UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Trabajo Final de Grado en Ingeniería Informática

Integración de redes de microcontroladores distribuidos basados en bus CAN. Lado microcontrolador

John Wu Wu

Tutores
Antonio Carlos Domínguez Brito
Jorge Cabrera Gámez
Enrique Fernández Perdomo

31 de marzo de 2013 version 1.0

Agradecimientos

Antes de comenzar con el presente documento me gustaría mostrar mis agradecimientos a todos aquellas personas que han aportado su granito de arena tanto de forma directa como indirecta a este proyecto.

Mi gratitud a Raúl Milla Pérez por haber elaborado la librería Arcan y haberla puesto a disposición pública, lo cual nos ha ahorrado muchísimo tiempo, a stevenh de modelrail.otenko.com/ por resolver nuestras dudas acerca de su librería y a toda la comunidad en torno a Arduino.

Un agradecimiento especial a nuestro tutor, Antonio Carlos Domínguez Brito, que ha demostrado gran interés y nos ha guiado a lo largo del trabajo, a mi compañero José Antonio Lareo Domínguez por su inestimable ayuda durante el transcurso de este TFG y a nuestros amigos que nos han brindado su apoyo y ánimos y sin los que no estaríamos donde estamos.

Por último, quiero mostrar mi más sincero agradecimiento a mi familia que ha estado conmigo en los buenos y los malos momentos, tanto a los que están con nosotros, como a los que se fueron.

A todos, gracias.

Resumen

El principal objetivo de este TFG fue la creación de un protocolo basado en CAN que facilitase la integración de redes de microcontroladores. Dicho protocolo tendría que ser sencillo de usar pero con funcionalidades potentes. Se eligió CAN como base puesto que se trataba de un estándar robusto y ampliamente reconocido. El resultado obtenido fue TouCAN, una librería potente pero amigable al usuario. TouCAN posee dos partes claramente diferenciadas pero estrechamente relacionadas, un lado microcontrolador y un lado supervisor.

El lado microncontrolador que es sobre el que versa este TFG, está diseñado sobre Arduino, una tecnología muy en boga actualmente dada la facilidad de desarrollo y a una comunidad entusiasta. El objetivo principal de esta parte es la de interconectar los microcontroladores entre sí mediante el protocolo definido en TouCAN, proporcionando las clases y los métodos necesarias para ello. Por otra parte proporciona una serie de métodos de comunicación por el puerto serie para la interacción con un PC supervisor.

El lado supervisor está basado en sistemas UNIX, por lo que es compatible con las diversas distribuciones Linux existentes además de ser fácilmente portables a otros sistemas como Mac OS X. Su principal función es la de servir como supervisor del lado microcontrolador. Conectándose a uno de los nodos maestros es capaz de interactuar con el resto de la red, permitiéndole al usuario comunicarse con sus dispositivos en todo momento.

TouCAN tiene el potencial necesario para convertirse en una herramienta libre de amplio uso puesto que es sencillo pero potente, sostenida por una tecnología ampliamente conocida.

Abstract

The main objective of this TFG was the creation of a protocol based in CAN that could make easier the integration of microcontroller networks. This protocol would have to be not only easy to use but also would have to provide powerful functionalities.

We chose CANBUS as base because it was a sturdy and widely known protocol. As a result, we obtained TouCAN, a powerful but user-friendly library. TouCANcan be clearly divided into two closely related parts, the microcontroller side and the supervisor side.

The microcontroller side is the main topic of this TFG, it is designed for Arduino, a very trendy technology because of its easy development and enthusiastic community. The objective of this part is to intercommunicate the microcontroller nodes using the protocol defined by TouCAN, providing the necessary clases and methods. On the other hand, it also provides a number of methods to communicate through the serial-port with a supervisor PC.

The supervisor side is based on UNIX systems, so it is compatible with several Linux distributions and can be easily ported to other systems as Mac OS X. Its main function is to work as the supervisor of the microcontroller side. Connected to a master node through a serial port, the supervisor can interact with the rest of the network, allowing the user to communicate with his devices whenever he wants.

TouCAN has the potential to become a widely used tool because it is easy to use but powerful and backed by a well-known technology.

Índice General

Capítulo 1

Introducción

1.1. Estado actual

Es innegable que los avances tecnológicos que hemos venido experimentando las últimas décadas han facilitado enormemente la vida del ser humano, ya sea en la vida cotidiana o en el diseño electrónico. Uno de los impulsores destacables de este crecimiento han sido los microcontroladores[4], pequeños chips programables que han permitido el abaratamiento de los costes de producción y han facilitado el diseño en el ámbito electrónico.

Si nos paramos a observar un momento, veremos que actualmente vivimos rodedados de microcontroladores: relojes, cafeteras, lavadoras, equipos informáticos, teléfonos móviles, aviones, ropa... La versatilidad de estos dispositivos ha propiciado que se hayan convertido en elementos imprescindible tanto en la electrónica de consumo, como en la industrial. Es en este segundo aspecto donde centraremos nuestra atención.

La industria automovilística observó rápidamente el potencial de estos dispositivos por lo que no tardó en incluirlos en sus vehículos, permitiendo la inclusión de electrónica de bajo coste en los automóviles, como sensores de temperatura, aceite, gasolina, etc. gestionados por microcontroladores. No obstante, se llegó a un punto en que era tal la cantidad de elementos electrónicos en un automóvil que el cableado necesario para realizar las interconexiones resultaba abrumador, por no hablar de que cada fabricante poseía un estándar distinto que dificultaba la inclusión de su tecnología a la de otro. Ante esta situación, a mediados de los 80, la multinacional alemana Robert Bosch GmbH comenzó el desarrollo de un estándar de comunicaciones basado en un bus de datos de comunicación serie destinado a paliar los problemas anteriormente mencionados. El protocolo[6] fue presentado oficialmente en 1986 bajo el nombre de CAN (Controller Area Network)[3] y no sólo solucionaría los problemas de implementación, comunicación y coste de la interconexión entre distintos dispositivos sino que además añadiría una capa adicional para control de errores, ancho de banda, gestión de prioridades

y protocolo de mensajes. Con el tiempo, este protocolo se extendería a otros campos ajenos al automovilístico como es el de la maquinaria industrial, trenes, monorraíles, medicina, aplicaciones en hostelería, aeronáutica, etc.

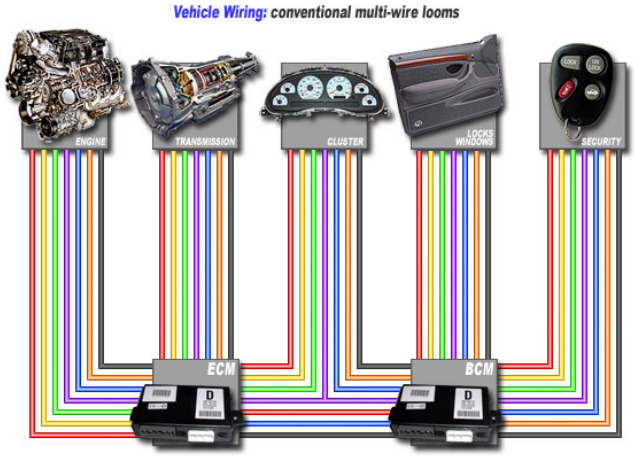


Figura 1.1: Esquema del cableado típico de la electrónica automovilística previa al bus CAN. *Fuente: http://www.canbuskit.com/.*

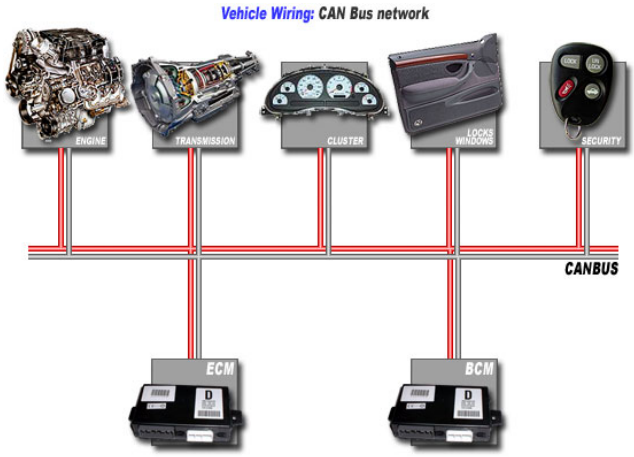


Figura 1.2: Esquema del cableado típico de la electrónica automovilística posterior a la aparición del bus CAN. *Fuente: http://www.canbuskit.com/.*

1.2. Objetivos

El trabajo llevado a cabo y que se explicará a continuación está basado en este protocolo CAN. Nuestra tarea consistió en implementar una API amigable con el usuario que integrase redes de microcontroladores de forma distribuida, con este

objetivo nace TouCAN del cual podemos distinguir claramente dos partes:

- **Lado microcontrolador:** los microcontroladores que componen la red propiamente dicha.
- **Lado supervisor:** un equipo informático algo más potente que los microcontroladores capaz de gestionar en cierta medida la red.

Si bien ambas partes son diferenciables, están íntimamente ligadas, cumpliendo una requisitos de la otra y viceversa por lo que en el siguiente documento será necesario explicarlas ambas para entender las decisiones de diseño tomadas.

Figura 1.3: Propuesta de logo para TouCAN.

Desde el punto de vista académico, este TFG se orientó al aprovechamiento de los conocimientos adquiridos durante la carrera, por lo que también nos marcamos los siguientes elementos de la titulación como objetivos a integrar:

- Algoritmos, programación y estructuras de datos.
- Diseño de Sistemas Basados en Microcontroladores.
- Sistemas Empotadros y de Tiempo Real.
- Algoritmos y Programación Paralela.
- Sistemas Operativos.

Explicamos a continuación los capítulos siguientes que componen la memoria:

- **Análisis del problema:** En este capítulo relataremos cómo se abordó el problema de la creación de un nuevo estándar, así como explicar los detalles que componen el bus CAN para poder entender los posteriores capítulos.
- **Competencias:** De qué manera se cubrieron las competencias exigidas en este TFG.
- **Aportaciones:** Aportaciones realizadas al estado del arte mediante este TFG.
- **Normativa y legislación:** Información sobre la legislación vigente sobre proyectos informáticos que afectan al TFG.
- **Pliego de condiciones:**

- **Metodología/ Plan de trabajo:** una vez se adquirió la suficiente comprensión del protocolo base, tuvimos que familiarizarnos con las placas Arduino, tras lo cual nos dispusimos a comenzar a diseñar el protocolo y su posterior implementación de forma general, antes de ponernos a escribir cualquier código. El capítulo tratará sobre el plan de trabajo durante la semana previa a la parte de la implementación.
- **Requisitos:** Elementos necesarios para la realización de este proyecto.
- **Diseño/ Implementación:** Las clases, métodos y decisiones de diseño se explicarán en este capítulo.
- **Pruebas:** Las pruebas realizadas tanto para comprobar el funcionamiento del protocolo como las pruebas de estrés y medición de velocidades.
- **Conclusiones:** Finalmente, en este capítulo relataremos la experiencia durante la realización de este Trabajo de Final de Grado.
- **Manual de Usuario:** Un manual de uso de cara al usuario final.

Capítulo 2

Análisis del Problema

El objetivo de este proyecto es la integración de redes de microcontroladores para lo cual, como se mencionó en las líneas anteriores, se escogió como protocolo base el bus CAN. Los motivos de nuestra decisión fueron básicamente a que se trata de un estándar "de facto.ªmpliamente extendido, por lo que disponíamos de una gran cantidad de documentación. El primer problema con el que nos encontramos fue entender cómo funcionaba el protocolo a bajo nivel, y es lo que explicaremos a continuación.

2.1. ¿Qué es el bus CAN?

El bus CAN un bus de datos de comunicación serie empleado en sistemas distribuidos en tiempo real. Fue ideado originalmente para la industria del automóvil, no obstante, su robustez y relación calidad/precio ha provocado su difusión a otros sectores. Para la correcta comprensión de la tecnología, a continuación explicaremos en más profundidad las características nombradas en el anterior apartado.

- Económico y sencillo: Uno de los principales objetivos durante su desarrollo fue la simplificación del cableado para la interconexión entre distintos dispositivos, haciendo que el bus sólo requiriese de dos líneas lo cual se tradujo colateralmente en una reducción de los costes de implementación.

- Mensajes o CAN frames: Se trata de las tramas transmitidas por el bus. Están compuestas por una serie de cabeceras, un identificador que define el tipo de mensaje y/o la prioridad y una serie de bits para el dato. En la especificación 2.0A (estándar) la longitud del mensaje está comprendida entre 44 y 108 bits mientras que para la 2.0B (extendida) ésta puede estar entre los 64 y 128 bits.

- Orientado a mensajes: Los nodos no poseen una dirección como tal, ya que el protocolo no contempla unas cabeceras específicas, por lo que cada tipo de

mensaje tendrá un identificador en la red, pudiendo ser aceptado o no por un determinado nodo en función del filtro que tenga programado.

- Detección de errores: Es un protocolo muy robusto frente a problemas de ruido ya que se diseñó específicamente para entornos industriales. Esto lo logra gracias a que el protocolo define una de las cabeceras del CAN frame para la realización de una suma de verificación o checksum.
- Tolerancia a errores: Si se produjese algún error en uno de los nodos, éste no comprometería la integridad de la red ya que el protocolo mismo se encargaría de aislarlo del resto.
- Funcionamiento en tiempo real: Como se ha mencionado previamente, cada mensaje parte con una cierta prioridad determinada por lo que, haciendo uso de un mecanismo especial de arbitraje, se asegura que mensajes más prioritarios lleguen antes en función de su identificador, que es lo que define su prioridad.
- Ancho de banda regulable: para cualquiera de los nodos que componen la red del bus CAN es posible regular su velocidad de transmisión de forma sencilla, pudiendo esta variar desde 125 Kbps (baja velocidad tolerante a fallos) a 1 Mbps.

2.1.1. Capa física

Originalmente, la especificación[10] del protocolo CAN definía únicamente una capa de enlace de datos, haciendo referencia a la capa física como un término abstracto, no obstante, a día de hoy, el estándar define de forma concreta la definición para la parte eléctrica de la capa física.

Esta especificación está basada en dos estándares de transmisión, ISO 11519 (baja velocidad tolerante a fallos, destinada originalmente al control de mecanismos no críticos del automóvil tales como puertas, techo corredizo, luces, etc.) e ISO 11898 (alta velocidad destinado a sistemas como el control del motor).

Para poder lograr una mejor comprensión de los posibles valores del bus se ha de definir el concepto de valor dominante y valor recesivo. Valor dominante es aquel que, siempre que varios nodos transmitan a la vez en el bus, sea el que marque el valor. Para el bus CAN dominante sería el “0” y el recesivo el “1”. Los niveles eléctricos que definen estos valores son la diferencia de potencial entre las dos líneas que componen el bus, CAN L y CAN H.

2.1.1.1. Baja velocidad tolerante a fallos

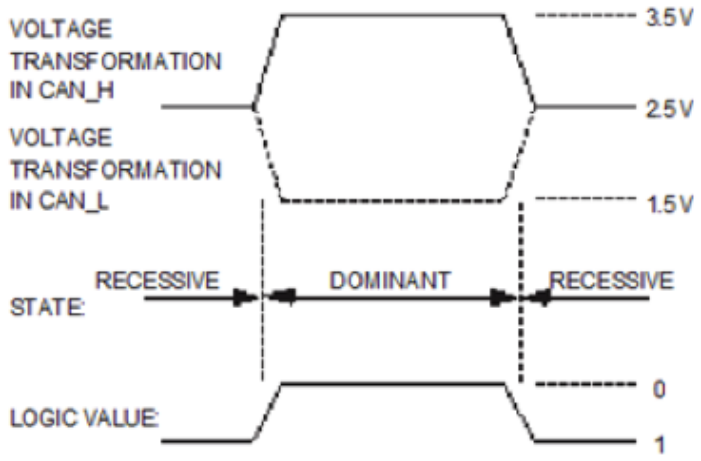


Figura 2.1: Niveles eléctricos para la definición del valor dominante y recesivo (baja velocidad)

Es necesario que cada uno de los transceptores de los nodos CAN estén conectados a una resistencia de 120 ohms, para reducir la velocidad de transmisión y sean detectables los fallos en la red.

2.1.1.2. Alta velocidad

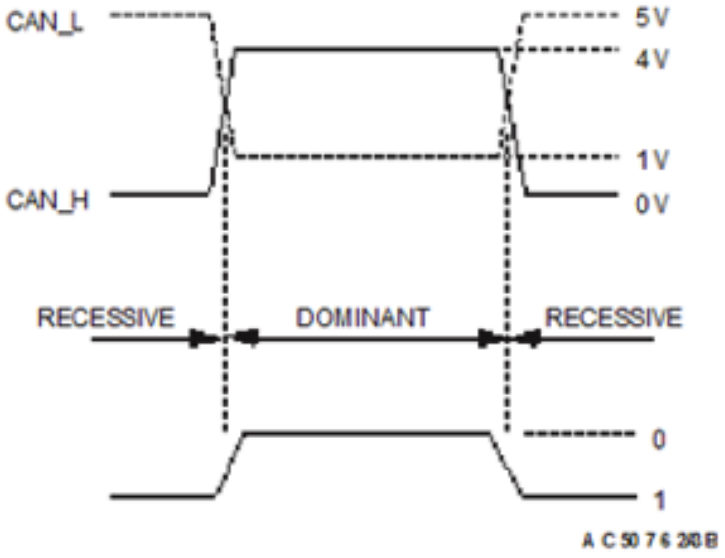


Figura 2.2: Niveles eléctricos para la definición del valor dominante y recesivo (alta velocidad)

En este caso, la red CAN ha de estar conectada a cada extremo a una resistencia de 120 ohms, en vez de conectarlos a cada transceptor de cada nodo CAN ya que

limita la velocidad de transmisión. De esta forma se alcanza a 1Mbps de velocidad de transmisión, no obstante,los errores de transmisión a este nivel, deberán de ser controlados por otros niveles o capas.

2.1.1.3. Reloj, sincronización, bit timing

Es en la capa física donde se lleva a cabo la sincronización para la transmisión/-recepción de tramas. A pesar de no tener una línea separada para un reloj, éste tampoco se transmite en el bus como puede pasar en otros protocolos. Es cada nodo el que se encarga del timing de cada bit, y esto se logra dividiendo el tiempo de transmisión de bit en distintas etapas. Es preciso definir una serie de términos para comprender este mecanismo. Tasa de bit nominal: número de bits transmitidos por segundos. Tiempo de bit nominal: tiempo necesario para la transmisión de un bit. Este tiempo se divide en las siguientes etapas:

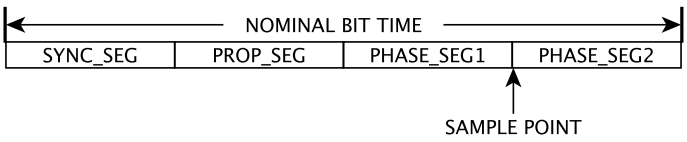


Figura 2.3: Fases de muestreo de bit.

Cada etapa requiere una serie de cuantos de tiempo (quantum), el cual es la unidad discreta de tiempo mínima con la que trabaja cada nodo CAN. Dicho cuanto es resultado de una división programable (prescaler) de la frecuencia oscilador a la que funciona el nodo CAN.

Etapa	*Longitud en cuanto*	*Descripción*
SYNC_SEG (Segmento de sincronización)	1	Se usa para sincronizar los nodos del bus.
PROP_SEG (Segmento de propagación de tiempo)	Programable 1-8	Este segmento del tiempo de bit se usa para compensar el retardo físico intrínseco a la red.
PHASE_SEG1 (Segmento de Fase de Buffer 1)	Programable 1-8	Se usan para compensar los errores por cambio de flanco.
PHASE_SEG2 (Segmento de Fase de Buffer 2)	El máximo entre Tiempo de Proceso de Información y PHASE_SEG1	
SAMPLE POINT (Punto de Muestreo)	0	Se trata del punto donde se muestrea el estado del bus para su lectura y se interpreta el valor del bit. Se produce entre PHASE_SEG1 y PHASE_SEG2.
Information Processing Time (Tiempo de Proceso de Información)	<= 2	Es el segmento de tiempo, comenzando con el punto de muestreo reservado para el cálculo del bit posterior.

Figura 2.4: Fases de muestreo de un bit en detalle.

Tipos de sincronización

- **Hard Synchronization:** Se realiza en el bit inicial. Consiste en resetear la cuenta del tiempo de bit.
- **Resynchronization:** Se realiza para compensar los desfases entre el oscilador del emisor y el receptor, alargando o reduciendo PHASE_SEG1 y/o PHASE_SEG2

2.1.2. Capa de enlace de datos

Define el método de acceso al medio así como los tipos de tramas para el envío de mensajes, las cuales describiremos a continuación.

2.1.2.1. Tipos de trama

- **Trama de datos (Data Frame):** Se trata de la trama que se transmiten usualmente a través del bus para el envío de datos. Dada su importancia se detallará en el siguiente sub-apartado.
- **Trama remota (Remote Frame):** Es la trama enviada por el nodo que solicita un dato. Destacamos que si bien esta trama, generalmente es importante al hacer uso del bus CAN de forma normal, era incompatible con nuestro protocolo por lo que no se llegó a usar.
- **Trama de error (Error Frame):** Se transmite cuando uno de los nodos de la red detecta un error en el bus.
- **Trama de sobrecarga (Overload Frame):** Indica que el nodo que la transmite, requerirá de cierto tiempo extra antes de poder recibir otra trama de datos o remota.
- **Espaciado entre tramas (Inter-Mission):** Entre cada trama transmitida, se transmite una secuencia predefinida para establecer el final de trama.
- **Bus ocioso:** La longitud de este periodo es arbitraria. El bus se detecta como libre y cualquier nodo con intención de transmitir una trama podrá acceder a él.

2.1.2.2. Trama de datos y trama remota

Ambas tramas comparten una serie de campos comunes que son los que se muestran en la siguiente figura:

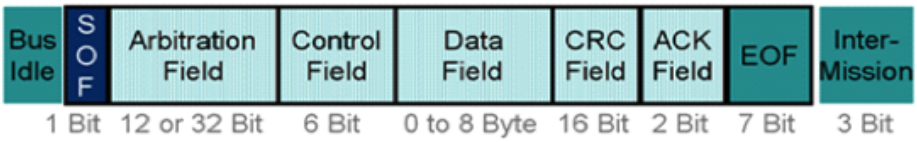


Figura 2.5: Bits de trama de datos y trama remota. *Fuente: http://www.softing.com/*

Veamos a continuación los bits que componen cada campo y su significado:

Nombre	***Longitud en bits***	***Descripción***
SOF (Start of frame)	1	0
Arbitration Field: Identifier	11	Identifica tanto al tipo de mensaje como su prioridad
Arbitration Field: RTR (Remote Transmission Request)	1	Indica si se trata de una trama de datos (0) o remota (1)
Control Field: IDE (Identifier Extended Bit)/r1 (reserved bit 1)	1	Si se trata de una trama estándar (0) o extendida (1)
Control Field: r0 (reserved bit 0)	1	Bit reservado
Control Field: Data Lenght Code	4	Indica la longitud del segmento de datos (nº de bytes)
Data Field	0-64	Los bits del datos transmitidos en la trama, 0 en caso de una trama remota
CRC Field: CRC	15	Checksum de la trama
CRC Field: CRC delimiter	1	Separa el campo CRC del resto de la trama
ACK Field: ACK Field	1	Indica que al menos un nodo ha recibido la trama con éxito
ACK Field: ACK Delimiter	1	Análogo al CRC delimiter
EOF: End Of Frame	7	Entre cada trama de datos y/o trama remota han de haber 7 bits consecutivos con valor recesivo (1)

Figura 2.6: Campos de una trama de datos o remota CAN.

2.1.2.3. Gestión de Acceso al Bus

Existe la posibilidad de que en un instante de tiempo determinado varios nodos quieran transmitir de forma simultánea a través del bus, para estos casos, el protocolo CAN proporciona un método de arbitraje, que es el que se muestra a continuación:

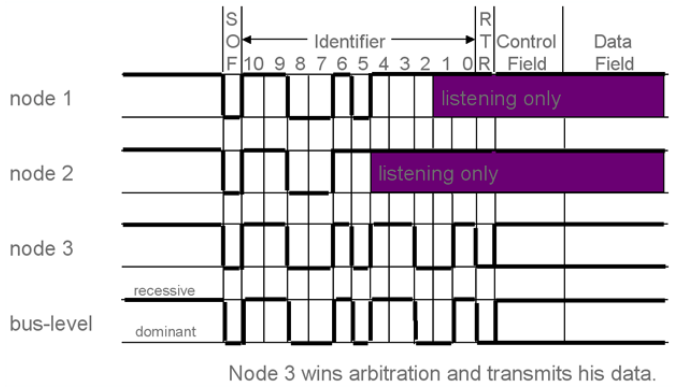


Figura 2.7: Ejemplo de arbitraje de bus: El nodo 3 gana el arbitraje. *Fuente: http://www.softing.com/*

Todos comienzan transmitiendo el identificador del mensaje (aparte obviamente del bit SOF). Mientras todos los nodos coincidan en el envío del un valor recesivo o dominante de forma simultánea, seguirán transmitiendo, no obstante; llegados a un punto, si uno o varios de los nodos transmiten un valor recesivo mientras existe al menos un nodo transmitiendo un valor dominante, será este último el que gane el arbitraje del bus, deteniendo los demás su transmisión y quedándose a la escucha. De esto se deduce que la forma de crear mensajes más prioritarios es dándoles un identificador de bajo valor decimal, ya que comenzaría con más valores dominantes (0).

2.1.2.4. Filtrado de mensajes, enmascaramiento

La subcapa LLC de la capa de enlace de datos proporciona a los nodos un mecanismo de filtros y máscaras para la aceptación de mensajes. Las máscaras son un conjunto de 11 bits por defecto a 0 donde se ponen a 1 indicando la posición del bit en el identificador del mensaje que se quiere verificar mediante el filtro.

Veamos ahora un ejemplo, tenemos dos identificadores en formato estándar de 11 bits que representarán un rango de mensajes el 0x120 y el 0x13F.

Observando individualmente los bits:

	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x120	0	0	1	0	0	1	0	0	0	0	0
0x13F	0	0	1	0	0	1	1	1	1	1	1

Figura 2.8: Bits de id de mensaje

Como queremos definir un rango, nos fijamos en la parte alta de cada tetra de bits (excepto los más significativos que son tres) que es invariante. Si obligamos a que se verifiquen dichos bits tendríamos las siguientes máscaras y filtros.

	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Mask	1	1	1	1	1	1	0	0	0	0	0
Filter	0	0	1	0	0	1	0	0	0	0	0
Passed	0	0	1	0	0	1	x	x	x	x	x

Figura 2.9: Configuración de máscaras y filtros

Obligamos a que se verifique la parte invariante entre los dos identificadores, mediante la máscara, a continuación se comparará con los filtros qué tipo de mensaje es. Si está en el rango entre 0x120 y 0x13F se aceptará, cualquier otro identificador será descartado.

2.2. Arduino

Como hemos podido observar, el protocolo conlleva un gran nivel de procesamiento de datos por lo que implementarlo mediante software sería impensable, sobre todo si tenemos en cuenta que el código habría de correr sobre un microcontrolador. Es por ello que nos decidimos por una solución hardware para sustentar este protocolo. Qué microcontrolador escoger fue también otro escollo que hubo que superar, por lo que tras pensarlo durante un tiempo escogimos la tecnología Arduino[1] por su versatilidad, amplio soporte de la comunidad y la relación cantidad precio. En este sentido nos decantamos por la placa **Arduino UNO[2]** ya que era la que más se ajustaba a nuestros intereses.

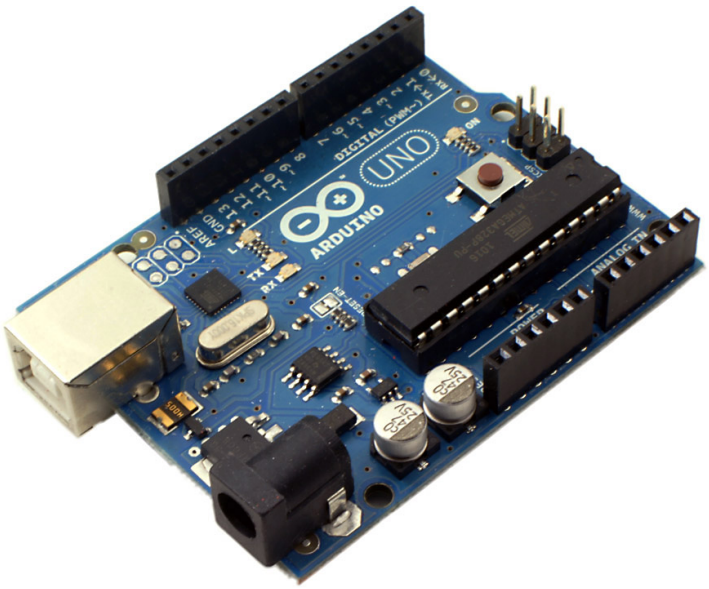


Figura 2.10: Placa Arduino UNO. *Fuente: http://arduino.yvision.kz/*

- **Microcontrolador:** ATmega328.

- **Voltaje de funcionamiento:** 5V.
- **Memoria Flash:** 32KB de los cuales 0,5 son usados por el gestor de arranque.
- **SRAM:** 2KB.
- **EEPROM:** 1KB.
- **Velocidad de reloj:** 16Mhz.

Una de las virtudes de Arduino es que sus características son ampliables mediante otras placas denominadas **shields[8]**, por lo que si bien el Arduino UNO no poseía soporte para CAN, podíamos incorporarle un shield que le añadiese esa funcionalidad. El shield escogido fue el de la marca **Sparkfun, CAN Bus Shield[9]**, el cual se basaba en un **PIC MCP2515[11]** como crontrolador para el protocolo CAN. Así pues, para poder desarrollar sobre esta plataforma fue necesario conocer su configuración interna.

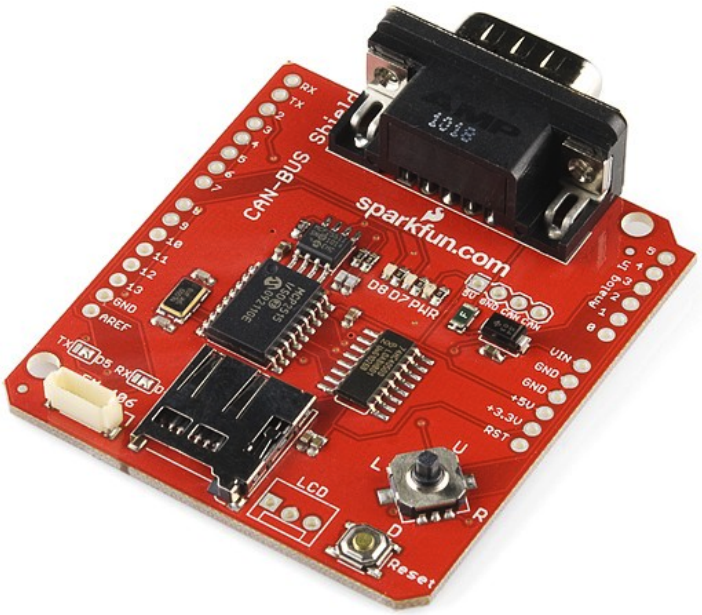


Figura 2.11: Shield "Sparkfun CAN Shield". *Fuente: http://reflexiona.biz/*

En el siguiente apartado detallaremos más acerca del MCP2515, el controlador que gobierna el shield

2.3. El MCP2515

El MCP2515 es un controlador CAN autónomo desarrollado para simplificar aplicaciones que requieran una interfaz con un bus CAN. El dispositivo consta de tres bloques principales

- El módulo CAN, que incluye el gestor del protocolo CAN, máscaras, filtros y buffers de recepción y transmisión.
- La lógica de control y los registros usados para la configuración del dispositivo.
- El bloque de protocolo SPI.

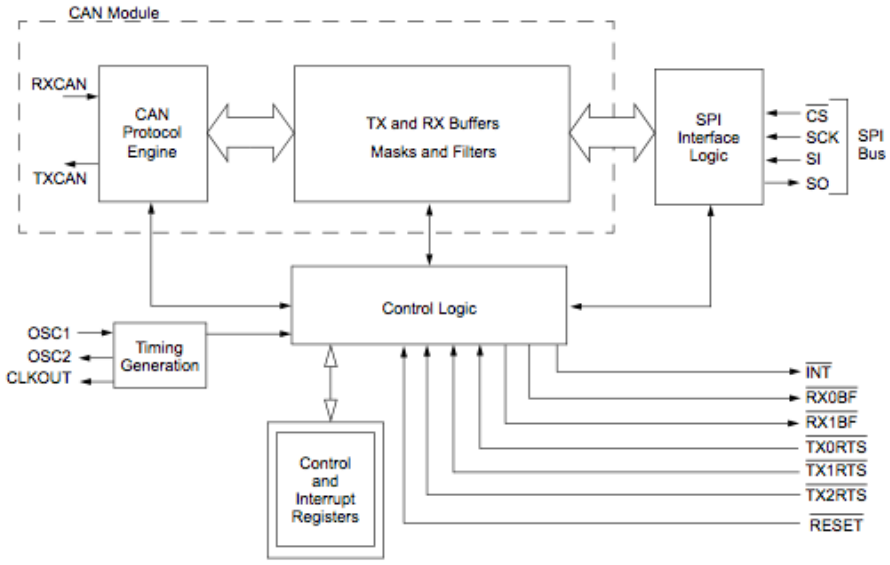


Figura 2.12: Diagrama de los principales bloques del MCP2515. *Fuente: Microchip MCP2515 Datasheet.*

2.3.1. El módulo CAN

El módulo CAN es el encargado de manejar todas las funciones tanto para recibir como para transmitir mensajes sobre el bus. Los mensajes son transmitidos tras haber cargado los registros de control y el buffer de mensaje adecuados. La transmisión es iniciada usando los bits del registro de control vía la interfaz SPI o mediante los pines que permiten la transmisión. El estado del dispositivo así como los errores detectados pueden ser comprobados en los registros apropiados. Cualquier mensaje detectado en el bus es revisado por si contiene algún error y en caso de no tenerlo, para ver si cumple alguno de los filtros del usuario para guardarlo; a posteriori, en uno de los dos buffers de recepción. Cabe destacar que éste es el módulo que a nivel de hardware se encarga de gestionar el protocolo por lo que

abstrae al usuario de toda la complicación que conlleva el bajo nivel, dejándole sólo la lógica de control.

2.3.2. La lógica de control

Estamos ante el bloque que controla la configuración y las operaciones del MCP2515, interconectando los otros bloques con el fin de pasar datos e información de control.

Los pines de interrupción se proporcionan para dar una mayor flexibilidad al sistema. Existe un pin de propósito múltiple para cada uno de los buffers de recepción. El uso de los pines específicos es opcional. Los pines de propósito general así como los registros de estado (accesibles mediante la interfaz SPI) tambien son válidos para comprobar la correcta recepción de un mensaje.

De forma adicional, existen también tres pines disponibles para iniciar inmediatemante una transmisión de un mensaje que ya haya sido cargado en uno de los tres buffers de transmisión. El uso de estos pines es opcional puesto que la transmisión de un mensaje puede ser iniciada usando los registros de control (accesibles mediante la interfaz SPI).

2.3.3. El bloque de protocolo SPI

El microcontrolador(en nuestro caso un ATmega328 que se encuentra en el Arduino UNO) se conecta al dispositivo a través de la interfaz SPI. Mediante los comando de lectura y escritura SPI, se logra la lectura y escritura en los registros. Por lo tanto, este bloque proporciona otro protocolo de comunicación, el SPI , que es accesible con el único fin de comunicar la placa Arduino con el shield.

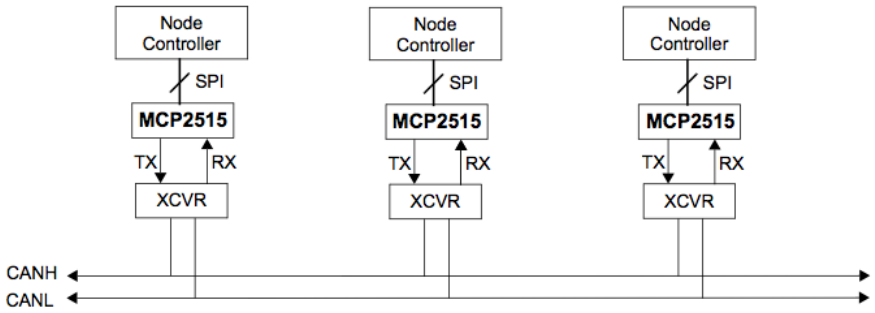


Figura 2.13: Ejemplo de red CAN: El microcontrolador conectado al MCP215 y éste, a su vez, conectado al bus. *Fuente: Microchip MCP2515 Datasheet.*

A continuación se muestra el juego de instrucciones que provee SPI para la comunicación.

2.3.3.1. Juego de Instrucciones

Instrucción	Formato	Descripción
RESET	1100 0000	Resetea los registros internos al valor por defecto y establece el modo configuración.
READ	0000 0011	Lee el dato de un registro dado.
READ RX Buffer	1001 0nm0	Operación de lectura específica sobre los buffers de entrada (menor overhead que READ).
WRITE	0000 0010	Escribe un dato en un registro dado.
Load TX Buffer	0100 0abc	Operación de escritura específica sobre los buffers de salida (menor overhead que WRITE).
RTS (Message Request-To-Send)	1000 0nnn	Se usa en el comienzo de cada transmisión de dato.
Read Status	1010 0000	Lee el estado del registro status.
RX Status	1011 0000	Indica si se ha cumplido un filtro y el tipo de mensaje recibido.
Bit Modify	0000 0101	Permite modificar bits de registros (aquellos que permitan dicha acción).

Tabla 2.1: Tabla correspondiente al juego de instrucciones de la Interfaz SPI.

A continuación se adjutan los croonogramas correspondientes a las instrucciones SPI. Sin embargo, para su compresión, resulta imprescindible comprender como funciona la comunicación con la Interfaz.

En Atmega, el inicio del ciclo de comunicación la realiza el maestro al poner la señal CS a bajo nivel. Cuando el maestro desea intercambiar datos, genera un pulso de reloj en la señal SCK. Por otra parte, los datos que van del maestro al esclavo se intercambian a través de la señal SI/MOSI (Master Output Slave Input), análogamente, para el envío del esclavo al maestro se hará uso de la señal SO/MISO (Master Input Slave Output). Por último, hay que comentar que en el final de cada transacción, el maestro se debe sincronizar con el esclavo llevando la señal CS a nivel alto.

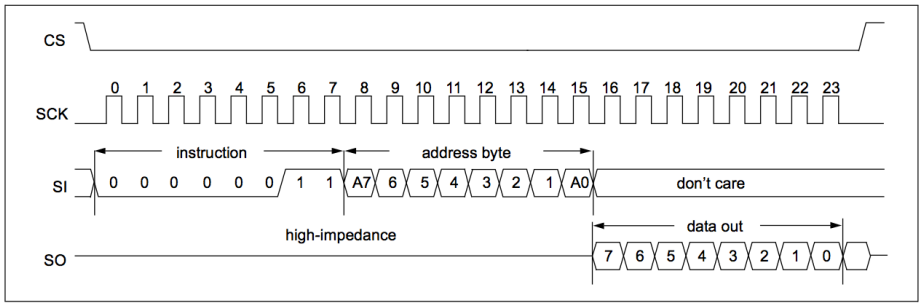
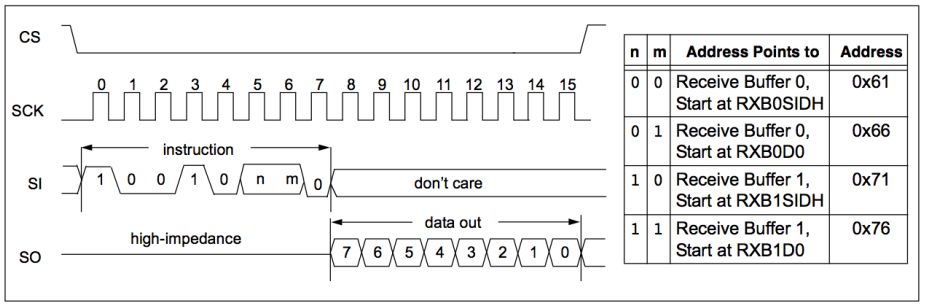


Figura 2.14: Cronograma correspondiente a la operación READ de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*

En el cronograma de la operación READ se puede observar que tras enviar el byte correspondiente al tipo de comando, se deberá indicar la dirección del registro sobre el que hacer la lectura, tras lo cual se obtendrá el byte correspondiente al dato contenido en el registro.



n	m	Address Points to	Address
0	0	Receive Buffer 0, Start at RXB0SIDH	0x61
0	1	Receive Buffer 0, Start at RXB0D0	0x66
1	0	Receive Buffer 1, Start at RXB1SIDH	0x71
1	1	Receive Buffer 1, Start at RXB1D0	0x76

Figura 2.15: Cronograma correspondiente a la operación READ RX BUFFER de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*

Como vemos en el cronograma anterior, los bits n y m se usarán para indicar el buffer del cual se quiere leer, siguiendo la correspondencia que se muestra en la tabla.

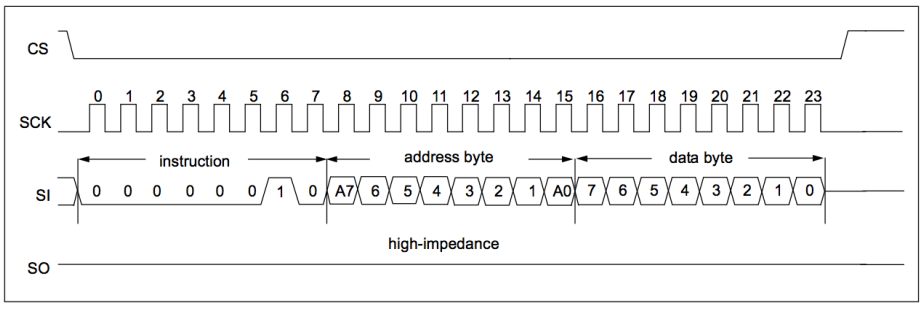
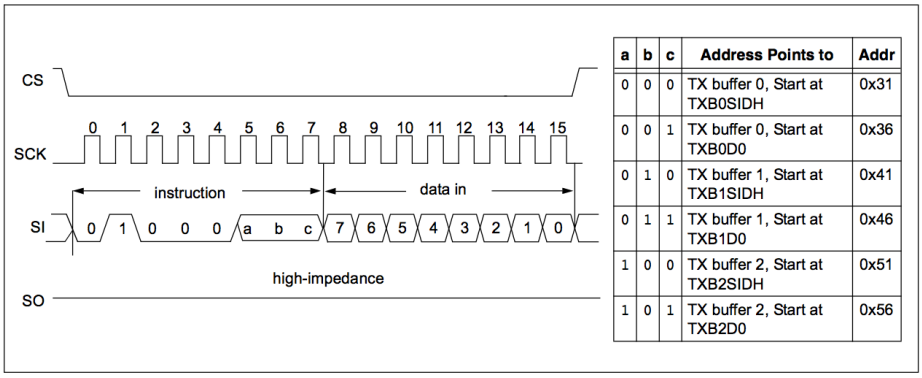


Figura 2.16: Cronograma correspondiente a la operación BYTE WRITE Buffer de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*

La operación WRITE, es análoga a la operación READ con la salvedad de que

en esta ocasión tras enviar el byte de dirección, escribimos el byte correspondiente al contenido del registro.



a	b	c	Address Points to	Addr
0	0	0	TX buffer 0, Start at TXB0SIDH	0x31
0	0	1	TX buffer 0, Start at TXB0D0	0x36
0	1	0	TX buffer 1, Start at TXB1SIDH	0x41
0	1	1	TX buffer 1, Start at TXB1D0	0x46
1	0	0	TX buffer 2, Start at TXB2SIDH	0x51
1	0	1	TX buffer 2, Start at TXB2D0	0x56

Figura 2.17: Cronograma correspondiente a la operación LOAD TX BUFFER de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*

Para llevar a cabo la operación de LOAD TX BUFFER, se deberá indicar en los bits a, b y c el buffer sobre el cual se quiere escribir el dato.

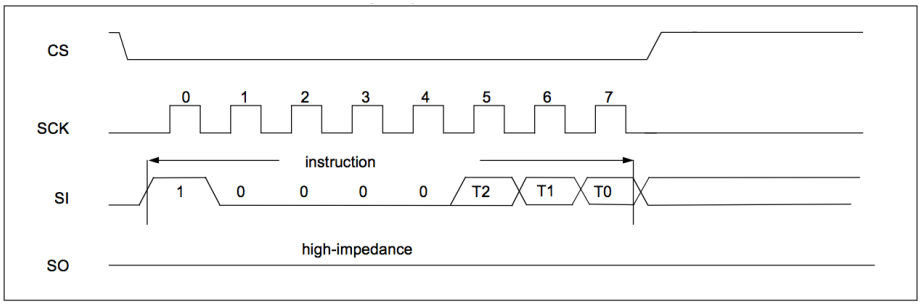


Figura 2.18: Cronograma correspondiente a la operación REQUEST-TO-SEND (RTS) de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*

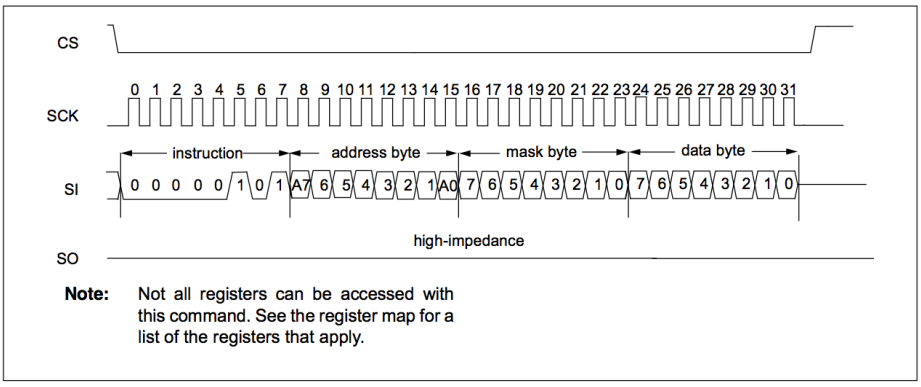


Figura 2.19: Cronograma correspondiente a la operación BIT MODIFY de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*

Dado que en la operación BIT MODIFY está destinada a la modificación de determinados bits de un registro, se hará uso de una máscara que indique los bits a modificar del registro.

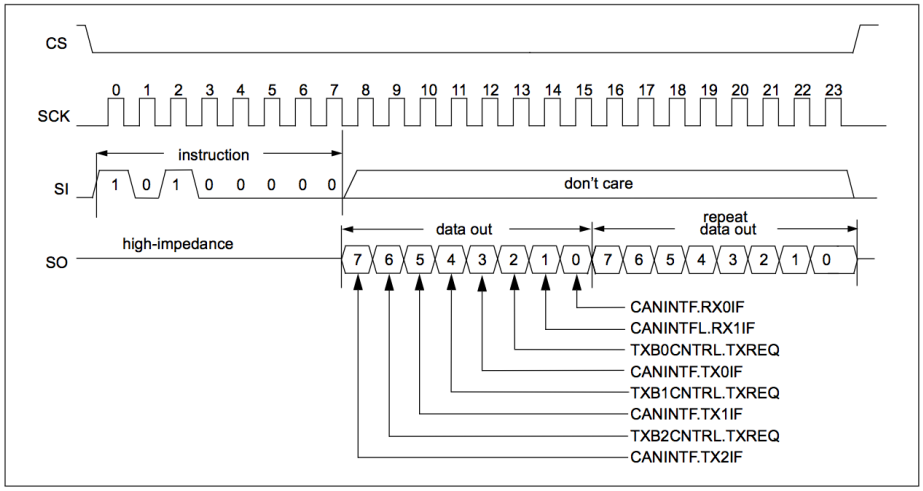
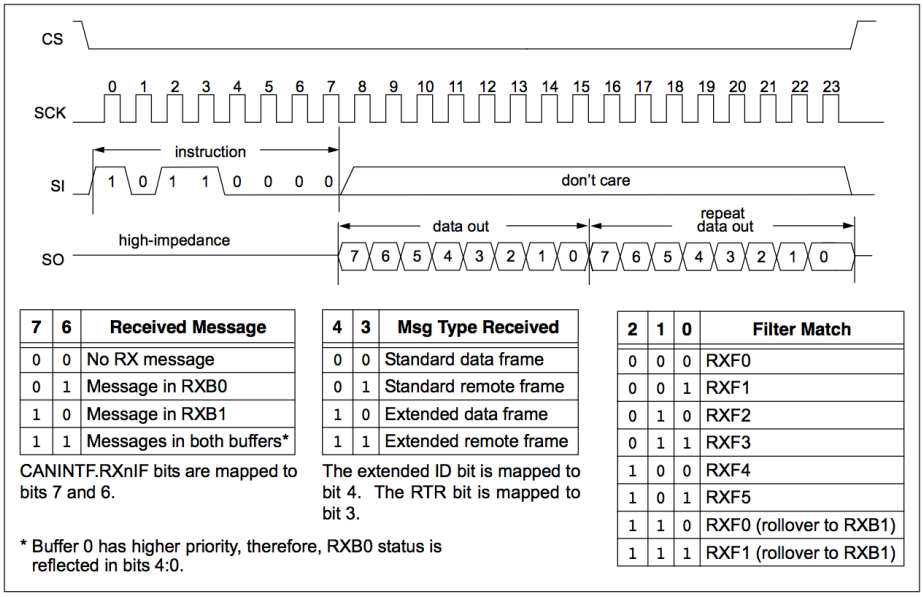


Figura 2.20: Cronograma correspondiente a la operación READ STATUS de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*



7	6	Received Message
0	0	No RX message
0	1	Message in RXB0
1	0	Message in RXB1
1	1	Messages in both buffers*

CANINTF.RXnIF bits are mapped to bits 7 and 6.

4	3	Msg Type Received
0	0	Standard data frame
0	1	Standard remote frame
1	0	Extended data frame
1	1	Extended remote frame

The extended ID bit is mapped to bit 4. The RTR bit is mapped to bit 3.

2	1	0	Filter Match
0	0	0	RXF0
0	0	1	RXF1
0	1	0	RXF2
0	1	1	RXF3
1	0	0	RXF4
1	0	1	RXF5
1	1	0	RXF0 (rollover to RXB1)
1	1	1	RXF1 (rollover to RXB1)

* Buffer 0 has higher priority, therefore, RXB0 status is reflected in bits 4:0.

Figura 2.21: Cronograma correspondiente a la operación RX STATUS de la Interfaz SPI. *Fuente: Microchip MCP2515 Datasheet.*

2.3.4. Banco de Registros

Dada la cantidad de registros que dispone el MCP2515, se ha decidido seleccionar los registros que a nuestro criterio resultan más relevantes.

2.3.4.1. Registros asociados a la trasmisión de tramas

Los registros que se muestran a continuación están íntimamente relacionados con la transmisión de tramas, ya sea actuando como buffers o como registros de configuración. Es importante tener en cuenta que el MCP2515 dispone de tres buffers de transmisión, por lo varios de los registros que se muestran a continuación estarán triplicados.

2.3.4.1.1. TXBnCTRL - Registro de control para el buffer de transmisión n: Se trata de los registros encargados de configurar los pines para la transmisión de datos. En TouCAN se manipula de forma directa el bit TXREQ con el fin de indicar sobre cuál de los tres buffers de transmisión realizamos la petición para el envío.

U-0	R-0	R-0	R-0	R/W-0	U-0	R/W-0	R/W-0
—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0
bit 7							bit 0

Figura 2.22: Registro TXBnCTRL. *Fuente: Microchip MCP2515 Datasheet.*

- bit 7 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 6 - ABTF: Flag para indicar que el mensaje ha sido abortado.
  1 = El mensaje fue abortado.
  0 = Se completó la transmisión del mensaje de forma existosa.
- bit 5 - MLOA: Flag para indicar la pérdida del arbitraje.
  1 = El mensaje perdió el arbitraje mientras era enviado.
  0 = El mensaje no perdió el arbitraje mientras era enviado.
- bit 4 - TXERR: Flag para indicar error durante la transmisión.
  1 = Un error en el bus ocurrió mientras se transmitía el mensaje.
  0 = Sin errores en el bus durante la transmisión del mensaje.
- bit 3 - TXREQ: Bit para indicar que se desear realizar un envío de trama.
  1 = El buffer se encuentra actualmente pendiente de una transmisión(El micrcontrolador puede ponerlo a 1 para indicar que se va a enviar un mensaje - El bit se pone a 0 automáticamente cuando se transmite el mensaje).
  0 = El buffer no se encuentra actualmente pendiente de una transmisión(El microntrolador puede ponerlo a 0 para indicar que se desea abortar el envío de un mensaje).
- bit 2 - Sin implementar: Su lectura proporciona el valor '0'.
- bits 1-0 - TXP: Prioridad del buffer de transmisión.
  11= Prioridad máxima para el mensaje.
  10= Prioridad alta para el mensaje.
  01= Prioridad media-baja para el mensaje.
  00= Prioridad baja para el mensaje.

2.3.4.1.2. TXBnSIDH – Buffer de transmisión n. Parte alta del identificador estándar: Se trata de los registros que actúan de buffer para almacenar la parta alta del identificador estándar. Como veremos en el capítulo de "diseño e implementación", se almacenará la parte alta de la dirección de destino de una trama TouCAN.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

Figura 2.23: Registro TXBnSIDH. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - Bits del identificador estándar(bits $< 10 : 3 >$).

2.3.4.1.3. Buffer de transmisión n. Parte baja del identificador estándar: Se trata de los registros que actúan de buffer para almacenar la parta baja del identificador estándar y por el otro lado hace las veces de registro de configuración ya que el bit EXIDE indica si se transmitirá una trama estándar o extendida.También podemos encontrar aquí, los 2 bits más significativos del identificador extendido(veremos en el capítulo correspondiente,"diseño e implementación", que estos dos bits son desechados en TouCAN).

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

Figura 2.24: Registro TXBnSIDL.

- bits 7-5 - SID: Bits del identificador estándar (bits $< 2 : 0 >$)
- bit 4 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 3 - EXIDE: Indica que se ha activado la opción de trama extendida.
  1 = El mensaje transmitirá un identificador extendido.
  0 = El mensaje transmitirá un identificador estándar.
- bit 2 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 1-0 - EID: Bits del identificadr extendido (bits $< 17 : 16 >$).

2.3.4.1.4. TXBnEID8 - Buffer de transmisión n. Parte alta del identificador extendido: Registros análogos a los TXBnSIDH pero aplicado a las tramas extendidas. Almacenan la parte alta del identificador extendido (salvo los dos bits más significativos $< 17 : 16 >$).

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

Figura 2.25: Registro TXBnEID8. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - Bits del identificador extendido(bits $< 15 : 8 >$).

2.3.4.1.5. TXBnEID0 - Buffer de transmisión n. Parte baja del identificador extendido: Almacenan la parte baja del identificador extendido de la trama a transmitir.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7							bit 0

Figura 2.26: Registro TXBnEID0. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - Bits del identificador extendido(bits $< 7 : 0 >$).

2.3.4.1.6. TXBnDLC - Buffer de transmisión n. Longitud del campo de datos: La función de estos registrso es la de almacenar el valor de la longitud del campo de datos de la trama a transmitir. De forma adicional también indican si la trama es remota o no.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	RTR	—	—	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

Figura 2.27: Registro TXBnDLC. *Fuente: Microchip MCP2515 Datasheet.*

- bit 7 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 6 - RTR: Bit de petición de transmisión remota.
  1 = La trama transmitida será del tipo remot..
  0 = La trama transmitida será del tipo de datos.
- bits 5-4 - Sin implementar: Su lectura proporciona el valor '0'.
- bits 3-0 - DLC: Longitud del campo de datos.
  Establece el número de bytes de datos a transmitir (de 0 a 8 bytes).
  Nota: Al tener 4 bits para codificar la longitud, es posible establecer valores mayores que 8, no obstante sólo 8 bytes serán transmitidos.

2.3.4.1.7. TXBnDm. Buffer de transmisión n. Byte m: Estos registros funcionan como buffers almacenando los bytes de datos de la trama a transmitir.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
TXBnDm7	TXBnDm6	TXBnDm5	TXBnDm4	TXBnDm3	TXBnDm2	TXBnDm1	TXBnDm0
bit 7							bit 0

Figura 2.28: Registro TXBnDm. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - TXBnDM7:TXBnDM0: Byte m del campo de bits del buffer de transmisión n

2.3.4.2. Registros asociados a la recepción de tramas

Los registros que se muestran a continuación están estrechamente relacionados con la recepción de mensajes ya sea actuando como registros de configuración o buffers de recepción de tramas. Como es lógico, se puede observar una cierta correspondencia con los registros asociados a la transmisión de tramas. Es importante tener en cuenta que el MCP2515 dispone de dos buffers de transmisión, por lo varios de los registros que se muestran a continuación estarán duplicados.

2.3.4.2.1. RXB0CTRL - Control del buffer de recepción 0: La finalidad de este registro es la de configurar el modo en el que el buffer 0 de recepción del MCP2515 recibe las tramas.

U-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	BUKT	BUKT1	FILHIT0
bit 7							bit 0

Figura 2.29: Registro RXB0CTRL. *Fuente: Microchip MCP2515 Datasheet.*

- bit 7 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 6-5 - RXM: Modo de operación del buffer de recepción.
  11= Desactiva las máscaras y los filtros, recibe cualquier mensaje.
  10= Recibe únicamente mensajes extendidos válidos que cumplan el criterio de los filtros.
  01= Recibe únicamente mensajes estándars válidos que cumplan el criterio de los filtros.
  00= Recibe únicamente mensajes extendidos o estándar válidos que cumplan el criterio de los filtros.
- bit 4 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 3 - RXRTR: Bit que indica si se ha recibido una trama remota o no.
  1 = Trama remota recibida.
  0 = Trama remota no recibida.
- bit 2 - BUKT: Bit de establecimiento de rollover.
  1 = El mensaje destinado al buffer RXB0 será pasado a RXB1 s RXB0 se encuentra lleno.
  0 = Rollover desactivado.
- bit 1 - BUKT1: Copia de sólo lectura de BUKT (usado internamente por el MCP2515).
- bit 0 - FILHIT: Hit-bit de filtro - Indica cuál de los filtros aceptó la recepción del mensaje.
  1 = Filtro de aceptación 1 (RXF1).
  0 = Filtro de aceptación 0 (RXF0).
  Nota: Si ocurre un rollover de RXB0 a RXB1, FILHIT reflejará cuál de los filtros aceptó el mensaje que produjo dicho rollover.

2.3.4.2.2. RXB1CTRL - Control del buffer de recepción 1: La finalidad de este registro es la de configurar el modo en el que el buffer 1 de recepción del MCP2515 recibe las tramas.

U-0	R/W-0	R/W-0	U-0	R-0	R-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	FILHIT2	FILHIT1	FILHIT0
bit 7							bit 0

Figura 2.30: Registro RXB1CTRL. *Fuente: Microchip MCP2515 Datasheet.*

- bit 7 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 6-5 - RXM: Modo de operación del buffer de recepción
  11= Desactiva las máscaras y los filtros, recibe cualquier mensaje.
  10= Recibe únicamente mensajes extendidos válidos que cumplan el criterio de los filtros.
  01= Recibe únicamente mensajes estándars válidos que cumplan el criterio de los filtros.
  00= Recibe únicamente mensajes extendidos o estándar válidos que cumplan el criterio de los filtros.
- bit 4 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 3 - RXRTR: Bit que indica si se ha recibido una trama remota o no.
  1 = Trama remota recibida.
  0 = Trama remota no recibida.
- bits 2-0 - FILHIT: Hit-bit de filtro - Indica cuál de los filtros aceptó la recepción del mensaje.
  101 = Filtro de aceptación 5 (RXF5).
  100 = Filtro de aceptación 4 (RXF4).
  011 = Filtro de aceptación 3 (RXF3).
  010 = Filtro de aceptación 2 (RXF2).
  001 = Filtro de aceptación 1 (RXF1) (Sólo si el bit BUKT está a 1 en RXB0CTRL).
  000 = Filtro de aceptación 0 (RXF0) (Sólo si el bit BUKT está a 1 en RXB0CTRL).

2.3.4.2.3. RXBnSIDH - Buffer de recepción n. Parte alta del identificador estándar: Estos registros actúan como buffers de recepción de la parte alta del identificador estándar de la trama recibida.

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

Figura 2.31: Registro RXBnSIDH. *Fuente: Microchip MCP2515 Datasheet.*

- SID: Bits del identificador estándar (bits $< 10 : 3 >$).
  Estos bits contienen los ocho bits más significativos del identificador estándar de la trama recibida.

2.3.4.2.4. RXBnSIDL - Buffer de recepción n. Parte baja del identificador estándar: Estos registros actúan como buffers de recepción de la parte baja del identificador estándar de la trama recibida. De forma adicional también indica si la trama recibida es una estándar o extendida, incluyendo también los dos bits más significativos del identificador de trama extendida.

R-x	R-x	R-x	R-x	R-x	U-0	R-x	R-x
SID2	SID1	SID0	SRR	IDE	—	EID17	EID16
bit 7							bit 0

Figura 2.32: Registro RXBnSIDL. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-5 - SID: Bits del identificador estándar (bits $< 2 : 0 >$).
  Estos bits contienen los tres bits menos significativos del identificador estándar de la trama recibida.
- bit 4 - SRR: Bit de trama remota recibida(válido sólo si el bit IDE está a '0').
  1 = Trama remota estándar recibida.
  0 = Trama de datos estándar recibida.
  Aporta una información similar al bit RXRTR de RXB0CTRL o RXB1CTRL.
- bit 3 - IDE: Flag que indica si se ha recibido una trama extendida o no.
  Este bit indica si se ha recibido una trama extendida o una estándar.
  1 = El mensaje recibido era una trama extendida.
  0 = El mensaje recibido era una trama estándar.
- bit 2 - Sin implementar: Su lectura proporciona el valor '0'.
- bits 1-0 - EID: Bits del identificador extendido (bits $< 17 : 16 >$)
  Contienen los dos bits más significativos del identificador de trama extendido.

2.3.4.2.5. RXBnEID8 - Buffer de recepción. Parte alta del identificador extendido: Registros análogos a los RXBnSIDH pero para las tramas extendidas. Almacenan la parte alta del identificador extendido de la trama recibida.

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

Figura 2.33: Registro RXBnEID8. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - EID: Bits del identificador extendido (bits $< 15 : 8 >$).
  Almacenan los bits del 15 al 8 del identificador extendido del mensaje recibido.

2.3.4.2.6. RXBnEID0 - Buffer de recepción. Parte baja del identificador extendido: Registros análogos a los RXBnSIDL pero para las tramas extendidas. Almacenan la parte baja del identificador extendido de la trama recibida.

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7							bit 0

Figura 2.34: Registro RXBnEID0. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - EID: Bits del identificador extendido (bits $< 7 : 0 >$).
  Almacenan los 8 bits menos significativos del identificador extendido del mensaje recibido.

2.3.4.2.7. RXBnDLC - Buffer de recepción. Longitud del campo de datos: Este registro codifica el la longitud en bytes del campo de datos.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	RTR	RB1	RB0	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

Figura 2.35: Registro RXBnDLC. *Fuente: Microchip MCP2515 Datasheet.*

- bit 7 - Sin implementar: Su lectura proporciona el valor '0'.
- bit 6 - RTR: Bit de petición de transmisión remota.
  1 = La trama transmitida será del tipo remot..
  0 = La trama transmitida será del tipo de datos.
- bits 5 - Bit reservado 1.
- bits 4 - Bit reservado 0.
- bits 3-0 - DLC: Código para la longitud del campo de datos.
  Indica el número de bytes de datos a recibidos (de 0 a 8 bytes).

2.3.4.2.8. RXBnDM - Buffer de recepción n. Byte m: Estos registros funcionan como buffers almacenando los bytes de datos de la trama recibida.

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
RBnDm7	RBnDm6	RBnDm5	RBnDm4	RBnDm3	RBnDm2	RBnDm1	RBnDm0
bit 7							bit 0

Figura 2.36: Registro RXBnDM. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - RBnDM7:RBnDM0: Byte m del campo de bits del buffer de recepción n

2.3.4.3. Registros asociados a filtros

Los registros que se muestran a continuación están íntimamente relacionados con los filtros para los identificadores, necesarios para la correcta aceptación de los mensajes. Es importante tener en cuenta que el MCP2515 dispone de 6 buffers de filtros (estándares y extendidos).

2.3.4.3.1. RXFnSIDH - Filtro n. Parte alta del identificador estándar: Estamos ante los registros que actuarán como filtro para la parte alta del identificador estándar. En el capítulo "diseño e implementación."plicaremos más detalladamente por qué no se utilizan en TouCAN.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

Figura 2.37: Registro RXFnSIDH. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - SID: Bits de filtro para identificador estándar (bits $< 10 : 3 >$).
  Estos bits se aplican como filtro al rango de bits $< 10 : 3 >$ de la porción del identificador estándar del mensaje recibido.

2.3.4.3.2. RXFnSIDL - Filtro n. Parte baja del identificador estándar: Estamos ante los registros que actuarán como filtro para la parte baja del identificador estándar. En el capítulo "diseño e implementación."plicaremos más detalladamente por qué no se utilizan en TouCAN.

R/W-x	R/W-x	R/W-x	U-0	R/W-x	U-0	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

Figura 2.38: Registro RXFnSIDL. *Fuente: Microchip MCP2515 Datasheet.*

- bit 7-5 - SID: Filtro para identificador estándar(bits $< 2 : 0 >$)
  Estos bits se aplican como filtro al rango de bits $< 2 : 0 >$ de la porción del identificador estándar del mensaje recibido.

- bit 4 - Sin implementar: Su lectura proporciona el valor '0'.

- bit 3 - EXIDE: Bit de indicio de identificador extendido.
  1 = El filtro se aplica únicamente a tramas extendidas.
  0 = El filtro se aplica únicamente a tramas estándard.

- bit 2 - Sin implementar: Su lectura proporciona el valor '0'.

- bit 1-0 - EID: Bits para el filtrado del identificador extendido(bits $< 17 : 16 >$)
  Estos bits se aplican como filtro al rango de bits $< 17 : 16 >$ de la porción del identificador extendido del mensaje recibido.

2.3.4.3.3. RXFnEID8 - Filtro n. Parte alta del identificador extendido: Registros análogos a los RXFnSIDH pero destinado al filtrado de tramas extendidas. Almacenan la parte alta del filtro del identificador extendido.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

Figura 2.39: Registro RXFnEID8. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - EID: Bits de filtro para el identificador extendido(bits $< 15 : 8 >$) Estos bits se aplican como filtro al rango de bits $< 15 : 8 >$ de la porción del identificador extendido del mensaje recibido.

2.3.4.3.4. RXFnEID0 - Filtro n. Parte baja del identificador extendido: Estos registros almacenan la parte baja del filtro destinado a la parte baja del identificador extendido de la trama recibida.

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7							bit 0

Figura 2.40: Registro RXFnEID0. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - EID: Bits de filtro para el identificador extendido(bits $< 7 : 0 >$) Estos bits se aplican como filtro para la parte baja del identificador extendido del mensaje recibido.

2.3.4.4. Registros asociados a máscaras

Los registros que aquí se presentan están estrechamente relacionados con las máscaras del protocolo CAN. Es importante tener en cuenta que el MCP2515 dispone de dos máscaras (estándar y extendidas).

2.3.4.4.1. RXMnSIDH - Máscara n. Parte alta del identificador estándar: Almacena la máscara necesaria para la comprobación de filtros de la parte alta del identificador estándar. En el capítulo "diseño e implementación.[ex]plicaremos más detalladamente por qué no se utilizan en TouCAN.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

Figura 2.41: Registro RXMnSIDH. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - SID: Máscara para los bits de la parte alta del identificador estándar (bits $< 10 : 3 >$).
  Estos bits se aplican como máscara al rango de bits $< 10 : 3 >$ de la porción del identificador estándar del mensaje recibido.

2.3.4.4.2. RXMnSIDL - Máscara n. Parte baja del identificador estándar Estos registros almacenan los bits necesarios para la máscara de la parte baja del identificador estándar a ser filtrado. En el capítulo "diseño e implementación.[ex]plicaremos más detalladamente por qué no se utilizan en TouCAN.

R/W-0	R/W-0	R/W-0	U-0	U-0	U-0	R/W-0	R/W-0
SID2	SID1	SID0	—	—	—	EID17	EID16
bit 7							bit 0

Figura 2.42: Registro RXMnSIDL. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-5 - SID: Máscara para el identificador estándar(bits $< 2 : 0 >$).
  Estos bits se aplican como máscara al rango de bits $< 2 : 0 >$ de la porción del identificador estándar del mensaje recibido.
- bits 4-2 - Sin implementar: Su lectura proporciona el valor '0'.
- bits 1-0 - EID: Máscara para el identificador estándar (bits $< 17 : 16 >$).
  Estos bits se aplican como máscara al rango de bits $< 17 : 6 >$ de la porción del identificador extendido del mensaje recibido.

2.3.4.4.3. RXMnEID8 - Máscara n. Parte alta del identificador extendido: Registros análogos a los RXMnSIDH pero destinado a las máscaras extendidas. Almacenan la parte alta de la máscara extendida del rango.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

Figura 2.43: Registro RXMnEID8. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - EID: Bits de máscara para el identificador extendido (bits $< 15 : 8 >$).
  Estos bits se aplican como máscara al rango de bits $< 15 : 8 >$ de la porción del identificador extendido del mensaje recibido.

2.3.4.4.4. RXMnEID0 - Máscara n. Parte baja del identificador extendido: Estos registros almacenan los bits necesarios para la máscara extendida de la parte baja identificador extendido.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7							bit 0

Figura 2.44: Registro RXMnEID0. *Fuente: Microchip MCP2515 Datasheet.*

- bits 7-0 - EID: Bits de máscara para el identificador extendido (bits $< 7 : 0 >$).
  Estos bits se aplican como máscara al rango de bits $< 7 : 0 >$ (parte baja) de la porción del identificador extendido del mensaje recibido.

Capítulo 3

Competencias

En este capítulo describiremos la manera en la que hemos cubierto las competencias exigidas al Trabajo de Fin de Grado.

3.1. CII01

Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente.

Esta competencia ha sido desarrollada en los capítulos de ̈Introducción", .ºnálisis del problemaz "Diseño e implementación". En dichos capítulos mostramos las causas y los hechos derivados de las decisiones de diseño tomadas.

3.2. CII02

Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social.
En el capítulo "Metodología y planificaciónrelatamos el sistema de organización usado para alcanzar los objetivos propuestos. La valoración del impacto social y económico de este TFG queda reflejada por su parte en el capítulo .ºportaciones".
Capacidad para elaborar el pliego de condiciones técnicas de una instalación informática que cumpla los estándares y normativas vigentes.
En el capítulo de "Pliego de condiciones"queda cubierta esta competencia especificando las condiciones asociadas a este Trabajo de Fin de Grado.

3.3. CII18

Conocimiento de la normativa y la regulación de la informática en los ámbitos nacional, europeo e internacional.

El cumplimiento de esta competencia queda descrita en el capítulo de "Normativa y Legislación".

Capítulo 4

Aportaciones

Siendo sinceros, en el momento de realización de este TFG, Arduino ya poseía una comunidad bastante fuerte en torno a él y por el otro lado, CAN era ya un protocolo asentado. No obstante, no existía aún ninguna librería libre y sencilla que aprovechase las capacidades de la dupla Arduino+CAN. Si bien es cierto que existen herramientas con propósitos similares como CANOpen, por lo general su uso queda restringido a los miembros de determinados consorcios. Por su parte, muchas de las herramientas libres podían pecar en simpleza en cuanto a funcionalidades y complejidad en cuanto al manejo.

Es por ello que creemos, en nuestra humilde opinión, que hemos aportado una herramienta de fácil manejo y gran potencial a una comunidad entusiasta y creativa. El estado del arte no podía ser más interesante, Arduino se utiliza en una gran variedad de proyectos open-source y open-hardware en campos tan variados como robótica, domótica, videojuegos y otros entretenimientos electrónicos, etc. por lo que estamos seguros de que nuestra herramienta será de gran ayuda y propiciará futuros proyectos en los que se desee integrar redes de microcontroladores de forma barata, sencilla, eficiente y libre.

Capítulo 5

Normativa y Legislación

5.1. Normativa

5.1.1. Ley de Protección de Datos

La Ley orgánica de Protección de Datos Personales tiene por objetivo el garantizar y proteger, en lo que concierne al tratamiento de los datos personales, las libertades públicas y los derechos fundamentales de las personas físicas, y especialmente de su honor e intimidad personal y familiar. Dicha ley será aplicada a los datos de carácter personal registrados en soporte físico que los haga susceptibles de tratamiento, y a toda modalidad de uso posterior de estos datos por los sectores público y privado.

Si la LOPD no estuviera en vigor, nuestros datos estarían en manos de la especulación. Un ejemplo claro lo vemos cuando nos registramos a una página web, la cual normalmente suele pedir tu nombre, tus apellidos, dirección, correo electrónico, gustos personales, etc. Sin duda estos datos deberían estar protegidos, sin embargo hoy en día es muy común por ejemplo que tu correo electrónico se proporcione sin tu consentimiento a otra empresa, lo cual le da vía libre para enviar correo basura de forma indiscriminada, es por ello que es preciso denunciar este tipo de acciones que van en contra de la LOPD.

5.1.2. Código tipo

Son códigos deontológico o de buena conducta o práctica profesionales. Están regulados en la Ley Orgánica de Protección de Datos de Carácter Personal, que es la que establece en nuestro país los cimientos básicos de los aspectos legales en cuanto a protección de datos de carácter personal se refiere. Gracias al código tipo se logra que el cliente o usuario se quede más tranquilo ya que los subscriptores a un código tipo ponen mayor interés en la protección de los datos. Por otra parte la Agencia de Protección de Datos no podrá sancionarnos por alguna práctica especificada en

el código siempre y cuando dicho código estuviese inscrito en el Registro General de Protección de Datos.

5.1.3. Inscripción de un programa informático

Los requisitos específicos para la inscripción de un programa informático son:

- La totalidad del código fuente, en CD-ROM o en soporte papel, debidamente encuadernada y paginada.
- En los programas de ordenador editados ha de presentarse un resumen por escrito de al menos 20 folios del código fuente, siempre y cuando reproduzcan elementos esenciales del mismo. Deberá ir encuadernado y con portada en que figure título y autor de la obra.
- Una memoria en soporte papel, debidamente encuadernada y paginada con el resumen de la aplicación, el lenguaje de programación, el entorno operativo, el listado de nombres de los ficheros que contiene y el diagrama de flujo.
- Puede aportarse el ejecutable del programa, de forma optativa.
- Los escritos y solicitudes que se dirijan a cualquiera de las oficinas del Registro podrán presentarse en las formas y ante los órganos que prevé la Ley 30/1992, de Régimen Jurídico de las Administraciones Públicas y del Procedimiento Administrativo Común. También podrán presentarse en cualquiera de los registros a que se refiere el presente Reglamento, tanto Central como territoriales, los cuales los remitirán, el día siguiente al de su presentación, al Registro indicado en la solicitud.

5.2. Licencias

5.2.1. GNU GPL

Conocida como licencia pública general (GNU Generical Public License), se trata de una licencia software elaborada por la fundación del software libre en el año 1989. Esta licencia nace de la necesidad de proteger la distribución, modificación y uso de sofware evitando la apropiación indebida del código que pueda dar lugar a las restricciones y libertades de los usuarios. La licencia GPL permite que el software se distribuya y modifique libremente siempre y cuando no se altere la licencia original. Por otra parte, cuando surja la necesidad de incluir código sujeto a otras licencias libres, el resultado final deberá estar bajo la licencia GPL.

5.2.2. LGPL

Licencia similar a la GPL que permite que una biblioteca determinada pueda ser usada tanto por programas libres como por programas no libres. Con lo cual, se da la posibilidad de poder distribuir programas bajo cualquier licencia con la salvedad de aquellos trabajos derivados, cuyos términos de modificación los establece la licencia por la cual se rigen.

5.2.3. Creative Commons

Las licencias Creative commons ó CC tienen su origen en la licencia GNU GPL, cuyo objetivo es el de ofrecer al autor de una obra un mecanismo sencillo y eficáz a la hora de establecer una serie de condiciones asociadas a la obra. Las condiciones que se pueden imponer son las siguientes:

- **Reconocimiento (Attribution):** *En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.*

Figura 5.1: Reconocimiento.

- **No Comercial (Non commercial):** *La explotación de la obra queda limitada a usos no comerciales.*ó

Figura 5.2: No Comercial.

- **Sin obras derivadas (No Derivate Works):** *La autorización para explotar la obra no incluye la transformación para crear una obra derivada.*

Figura 5.3: Sin obras derivadas.

- **Compartir Igual (Share alike):** *La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.*

Figura 5.4: Compartir Igual.

Capítulo 6

Pliego de condiciones

6.1. Objeto de este pliego

El presente Pliego de Condiciones Técnicas, tiene por objeto la definición de las condiciones técnicas y económicas asociadas en la ejecución y utilización del software del lado microcontrolador de redes de microcontroladores distribuidos TouCAN.

6.2. Pliego de Condiciones Generales

Las características principales del presente proyecto se corresponde con las siguientes:

- Capacidad para realizar peticiones síncronas a un nodo de la red.
- Capacidad para realizar peticiones asíncronas a un nodo de la red.
- Almacenamiento de las respuestas en buffers internos de la librería.
- Atender a peticiones de un equipo supervisor actuando en consecuencia.

6.3. Pliegos de especificaciones técnicas

6.3.1. Especificaciones de materiales, equipos y sofware

Las especificaciones de todos los materiales de los cuales se compone el proyecto se describe ampliamente en el apartado de requisitos, en la cual se analiza de forma detallada tanto los requisitos a nivel de hardware como a nivel de software.

6.3.2. Especificaciones de ejecución

El proceso llevado a cabo en la elaboración del proyecto se puede encontrar especificado dentro de capítulo diseño e implementación.

6.4. Pliego de Clausulas Administrativas Particulares

El presupuesto asociado al proyecto asciende a un total de 16.980 euros.

Recurso	Coste
Programador (120 horas)	40 euros.
Analista Funcional (60 horas)	50 euros.
Jefe de Proyecto (150 horas)	60 euros.
Hardware	180 euros.
Total	16980 euros (IGIG incl.).

Tabla 6.1: Tabla correspondiente al presupuesto asociado al proyecto.

6.5. Licencia de Uso

Al adquirir el presente proyecto, el comprador no adquiere ningún poder ni titularidad sobre el software que contiene. Sin embargo podrá modificarlo y distribuirlo si así lo desea, tal y como establece la licencia pública general GNU versión 3.

Capítulo 7

Metodología y Plan de Trabajo

En este capítulo relataremos las distintas etapas que compusieron la elaboración de este trabajo, así como las incidencias que tuvimos que obligaron a adaptar la planificación inicial.

Como metodología de desarrollo se utilizó las técnicas de desarrollo software propias del Proceso Unificado (Unified Process)[5]. El Proceso Unificado proporciona un marco de desarrollo genérico adaptable a proyectos específicos cuyas características son las siguientes:

1. **Iterativo e Incremental:** El Proceso Unificado de compone por cuatros fases distintas, Inicio, Elaboración, Construcción y Transición. En cada una de las fases se hará un determinado número de iteraciones cuyo objetivo será la mejora e inclusión de mejoras de funcionalidades dando lugar a un incremento del proyecto en desarrollo.

2. **Dirigido por los casos de uso:** Por medio de los casos de uso se especificará en las iteraciones los requisitos funcionales y los contenidos.

3. **Centrado en la arquitectura:** En el desarrollo del software no se usará un único modelo que describa todo el sistema. Se optará por un enfoque con múltiples modelos y vistas que describan la arquitectura de software del sistema.

A continuación se puede observar de forma gráfica la evolución típica de un proyecto que sigue el Proceso Unificado.

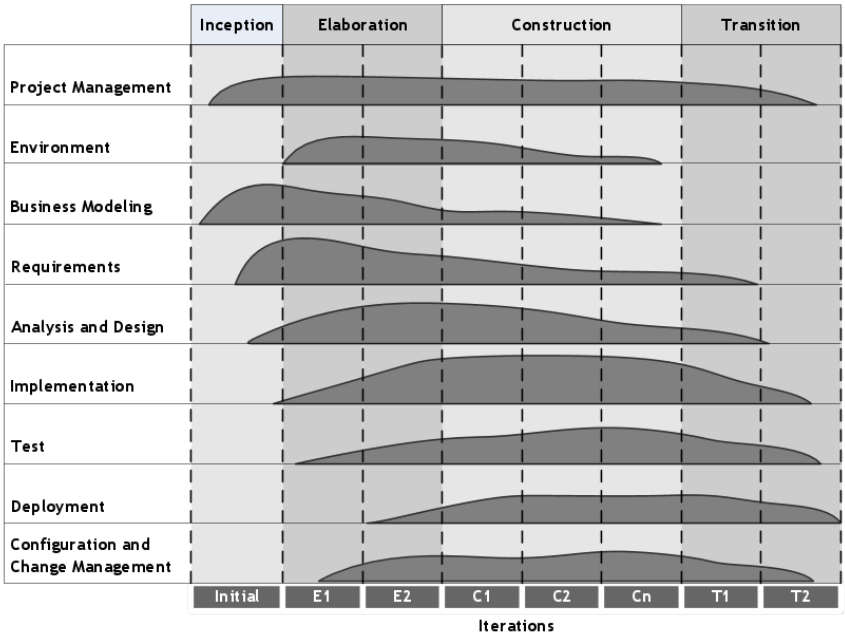


Figura 7.1: Evolución de las diferentes etapas de un proyecto desarrollado mediante el Proceso Unificado. *Fuente: http://es.wikipedia.org/*

En relación al Plan de Trabajo a llevar a cabo, las etapas a cubrir durante el desarrollo del TFG propuesto fueron las siguientes:

1. **Familiarización Arduino y Bus CAN**. Estudio, familiarización y análisis de la arquitectura de desarrollo para sistemas empotrados basada en microcontroladores Arduino. Estudio, familiarización y análisis del bus de comunicaciones CAN y su integración con la mencionada arquitectura de sistemas empotrados.

2. **Diseño y desarrollo en el lado microcontrolador**. Análisis, diseño, desarrollo e implementación de infraestructura software (bibliotecas) y hardware para la integración de sistemas de propósito general utilizando bus CAN.

3. **Diseño y desarrollo de prototipo para el lado microcontrolador**. Desarrollo del prototipo demostrador final del sistema, en el que inicialmente se integren sólo sistemas de propósito general. Prueba del prototipo.

4. **Integración prototipos lado microcontrolador/lado supervisor.**

5. **Pruebas.**

6. **Documentación y Defensa**. Elaboración del Documento de Trabajo de Fin de Grado a partir de toda la documentación y datos obtenidos durante la realización de las diferentes etapas en las que se ha organizado.

La planificación temporal del desarrollo del TFG en base a las tareas previstas en cada etapa que se llevó a cabo fue la siguiente:

Etapas	Dedicación Estimada
Etapa 1: Familiarización Arduino y Bus CAN	25 horas
Etapa 2: Diseño y desarrollo en el lado microcontrolador/supervisor	100 horas
Etapa 3: Diseño y desarrollo de prototipo para el lado microcontrolador/supervisor	75 horas
Etapa 4: Integración prototipos lado microcontrolador/lado supervisor	50 horas
Etapa 5: Pruebas	30 horas
Etapa 6: Documentación y Defensa	50 horas
Dedicación Estimada Total	330 horas

Tabla 7.1: Tabla correspondiente al a la dedicación estimada en cada etapa.

Capítulo 8

Requisitos

A continuación se describirán los requisitos tanto a nivel de hardware como a nivel de software necesarios para llevar a cabo el desarrollo del proyecto.

8.1. Hardware

A nivel de hardware las necesidades fueron las siguientes:

1. **Arduino Uno rev3**. 4 unidades.
2. **CAN-BUS shield**. 4 unidades.
3. **Protoboard**.
4. **Resistencias**. 2 unidades de 120 ohm.
5. **Cables**. Par de cobre trenzado.
6. **Cables USB de A a B**. 4 Unidades
7. **PC**. En concreto se ha utilizado un PC con CPU Intel Core 2 Duo a 2Ghz, Memoria ram de 2GB y una gráfica GeForce 9400M.

8.2. Sofware

A nivel de software fue necesario:

1. **Distribución GNU/Linux** En concreto se utilizó Ubuntu 12.04 de 32 bits.
2. **Sofware de desarrollo** TextMate, GNU g++, KDevelop y Arduino IDE(processing).
3. **Sofware documental** LaTexT, TextWorks, yEd.

Capítulo 9

Diseño e Implementación

En este capítulo explicaremos las distintas funciones que componen la librería y explicaremos la naturaleza de su diseño, haciendo un especial hincapié sobre el lado del microcontrolador pero sin dejar de lado el lado supervisor.

Como hemos explicado con anterioridad, al haber hecho uso de una librería previa que soportase CAN, el trabajo se agilizó en varios sentidos. Basándonos en la librería ArCAN[12] para el lado microcontrolador, creamos una nueva librería que sostuviese nuestro protocolo y añadiese más funcionalidades.

En líneas generales tratamos de crear una librería lo más amigable posible para el usuario, abstrayéndolo de toda la complejidad del protocolo, por lo que únicamente tendría que rellenar una serie de callbacks para procesar sus datos y compilar. Por ello la única dificultad de cara al usuario es conocer la interfaz de los métodos y el significado de cada parámetro.

Por otro lado, nos propusimos también hacer que el código fuera ampliable de forma sencilla. Tratamos de crear una librería flexible en la que si no existiese alguna funcionalidad necesaria para un usuario en un determinado momento, ésta fuera fácil de implementar.

Nuestra topología[7], cómo no, tenía que estar basada en el tipo bus. En este nuevo protocolo definimos los siguientes elementos:

- **Nodo (nodo común):** Son los nodos pasivos de la red. En un estado de bus ocioso se mantienen a la escucha es decir, no inician nunca transmisiones por iniciativa propia, sólo atienden a las peticiones de los maestros.
- **Maestro (nodo maestro):** Es un nodo especial más potente que un nodo común en el sentido de que posee iniciativa propia para iniciar peticiones o conectarse a un supervisor.
- **Supervisor:** No se trata de un nodo de la red propiamente dicho puesto que no

se conecta nunca a la red directamente sino a través de alguno de los maestros (no posee un identicador propio en la red). Es el elemento más potente de los aquí nombrados puesto que es capaz de realizar diversas tareas como la monitorización del bus, iniciar peticiones mediante alguno de los maestros, establecer velocidades de transmisión, etc. Es por ello que no forma parte del lado microcontrolador, sino que será explicado en más profundidad en la memoria correspondiente.

Veamos la concepción esquemática de los elementos de la red:

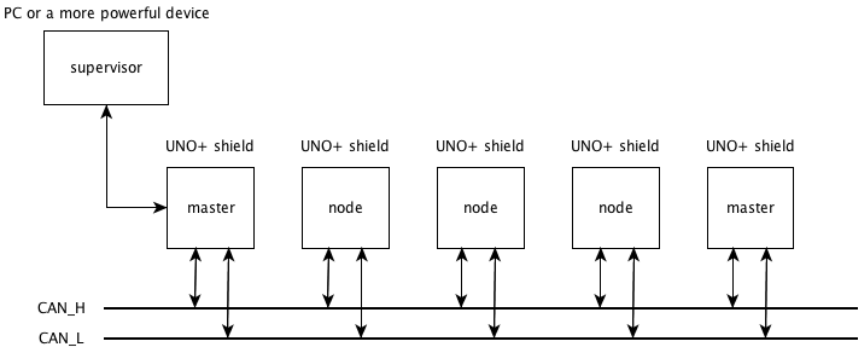


Figura 9.1: Esquema de la red TouCAN

9.1. Lado microcontrolador

Apartando la vista del supervisor de forma momentánea, nos encontramos a los maestros y a los nodos. Puesto que nuestro protocolo podía ser usado como red de comunicación en aparatos como pequeños robots móviles decidimos implementar dos tipos de comunicación:

- **Síncrona:** Este tipo de comunicación garantiza la respuesta por cada petición, es por ello que es la más costosa de tiempo debido al overhead añadido por el procesamiento de datos en ambos extremos de la comunicación.

- **Asíncrona:** Esta comunicación no garantiza la llegada de todas los mensajes. El maestro, al lanzar una única petición de este tipo, esperará recibir una gran cantidad de mensajes según un reloj establecido por sí mismo en el nodo mientras no lance un mensaje de parada.

Asimismo, establecimos que cada nodo en la red debería poseer un identificador.

Una vez tuvimos claro los elementos y mensajes en la red, tuvimos que ver cómo podíamos aprovechar el protocolo CAN para nuestros propósitos. El problema del identificador era fácil de resolver puesto que si bien CAN es un protocolo orientado a mensaje, podíamos aprovechar los filtros y las máscaras para establecer un protocolo orientado a camino, por lo que no desaprovecharíamos ninguno de los

bytes de datos con este fin como sí ocurría en ArCAN.

Como hemos visto en anteriores capítulos, CAN admite tanto paquetes estándar como extendidos, diferenciándose entre estos la longitud del campo ID. Vimos que podíamos hacer uso de los **mensajes extendidos** puesto que la longitud del campo ID poseía el número suficiente de bits para indicar una dirección de **origen** y una dirección de **destino**. Así pues, dividimos la ID extendida de CAN en dos partes: una parte alta para el origen del mensaje y una parte baja para indicar el destino y el tipo de mensaje. Por ello, cada nodo haría uso únicamente de los filtros y las máscaras extendidas puesto que éstas en conjunto a la parte extendida de la ID CAN, filtrarían los mensajes dirigidos a ellos.

En resumen, tendríamos:

- **11 bits de la ID estándar:** Para indicar quién originó el mensaje.
- **18 bits de la ID extendida:** Bits del EID0 a EID10 a quien envía. Bits del EID11 a EID15 tipo de trama. EID16 a EID17 inutilizado.

En total, con 11 bits de identificación tendríamos para 2048 nodos en la red (tanto maestros como nodos comunes), un número más que aceptable y con 5 bits de tipo de trama tendríamos 32 tipos de trama posible, de las cuales usamos 6, por lo que dejamos un amplio margen para la ampliación del protocolo en un futuro.

9.1.1. Tipos de trama TouCAN

En TouCAN podemos distinguir dos grupos de tramas en función al tipo de comunicación, ya sea síncrona o asíncrona:

Si atendemos al tipo síncrono vemos que tenemos:

- **Request (petición):** La codificación en el protocolo sería 00000. La función de esta trama es la de petición de datos a otro nodo. Únicamente puede ser llamada por un maestro o un supervisor. Garantiza la respuesta o en su defecto, informa la pérdida del paquete para el reintento. Se le ofrece libertad al usuario para que utilice los datos de este mensaje para implementar un protocolo sobre TouCAN.
- **Answer (respuesta):** La codificación en el protocolo sería 00001. Este paquete es emitido tanto por los nodos como por los maestro con el fin de responder a un Request.

Si se tratasen de comunicaciones asíncronas observamos:

- **Start (comienzo):** La codificación en el protocolo sería 00011 (macro STRT). Es enviada por el maestro con el fin de indicarle al destinatario que desea que se le envíe un gran número de datos de forma periódica. La frecuencia de envío se indica en el primer byte de datos del mensaje, indicándose en milisegundos. El resto de datos son libres para que el usuario los utilice para implementar otro protocolo a un nivel más superior.
- **Trama acknowledge (reconocimiento):** La codificación en el protocolo sería 01000 (macro ACKN). Esta trama es esperada por el maestro inmediatamente después de envíar un "start".

- **Trama de bulk (envío masivo):** La codificación en el protocolo sería 00101(macro BFRM). Esta trama forma parte de un conjunto de envíos y no garantiza la llegada, por lo que por lo general no se debería de utilizar para la petición de datos de forma unitaria. Es enviada por el nodo (o el maestro) al maestro que inició la comunicación con un "start*z* tras transmitir la trama de reconocimiento.

- **End (finalización):** La codificación en el protocolo sería 00100 (macro STOP). Es enviada por el maestro con el fin de indicarle al destinatario que cese el envío de datos iniciado por el "start".

Adicionalmente, incluimos también un paquete para indicar que el destinatario no podía atender a una petición ya fuese síncrona o asíncrona en un determinado momento.

- **Reject:** La codificación en el protocolo sería 00010. Es una trama general que informa de que en estos momentos no se ha podido atender la petición del maestro/supervisor, indicando qué petición se está rechazando (recordemos que un maestro o un supervisor puede mantener tanto comunicaciones síncronas como asíncronas de forma simultánea con el mismo nodo) y debido a qué motivo.

La existencia de peticiones de transmisiones síncronas y asíncronas hacían nuestro protocolo incompatible con las tramas remotas de CAN, por lo que optamos por hacer uso de la trama de datos como base para todos nuestros posibles mensajes.

De esta forma una trama genérica en TouCAN tendría el siguiente aspecto (ver siguiente página):

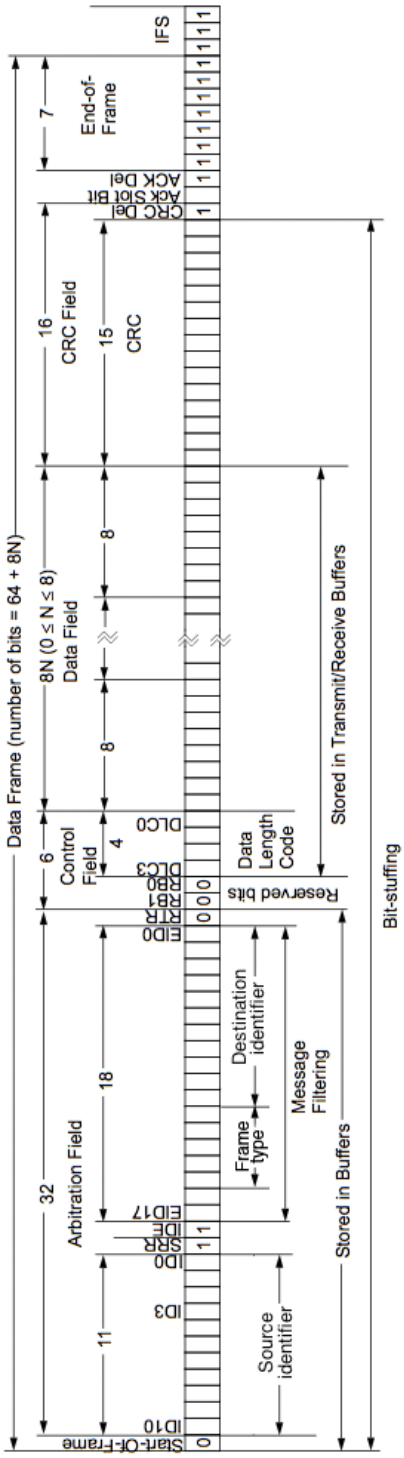


Figura 9.2: Trama de TouCAN basada en una trama de datos CAN

Los distintos tipos de trama son generados a partir de esta trama genérica modificando los bits de "frame type".

9.1.2. Organización del código

Llegados a este punto no sólo bastaba con haber diseñado el protocolo, puesto que había también que organizar el código de los microcontroladores de manera que se ajustasen a nuestras expectativas. Como hemos indicado en anteriores capítulos, el código Processing compilable consta de al menos dos funciones un setup() y un loop(), siendo el segundo de vital importancia para alojar el código del usuario, por lo que sabiendo esto y tras varias pruebas previas, decidimos organizar los códigos para maestros y nodos de las siguientes maneras:

9.1.2.1. Maestro

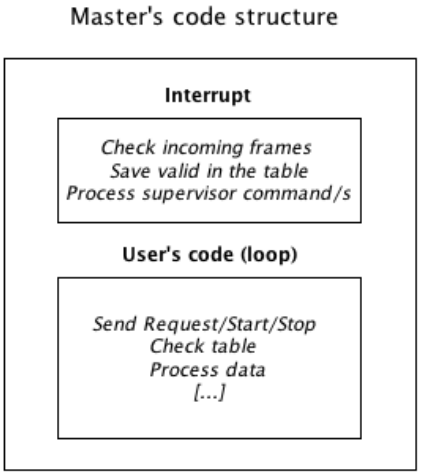


Figura 9.3: Código compilado en el maestro

Como se puede observar en el esquema, el maestro posee una interrupción (activada por tiempo) que conprueba de forma periódica si existen mensajes en alguno de los dos búferes de recepción, de haberlos los procesa y en caso de ser válidos son guardados en la tabla de comunicaciones del maestro. Mientras, en bucle principal se le otorga libertad al usuario para incluir su propio código y parar comprobar si existen datos válidos en la tabla, permitiéndole establecer de forma simultánea comunicaciones con varios nodos distintos.

9.1.2.2. Nodo común

Para los nodos describimos tres alternativas dependiendo de la carga de trabajo que vayan a realizar.

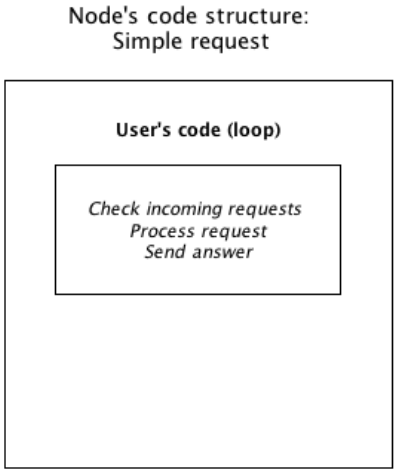


Figura 9.4: Código compilado en un nodo para una respuesta síncrona sencilla

Esta estructuración es apta cuando el nodo sólo responde a peticiones síncronas. Adecuado cuando se quiere un código ligero y sencillo, en cambio, cuando se requieren acciones más sofisticadas como el procesamiento de peticiones asíncronas en las que necesariamente hay que establecer tiempos, presentamos dos aproximaciones al problema.

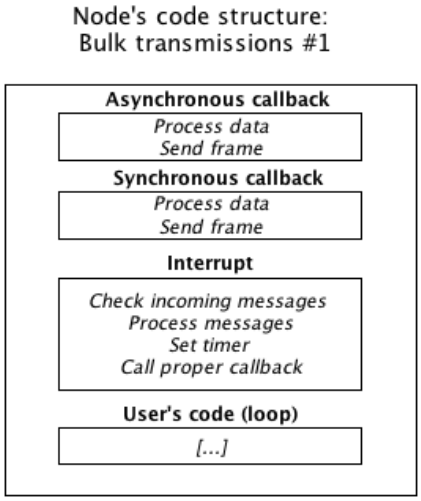


Figura 9.5: Código compilado en un nodo para atender peticiones asíncronas y síncronas en una interrupción

En esta primera organización se hace uso únicamente de una interrupción para gestionar todo el tránsito de datos, ya sea recepción o emisión. La ventaja que conlleva a la siguiente solución es que proporciona todo el bucle principal para el código de usuario, pero posee la desventaja de tener una frecuencia de envío limitiada a la frecuencia de salto de la interrupción la cual viene establecida por el tiempo de ejecución del código.

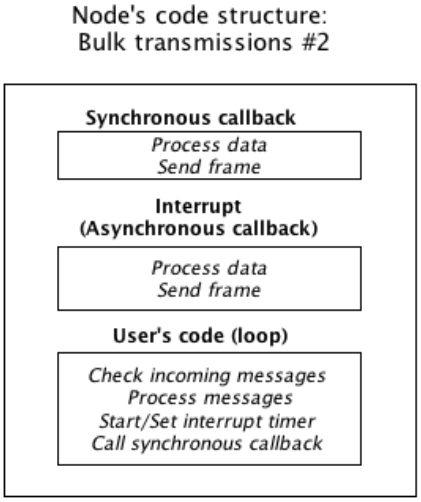


Figura 9.6: Código compilado en un nodo para atender peticiones asíncronas y síncronas en un loop

En esta segunda solución, la comprobación de la llegada de mensajes de comienzo o parada asíncronos se realiza en el bucle principal, restando espacio de código al usuario. Pese a ello, esta versión puede ser ventajosa sobre la anterior en los siguientes casos:

- Poco código de usuario: En los casos en los que el usuario requiera que el nodo simplemente se dedique a responder a sus peticiones y no se hagan procesos de datos ajenos a los envíos.
- Envíos muy rápidos: En este caso, la velocidad del envío no está limitada por el código de la librería sino por el propio código del usuario. En las pruebas se ha llegado de forma empírica a un período de 5ms, mientras que con la anterior versión el mínimo se mantiene alrededor de los 75ms para no quedarse atrapado en la interrupción.

Una vez definido el protocolo de forma conceptual pasemos a explicar la implementación sobre Processing.

9.1.3. Estructuras de datos. Clase TouCAN

Partiendo de la base de que Processing es un subconjunto de c++, podremos observar que las estructuras usadas para la implementación del protocolo son comunes a dicho lenguaje.

9.1.3.1. node

```
typedef struct{
        int id; //node id

        //Synchronous part
        boolean S_supervisor_op;
        float S_timestamp;
        byte S_answer;
        byte S_length;
        byte S_data[8];

        //Asynchronous part
        boolean A_superviso_op;
        float A_timestamp; //asynchronous part
        byte A_ack;
        byte written;
        byte A_length;
        byte A_data[8];
}node;
```

Se trata de un tipo basado en un struct. Generalmente representa de forma lógica un nodo de la red de cara al maestro. Al compilar el código hace falta declarar un vector de este tipo node de tantas posiciones como nodos haya en la red (contando al maestro). Un ejemplo de declaración sería el siguiente:

```
node nodes[NNODES+1];
```

Este vector simula una tabla, siendo capaz de almacenar información sobre las comunicaciones activas del maestro. Nace de la necesidad de que el maestro no se quede esperando por un dato pedido, sino que pueda comprobar la recepción a posteriori. Como se puede observar se ha declarado una posición más. Esta posición es especial puesto que representa al propio maestro, teniendo alguno de los campos un significado distinto al resto de los nodos. Para ahorrar tiempo en la búsqueda de elementos en la tabla se establecieron dos tipos de identificadores: reales y virtuales.

9.1.3.1.1. Identificadores reales: Son los conocidos de antemano por el usuario administrador de la red. Representan el valor real de identificador que posee cada dispositivo.

9.1.3.1.2. Identificadores virtuales: Se asignan en función de la posición del nodo en la tabla del maestro.
De esta manera, indicar el establecimiento de comunicación en la tabla se convierte

en un acceso directo, ahorrándonos la búsqueda secuencial.
Pasemos a ver los campos de la parte síncrona:

- **boolean S_supervisor_op:** Se trata de un campo que indica si la **comunicación síncrona** con el nodo ha sido iniciada por el maestro o por el supervisor (recordemos que si bien un supervisor no está conectado directamente a la red, puede realizar peticiones a otros nodos a través del maestro al que se encuentra conectado). Sus valores posibles son true si es una comunicación del supervisor y false si es del maestro.
- **float S_timestamp:** Indica que existe una comunicación síncrona con el nodo (u otro maestro de ser la entrada del maestro) si el valor es mayor que 0.
- **byte S_answer:** Indica si ha habido una respuesta síncrona del nodo destino. Si el valor es 0 es que no han habido respuestas, si es 1 es que sí la ha habido y en caso de ser 2 es que la petición ha sido rechazada. Si se trata de la entrada del maestro en la tabla, adquiere otro significado, siendo 0 que el maestro no ha recibido peticiones de otros maestros y 1 en caso contrrario.
- **byte S_length:** Como su nombre indica, se trata de la longitud de datos recibidos en la respuesta o petición en caso de ser la entrada del maestro) síncrona.
- **byte S_data[8]:** Es un vector que con el tamaño suficiente para almacenar los datos de una respuesta (o petición en caso de ser la entrada del maestro) de tamaño máximo (8 bytes). Se trata de una especie de buffer que almacena la respuesta para que a posteriori, cuando el maestro lo desee o pueda, lo procese.

Los campos de la parte asíncrona comparten muchas cosas en común con el lado síncrono pero dada su naturaleza, posee unos elementos adicionales.

- **boolean A_supervisor_op:** Se trata de un campo que indica si la **comunicación síncrona** con el nodo ha sido iniciada por el maestro o por el supervisor (recordemos que si bien un supervisor no está conectado directamente a la red, puede realizar peticiones a otros nodos a través del maestro al que se encuentra conectado). Sus valores posibles son true si es una comunicación del supervisor y false si es del maestro.
- **float A_timestamp:** Indica que existe una comunicación asíncrona con el nodo (u otro maestro de ser la entrada del maestro) si el valor es mayor que 0.
- **byte A_ack:** Indica si ha habido una trama de reconocimiento emitida desde el nodo destino. Si el valor es 0 es que no han habido respuestas, si es 1 es que sí la ha habido y en caso de ser 2 es que el "start.º "stop"previo ha sido rechazado. Si se trata de la entrada del maestro en la tabla, adquiere otro significado, siendo 0 que el maestro no ha recibido ningún "start"de otro maestro, 1 en caso contrrario, 2 que se ha recibido un "stop"válido de otro maestro y 3 que el maestro se encuentra en un estado de envío activo.
- **byte A_written:** Como supusimos inicialmente (y más tarde se confirmó en las pruebas), existe la posibilidad de que los envíos de tramas asíncronas se

produzcan a intervalos de tiempo menores a los que el maestro comprueba si ha llegado algún mensaje. Como este tipo de transmisión no garantiza la pérdida de un paquete, decidimos que al menos debíamos informar al usuario de que había sido posible una sobreescritura en el buffer de la entrada correspondinete de la tabla de nodos.

- **byte A_length:** Como su nombre indica, se trata de la longitud de datos recibidos en la respuesta o petición en caso de ser la entrada del maestro) síncrona.
- **byte A_data[8]:** Es un vector que con el tamaño suficiente para almacenar los datos de una trama asíncrona de tamaño máximo (8 bytes). Se trata de una especie de buffer que almacena la respuesta para que a posteriori, cuando el maestro lo desee o pueda, lo procese.

Como se puede notar sólo existe una entrada para el propio maestro, lo cual limita el número de peticiones que éste puede atender. Por cuestiones de diseño, el maestro sólo atenderá a un único maestro, y sólo en el caso de terminar todas las comunicaciones con el maestro que inició las transmisiones, se atenderá a otro. Cualquier petición llegada antes de esta situación será rechazado con el valor BUSY en la segunda posición del campo de datos.

Por otro lado, tenemos la clase TouCAN, basada en ArCAN. Internamente esta clase posee una estructura que representa el mensaje que será transmitido por le bus CAN.

9.1.3.2. tCAN

```
typedef struct{
        int id_s;
        int id_d;
        int frametype;
        struct {
                byte rtr : 0;
                byte length : 0;
        }header;
        byte data[8];
}tCAN;
```

Nos encontramos ante la representación del mensaje de cara al usuario. Sobre esta estructura se almacenará el mensaje obtenido de los buffers de recepción del MCPS2515 y de esta estructura se obtendrán los datos para el envío del mensaje a través de los buffers de transmisión. Pasemos a ver los campos:

- **int id_s:** Se trata de la dirección de origen del mensaje.
- **int id_d:** Se trata de la dirección de destino del mensaje.
- **int frametype:** Indica de qué tipo de trama se trata.
- **struct header:** Este struct almacena el tipo de trama a nivel de CAN (remota o de datos) y la longitud del campo de datos
- **byte data[8]:** Los datos del mensaje propiamente dicho.

9.1.3.3. Métodos de TouCAN

Podemos separar los métodos usados internamente por TouCAN (muchos de ellos basados en ArCAN pero con ciertas mejoras y correcciones) y los métodos públicos para el usuario. Nombramos a continuación todos los métodos disponibles en la clase. Los métodos más importantes serán explicados en mayor profundidad en las siguientes secciones.

- **tTouCAN(void):** constructor de la clase Arcan.
- **boolean set_mode(int):** Indica en qué modo ha de funcionar el dispositivo. Por lo general hacemos uso de dos modos: configuración durante el setup y normal una vez comienza el loop.
- **boolean init(void):** Establece los registros de la lógica de control mediante comandos SPI para la correcta configuración del shield. Es invocado una única vez durante el setup.
- **boolean check_message(void):** Comprueba la existencia de mensajes en los bufferes de recepción del MCP2515.
- **boolean check_free_buffer(void):**Comprueba la existencia de buffers libre para la transmisión de una trama.
- **byte get_message(tCAN *):** Comprueba la existencia de mensajes en los bufferes de recepción del MCP2515 y de haber alguno, lo guarda sobre la estructura tCAN pasada como parámetro.
- **byte get_messageB(tCAN *):** Versión bloqueante de get_message. No puede ser interrumpido.
- **byte send_message(tCAN *):** Comprueba la existencia de buffers libre para la transmisión de una trama. y de haberlos, deposita el mensaje pasado como argumento para el envío.
- **byte send_messageB(tCAN *):** Versión bloqueante de send_message. No puede ser interrumpido.
- **boolean set_mask(word,int):** Establece una máscara estándar para el dispositivo. Como norma general ha de ser llamado dos veces, una por cada máscara extendida existente.
- **boolean set_extended_mask(word,int):** Establece una máscara extendida para el dispositivo. Como norma general ha de ser llamado dos veces, una por cada máscara extendida existente.
- **boolean set_filter(word,int):** Establece uno de los seis filtros estándar del dispositivo indicado. En general ha de ser llamado seis veces durante el setup, indicando en el primer parámetro el valor del filtro y en el segundo de cuál de los filtros se trata.
- **boolean set_extended_filter(word,int):** Establece uno de los seis filtros estándar del dispositivo.En general ha de ser llamado seis veces durante el setup, indicando en el primer parámetro el valor del filtro y en el segundo de cuál de los filtros se trata.

- **byte spi_putc(byte data):** Envía un comando SPI.
- **void write_register(byte direction, byte data):** Realiza la escritura de un registro via el comando SPI correspondiente.
- **byte read_register(byte direction):** Realiza una lectura de un registro vía el comando SPI correspondiente.
- **void bit_modify(byte direction, byte mask, byte data):** Modifica un bit de un registro via el comando SPI correspondiente.
- **byte read_status(byte type):** Comprueba el estado del MCP2515.
- **int init_nodes(node *, int):** Inicializa la lista de nodos del maestro.
- **int do_request(tCAN *, int, int, int, int):** Se encarga de llamar a send_message o send_messageB (según se indique en el último parámetro) con el fin de realizar una petición síncrona.
- **int is_request(tCAN *):** Pregunta si el mensaje es una petición síncrona, devolviendo 1 en caso afirmativo y 0 en caso contrario.
- **int request(tCAN *, int, int, int, int , node *, int, int):** Es la función de usuario para realizar una petición síncrona.
- **int do_answer(tCAN *, int, int, int, int):** Se encarga de llamar a send_message o send_messageB (según se indique en el último parámetro) con el fin de realizar una respuesta síncrona.
- **int is_answer(tCAN *, int):** Pregunta si el mensaje es una respuesta síncrona, devolviendo 1 en caso afirmativo y 0 en caso contrario.
- **int answer(tCAN *, int, int, int, node *, int, int):** Es la función de usuario para realizar una respuesta síncrona.
- **int do_start_transmission(tCAN *, int, int, int, float, boolean):** Se encarga de llamar a send_message o send_messageB (según se indique en el último parámetro) con el fin de realizar un comienzo de comunicación asíncrona.
- **int is_start(tCAN *msje):** Pregunta si el mensaje es start, devolviendo 1 en caso afirmativo y 0 en caso contrario.
- **int start_transmission(tCAN *, int, int, boolean, int, float, float, node *,boolean):** Es la función de usuario para realizar un comienzo de comunicación asíncrona.
- **void get_start(tCAN *, int, void (*time_handler)(), int (*sync_callback)(int , byte *, int), byte *):** Se trata de la función principal de un nodo que siga la estructuración de la figura 4.6. Se encarga de obtener mensajes en el loop y procesarlos para dar una respuesta adecuada al remitente, establece la interrupción disparada por tiempo y procesa los datos llamando a los callbacks adecuados.

- **void get_start_interrupt(tCAN *, int, int , int (*sync_callback)(int , byte *, int),int (*async_callback)(int , byte *, int)):** Se trata de la función principal de un nodo que siga la estructuración de la figura 4.5. Esta función únicamente ha de ser colocada en una interrupción controlada por un timer. Se encarga de controlar la llegada de mensajes, procesarlos y establecer contadores de tiempo en caso de que sean starts válidos. Por establecer una analogía con la función anterior, el procesamiento de datos asíncronos se realiza en un callback.

- **int stop_transmission(tCAN *, int, int, int, boolean, node *, boolean):** Es la función de usuario para realizar un comienzo de comunicación asíncrona. Establece el período de envío en el primer byte del campo de datos.

- **int bulk_frame(tCAN *, int, int, int, int):** Es la función destinada a realizar el envío de un paquete de datos asíncrono.

- **int is_stop(tCAN *, int):** Pregunta si el mensaje es un stop, devolviendo 1 en caso afirmativo y 0 en caso contrario.

- **int is_ack(tCAN *, int):** Pregunta si el mensaje es un ack(reconocimiento), devolviendo 1 en caso afirmativo y 0 en caso contrario.

- **int acknowledge(tCAN *, int, int, boolean):** Es la función destinada a realizar el envío de una trama de reconocimiento TouCAN.

- **int reject(tCAN *, int, int, int, int):** Es la función destinada a realizar el envío de una trama de rechazo TouCAN.

- **int is_reject(tCAN *):** Pregunta si el mensaje es un reject, devolviendo 1 en caso afirmativo y 0 en caso contrario.

- **int ifreceived(node *, int, int);** Es una función de maestro con el fin de comprobar en la lista de nodos si ha llegado una determinada respuesta síncrona o asíncrona.

- **int ifreceivedB(node *, int, int, float):** Versión bloqueante de ifreceived(). Bloquea al programa hasta recibir la respuesta.

- **int do_attend_petition(node *, byte *, byte *, int):** Comprueba si existen peticiones de un maestro en la tabla de nodos, devolviendo en los parámetros de entrada/salida 0 si no lo ha habido y 1 en caso contrario.

- **int attend_petition(int, node *, int, int * int, float *, int (*sync_callback)(int , byte *, int));:** Es una función de maestro que tiene la finalidad de comprobar en la tabla si existe alguna petición de otro maestro, con el fin de atenderla llamando al callback síncrono o activando el envío en su interrupción según el caso.

- **int save_data(tCAN *, int, node *, int):** Tiene como objetivo guardar los mensajes válidos en la lista de nodos del maestro. De forma general ha de situarse en el código de la interrupción lanzada por el timer.

- **int check_received_command(tCAN *):** Esta función y las dos siguientes nacen como requerimientos del supervisor. Comprueba si existe un comando del supervisor por el puerto serial y, en caso de haberlos, construye un mensaje de tipo tCAN a partir de éste. Devuelve como valor el tipo de comando de supervisor (compatible con los mensajes TouCAN: request, start, stop).
- **void send_frame(tCAN *):** Procesa el mensaje pasado por parámetro y lo deposita en el buffer serial, desde donde será enviado al supervisor.
- **void perform_supervisor_operation(tCAN *, int, int *, float *_t, node *, int, void (*self_sync_callback)(tCAN), void (*self_async_callback)(tCAN)):** Realiza el comando indicado por check_received_command().
- voidsend_device_list(int, node *, int): Se encarga de enviar los identificadores de los dispositivos conectados en la red al supervisor. Se ha de llamar durante el setup y después de haber configurado la tabla de nodos.

Pasamos ahora a explicar de forma más detallada las funciones más destacables de cara al usuario.

9.1.3.4. Métodos de configuración

Este conjunto de funciones han de ser llamadas durante el setup puesto que son funciones de inicialización o configuración del dispositivo.

9.1.3.4.1. set_extended_filter(): Esta función es la encargada de establecer el filtro extendido para el dispositivo. Puesto que el destinatario es enviado en la parte extendida del mensaje, este será el único tipo de filtro necesario para el dispositivo.

```
boolean set_extended_filter(word, int);
```

Sus parámetros de entrada son:

- **word data:** Este será el valor que adquirirá el filtro. Por exigencias del protocolo TouCAN, es recomendable que sea igual al valor de la ID del nodo en la red.
- **int index:** Indica cuál de los seis filtros es el que va a ser configurado.

Al haber seis filtros extendidos en el MCP2515, será necesario llamar seis veces a esta función.

9.1.3.4.2. set_extended_mask(): Esta función es la encargada de establecer la máscara extendida para el dispositivo. Puesto que el destinatario es enviado en la parte extendida del mensaje, este será el único tipo de máscara necesaria para el dispositivo.

```
boolean set_extended_mask(word data, int index);
```

Sus parámetros de entrada son:

- **word data:** Este será el valor que adquirirá la máscara. Tal y como definimos el funcionamiento de las máscaras en CAN en el capítulo .ªnálisis del problema", es recomendable colocarlo a 0xFF para forzar la exactitud del identificador de destino con los filtros.

- **int index:** Indica cuál de las dos máscaras es la que va a ser configurada.

Al haber dos máscaras extendidas en el MCP2515, será necesario llamar dos veces a esta función.

9.1.3.4.3. init_nodes(): Simple y llanamente se encarga de dar valores iniciales a la tabla de nodos.Únicamente ha de ser llamada por un maestro puesto que es el único elemento de la red en tener esta tabla.

```
int init_nodes(node *nodes, int length);
```

Su parámetros son:

- **node *nodes:** Es un parámetro de entrada salida que representa la tabla/ lista de nodos en la red
- **int length:** Indica la longitud de la tabla.

9.1.3.5. Métodos de comunicación síncrona

Este conjunto de métodos son los necesarios para establecer una comunicación petición-respuesta de tipo síncrono.

9.1.3.5.1. request(): Método para el maestro que encargado de enviar una trama de tipo request al destinatario indicándole que desea un dato de forma síncrona. El campo de datos queda totalmente libre para que el usuario incluya su propio protocolo en él. Señala, de forma automática, un establecimiento de comunicación síncrona en la tabla de nodos mediante una marca de tiempo.

```
int request(tCAN *msge, int id_s, int id_d, \
        int length, int timeout, node *nodes, \
        boolean is_supervisor_op, boolean is_blocking);
```

Parámetros:

- **tCAN *msge:** El mensaje a enviar. En caso de querer enviar datos, éstos han de estar preparados antes de llamar al método.
- **int id_s:** Indica el origen del mensaje. Para un correcto funcionamiento, el protocolo obliga a que se ponga la ID real del maestro.
- **int id_d:** Indica el destinatario del mensaje usando su ID virtual.
- **int length:** Indica la longitud del campo de datos.
- **int timeout:** Tiempo mínimo necesario para poder ser capaz de hacer otro request si hubiese un anterior no respondido..
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.

- **boolean is_supervisor_op:** Indica si se trata de una operación realizada por el maestro por iniciativa propia o si ha sido iniciada por el supervisor. En el segundo caso no se guardará el dato de respuesta sobre la tabla sino que será enviado directamente al supervisor mediante el puerto serial.
- **boolean is_blocking:** Indica si la función ha de ser bloqueante o no.

Los valores devueltos posibles son: 1 si la petición se ha transmitido de forma exitosa, 0 si no ha sido posible o PREVSTILLACTIVE en caso de que hubiese una petición anterior en curso y válida según el parámetro timeout.

9.1.3.5.2. answer(): Método tanto para el maestro como para el nodo encargado de envíar una trama de tipo answer al destinatario que previamente envío una trama request.

```
int answer(tCAN *msge, int id_s, int id_d, \
        int length, node *nodes, int nnodes, \
        boolean is_blocking);
```

Parámetros:

- **tCAN *msge:** El mensaje a enviar. En caso de querer enviar datos, éstos han de estar preparados antes de llamar al método.
- **int id_s:** Indica el origen del mensaje. Para un correcto funcionamiento, el protocolo obliga a que se ponga la ID real.
- **int id_d:** Indica el destinatario del mensaje usando su ID real.
- **int length:** Indica la longitud del campo de datos.
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red. El valor ha de ser **NULL** si se trata de un nodo común.
- **boolean is_blocking:** Indica si la función ha de ser bloqueante o no.

Los valores devueltos posibles son: 1 si la respuesta se ha transmitido de forma exitosa, 0 si no ha sido posible.

9.1.3.6. Métodos de comunicación asíncrona

Estos métodos son los indicados en caso de querer establecer una comunicación con un envío de datos grande y constante.

9.1.3.6.1. start_transmission(): Método para el maestro que encargado de enviar una trama de tipo start al destinatario indicándole que desea un datos de forma asíncrona. El campo de datos queda totalmente libre para que el usuario incluya su propio protocolo en él. Señala, de forma automática, un establecimiento de comunicación asíncrona en la tabla de nodos mediante una marca de tiempo.

```
int start_transmission(tCAN *msge, int id_s, \
        int id_d, int length,\
        boolean is_supervisor_op, float sampling,\
        float timeout, node *nodes, \
        boolean is_blocking);
```

Parámetros:

- **tCAN *msge:** El mensaje a enviar. En caso de querer enviar datos, éstos han de estar preparados antes de llamar al método.
- **int id_s:** Indica el origen del mensaje. Para un correcto funcionamiento, el protocolo obliga a que se ponga la ID real del maestro.
- **int id_d:** Indica el destinatario del mensaje usando su ID virtual.
- **int length:** Indica la longitud del campo de datos.
- **boolean is_supervisor_op:** Indica si se trata de una operación realizada por el maestro por iniciativa propia o si ha sido iniciada por el supervisor. En el segundo caso no se guardará el dato de respuesta sobre la tabla sino que será enviado directamente al supervisor mediante el puerto serial.
- **float sampling:** Frecuencia deseada para el envío de datos por parte del destinatario. Expresado en milisegundos.
- **float timeout:** Se mantiene por cuestiones de compatibilidad.
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.
- **boolean is_blocking:** Indica si la función ha de ser bloqueante o no.

Los valores devueltos posibles son: 1 si la mensaje se ha transmitido de forma exitosa, 0 si no ha sido posible.

9.1.3.6.2. acknowledge(): Método que envía una trama ACKN (reconocimiento) con el fin de informar al maestro que inició o intentó terminar la comunicación asíncrona ha sido establecida o terminada correctamente. Ha de ser enviada por el nodo destinatario de un start si éste fuese válido. Por lo general, el usuario no debería usar este método puesto que existen otros métodos de recolección de datos que de forma automática realizan el envío de esta trama, no obstante; consideramos que es de gran importancia puesto que sin ella el protocolo no lograría funcionar de manera correcta por lo que es vital conocer su existencia.

Internamente, la trama enviada posee un campo de datos de longitud 1, el cual indica a qué trama se hace reconocimiento, si a un start o a un stop. Este dato ha de ser rellenado antes de llamar al método.

```
int acknowledge(tCAN *msge, int id_s, \
        int id_d, boolean is_blocking);
```

Parámetros:

- **tCAN *msge:** El mensaje a enviar. En caso de querer enviar datos, éstos han de estar preparados antes de llamar al método.

- **int id_s:** Indica el origen del mensaje. Para un correcto funcionamiento, el protocolo obliga a que se ponga la ID real.
- **int id_d:** Indica el destinatario del mensaje usando su ID real.
- **boolean is_blocking:** Indica si la función ha de ser bloqueante o no.

Los valores devueltos posibles son: 1 si el reconocimiento se ha transmitido de forma exitosa, 0 si no ha sido posible.

9.1.3.6.3. bulk_frame(): Método para realizar envíos de datos al maestro que inició la comunicación asíncrona. No requiere de ningún procesamiento de datos extra aparte de los del usuario por lo que se pueden realizar envíos masivos a altas velocidades. No obstante la llegada de todos los mensajes no está garantizada. El campo de datos, lógicamente queda totalmente a disposición del usuario.

```
int bulk_frame(tCAN *msge, int id_s, \
int id_d, int length, boolean is_blocking);
```

Parámetros:

- **tCAN *msge:** El mensaje a enviar. En caso de querer enviar datos, éstos han de estar preparados antes de llamar al método.
- **int id_s:** Indica el origen del mensaje. Para un correcto funcionamiento, el protocolo obliga a que se ponga la ID real.
- **int id_d:** Indica el destinatario del mensaje usando su ID real.
- **int length:** Indica la longitud del campo de datos.
- **boolean is_blocking:** Indica si la función ha de ser bloqueante o no.

Los valores devueltos posibles son: 1 si la mensaje se ha transmitido de forma exitosa, 0 si no ha sido posible.

9.1.3.6.4. stop_transmission(): Método para detener el envío de tramas asíncronas de tipo bulk. Señala, de forma automática, un intento de finalizacion de comunicación asíncrona en la tabla de nodos mediante una marca de tiempo negativa. El campo de datos queda a disposición del usuario.

```
int stop_transmission(tCAN *msge, int id_s, \
        int id_d, int length, boolean is_supervisor_op\
        node *nodes, boolean is_blocking);
```

Parámetros:

- **tCAN *msge:** El mensaje a enviar. En caso de querer enviar datos, éstos han de estar preparados antes de llamar al método.
- **int id_s:** Indica el origen del mensaje. Para un correcto funcionamiento, el protocolo obliga a que se ponga la ID real del maestro.
- **int id_d:** Indica el destinatario del mensaje usando su ID virtual.
- **int length:** Indica la longitud del campo de datos.

- **boolean is_supervisor_op:** Indica si se trata de una operación realizada por el maestro por iniciativa propia o si ha sido iniciada por el supervisor. En el segundo caso no se guardará el dato de respuesta sobre la tabla sino que será enviado directamente al supervisor mediante el puerto serial.
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.
- **boolean is_blocking:** Indica si la función ha de ser bloqueante o no.

Los valores devueltos posibles son: 1 si la mensaje se ha transmitido de forma exitosa, 0 si no ha sido posible.

9.1.3.6.5. get_start(): Método exclusivo de nodo que corresponde a la estructuración de código mostrada en la figura 4.6. Ha de colocarse por definición en el bucle principal (loop). Se encarga de la recepción de mensajes y su procesamiento, llamando a los callbacks adecuados dependiendo de la petición.

```
void get_start(tCAN *msge, int id_s, \
        void (*time_handler)(), \
        int (*sync_callback)(int , byte *, int), \
        byte *interrupt_active );
```

Parámetros:

- **tCAN *msge:** Esta estructura tCAN actuará como buffer de recepción/transmisión para el método.
- **int id_s:** Indica al método quién es el nodo fuente para los posibles envíos.
- **void (*time_handler)():** Manejador de la interrupción activada por tiempo. De existir comunicaciones asíncronas, de forma periódica este manejador será llamado con el fin de enviar tramas de tipo bulk procesadas. El manejador debe incluir **obligatoriamente** una llamada al método bulk_frame() puesto que internamente get_start() no lo realiza.
- **int (*sync_callback)(int , byte *, int):** Puntero a un manejador de peticiones síncronas. Por orden, sus tres parámetros son: el identificador real del dispositivo, un vector de bytes de datos y la longitud de dicho vector. La función debe incluir **obligatoriamente** una llamada al método answer() puesto que internamente get_start() no lo realiza.
- **byte *interrupt_active:** Parámetro de entrada salida para indicar si se ha activado la interrupción para el manejador de tiempo. La existencia de este campo es debido a que el método ha de comportarse de forma distinta si existe una interrupción activa o no y tratar de propagar este comportamiento al resto de métodos que posean un parámetro is_blocking que pudieran haber.

9.1.3.6.6. get_start_interrupt(): Método exclusivo de nodo que corresponde a la estructuración de código mostrada en la figura 4.5. Ha de colocarse por definición en el manejador de una interrupción activada por tiempo. Se encarga de la recepción de mensajes y su procesamiento, llamando a los callbacks adecuados dependiendo de la petición.

```
void get_start_interrupt(tCAN *msge,\
        int id_s, int period,
        int (*sync_callback)(int , byte *, int),\
        int (*async_callback)(int , byte *, int));
```

Parámetros:

- **tCAN *msge:** Esta estructura tCAN actuará como buffer de recepción/transmisión para el método.
- **int id_s:** Indica al método quién es el nodo fuente para los posibles envíos.
- **int period:** Valor del período al que funciona actualmente la interrupción disparada por tiempo.
- **int (*sync_callback)(int , byte *, int):** Se trata un callback para el procesamiento y envío de respuestas síncronas. Por orden, sus tres parámetros son: el identificador real del dispositivo, un vector de bytes de datos y la longitud de dicho vector. La función debe incluir **obligatoriamente** una llamada al método answer() puesto que internamente get_start_interrupt() no lo realiza.
- **int (*async_callback)(int , byte *, int):** Se trata un callback para el procesamiento y envío de tramas bulk asíncronas. Por orden, sus tres parámetros son: el identificador real del dispositivo, un vector de bytes de datos y la longitud de dicho vector.La función debe incluir **obligatoriamente** una llamada al método bulk_frame() puesto que internamente get_star_interruptt() no lo realiza.

Es importante indicar que las pruebas han demostrado que es recomendable un período de **75ms** para la interrupción de tiempo. Un valor menor puede conllevar a comportamientos indeseados.

9.1.3.7. Métodos de guardado y comprobación de mensajes

En este conjunto de métodos, nos centraremos en aquellos que le proporcionan al maestro la habilidad de establecer varias comunicaciones de forma simultánea sin necesidad de que esto interfiera en gran medida en su propio tiempo para el procesamiento de datos.

9.1.3.7.1. save_data(): Probablemente el método más importante de lo que se vayan a citar del conjunto puesto que sin él, el resto simplemente no funcionaría. Se encarga del guardado de todos los mensajes válidos fruto de una comunicación de un maestro con un dispositvo salvo que la comunicación haya sido establecida por el supervisor, en cuyo caso enviará el dato por el puerto serial. Es posible situarlo tanto en el bucle principal como en una interrupción y aunque ambas estructuraciones funcionan, recomendamos la segunda puesto que no ocupa espacio en el bucle principal que es donde, por lo general, estará el código de usuario.

```
int save_data(tCAN *msge, int id_s, \
        node *nodes, int nnodes);
```

Parámetros:

- **tCAN *msge:** Esta estructura tCAN actuará como buffer de recepción/transmisión para el método.
- **int id_s:** Indica al método quién es el nodo fuente para los posibles envíos. El protocolo exige que sea el identificador real del maestro donde se llame al método.
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.
- **int nnodes:** Longitud de la tabla de nodos.

Los valores devueltos posibles son: -1 si no ha habido ningún mensaje en ninguno de los buffers de recepción o un valor entre 0 y nnodes indicando en qué posición de la tabla se guardó el mensaje.

9.1.3.7.2. received(): Este método ha de ser llamado por el maestro cuando desee comprobar en la tabla si ha recibido un dato esperado. En caso de que exista una mensaje, limpia la entrada de la tabla para el tipo de comunicación con ese nodo, por lo que en casos de transmisiones asíncronas se recomienda procesar, copiar el dato lo antes posible o detener la interrupción para no perder el valor.

```
int ifreceived(node *nodes, int id, int mode);
```

Parámetros:

- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.
- **int id:** Indica qué entrada en la tabla de nodos es la que se va a inspeccionar. Ha de coincidir con el valor de identificador virtual.
- **int mode:** Indica qué clase de respuesta busca: una respuesta síncrona (ANSW), una trama de reconocimiento(ACKN) o un paquete asíncrono(BFRM).

Los valores devueltos posibles son:

- **0:** Si no ha habido ningún mensaje la entrada inspeccionada (sea cual sea por el tipo de trama preguntado).
- **1:** Si se ha preguntado por una repuesta síncrona(ANSW) o una trama de reconocimiento(ACKN) y ésta se encontraba en la entrada correspondiente en la tabla.
- **1, 2:** Se se ha preguntado por una trama de tipo asíncrono(BFRM) y ésta estaba. El primer valor representa una trama sin pérdida previa, el segundo valor representa una que ha sobreescrito una trama anterior.

9.1.3.7.3. receivedB(): Versión bloqueante del anterior método. Al preguntar por un determinado mensaje, se quedará bloqueado hasta recibirlo o hasta que el valor del timeout le haga salir. Se recomienda su uso cuando se esperan datos vitales de los que dependerán las siguientes líneas de código.

```
int ifreceivedB(node *nodes, int id, int mode, \
        float timeout);
```

Los valores devueltos posibles son:

- **0:** Si no ha habido ningún mensaje la entrada inspeccionada (sea cual sea por el tipo de trama preguntado).
- **1:** Si se ha preguntado por una repuesta síncrona(ANSW) o una trama de reconocimiento(ACKN) y ésta se encontraba en la entrada correspondiente en la tabla.
- **1, 2:** Se se ha preguntado por una trama de tipo asíncrono(BFRM) y ésta estaba. El primer valor representa una trama sin pérdida previa, el segundo valor representa una que ha sobreescrito una trama anterior.

NOTA IMPORTANTE: Recalcamos que tanto este método como el anterior sólo buscan por respuestas a peticiones (síncronas o asíncronas) generadas por el propio maestro. No son válidos para buscar comunicaciones establecidas por el supervisor o peticiones de otros maestros. Para atender comunicaciones establecidas por otros maestros existe el método attend_petition().

9.1.3.7.4. attend_petition(): Método para comprobar si existen operaciones pendientes de comunicaciones iniciadas por otros maestros de la red. Si existieran, las atiende llamando a los callbacks adecuados o activando contadores para los envíos.

```
void attend_petition(int id_s, node *nodes, \
        int nnodes, int *start, int period, \
        float *send_t,\
        int (*sync_callback)(int , byte *, int));
```

Parámetros:

- **int id_s:** Indica al método quién es el nodo fuente para los posibles envíos. El protocolo exige que sea el identificador real del maestro donde se llame al método.
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.
- **int nnodes:** Longitud de la tabla de nodos.
- **int *start:** Parámetro de entrada/salida para indicar si ha de comenzar la cuenta para realizar los envíos. Inicialmente ha de estar a 0. El método se encargará luego de configurarlo al valor necesario.
- **int period:** Valor del período con el que funciona la interrupción del maestro

- **float *send_t:** Valor del período para la realización del envío de datos asíncronos. Se calcula internamente en el método al llegar un mensaje de tipo start válido.
- **int (*sync_callback)(int , byte *, int):** Puntero a una función para el procesado y envío de respuestas síncronas. La función **obligatoriamente** ha de incluir una llamada al método answer() puesto que internamente, attend_petition() no lo realiza.

9.1.3.8. Métodos para servir al supervisor

Cronológicamente hablando, estos son los métodos más recientes en incluirse en el lado microcontrolador de TouCAN. Nacen de la necesidad de tener algún mecanismo con el que comunicarse con el supervisor. Recordemos que la figura del supervisor es la de un equipo informático más potente que los microcontroladores, conectado via puerto serie a un maestro. Es por ello que los siguientes métodos hacen un uso intensivo del puerto serial como canal de comunicación.

9.1.3.8.1. check_received_command(): Este método recoge byte a byte el comando enviado por el supervisor al maestro.

```
int check_received_command(tCAN *msge);
```

Parámetros:

- **tCAN *msge:** Esta estructura tCAN servirá como estructura de almacenamiento para el comando enviado por puerto serie por el supervisor.

Los valores devueltos son o bien el tipo de mensaje recibido si todo ha ido bien o -1 si ha habido algún problema o no había mensajes en ese momento.

9.1.3.8.2. send_frame(): Se encarga de enviar respuestas mediante el puerto serial al supervisort. No es un método especialmente destacable de cara al usuario final, no obstante su importancia reside en que forma parte vital de los mecanismos de comunicación maestro-supervisor.

```
void send_frame(tCAN *msge);
```

Parámetros:

- **tCAN *msge:** Mensaje que será transmitido por puerto serial.

9.1.3.8.3. void perform_supervisor_operation(: Este método recoge byte a byte el comando enviado por el supervisor al maestro. y realiza el envío de la trama TouCAN adecuada. Se puede entender como el equivalente de attend_petition() para el supervisor.

```
void perform_supervisor_operation(tCAN *msge,\
        int id_s, int *start, float *send_t, \
        node *nodes, int nnodes, \
        void (*self_sync_callback)(tCAN), \
        void (*self_async_callback)(tCAN));
```

Parámetros:

- **tCAN *msge:** Esta estructura tCAN servirá como estructura de almacenamiento para el comando enviado por puerto serie por el supervisor y posteriormente será enviado por la red TouCAN si fuera preciso.
- **int id_s:** Indica al método quién es el nodo fuente para los posibles envíos. El protocolo exige que sea el identificador real del maestro donde se llame al método.
- **int *start:** Parámetro de entrada/salida para indicar si ha de comenzar la cuenta para realizar los envíos. Inicialmente ha de estar a 0. El método se encargará luego de configurarlo al valor necesario.
- **float *send_t:** Valor del período para la realización del envío de datos asíncronos. Se calcula internamente en el método al llegar un mensaje de tipo start válido.
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.
- **int nnodes:** Longitud de la tabla de nodos.
- **int (*self_sync_callback)(int , byte *, int):** Puntero a una función para el procesado y envío de respuestas síncronas cuando el destinatario de la petición síncrona es el propio maestro. La función **obligatoriamente** ha de incluir una llamada al método send_frame() puesto que internamente, perform_supervisor_operation() no lo realiza.
- **int (*self_async_callback)(int , byte *, int):** Puntero a una función para el procesado y envío de tramas asíncronas cuando el destinatario de la petición asíncrona es el propio maestro. La función **obligatoriamente** ha de incluir una llamada al método send_frame() puesto que internamente, perform_supervisor_operation() no lo realiza.

9.1.3.8.4. send_device_list(): Envía al supervisor los identificadores de los nodos que componen la red TouCAN.

```
void send_device_list(int nnodes, \
        node *nodes, int id_s);
```

Parámetros:

- **int nnodes:** Longitud de la tabla de nodos.
- **node *nodes:** Parámetro de entrada salida que representa la tabla/ lista de nodos en la red.
- **int id_s:** Identificador real del maestro, es necesario pasarlo como parámetro aparte puesto que no se encuentra en la tabla.

9.1.4. El timer. MsTimer2

Tras haber leído la documentación sobre el lado microcontrolador, se habrá notado que nombramos continuamente una interrupción disparada por un timer pero no hemos contado exactamente cómo hacer uso exactamente de este recurso Hardware proporcionado por la placa Arduino. En nuestro caso optamos por hacer uso de la librería MsTimer2, la cual gobierna el Timer2 de la placa Arduino UNO. Se trata de una librería sencilla (de ahí su elección por encima de otras) que permite gestionar el Timer 2. Su funciones de cara al usuario más importantes son:

- **void set(unsigned long ms, void (*f)()):** Establece el período de la interrupción de Timer 2 en el primer parámetro y un manejador en el segundo.
- **void start():** Arranca el timer.
- **void stop():** Detiene el timer.

A simple vista podrían parecer un conjunto de funciones suficientes para un usuario común, no obstante, muchos de los métodos TouCAN son bloqueantes o poseen una versión bloqueante. El mecanismo para adquirir esta ïninterrumpibilidad.[es] parando el timer de la interrupción y reanudándolo. Inicialmente usamos la siguiente estructura de código:

```
//interruptible code
[...]
MsTimer2::stop();
//impotant and uninterruptible code
[...]
MsTimer2::start();
//interruptible code
[...]
```

Sin embargo, al realizar varias pruebas nos dimos cuenta de que el comportamiento con funciones bloqueantes era impredecible por lo que indagando un poco más descubrimos que el culpable era la función start(). Esta función resetea los contadores del timer por lo que cada vez que lo llamábamos reiniciábamos el conteo de tics de la interrupción, por lo que tuvimos que implementar una nueva función a la librería: la función resume().

- **void resume():** Reanuda el timer tras un stop, siguiendo la cuenta de tics previa.

Así pues, el código bloqueante tendría el siguiente aspecto:

```
//interruptible code
[...]
MsTimer2::stop();
//impotant and uninterruptible code
[...]
MsTimer2::resume();
//interruptible code
[...]
```

Al ser resume() una función sencilla es posible portarla a futuras versiones de la librería MsTimer2.

9.2. Lado supervisor

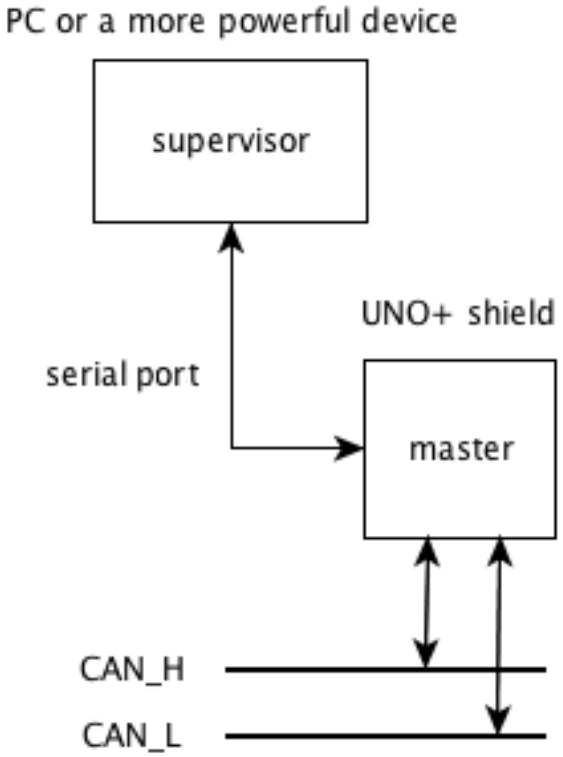


Figura 9.7: Supervisor conectado a maestro vía puerto serie.

Es necesario explicar el funcionamiento de este lado puesto que sin él no se entendería por qué se tomaron ciertas decisiones en el diseño. Como ya hemos explicado antes, el supervisor se conecta al microcontrolador maestro vía una interfaz serie. La transmisión de datos a través de este bus se realiza en paquetes de tamaño byte, por lo que resulta imposible trabajar con él como lo haríamos con le bus CAN. Es por ello que se tuvieron que crear métodos específicos de comunicación serie con el supervisor, los cuales han sido explicados en la anterior sección.

Nuestra principal meta fue lograr adaptar el protocolo del lado microcontroloador al lado supervisor, puesto que deseábamos que fuese compatible con las tramas y que éstas no necesitasen conversión alguna. Es por ello que las tramas enviadas por el supervisor son reconocidas por el maestro sin necesidad de grandes conversiones y viceversa.

El supervisor es compatible con los tipos de trama de TouCAN del lado microcontrolador en el sentido de que la codificación binaria y el significado son los mismos. Aún así, no existe una trama como una estructura definida como en el lado microcontrolador, sino una serie de bytes que han de ser interpretados según

el orden de llegada. Es por ello que decidimos denominar a las tramas del lado supervisor **comandos** o **datos**.

Frametype	Destination id (HIGH)	Destination id (LOW)	Data length	Data 0	...	Data 8
byte 0	byte 1	byte 2	byte 3	byte 4		byte 11

Figura 9.8: Bytes de comando/dato del supervisor.

Como se puede observar no existe un identificador de origen, esto hace que el supervisor sea invisible para los demás nodos. En caso de que el supervisor deseara establecer comunicaciones con otros nodos, delegaría los comandos sobre el maestro, y éste a su vez le pasaría las respuestas.

9.2.1. Organización de código

Al ser esta librería una adaptación del lado microcontrolador repercutió en que la estructuración de código deseada fuese muy similar a la de un microcontrolador maestro.

9.2.2. Tipos de trama TouCAN (supervisor)

Durante las fases de pruebas, descubrimos que, al conectar un supervisor mediante el puerto serial al microcontrolador, se producía un reseteo inevitable del segundo, lo cual conllevaba a que ejecutase de nuevo sus rutinas de configuración. Por otra parte, el supervisor, al ser más potente que el microcontrolador, requería esperar muchas veces a que éste se configurase y saber cuándo exactamente había terminado para poder comenzar su labor, por ello, para todos los maestros potenciales de ser conectados a supervisores, se creó una especie de "handshake.º apretón de manos para establecer correctamente la comunicación entre maestro y supervisor. Esta rutina habría de ser añadida al final del setup() del microcontrolador y tras completar las rutinas de configuración:

```
void setup(){
        [...]

        //Handshake routine
        while(Serial.read() != IS_READY);
        Serial.write(AVAILABLE);

        //Send the nodes list
        Serial.write(NNODES);
        for(int i = 0; i< NNODES; i++)Serial.write(nodes[i].id);
}
```

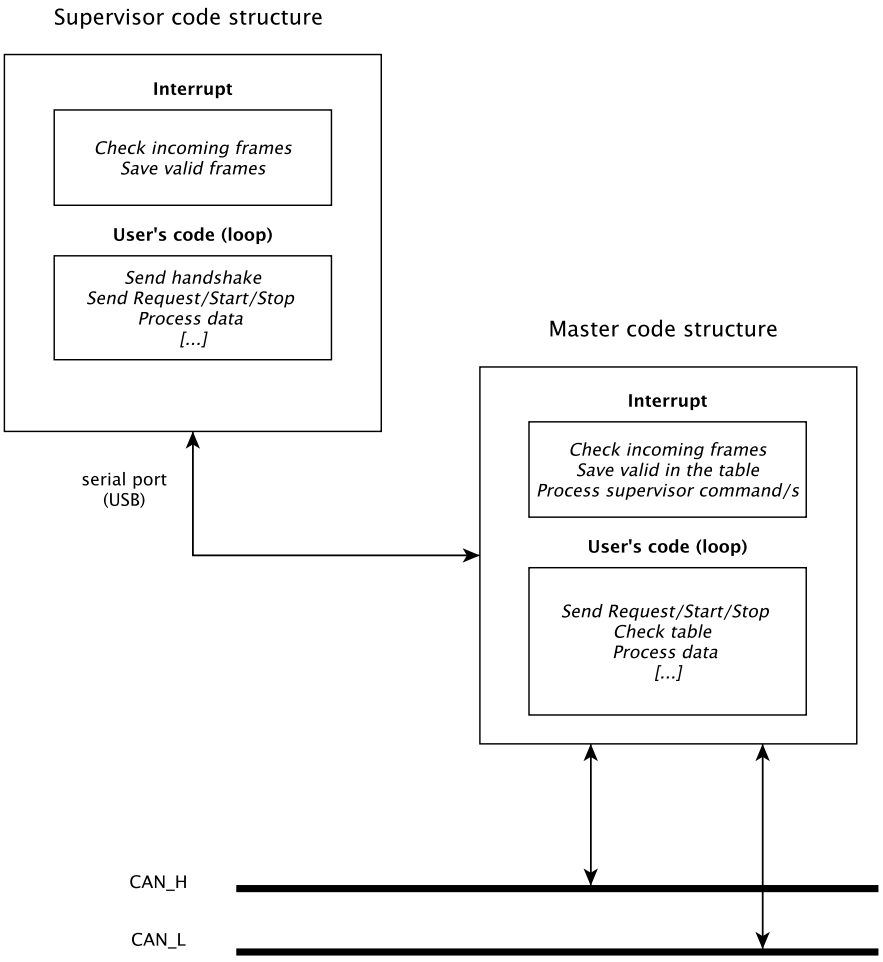


Figura 9.9: Estructuración de código de supervisor y maestro.

Las macros IS_READY y AVAILABLE se encuentran definidas en el fichero TouCAN_def.h

Por otra parte, el supervisor ha de poder actuar como si se tratase de un maestro más en la red y para ello delega todas su peticiones en el maestro al que se encuentra conectado. De esta capacidad nace la necesidad de que el supervisor tuviese su propia tabla de nodos no obstante esto presentaba varios inconvenientes:

- El crear un duplicado de la tabla del maestro provocaría tarde o temprano inconsistencias entre la tabla del supervisor y la del maestro
- Derivado del primer caso, si optásemos por actualizar ambas tablas con cada transmisión, el maestro tendría poco tiempo para sí mismo y teniendo en cuenta que se trata de un dispositivo que funciona a una velocidad mucho menor que la del supervisor podría acabar saturado.

- El buffer de recepción serial del maestro sólo posee capacidad para 64 bytes. Las pruebas demostraron que era muy fácil saturar este buffer.
- El supervisor puede actuar sobre la red pero es invisible a ojos de los microcontroladores salvo al maestro al que se encuentra conectado. Esto podría llevar a situaciones en las que otros maestros pidiesen datos al maestro conectado al supervisor y si éste se encontraba actualizando la tabla podría acabar con los buffers de recepción CAN saturados.

Debido a esto, decidimos optar por enviar sólo los nodos disponibles en la red, si no fuese posible atender a la petición del supervisor por estar ocupada la entrada deseada en la tabla se emitiría una trama de tipo reject explicando a qué se debe el rechazo.

Por su parte, teniendo en cuenta que la diferencia de velocidad entre ambos dispositivos, en general los tiempos entre envíos sufrieron un gran aumento. Por ejemplo, si bien una transmisión de datos asíncrona sin supervisor de nodo común a maestro se podía realizar con un mínimo de cada 5 milisegundos, el añadir un supervisor hacía que la capacidad del maestro para recibir datos por el puerto CAN cayese drásticamente al no poder acceder a su buffer de recepción CAN con tanta frecuencia. Las pruebas demostraron que el tiempo mínimo entre envío y envío había de ser aumentado a unos 75 milisegundos debido al overhead añadido por el supervisor.

Capítulo 10

Pruebas

En este capítulo no relataremos únicamente las pruebas realizadas con el fin de depurar, sino las que se realizaron como pruebas de estrés para comprobar los límites de las placas.

10.1. Pruebas de depuración

En esta sección relataremos las pruebas realizadas con el fin de depurar el código durante las fases de desarrollo, así como las conclusiones extraídas de éstas.

10.1.1. Maestro a nodo, petición síncrona

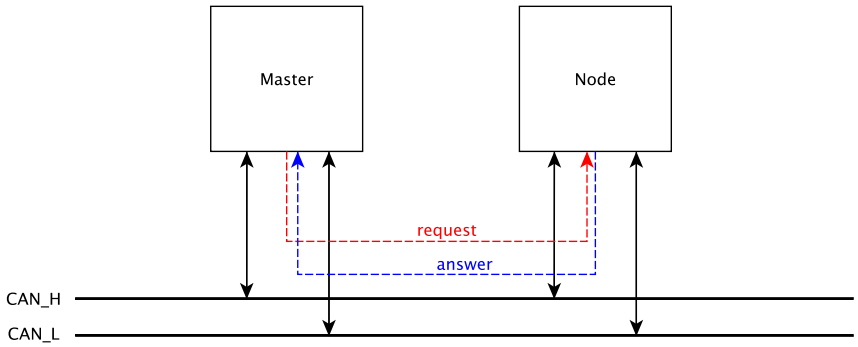


Figura 10.1: Maestro y nodo.

Con esta prueba sencilla tratamos de ver si nuestro protocolo funcionaba en sus primeras etapas puesto que el request fue de las primeras funciones que implementamos. Cada vez que se realizaba un cambio grande se realizaba primero esta

prueba que considerábamos básica. Inicialmente la implementación del request era bloqueante puesto que se quedaba bloqueado a la espera de una trama de respuesta. Las pruebas demostraron que si bien el funcionamiento era correcto, el hecho de ser bloqueante hacía que perdiese tiempo de ejecución para el código del usuario.

10.1.2. Maestro a nodo, petición asíncrona

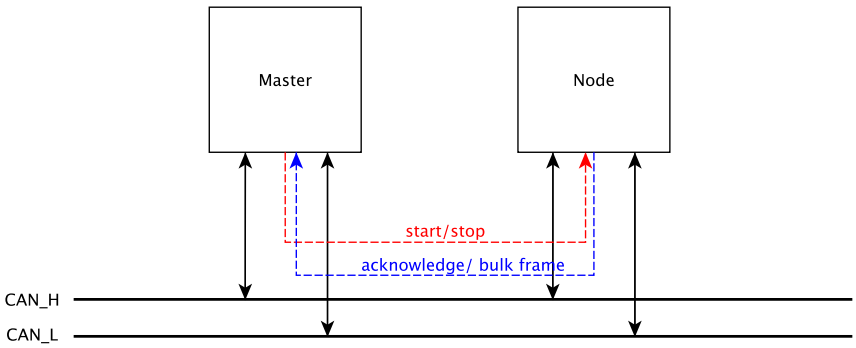


Figura 10.2: Maestro y nodo.

El propósito de esta prueba es doble. Por un lado comprueba si los tiempos de interrupción son adecuados para la recepción rápida de mensajes y por el otro, si se establecen correctamente los contadores y las condiciones de parada de la interrupción del nodo.

10.1.3. Maestro a nodo, petición asíncrona y síncrona

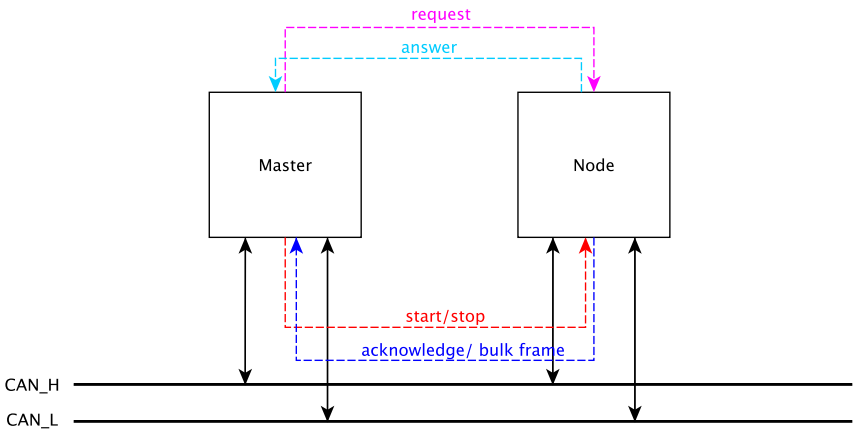


Figura 10.3: Maestro y nodo.

Con esta prueba tratamos de comprobar el funcionamiento de get_start() y get_start_interrupt(). Tras implementar la tabla de nodos, esta prueba nos sirvió además para comprobar si efectivamente las escrituras de los establecimientos de comunicación con un nodo se realizaban de forma correcta.

10.1.4. Maestro a nodo A, petición síncrona, a nodo B, asíncrona

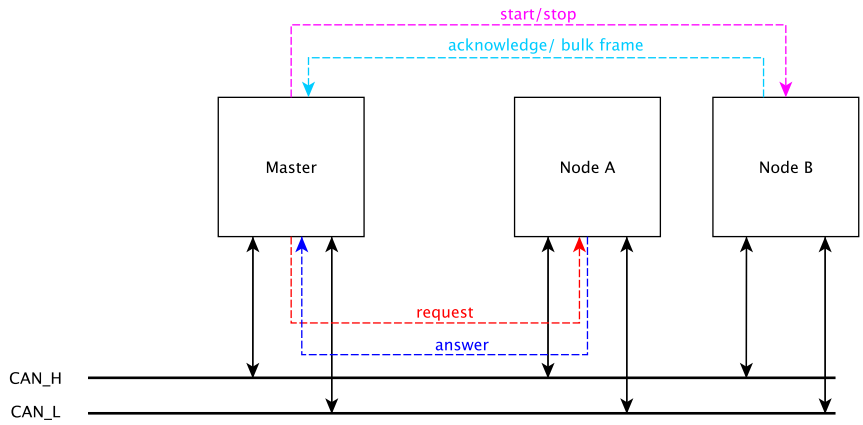


Figura 10.4: Maestro y nodos.

Con esta prueba tratamos de comprobar el correcto funcionamiento de la tabla de nodos. Establecer comunicaciones entre dos nodos distintos permitía escribir en más de una entrada de la tabla.

10.1.5. Maestro A a nodo, petición asíncrona, maestro B a maestro A, petición síncrona

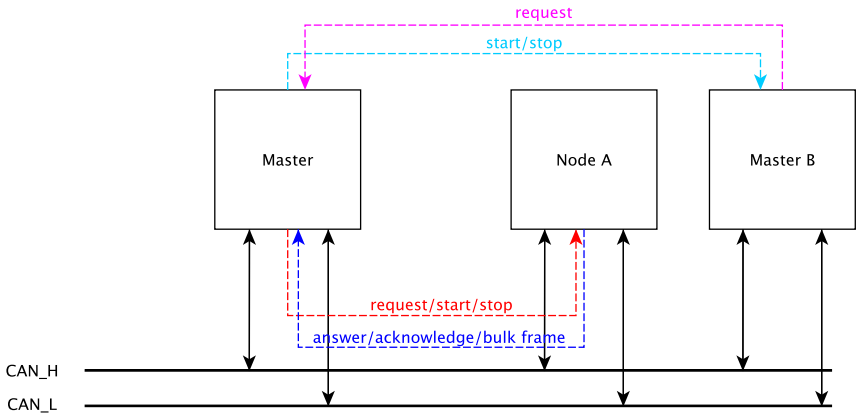


Figura 10.5: Maestro, nodo y maestro.

Esta prueba se realizó con el fin de comprobar el correcto funcionamiento de la función attend_petition(). Al tratarse de una función que trata de resolver un caso especial de la tabla, las pruebas anteriores no valían para ello. De esta prueba empezamos a sospechar los límites a los que es capaz de trabajar un nodo maestro cuando se le añade una carga de trabajo extra.

10.1.6. Maestro A a nodo, petición asíncrona, maestro B a maestro A, petición asíncrona

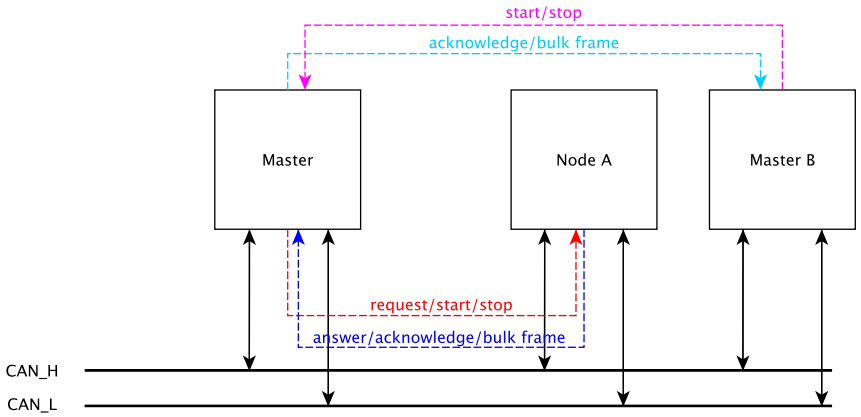


Figura 10.6: Maestro, nodo y maestro.

La finalidad de este test es similar al anterior, pero añadiendo además la comprobación del correcto establecimiento de contadores sobre la interrupción del maestro.

10.1.7. Supervisor a nodo, petición síncrona vía maestro

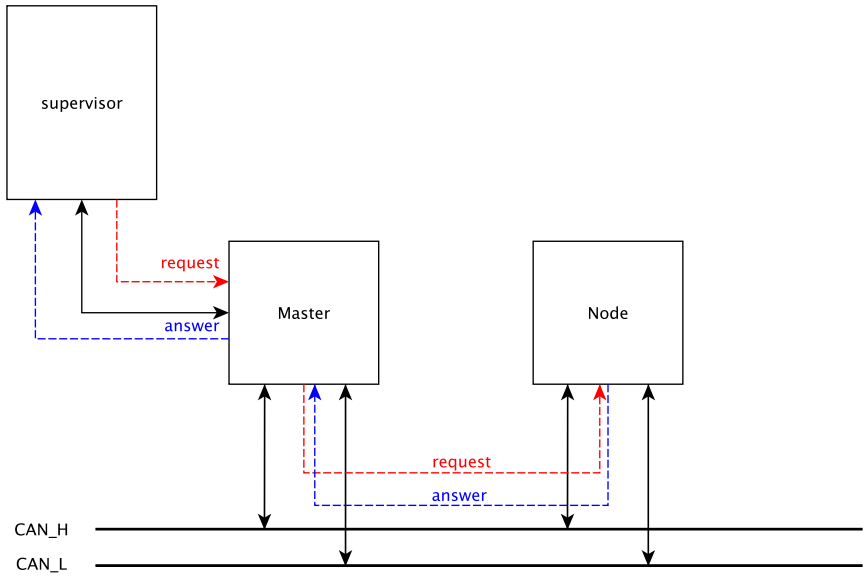


Figura 10.7: Supervisor, maestro y nodo.

Una vez finalizadas las pruebas sobre el lado microcontrolador y tras tener aproximaciones funcionales del lado supervisor comenzamos con esta prueba. La prueba tiene varios objetivos: testear la comunicación por el puerto serie entre el supervisor y el maestro y comprobar la compatibilidad de los comandos del supervisor con los del microcontrolador.

10.1.8. Supervisor a nodo, petición asíncrona vía maestro

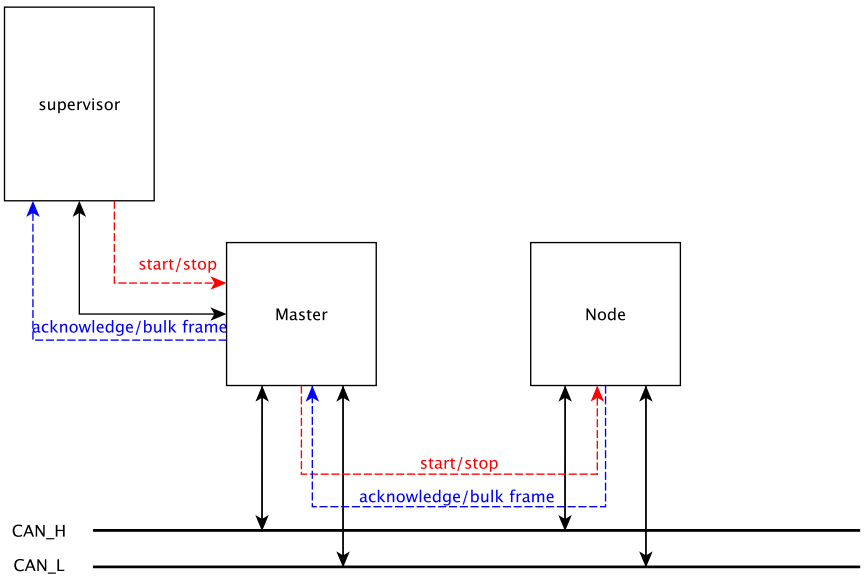


Figura 10.8: Supervisor, maestro y nodo.

Prueba de la capacidad de reenvío de datos asíncronos del maestro al supervisor. Esta prueba nos ayudó a comprobar las correctas marcas sobre la tabla de nodos puesto que la comunicación asíncrona requiere de un chequeo más completo de los distintos flags asíncronos involucrados. Por otra parte, el hecho de estar involucrado el puerto serial para tantas transmisiones del maestro al supervisor hizo que sirviese de testeo de las capacidades del buffer serie.

10.1.9. Supervisor a nodo, petición síncrona/asíncrona vía maestro

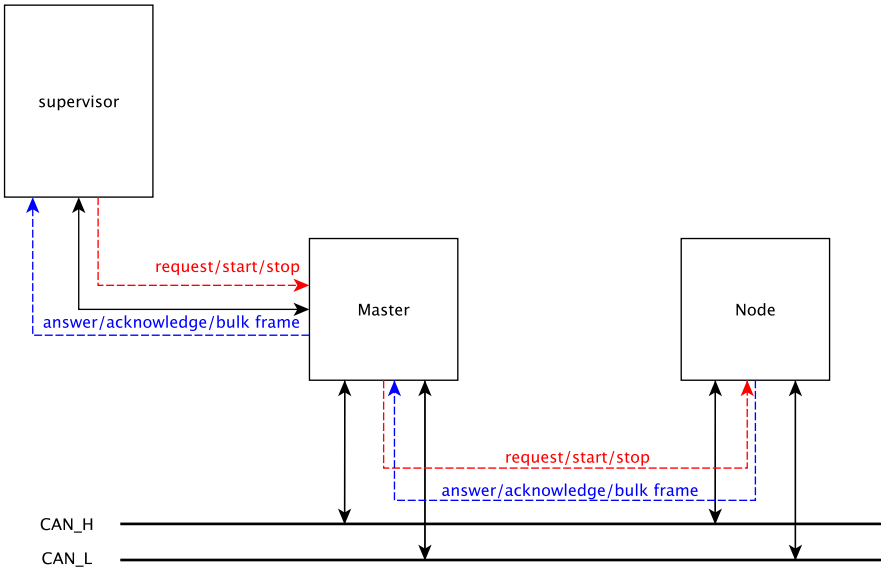


Figura 10.9: Supervisor, maestro y nodo.

Prueba con la misma finalidad de las dos anteriores pero añadiendo cierto grado de complejidad para comprobar la correcta escritura y lectura sobre la tabla de nodos. También funcionó como una de las primeras pruebas de estrés. De esta prueba se extrajeron, entre otras cosas, aproximaciones al período de envío más adecuado y los valores de timeout mínimos para no descartar paquetes tardíos.

10.1.10. Supervisor a maestro, petición síncrona

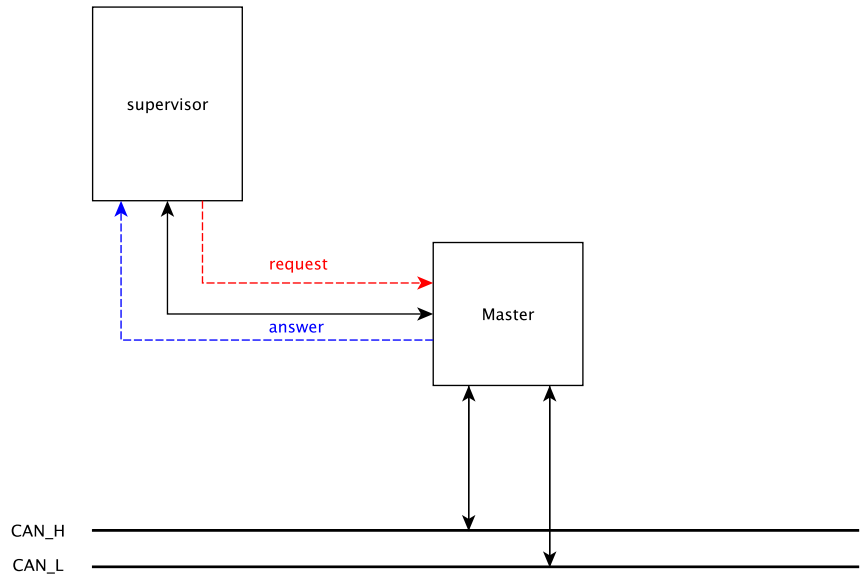


Figura 10.10: Supervisor y maestro.

La prueba se realizó con el fin de probar la recientemente (por aquel entonces) añadida capacidad de que el maestro funcionase como un nodo más para el supervisor.

10.1.11. Supervisor a maestro, petición asíncrona

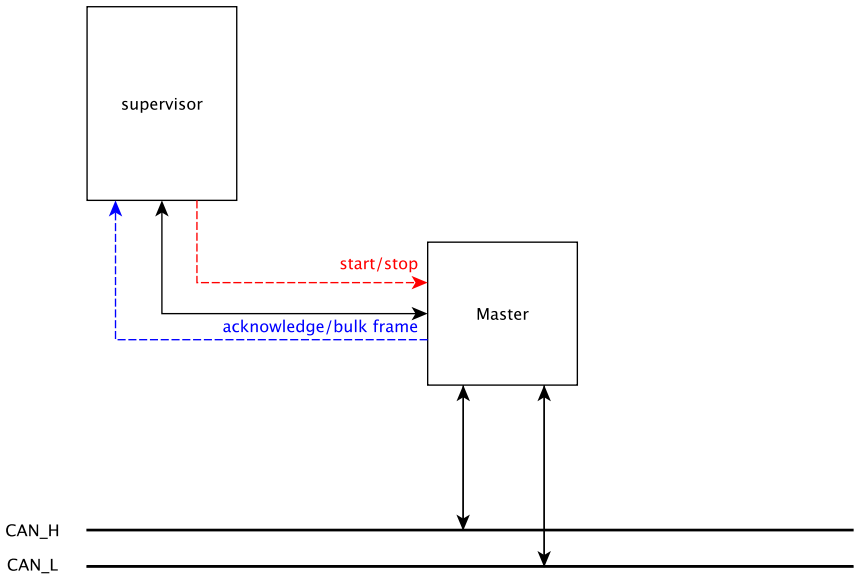


Figura 10.11: Supervisor y maestro.

Esta prueba se creó con el mismo propósito que la anterior, pero tratando las comunicaciones asíncronas.

10.1.12. Supervisor a maestro, petición síncrona/asíncrona

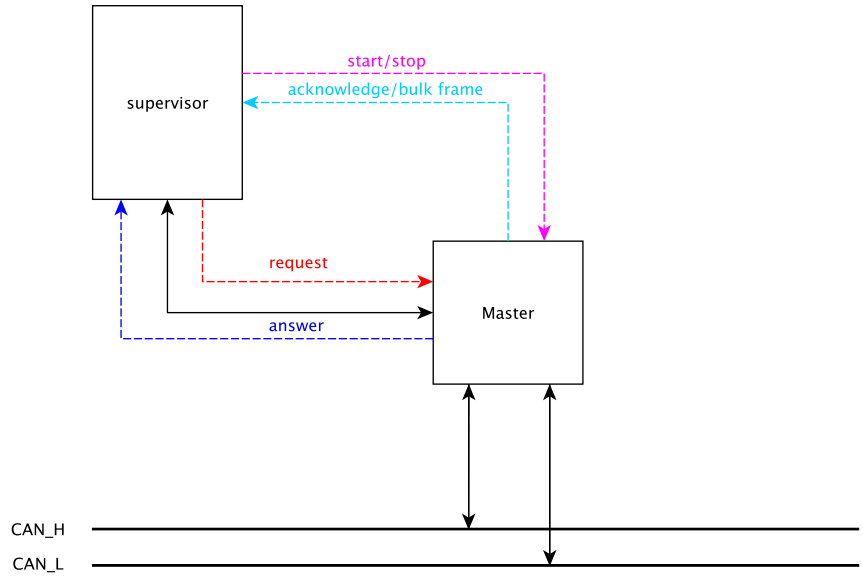


Figura 10.12: Supervisor y maestro.

Última prueba de debug realizada. Se trató de observar si la escritura de la tabla era la correcta cuando se realizaban ambos tipos de comunicación. Junto a las dos anteriores, sacamos la deducción de que el maestro trabaja mejor cuando sólo tiene que procesar un lado de la comunicación, es decir si sólo atiende a conexiones por el lado supervisor o si sólo establece comunicaciones por el lado microcontrolador

10.2. Pruebas de estrés

Este grupo de pruebas se realizaron con el fin de conocer los límites soportados por el microcontrolador. Como resultado colateral, también aprendimos a predecir comportamientos que antes nos parecían anómalos ya que en muchos de los casos se producían timeouts por sobrecargas de buffers, lo cual era imposible de depurar con métodos tradicionales sin interferir en el puerto serie. Este control de los tiempos fue primordial para asegurar los tiempos que se mencionan en esta memoria.

10.2.1. Test de velocidad simple

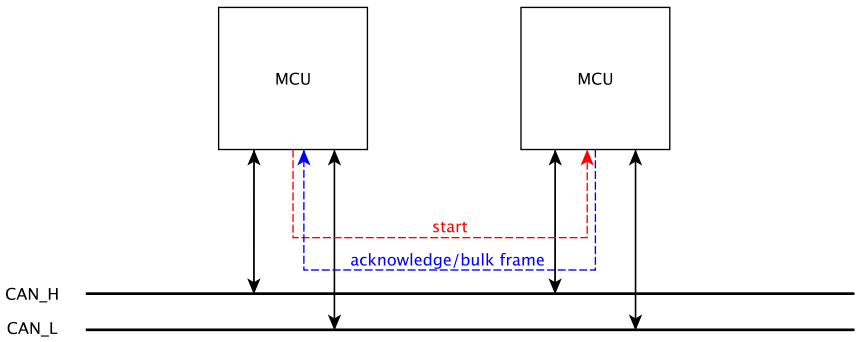


Figura 10.13: Microcontrolador a microcontrolador.

Se trata de una prueba sencilla en la que se trató de comprobar la velocidad máxima de transmisión y recepción posible. Conectando dos microcontroladores (MCU) entre sí y mediante una transmisión asíncrona (la más rápida existente) se transmitieron 1000 tramas TouCAN de longitud de campo datos 8. La prueba se realizó diez veces con los siguientes resultados:

Prueba	Tiempos (microsegundos)
Prueba 1	11891728
Prueba 2	11908108
Prueba 3	11898956
Prueba 4	11899947
Prueba 5	11908731
Prueba 6	11898836
Prueba 7	11890858
Prueba 8	11895721
Prueba 9	11906748
Prueba 10	11901288

Tabla 10.1: Tabla correspondiente a las pruebas de testeo de tiempo.

Lo cual arroja una media de 11900092,1 microsegundos, es decir; casi 12 segundos para transmitir y recibir correctamente 1000 mensajes. Éste sería el máximo de velocidad al que podíamos aspirar, 83 mensajes por segundo lo que vienen siendo aproximadamente 1344 bytes por segundo de los cuales, 672 bytes son datos netos por segundo.

10.2.2. Prueba completa

En esta prueba tratamos de crear una red con todos los elementos posibles al mismo tiempo y realizando el máximo de tráfico. Como se puede ver en el esquema, la red se compone de dos maestros, A y B, un supervisor conectado a A y nodo común. Las peticiones son originadas por el supervisor hacia el nodo, pasando antes por el maestro A. Mientras, por el otro lado el maestro B se encuentra realizándole todo tipo de peticiones a A. Se partió de un período establecido de 100ms para la interrupción de envío y se fue disminuyendo de forma gradual hasta observar fallos, siendo 75ms el punto de ruptura.

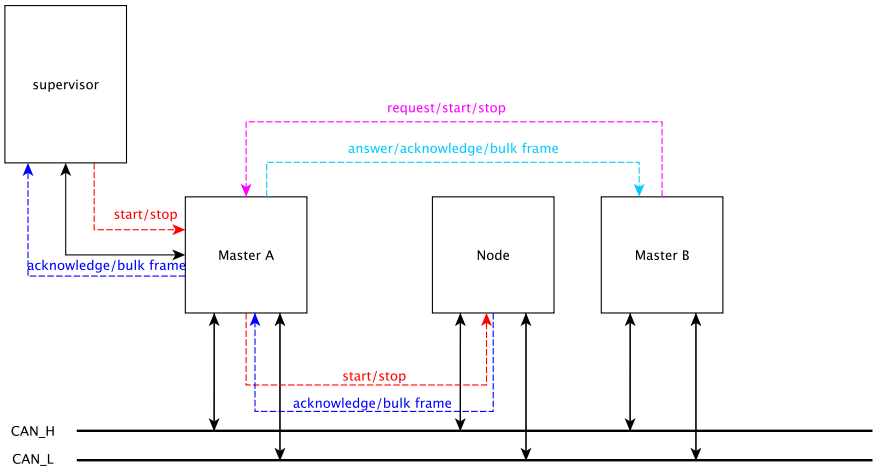


Figura 10.14: Supervisor, maestros y nodo.

Esta prueba añade una carga de trabajo muy dura sobre el maestro A, y es en esta situación en la que comienzan a haber pérdidas de datos. Esto es comprensible puesto que los buffers de A quedan saturados, ya que éste no tiene tiempo de realizar lecturas sobre ellos debido al gran volumen de procesamiento que requiere atender todas la recepciones de datos y peticiones.

Otra variante de esta prueba es la siguiente.

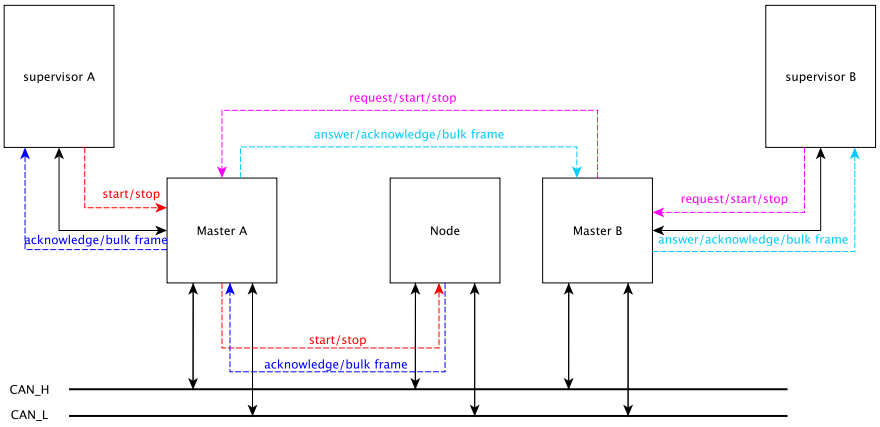


Figura 10.15: Supervisores, maestros y nodo.

Es similar a la anterior pero ahora, las peticiones del maestro B no son generadas por iniciativa propia del maestro sino por un supervisor B conectado a él. La diferencia de cargas de trabajo quedó demostrada puesto que el maestro B respondía correctamente a las peticiones de su supervisor, mientras que A generaba a veces reintentos por timeout. Se partió de un período establecido de 100ms para la interrupción de envío y se fue disminuyendo de forma gradual hasta observar los fallos, siendo 75ms el punto de ruptura.

Capítulo 11

Conclusiones

El objetivo de este Trabajo de Fin de Grado fue la integración de redes basadas en microcontroladores, creando un protocolo basado en CAN. Del desarrollo del trabajo para alcanzar dicho objetivo, hemos extraído las siguientes conclusiones:

- El tiempo para la planificación previa es tan o más importante que el código en sí. Sin lugar a dudas la llegada de este proyecto a buen puerto ha sido en gran medida gracias a la planificación previia de cada una de sus etapas y al diseño conceptual previo a la implementación.

- La latencia y los tiempos son aspectos a tener muy en cuenta cuando se trabaja con esta clase de dispositivos. Respuestas muy lentas o muy rápidas debido a periodos mal establecidos, pueden conllevar a comportamientos inesperados.

- Dada la escasa memoria de estos aparatos conviene crear programas lo más eficientes posibles en consumo.

- Una comunidad activa es lo que le aporta interés a una tecnología. Sin la ayuda de la comunidad es innegable que este trabajo no se podría haber llevado a cabo en los plazos establecidos.

Creemos asimismo que los objetivos académicos también se han logrado alcanzar de forma exitosa:

- Algoritmos, programación y estructuras de datos.

- Diseño de Sistemas Basados en Microcontroladores.

- Sistemas Empotadros y de Tiempo Real.

- Algoritmos y Programación Paralela.

- Sistemas Operativos.

11.1. Mejoras futuras

A pesar de que consideramos a TouCAN como eficiente y con un gran potencial, somos conscientes de que la perfección no existe y de que las actualizaciones y mejoras constantes mantienen vivo un proyecto como éste. Es por ello que existen una serie de mejoras que nos gustaría proponer:

- Uso de interrupciones para la obtención de mensajes: Este punto conllevaría a una mejora sustancial del rendimiento puesto que se obtendrían mensajes con cada salto al manejador y no se malgastaría tiempo uno de los timers con el fin de realizar chequeos periódicos de los buffers.
- Mejora de latencias: Si bien el tiempo de respuesta de TouCAN en condiciones de estrés es bastante rápido consideramos que existen márgenes de mejora.
- Mejora de la escalabilidad: Actualmente TouCAN funciona sobre placas Arduino UNO y shields basados en MCP2515, no obstante, teniendo en cuenta que los microcontroladores de la misma familia pero distinta gama son muy similares entre sí, la adaptación de la librería podría ser sencilla e interesante de cara a ofrecer más variedad al usuario final.
- Port del lado supervisor a Mac OS X: Puesto que se trata de un sistema basado en UNIX, no sería difícil realizar un port en un futuro cercano. Durante el desarrollo del TFG ya realizamos compilaciones sobre esta plataforma con resultados bastante satisfactorios y a los que sólo les faltaba adaptarse a ciertas características del sistema operativo.

Capítulo 12

Manual de usuario

12.1. Lado microcontrolador

12.1.1. Preparación del Entorno de Desarrollo

Es necesario instalar una serie de dependencias antes de poder ejecutar el entorno de desarrollo de Arduino, necesario para grabar los programas sobre el microcontrolador.

1. **Instalación de OpenJDK7:** El entorno de desarrollo de Arduino está basado en Java por lo que necesitaremos el kit de desarrollo para esta plataforma. Para instalarlo, abrimos un terminal y tecleamos el comando *sudo apt-get install openjdk-7-jdk*, aceptamos y listo.

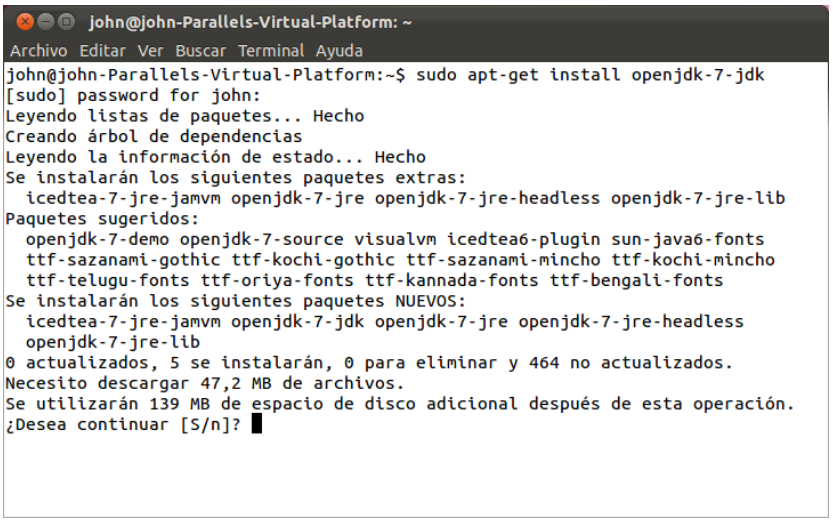


Figura 12.1: Instalación de OpenJDK7 sobre Ubuntu.

2. **Instalación del compilador AVR:** El siguiente paso será descargar el compilador para la arquitectura del microcontrolador, AVR. De nuevo, sobre un terminal, introducimos el siguiente comando: *sudo apt-get install gcc-avr avr-libc binutils-avr avrdude*, aceptamos los requisitos y terminamos.

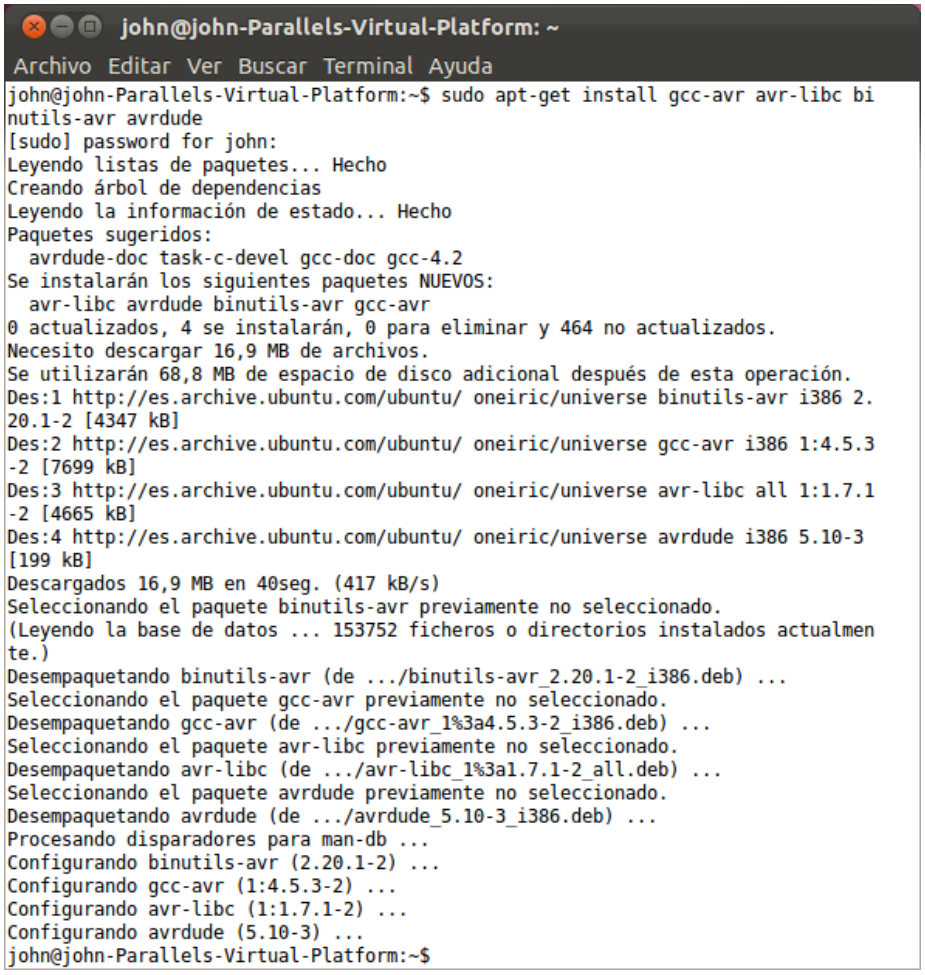


Figura 12.2: Instalación del compilador AVR sobre Ubuntu.

3. **Descargar el entorno de programación Arduino IDE:** Finalmente, instalamos el entorno de desarrollo con interfaz gráfica de Arduino. Existen dos sencillas maneras de hacerlo:

   - Si nos encontramos en Ubuntu, la opción más sencill es acudir al Centro de Software y descargar la versión disponible desde allí.
   - De forma general, podemos acudir a la página oficial de Arduino y descargar la última versión del entorno. El link es el siguiente: `http://arduino.cc/en/Main/Software`

12.1.2. Instalación de la librería

En la carpeta descargada acceder a la carpeta *libraries* y copiar ahí la librería TouCAN y la versión modificada de MsTimer2.

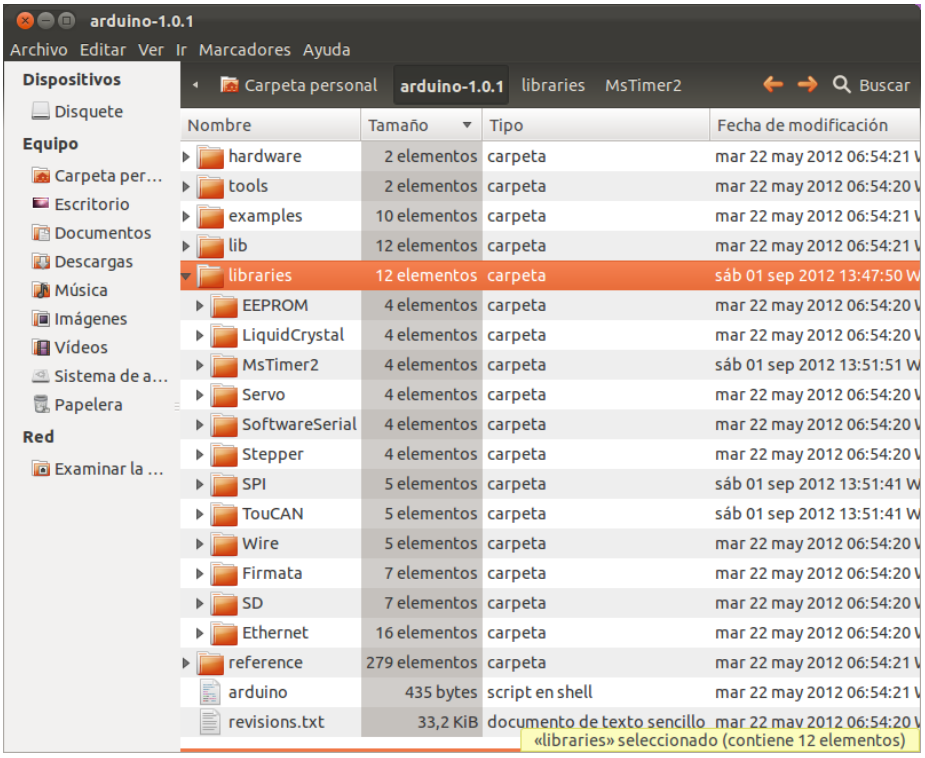


Figura 12.3: Carpeta para las librerías de Arduino.

12.1.3. Compilación

Por defecto, TouCAN viene con una serie de plantillas compilables para el usuario. Estas plantillas están enfocadas a tareas comunes, haciendo necesario únicamente que el usuario procese los datos en los callbacks adecuados y asigne los identificadores.

- *master_code.ino:* Código genérico para un maestro sin conexión a un supervisor.
  - Posee un callback síncrono a rellenar.
  - Las peticiones asíncronas se atienden en la interrupción manejada por timer_handler.
  - Durante el setup() asignar los identificadores en la tabla, como en el ejemplo adjunto.

- *supervisor_toucan.ino:* Código genérico para un maestro con conexión a un supervisor.
  - Posee 2 callbacks para atender peticiones del supervisor, uno síncrono y otro asíncrono.

- El resto de peticiones se han de manejar de la misma manera que en master_code.ino.
- Durante el setup() asignar los identificadores en la tabla, como en el ejemplo adjunto.

- *receiver_bulk_interrupt.ino:* Plantilla genérica para un nodo correspondiente a la figura 9.5.
  - Período mínimo admisible para la interrupción: 75ms.
  - Incluye 2 callbacks a rellenar, uno síncrono y otro asíncrono.

- *receiver_bulk_loop.ino:*Plantilla genérica para un nodo correspondiente a la figura 9.6
  - Período mínimo admisible para la interrupción: 5ms.
  - Incluye 2 callbacks a rellenar, uno síncrono y otro asíncrono.

Para realizar la compilación de las plantillas existen varias maneras, o bien arrastrar y soltar el fichero ".ino"sobre una ventana del entorno de desarrollo o usar el menú correspondiente de la barra de herramientas.

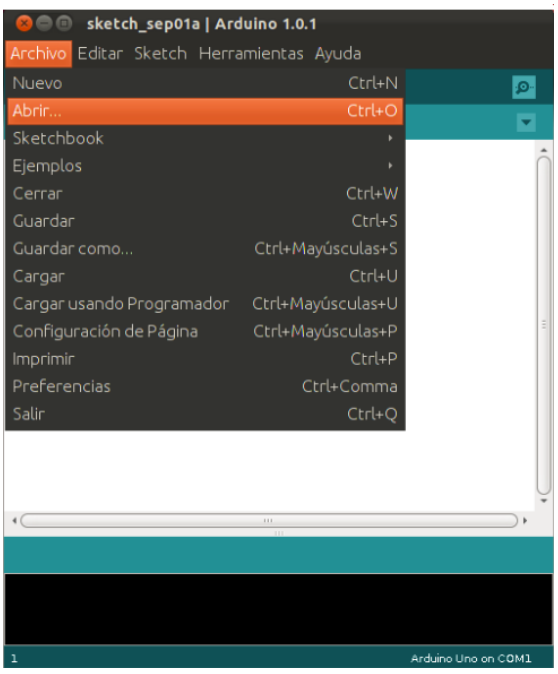


Figura 12.4: Apertura de un fichero .ino.

Una vez abierto, el archivo, elegimos cuál será el microcontrolador destino de la grabación. Para ello, desde la barra de herramientas accedemos a Herramientas -> Puerto Serial y escoger cuál de los dispositivos conectados vía usb es el que queremos grabar.

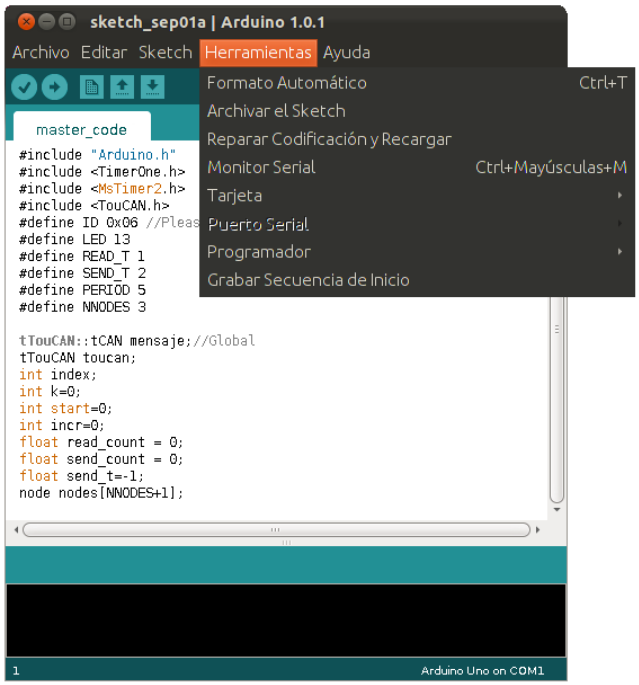


Figura 12.5: Elección del dispositivo.

Finalmente, cargar el programa sobre el microcontrolador haciendo click sobre el botón indicado en la figura:

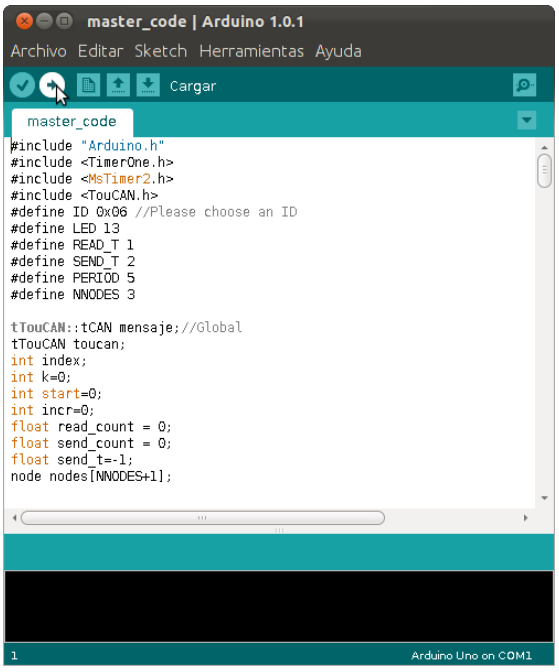


Figura 12.6: Carga del programa.

Para más información sobre los métodos y las clases, ver el capítulo "Diseño e Implementación".

12.1.4. Modo depuración

TouCAN proporciona un modo depuración. Acudiendo al fichero TouCAN_def.h y descomentando la macro *DEBUG_MODE*, se mostrará por el monitor serial del entorno Arduino IDE una traza de la ejecución.

Es importante tener en cuenta que al hacer uso del puerto serial para enviar los datos para realizar impresiones por pantalla, este modo es incompatible cuando se ejecuta un maestro conectado a un supervisor o cualquier otro nodo que haga un uso intensivo del puerto serial ya que existiría una colisión de datos.

12.2. Lado supervisor

Los programas del lado supervisor se incluyen en el CD del TFG asociado a dicho tema.

12.2.1. Instalación del compilador g++

Para compilar la librería y los programas necesarios para el supervisor, es necesario un compilador de C++. Para realizar la instalación, abrir un terminal y escribir *sudo apt-get install g++*

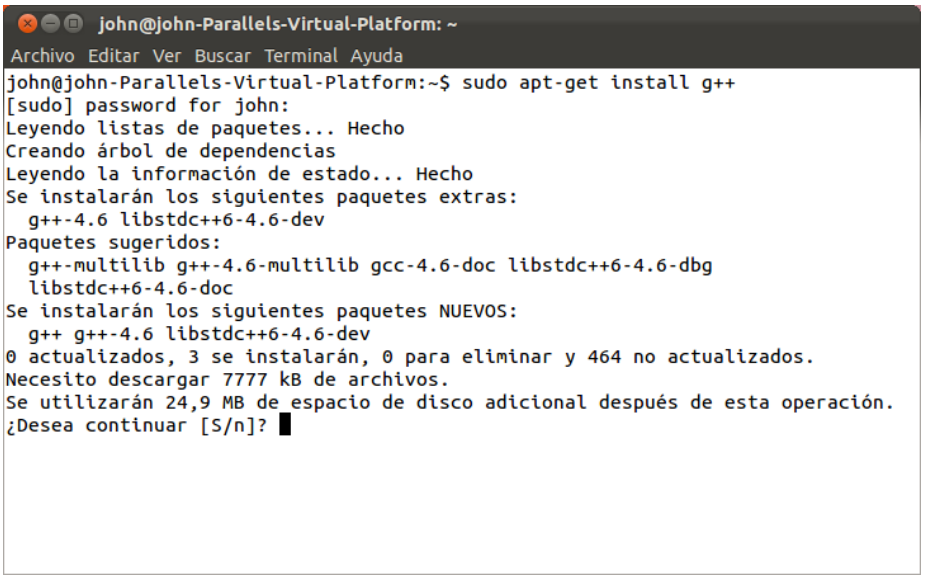


Figura 12.7: Instalación del compilador de C++.

12.2.2. Compilación y ejecución

Una vez instalado el compilador de C++, nos movemos a la carpeta TouCAN (para el lado supervisor, no confundir con el lado microcontrolador), para compilar el programa del lado supervisor. Sobre un terminal tecleamos *g++ -o nombre_de_ejecutable main.cpp*

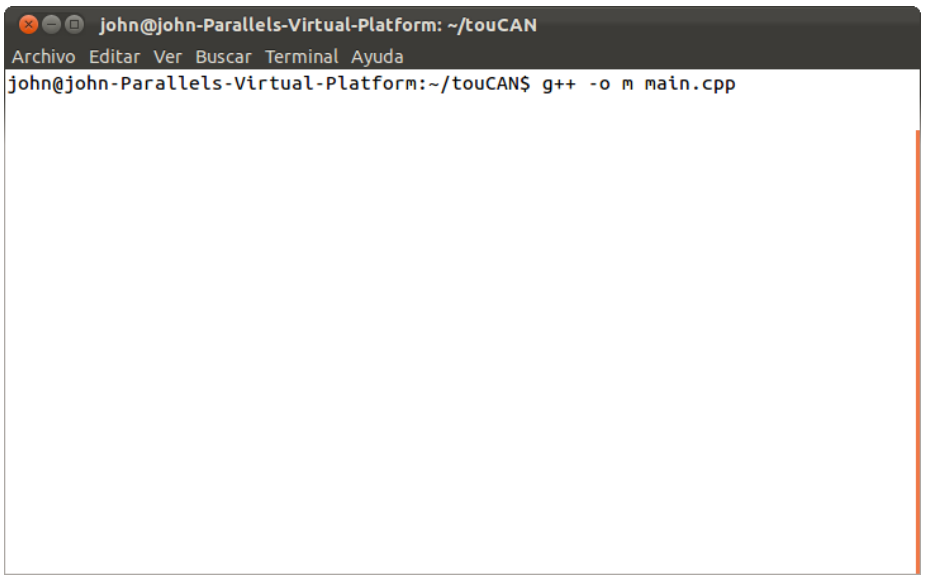


Figura 12.8: Compilación de TouCAN en el lado supervisor.

Por último conectamos via usb el equipo a la placa Arduino maestra y ejecutamos el programa.

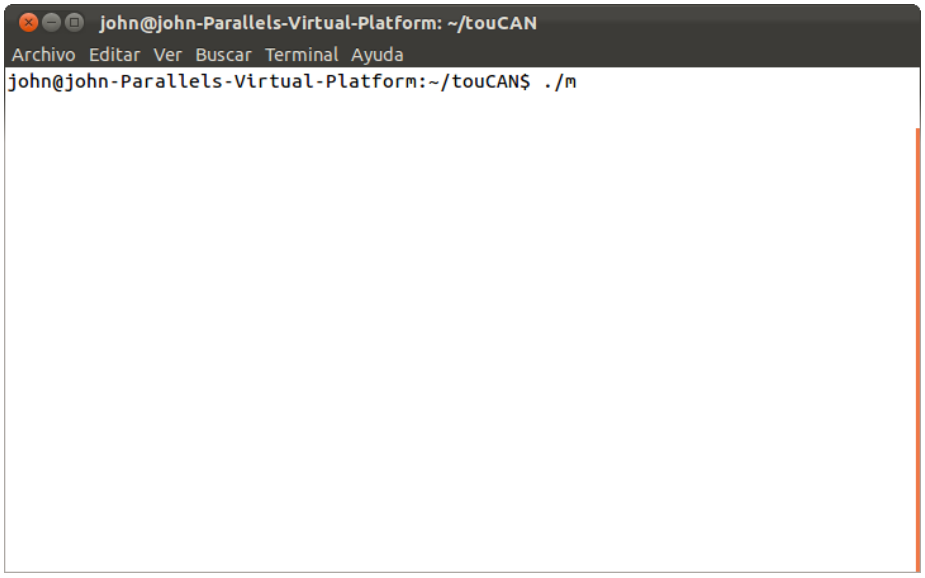


Figura 12.9: Arranque del programa del lado supervisor.

Bibliografía

[1] Arduino, artículo en wikipedia. http://es.wikipedia.org/wiki/Arduino.

[2] Arduino uno, página oficial de arduino (inglés). http://arduino.cc/en/Main/ArduinoBoardUno.

[3] Can bus, atículo en wikipedia (inglés). http://en.wikipedia.org/wiki/CAN_bus.

[4] Microcontrolador, artículo en wikipedia. http://es.wikipedia.org/wiki/Microcontrolador.

[5] Proceso unificado, artículo en wikipedia. http://es.wikipedia.org/wiki/Proceso_Unificado.

[6] Protocolo de comunicaciones, artículo en wikipedia. http://es.wikipedia.org/wiki/Protocolo_de_comunicaciones.

[7] Red en bus, artículo en wikipedia. http://es.wikipedia.org/wiki/Red_en_bus.

[8] Shields, página oficial de arduino (inglés). http://arduino.cc/en/Main/ArduinoShields.

[9] Sparkfun can-bus shield, página oficial del fabricante sparkfun (inglés). https://www.sparkfun.com/products/10039.

[10] Robert Bosch GmbH. Can specification, version 2.0. http://www.gaw.ru/data/Interface/CAN_BUS.PDF, 1991.

[11] Microchip. Stand-alone can controller with spi interface. http://www.gaw.ru/data/Interface/CAN_BUS.PDF, 2012.

[12] Raúl Milla Pérez. Página oficial del proyecto arcan. `http://www.arcan.es/`.