Universidad de Valladolid

E.T.S.I.T

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías Específicas de Telecomunicación

Diseño y construcción de un dispositivo de adquisición de datos para bus CAN

Autor:

D. Ismael Sanz Martín

Tutor:

Dr. D. Juan Carlos Aguado Manzano

Valladolid, Julio de 2017

Universidad de Valladolid

Trabajo Fin de Grado

Título:	Diseño y construcción de un dispositivo de adquisición de datos para bus CAN
Autor:	D. Ismael Sanz Martín
Tutor:	Dr. D. Juan Carlos Aguado Manzano
Departamento:	Teoría de la Señal y Comunicaciones e Ingeniería Telemática

Tribunal

Presidente:	Ignacio de Miguel Jiménez
Vocal:	Noemí Merayo Álvarez
Secretario:	Juan Carlos Aguado Manzano
Suplente 1:	Ramón Durán Barroso
Suplente 2:	Patricia Fernández Reguero

Fecha:	Julio 2017
Calificación:	

Universidad de Valladolid

Resumen

Este trabajo presenta el desarrollo tanto hardware como software de un sistema de adquisición de datos para los buses de comunicación CAN utilizando el programa de software libre BUSMASTER. Está enfocado al sector de la automoción, aunque puede ser utilizado en otros sectores que usen la misma tecnología.

Dentro del proyecto se hace una comparativa entre el software libre de BUSMASTER y CANoe, que es utilizado en la asignatura de Tecnologías de la Información y la Comunicación en Automoción. Además, se diseña y desarrolla una dispositivo hardware de adquisición de datos para utilizar junto con BUSMASTER que, una vez desarrollada la librería software correspondiente, sustituirá el elemento más caro de la gama de CANoe, el CANCaseXL.

Palabras clave: CAN, BUSMASTER, GUI, Socket, Arduino, CANoe.

Abstract

This work presents the hardware and software development of a data acquisition system for CAN communication buses, using a free software program BUSMASTER. It is focused on the automotive area, although it can be used in other areas that use the same technology.

BUSMASTER software is compared with CANoe software, which is used in the subject “Information and Communication Technologies in Automotive Industry”. Moreover, a hardware device for data acquisition is designed and developed. This device can be used together with BUSMASTER as long as the library software is programmed. This device can substituted the CANCaseXL, one of the more expensive devices of CANoe suite.

Keywords: CAN, BUSMASTER, GUI, Socket, Arduino,CANoe.

Universidad de Valladolid

AGRADECIMIENTOS

Lo primero de todo, me gustaría dar las gracias a mi familia por todo el apoyo que me ha ofrecido a lo largo de esta etapa y sin quienes no hubiese sido capaz de llegar tan lejos. Habéis sabido comprenderme y apoyarme cuando más lo necesitaba.

También me gustaría agradecer a todos los grandes compañeros, y a amigos, con quienes he compartido sufrimientos y alegrías durante estos años y han hecho de ellos algo inolvidable.

Por último, dar las gracias a mi tutor Juan Carlos Aguado por darme la oportunidad de trabajar juntos, por todo su apoyo, por enseñarme a aprender y saber guiarme en los momentos adecuados.

A todos ellos, muchas gracias.

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

CAPÍTULO 1: INTRODUCCIÓN

1.1 – INTRODUCCIÓN

Este proyecto nace de la necesidad de disponer de una herramienta sencilla y mucho más económica que la CANcaseXL propietaria de Vector, la cual es utilizada en el Aula Mercedes-Benz de la Universidad de Valladolid. En la actualidad la herramienta más potente y la más extendida en el mundo de la automoción para desarrollo y pruebas de software es CANoe [26]. Esta herramienta propietaria es de Vector y únicamente compatible con el hardware que ellos te proporcionan. Todo esto, junto con la licencia interna que se debe adquirir, hace que el coste sea muy elevado.

A su vez, la industria automovilística no es ajena al movimiento de software libre, y con lo cual la aparición del software libre BUSMASTER [1] supone una alternativa muy interesante para la substitución del software CANoe La herramienta BUSMASTER surge como una alternativa libre la cual puede ser utilizada con hardware de bajo coste y que no pasa desapercibida para grandes empresas las cuales necesitan del uso de estas herramientas. BUSMASTER puede ser usado tanto por fabricantes de automóviles como por proveedores de ECUs en labores de desarrollo, análisis, simulación, testing y diagnóstico. Debido a las ventajas económicas y a las similitudes con CANoe, BUSMASTER sigue desarrollándose diariamente y adquiriendo nuevas mejoras y funcionalidades, haciéndose cabida en importantes compañías.

Estas dos herramientas conjuntamente, nos permitirán recibir y transmitir tráfico CAN y podrán ser utilizadas tanto en labores de desarrollo, análisis, simulación, testing y diagnóstico.

Los buses de comunicación han adquirido una gran importancia en el sector de la automoción debido al avance de la industria electrónica y a las diferentes normativas que exigen una mayor eficiencia del vehículo tanto a nivel de prestaciones como a nivel medioambiental. Así pues, los vehículos actuales contienen siempre elementos electrónicos. Cada uno de estos elementos suele requerir información de otros módulos o ECUs para poder ejercer su función de forma eficiente. Este intercambio de información se realizaba inicialmente sin ningún estándar, haciendo que el diseño del cableado fuera costoso y poco práctico, debido a que si se cambiaba algún módulo probablemente se tuviera que cambiar el cableado. En la Ilustración 1 se puede observar una comparativa entre el cableado necesario en los sistemas de automoción convencionales y el cableado actual.

La solución a este problema apareció con los buses de comunicación [8], que reducen el coste del cableado, permiten modularidad en el vehículo, mayor flexibilidad, reducción del peso del cableado, evitar información redundante de los sensores, así como la facilidad para su fabricación en planta. Estos buses se han diseñado bajo los siguientes principios:

- Bajo coste
- Inmunidad ante interferencias y ambientes hostiles
- Robustez y fiabilidad

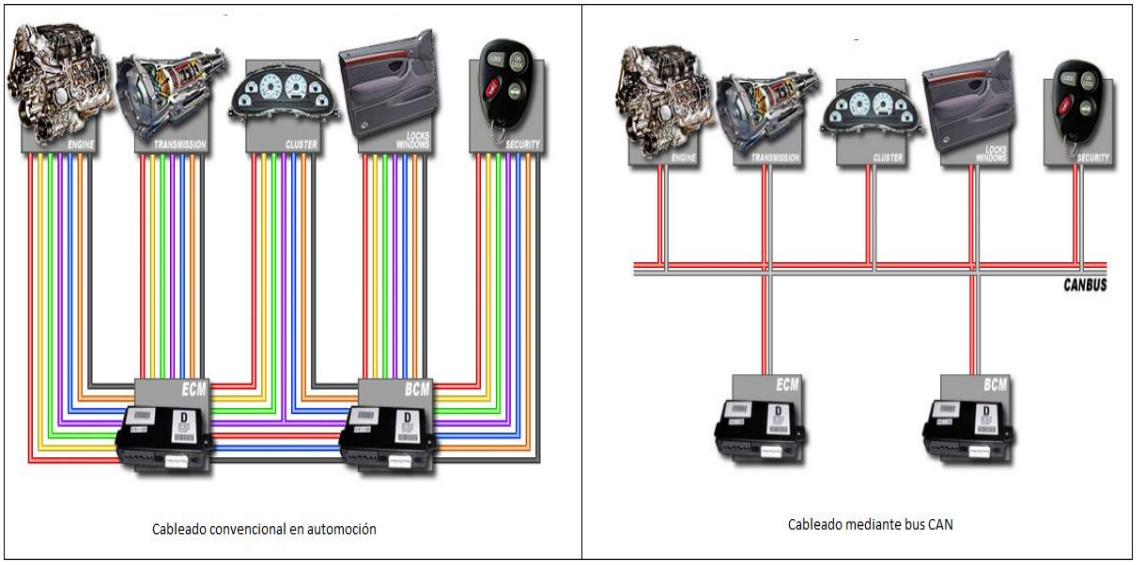


Ilustración 1: Comparativa de cableado entre sistemas de automoción [24]

El aumento de ECUs que se van añadiendo a estos buses debido a nuevos avances, supone una clara necesidad de dividir estos sistemas de comunicación según sus prestaciones, velocidad de respuesta y robustez, entre otros parámetros. En la Ilustración 2 podemos ver una clasificación de estos sistemas de comunicaciones en función de su tasa de transmisión y coste relativo.

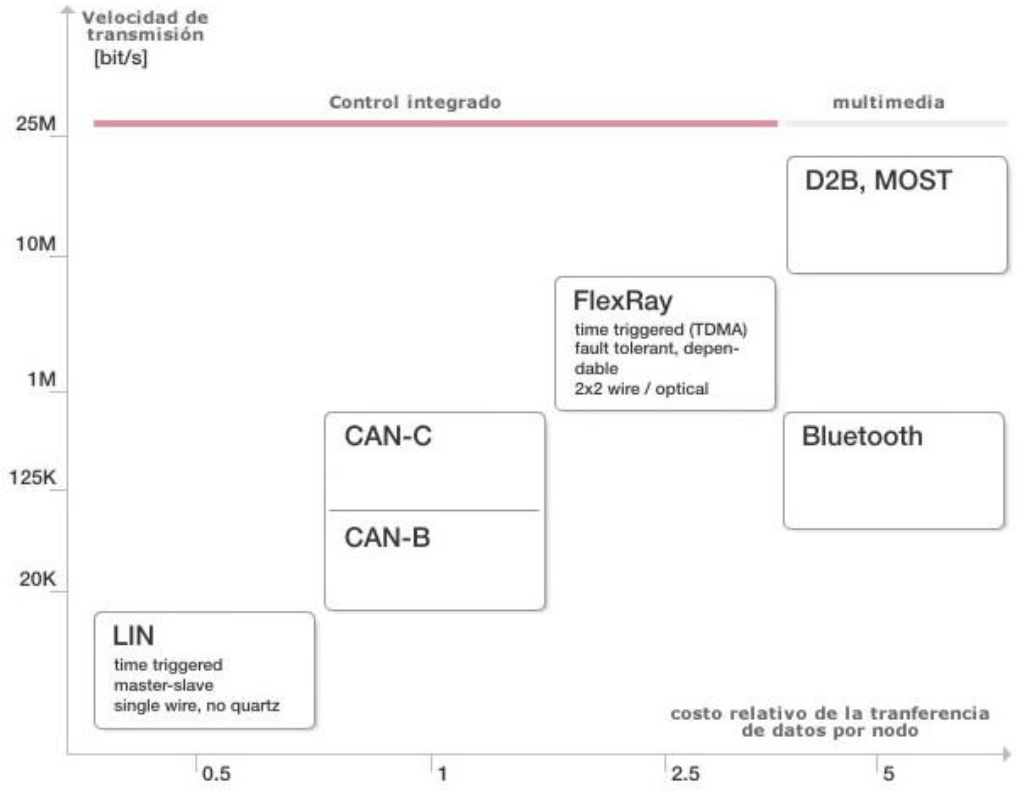


Ilustración 2: Clasificación de los protocolos de comunicación en automoción [25]

1.2 – OBJETIVOS

Inicialmente, tal y como se titula este Trabajo Final de Grado, el objetivo principal era el diseño y construcción de un dispositivo de adquisición de datos para bus CAN. Sin perder de vista este objetivo como meta del TFG, se fueron desarrollando una serie de objetivos secundarios que añaden utilidad y desarrollo a nuestro dispositivo. A continuación se mencionan dichos objetivos:

- Análisis de las prestaciones ofrecidas por el software de código abierto BUSMASTER. En base a ello realizar una comparativa en el Aula Mercedes-Benz de la Universidad de Valladolid con el software CANoe.
- Proporcionar a BUSMASTER una Interfaz Gráfica de Usuario para, de este modo, equipararlo al uso que se le da a CANoe en el Aula Mercedes-Benz.
- Diseño, construcción y encapsulación del dispositivo de adquisición de datos para bus CAN.
- Estudio general para la integración de nuestro dispositivo de adquisición de datos como un nuevo hardware compatible dentro del software BUSMASTER.

1.3 – FASES Y MÉTODOS

Para la consecución de los objetivos anteriormente mencionados se han seguido las siguientes fases:

- Realización en el Aula Mercedes-Benz de las prácticas de laboratorio ofrecidas por la asignatura de la Universidad de Valladolid "Tecnologías de la Información y las Comunicaciones en Automoción". Se realizaron en paralelo usando los software CANoe y BUSMASTER para completar así el primero de los objetivos.
- Análisis de las diferentes alternativas para realizar una GUI, buscando siempre aquellas que pudiesen ser compatibles con el software BUSMASTER.
- Búsqueda de información, investigación y desarrollo de los diferentes métodos encontrados para la implementación de la GUI.
- Creación de un socket cliente-servidor en el Entorno de Desarrollo Integrado Qt para su uso como GUI.
- Adaptar el cliente del socket creado a las librerías de Windows para su posterior intento de integración en BUSMASTER.
- Estudio del IDE de Arduino y de las librerías disponibles necesarias para nuestro sistema de adquisición de datos.
- Estudio del funcionamiento de los MCPs, tanto el controlador CAN como el transceptor CAN, para crear el diseño hardware de nuestro dispositivo, basándonos en el diseño de la ECU de Telemetría de la asociación universitaria Pisuerga Sport.
- Diseño de la placa PCB de nuestro sistema con ayuda de Proteus.

- Montaje de la placa en una protoboard para comprobar el funcionamiento correcto de la misma.
- Desarrollo del código en el IDE de Arduino para nuestro sistema.
- Testeo y validación de nuestro diseño en el aula Mercedes-Benz de la Universidad de Valladolid.
- Diseño de nuestro sistema para ser implementado en una placa de tiras en el espacio más reducido posible.
- Montaje y encapsulación de nuestro sistema en una placa de tiras.
- Testeo y validación del diseño final en el aula Mercedes-Benz de la Universidad de Valladolid.
- Estudio del funcionamiento de BUSMASTER para la integración de un nuevo hardware compatible para bus CAN.

CAPÍTULO 2: ENTORNO DE DESARROLLO INTEGRADO

2.1 – BUS CAN

El bus CAN (Controller Area Network) es un protocolo de comunicación en serie desarrollado para el intercambio de información entre unidades de control electrónicas del automóvil.

Este sistema permite compartir una gran cantidad de información entre las unidades de control abonadas al sistema, lo que provoca una reducción importante tanto del número de sensores utilizados como de la cantidad de cables que componen la instalación eléctrica.

De esta forma aumentan considerablemente las funciones presentes en los sistemas del automóvil donde se emplea el bus CAN sin aumentar los costes, además de que estas funciones pueden estar repartidas entre dichas unidades de control.

Las características principales de bus CAN [3]:

- Económico y sencillo: Dos de las razones que motivaron su desarrollo fueron precisamente la necesidad de economizar el coste monetario y el de minimizar la complejidad del cableado, por parte del sector automovilístico.
- Estandarizado: Se trata de un estándar definido en las normas ISO (Internacional Organization for Standardization), concretamente la ISO 11898, que se divide a su vez en varias partes, cada una de las cuales aborda diferentes aspectos de CAN.
- Medio de transmisión adaptable: El cableado es muy reducido en comparación con otros sistemas. Además, a pesar de que por diversas razones el estándar de hardware de transmisión sea un par trenzado de cables, el sistema de bus CAN también es capaz de trabajar con un solo cable u otros medios físicos como los enlaces ópticos o los enlaces de radio.
- Número de nodos: En principio es posible conectar un número tan alto como se desee de dispositivos a una red CAN, pero por diversas razones como mantener la impedancia del medio o cuestiones prácticas de comunicación se suele considerar como número máximo hasta 110 dispositivos en una sola red CAN.
- Optimización del ancho de banda: Los métodos utilizados para distribuir los mensajes en la red, como el envío de estos según su prioridad, contribuyen a un mejor empleo del ancho de banda disponible.
- Desconexión autónoma de nodos defectuosos: Si un nodo de red cae, sea cual sea la causa, la red puede seguir funcionado, ya que es capaz de desconectarlo o aislarlo del resto. De forma contraria, también se pueden añadir nodos al bus sin afectar al resto del sistema, y sin necesidad de reprogramación.
- Velocidad flexible: La ISO define dos tipos de redes CAN: una 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. En la Tabla 1 se puede observar la dependencia de la velocidad con respecto a la distancia.

Velocidad (Kbps)	Tiempo de bit (µs)	Longitud máxima del bus (m)
1000	1	30
800	1.25	50
500	2	100
250	4	250
125	8	500
50	20	1000
20	50	2500
10	100	5000

Tabla 1: Relación velocidad-distancia en el bus CAN

2.1.1 – Topología

La topología del bus CAN se muestra en la Ilustración 3.

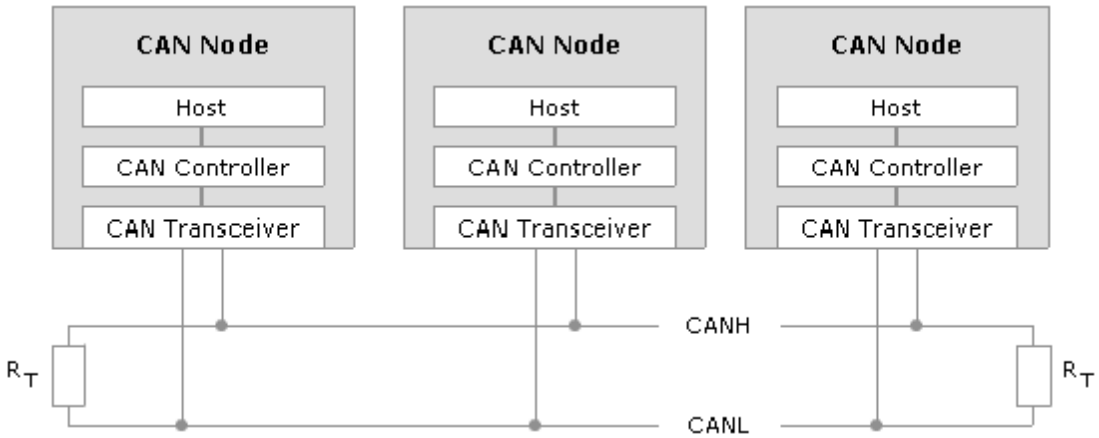


Ilustración 3: Topología del bus CAN

Se trata de un bus multimaestro, donde todos los nodos comparten las mismas características, sin necesidad de gestión del arbitraje. La prioridad queda así determinada por el contenido del mensaje, en CAN es un campo determinado, el identificador de mensaje, el que determina la prioridad.

Cada nodo CAN se compone de los siguientes elementos:

- Host: suele tratarse de un microcontrolador que interpreta los mensajes recibidos y decide qué mensajes se han de transmitir.
- Controlador CAN: recibe del microprocesador los datos que han de ser transmitidos. Los acondiciona y los pasa al transceptor CAN. Asimismo recibe los datos procedentes del transceptor CAN, los acondiciona y los pasa al microprocesador en la unidad de control. Puede tener funciones de filtrado para agilizar el trabajo del Host.

- Transceptor CAN: es un transmisor y un receptor. Transforma los datos del controlador CAN en señales eléctricas y transmite éstas sobre los cables del bus CAN. Además recibe los datos y los transforma para el controlador CAN.

2.1.2 – Capa física y Acceso al medio

El protocolo CAN utiliza, en su versión más extendida, un par de cables trenzados para conectarse a todas las unidades del sistema (tal y como se ve en la Ilustración 3 los cables recibirán el nombre de CAN_L y CAN_H respectivamente). Esta disposición permite inmunidad ante interferencias electromagnéticas externas. Las señales que circulan a través de estos cables son diferenciales. Existen diversos niveles de tensión en función del estándar de CAN que se utilice. Esta será la versión que utilicemos en este trabajo. Para conocer otras posibilidades en cuanto a la capa física se refiere se puede leer [5].

El bus tiene dos estados definidos: estado dominante y estado recesivo. El estado dominante corresponde al nivel lógico 0 y el estado recesivo al nivel lógico 1. Cuando dos nodos intentan transmitir bits diferentes se produce una colisión y el valor del bit dominante prevalece sobre el valor del bit recesivo. En ese caso el nodo que intentaba transmitir el valor recesivo detecta la colisión y pasa a modo pasivo, es decir, deja de transmitir para escuchar lo que transmite el otro nodo. Por lo tanto la forma de acceso al medio es una CSMA/BA (Carrier Sense Multiple Access/Bit Arbitration). Esta forma de acceso permite aprovechar al máximo el ancho de banda disponible y dar prioridad a unos mensajes sobre otros.

La definición de capa física del estándar asocia a cada rango de velocidades de bus una serie de rango de voltajes y características de las señales que se especifican a continuación.

2.1.2.1 – Highspeed – CAN (ISO 11898-2)

La tasa de transmisión para esta definición de la capa física va desde 125 kbps hasta 1 Mbps. Los niveles de tensión aparecen reflejados en la Ilustración 4.

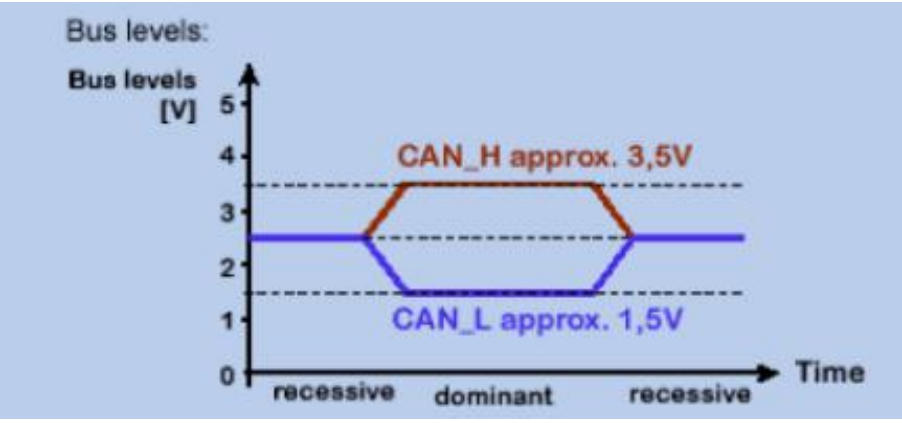


Ilustración 4: Niveles de tensión en highspeed CAN

	CAN_H – CAN_L
Estado dominante	> 0.9 V
Estado recesivo	< 0.5 V

Tabla 2: Niveles lógicos en highspeed CAN

En la Tabla 2 se interpretan los niveles lógicos dominante y recesivo.

El par de cables trenzados (CAN_H y CAN_L) constituyen una línea de transmisión. Si dicha línea de transmisión no está configurada con los valores correctos, cada trama transferida causa una reflexión que puede originar fallos de comunicación. Como la comunicación en el bus CAN fluye en ambos sentidos, ambos extremos de red deben de estar cerrados mediante una resistencia de 120 Ω, tal y como se puede ver en la Ilustración 5. Ambas resistencias deberían poder disipar al menos 0.25 W de potencia.

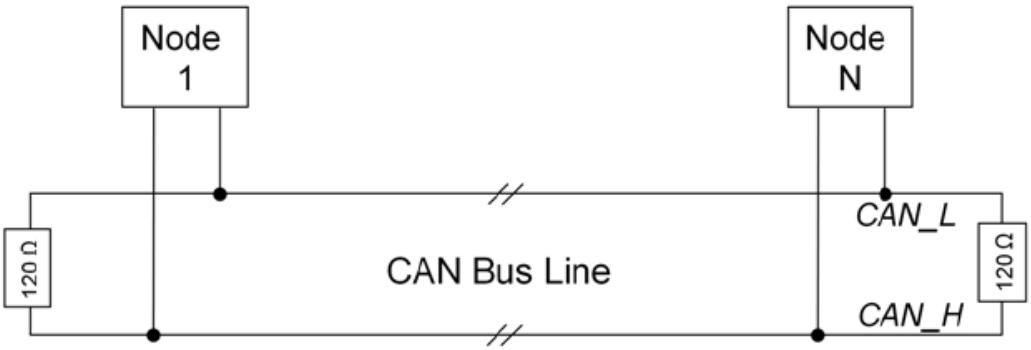


Ilustración 5: Topología highspeed CAN

2.1.2.2 – Lowspeed – CAN (ISO 11519-2 / ISO11898-3)

La tasa de transmisión para esta definición de la capa física va desde 10 kbps hasta 100 kbps. Los niveles de tensión aparecen reflejados en la Ilustración 6. En la Tabla 3 se interpretan los niveles lógicos dominante y recesivo.

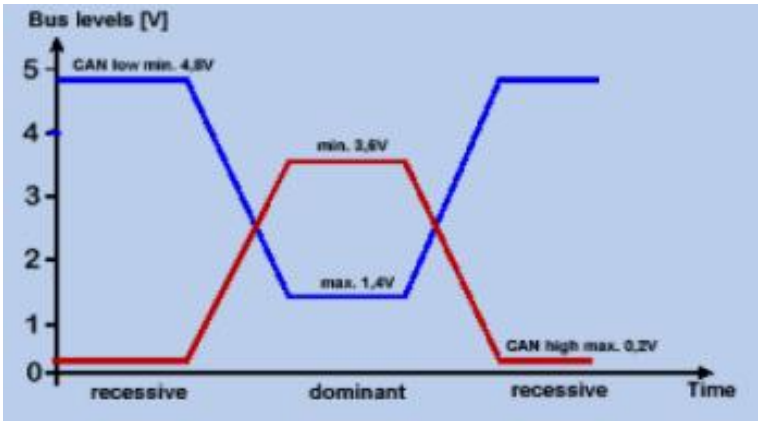


Ilustración 6: Niveles de tensión en lowspeed CAN

	CAN_H – CAN_L
Estado dominante	2 V
Estado recesivo	5 V

Tabla 3: Niveles lógicos en lowspeed CAN

Como podemos observar lowspeed CAN requiere mayor diferencia de voltaje entre los datos recesivos y dominantes. Esto es debido a que, a diferencia de highspeed CAN, es un bus activo y necesita una fuente de alimentación, haciendo el bus más lento pero más resistente frente a interferencias.

A diferencia de highspeed CAN, lowspeed CAN requiere dos resistencias en cada transceptor: una para la señal CAN_H y otra para la señal CAN_L. Esta configuración permite al transceptor detectar fallos en la red. La suma de todas las resistencias en paralelo, debe estar en el rango de 100-500 Ω. En la Ilustración 7 se puede ver el esquema eléctrico de esta configuración.

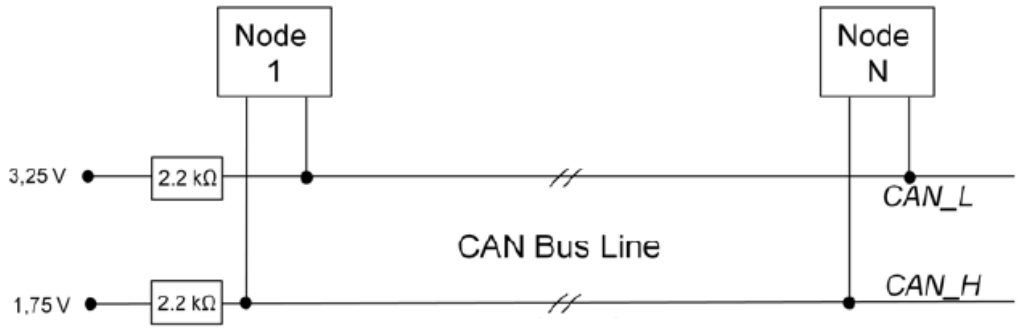


Ilustración 7: Topología lowspeed CAN

2.1.3 – Capa de enlace de datos

La capa de enlace de datos es responsable del acceso al medio y el control lógico y está dividida a su vez en dos niveles.

- El subnivel LLC (Logical Link Control): Gestiona el filtrado de los mensajes, las notificaciones de sobrecarga y la administración de la recuperación.
- El subnivel MAC (Medium Access Control): Es el núcleo del protocolo CAN y gestiona el entramado y desentramado de los mensajes, el arbitraje a la hora de acceder al bus y el reconocimiento de los mensajes, así como el chequeo de posibles errores y su señalización, el aislamiento de fallos en unidades de control y la identificación del estado libre del bus para iniciar una transmisión o recepción de un nuevo mensaje.

2.1.4 – Codificación y sincronización de datos

La codificación de bits se realiza por el método NRZ (Non-Return-to Zero) que se caracteriza por que el nivel de señal puede permanecer constante durante largos periodos de tiempo y habrá que tomar medidas para asegurarse de que el intervalo máximo permitido entre dos señales no es superado.

Este tipo de codificación requiere poco ancho de banda para transmitir, pero en cambio, no puede garantizar la sincronización de la trama transmitida. Para resolver esta falta de sincronismo se emplea la técnica del "bit stuffing": cada 5 bits consecutivos con el mismo estado lógico en una trama (a excepción de los bloques CRC, ACK, del delimitador de final de trama y el espacio entre tramas), se inserta un bit del estado lógico contrario, no perdiéndose así la sincronización. Por otro lado este bit extra debe ser eliminado por el receptor de la trama, que sólo lo utilizará para sincronizar la transmisión.

No hay flanco de subida ni de bajada para cada bit y disminuye la frecuencia de señal respecto a otras codificaciones.

2.1.5 – Formatos de trama

Existen cuatro tipos de trama CAN:

- Trama de datos (data frame)
- Trama remota (remote frame)
- Trama de error (error frame)
- Trama de sobrecarga (overload frame)

Trama de datos: es el tipo de mensaje más común. Podemos tener tramas estándar (standard frames) para CAN 2.0A o tramas extendidas (extended frames) para CAN 2.0B. En la Ilustración 8 se muestra el formato de trama estándar para CAN 2.0A. En la Tabla 4 se explican cada uno de los campos de este formato de trama.

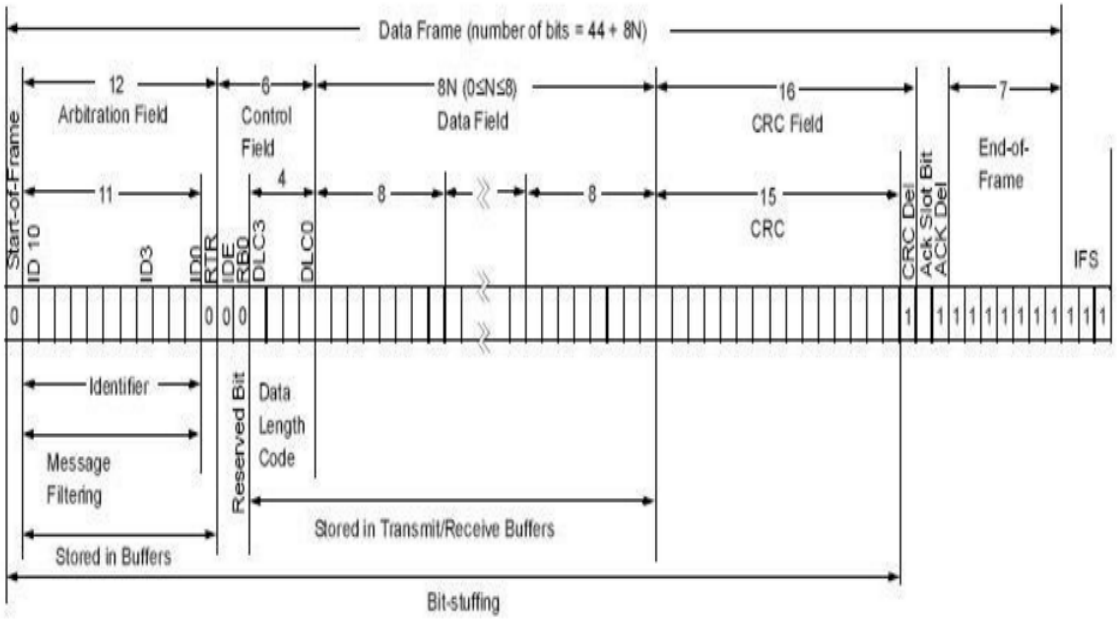


Ilustración 8: Formato de trama CAN estándar

Nombre del campo	Longitud (bits)	Descripción
Inicio de trama	1	Indica el comienzo de una transmisión.
Identificador ID	11	Un identificador (único) que también representa la prioridad de la trama.
Petición de transmisión remota RTR	1	Dominante (0) para tramas de datos y recesivo (1) para tramas de peticiones remotas.
Bit de extensión de identificador IDE	1	Dominante (0) para el identificador de 11 bits.
Bit reservado RB0	1	Bit reservado. Debe ser dominante (0).
Código de longitud de datos DLC	4	Número de bytes de datos en el mensaje, entre 0 y 8.
Campo de datos	0 – 64	Datos de la trama (la longitud del campo viene dada por el DLC).
CRC	15	Verificación por redundancia cíclica. Código que verifica que los datos fueron transmitidos correctamente.
Delimitador CRC	1	Debe ser recesivo (1).
Acuse de recibo ACK	1	El transmisor emite recesivo (1) y cualquier receptor emite dominante (0).
Delimitador ACK	1	Debe ser recesivo (1).
Fin de trama EOF	7	Debe ser recesivo (1).

Tabla 4: Campos del formato de tramas CAN

Vale la pena señalar que la presencia de un acuse de recibo (Acknowledgement) en el bus no significa que los destinatarios previstos hayan recibido el mensaje. Lo único que sabemos es que uno o más nodos en el bus han recibido el mensaje correctamente.

En la Ilustración 9 se muestran las diferencias del formato de trama extendida para CAN 2.0B con respecto al formato estándar. Como se puede observar, las diferencias entre el formato de trama estándar y el formato de trama extendida residen en la longitud del identificador. Para el caso de tramas extendidas tenemos 29 bis de identificador.

Para distinguir entre los dos formatos, el bit IDE estará en estado dominante (0) para el caso de trama estándar y en estado recesivo (1) para el caso de trama extendida. Los controladores CAN que soportan el formato de trama extendida también son capaces de enviar y recibir mensajes en formato estándar.

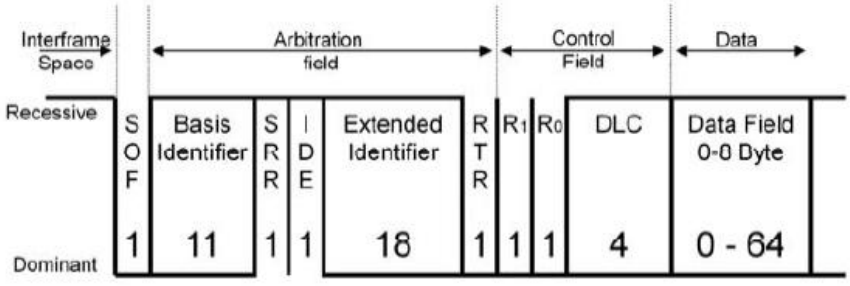


Ilustración 9: Formato de trama CAN extendida

Trama remota: Un nodo de destino puede solicitar datos de la fuente mediante el envío de una trama remota con un identificador que coincide con el identificador de la trama de datos requerida. Tiene el formato de trama estándar pero sin campo de datos y con el bit RTR recesivo (1) para dar prioridad a los mensajes de datos con ese identificador.

Trama de error: Se transmite cuando un nodo detecta un fallo y hace que todos los demás nodos detecten un fallo, por lo que también enviarán tramas de error. El transmisor intentará entonces retransmitir el mensaje.

Como se muestra en la Ilustración 10, la trama de error contiene un "error flag" con 6 bits dominantes (violando la regla del bit-stuffing) para sobrescribir los datos corruptos y un delimitador con 8 bits recesivos. El delimitador de error proporciona un espacio para que los otros nodos en el bus puedan reiniciar las comunicaciones después del error.

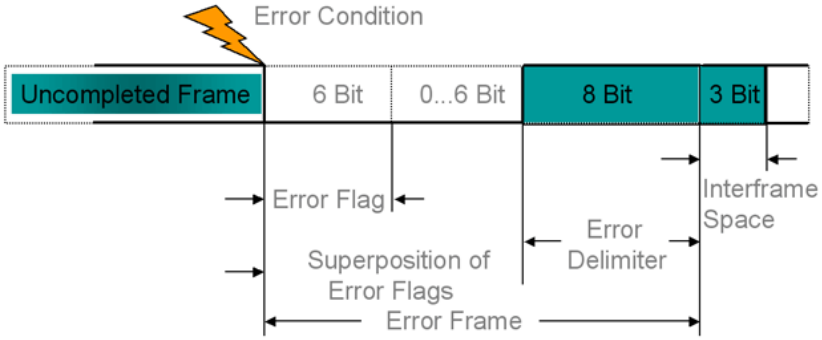


Ilustración 10: Formato de trama de error CAN

Trama de sobrecarga: Un nodo envía este tipo de tramas cuando está muy ocupado y no está preparado para recibir el siguiente mensaje. Tiene un formato similar al de las tramas de error. No se usa muy a menudo ya que los controladores CAN modernos no las necesitan.

2.1.6 – Ciclo en la transmisión de datos

En la Ilustración 11 se muestran los diferentes comportamientos de los nodos del bus CAN durante la transmisión. Estos comportamientos son:

- Proveer datos: La unidad de control provee los datos al controlador CAN para su transmisión.
- Transmitir datos: El transceptor CAN recibe los datos del controlador CAN, los transforma en señales eléctricas y los transmite.
- Recibir datos: Todas las demás unidades de control que están interconectadas a través del bus CAN se transforman en receptores.
- Revisar datos: Las unidades de control revisan si necesitan los datos recibidos para la ejecución de sus funciones o si no los necesitan.
- Adoptar datos: Si se trata de datos esperados por la ECU, la unidad de control en cuestión los adopta y procesa; si no son esperados, los desprecia.

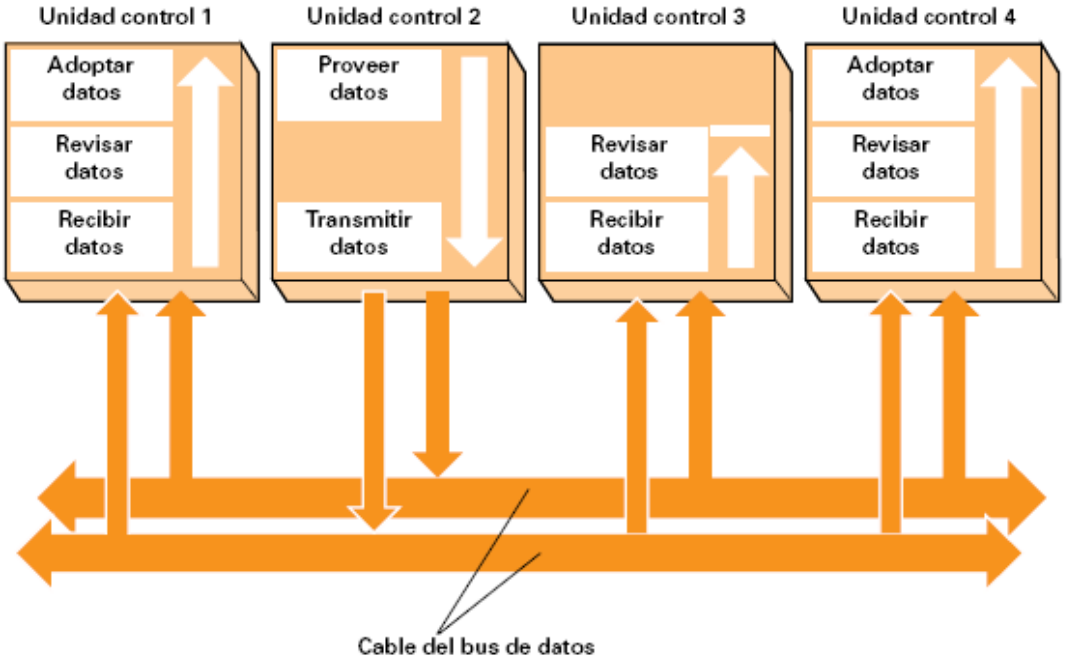


Ilustración 11: Comportamiento de los nodos

2.2 – CANoe

CANoe es una herramienta de desarrollo y pruebas de software de la empresa Vector Informatik GmbH. El software es utilizado principalmente por los fabricantes de automóviles y proveedores de unidades electrónicas de control (ECU) para el desarrollo, análisis, simulación, testing, diagnóstico y puesta en marcha de redes de ECUs y ECUs individuales. Su uso generalizado y gran número de sistemas buses soportados hace que sea especialmente adecuado para el desarrollo de ECUs en vehículos convencionales, vehículos híbridos y vehículos eléctricos.

2.2.1 – Visión general

Lo primero que tenemos que hacer cuando utilicemos CANoe es crear una nueva configuración y ajustar los parámetros de hardware, driver utilizado y asociación de las bases de datos.

Cuando queramos ejecutar una simulación tendremos que abrir Measurement Setup, como se muestra en la Ilustración 12, desde donde podemos configurar si queremos utilizar el modo online o modo offline y activar o desactivar los bloques que deseemos.

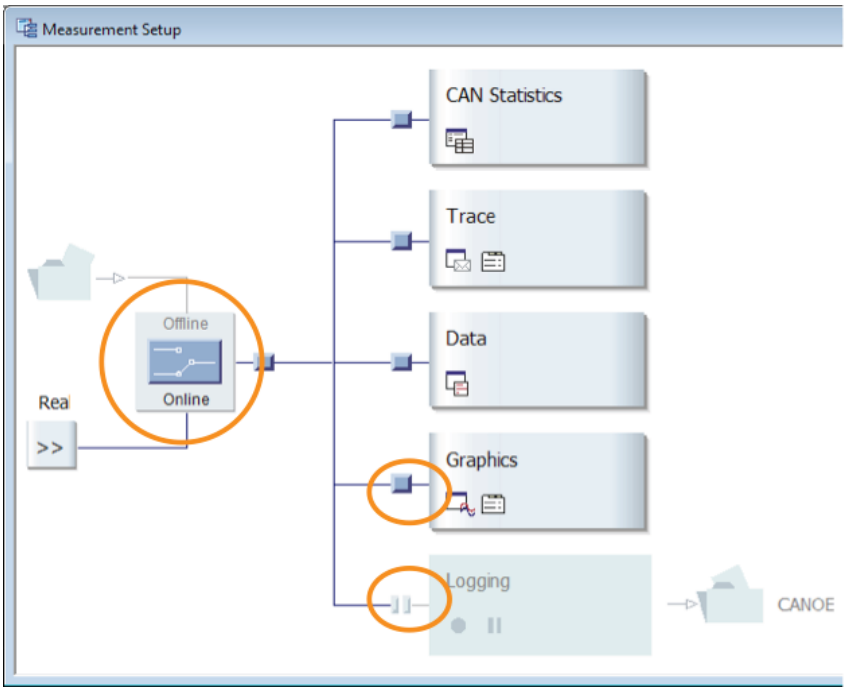


Ilustración 12: Measurement Setup

Para mostrar gráficamente el bus CAN y todos los nodos de red simulados abriremos Simulation Setup Window (Ilustración 13). Tenemos dos estados de los nodos simulados:

- Off state: el nodo se muestra en colores sombreados y unido a una línea negra. El nodo está inactivo o está trabajando un nodo real en el bus real con la misma función.
- Simulated Bus: el nodo se muestra de manera clara y unido a una línea roja. El nodo está activo y no está trabajando ningún nodo real en el bus real con la misma función.

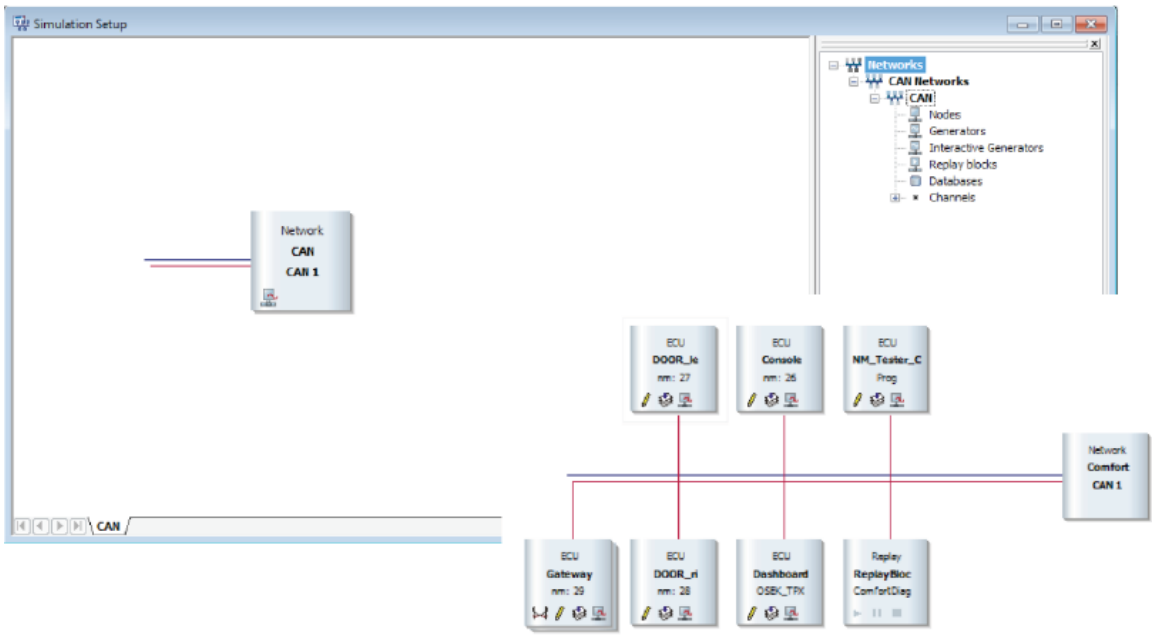


Ilustración 13: Simulation Setup

Modos de operación

- Modo online: traza los mensajes que se envían en el bus. El bus puede ser real o simulado.
    - Bus real: los mensajes de Simulation Setup se transmiten en uno o más buses reales, y en Measurement Setup son recibidos por uno o más buses reales. Los canales definidos corresponden a estos buses reales con sus controladores.
    - Bus simulado: buses y nodos de la red completamente simulados, no se retransmiten ni se reciben de redes reales.
- Modo offline: reproduce los mensajes que fueron trazados y grabados.

Crear una simulación

Para crear una simulación tenemos que tener definida la base de datos utilizando para ello CANdb++ Editor (Ilustración 14). Tal y como se ve en la Tabla 5 sus tareas son:

- Definir los nodos de red.
- Definir los mensajes y sus propiedades.
- Definir y posicionar elementos o señales de datos en un mensaje CAN.
- Definir las unidades físicas de una señal, si es apropiado.
- Definir los valores simbólicos de una señal, si es apropiado.
- Definir las variables comunes (conocidas como atributos), si es apropiado.

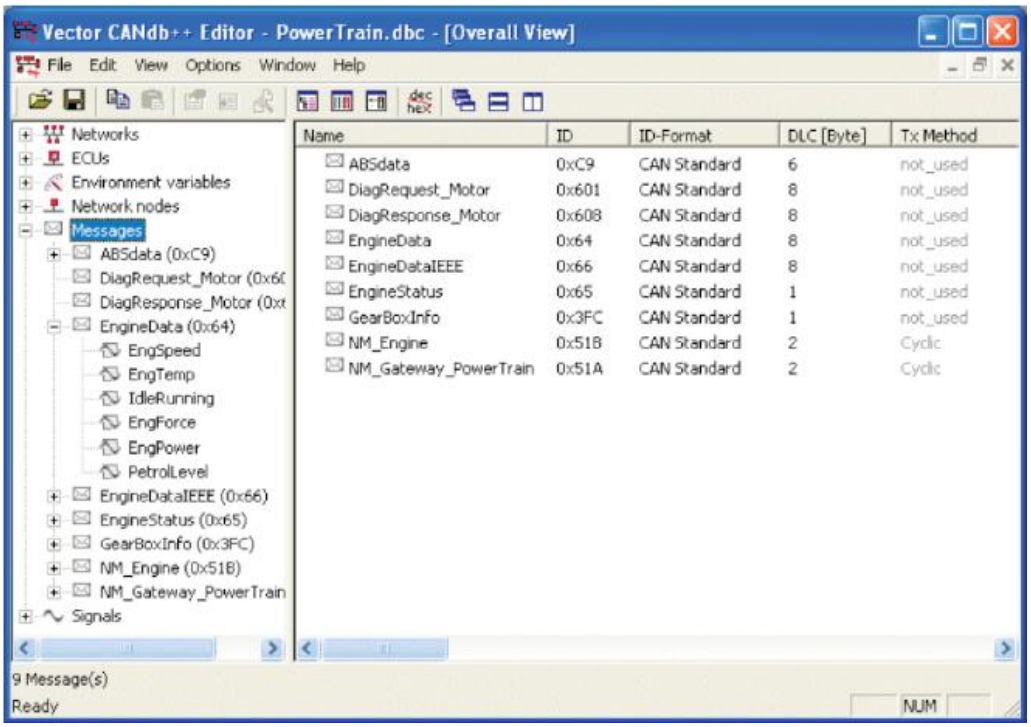


Ilustración 14: CANdb++ Editor

TIPO DE OBJETO	DESCRIPCIÓN
Netwoks	Una red de bus CAN que contiene múltiples ECUs para intercambiar datos.
ECUs	Unidades distribuidas de procesamiento dentro de una red (control de tracción, ABS…)
Nodes	Interfaces de red de una ECU usadas para transmitir y recibir datos desde el bus CAN.
Signals	Regiones de datos independientes en el campo de datos de un mensaje.
Messages	Contenedores de información que son transmitidos en el bus CAN.
Environment Variables	Variables de entrada y salida de los nodos de la red, tales como señales del interruptor del sensor de posición y señales del actuador.

Tabla 5: Objetos de la base de datos

El lenguaje de programación que usaremos para programar los nodos es CAPL, un lenguaje basado en C para herramientas de entorno basadas en PC de CANalyzer y CANoe. Es un lenguaje orientado a evento que nos permite simular:

- Nodo o comportamiento del sistema mediante instrucciones y valores legibles en lugar de valores hexadecimales.
- Mensajes de eventos, mensajes periódicos o mensajes repetitivos condicionados.
- Eventos humanos (presionar una tecla).
- Nodos temporizados o eventos de red.
- Múltiples eventos temporales, cada uno con su propio comportamiento programable.
- Operación normal, operación de diagnóstico u operación de fabricación.
- Los cambios en los parámetros físicos o valores simbólicos.
- Módulos y fallos de red para evaluar una estrategia de operación limitada.
- Funciones simples o complejas.

Modelado

CANoe soporta interfaces gráficas definidas por el usuario llamadas paneles. Contienen controles visibles que reaccionan tanto a eventos externos (activación de un interruptor) como internos (recepción de mensajes desde el bus CAN).

Para crear estos paneles gráficos utilizaremos el entorno Panel Designer (Ilustración 15).

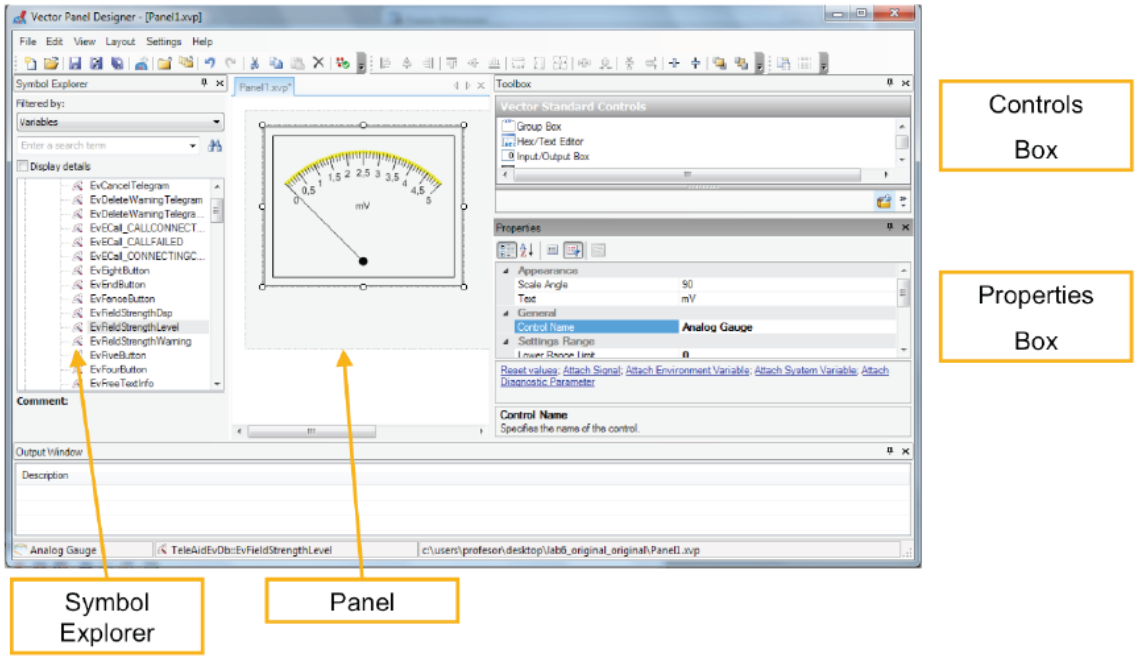


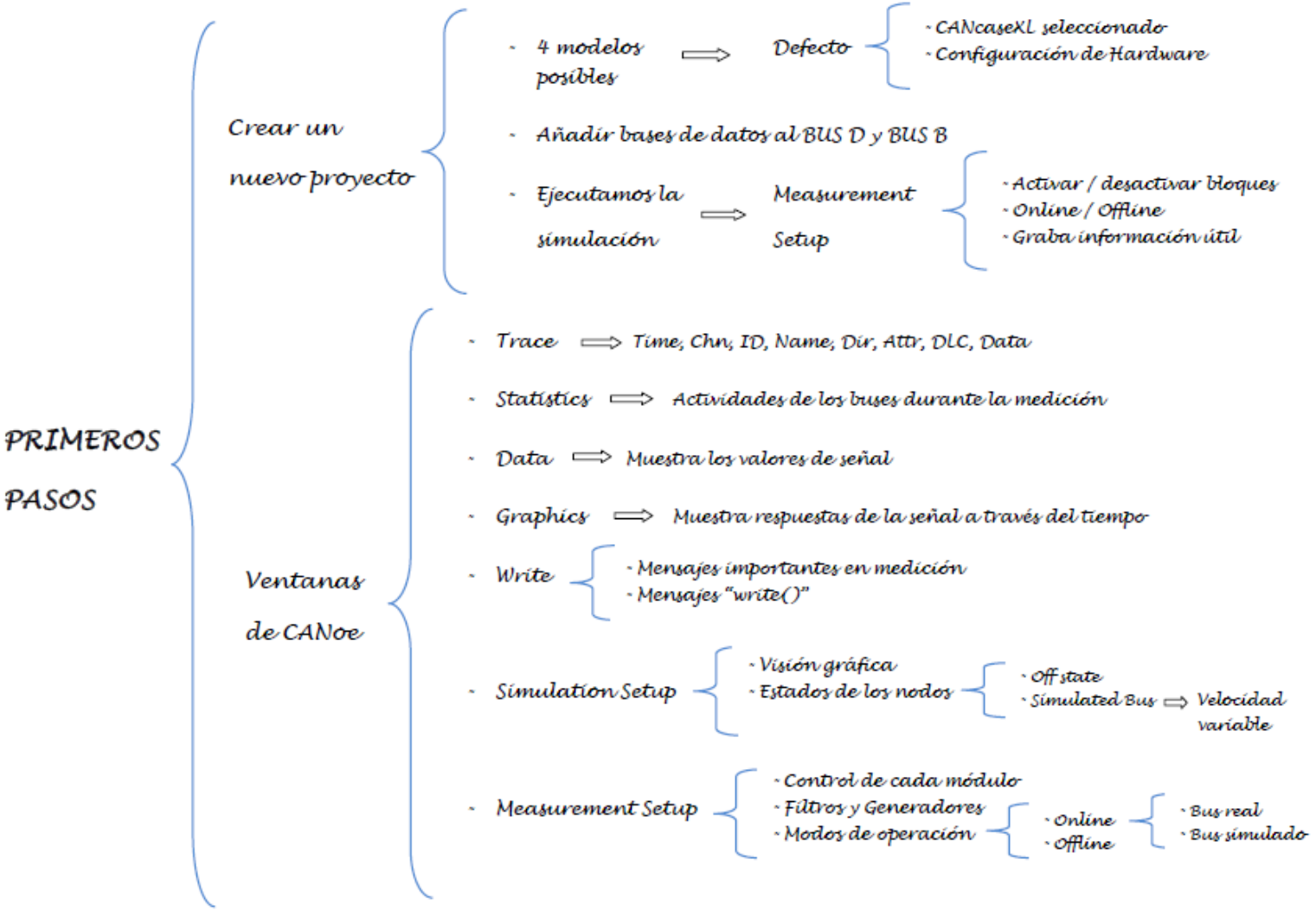
Ilustración 15: Panel Designer

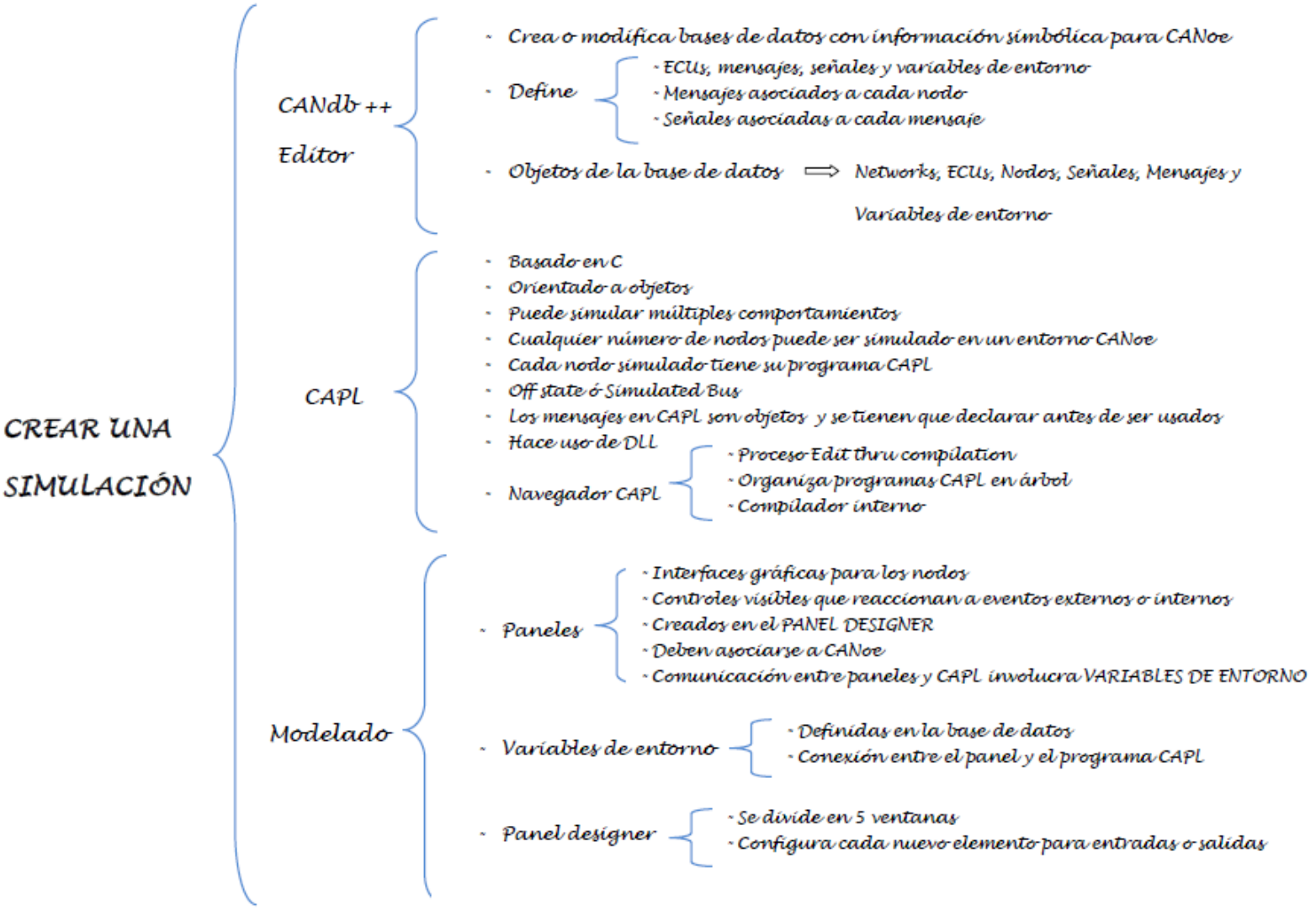
Una vez creados los paneles deben asociarse a CANoe. La comunicación entre los paneles y CAPL involucra variables de entorno. Todos los elementos dinámicos de control dentro de un panel deben estar asociados a una variable de entorno, la cual debe estar definida en la base de datos junto con los mensajes y señales.

2.2.2 – Diagramas

En este apartado hemos creado unos esquemas que nos van a permitir visualizar de una manera rápida la estructura y funcionalidades que nos ofrece el software CANoe para nuestros objetivos en el Aula Mercedes. Estos diagramas nos servirán de base para comparar las funcionalidades que ofrecen CANoe y BUSMASTER.

CANoe
Hardware
- PCMIA
- USB / Serie
- PCI 7 PCIe / PXI
Primeros pasos
- Crear un nuevo proyecto
- Ventanas de CANoe
Crear una simulación
- CANdb++ Editor
- Programar CAPL
- Modelado
Bloques funcionales
- Generator Block
- Interactive Generator Block
- Replay Block


PRIMEROS PASOS
Crear un nuevo proyecto
4 modelos posibles
Defecto
CANcaseXL seleccionado
Configuración de Hardware
Añadir bases de datos al BUS D y BUS B
Ejecutamos la simulación
Measurement Setup
Activar / desactivar bloques
Online / Offline
Graba información útil
Ventanas de CANoe
Trace
Time, Chn, ID, Name, Dir, Attr, DLC, Data
Statistics
Actividades de los buses durante la medición
Data
Muestra los valores de señal
Graphics
Muestra respuestas de la señal a través del tiempo
Write
Mensajes importantes en medición
Mensajes "write()"
Simulation Setup
Visión gráfica
Estados de los nodos
Off state
Simulated Bus
Velocidad variable
Measurement Setup
Control de cada módulo
Filtros y Generadores
Modos de operación
Online
Offline
Bus real
Bus simulado


CREAR UNA SIMULACIÓN
CANdb ++ Editor
Crea o modifica bases de datos con información simbólica para CANoe
Define
ECUs, mensajes, señales y variables de entorno
Mensajes asociados a cada nodo
Señales asociadas a cada mensaje
Objetos de la base de datos ⟹ Networks, ECUs, Nodos, Señales, Mensajes y Variables de entorno
CAPL
Basado en C
Orientado a objetos
Puede simular múltiples comportamientos
Cualquier número de nodos puede ser simulado en un entorno CANoe
Cada nodo simulado tiene su programa CAPL
Off state ó Simulated Bus
Los mensajes en CAPL son objetos y se tienen que declarar antes de ser usados
Hace uso de DLL
Navegador CAPL
Proceso Edit thru compilation
Organiza programas CAPL en árbol
Compilador interno
Modelado
Paneles
Interfaces gráficas para los nodos
Controles visibles que reaccionan a eventos externos o internos
Creados en el PANEL DESIGNER
Deben asociarse a CANoe
Comunicación entre paneles y CAPL involucra VARIABLES DE ENTORNO
Variables de entorno
Definidas en la base de datos
Conexión entre el panel y el programa CAPL
Panel designer
Se divide en 5 ventanas
Configura cada nuevo elemento para entradas o salidas

2.3 – BUSMASTER

BUSMASTER es una herramienta software de código abierto para simular, analizar y testear buses de sistemas de datos tales como CAN, LIN y FlexRay. Fue concebida, diseñada e implementada por Robert Bosch Engineering and Business Solutions (RBEI).

El software de código abierto BUSMASTER se puede descargar desde el repositorio virtual GitHub. El proyecto de software BUSMASTER es patrocinado por RBEI ETAS y está abierto a las contribuciones de la investigación y la industria. La apertura de la gestión del proyecto por los patrocinadores provee flexibilidad ante modificaciones y ampliaciones con respecto a los sistemas de bus, protocolos e interfaces de hardware. Asimismo, ETAS ofrece apoyo profesional y brinda servicios de ingeniería, incluyendo la personalización, tutorías y la formación.

El software puede ser desarrollado y gestionado con herramientas de software libre, como Microsoft Visual C Express. Gracias a la arquitectura modular de la herramienta, los desarrolladores de software pueden añadir fácilmente nuevas funciones al software.

2.3.1 – Características principales

Lo primero que nos planteamos a la hora de abordar un nuevo software es conocer su verdadero potencial. Para ello se deben conocer las herramientas disponibles y el funcionamiento de las mismas. En este punto vamos a tratar de explicar las diferentes características que nos ofrece el software BUSMASTER para el análisis del tráfico en el bus CAN. A continuación mostraremos las características más reseñables de BUSMASTER:

- Compatibilidad con Windows 2000, Windows XP y Windows 7; requiere la instalación del paquete MinGW, dado que Windows no incluye un compilador GCC de serie.
- Soporte de múltiples módulos USB para interfaz de bus CAN.
- Creación y edición de bases de datos CAN.
- Filtrado de mensajes por hardware o software.
- Registro y repetición de mensajes CAN.
- Creación de nodos programables usando un editor de funciones y lenguaje ANSI C.
- Filtros de importación para archivos de bases de dato DBC y programas CAPL.

2.3.1.1 – Modos de operación

Por defecto trabaja en modo activo.

- Activo: en este modo el controlador va a participar en la actividad del bus, es decir, que en la recepción de mensajes va a generar la señal de acuse de recibo (acknowledgement), señales de error, etc; y también será capaz de transmitir mensajes.
- Pasivo: en la actualidad este modo está desactivado.

2.3.1.3 – FILTRADO

Tenemos dos alternativas para hacer el filtrado de mensajes en BUSMASTER:

Filtro de aceptación: este tipo de filtrado se realiza mediante hardware. Nos permite aumentar la respuesta de la herramienta, pudiéndose configurar una alta tasa de mensajes. El controlador CAN filtra los mensajes en función de los valores establecidos en la máscara y en el código de aceptación. Este filtrado es más rápido que el realizado mediante software. Para configurar este tipo de filtrado debemos ir a CAN -> Channel Configuration. El driver será el que se encargue de configurar el controlador CAN para aplicar los filtros hardware deseados. Vale la pena señalar que esto implica poder configurar el controlador CAN a través de Arduino desde el ordenador, lo cual puede complicar significativamente el programa embebido dentro del Arduino.

Filtros App: este tipo de filtrado se realiza mediante software. Se puede optar por ver y grabar los mensajes de nuestra elección. El filtrado se puede aplicar a la ventana de visualización de mensajes, a los filtros y / o a la grabación de mensajes.

Podemos configurar la lista de filtros y elegir los mensajes a filtrar. Para configurar la lista de filtros, se siguen los siguientes pasos:

- Seleccionamos CAN -> Filter Configuration.
- Se muestra el cuadro de diálogo que se especifica en la Ilustración 16

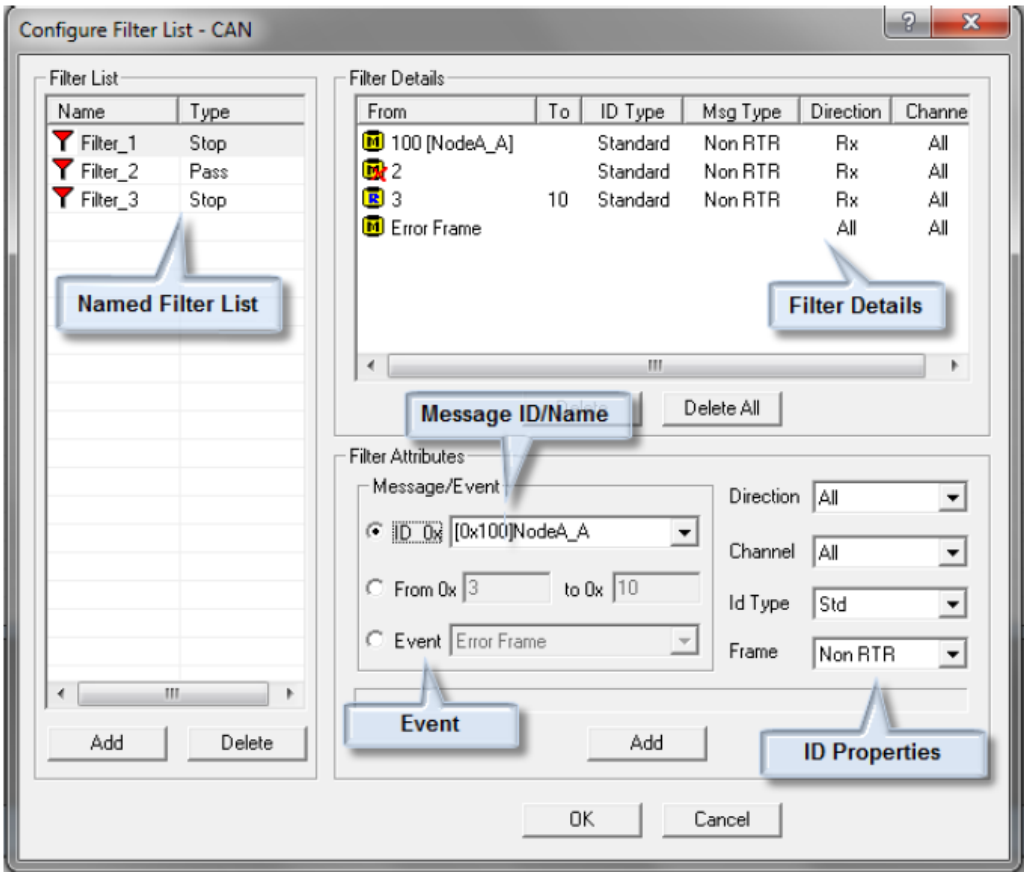


Ilustración 16: Lista de configuración de filtros

FILTER LIST -> Es una lista de filtros que se identifican por el nombre. El nombre del filtro debe ser único. El segundo parámetro indica el tipo de filtro, de paso o parada.

FILTER DETAILS -> En esta sección se muestra una lista con los nombres de los mensajes, identificador y rango junto con el tipo de identificación, tipo de mensaje, dirección y número de canal.

FILTER ATTRIBUTES -> Da más detalles del filtro seleccionado. El botón Add de la sección añadirá un filtro que esté configurado en la lista de filtros. Este botón se desactivará en el caso de que haya un parámetro no válido introducido por el usuario y se mostrará en la barra de estado el mensaje de error correspondiente.

Habilitar/Deshabilitar filtros: se puede hacer a través del menú CAN o mediante los iconos de la Ilustración 17. Se puede aplicar para filtrado, grabación o repeticiones de mensajes.

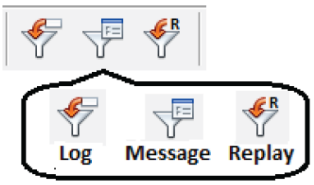


Ilustración 17: Habilitar/Deshabilitar filtros

2.3.1.4 – INTERACCIÓN CON MENSAJES

Grabación de mensajes: se puede seleccionar un archivo en el cual grabar todos los mensajes transmitidos y recibidos. Se puede configurar una sesión de grabación para formato decimal o para formato hexadecimal. Antes de iniciar la grabación de mensajes debemos configurar el tipo de grabación desde la ventana de la Ilustración 18. Para ello nos vamos a CAN --> Logging --> Configure.

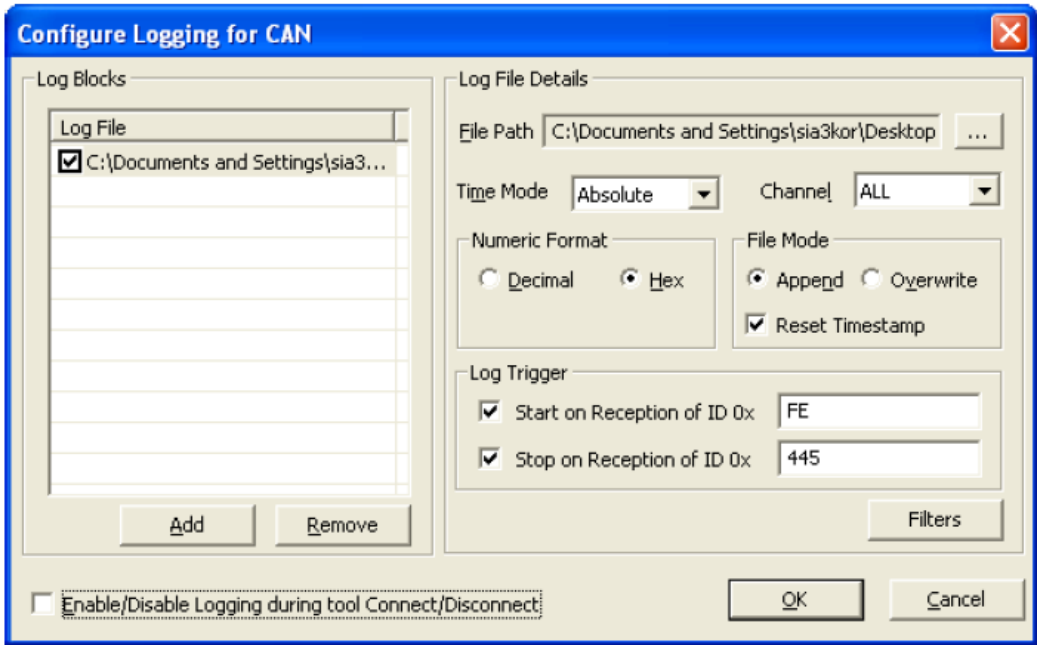


Ilustración 18: Configuración para grabar mensajes CAN

El registro de mensajes se puede realizar en tres modos de tiempo diferentes: “System time”, “Absolute time” y “Relative time”. En el modo “System time” el tiempo asociado a cada mensaje es el tiempo real del sistema. En el modo “Absolute time” la grabación se realiza con respecto al tiempo que se declaró durante la conexión. En el modo “Relative time” el tiempo asociado a un mensaje es el transcurrido respecto al mensaje del mismo tipo recibido previamente.

Así mismo debemos seleccionar la ruta en la cual queremos grabar el contenido del bus CAN.

Para iniciar la grabación de mensajes, pulsamos el botón de la barra de herramientas que se muestra en la Ilustración 19.

Ilustración 19: Botón para grabación de mensajes

Repetición de grabaciones: cuando tengamos alguna grabación almacenada y deseemos reproducirla tenemos que buscarla desde la ventana que aparece en la Ilustración 20. Para ello nos vamos a CAN --> Replay --> Configure.

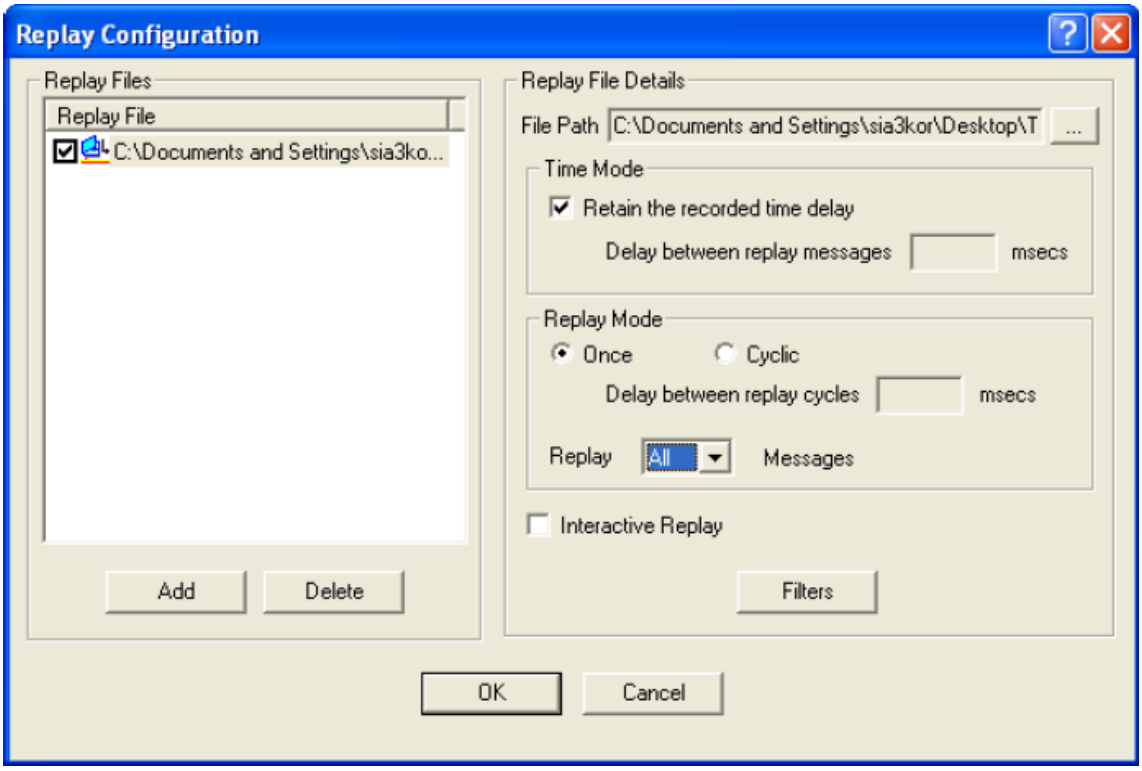


Ilustración 20: Configuración de reproducción de grabaciones

Seleccionaremos los archivos que deseemos reproducir. Tenemos dos tipos de reproducción:

- Interactive Replay: el contenido del archivo se muestra como una lista de mensajes y podemos navegar en esta lista y transmitir los mensajes de interés usando los botones de la Ilustración 21. Para desplegar la lista de mensajes debemos conectar la herramienta.
- Non Interactive Replay: La repetición empezará al conectarse la herramienta y se parará al desconectarse.

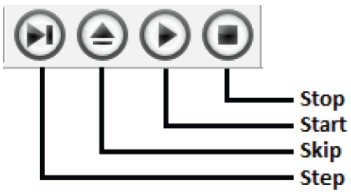


Ilustración 21: Botones para la reproducción de grabaciones en modo Interactivo

ERRORES OBSERVADOS: cuando hacemos una reproducción y ponemos el modo de tiempo relativo, observamos que los mensajes que se transmiten de forma cíclica se han grabado correctamente pero a la hora de reproducirse comenten un error. El valor del tiempo relativo no permanece constante sino que aparecen valores oscilantes.

Transmitir mensajes: Podemos definir mensajes para enviar en el bus CAN. Para configurar los mensajes debemos hacerlo desde el cuadro de la Ilustración 22, al cual accedemos mediante CAN -> Transmit -> Configure.

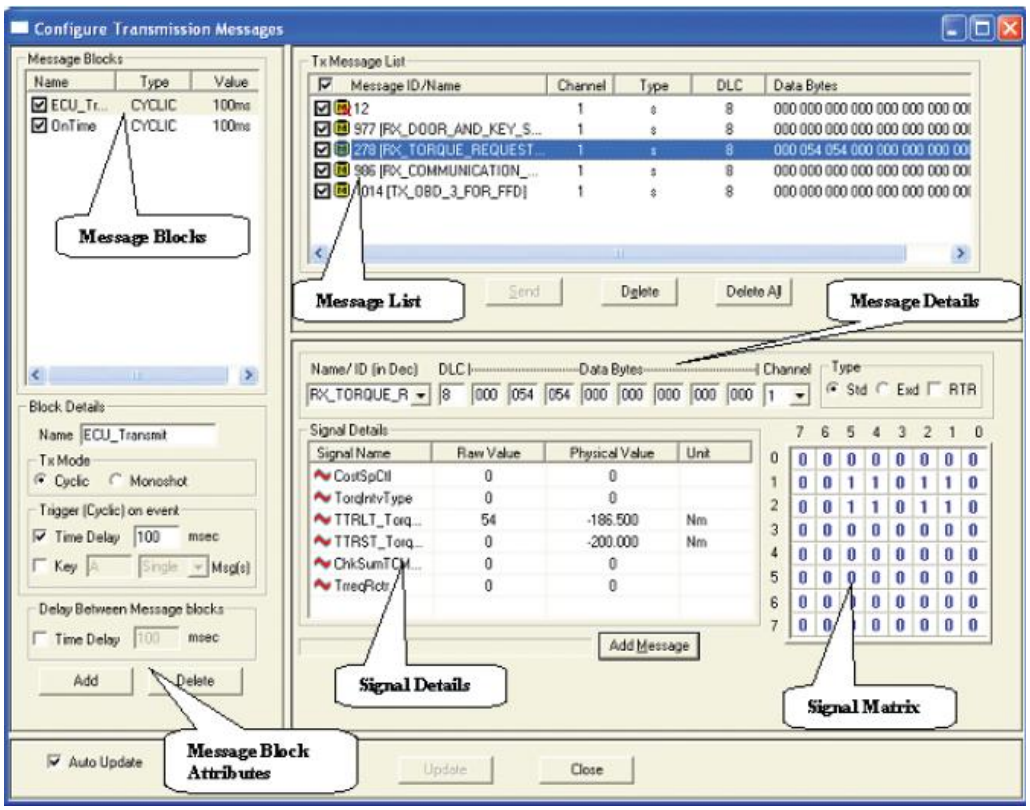


Ilustración 22: Configuración de transmisión de mensajes

Lo primero que tenemos que hacer es crear bloques de mensajes. En cada bloque tenemos una lista de mensajes, la cual podemos editar con mensajes que estén en la base de datos (para ello hay que tener una base de datos asociada) o crear nosotros un mensaje nuevo. Además, los mensajes que se envían se pueden definir como mensajes standard, mensajes extended o como mensajes RTR.

Veremos la salida en Message Window cuando inicialicemos el controlador, es decir, cuando conectemos la herramienta y cuando iniciemos la transmisión pulsando en CAN -> Transmit -> Enable.

2.3.1.5 – Nodos simulados

Podemos crear nodos simulados los cuales pueden ser configurados bajo el bus CAN. Para crear simulaciones tenemos la ventana de la Ilustración 23, la cual se puede abrir desde CAN --> Node Simulation --> Configure.

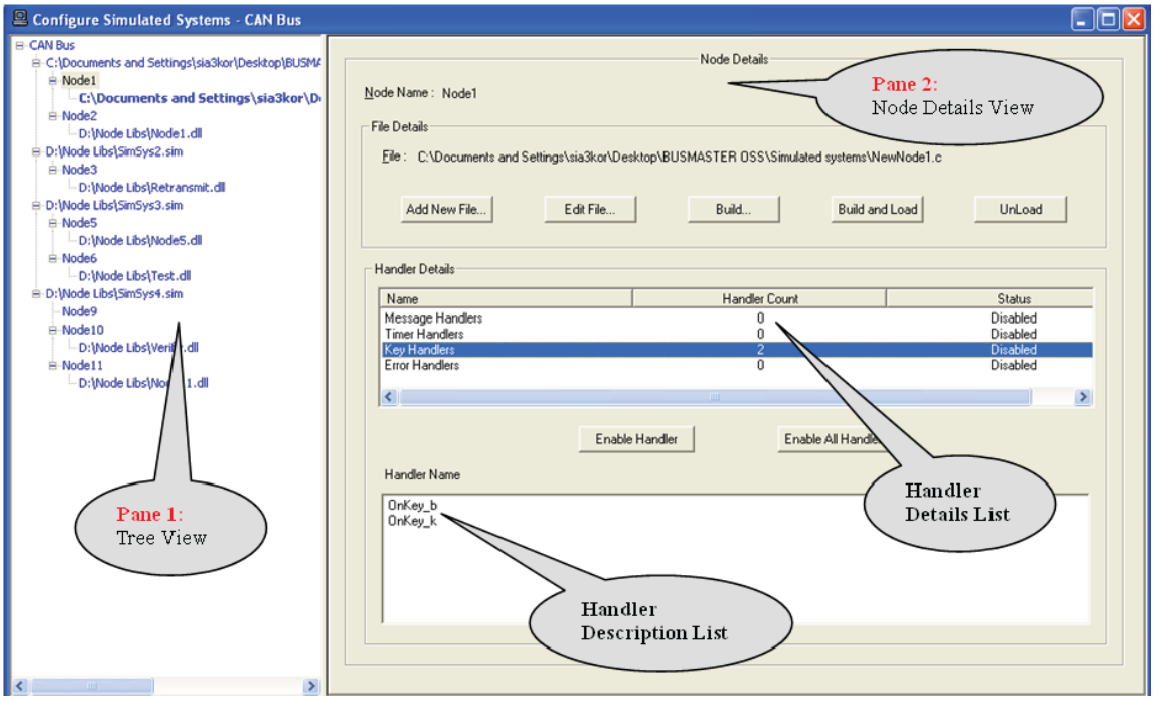


Ilustración 23: Configuración de sistemas simulados

Cada nodo que creemos va tener asociada una librería dinámica DLL. Podremos tener varios nodos creados y funcionando simultáneamente.

Editor de funciones: Busmaster puede funcionar como un nodo programable en el bus CAN. Desde el editor de funciones podemos programar diferentes controladores de eventos. El lenguaje de programación es C.

Podemos programar los siguientes controladores de eventos:

CONTROLADOR DE EVENTO	EJECUTADAS EN
Message Handlers	La recepción de un mensaje
Timer Handlers	Transcurso de un intervalo de tiempo
Key Handlers	Pulsar una tecla
Error Handlers	Detección de error o cambio en el estado de error
DLL Handlers	Carga / descarga de DLL

Tabla 6: Controladores de eventos

También se pueden incluir nombres de archivos de cabecera, añadir variables globales y funciones de utilidad mientras se programan los controladores de eventos. Todas estas funciones se pueden editar y guardar en un archivo con extensión ".c". En la Ilustración 24 se muestra la estructura del editor de funciones.

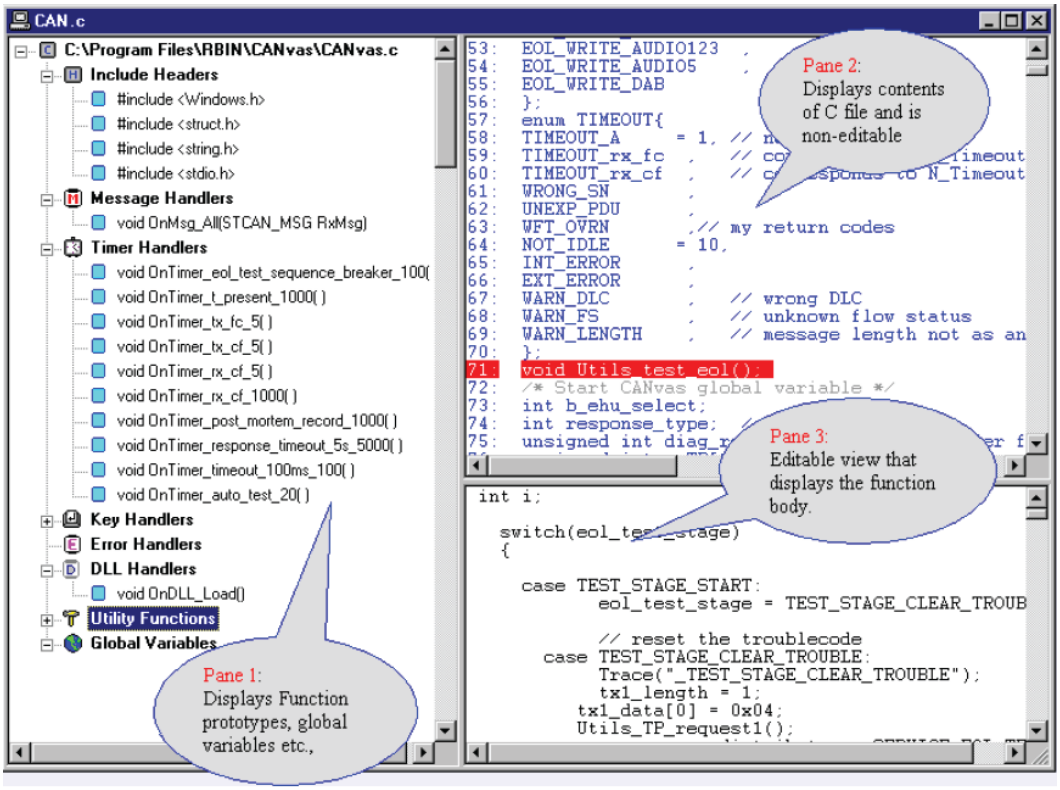


Ilustración 24: Editor de funciones de BUSMASTER

Simulación de nodos: Una vez que el programa está escrito se puede compilar como un archivo DLL y se puede cargar. Utilizaremos el compilador GCC[1].

Una vez hecho esto podemos activar los controladores de eventos de manera individual o todos a la vez mediante los botones de la Ilustración 25.

[1] Para instalar este compilador seguiremos las instrucciones de la documentación "Help" proporcionada en el repositorio de BUSMASTER. Utilizaremos el procedimiento MinGW Installation using TDM-GCC Installer.

Ilustración 25: Activar controladores de eventos

2.3.1.6 – EDITOR DE LA BASE DE DATOS

La base de datos de BUSMASTER consiste en información sobre el mensaje esperado. Podemos crear una base de datos con los mensajes que se van a transmitir o recibir a través del bus CAN y modificar las señales asociadas a cada mensaje. Seremos capaces de crear nuevos mensajes y nuevas señales, las cuales serán exclusivas para cada mensaje.

Gracias a los convertidores de formato, podremos utilizar las bases de datos que tengamos de CANoe haciendo la conversión desde archivos *.dbc (propios de las bases de datos de CANoe) a archivos *.dbf (correspondientes a las bases de datos de BUSMASTER).

A diferencia de BUSMASTER, CANoe utiliza bases de datos relacionales teniendo como objetos networks, ECUs, nodes, signals, messages y environment variables. Al hacer un cambio de formato desde archivos *.dbc a archivos *.dbf, sólo se mantendrá la información compatible con las bases de datos de BUSMASTER, la cual es cada uno de los mensajes y las señales asociadas a cada mensaje.

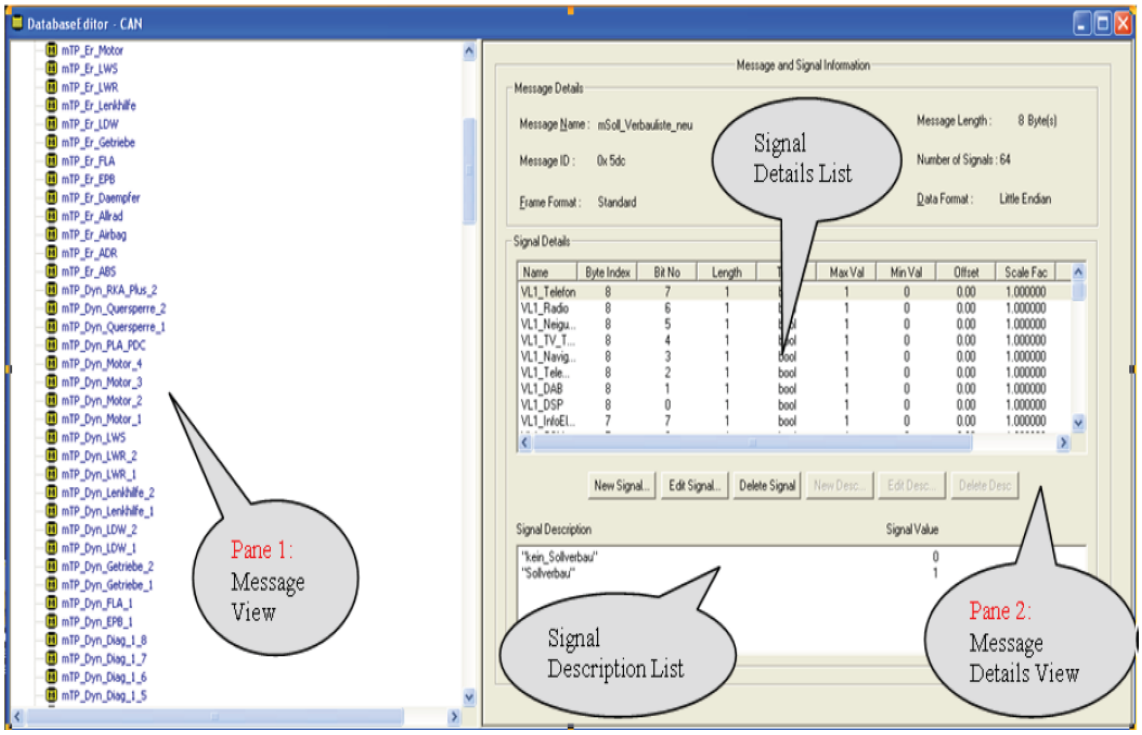


Ilustración 26: Editor de bases de datos de BUSMASTER

Para abrir el editor de bases de datos de la Ilustración 26, tenemos que seleccionar CAN -> Database -> New. Características de los mensajes en la base de datos:

- Cada mensaje tiene un identificador y nombre único.
- Cada mensaje tiene un máximo de ocho bytes de datos.

- Se puede definir la longitud del mensaje en bytes.
- Cada mensaje puede constar de una o más señales.
- La señal puede tener un offset (desplazamiento), un factor de multiplicación y unidades de ingeniería. Estas tres informaciones juntas se utilizan para mostrar el valor de la señal recibida en unidades de ingeniería.

2.3.1.7 – VENTANAS

Message Window: BUSMASTER nos informa de todo tipo de mensajes a través de esta ventana. Un mensaje puede surgir de cualquiera de las dos categorías que se mencionan a continuación:

- Mensaje transmitido a través del bus CAN, incluyendo el generado por Busmaster.
- Mensajes de error.

Como podemos ver en la Ilustración 27, cada mensaje tiene diferentes campos que nos dan la siguiente información:

- Time: El tiempo puede ser visto en tres modos diferentes:
  - System - En este modo, el mensaje se mostrará con el tiempo del PC / Sistema.
  - Relative - En este modo, el mensaje se mostrará con el tiempo desde que se recibió el mensaje previo con el mismo identificador.
  - Absolute - En este modo, la referencia es el momento de la conexión.
- Tx / Rx - Un mensaje transmitido desde Busmaster está etiquetado como Tx mientras que para un mensaje recibido por Busmaster está etiquetado como Rx.
- Channel – Muestra el canal por el que se reciben las tramas.
- Type - Indica si el mensaje es de tipo standard, extended o RTR, la convención seguida es:
  - S – Standard frame
  - X – Extended frame
  - Sr – Standard RTR frame
  - Xr – Extended RTR frame
- ID – Muestra el ID del mensaje.
- Message – Contiene el nombre asociado al ID del mensaje recibido.
- DLC - Es la abreviatura de Data Length Count. Se muestra el número de bytes de datos en el cuerpo del mensaje.
- Data Byte(s) - Se visualizan los datos de los bytes ya sea en hexadecimal o en modo decimal.

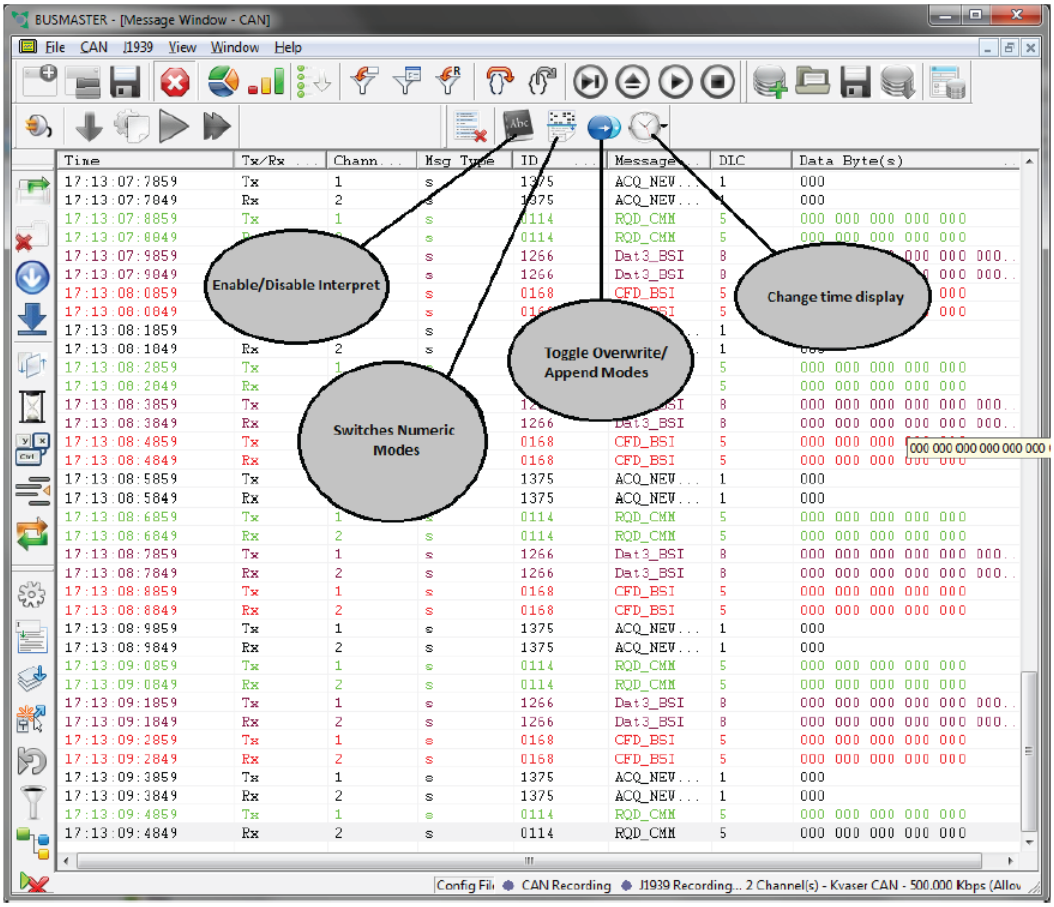


Ilustración 27: Message Window

Pulsando a expandir (+) o comprimir (-) en un mensaje concreto podemos mostrar los valores de la señal en valores Físicos o Puros. Haciendo doble clic en un mensaje se abrirá un dialogo de interpretación para ese mensaje concreto.

Trace Window: Da detalles sobre el resultado de la última operación. El resultado puede ser información, advertencia o error.

2.3.1.8 – SEÑALES

Generación de señales: Podemos configurar cada señal en los mensajes de la base de datos con una forma de onda particular y enviar los mensajes con estos valores de señal en un período de tiempo de muestreo particular.

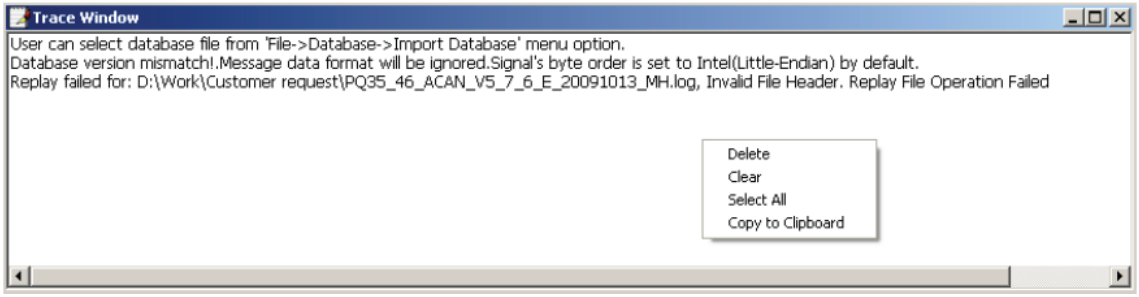


Ilustración 28: Trace Window

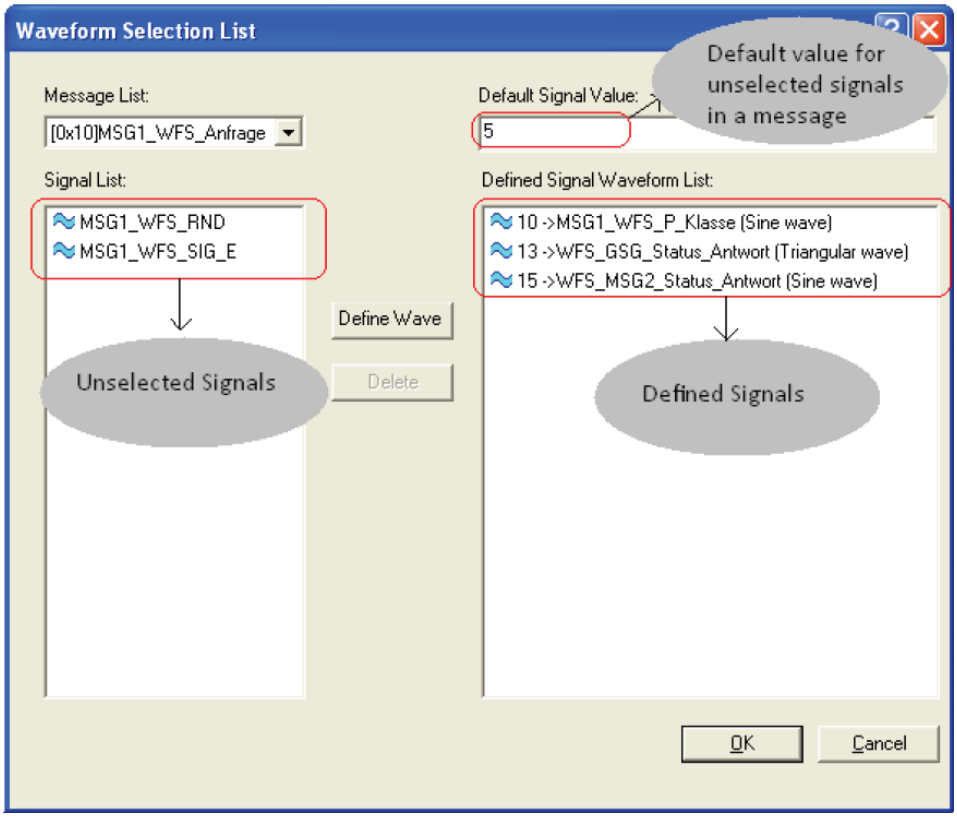


Ilustración 29: Configuración de señales en BUSMASTER

Actualmente sólo dos formas de onda son posibles, onda sinusoidal y triangular.

Para configurar las señales, vamos a la opción de menú, CAN -> Signal Graph -> Configure y se mostrará cuadro de diálogo de la Ilustración 29.

Tendremos una lista con los mensajes de la base de datos asociada en ese momento y una lista con las señales del mensaje seleccionado.

A cada señal podemos darle la forma de onda y valor deseado.

Si queremos transmitir las señales definidas, tenemos que asegurarnos de que la aplicación está en estado conectado y la opción de menú CAN -> Waveform Messages -> Enable esté activada.

Visualizar una señal: Podemos ver el valor de una señal utilizando la ventana de visualización de la señal siempre y cuando se haya recibido un mensaje que tiene esa señal. Los valores Physical y Raw serán listados y actualizados a medida que el mensaje llegue. Haga clic en el botón de la barra de herramientas de la Ilustración 30 para mostrar la ventana de visualización de la señal.

Ilustración 30: Abrir ventana de visualización de la señal en BUSMASTER

Gráfico de la señal: BUSMASTER soporta el trazado gráfico de valores de señales y parámetros estadísticos. Esto incluye valores brutos y físicos (raw and physical) de una señal. Se pueden añadir para trazar en el gráfico los parámetros de estadísticas de la red. Los datos del gráfico se pueden exportar en varios formatos. Podemos acceder a ello a través de CAN --> Signal Graph --> Activate.

2.3.1.9 – Automatización de pruebas

Es un proceso para optimizar el esfuerzo en las pruebas donde sólo tenemos que definir los casos de prueba (test cases) en lugar de escribir los códigos en el nodo. Los casos de prueba se pueden tomar directamente como parámetros de entrada para la ejecución de una sesión de evaluación llevada a cabo por el módulo de automatización de pruebas (Test Automation Module) de Busmaster.

La automatización de pruebas es una extensión de la herramienta para simplificar el proceso de llevar a cabo pruebas de un género y generar el informe.

2.3.1.10 – Convertidores de formato

BUSMASTER nos permite hacer las siguientes conversiones de formato:

- CAPL To CPP Converter: Converts CAPL (*.can) file into BUSMASTER CPP (*.cpp) file
- DBC To DBF Converter: Converts DBC (*.dbc) file into BUSMASTER database (*.dbf) file
- DBF To DBC Converter: Converts BUSMASTER database (*.dbf) file into DBC (*.dbc) file
- ASC TO LOG Converter: Converts CANoe log file (*.asc) to BUSMASTER log file(*.log)
- LOG To ASC Converter: Converts BUSMASTER log file (*.log) to CANoe Log file (*.log)
- Log To Excel Converter: Exports BUSMASTER log file (*.log) to CSV (Comma Seperated Values) Format
- BLF To LOG Converter: Converts BLF (*.blf) file into BUSMASTER log file (*.log)

2.3.2 – Caso práctico

En esta sección hemos tratado de ejemplificar las funcionalidades del software BUSMASTER con las herramientas disponibles en el Aula Mercedes. Crearemos un nuevo sistema simulado en el cual programaremos un nodo para que envíe cada tres segundos un mensaje que previamente crearemos y añadiremos a nuestra base de datos. Después comprobaremos en Message Window su correcto funcionamiento.

Para ello hemos hecho uso del hardware CANcaseXL de Vector con lo que previamente ha de instalarse el driver de Vector para el CANcaseXL[2].

[2] Instalar “Vector XL Driver Library”, disponible en el repositorio de BUSMASTER, en C:\XL Driver Library.

2.3.2.1 – Configuración inicial

Lo primero que tenemos que tener claro son los pasos previos para hacer funcionar una simulación. Debemos de seguir las siguientes indicaciones:

1) Crear una nueva configuración de BUSMASTER en la carpeta deseada.
2) Para poder hacer la compilación del nodo en el bus CAN tenemos que hacer uso del Makefile que se encuentra en la carpeta de instalación de BUSMASTER: GCCDLLMakeTemplate_CAN[3]. Tenemos dos alternativas:
    a. Cuando cargamos la configuración de BUSMASTER directamente desde BUSMASTER (File/Load…), se utiliza el Makefile que se encuentra en la carpeta de instalación de BUSMASTER.
    b. Cuando cargamos la configuración de BUSMASTER haciendo doble clic en el archivo "nuestraConfiguracion.cfx", la compilación se realizará mediante el Makefile que tengamos en la misma ubicación que el archivo ".cfx". Por este motivo debemos copiar el Makefile GCCDLLMakeTemplate_CAN en esa ubicación.
3) Asegurarse de tener copiadas las cabeceras del directorio "<INSTALLDIR>\SimulatedSystems\include" dentro de la carpeta "include" del paquete MinGW.
4) Modificar el contenido del Makefile GCCDLLMakeTemplate_CAN (ver Anexo página 97) para poder acceder a las cabeceras y a las librerías de los sistemas simulados de Busmaster. En caso contrario puede dar errores de compilación dado que el Makefile hace uso de la variable global del Makefile <INSTALLDIR>. Tenemos que cambiar la expresión <INSTALLDIR> por la ruta de instalación de BUSMASTER dentro del Makefile GCCDLLMakeTemplate_CAN para que sea interpretado por nuestro compilador. En nuestro caso hay que modificar la expresión <INSTALLDIR> por la siguiente expresión "C:\Program Files (x86)\BUSMASTER_v2.6.3" donde tenemos instalado BUSMASTER.
5) Para poder ejecutar el compilador G++ de MinGW tenemos que añadir la ruta de la carpeta C:\MinGW\bin\ a la variable de entorno Path[4].
6) Configurar los drivers, canales y bases de datos que se vayan a utilizar.
7) Añadir un nuevo sistema simulado en la ruta deseada.
8) Añadir un nuevo nodo en la ruta deseada con la DLL asociada al nodo y si es un nodo nuevo y no tenemos una DLL asociada no añadiremos ninguna, se añadirá sola una vez que se compile y se genere la DLL correctamente. Usar nombres para el nodo como "Nodo.cpp" puede producir errores.
9) Programar los controladores y después compilarlos (Build) y subirlos (Load).
10) Activar los controladores necesarios para que funcione el sistema simulado.

[3] Consultar la sección de Anexos para ver el formato del Makefile GCCDLLMakeTemplate_CAN
[4] Consultar la sección de Anexos para más información.

2.3.2.2 – Nodo simulado

Vamos a crear una base de datos con un mensaje nuevo y enviarlo cada tres segundos mediante un controlador de tiempo.

Creo una nueva configuración y ajusto los parámetros iniciales:

- Elegimos el driver de Vector instalado: CAN->Driver Selection->Vector XL
- Elegimos el canal: CAN->Channel Selection. Con CAN 1 seleccionado pinchamos en Select y nos selecciona el Hardware de CAN 1.

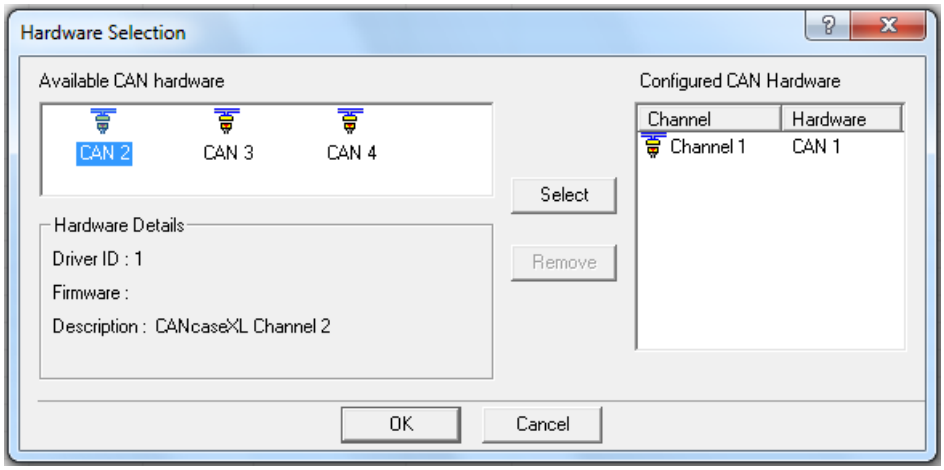


Ilustración 31: Selección del canal en BUSMASTER

- En el canal 1, que es el que hemos seleccionado, marcamos los baudios dependiendo del canal en el que estemos (CAN B, CAN D, CAN C). Para ello vamos a CAN->Channel Configuration y tendremos algo como en la Ilustración 32.

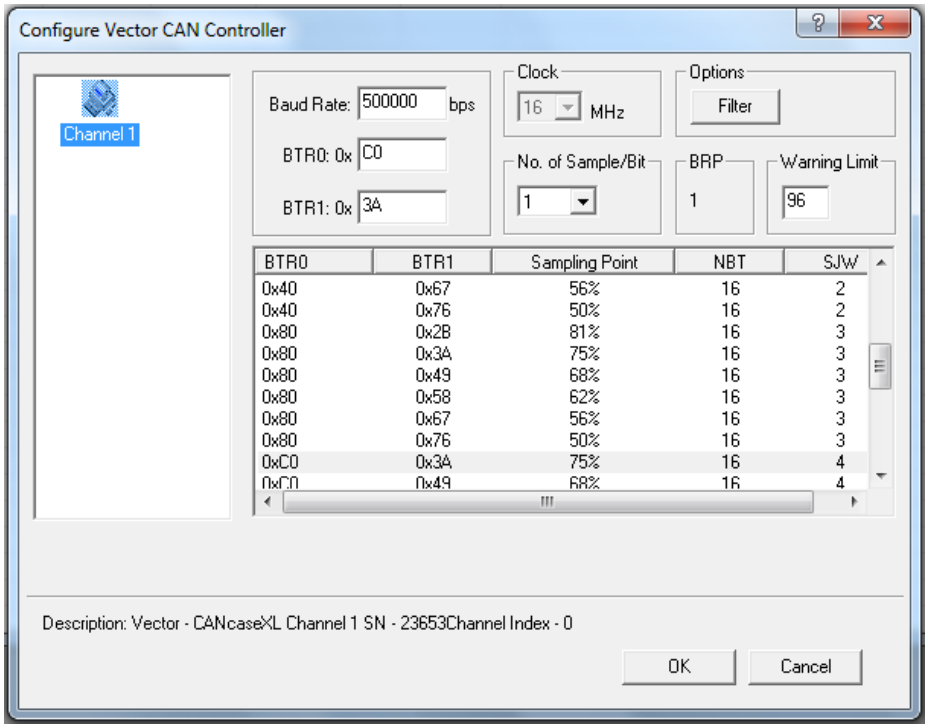


Ilustración 32: Configuración del canal seleccionado

- Asociamos las bases de datos que necesitemos en nuestra configuración mediante CAN->Database->Associate.

Creamos una nueva base de datos con un mensaje asociado:

- Vamos a CAN->Database->New y generamos la nueva base de datos en la carpeta deseada.

- Se nos abrirá el editor de la base de datos. En él pincharemos sobre la dirección de nuestra base de datos creada con el botón derecho del ratón y pinchamos en New message. Tras hacer esto tendremos algo como la Ilustración 33.

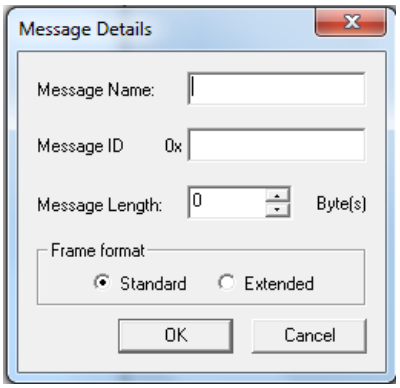


Ilustración 33: Nuevo mensaje de la base de datos

- Rellenamos el nombre del mensaje, ID, número de bytes y tipo de formato de trama. Tras hacer esto tendremos algo como la Ilustración 34. Si pinchamos sobre el mensaje creado, nos aparecerá a la derecha de la ventana una descripción del mensaje y las señales que incluye; junto con la posibilidad de crear nuevas señales o editarlas.

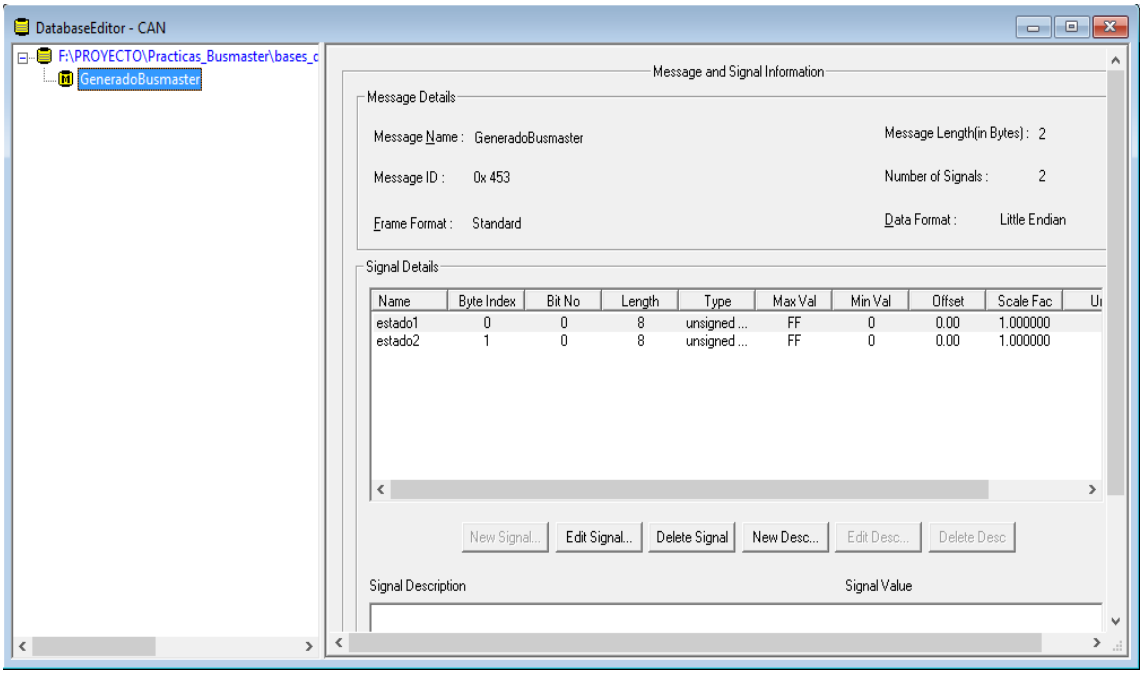


Ilustración 34: Editor de bases de datos

- Creamos una nueva señal. Para ello pinchamos en New Signal y tendremos la Ilustración 35. En esta ventana definimos los detalles de la señal que queremos crear. En el momento que generemos señales suficientes para llenar el tamaño del mensaje, ya no nos permitirá crear nuevas señales para este mensaje.

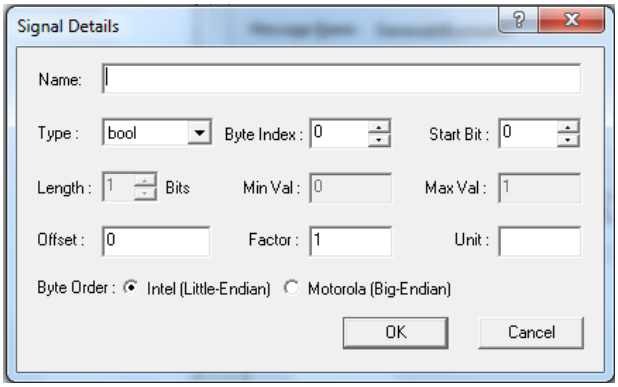


Ilustración 35: Detalles de la señal de BUSMASTER

A continuación crearemos un nuevo sistema simulado y un nodo asociado. Una vez hecho esto podremos seleccionar la opción "Edit File" en la que nos aparecerá el editor de funciones con las cabeceras necesarias ya incluidas.

Para crear un controlador tenemos que clicar con el botón derecho del ratón sobre el controlador que deseemos añadir y seleccionar Add. En cada caso nos pedirá completar con la información requerida.

Programamos un controlador de tiempo con 3000 ms (3 segundos) y añadimos el código de la Ilustración 36. Este código nos generará un mensaje Standar con ID = 0x20 y con una longitud de 8 bytes. Se inicializará el mensaje con unos valores predeterminados.

```
/* Start BUSMASTER generated function - OnTimer_myTimer_3000 */
void OnTimer_myTimer_3000( )
{
/* TODO */
STCAN_MSG sMsg;
sMsg.m_unMsgID = 0x20; // Message ID
sMsg.m_ucEXTENDED = FALSE; // Standard Message type
sMsg.m_ucRTR = FALSE; // Not RTR type
sMsg.m_ucDLC = 8; // Length is 8 Bytes
sMsg.m_sWhichBit.m_aulData[0] = 45; // Lower 4 Bytes
sMsg.m_sWhichBit.m_aulData[1] = 65462; // Upper 4 Bytes
sMsg.m_ucChannel = 1;
SendMsg(sMsg);
}/* End BUSMASTER generated function - OnTimer_myTimer_3000 */
```

Ilustración 36: Código para enviar mensaje desde BUSMASTER

Debemos activar los controladores de eventos correspondientes y conectar la herramienta pulsando el botón de la Ilustración 37.

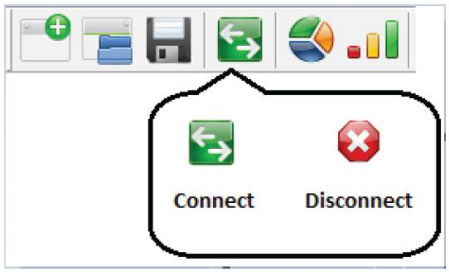


Ilustración 37: Conectarse al bus desde BUSMASTER

La salida en la ventana Message Window (Ilustración 38) es la de un mensaje de tipo Standard, con longitud de 8 bytes, con dichos bytes inicializados, que se transmite por el canal 1, con ID 0x20 y que se transmite cada 3 segundos.

	00:00:21:3065	Rx	1	s	0x1C1	0x1C1	4	00 00 00 00
⊞	00:00:21:3090	Rx	1	s	0x3E7	TELEAID_POS3	8	5D 4E FD F9 4B 7C A0 7F
⊞	00:00:21:3190	Rx	1	s	0x3E5	TELEAID_POS1	8	0C 00 00 00 01 FF FF 00
⊞	00:00:21:3490	Rx	1	s	0x3E3	TP_AGW_TELEAID6	8	59 99 99 12 E0 B6 0A 0D
	00:00:21:5081	Rx	1	s	0x020	0x20	8	2D 00 00 00 B6 FF 00 00
⊞	00:00:20:8198	Rx	1	s	0x39F	IC_A5	8	00 31 0D 01 01 07 0A FF
⊞	00:00:20:8248	Rx	1	s	0x19F	IC_A1	8	0F FF FF F0 31 01 86 A0
⊞	00:00:20:8661	Rx	1	s	0x208	MSSK_A2	8	04 03 00 00 00 00 00 A9
⊞	00:00:20:8910	Rx	1	s	0x207	CVI	8	10 C3 7F 7F FF 00 EF D1

Ilustración 38: Tráfico en Message Window

Si modificamos el identificador del mensaje de la Ilustración 36 de 0x20 a 0x453 conseguimos enviar el mensaje creado anteriormente en nuestra base de datos y modificar el valor de sus señales como se muestra en la Ilustración 39.

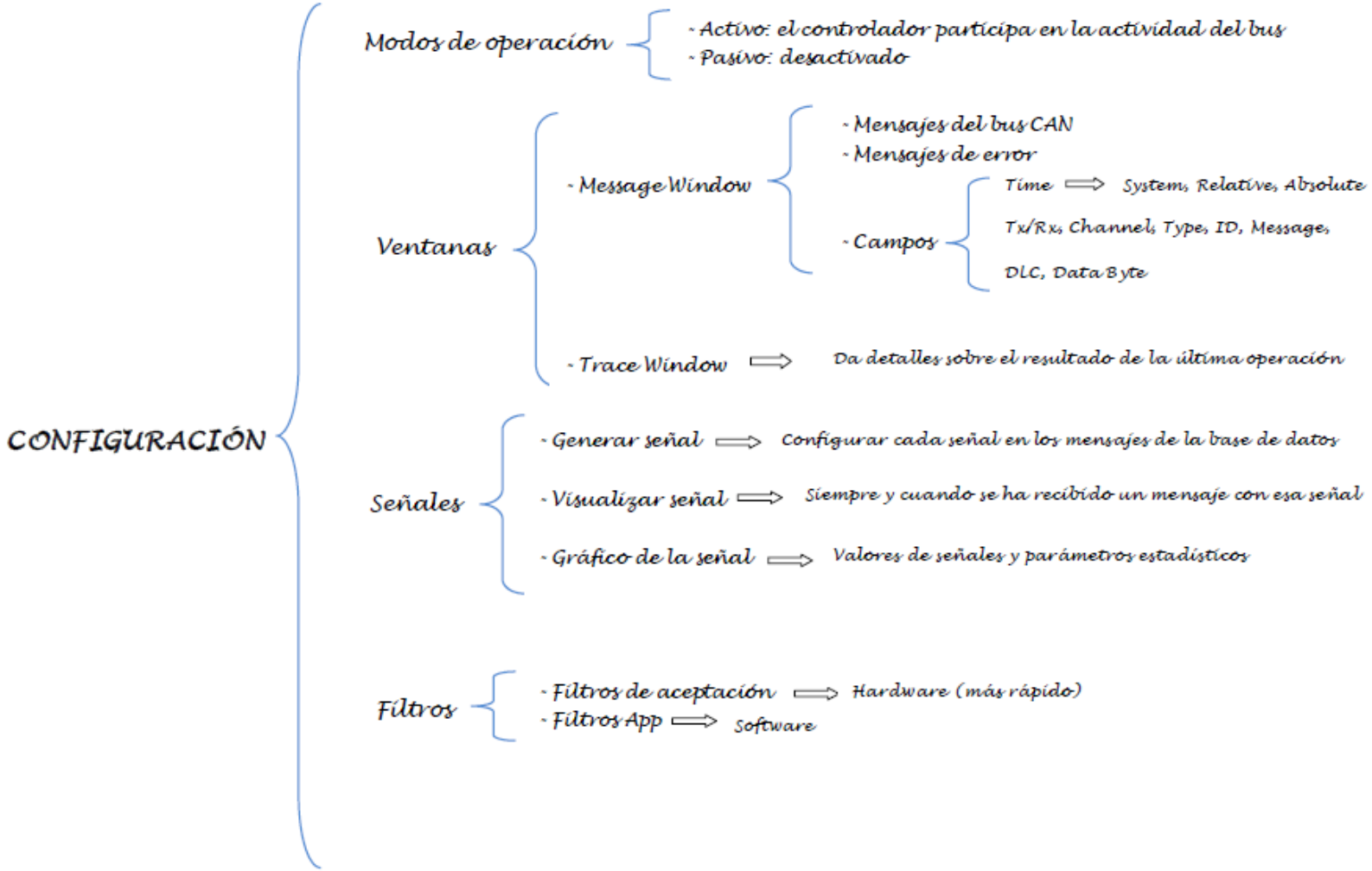
⊞	00:00:27:5750	Rx	1	s	0x207	CVI	8	10 C3 7F 7F FF 00 EF D1
⊞	00:00:27:5950	Rx	1	s	0x206	AVI	8	00 00 00 00 00 C0 FA 84
	00:00:24:9263	Rx	1	s	0x020	0x20	8	2D 00 00 00 B6 FF 00 00
⊟	00:00:26:9253	Rx	1	s	0x453	GeneradoBusmaster	8	2D 00 00 00 B6 FF 00 00
	estado1		0x2D	45.000				
	estado2		0x0	0.000				
⊞	00:00:26:9305	Rx	1	s	0x3EB	NAVI_POS2	8	00 00 00 00 FF FF 0E 25
⊞	00:00:26:9406	Rx	1	s	0x3E9	TELEAID_POS5	8	07 D0 01 01 00 39 10 CC

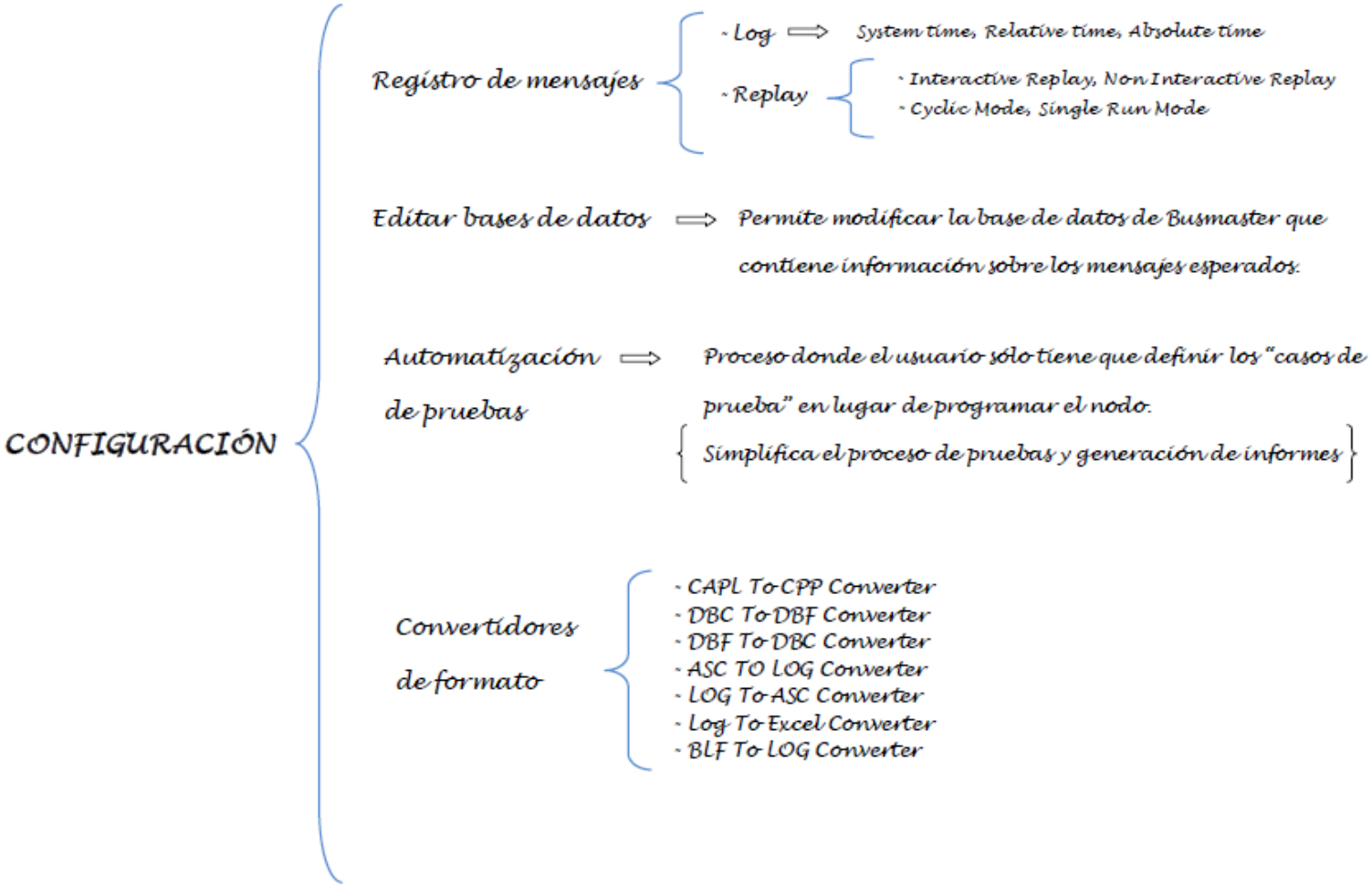
Ilustración 39: Mensaje de la base de datos en Message Window

2.3.3 – Diagramas

A continuación hemos tratado de esquematizar, de una forma análoga a lo hecho en CANoe, las características y funcionalidades que nos ofrece BUSMASTER en el protocolo de comunicaciones CAN.

BUSMASTER
Hardware
- CANcardXL
- CANcaseXL
- CAN PCMCIA family
- CAN PCI family
Protocolos
- CAN
- J1939
- FlexRay
- LIN
Configuración
- Modos de operación
- Ventanas
- Filtros
- Registro de mensajes
- Convertidores de formato
- Editar base de datos
- Señal
- Automatización de pruebas
Sistemas de Simulación
- Handlers
- Editor de funciones
- Simulación de nodos


CONFIGURACIÓN
Modos de operación
- Activo: el controlador participa en la actividad del bus
- Pasivo: desactivado
Ventanas
- Message Window
- Mensajes del bus CAN
- Mensajes de error
- Campos
Time ⟹ System, Relative, Absolute
Tx/Rx, Channel, Type, ID, Message,
DLC, Data Byte
- Trace Window ⟹ Da detalles sobre el resultado de la última operación
Señales
- Generar señal ⟹ Configurar cada señal en los mensajes de la base de datos
- Visualizar señal ⟹ Siempre y cuando se ha recibido un mensaje con esa señal
- Gráfico de la señal ⟹ Valores de señales y parámetros estadísticos
Filtros
- Filtros de aceptación ⟹ Hardware (más rápido)
- Filtros App ⟹ Software


CONFIGURACIÓN
Registro de mensajes
- Log ⟹ System time, Relative time, Absolute time
- Replay
- Interactive Replay, Non Interactive Replay
- Cyclic Mode, Single Run Mode
Editar bases de datos ⟹ Permite modificar la base de datos de Busmaster que contiene información sobre los mensajes esperados.
Automatización de pruebas ⟹ Proceso donde el usuario sólo tiene que definir los "casos de prueba" en lugar de programar el nodo.
Simplifica el proceso de pruebas y generación de informes
Convertidores de formato
- CAPL To CPP Converter
- DBC To DBF Converter
- DBF To DBC Converter
- ASC TO LOG Converter
- LOG To ASC Converter
- Log To Excel Converter
- BLF To LOG Converter

SISTEMAS DE SIMULACIÓN

- **Handlers** ejecutados en:
  - Message Handlers
  - Timer Handlers
  - Key Handlers
  - Error Handlers
  - DLL Handlers
  - La recepción de un mensaje
  - Transcurso de intervalo de tiempo
  - Pulsar una tecla
  - Detección/cambio de un error
  - Carga/descarga de DLL
- **Editor de funciones** ⟹
  - Permite programar los controladores de eventos en C, incluyendo archivos de cabecera, variables globales y funciones de utilidad.
  - Archivo de origen construido como un archivo DLL
- **Simulación de nodos**
  - Compilador GCC ⟹ compila y carga archivos DLL
  - Opciones:
    - Compilar
    - Compilar y cargar
    - Cargar
    - Descargar

2.4 – Comparativa CANoe vs BUSMASTER

A continuación, en la Ilustración 40, hemos mostrado una comparativa entre CANoe y BUSMASTER con las funcionalidades que creemos son interesantes para nuestros objetivos.

Podemos sacar en conclusión que BUSMASTER cubre las funcionalidades de CANoe a excepción de un entorno gráfico o modelado, como los paneles que ofrece CANoe a través del Panel Designer.

		CANoe	Busmaster	Ubicación CANoe	Ubicación Busmaster
Ventanas:		Trace Window	Message Window	View -> Trace	CAN -> Message Window -> Activate
		Write Window	Trace Window	View -> Write	Window -> Trace Window
		Statistics Window	Network Statistics	View -> Bus Statistics	CAN -> Network Statistics
		Data Window	Signal Watch List	View -> Data	Show/Hide Signal Watch
		Graphics Window	Signal Graph	View -> Graphics	CAN -> Signal Graph -> Activate
Transmitir mensaje desde el software		Interactive Generator Block CAN	Transmit Messages	View -> Simulation Setup-> Clic derecho en cable rojo de Network CAN -> Insert Interactive Generator Block	CAN -> Transmit -> Configure
Simulación		Simulation Setup	Configure Simulated Systems	View -> Simulation Setup	CAN -> Node Simulation -> Configure
Filtros	Software	Filter Block	Filtros App	View -> CAN Filter	CAN -> Filter Configuration
	Hardware	CAN controllers	Filtros de aceptación	___	___
Modos de operación		Online: Bus real	Activo	View -> Measurement Setup -> Online	CAN -> Driver Selection -> Vector XL
		Online: Bus simulado	Activo + Sistema Simulado	View -> Measurement Setup -> Online	CAN -> Driver Selection -> Vector XL
		Offline	Activo	View -> Measurement Setup -> Offline	CAN -> Driver Selection -> Simulation
Modelado		Panel Designer	___	Mediante variables de entorno	___
Lenguaje de programación		CAPL	ANSI C	___	___
Base de datos	Editable	Si	Si	___	___
	Mensajes y señales	Si	Si	___	___
	Plataforma	CANdb++ Editor	Database Editor	Tools -> CANdb++ Editor	CAN -> Database -> Open
Automatización de pruebas		Interactive Generator	Test Setup File	View -> IG	___
		Replay Block	Test Setup Editor	View -> Simulation Setup -> Clic derecho -> Insert Replay Block CAN	CAN -> Test Automation -> Editor
		___	Test Executor	___	CAN -> Test Automation -> Executor
Protocolos de comunicación		CAN	CAN	___	___
		LIN	LIN	___	___
		MOST		___	___
		FlexRay	FlexRay	___	___
		Ethernet		___	___
		J1708		___	___
		WLAN		___	___

Ilustración 40: Comparativa CANoe vs BUSMASTER

2.5 – Desarrollo de una GUI para BUSMASTER

Uno de los objetivos marcados en este Trabajo Fin de Grado es equiparar a BUSMASTER con las funcionalidades ofrecidas por CANoe en el Aula Mercedes Benz de la Universidad de Valladolid. Por este motivo hemos tratado de diseñar una Interfaz Gráfica de Usuario (GUI) con la que poder hacer simulaciones a través de la herramienta BUSMASTER.

Hemos elegido Qt Creator como IDE para desarrollar los paneles de BUSMASTER por ser multiplataforma y disponer de un entorno de diseño intuitivo como el Panel Designer que nos ofrece CANoe. Este entorno de diseño se llama Qt Designer la cual es una herramienta muy potente que permite crear "widgets" o elementos gráficos de manera visual. Tras diseñar de manera gráfica un widget, se genera un archivo con extensión '.ui', que no es más que un archivo XML que posteriormente se traduce a una clase en C++.

2.5.1 – Dificultades

La gran dificultad de este apartado ha sido la imposibilidad para realizar una compilación secuencial de los dos programas creados a través del Makefile de BUSMASTER. Para poder entender esta dificultad, y dado que nos vamos a basar en esta plataforma, es importante mencionar las bases de la programación de eventos en Qt.

Señales y slots: son las herramientas que utiliza Qt para gestionar los eventos al programar un interfaz gráfico de usuario. Una señal en Qt se emite cuando ocurre un evento en particular. Los componentes de Qt tienen muchas señales predefinidas, pero podemos crear componentes derivados y añadir nuestras propias señales. Un slot es una función que es llamada como respuesta a una señal en concreto. Los componentes de Qt tienen muchos slots predefinidos, pero lo normal es añadir nuestros propios slots para manejar aquellas señales que nos interesen.

Un objeto no sabe si alguien recibirá sus señales al igual que no sabe si será llamado alguno de sus métodos como consecuencia de la generación de una señal. Se pueden conectar varias señales a un mismo slot, y una señal se puede conectar a varios slots distintos. Incluso se pueden encadenar señales, para que una se produzca como reacción a la primera.

Como se muestra en la Ilustración 41, las señales y slots pueden ser interconectados libremente a través del método 'connect'.

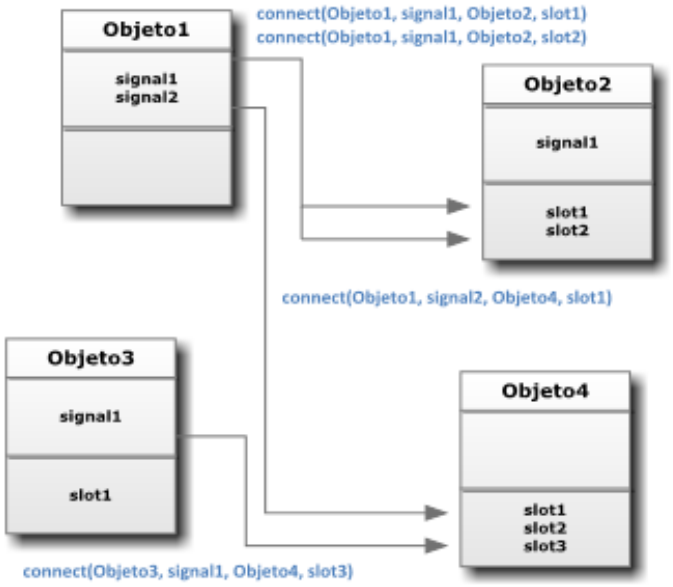


Ilustración 41: Conexiones entre señales y slots

MOC (Meta Object Compiler): o compilador de metaobjetos crea metaobjetos que describen las clases haciendo uso de las señales y slots.

Para poder usar las señales y slots cada clase tiene que declararse en un fichero de cabecera y la implementación se sitúa en un archivo '.cpp' por separado. Este archivo de cabecera se pasa entonces a través del MOC. Este produce un '.cpp' el cual contiene el código que permite que funcionen las señales y slots.

En la Ilustración 42 se puede ver el flujo desde que se crea una clase derivada de QObject hasta que se compila:

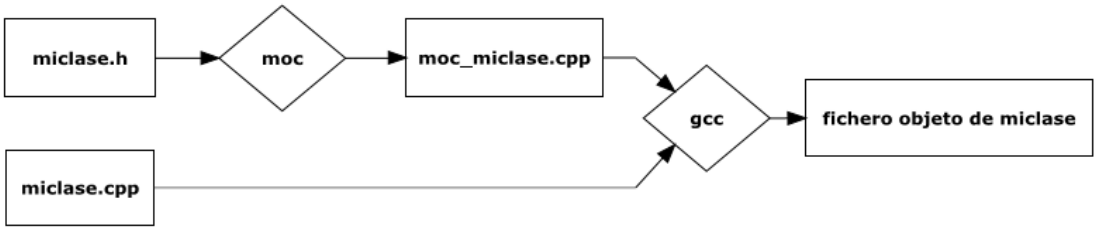


Ilustración 42: Flujo del MOC

Las palabras *signals* y *slots* son palabras reservadas o keywords del MOC, el cual convierte en código C++ real. No podremos definir estas palabras en librerías de terceros a no ser que se especifique y para lo cual deberán definirse macros de Qt que puedan ser interpretadas por el MOC como signals y slots.

UIC (User Interface Compiler): o compilador de interfaces de usuario traduce los ficheros '.ui' generados por Qt Designer a clases C++. Qmake configura esta herramienta de manera automática.

Debido a que las aplicaciones hechas con Qt necesitan de estas compilaciones previas y dependen de muchas librerías, la compilación se convierte en un proceso muy complejo incluso con la herramienta make. Por eso Qt dispone de la herramienta Qmake que simplifica el proceso. Por esta razón, no hemos podido enlazar BUSMASTER y Qt, porque a pesar de usar el mismo compilador cada entorno de trabajo utiliza variables y procesos ajenos a nosotros y no sabemos cómo tratan exactamente las compilaciones.

Finalmente nos decantamos por el uso de sockets dado que la abstracción de este método no debería imposibilitar su creación, aunque previamente hicimos varios intentos para realizar una compilación secuencial para integrar las interfaces generadas por Qt en la DLL generada por BUSMASTER.

2.5.1.1 – Compilación de la GUI a través del Makefile de BUSMASTER

Tanto Qt como BUSMASTER utilizan MinGW por lo tanto optamos como primera opción hacer una compilación secuencial del panel creado en Qt y el sistema simulado de BUSMASTER. Para ello usamos el Makefile de BUSMASTER llamado GCCDLLMakeTemplate_CAN.

Partiremos de un sistema simulado de BUSMASTER sencillo como el de la Ilustración 36.

Por otro lado, hemos creado una aplicación en Qt llamada 'panelPrueba'.

En Qt se pueden hacer dos tipos de depuraciones:

- Modo Debug: Es más adecuado cuando estamos en la fase de desarrollo del proyecto. Se genera toda la información de depuración.
- Modo Release: Se usa una vez acabado el proyecto, para entregar el programa al cliente. El código está más optimizado en tiempo y espacio.

Utilizaremos la depuración en modo Release[5]. Tras la compilación Qt habrá creado el ejecutable '.exe' junto con los archivos '.o' correspondientes, incluidos los del MOC y el UIC.

En la Ilustración 43 se muestra el procedimiento de compilación individual tanto de Qt como de BUSMASTER y el procedimiento de compilación secuencial que se desea conseguir.

Para su desarrollo hemos modificado el Makefile añadiendo las dependencias correspondientes a la aplicación desarrollada en Qt, con el objetivo de generar una única librería dinámica que contenga la '.dll' de BUSMASTER y el '.exe' correspondiente a Qt. El problema que hubo es que el compilador de BUSMASTER no entendía las cabeceras (ficheros '.h') relativas de Qt y dado que existen multitud de relaciones entre cabeceras es inviable modificar todas ellas a su ruta absoluta.

[5] Consultar Anexos para configurar una aplicación en modo Release

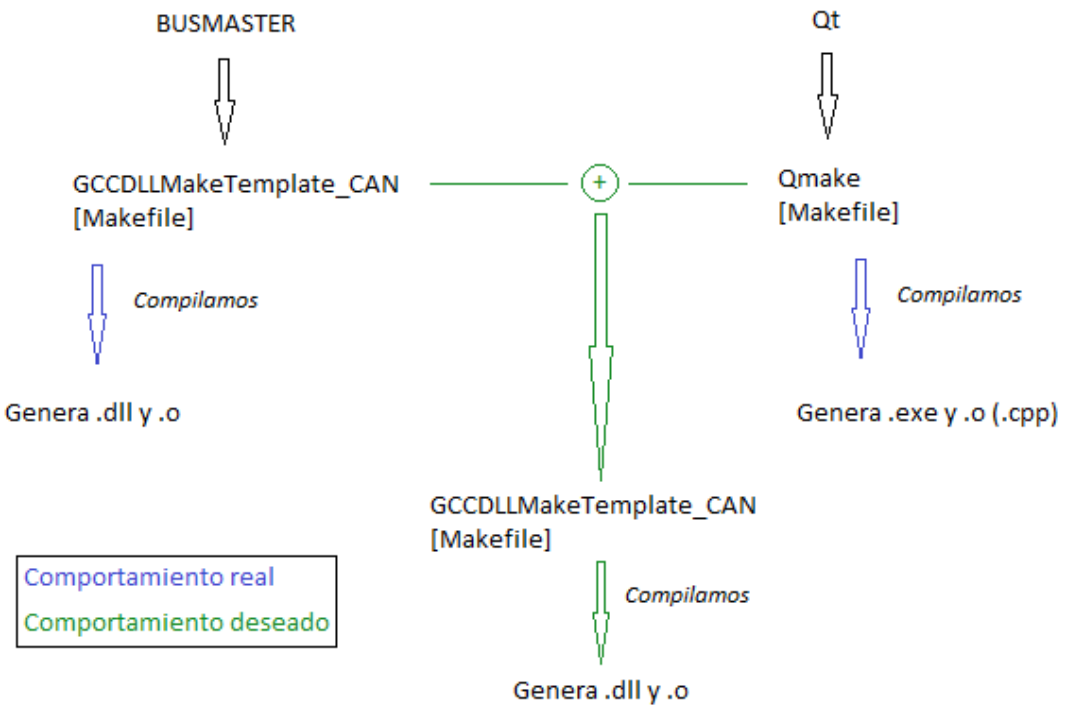


Ilustración 43: Flujo de la compilación real y deseada

2.5.1.2 – Precompilación de la GUI a través de Qmake

Debido a la dificultad encontrada con las cabeceras de Qt, decidimos realizar una precompilación de la aplicación en Qt mediante Qmake para posteriormente hacer la compilación secuencial desde el Makefile de BUSMASTER.

Tras generar una precompilación con Qmake y modificar el Makefile de BUSMASTER acorde a ello, no fuimos capaces de terminar la compilación secuencial. La dificultad fue debida a que la precompilación de Qmake genera un archivo del tipo 'PCH.h.gch' y, para que el compilador de GNU use este fichero, debe estar compilado exactamente igual que el '.cpp' que lo necesita. Eso incluye que tenga el mismo formato de compilación tanto Qmake como el Makefile de BUSMASTER y debido a la complejidad del Makefile de Qt se convierte en una tarea ardua.

2.5.1.3 – Compilación de la GUI a través de MSYS

Otra alternativa que tomamos para resolver la dificultad de las cabeceras de Qt fue instalar el entorno MSYS para hacer la compilación. En este entorno fuimos capaces de compilar la aplicación de Qt, pero al unir todas las dependencias en el Makefile de BUSMASTER no fuimos capaces de generar la DLL con ambos programas.

2.5.1.4 – Exportar la GUI en una librería dinámica

El siguiente camino que tomamos fue generar una aplicación en Qt con un ejecutable '.exe' que cargue sus librerías en una DLL. La mejor forma para hacerlo es a través de un plugin para ser añadido como una funcionalidad a BUSMASTER. El principal problema que encontramos es que no se puede exportar una GUI en Qt ya que los métodos de Qt Designer son declarados como privados y deben ser públicos. Una opción sería crear un nuevo espacio de nombres pero estaría fuera de la clase que se quiere exportar. Otra opción sería usar Qt/MFC Migration Framework, aunque no nos conviene dado que nos meteríamos en una relación a tres programas.

2.5.2 – GUI mediante sockets

Un socket es un mecanismo de comunicación entre procesos que permite que un proceso hable (emita o reciba información) con otro proceso incluso estando en distintas máquinas. Es una forma de conseguir que dos programas se transmitan datos. Los sockets se pueden ver como una interfaz con la capa de transporte.

La comunicación entre procesos a través de sockets se basa en la filosofía de Cliente-Servidor. En la Ilustración 44 mostramos un diagrama del funcionamiento básico.

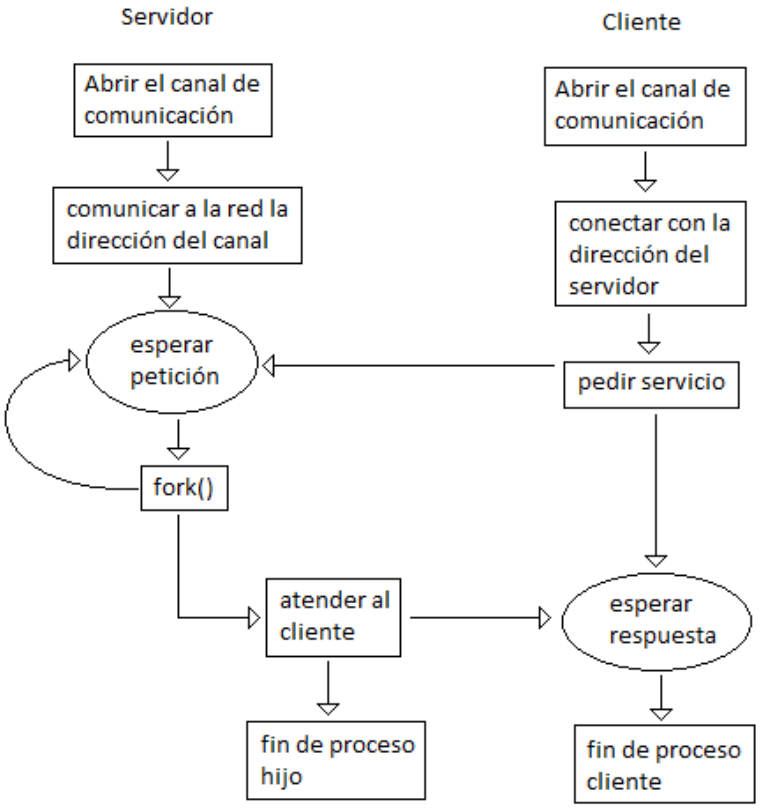


Ilustración 44: Filosofía Cliente-Servidor

Como se puede ver el cliente es el programa que solicita la conexión para pedir datos al servidor, mientras que el servidor permanece pasivo a la espera de que alguien solicite conexión con él.

Se pueden tener dos tipos de sockets:

- Socket de flujo: al conectar se realiza una búsqueda de un camino libre entre origen y destino. Se mantiene el camino en toda la conexión. Basado en TCP.
- Socket de datagrama: no se fija un camino. Cada paquete podrá ir por cualquier sitio. No se garantiza la recepción secuencial. Basado en UDP.

Dado que nuestro objetivo es crear un panel para BUSMASTER utilizaremos el socket de flujo basado en TCP. Nos proporciona una comunicación bidireccional fiable y los paquetes llegan sin errores y en secuencia.

Qt nos proporciona la clase QTcpSocket, que implementa el protocolo de transporte TCP, para crear aplicaciones cliente y servidor de red. Para los servidores también necesitaremos la clase QTcpServer para manejar las conexiones TCP entrantes. En la Ilustración 45 se puede ver el esquema que hemos seguido para crear una aplicación Cliente-Servidor íntegra en Qt, en la cual se realiza una conexión directa a una IP y puerto concretos.

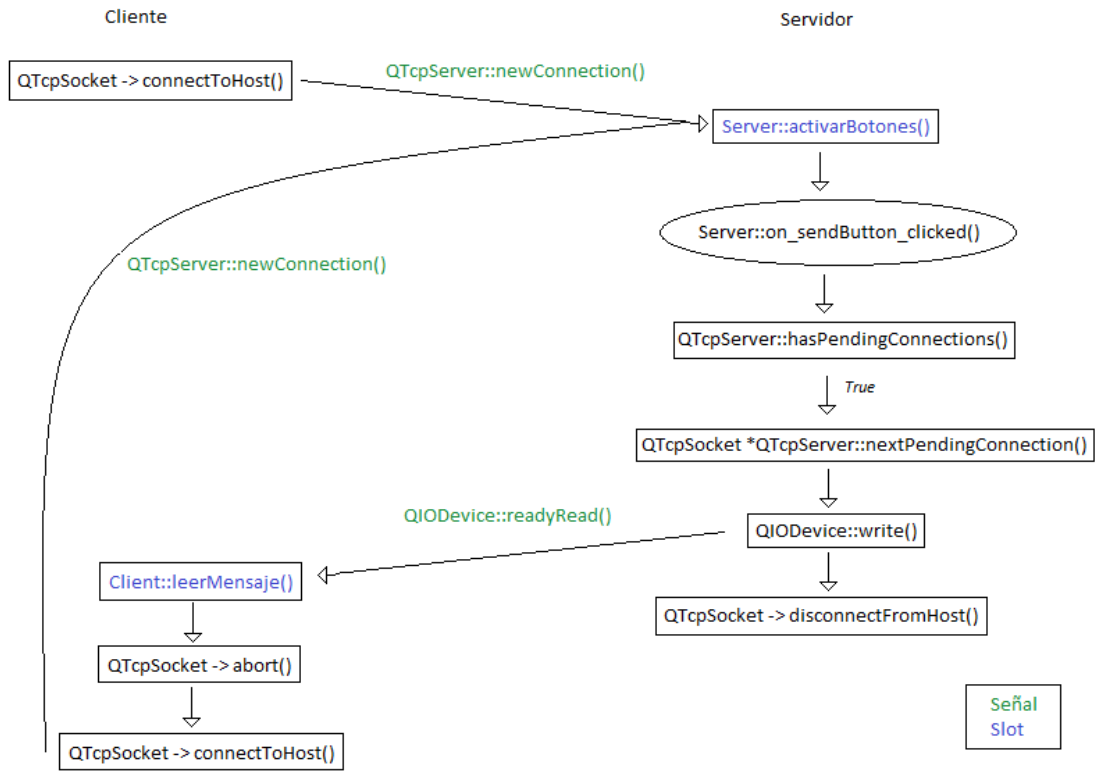


Ilustración 45: Flujo Cliente-Servidor en Qt

El cliente se conecta a una dirección IP y puerto específico y la señal 'newConnection' es detectada por el servidor, el cual ejecuta el slot 'activarBotones'. Este slot nos permite poder activar los botones de la GUI del lado del servidor. Cuando un botón sea pulsado el servidor buscará si tiene conexiones pendientes en cuyo caso realizará dicha conexión. El servidor

enviará los datos correspondientes, lo que activará la señal 'readyRead' en el lado del cliente haciendo que el slot 'leerMensaje' se ejecute. Tras esto el cliente aborta la comunicación activa y se conecta de nuevo a la dirección IP y puerto específico.

El problema que nos surge al crear un socket en Qt, dada su filosofía, es la comunicación bidireccional. El cliente debe mantenerse conectado al servidor a la espera de que éste vuelva a enviar nueva información pero sin la posibilidad de ser el cliente quien envíe datos al servidor.

2.5.2.1 – INTEGRACIÓN DEL CLIENTE EN BUSMASTER

Una vez creada una comunicación entre procesos en Qt debemos modificar el cliente para que pueda ser utilizado por BUSMASTER. Para ello hemos creado el cliente con un socket de Windows desde Visual Studio Community 2015[6].

Los pasos que hemos seguido para crear el cliente en Windows son:

1. Inicializar Winsock
2. Crear un socket
3. Conectarse al servidor
4. Recibir datos
5. Desconectarse

Primero de todo hemos **inicializado Winsock**. Todos los procesos que usan las funciones de Winsock deben inicializar el uso de las 'Windows Socket DLL' antes de hacer alguna llamada a sus funciones. Está inicialización también nos permite asegurarnos de que nuestro sistema soporta las librerías de Winsock. Para ello hemos creado un objeto wsaData de la clase WSADATA. Después, como se muestra en la Ilustración 46, llamamos a la función WSAStartup para iniciar el uso de WS2_32.dll.

```
iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != 0)
{
    printf("WSAStartup failed with error: %d\n", iResult);
    return 1;
}
```

Ilustración 46: Inicializar Winsock

El entero iResult nos permitirá identificar si ha habido algún error en la inicialización de la versión 2.2 de Winsock.

Una vez inicializado Winsock hemos **creado un socket**. Para ello hemos creado el objeto addrinfo que tiene la estructura de sockaddr e inicializamos sus valores para tener un socket de flujo basado en TCP y compatible con IP4 e IP6.

[6] Consultar Anexos para conocer las condiciones de instalación de Visual Studio Community 2015.

```
struct addrinfo *result = NULL,

    *ptr = NULL,
    hints;

ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
```

Ilustración 47: Definición de las características del socket

La función getaddrinfo nos permite conectarnos a la dirección IP y el puerto que ya habíamos definido en el servidor de Qt. Devuelve un valor entero con el que podemos comprobar errores de conexión.

```
iResult = getaddrinfo("localhost", DEFAULT_PORT, &hints, &result);
if (iResult != 0)
{
    printf("getaddrinfo failed with error: %d\n", iResult);
    WSACleanup();
    return 1;
}
```

Ilustración 48: Comprobar conexión con el servidor

La llamada a la función socket debe devolver su valor en el objeto ConnectSocket de la clase SOCKET para después comprobar que es un socket válido.

```
ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
if (ConnectSocket == INVALID_SOCKET) {
    printf("Socket failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
```

Ilustración 49: Crear socket de Windows

Para **conectarnos al servidor** de Qt hemos llamado a la función connect pasando el socket creado y la estructura sockaddr como parámetros.

```
iResult = connect(ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
if (iResult == SOCKET_ERROR)
{
    closesocket(ConnectSocket);
    ConnectSocket = INVALID_SOCKET;
    continue;
}
```

Ilustración 50: Conectarse al servidor

Tras limpiar el contenido de la memoria del buffer de recepción llamamos a la función recv para **recibir los datos** enviados por el servidor. La función devuelve el número de bytes recibidos y almacena en la variable recvbuf el contenido de los datos recibidos.

```
memset(&recvbuf[0], 0, DEFAULT_BUFLEN);

iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
```

Ilustración 51: Recibir datos del servidor

Para **desconectar** el socket llamamos a la función closesocket y luego limpiamos los recursos de 'Windows Sockets DLL' mediante la función WSACleanup.

```
closesocket(ConnectSocket);
WSACleanup();
```

Ilustración 52: Desconectar socket

En la Ilustración 53 e Ilustración 54 se muestra el resultado de la GUI creada para BUSMASTER en la cual seremos capaces de gestionar la posición de la llave y en consecuencia modificar las posiciones de encendido y el estado del vehículo.

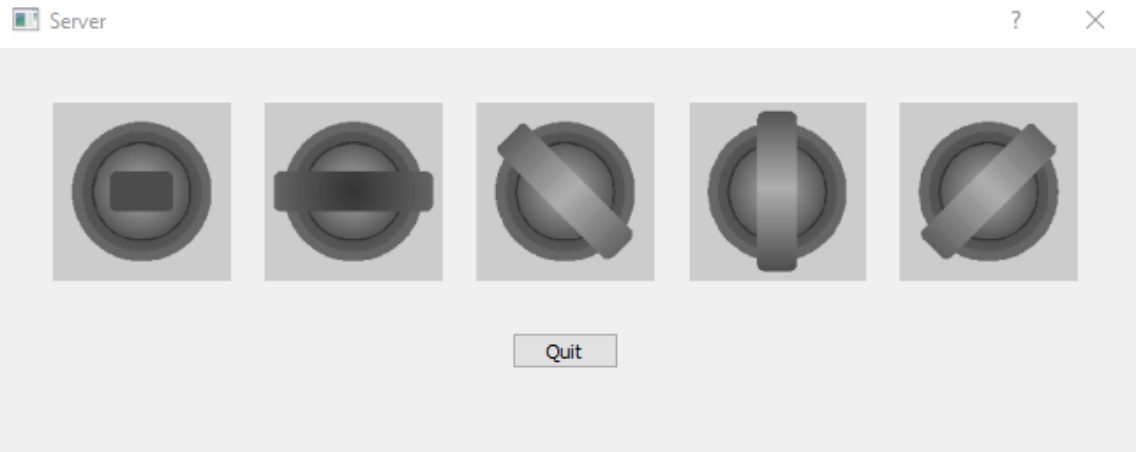


Ilustración 53: GUI del lado del Servidor

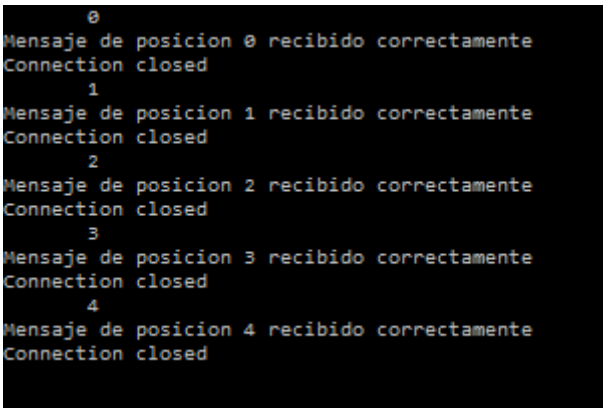


Ilustración 54: GUI del lado del Cliente

Cuando pulsamos en cualquiera de las imágenes del servidor, este enviará un mensaje al cliente con la posición asociada a dicha imagen. Del lado del cliente hemos creado una aplicación de consola en la cual podemos ver, para cada uno de los casos, el resultado de haber procesado y filtrado el mensaje recibido del servidor. Como hemos mencionado

anteriormente, tras cada envío la conexión entre cliente y servidor se cierra y vuelve a reiniciar.

Dado que el código generado en el cliente es ANSI C, este se puede utilizar en la programación de nodos de BUSMASTER. Para integrar el código en BUSMASTER hemos generado un controlador que sea ejecutado al cargarse las DLLs 'OnDLL_Load'. Dentro de este controlador añadimos el código generado en el socket cliente de Windows incluyendo las correspondientes cabeceras y librerías, así como modificar el Makefile de BUSMASTER para añadir las rutas necesarias.

El objetivo es que una vez recibido y procesado el mensaje enviado desde el servidor, al pinchar en la posición de la llave deseada, modifiquemos el mensaje correspondiente a la posición de la llave en el bus CAN al que estemos conectados, utilizando para ello un programa con estructura igual al de la Ilustración 36. Se puede hacer de manera sencilla añadiéndolo dentro del bucle if que procesa la posición de la llave recibida.

A la hora de compilar no hemos sido capaces de tener éxito. Obteníamos multitud de errores de compilación del tipo de la Ilustración 55.

```
C:/Program Files (x86)/Microsoft SDKs/Windows/v7.1A/Include/oaidl.h:445:17: note: in expansion of macro '_VARIANT_BOOL'
_VARIANT_BOOL bool;
^
C:/Program Files (x86)/Microsoft SDKs/Windows/v7.1A/Include/wtypes.h:1121:26: error: pasting "/" and "/" does not give a v
#define _VARIANT_BOOL    /##/
^
C:/Program Files (x86)/Microsoft SDKs/Windows/v7.1A/Include/oaidl.h:460:17: note: in expansion of macro '_VARIANT_BOOL'
_VARIANT_BOOL *pbool;
^
```

Ilustración 55: Error de compilación cliente en BUSMASTER

La conclusión obtenida en referencia a estos errores es que el socket de Windows está diseñado en ANSI C, sin embargo las librerías que utiliza el socket son C++ con lo que el compilador del paquete MinGW no es capaz de compilarlas.

CAPÍTULO 3: DISPOSITIVO DE ADQUISICIÓN DE DATOS PARA BUS CAN

El objetivo de este capítulo es explicar el desarrollo de la herramienta sustitutiva de la propietaria CANCaseXL, de Vector, basándonos en Arduino. En la Ilustración 56 se muestra la apariencia de una CANCaseXL. Se trata de un interfaz USB con dos controladores CAN el cual puede enviar y recibir mensajes CAN con identificadores de 11 y 29 bits así como tramas remotas sin restricciones. Además, es capaz de detectar y generar tramas de error en el bus.

Ilustración 56: Apariencia de la CANCaseXL

Este dispositivo necesita un conector DB9 hembra que vaya correctamente pineado a un OBD-II macho. De esta forma podremos conectar el dispositivo al OBD-II del vehículo. Por el otro extremo se conectará mediante USB al ordenador. La misma filosofía seguirá nuestro dispositivo de adquisición de datos para bus CAN.

3.1 – DESARROLLO HARDWARE

En la Ilustración 57 se puede ver el esquema de conexión de nuestro dispositivo de adquisición de datos desde el vehículo hasta el ordenador.

Ilustración 57: Esquema de las conexiones del dispositivo de adquisición de datos

El conector DB9 macho que sale de nuestro dispositivo está apropiadamente pineado para ser compatible con el DB9 hembra utilizado en la maqueta del Aula Mercedes de la Universidad de Valladolid de tal forma que lleguen las señales CAN-H, CAN-L, CAN_GND y CAN_V+ necesarias para hacer funcionar nuestro dispositivo.

3.1.1 – Elementos utilizados

La base para el dispositivo será un microcontrolador Arduino nano junto con un controlador y transceptor CAN necesarios para comunicarse a través del bus. Los siguientes elementos han sido necesarios para el diseño de nuestro dispositivo:

- Arduino nano [14]: es el microcontrolador y la unidad principal del módulo que nos permite manejar al resto de dispositivos. Se encarga de la inicialización y configuración del controlador CAN. Características principales:
    - Microcontrolador ATmega328 con cargador de inicio preprogramado.
    - Tensión de entrada (recomendada): +7 a + 12 V.
    - Tensión de entrada (límites): +6 a + 20 V.
    - 14 pines GPIO (de los que 6 ofrecen salida PWM).
    - 6 pines de entrada analógica.
    - SRAM de 2 KB.
    - EEPROM de 1 KB.
    - Admite comunicación serie IC.
    - Frecuencia de reloj: 16 MHZ.
- MCP 2515 [15]: es el controlador CAN que se comunica con el microcontrolador a través del protocolo SPI a una tasa de hasta 1 Mbps. Recibe del microprocesador los datos que han de ser transmitidos, los acondiciona y los pasa al transceptor CAN. Asimismo recibe los datos procedentes del transceptor CAN, los acondiciona y los pasa al microprocesador en la unidad de control. Es capaz de transmitir y recibir tramas estándar, extendidas y remotas.
- MCP 2551 [17]: es el transceptor CAN, es decir, hace de interfaz entre el controlador CAN y el bus físico, transformando los datos del controlador CAN en señales eléctricas a una tasa de 1 Mbps.
- RTC: Real Time Clock. Es el reloj de nuestro sistema. Se comunica con el microcontrolador mediante I2C.
- Resistencias: hemos utilizado resistencias con valores de 470 Ω, 1 KΩ y 10 KΩ con el fin de seleccionar un determinado modo de operación en el controlador CAN, para adaptar el valor de los voltajes y para el desarrollo del circuito conversor de voltaje.
- Condensadores: hemos utilizado condensadores con valores de 10 µF, 15 pf, 1 µF y 0,1 µF. Estos condensadores han sido necesarios para el correcto funcionamiento de los

distintos módulos, porque forman parte del circuito conversor de voltaje o bien para aislar el elemento del ruido que provengan de la línea de alimentación.

- LED: ha sido necesario uno para identificar cuando se están transmitiendo tramas y otro para identificar cuando se están recibiendo.
- Diodos 1N4001: han sido necesarios dos para el desarrollo del circuito conversor de voltaje.
- Oscilador de cristal: ha sido conectado en paralelo con dos condensadores de 15 pF para que el oscilador del controlador CAN funcione a 20 MHz.
- Conversor de voltaje LM317t [16]: nos permite obtener valores de voltaje a la salida entre 1,2 V y 37 V para valores a la entrada de hasta 40 V.
- Conector DB9 macho: necesario para conectarse al bus CAN a través del OBD-II del vehículo como se muestra en la Ilustración 57. El pineado utilizado se muestra en la Ilustración 58.

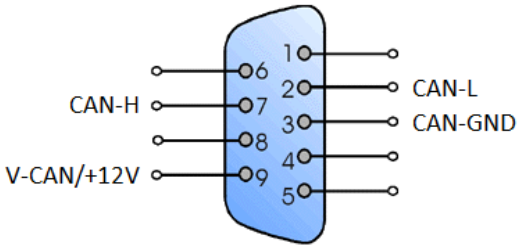


Ilustración 58: DB9 pinout

Con todos estos elementos y con la ayuda de las hojas de especificaciones[7] hemos sido capaces de desarrollar el diseño. Cabe mencionar que tanto el RTC como el circuito conversor de voltaje no son necesarios para el funcionamiento de nuestro dispositivo, sin embargo hemos decidido añadirlos para futuras aplicaciones. El RTC nos puede servir para tener una medida de tiempo absoluto que nos permita fechar las tramas recibidas y tener un control sobre ellas. Esto nos puede ser útil si queremos añadir la funcionalidad de datalogger a nuestro sistema, utilizando por ejemplo un módulo SD. Por su parte, el conversor de voltaje es innecesario si tenemos el Arduino nano alimentado mediante USB a través del ordenador. Eso sí, será necesario si queremos hacer las funciones de datalogger ya que tener el ordenador conectado restaría utilidad a este hipotético sistema. De cualquier modo es interesante utilizarlo dado que su función no es sólo proveernos los 7 V necesarios para alimentar el Arduino nano, sino que también nos protege de posibles picos de tensión.

3.1.2 – Protocolos de comunicación

Para el diseño de nuestro sistema y como se ha mencionado previamente, son necesarios los protocolos de comunicación SPI e I2C. En el caso de SPI es necesario para la comunicación con el controlador CAN MCP2515 e I2C es necesario para el RTC. A continuación se hace mención al funcionamiento básico de cada uno de estos protocolos.

[7] Consultar sección de Referencias

3.1.2.1 – SPI (Serial Peripheral Interface)

El protocolo SPI [20] fue desarrollado por Motorola y consiste en un bus de tres líneas, sobre el cual se transmiten paquetes de información de 8 bits de manera síncrona. Cada una de estas tres líneas porta la información entre los diferentes dispositivos conectados al bus. Cada dispositivo conectado al bus puede actuar como transmisor y receptor al mismo tiempo, por lo que este tipo de comunicación serie es full-duplex. Dos de estas líneas transfieren los datos (una en cada dirección) y la tercera línea es la del reloj.

Los dispositivos conectados al bus son definidos como maestros y esclavos. Un maestro es aquel que inicia la transferencia de información sobre el bus y genera las señales de reloj y control. Un esclavo es un dispositivo controlado por el maestro. Cada esclavo es controlado sobre el bus a través de una línea selectora llamada Chip Select o Select Slave, por lo tanto el esclavo es activado solo cuando esta línea es seleccionada. Generalmente existe una línea de selección dedicada para cada esclavo.

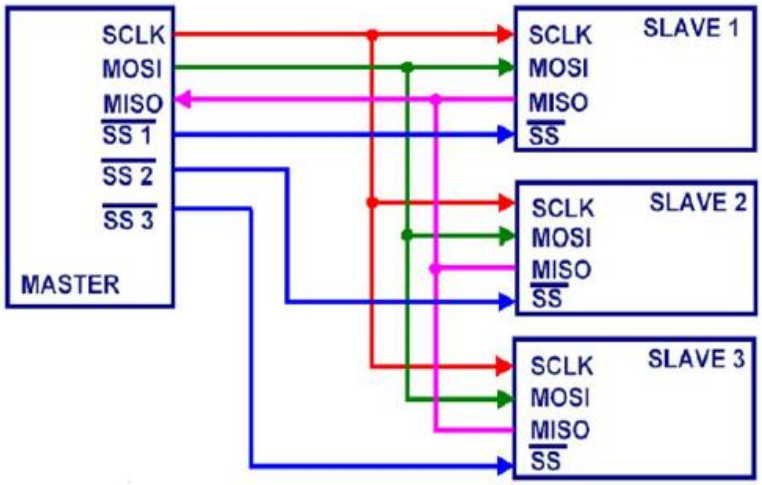


Ilustración 59: Disposición de nodos y conexionado en el protocolo SPI

El bus SPI emplea un registro de desplazamiento para transmitir la información. En la Ilustración 59 se puede ver la disposición de una comunicación SPI.

- La señal sobre la línea de reloj (SCLK) es generada por el maestro y sincroniza la transferencia de datos.
- La línea MOSI (Master Out Slave In) transporta los datos del maestro hacia el esclavo.
- La línea MISO (Master In Slave Out) transporta los datos del esclavo hacia el maestro.
- Cada esclavo es seleccionado por un nivel lógico bajo (‘0’) a través de la línea (CS = Chip Select o SS Slave Select).

Los datos sobre este bus pueden ser transmitidos a una tasa de hasta 1 Mbps en bloques de 8 bits, en donde el bit más significativo (MSB) se transmite primero.

3.1.2.2 – I2C (Inter-Integrated Circuit)

El protocolo I2C [21] es un tipo de bus diseñado por Philips Semiconductors a principios de los 80s que se utiliza para conectar circuitos integrados (ICs). El I2C es un bus con múltiples maestros, lo que significa que se pueden conectar varios chips al mismo bus y que todos ellos pueden actuar como maestro sólo con iniciar la transferencia de datos. Este estándar facilita la comunicación entre microcontroladores, memorias y otros dispositivos, utilizando para ello dos líneas de señal y una masa. Permite el intercambio de información entre muchos dispositivos a una velocidad de 100 Kbps.

La metodología de comunicación de datos del bus I2C es en serie y sincrónica. Las señales transmitidas son:

- SCL (System Clock) es la línea de los pulsos de reloj que sincronizan el sistema.
- SDA (System Data) es la línea por la que se mueven los datos entre los dispositivos.

La disposición de un bus I2C se muestra en la Ilustración 60 que tendría además la GND común de todos los dispositivos conectados al bus.

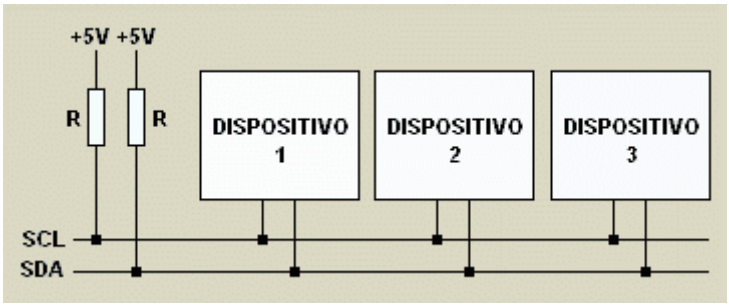


Ilustración 60: Disposición de nodos y conexionado en el protocolo I2C

Las resistencias de polarización son opcionales y tienen la misión de que las entradas lógicas del sistema se mantengan en los niveles correctos en caso de que otros dispositivos o circuitos se conecten o desconecten del sistema.

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, esta característica se la pueden ir pasando los dispositivos que tengan esa capacidad.

3.1.3 – Diseño del módulo

Después de saber los elementos que vamos a utilizar y cómo funcionan los protocolos de comunicación necesarios, vamos a diseñar la placa PCB para nuestro dispositivo. Para ello nos hemos ayudado del programa Proteus. Este software consta de dos programas principales: Ares e Isis.

3.1.3.1 – Diseño en Isis

Lo primero de todo realizamos el diseño del plano eléctrico del circuito mediante Isis. En la Ilustración 61 se pueden ver las entradas y salidas del Arduino nano. Se pueden identificar las

señales necesarias para la comunicación de los protocolos SPI e I2C y el voltaje de alimentación del Arduino Vin, que debe tener un valor de 7 V.

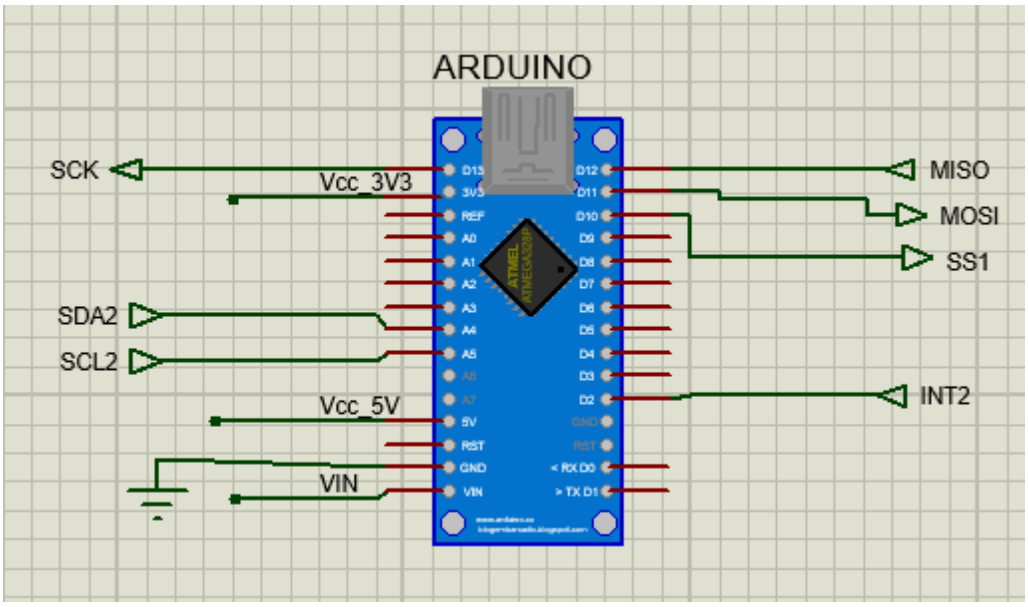


Ilustración 61: Conexionado del Arduino en Isis

En la Ilustración 62 encontramos las señales de entrada y salida que llegan a través del conector DB9 desde el bus CAN.

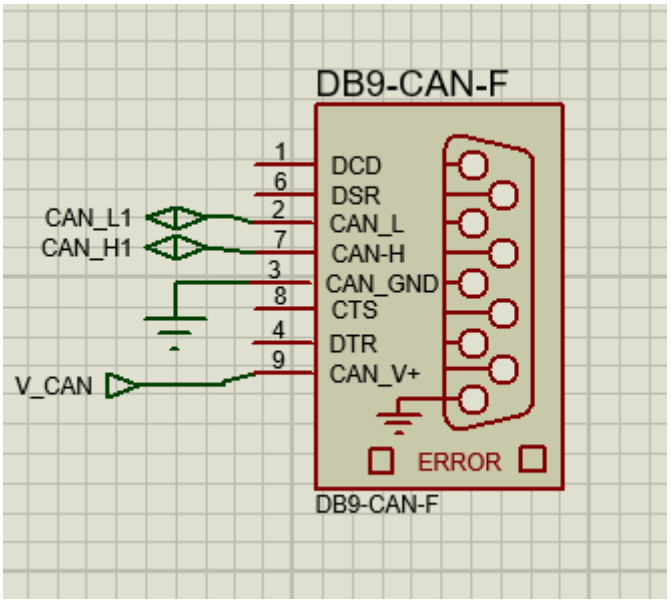


Ilustración 62: Conexionado del DB9 en Isis

En la Ilustración 63 vemos la disposición del conversor de voltaje, para el cual nos hemos basado en la hoja de especificaciones del conversor de voltaje LM317t [16] y hemos tomado los valores de los elementos de tal forma que a la salida tengamos los 7 V necesarios para alimentar el Arduino.

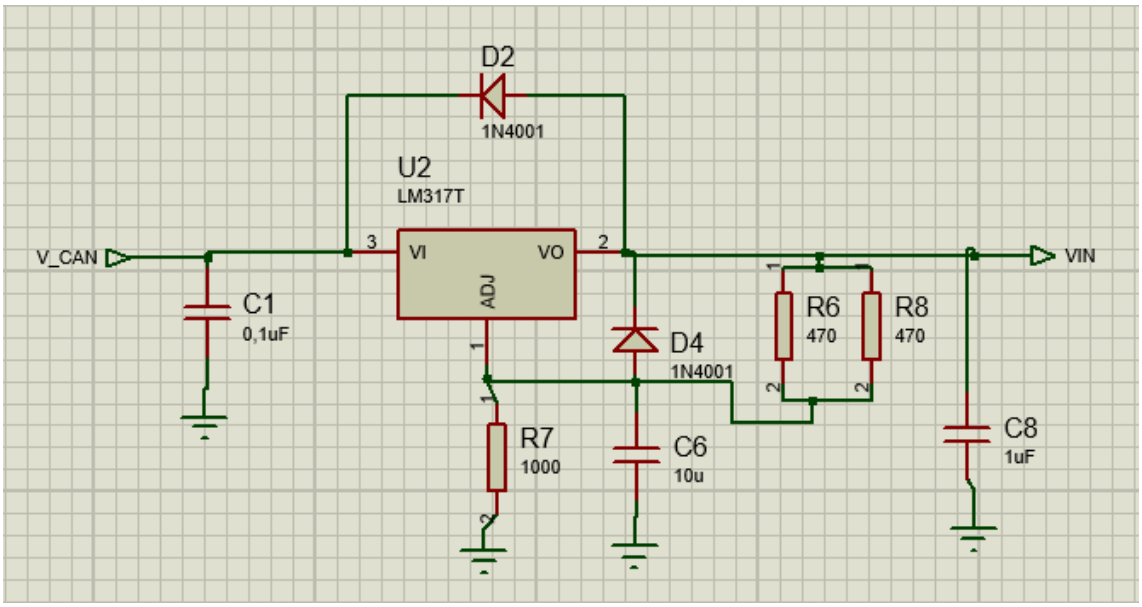


Ilustración 63: Conexionado del conversor de voltaje en Isis

En la Ilustración 64 tenemos al controlador CAN. Se pueden ver las señales necesarias para el protocolo SPI, los LED de transmisión y recepción y el oscilador de cristal.

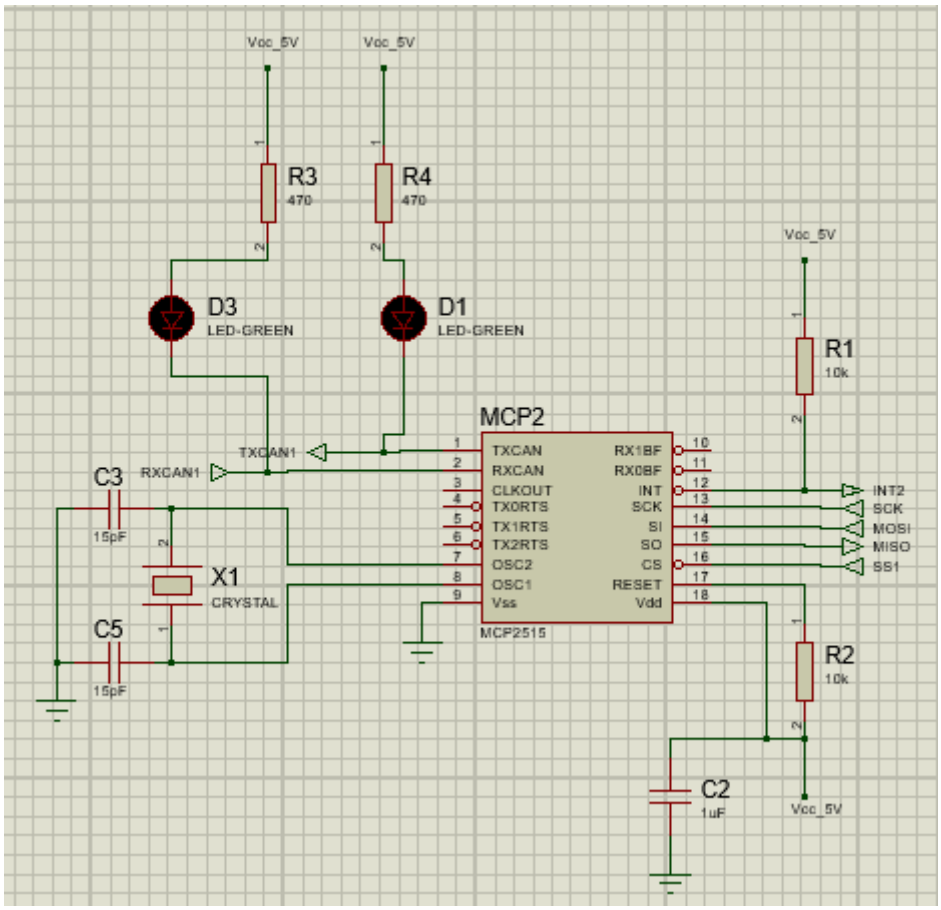


Ilustración 64: Conexionado del MCP2515 en Isis

En la Ilustración 65 vemos el transceptor CAN que por un lado tiene comunicación directa con las señales del bus CAN (CAN High y CAN Low) y por otro lado envía y recibe datos directamente con el controlador CAN.

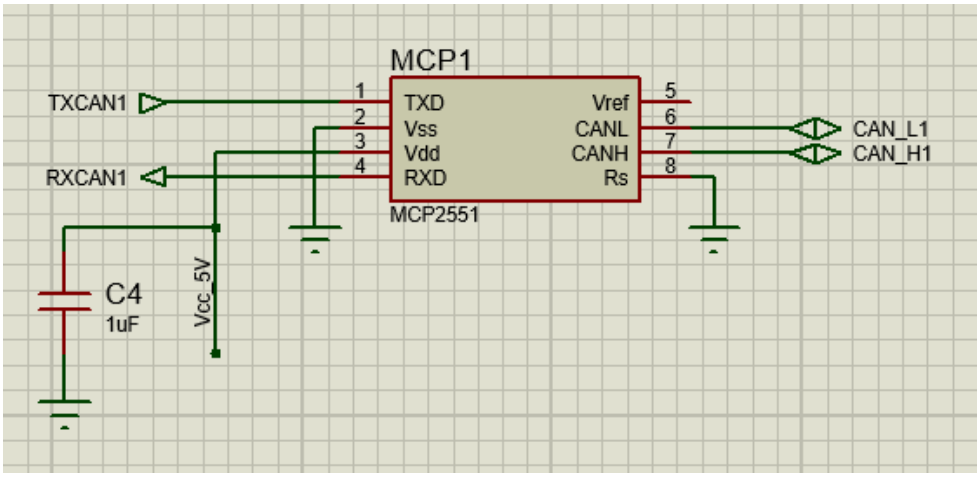


Ilustración 65: Conexionado del MCP2551 en Isis

En la Ilustración 66 tenemos las conexiones del RTC las cuales corresponden al protocolo I2C y la alimentación de 3,3 V suficiente para este módulo, que se puede extraer de una de las salidas del Arduino.

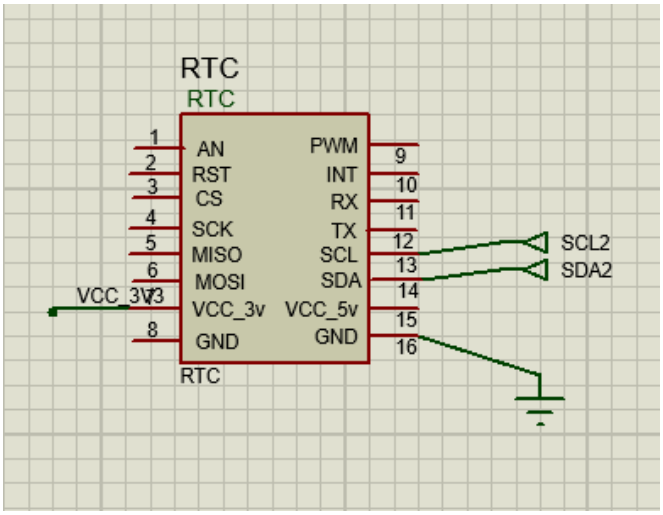


Ilustración 66: Conexionado del RTC en Isis

3.1.3.2 – Diseño en Ares

Una vez tenemos el plano eléctrico creado en Isis, podemos hacer uso de la herramienta Ares. Esta herramienta nos permite hacer el enrutado, la ubicación y la edición de los componentes. En la Ilustración 67 vemos el resultado de cómo quedaría nuestra PCB usando pistas de un milímetro de grosor.

Hemos tratado de hacer el diseño lo más reducido posible, obteniendo unas medidas de 6,5 cm de alto y 8,5 cm de ancho. Como podemos ver más claramente en el resultado 3D de la Ilustración 70, nuestro diseño cuenta con dos capas.

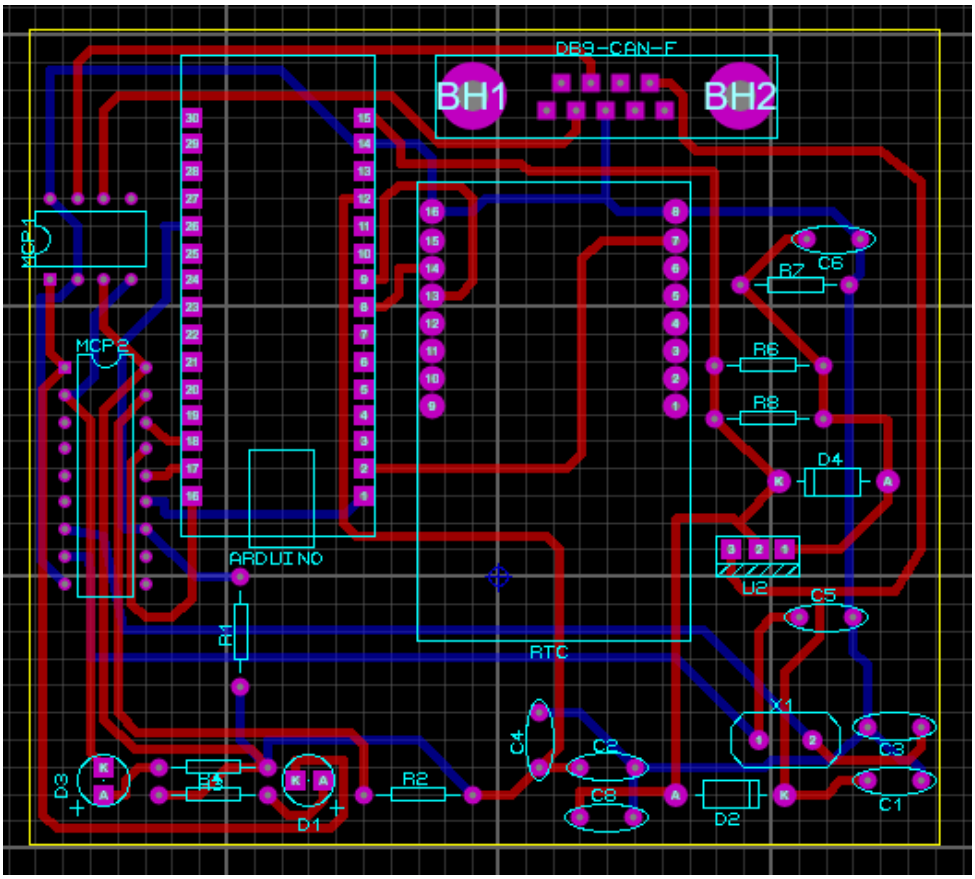


Ilustración 67: Diseño de la PCB en Ares

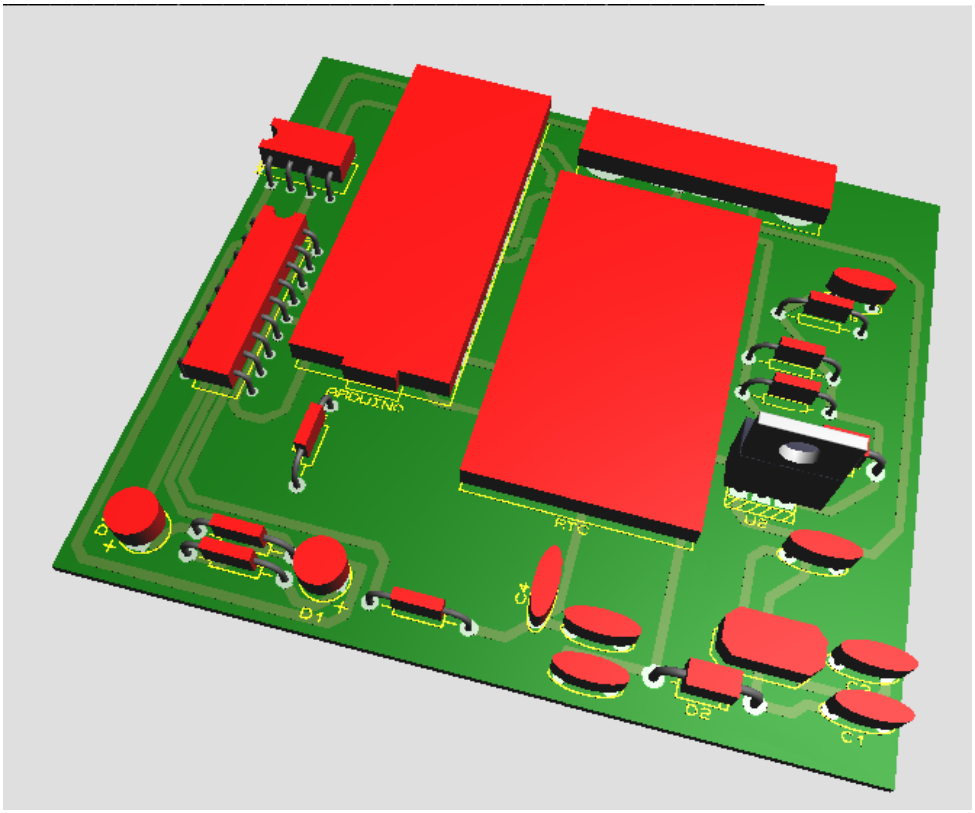


Ilustración 68: Diseño de la PCB en 3D vista por el frente con componentes

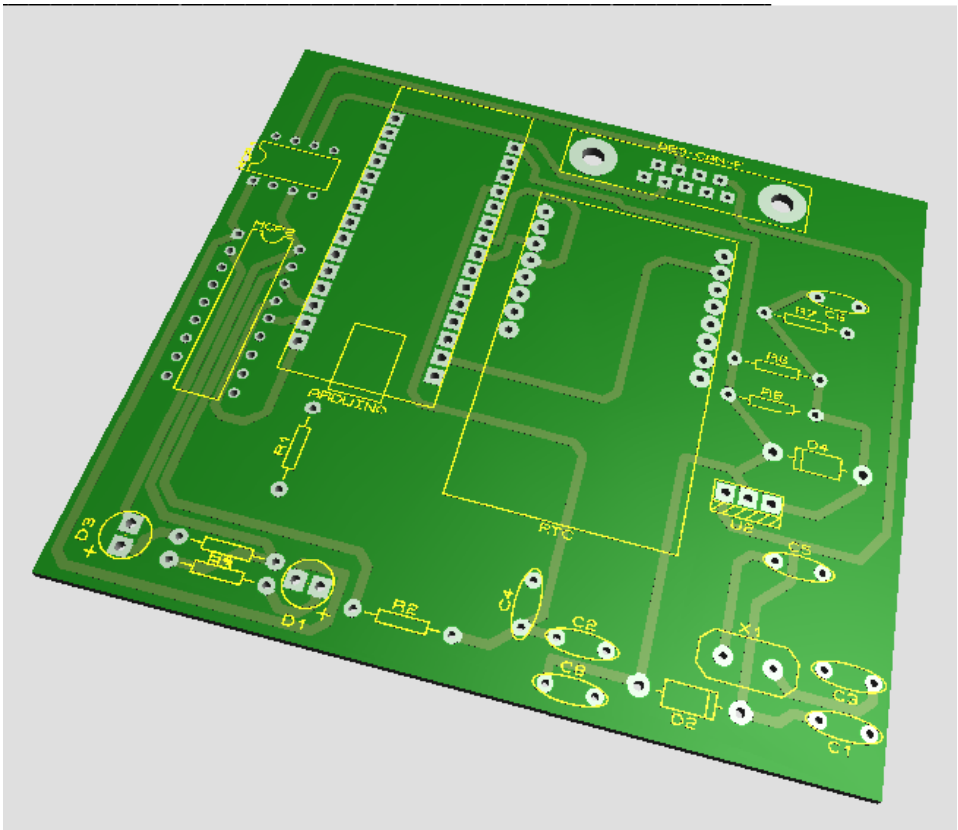


Ilustración 69: Diseño de la PCB en 3D vista por el frente

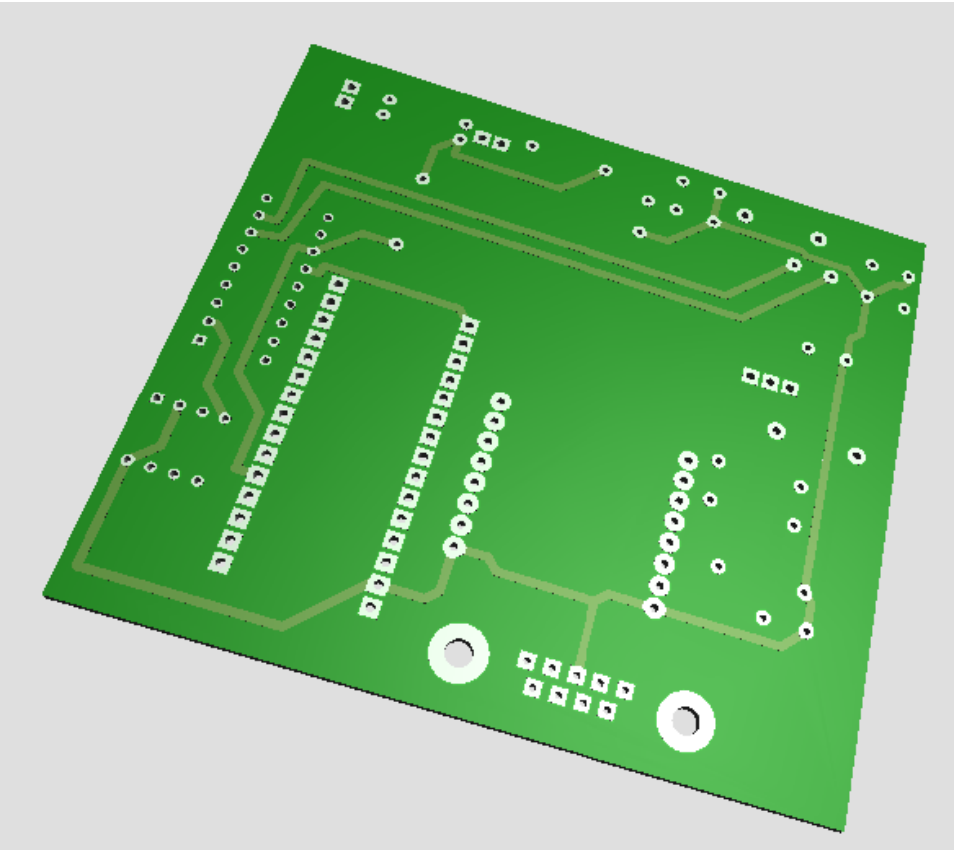


Ilustración 70: Diseño de la PCB en 3D vista por atrás

En la Ilustración 71 se muestra el montaje del módulo en una protoboard usado en un estudio previo que hicimos para poder dar fiabilidad a nuestro diseño antes de fabricar la placa.

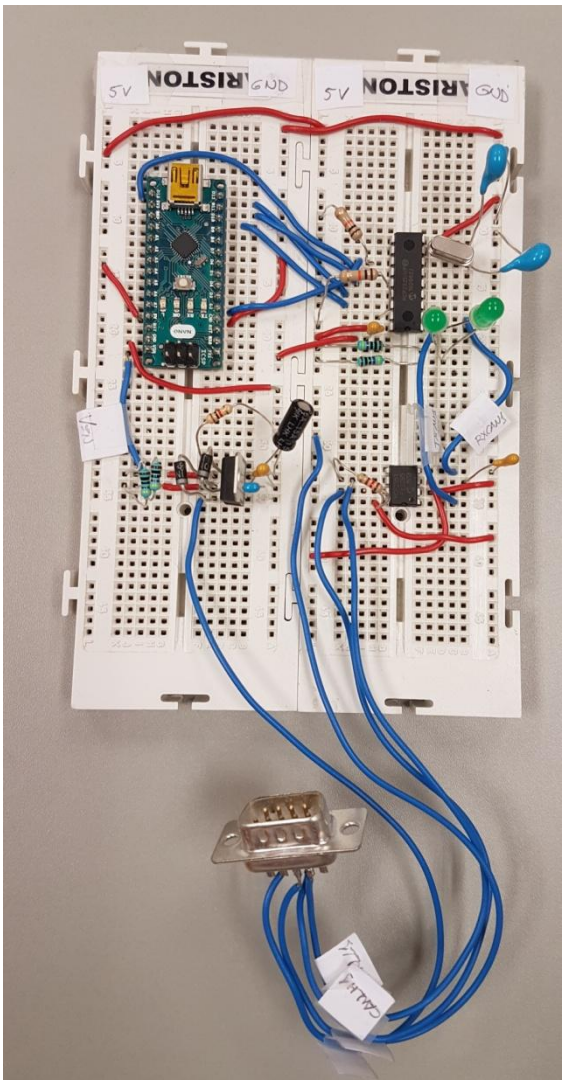


Ilustración 71: Montaje del módulo en una protoboard

3.1.3.3 – Montaje de la PCB

Una vez testeado el funcionamiento de nuestro sistema, hemos desarrollado el diseño en una placa de tiras para poder encapsularlo y facilitar así su protección y utilización. Durante todo el proceso hemos respetado la máxima reducción del tamaño de nuestro sistema, razón por la cual hemos decidido prescindir del módulo RTC.

El primer paso ha sido realizar el nuevo diseño de las conexiones para ser utilizado en una placa de tiras longitudinales y ajustándonos al tamaño de la caja de encapsulación. En la Ilustración 72 se muestra el resultado.

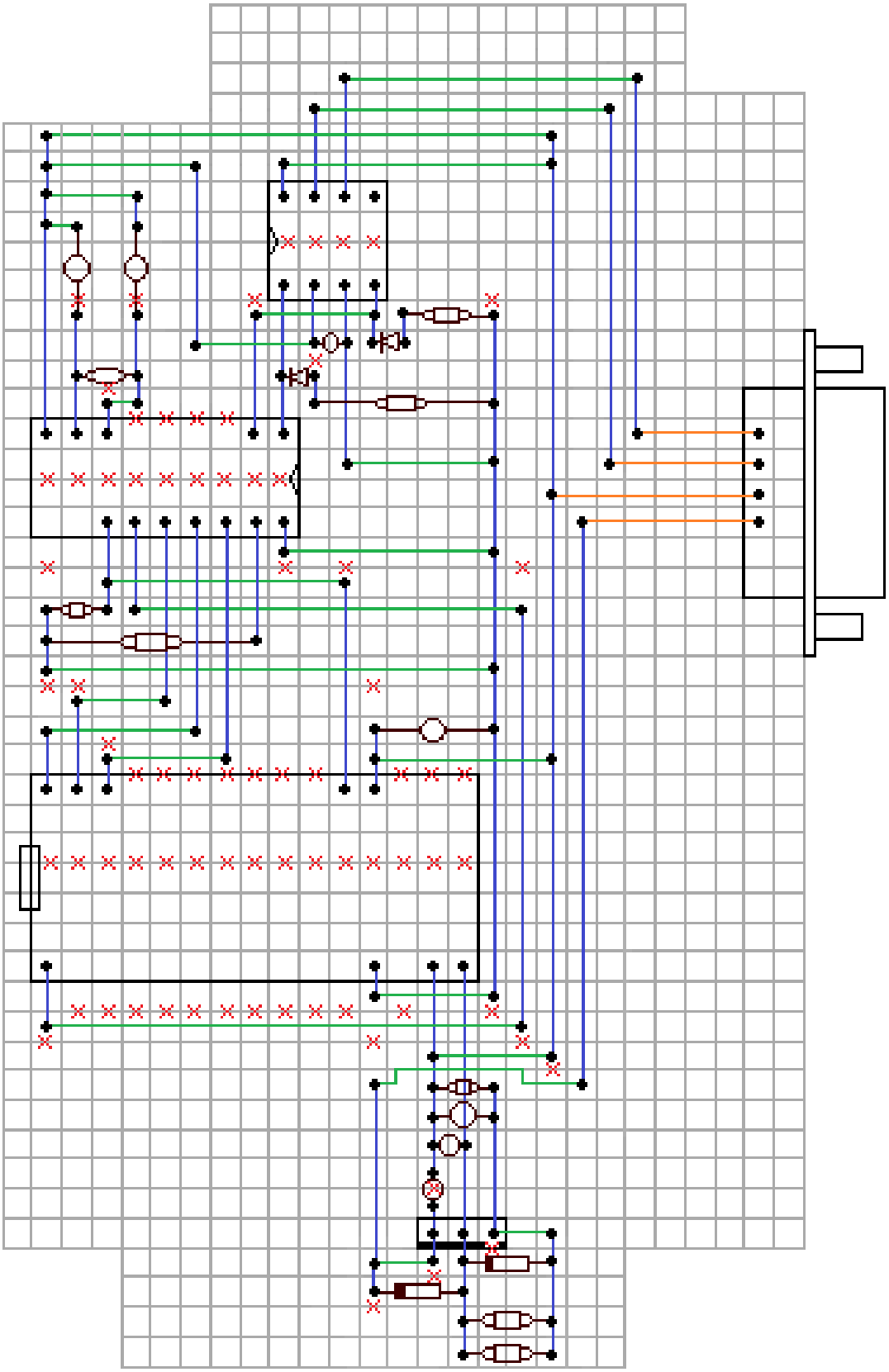


Ilustración 72: Diseño del sistema para la placa de tiras

Una vez hechas todas las soldaduras y los cortes necesarios obtuvimos un resultado como el de la Ilustración 73.

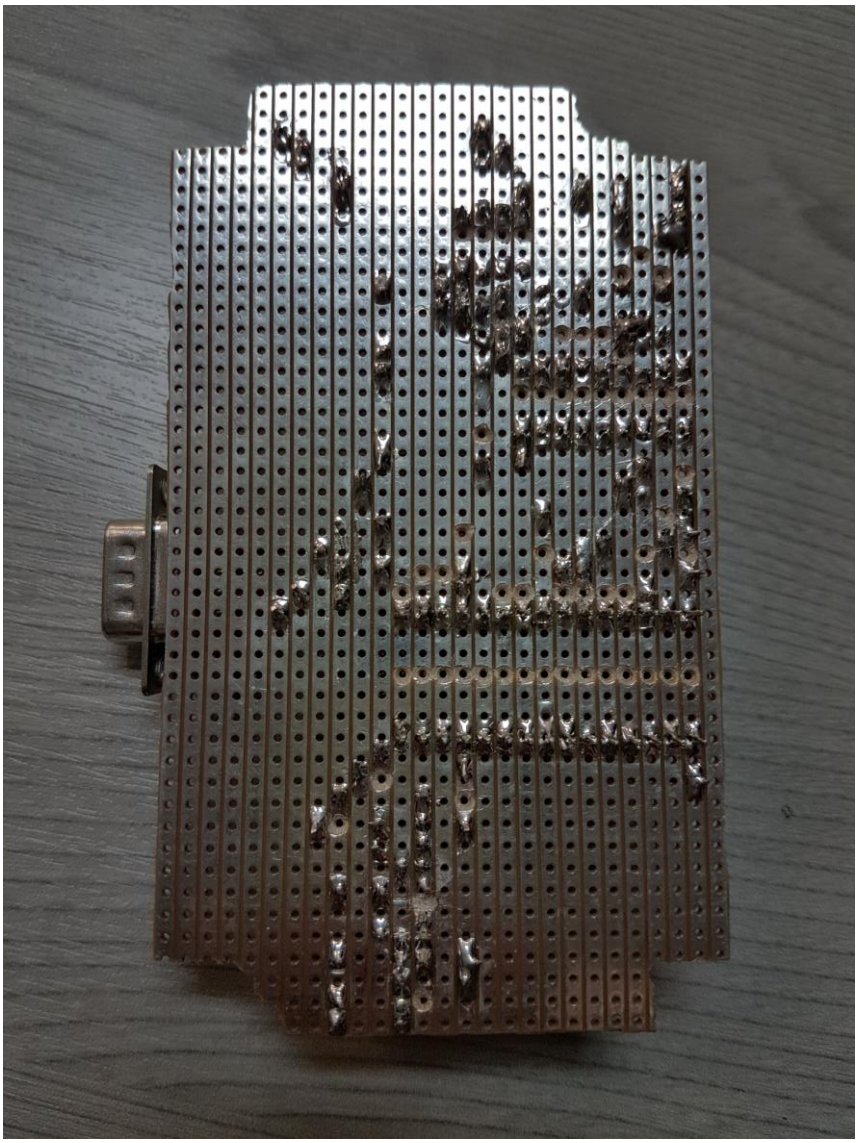


Ilustración 73: Soldaduras y cortes de la placa de tiras

En la Ilustración 74 se puede ver el frontal de la placa después de haber adaptado la caja en la que irá insertada.

Finalmente se puede ver en la Ilustración 75 la apariencia superficial de nuestro módulo de adquisición de datos cuyas dimensiones son de 12 cm x 7,2 cm x 5 cm.

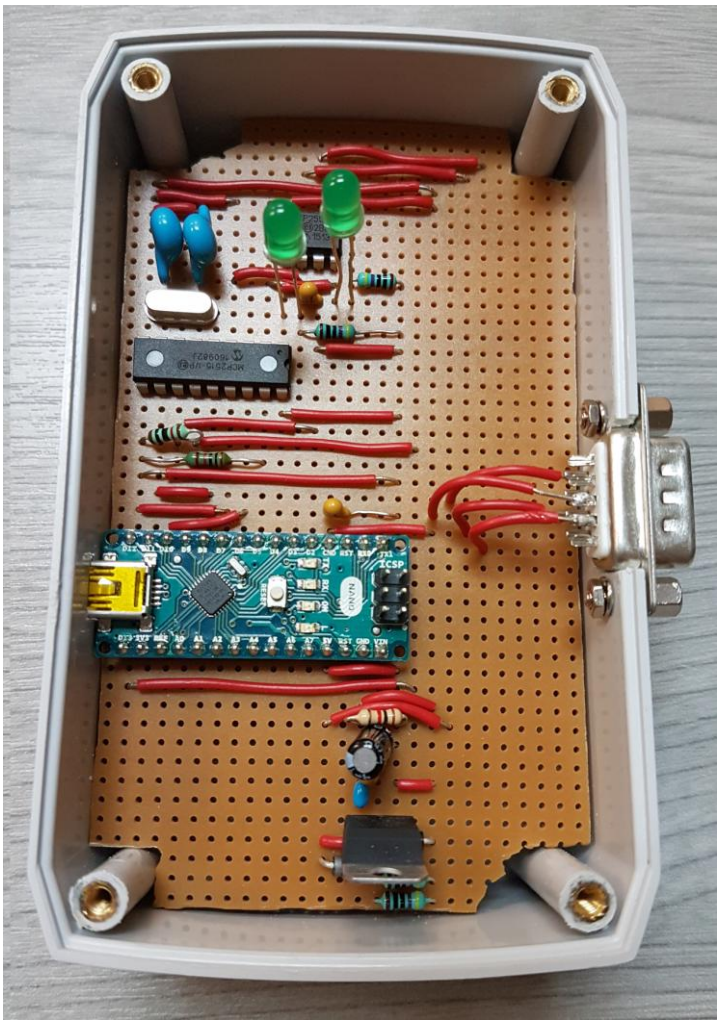


Ilustración 74: Frontal de la placa de tiras

Ilustración 75: Vista superficial del módulo de adquisición de datos

Para desarrollar el software de nuestro módulo, y dado que estamos utilizando un Arduino nano como microcontrolador, tenemos que utilizar el IDE de Arduino. El lenguaje de programación de Arduino está basado en C++ y tiene sus propias estructuras, variables y funciones[8] aunque también es posible usar comandos estándar de C++ en la programación de Arduino. Para nuestro diseño será necesario el uso de la librería SPI de Arduino, la cual nos permitirá comunicarnos con el controlador CAN mediante este protocolo.

3.2.1 – FUNCIONAMIENTO DEL CONTROLADOR CAN

Es importante entender las bases del funcionamiento del controlador CAN MCP2515. Para ello hacemos uso de su hoja de especificaciones.

3.2.1.1 – ESTRUCTURA DEL TIEMPO DE BIT

El tiempo de bit CAN se divide en cuatro segmentos, como se muestra en la Ilustración 77; el segmento de sincronización (Sync), el segmento de tiempo de propagación (PropSeg), el segmento de buffer de fase 1 (PS1) y el segmento de buffer de fase 2 (PS2). Cada segmento cuenta con un número específico de TQ (Time Quanta) que son programables mediante el BRP (Baud Rate Prescaler) como se muestra en la Ilustración 76.

$$TQ = 2 \cdot BRP \cdot T_{OSC} = \frac{2 \cdot BRP}{F_{OSC}}$$

Ilustración 76: Relación BPR y TQ

La longitud de TQ, que es la unidad de tiempo básica del tiempo de bit, depende del periodo de oscilación del reloj del controlador CAN.

El segmento de sincronización es la parte del tiempo de bit en la que se espera que se produzcan los bordes o picos de nivel del bus CAN. La distancia entre un borde fuera o dentro de Sync se denomina error de fase. El segmento PropSeg está destinado a compensar los tiempos de retardo físicos dentro de la red CAN, que pueden ser producidos por retardos en el tiempo de propagación de la señal en el bus o por retardos internos en los nodos CAN. Los segmentos de fase PS1 y PS2 son los apropiados para buscar un punto de muestreo.

El SJW (Synchronization Jump Width) define hasta que punto una resincronización puede mover el punto de muestreo dentro de los límites definidos por PS1 y PS2 para compensar los errores de fase. Un valor largo de SJW es sólo necesario cuando la señal de reloj es inestable, como cuando se usan osciladores cerámicos. En nuestro caso, al usar un oscilador de cristal, es suficiente con un valor de 1 TQ.

[8] Consultar la sección Referencias para acceder a la dirección web de Arduino

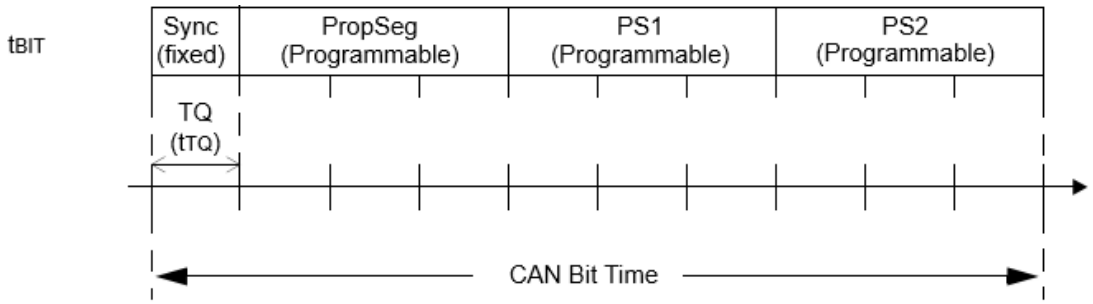


Ilustración 77: Tiempo de bit CAN y TQ

3.2.1.2 – Velocidad del bus CAN

Para configurar la velocidad del bus CAN en nuestro controlador debemos ajustar los valores de los registros CNF1, CNF2 y CNF3. Para hacer los cálculos vamos a establecer 10 TQ por cada tiempo de bit, siempre y cuando se cumpla que BRP nos quede un número entero. Por otro lado tenemos una frecuencia de oscilación fijada en 20 MHz.

Hay tres igualdades que se deben cumplir:

1) PropSeg + PS1 >= PS2
2) PropSeg + PS1 >= TDelay
3) PS2 > SJW
4) PS2 >= 2 TQ

Típicamente el valor del tiempo de retardo o TDelay es 1-2 TQ.

Para los casos que nos interesan vamos a configurar las velocidades High Speed CAN a 500 Kbps y Medium Speed CAN a 125 Kbps.

Primero vamos a calcular los valores para **MS CAN** aplicando las ecuaciones:

Ecuación 1: TQ = (1 / Baudios bus CAN) x (1 / TQ_{time})

Ecuación 2: TQ_{time} = (2 x (BRP + 1)) / Fosc

Siendo TQ = 10, Baudios del bus CAN = 125 Kbps, Fosc = 20 MHz. De la Ecuación 1 obtenemos que TQ_{time} = 0,8 µs. Despejando en la Ecuación 2 obtenemos un valor de BRP = 7 el cual, al ser un número entero, es válido.

Fijaremos el valor suficiente de SJW = 1 TQ.

El valor del registro CNF1 será '0x07' cómo se muestra en la Ilustración 78.

Estableceremos un valor de 4 TQ para PS1 y 1 TQ para PropSeg con lo que el valor del registro CNF2 será '0x98' cómo se muestra en la Ilustración 79.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
bit 7							bit 0
0	0	0	0	0	1	1	1

bit 7-6 **SJW**: Synchronization Jump Width Length bits <1:0>
11 = Length = 4 x TQ
10 = Length = 3 x TQ
01 = Length = 2 x TQ
00 = Length = 1 x TQ

bit 5-0 **BRP**: Baud Rate Prescaler bits <5:0>
TQ = 2 x (BRP + 1)/FOSC

Ilustración 78: CNF1 para 125 Kbps

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BTLMODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0
bit 7							bit 0
1	0	0	1	1	0	0	0

bit 7 **BTLMODE**: PS2 Bit Time Length bit
1 = Length of PS2 determined by PHSEG22:PHSEG20 bits of CNF3
0 = Length of PS2 is the greater of PS1 and IPT (2 TQ)

bit 6 **SAM**: Sample Point Configuration bit
1 = Bus line is sampled three times at the sample point
0 = Bus line is sampled once at the sample point

bit 5-3 **PHSEG1**: PS1 Length bits<2:0>
(PHSEG1 + 1) x TQ

bit 2-0 **PRSEG**: Propagation Segment Length bits <2:0>
(PRSEG + 1) x TQ

Ilustración 79: CNF2 para 125 Kbps

Para el valor de PS2 nos restan 4 TQ para completar los 10 que contiene cada tiempo de bit, con lo que el valor del registro CNF3 será '0x03' como se muestra en la Ilustración 80.

R/W-0	R/W-0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0
SOF	WAKFIL	—	—	—	PHSEG22	PHSEG21	PHSEG20
bit 7							bit 0
0	0	-	-	-	0	1	1

bit 7 **SOF:** Start-of-Frame signal bit
If CANCTRL.CLKEN = 1:
1 = CLKOUT pin enabled for SOF signal
0 = CLKOUT pin enabled for clockout function
If CANCTRL.CLKEN = 0, Bit is don't care.

bit 6 **WAKFIL**: Wake-up Filter bit
1 = Wake-up filter enabled
0 = Wake-up filter disabled

bit 5-3 **Unimplemented**: Reads as '0'

bit 2-0 **PHSEG2**: PS2 Length bits<2:0>
(PHSEG2 + 1) x TQ
Minimum valid setting for PS2 is 2 TQ

Ilustración 80: CNF3 para 125 Kbps

Comprobamos que estos valores efectivamente cumplen las cuatro condiciones impuestas:

Condición	Ecuación	Comprobación
1	PropSeg + PS1 >= PS2	TQ + 4 TQ >= 4 TQ
2	PropSeg + PS1 >= TDelay	TQ + 4 TQ >= 4 TQ
3	PS2 > SJW	4 TQ > TQ
4	PS2 >= 2 TQ	4 TQ >= 2 TQ

Tabla 7: Condiciones necesarias para establecer la velocidad del bus MS CAN

Para **HS CAN** mantendremos los valores de TQ = 10 y Fosc = 20 MHz, siendo ahora los baudios del bus CAN = 500 Kbps. En este caso de la Ecuación 1 obtenemos que TQ_{time} = 0,2 µs. Despejando en la Ecuación 2 obtenemos un valor de BRP = 1 el cual, al ser un número entero, es válido. Al igual que en MS CAN mantendremos el valor de SJW = 1 TQ, PS1 = 4 TQ, PropSeg = 1 TQ y PS2 = 4 TQ. De esta forma se siguen cumpliendo las cuatro condiciones impuestas. Entonces, los valores de los registros para HS CAN serán: CNF1 = 0x01, CNF2 = 0x98, CNF3 = 0x03.

3.2.1.3 – Modos de operación

El controlador CAN tiene cinco modos de operación diferentes que se eligen mediante los bits CANCRTL.REQOP:

- Modo de configuración: se inicia al encenderse el MCP, con un reset o con CANCRTL.REQOP = '100'. Es el único modo que nos permite modificar los registros CNF1, CNF2, CNF3, TXRTSCTRL, registros de filtrado y registros de máscara.
- Modo dormido: el interfaz SPI permanece activo para lectura. Se inicia con CANCRTL.REQOP = '111'.
- Modo solo-escucha: el MCP sólo permite recibir mensajes. Desactiva y resetea contadores de error.
- Modo bucle: permite transmisión interna de mensajes desde los buffer de transmisión a los buffer de recepción sin actualizar los mensajes recibidos en el bus CAN. Indicado para labores de testeo.
- Modo normal: modo estándar en el que se envían acknowledgements. Único modo del MCP en el cual se transmiten mensajes sobre el bus CAN.

3.2.1.4 – Transmisión y recepción de mensajes

Dado que tenemos que programar nuestro microcontrolador tanto para recibir como para transmitir mensajes al bus CAN, es importante saber cómo el controlador CAN procede y cuál es nuestra función como programadores.

3.2.1.4.1 – Transmisión de mensajes CAN

En la Ilustración 81 se muestra el esquema de flujo en la transmisión de mensajes usando el controlador CAN MCP2515.

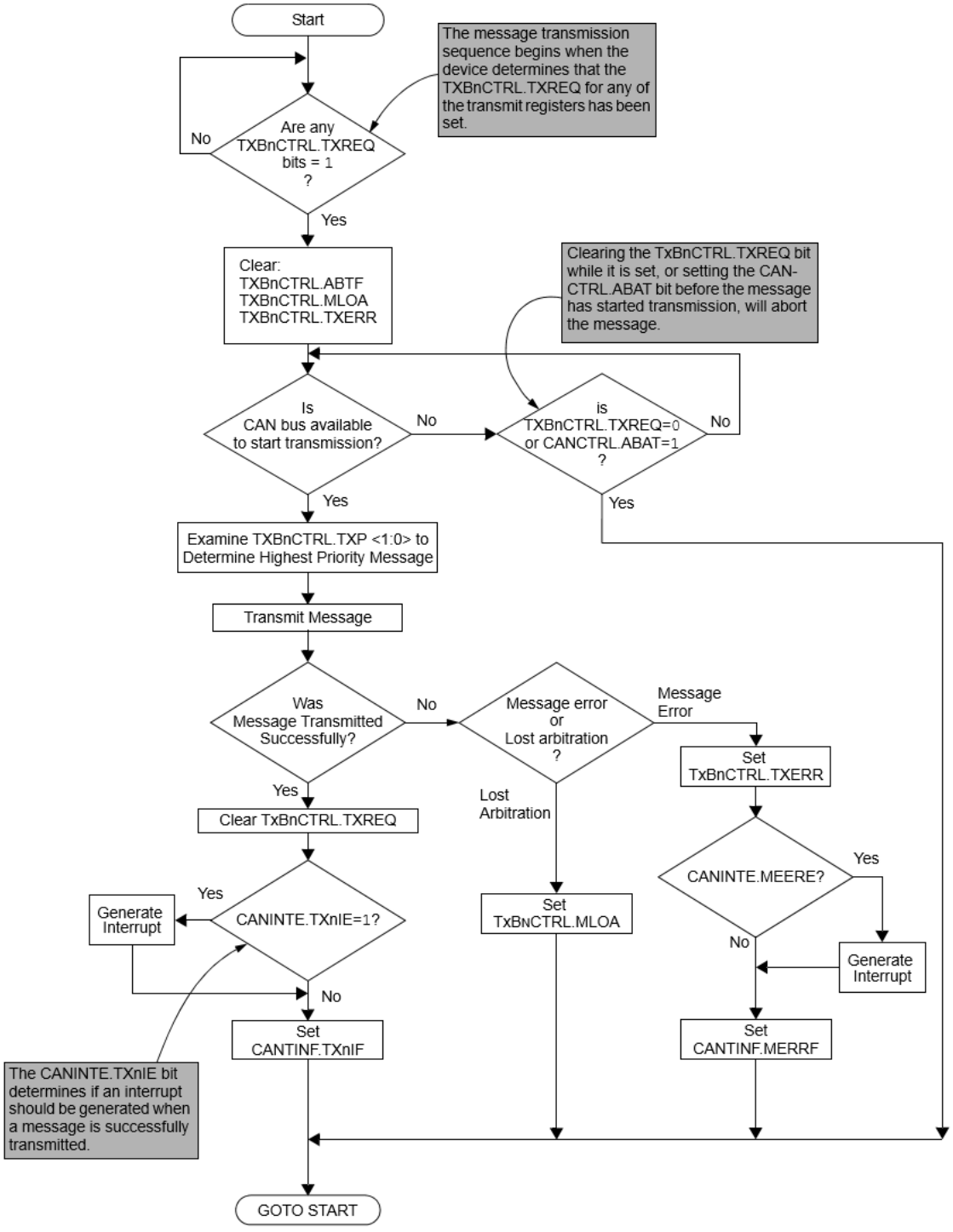


Ilustración 81: Esquema de flujo para la transmisión de mensajes en MCP2515

El MCP2515 implementa tres buffers de transmisión. Cada uno de estos buffers ocupa 14 bytes de SRAM y son mapeados dentro de la memoria del dispositivo.

El primer byte, TXBnCTRL, es un registro de control asociado con el buffer del mensaje. La información en este registro determina las condiciones bajo las cuales el mensaje será transmitido e indica el estado de la transmisión. Los siguientes cinco bytes soportan los identificadores estándar y extendidos así como otra información arbitraria del mensaje. Los

últimos ocho bytes son para los ocho posibles bytes de datos del mensaje que se quiera transmitir.

Cuando ponemos el bit TXBnCTRL.TXREQ = 1 en alguno de los registros de transmisión, el dispositivo inicia el proceso de transmisión del mensaje. Cuando el mensaje ha sido enviado correctamente a través del bus CAN, el controlador limpiará el bit TXBnCTRL.TXREQ para indicar que el buffer de transmisión que se ha utilizado está libre para su uso con un nuevo mensaje. Cuando activemos de nuevo el bit TXBnCTRL.TXREQ el controlador escribirá en el buffer de transmisión el nuevo mensaje que se desee enviar. En una transmisión, como mínimo se deben cargar los registros TXBnSIDH, TXBnSIDL y TXBnDLC. En caso de haber bytes de datos, se tienen que cargar en los registros TXBnDm. Si se usan identificadores extendidos se debe cargar TXBnEIDm y activar TXBnSIDL.EXIDE.

Para que el Arduino reciba una señal de interrupción cuando el mensaje ha sido efectivamente transmitido, es necesario que inicialicemos el bit CANINTE.TXnIE. En caso de generarse una interrupción debida a una transmisión, el controlador activará automáticamente el bit CANINTF.TXnIF.

3.2.1.4.2 – Recepción de mensajes CAN

En la Ilustración 82 se muestra el esquema de flujo en la recepción de mensajes usando el controlador CAN MCP2515.

El MCP2515 incluye tres buffers de recepción. El buffer de ensamblado de mensajes (MAB) recibe el siguiente mensaje disponible en el bus y lo ensambla junto con los demás mensajes recibidos. Si los filtros de aceptación (programables) aceptan los mensajes, estos son transferidos desde el MAB a los otros dos buffers RXB0 y RXB1.

Cuando un mensaje se mueve al buffer de recepción el bit CANINTF.RXnIF correspondiente se activa. Este debe ser limpiado por el MCU para permitir que se reciba un nuevo mensaje en el buffer.

Si el bit CANINTE.RXnIE está activado, se genera una interrupción en el pin INT del MCP2515 para indicar que se ha recibido un mensaje válido.

RXB0 tiene prioridad ante RXB1. Si RXB0 contiene un mensaje válido y se recibe otro, este se guardará en RXB1 (siempre y cuando se haya configurado así, en otro caso el mensaje se descarta).

Los bits de RXBnCTRL.RXM nos permiten aceptar mensajes estándar (00), aceptar mensajes extendidos (10) o recibir todos los mensajes independientemente de los filtros de aceptación (11).

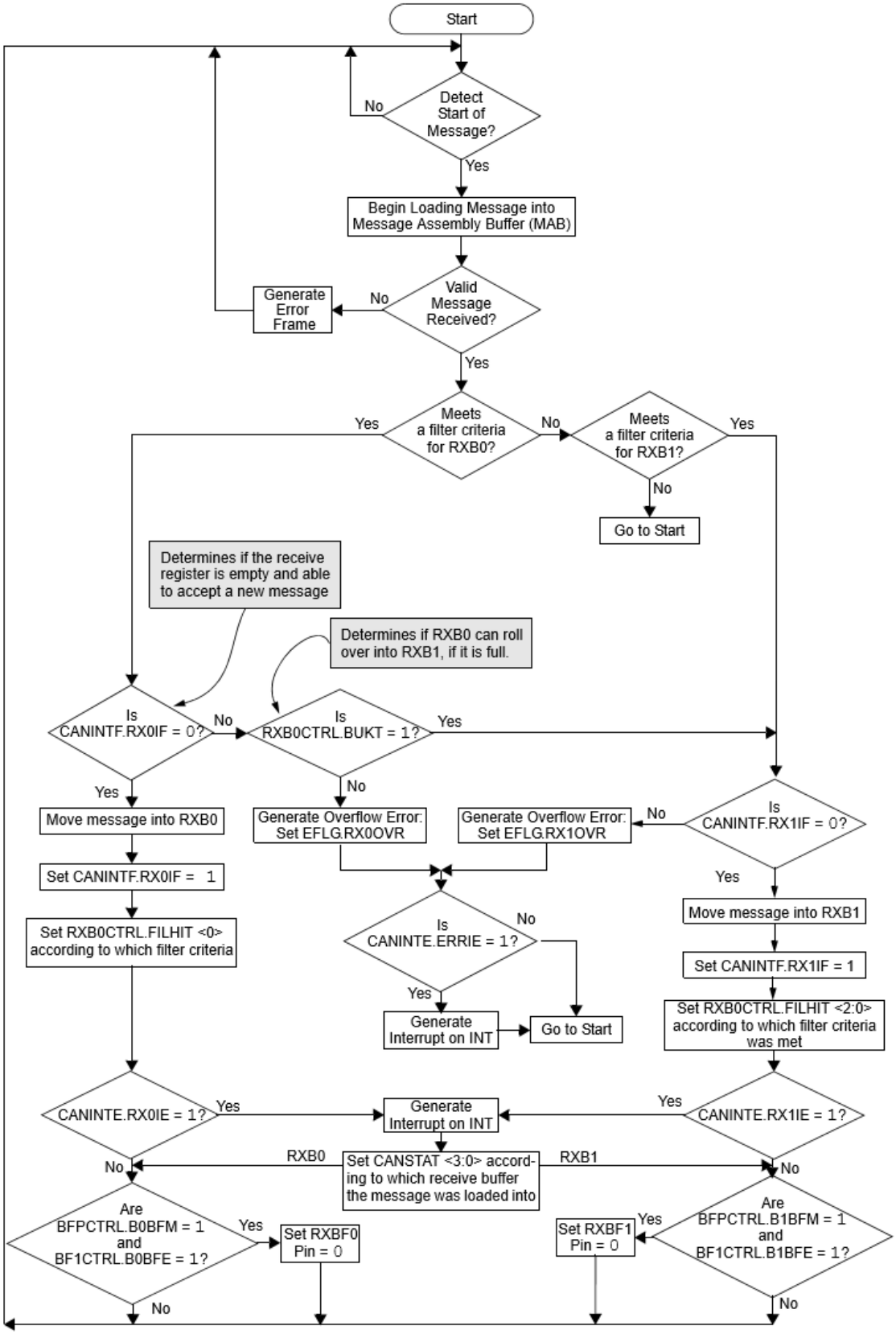


Ilustración 82: Esquema de flujo para la recepción de mensajes en MCP2515

3.2.2 – Flujo del programa

Nuestro código desarrollado para el Arduino es lo más sencillo posible para poder reducir al máximo la carga de procesamiento. Todo ese procesamiento se llevará a cabo una capa más arriba, cuando se implemente la DLL necesaria para añadir nuestro hardware a BUSMASTER, la cual debe encargarse de procesar y adaptar la información entre el controlador CAN y BUSMASTER. Hacemos esto dado que el procesador del Arduino nano tiene menos potencia que cualquier procesador que se use en un ordenador moderno y necesitamos reducir la carga en el Arduino para ser capaces de recibir y enviar la mayor cantidad de tramas posibles. En la Ilustración 83 se muestra la secuencia de funcionamiento seguida por nuestro software.

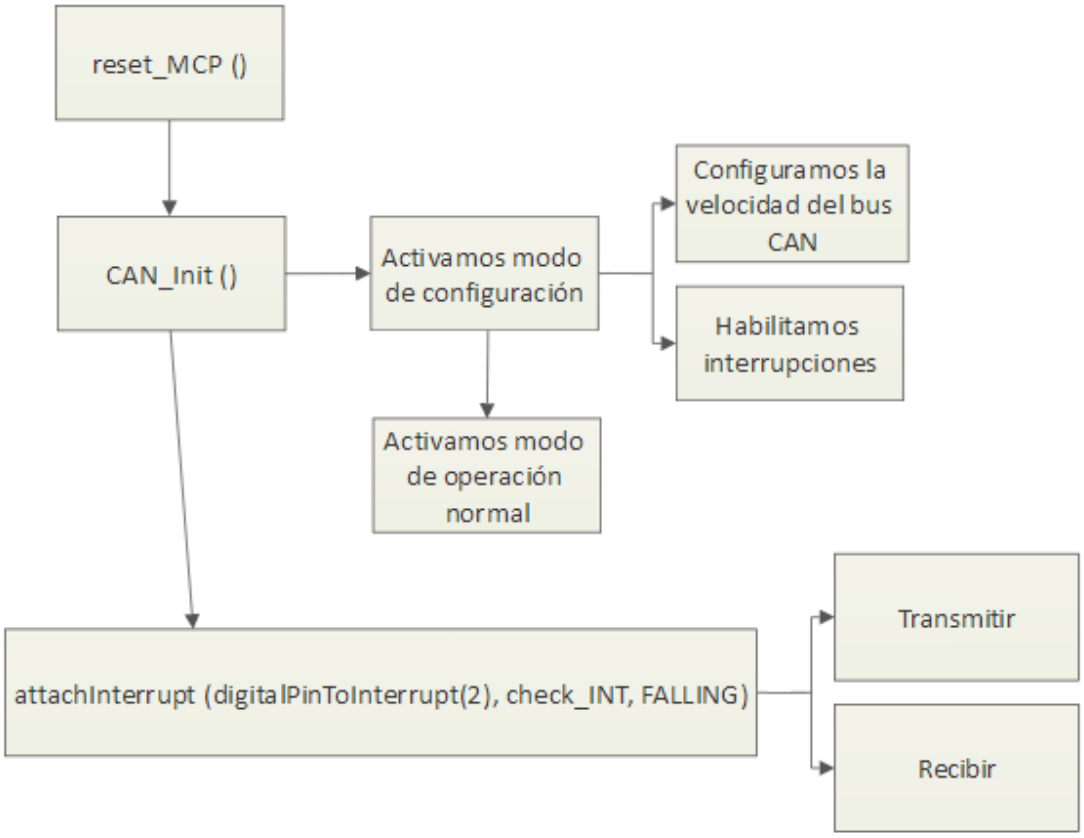


Ilustración 83: Secuencia de funcionamiento del SW de Arduino

Al iniciar nuestro software lo primero que hacemos es reiniciar el MCP2515 para tenerlo en un estado inicial mediante la llamada a la función reset_MCP.

Después se llamará a la función CAN_Init. En este punto es donde tenemos que inicializar nuestro controlador CAN. Lo primero que debemos hacer es activar el modo de configuración el cual nos permitirá modificar los registros del controlador (modificar el registro CANCTRL = 0x80). Una vez que estamos en modo de configuración podremos establecer la velocidad del bus CAN al cual nos vamos a conectar, modificando los registros CNF1, CNF2 y CNF3 en consonancia a lo explicado en el punto 3.2.1.2. Después habilitaremos las interrupciones que deseemos en el registro CANINTE. En nuestro caso, y como se muestra en la Ilustración 84,

hemos activado las interrupciones para los dos registros de recepción y para el primer registro de transmisión TXB0.

```
Can.CAN_Write(MCP_CANINTE, 0x07); // 00000111 Interruption enable
                                  //        . RXB0=1  RXB0 BUFFER 0               ENABLED
                                  //       .  RXB1=1  RXB1 BUFFER 1               ENABLED
                                  //      .   TXB0=1  TXB0 BECOMING EMPTY         ENABLED
                                  //     .    TXB1=0  TXB1 BECOMING EMPTY         DISABLED
                                  //    .     TXB2=0  TXB2 BECOMING EMPTY         DISABLED
                                  //   .      ERRIE=0 EFLG ERROR CONDITION CHANGE DISABLED
                                  //  .       WAKIE=0 WAKEUP INTERRUPT            DISABLED
                                  //.         MERRE=0 ERROR DURING MESSAGE RX/TX  DISABLED
```

Ilustración 84: Activar interrupciones del controlador

Una vez hemos terminado la inicialización del controlador volveremos a activar el modo de operación normal (modificar el registro CANCTRL = 0x00).

La función "attachInterrupt (digitalPinToInterrupt(2), check_INT, FALLING)" nos va a permitir interactuar con el controlador a través de las interrupciones que éste produce. Una interrupción es una señal que interrumpe la actividad normal de nuestro microprocesador y salta a atenderla. En nuestro diseño hemos conectado la señal de las interrupciones del controlador $\overline{\text{INT}}$ (activo en baja) en el pin 12, a la entrada digital D2 del Arduino. Según esto, la función check_INT se iniciará cada vez que la entrada digital D2 del Arduino pase de un estado lógico 1 (HIGH) al estado 0 (LOW).

PROCESO DE RECEPCIÓN

Cuando la interrupción de recepción está activada con CANINTE.RXnIE = 1 y aparece un mensaje nuevo del bus CAN, se generará una interrupción en el pin $\overline{\text{INT}}$ una vez que el mensaje haya sido recibido y cargado en el buffer de recepción correspondiente de forma satisfactoria. El bit CANINTF.RXnIF correspondiente al buffer de recepción que ha cargado el mensaje se activará. Es entonces cuando nuestro programa realizará la llamada a la función check_INT y hará una lectura del registro CANINTF para comprobar cuál de las interrupciones habilitadas ha causado su llamada. Una vez que ha detectado que el bit CANINTF.RXnIF correspondiente está activo, leeremos el identificador, la longitud y el campo de datos del buffer de recepción. Estos datos serán enviados al ordenador para ser interpretados por la DLL de BUSMASTER en el formato de la Ilustración 85.

\n	IDH	IDL	Lenght	Data								\n
Byte 0	1	2	3	4	5	6	7	8	9	10	11	Byte 12

Ilustración 85: Formato de trama enviada a la DLL de BUSMASTER

La trama será enviada en hexadecimal con lo que será posible utilizar como delimitadores a principio y final de trama un salto de línea (\n).

Finalmente limpiaremos el registro CANINTF.RXnIF para permitir que se cargue un nuevo mensaje en el buffer de recepción.

PROCESO DE TRANSMISIÓN

Cuando la interrupción de transmisión está activada con CANINTE.TXnIE = 1 y se desea enviar un mensaje nuevo al bus CAN, se generará una interrupción en el pin $\overline{\text{INT}}$ una vez que el buffer de transmisión asociado esté vacío y listo para ser cargado con un nuevo mensaje. Se activará entonces el bit CANINTF.TXnIF.

Partiendo de la base de que el controlador ha sido reiniciado y los buffer de transmisión están disponibles, para iniciar el proceso de transmisión debe estar activado el bit TXBnCTRL.TXREQ en alguno de los registros de transmisión. Dado que no estamos enlazados a BUSMASTER, si queremos comprobar la transmisión debemos inicializarlo nosotros. Hemos añadido la activación del bit TXB0CTRL.TXREQ, para el cual tenemos también habilitadas las interrupciones, en la función de inicialización CAN_Init. En estas condiciones, se habrá generado una interrupción y ejecutado check_INT. Cuando se detecte que el bit CANINTF.TX0IF está activo, inicializaremos nuestra estructura que contiene los datos que se desean transmitir. A continuación, escribiremos los valores de la estructura en los registros TXB0SIDH, TXB0SIDL, TXB0DLC, TXB0Dn.

Cuando el mensaje se ha transmitido correctamente, el controlador limpiará automáticamente el bit TXBnCTRL.TXREQ para indicar que no tiene transmisiones pendientes.

En el caso de llegar nuevas tramas para transmitir se inicializará de nuevo alguno de los bits TXBnCTRL.TXREQ y se repetirá el proceso.

A continuación se muestran los resultados obtenidos durante el proceso de transmisión y recepción de nuestro sistema. Cabe destacar que estos resultados son válidos para las funcionalidades básicas que se han implementado, sin embargo, habría que añadirle mayor carga de procesamiento a nuestro microcontrolador Arduino cuando queramos que se comunique con la DLL de BUSMASTER. Esta DLL será la que, mediante una comunicación con el Arduino a través del puerto serie, dirá a nuestro microcontrolador cómo deben establecerse los filtros hardware del controlador CAN (filtros de aceptación), cuándo inicializar las transmisiones (activando el bit TXBnCTRL.TXREQ) y qué velocidad debe establecerse para el bus CAN. Esto implicará un programa mayor en el microcontrolador reduciendo significativamente la cantidad de memoria disponible en él, que de por sí es bastante pequeña. Además se le puede añadir más carga al microcontrolador haciéndole distinguir entre tramas con formato estándar o extendido, y se le tiene que habilitar la capacidad de transmitir tramas, para lo cual tiene que escuchar el puerto conectado al ordenador y reenviar al MCP lo que le llegue desde allí. En este caso no solo consumirá más memoria, sino que consumirá ciclos de reloj, haciendo que los tiempos mínimos entre tramas de recepción puedan llegar a aumentar de forma significativa.

3.3 – RESULTADOS

En este apartado mostraremos los resultados obtenidos en el Aula Mercedes-Benz usando el software y hardware diseñado. Durante el proceso de pruebas, hemos conectado nuestro dispositivo a una maqueta junto con la CANcaseXL, la cual usaremos para mostrar los resultados en CANoe.

RECEPCIÓN

En la Ilustración 86 se muestra el resultado de las tramas recibidas del bus CAN en el formato de trama el cual enviaremos a la DLL de BUSMASTER.

```
0060080D1F100004FF80B2
73E008012B020101070AFF
64A0060000000301F3
4100080403000000000A9
7100083F0000000000003F
33E0080FFFFFFFFF0186A0
6AA008FFFFFFFFFFFFFFF
5D200183
704003000000
0D20020C04
71C0080FFF0FFF0FFF0FFF
09600600C1FF3EC100
05E007003C643C000000
5EE006FFFFFFFFF7F
```

Ilustración 86: Tramas recibidas del bus CAN hacia BUSMASTER

En Ilustración 87 la se muestra el resultado de las tramas recibidas del bus CAN con el identificador, RTR y longitud procesados.

```
ID:45  RTR:0  lenght:8  Data  0  FF  0  0  0  0  80  25
ID:401  RTR:0  lenght:8  Data  FD  2  4  F  10  0  0  6
ID:3  RTR:0  lenght:8  Data  D  1F  10  0  4  FF  20  13
ID:208  RTR:0  lenght:8  Data  4  3  0  0  0  0  0  A9
ID:388  RTR:0  lenght:8  Data  3F  0  0  0  0  0  0  3F
ID:355  RTR:0  lenght:8  Data  FF  FF  FF  FF  FF  FF  FF  FF
ID:382  RTR:0  lenght:3  Data  0  0  0
ID:12D  RTR:0  lenght:8  Data  0  F  0  0  20  53  87  1
ID:1C0  RTR:0  lenght:2  Data  0  0
ID:207  RTR:0  lenght:8  Data  10  C3  7F  7F  FF  0  EF  D1
ID:4B  RTR:0  lenght:6  Data  0  C1  FF  3E  C1  0
ID:1  RTR:0  lenght:8  Data  CC  80  CF  FC  FF  3  41  66
ID:38E  RTR:0  lenght:8  Data  F  FF  F  FF  F  FF  F  FF
ID:6  RTR:0  lenght:5  Data  3  3D  0  FF  9
```

Ilustración 87: Tramas recibidas del bus CAN procesadas

TRANSMISIÓN

El proceso de transmisión se ha realizado inicializando las variables de la estructura que posteriormente se cargaba al buffer de transmisión. En la Ilustración 88 e Ilustración 89 se pueden ver las tramas enviadas de forma consecutiva desde el Arduino y cómo CANoe recibe las tramas de forma correcta del bus CAN. En CANoe se ha aplicado un filtro de ID para mostrar únicamente las tramas que hemos enviado.

```
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
ID:27 Longitud:5 Data:EF101112
```

Ilustración 88: Tramas enviadas desde Arduino

Time	Chn	ID	Name	Event Type	Dir	DLC	Data length	Data
4.724655	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.725430	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.726205	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.726981	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.728452	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.730027	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.731523	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.733250	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12
4.734521	CAN 1	1B		CAN Frame	Rx	5	5	0E 0F 10 11 12

Ilustración 89: Tramas recibidas desde el Arduino

Notar que el identificador mostrado en el terminal de Arduino está en decimal, cuyo equivalente en hexadecimal es efectivamente 1B.

3.3.1 – Tasa máxima de recepción

Dado que hemos implementado una solución de bajo coste para sustituir a una herramienta con un precio mucho más elevado, somos conscientes de que debe tener sus limitaciones. En este apartado vamos a analizar la velocidad máxima de recepción que es capaz de soportar nuestro dispositivo con el programa actual (se debe entender que cualquier modificación del programa puede afectar de forma severa al rendimiento aquí observado). Para ello, hemos creado una nueva configuración en CANoe con un nodo CAPL asociado que nos permitirá analizar cuándo nuestro dispositivo empieza a fallar. Ajustamos un envío de tramas cíclico cada 2 ms como se muestra en la Ilustración 90.

En la Ilustración 91 se observa como nuestro dispositivo, el cual está conectado en el mismo bus CAN, es capaz de recibir sin fallos todas las tramas enviadas desde CANoe.

Cambiamos a 1 ms la tasa de transmisión de nuestro nodo CAPL y obtenemos los resultados de la Ilustración 92.

```
variables
{
  message ismael msg = {dlc = 5};
  msTimer timer1;
  int espacio_mensajes=2;
}

on start
{
  setTimer(timer1,espacio_mensajes);
}

on timer timer1
{
  msg.byte(0) = msg.byte(0) + 1;
  msg.byte(1) = msg.byte(1)+1;
 msg.byte(2) = msg.byte(2)+1;
 msg.byte(3) = msg.byte(3)+1;
 msg.byte(4) = msg.byte(4)+1;
  output(msg);
  setTimer(timer1,espacio_mensajes);
}
```

Ilustración 90: Nodo CAPL para envío de tramas cada 2 ms

```
002005B9B9B9B9B9
002005BABABABABA
002005BBBBBBBBBB
002005BCBCBCBCBC
002005BDBDBDBDBD
002005BEBEBEBEBE
002005BFBFBFBFBF
002005C0C0C0C0C0
002005C1C1C1C1C1
002005C2C2C2C2C2
002005C3C3C3C3C3
002005C4C4C4C4C4
```

Ilustración 91: Recepción de tramas desde CAPL cada 2 ms

```
0020050303030303
0020050404040404
0020050606060606
0020050808080808
0020050909090909
0020050B0B0B0B0B
0020050D0D0D0D0D
0020050E0E0E0E0E
0020050F0F0F0F0F
0020051111111111
0020051313131313
0020051414141414
```

Ilustración 92: Recepción de tramas en Arduino IDE desde CAPL cada 1 ms

Como podemos observar, hay tramas que se pierden para esta tasa de recepción. En este punto nos preguntamos si el retardo debido al IDE de Arduino puede ser el causante de que no veamos las tramas reflejadas por el terminal aunque sí sea capaz de recibirlas. Para ello, hemos utilizado la herramienta RealTerm que nos permite recoger los datos recibidos por el puerto USB y guardarlos directamente en un archivo de texto. En la Ilustración 93 se pueden observar los resultados obtenidos.

```
0020056767676767
0020056868686868
0020056A6A6A6A6A
0020056B6B6B6B6B
0020056C6C6C6C6C
0020056E6E6E6E6E
0020057070707070
0020057171717171
0020057272727272
0020057474747474
0020057575757575
0020057676767676
0020057878787878
0020057A7A7A7A7A
```

Ilustración 93: Recepción de tramas en RealTerm desde CAPL cada 1 ms

En ambos casos se pierden tramas de igual manera, y por lo tanto no parece haber relación entre las pérdidas de tramas y su representación por pantalla. En cualquier caso seguiremos usando el RealTerm para las posteriores comprobaciones.

A continuación vamos a realizar una comparativa entre nuestro dispositivo y la CANcaseXL para determinar de forma más ajustada la tasa máxima de recepción (suponiendo que la CANcaseXL es capaz de recibir todas las tramas del bus CAN al que estamos conectados), dado que con los resultados actuales sólo somos capaces de confirmar que la tasa máxima de recepción está entre 1 y 2 milisegundos.

Para ello vamos a guardar las tramas que recibamos durante 1 segundo. Hemos recibido un total de 262 tramas con nuestro dispositivo mientras que con la CANcaseXL se han recibido 266 tramas. Podemos determinar entonces que se pierden un 1.5 % de las tramas del bus CAN con respecto a la CANcaseXL. En la Tabla 8 y la Tabla 9 se muestran fragmentos de las tramas recibidas durante ese segundo, mostrando también las tramas que se han perdido. Podemos ver el tiempo en el cual han sido enviadas, los datos enviados y el incremento de tiempo desde la última trama recibida Δt.

Tiempo	Datos	Δt	Trama recibida
1.725017	09600600C1FF3EC100		Si
1.729217	38200400000000	0.0042	Si
1.730120	05E00700006400000000	0.000903	Si
1.731816	5EE006FFFFFFFFFF7F	0.001696	Si
1.734071	7F600200BC	0.002255	Si
1.734815	00C005023D00FF01	0.000744	Si
1.737086	48A008FFFFFFFFFFFF00FF	0.002271	Si
1.739478	7C4007FFFFFFFFFFFF0F	0.002392	Si
1.740021		0.000543	No
1.741069	25A008000F000020538701	0.001048	Si

1.741981		0.000912	No
1.742708	3800020000	0.000727	Si
1.744508	7C200801037FFD483FFFFF	0.0018	Si
1.745484	28200800A17CFF0000FF00	0.000976	Si
1.749483	73E008001B330101070AFF	0.003999	Si
1.751058	002008C280CFFCFF03F10B	0.001575	Si
1.752018	71C0080FFF0FFF0FFF0FFF	0.00096	Si
1.754337	64A0060000000301F3	0.002319	Si
1.756809	406008FFFFFFFFFFFFFF	0.002472	Si
1.759472	33E0080FFFFFF01B0186A0	0.002663	Si

Tabla 8: Tramas recibidas y perdidas del bus CAN 1

Tiempo	Datos	Δt	Trama recibida
2.011964	6AA008FFFFFFFFFFFFFF		Si
2.014404	704003000000	0.00244	Si
2.019275	7EC0050200000000	0.004871	Si
2.019818	0D20020000	0.000543	Si
2.021658	75E008FF7F7FFFFC077FFF	0.00184	Si
2.024594	09600600C1FF3EC100	0.002936	Si
2.029673	05E00700006400000000	0.005079	Si
2.030681		0.001008	No
2.031824	40E00810C37F7FFF00EFD1	0.001143	Si
2.032680	5EE006FFFFFFFFFF7F	0.000856	Si
2.034504	00C005023D00FF01	0.001824	Si
2.037103	48A008FFFFFFFFFFFF00FF	0.002599	Si
2.041064	25A008000F000020538701	0.003961	Si
2.050487	40800801FFFFFFFFFFFFFF	0.009423	Si
2.051478	756008FF7FFFFFFF7FFF00	0.000991	Si
2.053054	002008C280CFFCFF0321F3	0.001576	Si
2.056822	406008FFFFFFFFFFFFFF	0.003768	Si
2.057798		0.000976	No
2.058765	802008FD02040F10000006	0.000967	Si
2.060677	74C0023F7F	0.001912	Si
2.061860	402008FFFFFFFFFF0000FF	0.001183	Si
2.070356	40800802FFFFFFFFFFFFFF	0.008496	Si
2.081796	2180083BFFFF0303FFFFFB	0.01144	Si
2.086812	20A0083FFFFFFFFF3F07FF	0.005016	Si
2.090644	7500023F7F	0.003832	Si

Tabla 9: Tramas recibidas y perdidas del bus CAN 2

Como podemos observar, las cuatro tramas perdidas tienen un Δt de 0.000543 s, 0.000912 s, 0.001008 s y 0.000976 s. Con estos datos podríamos determinar que a partir de un Δt de 0.001008 segundos las tramas se reciben de forma correcta. Sin embargo, también podemos observar que para los valores de Δt inferiores a 0.001008 segundos e iguales a 0.000903 s,

0.000744 s, 0.000976 s, 0.00096 s, 0.000543 s, 0.000856 s, 0.000991 s y 0.000967 s las tramas se reciben de forma correcta.

Con todo esto, podemos determinar que el periodo mínimo de recepción es de 0.001008 segundos, permitiendo recibir tramas con una tasa inferior pero sin una alta fiabilidad y con un porcentaje de recepción del 98,5% con respecto a la CANcaseXL.

3.4 – INTEGRACIÓN EN BUSMASTER

Con vistas a poder utilizar nuestro sistema de adquisición de datos en BUSMASTER, debemos añadirlo a la lista de hardware soportados por este. Para ello vamos a dar unas pautas a seguir.

Lo primero que debemos hacer es entender cómo podemos hacer tal cosa y con lo cual accederemos a la página oficial de BUSMASTER [1]. Una vez aquí podremos descargarnos el controlador de versiones GIT junto con el código del programa BUSMASTER. Antes de nada conviene leer la documentación proporcionada por los desarrolladores [23].

A continuación se explicarán las bases y la arquitectura de diseño utilizada por los desarrolladores de BUSMASTER de manera breve.

En la Ilustración 94 podemos ver a modo de presentación, una vista de un diagrama de la aplicación BUSMASTER que incluye interacciones, componentes y despliegue.

Nuestro objetivo sería añadir un nuevo software de terceros como los que están en color gris.

DIL_Interface

Este módulo actúa como interfaz entre la aplicación (BUSMASTER) y la DIL (Database Interface Layer) esperada para el bus deseado. Hace las funciones de manager para los requerimientos de la aplicación.

El proceso comienza con la petición por parte del cliente del DIL a través de la función QueryInterfaceDIL(). Si está disponible, DIL_Interface devuelve al cliente el puntero a la interfaz (IDIL_<Bus>). Para tomar el control sobre la interfaz del controlador, la aplicación debe hacer una petición usando el puntero a la interfaz siguiendo la secuencia de la Ilustración 96.

Como podemos ver en la Ilustración 95 tenemos GFS_CAN_DLL, que es la Global Function Set para el driver CAN. Se trata de un conjunto de funciones globales con el mismo conjunto de prototipos, desplegados como DLLs independientes. Sólo uno de los módulos GFS, en un bus concreto, es accesible al mismo tiempo.

Los routers se encargan del mapeado entre IDIL_<Bus> y el GFS_<BUS>_DLL.

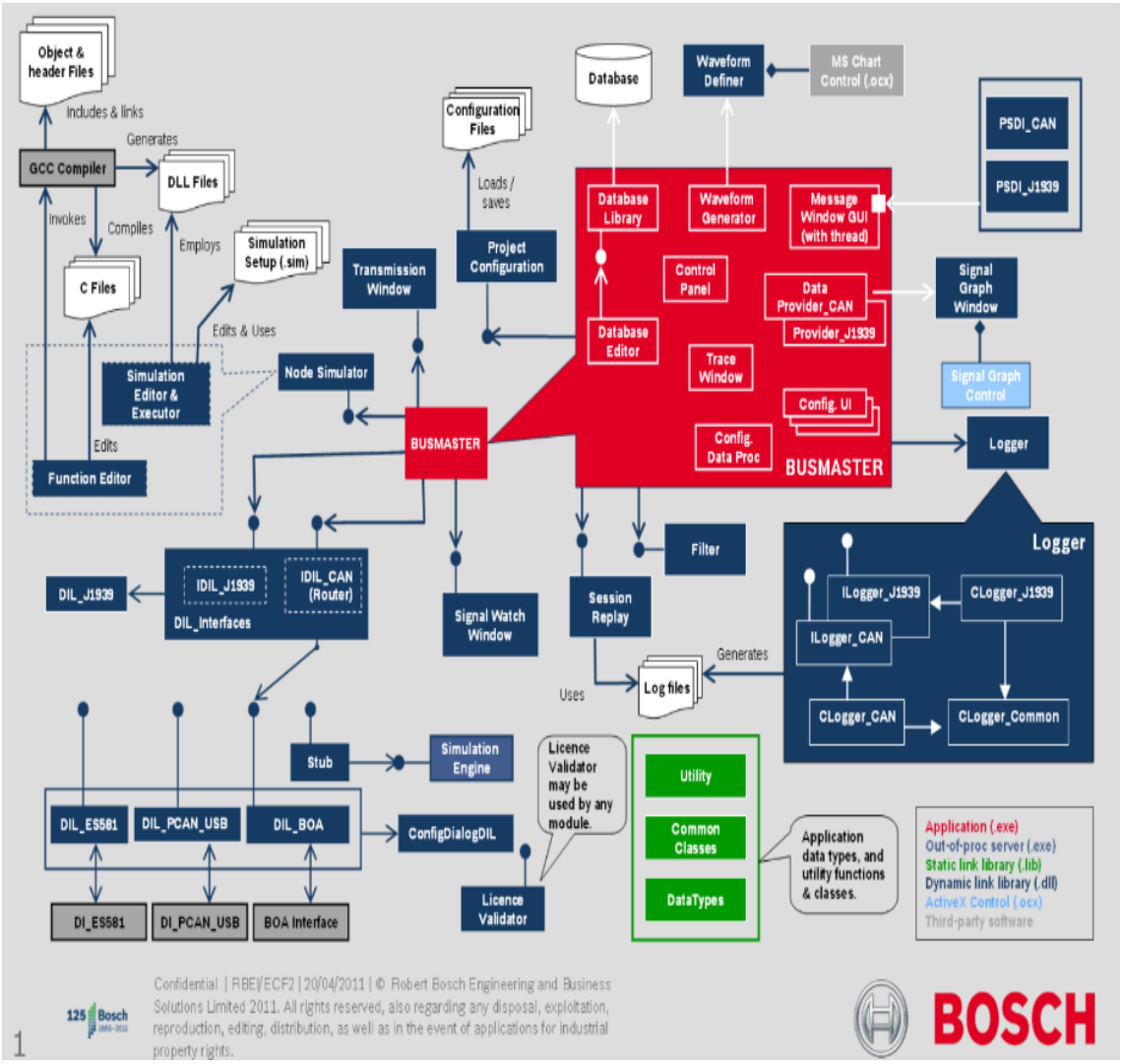


Ilustración 94: Diagrama de la aplicación BUSMASTER [23]

IDIL_CAN

Se trata de la interfaz DIL para el bus CAN. La clase CDIL_CAN será la que haga las funciones de router para mantener un mapeado dinámico entre todas las llamadas del DIL_CAN, desde el cliente y el apropiado interfaz del controlador. Todas las interfaces DIL para un controlador en un bus particular, deben tener el mismo formato.

Para poder soportar un controlador, este debe ser escrito como un socket. Las funciones deben seguir el prototipo de los otros controladores y será implementada como una DLL en BUSMASTER.

Para nuestro caso particular, centrados en el bus CAN y para nuestro controlador (llamémosle SAD), el proceso se resume en los dos puntos siguientes:

1) Crear una extensión en el módulo DIL_CAN, añadiendo para ello otra GFS_CAN_DLL para nuestro controlador DIL::SAD.

2) Programar un socket, que será cargado como una DLL en BUSMASTER, el cual implementará las funciones prototipadas por los otros controladores CAN y que puedan ser interpretadas por CDIL_CAN.

En esta DLL debemos ser capaces de programar las diferentes funciones para poder interpretar el formato de trama enviado desde nuestro dispositivo de adquisición de datos.

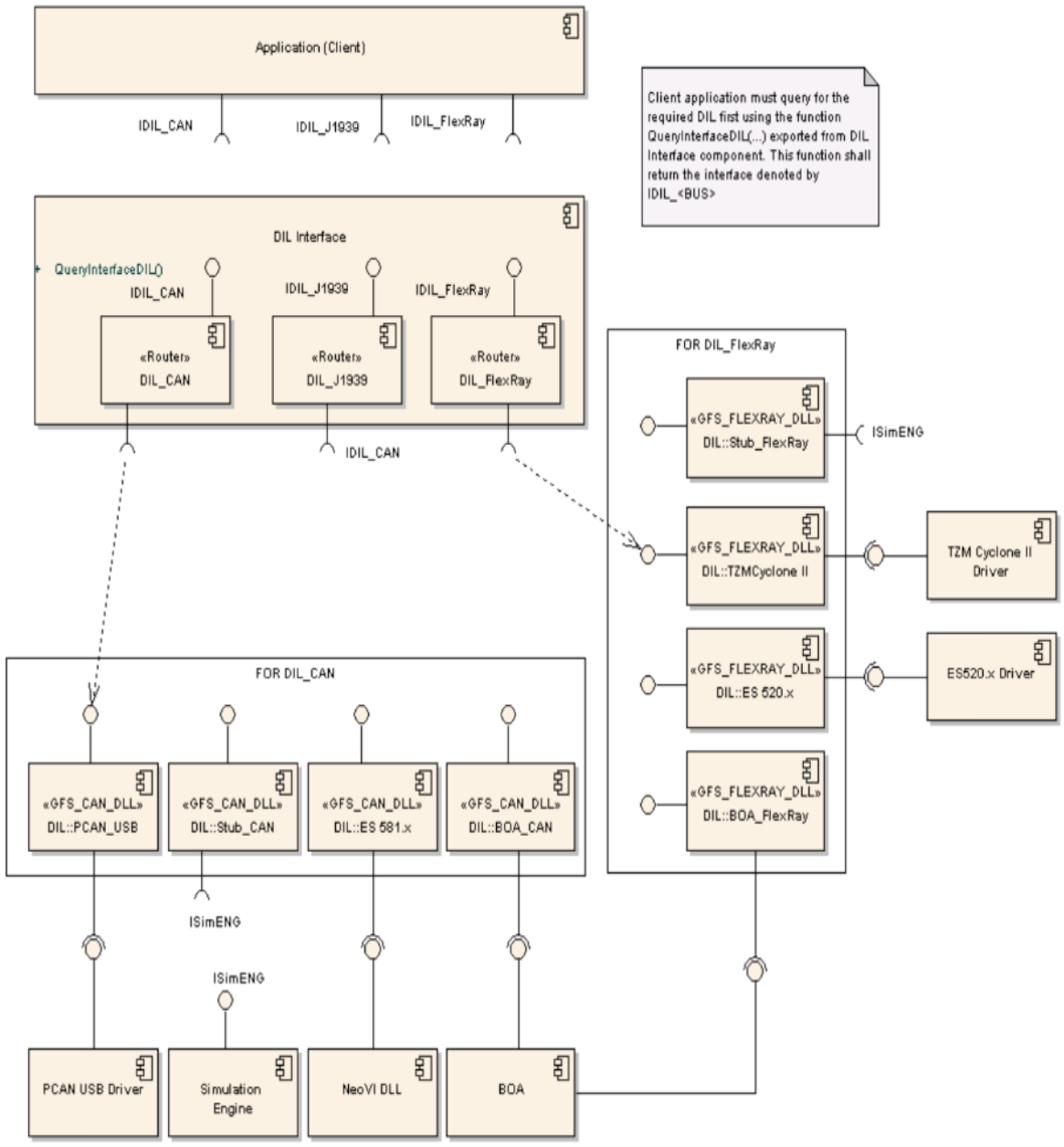


Ilustración 95: Visión de la comunicación entre módulos en BUSMASTER

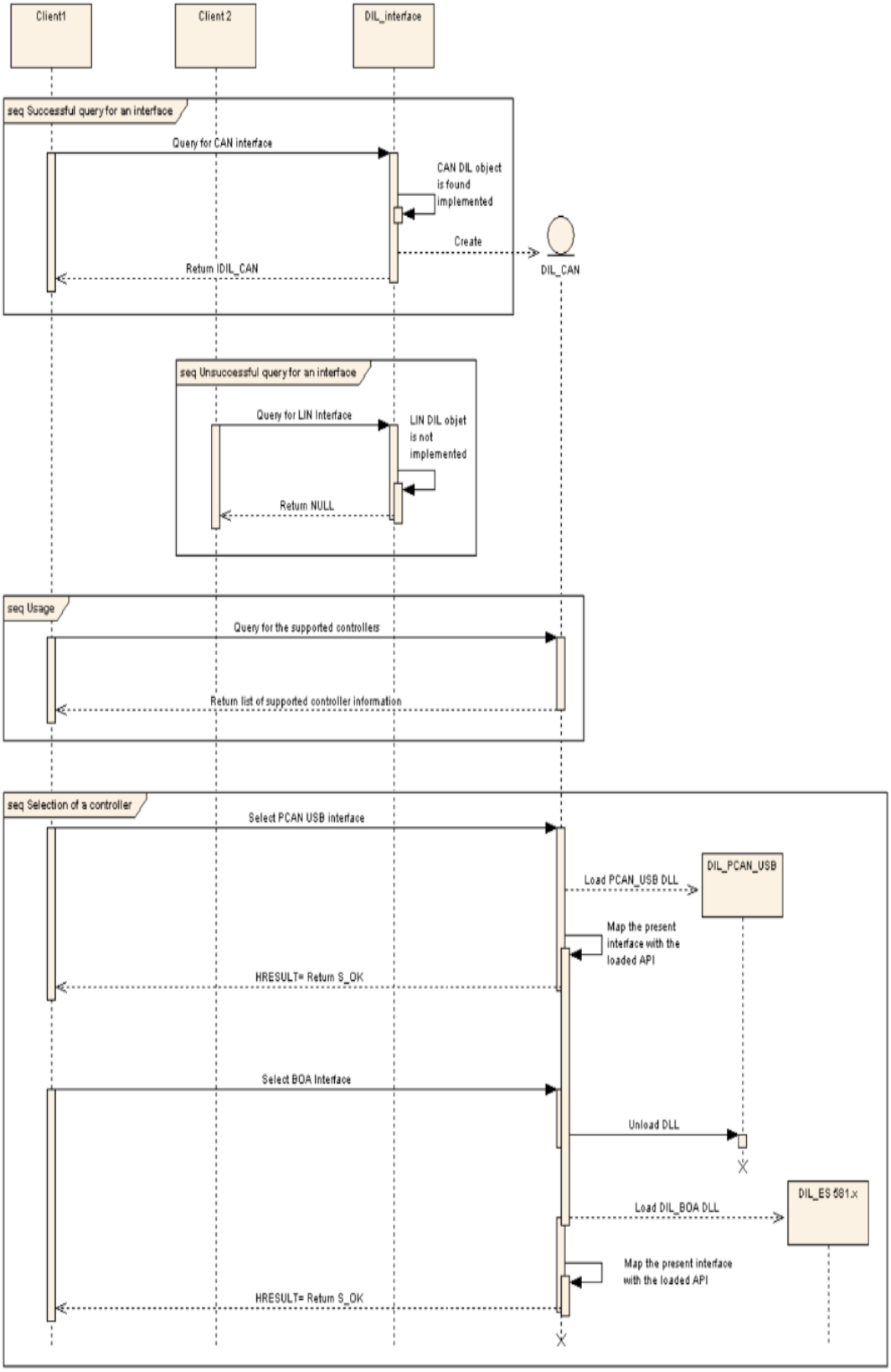


Ilustración 96: Diagrama de flujo entre la Aplicación BUSMASTER y los diferentes módulos

CAPÍTULO 4: CONCLUSIONES Y LÍNEAS FUTURAS

4.1 – CONCLUSIONES

Tras los resultados obtenidos podemos concluir que se han conseguido con éxito los objetivos propuestos al inicio de este Trabajo Final de Grado. Hemos sido capaces de abordar las diferentes fases del proyecto y sentar las bases para dar continuidad a futuras líneas de desarrollo. A continuación desglosaremos cada uno de los objetivos propuestos con las conclusiones sacadas para cada uno de ellos.

- **Diseño y construcción de un dispositivo de adquisición de datos para bus CAN.**

  Este era el objetivo principal y el cual se perseguía desde un primer momento. Hemos sido capaces de desarrollar una herramienta con un funcionamiento robusto. A nivel de hardware finalmente ha sido desarrollada y encapsulada con un diseño sencillo pero completo, el cual nos da confianza para futuras aplicaciones. En cuanto al software, hemos logrado un funcionamiento satisfactorio aunque con proyección para desarrollar filtros de aceptación y recoger tramas de datos extendidas.

  Objetivamente, se buscaba substituir a la herramienta CANcaseXL. Comparándonos con dicha herramienta somos capaces de capturar el 98,5% del tráfico con una tasa máxima de recepción de 0,001008 segundos.

- **Análisis de las prestaciones ofrecidas por el software de código abierto BUSMASTER.**

  A lo largo de este documento se ha facilitado, a modo de guía, la información necesaria para desenvolverse en el entorno de desarrollo BUSMASTER y poder dar el uso a esta herramienta que nosotros pretendíamos. Se ha realizado una comparativa con la herramienta CANoe y podemos concluir que BUSMASTER es suficientemente completo para su uso en docencia aunque con ciertas limitaciones que están en fases de desarrollo.

- **Proporcionar a BUSMASTER una Interfaz Gráfica de Usuario.**

  Este objetivo apareció como una necesidad para equiparar a BUSMASTER con una de las funcionalidades más prácticas que nos proporciona CANoe.

  Hemos sido capaces de sentar las bases desarrollando un socket que funciona de manera correcta para comunicar una aplicación de la herramienta Qt, haciendo las veces de Interfaz Gráfica de Usuario, con otro proceso que debe ser añadido a BUSMASTER. Para completar este objetivo, haría falta compilar la parte del cliente de nuestro socket conjuntamente con BUSMASTER.

- **Estudio general para la integración de nuestro dispositivo de adquisición de datos como un nuevo hardware compatible dentro del software BUSMASTER.**

  Como nos propusimos, hemos realizado un estudio del proceso que deberíamos seguir para añadir nuestro sistema de adquisición de datos a la herramienta BUSMASTER, sentando así las bases para posibles líneas futuras.

- Integrar los paneles a BUSMASTER. Conseguir comunicar el socket creado como GUI para BUSMASTER desde el servidor de Windows hasta el cliente en BUSMASTER.
- Desarrollar la DLL para añadir el dispositivo de adquisición de datos a BUSMASTER. Para completar el uso de nuestra alternativa de bajo presupuesto, es importante ser capaces de dar uso a nuestro sistema de adquisición de datos en el software libre BUSMASTER.
- Adaptar el dispositivo de adquisición de datos para ser compatible con CAN FD. La cantidad de datos generada por un vehículo está creciendo de manera muy rápida, y cada vez más la velocidad del bus cobra mucha importancia, adaptándose a nuevas tecnologías como CAN FD. Sería muy interesante compatibilizar nuestro dispositivo con CAN FD y aprovechar las tasas de datos superiores a 1 Mbps.
- Adaptar el dispositivo de adquisición de datos para que sea inalámbrico y multipunto. Dado el uso tan extendido de estos dispositivos en las fases de testeo, es muy práctico poder conectarse de manera inalámbrica desde el OBD II que se conecta al coche hasta el conector USB del portátil, utilizando por ejemplo la tecnología Bluetooth. Sería interesante poder usar cada emisor/receptor OBD II con cada receptor/emisor USB, sincronizándose mediante un botón, como se hace por ejemplo en la tecnología de los ratones inalámbricos. El dispositivo de adquisición de datos podría comunicarse de manera inalámbrica con cada uno de los conectores o dejarse conectado a uno de los extremos, dependiendo del entorno de trabajo en el que nos encontremos (vehículo real, HIL, VIL,…). Además el poder sincronizar más de un ordenador a un mismo OBD II de forma tan sencilla, puede ser muy útil para trabajos compatibles en paralelo, dado que en la mayoría de los casos los coches o maquetas (dado su elevado precio) son limitados y compartidos por diferentes usuarios.

CAPÍTULO 5: REFERENCIAS Y GLOSARIO DE TÉRMINOS

5.1 – REFERENCIAS

Página oficial de BUSMASTER:

[1] https://rbei-etas.github.io/busmaster/

Bus CAN

[2] http://server-die.alc.upv.es/asignaturas/PAEEES/2005-06/A03-A04%20-%20Bus%20CAN.pdf

[3] https://riunet.upv.es/bitstream/handle/10251/17246/Memoria.pdf?sequence=1

[4] http://www.can-cia.org/can-knowledge/can/high-speed-transmission/

[5] http://www.ti.com/lit/an/slla270/slla270.pdf

CANBUS.pdf

[6] http://www.canbus.galeon.com/electronica/canbus.htm

[7] https://www.kvaser.com/can-protocol-tutorial/

[8] *Apuntes de la asignatura TICA de la Escuela de Telecomunicaciones de la Universidad de Valladolid*

Página oficial del Aula Mercedes de la UVA

[9] http://www.tel.uva.es/personales/mercedes/content1.html

CANcaseXL

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

Makefile

[11] http://www.mingw.org/wiki/LibraryPathHOWTO

Qt

[12] http://bibing.us.es/proyectos/abreproy/12046/fichero/7_Capitulo7.pdf

[13] https://www.qt.io/ide/

Hojas de especificaciones utilizadas para el diseño del sistema de adquisición de datos:

Arduino Nano.

[14] http://docs-europe.electrocomponents.com/webdocs/0db9/0900766b80db99cb.pdf

Controlador CAN-MCP 2515.

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

Conversor de voltaje LM317t.

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

Transceptor CAN-MCP2551.

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

Arduino

[18] http://arduino.cc/en/Reference/HomePage

[19] *Asociación Pisuerga Sport de la Universidad de Valladolid*

SPI

[20] http://www.i-micro.com/pdf/articulos/spi.pdf

I2C

[21] http://www.prometec.net/bus-i2c/

Windows sockets

[22] https://msdn.microsoft.com/es-es/library/windows/desktop/ms737520(v=vs.85).aspx

Documentos de desarrollo BUSMASTER

[23] https://github.com/rbei-etas/busmaster-documents

Otros

[24] http://docplayer.es/6041691-Proyecto-final-carrera-analizador-de-trafico-modular-para-buses-de-automocion-modular-automotive-bus-traffic-analyzer.html

[25] http://www.aficionadosalamecanica.net/can-lin-most-bluetooth-intro.htm

[26] https://vector.com/vi_canoe_en.html

5.2 – GLOSARIO DE TÉRMINOS

ACK: Acknowledgement.

BRP: Bad Rate Prescaler.

CAN: Controller Area Network.

CAN (FD): CAN Flexible Data-rate.

CRC: Cyclic Redundancy Check.

CS: Chip Select.

DIL: Database Interface Layer.

DLC: Data Length Code.

DLL: Dynamic-Link Library.

ECU: Electronic Control Unit.

EOF: End Of Frame.

GCC: GNU Compiler Collection.

GND: Ground.

GUI: Graphical User Interface.

HIL: Hardware-In-the-Loop.

HS CAN: High Speed CAN.

ID: Identifier.

IDE: Integrated Development Environment.

IP: Internet Protocol.

ISO: International Organization for Standardization.

I2C: Inter-Integrated Circuit.

LED: Light-Emitting Diode.

LIN: Local Interconnect Network.

LLC: Logical Link Control.

MAC: Medium Access Control.

MCP: Microchip.

MCU: Microcontroller Unit.

MISO: Master In Slave Out.

MOC: Meta Object Compiler.

MOSI: Master Out Slave In.

MOST: Media Oriented Systems Transport.

MS CAN: Medium Speed CAN.

MSB: Most Significant Bit.

NRZ: Non-Return-to Zero.

OBD: On Board Diagnostic.

PCB: Printed Circuit Board.

RTC: Real Time Clock.

RTR: Remote Transmission Request.

Rx: Receiver.

SAM: Sample Point Configuration.

SJW: Synchronization Jump Width.

SOF: Start-Of-Frame.

SPI: Serial Peripheral Interface.

TCP: Transmission Control Protocol.

TQ: Time Quanta.

Tx: Transmitter.

UDP: User Datagram Protocol.

UIC: User Interface Compiler.

VIL: Vehicle-In-the-Loop.

WLAN: Wireless Local Area Network.

ANEXOS

Añadir nueva ruta a la variable de entorno PATH de Windows

Abrimos el Panel de control de Windows y pulsamos sobre Sistema. Se abre la utilidad de la Ilustración 97:

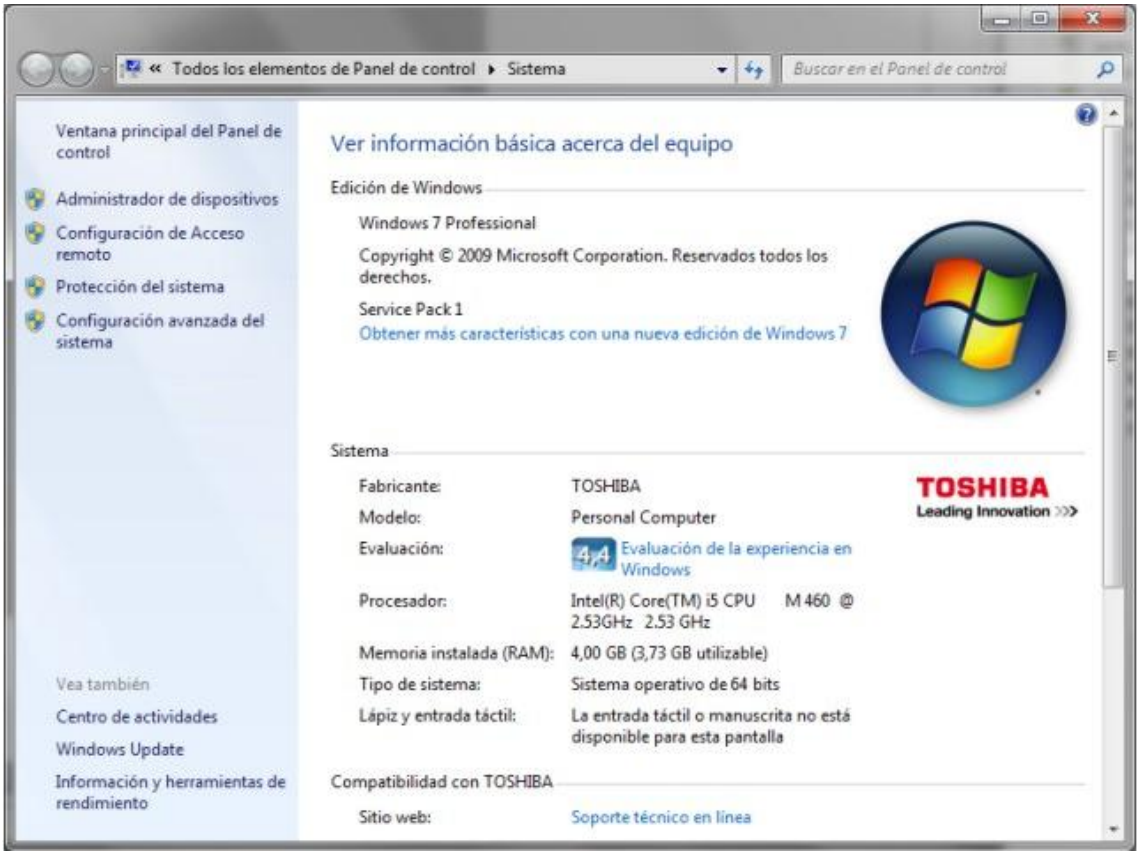


Ilustración 97: Sistema del Panel de Control

Pulsamos en Configuración avanzada del sistema:

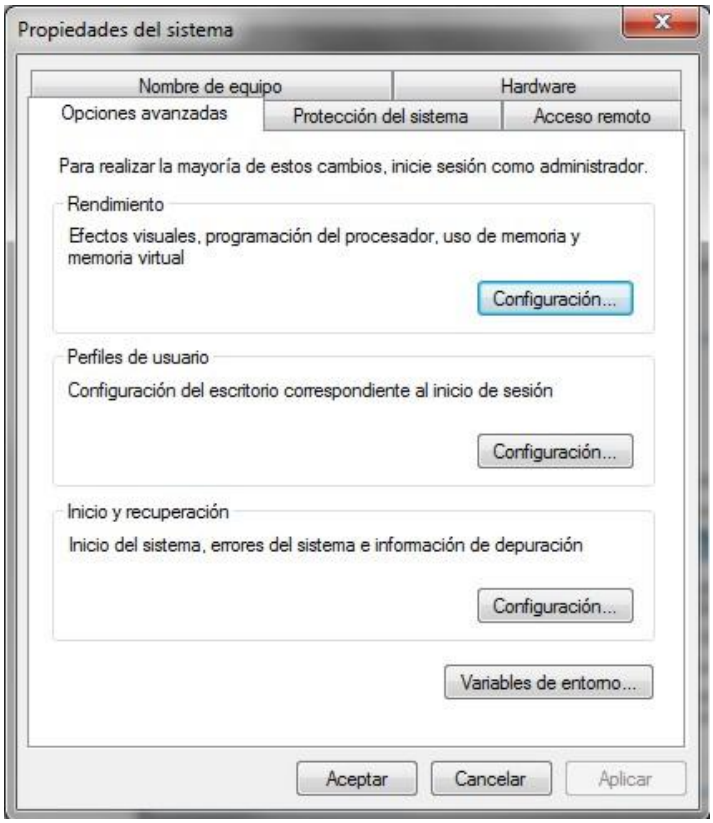


Ilustración 98: Configuración avanzada del sistema

El botón Variables de entorno nos permite ver y modificar las variables de entorno. Son variables que usa el Sistema Operativo para mantener valores globales que puede necesitar en cualquier momento, o listas de carpetas en las que buscar algo, como en el caso de la variable de entorno Path, que contiene las carpetas en las que puede haber archivos ejecutables:

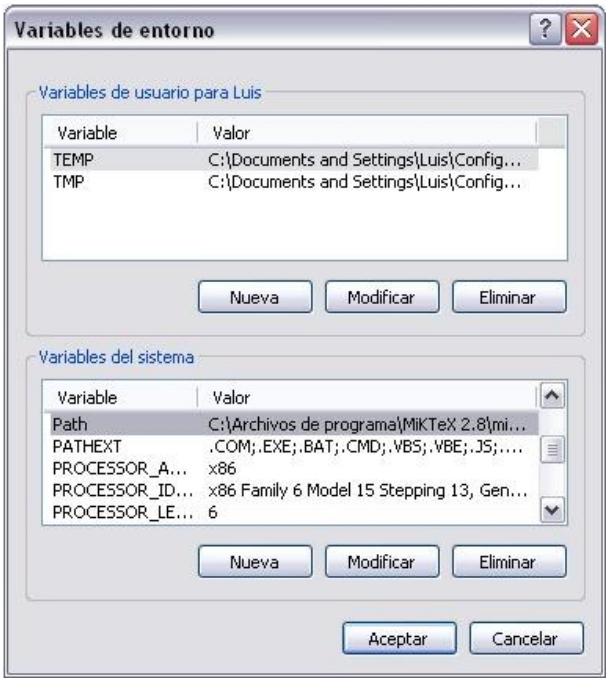


Ilustración 99: Variables de entorno de Windows

Si se pide ejecutar un archivo y ese archivo no está en la carpeta actual, Windows buscará el archivo en las carpetas que se hayan incluido en la variable de entorno Path. Si lo encuentra en alguna de ellas, lo ejecuta. Si no está en ninguna, simplemente indicará que no ha encontrado el programa. Para añadir una nueva ruta, seleccionamos la variable en la lista Variables del sistema, como se muestra en la Ilustración 99, y pulso el botón Modificar.

Para añadir la ruta de MinGW escribo ";C:\MinGW\bin\" (sin las comillas, sin espacios y con el punto y coma del principio), ya que esa es la ruta donde se encuentra la carpeta bin de MinGW en mi caso, a continuación del resto de variables; como se muestra en la Ilustración 100.

Ilustración 100: Modificar la variable Path de Windows

Formato del Makefile de BUSMASTER GCCDLLMakeTemplate_CAN

#Make file for building DLL

DllFile.dll : ObjFile.o

@mingw32-g++.exe -shared -Wl,--dll "<INPUTFILE>.o" -o "<INPUTFILE>.dll" "C:\Program Files (x86)\BUSMASTER_v2.6.3\SimulatedSystems\OBJ\libWrapper_CAN.a"

ObjFile.o : <CPPFILE>

g++ -x c++ -I"<INSTALLDIR>\SimulatedSystems\include" -c "<INPUTFILE>.cpp" -o "<INPUTFILE>.o" -mwindows

Configuración de una aplicación Qt en modo Release

Tras depurar el código en modo Release se generará un archivo ejecutable en un directorio cuyo nombre será "build-panelPruebaQtDesktop_Qt_5_5_1_MinGW_32bit-Release". Si queremos ser capaces de ejecutar al aplicación en máquinas que no tengan instalado Qt, debemos pegar en el mismo directorio donde este el .exe las librerías .dll que se nos soliciten. Estas librerías se pueden encontrar en la ruta "C:\Qt\Qt5.1.1\5.1.1\mingw48_32\bin\".

Condiciones de instalación de Visual Studio Community 2015

Dado que vamos a utilizar librerías de Windows, es importante tener actualizados a la última versión el 'Microsoft .NET Framework SDK'. Instalando Visual Studio Community 2015 nos añadirá por defecto la versión de SDK correspondiente a esta versión del IDE.

Por otro lado seleccionar las funcionalidades durante la instalación: Lenguajes de programación -> Visual C++, Desarrollo de Web y de Windows -> Microsoft Web Developer Tools, Herramientas comunes -> Herramientas de extensibilidad de Visual Studio.