a capturar-almacenar en cada ejecución o *loop()* del programa. [25]

Se ha calculado que cada mensaje formado por una trama y su etiquetado temporal ocupa 55 bytes. Considerando esto y que la memoria SRAM del Arduino no debe de

superar el 80% de su capacidad para que funcione de una manera estable se ha estimado que se pueden almacenar en la pila un total de 7 mensajes, sin contar las otras variables del programa, por lo que finalmente son 5 los mensajes que pueden almacenarse de cada vez.

A la hora de desarrollar este código se han encontrado varios contratiempos, como es la ocupación de las memorias SRAM de los Arduinos y los tiempos que emplea Arduino para construir los arrays internos donde se almacenan los mensajes CAN que se capturan en tiempo real.

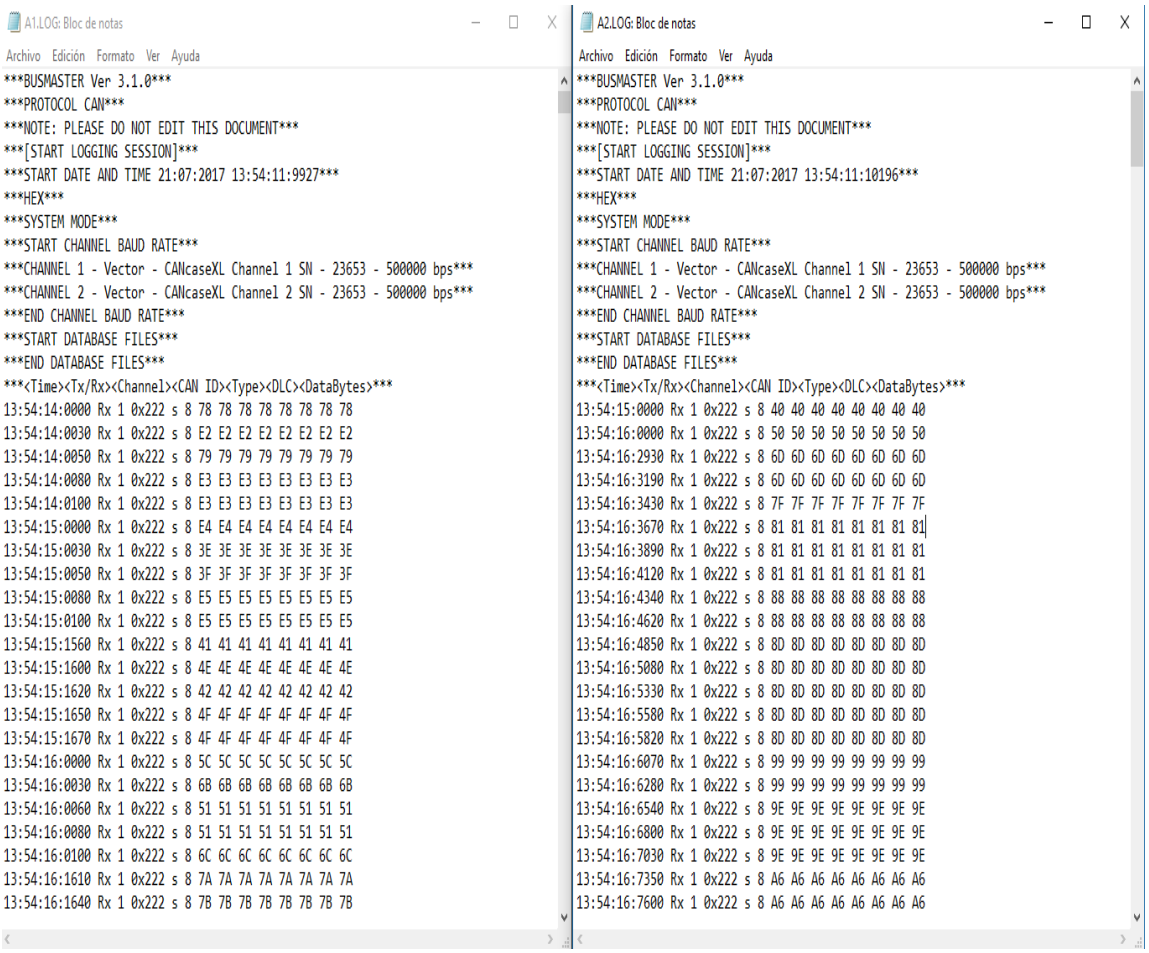
Para el problema de la ocupación de la memoria SRAM se ha necesitado optimizar el uso de ésta, liberando parte de su uso con varios mecanismos a partir de la información encontrada en [26].

Por su parte, para la construcción de los arrays se ha necesitado dividir cada mensaje CAN en dos variables, una para el etiquetado, ID y longitud de la trama y otra variable para los datos de la trama. La razón es la siguiente, obligatoriamente se deben de recoger los datos de la trama byte por byte y si solo usáramos una variable la concatenación permanente de esta variable en si misma ocupa un gran tiempo en Arduino, por lo que de esta manera con una variable o array de 8 bytes (datos trama CAN) la concatenación se hace mucho más rápida.

Análisis del programa final

Los resultados de los numerosos testeos que se han realizado nos arrojan muchos problemas en la SD ya que o un Arduino no es capaz de escribir en ella o se generan archivos extraños y muy pesados. Se piensa que este problema de archivos no deseados puede ser debido al momento en el que la SD es introducida en un sistema operativo Windows para su lectura, ya que el sistema introduce información en la SD la cual el Arduino posteriormente no es capaz de interpretar o gestionar.

A favor, se han logrado llevar a cabo varias pruebas con resultados favorables. Se consigue capturar las tramas con un intervalo de 2 milisegundos entre ellas, sin dejar pasar las tramas sin capturar mientras se hace la escritura en la SD como se buscaba. En la Figura 5.8 se muestra los archivos de salida de cada uno de los Arduinos, quedando en la parte izquierda el archivo del Arduino Maestro y en la parte derecha el archivo del Arduino Esclavo. Destacar que se ha necesitado disminuir el número inicial de tramas que se pretendía para la captura por cada turno por los problemas del tamaño de la memoria SRAM en los Arduino, realizándose una captura de tres tramas por cada vez en el Maestro y una trama para el Esclavo. Nótese que el objetivo de este prototipo se marcó en poder capturar una secuencia de 6-9 mensajes por cada turno y la diferencia entre mensajes fuera de 2 milisegundos.



A1.LOG: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
***BUSMASTER Ver 3.1.0***
***PROTOCOL CAN***
***NOTE: PLEASE DO NOT EDIT THIS DOCUMENT***
***[START LOGGING SESSION]***
***START DATE AND TIME 21:07:2017 13:54:11:9927***
***HEX***
***SYSTEM MODE***
***START CHANNEL BAUD RATE***
***CHANNEL 1 - Vector - CANcaseXL Channel 1 SN - 23653 - 500000 bps***
***CHANNEL 2 - Vector - CANcaseXL Channel 2 SN - 23653 - 500000 bps***
***END CHANNEL BAUD RATE***
***START DATABASE FILES***
***END DATABASE FILES***
***<Time><Tx/Rx><Channel><CAN ID><Type><DLC><DataBytes>***
13:54:14:0000 Rx 1 0x222 s 8 78 78 78 78 78 78 78 78
13:54:14:0030 Rx 1 0x222 s 8 E2 E2 E2 E2 E2 E2 E2 E2
13:54:14:0050 Rx 1 0x222 s 8 79 79 79 79 79 79 79 79
13:54:14:0080 Rx 1 0x222 s 8 E3 E3 E3 E3 E3 E3 E3 E3
13:54:14:0100 Rx 1 0x222 s 8 E3 E3 E3 E3 E3 E3 E3 E3
13:54:15:0000 Rx 1 0x222 s 8 E4 E4 E4 E4 E4 E4 E4 E4
13:54:15:0030 Rx 1 0x222 s 8 3E 3E 3E 3E 3E 3E 3E 3E
13:54:15:0050 Rx 1 0x222 s 8 3F 3F 3F 3F 3F 3F 3F 3F
13:54:15:0080 Rx 1 0x222 s 8 E5 E5 E5 E5 E5 E5 E5 E5
13:54:15:0100 Rx 1 0x222 s 8 E5 E5 E5 E5 E5 E5 E5 E5
13:54:15:1560 Rx 1 0x222 s 8 41 41 41 41 41 41 41 41
13:54:15:1600 Rx 1 0x222 s 8 4E 4E 4E 4E 4E 4E 4E 4E
13:54:15:1620 Rx 1 0x222 s 8 42 42 42 42 42 42 42 42
13:54:15:1650 Rx 1 0x222 s 8 4F 4F 4F 4F 4F 4F 4F 4F
13:54:15:1670 Rx 1 0x222 s 8 4F 4F 4F 4F 4F 4F 4F 4F
13:54:16:0000 Rx 1 0x222 s 8 5C 5C 5C 5C 5C 5C 5C 5C
13:54:16:0030 Rx 1 0x222 s 8 6B 6B 6B 6B 6B 6B 6B 6B
13:54:16:0060 Rx 1 0x222 s 8 51 51 51 51 51 51 51 51
13:54:16:0080 Rx 1 0x222 s 8 51 51 51 51 51 51 51 51
13:54:16:0100 Rx 1 0x222 s 8 6C 6C 6C 6C 6C 6C 6C 6C
13:54:16:1610 Rx 1 0x222 s 8 7A 7A 7A 7A 7A 7A 7A 7A
13:54:16:1640 Rx 1 0x222 s 8 7B 7B 7B 7B 7B 7B 7B 7B
```

A2.LOG: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
***BUSMASTER Ver 3.1.0***
***PROTOCOL CAN***
***NOTE: PLEASE DO NOT EDIT THIS DOCUMENT***
***[START LOGGING SESSION]***
***START DATE AND TIME 21:07:2017 13:54:11:10196***
***HEX***
***SYSTEM MODE***
***START CHANNEL BAUD RATE***
***CHANNEL 1 - Vector - CANcaseXL Channel 1 SN - 23653 - 500000 bps***
***CHANNEL 2 - Vector - CANcaseXL Channel 2 SN - 23653 - 500000 bps***
***END CHANNEL BAUD RATE***
***START DATABASE FILES***
***END DATABASE FILES***
***<Time><Tx/Rx><Channel><CAN ID><Type><DLC><DataBytes>***
13:54:15:0000 Rx 1 0x222 s 8 40 40 40 40 40 40 40 40
13:54:16:0000 Rx 1 0x222 s 8 50 50 50 50 50 50 50 50
13:54:16:2930 Rx 1 0x222 s 8 6D 6D 6D 6D 6D 6D 6D 6D
13:54:16:3190 Rx 1 0x222 s 8 6D 6D 6D 6D 6D 6D 6D 6D
13:54:16:3430 Rx 1 0x222 s 8 7F 7F 7F 7F 7F 7F 7F 7F
13:54:16:3670 Rx 1 0x222 s 8 81 81 81 81 81 81 81 81
13:54:16:3890 Rx 1 0x222 s 8 81 81 81 81 81 81 81 81
13:54:16:4120 Rx 1 0x222 s 8 81 81 81 81 81 81 81 81
13:54:16:4340 Rx 1 0x222 s 8 88 88 88 88 88 88 88 88
13:54:16:4620 Rx 1 0x222 s 8 88 88 88 88 88 88 88 88
13:54:16:4850 Rx 1 0x222 s 8 8D 8D 8D 8D 8D 8D 8D 8D
13:54:16:5080 Rx 1 0x222 s 8 8D 8D 8D 8D 8D 8D 8D 8D
13:54:16:5330 Rx 1 0x222 s 8 8D 8D 8D 8D 8D 8D 8D 8D
13:54:16:5580 Rx 1 0x222 s 8 8D 8D 8D 8D 8D 8D 8D 8D
13:54:16:5820 Rx 1 0x222 s 8 8D 8D 8D 8D 8D 8D 8D 8D
13:54:16:6070 Rx 1 0x222 s 8 99 99 99 99 99 99 99 99
13:54:16:6280 Rx 1 0x222 s 8 99 99 99 99 99 99 99 99
13:54:16:6540 Rx 1 0x222 s 8 9E 9E 9E 9E 9E 9E 9E 9E
13:54:16:6800 Rx 1 0x222 s 8 9E 9E 9E 9E 9E 9E 9E 9E
13:54:16:7030 Rx 1 0x222 s 8 9E 9E 9E 9E 9E 9E 9E 9E
13:54:16:7350 Rx 1 0x222 s 8 A6 A6 A6 A6 A6 A6 A6 A6
13:54:16:7600 Rx 1 0x222 s 8 A6 A6 A6 A6 A6 A6 A6 A6
```

Figura 5.8. Archivo de salida de nuestro datalogger con dos arduinos

CAPÍTULO VI: CONCLUSIONES

En este capítulo se presentan las conclusiones que se han alcanzado tras la realización de todo el trabajo, además de las mejoras y objetivos futuros que se piensan que se deberían de seguir para alcanzar los resultados deseados.

Con respecto al primer modelo del datalogger (un Arduino Nano):

- Se trata de un prototipo fiable y de un tamaño reducido.
- Se ha logrado que el sistema capture y almacene los datos del bus CAN de una manera transparente al bus.
- Existen limitaciones en las comunicaciones SPI entre el Arduino y la tarjeta SD que desembocan en una pérdida de tramas, haciendo que el modelo no cumpla con las especificaciones.

Se piensa que el sistema puede ser mejorado de varias maneras:

- Revisando y optimizando el código de las librerías SD.h y SdFat.h con el objetivo de aumentar la tasa de transferencia en la comunicación entre Arduino y tarjeta SD. Como ejemplo, se podría suprimir partes del código innecesarias para nuestro uso. También existen otras librerías similares de uso libre en Internet que pueden hacer que el sistema sea más veloz.
- Sustituyendo el modelo Arduino Nano por algún modelo de prestaciones superiores. Sería una solución relativamente sencilla ya que la programación de nuestro datalogger es compatible con todos los modelos Arduino. Por el contrario, nuestro datalogger se vería afectado en la especificación de ser un prototipo de tamaño reducido. También existen otros microcontroladores que se pueden adaptar al proyecto como es el caso de las placas pingüino, las cuales son compatibles con Arduino.
- Añadiendo otro Arduino Nano con el objetivo de agregar otro bus SPI al sistema y así tener cada esclavo (controladores CAN y tarjeta SD) separado en un bus propio. Es la idea que se siguió.

Con respecto al segundo modelo del datalogger (dos Arduino Nano):

- Se trata de un prototipo con dos Arduino trabajando conjuntamente para el sistema. Se logra que cada Arduino lleve a cabo las comunicaciones en

ambos buses SPI, sin interferirse, con multiplexores y con el diseño de mecanismos de gestión de turnos entre ambos.

- Se ha logrado tanto llevar a cabo una correcta gestión de turnos entre los Arduino como realizar algunos testeos correctos, sin embargo, han surgido errores tales como la aparición de archivos corruptos en la SD y problemas de capacidad de memoria en la SRAM de los Arduino

Se piensa que el sistema puede ser mejorado de dos maneras:

- Solucionando el error de archivos corruptos en la SD.

- Ampliando la memoria SRAM de ambos Arduino Nano con el objetivo de que cada Arduino sea capaz de almacenar en una pila interna 512 bytes para la transferencia posterior a la SD, ya que es el valor de bytes limitación de las librerías que se manejan y también el valor que hace una transferencia más limpia y eficaz. Para ello existen a la venta módulos en modo cabecera para Arduino.

- Sustituyendo el modelo Arduino Nano por algún otro modelo Arduino de prestaciones superiores.

En una línea futura y dado que se han encontrado problemas de implementación se podría revisar el hardware que utiliza el dispositivo CANcaseXL para orientarse en las prestaciones de nuestro datalogger. En concreto, el microcontrolador CAN que utiliza y su número de buffers y el tamaño de memoria del microprocesador principal. .

CAPÍTULO VII: REFERENCIAS Y GLOSARIO DE TÉRMINOS

[1] Documentación Manual CANcaseXL

https://vector.com/portal/medien/cmc/manuals/CANcaseXL_Manual_EN.pdf

[2] Arduino Nano. Consultado en Febrero de 2017.

https://www.arduino.cc/en/Guide/ArduinoNano

[3] TFG Sergio Sánchez Romero (2016). Construcción y testeo de una tabla de simulación de la arquitectura eléctrica del modelo Captur de Renault.

http://uvadoc.uva.es/bitstream/10324/20956/1/TFG-G%202268.pdf

[4] Arduino Software (IDE). Consultado en Marzo de 2017.

http://arduino.cc/en/Main/Software

[5] Buses de comunicaciones en Arduino. Consultado en Mayo 2017.

http://saber.patagoniatec.com/arduino-nano-328-arduino-atmega-clon-compatible-arduino-argentina-ptec

[6] Bus SPI en Arduino. Consultado en Mayo 2017.

https://aprendiendoarduino.wordpress.com/2016/11/13/bus-spi/

[7] Configuración bus SPI en Arduino. Consultado en Mayo 2017.

https://www.arduino.cc/en/Reference/SPISettings

[8] Librería SPI para Arduino. Consultado en Junio 2017.

http://arduino.cc/en/Reference/SPI

[9] Bus I2C en Arduino. Consultado en Mayo 2017.

https://aprendiendoarduino.wordpress.com/2016/11/14/bus-i2ctwi/

[10] Librería Wire (I2C) para Arduino. Consultado en Mayo 2017.

https://www.arduino.cc/en/Reference/Wire

[11] TFG Alejando García Osés, Junio 2015. Diseño de una red CAN bus con Arduino. Consultado en Abril 2017.

https://academica-e.unavarra.es/bitstream/handle/2454/19115/TFG%20Dise%C3%B1o%20de%20una%20Red%20Can%20bus%20-%20Alejandro%20Garc%C3%ADa%20Os%C3%A9s.pdf?sequence=1&isAllowed=y

[12] Apuntes Aula Mercedes 2017. Consultado en Abril 2017.

[13] Datasheet Convertidor de voltaje LM317t

http://docs-europe.electrocomponents.com/webdocs/1386/0900766b813862fa.pdf

[14] Datasheet RTC

http://es.rs-online.com/web/p/kits-de-desarrollo-de-procesador-y-microcontrolador/8209832/

[15] Datasheet Controlador CAN MCP-2515

http://docs-europe.electrocomponents.com/webdocs/137e/0900766b8137ef40.pdf

[16] Datasheet Transceptor CAN MCP 2551

http://docs-europe.electrocomponents.com/webdocs/14f3/0900766b814f3bfd.pdf

[17] Datasheet SD

http://store.digilentinc.com/pmod-sd-full-sized-sd-card-slot/

[18] Mecanismo de interrupciones y cómo usarlas en Arduino. Consultado en Mayo 2017.

https://www.luisllamas.es/que-son-y-como-usar-interrupciones-en-arduino/
http://www.prometec.net/interrupciones/

[19] Librería software SD para Arduino. Consultado en Junio 2017.

https://www.arduino.cc/en/Reference/SD

[20] Librería SdFat para Arduino. Consultado en Junio 2017.

https://github.com/greiman/SdFat

[21] Librería MCP para Arduino. Consultado en Mayo 2017.

https://github.com/coryjfowler/MCP_CAN_lib

[22] Librería RTC para Arduino. Consultado en Mayo 2017.

https://github.com/edebill/PCF8583/blob/master/PCF8583.h

[23] Forum oficial de Arduino. Consultado en Junio 2017.

http://forum.arduino.cc/

[24] Datasheet multiplexor Quad 2:1 ADG774

http://docs-europe.electrocomponents.com/webdocs/077f/0900766b8077ff33.pdf

[25] Almacenamiento de datos en una tarjeta SD conectada a Arduino por SPI. Consultado en Junio 2017.

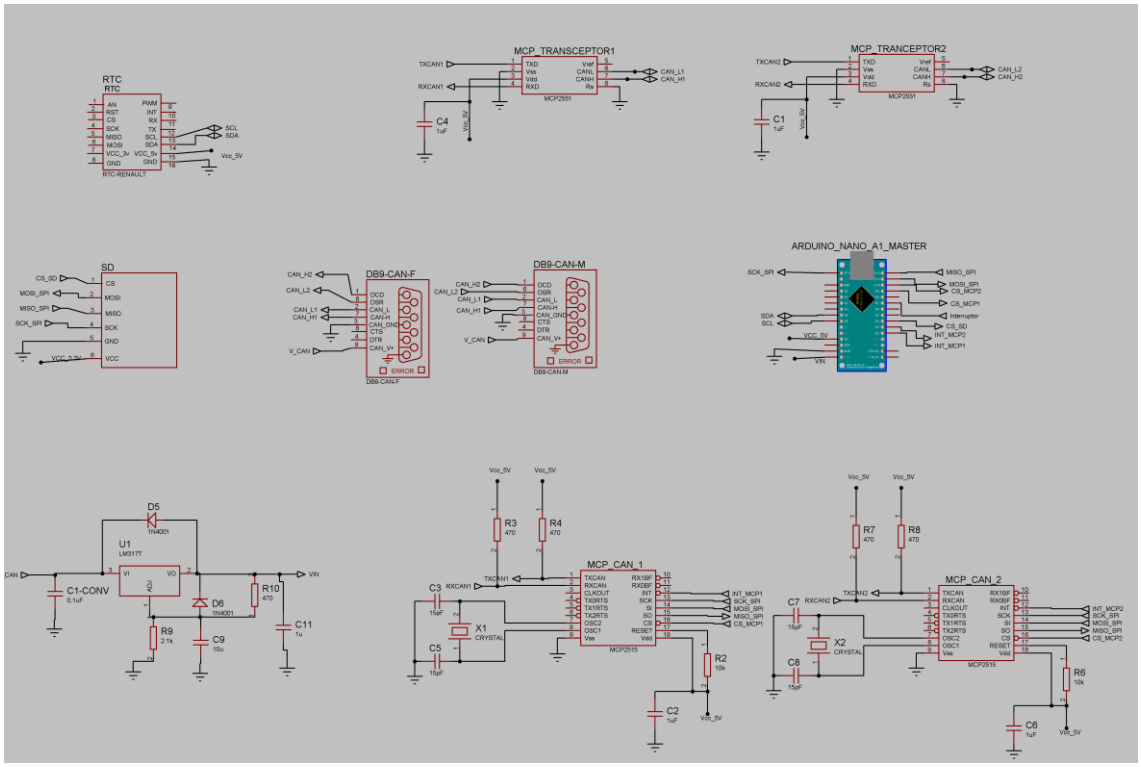
https://polaridad.es/almacenar-tarjeta-sd-spi-arduino/

[26] Aprovechamiento de la memoria de un Arduino. Consultado en Junio 2017.

http://cursos.olimex.cl/como-aprovechar-al-maximo-la-memoria-de-tu-arduino/

CAPÍTULO VIII: ANEXOS

Esquema Proteus Primer Prototipo con Un Arduino



Esquema Proteus Segundo Prototipo con Dos Arduinos